UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

ESCUELA NACIONAL DE ESTUDIOS PROFESIONALES
"A C A T L A N"

"ANALISIS DE ALGORITMOS: UN ESTUDIO PARA LA SOLUCION DE PROBLEMAS POR COMPUTADORA."

T E S I S

QUE PARA OBTENER EL TITULO DE:

LIC. EN MATEMATICAS APLICADAS Y COMPUTACION

P R E S E N T A N :

CAMACHO CANCINO SARA
MARTINEZ SANTOS GABRIELA E.

MEXICO, D. F. 1992

UNAM – Dirección General de Bibliotecas Tesis Digitales Restricciones de uso

PROLOGO

El propósito de esta tesis es presentar algoritmos que se usan frecuentemente en la solución de problemas empleando la computadora, con la finalidad de desarrollar en el lector el hábito de cuestionarse al seleccionar un algoritmo preguntas tales como: ¿Qué tan bueno es el proceso escogido? ¿Cuánto tiempo y espacio requerirá para resolver el problema? ¿Cuáles son las características propias del problema? ¿Cuáles son los recursos que se necesitan para su aplicación? En fin, hacerlos comprender que el análisis de un algoritmo es tan importante como la propia solución del problema.

Nuestra estrategia ha sido organizar la tesis en capítulos que traten algoritmos pertenecientes a una clase de problemas, lo cual implica que cada uno de ellos posee un alto grado de independencia y pueden ser analizados sin la necesidad estricta del estudio de un capítulo previo.

Consideramos que los temas presentados son un conjunto materias que permiten analizar los algoritmos en forma práctica y para ilustrar dichos algoritmos recurrimos a la ayuda de **pseudocódigos** con el objetivo de facilitar en algunos casos la comprensión del método, pero también con la idea de crear en el lector una sensación de irealidad y dificultades futuras cuando intente programarlo. Empleamos también **código** (escrito en lenguaje PASCAL o C) con la finalidad de otorgarle al lector una idea más firme y básicamente ya establecida del algoritmo. El determinar realizarlo de esta forma fue un tanto difícil pero consideramos que en un momento dado despierta mayor interés en un tema.

Antes de concluir deseamos expresar nuestro agradecimiento a quienes hicieron posible este trabajo: Nuestros padres por su gran comprensión; los profesores **Eduardo Padilla, Fernado Vera Badillo, Jorge Alberto Márquez Carbonell, Teodoro Sánchez García Daniel Reyes Espinos, Jorge Luis Suarez Madariaga y María de la Gracia Barquera Díaz** por apoyarnos con bibliografía y consejos muy certeros; en especial al **Ingeniero Rubén Romero Ruiz**, asesor de esta tesis por su gran paciencia y valiosa ayuda; Así como a **CADASA** departamento de Informática de Cannon Mills por las facilidades otorgadas para la elaboración del presente trabajo.

ANALISIS DE ALGORITMOS

Un estudio para la solución de problemas por computadora

INDICE

INTRODUCCION 1

CAPITULO 1: INTRODUCCION A LOS ALGORITMOS 3

1.1 ¿Qué es un algoritmo? 4
1.2 Breve historia de los algoritmos 9
1.3 ¿Qué es el análisis de algoritmos? 11
1.4 Herramientas empleadas en la construcción de algoritmos eficientes 12
1.5 Problema-Algoritmo-Programa 27
1.6 ¿Cómo elegir un algoritmo? 33

CAPITULO 2: ALGORITMOS SEMINUMERICOS 35

2.1 Definición y características de los algoritmos seminuméricos 36
2.2 Descripción y ejemplos de algunos algoritmos seminuméricos 36

CAPITULO 3: ALGORITMOS MATEMATICOS 67

3.1 Definición y características de los algoritmos matemáticos 67
3.2 Errores 69
3.3 Análisis de algoritmos matemáticos 75

CAPITULO 4: ALGORITMOS DE ORDENACION Y BUSQUEDA 109

4.1 Algoritmos de ordenación 110
4.2 Algoritmos de búsqueda 152

CAPITULO 5: ALGORITMOS PARA MANIPULAR CADENAS DE CARACTERES. 173

5.1 Algoritmos de manipulación de cadenas de caracteres 174
5.2 Algoritmos de búsqueda de cadenas 191
5.3 Criptología 199
5.4 Compactación de datos 227

CAPITULO 6: ALGORITMOS ESPECIFICOS (ALGUNOS EJEMPLOS)..... 249

6.2 Algoritmos para teoría de gráficas..... 250
6.1 Algoritmos para estadística y probabilidad..... 301

CAPITULO 7: EFICIENCIA DE LOS ALGORITMOS..... 319

7.1 Complejidad de un algoritmo..... 320
7.2 Técnicas de análisis de algoritmos..... 332
7.3 Técnicas de diseño de algoritmos..... 346
7.4 Recomendaciones al seleccionar y programar un algoritmo 355

CONCLUSIONES Y RECOMENDACIONES..... 361

REFERENCIAS BIBLIOGRAFICAS..... 363

APENDICE:

A. La estadística y graficación como herramientas auxiliares en el estudio de algoritmos..... 367
B. Descripción del equipo y lenguaje empleados..... 369

INTRODUCCION

Aún cuando los algoritmos ya se conocían y se usaban en la antigüedad, casi toda su teoría se ha desarrollado en el siglo XX. El papel que juega un algoritmo es fundamental: sin un algoritmo no puede existir programa alguno, y sin programa no hay cosa alguna que ejecutar en una computadora; también son importantes en el sentido de que un algoritmo es independiente tanto del lenguaje en que esté escrito como de la computadora que los ejecute. Por todo esto es posible crear y estudiar algoritmos independientemente de la tecnología del momento, dado que sus resultados siempre permanecerán válidos. Tanto el lenguaje de programación (medio conveniente para expresar un algoritmo) como la computadora (procesador para ejecutarlo) son los medios para obtener un fin: lograr que el algoritmo se ejecute y con ello efectuar el proceso correspondiente.

Cuando manejamos una computadora para resolver un problema debemos tener presente dos aspectos:

1) Aún el problema más simple puede resolverse por una gran variedad de algoritmos, de ahí que la habilidad para manejar algoritmos tenga un valor estratégico muy importante.

2) Una computadora sólo puede resolver un problema después de que se le dice como hacerlo; esto significa que se requiere de un esfuerzo humano para desarrollar un procedimiento de solución que se comunicará a la computadora para que ésta lo utilice.

Existen diversas formas de evaluar y comparar un algoritmo, unas hacen énfasis en su legibilidad y documentación; otras, en la dificultad de su programación y algunas otras en la eficiencia de su operación, la forma seleccionada dependerá del problema en cuestión y de los medios con que se cuenten para resolverlo.

Para estudiar esta tesis el lector sólo necesita poseer los conocimientos básicos de computación; no obstante debe estar preparado para seguir una cadena un tanto complicada de distintas áreas que juegan un importante papel en la carrera de **MATEMATICAS APLICADAS Y COMPUTACION**, ya que son los estudiantes de esta disciplina nuestro principal objetivo y deseamos orientarlos en el área del análisis de algoritmos que muchas veces se ve relegada a segundo plano cuando vemos que nuestro programa da los resultados requeridos. Tratamos de exponer los temas en forma sencilla para despertar el interés en el área, no ahogar al lector desde las primeras páginas e invitarlo a continuar con la lectura.

La tesis consta de los siguientes capítulos:

CAPITULO I (Introducción a los algoritmos): En él se desglosa la fundamentación teórica de los algoritmos (conceptos, clasificación y terminología empleada); así como observaciones introductorias en la elaboración o selección de un algoritmo a partir de la necesidad de resolver un problema.

CAPITULO II (Algoritmos seminuméricos): Se mencionan las características de los algoritmos seminuméricos y se desarrollan algunos ejemplos de este tipo de procesos.

CAPITULO III (Algoritmos matemáticos): Se describen las principales características de los algoritmos matemáticos y se analizan algunos de ellos.

CAPITULO IV (Algoritmos de ordenación y búsqueda): Comprende el desarrollo, descripción y comparación de los principales métodos de ordenación y búsqueda.

CAPITULO V (Algoritmos para manipular cadenas de caracteres): Se realiza un estudio teórico de las diferentes técnicas empleadas en la manipulación de cadenas de caracteres.

CAPITULO VI (Algoritmos específicos): Dada la amplitud del campo de desarrollo y aplicación de los algoritmos no es posible abarcar todo en los capítulos anteriores (ni en toda la tesis) por lo que seleccionamos algunos algoritmos que de alguna forma resulten interesantes para nuestra carrera. Dichos algoritmos se presentan en este capítulo.

CAPITULO VII (Eficiencia de los algoritmos): Se presenta un estudio de la complejidad de un algoritmo así como de las diferentes técnicas de análisis y diseño de ellos; se describen también aspectos relevantes que hay que tener presentes en las fases de selección y programación con la finalidad de que el lector tenga argumentos válidos que respalden la elección de un determinado proceso.

Por último el lector deberá tener presente que a medida que las computadoras aumenten su rapidez y disminuyan su precio, también el deseo de resolver problemas más grandes y complejos seguirá creciendo por lo que la importancia del descubrimiento y el empleo de algoritmos eficientes también se incrementará.

CAPITULO I

INTRODUCCION A LOS ALGORITMOS

Algoritmo, un término bastante común en el área de las ciencias de la computación, que inadvertidamente se encuentra presente en nuestra vida diaria, ya que todos seguimos ciertos procedimientos para realizar determinadas tareas (por ejemplo la acción de cambiarle las llantas a un automóvil) pero, ¿cual es el concepto real de un algoritmo? El descubrimiento y formulación de algoritmos fue uno de los acontecimientos más importantes de las matemáticas desde el momento en que se convirtieron en ciencia. Al respecto el académico A. N. Kolmogorov escribe: "En todos los casos que sea posible, el descubrimiento de algoritmos es la meta natural de las matemáticas".

En su proceso de desarrollo, las matemáticas ha tratado de encontrar algoritmos eficientes y generales que permitan la solución uniforme de clases más amplias de problemas.

1.1 ¿QUE ES UN ALGORITMO?

Un algoritmo es un conjunto finito de instrucciones, cada una de las cuales tiene un significado preciso y puede ejecutarse en un tiempo finito. Las instrucciones de un algoritmo, son una secuencia ordenada de operaciones a realizar para resolver un problema especifico o una clase de problemas. En otras palabras un algoritmo es **una fórmula para la solución de un PROBLEMA**.

En computación el término algoritmo posee un significado especial, aún cuando todo algoritmo es un procedimiento, no todo procedimiento es un algoritmo. La palabra procedimiento se emplea a veces en un sentido muy general para describir una serie de pasos no muy bien definidos; en tanto que un algoritmo es un método cuya definición es muy precisa.

1.1.1 Propiedades de un algoritmo.

Un procedimiento que tiene las siguientes propiedades, según varios autores, es un algoritmo:

A. FINITUD

Cada instrucción debe ser efectiva; esto es los componentes de una instrucción deben ser básicos y entendibles, para que esta pueda ejecutarse en una cantidad finita de tiempo y esfuerzo; así un algoritmo debe terminar después de ejecutar un número finito de instrucciones, sin importar cuales fueron los valores de entrada.

B. AUSENCIA DE AMBIGUEDAD.

Cada instrucción debe estar perfectamante definida; esta condición significa que cada vez que se presente para su ejecución un algoritmo, con los mismos datos de entrada, se obtendrán los mismos resultados.

C. DEFINICION DE SECUENCIA.

La secuencia que deben llevar a cabo los pasos del algoritmo se deben especificar sin lugar a dudas: Todo algoritmo tiene una instrucción inicial única y cada instrucción debe tener un sucesor único para un dato de entrada dado.

D. DEFINICION DE ENTRADA Y SALIDA.

Un proceso con cero o más datos de entrada (datos necesarios para poder iniciar el algoritmo) obtiene uno o más datos de salida (datos presentados al exterior como el resultado del algoritmo).

E. EFECTIVIDAD.

Las instrucciones de un algoritmo deben ordenar tareas posibles de realizar, no se puede llevar a cabo una instrucción sino se tiene la información suficiente o si el resultado de la ejecución de la orden no está definido.

F. DEFINICION DE ALCANCE.

Un algoritmo se aplica a un problema o a una clase de problemas específicos, el rango de las entradas se tiene que definir previamente y es éste el que determina la generalidad de un algoritmo.

1.1.2 Naturaleza de los algoritmos.

Por la cantidad de pasos que un algoritmo emplea para resolver un problema, los algoritmos se clasifican en:

ALGORITMOS DIRECTOS: En este tipo de algoritmos el número de pasos es fijo o se tiene un número máximo fijo; por ejemplo si cada paso de un algoritmo para jugar gato es una jugada, hay cuando más nueve pasos. Los algoritmos de esta clase pueden o no ser de naturaleza cíclica.

ALGORITMO INDIRECTOS: Esta clase de algoritmos son generalmente iterativos y el número máximo de pasos se encuentra relacionado con la cantidad de datos, de tal forma que puede ser calculado apriori. Por ejemplo el procesador de cheques para el pago de una nómina requiere cuando más nk pasos, donde n es el número de empleados y k el número máximo de pasos por cheque.

ALGORITMOS CONVERGENTES: El número de pasos para esta clase de algoritmos está también relacionado con los datos; pero de tal forma que no podemos predecir el número de pasos involucrados. Tales procesos se encuentran a menudo relacionados con investigaciones sobre conjuntos infinitos o procesos convergentes. Por ejemplo, al calcular la raíz de un polinomio el algoritmo puede completar su proceso en forma satisfactoria si la solución tiene un rango de error aceptable. De ahí que se diga que este tipo de algoritmos deben generar una serie de estimaciones (presumibles de la respuesta de algún problema), y siempre se debe tener alguna información respecto al grado de convergencia.

1.1.3 Hipótesis básica de la teoría de algoritmos.

Para poder estudiar la teoría de algoritmos, se requiere de una definición matemática precisa de un algoritmo:

"Un algoritmo es un procedimiento general tal que para alguna pregunta apropiada, la respuesta puede ser obtenida por el uso de un simple proceso computacional o cálculo de acuerdo con un método específico... Un procedimiento general es un proceso en el cual la ejecución se especifica claramente aún en los más pequeños detalles"

HERMES (1965).

"...Un algoritmo es un procedimiento determinístico, al cual se le pueden aplicar cierta clase de entradas simbólicas, produciendo eventualmente una correspondiente salida simbólica"

ROGERS (1967).

"Un procedimiento es una secuencia finita de instrucciones que pueden ser mecánicamente llevadas a cabo, tal como un programa en la computadora...Un procedimiento el cual siempre termina es llamado algoritmo"

HOPCROFR Y ULLMAN (1969).

Estas sólo son algunas de las diferentes definiciones formales del concepto algoritmo que se han dado a través de la historia sin embargo, se ha demostrado que todas ellas son equivalentes al concepto de máquinas de Turing, que se aceptan como las bases formales de la computación:

" TODO ALGORITMO ESTA DADO EN FORMA DE MATRICES FUNCIONALES Y ES EJECUTADO POR SU CORRESPONDIENTE MAQUINA DE TURING "

De esta definición se deduce que un procedimiento resuelva un problema sólo si es posible programar una máquina de Turing que lo intente resolver; este procedimiento se convertirá en un algoritmo si la máquina de Turing se detiene en un momento dado de su ejecución

1.1.4 Máquina de Turing.

Una máquina de Turing **M** consta de 3 partes: una Unidad de Memoria, una Unidad de Control y un Programa

A. UNIDAD DE MEMORIA.

La memoria de una máquina de Turing consiste de una cinta infinita dividida en cuadros, cada cuadro puede contener un símbolo que pertenece a un alfabeto (conjunto de símbolos predefinidos), cada símbolo en la cinta representa un dato. Existen dos aspectos importantes a considerar.

1) La capacidad de memoria de una máquina de Turing conceptualmente es infinita, lo cual implica la posibilidad de

que el desarrollo computacional realizado por la máquina nunca termina, sin embargo la terminación computacional es la característica distintiva entre algoritmos y procedimientos.

2) El programa es parte de la descripción de una maquina de Turing: cada algoritmo o procedimiento específico puede representarse por una máquina de Turing distinta con su respectivo alfabeto.

Cuando se formaliza un procedimiento (representándolo como una máquina de Turing) es importante definir el alfabeto de la máquina de Turing en el que el proceso puede ejecutarse.

Ejemplo:

Si se escogiera una representación particular de una máquina de Turing, que emplea un alfabeto con sólo dos símbolos 0 y 1 más un símbolo especial B que representa un cuadro blanco; la entrada/salida se regiría por la siguiente convención:

I) Inicialmente la cinta se encuentra en blanco (la cinta contiene sólo cuadros con el símbolo B).

II) Antes de que el programa inicie, la entrada se coloca en la cinta y la cabeza de lectura y escritura se posesiona sobre el primer símbolo de la cadena de entrada, que es el primer símbolo que está más a la izquierda de la cinta el cual no es B.

III) Después de que el programa para, los caracteres bajo y hacia la derecha de la cabeza de lectura y escritura antes del primer blanco son la salida.

B. UNIDAD DE CONTROL.

Esta parte determina la instrucción que va ejecutarse y la ejecuta . La unidad de control se comunica con la cinta infinita por medio de la cabeza de lectura y escritura (RWH) y con el programa por medio de la cabeza de sólo lectura (ROH). La RWH es la encargada de leer o escribir el contenido de un sólo cuadro, mientras que la ROH lee una sola instrucción de la máquina de Turing en un tiempo. La máquina de Turing parará sólo si encuentra una instrucción HALT.

C. PROGRAMA.

Es una secuencia finita de instrucciones primitivas que la máquina de Turing es capaz de ejecutar; algunas de esas instrucciones están referenciadas por una etiqueta única.

Las instrucciones primitivas y su interpretación son:

- LEFT Mover la RWH un cuadro a la izquierda.

- RIGHT Mover la RWH un cuadro a la derecha.

- WRITE "a" Reemplazar el símbolo en el cuadro prevaleciente bajo la RWH por el símbolo "a".

- GO TO n Mover la ROH a la instrucción etiquetada con n.

- IF "a" GO TO n Si el símbolo bajo la RWH es "a" mueve la ROH a la instrucción etiquetada con n; si ocurre otra cosa mover la ROH a la siguiente instrucción.

- HALT Termina el proceso.

Actualmente estas instrucciones representan el conjunto mínimo requerido para evaluar una función "computable"; su inherente simplicidad permiten definir la semántica de cada instrucción sin ambigüedad pero provoca también, una tarea ardua al intentar escribir con ellas un procedimiento realista

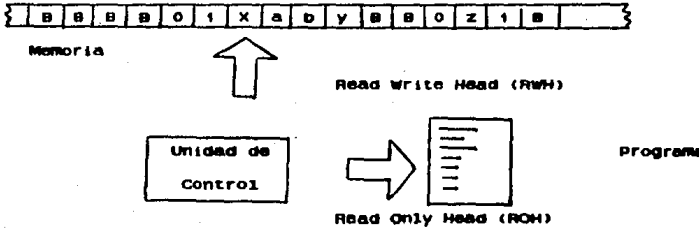
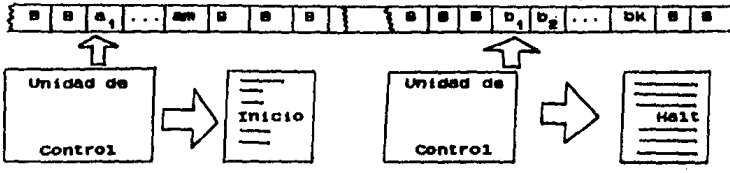


Fig. 1.1 Componentes de la máquina de Turing

Un carácter de entrada a una máquina de Turing, es una cadena finita $x=a_1a_2...a_m$ de símbolos definidos por el alfabeto X (si X no contiene símbolos se dice que es nulo), inicialmente esta cadena se escribe en la cinta con un número infinito de blancos a la derecha e izquierda de la cabeza de lectura y escritura, esta cabeza busca el símbolo a_1 e inicia la ejecución del programa con la primera instrucción. La máquina de Turing puede o no parar con una entrada "x". Si se detiene, la cinta puede contener una cadena $y=b_1b_2...b_k$ con un número infinito de blancos a la derecha o izquierda y se dice que "y" es la salida de la máquina correspondiente a la entrada "x". Por ello se dice que la máquina de Turing existe al definir su alfabeto y su programa.



Configuración Inicial Configuración Final

Fig. 1.2 Configuraciones de la Máquina de Turing

1.2 BREVE HISTORIA DE LOS ALGORITMOS.

El término algoritmo, antes algorismo, originalmente se refiere a cualquier desarrollo computacional vía un conjunto de reglas aplicadas a números escritos en forma decimal. La importancia de tener algoritmos o métodos de cálculo ya era conocida por los egipcios y los griegos de la antigüedad.

Durante la Edad Media el origen de esta palabra aún estaba en duda; algunos lingüistas suponían que se derivaba de hacer la combinación **algiros** (arduo) + **arithmos** (número), otros decían que la palabra provenía de "El Rey Algor de Castilla". Finalmente los historiadores de las matemáticas encuentran su verdadero origen: La pronunciación fonética de **Al-Khowârizmi**, matemático árabe cuyo nombre completo es Abu Ja'far Mohammed (literariamente "Padre de Ja'far, Mohammed, hermano de Moses, nativo de Khowarizm", Khowâri es hoy una pequeña ciudad soviética de Khiva.) quien escribió el célebre libro Kitab al jabr W'al-muqabala (reglas de restauración y reducción) en los años 825 o 830 (la fecha exacta es desconocida).

Aún cuando la evaluación de la lógica comienza con Aristóteles, 384-322 A.C. y otros griegos de la antigüedad, no fue sino hasta por el tiempo de Whitehead y Russell que el estudio de esta disciplina se desarrolló en diferentes direcciones como la de los **ALGORITMOS** y las **COMPUTADORAS**. En 1900 el matemático alemán David Hilbet propuso una lista de 23 problemas cuya solución consideraba esencial para el avance de las matemáticas. El programa que Hilbert tenía en mente era el desarrollo de un sistema lógico-matemático dentro del cual estuvieran inmersas todas las matemáticas y que fuera probablemente consistente. Es decir, sería posible probar que de los axiomas de este sistema no

se puede llegar a una contradicción como teorema; sin embargo en 1931, un joven lógico austriaco Kurt Gödel publicó un artículo en el cual demostró que no importa cuán fuerte sea el sistema formal que se invente, siempre habrá problemas que podrán ser formulados dentro del sistema, pero que no podrán resolverse dentro de él. Es decir, hay problemas matemáticos que, forzosamente, nunca podrán resolverse; no solamente porque no se haya descubierto cómo resolverlos, sino porque el hombre parece incapaz de encontrarles una solución.

El trabajo de Gödel destruyó el programa de Hilbert, pero condujo al rápido desarrollo de un campo descuidado de las matemáticas: determinar qué métodos son válidos en la resolución de problemas. Las cuestiones básicas, aquí, son exactamente lo que queremos decir cuando afirmamos que tenemos un algoritmo para resolver un problema, que podemos "calcular efectivamente" una función, o que podemos "enumerar efectivamente " un conjunto.

La palabra **algorismo** no apareció en el Diccionario de Nuevas Palabras de Webster sino hasta 1954, con el siguiente significado "proceso de hacer aritmética usando números arábigos". Gradualmente su forma y significado fue cambiando por representar confusión con la palabra aritmética. El cambio de algorismo a algoritmo es difícil de entender ya que la gente ha olvidado la derivación original de la palabra.

Un reciente diccionario matemático alemán VOLLSTÄNDIGES MATHEMATISCHES LEXICON da la siguiente definición para la palabra algoritmo: "bajo esta palabra se combinan las nociones de los cuatro tipos de cálculos aritméticos, llamados, adición, multiplicación, substracción y división". La frase latina algoritmo INFINITESIMAL fue empleada para denotar "métodos de cálculo con infinidad de pequeñas cantidades" y, fue inventada por Leibnitz.

Para 1950 la palabra algoritmo fue asociada con el algoritmo de Euclides, el cual es un proceso para encontrar el máximo común divisor de dos números.

Desde 1952 el desarrollo de las computadoras ha dado ímpetu a los estudios de la teoría de algoritmos y lógica.

Uno de los más recientes procedimientos para abordar la resolución de problemas fue el propuesto por el matemático ruso A. A. Markov en los primeros años de la década de los cincuenta, con lo que el llamaba "algoritmos normales" y que nosotros conocemos como "algoritmos de Markov". El tipo general de problemas que Markov atacó, es el de las transformaciones de cadenas de símbolos : Dada la sucesión A, transformarla a la sucesión B de un modo mecánico. Este problema es una extracción de muchos problemas comunes; por ejemplo sumar dos números A y B puede considerarse como el problema de transformar la sucesión "A+B" en la sucesión "C".

Actualmente un algoritmo es formulado y presentado como la solución de un problema, su significado es bastante similar al de fórmula, proceso, método, o técnica; sin embargo esta palabra implica aspectos un poco diferentes: es un conjunto finito de instrucciones que especifican el camino en el que las entradas se procesan en orden para producir resultados; el hecho de que la relación ENTRADA/SALIDA se complete depende no sólo de las instrucciones sino de la calidad de los datos, datos equivocados producen malos resultados.

1.3 ¿QUE ES EL ANALISIS DE ALGORITMOS?

El análisis de algoritmos es el estudio de todos los aspectos involucrados en la solución de un problema por computadora, desde su formulación preliminar, la programación, y así sucesivamente hasta la interpretación de los resultados obtenidos.

Es conveniente distinguir entre el análisis de un algoritmo en particular y el análisis de una clase de algoritmos. En el primer caso se investigan las características más importantes tales como el tiempo y la memoria empleada (complejidad), para determinar si ese algoritmo es o no óptimo en algún sentido. Este tipo de análisis se realiza desde los primeros días de la programación computacional.

En el análisis de una clase de algoritmos (algoritmos agrupados según problemas matemáticos bien definidos, por ejemplo la ordenación y la búsqueda), se estudia una familia entera de algoritmos para la solución de un problema particular y se selecciona el que resulte más eficiente para solucionar dicho problema, el realizar esta tarea de selección puede ser un proceso complicado que envuelva un sofisticado análisis matemático.

El análisis de algoritmos forma parte de las disciplinas más generales de la ciencia computacional y a grandes rasgos, engloba lo que sería la evaluación de la eficiencia de los algoritmos; dado que actualmente existen varios algoritmos disponibles para una aplicación particular, el análisis de algoritmos, nos ayuda a conocer cuál es el algoritmo que nos permite resolver de una forma óptima nuestro problema.

Es importante mencionar que en el análisis de un algoritmo se recurre a la notación matemática que se emplea con dos principales propósitos:

1) Describir una porción de un algoritmo y,

2) Analizar las características de ejecución de un algoritmo.

1.4 HERRAMIENTAS EMPLEADAS EN LA CONSTRUCCION DE ALGORITMOS EFICIENTES.

Un problema de gran importancia en la computación, es el encontrar representaciones de los objetos que se manipulan, objetos que pueden ser tan simples como un tipo de dato (Integer, Char),o tan complicados como las redes de comunicación (Gráficas). Este problema surge porque existen muchas formas de representar los datos,en términos de estructuras simples provistas directamente por un lenguaje de programación.

Siempre hay que tener presente que los programas consisten de **ALGORITMOS** y **ESTRUCTURAS DE DATOS** y un "buen" programa es la combinación adecuada de ambos. El elegir e implementar una estructura de datos es tan importante como las rutinas encargadas de su manipulación y tiene una influencia significativa en el costo y eficiencia de la implementación de un algoritmo, su empleo responsable produce algoritmos más claros y concisos y por tanto simplifican el programa total.

1.4.1 Clasificación de las estructuras de datos.

Las estructuras de datos más comunes se clasifican según Arthur H. Sedman (The Handbook of Computer and Computing) en dos grupos:

1) ESTRUCTURAS REGULARES.

2) ESTRUCTURAS COMPLEJAS.

Las estructuras regulares se conocen también con el nombre de **TIPO DE DATOS** y son un conjunto de ítems elementales o átomos. Un átomo usualmente consiste de un sólo elemento: un entero, un bit o un conjunto de ítems.

Los tipos de datos pueden ser clasificados como escalares y estructurados. Los datos escalares incluyen el tipo real, el tipo entero, el de doble precisión, el complejo, el lógico, los apuntadores y las etiquetas; se les denomina así por ser un conjunto ordenado en forma creciente (el real es el único tipo de dato considerado en esta clasificación que no cumple con esta propiedad).

Los datos estructurados son colecciones de elementos individuales de igual o distinto tipo de dato. Por ejemplo

```
Alumno : RECORD
            Nombre : STRING [30];
            Edad   : INTEGER;
         END;
```

Alumno en este caso, es un tipo de dato estructurado compuesto por los elementos nombre y edad que corresponden a tipos de datos distintos.

Las **ESTRUCTURAS IRREGULARES** o **ESTRUCTURAS DE DATOS** se encuentran definidas por los posibles caminos en los cuales los tipos de datos son lógicamente relacionados; las relaciones que pueden existir entre éstos son:

a) CONJUNTO: En este tipo de estructura no existe una relación directa entre los elementos, ésta se encuentra dada hacia el conjunto a quien pertenecen.

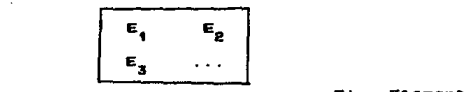


Fig 1.3 Ejemplo de un conjunto.

b) LINEAL: Este tipo de relación se presenta cuando los elementos guardan una relación de uno a uno.

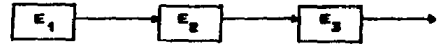


Fig 1.4 Ejemplo de una estructura lineal.

c) NO LINEAL: La relación que se establece entre los tipos de datos de una estructura es múltiple (Un elemento puede estar relacionado con dos o más elementos).

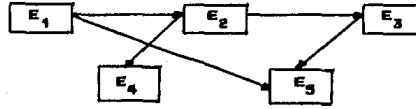


Fig 1.5 Ejemplo de una estructura no lineal.

1.4.2 Listas.

Una lista es una estructura de datos que tiene un número variable de elementos (objetos del mismo tipo, denominados también átomos) que se encuentran relacionados. Las listas deben estar almacenadas en un soporte direccionable (memoria central o periférico de acceso seleccionable).

Para hacer referencia a una lista se debe conocer donde comienza y donde termina. El tratamiento o manipulación de una lista exige poder:

a) Accesar al primer elemento (cabeza de la lista)

b) Accesar a otros elementos.

1.4.3 Formas de almacenar una lista.

Existen dos formas de almacenar una lista denominadas forma **contigua** y forma **ligada**.

En el almacenamiento contiguo los miembros de la lista se encuentran almacenados en localidades de memoria contigua. Esto es si S_1 es almacenado en la localidad L_1, S_2 en $L_2 = L_1 + d$, entonces S_n se encuentra almacenado en $L_n = L_1 + (n-1)d$ donde d es el número de localidades que se requieren para almacenar un sólo elemento de la lista.

Fig 1.6 Almacenamiento Contiguo.

Para implementar de esta forma las listas se requiere del empleo de arreglos, que son estructuras homogeneas, constituidas por elementos del mismo tipo, en la cual sus miembros son accesados por selectores enteros llámados indices. El tipo y tamaño del arreglo debe ser definido antes de iniciar el proceso.

La otra forma de almacenar una lista es la ligada; en la cual los componentes de la estructura se encuentran enlazados mediante apuntadores. En este tipo de almacenamiento cada elemento de la lista incluye un campo que indica donde se encuentra el siguiente elemento, lo que implica que el orden relativo de los componentes puede ser modificado al alterar los apuntadores.

Una lista enlazada no esta limitada a contener un número máximo de elementos por lo que puede expandir (si existe memoria disponible) o contraer su tamaño mientras se ejecuta el procedimiento.

La mayor ventaja de las listas ligadas es la utilización eficaz de la memoria, ya que puede emplear cualquier espacio vacio por no exigir posiciones secuenciales de memoria; por el contrario su mayor inconveniente es la lentitud del proceso, ya que es necesario ir recorriendo elemento a elemento a traves de apuntadores para acceder a un dato especifico.

Dos de las aplicaciones más comunes de las listas ligadas son:

- En un editor de lineas puede mantenerse una lista enlazada de nodos de linea, cada uno conteniendo un número de linea, una linea de texto y un apuntador al siguiente nodo de la lista. puesto que no se puede predecir cuantas lineas se necesitarán, ésta podria ser una aplicación adecuada para implementarla con nodos creados dinamicamente.

- Para implementar arreglos esparcidos (vectores o matrices con pocos elementos distintos de cero) y evitar el desperdicio excesivo de memoria.

1.4.4 Consideraciones sobre el almacenamiento contiguo y ligado.

- El almacenamiento contiguo es facil de implementar, pero es deficiente por la cantidad de localidades de memoria que utiliza ya que esto en muchos casos delimita el area de implementación.

- El almacenamiento ligado emplea espacio adicional de memoria para las ligas y esto podria ser un factor determinante en algunas situaciones; pero como la información no siempre emplea toda la memoria reservada para ella; la restante se utiliza para el campo apuntador.

- Cuando la información está almacenada en forma contigua, su manipulación resulta aparentemente más sencilla, sin embargo para retirar un elemento hay que desplazar una gran cantidad de ellos. En cambio en el almacenamiento ligado bastará con alterar las ligas.

- Para recuperar un elemento el almacenamiento contiguo es más eficiente ya que puede practicarse en forma directa; mientras que en el ligado se requiere un número mayor de comparaciones por el recorrido que se tiene que hacer (de acuerdo con el campo liga).

- En el almacenamiento ligado es más fácil unir o separar dos listas .

1.4.5 Clasificación de las listas.

De acuerdo con la relación que guardan entre sus elementos las listas se clasifican en lineales y no lineales.

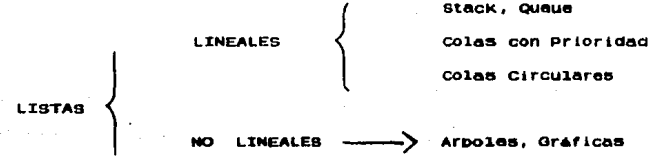


Fig 1.7 Clasificación de las listas.

Las listas NO LINEALES serán estudiadas en el capitulo 6.

1.4.6 Listas lineales.

Una lista lineal es una estructura, cuyos nodos están ordenados por un sólo criterio, y en donde el último y primer nodo no tienen sucesor ni antecesor respectivamente.

Formalmente una lista lineal se define como un conjunto de n >= 0 nodos X(1),X(2),...X(n) lo cual involucra una relación lineal (una dimensión) relativa a la posición de los nodos. Si n>0, X(1) es el primer nodo, donde 1<K<n el K-nodo X(K) es precedido por X(K-1) y seguido por X(K+1) y X(n) es el último nodo.

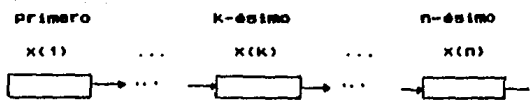


Fig 1.8 Representación gráfica de las listas

Dado que el tamaño de una lista lineal es variable la supresión e inserción de elementos son las operaciones principales, pero también se pueden realizar otras como:

- Determinar el tamaño de la lista.
- Acceder al K-ésimo nodo de la lista para examinarlo y cambiar si es preciso el contenido de sus campos.
- Insertar un nuevo nodo antes del K-ésimo.
- Borrar el K-ésimo nodo.
- Combinar una o más listas en una sola lista.
- Dividir una lista en dos o más listas.
- Determinar el número de nodos de la lista.
- Buscar la ocurrencia de un nodo con un particular valor en un campo determinado.
- Copiar una lista lineal.
- Ordenar los nodos de una lista basados en ciertos campos de los nodos.

Al manipular las listas, se pueden presentar dos tipos de errores denominados, OVERFLOW y UNDERFLOW que significan un exceso o deficiencia de elementos. En el caso de un error UNDERFLOW se tratan de borrar elementos que no existen y en el caso de un error OVERFLOW se intenta realizar una inserción cuando ya no tenemos memoria disponible para hacerlo. Es importante tener presente que todas las operaciones asociadas a una lista se realizan en la memoria principal.

1.4.7 Tipos de listas lineales.

La organización de los elementos en una estructura generalmente tiene el propósito de ejecutar operaciones con la estructura, más

que con sus componentes y de acuerdo con su comportamiento (forma en la cual realizan sus operaciones) las estructuras lineales se clasifican en stack, queue, cola con prioridad, colas circulares y, colas doblemente ligadas.

A. STACK.

Si una estructura sólo puede operar con su elemento extremo se obtiene la estructura stack. Un stack es una estructura tipo LIFO (LAST IN, FIRST OUT) lo cual implica que los elementos se sacan de la lista en el orden inverso al que se insertaron en ella.

Este tipo de estructura también es llamada push-down, almacenamiento inverso, pila, lista último en entrar primero en salir y aún listas yo-yo.

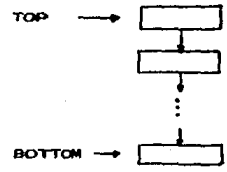


Fig 1.9 Representación gráfica del STACK

El primer elemento accesible del stack se referencia por el apuntador TOP; el apuntador BOTTOM indica el último elemento accesible y no puede ser borrado hasta que todos los elementos sobre él sean eliminados.

Por razones históricas, las dos operaciones primarias en pilas se llaman usualmente PUSH (insertar) y POP (borrar).

El esquema de un stack empleando almacenamiento dinámico sería:

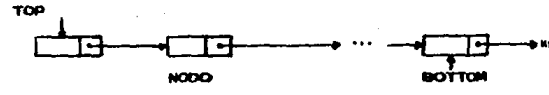


Fig 1.10 Representación ligada del STACK.

Donde cada nodo está constituido por dos campos (un campo tipo apuntador y otro campo que contiene la información:

El almacenamiento secuencial para un stack es muy simple, sólo se requiere de dos variables, una que indique el último elemento insertado (TOP) y otra que indique el número máximo de elementos (MAX-PILA); Cuando la variable TOP es igual a cero la pila está vacía

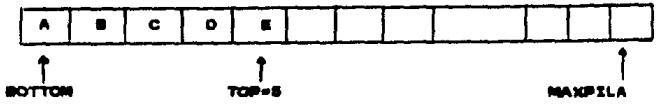


Fig 1.11 Representación ligada del STACK.

APLICACIONES DEL STACK.

- Una pila es la estructura de datos adecuada cuando la información debe guardarse y luego recuperarse en orden inverso. Una situación que requera volver atrás alguna posición puede ser una buena ocasión para usar una pila. Por ejemplo en la resolución de un laberinto, se puede dar con un muro y necesitar volver atrás buscando una salida. Si se emplea el stack para guardar los caminos alternativos a aquellos por los que pasó, podría volver a la ruta en una posición anterior.

- En procesos recursivos o llamadas a procedimientos. Muchos sistemas utilizan una pila para llevar las direcciones de vuelta, los valores de los parámetros o direcciones e, información utilizada en los subprogramas. Puesto que el orden de las llamadas a los procedimientos dentro de un programa es dinámico, la pila que almacena estos datos puede crecer y disminuir durante la ejecución de acuerdo con el nivel de anidamiento de los subprogramas.

8. COLAS SIMPLES (QUEUES).

Una cola es una estructura de datos lineal en la que se emplean sus dos extremos para realizar las operaciones; lo cual implica que deben existir dos apuntadores: el apuntador TAIL que indica el extremo empleado para insertar, y el apuntador HEAD que indica el extremo por el que se podrá suprimir y realizar generalmente todos los accesos.

Las colas son estructuras tipo FIFO (FIRST IN FIRST OUT), ya que el primer elemento que llegue a la cola será el primer elemento en salir de ella. El esquema de este tipo de estructura en forma dinámica es:

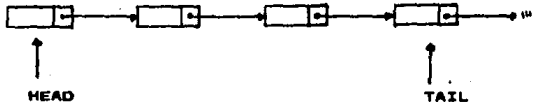


Fig 1.12 Representación ligada de una cola simple.

Si se desea implementar una cola en forma estática se requiere de tres variables: HEAD, TAIL que guardan la posición de los elementos extremos y MAX que indica el número máximo de elementos que podría contener la cola.

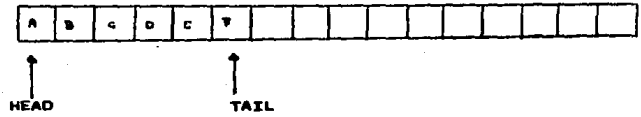


Fig 1.13

Si después de realizar varias operaciones nuestra estructura tiene la siguiente forma:

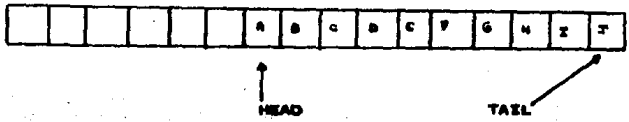


Fig 1.14

y se desea realizar una operación de inserción es necesario aplicar un desplazamiento al principio de la lista para poder hacerlo.

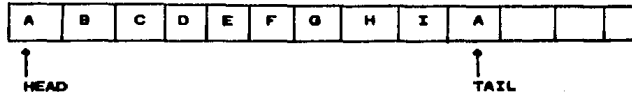


Fig 1.15

APLICACIONES DE UNA QUEUE.

- Una aplicación donde las colas figuran como la estructura de datos predominante es la simulación por computadora de una situación del mundo real. De hecho, el sistema del mundo real se llama sistema de cola y se encuentra formado por servidores y colas de objetos a servir. El factor que se examina es el tiempo de espera.

Los objetivos de un sistema de colas es emplear al máximo a los servidores y mantener el tiempo medio de espera dentro de unos límites razonables. Estos objetivos necesitan frecuentemente de un compromiso entre el costo y la satisfacción del cliente.

- Las colas también se emplean en muchas partes del sistema operativo, en el cual existe un programa que clasifica y asigna los recursos de una computadora. Uno de esos recursos es la propia unidad de procesamiento.

- Todos los dispositivos de ENTRADA/SALIDA tienen su propia cola de peticiones para imprimir, leer o escribir en esos dispositivos.

C. COLA DOBLE (DEQUE).

La cola doble es una estructura de datos lineal que permite que las operaciones de borrar o insertar un elemento se realicen por ambos extremos. Por la forma en la cual se opera esta estructura se puede tener el comportamiento de un stack o de una cola simple.

Existen dos variaciones de las colas dobles:

1) COLA DOBLE DE SALIDA RESTRINGIDA: Sólo se permite la eliminación de elementos por un sólo extremo y la inserción por ambos.

2) COLA DOBLE DE ENTRADA RESTRINGIDA: Las inserciones se realizan por un solo extremo y las eliminaciones por ambos.

El esquema de este tipo de estructura sería:

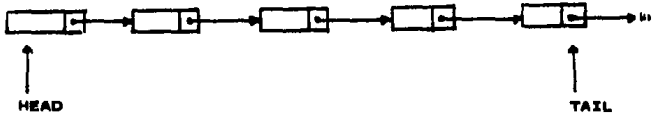


Fig 1.16.

Las DEQUE o colas dobles son una generalización de las QUEUES y los STACK.

D. COLAS CON PRIORIDAD.

Una cola con prioridad es un conjunto de elementos en los cuales cada uno tiene asignado una prioridad. La forma en la cual se anexa o elimina un elemento de este tipo de estructura sigue las siguientes reglas:

1) El elemento de mayor prioridad se procesa primero.

2) Dos elementos con la misma prioridad son procesados de acuerdo con el orden en que fueron anexados a la cola

3) Los números que representan la prioridad tendrán el siguiente significado: "a menor número, mayor prioridad"

El esquema de una cola con prioridad empleando almacenamiento dinámico es:

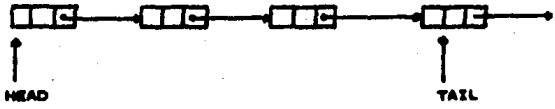


Fig 1.17

Donde cada nodo conteun nuevo campo en el que se indica la prioridad para ser procesado.

La implementación de este tipo de estructuras en forma estática es un poco más complicada ya que requiere de una matriz (prioridad, orden) y dos vectores que manejen los apuntadores TAIL y HEAD de cada nivel de prioridad.

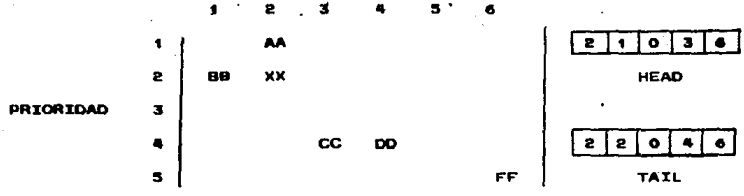


Fig 1.18 Cola con prioridad en forma estática

Cada localidad de los arreglos corresponde a un nivel de prioridad.

Al manejar una cola con prioridades en forma contigua se debe tener cuidado porque los errores de UNDERFLOW y OVERFLOW se presntan para cada nivel de prioridad:

APLICACIONES DE UNA COLA CON PRIORIDAD.

- Un prototipo de una cola con prioridades es un sistema con tiempo compartido en el cual los programas tienen cierta jerarquia.

- En la vida diaria la prioridad en las colas de espera se logra ya sea por la finalidad, parentesco o, por alguna propina.

E. LISTAS DOBLEMENTE LIGADAS.

Las listas ligadasen una sola direccion tienen un gran inconveniente: deben recorrerse de izquierda a derecha, pero debido a que en muchas ocasiones se necesita recorrerlas en ambas direcciones, se crearon las listas doblemente ligadas; lo cual viene a darle mayor flexibilidad a la manipulación de las listas.

En este tipo de estructura cada nodo incluye dos campos tipo apuntador llamados por ejemplo liga izquierda y liga derecha que señalan a su nodo antecesor y sucesor respectivamente.

Gráficamente:

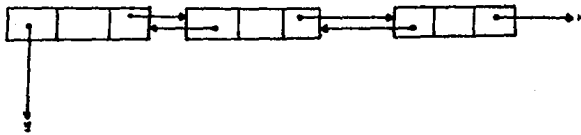


Fig 1.19

En cualquiera de las estructuras vistas anteriormente se puede tener doble campo tipo apuntador pero es necesario tener presente la utilidad que esto representa; por ejemplo, un stack puede ser doblemente ligado pero una liga nunca se empleará.

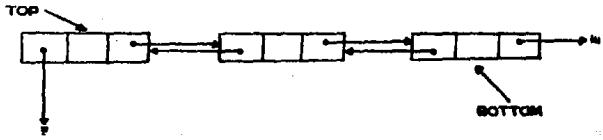


Fig 1.20 Stack doblemente ligado

APLICACIONES DE LAS LISTAS DOBLEMENTE LIGADAS.

- En cualquier proceso donde se requiera recorrer la lista en ambos sentidos por ejemplo: En el caso de una base de datos que permite al usuario explotar la lista en cualquier dirección. Debido a que la lista se puede leer empleando cualquiera de los dos enlaces, si en un momento dado un enlace no es valido, la lista se puede reconstruir usando el otro enlace. Esto tiene sentido sólo en caso de un fallo de equipo.

F. LISTAS CIRCULARES.

Una lista circular es una estructura de datos que tiene la propiedad de que el último nodo de la lista se encuentra relacionado con el primer elemento de la misma.

En el almacenamiento ligado de este tipo de estructura no es necesario llevar el control del primer o último elemento, sólo del actual (PTR), lo cual es equivalente a una lista con los punteros (HEAD y TAIL).

Para poder insertar o borrar un elemento de este tipo de estructura, se necesita hacerlo después del que está siendo apuntado por PTR que es el único elemento del cual se tiene información.

Si el apuntador PTR se mantiene fijo la lista se comporta como un stack.

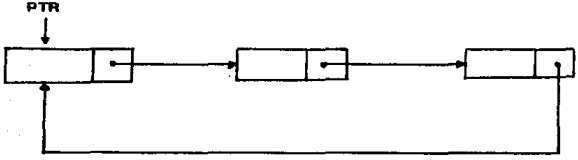


Fig. 1.21 Lista Circular.

En el almacenamiento contiguo este tipo de estructura viene a solucionar el problema de desplazamiento de los nodos en los arreglos, lo cual puede ser una operación muy costosa si la cola es muy grande.

Ejemplo:

Si después de una serie de operaciones, se tiene la siguiente estructura

HEAD = 4

TAIL = 10

Fig 1.22

y se desean realizar más inserciones, se toman las localidades disponibles al inicio del arreglo

HEAD = 4

TAIL = 2

Fig 1.23

APLICACIONES DE LAS LISTAS CIRCULARES.

- La aplicación más común de una cola circular puede ser en los sistemas operativos que regulan la información que se lee y se escribe en archivos de disco o en la consola.

- En los programas de aplicaciones en tiempo real, el programa puede realizar una tarea mientras el usuario introduce datos por el teclado. Muchos procesadores de texto hacen esto cuando se reformatea un párrafo o se ajusta una línea por ello existe un breve período en el cual lo que se está tecleando no se visualiza hasta que el proceso que se está realizando termine. Sin embargo el programa de aplicación tiene que continuar checando el teclado durante la ejecución del otro proceso, y si se ha pulsado una tecla, se colocaría rápidamente en la cola y continuar el proceso. Cuando el proceso se ha completado, los caracteres se recuperan de la cola y se continúa de forma normal.

Nota : Algunos lenguajes como por ejemplo FORTRAN no tienen apuntadores pero este tipo de datos pueden implementarse en almacenamiento contiguo; o simulando apuntadores mediante cursores (enteros que indican la posición del elemento siguiente en el arreglo),ejemplo:

```
VAR

  Espacio : ARRAY [1..Max3] of record

            Elemento : tipo de elemento;

            Sig      INTEGER;

            END;
```

1.5 PROBLEMA-ALGORITMO-PROGRAMA.

Escribir un programa para resolver un problema comprende varios pasos que van desde la formulación del problema, el diseño de la solución, la implementación, prueba y documentación, hasta la evaluación de la solución.

Los pasos que proponemos para el desarrollo de un buen programa son:

A. DEFINICION DEL PROBLEMA.

Antes de entender el problema, es necesario tener una definición precisa de éste; tal condición por sí misma no es suficiente para entenderlo, pero es absolutamente indispensable.

Esta fase, muchas veces se pasa por alto siendo absolutamente necesaria para una compresión exacta y clara de lo que se requiere en realidad del programa, para poder crear un ambiente propicio de desarrollo.

Para desarrollar una definición precisa es necesario realizarse preguntas como:

¿Todo el vocabulario empleado en la formulación se entiende?

¿Cuál es el resultado que deseo?

¿Cómo podría reconocer la solución?

¿Qué información me están dando? ¿Qué información voy a emplear?

¿Qué suposiciones se hicieron?

y demás preguntas que dependerán básicamente del problema.

B. DESARROLLO DE UN MODELO.

Una vez que el problema ha sido establecido en forma clara, se debe formular un modelo. Este es un paso muy importante en todo proceso de solución y debe someterse a una considerable reflexión, ya que el modelo diseñado tendrá una influencia substancial en la solución del problema.

Existen dos preguntas básicas que es necesario realizar para escoger un modelo:

¿Cuales son las estructuras de datos que mejor se acoplan al problema? y,

¿Existen otros problemas resueltos, que se parecen a éste?

Para poder escoger las estructuras de datos es necesario entender la relación que existe entre los datos importantes en la solución del problema y decidir las operaciones que se van a desarrollar con ellos, sin embargo esta selección puede estar influenciada por los siguientes factores:

- Tener un conocimiento limitado respecto a las estructuras de datos.

- La conveniencia propia de la representación

- La simplicidad computacional.

Se dice que se tiene un modelo candidato si las siguientes preguntas se contestan en forma afirmativa:

¿Las estructuras de datos empleadas, representan en forma clara toda la información importante del problema?

¿Es posible reconocer todas las relaciones que se establecen entre los objetos?

¿Es posible trabajar con el modelo? ¿Es razonable su manipulación?

C. DISEÑO DE UN ALGORITMO.

Una vez que el problema se comprendio y se desarrollo un modelo se procede a formular un algoritmo basado en ese modelo.

Las versiones iniciales de un algoritmo a menudo se encuentran mezcladas en proposiciones generales que deben convertirse en instrucciones más pequeñas y definidas en forma concreta, por lo que se propone representar el algoritmo de una forma adecuada.

1) FORMAS DE REPRESENTAR UN ALGORITMO.

El lenguaje humano nos proporciona el medio menos aceptable para la comunicación de alguna tarea ya que fácilmente pueden surgir ambigüedades y malas interpretaciones, a menos que se recurra a una cuidadosa elección de las palabras empleadas en la descripción del proceso. Estas complicaciones, se pueden evitar adoptando ciertas convenciones, que nos permitan describir el algoritmo de otra forma tal como el empleo de DIAGRAMAS DE FLUJO y los PSEUDOCODIGOS.

a) Diagramas de Flujo (Representación Gráfica):

Un diagrama de flujo es una colección de dibujos o símbolos, cada uno representa un tipo de actividad diferente; se encuentran conectados por segmentos de línea provistos de flechas, que indican un sentido de dirección. Los detalles específicos que se requieren para efectuar las diferentes actividades se colocan dentro de los dibujos, según a convenciones bien establecidas.

Para representar un algoritmo por medio de diagramas de flujo se emplean los siguientes símbolos:

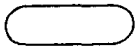
SIMBOLO	NOMBRE	FUNCION
	Terminal	Inicio o final de un programa principal.
	Proceso	Asignación de un valor a una variable.
	Decisión	Expresa ramificación del flujo lógico.
	Documento	Señala que la información se imprime en papel.
	Entrada	Los datos se alimentan desde el teclado.
	Preparación	Inicio de un procedimiento repetitivo.

Fig 1.24 Simbolos empleados en la representación gráfica

La ventaja más significativa de este tipo de diagramas es la representación clara del flujo de control del algoritmo, es decir, la secuencia en la que se van a llevar a cabo las operaciones.

Los diagramas de flujo son apropiados para el método de diseño TOP-DOWN (de arriba hacia abajo), en la que se establece inicialmente la estrategia general del algoritmo y los refinamientos se introducen después.

b) Pseudocódigos (Representación Inscrita):

En este tipo de representación se describe el flujo del programa utilizando el lenguaje inscrito y la notación algebraica.

Al escribir los algoritmos mediante esta representación se recomienda tener presente las siguientes convenciones:

- La indicación clara del principio y fin del proceso.

- El uso de palabras claves en mayúsculas.

- Sangrar por la izquierda las instrucciones que se encuentran bajo el control de una secuencia principal.

El algoritmo seleccionado en esta fase puede ser bien, uno desarrollado previamente (obtenido de alguna revista o libro especializado), o uno ideado por la persona interesada.

D. DEPURACION O CORRECCION DE UN ALGORITMO.

Uno de los pasos más tediosos y difíciles en el desarrollo de un algoritmo es el verificar que el proceso sea el correcto, la forma más común para comprobar esto es probandolo manualmente para diferentes casos (distintos datos)

E. CODIFICACION E IMPLEMENTACION.

Una vez que se tiene el problema en [illegible] de una secuencia de pasos y nos convencimos de que son correctos, es tiempo de codificar e introducir el algoritmo (programa) a la computadora

La selección del lenguaje de programación es importante, y deberá estar sujeto a las siguientes consideraciones:

a) Naturaleza del problema.

b) Lenguajes de programación disponibles (Si se tienen más de dos lenguajes que cumplan los requisitos para ser empleados, se recomienda aquel que permita expresar de una manera natural las operaciones que se desean realizar con los datos).

c) Limitaciones del equipo.

El problema que se puede presentar al codificar un algoritmo es que determinados pasos de este no pueden ser transferidos en forma directa; ademas para poder realizar la codificación es necesario contestar el siguiente tipo de preguntas:

¿Cuáles son las variables?

¿Cuáles son los tipos?

¿Podríamos usar listas ligadas?

¿Qué subrutinas son necesarias?

Al desarrollar la codificación del programa se debe tener presente que su algoritmo respectivo fue realizado en forma

sistemática y lógica, y por lo tanto, el programa final deberá presentar las siguientes características:

- El programa debe ser fácil de entender ya que sus partes no fueron realizadas al azar.

- Las estructuras de datos empleadas deben ser transparentes ya que fueron desarrolladas mucho antes, de que el programa actual fuera escrito.

- La cantidad de tiempo requerido por el programa final debe ser reducido.

F. ELIMINACION DE ERRORES (DEPURACION).

Por lo regular siempre será necesario corregir y detectar los inevitables errores de ejecución y lógica que se presentan en los programas implementados, debido a descuidos o a una mala codificación.

G. ANALISIS Y COMPLEJIDAD DE UN ALGORITMO.

Cuando un programa se va a ejecutar varias veces y/o por muchas personas es crucial su eficiencia, lo cual, es posible medir mediante un buen análisis.

Existen varias razones para analizar un algoritmo, la principal es que, es necesario tener estimaciones de la memoria y tiempo que el algoritmo requiere (dos recursos relativamente escasos y caros); así como también el desear tener cantidades estándares para comparar en un momento dado algoritmos que resuelvan el mismo problema.

Este paso es importante, porque en él se investigan los posibles algoritmos que resuelven la misma clase de problemas y se determina cual de ellos es el mejor para la solución del problema en cuestión.

H. PRUEBA Y VALIDACION.

Una vez que el programa se ha implementado y corregido se procede a realizar pruebas de este, que podrían ser definitivas como la verificación experimental de lo que el programa debe hacer, en otras palabras esto significa establecer sus límites, pero ¿Cómo escoger los datos que debe admitir como entrada? la respuesta a esta pregunta depende no solo de la complejidad del programa, sino también del tiempo disponible para probarlo.

I. DOCUMENTACION.

Realmente la documentación no es uno de los últimos pasos, ya que ésta se podría realizar conforme se desarrolla el programa. La razón más obvia para documentar un programa es que éste, pueda ser entendido en forma fácil por otros usuarios.

La documentación consta de la descripción del algoritmo, diagrama de flujo, descripción de procesos, flujo de datos, organización del programa, así como de todos los elementos que nos ayuden a entender cómo se llegó a la solución del problema.

El nivel de documentación de un programa dependerá del tipo de problema que se desea resolver:

- Si se trata de un problema, cuyo programa se empleará para correrse una o muy pocas veces, se requerirá muy poca documentación.

- Si el programa fue diseñado para un pequeño grupo de especialistas requerirá de un manual de usuario y mensajes apropiados de error (por ejemplo la maquina de Turing).

- Los programas que van a ser empleados por una gran cantidad de usuarios, por ejemplo las subrutinas y funciones de FORTRAN, requieren de buena documentación en todos sus niveles; pero

- Los programas extremadamente grandes, complejos y diseñados para ser corridos en mucho tiempo, requieren documentación bien detallada en todos los niveles, incluyendo manuales de usuario; por ejemplo la actualización de registros de una compañía de seguros.

J. MANTENIMIENTO.

Este último punto se refiere al hecho de que una vez que se hizo el programa, éste no es una entidad estática, que deberá permanecer sin alteraciones. Generalmente un programa empleado en el mundo real deberá sufrir cambios debido a situaciones no previstas, problemas nuevos que es necesario resolver, cambios en su ambiente de trabajo, y muchas otras causas. A las modificaciones que sufren los programas durante su empleo se les denomina MANTENIMIENTO.

1.6 ¿COMO ELEGIR UN ALGORITMO?

Uno de los principales objetivos, al resolver un problema es encontrar su solución y no se diga la solución más eficiente; para ello se recurre muchas veces a procesos guiados por la intuición y el conocimiento que se tiene del problema (Método de prueba y error). Una vez que se conoce perfectamente el problema se procede a buscar "un buen algoritmo" para darle solución, pero, ¿qué se entiende por un buen algoritmo? No es fácil responder a esta pregunta, ya que existen muchos criterios para definir este término, criterios que incluyan cuestiones muy

subjetivas como simplicidad, claridad, adecuación a los datos manejados, adaptabilidad del algoritmo a la computadora, y la elegancia; siendo el criterio más objetivo (lo cual no implica que sea el más importante) el de la eficiencia en tiempo de ejecución, que puede ser expresado en términos de la cantidad de tiempo de ejecución de cada paso.

Dado que diferentes algoritmos y procedimientos pueden producir la solución de un problema determinado ¿cómo se debe elegir uno de ellos? La selección de uno o más algoritmos depende de varios factores, entre ellos:

- Si el programa se va a emplear muchas veces, el costo de ejecución del programa puede superar en mucho el de escritura, en especial si en la mayor parte de las ejecuciones se dan entradas de gran tamaño. En este caso, es más ventajoso desde el punto de vista económico, realizar o seleccionar un algoritmo complejo, que emplee eficientemente los recursos de la computadora y, en especial, que se ejecute con la mayor rapidez posible (su tiempo de ejecución, debe ser significativamente menor que el de un algoritmo más evidente). En algunos casos y en determinadas situaciones quizá, es conveniente aplicar primero un algoritmo simple con el objeto de determinar el beneficio real que se obtendría escribiendo o seleccionando un programa más complicado, por ejemplo en la construcción de un sistema complejo, a menudo es deseable aplicar un prototipo sencillo, en el cual se puedan efectuar simulaciones y mediciones, antes de dedicarse al diseño definitivo.

- Si el programa se va a ejecutar sólo con entradas **pequeñas**, la velocidad de crecimiento del tiempo de ejecución puede ser menos importante que el factor constante del tiempo de ejecución (Ver capítulo VII). Determinar que es una entrada **pequeña**, depende de los tiempos de ejecución exactos de los algoritmos implicados. Hay algoritmos como el de la multiplicación de enteros de Schonhage y Strossen (1971), que son asintóticamente los más eficientes para sus problemas, pero nunca se han llevado a la práctica (ni siquiera con los problemas más grandes) debido a que su constante de proporcionalidad es demasiado grande comparada con otros algoritmos "menos" eficientes.

- Un algoritmo eficiente pero complicado puede no ser apropiado, porque posteriormente la persona encargada del mantenimiento puede ser alguien distinto al desarrollador. Aún cuando se espera que al difundir el conocimiento de las principales técnicas de diseño de algoritmos eficientes se puedan utilizar algoritmos con un alto grado de complejidad, debe considerarse la probabilidad de que un programa resulte inútil debido a que nadie entienda sus útiles y eficientes rutinas.

- Existen ejemplos de algoritmos eficientes que ocupan demasiado espacio para ser aplicados sin almacenamiento secundario, lo cual puede anular su eficiencia.

- Si se trata de algoritmos numéricos, la precisión y estabilidad son tan importantes como la eficiencia.

Por último, al seleccionar un algoritmo es necesario tener presente que aún el proceso más deficiente, puede ser transformado en el algoritmo más eficiente a nuestro problema con sólo un poco de esfuerzo.

CAPITULO II

ALGORITMOS SEMINUMERICOS

Los algoritmos seminuméricos son algoritmos "simples" que desarrollan métodos elementales como generar números aleatorios, desarrollar aritmética básica (operar enteros, polinomios o matrices) y otros métodos empleados para resolver una gran cantidad de problemas.

El propósito de este capítulo es realizar un estudio general de este tipo de algoritmos con el objetivo de que el lector tenga presente que el empleo de algoritmos seminuméricos eficientes es esencial en muchas aplicaciones computacionales; sin embargo, debido a la amplitud del tema sólo se tratan algunos ejemplos ya que en general estos algoritmos son métodos de conocimiento básico en la carrera de Matemáticas Aplicadas y Computación y su desarrollo podría ser muy amplio y tedioso. Para las personas interesadas en ampliar su conocimiento con respecto a este tema se recomienda principalmente el libro: THE ART OF COMPUTER PROGRAMMING (Seminumerical Algorithms) de Donald E. Knuth, Volumen 2.

2.1 DEFINICION Y CARACTERISTICAS DE LOS ALGORITMOS SEMINUMERICOS.

Un algoritmo es SEMINUMERICO si:

1. Sus objetos son numéricos, es decir, si los datos que opera dicho algoritmo son números.

2. Sus efectos son numéricos, o sea, si sus salidas son números, lo cual quizá implique que las etapas del algoritmo involucran algún tipo de operaciones matemáticas, pero hay que tener presente que no siempre un algoritmo que genera datos numéricos ejecuta cálculos aritméticos.

Este tipo de algoritmos que tratan con números (procesan y generan números), cumplen además con las propiedades **Seminuméricas** porque se dice que se encuentran en la frontera de los algoritmos numéricos y algoritmos simbólicos dado que cada proceso no pretende únicamente dar respuestas inmediatas a un problema específico, sino que intenta mezclar satisfactoriamente todas las instrucciones de un algoritmo, con las operaciones internas de una computadora digital.

En los algoritmos seminuméricos existe (en forma implícita) un contador que permite conocer apriori el número de veces que se va a ejecutar; lo cual implica en un momento dado el poder calcular su costo, la cantidad de memoria que requiere y el tiempo de ejecución en la máquina. La minimización de estos tres factores es primordial en este tipo de algoritmos, dado que normalmente se encuentran implementados en el Hardware de la computadora y aún cuando el usuario no tiene que preocuparse de los detalles relacionados con los mismos, se debe tener presente que el uso de algoritmos "prefabricados" limita el uso de cifras a determinada longitud y por lo tanto si se desea trabajar con números de longitud mayor a la permitida, es necesario abastecer a la computadora de los algoritmos necesarios, lo cual implica conocer la forma en la cual este tipo de algoritmos realizan su proceso.

2.2 DESCRIPCION Y EJEMPLOS DE ALGUNOS ALGORITMOS SEMINUMERICOS.

2.2.1 Aritmética básica.

Los algoritmos empleados para realizar las operaciones aritméticas básicas, tienen una historia muy antigua, datan del estudio original de los algoritmos en el trabajo del matemático árabe Alkhowârizmi; y aún cuando muchas veces se ve a la aritmética como algo trivial, es de hecho un nivel subjetivo que

ha jugado un papel muy importante en la historia del mundo ya que sirve como base para desarrollar aspectos matemáticos más complicados, tales como los polinomios y las matrices.

SISTEMA NUMERICO.

La notación numérica base b (alternativamente llamada radix b) se define por la regla:

$$(\ldots a_3 a_2 a_1 a_0 a_{-1} a_{-2} \ldots)_b = \ldots + a_3 b^3 + a_2 b^2 + a_1 b^1 + a_0 + a_{-1} b^{-1} + a_{-2} b^{-2} + \ldots$$

por ejemplo:

$$(520.3)_6 = 5(6)^2 + 2(6) + 0 + 1/3(6)^{-1} = 192.5$$

El sistema numérico que empleamos convencionalmente, es por supuesto, un caso especial de la notación anterior cuando b=10 y cuando las ai son seleccionadas de los digitos decimales 0,1,2,3,4,5,6,7,8,9.

ARITMETICA DE PUNTO FLOTANTE.

Cuando el punto decimal tiene un lugar fijo a través de toda la ejecución del programa se dice que se trabaja con una aritmética de punto fijo, sin embargo para muchas aplicaciones es conveniente que el lugar del punto sea variable o flotante, es decir, que cuando el programa se esté ejecutando en cada número se tenga una indicación de la posición correspondiente del punto. Esto se emplea en muchos cálculos científicos, especialmente para expresar números muy grandes: por ejemplo $N=6.02250\times10^{23}$ o números muy pequeños como $N=1.0545\times10^{-27}$.

La forma general de representar un número de esta forma es (e,f)=fxb##e-q, es decir, cada número se representa como un número multiplicado por la base de su notación elevada a la potencia e menos un número q (indica la potencia mayor). Donde e es un entero que tiene un rango específico y f es una fracción signada de una determinada longitud en la cual $1/b \le f < 1$.

Por ejemplo:

Expresar a N y H con la notación anterior empleando a q=50 y b=10 y la longitud de f igual a 8.

$N=6.02250\times10^{23}$	equivalente a	$0.\,60225000\times10^{24}$	e-q = 24
			e-50= 24
N=(74,+0.60225000)			e = 74
$H=1.0545\times10^{-27}$	equivalente a	$0.\,10545000\times10^{-26}$	e-q =-26
			e-50=-26

Los dos componentes e y f de un número de punto flotante se llaman exponente y parte fraccional respectivamente (también llamados carácteristica y mantisa) y se dice que está normalizado si el dígito más significativo de la representación de f es distinto de cero o sea:

1/b <= /f/ < 1

o si f=0 y e tiene el valor más pequeño posible.

Para poder operar con números de punto flotante por ejemplo,

Adición de punto flotante

(eu,fu) (+) (ev,fv) = (ew,fw)

se emplean los símbolos: ⊕, ⊖, ⊗ y ⊘ dado que la aritmética de punto flotante es aproximadamente igual (no exacta). La idea que se involucra en la adición de punto flotante es muy sencilla: Es necesario que ambos números tengan el mismo exponente y el punto decimal esté alineado.

Para poder realizar el algoritmo que ejecute este proceso se asume que eu>=ev, y que (eu,fu) y (ev,fv) están normalizados.

Algoritmo para normalizar números de punto flotante.

Dado un número (e,f), su parte exponencial e y su parte fraccional f, se convierte a una forma normalizada redondeando si es necesario a p dígitos, se asume /f/<b y que b es par (b es la base del número).

1. Si /f/ >= 1 (la parte fraccional se excede) ir al paso 4.

   Si f=0 asignar a ew el valor más pequeño posible e ir al paso 5.

2. Si /f/ >= 1/b ir al paso 5.

3. Realizar un corrimiento a la izquierda en una posición es decir multiplicar f por b y decrementar en 1 e. Ir al paso 2.

4. Realizar un corrimiento a la derecha en una posición (dividir f entre b), incrementar e en 1.

5. Redondear f a p lugares (Se asume que f=b$$-p(b$$pf+1/2). Cuando f>0 y f=b$$-p((b$$p)f-1/2)). Cuando p<0 se pueden emplear otras reglas de redondeo.

En este algoritmo es importante hacer notar que la operación de redondeo puede hacer que /f/=1; en tal caso se debe regresar al paso 4.

Algoritmo para sumar dos números de punto flotante.

Dada la base B, q, p dígitos y dos números normalizados de punto flotante u=(eu,fu), v=(ev,fv), se puede realizar su suma empleando el siguiente algoritmo (también puede ser utilizado para la substracción si -v se sustituye por v).

1. Separar la parte exponencial y fraccional de las representaciones de u y v.

2. Si eu < ev intercambiar u y v.

3. ew = ev.

4. Si eu-ev >= MAXIMO (existe una gran diferencia en los exponentes) entonces fw = fu ir al paso 7 (en este caso se va a normalizar un número que posiblemente no se haya normalizado).

5. Realizar un corrimiento fv a la derecha eu-ev lugares es decir dividir fv entre Bes(eu-ev), lo cual en un momento dado puede implicar un corrimiento de mas de p+1 lugares requiriendo con ello un espacio adicional (en el próximo paso) de 2p+1 dígitos de base B a la derecha del punto para una operación exitosa.

6. fw = fu + fv

7. Normalizar (ew,fw)

8. Terminar.

El tiempo de ejecución para este algoritmo depende de la diferencia inicial entre los exponentes y del número de pasos requeridos para normalizar el número.

Algoritmo para multiplicar dos números de punto flotante.

Dada la base B, q, p dígitos y dos números normalizados de punto flotante u=(eu,fu) y v=(ev,fv) se forma el producto w=u ⊗ v o el cociente w=u⊘v donde p es impar.

1. Separar la parte fraccional y la parte exponencial de las representaciones de u y v (a veces es conveniente pero no necesario comparar los operandos con cero durante este paso).

2. ew = eu + ev - q fw=fu*fv (para la multiplicacion)

   ew = eu - ev + q+1 fw=(1/Bfu)/fv (para la división)

Dado que se asume que los números de entrada están normalizados nunca fw=0 o $1/B^2 <= |fw| < 1$, lo cual implica que nunca existirá un error de división entre cero. Necesariamente fw debe ser truncada a p dígitos.

3. Normalizar (ew,fw)

FALLAS COMUNES DEL PROGRAMADOR O DEL DISEÑO DE LA COMPUTADORA CUANDO SE IMPLEMENTAN LAS RUTINAS DE PUNTO FLOTANTE.

PERDIDA DEL SIGNO: En muchas computadoras, las instrucciones de corrimiento de bits en los registros pueden afectar el signo por lo que deben emplearse cuidadosamente.

OVERFLOW, UNDERFLOW: Cuando se presenta este tipo de error, implica que **ew** tiene un dato erróneo, esto puede ocurrir tanto en las operaciones de adición como en las de multiplicación.

Cuando se realiza un corrimiento a la derecha es importante introducir ceros a la izquierda.

Redondear antes de normalizar puede ocasionar errores prematuros en dígitos de posición equivocada.

Por lo general la aritmética de punto flotante es parte del hardware por lo que puede incluir algunas anomalías con resultados muy pobres en ciertas circunstancias.

ALGORITMOS CLASICOS.

Se les denomina algoritmos clásicos a las operaciones aritméticas con enteros, en el cual cada dígito del número debe ser menor que **b** (la base empleada). Dichas operaciones involucran en cada paso operaciones primitivas entre dos dígitos.

OPERACIONES PRIMITIVAS.

a) Sumar o restar enteros de longitud 1, que genera como resultado un entero de longitud 1 y un acarreo.

b) Multiplicar dos enteros de longitud 1, que genera un resultado de longitud 2.

c) Dividir un entero de longitud 2 entre un entero de longitud 1, lo cual implica que el cociente es un entero de longitud 1 y el residuo de longitud 1.

Algoritmo de adición.

Dados dos enteros no negativos $u_1u_2u_3u_4...un$ y $v_1v_2v_3v_4...vn$ con base **b**, el algoritmo genera su suma $(w_1w_2w_3w_4...wn)_b$, w_0 se emplea como acarreo.

1. j = n (posición de los dígitos)

2. k = 0 (acarreo en cada paso)

3. Sumar los dígitos wj = (uj+vj+k) mod b

4. k = ((uj+vj+k)/b))

5. j = j-1

6. Si j>0 ir al paso 2

7. w0=k

8. FIN

Algoritmo de la multiplicación.

Dados dos enteros $u_1u_2u_3...un$ y $v_1v_2v_3...vm$ de base b; este algoritmo obtiene su producto $(w_1w_2w_3...wm+n)b$. El proceso manual es la base de este algoritmo: se forman primero los productos parciales $(u_1u_2u_3...u)vj$ para $1<=j<=m$ y posteriormente se suman dichos productos; pero para efectos del algoritmo sumaremos concurrentemente con la multiplicación.

1. Inicializar $wm+_1wm+_2...wm+n$ con ceros. Sea j=m

2. Si vj = 0 asignar wj = 0 e ir al paso 6. (Esta prueba minimiza una considerable cantidad de tiempo si por suerte vj es cero, pero puede ser *omitida* sin afectar la validez del algoritmo.

3. i = n, k=0

4. t = ujvj + wi+j + k, wi+j = t mod b

   k = t/b (el acarreo siempre está en el rango 0<=k<=b)

5. i = i - 1

   Si i > 0 ir al paso 4

   wj = k

6. j = j - 1

   Si j > 0 ir al paso 2

7. FIN

La implementación de los algoritmos vistos hasta esta parte del capitulo es fácil de realizar y pueden ser modificados en ciertos pasos para lograr una mayor eficiencia.

2.2.2 Polinomios y Matrices.

Las operaciones aritméticas juegan un papel muy importante cuando se desarrollan en objetos matemáticos un poco más complicados tales como los polinomios y matrices. La cantidad de operaciones que se desarrollen para evaluar dichos objetos es determinante en la eficiencia de los algoritmos.

DEFINICION.

Un polinomio sobre S es una expresión de la forma:

$$p(x) = anx^n an-_1x^n-^1+ \ldots+a_2x^2+a_1x+a_0$$

donde la variable x puede ser considerada como un símbolo formal con un significado indeterminado y los coeficientes $a_0,a_1,a_2,\ldots$an son elementos de algún sistema algebraico S, el cual es un anillo conmutativo con identidad; esto significa que S admite las operaciones de adición, substracción y multiplicación, satisfaciendo además las propiedades asociativas y conmutativas de la adición y de la multiplicación y la propiedad distributiva de la multiplicación sobre la adición. En dicho sistema existe además un elemento neutro aditivo (0) tal que a+0=a y un elemento neutro multiplicativo (1) tal que a(1)=a para toda a que pertenezca a S.

La aritmética de polinomios consiste primordialmente de adición, substracción y multiplicación de polinomios, operaciones que serán tratadas en este capítulo

Las operaciones de adición y substracción son dadas al sumar o restar los coeficientes con igual potencia de x. La multiplicación se encuentra dada por la siguiente regla:

$$(anx^n+\ldots+a_1x+a_0)(bmx^m+\ldots+b_1x+b_0)=c(n+m)x^{n+m}+\ldots+c_1x+c_0$$

donde

$$ck = a_0bk + a_1bk-_1+\ldots+akb_0$$

SIMILITUD ENTRE ENTEROS Y POLINOMIOS.

Dadas las características de los polinomios podemos observar que existe una gran similitud de estos objetos matemáticos con los enteros, algunas de ellas son:

La semejanza más obvia es que ambos pueden representarse como una serie finita de potencias: $\Sigma^{n-1}_{i=0}aix^i$. En el caso de un entero en base 10 las ai pueden ser seleccionadas de el conjunto (1,2,3,4,5,6,7,8,9,0) con x=10 y en el caso de un polinomio las ai pueden ser escogidas de algún conjunto de coeficientes con una x indeterminada.

En ambos, existe un tamaño que es esencialmente la longitud de la serie de potencias para representar el entero o el polinomio, en el caso del polinomio el tamaño es el número de coeficientes, y en el del entero es el número de dígitos.

Los resultados para un polinomio y para la aritmética entera tienen un proceso similar.

SUMA DE POLINOMIOS.

Dado que la suma y la substracción son operaciones inversas, todo lo que se vea en este apartado para la suma es aplicable para la substracción.

Suponiendo que se desea calcular r(x) = p(x)+q(x), donde p y q son polinomios con n coeficientes, el siguiente programa es una implementación de esto:

```
PROGRAM SUMA_POLINOMIOS (INPUT, OUTPUT);
  CONST max = 100;
 VAR
      p,q,r : ARRAY [ 0..max ]  OF REAL;
      n,i   : INTEGER;
 BEGIN
      READLN(n);
      FOR i:=0 TO n-1 DO READ(p [i] );
      FOR i:=0 TO n-1 DO READ(q [i] );
      FOR i:=0 TO n-1 DO r [i]  := p [i]   + q [i]  ;
      FOR i:=0 TO n-1 DO WRITE(r [i] );
      WRITELN;
 END;
```

El polinomio $p(x) = p_0+p_1x+\ldots+p_{n-1}x^{n-1}$ está representado por el arreglo p [0..n-1] con p [j] =pj para j=0,1,2...n-1. Un polinomio de grado n-1 está definido por n coeficientes.

La principal ventaja de representar un polinomio por medio de un arreglo es la facilidad de referenciar sus coeficientes, la desventaja es el espacio que es necesario reservar (ya que siempre se debe reservar para más números de los necesarios), por ejemplo el programa anterior no podría ser usado para sumar

$$(x^{100}+1)+(x^{10}+2)=x^{100}+x^{10}+3$$

aún cuando la entrada involucra sólo cuatro coeficientes y la salida tres. Una alternativa para representar los polinomios es el empleo de las listas ligadas, lo cual involucra almacenamiento en localidades de memoria no contigua donde cada elemento contiene la dirección del próximo.

```
PROGRAM SUMA_POLINOMIOS(INPUT, OUTPUT);
 TYPE
    liga = ^nodo;
    nodo = RECORD
             c       : REAL;
             proximo : liga;
           END;
 VAR
   n : INTEGER;
   z : liga;

PROCEDURE ESCRIBE_COEFICIENTE(r:liga);
 BEGIN
    WHILE  r<>z  DO
```

```
     BEGIN
          WRITE(r^.c);
          r := r^.proximo;
     END;
     WRITELN;
 END;

FUNCTION LEE_COEFICIENTE(n:INTEGER):liga;
 VAR
   i : INTEGER;
   t : liga;
 BEGIN
     t:=z;
     FOR i:=1 TO n-1 DO
     BEGIN
          NEW(t^.proximo);
          t:=t^.proximo;
          read(t^.c);
     END;
     t^.proximo := z;
     LEE_COEFICIENTE := z^.proximo;
     z^.proximo := z;
 END;

FUNCTION SUMA(p,q:liga):liga;
 VAR
   t : liga;
 BEGIN
     t := z;
     REPEAT
          NEW(t .proximo);
          t     := t^.proximo;
          t^.c := p^.c + q^.c;
          p     := p^.proximo;
          q     := p^.proximo;
     UNTIL (p=z) AND (q=z);
     t^.proximo := z;
     SUMA       := z^.proximo;
 END;

BEGIN
 READLN(n);
 NEW(z);
 ESCRIBE_COEFICIENTE(SUMA(LEE_COEFICIENTE(n), LEE_COEFICIENTE(n))
END.
```

Como podemos analizar, en las listas ligadas, se emplean sólo los nodos que son requeridos; sin embargo se puede tener la desventaja de que muchos de los coeficientes del polinomio pueden ser cero, por lo que se puede mejorar el algoritmo implementando una lista de nodos con sólo los coeficientes distintos de cero, esto se puede lograr si cada nodo de la lista contiene el valor del coeficiente y del exponente:

```
PROGRAM SUMA_POLINOMIOS(INPUT, OUTPUT);
 TYPE
    liga : ^.nodo;
    nodo = RECORD
              c       : REAL;
              j       : INTEGER;
              proximo : liga;
           END;

PROCEDURE ESCRIBE_COEFICIENTE(r:liga);
 BEGIN
     WHILE r<>z AND r^.c <> 0
     BEGIN
          WRITE(r^.c);
          r := r^.proximo;
     END;
     WRITELN;
 END;

FUNCTION LEE_COEFICIENTE(n:INTEGER):liga ;
 VAR
   i : INTEGER;
   e : INTEGER;
   t : liga;
 BEGIN
   t := z;
   FOR i:=0 TO n-1 DO
   BEGIN
        READ(e);
        IF e <>0 THEN
        BEGIN
             NEW (t^.proximo);
             t    := t^.proximo;
             t^.c := e;
        END;
   END;
   t^.proximo       := z;
   LEE_COEFICIENTE := z^.proximo;
   z^.proximo       := z;
 END;

FUNCTION ADICIONA_LISTA(t:liga;c:REAL;j:INTEGER):liga;
 BEGIN
     NEW(t^.proximo);
     t    := t^.proximo;
     t^.c := c;
     t^.j := j;
     ADICIONA_LISTA :=t;
 END;

FUNCTION SUMA(p,q:liga):liga;
 BEGIN
     t    := z;
     z^.j := n+1;
```

```
    REPEAT
          IF (p^.j=q^.j) AND (p^.c + q^.c<>0) THEN
          BEGIN
               t := ADICIONA_LISTA(t,p^.c+q^.c,p^.j);
               p := p^.proximo;
               q := q^.proximo;
          END
          ELSE
               IF p^.j < q^.j THEN
               BEGIN
                    t:=ADICIONA_LISTA(t,p^.c,p^.j);
                    p:=p^.proximo;
               END
               ELSE
                    IF q^.j < p^.j THEN
                    BEGIN
                         t:= ADICIONA_LISTA(t,q^.c,q^.j);
                         q:= q^.proximo;
                    END;
    UNTIL (p=z) AND (q=z);
END;

BEGIN
 READLN(n);
 NEW(z);
 ESCRIBE_COEFICIENTE(SUMA(LEE_COEFICIENTE(N),LEE_COEFICIENTE(N)));
END.
```

Si analizamos el programa nos podemos dar cuenta que en la rutina de escritura y lectura se introdujo un filtro que no permite incorporar a la lista correspondiente coeficientes iguales a cero y la función que suma se modifico para permitir la adición correcta del polinomio.

Este tipo de implementación se recomienda para polinomios esparcidos (polinomios con muchos coeficientes iguales a cero). Similares implementaciones se recomiendan para realizar otras operaciones con polinomios por ejemplo, la multiplicación.

SUMA DE MATRICES.

Se desea calcular la suma de dos matrices, para poderla realizar, es necesario tener matrices de igual dimensión y realizar una suma de término por término como en los polinomios, por lo que el programa que realiza esta operación es sólo una generalización del programa para sumar polinomios.

$$P = \begin{bmatrix} p_{11} & p_{12} & \cdots & p_{1n} \\ \cdots & & & \\ p_{m1} & p_{m2} & \cdots & p_{mn} \end{bmatrix} \quad + \quad Q = \begin{bmatrix} q_{11} & q_{12} & \cdots & q_{1n} \\ \cdots & & & \\ q_{m1} & q_{m2} & \cdots & q_{mn} \end{bmatrix}$$

$$= \begin{pmatrix} p_{11}+q_{11} & p_{12}+q_{12} & \cdots & p_{1n}+q_{1n} \\ \cdots & & & \\ p_{m_1}+q_{m_1} & p_{m_2}+q_{m_2} & \cdots & p_{mn}+q_{mn} \end{pmatrix}.$$

```
PROGRAM SUMA_MATRICES(INPUT,OUTPUT);
 CONST
     max = 10;
 VAR
     p,q,r    : ARRAY [0..max,0..max]   OF REAL;
     n,m,i,j  : INTEGER;
 BEGIN
     READLN(n,m);
     FOR i:=0 TO n-1 DO
       FOR j:=0 TO m-1 DO
           READ(p [i,j] );
     FOR i:=0 TO n-1 DO
       FOR j:=0 TO m-1 DO
           READ(q [i,j] );
     FOR i:=0 TO n-1 DO
       FOR J:=0 TO m-1 DO
           r [i,j]   := p [i,j]   + q [i,j]  ;
     FOR i:=0 To n-1 DO
       FOR j:=0 TO m-1 DO
       BEGIN
            IF j=m THEN WRITELN;
            WRITE(r [i,j] );
       END;
 END;
```

Al igual que con los polinomios las matrices esparcidas (con muchos elementos iguales a cero) pueden procesarse empleando listas ligadas.

Cada elemento de la matriz distinto de cero se representa por un nodo ligado que contiene dos ligas: una apuntando al próximo elemento distinto de cero en el mismo renglón y otro apuntando al elemento distinto de cero en la misma columna, por lo que se puede decir que la implementación de la suma de matrices esparcidas, es un poco más complicado si cada nodo contiene dos campos tipo apuntador (lo cual no es estrictamente necesario, ver capitulo VI).

Por ejemplo

$$\begin{vmatrix} 0 & 2 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{vmatrix}$$

puede ser implementado como:

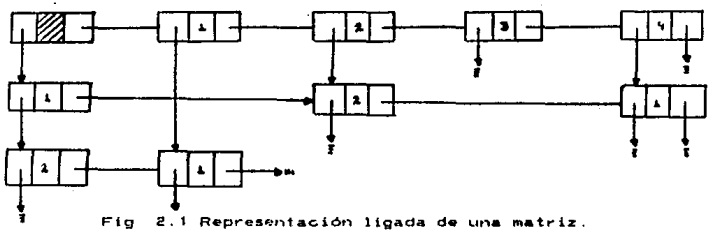


Fig 2.1 Representación ligada de una matriz.

MULTIPLICACION DE POLINOMIOS.

El problema de la multiplicación de polinomios consiste en que dados dos polinomios **p(x)** y **q(x)** se debe calcular su producto **p(x)q(x)**; para ello se debe tener presente que un polinomio de grado **n-1** puede tener n términos (incluyendo la constante) y el producto de este tipo de polinomios de grado **2n-2** y a lo más **2n-1** términos, por ejemplo:

$$(-4x^3+3x^2+x+1)(-3x^3-5x^2+2x+1)=12x^6+11x^5-26x^4-6x^3+3x+1$$

Una implementación simple de este proceso es la siguiente:

```
FUNCTION MULTIPLICA(p,q:ARRAY [0..n-1]  :OF REAL;n:INTEGER);
 BEGIN
     FOR i:=0 TO 2*(n-1) DO r[i]  :=0;
     FOR i:=0 TO n-1 DO
         FOR j:=0 TO n-1 DO
             r[i+j]   := r[i+j]  +p[i]  *q[i]   ;
 END;
```

Este proceso requiere n^2 multiplicaciones para polinomios de grado **n-1** ya que cada uno de los **n** términos de **p(x)** es multiplicado por los **n** términos de **q(x)**.

Otra técnica de resolver este problema es dividiendo la longitud del polinomio exactamente a la mitad (técnica divide y vencerás): Dado un polinomio de grado **n-1** (n coeficientes) es posible tener dos polinomios con **n/2** coeficientes (asumiendo que n es par) y empleando los **n/2** coeficientes de menor orden para un polinomio y los **n/2** coeficientes de mayor orden para otro.

Para $p(x) = p_0+p_1x+p_2x^2+\ldots p_{n-1}x^{n-1}$ definimos:

$$p_1(x) = p_0+p_1x+p_2x^2+\ldots p_{n/2-1}x^{n/2-1}$$

$$p_2(x) = p_{n/2}x^{n/2}+p_{n/2+1}x^{n/2+1}+\ldots p_{n-1}x^{n-1} = x^{n/2}(p_{n/2}+p_{n/2+1}x+p_{n/2+2}x^{n-2}+\ldots+p_{n-1}x^{n-1-n/2})$$

Si empleamos el mismo criterio para $q(x)$:

$$q(x) = q_0+q_1x+q_2x^2+\ldots q_{n-1}x^{n-1}$$

$$q_1(x) = q_0+q_1x+q_2x^2+\ldots q_{n/2-1}x^{n/2-1}$$

$$q_2(x) = q_{n/2}x^{n/2}+q_{n/2+1}x^{n/2+1}+\ldots q_{n-1}x^{n-1} = x^{n/2}(q_{n/2}+q_{n/2+1}x+q_{n/2+2}x^{n-2}+\ldots+q_{n-1}x^{n-1-n/2})$$

donde

$$p(x) = p_1(x) + x^{n/2}p_2(x)$$

$$q(x) = q_1(x) + x^{n/2}q_2(x)$$

Ahora estos términos son polinomios más pequeños y su producto está dado por:-

$$p(x)q(x) = p_1(x)q_1(x) + (p_1(x)q_2(x) + p_2(x)q_1(x))x^{n/2} + p_2(x)q_2(x)x^n$$

Lo cual resulta interesante ya que sólo requerirímos tres multiplicaciones:

$$r_1(x) = p_1(x)q_1(x)$$

$$r_2(x) = p_2(x)q_2(x)$$

$$r_3(x) = (p_1(x)+p_2(x))(q_1(x)+q_2(x)) = p_1(x)q_1(x) + p_2(x)q_1(x) + p_2(x)q_1(x) + p_2(x)q_2(x)$$

y podemos obtener el producto $p(x)q(x)$ con sólo

$$p(x)q(x) = r_1(x)+(r_3(x) - r_1(x) - r_2(x))x^{n/2}+r_2(x)x^n$$

con lo que se reduce la complejidad del algoritmo ya que el proceso para sumar polinomios es lineal y para multiplicarlos es cuadrático.

Ejemplo:

Sea $p(x) = -4x^3+3x^2+1$

$q(x) = -3x^3-5x^2+2x+1$

entonces

$$r_1(x)=(x+1)(2x+1) = 2x^2+3x+1$$

$$r_2(x)=(-4x+3)(-3x-5) = 12x^2+11x-15$$

$$r_3(x)=(-3x+4)(-x-4) = 3x^2+8x-16$$

entonces $r_3(x) - r_1(x) - r_2(x) = 11x^2-6x-2$ y el producto se calcula como $p(x)q(x)=(2x^2+3x+1)+(-11x^2-6x-2)x^2+(12x^2+11x-15)x^4$ Este diseño del problema por medio del método divide y vencerás (capitulo VII) resuelve una multiplicación de polinomios de tamaño n, dando solución a tres subproblemas de tamaño n/2, empleando sólo sumas sobre los subproblemas y combinando sus soluciones. Puede ser implementado como un proceso recursivo:

```
FUNCTION MULTIPLICA(p,q: ARRAY [0..n-1]    OF REAL;  n:INTEGER):
ARRAY  0..2*n-2] OF REAL;
 VAR
   p1,q1,p2,q2,t1,t2 : ARRAY [0..(nDIV2)-1]   OF REAL;
   r1,r2,r3          : ARRAY [0..n-1]   OF REAL;
   i,n2              : INTEGER;
 BEGIN
     IF n=1 THEN mult[0]  :=p[0]  *q[0]      .
     ELSE
     BEGIN
          n2:=n DIV 2;
          FOR i:=0 TO n2-1 DO
          BEGIN
               p1[i]    := p[i]  ;
               q1[i]    := q[i]  ;
          END;
          FOR i:=n2 TO n-1 DO
          BEGIN
               p2[i-n2]  :=p[i]  ;
               q2[i-n2]  :=q[i]  ;
          END;
          FOR i:=0 TO n2-1 DO
              t1[i]  :=p1[i]  +p2[i]  ;
          FOR i:=0 TO n2-1 DO
              t2[i]  :=q1[i]  +q2[i]  ;
          r3:=MULTIPLICA(t1,t2,n2);
          r1:=MULTIPLICA(p1,q1,n2);
          r2:=MULTIPLICA(p2,q2,n2);
          FOR i:=0 TO n-2 DO MULTIPLICA[i]  :=r1[i]  ;
          mult[n-1]  :=0;
          FOR i:=0 to n-2 DO MULTIPLICA[n+i]  :=r2[i]  ;

          FOR i:=0 TO n-2 DO
          MULTIPLICA[n2+i]  :=MULTIPLICA[n2+i]  +r3[i]  -(r1
                    [i]  +r2[i] );
     END;
 END;
```

Este programa asume que n es una potencia de 2, los detalles para que pueda ser ejecutado para una n cualquiera son fáciles de

implementar, pero hay que tener cuidado para que la recursión termine bien y que los polinomios se dividan en forma correcta. Así mismo este método puede usarse para multiplicar enteros si se maneja el acarreo en forma adecuada después de las llamadas recursivas.

Si se observa con cuidado el método, debe resultar claro que el número de multiplicaciones enteras requeridas al multiplicar dos polinomios de tamaño n es el mismo que el requerido para multiplicar tres polinomios de tamaño n/2; por lo que si M(n) es el número de multiplicaciones requeridas al multiplicar dos polinomios de tamaño n tenemos M(n)=3M(n/2) para n>1; para distintos valores de M (recuerde que se está aplicando un método recursivo).

M(1) = 1 M(2) = 3 M(3)= 9 M(4)=27

En general si N=2**n (por la condición n potencia de 2) se obtendrá la solución: $M(2^{**}n) = 3M(2^{**}(n-1)) = 3^2M(2^{**}(n-2)) = 3^3M(2^{**}(n-3)) = \ldots = 3^{**}nM(1) = 3^{**}n$. Entonces aplicando logaritmos: $3^{**}n = 2^{**}(\log 3)n = n^{**}\log 3 = n^{**}1.58$; solución que es exacta sólo para n=2**n, pero en general $M(n) = n^{**}\log 3 = n^{**}1.58$.

MULTIPLICACION DE MATRICES.

La multiplicación de matrices es una operación un poco más complicada, por ejemplo:

$$\begin{bmatrix} 1 & 3 & -4 \\ 1 & 1 & -2 \\ -1 & -2 & 5 \end{bmatrix} \begin{bmatrix} 8 & 3 & 0 \\ 3 & 10 & 2 \\ 0 & 2 & 6 \end{bmatrix} = \begin{bmatrix} 17 & 25 & -18 \\ 11 & 9 & -10 \\ -14 & -13 & 26 \end{bmatrix}$$

el elemento r(i,j) es el producto punto del i-ésimo renglón de p con la j-ésima columna de q. Recordando que el producto punto es simplemente la suma de los n términos de la matriz p por los n términos de la matriz q lo cual implica que para poder multiplicar dos matrices se requiere que el número de columnas de la matriz p sea igual al numero de renglones de la matriz q; la forma más facil de implementar esta operación es por medio de almacenamiento contiguo lo cual estaría dado por el siguiente procedimiento:

```
FOR i:=0 TO m-1 DO
  FOR j:=0 TO n-1 DO
  BEGIN
        t:=0;
        FOR k:=0 TO n-1 DO
            t:=t+p[i,k] *q[k,j] ;
```

```
            r[i,j] :=t;
    END;
```

Cada uno de los elementos de la matriz resultado se calcula por medio de n multiplicaciones por lo que se requieren n^3 operaciones para multiplicar matrices de dimensión nxn (n^2 elementos). En el caso de multiplicar una matriz de mxn con una matriz de nxr, se requieren de (m-1)(n-1)(r-1) multiplicaciones y sumas.

EVALUACION DE POLINOMIOS.

Supóngase que se tiene el siguiente polinomio

$$p(x)=a_n x^n+a_{n-1} x^{n-1}+\ldots+a_1 x+a_0$$

cuyos coeficientes son valores dados y se nos pide evaluar dicha expresión en un punto dado, por ejemplo evaluar

$$p(x) = x^4+3x^3+6x^2+5x+1$$

Tal vez el camino más obvio para resolver el problema es calcular cada término y adicionarlo a la suma de los otros previamente calculados (por medio del recálculo de las potencias de x) como lo realiza el siguiente algoritmo:

```
PROCEDURE EVALUA_POLINOMIO  (p:ARRAY [ 0..n-1]      OF  REAL;  x:
INTEGER);
 VAR
    i,potencia: INTEGER;
 BEGIN
     valor     := p[0] ;
     potencia := 1;
     FOR i:=1 TO n-1 DO
     BEGIN
          potencia := potencia * x;
          valor    := valor + p[i] *potencia;
     END;
 END;
```

El cual realiza 2(n-1) multiplicaciones y n-1 sumas; sin embargo, existe un método simple que reduce la cantidad de cálculos realizados a sólo n-1 multiplicaciones y n sumas; este método se conoce con el nombre de REGLA DE HORNER: "alternando la multiplicación y las operaciones de suma apropiadamente, un polinomio de grado n puede ser evaluado usando sólo n-1 multiplicaciones y n sumas"; en otras palabras la regla de Horner es la factorización del polimonio de la siguiente forma:

$$p(x) = (\ldots((a_n x+a_{n-1})x+a_{n-2})x+\ldots+a_1)x+a_0$$

cuyo cálculo esta dado por el ciclo

```
y:=p [n];
```

```
FOR i := n-1 DOWNTO 0 DO
    y:= x*y + p[i] ;
```

donde se asume que el polinomio esta representado por un arreglo.

Otro problema es evaluar un polinomio dado en varios puntos, para resolverlo existen muchos algoritmos diferentes cuya eficiencia depende del número de evaluaciones que se van a realizar y si se van a dar o no simultáneamente. Si se va a realizar un gran número de evaluaciones es recomendable realizar precálculos que puedan reducir el costo de las evaluaciones posteriores dado que si por ejemplo, se emplea el método de Horner se requerirán cerca de n^2 multiplicaciones para evaluar un polinomio de grado n en n puntos.

Si el polinomio dado sólo tiene un término, entonces el problema de la evaluación de un polinomio se reduce a calcular x^n, si se emplea la regla de Horner el algoritmo degeneraría a un proceso trivial que requiere n-1 multiplicaciones. Para comprender como evaluar ese polinomio de otra forma veamos el siguiente Ejemplo:

Para calcular x^{32} consideremos la siguiente secuencia:

$$x, x^2, x^4, x^8, x^{16}, x^{32}$$

cada término se obtiene elevando al cuadrado el término previo y sólo se requieren 5 multiplicaciones. Este método se conoce con el nombre de **Método de Cuadrados** que puede ser expandido a una n general si se salvan los valores calculados.

Por ejemplo x^{55} puede ser calculado empleando los valores $x^{55}=x^{32}x^{16}x^4x^2x$, si observamos con detenimiento esta secuencia podríamos percatarnos que la representación binaria del exponente nos ayuda a seleccionar los valores que vamos a emplear $55=(110111)_2$ en este caso se emplean todos los exponentes a excepción de x^8. Una implementación fácil de este método es la siguiente:

1) acumula := 1;

2) Buscar en la representación binaria de izquierda a derecha bit a bit;

3) Calcular el cuadrado de acumula; si el bit es uno multiplicar acumula por x.

Si este algoritmo se aplica a n=55 se genera la siguiente secuencia de valores:

$$1, 1, x, x^2, x^3, x^6, x^{12}, x^{13}, x^{26}, x^{27}, x^{54}, x^{55}$$

Otro método para evaluar

$p(x) = x^n + a_{n-1}x^{n-1} + \ldots + a_1 x + a_0$ en el cual $n = 2^k - 1$ para alguna $k>1$, es decir, p tiene 2^k términos algunos de los cuales son cero y $a_n = 1$ entonces, si $n=1$, $p(x)=x+a_0$, y se evalua al realizar una suma y si $n>1$ para alguna j y b dadas entonces, p puede ser escrita como sigue:

$$p(x) = (x^j + b)q(x) + r(x)$$

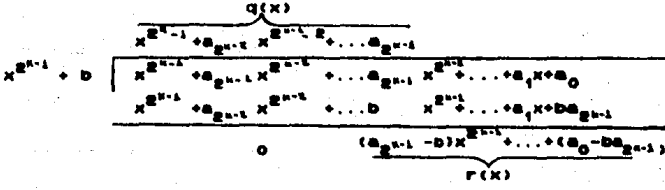
donde q(x) y r(x) tienen las mismas caracteristicas que p ($_1$(el coeficiente del grado del polinomio es 1) y son de grado $2^{n-1}-1$. Entonces, p(x) puede ser evaluado por medio de los siguientes pasos:

1) Evaluar q(x) y r(x)

2) Calcular x^j

3) Multiplicar $(x^j + b)$ por q(x) y sumarle r(x).

Dado que q y r satisfacen las mismas condiciones que p pueden ser evaluados en forma recursiva, donde j = grado(p) - grado(q)=$2^k-1-(2^{k-1}-1) = 2^{k-1}$ dado que j es una potencia de 2, x^j puede ser calculado rápidamente. Si dividimos p(x) entre $x^j + b$ obtenemos q(x) en el cociente y a r(x) en el residuo.



para garantizar que el coeficiente del termino $x^{2^{k-1}}$ de r(x) sea 1 $a_{2^{k-1}} - b = 1$ por lo tanto $b=a_{2^{k-1}} - 1$. El algoritmo para este método seria:

$$p(x)=(x^{2^{k-1}} + (a_{2^{k-1}} - 1))q(x) + r(x).$$

El cual es un procedimiento recursivo que termina cuando q y r son de grado 1.

Ejemplo

$p(x)=x^7+6x^6+5x^5+4x^4+3x^3+2x^2x+1$

$k=3 \qquad j=2^{k-1}=4 \qquad b=a_{2^{k-1}-1}-1=a_3-1=2 \Rightarrow x^j + b = x^4+2$

1) $q(x)$

$$x^3+6x^2+5x+4$$

$$x^4+2 \;\Big|\; x^7+6x^6+5x^5+4x^4 \;\Big|\; +3x^3+2x^2+x+1$$

$$x^7+6x^6+5x^5+4x^4 \;\Big|\; +2x^3+12x^2+10x+8$$

$$x^3+10x^2-9x-7 \longrightarrow r(x)$$

esto es $p(x)=(x^4+2)(x^3+6x^2+5x+4)+(x^3-10x^2-9x-7)$.

Factorizando q(x) y r(x) por medio de la recursividad:

$q(x) = x^3+6x^2+5x+4$ $\qquad$ $r(x)=x^3-10x^2-9x-7$

$k=2 \qquad j=2^{k-1}=2$ $\qquad$ $k=2 \qquad j=2^{k-1}=2$

$b=a_{2^{k-1}-1}=a_1=1=4$ $\qquad$ $b=a_{2^{k-1}+1}=a_1=1=-10$

$$x^2+4 \;\Big|\; x^3+6x^2 \;\Big|\; x+6 \;\Big|\; +5x+4x \;\Big|\; x-20$$

$$x^3+6x^2 \;\Big|\; +4x+24$$

$$x^2-10 \;\Big|\; x^3-10x^2 \;\Big|\; x-10 \;\Big|\; -9x-7 \;\Big|\; x-107$$

$$x^3-10x^2 \;\Big|\; -10x+100$$

esto es $q(x)=(x^2+4)(x+6)(x-20)$ y $r(x)=(x^2-10)+(x-107)$ por lo tanto $p(x)=(x^4+2)((x^2+4)(x+6)(x-20))+((x^2-10)(x-10)+(x-107))$.

El empleo de esta fórmula para evaluar p(x) requiere cinco multiplicaciones, tres que aparecen explícitamente en la factorización y dos para calcular x^2 y x^4, además de 10 sumas; si se aplica el método de Horner se realizaría con siete multiplicaciones y siete sumas. Por lo cual, es necesario que la persona que desee seleccionar un método comprenda que la aplicación de criterios propios es determinante para poder decir porqué se inclina por un método determinado.

2.2.3 Números aleatorios (RANDOM).

Las secuencias de números aleatorios se emplean en infinidad de situaciones dentro del campo de la programación, desde la simulación que es lo más común, hasta los juegos y otro tipo de software recreativo. La forma de obtener estos números es muy

variada: las ruletas, dados y barajas constituyen los métodos manuales de generación de números aleatorios. Estas técnicas tienen como desventaja ser muy lentas, sólo es posible obtener muestras grandes después de un trabajo laborioso; además, es imposible volver a generar la misma secuencia de números, lo cual puede ser útil en un momento determinado para comprobar algún resultado.

Recientemente, se han desarrollado algunas técnicas para obtener números aleatorios, entre los que destacan las tablas, las computadoras analógicas y las computadoras digitales. En las tablas, los números que se tienen han sido generados por algún otro medio. Esta técnica tiene la ventaja de poder repetir la misma secuencia de números, pero aún si se maneja una computadora es una técnica lenta y puede quedar limitada si se requiere de una muestra grande, además de que los números que contiene son constantes para cualquier problema.

Las computadoras analógicas generan números a través de algún proceso físico, lo cual permite obtener números que son verdaderamente aleatorios con una alta frecuencia; su desventaja principal es el no poder repetir la misma secuencia.

En las computadoras digitales los números se pueden generar por medio de tablas, con algún aditamento analógico que establezca un proceso físico, o bien, por generación interna empleando una relación de recurrencia (números pseudoaleatorios). Debido a las características innatas del funcionamiento de las computadoras digitales, se puede decir que el manejo de tablas y el uso de algún aditamento es artificial, por lo cual, en el caso de las tablas, la generación de los números es lenta, además de que ocupa gran espacio de memoria y en el caso de aditamentos analógicos no es posible repetir la misma secuencia.

En muchos compiladores, la calidad de los generadores de números aleatorios no es suficiente para el problema en cuestión, así que puede ser necesario disponer de algún tipo de control sobre la forma en la cual se producen los números aleatorios con la finalidad de mejorar o implementar detalles o algoritmos referentes a este punto que produzcan resultados más satisfactorios.

DEFINICION DE GENERADOR DE NUMEROS ALEATORIOS.

Generar números aleatorios, significa informalmente crear una secuencia de números que aparentan estar en un orden aleatorio, es decir, números que no se encuentren relacionados en absoluto, lo cual implica que cualquier secuencia puede y no ser aleatoria, ello depende de la forma en la cual fue generada y no de cual sea la secuencia en si.

Un generador de números aleatorios es básicamente un conjunto de instrucciones que asocian o combinan los dígitos por medio de operaciones computables simples, produciendo un nuevo entero.

El generador debe producir números aleatorios uniformemente distribuidos (cada número posible es igualmente probable), ser rápido, no debe emplear demasiado espacio de memoria, debe tener un periodo grande antes de repetir su ciclo, es decir debe transcurrir un tiempo considerable antes de que obtenga la misma secuencia de valores.

Después de que se introdujerón las computadoras, la gente inicia la búsqueda de caminos eficientes para generar números aleatorios por medio de programas, y aún cuando muchos generadores de números aleatorios usados hoy en día son muy buenos, existe la tendencia de la gente a evitar aprender cuestiones sobre los generadores de números aleatorios, y por ello métodos viejos que son comparativamente insatisfactorios han pasado de programadores a programadores sin que éstos se preocupen realmente de sus limitaciones.

Una de las fallas más comunes encontradas en el empleo de los generadores de números aleatorios es la idea de que si se tiene un buen generador y se modifica su orden, se producirá una secuencia más aleatoria; esto no es siempre cierto y hay que tenerlo presente para no decrementar la eficiencia del método.

ALGUNOS METODOS PARA GENERAR NUMEROS ALEATORIOS.

Cuando se emplea una computadora para generar secuencias de números aleatorios, hay que tener presente que son secuencias deterministicas, es decir cada número, excepto el primero, depende del número que le precede. Técnicamente esto significa que en una computadora sólo se pueden crear secuencias de números "cuasi-aleatorios", sin embargo, esto es suficiente para la mayoría de problemas. Los métodos más comunes para generar números aleatorios empleando la computadora son:

A. Método de cuadrados centrales.

Es un método particularmente interesante desarrollado por Von Neumann (Padre de las computadoras modernas), consiste en elevar al cuadrado el número aleatorio anterior y extraer los dígitos centrales. Por ejemplo si se generan números de tres dígitos y el número aleatorio anterior es 513, entonces se eleva este número al cuadrado para obtener 257169 y se extrae 571 como número central y el siguiente elemento de la secuencia. Las principales desventajas de este método son: que tiende a degenerar con rapidez es decir, tiende rápido hacia una secuencia repetitiva denominada ciclo o periodo, especialmente si encuentra el cero como número central; por ello este método no se emplea en forma común.

ALGORITMO.

1. Generar un número de M dígitos (semilla).

2. Elevar dicho número al cuadrado.

3. Tomar un número central de N dígitos.

4. Repetir 2-3 tantas veces como se desee.

B. Método lineal Congruente.

Los métodos más conocidos para generar números aleatorios fueron introducidos por D. Lehmer en 1951 y se denominan métodos lineales congruentes. Para generar el i-ésimo se basa en la siguiente fórmula:

$$X_{i+1} = (bX_i + c) \bmod m$$

Cuando b=1 el método recibe el nombre de método lineal aditivo; si c=0 se denomina multiplicativo y en otro caso mixto.

Si por ejemplo X[i] contiene algún número arbitrario, entonces la siguiente instrucción genera un arreglo con n números aleatorios usando este método:

```
FOR  i:=2  TO  n  DO

  X[i]   := (X[i-1] *b + c) mod m.
```

Esto, es se genera el nuevo número aleatorio, tomando el número anterior multiplicado por una constante **b**, sumándole **c** y tomando el residuo cuando se divide por una segunda constante **m**. El resultado es siempre un entero entre 0 y m-1. Como se puede observar este método es de uso atractivo para las computadoras porque la función **mod** es usualmente trivial de implementar, además el empleo de esta ecuación para la generación de números aleatorios puede parecer algo muy sencillo, sin embargo, lo bien que funcione depende fuertemente de los valores de **a**, **c** y **m**. La elección de esos valores ha sido objeto de intensos, detallados y difíciles análisis matemáticos, veamos algunas reglas sencillas que Knuth propone:

1. El número X_0 puede ser escogido arbitrariamente. Si el programa se va a ejecutar varias veces se puede emplear el último valor de X de la ejecución precedente, si no se puede dar un valor inicial a X (preferentemente un número primo distinto a 5).

2. Para una adecuada elección de **m** se recomienda un número grande por ejemplo, el tamaño de la palabra de la computadora, o un número grande que sea potencia de 2 o 10 (existen computadoras con longitud de palabra variable, en este caso la elección de **m** se deja a criterio del programador), dado que esto hace que el cálculo de **(bx+c) mod m** sea muy eficiente.

3. Si **m** es una potencia de 2 se escoge **b** de tal manera que cumpla: **b mod 8 = 5**; si **m** es una potencia de 10 se escoge una **b** tal que **b mod 200 = 21**.

4. El multiplicador b puede ser más grande que $\sqrt{m}$, preferiblemente mayor a m/100; pero más pequeño que $m-\sqrt{m}$. Una elección adecuada de b es un número con un dígito menor que m.

Al seleccionar el valor de las constantes no nos podemos permitir ignorar el overflow ya que esto puede conducir a resultados impredecibles. Suponiendo que tenemos una computadora con una palabra de 32 bits, y escogemos a m=100000000, b=31415821 e iniciamos x_0=1234567. Todos esos valores son menores al entero más grande que puede representarse, pero la operación x_{n+1} causa un overflow, que no es muy relevante ya que sólo estamos interesados en los últimos ocho dígitos, para eludirlo se puede realizar la multiplicación en partes, sea $p=10^4p_1+p_0$ y $q=10^4q_1+q_0$ tal producto sería:

$$pq = (10^4p_1 + p_0)(10^4q_1 + q_0) = 10^8p_1q_1 + 10^4(p_1q_0 + p_0q_1) + p_0q_0$$

Ahora como sólo se está interesado en los 8 primeros dígitos se puede ignorar el primer término y los primeros cuatro dígitos del segundo término. Lo cual genera el siguiente programa:

```
PROGRAM RANDOM (INPUT,OUTPUT);
 CONST
      m = 100000000;
      m1= 10000;
      b = 31415821;
 VAR
      i, a, n : INTEGER;

FUNCTION MULT(p,q: INTEGER): INTEGER;
 VAR
      p1,p0,q1,q0 : INTEGER;
 BEGIN
      p1  := p DIV m1;     p0 := p MOD m1;
      q1  := q DIV m1;     q0 := q MOD m1;
      mult:=(((p0*q1 + p1*q0) MOD m1)*m1 + p0*q0 mod m;
 END;

BEGIN
      READ(n,x);
      FOR i:=1 TO n DO
      BEGIN
           x1 := (mult(b,x)+1) MOD m;
           WRITE(x1);
           x := x1;
      END;
END;
```

La función mult en este programa calcula (p*q+1) MOD m sin overflow.

Los números producidos por este programa para una entrada n=10 y x=1234567 son

35884508

80001069

63512650

43635651

01034472

87181513

06917174

00209855

67115956

59939877

Si observamos bien esos números podemos darnos cuenta que su unidad tiene un comportamiento determinado: circula entre cero y nueve, lo cual es un problema común entre este tipo de métodos y puede tener solución con el siguiente proceso:

```
FUNCTION RANDOM(R:INTEGER):INTEGER;
 BEGIN
     X:=(MULT(b,x) MOD m;
     RANDOM := X DIV r;
 END;
```

Otra técnica común es generar números aleatorios entre cero y uno. Esto puede ser implementado simplemente regresando un número real x/m por r y truncar al entero más cercano.

c. Método por corrimiento de registros.

Este método se basa en una regeneración de corrimientos lineales de registros y se emplea principalmente en máquinas de encriptación.

El método inicia con un registro fijo (arbitrario), para generar el siguiente número se realiza un corrimiento a la derecha y se coloca un bit en la posición vacante producto de una operación con los bits iniciales. Por ejemplo el siguiente diagrama muestra un registro simple de cuatro bits, en el cual el nuevo bit es un "or exclusivo" de los dos bits más a la derecha:

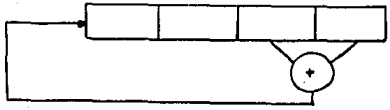


Fig 2.2

La siguiente lista muestra el contenido del registro para los primeros 16 pasos del proceso:

0	1	2	3	4	5	6	7
1011	0101	1010	1101	1110	1111	0111	0011
8	9	10	11	12	13	14	15
0001	1000	0100	0100	0010	1001	0110	1011

Se puede observar que todos los modelos excepto el 0000 ocurren y el valor inicial se repite después de 15 pasos; se han realizado extensos estudios para seleccionar el valor inicial y el tamaño de los registros; por ejemplo, si aplicamos el or exclusivo a dos palabras 1011 $\oplus$ 0101 se obtiene 1110 lo cual aparece tres lugares después en la lista. Esto nos da la idea de generar números aleatorios en forma fácil empleando una retroalimentación de registros:

$$a[k] = (a[k-b] + a[k-c]) \text{ MOD } m$$

lo cual guarda una correspondencia con el modelo de corrimiento de registros, el "+" en esta recursión podría ser un "or exclusivo", sin embargo se ha demostrado que esta fórmula genera buenos números aleatorios aún con una suma normal; pero su desventaja principal es que es necesario conservar una tabla que contenga los números aleatorios más recientes. La tabla puede inicializarse con números ni tan pequeños ni tan grandes, para seleccionar los valores de las constantes Knuth recomienda escoger b=31 y c=55 con los cuales se pueden realizar buenas aplicaciones.

¿COMO SE PUEDE DETERMINAR LA CALIDAD DE UN GENERADOR?

Se pueden emplear distintas pruebas para determinar el grado de aleatoriedad de una secuencia de números, sin embargo, ninguna de ellas dirá si la secuencia es aleatoria sólo si no lo es, ya que estas pruebas sólo muestran el grado de confianza que se puede tener en el generador de números aleatorios que produce la secuencia.

La teoría de la estadística propone algunas medidas cuantitativas para la aleatoriedad. Literalmente no existe un

número determinado de pruebas que se le deban aplicar a una secuencia para determinar su comportamiento aleatorio; ya que si una secuencia se comporta aleatoriamente con respecto a las pruebas $T_1, T_2, T_3, \ldots T_n$, no se puede asegurar que tenga éxito en la prueba siguiente ya que cada prueba se limita a proporcionar sólo un mayor grado de confianza. En la práctica se aplican cerca de media docena de pruebas estadísticas diferentes a una secuencia y si ésta las pasa satisfactoriamente puede ser considerada aleatoria. Se recomienda que si una secuencia se va a utilizar frecuentemente se pruebe cuidadosamente.

Knuth propone diferentes pruebas, sin embargo en este apartado sólo estudiaremos una de ellas con la finalidad de analizar como se realizan éstas; ya que el análisis de las diferentes pruebas no es el objetivo de la tesis.

PRUEBA CHI-CUADRADO.

La prueba **Chi-cuadrado** (prueba λ^2) es una de las mejores pruebas de toda la estadística y es un método básico usado en otras pruebas, además de ser fácil de implementar ya que se basa en la diferencia del número esperado de ocurrencias de todos los posibles resultados, la fórmula es:

$$V = \sum_{i=1}^{N} (Oi - Ei)^2 / Ei$$

Donde **Oi** es el número de ocurrencias observadas, **Ei** es el número de ocurrencias esperadas y **N** el número de elementos discretos. Una vez que se tiene el valor de V se procede a buscar este número en la tabla de valores de desviaciones cuadráticas para determinar la aleatoriedad de la secuencia.

A manera de ejemplo, veamos como probar cuánto se aproxima la distribución de los números de una secuencia a lo que se espera sea una distribución aleatoria, supóngase que se desean generar secuencias aleatorias de dígitos entre cero y nueve. La probabilidad de que se de cada dígito es 1/10, ya que existen diez posibles números en la secuencia, todos igualmente probables

Supongamos que se desea generar la siguiente secuencia : 91824637582904247862 si se contabiliza el número de veces que se da cada dígito se obtiene:

Dígito :	0	1	2	3	4	5	6	7	8	9
Ocurrencia :	1	1	4	1	3	1	2	2	3	2

Para determinar la aleatoriedad de la secuencia se determina el valor de **V** donde **n=10** (número de posibles elementos) y el número de observaciones es igual a 20, por lo que **Ei=2 i=1,2,...,10.**

$$V = (1-2)^2/2 + (1-2)^2/2 + (4-2)^2/2 + (1-2)^2/2 + (3-2)^2/2 +$$

$$(1-2)^2/2 + (2-2)^2/2 + (3-2)^2/2 + (2-2)^2/2 = 5$$

Posteriormente se busca en la tabla de desviaciones cuadráticas el grado de confianza (En el renglón igual al número de observaciones (20) y la columna con el número mayor o igual a V)

PROBABILIDAD

	99%	95%	75%	50%	25%	5%
n=5	0.5543	1.1455	2.6750	4.3510	6.6260	11.0700
n=10	2.5580	3.9400	6.7370	9.3420	12.5500	18.3100
n=15	5.2290	7.2610	11.0400	14.3400	18.2500	25.0000
n=20	8.2600	10.8500	15.4500	19.3400	23.8300	31.4100

Lo cual significa que existe un 99% de probabilidad de que una muestra de 20 elementos tenga un valor V mayor a 8.60, por lo que sólo existe el 1% de probabilidad de que la secuencia sea aleatoria. Para que pase esta prueba la probabilidad de V debe estar entre el 25% y el 75%; sin embargo es importante cuestionarse ¿Cómo es posible que esta secuencia sólo tenga el 1% de probabilidad de ser aleatoria si todas las secuencias son posibles? La respuesta es que se trata sólo de una probabilidad y que la prueba de desviación cuadrática sólo da un grado de confianza, de hecho si se emplea esta prueba, se debe aplicar a diferentes secuencias y obtener la media de los resultados, para impedir que se deseche un buen generador de números aleatorios.

Por otro lado puede que una secuencia pase la prueba y no ser aleatoria, por ejemplo: 135792468 es una secuencia que pasaría la prueba a pesar de no parecer muy aleatoria.

Otra característica que hay que comprobar es la longitud del período (cuántos números se pueden generar antes de que se empiece a repetir la secuencia, o peor, antes de que degenere en un ciclo corto), a mayor período mayor es la calidad del generador; generalmente un período de varios miles de números es suficiente para la mayoría de aplicaciones.

APLICACIONES DE LOS NUMEROS ALEATORIOS.

Los números aleatorios se emplean en una gran variedad de aplicaciones, por ejemplo:

A. Simulación

Una simulación es un modelo de una situación real. Cualquier cosa puede ser simulada, y el éxito de ello depende de lo bien

que comprenda el programador la situación a ser simulada. Debido a que los modelos reales a menudo tienen miles de variables, siempre es difícil realizar una simulación totalmente efectiva. Las simulaciones son muy importantes porque no sólo permiten alterar los parámetros de la situación y comprobar los posibles resultados (en la vida real esta situación podría resultar muy costosa o muy peligrosa), sino que también permite crear situaciones que no se dan en el mundo real (por ejemplo aumentar gradualmente la inteligencia de un ratón para ver en que punto resuelve un laberinto dado; lo cual puede permitir comparar la naturaleza de la inteligencia con el instinto).

Ejemplos

- **Simulación de colas de espera:** En la simulación de colas de espera, a medida que se ejecuta la simulación, un generador de números crea clientes, otro generador determina cuánto se tarda en atender un cliente y un tercer generador a qué cola de espera va cada cliente. Lo que se pretende con esta simulación es ayudar a administrar un número óptimo de colas que hay que tener disponibles en un típico día de compras, limitando el número de personas en la cola a diez o menos y sin dejar ninguna caja abierta sin clientes esperando.

La clave de este tipo de simulaciones está en crear procesos múltiples, se puede simular el multiprocesamiento teniendo cada función dentro de un bucle de programa principal haciendo algún trabajo y volviendo a repartir el tiempo entre las funciones. Hay que tener en cuenta que la simulación no tiene en cuenta situaciones accidentales que pueden hacer que la cola se detenga temporalmente.

-**Administración de una cartera de clientes:** El administrar una cartera de acciones, se basa en distintas teorías y suposiciones de muchos factores, que no se pueden conocer fácilmente a no ser que se pertenezca a ese mundo. Existen estratégias de venta y compra que se basan en el análisis estadístico sobre los precios de las acciones y diferentes factores: tal como la correlación con el precio del petróleo y hasta con los ciclos de la luna. Por lo que se puede pensar que la bolsa es difícil de simular, sin embargo, el problema en sí constituye la solución; debido a que la bolsa es tan compleja, se puede ver como la composición de sucesos ocurren aleatoriamente. Esto significa que se puede simular la bolsa como una serie de sucesos aleatorios inconexos. A este tipo de simulación se le denomina "método de paseo aleatorio", cuyo término viene del clásico experimento del borracho deambulando por la calle pasando aleatoriamente de farola en farola.

Las operaciones de comprar, vender y mantener las acciones son obvias. Cuando se vende al descubierto, se venden acciones que no se tienen, con la esperanza de poder comprarlas en poco tiempo más baratas. La venta al descubierto es una manera de hacer dinero cuando el mercado va hacia abajo.

Cuando se compra a crédito, se usa, por un pequeño interés, el dinero de la agencia de cambio y bolsa, para financiar parte del costo de las acciones que se compran. La idea que se esconde tras la compra a crédito es que si las acciones suben mucho, se saca más dinero del que se podría haber sacado comprando menos acciones con el dinero realmente disponible. Con esto se saca dinero sólo cuando la bolsa sube.

B. Muestreo.

Cuando se analiza una cantidad muy grande de datos, puede resultar impráctico examinarlos todos; una cantidad pequeña de datos seleccionados de acuerdo con un método aleatorio puede servir como resultado válido.

C. Criptografía.

En esta área lo principal es codificar un mensaje de tal forma que éste no pueda leerlo nadie, excepto la persona indicada. Un camino para escribir el mensaje de esta forma es la utilización de una secuencia pseudo-random al codificar el mensaje.

D. Análisis Numérico.

Existen ingeniosas técnicas para resolver complicados problemas numéricos usando números aleatorios. Varios libros han sido escritos con este objetivo.

E. Programas.

Los valores aleatorios son buen origen de los datos para pruebas de eficiencia de los programas implementados en las computadoras.

F. Toma de desiciones.

La habilidad de tomar decisiones fuera de prejuicio es útil en programas implementados en computadoras por ejemplo, en situaciones en la cual una decisión fija implica una ejecución lenta. La aleatoriedad es una parte esencial en las estrategias de optimización en la teoría de juegos.

CAPITULO III

ALGORITMOS MATEMATICOS

Los **algoritmos matemáticos** -también llamados numéricos- se basan en el uso de las cuatro operaciones aritméticas; se consideran los algoritmos más importantes dentro de la ciencia de la computación, tanto que, una rama especial de las matemáticas, EL ANALISIS NUMERICO, se encarga de analizar este tipo de algoritmos diseñando también métodos para aproximar de una manera eficiente, las soluciones de los problemas expresados matemáticamente.

El objetivo primordial del análisis numérico es desarrollar métodos que reduzcan a las cuatro operaciones aritméticas, operaciones más complicadas tales como la integración, la diferenciación y la resolución de varios tipos de ecuaciones. Esta reducción comúnmente no da la respuesta exacta, pero si la da con cierto grado de precisión.

En resumen a esta breve introducción, podemos decir que los algoritmos matemáticos consisten de una secuencia de operaciones algebraicas y lógicas que producen una aproximación al problema con una tolerancia o precisión predeterminada, lo cual, aunado al grado de dificultad de implantación del algoritmo darán la medida de eficiencia de dicho método.

1.1 DEFINICION Y CARACTERISTICAS DE LOS ALGORITMOS MATEMATICOS.

En el capítulo anterior definimos a un algoritmo seminumérico como un algoritmo que trata con objetos numéricos (la entrada y la salida son números); si además este algoritmo procesa esos objetos matemáticamente recibe el nombre de algoritmo numérico o matemático, por lo que se puede decir que todo algoritmo numérico es seminumérico y no viceversa. Este tipo de algoritmos se basan en el empleo de las cuatro operaciones aritméticas y normalmente se dan en forma de instrucciones verbales o diversas clases de fórmulas o esquemas.

Dado que la mayoría de procesos matemáticos pueden reducirse a las cuatro operaciones aritméticas, los algoritmos numéricos se emplean ampliamente, ya que aún cuando esta reducción no proporciona la respuesta exacta, la da hasta con cualquier grado de exactitud deseado. Por ello, cuando se emplean los algoritmos numéricos es necesario introducir una condición que imponga un número máximo de iteraciones para eliminar la posibilidad de caer en un ciclo infinito, posibilidad que puede surgir cuando el método no es apropiado para la función dada (la sucesión diverge).

Una de las más importantes aplicaciones de las computadoras es precisamente la solución de problemas matemáticos por medio de algoritmos numéricos. El estudio de esos problemas y de las técnicas para su solución es un importante campo de las matemáticas y de la computación denominado ANALISIS NUMERICO, cuyo objetivo principal es diseñar y mejorar métodos para aproximar de una manera eficiente la solución de problemas expresados matemáticamente (métodos complicados tales como la integración, la diferenciación y la resolución de varios tipos de ecuaciones). Las ideas básicas sobre las cuales se apoyan la mayoría de las técnicas numéricas actuales se conocen ya desde hace algún tiempo, al igual que los métodos usados para predecir las cotas de error máximo que se pueda suscitar al aplicar el método.

La eficiencia de los algoritmos numéricos depende tanto de la precisión que se requiera como de la facilidad con que pueda implementarse, por lo que la elección del método apropiado para aproximar la solución de un problema se encuentra influenciado por los cambios tecnológicos en las computadoras y calculadoras. Un factor limitante respecto a la eficiencia de un algoritmo podría ser la capacidad de almacenamiento en la computadora (sin embargo en la actualidad ya no lo es tanto), a pesar de que el costo asociado con los tiempos de cómputo es desde luego un factor importante.

Por último, es importante tener presente que en el área de las matemáticas una clase de problemas se considera resuelto cuando se encuentra un algoritmo que la resuelve y el descubrir esos algoritmos es la meta natural de las matemáticas. Si no existe un algoritmo para resolver todos los problemas de un tipo dado, siempre es posible descubrir un procedimiento que resuelva ciertos problemas de ese tipo, aún cuando dicho algoritmo no sea aplicable a otros casos.

3.2 ERRORES.

Los errores no son inherentes al algoritmo por sí mismo, más bien son una función de la naturaleza y estructura del dispositivo seleccionado para ejecutar el algoritmo. La computadora es naturalmente, un dispositivo finito cuya organización convencional de su memoria impone ciertos contrastes en la representación de los números, la longitud física de la palabra indica no sólo el máximo valor que esa medida pueda contener sino también la ocurrencia de la representación. Dado que el número de decimales que pueden ser acarreados es finito, no es posible obtener una representación exacta de muchos números reales. La representación en una computadora de 1/3 por ejemplo, termina después de cierto número de dígitos decimales por lo que en ese momento ya se está introduciendo un error.

Como ya se ejemplificó la translación de un número de un sistema real numérico a una representación de computadora puede introducir un error inicial muy grande en los datos de los cuales depende el resultado final. Este error puede ser propagado por el desarrollo de las operaciones aritméticas involucradas, lo cual implica que el resultado final no será el correcto y/o el esperado. Por lo tanto la selección de un algoritmo eficiente depende no sólo de un cuidadoso análisis sino también de las características físicas de la computadora.

Un cálculo que involucra números reales en una computadora está sujeto a diferentes tipos de error, de los cuales algunos siempre están presentes.

3.2.1 Tipos de error.

En los cálculos numéricos existen tres tipos de error comunes: error inherente, error por truncamiento y error de redondeo.

A. ERROR INHERENTE.

La mayoría de valores numéricos obtenidos experimentalmente contienen este tipo de error, el cual surge por la incertidumbre

del instrumento de medición. Generalmente es conveniente expresar explícitamente el error límite de un resultado experimental; por ejemplo una medida 22.8 0.04 grados centígrados indica que la temperatura no es menor que 22.76 °c y no es mayor que 22.84 °c. Este tipo de error también está presente en aproximaciones decimales infinitas tales como: π ,e, $\sqrt{2}$, y 1/3 ya que no tienen representación finita exacta. algunos números de hecho tienen una representación finita en un sistema numérico pero no en otro; por ejemplo 1/10 tiene una representación decimal finita pero no una representación binaria finita, por lo que una computadora binaria no da exactamente 1.0 como respuesta a 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1 + 0.1+ 0.1

B. ERROR POR TRUNCAMIENTO.

Este tipo de error ocurre cuando

1. Un proceso matemático infinito se representa por un proceso finito; por ejemplo el cálculo del seno $X = x + x^3/3! + x^5/5! + x^7/7! + x^9/9! + \ldots$ en el cálculo práctico, la fórmula es truncada después de un número finito de términos, lo cual implica que el cálculo del valor es inexacto.

2. Cuando un proceso finito es aproximado por un número menor de iteraciones que las que requiere.

   Por ejemplo una sumatoria de un número grande de términos, muchos de los cuales son muy pequeños:

   $$\sum_{n=1}^{1000\,000} 1/n! = 1 + 1/2 + 1/6 + 1/24 + 1/120 + \ldots + 1/10000000!$$

   como se puede observar los números decrecen muy rápidamente y tal suma podría luego, ser truncada.

3. El error que ocurre cuando se eliminan dígitos a la derecha después del punto decimal de un número sin redondeo también se denomina error por truncamiento; por ejemplo 9.2344778 podría ser truncado a 9.2344 introduciendo en este último número dicho error

C. ERROR DE REDONDEO.

Son los errores resultantes de redondear un número decimal a **n** lugares decimales sumando 5 a el **(n+1)-ésimo** dígito a la derecha del punto decimal y entonces eliminar todos los dígitos a la derecha del **n-ésimo** dígito.

Ejemplo

Redondear 3.14159 a 3 lugares decimales

3.14159 + 0.0005 = 3.14209 Resultado 3.142

El error resultante es (3.14200 - 3.1419) = 0.00041.

Cuando un número es redondeado a n lugares después del punto decimal, el error siempre es menor o igual a $5 \times 10^{-(n-1)}$. El error de redondeo puede ser resultado del corrimiento de valores previo al cálculo; suponga por ejemplo que se desea sumar dos números reales digamos 999.0 y 1.12954, si asumimos la representación de punto flotante a 7 lugares decimales, entonces esos valores se almacenan como 0.9990000×10^{3} y 0.1129540×10^{1}; antes de que se realice la suma, se debe recorrer la parte fraccional del número más pequeño, esto es 0.1129540×10^{3} se cambia a 0.0011295×10^{3}, por lo que en tal operación cierta precisión se pierde.

3.2.2 Propagación del error.

Un error puede ser expresado en términos de absoluto, relativo y porcentaje. El error absoluto es simplemente la diferencia entre el valor exacto de un número y su aproximación. El error relativo es el valor absoluto dividido por el valor exacto, por lo tanto se encuentra en relación a la distancia y la medida. Entre más pequeña sea la distancia, mayor será el error y entre más pequeña sea la medida menor será el error. El error de porcentaje es el error relativo multiplicado por el 100%. Por ejemplo si $\bar{x}$ es una aproximación del valor exacto de x. El error absoluto de $\bar{x}$ es $|x-\bar{x}|$; el error relativo es $(x-\bar{x})/x$ y el error porcentaje es $((x-\bar{x})/x)*100\%$.

Dado que el valor de x no se conoce, es conveniente definir el error relativo como $(x-\bar{x})/\bar{x}$ y el error porcentaje como $((x-\bar{x})/\bar{x})*100\%$.

Cuando las operaciones aritméticas involucran dos números aproximados, el resultado, es por lo tanto una aproximación, cuyo error puede ser muy grande y se dice que al efectuar este tipo de operaciones el error se ACUMULA o se PROPAGA, si se conoce el error obligado de los operandos, entonces el error del resultado puede estimarse.

Ahora $x = \bar{x} + e \qquad y = \bar{y} + e$

donde

$\bar{x}$ es una aproximación de x

$\bar{y}$ es una aproximación de y

e_x es el error absoluto de $\bar{x}$

e_y es el error absoluto de $\bar{y}$

Considerando las operaciones aritméticas básicas de adición, substracción, multiplicación y división:

1. **Adición.**

Para la suma de X y Y tenemos

$$X + Y = (\bar{x} + e_x) + (\bar{y} + e_y) = (\bar{x} + \bar{y}) + (e_x + e_y)$$

por lo que el error absoluto de la suma es

$$e_{x+y} = e_x + e_y$$

y el error relativo

$$r_{x+y} = (e_{x+y}/(\bar{x} + \bar{y})) = (e_x + e_y)/(\bar{x} + \bar{y})$$

Note que el error relativo de la suma es un valor intermedio de los errores relativos de los dos operadores.

2. **Substracción.**

La diferencia entre X y Y es

$$X - Y = (\bar{x} + e_x) - (\bar{y} + e_y) = (\bar{x} - \bar{y}) + (e_x - e_y)$$

entonces

$$e_{x-y} = e_x - e_y$$

y

$$r_{x-y} = (e_{x-y}/(\bar{x} - \bar{y})) = (e_x - e_y)/(\bar{x} - \bar{y})$$

Cuando X y Y son aproximadamente iguales el denominador es muy pequeño y entonces el error relativo puede ser extremadamente grande.

3. **Multiplicación.**

Para la multiplicación de X y Y

$$XY = (\bar{x} + e_x)(\bar{y} + e_y) = \bar{x}\bar{y} + \bar{x}e_y + \bar{y}e_x + e_x e_y$$

Asumiendo que e_x y e_y son mucho más pequeños que $\bar{x}$ y $\bar{y}$, se espera que los términos que involucran $\bar{x}$ y $\bar{y}$ dominen. Esto es que el término $e_x e_y$ contribuya muy poco al tamaño de e_{xy}. Entonces el valor absoluto del producto es $e_{xy} \cong \bar{x}e_y + \bar{y}e_x$; donde el símbolo $\cong$ se lee como "aproximadamente igual a". El error relativo del producto es

$$r_{xy} = e_{xy}/\bar{x}\bar{y} = (\bar{x}e_y + \bar{y}e_x)/\bar{x}\bar{y} = e_x/\bar{x} + e_y/\bar{y}$$

aproximadamente igual a la suma de los errores relativos de los operandos.

4. División.

Para la división de X y Y

$$X/Y = (\bar{x} + e_x)/(\bar{y} + e_y)$$

racionando el denominador (multiplicándolo por $(\bar{y} - e_y)/(\bar{y} - e_y)$) tenemos

$$x/y = (\bar{x}\bar{y} + \bar{y}e_x - \bar{x}e_y - e_x e_y)/(\bar{y}^2 - e^2 y)$$

Dado que esperamos que e_x y e_y sean muy pequeñas nos podemos olvidar de los términos que involucran potencias de productos de e_x y e_y .

$$x/y \cong (\bar{x}\bar{y} + ye_x - \bar{x}e_y)/y^2$$

lo cual puede escribirse como

$$x/y \cong \bar{x}/\bar{y} + \bar{x}/\bar{y}(e_x/\bar{x} - e_y/\bar{y})$$

donde el error absoluto es

$$e_{x/y} = \bar{x}/\bar{y}(e_x/\bar{x} - e_y/\bar{y})$$

y el error relativo del cociente

$$r_{x/y} = (e_x/y)/(\bar{x}/\bar{y}) = e_x/\bar{x} - e_y/\bar{y}$$

es aproximadamente igual a la diferencia entre los errores relativos del numerador y del denominador.

Note que el signo de un error puede ser positivo o negativo, consecuentemente, los errores que incurren en la multiplicación y división no son necesariamente más grandes que el de la substracción y división.

Ahora bien, supongamos que E_n representa el crecimiento del error después de n operaciones subsecuentes si $/E_n/ \cong CnE$; donde C es una constante independiente de n se dice que el crecimiento del error es lineal. Si $/E_n/ = K^n E$ para alguna $K>1$, el crecimiento del error es exponencial.

El crecimiento lineal del error generalmente es inevitable y cuando c y E son pequeños, los resultados son generalmente aceptables. El crecimiento exponencial del error debe ser evitado, ya que el término K^n será grande aún para valores relativamente pequeños de n. Esto lleva a imprecisiones inaceptables, no importando el tamaño de E, por lo tanto un algoritmo que exhibe un crecimiento de error lineal es estable, mientras que un algoritmo cuyo crecimiento de error es exponencial se dice que es inestable.

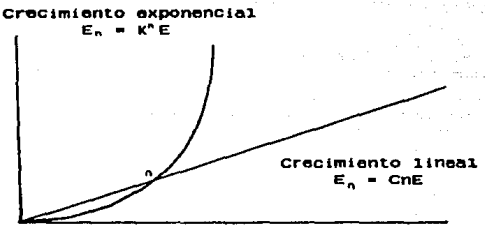


Fig 3.1 Crecimiento del error.

3.2.3 Métodos iterativos.

El error nunca puede ser eliminado totalmente; pero en muchos casos a través de técnicas cuidadosamente consideradas es posible tener el error en límites aceptables.

El estudio de cálculos numéricos se centran principalmente en métodos denominados **Métodos iterativos**; este tipo de métodos inician con una aproximación inicial a la solución del problema S_0, de esta solución se deriva una mejor aproximación S_1 y así sucesivamente; este proceso iterativo puede continuar a través de un número de soluciones subsecuentes S_2, S_3, S_4, ... Sn; sin embargo la precisión sólo puede llevarse hasta cierto punto. La persona interesada en la solución puede especificar qué considera como error aceptable, asumiendo por cuestiones de simplicidad que el error absoluto E ha sido declarado aceptable. Nuestro proceso iterativo termina cuando dos soluciones sucesivas consienten este error de tolerancia, esto es, cuando

$$/Si - Si-_1/ <= E$$

En este punto se dice que la técnica ha convergido al resultado Si.

El estudio de convergencia es muy importante en los cálculos numéricos. Existen distintas razones por las cuales una técnica de solución puede fallar al converger, por ejemplo, la tolerancia deseada por el usuario puede ser muy pequeña, o la técnica de solución por sí misma puede ser inapropiada para circunstancias particulares del problema en cuestión. Una buena práctica de programación es imponer un criterio de paro adicional: terminar el proceso después de un número dado de iteraciones.

Las técnicas de solución numérica se emplean por varias razones, por ejemplo, cuando la solución no puede ser obtenida por otros medios, tal como la técnica analítica. Para algunas categorías del problema, las técnicas de solución analítica simplemente no existen. En otros casos aún cuando las técnicas analíticas existan, la solución puede ser muy difícil de derivar mientras que las técnicas numéricas ofrecen métodos directos para calcular soluciones satisfactorias.

3.3 ANALISIS DE ALGORITMOS MATEMATICOS.

Describiremos a continuación algunos algoritmos matemáticos relacionados con problemas matemáticos muy conocidos y de gran importancia.

3.3.1 Búsqueda de raíces de funciones no lineales.

La raíz de una función de una sola variable se define como el valor de la variable que da como resultado cero para la función; por lo tanto la raíz es el punto o puntos en los cuales una función cruza el eje de las x.

Para las funciones lineales, las raíces son fáciles de encontrar; por ejemplo, la raíz de la función f(x)=x-3 se encuentra igualando la ecuación x-3 con cero y despejando x:

$$x - 3 = 0 \qquad \rightarrow \qquad x = 3$$

Para funciones no lineales (funciones con potencia de x mayores a 1) las técnicas de solución no son tan simples; para polinomios de segundo grado tenemos funciones de la siguiente forma:

$$f(x) = ax^2 + bx + c$$

cuyas raíces son dadas por la fórmula siguiente

$$x = (-b \pm \sqrt{b^2-4ac})/2a$$

La solución para polinomios de tercer y cuarto grado son más complicadas y fórmulas para polinomios de grado más grande no existen.

Ahora supongamos que deseamos considerar funciones no polinómicas de x tal como

$$\sin(x)-x+1 = 0$$

dado que no existe una fórmula general para las raíces de este tipo de ecuaciones, se desea un método de solución por ejemplo: la ecuación anterior se puede escribir como:

$$\sin(x) = x-1$$

si graficamos estas dos ecuaciones:

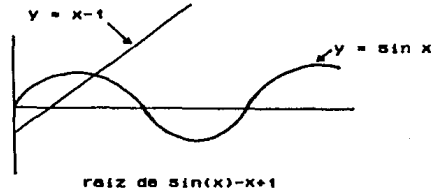


Fig 3.2

La gráfica muestra la intersección de $y = \sin(x)$ y $y=x-1$; la solución es entonces el punto de intersección de las dos curvas. Este método no es aceptado porque su aproximación es muy pobre.

En general las funciones trigonométricas o logaritmicas, son funciones trascendentales y no existen fórmulas generales para encontrar sus raíces.

En esta sección analizaremos algunos métodos para encontrar las raíces de una función; es decir para encontrar los valores de a que satisfagan $f(x) = 0$.

A. METODO DE ITERACIONES DE PUNTO FIJO.

Se desean encontrar todas las soluciones de la ecuación $f(x)=0$ para una $f(x)$ dada; esto es, se desea encontrar todos los valores de x los cuales hacen $f(x)=0$. Un método para realizar esto es llamado **Método de iteración de punto fijo** y está dado como sigue:

La iteración de punto fijo es un proceso que realiza aproximaciones sucesivas las cuales progresivamente encierran a la raíz. Un punto inicial x_0 es usado como el punto de partida de la iteración, después cada nueva aproximación x_{n+1} se obtiene por medio de la ecuación $x_{n+1} = g(x_n)$. La función g es seleccionada si $x=r$ es una raíz de $f(x)=0$ entonces $g(r)=r$; lo cual implica que si en una aproximación se llega exactamente al valor de la raíz, este valor permanecerá a través de iteraciones sucesivas.

La función g es obtenida por manipulación algebraica de la ecuación f(x)=0 en la forma g(x)=x.

Existen muchos caminos para encontrar g para una función dada f. Algunos de esos caminos pueden arrojar una función g que logre que la iteración de punto fijo converga; otros caminos sin embargo, pueden dar una función g que ocasione que la iteración de punto fijo no converja. La condición suficiente para garantizar la convergencia de g es $|g'(x)|<1$; para toda x en la región que contiene la raiz r y todas las aproximaciones $x_0, x_1, x_2, \ldots$ donde g'(x) es la derivada de g en x.

Considerando la función f(x) = (x+1)/(x-1) se desea encontrar la raiz de la ecuación, o sea (x+1)/(x-1)=0 esta ecuación se puede reescribir como:

$$(x+1)/(x-1)+1 = 1$$

$$(x+1)/(x-1) + (x-1)/(x-1) = 1$$

$$(x+1+x+1)/(x-1) = 1$$

$$2x/(x-1) = 1$$

$$2x = x-1$$

$$2x+1 = x$$

Ecuación que tiene la forma requerida g(x)=x. Ahora examinaremos la derivada de g para determinar si las iteraciones de punto fijo pueden o no converger:

$$g(x) = 2x+1$$

$$g'(x)= 2$$

Note que $|g'(x)|<1$ para cualquier x; por lo tanto no se puede garantizar que la iteración de punto fijo pueda converger.

Ahora, si aplicamos el metodo con un valor inicial $x_0=0$ las iteraciones serían:

$$x_0 = 0$$

$$x_1 = g(x_0) = 2x_0 + 1 = 0 + 1 = 1$$

$$x_2 = g(x_1) = 2x_1 + 1 = 2 + 1 = 3$$

$$x_3 = g(x_2) = 2x_2 + 1 = 6 + 1 = 7$$

Claramente esta iteración no converge a la raiz (la cual es r=-1). Ahora, si se realiza otra formulación g(x)=x podemos encontrar una iteración en la cual converja:

$$f(x)=(x+1)/(x-1) = 0$$

$$(x+1+1-1)/(x-1) = 0$$

$$(x-1+2)/(x-1) = 0$$

$$(x-1)/(x-1) + 2/(x-1) = 0$$

$$1 + 2/(x-1) = 0$$

$$2/(x-1) = -1$$

$$-2/(x-1) = 1$$

$$-2 = x-1$$

$$-2+1 = x$$

$$x = -1$$

Esta ecuación es de la forma g(x)=x con g(x)=-1, la derivada g'(x)=0; obviamente /g'(x)/<1 para toda x; por lo que se garantiza la convergencia. Iniciando el método con $X_0 = 0$ tenemos una convergencia muy rápida

$$X_0 = 0$$

$$X_1 = g(X_0) = -1$$

$$X_2 = g(X_1) = -1$$

...

$$Xn = g(Xn-_1) = -1$$

En general la convergencia por el método de iteración de punto fijo es muy lenta y requiere necesariamente de derivadas para determinar la convergencia, por lo cual el método no funciona cuando la derivada es desconocida o no existe.

8. METODO DE BISECCION (METODO DE BUSQUEDA BINARIA).

Considerando la función f(x), se desea encontrar un valor x que satisfaga f(x)=0. El método de bisección inicia seleccionando dos valores de x: x_1 y x_2 en los cuales la función evaluada tenga diferentes signos, esto es un $f(x_1)$ positivo y un $f(x_2)$ negativo, por lo que $f(x_1)f(x_2)<0$. Si se asume que f(x) es continua en (x_1, x_2) entonces, existe una raiz entre esos puntos; si la función nunca cambia de signo este método no puede ser aplicado ya que dicha función nunca cruza el eje de las x y por lo tanto no tiene raices.

El procedimiento de este método es el siguiente:

1) Encontrar r, el punto medio de (x_1,x_2), donde $f(x_1)$ y $f(x_2)$ son opuestos en signo.

2) Si $f(r) = 0$ entonces la raiz es r. Terminar.

   Si $/x_1 - x_2/$ es algún estado de tolerancia entonces la raiz es x_1 o x_2. Terminar.

3) Si $f(r)$ tiene el mismo signo que $f(x_1)$ entonces r puede estar entre x_2 y r, entonces repetir el procedimiento con $x_1=r$.

4) Si $f(r)$ tiene el mismo signo que $f(x_2)$ entonces repetir el procedimiento con $x_2 = r$.

Por este método el intervalo de búsqueda (x_1,x_2) siempre contiene la raiz. El intervalo siempre se divide por la mitad hasta que el valor de la función en el punto medio del intervalo sea suficientemente cercano a cero o hasta que x_1 y x_2 se encuentren demasiado juntos.

ALGORITMO DEL METODO DE BISECCION.

Dada x_1 y x_2, dos valores tales que $f(x_1)f(x_2) < 0$ y E la aproximación deseada. Este algoritmo encuentra la raiz: un valor de x para el cual $f(x)=0$. Asumiendo que todos los valores son reales.

```
FOR  i:=1 TO 30 DO
|Realizar a lo mucho 30 iteraciones|
BEGIN
     raiz:=(x2+x1)/2;
     IF FUNCION(raiz)=0 or ABS(x2-x1) < E THEN
     BEGIN
          WRITE('RAIZ' :, FUNCION(RAIZ));
          EXIT;
     END;
     IF FUNCION(raiz)*FUNCION(x1) < 0 THEN
        x2 := raiz
     ELSE
        x1 := raiz;
END;
WRITE ('RAIZ no encontrada en 30 iteraciones');
EXIT.
```

Gráficamente esto sería:

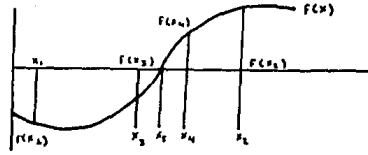


Fig 3.3 Método de bisección.

Iteración	Intervalo	Punto Medio	
1	(x_1, x_2)	x_3	
2	(x_3, x_2)	x_4	
3	(x_3, x_4)	x_5	Raíz

El método de bisección aún cuando conceptualmente es un proceso claro, tiene el inconveniente de que converge muy lentamente (ya que el intervalo puede ser muy grande antes de que /xi-xj/ sea suficientemente pequeño y más aún, una buena aproximación intermedia puede ser desechada sin darnos cuenta (si existe más de una raíz). Sin embargo, el método tiene una propiedad importante: siempre converge a la solución.

C. METODO DE LA SECANTE

El método de la secante converge más rápidamente que el método de bisección. Este método requiere de dos puntos iniciales x_1 y x_2. La curva de la función es aproximada por una línea recta (la línea secante) la cual es extrapolada o interpolada (dependiendo de los signos de los valores de la función) a un tercer punto.

La aproximación a las raíces es el punto en el cual la línea calculada cruza el eje de las x. El proceso se repite usando siempre los dos últimos puntos calculados para obtener la siguiente aproximación, hasta que se obtiene un valor con la tolerancia deseada.

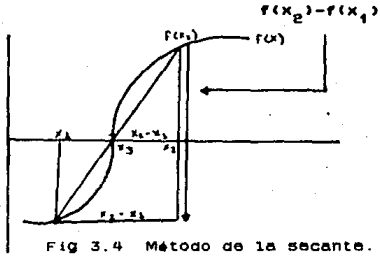


Fig 3.4 Método de la secante.

Por ejemplo encontrar la raiz de f(x)=0 por el método de la secanta donde

a) x_2 y x_3 son de signo opuesto

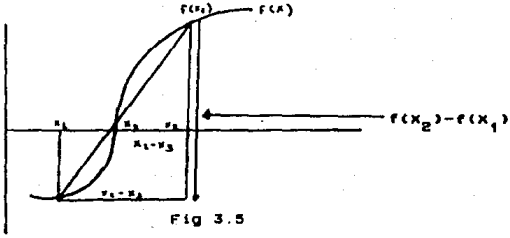


Fig 3.5

De este dibujo podemos ver que:

$(x_2-x_3)/(x_2-x_1) = f(x_2)/(f(x_2)-f(x_1))$

y entonces $x_3 = x_2 - ((x_2-x_1)/(f(x_2)-f(x_1)))f(x_2)$

y entonces $x_3 = x_2 - ((x_2-x_1)/(f(x_2)-f(x_1)))f(x_2)$

b) x_1 y x_2 son del mismo signo

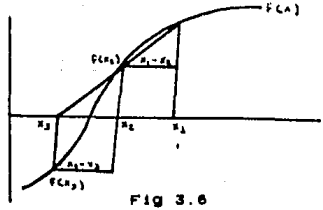


Fig 3.6

Como podemos observar existen dos triángulos semejantes y tenemos que:

$$(x_2-x_3)/(x_1-x_2) = f(x_2)/(f(x_1)-f(x_2))$$

lo cual puede expresarse como:

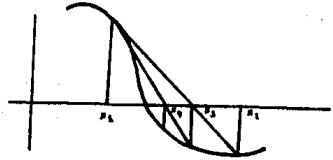
$$x_3 = x_2 - ((x_2-x_1)/(f(x_2)-f(x_1)))f(x_2)$$

en ambos casos resulta la misma fórmula, por lo que esto puede ser generalizado como:

$$Xn+_1 = Xn - ((Xn - Xn-_1)/(f(Xn)-f(Xn-_1)))f(Xn)$$

Los siguientes diagramas muestran cómo aplicando el método de la secante podemos encontrar f(x)=0

Aproximándose a la raíz por un lado



Aproximándose a la raíz por ambos lados

Fig 3.7

ALGORITMO DEL METODO DE LA SECANTE

Dado x_1 y x_2, los dos valores iniciales y E (grado de precisión); este algoritmo encuentra la raiz (valor de x para f(x)=0), asumiendo que todas las variables son reales.

```
1. Limite_inferior := x1;
   Limite_superior := x2;
2. FOR  i:=1  TO  30  DO
   {Desarrollar a lo mucho 30 iteraciones}
   BEGIN
      raiz := Limite_superior - ((Limite_superior -
           Limite_inferior)/(FUNCION(Limite_superior) -
           FUNCION(Limite_inferior)))*FUNCION(Limite_superior);
      IF ABS(FUNCION(raiz)) < E THEN
      BEGIN
           WRITE('RAIZ',FUNCION(RAIZ));
           EXIT;
      END;
      Limite_inferior := Limite_superior;
      Limite_superior := raiz;
   END;
3  WRITE ('No se encontro la raiz en 30 iteraciones');

4.  EXIT.
```

El método de la secante puede fallar en algunos casos donde el método de bisección tiene éxito; De hecho si los valores iniciales no son cuidadosamente seleccionados, el método de la secante puede ser cero con un mínimo relativo en lugar de una raiz.

D. METODO DE NEWTON.

El método de Newton fue desarrollado por el señor Isacc Newton. En dicho método la curva de la función es aproximada por la tangente a la curva en el punto del valor previo.

Dada que la derivada de la función en x_1 -denotada por $f'(x_1)$- es igual a la inclinación de la linea de la tangente en x_1. En la figura (c) se puede observar que

$$f'(x_1) = f(x_1)/(x_1-x_2)$$

despejando x_2 $\quad x_2 = x_1 - f(x_1)/f'(x_1)$

más generalmente $\quad xn+_1 = xn - f(x)/f'(xn)$

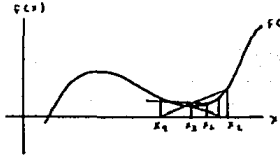


Figura (a)

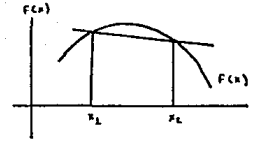


Figura (b)

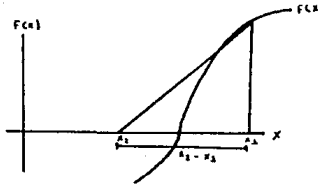


Figura (c)

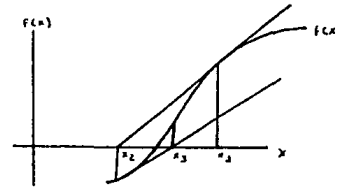


Figura (d)

Fig 3 8

Las figuras anteriores muestran aproximaciones sucesivas a las raíces por el método de Newton.

Las aproximaciones sucesivas se realizan hasta que f(Xn) es muy pequeña o hasta que Xn y Xn-1 son demasiado cercanas. El siguiente algoritmo para cuando f(Xn) es muy pequeña.

ALGORITMO DE NEWTON.

Dada una x inicial supuestamente la raíz de f(x) y el valor de aproximación deseada E, este algoritmo encuentra la raíz de f(x).

```
1. FOR  i:=1  TO  30  DO
   BEGIN
        raiz := x - FUNCION(x)/DERIVADA(x)
        IF ABS(FUNCION(raiz)) < E  THEN
        BEGIN
           WRITE ('raiz',FUNCION(raiz));
           EXIT;
        END;
        x := raiz;
   END;
```

2 WRITE('La raiz no se encontró en 30 iteraciones');
3. EXIT.

El método de NEWTON es más eficiente que los métodos anteriores pero tiene ciertas desventajas requiere de la derivada de la función (y frecuentemente ocurre que f'(x) es mucho más complicada y para calcularla requiere de un mayor número de operaciones aritméticas que f(x)), también es probable que el calculo de una linea falle como lo muestra la siguiente figura:

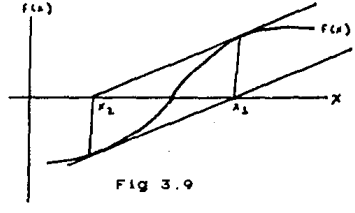


Fig 3.9

El método de NEWTON requiere de algunos preliminares del problema para una mejor solución.

Empleando este método podemos encontrar una fórmula para X_{n+1} que pueda ser utilizada para encontrar la raiz cuadrada de un número no negativo, por ejemplo $x = \sqrt{N}$ lo cual puede ser escrito como $x^2 - N = 0$.

Ahora $f(x) = x^2 - N$

La raiz de $f(x)=0$ puede ser la raiz cuadrada de N para alguna n, $f(x)=2x_n$ por lo tanto

$$x_{n+1} = x_n - (x_n^2 + N)/2x_n = 1/2(x_n + N/x_n)$$

es la fórmula a ser usada; Esta es de hecho, la fórmula empleada por muchas calculadoras electrónicas que sustituye la fórmula de la raiz cuadrada. De una forma similar se pueden encontrar fórmulas para raices cúbicas, cuartas, etc.

COMPARACION DE LOS METODOS.

El método de NEWTON converge más rapidamente que el método de la secante y el más lento es el de la bisección y el de iteración de punto fijo.

El método más seguro y fácil de calcular es el de bisección.

Cuando se tiene una buena aproximación y se puede calcular la derivada de la función el método de NEWTON es el mejor.

El método de la secante se debe emplear cuando se ésta familiarizado con la función pero no se conoce su derivada; En el caso en el cual no se está seguro del comportamiento de la función el método preferible es el de bisección.

3.3.2 Métodos de integración numérica.

La integración es una técnica estandar matemática para calcular el "área" de una figura aproximada:

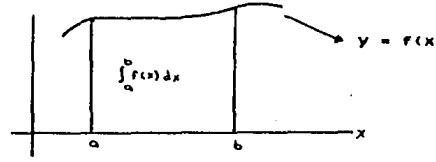


Fig 3.10

La integral está dada por la expresión $\int_a^b f(x)dx$ donde la función f(x) (la cual es continua) produce la curva mostrada y, a y b son dos puntos en el eje de las x.

Existen varios aspectos por los cuales la integración numérica (también conocida como cuadratura numérica) debe ser empleada en lugar de las técnicas clásicas analíticas:

1) Cuando sólo se conocen ciertos valores funcionales, pero no la función por sí misma.

2) Cuando la función es difícil o imposible de integrar analíticamente.

3) Cuando se emplea una computadora para evaluar la integral.

En este apartado examinaremos algunas técnicas que han sido desarrolladas para calcular aproximadamente el valor de las integrales reales.

A. REGLA DEL RECTANGULO.

La regla del rectángulo para la integración numérica ésta basado en la suposición de que el área bajo la curva puede ser aproximado sumando las áreas de un número finito de rectángulos.

Fig 3.11 Regla del Rectángulo.

Esta suposición da lugar a la siguiente aproximación:

$$\int_a^b f(x)dx \simeq \sum_{i=1}^{n-1} f(xi)h = h(f(x_0)+f(x_0+h)+\ldots+f(x_0+(n-1)h))$$

donde $h = (b-a)/n$ y $xi+_1 = xi + h$ $x_0 = a$

Claramente la precisión de la aproximación está en función a n (a menor tamaño del rectángulo mayor precisión); de hecho el límite cuando $n \to \infty$ constituye la definición formal de la integral. La regla del rectángulo generalmente se emplea en procedimientos iterativos llevados a cabo con alguna precisión provista, digamos E. La integral es evaluada repetidamente por valores progresivamente más grandes de n (o equivalentemente, valores más pequeños de h).

Existen dos criterios para determinar cuando la precisión requerida se ha alcanzado:

1) Criterio absoluto.

Ahora In es la última aproximación realizada y $In-_1$ la penúltima. El criterio absoluto se realiza cuando

$$/In - In-_1/ < E$$

2) Criterio relativo.

Es una indicación más formal:

$$(/In - In-_1/)/In < E$$

ALGORITMO DE LA REGLA DEL RECTANGULO

Dados a y b, los puntos extremos del intervalo de integración y E el grado de precisión; este algoritmo calcula AREA que es la aproximación de la regla del rectángulo de $\int_a^b f(x)dx$ donde f(x) es ya conocida por el algoritmo.

```
1.|Inicializar|
      AREA:=0;
```

```
        h     := b-a;
2.|Desarrollar máximo 20 iteraciones|
        FOR   i:=1   TO   20   DO
        BEGIN
3.        aproximacion_pasada := AREA;
4.        h := (b-a)/h;
5.        sum    := 0;
          FOR   j:= 0   TO   n-1   DO
            sum := sum + f(a+j*h);
6.        AREA   := h*sum;
7.        IF /((AREA - aproximacion_pasada)/AREA)/ < E THEN
          BEGIN
            WRITE (AREA,'Tolerancia encontrada':,i, iteraciones);
            EXIT;
          END;
8.        h := h/2;  |particiona el tamaño de los intervalos|
        END;
9.      WRITE('No se encontró la tolerancia después de 20
                iteraciones');
        EXIT
```

Este proceso puede ser muy costoso ya que la evaluación de la función f puede requerir cálculos extensivos, por lo tanto puede valer la pena guardar los valores de la función en un arreglo cuando son calculados y si van a ser empleados más tarde.

Dado que hi y ni denotan la longitud y número de subintervalos empleados en la **i-ésima** iteración respectivamente, **sumi** denota la suma de los valores de la función obtenida durante la **i-ésima** iteración y dado que $hi = hi_{-1}/2$ y $ni = 2*ni_{-1}$; note que:

$$sumi = f(a)+f(a+hi)+f(a+2*hi)+ \ldots +f(a+(ni_{-1})hi)$$

$$= f(a)+f(a+hi)+f(a+hi)+ \ldots +f(a+(ni_{-1}-1)hi_{-1})+f(a+ni_{-1}h)$$

$$= sumi_{-1} +f(a+hi)+f(a+3hi)+\ldots+f(a+(ni_{-1})hi)$$

Se puede optimizar calculo si se emplea el valor ya calculado de $sumi_{-1}$. Al incorporar esta modificación al algoritmo anterior sólo los pasos 1 y 5 sufren cambios:

```
1.  AREA := 0;
    h    := b - a;
    sum  := f(a);
5.  FOR  j:=1   TO   n-1   DO
      sum := sum + f(a+j*h)
```

Ejemplo

Encontrar la aproximación de la integral de e^x sobre el intervalo (1.6, 3.2) con h=0.2 y sólo una iteración

$$\int_{1.6}^{3.2} f(x)dx \approx 0.2\ (4.953 + 6.050 + 7.389 + 9.025 + 11.023 + 13.464 + 16.445 + 20.086) = 17.687$$

El valor real de la integral es $e^{3.2} - e^{1.6} = 19.580$ que comparado con el valor que arroja el método (17.687) denota un error muy grande; pero conforme n crece (y h decrece) se logra una mejor aproximación.

Una variación adicional de la regla podría ser emplear el valor de la función en el punto más alto de la función comprendido en el intervalo, o el punto medio del intervalo: $f((xi_{+1} + xi)/2)$; también es posible usar la regla del rectángulo con h_{+} distintas para cada componente de la suma (intervalos de longitud variable).

8. REGLA DEL TRAPEZOIDE.

La regla del trapezoide proporciona una aproximación más exacta a la integral que la regla del rectángulo; aún cuando son métodos muy similares: en lugar de usar rectángulos para aproximar el área, se emplean trapezoides formados por la secante de la curva en cada subintervalo, esto es la línea formada por unir los dos puntos funcionales de la curva en el subintervalo.

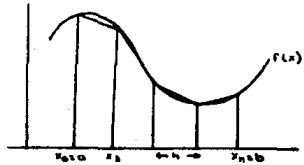


Fig 3.12 Regla del Trapezoide.

Aproximación por la regla del trapezoide de $\int_a^b f(x)dx$.

Este método pretende seguir la curvatura de la función más aproximadamente. El área de un trapezoide (una figura de cuatro lados con dos de ellos paralelos) está dada por la longitud promedio de los lados paralelos multiplicados por la distancia entre ellos; esto es, el área de un trapezoide entre xi y xi_{+1} está dada por la fórmula:

$$(f(xi) + f(xi_{+1}))/2(xi_{+1} - xi) = h/2(f(xi)+f(xi_{+1})$$

por lo que el área total de la integral está dada por:

$$\int_a^b f(x)dx = \sum_{i=0}^{n-1} h/2(f(xi)+f(xi_{+1}))$$

$$= h/2(f(x_0)+f(x_1)+f(x_1)+f(x_2)+\ldots+f(xn-_1)+f(xn))$$

$$= h/2(f(x_0)+2f(x_1)+2f(x_2)+\ldots 2f(xn-_1)+f(xn))$$

por ejemplo

$$\int_{1.6}^{3.1} f(x)dx = 0.2/2\ (4.953 + 2(6.050) + 2(7.389) + 2(9.025) + 2(11.023) + 2(13.464) + 2(16.445) + 2(20.086) + 24.353) = 19.645$$

Lo cual es mucho más cercana que la aproximación de 19.580 del ejemplo de la regla del rectángulo; pero, presenta aún un error de 0.065. La regla del trapezoide tiene de hecho un error de $O(h^2)$ que implica que conforme h se aproxima a 0; el error es más pequeño.

Esta regla dará el resultado exacto cuando se aplique a cualquier función cuya segunda derivada sea cero o en cualquier poliniomio de grado 1 o menor.

ALGORITMO DE LA REGLA DEL TRAPEZOIDE.

```
1.  AREA := 0;
    h    := b-a;
2.  FOR  i:=1  TO  20  DO
    BEGIN
3.        ultima_aproximacion := AREA;
4.        h := (b-a)/h;
5.        sum := 0;
          FOR  j:=0  TO  n-1  DO
            sum := sum + f(a+j*h) + f(a+(j+1)*h);
6.        AREA := h/2*sum;
7.        IF ABS((AREA - ultima_aproximacion)/AREA) < E  THEN
          BEGIN
               WRITE('La integral es', AREA,'Tolerancia
                     encontrada en,i,iteraciones');
               EXIT;
          END;
8.        h := h/2;
    END;
9.  WRITE ('La tolerancia no se encontró en 20 iteraciones');
    EXIT.
```

Al igual que en la regla del rectángulo es posible optimizar la regla del trapezoide usando los resultados de los cálculos previos en iteraciones subsecuentes; es también posible eliminar operaciones de multiplicaciones y divisiones empleando una fórmula reacomodada para la aproximación de la integral:

$$\int_a^b f(x)dx = h/2(f(x_0)+2f(x_1)+2f(x_2)+\ldots+2f(xn-_1)+f(xn))$$

$$= h((f(x_0)/2+f(x_1)+f(x_2)+f(x_3)+\ldots+f(xn-_1)+f(xn)/2)$$

Finalmente esta regla también se puede emplear para intervalos de longitud variable.

C. REGLA DE SIMPSON.

En la **regla de SIMPSON** los polinomios de segundo grado son aproximados a la curva; se emplea un polinomio para cada par de subintervalos calculando y adicionando su área a la aproximación de la integral al mismo tiempo, como se muestra en la siguiente figura:

Fig 3.13 Regla de Simpson.

En la regla del rectángulo se asume que el valor de la función en el intervalo (xi,xi+1) es f(xi) y f(xi+1); en la regla del trapezoide la función en el intervalo es aproximado a la línea de la secante. Por la regla de **SIMPSON** la función en el intervalo es aproximada por un polinomio de segundo grado, que debe de coincidir con los valores de la función en tres puntos consecutivos.

Al aproximar la integral $\int_a^b f(x)dx$ el intervalo (a,b) es dividido en n subintervalos iguales $(x_0,x_1),(x_1,x_2),\ldots (xn-_1,xn)$ donde n podría ser un número par. Ahora

$$\int_{a=x_0}^{b=x_n} f(x)dx = \int_{x_0}^{x_2} f(x)dx + \int_{x_2}^{x_4} f(x)dx + \int_{x_4}^{x_6} f(x)dx + \ldots \int_{x_{n-2}}^{x_n} f(x)dx$$

primero consideremos la regla de aproximación de $\int_{x_0}^{x_2} f(x)dx$ en la cual se desea encontrar una función cuadrática de la forma $g(x) = C_0 + C_1X + C_2X^2$ tal que

$$g(x_0) = f(x_0)$$

$$g(x_1) = f(x_1)$$

$$g(x_2) = f(x_2)$$

se podría entonces usar la función g(x) para aproximar f(x) en el intervalo (x_0,x_2)

$$\int_{x_0}^{x_2} f(x)dx = \int_{x_0}^{x_2} g(x)dx$$

Resolviendola para C_0,C_1,C_2 en las ecuaciones

$$f(x_0) = C_0 + C_1x_0 + C_2x_0^2$$

$$f(x_1) = C_0 + C_1x_1 + C_2x_1^2$$

$$f(x_2) = C_0 + C_1x_2 + C_2x_2^2$$

Se podría obtener la función de aproximación

$$g(x) = C_0 + C_1x + C_2x^2$$

haciendo h = xi - xi-1 se encuentra

$C_0 = f(x_0)$

$C_1 = 1/h(-3/2f(x_0) + 2f(x_1) - 1/2f(x2))$

$C_2 = 1/h^2(1/2f(x_0) - f(x_1) + 1/2f(x2))$

Ahora

$$g(x) = f(x_0) + x/h(-3/2f(x_0) + 2f(x_1) - 1/2f(x_2)) + x^2/h^2(1/2f(x_0) - f(x_1) + 1/2f(x_2))$$

y

$$\int_{x_0}^{x_2} f(x)dx = \int_{x_0}^{x_2} g(x)dx = \int_{x_0}^{x_2} (xC_0 + x^2/2C_1 + x^3/3C_2)$$

$$= (x_2 - x_0)C_0 + (x_2 - x_0)^2/2C_1 + (x_2 - x_0)^3/3C_2$$

$$= 2hC_0 + 2h^2C_1 + 8/3h^3C_2$$

Substituyendo los valores obtenidos para los coeficientes C_0, C_1, y C_2

$$2hC_0 + 2h^2C_1 + 8/3h^3C_2 = 2hf(x_0) + 2h(-3/2f(x_0) + 2f(x_1) - 1/2f(x_2))$$
$$+ 8/3h(1/2f(x_0) + f(x_1) + 1/2f(x_2))$$
$$= h(2f(x_0) - 3f(x_0) + 4f(x_1) - f(x_2))$$
$$+ 4/3f(x_0) - 8/3f(x_1) + 4/3f(x_2))$$
$$= h/3(f(x_0) + 4f(x_1) + f(x_2))$$

generalizando a todo el intervalo (a,b)

$$\int_a^b f(x)dx = h/3(f(x_0) + 4f(x_1) + f(x_2)) + h/3(f(x_2) + 4f(x_3) + f(x_4))$$
$$+ \ldots + h/3(f(xn-_2) + 4f(xn-_1) + f(xn))$$
$$= h/3(f(x_0) + 4f(x_1) + 2f(x_2) + 4f(x_3) + \ldots + 2f(xn-_2)$$
$$+ 4f(xn-_1) + f(xn))$$

Esta fórmula constituye la definición de la regla de SIMPSON. Recordando que n puede ser siempre un número par. La regla de

SIMPSON es muy popular porque la fórmula puede girar sobre el valor exacto para todos los polinomios de tercer grado.

Regla de aproximación de SIMPSON

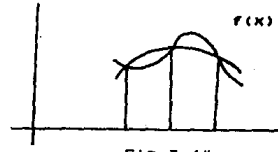


Fig 3.14

Otra importante razón para el empleo de esta regla es su pequeño error, sólo el $O(h^4)$.

Ejemplo

Encontrar la integral del siguiente polinomio x^3+2x^2+4 en el intervalo (0,4) con h=0.5.

$$\int_0^4 f(x)dx = 0.5/3\,(4 + 4(4.625) + 2(7) + 4(11.875) + 2(20) + 4(32.125) + 2(49) + 4(71.375) + 100) = 122.6666...$$

resultado que es exactamente igual a la respuesta analítica.

$$\int_0^4 x^3+2x^2+4 = x^4/4 + 2x^3/3 + 4x = 4^4/4 + 2(4)^3/3 + 4(4) = 122.666...$$

ALGORITMO DE SIMPSON.

Dado a,b y E, este algoritmo calcula AREA, la aproximación de la regla de SIMPSON de $\int_a^b f(x)dx$ con una aproximación de E.

```
1. AREA := 0;
   h    := (b-a)/2;
2. FOR  i:=1  TO  20  DO
   BEGIN
3.       aproximacion_pasada := AREA;
4.       h    := (b-a)/2;
5.       sum := f(a)+4*f(a+h)+f(b);
6.       FOR  j:=2  TO  n-2  DO
           sum := sum+2*f(a+j*h)+4*f(a+(j+1)*h);
7.       AREA := h/3*sum;
8.       IF ABS((AREA - aproximacion_pasada)/AREA) < E  THEN
         BEGIN
           WRITE('La respuesta es:',AREA,'tolerancia encontrada
                 en', i, 'iteraciones');
           EXIT.
         END;
   END;
9. WRITE ('La tolerancia no fue encontrada');
   EXIT;
```

COMPARACION DE LOS METODOS.

De los métodos vistos en este capítulo podemos decir que la regla del rectángulo se aproxima al área empleando una serie de rectángulos, es una regla simple de aplicar, pero involucra un error muy grande; la regla del trapezoide emplea trapezoides y es una mejor aproximación de la integral especialmente cuando se emplean intervalos muy pequeños. Empleando parábolas la regla de Simpson se aproxima a la curvatura tanto como es posible y presenta un error relativamente pequeño pero involucra mayor cálculo. Por lo que la eficiencia del método seleccionado está en relación a la complejidad y precisión que se pretenda tener en el problema que se va a resolver.

Cuando se escoge un método para una aplicación dada el tiempo de cálculo y la precisión del resultado son factores que juegan un papel muy importante. Para solucionar el problema de integración existe una gran variedad de métodos que minimizan el error pero su cálculo es más complicado (emplean mayor tiempo de cómputo), es recomendable el conocerlos para poder tener un panorama más amplio al analizar los métodos y poder seleccionar el método óptimo para el problema en cuestión.

3.3.3 Ecuaciones lineales simultáneas.

Como un tercer ejemplo de algoritmos matemáticos, mencionaremos algunos métodos conocidos para resolver conjuntos de ecuaciones lineales simultáneas. Este tema fue seleccionado porque al igual que los problemas anteriores (búsqueda de raíces e integración) es un problema frecuente en muchas áreas y por lo general involucra una considerable cantidad de datos.

La solución de un sistema de ecuaciones, es uno de los problemas más antiguos de las matemáticas que se presenta con frecuencia en la solución de una gran variedad de problemas reales. Se ha desarrollado una amplia colección de algoritmos para llevar a cabo la solución de los sistemas lineales, lo que indica que es engañoso el aparente carácter elemental del problema.

DEFINICIONES.

Ecuación algebraica lineal: Es aquella en donde en cada término de la ecuación aparece únicamente una variable o incógnita elevada a la primera potencia; por ejemplo

$$a_{11}x1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_{1n}xn = b_1$$

es una ecuación algebraica lineal, con variables $x_1, x_2, x_3, \ldots, xn$; en la cual los coeficientes $a_{11}, a_{12}, a_{13}, \ldots, a_1n$ y el término independiente b_1, son constantes reales (pueden ser complejas).

Un sistema de ecuaciones: Es un conjunto de ecuaciones que deben resolverse simultaneamente:

$$a_{11}x_1 + a_{12}x_2 + a_{13}x_3 + \ldots + a_1 x_n = b_1$$
$$a_{21}x_1 + a_{22}x_2 + a_{23}x_3 + \ldots + a_2 x_n = b_2$$
$$\ldots$$
$$a_{n1}x_1 + a_{n2}x_2 + a_{n3}x_3 + \ldots + a_n x_n = b_n$$

El siguiente esquema muestra la clasificación general de un sistema de ecuaciones (S.E.):

- S.E.
  - COMPATIBLES
    - DETERMINISTICO : Tiene una solución única
    - INDETERMINISTICO: Infinidad de soluciones.
  - INCOMPATIBLES

Si se aplica la definición de producto entre matrices un sistema de n ecuaciones algebraicas lineales con n incognitas puede escribirse en forma matricial:

$$\begin{bmatrix} a_{11} & a_{12} & a_{13} & \ldots & a_1 n \\ a_{21} & a_{22} & a_{23} & \ldots & a_2 n \\ . & . & . & \ldots & . \\ . & . & . & \ldots & . \\ . & . & . & \ldots & . \\ a_{n1} & a_{n2} & a_{n3} & \ldots & a_n n \end{bmatrix} \begin{bmatrix} x_1 \\ x_2 \\ . \\ . \\ . \\ xn \end{bmatrix} \begin{bmatrix} b_1 \\ b_2 \\ . \\ . \\ . \\ bn \end{bmatrix}$$

lo cual se define simbolicamente como AX=B; en donde A es la matriz del sistema y B el vector de términos independientes; cuando a la matriz A se le agrega el vector de términos independientes como última columna recibe el nombre de matriz expandida o aumentada y se representa como (A,B):

$$\left[\begin{array}{ccccc|c} a_{11} & a_{12} & a_{13} & \cdots & a_1n & b_1 \\ a_{21} & a_{22} & a_{23} & \cdots & a_2n & b_2 \\ . & . & . & \cdots & . & . \\ . & . & . & \cdots & . & . \\ . & . & . & \cdots & . & . \\ a_{n1} & a_{n2} & a_{n3} & \cdots & a_nn & bn \end{array}\right]$$

Solución de un sistema de ecuaciones: La solución de un sistema de ecuaciones es un conjunto de valores de las incógnitas que satisfacen simultaneamente a todas y cada una de las ecuaciones del sistema.

Combinación lineal de las ecuaciones: Una combinación lineal de las ecuaciones de un sistema es una ecuación obtenida al sumar miembro a miembro algunas de las ecuaciones del sistema multiplicando previamente cada una de ellas por alguna constante.

Ecuación consecuencia de un sistema: Una ecuación consecuencia de un sistema es aquella ecuación que se verifica para el mismo sistema de valores de las incógnitas que constituyen la solución del sistema; toda combinación lineal de las ecuaciones de un sistema es una consecuencia del mismo.

Sistemas equivalentes: Dos sistemas son equivalentes si ambos admiten como solución el mismo conjunto de valores de las incógnitas.

Teorema Fundamental de Equivalencia: Si en un sistema de ecuaciones, una ecuación es substituida por una combinación lineal de ecuaciones del sistema, se obtiene un nuevo sistema que es equivalente al anterior.

Operaciones permitidas en un sistema de ecuaciones: Existen tres operaciones que pueden ser desarrolladas en un sistema de ecuaciones sin alterar su solución:

1. La ecuación **Ei** se puede multiplicar por cualquier constante λ diferente de cero y se puede usar la ecuación resultante en lugar de **Ei**. Esta operación se denota como:

$$(\lambda Ei) \quad -> \quad (Ej) \qquad \lambda <> 0$$

2. La ecuación **Ej** puede multiplicarse por cualquier constante λ, sumarla a la ecuación **Ei** y emplear la ecuación resultante en lugar de **Ei**. Esta operación se denotará como:

$$(Ei + \lambda Ej) \quad -> \quad (Ei) \qquad \lambda <> 0$$

3. Las ecuaciones Ei y Ej se pueden intercambiar. Esta operación se denota como:

(Ei) <-> (Ej)

Por medio de una secuencia de las operaciones anteriores, un sistema lineal se puede transformar en un sistema lineal equivalente más fácil de resolver.

A. METODOS DIRECTOS DE SOLUCION DE ECUACIONES LINEALES.

Considere el siguiente sistema de dos ecuaciones con dos variables:

$$2x + 3y = 7$$

$$3x + 5y = 11$$

Esas ecuaciones pueden ser resueltas para x y para y de la siguiente manera:

Eliminar una variable: Para eliminar x de la segunda ecuación se divide primero la primer ecuación por 2, y se substrae después 3 veces la primer ecuación de la segunda

$$x + 3/2y = 7/2 \qquad 1/2y = 1/2 \qquad y = 1$$

subtituyendo el valor de y en la primer ecuación se obtiene x=2.

Este sistema ha sido resuelto por un método conocido como **ELIMINACION GAUSSIANA**, el cual emplea la matriz aumentada y consiste de los siguientes pasos:

1) ELIMINACION ADELANTADA

Iniciando en el primer renglón, se divide el renglón por el elemento de la diagonal (llamado pivote) para tener 1 en la diagonal y después, se manipula la matriz para tener ceros bajo la diagonal; se procede igual con los renglones restantes.

2) SUBSTITUCION HACIA ATRAS

Se manipula la matriz para tener ceros en toda la matriz excepto en la diagonal.

Cuando el algoritmo termina la solución del sistema se encuentra en la última columna de la matriz solución.

Ilustrando esto:

$$\left[\begin{array}{cc|c} 2 & 3 & 7 \\ 3 & 5 & 11 \end{array}\right]$$

1) Eliminación adelantada

$R_1/2$ $\left[\begin{array}{cc|c} \textcircled{1} & 3/2 & 7/2 \\ 3 & 5 & 11 \end{array}\right]$ R_2-3*R_1 $\left[\begin{array}{cc|c} 1 & 3/2 & 7/2 \\ 0 & 1/2 & 1/2 \end{array}\right]$

$R_2/(1/2)$ $\left[\begin{array}{cc|c} 1 & 3/2 & 7/2 \\ 0 & \textcircled{1} & 1 \end{array}\right]$

2) Substitución hacia atras

$R_1-3/2*R_2$ $\left[\begin{array}{cc|c} 1 & 0 & 2 \\ 0 & 1 & 1 \end{array}\right]$

Donde Ri denota el renglón i y los pivotes son encerrados en circulos

Las tres primeras matrices muestran los pasos de la eliminación adelantada y el último de la sustitución hacia atras, la solución se encuentra en la tercer columna de la matriz.

ALGORITMO DE ELIMINACION GAUSSIANA.

Dado n, **m=n+1** y una matriz aumentada, este algoritmo encuentra la matriz solución donde **AX=B** son variables e i,j y k son usados como indices de la matriz.

```
1. Eliminación adelantada.
   BEGIN
      FOR  i:=1  TO  n  DO
      BEGIN
2.       {Dividir cada elemento por el renglón pivote.}
          FOR  j:=1  TO  m  DO
              a[i,j] := a[i,j]/a[i,i];
3.       {Substraer un multiplicador de el renglón de cada
          renglón más bajo.}
          FOR  k:=i+1  TO  n  DO
             FOR  j:=i  TO  m  DO
                a[k,j]:=a[k,j]*a[i,j];
      END;
```

```
4.    {Substitución hacia atrás.}
      FOR  i:=n  DOWNTO  2  DO
         FOR  k:=1  TO  i-1  DO
         BEGIN
             a[k,m] - a[k,i]*a[i,m];
             a[k,i]:= 0;
         END;
5. END;
```

Existen diferentes variaciones al esquema de GAUSS: los renglones pueden ser divididos por su elemento diagonal después o al mismo tiempo que se está realizando el paso de eliminación adelantada, o aún después de la substitución hacia atrás; otra variación es la implementación del pivoteo (búsqueda del número más grande en magnitud para ser empleado como pivote y minimizar con ello los errores de redondeo); cuando se intercambian renglón y columna se dice que se emplea eliminación por pivoteo o pivoteo completo; cuando sólo se intercambian renglones se emplea la eliminación por pivoteo parcial (un intercambio de renglones corresponde a un simple cambio en el orden de ecuaciones y un intercambio de columnas es equivalente a modificar el orden de las variables).

Para ilustrar estas técnicas resolveremos el siguiente sistema de ecuaciones empleando **ELIMINACION GAUSSIANA** sin pivoteo, con pivoteo parcial y con pivoteo completo:

$$
\begin{aligned}
x_1 + 3x_2 - 2x_3 &= 7 \\
4x_1 - x_2 + 3x_3 &= 10 \\
-5x_1 + 2x_2 + 3x_3 &= 7
\end{aligned}
$$

I) SIN PIVOTEO

$$
\begin{bmatrix} 1 & 3 & -2 & 7 \\ 4 & -1 & 3 & 10 \\ -5 & 2 & 3 & 7 \end{bmatrix}
\begin{matrix} \\ R_2 - 4R_1 \\ R_3 + 5R_1 \end{matrix}
\begin{bmatrix} 1 & 3 & -2 & 7 \\ 0 & -13 & 11 & -18 \\ 0 & 17 & -7 & 42 \end{bmatrix}
$$

$$
\begin{matrix} R_2/-13 \\ \\ \end{matrix}
\begin{bmatrix} 1 & 3 & -2 & 7 \\ 0 & 1 & -0.846 & 1.384 \\ 0 & 17 & -7 & 42 \end{bmatrix}
\begin{matrix} R_1 + 2R_3 \\ R_2 + .846R_3 \\ \end{matrix}
\begin{bmatrix} 1 & 3 & 0 & 11.999 \\ 0 & 1 & 0 & 3.499 \\ 0 & 0 & 1 & 2.499 \end{bmatrix}
$$

$$R_1-3*R_2 \begin{bmatrix} 1 & 0 & 0 & 1.499 \\ 0 & 1 & 0 & 3.499 \\ 0 & 0 & 1 & 2.499 \end{bmatrix}$$

II) CON PIVOTEO PARCIAL

$$\begin{bmatrix} 1 & 3 & -2 & 7 \\ 4 & -1 & 3 & 10 \\ -5 & 2 & 3 & 7 \end{bmatrix} \quad R_1 <-> R_3 \begin{bmatrix} -5 & 2 & 3 & 7 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix}$$

$$R_1/-5 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix} \quad \begin{matrix} R_2-4*R_1 \\ R_3-1*R_1 \end{matrix} \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & .6 & 5.4 & 15.6 \\ 0 & 3.4 & -1.4 & 8.4 \end{bmatrix}$$

$$R_2<->R_3 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & 3.4 & -1.4 & 8.4 \\ 0 & 0.6 & 5.4 & 15.6 \end{bmatrix} \quad R_2/3.4 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & 1 & -.411 & 2.47 \\ 0 & .6 & 5.4 & 15.6 \end{bmatrix}$$

$$R_3/5.4 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & 1 & -.411 & 2.470 \\ 0 & 0 & 1 & 2.500 \end{bmatrix} \quad \begin{matrix} R_1+.6*R_3 \\ R_2+.411*R_3 \end{matrix} \begin{bmatrix} 1 & -.4 & 0 & 0.100 \\ 0 & 1 & 0 & 3.500 \\ 0 & 0 & 1 & 2.500 \end{bmatrix}$$

$$R_1+.4*R_2 \begin{bmatrix} 1 & 0 & 0 & 1.500 \\ 0 & 1 & 0 & 3.500 \\ 0 & 0 & 1 & 2.500 \end{bmatrix}$$

III) CON PIVOTEO COMPLETO

$$\begin{bmatrix} 1 & 3 & -2 & 7 \\ 4 & -1 & 3 & 10 \\ -5 & 2 & 3 & 7 \end{bmatrix} \quad R_1 <-> R_3 \quad \begin{bmatrix} -5 & 2 & 3 & 7 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix}$$

$$R_1/-5 \quad \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix} \quad \begin{matrix} R_2-4*R_1 \\ R_3-1*R_1 \end{matrix} \quad \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & .6 & 5.4 & 15.6 \\ 0 & 3.4 & -1.4 & 8.4 \end{bmatrix}$$

$$C_2 <-> C_3 \quad \begin{bmatrix} 1 & -.6 & -.4 & -1.4 \\ 0 & 5.4 & .6 & 15.6 \\ 0 & -1.4 & 3.4 & 8.4 \end{bmatrix} \quad \begin{matrix} R_3/3.555 \\ x_3 \\ x_2 \end{matrix} \quad \begin{bmatrix} 1 & -.6 & -.4 & -1.4 \\ 0 & 1 & .111 & 2.888 \\ 0 & -1.4 & 3.4 & 8.4 \end{bmatrix}$$

$$R_1+1.4*R_2 \quad \begin{bmatrix} 1 & -.6 & -.4 & -1.4 \\ 0 & 1 & .111 & 2.888 \\ 0 & 0 & 3.555 & 12.444 \end{bmatrix} \quad R_3/3.55 \quad \begin{bmatrix} 1 & -.6 & -.4 & -1.4 \\ 0 & 1 & .111 & 2.888 \\ 0 & 0 & 1 & 3.500 \end{bmatrix}$$

$$\begin{matrix} R_1+.4*R_3 \\ R_2-.111R_3 \end{matrix} \quad \begin{bmatrix} 1 & -.6 & 0 & 0 \\ 0 & 1 & 0 & 2.500 \\ 0 & 0 & 1 & 3.500 \end{bmatrix} \quad R_1+.6R_2 \quad \begin{bmatrix} 1 & 0 & 0 & 1.500 \\ 0 & 1 & 0 & 2.500 \\ 0 & 0 & 1 & 3.500 \end{bmatrix} \begin{matrix} x_1 \\ x_3 \\ x_2 \end{matrix}$$

La respuesta exacta de este sistema de ecuaciones es $x_1=1.5$, $x_2=3.5$ y $x_3=2.5$. El error total absoluto de la eliminación gaussiana sin pivoteo, con pivoteo parcial y con pivoteo completo es 0.0004, 0.0001 y 0.0001 respectivamente; como se puede observar pivoteando se reduce el error.

ALGORITMO DE ELIMINACION DE GAUSS-JORDAN.

Consiste en dividir el renglón por el elemento de la diagonal para tener 1 en esa posición y después manipular la matriz para tener ceros sobre y bajo la diagonal de la columna correspondiente. Esto se repite para todos los renglones de la matriz.

```
1. FOR  i:=1  TO  n  DO
   BEGIN
       FOR  j:=i  TO  m  DO
2.          a[i,j] := a[i,j] / a[i,i];
3.      FOR  k:=1  TO  n  DO
4.          FOR  j:=i  TO  m  DO
                a[k,j] := a[k,j] -  a[k,i]*a[i,j];
   END;
```

Es recomendable tener presente que si uno de los elementos de la diagonal principal es cero, los dos algoritmos fallan ya que el pivote se emplea como divisor; una posible solución para ésto es incorporar en el paso dos un filtro que considere esta posibilidad.

Ejemplo

$$x_1 + 3x_2 - 2x_3 = 7$$
$$4x_1 - x_2 + 3x_3 = 10$$
$$-5x_1 + 2x_2 + 3x_3 = 7$$

I) GAUSS-JORDAN SIN PIVOTEO

$$\begin{pmatrix} 1 & 3 & -2 & 7 \\ 4 & -1 & 3 & 10 \\ -5 & 2 & 3 & 7 \end{pmatrix} \quad R_3+5*R_3 \quad R_2-4*R_1 \quad \begin{pmatrix} 1 & 3 & -2 & 7 \\ 0 & -13 & 11 & -18 \\ 0 & 17 & -7 & 42 \end{pmatrix}$$

$$R_2/-13 \quad \begin{pmatrix} 1 & 3 & -2 & 7 \\ 0 & 1 & -0.846 & 1.384 \\ 0 & 17 & -7 & 42 \end{pmatrix} \quad R_1-3*R_2 \quad R_3-17*R_2 \quad \begin{pmatrix} 1 & 0 & 0.538 & 11.999 \\ 0 & 1 & -.846 & 1.384 \\ 0 & 0 & 7.385 & 18.461 \end{pmatrix}$$

$$R_3/7.385 \quad \begin{pmatrix} 1 & 0 & 0.538 & 2.846 \\ 0 & 1 & -0.846 & 1.384 \\ 0 & 0 & 1 & 2.499 \end{pmatrix} \quad R_1-.538*R_3 \quad R_2+.846*R_3 \quad \begin{pmatrix} 1 & 0 & 0 & 1.499 \\ 0 & 1 & 0 & 3.499 \\ 0 & 0 & 1 & 2.499 \end{pmatrix}$$

II) CON PIVOTEO PARCIAL

$$\begin{bmatrix} 1 & 3 & -2 & 7 \\ 4 & -1 & 3 & 10 \\ -5 & 2 & 3 & 7 \end{bmatrix} \quad R_1 <-> R_3 \begin{bmatrix} -5 & 2 & 3 & 7 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix}$$

$$R_1/-5 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 4 & -1 & 3 & 10 \\ 1 & 3 & -2 & 7 \end{bmatrix} \quad \begin{matrix} R_2-4*R_1 \\ R_3-1*R_1 \end{matrix} \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & .6 & 5.4 & 15.6 \\ 0 & 3.4 & -1.4 & 8.4 \end{bmatrix}$$

$$R_2<->R_3 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & 3.4 & -1.4 & 8.4 \\ 0 & 0.6 & 5.4 & 15.6 \end{bmatrix} \quad R_2/3.4 \begin{bmatrix} 1 & -.4 & -.6 & -1.4 \\ 0 & 1 & -.411 & 2.47 \\ 0 & .6 & 5.4 & 15.6 \end{bmatrix}$$

$$\begin{matrix} R_1+.4*R_2 \\ R_3-.6*R_2 \end{matrix} \begin{bmatrix} 1 & 0 & -.764 & -.411 \\ 0 & 1 & -.411 & 2.470 \\ 0 & 0 & 5.647 & 14.117 \end{bmatrix} \quad R_3/5.647 \begin{bmatrix} 1 & 0 & -.764 & -.411 \\ 0 & 1 & -.411 & 2.470 \\ 0 & 0 & 1 & 2.500 \end{bmatrix}$$

$$R_1+.764*R_3 \begin{bmatrix} 1 & 0 & 0 & 1.500 \\ 0 & 1 & 0 & 3.500 \\ 0 & 0 & 1 & 2.500 \end{bmatrix}$$

Observe nuevamente el decremento de error cuando se emplea pivoteo parcial.

8. METODOS ITERATIVOS DE SOLUCION DE ECUACIONES LINEALES.

Los métodos iterativos tienen ciertas ventajas sobre los métodos directos discutidos, se recomienda usarlos cuando muchos de los coeficientes de la matriz son cero; en general su implementación es muy facil.

METODO DE JACOBI

El primer paso es resolver el sistema para una variable en cada ecuación (se recomienda la variable cuyo coeficiente sea el más grande).

Por ejemplo, el conjunto de ecuaciones:

$$5x_1 + x_2 + 3x_3 = 10$$

$$x_1 + x_2 + 5x_3 = 8$$

$$2x_1 + 4x_2 + x_3 = 11$$

puede ser transformada en

$$x_1 = 2.00 - 0.20x_2 - 0.60x_3 \quad (1)$$

$$x_2 = 2.75 - 0.50x_1 - 0.25x_3 \quad (2)$$

$$x_3 = 1.60 - 0.20x_1 - 0.20x_2 \quad (3)$$

Se inicia el proceso dando valores iniciales a las variables (pueden ser cero, sino se conocen mejores valores), éstas aproximaciónes se colocan a la derecha de las ecuaciones, y se resuelven para obtener una mejor estimación de las variables; entonces se repite el procedimiento, siempre empleando la solución más reciente para obtener una mejor aproximación. Cuando dos conjuntos de estimaciones son muy cercanas el procedimiento debe parar y el último conjunto se toma como solución; otro criterio de parada podría ser cuando la suma de los cuadrados de las diferencias es menor que la tolerancia prescrita; esto es, si X^i es la **i-ésima** estimación de la variable Xj, y existen n variables a ser encontradas, entonces $X^i_1, X^i_2, \ldots, X^i_n$ puede ser tomada como solución si $\sum_{j=1}^{n} (X^i_j - X^{i-1}_j)^2 < E$ donde E es la tolerancia especificada.

ALGORITMO DE JACOBI

Dada una **n** (número de variables), un arreglo de valores iniciales y $f_1, f_2, f_3, \ldots fn$ las ecuaciones a ser usadas para $x_1, x_2, \ldots xn$ respectivamente; este algoritmo resuelve el sistema de ecuaciones:

$$x_1 = f_1(x_2, x_3, \ldots, xn)$$

$$x_2 = f_2(x_1, x_3, \ldots, xn)$$

$$\ldots$$

$$xn = fn(x_1, x_2, \ldots, xn-_1)$$

por el **método de JACOBI**, las aproximaciones se almacenan en el vector **nuevax** y el proceso para cuando la suma de los cuadrados de las diferencias es menor que E.

```
BEGIN
     FOR  i:=1  TO  30  DO
     BEGIN
          {Calcular el nuevo conjunto de aproximaciones}
          FOR  j:=1  TO  n  DO
             nuevax[j]:= fj(x[1],x[2],...,x[j-1],x[j+1],...x[n]);
             {Prueba de ocurrencia}
             sum:=0;
             FOR  j:=1  TO  n  DO
                sum := sum + (nuevax[j] - x[j])**2;
             IF  sum < E  THEN
             BEGIN
                  WRITE(Solución:);
                  FOR  j:=1  TO  n  DO
                       WRITE(x[j]);
             END;
            {Emplear el nuevo conjunto de valores en la
             siguiente aproximación}
             FOR  j:=1  TO  n  DO
                  x[j] := nuevax[j] ;
     END;
     WRITE('tolerancia  no  encontrada  en  30  iteraciones');
END;
```

Si aplicamos este método a los ejemplos previos obtenemos la siguiente secuencia de valores en 21 iteraciones:

x_1	x_2	x_3
0.00	0.00	0.00
2.00	2.70	1.60
0.49	1.35	0.65
1.34	2.3425	1.232
0.7923	1.7720	0.8635
1.1275	2.1379	1.0871
0.9201	1.9144	0.9469
1.0490	2.0532	1.0331
0.9695	1.9672	0.9796
1.0188	2.0203	1.0126
0.9883	1.9874	0.9921
1.0072	2.0078	1.0048
0.9955	1.9952	0.9970

1.0027	2.0030	1.0018
0.9983	1.9982	0.9988
1.0010	2.0011	1.0007
0.9993	1.9993	0.9995
1.0004	2.0004	1.0002
0.9998	1.9997	0.9998
1.0001	2.0001	1.0001
0.9999	1.9999	0.9999
1.0000	2.0000	1.0000

Como para $i > 1$, $x_1^{(k)}, x_2^{(k)}, \ldots x_{i-1}^{(k)}$ ya han sido calculados y supuestamente es una mejor aproximación a la solución real $x_1, x_2, \ldots, x_{i-1}$ que $x_1^{(k-1)}, x_2^{(k-1)}, \ldots, x_{i-1}^{(k-1)}$ resulta más razonable calcular $x_i^{(k+1)}$ usando los valores más recientes, a esta mejora del método se le conoce con el nombre de GAUSS-SEIDEL.

METODO DE GAUSS SEIDEL.

Dado n, x el arreglo que contiene la última aproximación, f1,f2,...fn y E como en el algoritmo anterior, este proceso aplica el **método de GAUSS SEIDEL** para calcular la solución de un sistema de ecuaciones (Observe que en este método ya no es necesario el arreglo **nuevax**).

```
BEGIN
    FOR  i:=1  TO  30  DO
    BEGIN
         SUM := 0;
         FOR  j:=1  TO  n  DO
         BEGIN
              valor_anterior := xcjj;
              xcjj:=fj(xc1),xc2),...,xcj-1),
                     xcj+1),...,xcn));
              sum:=sum+(xcjj- valor_anterior)**2;
         END;
         Prueba de ocurrencia
         IF sum  < E  THEN
         BEGIN
              FOR k:=1  TO  n  DO
                  WRITE (xci));
              EXIT.
         END;
    END;
    WRITE('EL METODO NO CONVERGIO');
END;
```

Si se aplica este método al siguiente sistema de ecuaciones:

$$X_1 = -1.4000 + 0.4000X_2 + 0.6000X_3$$

$$X_2 = 2.3333 - 0.3333X_1 + 0.6666X_3$$

$$X_3 = 3.3333 - 1.3333X_1 + 0.3333X_2$$

Se obtienen los siguientes resultado

X_1	X_2	X_3
0	0	0
2.0	1.75	0.85
1.14	1.9675	0.9785
1.0194	1.9956	0.9970
1.0026	1.9994	0.9996
1.0003	1.9999	0.9999
1.0000	2.0000	1.0000

COMPARACION DE LOS METODOS.

Cuando se comparan las técnicas para resolver sistemas lineales, es necesario considerar otras cuestiones además de la cantidad de memoria requerida. Uno de los principales aspectos es el efecto del error de redondeo y otro es la cantidad de tiempo requerido para realizar las operaciones aritméticas básicas; como podemos observar varios aspectos dependen del número de operaciones que el método requiera, en general, el tiempo requerido para realizar una multiplicación o una división en una computadora es aproximadamente el mismo y es considerablemente mayor que el requerido para realizar una suma o una resta. Las diferencias reales en el tiempo de ejecución dependen del sistema de cómputo particular que se esta empleando.

Los métodos más empleados para resolver sistemas lineales son el método de Eliminación Gaussiana con o sin pivoteo y el método de Gauss-Seidel.

Los ejemplos que se ilustraron en este capítulo son sólo una muestra de la gran variedad de algoritmos matemáticos que existen.

CAPITULO IV

ALGORITMOS DE ORDENACION Y BUSQUEDA

En el mundo de la informática, no hay quizá, tareas tan importantes, ni tan extensamente analizadas como la ordenación y la búsqueda; estas dos operaciones son fundamentales en la computación, y se emplean virtualmente en todos los programas de bases de datos, compiladores, intérpretes, y sistemas operativos.

La **ORDENACION**, se refiere a la operación de organizar datos con un criterio determinado, y la **BUSQUEDA** al proceso de localizar un dato. El aprender los diversos métodos que existen para resolver estas operaciones es de gran utilidad en el manejo de la información, y es necesario sobre todo para aquellas personas dedicadas al manejo de grandes volúmenes de datos que tengan que ser consultados de manera eficiente y rápida.

Entre los métodos de ordenación y búsqueda existe una relación estrecha, por lo que la primer pregunta que resulta en cualquier aplicación que requiera consulta de datos es, si vale la pena realizar un ordenamiento antes de proceder a buscar la información requerida. Algunas veces es mejor trabajar en buscar algún elemento en particular, sin importar el orden de los datos, pero en otras ocasiones y sobre todo, si es frecuente el uso del archivo para la búsqueda de algunos elementos específicos, puede ser más eficiente el ordenar primero el archivo. Precisamente por estas razones no se puede decir exactamente si es mejor el ordenar o no un archivo antes de efectuar una búsqueda; ello dependerá del problema en cuestión.

Los programadores deben tomar conciencia de la importancia de las técnicas de ordenación y búsqueda que existen, evitando tomar la vía más fácil y codificar algoritmos que emplean métodos poco eficientes para sistemas, en los cuales, estas dos operaciones son componentes claves.

4.1 ALGORITMOS DE ORDENACION.

En la mayoría de los centros de cómputo, el problema de la ordenación ocupa un lugar predominante, ya que un porcentaje considerable de los procesos que se realizan tienen como función ordenar diversos tipos de conjuntos.

Para hablar de la eficiencia de un algoritmo de ordenación, se deben considerar los medios con los que se cuenta para su aplicación y el esfuerzo o costo necesario para realizar las diversas tareas de comparación e intercambio.

En el presente subcapitulo se tiene la finalidad de presentar algunos de los algoritmos de ordenación más conocidos, y hacer un análisis general de sus diferentes propiedades y características, que permita obtener elementos de juicio para poder escoger el método apropiado para cada problema particular de ordenación.

4.1.1 Definición de ordenación.

Ordenar una estructura de datos significa establecer un orden de precedencia entre los elementos de la estructura especificamente, dada una lista i ordenada en forma creciente de n elementos, $i_1 <= i_2 <= \ldots <= i_n$. El campo de información empleado para ordenar los elementos comúnmente recibe el nombre de llave.

Un conjunto de elementos se ordena generalmente para simplificar la recuperación de la información manualmente (como en un directorio telefónico o la clasificación de libros), o para facilitar el acceso por medio de una máquina a los datos en una forma más eficiente (por cuestiones de consulta o actualización), y aún, cuando esto parece ser una operación trivial, en realidad es un proceso costoso que debe realizarse sólo cuando sea necesario y con el método apropiado.

4.1.2 Consideraciones para seleccionar un método de ordenamiento.

Desafortunadamente un algoritmo de ordenación no es el mejor para cualquier situación, debido a esto, existen diversos métodos que realizan este proceso y por ello el programador debe estar conciente de las diferentes consideraciones de eficiencia, para escoger el método apropiado de ordenamiento para el problema en particular que está analizando. Algunos de los puntos que se deben tomar en cuenta son:

-Tiempo requerido por el programador para codificar un algoritmo:

Si se tiene un archivo pequeño, las técnicas de ordenamiento sofisticadas para minimizar espacio y tiempo son generalmente menos eficientes que los métodos simples. Igualmente, si el programa de ordenación se va a ejecutar una sola vez, y existe suficiente capacidad de espacio y tiempo de máquina, sería ilógico para el programador, gastar recursos investigando el mejor método para obtener un poco más de eficiencia.

-Tipo de memoria en la cual se localizan los datos:

Los datos se pueden localizar en:

a) Memoria de acceso directo y alta velocidad.

b) Memoria de acceso directo y mediana velocidad.

c) Memoria de acceso secuencial.

-La eficiencia del método (Complejidad):

Cuando se ordena un conjunto de datos es necesario tener presente, que cualquier mejora en cuanto a la eficiencia de un método representa un gran ahorro computacional, por lo que al considerar un archivo de tamaño n y el tiempo requerido para ordenarlo, nos interesa conocer el comportamiento del método al modificar la cantidad de datos. Esto se puede llevar a cabo de dos maneras:

1) Realizar un análisis matemático cuyo resultado sea una función del tamaño del archivo (generalmente es el tiempo promedio requerido para algún ordenamiento en particular).

2) Correr en tiempo real el programa para medir su eficiencia con respecto a una cantidad determinada de datos.

Sin embargo es muy difícil decidir que algoritmo es el mejor para una situación dada, ya que existen, otra gran variedad de factores que influyan en la efectividad de un algoritmo de ordenación:

-Cantidad, tipo y distribución inicial de los datos .

-Características del sistema operativo (con respecto al manejo de archivos y de la memoria).

-Tiempo de acceso a los dispositivos de almacenamiento secundario o auxiliar y,

-Cantidad de espacio requerido por el algoritmo.

este tipo de restricción hoy en día es menos importante que las consideraciones de tiempo, ya que si se requiere más espacio, éste se puede encontrar utilizando almacenamiento secundario; pero hay que tener presente que este hecho influye notoriamente en el tiempo de ejecución.

4.1.3 Análisis de los algoritmos de ordenación.

Un ordenamiento a menudo consta de tres partes:

a) Una comparación que determina el orden de cualquier par de elementos,

b) Un intercambio que invierte el orden, y

c) Un algoritmo de ordenación que realiza comparaciones e intercambios hasta que los objetos estén ordenados.

En el área de la computación se han desarrollado una gran cantidad de algoritmos cuya complejidad estima el tiempo de ejecución como una función del número de elementos a ser ordenados, y las principales operaciones empleadas al realizar este proceso son intercambios, comparaciones e iteraciones; sin embargo, la función de complejidad normalmente computa sólo el número de comparaciones, ya que los intercambios e iteraciones son en la mayoría de los casos un factor constante del número de comparaciones.

Ahora bien, los criterios para juzgar un algoritmo de ordenación se basan en las siguientes preguntas

¿ Qué rapidez tiene el algoritmo en el caso medio ?

¿ Qué rapidez tiene en el mejor y peor caso ?

¿ Qué tipo de comportamiento exhibe el método?

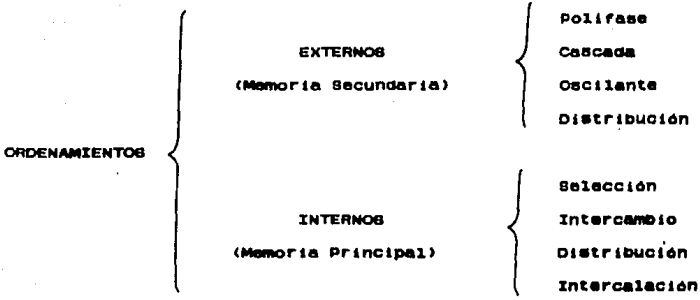
El tiempo requerido para un ordenamiento depende como ya se ha mencionado de la cantidad de datos, pero también puede depender de la secuencia original de éstos. Para algunos métodos si los datos de entrada están casi ordenados, su proceso se puede completar en un tiempo $O(n)$, mientras que si los datos de entrada están en orden casi inverso, se requiere un tiempo $O(n^2)$. También existen procesos cuyo tiempo requerido es $O(n\log_2 n)$ independientemente del orden inicial de los datos. El medir el tiempo de ejecución de un algoritmo con respecto a una secuencia de datos es importante sólo si se espera encontrar situaciones semejantes frecuentemente.

Un método de ordenación exhibe un comportamiento natural, si su trabajo se encuentra en función a la secuencia inicial de los datos "**a mayor desorden, mayor trabajo**"; en caso contrario se dice que el método tiene un comportamiento antinatural.

Por último y a manera de recomendación, cuando se selecciona un método de ordenación se debe intentar tener cierta visibilidad futura, porque aún cuando las ordenaciones sencillas, necesitan relativamente poco tiempo de programación para escribirlas y mantenerlas, conforme se le añaden características para mejorarlas, también se le está incrementando la complejidad a ese algoritmo, ya que no sólo se está incrementando el tiempo requerido por las rutinas, sino también el tiempo empleado por el programador para mantenerlo, lo cual implica un mayor costo.

4.1.4. Clasificación de los métodos de ordenamiento.

Los métodos de ordenamiento se clasifican por el tipo de memoria en la cual se encuentran almacenados los datos a ser ordenados:



ORDENAMIENTOS EXTERNOS.

Cuando se presenta el problema en el cual el conjunto de elementos por ordenar, emplea mayor memoria a la capacidad de almacenamiento de la memoria primaria, se debe recurrir a un método de ordenamiento externo, ya que los datos se encuentran en dispositivos de almacenamiento secundario.

El esquema general de los métodos de ordenamiento externo está compuesto de dos fases:

FASE I : Construcción de arreglos con datos ordenados.

FASE II : Intercalación de los arreglos de la fase I, hasta sólo tener un arreglo totalmente ordenado.

ORDENAMIENTOS INTERNOS.

Los métodos de ordenamiento que se utilizan para un conjunto de datos almacenados en una memoria de acceso directo de alta velocidad (memoria principal) son llamados métodos de ordenamiento interno. De acuerdo al principio en que se basan al realizar su proceso, este tipo de métodos se clasifican en métodos de **selección, intercambio, distribución e intercalación.**

A. METODOS POR SELECCION.

En este tipo de método se selecciona del conjunto de datos, el mayor o menor elemento (según el criterio) y se excluye; posteriormente, se procede de igual forma sobre los n-1 elementos restantes.

1) SELECCION DIRECTA.

El proceso consiste en localizar el elemento menor de la lista e intercambiarlo, con el elemento que se encuentra en la primer posición; Después localiza el segundo elemento menor y lo intercambia con el que se encuentra en la segunda casilla y así sucesivamente.

Ejemplo

ITERACION:	LISTA:							
0	10	9	7	8	6	5	4	8
1	4	9	7	8	6	5	10	8
2	4	5	7	8	6	9	10	8
3	4	5	6	8	7	9	10	8
4	4	5	6	7	8	9	10	8
5	4	5	6	7	8	9	10	8
6	4	5	6	7	8	8	10	9
7	4	5	6	7	8	8	9	10

CODIFICACION

La codificación de este método es sencilla, su empleo no se recomienda para archivos muy grandes, ya que no presenta ventaja alguna en cuanto a que el archivo de entrada esté completamente ordenado; debido a que el procedimiento se realiza n-1 veces, independientemente del estado original del archivo.

```
    PROCEDURE Seleccion-Directa (a: Arreglo; n: INTEGER);
    i,j,k : INTEGER;
    BEGIN
1)      FOR i:= 1 TO n-1 DO
        BEGIN
2)          k:=i;
3)          FOR j:= i+1 TO n DO
4)              IF a[j] < a[k] THEN k:= j;
6)          intercambia(a[j], a[k]);
        END;
    END;
```

COMPLEJIDAD DEL METODO

El ciclo **FOR** interno se ejecuta en un tiempo **O(n-i)**, puesto que j va desde i+1 hasta n, así que el tiempo que requiere el algoritmo es $C\sum_{i=1}^{n-1}(n-i)$ para alguna constante C cualquiera. De esta suma que es igual $Cn(n-1)/2$ se observa que el tiempo total de ejecución es $O(n^2)$.

COMPARACIONES.

El número de comparaciones de este método es independiente del orden inicial de los datos y está dado por el número de veces que se ejecuta la línea cuatro:

$$\sum_{i=1}^{n-1}(1) = (n-1) + (n-2) + \ldots + 2 + 1$$

$$= n(n-1)/2$$

lo cual hace que sea un algoritmo demasiado lento para un gran número de elementos.

INTERCAMBIOS.

La instrucción intercambia (6) se ejecuta exactamente n-1 veces por lo que la velocidad de crecimiento con respecto al número de intercambios en este método es **O(n)**.

En este método cada intercambio requiere tres movimientos

```
PROCEDURE intercambia (VAR x,y : elemento);
BEGIN
     aux  := x ;
     x    := y ;
     y    := aux;
END;
```

y el número total de ellos puede ser minimizado si sólo se invoca a intercambia cuando el elemento esté realmente fuera de la posición que le corresponda, ya que no tiene caso intercambiar dos elementos que se encuentran en la posición correcta.

3) SELECCION REPETITIVA (SELECCION CUADRATICA).

El método divide el conjunto de n datos en m grupos de m datos, donde m es la raíz cuadrada de n. De cada grupo se selecciona el dato menor y se coloca en un arreglo temporal (T) de longitud m, de este arreglo se selecciona el menor; este dato se elimina del arreglo T y del grupo de donde provino, y se sustituye por un valor muy grande . Posteriormente se selecciona el menor de este grupo, se lleva al arreglo temporal T y se continúa el procedimiento.

Ejemplo

LISTA ORIGINAL : 5 9 8 7 6 5 7 8 9

n = 9 m = $\sqrt{n}$ => m = 3

por lo que se realizan tres grupos con tres elementos cada uno de ellos:

GRUPOS	ARREGLO TEMPORAL (T)	ELEMENTO MENOR
(5 9 8) (7 6 5) (7 8 9)	(5 5 7)	5
(* 9 8) (7 6 5) (7 8 9)	(8 5 7)	5
(* 9 8) (7 6 *) (7 8 9)	(8 6 7)	6
(* 9 8) (7 * *) (7 8 9)	(8 7 7)	7
(* 9 8) (* * *) (7 8 9)	(8 * 7)	7
(* 9 8) (* * *) (* 8 9)	(8 * 8)	8
(* 9 *) (* * *) (* 8 9)	(9 * 8)	8
(* 9 *) (* * *) (* * 9)	(9 * 9)	9
(* * *) (* * *) (* * 9)	(* * 9)	9

Donde el simbolo * representa un valor muy grande

CODIFICACION.

```
PROCEDURE Selección_cuadratica(a: arreglo; n: INTEGER);
  PROCEDURE Busca_menor (VAR t,a: vector; indice,
            lim_inf,lim-sup: INTEGER; VAR aux: INTEGER);
  VAR
     i : INTEGER;
  BEGIN
       menor   := a[lim_inf];
       aux     := lim_inf;
       FOR  i := lim_inf + 1 to lim_sup DO
         IF  a[i]<menor THEN
         BEGIN
              aux   := i;
              menor := a[i];
         END;
       a[aux]     := infinito;
       t[indice] := menor;
  END;
BEGIN
     grupo := INT(SQRT(n));
     IF SQR(grupo) < n THEN
     BEGIN
```

```
        grupo := grupo + 1;
        FOR  i:= n+1 TO SQR(grupo) DO
            a[i] := infinito;
    END;
    FOR k := 0 TO grupo-1  DO
        busca_menor(t,a,k+1,k*grupo + 1,(k+1)*grupo,aux);
    FOR k := 1 TO n DO
    BEGIN
        busca_menor(t1,t,k,1,grupo,aux)
        busca_menor(t,a,aux,(aux-1)*grupo+1,aux*grupo,aux);
    END;
    WRITE('Lista Ordenada:',t1);
END.
```

COMPLEJIDAD DEL METODO.

Si la raíz de n es un número entero m el número de comparaciones es m-1 para cada grupo; en la primer pasada, el número de comparaciones totales es igual al número de grupos m+1 (se considera también el vector T) por el número de comparaciones para cada grupo: m-1; pero para las siguientes pasadas, sólo se examina el grupo de donde provino el menor y T, siendo el número de comparaciones igual a 2(m-1) por lo que el total de comparaciones realizadas son:

$$(m+1)(m-1)+(n-1)(2(m-1)) \text{ con } m=\sqrt{n} \qquad (1)$$

donde

$$(m+1)(m-1)=m^2-1 \qquad \text{como } m^2=n$$

$$=n-1$$

poniendo (1) en función de n

$$f(n) = n-1 + (n-1)(2\sqrt{n}-1)) \quad \text{comparaciones}$$

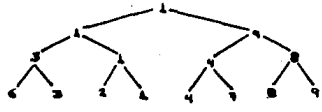
este tipo de métodos se puede generalizar a métodos de selección cúbica, etc.; pero se debe estar conciente que el número de iteraciones y comparaciones aumenta.

3) TORNEO O MONTICULO.

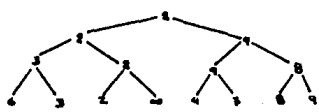
El conjunto de n datos se estructura como un árbol binario de m niveles, en el cual sus hojas son los datos originales. El nivel m-1 se tienen los menores de cada par; esto se repite para todos los niveles hasta tener en la raíz al menor de todos. En este momento debe substituirse el contenido de la hoja que posee al elemento menor por un valor muy grande y repetir el proceso n-1 veces.

Ejemplo

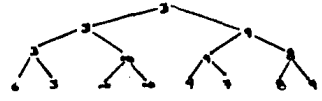
LISTA 6 3 2 1 4 7 8 9



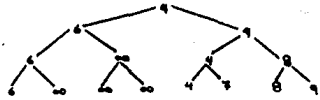
Lista: 1



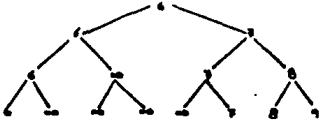
Lista: 1,2



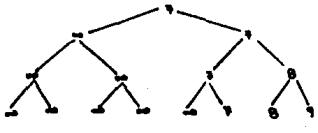
Lista: 1,2,3



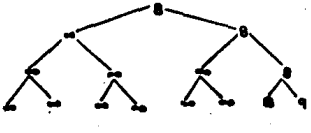
Lista: 1,2,3,4



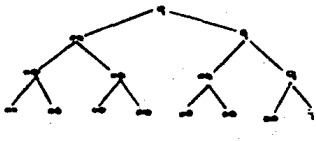
Lista: 1,2,3,4,6



Lista: 1,2,3,4,6,7



Lista: 1,2,3,4,6,7,8



Lista: 1,2,3,4,6,7,8,9

FIG 4.1 Torneo

COMPLEJIDAD DEL METODO.

El número de comparaciones para el método puede calcularse de la siguiente manera: en el nivel **m**, el número de comparaciones es **n/2**, en el **m-1** es **n/4**, y así sucesivamente hasta el nivel 2, donde se requiere de una comparación. De esta forma el número de comparaciones para la pasada uno es:

$$n/2 + n/4 + n/8 + \ldots + 4 + 2 + 1 = n-1$$

Para las siguientes pasadas, al obtener el número de comparaciones, sólo se consideran los nodos afectados por el subárbol donde se encuentra el menor por lo que se requieren tantas comparaciones como niveles tenga el árbol menos 1 ya que en la raíz no hay comparación.

Como el número de niveles de un árbol binario completo es de $\log_2 (n+1)$, el número de comparaciones en las **n-1** pasadas restantes es $(n-1)(\log_2 n)$. El total de comparaciones es entonces

$$(n-1) + (n-1)(\log_2 n)$$

lo cual en forma aproximada nos da un comportamiento proporcional a $O(n \log_2 n)$.

El espacio de memoria utilizado por este método es muy grande, ya que se requiere de una cantidad casi igual al número de datos si se desea un vector ordenado, el total de memoria adicional usada sería de **2n**.

Es muy difícil creer que este método de ordenación pueda ser eficiente, ya que es necesario mover el valor menor para varios sitios hasta llegar a la raíz, antes de ponerlo en su lugar final; y de hecho para cantidades de datos pequeñas no es método recomendable, lo cual no sucede para cantidades grandes ya que aún cuando cada elemento fuera una hoja del último nivel y tenga que pasar por todo el árbol, su tiempo de ejecución es $O(n \log_2 n)$, lo cual implica que a mayor cantidad de datos, mayor eficiencia.

4) HEAP SORT.

Este método se basa en el empleo de una estructura llamada **HEAP** y un recorrido denominado **BORRADO DEL HEAP**.

El **HEAP** es un árbol binario completo, al cual le pueden faltar algunas hojas situadas más a la derecha en el último nivel. En cualquier nodo el dato es mayor o menor (según el criterio) a los datos que se encuentran en sus subárboles.

El **BORRADO DEL HEAP** consiste en remover el dato de la raíz e intercambiarlo con otro nodo; este dato debe ser acomodado dentro del árbol manteniendo la estructura del HEAP con los nodos activos, se procede con los nodos nivel por nivel, de abajo hacia arriba y derecha a izquierda.

ALGORITMOS.

- CONSTRUCCION DEL HEAP.

```
FOR i := altura-1 DOWNTO 1 DO
BEGIN
    {PARA cada nodo p que no sea hoja en el nivel n}
       MIENTRAS p no sea hoja
       BEGIN
            m := hijo de p con mayor valor;
            IF valor(p) < valor(m) THEN
            BEGIN
                 INTERCAMBIAR nodos;
                 p := m;
            END
            ELSE
                 p := hoja;
       END;
END;
```

Ejemplo

LISTA 2 4 5 3 7

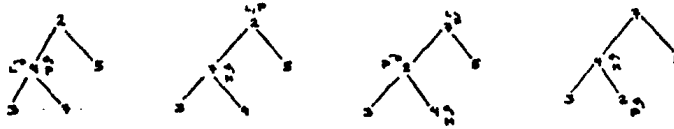


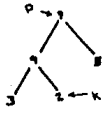
FIG 4.2 Construcción del HEAP.

- BORRADO DEL HEAP.

```
MIENTRAS el número de nodos del HEAP > 1
BEGIN
   k := hoja más a la derecha del último nivel;
   p := raiz;
   INTERCAMBIAR el nodo(k) con el nodo(p) y eliminar k;
   MIENTRAS p no sea hoja
   BEGIN
        m := apuntador al hijo mayor de p;
        IF valor(p) < valor(m)  THEN
        BEGIN
             INTERCAMBIALOS;
             p := m;
        END
        ELSE
```

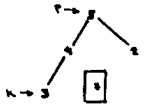
```
            p := hoja;
        END;
    END;
```

Ejemplo

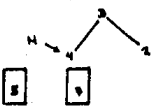


HEAP INICIAL

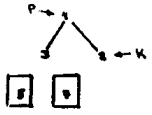
LISTA 7



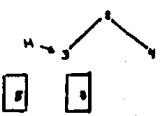
NUEVO HEAP



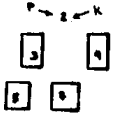
LISTA 7,5



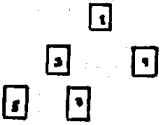
NUEVO HEAP



LISTA 7,5,4



NUEVO HEAP



LISTA 7,5,4,3,2

FIG 4.3 Borrado del HEAP

CODIFICACION

```
    PROCEDURE HEAP(a:arreglo; j,m:INTEGER);
    BEGIN
         WHILE (2*j+1 <= m) AND ((a[j]<a[2*j]) OR
               (a[j]<=a[2*j+1])) DO
         BEGIN
           IF a[2*j] > a[2*j+1] THEN
           BEGIN
                temp    := a[j];
                a[j]    := a[2*j];
                a[2*j]  := temp;
                j       := 2*j;
           END
           ELSE
           BEGIN
                temp    := a[j];
                a[j]    := a[2*j+1];
                a[2*j+1]:= temp;
                j       := 2*j+1;
           END;
           IF ((2*j) = m) AND (a[j] < a[2*j]THEN
           BEGIN
                temp    := a[j];
                a[j]    := a[2*j];
                a[2*j]:= temp;
           END;
         END;
  END;
  PROCEDURE HEAPSORT (a : arreglo; n : INTEGER);
  BEGIN
       i := n;
       m := n;
       WHILE   i >= 1 DO
       BEGIN
            HEAP(a,i,m);
            i := i-1;
       END;
       WHILE    m > 1  DO
       BEGIN
            temp := a[1];
            a[1] := a[m];
            a[m] := temp;
            m    := m-1;
            HEAP(a,1,m);
       END;
  END;
```

COMPLEJIDAD DEL METODO.

Analizar el peor caso es más facil que realizar el análisis del caso promedio; para ello se supondrá que se emplea un árbol binario completo.

Sea d=log n la altura del HEAP con n nodos. El número de comparaciones realizadas de la raíz a un nodo p es d-(nivel de p), entonces el número de comparaciones totales en la construcción del HEAP es al menos:

$$\sum_{i=0}^{d-1} 2(d-1) \text{ (nodos del nivel i)} = 2\sum_{i=0}^{d-1}(d-1)2^i$$

$$= 2d\sum_{i=0}^{d-1}2^i - 2\sum_{i=0}^{d-1}i2^i$$

$$= 2d\,(2^d-1) - 4\sum_{i=1}^{d-1}i2^{i-1}$$

$$= 2d(2^d-1) - 4(d2^{d-1} - 2^d + 1)$$

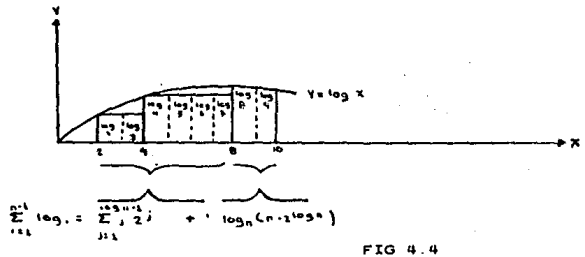
$$= 2^{d+2} - 2d - 4$$

En términos de n, el número de comparaciones realizadas en la construcción del HEAP es al menos Cn-2(1ogn) donde 2<=C<4.

El número de comparaciones realizadas al borrar un nodo del HEAP con i nodos requiere a lo mucho 2(log i), lo cual implica que el número total para todo el borrado en el peor caso es $2\sum_{i=1}^{n-1}(\log i)$.

Para poder evaluar la suma $\sum_{i=1}^{n-1}(\log i)$, se empleará el caso para n=10.

Sea la siguiente figura.



$$\sum_{i=1}^{n-1}(\log i) = \sum_{j=1}^{\log n-1} j2^j + (n-1-2^{\log n})\log n$$

FIG 4.4

En ella se puede observar que $\sum_{i=1}^{n-1}(\log i)$ es igual a la suma de las áreas de los rectángulos; esto es $\sum_{j=1}^{\log n-1} j2^j + \log n\,(n-2^{\log n})$ donde la suma del primer término incluye las áreas de todos los rectángulos completos (altura j y ancho 2^j), y el segundo término es el área del último rectángulo incompleto con altura log n y ancho $n-2^{\log n}$.

Por lo que el área total es

$$\sum_{i=1}^{n-1} \log i = 2((\log(n)2^{\log_2 n-1}-2^{\log_2 n}+1)+(\log(n))(n-2^{\log_2 n}))$$

$$= n(\log(n)-2^{\log_2 n+1}+2$$

Por lo que la construcción del HEAP requiere a lo mucho $2^{\log_2 n+2}-2(\log n)-4$ comparaciones y en su borrado a lo más $2(n(\log n)-2^{\log_2 n+1}+2)$; siendo el número total de comparaciones $2n(\log n)$ en el peor caso. El caso promedio es más complicado de analizar, pero se puede demostrar que su tiempo de ejecución es $O(n\log_2 n)$.

A pesar de su tiempo $O(n\log_2 n)$ en el peor caso, HEAP-SORT tiene interés intelectual por ser el primer algoritmo con ese tiempo de ejecución que se ha estudiado. Es de gran utilidad su práctica, cuando no se desean ordenar los n elementos, sino sólo los K mayores (menores) de ellos, con K mucho menor que n.

8. METODOS POR INTERCAMBIO.

Este tipo de métodos intercambian pares de datos que se encuentran fuera de orden hasta tener el conjunto ordenado.

1) BURBUJA.

Este método también se conoce con el nombre de intercambio por selección o propagación, el nombre de burbuja se debe a su forma de manipular los datos (semejante a burbujas subiendo a una superficie).

El proceso se inicia comparando los elementos n y n-1 e intercambiándolos sólo en el caso en el que elemento n sea menor al elemento n-1; después repite el proceso comparando a n-1 y n-2 hasta comparar los datos 1 y 2, logrando con ello tener al dato menor en la posición 1; se elimina esta posición y se vuelve a aplicar el método n-2 veces. Ejemplo:

Sea la lista a ordenar 10 12 8 1 6 19 4

primer iteración:

10	10	10	10	10	10	1
12	12	12	12	12	1	10
8	8	8	8	1	12	12
1	1	1	1	8	8	8
6	6	4	4	4	4	4
19	4	6	6	6	6	6
4	19	19	19	19	19	19

iteración:	0	1	2	3	4	5	6
	10	1					
	12	10	4				
	8	12	10	6			
	1	8	12	10	8		
	6	4	8	12	10	10	
	19	6	6	8	12	12	12
	4	19	19	19	19	19	19

CODIFICACION.

```
    PROCEDURE BURBUJA;
    BEGIN
1)      FOR  k := 1  TO  n-1  DO
2)         FOR  j := n   DOWNTO  k  DO
3)            IF a[j] < a[j-1]  THEN
              BEGIN       Intercambio
4)                aux     := a[j];
5)                a[j]    := a[j-1];
6)                a[j-1] := c;
              END;
    END;
```

COMPLEJIDAD DEL METODO.

La ordenación de burbuja en un intercambio lleva un tiempo constante, sea C_1 unidades de tiempo para alguna constante C_1; por tanto el ciclo externo (1) se ejecuta en $C_2(n-1)$ pasos para alguna constante C_2; esta última constante es algo mayor que C_1 para justificar los decrementos y pruebas de j. En consecuencia el programa completo requiere

$$C_3n + \sum_{i=1}^{n-1} C_2(n-i) = 1/2C_2n^2 + (C_3 - 1/2C_2)n \text{ pasos},$$

donde el término C_3n tiene en cuenta los incrementos y pruebas de i.

Factorizando el resultado:

$$1/2C_2n^2 + (C_3 - 1/2C_2)n = 1/2C_2n^2 + C_3n - 1/2C_2n$$

$$= 1/2C_2(n^2 + n) + C_3n$$

Como este resultado excede a $(C_3/2 + C_2)n^2$ para $n>=1$, se observa que la complejidad de tiempo de este método es de $O(n^2)$, por lo que también recibe el nombre de método de la **n-cuadrada**.

COMPARACIONES.

El número de comparaciones es el mismo para cualquier caso ya que los dos ciclos FOR se repiten el número de veces especificado, aunque la lista estuviese inicialmente ordenada.

$$\text{Comparaciones} = (n-1)(n/2) = 1/2(n^2-n)$$

ya que el ciclo externo se ejecuta n-1 veces y el interno n/2 veces.

INTERCAMBIOS.

Cuando la lista se encuentra ordenada, el método no realiza intercambio alguno, en el peor caso (cuando la lista se encuentra invertida) realiza tantos intercambios como comparaciones requiera es decir (n(n-1))/2.

En el caso promedio dado que existen n! permutaciones distintas para n datos, donde las llaves no ordenadas en el archivo L son $x_1, x_2, x_3, \ldots x_n$; existe una permutación alfa tal que para 1<=i<=n, alfa(i) es la posición correcta de xi, cuando el archivo está ordenado.

Sin perder la generalidad se puede asumir que las llaves son enteros 1,2,3,...n y se puede substituir 1 por la llave más pequeña o primera en orden, 2 por la segunda y así sucesivamente. Identifíquese entonces a la permutación alfa con el archivo alfa(1), alfa(2), alfa(3), ..., alfa(n), entonces el número promedio de comparaciones para un algoritmo de ordenación, se encuentra calculando cuantas comparaciones realiza el algoritmo para cada permutación.

Una inversión de una permutación alfa es un par (alfa(i),alfa(j)) tal que i<j y alfa(i)>alfa(j). Si (alfa(i),alfa(j)) es una inversión el i-ésimo y j-ésimo elemento en el archivo está fuera de un orden relativo con respecto al otro. Por ejemplo la permutación 2,4,1,5,3 tiene cuatro inversiones (2,1),(4,1),(4,3),(5,3).

Cuando se da una comparación en el método de la burbuja, puede suscitarse un intercambio de dos llaves, con ello exactamente una inversión será eliminada de la lista; sin embargo no todas las comparaciones implican un intercambio El número de comparaciones desarrolladas por el método de burbuja en la entrada alfa(1), alfa(2), ..., alfa(n) es al menos el número de inversiones de alfa, por lo que el número promedio de inversiones de las permutaciones en n elementos es aproximado al número promedio de comparaciones.

Para una permutación alfa, la secuencia B(alfa) = ($b_1, b_2, b_3, \ldots, b_n$) está definida por bi que es el número de elementos que se encuentran a su derecha, los cuales son menores a i. Para la permutación |2,4,1,5,3| $b_1=b_3=0$, $b_2=b=1$ y $b_4=3$; observa que bi es

el número de inversiones de alfa cuyo primer componente es i, por lo que el número total de inversiones es $\sum_{i=1}^{n} b_i$. También para $1<=i<=n$, $0<=b_i<=i-1$, si se da una secuencia $B=(b_1,b_2,\ldots,b_n)$ tal que $0<=b_i<=i-1$ para $1<=i<=n$; es posible encontrar una correspondencia entre tales secuencias y permutaciones, por lo que se puede concluir que para cada entero K no negativo, el número de permutaciones con exactamente K inversiones, es el número de secuencias $b_1,b_2,\ldots,b_n$ con $0<=b_i<=i-1$ tal que $\sum_{i=1}^{n} b_i = K$, se denota este número por S(K).

Dada una permutación con n componentes tiene al menos (n(n-1))/2 inversiones, entonces el número promedio de inversiones es:

$$1/n! \sum_{k=0}^{n(n-1)/2} K \cdot S(K)$$

Usando técnicas combinatorias, el valor de S(K) y el de la suma puede ser calculado, siendo el resultado (n(n-1))/4 Por lo tanto el número de intercambios promedio realizados por el método de burbuja es n(n-1)/4.

De todos los métodos, este es probablemente el menos eficiente, su única ventaja es que requiere poco espacio adicional (una posición de memoria para retener el valor temporal al realizar un intercambio); por lo que se recomienda sólo para un número pequeño de elementos ya que su tiempo de ejecución está directamente relacionado con el número de comparaciones e intercambios.

Como se puede observar, la clasificación de burbuja intercambia datos a lo sumo unas $n^2/2$ veces, pero dado que un intercambio se lleva a cabo sólo si dos elementos adyacentes se encuentran fuera de orden, cabe esperar que la cantidad real de intercambios sea mucho menor a $n^2/2$.

De hecho, este método en el caso promedio intercambia elementos exactamente la mitad de veces, así que el número de intercambios esperado, si todas las secuencias iniciales tienen igual probabilidad será aproximadamente $n^2/4$; para demostrar esto considerese dos listas iniciales de n elementos mutuamente inversas $L_1 = K_1,K_2,K_3,\ldots,Kn$ y $L_2 = Kn, Kn-1, \ldots K_1$. Un intercambio es la única forma en que Ki y Kj puedan cruzarse si están inicialmente fuera de orden, pero Ki y Kj pertenecen a sólo una de las listas L_1 y L_2; Así el número total de intercambios ejecutados cuando se aplica la clasificación de burbuja a L_1 y L_2 es igual al número de pares de elementos, esto es $\binom{n}{2}$ o $n(n-1)/2$; por lo que el número de intercambios promedio para L_1 y L_2 es $n(n-1)/4$ (aproximadamente $n^2/4$), ya que todas las ordenaciones posibles pueden compararse con su inversa, como sucedió con L_1 y L_2. De esto se concluye que el número promedio de intercambios en el método de burbuja es aproximadamente $n^2/4$.

2) MEJORAS AL METODO DE BURBUJA.

El análisis de complejidad de estos métodos no se incluye dado que es muy similar al realizado en el método de burbuja y en el peor caso exhiben prácticamente el mismo comportamiento.

- Una mejora inicial al método de burbuja, es parar el proceso cuando no existan intercambios en una determinada iteración; con esto se logra minimizar el número de comparaciones e intercambios realizados.

CODIFICACION.

```
PROCEDURE BURBUJA;
BEGIN
     K := 0;
     sigue := TRUE;
     WHILE  sigue  DO
     BEGIN
          K     := K+1;
          sigue := FALSE;
          FOR  j:=n DOWNTO K DO
            IF a[j] < a[j-1] THEN
            BEGIN
                 sigue    := TRUE;
                 aux      := a[j];
                 a[j]     := a[j-1];
                 a[j-1]   := aux;
            END;
     END;
END;
```

Ejemplo

ITERACION	0	1	2
	3	2	2
	4	3	3
	5	4	4
	2	5	5
	6	6	6
	7	7	7

Como en la segunda iteración no existió intercambio el método para.

-**SHAKE O EMBUDO** : Inicialmente denominado **COCKTAIL SHAKE**; emplea el método de la burbuja, alternado las pasadas, es decir, en la primer iteración se coloca al elemento menor, en la segunda

se coloca al mayor, logrando con esto minimizar el número de comparaciones.

Ejemplo

ITERACION	0	1	2	3	4	5	6	7	8
		↓	↑	↓	↑	↓	↑	↓	↑
	6	1							
	5	6	5	2					
	9	5	6	5	5				
	7	9	7	6	6	3			
	10	7	9	7	7	6	6	4	
	1	10	2	9	3	7	4	6	6
	3	2	3	3	4	4	7	7	7
	2	3	4	4	8	8	8		
	4	4	8	8	9				
	8	8	10						

CODIFICACION.

```
PROCEDURE SHAKE;
BEGIN
      K      := 0;
      sigue  := TRUE;
      WHILE sigue DO
      BEGIN
            K := K+1;
            sigue := FALSE;
            FOR j:= n-K+1 DOWNTO  K DO
              IF a[j] < a[j-1] THEN
              BEGIN
                    INTERCAMBIA (a[j],a[j-1]);
                    sigue := TRUE;
              END;
            FOR J:= K+1 TO n-K+1 DO
              IF a[j] > a[j+1] THEN
              BEGIN
                    INTERCAMBIA (a[j],a[j+1]);
                    sigue := TRUE;
              END;
      END;
END;
```

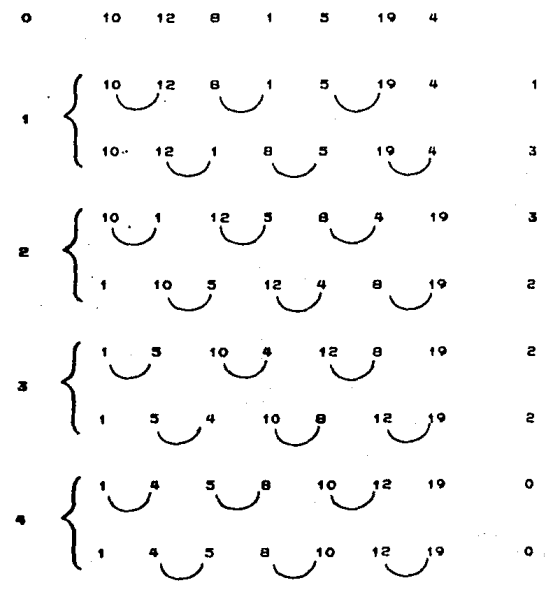
-TRANSPOSICION DE PARES Y NONES : Este método consta de dos etapas:

Primera: Compara datos que ocupan localidades nones con la localidad siguiente y los intercambia si es necesario.

Segunda: Compara datos que se encuentran en localidades pares, con los datos que se encuentran en la localidad siguiente, intercambiándolos si están en desorden.

Este proceso continúa hasta que en ambas etapas no se realice ningún intercambio.

Ejemplo



CODIFICACION (VERSION UNO).

```
    PROCEDURE PARES_NONES;
    BEGIN
          sigue := TRUE;
          aux   := n DIV 2;
          WHILE  sigue  DO
          BEGIN
                sigue := FALSE;
                i     := 1;
                FOR   j:=1  TO  aux  DO
                BEGIN
                     IF a[i] > a[i+1] THEN
                     BEGIN
                          sigue := TRUE;
                          INTERCAMBIA (a[i],a[i+1]);
                     END;
                     i := i+2;
                END;
                i := 2;
                FOR j:=1  TO AUX DO
                BEGIN
                     IF a[i] > a[i+1] THEN
                     BEGIN
                          sigue := TRUE;
                          INTERCAMBIA (a[i], a[i+1]);
                     END;
          END;
    END;
```

CODIFICACION (VERSION DOS).

```
    PROCEDURE ORDENA (VAR i: INTEGER; VAR cambio : BOOLEAN);
    BEGIN
          c := 0;
          REPEAT
                IF a[i] > a[i+1] THEN
                BEGIN
                     k       := a[i];
                     a[i]    := a[i+1];
                     a[i+1] := k;
                     c       := c+1;
                END;
                IF  c = 0  THEN
                   cambio :=  FALSE
                ELSE
                   cambio := TRUE;
                i := i+2;
          UNTIL i >= n;
     END;
     BEGIN
          REPEAT
                i := 1;
                ORDENA(i,BAN);
```

```
            i := 2;
            ORDENA(i,BAN);
        UNTIL BAN = FALSE;
    END.
```

3) QUICK SORT.

El método **QUICKSORT** fue desarrollado por C.A.R. Hoare; es el proceso más eficiente que se conoce para ordenar un conjunto de datos, ya que emplea particiones e intercambios; sin embargo, tiene la absurda propiedad de que trabaja mejor entre más desordenado esté el archivo.

El método consiste esencialmente en seleccionar un elemento llamado pivote del conjunto de datos, y reacomodar éste de tal forma que a la derecha de él se encuentren los elementos menores y a su izquierda los mayores. El proceso se repite hasta que las particiones generadas tengan un sólo elemento.

Los pasos detallados para aplicar este método son:

a) Seleccionar al pivote o comparando (x) del conjunto, lo cual se puede realizar de dos formas:

   - Escogerlo aleatoriamente.

   - Seleccionar el elemento central.

b) Revisar el conjunto desde el extremo izquierdo hacia el elemento pivote hasta encontrar un elemento >= x. Sea ai donde i es la posición.

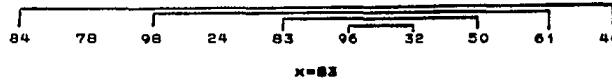
c) Revisar el conjunto desde el extremo derecho hacia el extremo pivote, hasta encontrar un elemento <= x, sea éste aj donde j es su posición.

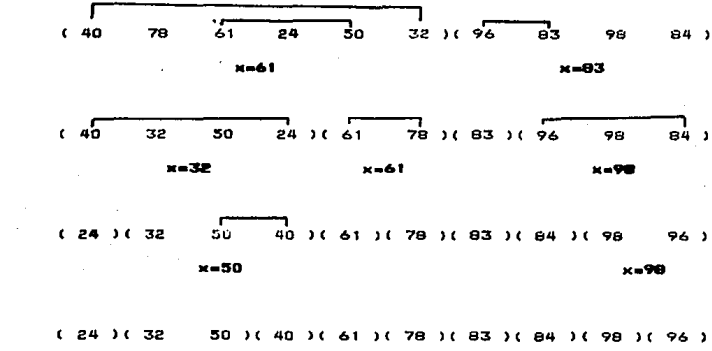
d) Intercambiar los elementos ai y aj para continuar con las búsquedas e intercambiarlos del lado izquierdo y derecho, hasta que i y j se crucen en alguna parte del conjunto.

e) Guardar los índices de los extremos del subconjunto con mayor longitud.

f) Analizar el subconjunto menor (regresar al paso a) hasta que no existan subtablas por ordenar.

Ejemplo





(24)(32)(50)(40)(61)(78)(83)(84)(98)(96)

CODIFICACION.

```
PROCEDURE QSORT (datos : vector; n: INTEGER);
   PROCEDURE QUICKSORT(datos: vector;n: INTEGER);
   VAR
        i,j,x : INTEGER;
   BEGIN
        i := lim_inf;
        j := lim_sup;
        x := (i+j) DIV 2;
       |posición  central|
        x := datos[x];
       |dato   central|
        REPEAT
             |encuentra un elemento > x|
              WHILE  datos[i] < x DO
                    i := i+1;
             |encuentra un elemento < x|
              WHILE  x < datos[j]  DO
                    j := j-1;
              IF i <= j THEN
              BEGIN
                   IF datos[i] <> datos[j] THEN
                   BEGIN
                        INTERCAMBIA (datos[i],datos[j]);
                        i := i+1;
                        j := j-1;
```

```
                END;
            END;
        UNTIL  i > j;
        IF lim_inf < j THEN QUICKSORT(lim_inf, j);
        IF lim_sup > i THEN QUICKSORT(i, lim_sup);
    END;
BEGIN
    QSORT(1,n);
END.
```

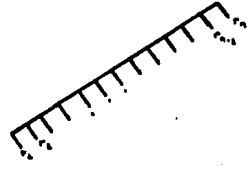
COMPLEJIDAD DEL METODO.

El método de ordenación rápida o **QUICKSORT** en el caso promedio consume un tiempo **O(nlog n)** y en el peor caso **O(n^2)** para ordenar **n** elementos. El particionar un conjunto (elementos menores al pivote a la izquierda y elementos mayores o iguales a la derecha) tiene un tiempo de ejecución proporcional al número de elementos que deberá separar es decir **O(j-i+1)**.

Con excepción de las llamadas recursivas que hace **QUICKSORT**, cada llamada individual se ejecuta en un tiempo máximo proporcional al número de elementos que se piden ordenar; en otras palabras el tiempo total consumido por **QUICKSORT** es la suma en todos los elementos, de las veces que el elemento forma parte del subconjunto con el cual se hizo la llamada a **QUICKSORT**. Es evidente que ningún elemento puede incluirse en dos llamadas al mismo nivel, así que el tiempo consumido por este método puede expresarse como la suma de la profundidad o máximo nivel de cada uno de los elementos.

ANALISIS DEL PEOR CASO.

El peor caso se presenta cuando en cada llamada a **QUICKSORT** se selecciona el peor pivote posible, por ejemplo el valor mayor del subconjunto que se está ordenando, ya que el conjunto se dividirá en dos subconjuntos, uno con un sólo elemento (el pivote) y el otro con los elementos restantes. Esta secuencia de particiones forma el siguiente árbol:



donde $r_1, r_2, r_3, \ldots, r_n$ es la secuencia de registros en el orden creciente de los elementos.

La profundidad de r_i es n-i+1 para $2<=i<=n$ y la profundidad de r_1 es n-1; así la suma de las profundidades es

$$n+1 + \sum_{i=2}^{n} (n-i+1) = n^2/2 + n/2 + 1$$

Por lo que el tiempo de ejecución en el peor caso es $O(n^2)$.

ANALISIS DEL CASO PROMEDIO.

Suponiendo que al llamar a **QUICKSORT(i,j)** todas las permutaciones a [i] ...,a [j] son igualmente probables (la justificación es que antes de la llamada no había pivote con los cuales a [i] ...,a [j] se pudieran comparar para distinguirlos entre sí, es decir, para que todos fueran menores que el pivote V, o para que todos fueran mayores que él).

Ahora T(n) es el tiempo promedio consumido por este método para ordenar n elementos. Es evidente que T(1) es alguna constante C_1, ya que para un elemento este método no realiza llamadas recursivas directas. Cuando n>1, como se supone que todos los elementos son distintos, este método tomará un pivote y dividirá el conjunto consumiendo un tiempo C_2n para alguna constante C_2, y después llamará al método para ordenar los dos subconjuntos; sería bueno poder pedir que el pivote tuviera la misma probabilidad de ser el primero, segundo o n-ésimo elemento del conjunto que está ordenando, sin embargo, para garantizar que el método encuentre por lo menos un elemento menor que cada pivote y al menos uno igual o mayor que éste, siempre se escoge el mayor de los dos primeros encontrados.

Resulta que esta selección no afecta a la distribución de tamaños de los subconjuntos, pero tiende a hacer los grupos izquierdos más grandes que los grupos derechos.

Se hará el desarrollo de la fórmula para la probabilidad de que el grupo izquierdo tenga i de los n elementos, en el supuesto de que todos los datos son distintos. Para que el grupo izquierdo tenga i elementos, el pivote debe ser el (i+1)-ésimo elemento. El pivote por el método de selección podría haber estado en la primer posición, con alguno de los i menores como primero (la probabilidad de que cualquier elemento en particular tal como el (i+1)-ésimo aparezca primero en una secuencia aleatoria es 1/n). Dado que apareció primero, la probabilidad de que el segundo elemento sea uno de los i elementos más pequeños de los n-1 elementos restantes es i/n(n-1); así la probabilidad de que el pivote aparezca en la primer posición y sea el número i+1 de los n en el orden apropiado es i/n(n-1). En forma semejante la probabilidad de que el pivote sea el de la segunda posición y el número i+1 de los n en el orden es i/n(n-1); así que la probabilidad de que el grupo izquierdo sea de tamaño i es Σi/n(n-1) para $1<=i<n$.

Ahora se puede escribir una ecuación de recurrencia para T(n):

$$T(n) <= \sum_{i=1}^{n-1} 2i/n(n-1) \cdot [T(i) + T(n-i)] + C_2 n \qquad (A)$$

según esta ecuación el tiempo promedio requerido por el método **QUICKSORT** es $C_2 n$ independientemente de las llamadas recursivas, más el tiempo promedio utilizado en las mismas. Este último tiempo se expresa como la suma de todas las posibles i, de la probabilidad de que el grupo izquierdo tenga tamaño i (y por lo tanto el grupo derecho tamaño n-i) multiplicado por el costo de las llamadas recursivas T(i) y T(n-i) respectivamente.

Simplificando la sumatoria para que tome la forma que tendría si se hubiera elegido un pivote verdaderamente aleatorio, se tendrá que realizar una transformación, obsérvese que para una función f(i) cualquiera, substituyendo i por n-i se puede probar

$$\sum_{i=1}^{n-1} f(i) = \sum_{i=1}^{n-1} f(n-i) \qquad (B)$$

multiplicando ambas por f(i) y despejando se tiene

$$\sum_{i=1}^{n-1} f(i) = 1/2 \sum_{i=1}^{n-1} (f(i) + f(n-i)) \qquad (C)$$

Aplicando (C) a (A) con f(i) igual a la expresión interna de la sumatoria de (A) se [illegible]

$$T(n) <= 1/2 \sum_{i=1}^{n-1} [\, 2i/n(n-1) \cdot T(i) + T(n-i) \,] +$$

$$2(n-i)/n(n-1) \cdot T(n-i) + T(i) \,] \,] + C_2 n$$

$$<= 1/(n-1) \sum_{i=1}^{n-1} \cdot T(i) + T(n-i) \,] + C_2 n \qquad (D)$$

Se aplica (C) a (D) con f(i) = T(i)

$$T(n) <= 2/(n-1) \sum_{i=1}^{n-1} T(i) + C_2(n) \qquad (E)$$

La solución propuesta es que $T(n) <= Cn\log n$ para alguna constante C y toda n>=2.

Para demostrar que esta suposición es correcta se efectúa una inducción sobre n. Si n=2, sólo se observa que para alguna constante C, $T(2) <= 2C\log 2 = 2C$. Para probar la inducción, se supone que $T(i) <= Ci\log i$ para i<n y se sustituye esta fórmula por T(i) en el lado derecho de (E) con lo cual se puede demostrar que la cantidad resultante no es mayor que $Cn\log n$. Así (E) es

$$T(n) <= 2C/(n-1) \sum_{i=1}^{n-1} i\log i + C_2 \qquad (F)$$

Como es necesario dividir la sumatoria anterior en términos menores, donde i<=n/2 y por lo tanto, logi no es mayor que log(n/2) que es (logn)-1, y en términos mayores, donde i>n/2 y logi puede ser tan grande como logn, F se transforma en:

$$T(n) <= 2C/(n-1) \cdot [\, \sum_{i=1}^{n/2} i\log i + \sum_{i=n/2+1}^{n-1} i\log i \,] + C_2 n$$

$$<= 2C/(n-1) \cdot [\, \sum_{i=1}^{n/2} i\log(n-1) + \sum_{i=n/2+1}^{n-1} i\log n \,] + C_2 n$$

$$<= 2C/(n-1)\,[\,n/4(n/2+1)\log n - n/4(n/2+1) +$$

$$3/4n(n/2+1)\log n\,] + C_2 n$$

$$<= 2C(n-1)\,[\,(n^2/2 - n/2)\log n - (n^2/8 + n/4)\,] + C_2 n$$

$$<= Cn\log n - Cn/4 + Cn/2(n-1) + C_2 n \qquad (6)$$

Al tomar $C>=4C_2$ la suma del segundo y cuarto termino de (6) no es mayor que cero, y como el tercer término hace una contribución negativa, $T(n)<=Cn\log n$ si $C=4C_2$. Esto completa la demostración de que la ordenación rapida requiere un tiempo O(nlogn) en el caso promedio.

UNA FORMA MAS SENCILLA DE ANALIZAR QUICKSORT.

El analisis del tiempo de ejecución del **QUICKSORT** esta dado por

$$T(n) = P(n) + T(J-LB) + T(UB-J)$$

donde **P(n)**, **T(J-LB)** y **T(UB-J)** representan el tiempo empleado para particionar el conjunto de datos, ordenar el subconjunto izquierdo y ordenar el subconjunto derecho respectivamente. El tiempo empleado para particionar un conjunto es **O(n)**.

El peor caso ocurre cuando en cada llamada al procedimiento, al particionar el conjunto en dos subconjuntos, uno de ellos está vacio (esto es, **J=LB** o **J=UB**); situación que se presenta cuando el elemento del conjunto seleccionado se encuentra ya ordenado.

El analisis del peor caso asumiendo **J=LB** es:

$$T(n) = P(n) + T(0) + T(n-1)$$

$$= Cn + T(n-1)$$

$$= Cn + C(n-1) + T(n-2)$$

$$= Cn + C(n-1) + C(n-2) + T(n-3)$$

$$\cdots$$

$$= \sum_{k=1}^{n} Ck + T(0)$$

$$= C((n+1)n/2)$$

$$= O(n^2).$$

El mejor caso ocurre cuando el conjunto es particionado en dos subconjuntos del mismo tamaño, esto es **J = ((LB+UB)/2)** entonces

$$
\begin{aligned}
T(n) &= P(n) + 2T(n/2) \\
&= C*n + 2T(n/2) \\
&= C*n + 2C(n/2) + 4T(n/4) \\
&= C*n + 2C(n/2) + 4C(n/4) + 8T(n/8) \\
&= 3C*n + 8T(n/8) \\
&\ \ \ldots \\
&= \log_2 nC*n + 2**\log nT(1) \\
&= O(n\log_2 n)
\end{aligned}
$$

MEJORAS AL QUICKSORT.

El método **QUICKSORT** es el proceso de ordenación más rápido que se conoce en la actualidad, ya que su tiempo promedio de ejecución es **O(nlogn)**; sin embargo, es posible mejorar aún más el factor constante al tomar pivotes que dividan cada subconjunto en partes similares.

Ejemplo

Se pueden tomar K elementos al azar de un conjunto, ordenarlos por una llamada recursiva a **QUICKSORT**, o por un algoritmo simple y elegir al elemento medio, es decir, el elemento (K+1)/2 como pivote; pero hay que tener presente que si K es muy pequeño se malgasta el tiempo porque en promedio, el pivote dividirá los elementos en forma desigual; y si K es muy grande tardará demasiado tiempo, el encontrar el elemento medio de los k-elementos. Así, cabe esperar un incremento en la velocidad del método de ordenación rápida si y sólo si se seleccionan los pivotes con cuidado.

Otra posible mejora es aplicar un método sencillo de ordenamiento cuando se tienen subconjuntos con un número reducido de datos, ya que los métodos simples $O(n^2)$ son mejores que los procesos **O(nlogn)** para una n pequeña. El tamaño adecuado de n depende de muchos factores tales como el tiempo empleado en una llamada recursiva -lo cual es una propiedad de la arquitectura de la máquina- y la estrategia utilizada por el compilador del lenguaje en el que se escribió el método de ordenación para realizar la llamada a procedimientos (KNUTH sugiere el nueve como el tamaño del subconjunto en que **QUICKSORT** debe llamar a un algoritmo de ordenación más simple).

Por último hay que tener presente que el método **QUICKSORT** recursivo es elegante pero no práctico ya que emplea demasiada memoria local.

C. METODOS POR INSERCION.

Este tipo de métodos suponen que el conjunto de datos se encuentra ordenado y se desea introducir un elemento nuevo, para ello se determina el lugar que debe ocupar y se desplazan los datos una posición de tal forma que el nuevo dato pueda ser colocado en el lugar correcto.

1) INSERCION DIRECTA.

EL método supone que el conjunto inicial sólo tiene un elemento y a partir de él se van ordenando los demás.

Ejemplo

ORDENAR:	5	1	7	9	1	4
Lista inicial:	5					
	1	5				
	1	5	7			
	1	5	7	9		
	1	1	5	7	9	
	1	1	4	5	7	9

CODIFICACION.

```
    PROCEDURE INSERCION;
    BEGIN
1)      a[0] := menos_infinito;
2)      FOR i:= 2 TO n DO
        BEGIN
3)          j := i;
4)          WHILE a[j] < a[j-1] DO
            BEGIN
5)              INTERCAMBIA(a[j],a[j-1]);
                j := j-1;
            END;
        END;
    END;
```

COMPLEJIDAD DEL METODO.

El ciclo WHILE no puede llevar más de O(i) pasos ya que j toma el valor inicial de i en la instrucción 3 y decrece en cada ciclo, éste termina cuando j=1, ya que a[0] es menos_infinito lo que provoca que la condición 4 sea falsa. Dado que el ciclo FOR se ejecuta n-1 veces, el método de inserción directa consume hasta $C\sum_{i=2}^{n} i$ pasos para alguna constante C. Siendo el valor de esta suma $n(n+1)/2 - 1$, lo cual implica un tiempo de ejecución $O(n^2)$.

COMPARACIONES.

El ciclo WHILE se ejecuta $\sum_{j=2}^{n}(1 + dj)$ veces; donde dj es el número de elementos mayores a a[j] que se encuentran a su izquierda y $(d_1,d_2,d_3, \ldots,dn)$ es la inversión del vector a[1], a[2],...,a[n], por lo que número total de comparaciones es n-1 + dj.

Si 0<=dj<=j-1 y las d_j's son independientes entonces el máximo y mínimo valor para $n-1 + \sum_{j=2}^{n} dj$ es 1/2(n-1)(n-2) y n-1.

El promedio de $\sum_{j=2}^{n} dj$ sobre todas las permutaciones es $1/2\sum_{j=2}^{n} j = 1/4n(n-1)$ por lo que el número de comparaciones para el caso promedio está dado por:

$$n-1 + 1/2\sum_{j=2}^{n} dj = n-1 + 1/4n(n-1)$$

$$= (4(n-1) + n(n-1))/4$$

$$= ((n-1)(n+4))/4$$

INTERCAMBIOS.

En este algoritmo cada intercambio implica tres movimientos (aux:=a[j], a[j]:=a[j+1], a[j+1]:=aux), por lo que el número total de ellos está dado por las comparaciones realizadas en cada caso multiplicadas por la constante 3.

Una leve mejora a este algoritmo, con el objeto de simplificar el número de intercambios es la siguiente:

```
   PROCEDURE INSERCION;
   BEGIN
1)      a[0] := menos_infinito;
2)      FOR i:=2  TO  n  DO
        BEGIN
3)          j:=i-1;
4)          x:=a[i];
5)          WHILE  x<a[j]  DO
            BEGIN
6)                a[j+1]:=a[j];
7)                j     :=j-1;
            END;
8)          IF  i<>j  THEN
9)                a[j]:=x;
        END;
   END;
```

Dado que el ciclo WHILE se ejecuta $\sum_{j=2}^{n} dj$ veces, el número de movimientos de los datos es (n-1) + (n-1) + $\sum_{j=2}^{n} dj$ esto es 2n-2 movimientos en el mejor caso 1/2(n-1)(n+4) en el peor caso y 1/4(n-1)(n+8) en el caso promedio:

Mejor caso = (n-1)(n-1)

= 2n + 2

Peor caso = (n-1) + 1/2(n-1)(n+2)

= (2(n-1) + (n-1)(n+2))/2

= (n-1)(n+4)/2

Caso promedio = (n-1) + $\underbrace{(n+1) + \Sigma_{j=2}^{n} dj}_{\text{comparaciones}}$

= (n-1) + ((n-1)(n+4))/4

= (4(n-1) + (n-1)(n+4))/4

= (n-1)(n+4+4)/4

= 1/4(n-1)(n+8)

También es posible al activar la instrucción 8 del algoritmo minimizar más el número de intercambios, pero también implica un gran incremento en el número de comparaciones (n+1) en cada caso.

2) **INSERCION BINARIA.**

El proceso de inserción binaria consiste en comparar inicialmente el primer dato con el elemento central del conjunto y determinar si las comparaciones continúan sobre los datos que se encuentran a la izquierda o derecha del elemento central.

Ejemplo

Lista:

503	87	512	61	170	897	275
87	512	61	170	503	897	275
512	61	87	170	503	897	275
61	87	170	503	512	897	275
61	87	170	275	503	512	897

COMPLEJIDAD DEL METODO.

El método de inserción binaria, resuelve sólo la mitad del problema, ya que después de que se determina donde debe ser insertado un elemento, se deben mover cerca de 1/2j registros que han sido ordenados para tener libre el lugar en el que se instalará a[j]; por lo tanto el tiempo de ejecución es:

$$1/2 \sum_{i=1}^{n} i = 1/2(1 + 2 + \ldots + n)$$

$$= 1/2(n(n+1))/2$$

$$= 1/4n(n+1)$$

o sea proporcional a $O(n^2)$.

INSERCION DE DOBLE ENTRADA.

Este método fue propuesto con el fin de reducir el número de intercambios, y consiste en colocar el primer elemento en el centro del área de salida, y posteriormente a los demas a su derecha o izquierda según corresponda. Ejemplo

ORDENAR:	503	87	512	61	70	897	275
Salida :							
503				503			
87			87	503			
512			87	503	512		
61		61	87	503	512		
70	61	70	87	503	512		
897	61	70	87	503	512	897	
275	61	70	87	503	512	897	275

CODIFICACION.

```
PROCEDURE ORDENA;
PROCEDURE INSERTA_BINARIO(VAR sup,inf,cen,elem : INTEGER);
    BEGIN
        IF elem > a[n] THEN
          a[n+1]:=elem
        ELSE
          BEGIN
              sup:=n;
              inf:=1;
```

```
                cen:=(sup + inf) DIV 2;
                REPEAT
                      IF a[cen] < elem THEN
                      BEGIN
                           inf:=cen;
                           cen:=(sup + inf) DIV 2;
                      END
                      ELSE
                        BEGIN
                           sup:=cen;
                           cen:=(inf + sup) DIV 2;
                        END;
                UNTIL    cen - inf = 0;
                IF a[cen] >= elem THEN
                   FOR i:=n DOWNTO cen DO
                        a[i+1] := a[i];
                   a [cen+1] := elem;
           END;
END;
BEGIN
     FOR i:=1  TO  n  DO
          INSERTA_BINARIO(sup,inf,cen,elem[i]);
END;
```

3) SHELL.

Este método es una generalización del ordenamiento de burbuja y debe su nombre a su inventor D.L. SHELL; el proceso consiste en comparar todas las parejas de elementos que se encuentran a una distancia razonable, e intercambiar aquellas que no se encuentran en orden.

Las distancias de comparación no deben ser múltiplos enteros para evitar comparar más de una vez los mismos elementos, y deben ser decrementadas en cada iteración, hasta que se vuelva unitaria. El método parará, cuando la distancia sea unitaria y no existan intercambios.

Una selección razonable de la distancia es:

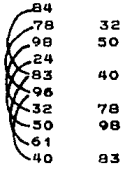
$$di = (di-_1 + 1)/2 \qquad i = 1,2,3, \ldots n \quad y \quad d_0 = n$$

Ejemplo

Lista inicial: 84 78 98 24 83 96 32 50 61 40

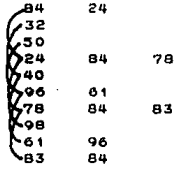
$d_1 = (10+1)/2$

$= 5$



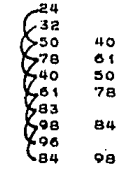
$d_2 = (5+1)/2$

$= 3$



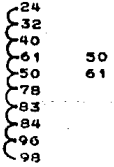
$d_3 = (3+1)/2$

$= 2$



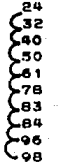
$d_4 = (2+1)/2$

$= 1$



$d_5 = (1+1)/2$

$= 1$



CODIFICACION.

```
PROCEDURE SHELL;
VAR
     distancia : INTEGER;
     bandera   : BOOLEAN;
 BEGIN
     bandera := FALSE;
     distancia := 0;
     WHILE NOT bandera DO
     BEGIN
         distancia := TRUNC((distancia+1)/2);
```

```
            IF distancia = 1 THEN
               bandera := TRUE;
            FOR i:=n DOWNTO  2  DO
                IF (i-distancia) >= 1  THEN
                BEGIN
                     IF datos[i] < datos[i-distancia] THEN
                     BEGIN
                        INTERCAMBIA(datos[i],datos[i-distancia]);
                         IF bandera = TRUE  THEN
                            bandera := FALSE;
                     END;
                END;
        END;
    END;
```

COMPLEJIDAD DEL METODO.

El número de comparaciones realizadas por el proceso **SHELL** es una función de la secuencia de incrementos usados. Su análisis es extremadamente difícil y requiere respuestas a algunos problemas matemáticos que aún no han sido resueltos.

La mejor secuencia de incrementos no se ha determinado aún, pero se han estudiado algunos casos específicos, como por ejemplo cuando el número de incrementos es igual a dos (t=2), empleando como distancias a h y 1; demostrando que el mejor valor para h es aproximadamente $1.72(n)^{1/3}$, con cuyo valor el tiempo promedio de ejecución es proporcional a $n^{5/3}$.

Si el número de incrementos es mayor a dos (t>2) la secuencia de valores está definida como $hk = 2^k - 1$ para $1<=k<=\log n$; siendo el número de comparaciones realizadas en el peor caso $O(n^{3/4})$; sin embargo estudios empíricos (con valores de n tan altos como 250,000), han demostrado que existen otros conjuntos de incrementos que aumentan la rapidez del programa, definiendo $hi=(3^i-1)/2$ para $1<=i<=t$, donde t es el entero más pequeño tal que $ht+2 >= n$ (Estos incrementos han sido calculados en forma iterativa).

Al inicio del ordenamiento se puede calcular ht empleando la relación $hs+1 = 3hs+1$ y comparando los resultados con n; en cada iteración hs puede ser recalculada empleando $hs = (hs+1-1)/3$. Se ha determinado la forma 2^i3^j < n (empleando un orden decreciente) entonces el número de comparaciones realizadas es $O(n(\log n)^2)$.

SHELL es un proceso eficiente ya que en cada iteración del método se involucra relativamente a pocos elementos, o a elementos que se encuentran a una distancia razonable, logrando incrementar considerablemente el orden de los datos en cada iteración.

D. METODOS POR DISTRIBUCION.

Los métodos por distribución constan de una colección de casilleros, marcados o etiquetados, donde se depositan los elementos que coinciden con su marca.

1) METODO RADIX SORT.

Cada iteración del método RADIX SORT; conocido también con el nombre de BUCKET SORT o CUBETAS; consta de dos fases. una fase de DISTRIBUCION en la cual los datos son colocados en las cubetas o urnas de acuerdo a un determinado criterio y, una fase de RECOLECCION donde los elementos contenidos en las cubetas son colocados en una lista con un determinado orden.

Ejemplo

Ordenar el siguiente conjunto de datos:

135	209	107	106	124	221	103	208	204

Fase I (distribución de acuerdo a la unidad)

CUBO		
0		
1	221	
2		
3	103	
4	124	204
5	135	
6	106	
7	107	
8	208	
9	209	

Fase II (recolección)

221	103	124	204	135	106	107	208	209

Fase I (distribución de acuerdo a la decena)

CUBO

0	103	204	106	107	208	209
1						
2	221	124				
3	135					
4						
5						
6						
7						
8						
9						

Fase II (recolección)

103 204 106 107 208 209 221 124 135

Fase I (distribución de acuerdo a la centena)

CUBO

0					
1	103	106	107	124	135
2	204	208	209	221	3
4					
5					
6					
7					
8					
9					

Fase II (recolección)

103 106 107 124 135 204 208 209 221

CODIFICACION DEL METODO EMPLEADO EN EL EJEMPLO.

```
PROCEDURE CUBETAS;
BEGIN
      a1 := 1;
      a2 := 10;
      FOR k1:=1  TO  num_digitos  DO
      BEGIN
           k := 1;
           FOR j:=0  TO  9  DO
                FOR  i:=1  TO  n  DO
                     IF (datos[i] MOD  a2)  DIV  a1  THEN
```

```
                                    BEGIN
                                      recoleccion[k] := datos[i];

                                k := k + 1;
                              END;
                    FOR i:=1  TO  n  DO
                        IF datos[i] <> recoleccion[i] THEN
                           datos[i] := recolecion[i];
                    a1 := a1 * 10;
                    a2 := a2 * 10;
            END;
    END;
```

COMPLEJIDAD DEL METODO.

La cantidad de trabajo realizado en la primer fase es proporcional al número de elementos (**n**), suponiendo que en cada fase las cubetas deben ser ordenadas empleando para ello un algoritmo que ordena por comparación de datos, entonces, se realizan **s(m)** comparaciones para una cubeta con **m** elementos.

Ahora si **ni** es el número de elementos que contiene la **i-ésima** cubeta, el algoritmo realiza $\sum_{i=1}^{k} s(ni)$ comparaciones durante la primer fase. La segunda fase puede requerir que todos los elementos sean copiados de una cubeta a un archivo, siendo la cantidad de trabajo **O(n)**. Lo cual implica que se realiza más trabajo mientras se ordenan las cubetas. Suponiendo que **s(m)** es **O(mlogm)**, entonces si los elementos son uniformemente distribuidos en las cubetas, el algoritmo realiza aproximadamente **Ck(n/k)log(n/k) = Cnlog(n/k)** comparaciones de elementos en la primer fase, donde **C** depende del algoritmo de ordenación empleado. Al incrementar **k** (número de cubetas) se decrementa el número de comparaciones realizadas; Si **k=n/10** entonces **nlog10** comparaciones podrían ser realizadas y el tiempo de ejecución de la ordenación de cubetas podría ser lineal, asumiendo que los datos son uniformemente distribuidos y el tiempo de la primer fase no depende de **k**.

Sin embargo, en el peor caso, todos los elementos pueden ser introducidos en una cubeta y el archivo entero puede ser ordenado en la segunda fase, haciendo todo el trabajo de la primer fase totalmente innecesario.

Si en la segunda fase no se empleó ningún algoritmo de ordenación como se muestra en el ejemplo entonces, el número de comparaciones realizadas por el método es cero y dado que existen n elementos a ordenar y **m** valores distintos de llaves (por lo tanto **m** cubetas distintas), el método consume **O(n+m)** si la estructura de datos empleada para las cubetas es la adecuada (se recomienda una lista enlazada).

En particular si **m<=n** la ordenación por cubetas se ejecuta en un tiempo **O(n)**.

De todo esto hay que tener presente que el método RADIX SORT es eficiente para un número no muy pequeño de elementos con una llave de longitud no muy larga, y aún, cuando varios métodos de ordenación se basan esencialmente en la comparación de llaves la contabilidad de esta operación no es el único camino para medir la eficiencia de este tipo de procesos.

E. METODOS POR INTERCALACION.

Los métodos por intercalación mezclan dos o más conjuntos de datos ordenados para formar un nuevo conjunto ordenado de datos.

1) MEZCLA.

El mínimo número de conjuntos ordenados requeridos para aplicar este método son dos; realizando para ello el siguiente proceso:

a) Comparar los dos primeros elementos del conjunto A y B.

b) Mover el elemento menor a un conjunto llamado C.

c) Tomar el dato siguiente al menor elegido y compararlo con el perdedor.

d) Repetir b y c hasta que uno de los conjuntos se agote.

e) Introducir los datos faltantes del conjunto no agotado a C

ejemplo

A : 3 5 7 8 9

B : 1 2 4 6

Comparación	C
3 > 1	1
3 > 2	1 2
3 < 4	1 2 3
4 < 5	1 2 3 4
5 < 6	1 2 3 4 5
6 < 7	1 2 3 4 5 6

como ya se agotó B se copian los elementos restantes de A a C.

1 2 3 4 5 6 7

CODIFICACION.

```
PROCEDURE INTERCALACION;
 BEGIN
      i:=1;
      j:=1;
      K:=1;
      WHILE (i<=n) AND (j<=M) DO
      BEGIN
           IF a[ i]<b[ j] THEN
           BEGIN
                c[kj:=a[ ij;
                i   :=i+1;
           END
           ELSE
           BEGIN
                c[kj:=b[ jj;
                j   :=j+1;
           END;
           K:=K+1,
      END;
      IF i<n THEN
        FOR 1:=j  TO  m  DO
        BEGIN
             K   :=K+1;
             c[kj:=b[ 1];
        END
      ELSE
        FOR 1:=i  TO  n  DO
        BEGIN
             K:=K+1;
             c[kj:=a[ 1];
        END;
 END;
```

COMPLEJIDAD DEL METODO.

Si n y m es el número de elementos de los conjuntos ordenados A y B, al aplicarles el método de mezcla para intercalarlos y obtener un conjunto ordenado, es fácil comprobar que el número máximo de comparaciones realizadas por el proceso en el peor caso es m+n-1, por lo tanto un algoritmo que mezcla dos conjuntos de tamaño n realiza a lo más 2n-1 comparaciones.

Los datos ai y bi pueden ser comparados para i<=i<=n y ai con b_{i+1} para i<=i<=n-1; si dos elementos ai y bj nunca son comparados, esto implica que si i<j, ai<bj y si i>=j, ai>bj. Si los elementos se comportan como $b_1<a_1<b_2<a_2<\ldots<b_n<a_n$ satisfacen esas condiciones pero el algoritmo puede ser capaz de no terminar en un orden correcto, similarmente si para alguna j nunca se compara aj con b_{j+1}; los datos se comportan como $b_1<a_1<b_2<a_2<\ldots<b_n<a_n$ donde el resultado puede ser consistente con los resultados de las comparaciones, pero nuevamente el proceso puede no terminar en el orden esperado.

Como se puede observar, cuando n es semejante a m el método de mezcla es bastante bueno,sin embargo cuando el tamaño de uno de los dos conjuntos es pequeño (cercano a uno); se puede buscar una forma más eficiente (empleando búsqueda binaria por ejemplo) para determinar el lugar en que esos elementos deben ser insertados

4.1.5 CONSIDERACIONES ADICIONALES RESPECTO A LOS METODOS DE ORDENAMIENTO.

Algunas de las consideraciones que un usuario debe tener presente para maximizar la eficiencia de un algoritmo de ordenación son:

- Cuando los registros sean muy grandes y los intercambios costosos, una estrategia muy útil para minimizar el trabajo y el costo del proceso, es mantener una matriz de apuntadores a registros por medio de cualquier algoritmo, para modificar las direcciones en lugar de intercambiar los registros.

- Algunos algoritmos intercambian a c[i] y a c[j] con i=j, o sea intercambian valores que se encuentran en la misma posición; los usuarios se han de preguntar el porqué de ello dado que no tiene caso realizar esta operación, pero hay que tener presente que el activar una instrucción condicional que considere esto, implica incrementar el número de comparaciones realizadas y sólo sería válido en el caso en el cual los intercambios sean una operación costosa.

- Si la rapidez, es determinante en la tarea de ordenar un conjunto de datos, se pueden evitar las llamadas a procedimientos y funciones, ya que estos requieren un tiempo extra; similarmente, dado que el procedimiento de ordenación rápida recursiva necesita un tiempo extra para la implementación de las llamadas recursivas, puede evitarse este tiempo codificando el procedimiento en forma no recursiva.

4.1.6 TABLA COMPARATIVA DE ALGUNOS METODOS DE ORDENACION.

ALGORITMO	COMPARACIONES		MEMORIA
	CASO PROMEDIO	PEOR CASO	ADICIONAL
INSERCION DIRECTA	(n-1)(n-2)/2	(n-1)(n+4)/4	NO
SELECCION DIRECTA	n(n-1)/2	n(n-1)/2	NO
HEAP SORT	n(logn)	2n(logn)	NO
BURBUJA	n(n-1)/2	n(n-1)/2	NO
QUICK SORT	n(logn)	n(n-1)/2	SI
RADIX SORT	m * n	m * n	SI

4.2 ALGORITMOS DE BUSQUEDA.

Los algoritmos de búsqueda aplicados a una estructura de datos permiten localizar un nodo en particular si es que éste existe, la eficiencia de este tipo de procesos se basa principalmente en el número de comparaciones que el método realiza al ejecutar su tarea.

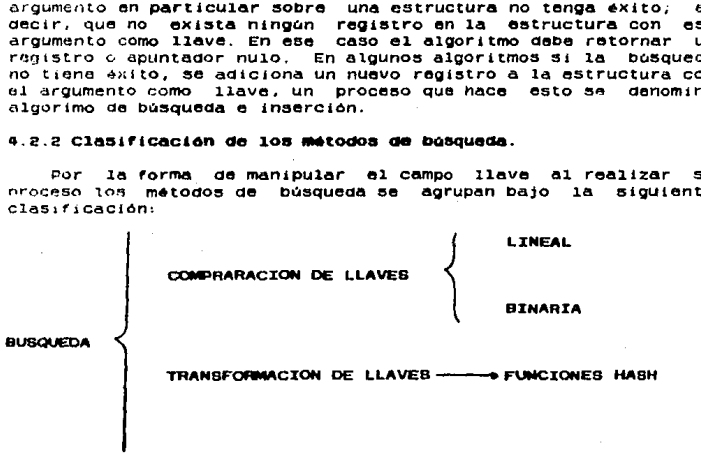
La búsqueda es una operación importante, ya que es el proceso que consume más tiempo en un programa, la elección o substitución de un buen método por un proceso malo repercute en un incremento substancial en la rapidez del programa; por ello es importante conocer las características de los diferentes métodos que existen para realizar esta actividad y, obtener criterios que nos permitan seleccionar el método más apropiado para un problema en particular.

4.2.1 Definición de los algoritmos de búsqueda.

Un algoritmo de búsqueda, es un proceso que acepta un argumento a y trata de encontrar un registro cuya llave sea a. El algoritmo puede retornar el registro completo o simplemente un apuntador a ese registro. Es posible que la búsqueda de algún argumento en particular sobre una estructura no tenga éxito; es decir, que no exista ningún registro en la estructura con ese argumento como llave. En ese caso el algoritmo debe retornar un registro o apuntador nulo. En algunos algoritmos si la búsqueda no tiene éxito, se adiciona un nuevo registro a la estructura con el argumento como llave, un proceso que hace esto se denomina algorimo de búsqueda e inserción.

4.2.2 Clasificación de los métodos de búsqueda.

Por la forma de manipular el campo llave al realizar su proceso los métodos de búsqueda se agrupan bajo la siguiente clasificación:



La elección de cualquiera de estos métodos depende del tiempo empleado en la ejecución de su proceso, y de la forma en la cual se encuentran almacenados los datos (ordenados, desordenados o almacenados con un criterio determinado).

A. METODOS DE BUSQUEDA POR COMPARACION DE LLAVES.

La característica común de los métodos agrupados bajo este nombre, es la de realizar la comparación directa de la llave buscada con las llaves de los nodos que pertenecen a la estructura de datos.

1) Búsqueda lineal.

La búsqueda lineal es el método más sencillo que se conoce, consiste en el recorrido secuencial desde el inicio hasta el final de los elementos de una estructura de datos.

El método es aplicable a estructuras de datos ordenadas y sin ordenar y se recomienda emplearlo cuando éstas son pequeñas y se encuentran almacenadas en memoria principal, ya que si no cumple estas características es necesario considerar en su procesamiento el tiempo de acceso a los dispositivos de almacenamiento secundario y el tiempo de recorrido de la estructura.

Ejemplo

Sea A una estructura de datos que contiene los siguiente elementos:

A =	6	7	3	15	0	4	23	22	11	10	9	17

a) Para saber si existe el número 23 se requieren de 7 comparaciones.

b) Para determinar si un elemento no pertenece a la estructura son necesarias 12 comparaciones.

c) El número promedio de comparaciones para buscar un elemento es 6 (n/2).

CODIFICACION.

```
PROCEDURE BUSQUEDA_LINEAL;
BEGIN
      exito := FALSE;
      i     := 1;
      WHILE  NOT exito  AND  i<n  DO
      BEGIN
            IF datos [ i ]   = x THEN
               exito := TRUE;
            i := i+1;
      END;
      IF  exito  THEN
          WRITE ('El nodo:',x, 'pertenece a la lista')
      ELSE
          WRITE ('Búsqueda sin éxito');
END;
```

COMPLEJIDAD.

El tiempo de ejecución del método lineal de búsqueda está en función al número de comparaciones realizadas, lo cual es relativamente fácil de calcular.

El mejor caso se presenta cuando el primer elemento de la estructura es el elemento buscado, por lo que el método realiza sólo una comparación.

El número de comparaciones promedio que realiza el método son (n+1)/2, lo cual está dado por la siguiente sumatoria:

$$1/n \sum_{i=1}^{n} i/n = 1/n^2(n(n+1)/2)$$

$$= (n+1)/2$$

donde i/n es la probabilidad de que el elemento buscado se encuentre en la **i-ésima** posición, siendo la eficiencia del método en el caso promedio **O(n)**.

En el peor caso se realizan n comparaciones puesto que se deben comparar todos los registros, ya que el elemento deseado puede encontrarse en el último nodo de la lista o no estar contenido en ella.

2) **Búsqueda binaria.**

También llamada búsqueda por bisección, debido a que el método está basado en la bipartición repetida del intervalo de búsqueda.

Es un método de gran utilidad debido a su sencillez y velocidad; se aplica sólo a datos ordenados, los cuales se encuentran almacenados en forma contigua. Al realizar su proceso desarrolla los siguientes pasos:

a) MIENTRAS EL INTERVALO DE BUSQUEDA NO ESTE VACIO

- Determinar el elemento central del intervalo.
- Comparar este elemento con el elemento buscado.
- Si son iguales el proceso termina

  Sino

  Si el elemento buscado es mayor

  desechar la primera mitad del intervalo y considerar la segunda como el nuevo intervalo de búsqueda.

  Si el elemento buscado es menor

desechar la segunda mitad del intervalo y considerar la primer mitad como el nuevo intervalo.

b) Imprimir "BUSQUEDA SIN EXITO".

Ejemplo

Dada la siguiente lista de datos ordenada.

0 3 4 6 7 9 10 11 15 17 22 23

Determinar si el número 5 pertenece a la lista.

Iteracion:

1 (0 3 4 6 7) 9 (10 11 15 17 22 23)

2 (0 3) 4 (6 7)

3 6 (7)

de donde se concluye que el dato 5 no pertenece a la lista.

CODIFICACION.

```
PROCEDURE BUSQUEDA_BINARIA (datos:arreglo);
BEGIN
     inf    := 1;
     sup    := n;
     exito := FALSE;
     WHILE inf <= sup  AND  NOT exito DO
     BEGIN
          med := INT((inf+sup)/2);
          IF x < datos [ med ]     THEN
             sup := med-1
          ELSE
            IF x > datos [ med ]     THEN
               inf := med+1
            ELSE
               exito := TRUE;
     END;
     IF NOT exito THEN
        WRITE ('Elemento no encontrado')
     ELSE
        WRITE ('Elemento encontrado').
END;
```

COMPLEJIDAD DEL METODO.

En el mejor caso el número de comparaciones realizadas es igual a 1, ya que el elemento central obtenido en la primer iteración es el nodo buscado.

Dado que cada iteración de este método divide el intervalo de búsqueda a la mitad, reduce el número posible de candidatos en un factor de 2, gráficamente esto sería:

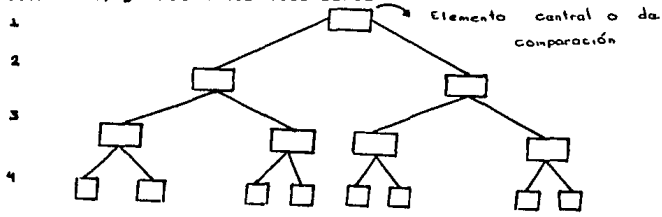


FIG 4.5

donde los dos subárboles representan las particiones realizadas en cada iteración.

El cálculo del número promedio de comparaciones es algo complicado de realizar, ya que cada elemento es igualmente probable de ocurrir.

Ahora el número de comparaciones realizadas para encontrar el i-ésimo elemento está dado por su nivel en el árbol más uno. Por lo tanto el promedio del número de comparaciones está dado por

$$\bar{C} = 1/n \sum_{i=1}^{n} NCOMP(i)$$

donde NCOMP(i) es el número de comparaciones realizadas para encontrar el elemento i.

Asumiendo que el árbol binario está lleno y n=2**k-1, hay 2**(k-1) nodos con altura k-1. Si ALFA denota la suma $\sum_{i=1}^{n} NCOMP(i)$, entonces ALFA es la suma de los primeros n términos de las series

$$ALFA = 1 + 2 + 2 + 3 + 3 + 3 + \ldots$$

$$= 1 + 2(2) + 2^2(3) + \ldots + 2^{-1}(j) + \ldots + 2^{-1}(k)$$

Una forma cerrada para el valor de ALFA puede ser obtenida usando una técnica de cálculo. Si se denota las k sumas parciales de las series geométricas, entonces

$$Sk(y) = 1 + y + y^2 + \ldots y^{k-1} = y^{k-1} /(y-1) \qquad y<>1$$

ahora, si y = 2z entonces

$$Sk(2z) = 1 + 2z + (2z)^2 + \ldots +(2z)^{k-1} = (2^k z^k - 1)(2z-1)$$

por lo que

$$d/dz(|Sk(2z)-1|\quad z)=2(2z)+2^2(3z^2)+\ldots+2^{k-1}(kz^{k-1})$$

de donde se tiene que diferenciar las series $|\ sk(2z)-1\ |\quad z$ término por término. Si evaluamos la derivación, consideramos z=1 y sumamos 1, se tiene que ALFA es:

$$ALFA = 1 + d/dz\ [\ |\ Sk(2z)-1\ |\quad z\ ]\quad y\ z=1$$

usando la forma cerrada para la suma parcial de la serie geométrica obtenemos

$$= 1 + 2\ (K-1)$$

y $$\bar{C} = \ /n = (1 + 2^k(K-1))/n$$

$$\bar{C} = (1 + (n+1)\ [\ log_2(n+1)-1\ ]\ )/n$$

$$\bar{C} = (n+1)/n\log_2(n+1)-1$$

de donde se deduce que la complejidad de la búsqueda binaria es $O(\log_2 n)$.

Por lo tanto, el número de comparaciones para el caso promedio y para el peor caso tiende hacia $\log_2 n$ a medida que n aumenta.

La búsqueda binaria no está garantizada que sea más rápida que otros métodos en listas muy pequeñas, ya que generalmente realiza pocas comparaciones, cada comparación requiere más cálculo.

COMPARACION DEL METODO LINEAL CON EL METODO BINARIO DE BUSQUEDA:

Gráficamente el comportamiento de estos métodos es:

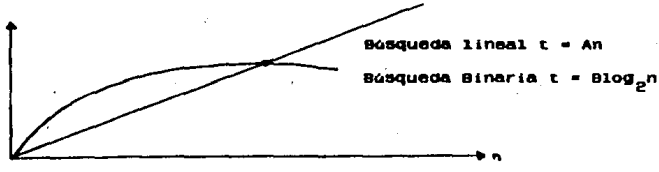


FIG 4.6

lo cual muestra que existe un punto óptimo de uso para cada uno de estos métodos. El punto de cruce de las curvas depende de la computadora empleada, ya que esto determina el valor de las constantes A y B.

3) Arbol binario de búsqueda.

Un árbol binario de búsqueda es aquel en el cual el valor que contiene un nodo, es mayor que cualquier valor de su subárbol izquierdo y menor o igual (si permite duplicaciones) que cualquier valor de su subárbol derecho; lo cual nos garantiza que el recorrido "enorden" del árbol da una lista ordenada.

Ejemplo

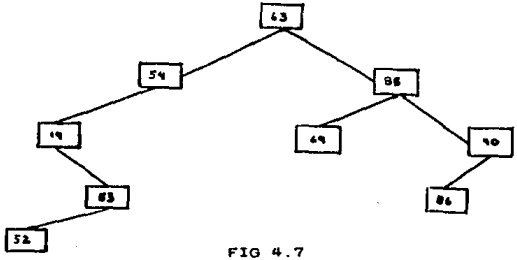


FIG 4.7

RECORRIDO ENORDEN: 14,52,53,54,63,64,85,86,90 ya que se visita primero el subárbol derecho en enorden, la raíz y después el subárbol izquierdo en enorden.

Ahora, un árbol binario de búsqueda sobre $x_1, x_2, \ldots, xn$, tal que el nombre simétrico de esos nodos coincide con el orden natural. Donde cada uno de los n+1 nodos externos corresponde a un espacio (gap) en la tabla.

Ejemplo

Tres formas distintas de árboles binarios de búsqueda sobre las llaves A,B,C,D

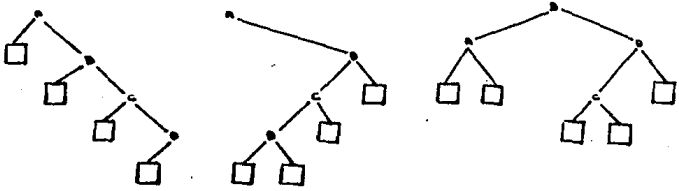


FIG 4.8

Ahora, buscando la llave z en un árbol binario de búsqueda se inicia comparando z con la llave que contiene la raíz, de ahí sucede que,

1) Si el árbol es vacio, z no pertenece a la tabla y la búsqueda termina sin éxito.

2) Si z es igual a la llave de la raíz, la búsqueda termina con éxito.

3) Si z precede a la llave de la raíz, la búsqueda continúa con el subárbol izquierdo de la raíz.

4) Si z sucede al valor de la raíz, la búsqueda continúa con el subárbol derecho de la raíz.

Los árboles binarios con muchas hojas tienen caminos largos. La longitud de un camino desde la raíz hasta una hoja proporciona el número de comparaciones hechas cuando la disposición representada por esa hoja es el orden clasificado para cierta lista de entrada L. Así, la longitud del camino más largo desde la raíz hasta una hoja es una cota inferior en el número de pasos efectuados por el algoritmo en el peor caso. En especial si L es la lista de entrada, el algoritmo efectuará al menos tantas comparaciones como la longitud del camino, probablemente en acción a otros pasos que no son comparaciones de claves.

Por lo tanto es necesario preguntarse lo cortos que pueden ser todos los caminos en un árbol binario con **k** hojas. Un árbol binario en el que todos sus caminos son de longitud **p** o menor, pueden tener una raíz, dos nodos en el nivel 1, cuatro en el nivel 2 y en general **2**i** nodos en el nivel **i**, así el número mayor de hojas de un árbol sin nodos en los niveles más altos que **p** es **2**p**. Es decir, un árbol binario con **k** hojas debe tener un camino de longitud no menor que **log k**. Si **k=n!**, entonces cualquier algoritmo de ordenación que sólo utilice comparaciones para determinar el orden, en el peor caso debe requerir un tiempo **O(log n!)**.

Una aproximación cercana a **n!** es **(n/c)** donde **e=2.7183** es la base de los logaritmos naturales.

Dado que **log(n/e) = nlogn - nloge** se observa que **log(n!)** es de orden **nlogn**. Es posible tomar una cota inferior exacta, observando que **n(n-1)(n-2)...(2)(1)** es el producto de por lo menos **n/2** factores, que a su vez son cada uno al menos **n/2**. Así **n! >= (n/2)**n/2** de aquí que:

$$\log(n!) \geq (n/2)\log(n/2) = (n/2)\log n - n/2,$$

por lo que la ordenación por comparaciones requiere un tiempo **O(nlogn)** en el peor caso.

CODIFICACION.

```
PROCEDURE BUSQUEDA(i:INTEGER; raiz:apuntador);
 BEGIN
      r:=FALSE;
      IF raiz = NIL  THEN
      BEGIN
         nodo := nil;
         padre:= nil;
      END;
      IF i=raiz^.elemento.el  THEN
      BEGIN
         nodo := raiz;
         padre:=NIL;
         r    := TRUE;
      END;
      IF i<raiz^.elemento.el THEN
      BEGIN
         nodo := raiz;
         padre:= NIL;
         r    :=  TRUE;
      END;
      IF i<raiz^.elemento.el THEN
      BEGIN
         nodoaux  := raiz^.li;
         padreaux := raiz;
      END
      ELSE
      BEGIN
         nodoaux  := raiz .ld;
         padreaux := raiz;
      END;
      WHILE (nodoaux <> NIL) AND (NOT r)  DO
      BEGIN
         IF i=nodoaux^.elemento.el THEN
         BEGIN
            nodo  := nodoaux;
            padre := padreaux;
            r     := TRUE;
         END
         ELSE
         BEGIN
            IF i<nodoaux^.elemento.el THEN
            BEGIN
               padreaux := nodoaux;
               nodoaux  := nodoaux^.li;
            END
            ELSE
            BEGIN
                padreaux := nodoaux;
                nodoaux  := nodoaux^.ld;
            END;
         END;
```

```
        END;
        IF NOT  r  THEN
        BEGIN
           padre := padreaux;
           nodo  := NIL;
        END;
    END;
```

4) Búsqueda secuencial indexada.

Existe otra técnica para mejorar la eficiencia de búsqueda en un archivo ordenado, pero que requiere una gran cantidad de espacio. Este método es denominado búsqueda secuencial indexada.

En este caso, se requiere una tabla auxiliar denominada índice, adicional al archivo ordenado; cada elemento en la tabla índice consta de una llave Kindex y un apuntador al registro en el archivo el cual corresponde al Kindex. Los elementos en la tabla índice deben estar ordenados en base a la llave.

La ventaja de este método es que los elementos del archivo pueden ser examinados en forma secuencial si todos los registros deben ser accesados; sin embargo, si sólo se busca algún elemento en particular la búsqueda secuencial se realiza en la tabla de índices (en lugar del archivo) minimizando con ello el factor tiempo, ya que la tabla de índices es proporcionalmente más pequeña que el archivo, por ejemplo, si la tabla índice es 1/8 del tamaño del archivo, entonces cada 8 registros del archivo están representados por uno de la tabla índice.

El empleo de una tabla de índices puede ser aplicable a una lista ordenada almacenada ya sea en forma ligada o en forma contigua. La utilización de una lista ligada implica mayor espacio para campos tipo apuntador pero las inserciones y supresiones se realizan en forma más eficaz.

En la implementación contigua, las eliminaciones de una tabla secuencial indexada se pueden realizar fácilmente mediante la asignación de banderas a las entradas que son eliminadas. Durante la búsqueda secuencial a través de la tabla se ignoran las entradas que han sido eliminadas, es decir, si un elemento fue borrado, aún cuando su llave permanezca en la tabla índice, en la tabla original la entrada estará marcada por una bandera que indique que ese elemento ya no debe tomarse en cuenta.

La inserción en una tabla secuencial indexada es un poco más difícil debido a que puede que no exista espacio entre dos entradas de la tabla, siendo necesario mover un gran número de elementos en la tabla. Sin embargo, si existe algún elemento que tenga una bandera de eliminado, se requiere mover sólo algunos elementos y emplear el espacio del elemento borrado para el nuevo elemento.

Si el archivo que contiene la información es tan grande que incluso el empleo de un índice no genera la suficiente eficiencia (ya sea por que la tabla índice es muy grande para reducir la búsqueda secuencial en la tabla, o porque la tabla índice es muy pequeña tal que las llaves adyacentes en los índices están muy lejos en el archivo) entonces se puede reducir a un índice secundario. El índice secundario actúa como un índice primario, que va a apuntar a las entradas en la tabla secuencial.

CODIFICACION.

```
PROCEDURE BUSQUEDA_INDEXADA;
 BEGIN
    found := FALSE;
    i     := 1;
    WHILE (i<=indexze)  AND  NOT found  DO
    IF kindex(i) > key THEN
       found := true
    ELSE
       i := i+1;
    IF i=1 THEN
        lowlim := 1;
    ELSE
        lowlim := pindex(i-1);
    IF found  THEN
        hilim := pindex(i)-1
    ELSE
        hilim := n;
   {busca en el archivo los registros comprendidos}
    entre lowlim y hilim
    j     := lowlim;
    found := false;
    WHILE (j<=hilim) AND (NOT found)  DO
        IF  K(j) = key  THEN
            found := TRUE
        ELSE
             j := j+1;
        IF  found  THEN
            posicion := j
        ELSE
            posicion := 0;
END;
```

B. METODOS DE BUSQUEDA POR TRANSFORMACION DE LLAVES.

Las técnicas de búsqueda por transformación de llaves (HASH), están basadas en el cálculo de la dirección en forma directa, a partir de la llave de un nodo. Este se emplea mediante una función de mapeo H, del conjunto K (dominio de esta función formado por el conjunto de llaves que pueden ser transformadas) sobre el conjunto L (rango de la función formada por números enteros) que representan las direcciones de memoria. Tal función:

$$H : K \rightarrow L$$

se llama función **HASH** o función de direccionamiento **HASH**

Con este método se tiene una búsqueda independiente del número de nodos de la estructura y su eficiencia es proporcional al tiempo utilizado para realizar la transformación.

Esta técnica no sólo se utiliza en la recuperación de la información sino también en su almacenamiento.

FUNCIONES HASH.

Existen muchas funciones de transformación que se pueden aplicar, pero es necesario tener en cuenta que, la elección de la función adecuada al problema repercute en la eficiencia de los algoritmos de almacenamiento y recuperación de la información.

Los dos criterios principales a tomar en cuenta al seleccionar una función **HASH** H : K->L son:

- La función H debe ser fácil y rapida de calcular.
- La función H debe, en la medida posible, distribuir uniformemente las direcciones **HASH** sobre el conjunto L, de tal forma que exista el menor número de colisiones posibles (es decir, de claves distintas que produzcan direcciones iguales).

A continuación se muestran algunas funciones **HASH**, fáciles y rapidas de evaluar en una computadora. Estas pueden ser aplicadas tanto a llaves numéricas, alfabéticas, o alfanuméricas.

1) Método de suma o plegado.

En este método la llave K se divide en varias partes $K_1, K_2, \ldots, Kr$, después estas se suman y del resultado se toman m bits (0,1) o dígitos decimales con los que se pueden formar $2^{**}m$ o $10^{**}m$ direcciones respectivamente.

$$H(K) = K_1 + K_2 + K_3 + \ldots + Kr$$

A veces se invierte el orden de los dígitos de las partes pares antes de sumarlas, con el objeto de afinar más las direcciones.

Ejemplos

1.- Empleando el método de suma, obtener la función de mapeo para las siguientes llaves:

$K_1 = 201 \qquad K_2 = 1024 \qquad K_3 = 564$

$H(K_1) = H(201) = 2 + 0 + 1 = 3$

$H(K_2) = H(1024) = 1 + 0 + 2 + 4 = 7$

$$H(K_3) = H(364) = 3 + 6 + 4 = 13$$

Usando **m=2** dígitos decimales, se tomarán los dígitos menos significativos (o sea, los de la extrema derecha de la suma).

Así, la dirección máxima que se puede formar es $10^2-1=100-1=99$ (la unidad que se le resta es por la dirección 0).

2.- Empleando el método de suma de la siguiente manera

a) Particionar la clave en dos partes y sumarlas,

b) Particionar la clave en dos partes, invertir la segunda y sumarlas

en ambas, considerar que sólo se tienen 99 direcciones. Obtener la función **HASH** de las siguientes claves:

	3205	7148	2345
a)	H(3205) = 32 + 05 = 37		
	H(7148) = 71 + 48 = 19		
	H(2345) = 23 + 45 = 68		
b)	H(3205) = 32 + 50 = 82		
	H(7148) = 71 + 84 = 55		
	H(2345) = 23 + 54 = 77		

2) **Método por multiplicación.**

Este método se basa en multiplicar la llave por cierta constante, que puede ser la misma llave, del resultado de ese producto se toman **m** bits o dígitos, para direccionar n (2^m o 10^m) localidades.

Ejemplo

Empleando el método de multiplicación, obtener la función **HASH** considerando los dos dígitos centrales para las llaves:

$$K_1 = 32 \qquad K_2 = 41 \qquad K_3 = 55$$

tomando como la constante en cada caso el mismo valor de la llave.

$$H(K_1) = H(32) = (32)(32) = 1024 = 02$$

$$H(K_2) = H(41) = (41)(41) = 1681 = 68$$

$$H(K_3) = H(55) = (55)(55) = 3025 = 02$$

Como se puede observar las llaves K_1 y K_3 producen direcciones iguales, lo que indica que se esta provocando una colisión.

3) Método de la mitad del cuadrado.

Este método es un caso especial del método por multiplicación, en el cual se calcula el cuadrado de la llave K, y la función HASH se define como:

$$H(K) = l$$

donde l se obtiene eliminando algunos dígitos de los extremos de K^2.

Es importante emplear las mismas posiciones de K^2 para todas las llaves.

Ejemplo

Obtener las siguientes funciones HASH mediante el método de la mitad del cuadrado, tomar para ello los dígitos cuarto y quinto por la derecha.

$K_1 = 3205 \qquad K_2 = 7148 \qquad K_3 = 2345$

$H(K_1) = H(3205) = K = 10272025 = 72$

$H(K_2) = H(7148) = K = 51093904 = 93$

$H(K_3) = H(2345) = K = 5499025 = 90$

4) Método de división.

El método consiste en tomar la llave K y dividirla por un número m (número cercano o mayor al número de llaves (K), es recomendable elegir un número primo o con pocos divisores, ya que así se minimiza el número de colisiones). El residuo de esta división es la dirección del elemento.

La función HASH de este método se define como:

$H(K) = K \text{ MOD } m$ con rango (0,1,2, ..., n-1)

$H(K) = K \text{ (MOD m)} + 1$ con rango (1,2,3, ..., n)

Ejemplo

Calcular por el método de división las siguientes direcciones:

$K_1 = 3205 \qquad K_2 = 7148 \qquad K_3 = 2345$ con m = 97.

Con rango (0,1,2, ... 98):

$H(K_1) = H(3205) = 3205 \text{ MOD } 97 = 4$

$H(K_2) = H(7148) = 7148 \text{ MOD } 97 = 67$

$$H(K_3) = H(2345) = 2345 \ MOD \ 97 = 17$$

Con rango (1,2,3, ... 99):

$$H(K_1) = H(3205) = (3205 \ MOD \ 97) + 1 = 5$$

$$H(K_2) = H(7148) = (7148 \ MOD \ 97) + 1 = 68$$

$$H(K_3) = H(2345) = (2345 \ MOD \ 97) + 1 = 18$$

5) Método de conversión de base.

Este método considera que los elementos que forman la llave representan un número en base P, entonces el método convierte este número a base Q (P>Q), de donde se toman m elementos para formar la dirección en un rango (0,1,2, ...,10^m-1).

Ejemplo

Empleando el método de conversión de base, obtener la función HASH para las llaves:

$$K_1 = 94 \qquad K_2 = 18 \qquad K_3 = 3.$$

Los números se encuentran en base 10, convertirlos a la base 8 y tomar los dos dígitos decimales del extremo izquierdo.

$$H(K_1) = H(94) = 136_8 = 13$$

$$H(K_2) = H(18) = 22_8 = 22$$

$$H(K_3) = H(03) = 3_8 = 3$$

Es importante tener presente que con los métodos antes mostrados, es posible inventar una gran cantidad de funciones HASH que pueden ser empleadas con éxito.

COLISIONES.

Las colisiones surgen cuando la dirección obtenida para un nodo o elemento ya ha sido asignado a otro nodo; este problema es casi imposible de evitar, por lo que se han diseñado varios métodos para buscarle un espacio disponible al nuevo nodo, a éstos, se les denomina Métodos de Resolución de Colisiones, siendo los mas importantes los de Direccionamiento abierto y los de Encadenamiento.

1) Direccionamiento abierto.

Este tipo de método considera que el archivo o tabla esta agrupado en forma circular, y consiste en revisar la estructura, hasta encontrar el elemento buscado o un espacio libre, lo cual indica:

- En el caso de querer insertar un elemento, el lugar disponible para hacerlo y,

- En el caso de querer realizar una búsqueda, que el elemento se encuentra en el archivo o tabla.

La gran desventaja del Direccionamiento abierto se presenta cuando se eliminan nodos de la estructura, esto es, un registro R se elimina de la posición r, ahora si se requiere buscar un elemento que se encuentra en una posicion posterior a r, al realizar el recorrido de la estructura y encontrar la posición r vacia, se concluirá que se llegó al final de la estructura, y esto significa que la búsqueda no ha tenido éxito, lo cual es FALSO, por lo que surge la necesidad de etiquetar la posición r para indicar que anteriormente contenía un registro y se pueda continuar con la búsqueda. Esto es la principal causa para que el direccionamiento abierto no se recomiende para archivos que serán modificados posteriormente.

FORMAS DE REVISAR LA ESTRUCTURA UTILIZANDO DIRECCIONAMIENTO ABIERTO.

a) Prueba y modificación lineal.

Se revisa la tabla en forma secuencial a partir del punto donde ocurrió la colisión hasta encontrar el nodo deseado o una posición vacia.

La desventaja fundamental de este tipo de direccionamiento directo es que los nodos tienden a agruparse, o sea, aparecen unos junto a otros (**CLUSTERING**), este fenómeno ocurre cuando la estructura se va llenando y provoca el aumento en el número de colisiones.

Ejemplo

Insertar el número 31 en la siguiente tabla, empleando el módulo 10.

H(31) = 31 MOD 10 = 1

TABLA INICIAL

0	60	
1	59	<- posición ocupada se procede a buscar una posición vacia
2		
3	23	
4	4	
5		
6	56	
7	66	
8	18	
9	29	

TABLA RESULTANTE:

0	60	
1	59	
2	31	<- elemento insertado
3	2	
4	4	
5		
6	56	
7	66	
8	18	
9	29	

b) Prueba y modificación cuadrática.

Este método en lugar de buscar las posiciones con direcciones h, h+1, h+2, ..., h+i, busca las posiciones con direcciones h, h+1, h+4, h+9, ..., h+i^2, con lo que se logra reducir el tiempo medio de búsqueda para un registro (al reducirse el agrupamiento).

Si el número de posiciones en la tabla es un número primo el empleo de este tipo de direccionamiento recorre en el peor caso la mitad de la tabla.

Ejemplo

Insertar el número 61 en la siguiente tabla, empleando el módulo 10.

$$H(61) = 61 \text{ MOD } 10 = 1$$

por lo que la secuencia a seguir es: 1,2,5,10

TABLA INICIAL			TABLA RESULTANTE		
0	60	<- ocupada	0	60	
1	59	<- ocupada	1	59	
2	31		2	31	
3	23		3	23	
4	4		4	4	
5		<- vacía	5	61	<- elemento
6	56		6	56	
7	66		7	66	
8	18		8	18	
9	29		9	29	

c) Doble direccionamiento HASH.

En esta técnica se emplea una segunda función HASH H, cada llave tiene dos direcciones: H(K)=h y H (K)=h <>m. La forma de buscar la posición de las direcciones es la siguiente: h, h+h, h+2h, h+3h, ..., h+ih sin embargo, si m es un número primo, la secuencia anterior accesará toda la tabla Ejemplo

TABLA INICIAL			TABLA RESULTANTE		
0	60	<- posición ocupada	0	60	
1	59		1	59	
2	31		2	31	
3	23	<- posición ocupada	3	23	
4	4		4	4	
5		<- posición vacía	5	61	<- elemento
6	56		6	56	insertado
7	66		7	66	
8	18		8	18	
9	29	<- posición ocupada	9	29	

2) Encadenamiento.

Esta técnica forma una lista ligada para cada clase de equivalencia (funciones **HASH** repetidas), por lo que debe ser revisada secuencialmente para poder insertar, eliminar y consultar nodos. Para lograr esto se requiere de un campo adicional conocido como **ENLACE**, que contenga la dirección del siguiente registro de la lista, el último nodo indica el fin de esta.

Esta técnica es más flexible y aumenta la rapidez de la búsqueda.

Ejemplo

Realizar la búsqueda del número 69 en la siguiente tabla, empleando mod 10.

H(69) = 9

	POSICION	INFORMACION	ENLACE	
	0	60	Δ	<- indica que es el último
	1	59	5	<- ir a 5
	2	31	Δ	
	3	23	Δ	
	4	4	Δ	
	5	69	Δ	<- el número 69 pertenece
	6	56	7	a la tabla. FIN
	7	66	Δ	
	8	18	Δ	
inicio	9	29	1	<- ir a 1

EFICIENCIA DE LOS METODOS DE BUSQUEDA POR TRANSFORMACION DE LLAVES.

La eficiencia de este tipo de métodos depende de la función de Mapeo **HASH** y la técnica de Resolución de colisiones. Para aumentar la eficiencia es recomendable asignar un número mayor de localidades de almacenamiento al número de nodos a insertar, ya que esto facilita la asignación de localidades vacías. A mayor factor carga mayor eficiencia:

Factor Carga = (# de nodos a insetar)/(# de nodos a almacenar)

Anora, considerando H: K->L, el factor de carga es la relación entre el número n de llaves de K y m el número de direcciones de L, o sea **n/m**.

C. METODO DE BUSQUEDA POR LLAVES SECUNDARIAS.

Algunas veces es necesario realizar una búsqueda basada en los valores de un campo que no es la llave primaria en un registro, a ese campo se le conoce con el nombre de llave secundaria o atributo del registro.

En general se asume que cada registro contiene varios atributos que con ciertos valores. La especificación de los registros que son requeridos se llama **QUERY**. Los **QUERIES** se encuentran restringidos a los tres siguientes tipos:

a) **QUERY SIMPLE:** en ella se da un valor especifico que se desea tenga el campo del registro.

b) **UN RANGO DE QUERY:** se da un rango especifico de valores para un atributo especifico, ejemplo costo < $18,000.

c) **QUERY BOOLEANA:** consiste de los dos tipos anteriores combinados con las operaciones AND, OR y NOT, ejemplo (residencia = México) AND (costo < $18,000).

El problema de descubrir búsquedas eficientes para estos tres tipos de **QUERIES** es difícil. El más conocido es denominado **Inversión de archivos.**

INVERSION DE ARCHIVOS.

El archivo se coloca en un orden distinto al original, esto significa que los registros y atributos son combinados, y en lugar de listar los atributos de un determinado registro, se listan los registros dado un determinado atributo.

En general, es necesario usar un archivo invertido junto con el archivo original, esto implica duplicidad y redundancia de la información pero, es compensada con la gran velocidad para localizar la llave secundaria.

Los componentes de un archivo invertido son las listas invertidas de todos los registros que tienen un valor dado en algún atributo.

Para las **QUERIES** booleanas se necesita insertar o unir mínimamente dos listas invertidas, esto se puede hacer de varias formas, por ejemplo, si nosotros ordenamos las listas, se puede seleccionar la más pequeña y revisar sus registros para checar sus atributos y listar las que cumplan con la condición:

(EDAD < 18) AND (COSTO > 2,000)

REGISTRO	EDAD	REGISTRO	COSTO
1	5	1	$1500
2	7	2	$2000
3	6	3	$1300
4	10	4	$ 500
.	.	.	.
.	.	.	.
.	.	.	.

LISTA: 1,2,3, ...

Este tipo de búsqueda es muy importante y los métodos empleados son perfeccionados día con día (ejemplo: el LMD o Lenguaje de Manipulación de Datos empleado en el manejo de base de datos).

CAPITULO V

ALGORITMOS PARA MANIPULAR CADENAS DE CARACTERES.

Si se observan los archivos contenidos en algún medio de almacenamiento desde un punto de vista general; encontramos en ellos sólo cadenas de caracteres con una función especial: realizar alguna tarea, ordenar realizarla (**Programas de Sistemas y Programas de Aplicaciones**), o contener información (**Archivos de Datos**). De ahí, la necesidad ineludible del manejo de las cadenas de caracteres en el área de la computación, que implica un estudio constante de los algoritmos involucrados en este tema. No sólo es importante tener algoritmos eficientes que manipulen este tipo de cadenas, sino también entender cómo realizan su función.

Así mismo, es importante el estudio de algoritmos que proporcionen seguridad en el manejo de la información contenida en los archivos (**CRIPTOLOGIA**) y de aquellos algoritmos que minimicen el área de almacenamiento empleado por los archivos (**COMPACTACION DE DATOS**).

5.1 ALGORITMOS DE MANIPULACION DE CADENAS DE CARACTERES.

Las cadenas de caracteres son el principal centro de los sistemas procesadores de textos, se definen como secuencias de letras, números y caracteres especiales. Estos objetos pueden ser muy grandes y los algoritmos eficientes juegan un papel muy importante en su manipulación, que se lleva acabo en todos aquellos métodos que relacionan las cadenas de caracteres.

DEFINICION DE SECUENCIA.

Tenemos que una sucesión es una lista de objetos, uno después del otro, la cual está numerada en el orden natural creciente de los números positivos $S_1 = 1$, $S_2 = 2$, ...

En la computación, la sucesión es un concepto denominado arreglo lineal o lista lineal, donde un arreglo se ve como una sucesión de posiciones, representadas como cajones.

Arreglo S :	S(1)	S(2)	S(3)	S(4)	...
	1	2	3	4	...

Por otra parte, dado un conjunto S se puede construir un conjunto S*, integrado por todas las sucesiones finitas de elementos de S, frecuentemente, el conjunto S no es un conjunto de números sino de símbolos, en este caso a S se le llama **Alfabeto** y a las sucesiones finitas de S* se les llama palabras o secuencias (cadenas de caracteres o string) de S. También se supone que S* tiene como elemento a la sucesión vacía o Λ secuencia vacía que no contiene símbolos y se indica como Λ o E.

Ejemplo:

S* = (0, 1, 2, 3, ..., 9, ...) **ALFABETO**

S = 3.1415... secuencia o cadena formada con caracteres tomados de S*.

Una secuencia finita

$$S_1, S_2, S_3, \ldots$$

se define formalmente como una función cuyo dominio es un conjunto de enteros positivos

$$f(i) = Si \quad , \quad i >= 1.$$

REPRESENTACION DEL ALMACENAMIENTO DE LAS CADENAS.

Debido a que una cadena es una secuencia de caracteres, la forma más apropiada para representarla es empleando una secuencia de localidades contiguas de almacenamiento en memoria.

5.1.1 Algoritmo de MARKOV.

El algoritmo de **MARKOV** llamado así por el matemático ruso A. A. Markov, es un sistema formal para la manipulación de cadenas. Originalmente se desarrolló como soporte a la teoría de la computación.

La estrategia general de un algoritmo de **MARKOV** es tomar una cadena X y por medio de un cierto número de pasos o producciones en el algoritmo, transformar X a una salida o cadena Y. Este proceso de transformación se desarrolla comúnmente en áreas de aplicación computacional tales como, la edición de texto o compilación de programas. En particular, la compilación puede ser a través de la transformación de cadenas desde un lenguaje origen a cadenas de un código objeto (tal como lenguaje ensamblador).

Una producción simple de Markov es una sentencia de la forma U -> W donde U y W representan cadenas en Vs, donde V no contiene el símbolo "->" y ".". En la producción, U se denomina antecedente y W consecuente, esta producción se aplica a cadenas de caracteres Z ∈ Vs, si existe por lo menos una ocurrencia de U e Z; de otra forma esta producción no es aplicable.

Si la producción se aplica, entonces la primera ocurrencia de U en Z es reemplazada por W, por ejemplo, si una producción **ba -> c** se aplica a la cadena de salida **abaaab** entonces el resultado es la cadena **acaab**, sin embargo la producción **baa -> c** no es aplicable a **abaaab** .

Un algoritmo de **MARKOV** consiste de un conjunto ordenado de producciones P_1, P_2, ..., P_n. El flujo de control del algoritmo de **MARKOV** depende de las producciones aplicables. Inicia con la producción 1 y termina de emplearla hasta que no sea aplicable continuando con las producciones siguientes. El proceso finaliza cuando se cumple una de las siguientes condiciones:

1) La última producción no es aplicable,

2) Una producción terminal es aplicable.

Ejemplo:

Considere que el algoritmo de Markov tiene las siguientes producciones

P_1 : ab -> b
P^1_2 : ac -> c
P^2_3 : aa -> a

en el alfabeto V = { a, b, c,¡como se observa este algoritmo borra todas las ocurrencias a en una cadena a excepción de una a siempre y cuando ésta aparezca como el último carácter.

Sea la cadena de salida bacaabaa, el símbolo --> se emplea para indicar el resultado de una transformación y una subcadena de caracteres es reemplazada,

bacaabaa --> bacabaa por P_1
bacabaa --> bacbaa por P^1_1
bacbaa --> bcbaa por P^2_2
bcbaa --> bcba por P^2_3

como la producción 3 no es aplicable el algoritmo finaliza.

SECUENCIA DE UN ALGORITMO DE MARKOV.

El siguiente algoritmo tiene una secuencia finita de producciones $P_1, P_2, \ldots, P_n$ las cuales se aplican a una cadena de salida $Z_0 \in V^*$, las variables i y j son índices y Zi denota el resultado después de aplicar la i-ésima transformación en la cadena de salida Z_0.

```
1. Inicializar i, el intermediario de la cadena indexada
   i := 0;
2. Inicializar j, la producción indexada
   j := 1;
3. Establecer condiciones para la ejecución.
   Repeat paso 4 While j <= n
4. Checar la producción aplicable y producción terminal,
   If  Pj es aplicable  Then
       aplicar la producción Pj
       i := i + 1  para obtener una nueva cadena Zi
       If  Pi  es una producción terminal  Then
           Write  Zi
           Exit
       Else
          j := 1
   Else
       j := j + 1;
5. El algoritmo finaliza
   Write (Zi);
   Exit.
```

FUNCIONES PRIMITIVAS.

Un análisis esencial cerrado de cadenas requeridas para sistemas de creación y edición podría requerir de la siguiente lista de funciones primitivas:

1) Creación de una cadena de texto.

2) Concatenación de dos cadenas formando una sola.

3) Buscar y reemplazar (si se desea) una subcadena dada por otra.

4) Probar la identidad de una cadena.

5) Calcular la longitud de una cadena.

La creación de una cadena implica no sólo la habilidad para construir una representación para una cadena, sino también la de conservar el valor de la cadena en una variable (o celda de memoria). Este hecho es inherente al modelo del Algoritmo de Markov, donde una cadena (la cadena **SUBCJET**) es creada y transformada eventualmente a una cadena de salida.

Por otra parte, la operación considerada como la más importante sobre una cadena es la concatenación, la cual se define de la siguiente manera:

La concatenación de dos caracteres alfabéticos **a** y **b** forman la secuencia **ab**, también se puede aplicar a cadenas de caracteres, por ejemplo, **ab** concatenada con **ab** da como resultado **abab** .

Por lo tanto, una cadena sobre un alfabeto es:

1) Una letra o un carácter del alfabeto, o

2) Una secuencia de letras o caracteres derivada de la concatenación de caracteres del alfabeto.

Por último, la longitud de la cadena, es el número de caracteres que ésta contiene, que es de gran importancia en el formato de salida para cadenas de caracteres.

5.1.2 Gramática.

El sistema formal para la manipulación de caracteres es la gramática.

En este momento existe menos interés en desarrollar manipulaciones en cadenas y cobra mayor importancia generar o reconocer un subconjunto específico de cadenas generadas desde un alfabeto.

DEFINICION: Un alfabeto es un conjunto de símbolos. Una cadena sobre un alfabeto Σ ; es una secuencia de símbolos de , en donde la cadena vacía (cadena que no contiene símbolos) es denotada por **E**.

Ejemplo:

= A, ..., Z es el alfabeto

"CASA" es una cadena.

Ahora, si XYZ es una cadena, "X" es llamada prefijo, "Y" es una subcadena y "Z" es el llamado sufijo de XYZ.

Ejemplo:

La longitud de una cadena X es denotada X y es el número total de símbolos que componen X.

Ejemplo:

Si X = CASA entonces X = 4

Un lenguaje sobre un alfabeto es un conjunto de cadenas de Σ, Si L1 y L2 son dos lenguajes, entonces L1L2 se denota como | XY/ X E L1 y Y E L2 | .

Ejemplo:

L1 = | A, ..., Z, E |
L2 = | 0, ..., 9, E |

X = VAR
Y = E por lo tanto XY = VAR,
X = VAR
Y = 2 por lo tanto XY = VAR2

DEFINICION: Si L es un lenguaje entonces se define L^0 = (E) y Lesi = LL ss(i-1) para toda i >= 1.

LA CERRADURA DE KLEEN: se denota L* y es el lenguaje L* = $\bigcup_{i=0}^{\infty}$ Lesi con i = 0 hasta infinito.

Ejemplo:

Si X = AB entonces X* = ABAB...AB

LA CERRADURA POSITIVA: de L se denota L+ y es el lenguaje L+ = $\bigcup_{i=1}^{\infty}$ L ssi con i = 1 hasta infinito, esta no acepta L^0 = (E) el Vacio.

Por supuesto, si es un alfabeto estara definida la operación binaria de concatenación para los elementos del alfabeto.

Por otra parte, una **GRAMATICA** consiste de un conjunto finito de reglas de reemplazamientos a producciones. Una producción

gramática y una producción del Algoritmo de **MARKOV**, son muy similares en su función de reemplazamientos específicos, pero tienen propósitos diferentes. Una producción de gramática se emplea para generar cadenas desde un lenguaje, mientras que una producción de **MARKOV** se usa en la manipulación de una cadena sujeto **(SUBJECT)** dada.

Un **METALENGUAJE** es un lenguaje empleado para describir otros lenguajes, definiendo su gramática. En el área de la Computación se han desarrollado mínimamente 4 metalenguajes para representar la sintaxis (gramática) de los lenguajes de programación. El más conocido y empleado es el BNF (Backus Naur Form o Backus Normal Form) que se hizo popular cuando se empleó para describir la sintaxis del lenguaje de programación ALGOL.

La **BNF**, emplea cuatro símbolos especiales (<, >, :=, /); donde una gramática se escribe como un conjunto de producciones, en la cual la parte izquierda está seguida por el símbolo :=, seguido a su vez por la lista de sus partes. La parte izquierda es un símbolo no terminal (por ejemplo: <dígito>) encerrados entre los símbolos < y >. La parte derecha es separada por los símbolos | si es una cadena terminal y / si no lo es. Un símbolo terminal es un carácter o una cadena de caracteres desde un alfabeto.

Existen dos importantes cuestiones en lo que concierne a la generación de cadenas de una gramática. Primero, se tiene el descuido (por motivos de legibilidad) de poner en los símbolos terminales de la gramática caracteres blancos, que aparecen normalmente entre las palabras de las sentencias. Segundo, se puede notar que no todas las cadenas compuestas de símbolos terminales forman sentencias, lo que hace necesario un método para analizar las diferentes partes de una cadena y determinar cuando la cadena es una sentencia en el lenguaje. Tal proceso existe y es conocido como **PARSEN**.

METODOS PARA ANALIZAR UN LENGUAJE.

Para realizar el análisis del lenguaje existen dos métodos esenciales, denominados análisis de "arriba-abajo" o **TOP-DOWN** y análisis de "abajo-arriba" o **BOTTOM-UP**.

A) TOP-DOWN.

El método parte del nivel más alto y procura encontrar una vía hacia abajo de tal modo que, termine por analizar cada símbolo del conjunto de sentencias.

Una prueba tentativa de este método se da al construir un árbol sintáctico que inicia el análisis comenzando en la raíz y procediendo con los hijos. El efecto de tales procesos es generar las sentencias sistemáticamente desde el lenguaje hasta dar con la cadena en cuestión. Si la cadena no se encuentra, se concluye

que ésta no es una sentencia del lenguaje. Esta generación de sentencias puede ser creada con la ayuda de una relación especial => donde x => y es interpretado como "la cadena x produce y" (o "y reduce a x") durante un paso del PARSEN.

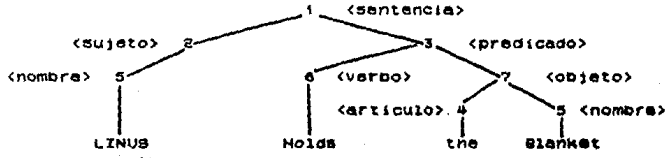
Ejemplo:

Los pasos en la generación de la sentencia "LINUS Holds the Blanket" dadas las reglas gramaticales para Gi, son

```
<sentencia> =>                  <sujeto> <predicado> por producción 1.
            =>                  <nombre> <predicado> por producción 2.
            => LINUS                     <predicado> por producción 5.
            => LINUS                <verbo> <objeto> por producción 3.
            => LINUS Holds                  <objeto> por producción 6.
            => LINUS Holds  <articulo> <nombre>      por producción 7.
            => LINUS Holds the         <nombre>      por producción 4.
            => LINUS Holds the Blanket               por producción 5.
```

Mapeo:

El árbol sintáctico es:



donde

```
Vn = { <sentencia>, <sujeto>, <predicado>, <articulo>,
       <nombre>, <verbo>, <objeto> }
Vt = { a, the, Linus, Charlie, Snoopy, blanket, dog, song,
       holds, pets, sings }
S =    <sentencia>.
P =    1 <sentencia> := <sujeto> <predicado>
       2 <sujeto>    := <articulo> <nombre> / <nombre>
       3 <predicado> := <verbo> <objeto>
       4 <articulo>  := a / the
       5 <nombre>    := Linus / Charlie / Snoopy   blanket
                        dog | song
       6 <verbo>     := holds | pets | sings
       7 <objeto>    := <articulo> <nombre> | <nombre> |
```

B) BOTTOM-UP

El método parte del nivel más bajo y ensaya sucesivamente el agrupamiento de símbolos para obtener conceptos de nivel más elevado.

En este caso, la realización del árbol sintáctico se inicia en los niveles más bajos (en los hijos) y va moviéndose hacia arriba (hacia la raiz). La relación especial => usada donde x => y significará " y reduce a x", ya que esta es la interpretación empleada en el **BOTTOM-UP** Parsen.

Ejemplo:

Las siguientes series de derivación son las que resultan en un Parsen **BOTTOM-UP** de la sentencia "LINUS Holds the Blanket",

```
<nombre> Holds the Blanket              =>  LINUS Holds the Blanket
                                                    por producción 5.
<sujeto> Holds the Blanket              =>          por producción 2.
<sujeto> <verbo> the Blanket            =>          por producción 6.
<sujeto> <verbo> <articulo> Blanket     =>          por producción 4.
<sujeto> <verbo> <articulo> <nombre>    =>          por producción 5.
<sujeto> <verbo> <objeto>               =>          por producción 7.
<sujeto> <predicado>                    =>          por producción 3.
         <sentencia>                    =>          por producción 1.
```

En la práctica se utilizan ambos métodos, el más apropiado puede variar de acuerdo con las circunstancias.

En una gramática pueden presentarse producciones recursivas, por lo que es necesario tener en cuenta lo siguiente.

La recursión es un proceso en el que un objeto se define en términos de él mismo.

Ejemplo:

```
Tenemos  G2 = (Vn, Vt, S, P)
donde    Vn = | <digito> <no.> <número> |
         Vt = | 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 |
         S  = | <número> |
         P  =   1 <número> := <no.>
                2 <no.>    := <digito> 1 <no.> <digito>
                3 <digito> := 0 1 1 1 2 1 3 1 4 1 5 1 6 1
                              7 1 8 1 9 |
```

Las producciones definidas recursivamente se emplean en instancias, en las que ciertas subcadenas pueden llamarse repetidamente en una sentencia de un lenguaje.

El poder de las producciones recursivas en una gramatica depende de que estas no sean mal empleadas, por lo que las siguientes dos declaraciones pueden ser propuestas como teoremas:

1) Una gramática que contiene producciones definidas en forma recursiva describe un lenguaje infinito, es decir, un lenguaje con un número infinito de sentencias.

2) Una gramática que no contiene producciones recursivas describe un lenguaje finito.

El **PARSEN** sólo es responsable de identificar sentencias sintácticamente correctas, que son consideradas el inicio de palabras compuestas de un lenguaje. El valor exacto de una palabra ligada o el significado real de una secuencia no son consecuencia del Parsen, esos valores son identificados por un proceso conocido como **SCANNER**.

En suma, los Algoritmos de **MARKOV** y las Gramáticas de frases estructuradas son equivalentes con respecto al poder computacional; sin embargo, ambos son modelos formales diseñados para diferentes propósitos. El modelo de **MARKOV** exhibe propiedades básicas que son determinantes en el manejo de cadenas y rudimentarias cadenas patrón o pattern), y la gramática, es un modelo de reconocimiento y generación de cadenas.

RECONOCIMIENTO DE LENGUAJE

En esta ocasión nos ocuparemos de la forma de reconocimiento de lenguaje, que es generado por una gramática diferente, conocida como expresión regular.

Anteriormente definimos un lenguaje L sobre un alfabeto Σ como un conjunto de cadenas de caracteres pertenecientes a Σ . En seguida presentamos algunas definiciones relacionadas con un lenguaje L.

DEFINICION: Si L es un lenguaje entonces se define $L^0 = (E)$ y $L^i = LL^{(i-1)}$ para toda $i >= 1$.

LA CERRADURA DE KLEEN se denota L^* y es el lenguaje $L^* = \bigcup_{i=0}^{\infty} L^i$ con i = 0 hasta infinito.

Ejemplo:

Si X = AB entonces $X^* = ABAB \ldots AB$

LA CERRADURA POSITIVA de L se denota L^+ y es el lenguaje $L^+ = \bigcup_{i=1}^{\infty} L^i$ con i = 1 hasta infinito, que no acepta $L^0 = (E)$ el vacío.

DEFINICION: La operación binaria de concatenación para elementos del alfabeto, es denotada como L^*.

LENGUAJE DE EXPRESIONES REGULARES.

Si definimos una expresión regular sobre un alfabeto:

$$\Sigma = \{A, \ldots, Z, 0, \ldots, 9, *, \ldots\}$$

sería

Identificación = letra (letra | digito)*

OR

Cerradura Positiva — Con el asterisco (*) se indica a la Cerradura de Kleen.

En este caso, con la cerradura positiva se garantiza que un identificador no tenga o sea el vacío.

Ejemplo:

Identificador1 = X
Identificador2 = X1

Así un identificador es una cadena de carácteres cuyo carácter inicial es una letra.

DEFINICION: Una expresión regular definida sobre un alfabeto es una expresión construida bajo una de las siguientes relaciones o reglas:

1) E, la cadena vacía se define como expresión regular (E.R.)

2) Para cada símbolo a E , a se dice que es una expresion regular.

3) Si R y S son dos expresiones regulares denotadas en los lenguajes LR y LS, entonces:

   i) R + S es una expresión regular denotada en el lenguaje LR U LS.

   ii) RS es una expresión regular en el lenguaje LR*LS

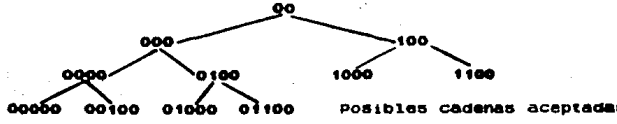
   iii) R* es una expresión regular.

   Además, la construcción de expresiones regulares es recursiva.

Ejemplo:

Sea = 0, 1 y E.R. = (0+1)* 00

parte final de la expresión empieza con el blanco o vacío.

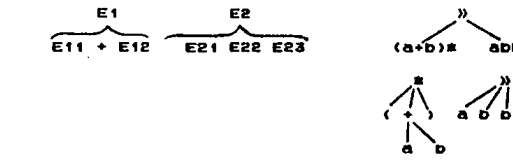


Una expresión regular compuesta puede dividirse en componentes elementales o primitivos, donde cada componente es una expresión regular simple.

Ejemplo:

$$\Sigma = \{a, b\} \quad y \quad R. = (a+b)^* abb$$

descomponer la expresión regular en sus componentes elementales o primitivos.



PASOS PARA GENERAR UN LENGUAJE.

Para generar un lenguaje por medio de una expresión regular es necesario seguir los siguientes pasos:

1) Encontrar la Expresión Regular que define el lenguaje.

2) Construir un Autómata Finito No Determinístico que reconozca el lenguaje.

3) Construir el Autómata Finito Determinístico que reconozca el Lenguaje

4) Minimizar el Autómata Finito Determinístico.

5) La Generación del Código.

ESQUEMA:

Generar Lenguaje ⟩ **Expresión Regular** => **Autómata Finito No Determinístico** => **Autómata Finito Determinístico** => **Minimización de Estados en el AFD** => **Generación de Código.**

Un Autómata es el mecanismo para reconocer lenguajes en términos de una Expresión Regular, y se dice que el Autómata es

No Determinístico si contiene signos "+", ya que ante esto no sabe que secuencia seguir.

AUTOMATA FINITO NO DETERMINISTICO:

DEFINICION: Un Autómata Finito No Determinístico (AFND o AFN) es una quintupla definida por (S, , d, S_0, F) donde:

1. S es el conjunto finito de los estados de control.
2. es el alfabeto de símbolos de entrada.
3. d es una función de transición de estados, la cual mapea Sx(U) a un conjunto de subconjuntos de S.
4. S_0 es el estado inicial de control finito.
5. F S es el conjunto de estados aceptantes o finales.

También, un AFN es una gráfica considerada como inteligente, formada por un conjunto \ S, d\ donde S es el conjunto de estados y d son los arcos que producen transiciones de un estado i a un estado j.

CONSTRUCCION DE UN AFN.

ENTRADA: Una expresión regular sobre un alfabeto Σ .

SALIDA : Un **AFN** el cual es el reconocedor o aceptador del lenguaje denotado por R.

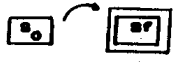
METODO : Primero - Descomponer la expresión regular en sus componentes primitivos o elementales.

Ejemplo: E.R. = abc entonces E.R.1 = a, E.R.2 = b, E.R.3 = c.

Segundo - Para cada componente construir un **AFN** de forma inductiva, como sigue:

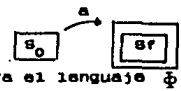
Se construye un **AFN** según los puntos 1 y 2 que forman la base, y el punto 3 es la forma inductiva para la formación del **AFN**.

Punto 1) Autómatas Finitos No Deterministicos que aceptan expresiones regulares de longitud 1.



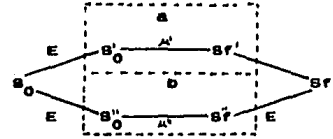
| E | **Transición espontánea, ya que tiene una transacción vacía.**

Punto 2) Para toda E Σ entonces se tiene



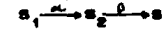
particularmente para el lenguaje Φ

Punto 3) Dados los AFN que se construyen del punto 2) según la expresión regular, entonces inductivamente se tiene lo siguiente:



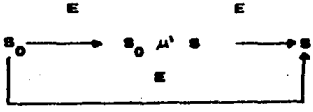
(a) μ

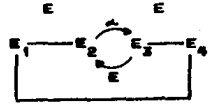
Sea α una E.R.1 con AFN μ' y sea β otra E.R.2 con AFN μ'' entonces la E.R. = $\alpha + \beta$ define inductivamente al AFN μ (a).



(b)

Sea $\alpha \in \Omega$ con AFN μ'
Sea $\beta \in \Omega$ con AFN μ''
$\Big\}$ $\alpha\beta$ es un E.R. con AFN μ (b).





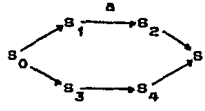
(c)

Sea α un E.R. con AFN μ' entonces la E.R. = α^* con AFN μ (c).

Ejemplo:

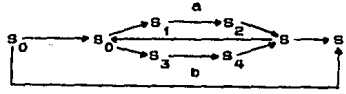
Sea Σ = {a, b} y la E.R. = (a+b)*ab, construir el Autómata Finito No Determinístico de la E.R. dada:

1) a+b (a ó b)

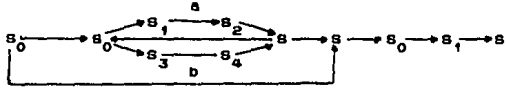


	a	b	E
S_0			S_0,S
S_0			S_1,S_3
S_1	S_2		
S_2			S
S_3		S_4	
S_4			S
S			S,S_0
S			S_0
S	S		
S		S	
S	ESTADO FINAL		

2) (a+b)



3) (a+b) ab



Lo siguiente es construir el Autómata Finito Determinístico a partir del **AFN** de la Expresión Regular dada.

Tenemos que dada una gráfica denominada **AFND**, que está constituida por nodos y arcos etiquetados por símbolos de un alfabeto Σ, su transposición a un **AFD** equivale a encontrar las clases de equivalencia (Sea **R** una relación de equivalencia sobre el conjunto **A**. Si **a E A** => **a** = **x E R / xRa** , se llama clase de equivalencia de **a** y nunca es vacía, porque **R** es reflexiva) de estados que son apuntados por arcos del mismo símbolo, ésto se consigue aplicando el siguiente algoritmo.

CONSTRUCCION DE UN AFD:

ENTRADA : Un **AFN**.

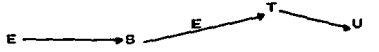
SALIDA : El **AFD** correspondiente.

METODO : Encontrar las clases de equivalencia de los estados, que están conectados por transiciones vacías. Al método se le denomina **Método de la E-Cerradura**. La clase de equivalencia está determinada por la cerradura y tiene como representante de clase al nuevo estado del **AFD**.

CERRADURA VACIA: La E-cerradura de (S) es el conjunto de estados que construye a partir de las siguientes reglas:

a) Se añade el estado S a la E-cerradura (S).

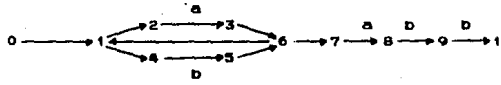
b) Si t pertenece a la E-cerradura (S) y existe una transición E de t al estado u entonces se añade u a la E-cerradura (S).



E-Cerradura (S) = {S,T,U}

Ejemplo:

Tenemos que el AFN de la expresión regular (a+b)*abb es:



Donde

R = (a + b)* abb

i) (a+b)* cumple con la Cerradura de Kleen donde a y b son E.R.

ii) abb es una concatenación.

Se quiere de obtener el AFD, por lo que se deben encontrar las clases de equivalencia:

A = E-cerradura (0) = {0, 1, 2, 4, 7} Son los estados a los que se puede ir con la transición E partiendo del estado 0.
con a vamos a {3, 8}
con b vamos a {5}

B = E-cerradura ({3, 8}) = {3, 8, 6, 1, 4, 2, 7}
se incluyen en el conjunto
Con a vamos a {3, 8}
con b vamos a {5, 9}

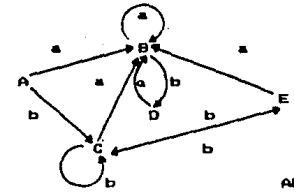
C = E-cerradura ({5}) = {1, 2, 4, 5, 6, 7}
con a vamos a {3, 8}
con b vamos a {5} como se repite ya no se toma en cuenta.

D = E-cerradura ({5, 9}) = {1, 2, 4, 5, 6, 7, 9}
con a vamos a {3, 8}
con b vamos a {5, 10}

E = E-cerradura (5, 10) = \1, 2, 4, 5, 6, 7, 10\
con a vamos a 3, 8
con b vamos a 5

Finalmente tenemos:

d	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C



AFD

Una vez obtenido el AFD se procede con la minimización de éste (reducción de estados), lo cual se logra mediante el algoritmo siguiente:

MINIMIZACION DEL NUMERO DE ESTADOS DE UN AFD:

ENTRADA : Un **AFD M** con un conjunto de estados S, entradas , transiciones definidas para todas las entradas y estados, estado inicial, un conjunto de estados finales F.

SALIDA : Un **AFD M** aceptando el mismo lenguaje que M y teniendo el menor número de estados posibles.

METODO : Construimos una partición TT de estados.

Inicialmente TT consiste de dos grupos, los estados finales F, y los estados no finales S-F. Entonces se construye una nueva partición TT nueva, que será siempre un "**refinamiento**" de TT, significando que TT nueva consta de los grupos de TT.

Si **TT nueva** <> TT reemplazamos TT por TT nueva y repetimos el procedimiento "**Encuentra TT nueva**". Si **TT nueva** = TT, entonces no pueden ocurrir más cambios, y termina esta parte del algoritmo. Note que cada TT nueva tiene al menos tantos grupos como la anterior.

PROCEDIMIENTO PARA ENCONTRAR TT - NUEVA:

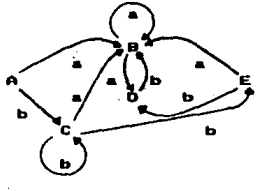
FOR cada grupo G de TT:
BEGIN
Particionar G en subgrupos, tal que dos estados s y t de G estén en el mismo subgrupo si y ,sólq si para todos los símbolos de entrada a, los estados s y t tienen

transiciones de estados en el mismo subgrupo de TT /* a lo más un estado estará en un subgrupo por sí mismo */
Poner todos los subgrupos formados en TT nueva.

END.

Ejemplo:

Sea = a, b, R = (a+b)*abb y el AFD M el siguiente:



d	a	b
A	B	C
B	B	D
C	B	C
D	B	E
E	B	C

obtener el AFD M minimizado.

TT = { A, B, C, D }, { E } F = E y S-F = { A, B, C, D }
Estado final.

TT nueva = { A, B, C }, { D }, { E }
Provoca que nos salgamos de la clase de equivalencia.

{ A, B, C, D }
con a vamos a { B }
con b vamos a { C, D, E } y salimos, porque D nos lleva a E con "b" y E no está en la partición.

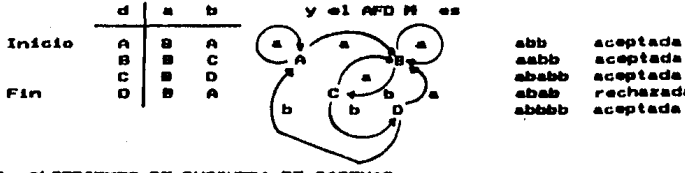
TT = TT nueva → se actualiza.

TT = { A, B, C }, { D }, { E }
A, B, C
con a vamos a { B }
con b vamos a { C, D } y salimos, porque B nos lleva a D con "b" y D no está en la partición.

TT = TT nueva → se actualiza.

TT = { A, C }, { B }, { D }, { E }
A, C
con a vamos a { B }
con b vamos a { C } Como ya se llegó a un estado terminal.

TT = { A, C }, { B }, { D }, { E }
A B C D



5.2 ALGORITMOS DE BUSQUEDA DE CADENAS.

En la manipulación de cadenas la operación más común es la búsqueda de una cadena o subcadena dada dentro de un texto o conjunto de cadenas.

5.2.1 Definición del problema.

La operación fundamental en las cadenas es conocida como **PATTERN MATCHING** (emparejamiento de patrones) la cual se define como: dada una cadena texto (Se usará el término texto para referirnos tanto a una secuencia de 0 y 1 s o a otro tipo especial de cadena) de longitud N (llamado **SUBJECT**) y una cadena patrón o modelo de longitud M (llamado **PATTERN**), se debe detectar la ocurrencia del pattern en el subject.

El problema de **PATTERN MATCHING** puede ser caracterizado como un problema de búsqueda con el pattern como la llave, pero los algoritmos de búsqueda que se estudiaron en el capítulo anterior, no pueden ser directamente aplicados puesto que el pattern puede ser grande y además porque no se sabe como se encuentran alineados los caracteres en el texto.

Este es un problema interesante para estudiar debido a que existen una gran variedad de algoritmos propuestos recientemente, que no sólo proveen un espectro de sus métodos de uso práctico, sino también ilustran algunas técnicas fundamentales de diseño.

Muchos algoritmos para este problema pueden ser fácilmente extendidos a encontrar todas las ocurrencias del pattern en el texto, ya que estos emplean la búsqueda secuencial y pueden ser reiniciados en un punto posterior al del inicio de una pareja, para intentar encontrar otra pareja si existe.

5.2.2 Una breve historia.

En 1970, S.A. Cook da a conocer un resultado teórico acerca de un tipo particular de máquina abstracta, insinuando que existe un algoritmo que resuelve el problema **PATTERN-MATCHING** en un tiempo proporcional a M + N en el peor caso. D.E. Knuth y V.R. Pratt siguieron con interés la construcción que Cook empleó en su teorema, generando un algoritmo, que tuvieron la habilidad de refinar obteniendo un algoritmo relativamente práctico y simple.

Este proceso se consideró un raro y sofisticado ejemplo de un resultado teórico que inmediatamente tuvo aplicación práctica.

Este algoritmo quedo fuera porqué J.H. Morrisio descubrió también como una solución de molestos problemas prácticos a los que se enfrentan al intentar la implementación de un editor de texto. El hecho de que el mismo algoritmo sirviera para dos aproximaciones diferentes aumentó la confiabilidad como la solución fundamental del problema.

Knuth, Morris, y Pratt publicaron su algoritmo hasta 1976, y mientras tanto R.S. Boyer y J.S. Moore (e independientemente R.W. Gosper) descubrían un algoritmo, mucho más rápido y eficiente en muchas aplicaciones, ya que sólo examina una fracción de los carácteres de la cadena en el texto. Muchos editores de texto usan este algoritmo para ejecutar un decremento notable en el tiempo empleado en la búsqueda de carácteres.

La historia ilustra que la búsqueda de un mejor algoritmo aún no ha sido bien justificado, pero existe la sospecha de que existen aún más desarrollos en el horizonte del problema **PATTERN-MATCHING**.

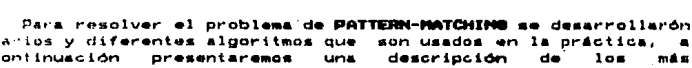
5.2.3 Algunos algoritmos.

Para resolver el problema de **PATTERN-MATCHING** se desarrollarón varios y diferentes algoritmos que son usados en la práctica, a continuación presentaremos una descripción de los más importantes.

ALGORITMO BRUTE FORCE

Un método obvio para el problema de Pattern Matching, es el chequeo para cada posible posición en el texto en la cual el pattern puede ser encontrado.

Por ejemplo:



Búsqueda exitosa.

El siguiente programa busca por este camino la primer ocurrencia de un pattern p[1..M] en la cadena texto a[1..N];

```
FUNCION Brutesearch: INTEGER;
VAR
   i, j : INTEGER;
BEGIN
   i := 1;
   j := 1;
   REPEAT
```

```
            IF a[i] = p[j] THEN
            BEGIN
               i := i + 1;
               j := j + 1;
            END
            ELSE
            BEGIN
               i := i - j + 2;
               j := 1;
            END;
         UNTIL (j>M) OR (i>N);
         IF j > M THEN
            Brutesearch := i - M
         ELSE
            Brutesearch := i;
      END;
```

El programa maneja un apuntador i en el texto y un apuntador j en el pattern. Si al final el pattern se encuentra (j>M) entonces se dice que una pareja ha sido detectada. Si el pattern no ocurre en el texto, entonces se regresa el valor N+1;

En una aplicación de edición de texto, el ciclo interno de este programa, es rara vez ejecutado y el tiempo de ejecución es aproximadamente proporcional al número de caracteres del texto examinado. Por ejemplo, suponemos que se quiere buscar el pattern STING en la cadena:

```
A STRING SEARCHING EXAMPLE CONSISTING OF SIMPLE TEXT.
  ||     |                                    |
  ST     S                                    S
```

donde la sentencia j := j + 1 se ejecuta sólo cuatro veces (una por cada S, pero dos veces por la primera ST), antes de comprobar que no se encuentra ninguna pareja. En otras palabras este programa puede ser muy lento para algunos patterns.

Por ejemplo, si el pattern es 0000001 y el texto

```
00000000000000000000000000000000000000000000000000001
                          52                          ^
                                                      pareja
```

en este caso, j se incrementa 7 * 45 = 315 antes de que la pareja sea encontrada.

Una versión diferente del Algoritmo de BRUTE FORCE es presentado a continuación.

```
ALGORITMO:
1       i := 0                              p = Pattern,
2       WHILE i < n DO                      s = Subjet.
```

```
        BEGIN
 3          i := i + 1;
 4          j := i;
 5          k := 1;
 6          WHILE Sj = Pk DO
            BEGIN
 7             IF k = m THEN RETURN 'EXITO'
 8             ELSE
               BEGIN
 9                j := j + 1;
10                k := k + 1;
               END;
            END;
11          WRITE 'NO SE ENCONTRO';
        END;
```

Para este algoritmo, se conociera la longitud de la cadena pattern, la prueba en la línea 2 podría ser **WHILE i<n-m+1** para terminar la operación cuando la longitud de la subcadena que falte por analizar sea más pequeña que el pattern (P).

ALGORITMO KNUTH - MORRIS - PRATT

La idea básica de este algoritmo es que cuando se detecte un error en el emparejamiento (entiéndase por emparejamiento a la ocurrencia del pattern en el texto), se observen todos los caracteres que se conocieron durante su desarrollo (ya que ellos están en el pattern) y de algún modo tener la habilidad de tomar ventaja de esta información, en lugar de retroceder el apuntador sobre todos los caracteres conocidos, se recorra hasta donde nos sea conveniente.

Por ejemplo:

Cuando se busca el pattern 10100111 en el texto 1010100111, el primer error que es detectado está en el quinto caracter, por lo tanto el apuntador debe retroceder, pero sólo hasta el tercer carácter y continuar la búsqueda, puesto que no sucede otro error se encuentra la cadena. No se puede decir con exactitud el tiempo que consume este proceso, puesto que, depende sólo del pattern, como se muestra en la siguiente tabla perteneciente a nuestro ejemplo:

j	p[1..j-1]	next[j]
2	1	1
3	10	1
4	10 1	2
5	10 10	3
6	10 100	1
7	10 100 1	2
8	10 100 1 1	2

donde el arreglo **next[1..M]** puede ser empleado para determinar qué tan lejos se debe o puede regresar cuando un error es detectado.

A continuación presentamos el programa que lleva a cabo el método mencionado:

```
FUNCION Kmpsearch: INTEGER;
VAR
   i, j :INTEGER;
BEGIN
   i := 1;
   j := 1;
   REPEAT
      IF (j=0) OR (a[i]=p[j]) THEN
      BEGIN
         i := i + 1;
         j := j + 1;
      END
      ELSE
         j := next[j];
   UNTIL (J>M) OR (i>N);
   IF j>M THEN
      Kmpsearch := i - M
   ELSE
      Kmpsearch := i;
End;
```

Funcionalmente, este programa es igual al proceso Brute Force, pero es probable que sea mucho más rápido para patterns, que sean altamente repetitivos.

El siguiente programa es básicamente el mismo que el anterior, excepto que se emplea para emparejar el pattern en contra de si mismo.

```
PROCEDURE Initnext;
VAR
   i, j : INTEGER;
BEGIN
   i := 1;
   j := 0;
   next[i] := 0;
   REPEAT
      IF (j=0) OR (p[i]=p[j]) THEN
      BEGIN
         i := i + 1;
         j := j + 1;
         next[i] := j;
      END
      ELSE
         j := next[j];
   UNTIL i>M;
END;
```

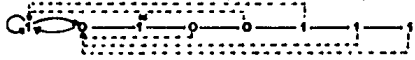
Por otra parte, el siguiente programa es exactamente equivalente al programa anterior, pero para el pattern 10100111 del ejemplo que hemos considerado, es mas eficiente:

```
BEGIN
      i := 0;
0:    i := i + 1;
1:    IF a[i]<>'1' THEN GOTO 0;  i := i + 1;
2:    IF a[i]<>'0' THEN GOTO 1;  i := i + 1;
3:    IF a[i]<>'1' THEN GOTO 1;  i := i + 1;
4:    IF a[i]<>'0' THEN GOTO 2;  i := i + 1;
5:    IF a[i]<>'0' THEN GOTO 3;  i := i + 1;
6:    IF a[i]<>'1' THEN GOTO 1;  i := i + 1;
7:    IF a[i]<>'1' THEN GOTO 2;  i := i + 1;
8:    IF a[i]<>'1' THEN GOTO 2;  i := i + 1;
SEARCH := i - 8;
```

Este programa emplea una mínima variedad de operaciones básicas para resolver el problema de la búsqueda de cadenas. Esto significa que puede ser fácilmente descrita en términos de un modelo de máquina simple llamado Estado de Máquina Finita (Autómata Finito).*

* Nota El conocimiento de la Teoría de Autómatas se hace necesario en este método.

La representación de máquina finita para el ejemplo es:



La máquina consta de estados (indicados como círculos etiquetados) y transiciones (indicadas por líneas). Cada estado tiene dos transiciones: una transición esperada (obteniendo una pareja) marcada con la línea sólida, y una transición no esperada (no se obtiene una pareja) marcada con la línea punteada.

La transición no esperada en el primer estado (marcado con una línea doble) también requiere que la máquina busque en el próximo carácter.

Por ejemplo, suponga que encontramos la subcadena 1011 en el texto cuando buscamos el pattern 10100111, después de emparejar 101, se encuentra un error en el cuarto carácter, la máquina indica que hay que checar el segundo carácter puesto a que ya encontramos el 1 en el tercer carácter, y sin embargo, no se tiene una pareja, debido a que el próximo carácter no es cero como se requiere por el pattern.

El algoritmo Knuth-Morris-Pratt probablemente no es significativamente más rápido que el método Brute-Force en muchas de las aplicaciones actuales, porque pocas aplicaciones involucran búsqueda de pattern altamente autorepetitivos en texto altamente repetitivo. Sin embargo, el método tiene una virtud principal desde un punto de vista práctico, esto es, el método es muy conveniente para usar un archivo grande leído desde algún dispositivo externo.

ALGORITMO BOYER - MOORE.

Boyer y Moore sugirieron combinar los dos métodos vistos anteriormente para realizar la búsqueda de derecha a izquierda del pattern, escogiendo el más grande de los dos saltos.

Nos presentan el siguiente algoritmo de caracteres mal emparejados, que obviamente no representa mucha ayuda para las cadenas binarias, debido a que sólo existen dos posibilidades para que se presenten caracteres que causen el mal emparejamiento (emparejamiento del pattern con el texto), y ambos tienen la misma probabilidad de estar en el pattern. Sin embargo, los bits pueden ser agrupados a un mismo tiempo, creando caracteres que pueden ser usados exactamente como un todo. Por lo que, si se toman b bits en un tiempo, entonces se necesitara una tabla de saltos con 2**b entradas.

```
FUNCTION Mismarsearch: INTEGER;
VAR
   i, j :INTEGER;
BEGIN
   i := M;  j := M;
   REPEAT
      IF a[i]=p[j] THEN
      BEGIN
         i := i - 1;
         j := j - 1;
      END
      ELSE
      BEGIN
         i := i + M - j + 1;
         j := M;
         SKIP (index(a[i]) - (M-j+1)) ;
       END;
   UNTIL (j<1) OR (i>N);
   Mismarsearch := i + 1;
END;
```

ALGORITMO RABIN KARP

Rabin Karp encontraron un camino fácil para resolver este problema por medio de la función **HASH H(K) = K mod q**, donde **q** (el tamaño de la tabla) es un número primo grande. Su método está basado en el cálculo de la función **HASH** para la posición i en el texto dando este valor para la posición i-1 (Este método viene directamente de la formulación matemática; ver Robert Sedgewick, "Algorithms", String Searching). Despues asumimos que transladamos los **M** caracteres a números empaquetados en una palabra, que podemos tratar como un entero. A esta correspondencia de escribir los caracteres como números se denomina sistema **base-d**, donde **d** es el número de caracteres posibles, el numero correspondiente a[i..i + M-1] es entonces

K = a[i] dM-1 + a[i+1] d**M-2 +...+ a[i+M-1]**

y podemos asumir que se conoce el valor de H(x) = x mod q. Pero cambiar a una posición a la derecha en el texto, simplemente corresponde a reemplazar x por

$$(x - a[i]\,d^{M-1})d + a[i+M].$$

Una propiedad fundamental de la operación, es que puede ser desarrollada en un tiempo, y obtener la misma respuesta. Otro camino sería, si tomamos el residuo resultante cuando dividimos por q (después de cada operación aritmética), entonces, tendremos la misma respuesta, si desarrollamos todas las operaciones aritméticas tomando el residuo cuando dividimos por q.

A continuación se presenta el algoritmo, que aplica el método de Rabin Karp:

```
FUNCTION Rksearch :INTEGER;
CONST
     q = 33554393;  d = 32;
VAR
    h1, h2, dM, i :INTEGER;
BEGIN
   dM := 1;
   FOR i := 1 TO M-1 DO
       dM := (d*dM) mod q;
   h1 := 0;
   FOR i := 1 TO M DO
       h1 := (h1*d + index(pat[i])) mod q;
   h2 := 0;
   FOR i := 1 TO M DO
       h2 := (h2*d + index(a[i])) mod q;
   i := 1;
   WHILE (h1<>h2) and (i<=M-N) DO
   BEGIN
       h1 := (h1 + d*q - index(a[i]*dM)) mod q;
       h2 := (h2*d + index(a[i+M])mod q;
       i := i + 1;
   END;
   Rksearch := i;
END;
```

La función, primero calcula un valor HASH h1 para el pattern, después un valor HASH para los primeros M caracteres del texto, calcula también, el valor de d^{M-1} mod q y lo deposita en la variable dM.

En el caso, de que se encuentre una posición en el texto que tenga el mismo valor HASH que el pattern, se puede realizar una comparación directa del texto con el pattern.

En cuanto al número de pasos que realiza este algoritmo se dice que: teóricamente puede realizar NM pasos en el peor caso, pero en la práctica se ha observado que realiza un poco más de N+M pasos.

BUSQUEDAS MULTIPLES.

Los algoritmos que se presentaron en este capítulo, están orientados a la búsqueda de cadenas específicas en un texto, pero en el caso, de que en un mismo texto se desee la búsqueda de varios patterns, entonces valdría la pena el empleo de algún proceso de búsqueda de cadenas subsecuente que sea eficiente. El hecho de que exista un gran número de búsquedas en el problema de búsqueda de cadenas, se considera un caso especial de problemas generales de búsqueda de cadenas.

5.3 CRIPTOLOGIA.

En los tiempos actuales donde muchas personas tienen acceso a la computación, el pirateo y el uso indebido de ciertos programas, han hecho que los expertos realicen algunos códigos sólo descifrables por ciertos métodos, que han tomado en los últimos años un gran auge.

El área que tiene como fin evitar o disminuir los abusos y delitos referentes al manejo de la información principalmente en el área de cómputo, que son más fáciles de describir que de prevenir, es la **CRIPTOLOGIA**.

La puesta en clave de los datos o encriptación de los mismos es una solución a problemas como robos de paquetes de discos ó el pirataje en la grabación mientras los datos son enviados por medio de líneas de comunicación; ya que la finalidad de los métodos consiste en escribir en clave secreta o de modo oculto la información para que ésta pueda ser manejada por el personal autorizado con un mayor grado de seguridad y con los fines planeados de antemano.

Es también importante tener presente que el codificar información es una forma de eliminar parte de la redundancia del texto original, y que este proceso puede ser desde muy simple hasta muy complejo, resultando obvio que entre más simple sea el esquema más fácil será para un intruso descubrir y descifrar los datos.

5.3.1 Definición.

La **CRIPTOLOGIA** es el estudio de los sistemas para la comunicación secreta, comprende dos grandes campos: **LA CRIPTOGRAFIA y EL CRIPTOANALISIS** (CRIPTOGRAPHY: A PRIMER; Alan G. Korheim).

A) **CRIPTOGRAFIA:** Esta palabra proviene del griego **Kryptos** (oculto) y **Graphein** (escritura); comprende el diseño de sistemas de comunicación secreta, es decir, es la ciencia que altera o encripta un mensaje de una forma original a una que no sea entendible por cualquier individuo, con el fin de incrementar la seguridad en la transmisión y almacenamiento de la información.

Las principales características de la Criptografía son.

- La transformación de un texto a una forma no entendible a través de un procedimiento algorítmico empleando una llave o parámetro secreto y,

- Sus principales usos son la privacidad de los datos, la autenticidad del mensaje y la validación de los usuarios autorizados.

B) **CRIPTOANALISIS:** Es el estudio de todos los posibles caminos para descifrar los sistemas de comunicación secreta; fue considerado un arte hasta el siglo XX, cuando la importancia de la inteligencia militar y diplomática hacia la conducta de la policía extranjera fue evidente. El concepto básico de criptoanálisis fue desarrollado como una rama de las matemáticas aplicadas, ya que emplea herramientas de la teoría de probabilidad y estadística, del algebra lineal, del algebra abstracta y de la teoría de la complejidad para desarrollar su función primordial (descifrar el mensaje).

La criptología mantiene estrechas relaciones con la ciencia de la comunicación y los **algoritmos**, especialmente con los aritméticos y con los de manipulación de cadenas de caracteres; así mismo, tiene estrechas relaciones con las ciencias computacionales, ya que emplea a las computadoras como herramientas.

Actualmente la criptología es una de las áreas más afectadas por las computadoras ya que aún cuando han permitido a los criptógrafos desarrollar máquinas de encriptación más poderosas, han dado lugar a más errores, ya que también les ha proporcionado a los criptoanalistas herramientas mucho más poderosas para descifrar el código, suscitando con ello una increíble explotación de los recursos computacionales.

5.3.2 Breve historia.

Los códigos y las cifras son tan antiguos como la historia de la escritura, sin embargo, nadie sabe con exactitud cuando comenzó la escritura secreta; uno de los primeros ejemplos de este tipo de escritura es una tabla cuneiforme que data de

alrededor del año 1500 A.C. que contiene una fórmula codificada para hacer cerámica de vidrio. Los griegos y los espartanos usaban los códigos allá por el año 475 A.C. y las clases altas de Roma durante el reinado de Julio Cesar. En el período de la Edad Media decrece el interés por este tipo de escritura. Con el Renacimiento italiano vuelve a florecer este arte y durante el reinado de Luis XIV de Francia se emplea un código basado en 587 claves seleccionadas en forma aleatoria para cifrar los mensajes del gobierno.

En el siglo XIX se dan dos factores que incrementan el interés por esta área. El primero fue la publicación del relato de Edgar Allan Poe "El escarabajo de oro", cuento cuya trama emplea mensajes cifrados, que logran excitar la imaginación de muchos lectores. El segundo, y más importante, fue la invención del telégrafo y del Código Morse, código que constituye la primera representación binaria (puntos y rayas) del alfabeto.

Durante la Primera Guerra Mundial muchas naciones construyeron máquinas codificadoras mecánicas que permitían una fácil codificación y decodificación empleando claves complejas y sofisticadas. Antes de emplear mecanismos para decodificar y codificar mensajes, las claves complejas se empleaban con poca frecuencia debido al costo (principalmente tiempo y esfuerzo) requerido para codificar y decodificar el mensaje. Por lo que la mayoría de los textos codificados podían ser descifrados en relativamente poco tiempo. Al emplear las máquinas mecánicas, el arte de descifrar un mensaje se hizo mucho más difícil. Aún así, aunque las modernas computadoras habrían descifrado fácilmente esos códigos, ni siquiera ellas pueden eludir el increíble talento de Herbert Yardley, considerado hoy en día como el maestro mayor en el arte de descifrar códigos, ya que no sólo descifró el código diplomático de los Estados Unidos en 1915, sino que incluso descifró el código diplomático japonés en 1922 empleando tablas de frecuencia del lenguaje nipón.

Durante la Segunda Guerra Mundial, se opta por robar las máquinas codificadoras al enemigo para evitar con ello el tedioso proceso de descifrar el código. De hecho, se considera que la posesión por parte de los aliados de una máquina codificadora alemana, contribuye en gran medida a la culminación de la guerra.

Posteriormente, el uso de métodos criptográficos para una comunicación segura estuvo en su mayor parte orientada a la jurisdicción del gobierno (seguridad militar y diplomática) celosamente guardados y dirigidos por los servidores nacionales de criptología.

Con la llegada de las computadoras la seguridad y existencia de códigos indescifrables ha cobrado un alto grado de importancia, ya no sólo es frecuente la necesidad de mantener secretos ciertos archivos, sino que el mismo acceso a la computadora debe ser controlado y regulado. Para ello se han desarrollado numerosos métodos de cifrado de archivos y el

algoritmo DES (Encriptación Estandar de Datos) aceptado por la Oficina Nacional de Estandares de Estados Unidos ya que aún cuando es un algoritmo difícil de implementar y puede no ser adecuado en todas las ocasiones, se considera seguro contra los intentos de descifrar el mensaje.

5.3.3 Criptosistema.

ENCRIPTAR es el proceso de transformar el texto original (TEXTO CLARO) en un texto encriptado que es un texto formado por símbolos que pertenecen a un alfabeto.

El alfabeto típico consiste de letras mayúsculas y letras minúsculas:

a, b, c, d, e, f, g, h, i, j, k, l, m, n, ñ, o, p, q, r, s, t, u, w, x, y, z

A, B, C, D, E, F, G, H, I, J, K, L, M, N, Ñ, O, P, Q, R, S, T, U, V, W, X, Y, Z.

argumentos numéricos:

0, 1, 2, 3, 4, 5, 6, 7, 8, 9

puntuación, espacios en blanco:

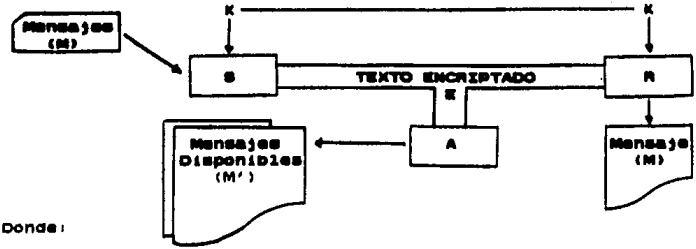
', ", , :, ;, ?, ¡,,, .

y símbolos especiales:

+, -, *, /, =,

Ahora bien un criptosistema es una familia T de transformaciones sobre un texto. Los miembros de la familia son indexados por un parámetro K llamado llave o clave. El espacio de la llave K es el conjunto de valores posibles para la llave que es es una secuencia de letras del alfabeto y la transformación Tk es un algoritmo determinado por K. La efectividad de la seguridad del encriptado depende de mantener la clave en secreto, ya que cualquier usuario puede tener acceso a los archivos codificados pero sólo el usuario autorizado conoce la forma de decodificar su contenido.

La estructura canónica de un criptosistema típico se muestra en el siguiente diagrama:



Donde:

$$E = T_K(M) \Rightarrow M = T_{K^{-1}}(M)$$

E	Mensaje encriptado
T_K	Transformación sobre el mensaje empleando el parámetro K
M	Mensaje original (Texto Claro)
$T_{K^{-1}}$	Transformación inversa sobre el mensaje empleando el parámetro K
M'	Mensajes producidos por el criptoanalista.

El transmisor (S) desea enviar un mensaje (llamado texto claro) a el receptor (R). Para ello transforma el texto claro a una forma secreta empleando un algoritmo criptográfico (T) y algún parámetro (K). Al leer el mensaje el receptor debe tener el algoritmo decodificador ($T_{K^{-1}}$) y los mismos parámetros (K), que emplea para poder transformar el texto encriptado en el texto claro. Usualmente se asume que el texto encriptado es enviado en una línea de comunicación insegura y se encuentra disponible al criptoanalista (A) quien también conoce los métodos de encriptación y desencriptación y se propone recuperar el texto claro desconociendo los parámetros (K).

Si se observa con detenimiento el esquema anterior, nos podemos dar cuenta que para que exista un criptosistema, el transmisor y el receptor deben ponerse de acuerdo en los parámetros de llave, como una regla, entre mayor número de parámetros existan como llave, mayor seguridad existirá; pero esto también representa un incremento en la complejidad de su empleo ya que es necesario recordar la combinación de los parámetros; además, un sistema criptográfico es sólo tan seguro como dignos de confianza sean los que conocen la llave (Robert Sedwick. ALGORITHMS).

Las cuestiones económicas juegan un papel central en cualquier área y si en los criptosistemas existe una motivación económica para construir dispositivos simples, pero efectivos de encriptación y desencriptación, también puede existir una determinada cantidad que ciertas personas estarían dispuestas a pagar al criptoanalista si descifra el mensaje.

5.3.4 Reglas básicas.

Uno de los problemas que siempre debe ser examinado es la eficiencia del método de encriptación ; en otras palabras, el evaluar si el texto encriptado realmente oculta la información y, si existe algún proceso mediante el cual el texto claro pueda ser recuperado sin el conocimiento de la llave; ésto nos permite cuantificar el esfuerzo, el tiempo computacional y el costo requerido por el criptoanalista para descifrar el mensaje. La criptología es pues, como se ha observado un conflicto constante entre dos adversarios:

1) El diseñador de un sistema específico que conoce la familia de transformación y los parámetros; y

2) El oponente o criptoanalista que se dedica a intentar recuperar el texto y/o la llave, pretendiendo con ello derrotar los efectos del encriptamiento.

Las reglas entre estos dos adversarios fueron formuladas en el siglo XIX por Kerckhoffs en su libro: **CRIPTOGRAFIA MILITAR**; ellas son:

K1 El sistema será sino teóricamente indecifrable, indecifrable en la práctica.

Esta regla implica que el método de encriptación empleado debe representar un alto grado de seguridad.

Algunos sistemas encriptados son indecifrables, significando esto que no existe ningún método criptoanalista para recuperar el texto original y/o la llave del texto encriptado.

Un sistema criptográfico en un espacio finito de llaves será efectivamente indecifrable si el tiempo y cálculo requerido para probar llaves es impráctico, así como si la búsqueda de llaves es exhausta.

K2 La decodificación del sistema no presentará inconvenientes para los correspondientes.

Los sistemas de encriptación consisten de dos tipos de información : Información Pública (reglas del algoritmo mediante el cual la información es encriptada) y la Información Privada (clave empleada para encriptar el texto), por lo que los errores presentados por el sistema no se deben derivar de la ignorancia de alguna persona autorizada y además deben poder ser menos preciados o inexistentes; es decir, aún con posibles errores durante la transmisión del mensaje éste debe ser descencriptado y entendido por la persona adecuada.

K3 La llave del método del sistema encriptográfico será fácil de memorizar.

K4 Las operaciones de encriptación deben ser sencillas.

Idealmente sólo se debe requerir de un lápiz y un papel para aplicar el método ya sea al texto claro o al encriptado y obtener el resultado deseado; lo costoso o complicado debe ser evitado.

K5 El texto cifrado debe ser fácil de transmitir y almacenar.

K6 El empleo del sistema no requerirá una lista grande de reglas o esfuerzo mental (minimización del tiempo).

Es importante considerar el factor tiempo en el empleo de los métodos de encriptación y desencriptación de datos ya que muchos usuarios tienen verdadera necesidad de encriptar sus archivos y si esta regla no se cumple optan por no hacerlo.

K7 El texto encriptado no debe ser más grande que el mensaje.

La expansión del mensaje incrementa el tiempo de transmisión y costo, y además, entre más corto esté el mensaje encriptado será más difícil de decodificar.

Es de suma importancia una valorización de los sistemas criptográficos tomando en cuenta estas reglas u objetivos; ya que ello nos permitirá evaluar la eficiencia de los algoritmos o métodos empleados para encriptar un texto.

5.3.5 Aplicaciones.

Actualmente, los sistemas de procesamiento de información juegan varios papeles en las operaciones de negocios tales como:

- El Análisis de datos: computando servicios, facturando, realizando pronósticos, etc.

- La Actualización de archivos: tal como el diseño de datos, bancos de información, listas de clientes, en organizaciones de crédito a consumidores y en las agencias de estados federales.

- El Manejo de Transacciones: tales como las reservaciones de viajes y ahorro de transacciones bancarias.

- Como un componente en la comunicación de sistemas.

- Como un surtido automático de negociaciones, lazos y comunicaciones.

- Como un componente del correo electrónico y facsimile.

Las aplicaciones antes mencionadas son sólo algunos ejemplos de transacciones que manejan datos electrónicamente, dando lugar a que el intercambio de servicios pueda ser ejecutado sin requerir voz o contacto físico de los participantes. La información procesada de esta forma da lugar al fomento de un problema ya existente con un amplio rango económico y de aplicación social: el problema de mantener segura la información.

Las comunicaciones son una de las áreas más vulnerables ya que las interferencias de conversación telefónica y la transmisión digital de datos por microondas, puede ser llevada acabo sin necesidad que las personas involucradas interfieran físicamente; los sistemas de negocios que identifican a sus usuarios con una tarjeta de crédito y un password favorecen también las actividades criminales, y en general, cualquier sistema de procesamiento de información se encuentra propenso a diferentes ataques, por lo que los componentes de un sistema computacional deben encontrarse dispersos, monitoreando y otorgando protección contra una clase limitada de amenazas: La Amenaza Pasiva (la información es copiada y liberada) y La Amenaza Activa (la información es borrada o alterada).

La criptografía es el área encargada de minimizar los problemas de ataque a la información, puesto que proporciona métodos enfocados a protegerla. Mientras no se garantice la absoluta seguridad de la información, ésta debe ser celosamente guardada y ya durante su transmisión se debe crear el sistema de seguridad, es decir, el texto claro debe ser transformado a un texto o formato clave, de tal forma que no pueda ser entendido por cualquiera.

La criptografía se ha aplicado ampliamente a la transmisión de datos, en la que existe un flujo continuo de caracteres a través de una línea. La longitud efectiva del mensaje puede ser sumamente grande y el método de encriptamiento aplicado en forma continua a la secuencia, sujeta solamente a una resincronización a fin de manejar los errores de transmisión.

Por último siempre hay que tener presente que las técnicas criptográficas proporcionan sólo un aspecto de protección contra la destrucción intencional o accidental, ésta no se obtiene mediante la puesta de la información en clave, ya que en realidad los datos se perderían si el texto encriptado se destruye o si la llave o parámetro de encriptación se pierde.

5.3.6 Ventajas de la Criptografía.

El encriptar textos involucra diversas ventajas; la más importante y la cual es el propósito de la criptografía es la de proteger la información, ya que no cualquier persona conoce la llave o clave para decodificar el texto encriptado, proporcionando así, si los métodos son seguros, no sólo un alto nivel de seguridad, sino además herramientas para el control del acceso a datos críticos.

Dado que existen diversos métodos de encriptación, el usuario puede elegir el más apropiado a sus necesidades, y aquel que crea es el más seguro; puede así mismo, utilizar la llave que considere apropiada, y, lograr con todo ello la protección adecuada de la información.

En general la criptografía puede llegar a proporcionar un sin número de ventajas, siempre y cuando se emplee adecuadamente y se consideren todas las contramedidas necesarias para proteger la información.

5.3.7 Desventajas de la criptografía.

La criptografía no es una garantía de absoluta seguridad, además es una actividad costosa, que implica codificar y decodificar archivos, por lo que sólo las personas que cuenten con los medios apropiados podrán emplearla.

La llave empleada al encriptar un texto, es un elemento sumamente importante y si no es segura, será fácil para los criptoanalistas decodificar lo encriptado, además es un elemento que se debe manejar con cuidado para evitar que sea interceptada o robada. Así, la persona encargada de encriptar la información juega un papel arriesgado que puede repercutir en la pérdida de una gran cantidad de dinero si la información es descifrada por los oponentes.

5.3.8 Características deseables en un algoritmo de encriptación.

Lo expuesto hasta esta parte del presente subcapítulo nos permite deducir los requerimientos o características que un buen algoritmo de encriptación debe tener:

- Debe permitir a los usuarios autorizados accesar datos con un mínimo costo o impacto operacional.

- Debe prevenir el acceso de personas no autorizadas, a la lectura, reemplazo y/o modificación de datos.

- Debe ofrecer un alto nivel de seguridad.

- Debe ser entendible y sin ambigüedad, y

- Debe requerir únicamente mantener la llave en secreto.

5.3.9 Métodos básicos de encriptación.

Los métodos de encriptación se encuentran siempre influenciados por la tecnología. Entre los métodos más tradicionales se encuentran dos tipos básicos: **El método por substitución y el método por transposición.** Un encriptamiento por substitución reemplaza un carácter por otro, dejando el mensaje en el mismo orden. Un encriptamiento por transposición modifica el orden de los componentes del mensaje de acuerdo a cierta regla. Estos

tipos de métodos se pueden emplear con cualquier nivel de complejidad e incluso pueden ser mezcladas. La computadora introduce un tercer método de encriptación: **El método por manipilación de bits**, que altera la representación interna de los datos en la computadora mediante algún algoritmo.

Estos tres métodos emplean una llave o clave constituida por una cadena de caracteres necesaria para decodificar el mensaje; es necesario tener presente que la llave por sí sola no basta para decodificar un texto encriptado, es necesario conocer también el algoritmo de encriptación. La llave personaliza un mensaje codificado, de forma tal que sólo aquellas personas que la conocen, pueden decodificarlo.

A) METODOS POR SUBSTITUCION.

El método por substitución más simple lleva acabo un desplazamiento del alfabeto en un cierto número de posiciones; es decir, si una letra del texto claro es la **n-ésima** letra del alfabeto, esta es substituida por la letra que se encuentra en la **(n+k)-ésima** posición del alfabeto, donde k es un entero fijo, conocida con el nombre de cifra **Caesar** (En honor a Caesar que empleaba **k=3**). Por ejemplo, si cada letra se desplaza 3 posiciones **(k=3)**, entonces

ALFABETO : a b c d e f g h i j k l m n ñ o p q r s t u v w x y z

ALFABETO DESPLAZADO : d e f g h i j k l m n ñ o p q r s t u v w x y z a b c

observe que las letras desplazadas del principio de la línea, se han anexado al final.

Si se desea codificar un mensaje empleando este método simplemente se sustituye el alfabeto real por el alfabeto desplazado.

Sea el texto claro:

A T A Q U E A L A M A N E C E R

El texto encriptado quedaría como:

D W D S X H D O D P D Q H F H U

Este método en realidad es muy fácil ya que el criptoanalista sólo tiene que adivinar el valor de k; después de todo sólo existen 26 desplazamientos, y es fácil probarlos todos en una cantidad relativamente pequeña de tiempo.

Un segundo fallo de este método simple de substitución es que respeta los espacios en blanco, una ligera mejora sería codificarlos (En realidad se deberían codificar todos los signos de puntuación).

ALGORITMO CODIFICADO.

```
# include "ctype.h"
# include "studio.h"
main (arg,argv)      /* cifrado por desplazamiento */
int argc;
char *argv[];
{
 if (argc!=5)
 {
  printf("uso: entrada salida codifica/decodifica
        desplaza\n");
     exit();
 }
  if (!isalpha (*argv 4))

   printf("la letra inicial debe ser un carácter alfabético
         n");
     exit(),

  if toupper(*argv[3]=='c')codifica(argv[1],argv[2],*argv[4]);
  else
      decodifica(argv[1],argv[2],*argv[4]);
}
codifica (entrada,salida,inicio)
char *entrada, *salida;
char inicio,
{
  int car;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"));==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  inicio= tolower (inicio);
  inicio= inicio-'a';
  do
  {
   car = tolower(getc(pf1));
   if (car==EOF) break;
   if (isalpha(car))
   {
     car+=inicio;
```

```
        if(car>'z') car-=26;
        }
        putc(car,pf2);
      }while (1);
    fclose(pf1); fclose(pf2);
    }
    decodifica (entrada,salida,inicio)
    char *entrada, *salida;
    char inicio;
    {
      int car;
      FILE *pf1, *pf2;
      if ((pf2=fopen (entrada,"w"))==0)
      {
       printf ("no se puede abrir el archivo de entrada\n");
       exit();
      }
      inicio= tolower (inicio);
      inicio= inicio-'a';
      do
      {
       car = tolower(getc(pf1));
       if (car==EOF) break;
       if (isalpha(car))
       {
         car-=inicio;
         if(car>'z') car+=26;

         putc(car,pf2);
      }while (1);
    fclose(pf1); fclose(pf2);
    }
```

Un mejor método de substitución es aquel que emplea una tabla general que define la substitución que va a ser realizada, es decir, para cada letra del texto claro, la tabla indica que letra se coloca en el texto clave. Por ejemplo, si la tabla tiene la siguiente correspondencia:

```
A B C D E F G H I J K L M N O P Q R S T U V W X Y Z
T H E   Q U I C K B R O W N F X J M P D V R A Z Y G L.
```

Entonces el texto claro

```
A T A Q U E   A L   A M A N E C E R
```

Quedaría encriptado como

```
T D T J Y Q L T Q L T W T N Q E Q M
```

Qué es mucho más difícil de descifrar ya que el analista tendrá que probar un número grande de tablas para estar seguro de leer el mensaje (cerca de 27! tablas > 10^{28} tablas); sin embargo aún este método es fácil de descifrar por la inherente frecuencia

de las letras en un lenguaje, y el criptoanalista puede tener un buen inicio en su tarea de descifrar el mensaje si centra su atención en las letras más frecuentes del texto encriptado y asume que estas pueden ser reemplazadas por las letras más frecuentes del lenguaje con el cual está tratando; por ejemplo, si se cuenta con una tabla de frecuencias del lenguaje español en la que se registra la información estadística del empleo de cada letra del alfabeto, se puede deducir fácilmente que la "Q" representa la letra "E" que es la letra más común del lenguaje español. De esta forma se puede seguir descifrando el resto del mensaje, más aún, cuanto mayor sea el texto codificado, más fácil será descifrarlo, empleando las tablas de frecuencia.

ALGORITMO CODIFICADO.

```
# include "ctype.h"
# include "studio.h"
char sub 28       = "qazwsxedcrfvtgbyhnujmikolp";
char alfabeto 28  = "abcdefghijklmnopqrstuvwxyz";
main (arg,argv)      /* método por sustitución */
int argc;
char *argv[3];
{
 if (argc!=4)
 {
  printf("uso: entrada salida codifica/decodifica
          desplaza\n");
      exit();
 }
  if (toupper (*argv[2])=='C') codifica(argv[1],argv[3]);
  else
       decodifica (argv[1],argv[3]);
}
codifica (entrada,salida)
char *entrada, *salida;
{
  int car;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  do
  {
   car = tolower(getc(pf1));
   if (car==EOF) break;
   if (isalpha(car))   car=' ')
  {
```

```
        car=sub[buscar(alfabeto,car)];
        putc(car,pf2);
      } while (1);
    fclose(pf1); fclose(pf2);
    }
    decodifica (entrada,salida)
    char *entrada, *salida;
    char inicio;
    {
      int car;
      FILE *pf1, *pf2;
      if ((pf1=fopen (entrada,"r"))==0)
      {
       printf ("no se puede abrir el archivo de entrada\n");
       exit();
      }
      if ((pf2=fopen (salida,"w"))==0)
      {
       printf ("no se puede abrir el archivo de salida\n");
       exit();
      }
      do
      {
       car = tolower(getc(pf1));
       if (car==EOF) break;
       if (isalpha(car) || car==' ')
       {
         car=buscar(sub,car) ;
         }
         putc(car,pf2);
      } while (1);
    fclose(pf1); fclose(pf2);
    }
    buscar (s, car)
    char *s;
    char car;
    {
      register int t;
      for (t=0); t<28; t++) if car==s[t] return t;
    }
```

Un camino para complicarle al analista la tarea de descifrar el mensaje es realizando una extensión de K conocida con el nombre de cifra vigenere, que es una llave pequeña repetida que se emplea para determinar el valor de K para cada letra. En cada paso del proceso de encriptación, la letra clave indexada se suma a la letra indexada del texto claro para determinar la letra indexada del texto cifrado. Por ejemplo, sea la cifra vigenere o llave: **ABC**, entonces

Llave	**A B C A B C A B C A B C A B C A B C**
Texto claro	**A T A Q U E A L A M A N E C E R**

Texto encriptado C V D R W H A C O A C P B P H D O U

Donde por ejemplo, la primer letra del texto encriptado es la segunda letra del alfabeto debido a que la letra correspondiente del texto claro es A (la primer letra del alfabeto) y su correspondiente letra de la llave es A (también la primer letra).

Este método de substitución empleando la cifra vigenere es obviamente mucho mas complicada ya que usa tablas generales para cada letra del texto claro, también es obvio que entre más larga sea la clave, el texto encriptado será más difícil de descifrar. De hecho, si la llave es tan larga como el texto claro estamos ante un método de substitución conocido como **METODO DE VERNAM** más comúnmente llamado **"Relleno en un tiempo"**. Este es probablemente el sistema de criptoanálisis más conocido y es empleado en aplicaciones vitales. Dado que cada letra es empleada sólo una vez, el criptoanalista no tiene otro camino que probar todas las posibles claves para el mensaje y obviamente con muy poca esperanza de éxito; pero por lo mismo que la letra clave se emplea sólo una vez, este método es usado para mensajes relativamente cortos, que no sean enviadas muy frecuentemente.

Para aumentar aún más la complejidad a la tarea de descifrar mensajes se pueden emplear los **METODOS POR SUBSTITUCION MULTIPLE** en los cuales la misma letra del texto claro no tiene necesariamente que tener la misma letra en el texto codificado, ésto puede ser llevado acabo incluyendo un segundo alfabeto aleatorio cambiando de un alfabeto a otro cada vez que se repita una letra.

Por ejemplo,

Sean los alfabetos:

ALFABETO : A B C D E F G H I J K L M N O P Q R S T U
V W X Y Z

ALFABETO ALEATORIO 1 : Q A Z W S X E D C R F V T G B Y H N U J M
I K O L P

ALFABETO ALEATORIO 2 : P O I U Y T R E W Q A S D F G H J K L M
N B V C X Z

Texto claro : A T A Q U E A L A M A N E C E R
Texto encriptado : Q J Q H M U Z P A Z Q S P D U Z S N

El proceso para encriptar el texto es el siguiente:

1) Se emplea el alfabeto aleatorio 1.

2) Se toma un carácter del texto claro.

3) Se substituye por el carácter correspondiente del alfabeto aleatorio considerado.

4) Se incrementa en 1 el contador del caracter encontrado.

5) Si el contador es igual a 2 se cambia de alfabeto aleatorio y el contador se inicializa (valor cero).

6) Si existe mas caracteres a encriptar en el texto claro ir a 2.

7) FIN.

ALGORITMO CODIFICADO.

```
# include "ctype.h"
# include "studio.h"
char sub1[28]       = "QAZWSXEDCRFVTGBYHNUJM IKOLP";
char sub2[28]       = "POI UVTREWQASDFGHJKLMNBVCXZ";
char alfabeto[28] = "ABCDEFGHIJKLMNOPQRSTUVWXYZ ";
main (arg,argv)      /* método por sustitución multiple */
int argc;
char *argv[3];
 }
 register int t;
 for (t=0; t<27; ++t) cont t =0;
 if (argc!=4)
 {
  printf("uso: entrada salida codifica/decodifica
            desplaza\n");
      exit();
  }
  if (toupper (*argv[3])=='E') codifica(argv[1],argv[2]);
  else
      decodifica (argv[1],argv[2]);
}
codifica (entrada,salida)
char *entrada, *salida;
{
  int car, cambio, t;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  cambio =1;
  do
  {
   car = tolower(getc(pf1));
   if (car==EOF) break;
   t = indice (car);
   cont t ++;
```

```
    if (isalpha(car)) || car==' ')
    {
      if (cambio)
            car=sub2[buscar(alfabeto,car)];
      else
        car=sub2[buscar(alfabeto,car)];
      }
      putc (car,pf2);
      if (cont_t ==2)
      {
        cambio =! cambio;
        cont_t  = 0;
      }
  } while (1);
fclose(pf1); fclose(pf2);
}
decodifica (entrada,salida)
char *entrada, *salida;
{
  int car,cambio;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  cambio = 1;
  do
  {
   car = tolower(getc(pf1));
   if (car==EOF) break;
   if (isalpha(car) || car==' ')
   {
     if (cambio)
      {
        car=alfabeto[buscar(sub,car)];
        cont_indice[car]  ++;
      }
   }
   putc(car,pf2);
   if (cont_indice[car]==2)
   {
    cambio =! cambio;
    cont_indice[car]=0;
   }
  } while (1);
fclose(pf1); fclose(pf2);
}
buscar (s, car)
```

```
char *s;
char car;
{
  register int t;
  for (t=0); t<28; t++) if car==s[t] return t,
}
indice (car)
char car;
{
 car = tolower(car);
 if (isalpha(car)) return car-'a' ;
 else
     return 26;
}
```

En general el uso del cifrado por sustitución múltiple hace más difícil descrifrar el código mediante tablas de frecuencia, ya que el método presentado en esta sección puede ser generalizado a un método que emplee varios alfabetos aleatorios y rutinas más complejas de cambio entre los alfabetos, que pueden hacer posible que todas las letras del alfabeto presenten la misma frecuencia; en este caso el empleo de tablas de frecuencia no sería útil.

B) METODOS POR TRANSPOSICION.

La esencia de los métodos de encriptación por transposición es intercambiar el orden de los caracteres que componen el texto claro de acuerdo a una regla.

Uno de los primeros códigos de transposición conocido fue diseñado por los espartacos sobre el año 475 a.c. los cuales empleaban un aparato llamado **"cuento de cielo"** compuesto básicamente de una cinta que se enrollaba alrededor de un cilindro y sobre la cual se encriptaba el mensaje transversalmente, para posteriormente desenrollarla y enviarla al receptor quién también disponía de un cilindro de igual tamaño. Teóricamente resulta imposible leer las cintas sin el cilindro, ya que las letras se encuentran en desorden; sin embargo en la práctica este método ha dejado de emplearse, ya que se puede probar con distintos cilindros, hasta dar con el mensaje.

Se puede crear una versión actual de "cuento de cielo" situando el mensaje en un arreglo bidimensional y escribiéndolo en una dirección distinta; por ejemplo

ATAQUE AL AMANECER

```
A T A Q U E
  A L   A M
A N E C E R
```

Si se escribe el arreglo por columnas

donde ... indica el cambio de columna.

CODIFICACION DEL METODO.

```
#include "ctype.h"
#include "studio.h"
union mensaje
{
  char s[100];
  char s2[100][5];
  cielo;
main (arg,argv)      /* ejemplo del metodo por transposición */
int argc;
char *argv[];
{
 register int t;
 for (t=0; t<100; ++t) cielo.s[t]=0;
 if (argc!=4)
 {
  printf("uso: entrada salida codifica/decodifica
          desplaza\n");
     exit();
 }
 if (toupper (*argv[3])== E) codifica(argv[1],argv[2]);
 else
      decodifica (argv 1,argv 2);
}
codifica (entrada,salida)
char *entrada, *salida;
{
  int car, cambio, t;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  for (t=0; t<100;++t)
  {
   cielo.s[t]=getc(pf1);
   if (cielo.s[t]==EOF)
   {
     cielo.s[t] = 0;
     break;
   }
  {
```

```
    for (t=0; t<5;++t)
       for (t2=0; t2<20; ++t2)
           putc(cielo.s2[t2][t],pf2);
        fclose(pf1); fclose(pf2);
}
decodifica (entrada,salida)
char *entrada, *salida;
{
   int t, t2;
   FILE *pf1, *pf2;
   if ((pf1=fopen (entrada,"r"))==0)
   {
    printf ("no se puede abrir el archivo de entrada  n");
    exit();
   }
   if ((pf2=fopen (salida,"w"))==0)
   {
    printf ("no se puede abrir el archivo de salida  n")
    exit();
   }
   for (t=0; t<5;++t)
      for (t2=0; t2<20; ++t2)
          if (cielo.s2[t2][t] = getc(pf1)) == EOF) break;
for (t=0; t<100;++t)
   putc (cielo.s[t],pf2);
fclose(pf1), fclose(pf2);
}
```

Por supuesto, existen otros métodos de obtener mensajes transpuestos, tantos como la imaginación del diseñador sea capaz de crear; siempre y cuando el intercambio de los caracteres se realice en base a cierta regla. Los métodos que se ajustan a las computadoras, son obviamente aquellos que intercambian las letras del texto claro de acuerdo a un algoritmo que puede ser codificado.

Por si mismos, los métodos por transposición pueden crear accidentalmente pistas para el criptoanalista en su proceso; ya que en el mensaje encriptado pueden aparecer palabras muy sugestivas que pueden ser fácilmente captadas por criptoanalistas muy hábiles o astutos.

C. METODOS POR MANIPULACION DE BITS.

Dado que la mayoría de las computadoras emplean el código binario para almacenar los archivos, se ha generado un nuevo método de encriptación: el método de manipulación de bits; este método es considerado por muchas personas como una variación de los métodos por sustitución, pero realmente los conceptos que envuelven un método por manipulación de bits difieren significativamente de lo que se puede considerar un método de encriptación.

Los métodos de manipulación de bits están exclusivamente pensados para ser ejecutados por una computadora, ya que las funciones requeridas por el método son fácilmente ejecutadas por el sistema.

Además, el texto encriptado empleando un método por manipulación de bits tiende a parecer completamente ilegible, lo que incrementa el grado de seguridad, ya que aparentemente el conjunto de datos parecen archivos no empleados o desechados, confundiendo así, a cualquier persona que desee accesar el archivo.

En general, los métodos por manipulación de bits se aplican sólo a los archivos manejados por una computadora; además es importante tener presente que no pueden ser impresos, ya que el proceso de encriptación tiende a crear caracteres no imprimibles; por lo que se debe asumir que cualquier archivo codificado por manipulación de bits ha de permanecer en la computadora.

La encriptación por manipulación de bits convierten el texto claro a texto cifrado, alternando los patrones reales de bits de cada carácter mediante el empleo de uno o más de los operadores lógicos.

El cifrado por manipulación de bits más simple y menos seguro emplea sólo el operador (operador de complemento a uno) el cual, invierte cada bit del byte, es decir, si un bit tiene el valor 1 lo convierte en 0 y si tiene el valor 0 lo convierte en 1; por lo tanto un byte complementado dos veces es el mismo.

Existen dos problemas inherentes a este esquema tan sencillo. El primero es que el programa de encriptación no emplea una clave para decodificar, por lo que cualquier persona con acceso al programa puede decodificar el archivo. El segundo y quizá el más importante, es que el método puede ser fácilmente descubierto por cualquier programador con experiencia.

CODIFICACION DEL METODO.

```
# include "studio.h"
main (arg,argv)   /* ejemplo del método por complemento a 1*/
int argc;
char *argvc];
{
if (argc!=4)
{
  printf("uso: entrada salida codifica/decodifica
          desplaza\n");
     exit();
  }
  if (toupper (*argvc3D=='E') codifica(argvc1],argvc2]);
  else
      decodifica (argvc1],argvc2]);
```

```
codifica (entrada,salida)
char *entrada, *salida;
{
  int car;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada \n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida \n");
   exit();
  }
  do
  {
   car = getc(pf1);
    if (car == EOF) break;
    car = ~car;
    if (car==EOF) car ++;
     putc (car,pf2);
  } while (1);
  fclose(pf1); fclose (pf2);
}
decodifica (entrada,salida)
char *entrada, *salida;
{
  int car;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada \n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida \n");
   exit();
  }
  do
  {
   car = getc(pf1);
    if (car == EOF) break;
    car = ~car;
    if (car==EOF) car --;
     putc (car,pf2);
  } while (1);
  fclose(pf1); fclose(pf2);
```

Un mejor método de manipulación de bits emplea el operador XOR, cuya tabla de verdad es la siguiente:

XOR	0	1
0	0	1
1	1	0

La tabla nos indica que la operación XOR es verdadera si y sólo si uno de los operandos es verdadero y el otro falso. Esto da al operador una propiedad única : si se aplica la operación XOR a un byte de texto claro y a otro denominado llave y luego se aplica nuevamente la operación XOR al resultado anterior y a la misma llave, se obtiene el byte original, tal y como se muestra en el siguiente ejemplo:

```
        11011001        (carácter del texto claro)
XOR     01010011        (llave)
        --------
        10001010        (carácter encriptado)
XOR     01010011        (llave)
        --------
        11011001        (carácter del texto claro)
```

Cuando se codifica un archivo, empleando este método se resuelven los dos problemas que implica el método de complemento a 1. En primer lugar, dado que se emplea una clave, el programa de codificación por sí mismo no puede decoficar un archivo; en segundo lugar, cada clave hace que el archivo sea único, lo cual ya no resulta tan evidente para el criptoanalista.

La clave no tiene necesariamente que ser de un byte, se puede emplear una clave de varios caracteres que pueden ser alternados a lo largo del archivo.

CODIFICACION DEL METODO.

```
# include "studio.h"
main (arg,argv)      /* ejemplo del método empleando XOR */
int argc;
char *argv  ;
{
if (argc!=5)
{
  printf("uso: entrada salida codifica/decodifica clave\n");
     exit();
  }
  if (toupper (*argv 3)== E) codifica(argvt1),argvc2),*argv
     t4));
  else
      decodifica (argv c1),argvc2),*argvc4));
}
codifica (entrada,salida,clave)
char *entrada, *salida;
char *clave;
{
```

```
    int car;
    FILE *pf1, *pf2;
    if ((pf1=fopen (entrada,"r"))==0)
    {
     printf ("no se puede abrir el archivo de entrada\n");
     exit();
    }
    if ((pf2=fopen (salida,"w"))==0)
    {
     printf ("no se puede abrir el archivo de salida\n");
     exit();
    }
    do
    {
     car = getc(pf1);
      if (car == EOF) break;
     car = car^clave;
      if (car==EOF) car ++;
       putc (car,pf2);
     } while (1);
    fclose(pf1); fclose (pf2);
}
decodifica (entrada,salida,clave)
char *entrada, *salida;
char clave;
{
  int car;
  FILE *pf1, *pf2;
  if ((pf1=fopen (entrada,"r"))==0)
  {
   printf ("no se puede abrir el archivo de entrada\n");
   exit();
  }
  if ((pf2=fopen (salida,"w"))==0)
  {
   printf ("no se puede abrir el archivo de salida\n");
   exit();
  }
  do
  {
   car = getc(pf1);
    if (car == EOF) break;
   car = car^clave;
    if (car==EOF+1) car --;
     putc (car,pf2);
  } while (1);
 fclose(pf1); fclose(pf2);
}
```

Inicialmente estos metodos resultaron sorprendentes pero actualmente muchos sistemas criptográficos que los emplean tienen la propiedad de que el criptoanalista puede descubrir la llave si conoce o tiene la idea de alguna selección del texto claro.

5.3.10. ¿Qué es el DES?

El **DES**, es el estándar de criptografía de datos (D encripryton Standar), fue publicado por la oficina nacional de estándares de los Estados Unidos, como un estándar de procesamiento de información federal, en Enero de 1977; fue aprobadoinicialmente para el uso exclusivo de la Agencia Federal en Aplicaciones de Cómputo no Clasificadas; sin embargo actualmente está siendo implementado en chip s por varios fabricantes.

El **DES** es un sistema simétrico de encriptación, que especifica un algoritmo que se implanta y se emplea en dispositivos de hardware electrónico para la protección criptográfica de los datos de la computadora.

Los datos almacenados por la computadora (en forma binaria) se protegen criptográficamente si se emplea el algoritmo **DES** conjuntamente con una clave; esta clave consta de 64 dígitos binarios, de los cuales 56 son seleccionados en forma aleatoria, se emplean por el algoritmo para desarrollar el proceso de encriptación y los 8 bits restantes son utilizados para detectar posibles errores durante el proceso de transmisión de datos.

La selección de la clave en forma aleatoria provoca que ésta sea un tanto exclusiva de cada proceso, lo que otorga seguridad criptográfica en los datos.

5.3.11. Criptoanálisis (Decodificación).

El criptoanálisis es el área encargada de transformar un mensaje encriptado a un mensaje claro sin conocimiento de la clave, para ello se basa principalmente en los métodos de prueba y error. Actualmente con el uso de la computadora, los métodos de encriptación relativamente sencillos se pueden descifrar fácilmente mediante un proceso exhaustivo de prueba y error; sin embargo, los textos que emplean métodos y claves complejas o bien no pueden ser descifrados, o requieren recursos de los que no se dispone normalmente.

La facilidad con la cual es posible decodificar un mensaje depende de cierto número de factores:

1. Conocer el método empleado de encriptación.
2. Conocer parte del texto claro.
3. Saber si se dispone de textos encriptados, que se suponen emplean la misma llave o clave.
4. Conocer si se dispone de textos claros que coincidan en cierto grado con el texto que se está estudiando.

Una herramienta básica que se emplea en la decodificación de un texto encriptado consiste en generar una tabla. Si el método es conocido y el número de llaves limitado, una tabla puede ser una lista que contenga todas las posible formas de desencriptarlo y probablemente mediante su revisión resalte el mensaje en forma clara.

Si se considera que el mensaje presenta un número grande de posibilidades para que pueda ser decodificado, pero aún si se trata de un número finito razonable, una selección heurística de texto decodificado, puede reducir el volumen de la tabla que se va a listar, eliminando aquellas que no muestren un patrón razonable.

Si se conoce parte del lenguaje fuente, los métodos de substitución pueden atacarse empleando estadísticas de frecuencia de letras conocidas. En general una substitución simple dará directamente el texto claro, si la letra más frecuente del texto claro es reemplazada por la letra de mayor frecuencia en la tabla y así sucesivamente.

Un texto en lenguaje de programación también será relativamente fácil de descifrar para un criptólogo diestro, ya que la sintaxis es limitada y no es necesario reconstruir los nombres originales de las variables o constantes empleadas.

Para decodificar los textos encriptados por algún método de transposición o manipulación de bits, se emplean secuencias de dígitos aleatorios obtenidos a partir de un generador de números aleatorios ; sin embargo estos métodos son más difíciles de descifrar. Las llaves largas proporcionan mayor protección que las llaves cortas, pero estas últimas no pueden emplearse con efectividad en la recuperación de texto claro, ya que para cada bloque o registro almacenado en un archivo, la llave debe volver a colocarse en su punto inicial para permitir la sincronización de la llave en la recuperación aleatoria. Debido a esto, la llave siempre está limitada a la longitud del registro.

Cuando la llave cambia, es necesario volver a empezar el proceso de criptoanálisis, pero el cambiar las llaves con frecuencia implica la transmisión constante de llaves entre el encargado de codificar el texto y la persona indicada para decodificarlo, aumentando con ello la posibilidad de que las llaves sean interceptadas o robadas.

La determinación de la longitud de la llave es importante para el criptógrafo que necesite reducir el espacio de búsqueda. Si se dispone de suficiente texto encriptado en relación con la longitud de llave, las pruebas estadísticas con diferentes agrupamientos del texto encriptado pueden proporcionar la llave empleada al realizar el proceso de encriptación.

Dado que se requiere un considerable esfuerzo para decodificar texto no trivial, existe siempre un gran esfuerzo por obtener información auxiliar extra y si el criptoanalista logra obtener un segmento coincidente de texto encriptado y texto claro, la tarea es mucho más sencilla, sobre todo si este segmento es de longitud mayor al de la llave empleada.

5.3.12. Contramedidas de protección.

En la sección anterior mencionamos algunas de las herramientas que un criptoanalista bien equipado debe conocer para facilitar su tarea (decodificar el texto); por lo que es necesario analizar algunas contramedidas de protección que se pueden tomar en cuenta para defenderse de los ataques en los puntos vulnerables del método empleado para encriptar.

Para minimizar la efectividad del empleo de las tablas, es necesario asegurarse de que el número de posibles llaves sea muy alto. El empleo de la combinación de los métodos básicos puede dar como resultado el producto o la suma de las posibilidades individuales.

El reducir la redundancia del texto ayuda a disfrazar el mensaje. En general, si se omitieran los espacios en blanco, signos de puntuación y sólo se emplearan letras mayúsculas el análisis del texto encriptado aumentaría en complejidad.

A fin de hacer aún más difícil la determinación de la llave puede utilizarse más de una llave, la selección entre un número determinado de llaves puede debilitar el análisis estadístico aplicado a grandes cantidades del texto encriptado; también el variar la longitud de la llave provocaría grandes dificultades a los algoritmos que un intruso pueda utilizar en sus intentos iniciales de conocer el texto encriptado.

Para evitar la liberación de texto claro y texto codificado coincidente, se deben considerar los problemas de acceso al sistema por parte de intrusos que deseen obtener texto que coincida, manejando los métodos existentes de identificación y bitácoras de acceso; ya que dado que la identificación toma tiempo, existen ciertas posibilidades de sorprender al intruso cuando se haya detectado un patrón suspendido o no usado de acceso durante una revisión periódica de la bitácora de acceso.

5.3.13. Criptosistemas de llaves públicas.

En aplicaciones comerciales, tales como la transferencia de fondos electrónicos y correo computarizado, el problema de la distribución de llaves es mucho más complicado que las aplicaciones tradicionales de la criptografía. El prospecto de proveer un número grande de llaves, por ejemplo a cada ciudadano, manteniendo o garantizando seguridad y costo efectivo, reprime el desarrollo de tales sistemas. Los métodos que han sido desarrollados recientemente, prometen eliminar el problema de la distribución de llaves completamente. Tales sistemas denominados sistemas de llaves públicas serán probablemente empleados en forma general en el futuro. La idea de una llave pública es emplear un **"directorio telefónico de claves encriptadas"**. Cada una de esas claves (denotadas por p) es conocida públicamente y podría aparecer cerca del número telefónico; así mismo, cada persona posee también una llave secreta con la cual decodifica el

mensaje; ésta llave secreta (denotada por S) es conocida sólo por el dueño. Al transmitir un mensaje M, el transmisor emplea la llave pública del receptor, y la utiliza para encriptar el mensaje, luego de realizar esto, transmite el mensaje denotado por C = P(M) (texto encriptado). El receptor emplea su llave privada de desencriptación para poder leer el mensaje.

Este sistema debe cumplir al menos las siguientes propiedades:

1. S((P(M)) = M para cada mensaje M.
2. Todo par (s,p) es distinto.
3. Derivar s de P es tan difícil como leer C.
4. Ambos S y P son fáciles de calcular.

La primera es una propiedad fundamental de la criptografía; las dos siguientes proveen seguridad y la cuarta permite que el sistema tenga un uso factible.

Un esquema inicial de este sistema fue dado a conocer por W. Diffie y M. Hellman en 1976, pero su método no satisfacía las propiedades antes citadas. Más tarde R. Rivest, A. Shamir y L. Adleman propusieron un esquema conocido con el nombre de **Criptosistema de llave pública**: RSA, que está basado en algoritmos aritméticos que desarrollan enteros muy grandes. La llave en encriptación es el par entero (N,P) y la llave de desencriptación S el par entero (N,S) donde S se mantenido en secreto.

Esos números deben ser muy grandes (típicamente, N podría ser de aproximadamente 200 dígitos y P y S podrían tener 100 dígitos).

Los métodos de encriptación y desencriptación son simples: primero el mensaje es particionado en números menores que N (Por ejemplo tomando log N bits en un paso desde la cadena binaria correspondiente a el carácter codificado del mensaje; entonces esos números son restados en forma independiente a un módulo N poderoso, el cual encripta el pedazo del mensaje M, calculando C=P(M)=(M**P) MOD N y desencripta el pedazo de texto encriptado C, calculando M = S(C) = C**S MOD N. Este cálculo se desarrolla fácil y rápidamente (No más de 2log N operaciones son requeridas para cada sección del mensaje, por lo que el número total de operaciones (100! número de dígitos) requeridos es lineal con respecto al número de bits en ese mensaje.

La propiedad cuatro también se cumple dado que N y S son fácil de calcular y la propiedad dos puede ser satisfecha si las criptovariables N, P y S se seleccionan de tal forma que satisfagan las propiedades uno y tres. Para ello es necesario generar tres números primos muy grandes en forma aleatoria (aproximadamente de tres dígitos); el mas grande puede ser S y se puede llamar a los otros dos X y Y. Entonces N se escoge del producto de X y Y, y P se escoge tal que PS mod (x-1)(y-1)=1. Es posible afirmar que con N, P y S escogidas por este camino, se tiene M**PS MOD N = M para todos los mensajes M.

S es difícil de calcular si se conoce P(Y,N), ya que encontrar S partiendo de P requiere del conocimiento de X y Y, cuyo producto es igual a N, lo cual implica la necesidad de un factor N; pero factorizar N es muy difícil, el mejor algoritmo de factorización conocido actualmente puede tardarse millones de años al factorizar un número de 200 dígitos (Robert Sedgewick. Análisis de Algoritmos).

Un hecho atractivo del sistema RSA es que los cálculos desarrollados para cada usuario que se encuentra subscrito al sistema al encriptar o desencriptar mensajes involucran sólo un procedimiento simple de exponenciación. Además este sistema de llaves públicas es muy atractivo para una comunicación segura, especialmente en sistemas computacionales y redes locales.

El método RSA presenta las siguientes limitantes: el procedimiento de exponenciación es un proceso fácil pero costoso para la criptografía estándar, y lo peor, existe una alta posibilidad de demora al leer el mensaje encriptado; pero es un método que podría resistir criptoanalistas muy competentes.

Diversos métodos han sido sugeridos para implementar criptosistemas de llaves públicas. Algunos de los más interesantes se encuentran relacionados con problemas muy importantes y difíciles de resolver; presentan propiedades muy interesantes y el éxito de ataque podría implicar perspicacia para intentar solucionar algunos problemas no resueltos (tal como la factorización para el método RSA).

La criptografía relacionada con los temas fundamentales de las ciencias computacionales y el empleo potencial expandido de los criptosistemas de llaves públicas, son actualmente un área activa en la búsqueda actual de nuevos avances tecnológicos y científicos.

5.4 COMPACTACION DE DATOS.

La necesidad primordial en un mundo industrializado es la información. En la actualidad, la implementación del manejo de información a través de sistemas de cómputo en casi todas las áreas de la vida, ha creado la necesidad de manipular grandes cantidades de información que requieren grandes cantidades de espacio de almacenamiento (memoria principal y auxiliar), esta consecuencia tiene varias desventajas, ya que el hecho de que cierta información esté almacenada en un espacio grande de memoria implica dificultad al querer accesarla y un incremento en el tiempo de acceso requerido, lo cual es bastante costoso; por otra parte, el empleo de grandes espacios de almacenamiento nos lleva a usar memoria auxiliar (dispositivos de almacenamiento secundario), que aparte de ser costoso ocasiona también, que el

acceso a la información sea lenta. Además de todo lo anterior, también proporciona cierta inseguridad en el manejo y almacenamiento de la información, puesto que una falla en el sistema o un error humano puede dañar la información.

Por consiguiente se han desarrollado algunas técnicas que hacen posible almacenar la información en un espacio menor al que es requerido realmente, estas técnicas son conocidas como **Técnicas de compactación o compresión de datos.**

El concepto de compactación o compresión de datos es muy cercano al de la criptografía. La compactación de datos se refiere a almacenar la información en un área menor a la que requiere realmente.

La compactación de datos se usa normalmente en sistemas informáticos para aumentar la capacidad de almacenamiento de los distintos tipos de dispositivos (reduciendo las necesidades de almacenamiento de los usuarios). La compactación también se aplica para la transmisión de datos, con el fin de duplicar y hasta triplicar muchas veces, la velocidad efectiva de la transmisión; ahorrar tiempo de transferencia (por ejemplo, como sucede en las líneas telefónicas), y proporcionar cierto grado de seguridad a la información.

Por otra parte, la compactación de datos permite lograr grandes economías, debido a que la mayoría de los archivos pueden reducirse por lo menos a la mitad de su volumen original y se conocen casos donde la reducción del tamaño ha sido entre el 80% y el 90%.

Además las técnicas de compactación tienen aplicación en todos los archivos de datos, pero su valor se destaca especialmente en los grandes archivos de escasos movimientos.

Existen varios métodos de compactación de datos, cuyos esquemas son distintos, pero tienen como principal objetivo reducir el área de memoria empleada en el almacenamiento de la información lo cual repercute principalmente en el costo y en la eficiencia de los algoritmos que manipulan la información contenida en los archivos.

5.4.1 Clasificación de los métodos de compactación de datos.

Los métodos de compactación de datos se clasifican en dos categorías:

- La primera es donde se encuentran los métodos dependientes de la estructura de los registros o del contenido de los datos, por lo que deben ser diseñados para cada aplicación particular.

- La segunda es donde están los métodos de uso más general, por lo que pueden ser incluidos en paquetes de software de uso general, en el hardware o en microprogramas.

Un proyecto para reducir el tamaño de un archivo puede iniciar por la aplicación de los métodos dependientes del contenido de los datos para luego continuar con los métodos independientes de la aplicación.

A. METODOS DEPENDIENTES DEL CONTENIDO DE LOS DATOS.

1) Eliminación de los items de datos redundantes.

Es uno de los métodos más importantes para reducir el espacio de almacenamiento de una base de datos; se basa en la eliminación de la redundancia inherente al almacenamiento múltiple de items de datos idénticos en distintos archivos; que es uno de los objetivos principales en los sistemas de administración de bases de datos.

2) Conversión de la notación humana en notación compacta.

Cuando se almacenan los datos de la forma en que los seres humanos prefieren para leerlos, a menudo emplean más caracteres de los necesarios. Las fechas por ejemplo, pueden escribirse como 30 SEP 91, o de manera más compacta 30.09.91, de modo que ellas se almacenan a menudo con seis bits en los archivos. Sin embargo, en la máquina el mes y el día no necesitan más de 4 bits, lo que hace un total de 16 bits (2 bytes). La conversión de la forma de 2 bytes a la forma legible para el hombre exige sólo algunas líneas de código.

Este tipo de compactación se aplica a muchos otros items, por ejemplo el número de pieza y las direcciones postales.

3) Supresión de caracteres repetidos.

Los campos numéricos de algunos archivos contienen a menudo muchos ceros a la izquierda y a la derecha. Cuando hay más de dos ceros, la codificación requiere en rigor sólo dos caracteres (decimales compactados), uno para indicar la repetición y el otro para indicar cuantas veces se repite el cero.

Algunos archivos de supresión de caracteres están basados en el empleo de un grupo de caracteres exclusivos para indicar que el carácter que sigue a este carácter especial es repetitivo. Por ejemplo, cuando se emplea la codificación convencional **ESCDIC** de 8 bits, la mayoría de las combinaciones que se dan con 8 bits no se usan por lo general para representar caracteres de datos. Las combinaciones que tienen un 0 en la segunda posición (posición de 1 bit), podrían utilizarse por consiguiente, para indicar la repetición de caracteres.

Indica que el carácter que sigue se repite

XOXXXXX

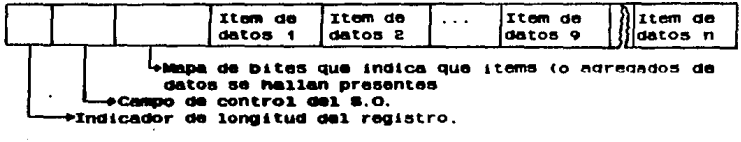
Número de veces en que se repite el carácter

Existen algunos tipos de archivos, en donde la supresión de los carácteres repetidos reportan un gran beneficio; y en otros, el efecto es apenas digno de consideración.

4) EVITACION DE LOS ESPACIOS VACIOS.

En algunos archivos, los registros y los items de los datos son de longitud muy variable. Por lo que la organización de un archivo para items variables es más compleja que la de un archivo para items de longitud fija.

Un planteamiento similar es el concerniente a registros que contienen un conjunto variable de items de datos en lugar de un conjunto fijo, o en otros términos, el de los registros que contienen un conjunto de items de datos de los cuales algunos suelen faltar. En ciertos casos es éste un problema muy serio: ya que en ocasiones la mayoría de los items de datos cuyo espacio está previsto en los registros no tienen valor alguno. Es posible idear un método de compactación, tal que los items de datos que no tienen valor no ocupen espacio. La siguiente figura ilustra un formato de registro para este método. En donde los items de datos están precedidos por un mapa de bits que indica si esos items se hallan o no presentes.



5) SUBSTITUCION DE LOS ITEMES DE DATOS DE USO HABITUAL.

Cuando existe un conjunto limitado de valores de atributos, no es necesario escribir el item de dato en forma completa. Con ciertos tipos de items de datos, por ejemplo, <<clase de licencia>> se utilizaría normalmente un código; y con otros como <<nombre>> que a menudo se deletrean de manera completa.

Por consiguiente, al decidirse por el uso de una tabla de sustitución, hay que llegar a una adecuada concertación entre el aumento del tiempo de procesamiento y el beneficio de la compactación. El uso de una tabla de sustitución de dos bytes para 65, 536 posibilidades parece irrazonable para la mayoría de los casos. Aún una tabla de sustitución de 8 bits requeriría cerca de 5,000 bytes de memoria, lo que también puede considerarse demasiado oneroso. Por lo que sería más conveniente limitar a 6 bits la tabla de sustitución, con lo que el byte de codificación podría también informar sobre el estado civil de la persona o dar otra información de interés.

6) SUSTITUCION DE TEXTO IDIOMATICO.

Cuando el texto escrito constituye una porción considerable de un archivo, parece una buena idea la de sustituir palabras y aún frases completas con caracteres de código. Supongamos que un archivo almacena bytes de 8 bits, cada byte permite combinaciones suficientes para representar más o menos 200 palabras de uso común, supongamos además que se reservan 32 de las 256 posibles combinaciones para indicar que ese carácter no da por sí sólo la palabra en cuestión sino que es necesario leer también el siguiente carácter. Se tiene así la posibilidad de codificar 32(256)=8192 palabras adicionales.

Por otra parte, la máquina que recibe esta sucesión de datos la convertirá en texto escrito mediante la operación de lectura de la tabla de equivalencias, además el vocabulario necesario para muchas aplicaciones específicas es muy breve, y puede no ser necesario almacenar más de, digamos, 250 palabras. Como alternativa, este método podría representar no palabras, sino mensajes o frases.

7) COMPACTACION DE DATOS ORDENADOS.

Al clasificar un conjunto de items de datos, los primeros caracteres tienden a repetirse. Estos items de datos pueden abreviarse reemplazando los caracteres repetidos con un dígito y aún más, si se codifican los nombres de pila más comunes.

También pueden reemplazarse algunas palabras descriptivas de uso corriente con caracteres especiales, por ejemplo:

| = Sra.

* = S.A.

Ocasionalmente esta técnica da lugar a duplicaciones, por lo que es necesario ir a una tabla de desborde en la que se almacena de manera unívoca el nombre original.

El empleo de tablas de desborde da valor práctico a muchos métodos de compactación muy poderosos. Estos métodos son particularmente valiosos en el caso de las guías y tablas que son parte esencial de muchas organizaciones de archivos.

Las tablas utilizadas están a menudo ordenadas en secuencia ascendente, con poco cambio al pasar de un item a otro.

Es necesario tener presente que la eficiencia de las técnicas consideradas en este apartado dependen de la naturaleza de los datos con que son empleadas. Algunas de ellas dependen completamente de la aplicación y no podrían ser incluidas en un algoritmo de compactación de uso general. Otras, en cambio, como la técnica de supresión de caracteres repetidos, podrían incluirse en un algoritmo de uso general.

8. METODOS DE USO GENERAL DE COMPACTACION DE DATOS.

Un algoritmo de compactación, diseñado para ser usado en sistemas diferentes puede implementarse de tres maneras: como software, como microprogramas o como hardware especial.

Además varias firmas ofrecen paquetes de software en los que se combinan la codificación de longitud variable con la supresión de caracteres repetidos, dos operaciones que pueden ser ejecutadas por el mismo algoritmo.

La mayoría de los métodos de compactación dan origen a registros de longitud variable, por lo que sólo se recomienda su uso en las organizaciones que admiten este tipo de registro.

Es deseable estimar con anticipación el beneficio que podría reportar el uso de la compactación en un archivo existente. Recurriendo para ello a la simulación, procesando los archivos en cuestión, o parte de ellos, con el algoritmo propuesto y acoplando los valores estadísticos. Así podrían compararse diversas técnicas y diferentes formas de codificación de longitud variable.

1) CODIFICACION DE CARaCTERES.

Una forma de obtener una compactación de datos que ahorre espacio de memoria y proporcione seguridad a los datos es el empleo de la codificación de caracteres, lo cual es muy utilizado en la transmisión de información.

Tenemos que los caracteres numéricos se almacenan a menudo con 4 bits; pero la forma de almacenamiento más económica de la información numérica es binaria y los datos alfabéticos y la mayoría de los signos podrían trasmitirse como caracteres de 5 bits.

CODIGO BAUT DE 5 BITS.

Este código utiliza caracteres de <<cambio de letras>> y <<cambio de cifras>> con el fin de aprovechar las mismas combinaciones principales de 5 bits, indistintamente, para representar letras o cifras (más caracteres especiales), algo así como la,<<tecla de mayúsculas>> de una máquina de escribir. Es interesante observar que un código de 5 bits con tres caracteres es suficiente para transportar todos los datos que la mayoría de los usuarios necesitan transmitir.

El código BAUT es el código telegráfico más utilizado fuera de los Estados Unidos.

CODIFICACION DE CARaCTERES DE LONGITUD VARIABLE.

Puede obtenerse una mejor compactación de los datos recurriendo a un código basado en el empleo de un número variable

de bits por carácter. Con este código, los caracteres de ocurrencia más frecuente resultarían cortos y, los de uso menos frecuente largos, ya que el carácter más corto (el que ocurre más frecuentemente) tendría un sólo bit.

Son muchos los códigos de longitud variable que se han ideado. La regla para determinar el número de bits por carácter es, en este caso, la siguiente:

"Si el primer bit es cero, el carácter tiene un bit de longitud, si los dos primeros dígitos son 10, el carácter tiene tres bits. En los demás casos, la longitud del carácter es igual al número de unos a la izquierda más tres".

En algunos archivos esta distinción corresponde a dos o más caracteres. Si hay dos caracteres mucho más populares que los demás, ellos se representan con dos bits cada uno.

DE OCHO A SIETE.

Es una técnica de compactación de datos cuyo esquema es la comprensión de bits, en donde en cada byte se guarda más de un carácter.

La mayoría de las computadoras actuales usan longitudes de byte que son potencias pares de dos, esto se debe a la representación binaria de los datos en la máquina. Las mayúsculas, las minúsculas y los signos de puntuación sólo requieren unos 63 códigos diferentes, por lo que bastan 6 bits para representar un byte (un byte de 6 bits podría tener valores entre 0 y 63). Sin embargo, la mayoría de las computadoras usan bytes de 8 bits, por lo tanto, en los archivos de texto normales se desperdicia el 23% del espacio de almacenamiento. Lo que nos sugiere que podríamos compactar realmente 4 caracteres en 3 bytes, si se pudieran usar los dos últimos bits de cada byte, la única limitante lo constituye la forma en que están organizados los códigos **ASCII**, hay mas de 63 códigos **ASCII** para los caracteres, y las letras mayúsculas y minúsculas son aproximadamente la mitad, ésto significa que algunos de los caracteres requieren al menos 7 bits. Por lo que una fácil solución es la de compactar 8 caracteres en 7 bytes, aprovechando el hecho de que ninguna letra, ni signo de puntuación usa el octavo bit del byte. Así, se puede ocupar el octavo bit de cada byte para guardar el octavo carácter. Sin embargo, hay que tener en cuenta que algunas computadoras usan los caracteres de 8 bits para representar caracteres especiales y caracteres gráficos.

Además, existen algunos procesadores de texto que usan el octavo bit para indicar instrucciones de procesamiento de texto. Por consiguiente, el empleo de este tipo de compactación de datos sólo funcionará con archivos **ASCII** que **"sencillamente"** no usen el octavo bit.

Ejemplo

Considérese los siguientes ocho caracteres representados como bytes de 8 bits:

byte 1	0111 0101
byte 2	0111 1101
byte 3	0010 0011
byte 4	0101 0110
byte 5	0001 0000
byte 6	0110 1101
byte 7	0010 1010
byte 8	0111 1001

Como se puede ver, el octavo bit siempre es cero, este siempre es el caso a no ser que se use para la comprobación de paridad.

La forma más simple de comprimir los ocho caracteres en 7 bytes es distribuir los 7 bits significativos del byte 1 en los octavos bits que no se usan de los otros 7 bytes (del 2 al 8). Así, los 7 bits restantes aparecerán como:

	byte 1 (leer hacia abajo)
byte 2	1111 1101
byte 3	1010 0011
byte 4	1101 0110
byte 5	0001 0000
byte 6	1110 1101
byte 7	0010 1010
byte 8	1111 1001

Para reconstruir el byte 1, simplemente se saca cada octavo bit de cada uno de los 7 bytes.

Esta técnica de compactación reduce el archivo 1/8, esto es, el 12.5% lo cual es ya un ahorro substancial.

ALGORITMO DE OCHO A SIETE.

```
# include "studio.h"
main (argc,argv) /* Comparación de bits */
int argc;
char *argv[3];
{
  if (argc!=4){
       printf("uso: entrada salida compactar/descompactar\n");
       exit ();
  }
  if (toupper(*argv 3 =='C')
       comprimir(argv[1],argc[2]);
  else
       descomprimir(argv[1],argc[2]);
}
comprimir(entrada,salida)
```

```
char *entrada, *salida;
\
  char car,ch2,t,hecho;
  FILE *pf1,pf2;
  if ((pf1=fopen(entrada,"r"))==0)\
     printf("no se puede abrir el archivo de entrada\n");
     exit();
  \
  if ((pf2=fopen(salida,"w"))==0)
     printf("no se puede abrir el archivo de salida\n");
     exit();
  \
  hecho = 0;
  do \
     car = getc(pf1);
     if (car==EOF) break;
     car = car<<1;
     for (t=0;t<7;++t)
          ch2 = getc(pf1);
          if (ch2==EOF)\
               ch2 = 0;
               hecho = 1;
          \
          ch2 = ch2 & 127; /* eliminar el primer bit */
          ch2 = ch2   ((car<<t)&128);
          putc(ch2,pf2);
       \
     \ while (!hecho);
     fclose(pf1); fclose(pf2);
\
descomprimir(entrada,salida)
char *entrada, *salida;
\
  unsigned char car,ch2,t,hecho;
  FILE *pf1,*pf2;
  if ((pf1=fopen(entrada,"r"))==0)\
     printf("no se puede abrir el archivo de entrada\n");
     exit();
  \
  if ((pf2=fopen(salida,"w"))==0)\
     printf("no se puede abrir el archivo de salida\n");
     exit();
  \
  hecho = 0;
  do \
     car = 0;
     for (t=0;t<7;++t)
          temp = getc(pf1);
          if (temp==EOF) break;
               ch2 = temp;          /* conversión de tipo */
               s[t]= ch2 & 127;     /* eliminar el primer bit */
               ch2 = ch2 &128;      /* eliminar todos los bits
                    menos el primero*/
               ch2 = ch2>>t+1;
```

```
            car = car | ch2;
        }
        putc(car,pf2);
        for (t=0; t<7;++t) put c (s<t1pf2);
    } while (temp!=EOF);
    fclose(pf1); fclose(pf2);
}
```

EL LENGUAJE DE 16 CARACTERES.

El esquema de esta técnica de compactación de datos es la eliminación de carácteres, en donde se eliminan realmente ciertos carácteres del archivo.

Aún cuando muchas veces no es recomendable, este método elimina las letras innecesarias de las palabras, convirtiendo esencialmente estas abreviaturas. Por lo que la compactación de datos consiste en almacenar sólo los carácteres que se usan. Así que el empleo de las abreviaturas para ahorrar espacio es muy común. Sin embargo, el método que aquí se explica, en vez de emplear las abreviaturas comunes, lo que hace es eliminar ciertas letras de los mensajes. Para hacer esto, se necesita un alfabeto mínimo. Un alfabeto mínimo es aquel en el que se han eliminado los carácteres que son usados raramente, dejando sólo los que son necesarios para formar la mayoría de las palabras o para evitar la ambigüedad. Por lo tanto, cualquier carácter que no aparezca en el alfabeto mínimo será eliminado de las palabras en que aparezca. La forma en la que se deciden los carácteres que aparecen en el alfabeto mínimo es un tanto subjetiva. Para ejemplificar el método se usarán las 14 letras más comunes, además del espacio y los saltos de línea como alfabeto mínimo. Por lo que la automatización del proceso de abreviación requiere que se conozcan las letras del alfabeto que se usan más frecuentemente para poder formar el alfabeto mínimo. En teoría, se podrían contar las letras de cada palabra del diccionario, pero, sin embargo, ciertos escritores usan diferentes frecuencias que otros, de tal forma que una tabla de frecuencias debe estar basada sólo en las palabras que conforman un lenguaje. Como una alternativa, se puede contabilizar la frecuencia de las letras del texto a ser comprimido y usarlas como base para el alfabeto mínimo.

Para contabilizar la frecuencia de las letras en un texto podemos utilizar el siguiente programa, que se salta todos los signos de puntuación excepto los puntos, las comas y los espacios.

```
#include "stdio.h"
#include "ctype.h"
main(argc,argv) /* programa de frecuencia de carácteres */
int argc;
char *argv[];
{
    FILE *pf1;
```

```
int alfac26],t;
int espacio = 0, punto = 0, coma = 0;
char car;
if (argc!=2)
    printf("favor de especificar el archivo de texto\n");
    exit();
}
if ((pf1=fopen(argv 1,"r"))==0)
    printf("no se puede abrir el archivo de entrada\n");
    exit();
}
for (t=0;t<26;t++) alfa t =0;
do  {
      car = getc(pf1);
      if (isalpha(car))
         alfa[toupper(car)-'A']++;
      else switch(car)
           case   ' '  : espacio ++;
                         break;
           case   '.'  : punto++;
                         break;
           case    ','  : coma++;
                         break;
  } while (car!=EOF);
  for (t=0; t<26; ++t)
      printf("%c:%d n",punto);
  printf("punto   :%d n",punto);
  printf("espacio:%d n",espacio);
  printf("coma    :%d n",coma);
  fclose(pf1);
}
```

Por lo visto, para obtener una compactación de datos substancial, es necesario recortar el alfabeto de forma significativa, eliminando las letras usadas con menor frecuencia. Aunque hay muchos puntos de vista sobre como debe de estar formado el alfabeto mínimo para ser operativo, las 14 letras más usadas y el espacio constituyen la mayor parte de los caracteres usados en un texto, además como el carácter de fin de línea es necesario para evitar la ruptura de las palabras debe incluirse también.

Ejemplo

El alfabeto que se empleará para ejemplificar el método, consta de 14 caracteres, un espacio y un salto de línea:

A C D E H I L M N O R S T U <espacio> <salto de línea>

En seguida, presentamos un programa, el cual elimina todos los caracteres excepto los anteriores.

ALGORITMO PARA UN LENGUAJE DE 16 CARACTERES.

```
# include "stdio.h"
main (argc,argv) /* programa de eliminación por compactació..
                    de caracteres */
int  *argc;
char *argv[];
{
  if (argc!=3){
       printf("uso:entrada salida\n");
       exit();
  }
  comp2(argv[1],argv[2])
  comp2(entrada,salida);
  char *entrada, *salida;
  char car;
  FILE *pf1, *pf2;
  if ((pf1=fopen(entrada,"r"))==0){
      printf("no se puede abrir el archivo de entrada \n");
      exit();
  }
  if ((pf2=fopen(salida,"w"))==0)
      printf("no se puede abrir el archivo de salida \n");
      exit();
  }
  do
   {
    car = getc(pf1);
    if (car==EOF) break;
    if (esta_en(toupper(car),"ACDEGILMNORSTU\n"))
    {
      if (car=='\n') putc('\r',pf2);
     putc(car,pf2);
    }
    while (1);
    fclose(pf1);
    fclose(pf2);
    }
    esta_en (car,s)
    char car,*s;
    {
      while (*s)
       {
        if (car==*s) return 1;
        s++;
       }
       return 0;
    }
```

Si al siguiente mensaje se le aplicara el programa anterior:

" Atención Alta Comandancia
Ataque con éxito. Por favor envíen mas víveres y tropas de relevo. Es esencial para mantener nuestra situación.
General Frashier "

El mensaje compactado sería:

" Atencion Alta Comandancia

Ataue con eito Por aor enien mas ieras troas de releo Es esencial ara mantener nuestra situacion

eneral roshier"

Como se puede observar el mensaje es bastante legible, aunque presenta cierta ambigüedad. La ambigüedad es el principal inconveniente de este método de compactación de datos. Sin embargo, si se está familiarizado con el vocabulario empleado en el texto, se puede seleccionar un mejor alfabeto mínimo que resuelva cierto grado de ambigüedad.

En el ejemplo se ahorró un 15% de espacio, ya que el mensaje original tenía 156 bytes y el mensaje compactado 137. Si también se hubiera aplicado el método de compactación de bits, se habría ahorrado un 28% de espacio lo que ya es un ahorro importante.

Tanto la compactación de bits como la eliminación de caracteres se usan en la encriptación de mensajes. Por otra parte, la compactación de bits es un paso más en la codificación de información y hace al mensaje más difícil de decodificar. Al usar la eliminación de bits antes de la encriptación se produce un efecto curioso : se trastoca la frecuencia de caracteres del lenguaje fuente.

5.4.2 Compactación de archivos.

Las siguientes técnicas que presentaremos para optimizar espacio son métodos de codificación de información teórica que fueron diseñados para minimizar la cantidad de información necesaria en sistemas de comunicaciones y originalmente fue un intento para optimizar tiempo, no espacio.

Las técnicas de compactación de archivos son usadas muchas veces para archivos de texto, archivos para decodificar pinturas (las cuales tienen grandes áreas homogéneas) y archivos para la representación digital de sonido y otras señales analógicas (con grandes modelos repetidos).

Es importante notar que un propósito general de los métodos de compactación es hacer los archivos más pequeños. La cantidad de espacio salvado por esos métodos depende de las características del archivo. ahorrando del 20% al 50% de espacio en los archivos típicos de texto y del 50% al 90% en los archivos binarios.

Sin embargo en la actualidad estas técnicas de compactación de archivos son menos importantes que en otros tiempos, debido a que el costo de almacenamiento en dispositivos ha descendido dramáticamente y el típico usuario puede disponer de mucho más espacio de almacenamiento que en el pasado. Pero, a pesar de

esto, se puede argumentar que las técnicas de comprensión de datos aún son importantes, porque si se emplea mucho almacenamiento será necesario recurrir a grandes tiempos de acceso y naturalmente esto es relativamente costoso.

METODO RUN-LENGTH.

Se tiene que el tipo más simple de redundancia en un archivo es la ejecución de carácteres repetidos. Por ejemplo, la siguiente cadena, **AAAABBBAABBBBBCCCCCCCCDABCBAAABBBBCCCD**, puede ser codificada más compactamente reemplazando cada repetición de caracteres por una sola instancia del carácter repetido al lado del número de veces que se repite. Así se puede decir que esta cadena consiste de **4A's**, seguido 3B's, 2A's,5B's,etc. A la compactación de una cadena por esta forma es llamada **RUN-LENGTH.** Existen varias formas para proceder con esta idea, las cuales dependen de las características de la aplicación (¿Qué hace que la ejecución tienda a ser relativamente larga?, ¿Cuántos bits son empleados para codificar los carácteres?) Por ejemplo si sabemos que nuestra cadena contiene letras, entonces se pueden codificar grandes cantidades con sólo intercalar los dígitos con las letras y la cadena puede ser codificada como sigue: **4A3BAA5B8CDABCB3A4B3CD**; DONDE "4A" significa "AAAA" y así sucesivamente. Además se nota que no vale la pena codificar subcadenas de longitud 1 ó 2, ya que dos carácteres son los que se necesitan para la codificación.

Para los archivos binarios se emplea una versión refinada de este método, en donde la idea es simplemente almacenar la cadena tomando como ventaja el hecho de que la cadena alterna entre 0 y 1, evitando almacenar los ceros y los unos por ellos mismos (Esto nos sugiere que el método trabaja más eficientemente con cadenas grandes).

Ejemplo

La siguiente figura es una representación de la letra **"q"** engañosa en su tamaño, que es representativa del tipo de información que tendría que ser procesada por un sistema de formato de texto, tal como el empleado para imprimir este trabajo; a la derecha de la misma hay una lista de números que pueden ser usados para almacenar una letra en forma compacta.

```
00000000000000000000000000011111111111111110000000000      28,14,09
00000000000000000000000000111111111111111110000000000      26,18,07
00000000000000000000000001111111111111111110000000000      23,24,04
00000000000000000000000011111111111111111111000000000      22,26,03
00000000000000000000001111111111111111111111111110000      20,30,01
00000000000000000000111111100000000000000000111111111      19,07,18,07
00000000000000000001111100000000000000000000011111111      19,05,22,05
00000000000000000011100000000000000000000000000000111      19,03,26,03
00000000000000000011100000000000000000000000000000111      19,03,26,03
00000000000000000011100000000000000000000000000000111      19,03,26,03
00000000000000000011100000000000000000000000000001110 20,04,23,03,1
00000000000000000000011100000000000000000000000111000 22,03,20,03,3
01111111111111111111111111111111111111111111111111111         01,50
01111111111111111111111111111111111111111111111111111         01,50
01111111111111111111111111111111111111111111111111111         01,50
01111111111111111111111111111111111111111111111111111         01,50
01111111111111111111111111111111111111111111111111111         01,50
01100000000000000000000000000000000000000000000000011 01,02,46,02
```

Contadores

Esto significa que la primer linea consta de 28 0's seguidos por 14 1's, 9 0's, etc. Los 63 contadores más el número de bits por linea (51 bits) contienen suficiente información para reconstruir los bits del arreglo (En particular, se nota que el indicador de fin de línea no es necesario). Si se emplean 6 bits para representar cada contador, entonces el archivo entero se representa con 384 bits, una optimización sustancial de los 975 bits necesarios para almacenarlo explícitamente.

La codificación Run-Length requiere una representación separada para el archivo que va ser codificado y la versión codificada del archivo, pero esto no se puede hacer con todos los archivos por resultar inconveniente. Por ejemplo, este método sugiere no trabajar con cadenas de caracteres que contienen dígitos ya que estos caracteres se emplean para codificar una cadena desde un alfabeto fijo de caracteres usando sólo caracteres de ese alfabeto.

¿Cómo se puede hacer para que algunas letras representen dígitos y que otras representen parte de la cadena que se va a codificar ? Una solución es usar algún carácter que sea poco probable que aparezca (a este carácter se le llama carácter de escape) y cada que aparece dicho carácter indica que las próximas dos letras (contador, carácter) forman un par, con contadores representados por la i-ésima letra del alfabeto que representa el número i.

Ejemplo

En este ejemplo la cadena anterior podría ser representada utilizando la letra **Q** como carácter de escape, así:

QDABBBAAQEBQHCDABCBAAAQDBCCCD

La combinación hecha del carácter de escape, con el contador y con una copia del carácter repetido es llamada secuencia de escape. Es de notar que no vale la pena codificar subcadenas de menos de cuatro caracteres de longitud puesto que para codificar por este método se requieren 3 caracteres.

Pero, ¿qué sucede si el mismo carácter de escape ocurre en la cadena de entrada? No podemos ignorar simplemente esa posibilidad, puesto que es muy difícil asegurar que un carácter determinado no ocurrirá (Por ejemplo alguien podría probar codificar una cadena que ya ha sido codificada). Una solución a este problema es usar una secuencia de caracteres de escape con un contador de cero para representar el carácter de escape. Así en nuestro ejemplo, el carácter de espacio podría representar el cero, y la secuencia de escape **"Q<espacio>"** podría ser usada para representar una ocurrencia de Q en la cadena de entrada. Es interesante notar que los archivos que contienen Q's son sólo aquellos que en lugar de disminuir su tamaño lo incrementan por este método de compactación. Puesto que si un archivo que ya ha sido compactado es compactado nuevamente, este crece por lo menos un número de caracteres iguales al número de secuencias de escape usados

Muchas secuencias largas han sido codificadas con secuencias de escape múltiple. Por ejemplo, una cadena de 51A's podría ser codificada como QZAQVA usando la convención anterior. Por lo que si se esperan muchas cadenas largas esto podría dar resultados satisfactorios a reserva de que más de un caracter codifique el contador

En la práctica es recomendable realizar programas de comprensión y expansión de errores sensitivos, lo que se puede dar incluyendo una cantidad de redundancia en los archivos compactados para que los programas de expansión puedan tolerar un mínimo cambio accidental en los archivos de compactación y expansión. Por ejemplo, probablemente valga la pena poner carácteres de fin de línea **"en la versión compactada de la letra Q"** para que el programa de expansión se pueda resincronizar por sí mismo en caso de error.

Por otra parte, la codificación de cadenas largas no es particularmente efectiva para archivos de texto, debido a que el carácter que se repite con mayor probabilidad es el blanco. En los sistemas modernos, no se repiten las cadenas de blancos ya que éstas jamás son cadenas de entrada y por consiguiente nunca se almacenan; las cadenas de blancos repetidas al inicio de las líneas son codificadas como **"tabs"** y **"tabuladores"** y los blancos al final de la línea son abreviados por el uso de indicadores de **"fin de línea"**. Una implementación de codificación de cadenas largas semejantes sólo óptimiza cerca del 4% de espacio, porcentaje nada satisfactorio, por tal motivo no se recomienda este tipo de codificación para archivos de texto

CODIFICACION DE LONGITUD VARIABLE (CODIGO HUFFMAN).

El código de Huffman optimiza una cantidad substancial de espacio en los archivos de texto. la idea de este código es abandonar el clásico almacenamiento en el cual los archivos de texto son almacenados, donde usualmente cada carácter emplea 7 de los 8 bits reservados para él. El método de Huffman usa sólo pocos bits para aquellos caracteres que son usados más frecuentemente y mas bits para aquellos que son usados rara vez.

Es conveniente examinar como se usa el código Huffman para poder entenderlo.

Suponemos que deseamos codificar la cadena "A SIMPLE STRING TO BE ENCODED USING A MINIMAL NUMBER OF BITS", haciéndolo en nuestro código binario standard compacto con 5 bits binarios, donde i representa la -ésima letra del alfabeto (0 para blanco), da la siguiente secuencia de bits:

```
00001000001001101001011011000011000010100000
10011101001001001001011100001101000011111
00000000100010100000001010111000011011100100
00101001000000010101100110100101110001100000
00001000000110101001011100100101101000101100
00000011101010101101000100010110010000001111
00110000000001001001101001011
```

Al decodificar este mensaje simplemente se leen 5 bits en un tiempo y se convierte de acuerdo al código binario definido anteriormente. En este código standard la "C", que aparece sólo una vez, requiere el mismo número de bits que la letra "I" que aparece seis veces.

El código Huffman logra economizar espacio por codificar caracteres usados frecuentemente con pocos bits como sea posible, por consiguiente el número total de bits empleados para el mensaje es minimizado.

El código de Huffman sigue los siguientes pasos para lograr su objetivo:

El primer paso es contabilizar la frecuencia de cada carácter en el mensaje que va ser codificado. El siguiente paso es llenar el código en un arreglo count[0..26] con la frecuencia contada de un mensaje en un arreglo de caracteres a 1..M .

```
FOR i:=0  TO 26  DO count[i] := 0;
FOR i:=1  TO  m  DO
    count[index(car[i])] := count[index(car[i])] + 1;
```

Este programa utiliza el procedimiento index (descrito en la parte inicial de este capitulo) que contabiliza la frecuencia de la i-ésima letra del alfabeto en count[i] con count[0] usado para blancos.

Para nuestra cadena por ejemplo, la tabla contadora producida es:

LETRA	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
	11	3	3	1	2	5	1	2	0	6	0	0	2	4	5

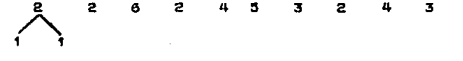
LETRA	15	16	17	18	19	20	21	22	23	24	25	26
	3	1	0	2	4	3	2	0	0	0	0	0

lo cual indica que hay 11 blancos, 3 A's, 3 B's, etc.

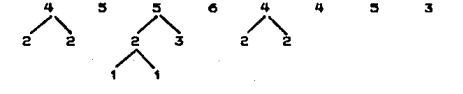
El próximo paso es construir un árbol de código bottom-up (de abajo hacia arriba y de derecha a izquierda) de acuerdo a las frecuencias. Primero, se crea un árbol con un nodo para cada frecuencia distinta de cero:

11	3	3	1	2	5	1	2	6	2	4	5	3	1	2	4	3	2
	A	B	C	D	E	F	G	I	L	M	N	O	P	R	S	T	U

Ahora se escogen los dos nodos con frecuencia más pequeña y se crea un nuevo nodo con esos dos nodos como hijos, con valor igual a la suma de los dos valores de los hijos:

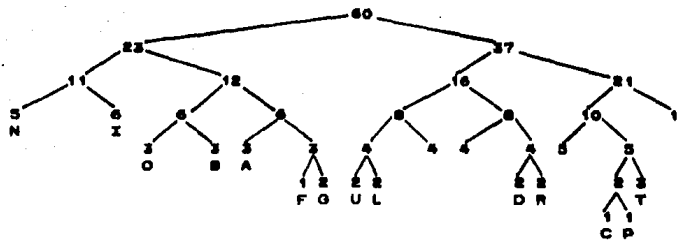


El bosque de los árboles después de que todos los nodos con frecuencia dos son tomados, es:



Ahora se escogen los nodos con frecuencia 3, creando nuevos nodos, etc. Continuando por este camino se construyen subárboles cada vez más grandes, al final todos los nodos se combinan en un sólo árbol.

Nota que los nodos con menor frecuencia se encuentran más lejos de la raíz, que los nodos con mayor frecuencia. Las etiquetas de los nodos externos en el árbol son los contadores de frecuencia mientras que las etiquetas de cada nodo interno es la suma de las etiquetas de los hijos. El número pequeño cerca de cada nodo en este árbol (siguiente figura) es el índice del arreglo count donde la etiqueta es almacenada, por referencia cuando examinamos el programa que construye el árbol anterior (las etiquetas para los nodos internos podrían ser almacenadas en count 27..51 en un orden determinado por lo dinámico de la construcción). Esto es, por ejemplo, el 5 en el nodo más externo a la izquierda (la frecuencia contada para N) es almacenada en count 14, el seis en el próximo nodo externo (la frecuencia contada para I) es almacenada en count 33, etc.



Si observamos la descripción de las estructuras de las frecuencias en forma de un árbol es exactamente lo que se necesita para crear una eficiente codificación. Antes de ver esta codificación veamos el código para construir el árbol. El propósito general involucra remover el elemento más pequeño desde un conjunto de elementos no ordenados, por lo cual nosotros podemos emplear el algoritmo **HEAP** para crear y mantener un heap indirecto en los valores de frecuencia. Dado que primeramente estamos interesados en los valores pequeños se usará el método HEAP en forma invertida. Una desventaja de usar la indirección es que es fácil ignorar el contador de frecuencia cero.

La siguiente tabla muestra el heap construido para el ejemplo:

K	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
H(k)	3	7	16	21	12	15	6	20	9	4	13	14	5	2	18	19	1	0
count H k	1	2	1	2	2	3	1	3	6	2	4	5	5	3	2	4	3	11

específicamente, este heap es construido, primero inicializando el arreglo Heap indicando las frecuencias distintas de cero, y posteriormente empleando el procedimiento **HEAP** como sigue:

```
n := 0;
FOR i:=0  TO  26  DO
    IF  count[i]<>0  THEN
    BEGIN
      n := n+1;
      H[n] := i;     { H es el Heap }
    END;
 FOR k:=n DOWNTO 1  DO  PQDOWNHEAP(k);
```

Como se mencionó antes, se asume que el sentido de las igualdades en el heap han sido invertidas.

Ahora, el uso de este procedimiento al construir el árbol es en forma directa: se toman los dos elementos más pequeños del heap, se suman y se coloca el resultado dentro del heap. En cada

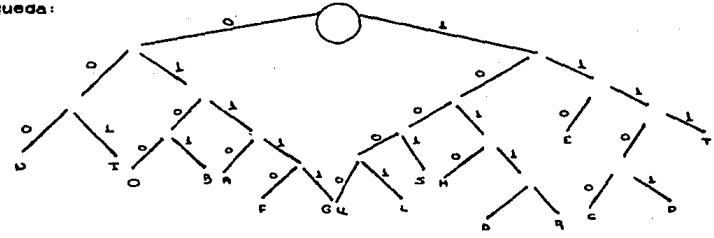
paso se crea un nuevo contador y se decrementa el tamaño del heap en uno. Este proceso crea N-1 nuevos contadores, uno para cada uno de los nodos internos del árbol creado, como en el siguiente procedimiento:

```
REPEAT
      t:=H[1]; H[1]:=H[n]; n:=n-1;
      PQDOWNHEAP(1);
      count[26+n]:=count[H[1]]+count[t] ;
      dad[t]:=26+n; dad[H[1]]:=-26-n;
      H[1]:=26+n; PQDOWNHEAP(1);
UNTIL n=1;
dad[26+n]:=0;
```

En las dos primeras líneas de este ciclo el tamaño del heap se decrementa en 1, y un nuevo interno es creado con índice 26+n y se le da un valor igual a la suma del valor en la raíz.

Después este nodo se coloca en la raíz, que altera esta prioridad haciendo necesario otra llamada a PQDOWNHEAP para restablecer el orden en el heap. El árbol por sí mismo es representado con un arreglo de "padres" ligados: dad[t] es el índice del padre del nodo cuyo peso está en count[t], la señal de dad[t] indica si el nodo que está a la derecha o izquierda es hijo de ese padre. Por ejemplo, en el árbol anterior se tiene dad[0]:=-30, count[30]:=21, dad[30] =-28, y count[28] =37 (lo que indica que el nodo de peso 21 tiene indexado 30 y este padre tiene indexado 28 y 37).

El código de Huffman es derivado de codificar el árbol, simplemente reemplazando las frecuencias de los nodos bajos con las letras asociadas y si vemos el árbol como una búsqueda radix queda:



Ahora el código puede ser leído directamente de este árbol. El código para N es 000, el código para I es 001, el código para C es 110100, etc.

El siguiente programa reconstruye los fragmentos de esta información para representar el código del árbol calculado durante el proceso. El código es representado por dos arreglos **code K** que da la representación binaria de la K-ésima letra y **len K** que contiene el número de bits de **code K** usado en el código. Por ejemplo, I es la novena letra y tiene código 001, por lo tanto **code 9 =1** y **len 9 =3**.

```
FOR  k:=0  TO  26  DO
     IF countCk]=0 THEN
     BEGIN
          codeCk] :=0;
          lenCk] :=0;
     END
     ELSE
     BEGIN
          i:=0;
          j:=1;
          t:=dadCk];
          x:=0;
          REPEAT
               IF  t<0  THEN
               BEGIN
                    x := x+j;
                    t := -t;
               END;
               t := dadCt];
               j := j+1;
               i := i+1;
          UNTIL t = 0;
          codeCk]:=x;
          lenCk]:=i;
     END;
```

Finalmente, podemos usar esa representación calculada del código para codificar el mensaje.

```
  FOR  j:=1  TO  m  DO
    FOR  i:=lenCindexCa[j]]] DOWNTO  1  DO
         WRITE (bitsCcodeCindexCa[j]],i-1,1):1);
```

Así nuestro mensaje es codificado en sólo 236 bits de los 300 bits usados por la codificación directa, optimizando con esto el 21%:

```
          01101111001001101011010110001110011110011011
          10111001000011111111101101001110101110011111
          00001101000100101101100101101111000010010010
          00111111101101111010001000001101001101000111
          00010000101001011100101111110100011101110101
          01110111001
```

Un interesante hecho del código de Huffman que sin duda ya fue notado, es que los delimitadores entre los carácteres no han sido

almacenados, aún diferentes caracteres pueden ser codificados con diferentes números de bits. ¿Cómo podemos determinar dónde un carácter termina y el próximo inicia en la decodificación del mensaje? La respuesta es utilizar la búsqueda radix de representación de código. Tal búsqueda tiene inicio en la raíz procediendo hacia abajo del árbol de acuerdo a los bits en el mensaje, cada vez que un nodo externo es encontrado se saca el carácter y se reinicia en la raíz. Pero el árbol se construye al mismo tiempo que se codifica el mensaje: esto significa que es necesario salvar el árbol junto con el mensaje en el orden en que se decodifica éste. Afortunadamente es necesario sólo almacenar el arreglo code, porque la búsqueda radix del árbol es la resultante de insertar las entradas desde el arreglo en una inicialización vacía del árbol.

El almacenamiento optimizado citado anteriormente no ocurre en forma integra, porque el mensaje no puede ser decodificado sin el árbol y por lo tanto tenemos que contabilizar el costo de almacenar el árbol junto al mensaje.

El código Huffman es sólo efectivo para archivos grandes donde la optimización del mensaje es bien compensado por el costo, o en situaciones donde el árbol codificado puede ser precalculado y usado para un gran número de mensajes. Por ejemplo, un árbol basado en las frecuencias de letras del lenguaje Inglés puede ser usado por documentos escritos en este idioma. Así mismo, un árbol basado en la frecuencia de caracteres de programas escritos en lenguaje **PASCAL** puede ser usado para codificar programas.

Como antes se vio, para archivos aleatorios verdaderos, aún este hábil esquema de codificación no funciona en forma óptima, porque cada carácter podría ocurrir aproximadamente el mismo número de veces, lo que ocasionaría un árbol codificado completamente balanceado y un número igual de bits por letra en el código.

EFICIENCIA DE ESTA TECNICA DE COMPACTACION.

En la mayoría de los archivos comerciales, el uso de un código de Huffman, cuidadosamente elegido de acuerdo con la distribución de los caracteres en el archivo, conduce a una reducción del 50% del tamaño del archivo, o a más. La supresión de los caracteres repetidos divide también por dos el tamaño del archivo. La eliminación de datos no utilizados tiene un importante efecto en algunos archivos.

Varios estudios del uso del código de Huffman para la compactación de programas arroja como resultado economías del orden del 35 al 45% para el código objeto y del 55 al 75% para el código fuente.

CAPITULO VI

ALGORITMOS ESPECIFICOS (ALGUNOS EJEMPLOS)

Dada la amplitud del campo de desarrollo y aplicación de los algoritmos, no es factible englobar todo ello en el presente trabajo, por lo que se optó por incluir este capítulo, presentando en él algoritmos de interés común a la **LICENCIATURA DE MATEMATICAS APLICADAS Y COMPUTACION**, no tratados en capítulos anteriores. Los algoritmos seleccionados fueron: **ALGORITMOS PARA TEORIA DE GRAFICAS Y ALGORITMOS PARA ESTADISTICA Y PROBABILIDAD**; que son algoritmos que pertenecen a materias consideradas de carácter obligatorio en el plan de estudios de esta carrera.

6.1 ALGORITMOS PARA TEORIA DE GRAFICAS.

Una gráfica G consiste de un conjunto de vértices, V=(v_1,v_2,v_3,...vn), llamados también puntos o nodos y un conjunto de arcos E=(e_1,e_2,e_3,....,em) que reciben también el nombre de líneas, segmentos, ejes o aristas. Cada arco ei que pertenece a E está relacionado a un único par de vértices que pertenecen a V.

La importancia de la teoría de gráficas radica en su utilidad como modelo matemático para representar cualquier sistema que contenga o involucre una relación entre sus elementos. Permite un enfoque intuitivo y estético del problema que se pretende resolver y, son probablemente la estructura matemática más empleada en la ciencia computacional ya que son utilizadas en el estudio de estructuras de datos, compiladores, lenguajes de programación, sistemas operativos, teoría de la computación, ordenación, búsqueda, ingeniería eléctrica, inteligencia artificial, economía, matemáticas, física, química, comunicaciones, teoría de juegos y otras áreas.

La eficiencia de un algoritmo para teoría de gráficas está en función al saber seleccionar la forma más apropiada para representar las gráficas involucradas en el problema a resolver por computadora.

6.1.1 CONCEPTOS INTRODUCTORIOS.

En todo estudio de gráficas es necesario tener presente los siguientes conceptos:

- **LINEAS PARALELAS:** La relación entre un par de vértices no es necesariamente única, si se tienen varias líneas que asocian un par de vértices se dice que se tienen líneas paralelas

Ejemplo

a y e son líneas paralelas:

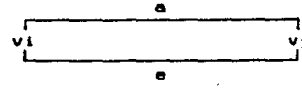


FIG 6.1 Líneas paralelas

- **LOOP O BUCLE:** Es un arco que une un vértice consigo mismo, su dirección no tiene significado y puede ser considerado dirigido o no dirigido.

Ejemplo

b es un bucle:

FIG 6.2 Bucle o ciclo

- **VERTICES EXTREMOS O TERMINALES:** Son los vértices que se encuentran unidos por un arco, también son llamados nodos adyacentes o vecinos.

Ejemplo

vi y vj son vértices terminales:

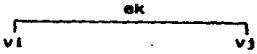


FIG 6.3 Vértices terminales

- **INCIDENCIA:** Se dice que una línea es incidente al vértice vi y vj, si dichos vértices son terminales de la línea ek.

Ejemplo

ek es incidente al vértice vi y vj:

FIG 6.4 Línea incidente

- **GRADO DE UN VERTICE:** Es el número de arcos que contiene un nodo, los loops que pueda contener un vértice se cuentan por dos, si el grado de un nodo es cero se dice que es un nodo nulo o aislado; si es uno se trata de un vértice colgante o final. El grado de un vértice se representa como g(vi) y se llama también valencia del vértice.

Ejemplo

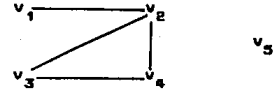


FIG 6.5 Grado de un vértice

El grado de cada vértice de la gráfica anterior es:

$g(v_1) = 1$ vértice colgante o final.
$g(v_2) = 3$
$g(v_3) = 2$
$g(v_4) = 2$
$g(v_5) = 0$ vértice aislado.

- **CAMINO**: Un camino o paseo P de longitud n en una gráfica G(V,E) que va de un nodo v_0 a vn, se define como una secuencia alternada (finita) de n+1 nodos y n líneas:

$$P = (v_0, e_1, v_1, e_2, \ldots, en, vn)$$

tal que el vértice vi es adyacente al vértice vi+1 para i=0,1,..n-1. Tal camino o paseo no debe cruzar más de una vez una línea, es decir, todas las líneas son distintas; sus vértices pueden ser cruzados más de una vez.

- **PASEO CERRADO**: Es aquel paseo donde el vértice inicial y el vértice final son iguales ($v_0 = vn$).

- **PASEO ABIERTO**: Es aquel paseo en el cual el vértice inicial y el vértice final son distintos ($v_0 <> vn$).

- **TRAYECTORIA O CAMINO SIMPLE**: Es un paseo abierto en el cual todos los vértices son distintos.

- **CIRCUITO**: Paseo cerrado con n vértices distintos, es decir, cada vértice aparece una vez, excepto el vértice inicial que es igual al vértice final.

Ejemplo

De la siguiente gráfica obtenga cuatro paseos distintos e indique de que tipo de paseo se trata y cual es su longitud:

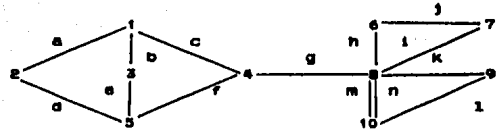


FIG 6.6 Gráfica

PASEO:	NOMBRE:	LONGITUD:
1,c,4,g,8	Trayectoria	2
1,a,2,d,5,f,4,c,1	Circuito	4
1,a,2,d,5,e,3,b,1,c,4	Paseo abierto	5
10,n,8,i,7,j,6,h,8,m,10	Paseo cerrado	5

-GRAFICA CONECTADA (CONEXA): Una gráfica G está conectada, si para todo par de vértices que pertenecen a ella, existe una trayectoria que los une, de lo contrario es una gráfica desconectada.

Una gráfica desconectada G está compuesta por dos o más subgráficas conectadas, a las cuales se les llama componentes.

Ejemplo

Sea la gráfica G ¿Es una gráfica conectada?

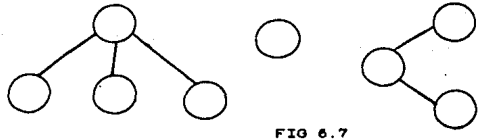


FIG 6.7

No, ya que no siempre existe una trayectoria que una cualquier par de vértices que pertenezcan a la gráfica. La gráfica G es desconectada y consta de tres componentes.

6.1.2 REPRESENTACION DE LAS GRAFICAS EN LA COMPUTADORA.

Como se ha visto, una gráfica es una colección de vértices y arcos; los vértices son objetos simples que pueden tener nombres y ciertas propiedades, y un arco es una relación entre dos vértices. Una gráfica puede ser dibujada por medio de puntos (vértices) y líneas (arcos); pero se debe tener presente que una

gráfica se define independientemente de su representación. Existen dos formas estándar de representar una gráfica en la memoria de la computadora:

-Representación contigua (Representación algebraica).

-Representación ligada.

a) REPRESENTACION CONTIGUA (REPRESENTACION ALGEBRAICA).

La representación contigua o algebraica se basa en el empleo de matrices y vectores. Las representaciones contiguas más usadas son:

1. MATRIZ DE ADYACENCIA.

La matriz de adyacencia A de una gráfica G, de n vértices (etiquetados) y sin líneas paralelas es la matriz de nxn A(G); donde:

$$a = \begin{cases} 1 & \text{si Vi es adyacente a Vj} \\ 0 & \text{otra cosa} \end{cases}$$

La matriz A(G) es una matriz binaria que requiere $/n/^2$ bits, donde n es el número de vértices de la gráfica.

OBSERVACIONES A LA MATRIZ DE ADYACENCIA:

- G es cuadrada.
- Un uno en la diagonal principal corresponde a un bucle.
- La definición de matriz de adyacencia no considera líneas paralelas (por ser una matriz binaria). Si no fuera binaria y existiera un elemento aij > 1 se dice que el valor de aij es el número de líneas paralelas entre los vértices i y j.
- Si la gráfica es simple (no contiene loops, ni líneas paralelas), la sumatoria por renglón o columna es el grado del vértice.
- En una gráfica simple el número de líneas es igual a la sumatoria de todos los elementos aij entre dos.
- Si una gráfica es completa (sin loops y una línea entre todo par de vértices), implica que tiene ceros en la diagonal principal y unos en todos los demás elementos.
- Las permutaciones de renglones y sus correspondientes columnas implican reordenar los vértices.
- Una gráfica desconectada con dos componentes G_1 y G_2, se puede representar como

$$A(G) = A(G_1) \cup A(G_2)$$

$$A(G) = \left[\begin{array}{c|c} A(G_1) & 0 \\ \hline 0 & A(G_2) \end{array}\right]$$

- Dada una matriz binaria, cuadrada y simétrica A de orden n, siempre se podrá construir la gráfica G correspondiente. Ejemplo

Sea la matriz A:

	1	2	3	4
1	1	0	1	1
2	0	0	1	0
3	1	1	0	0
4	1	0	0	1

La gráfica correspondiente a esa matriz es:

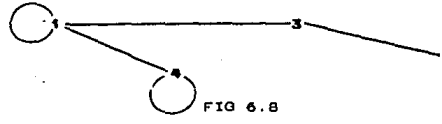


FIG 6.8

La principal desventaja de emplear una matriz de adyacencia para representar una gráfica dirigida, es que se requiere un espacio (n^2), aún si la gráfica posee menos de n^2 arcos. Sólo leer o examinar la matriz consume un tiempo $O(n^2)$, lo cual anula la eficiencia de los algoritmos que manipulan la gráfica en un tiempo $O(n)$.

2. MATRIZ POTENCIA.

Al multiplicar la matriz de adyacencia consigo misma o elevarla a una potencia, el resultado de esa multiplicación es una matriz cuadrada de orden n simétrica pero no binaria. El valor de aii, corresponde al grado del vértice i. Para la matriz $A^2(G)$ sus elementos también representan:

- El número de vértices que son adyacentes tanto al vértice i como al vértice j.
- Los elementos de la diagonal representan el grado del vértice correspondiente.
- El elemento a^2ij con i<>j representa el número de caminos o paseos distintos de longitud dos que existen del vértice i al vértice j; y en general dada una matriz de adyacencia A(G) de n vértices, entonces los elementos aij de A**m m >= 1, es el numero de caminos o paseos diferentes de longitud m del vértice i al vértice j.

3. MATRIZ DE INCIDENCIA.

La matriz de incidencia de una gráfica G de n vértices y e lineas, sin loops está dada por la matriz B(G) = bij de nxe elementos, donde

$$b = \begin{cases} 1 & \text{si la línea j es incidente a Vi} \\ 0 & \text{otra cosa} \end{cases}$$

OBSERVACIONES A LA MATRIZ DE INCIDENCIA.

- Es una matriz de incidencia vértice-línea.
- La matriz B(G) es una matriz binaria.
- En cada columna hay exactamente dos unos, excepto si se trata de un bucle; en tal caso aparece sólo un uno.
- La sumatoria por renglón (adicionando uno por cada bucle) es igual al grado del vértice correspondiente.
- Si la sumatoria de un renglón es cero, el vértice correspondiente es aislado.
- Cuando se tienen dos o más columnas iguales en B(G), las líneas correspondientes son líneas paralelas.
- Si se intercambian renglones o columnas y se reetiquetan los vértices y líneas se obtiene la misma gráfica.
- Si una gráfica G es desconectada y consta de dos o más componentes, G_1 y G_2, la matriz B(G) puede ser representada como:

$$B(G) = \begin{bmatrix} B(G_1) & 0 \\ 0 & B(G_2) \end{bmatrix}$$

- Dada una matriz de incidencia es posible generar el diagrama gráfico correspondiente.

Ejemplo

$$B(G) = \begin{array}{c} \\ 1 \\ 2 \\ 3 \\ 4 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 1 & 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 1 & 1 \end{bmatrix} \end{array}$$

La gráfica correspondiente a esa matriz de incidencia es:

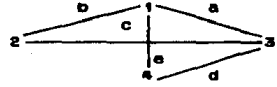


FIG 6.9

RELACION ENTRE LA MATRIZ DE ADYACENCIA A(G) Y LA MATRIZ DE INCIDENCIA B(G).

Si una gráfica no tiene líneas paralelas, la matriz de adyacencia A(G) tiene toda la información correspondiente a la gráfica G, pero por el contrario, si la gráfica tiene líneas

paralelas y no tiene loops, entonces la matriz de incidencia B(G) contiene toda la información de G; si se tiene una gráfica simple (sin loops o líneas paralelas) tanto A(G) como B(G) contienen toda la información de la gráfica G por lo que a partir de la matriz de adyacencia se puede construir la matriz de incidencia y viceversa.

4. MATRIZ CIRCUITO.

Dada una gráfica G con líneas y circuitos etiquetados, La matriz circuito C(G) = cij, tiene un renglón por cada circuito y una columna por cada línea de la gráfica G donde:

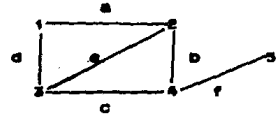
$$c_{ij} = \begin{cases} 1 & \text{si el circuito } i \text{ contiene la línea } j \\ 0 & \text{si el circuito } i \text{ no contiene la línea } j \end{cases}$$

OBSERVACIONES DE LA MATRIZ CIRCUITO.

- La sumatoria de los elementos de un renglón es la longitud del circuito correspondiente.
- Cuando una columna esta constituida en su totalidad por elementos iguales a cero indica que la línea no pertenece a ningún circuito.
- Si se realiza una permutación de renglones o columnas se deben restiquetar los circuitos y líneas corespondientes para obtener la misma gráfica.

Ejemplo

De la gráfica siguiente obtener la matriz circuito:



$$C = \begin{array}{c} \\ 1 \\ 2 \\ 3 \end{array} \begin{array}{c} \begin{array}{cccccc} a & b & c & d & e & f \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 1 & 1 & 0 \\ 0 & 1 & 1 & 0 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{bmatrix} \end{array}$$

circuito:
$C_1 = 1,2,3,1$
$C_2 = 2,4,3,2$
$C_3 = 1,2,4,3,1$

FIG 6.10

5. MATRIZ TRAYECTORIA.

La matriz trayectoria T(Vk,Vi) es conveniente para diferentes problemas tales como, el problema del transporte, de comunicación y de realización de proyectos; al igual que las matrices

anteriores es una matriz binaria, y se toma en cuenta para un par de vértices de la gráfica, es decir, la matriz se genera sólo de las trayectorias posibles entre un par de vértices, donde los renglones de T(VK,Vl) son las diferentes trayectorias de VK a Vl, y las columnas son todas las líneas de la gráfica.

La matriz trayectoria para los vértices VK y Vl se define como T(VK,VL) = tij donde:

$$t = \begin{cases} 1 & \text{si la trayectoria i contiene la línea j} \\ 0 & \text{si la trayectoria i no contiene la línea j} \end{cases}$$

OBSERVACIONES DE LA MATRIZ TRAYECTORIA.

- La suma de cada renglón es la longitud de la trayectoria.
- Cuando una columna está constituida por todos sus elementos iguales a uno la línea correspondiente es una línea de corte (línea que al ser eliminada desconecta la gráfica).
- La sumatoria de cada columna representa el número de trayectorias a las que pertenece la línea.
- Si la sumatoria de una columna es igual a cero, entonces esa línea no pertenece a ninguna trayectoria entre los vértices VK y Vl.
- La longitud más corta entre VK y Vl es el renglón cuya sumatoria es la menor.

Ejemplo:
De la siguiente gráfica obtener la matriz trayectoria del vértice uno al vértice dos:

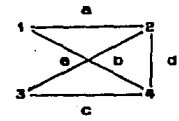


FIG 6.11

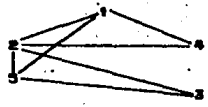
$$T(1,2) = \begin{array}{c} \\ T_1 \\ T_2^1 \\ T_3^2 \end{array} \begin{array}{c} \begin{array}{ccccc} a & b & c & d & e \end{array} \\ \begin{bmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \end{bmatrix} \end{array}$$

trayectoria:
T_1 = 1,2
T_2^1 = 1,4,2
T_3^2 = 1,4,3,2

6. VECTORES ADJUNTOS.

El i-ésimo renglón contiene un vector con todos los vértices adjuntos Vi; el orden de los elementos de este renglón está determinado por el orden en el cual los estados de G son dados.

Ejemplo



1	4	2	5	0
2	4	1	5	3
3	5	2	0	0
4	1	2	0	0
5	1	3	2	0

FIG 6.12

El número de columnas es el máximo grado de un vértice.

7. LISTAS DE ESTADOS.

Para gráficas esparcidas puede ser más eficiente listar los arcos representados en la gráfica como pares de vértices. Esta representación puede ser implementada con dos arreglos $g=(g_1,g_2,\ldots,g/E/)$ y $h=(h_1,h_2,\ldots,h/E/)$ donde E es el número de arcos. Cada entrada es un vértice etiquetado, el i-ésimo arco en la gráfica va del vértice gi al vértice hi. Por ejemplo:

g = (1,2,2,2,3,3,4,5,5,5,7,7)
h = (6,1,3,4,6,4,5,5,3,6,7,1,6)

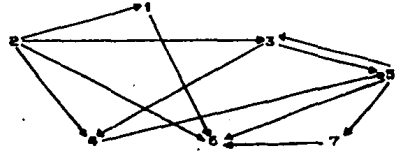


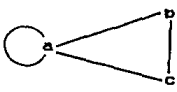
FIG 6.13

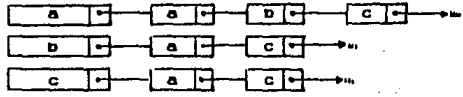
Aún cuando las matrices sirven como entrada para el procesamiento por computadora; una matriz destruye el aspecto gráfico (visual) y ciertas propiedades de las gráficas no puedan ser representadas en cualquier matriz; además en la representación contigua es difícil insertar y eliminar nodos y si el número de vértices es grande en comparación al número de arcos, la matriz contendrá un gran número de ceros (matriz esparcida) lo cual ocasiona un gran desperdicio de memoria.

b) REPRESENTACION LIGADA.

La representación ligada, es una alternativa para representar gráficas en la computadora.

1. Un posible formato para representarlas de esta manera es el siguiente:





nodos cabeza de lista — nodos con los que está relacionados

FIG 6.14

En gráficas no dirigidas y con esta representación se requieren 2m-r nodos y n cabezas de lista, donde m es el número de arcos, r el número de loops o bucles y n el número de nodos.

Ejemplo

En la gráfica anterior

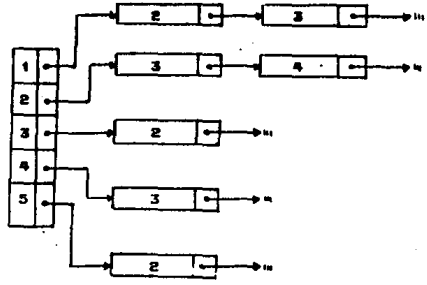
m = 4
r = 1
n = 3 por lo tanto 2m-r = 2(4)-1
= 7 nodos y 3 cabezas de lista.

2. LISTA DE ADYACENCIA.

La lista de adyacencia para un vértice i es una estructura, en algún orden, de todos los vértices adyacentes a i. La gráfica G se representa por medio de un vector cabeza, donde cabeza[i] es un apuntador a la lista de adyacencia del vértice i. Esta representación requiere un espacio proporcional a la suma del número de vértices más el número de arcos (si son no dirigidas se cuentan por dos); se recomienda emplearla cuando el número de arcos es mucho menor a n^2, sin embargo la desventaja potencial de la representación con listas de adyacencia es que puede consumir un tiempo O(n) el determinar si existe un arco del vértice i al vértice j, ya que pueden existir n vértices en la lista de adyacencia para el vértice i.

Si van a existir inserciones y supresiones en la manipulación de la gráfica, sería preferible que la matriz cabeza de la lista de adyacencia apunte a celdas de encabezado que no contengan vértices adyacentes (Fig 6.15). Por otra parte si se espera que la gráfica permanezca fija, sin cambios (o muy pocos) en la lista de adyacencia sería preferible que **cabeza[i]** fuera un apuntador a un vector **ADV**, donde **ADV[cabeza[i]]**, **ADV[cabeza[i]+1]**,... y así sucesivamente, contuvieran los vértices adyacentes al vértice i, hasta el punto en **ADV** donde se encuentra por primera vez un cero, el cual marca el fin de la lista de vértices adyacentes a i (Fig 6.16).

Representación de G por medio de listas de adyacencia:



vector cabeza celdas de encabezado

FIG 6.15.

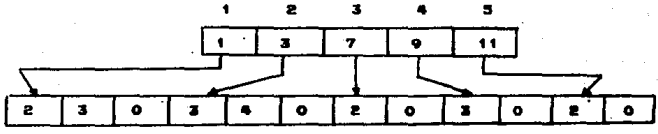


FIG 6.16.

En conclusión, para representar una gráfica, se pueden emplear diversas estructuras de datos, la selección apropiada de una de ellas depende de las operaciones que se aplicarán a los vértices y a los arcos de la gráfica.

6.1.3 ARBOLES.

Un árbol es un tipo especial de gráfica, cuya relación entre los elementos es de uno a varios de ellos; esta estructura se emplea principalmente para representar datos con una relación jerárquica determinada (relaciones de dependencia). En un árbol se distingue un nodo especial llamado raíz del árbol, los demás nodos pueden ser particionados en conjuntos diferentes

$T_1, T_2, \ldots Tn$ (n>=0) cada uno de los cuales representa un árbol y son llamados subárboles de la raíz.

Para que una gráfica reciba el nombre de árbol debe cumplir las siguientes características según Kurt . (FUNTAMENTALS OF THE COMPUTING SCIENCES).

1) El número de nodos es igual al número de arcos más uno.
2) Todos los nodos son de grado interno (número de líneas que llegan a él) igual a uno, excepto el nodo raíz que es de grado cero.
3) No tiene ciclos ni loops.
4) Cualquier trayectoria es simple (No se repiten vértices).
5) Entre cualquier par de nodos sólo existe una trayectoria.
6) Cualquier arco, es una línea de corte (si se elimina dicha línea, la gráfica queda desconectada).

TERMINOLOGIA EMPLEADA EN EL MANEJO DE ARBOLES.

Frecuentemente en el manejo de árboles se requiere del conocimiento de las siguientes definiciones:

- **Grado o grado externo de un nodo:** Es el número de subárboles que posee un nodo.
- **Hoja o nodo terminal:** Nodos de grado cero.
- **Nodo padre:** Nodo antecesor directo.
- **Nodo hijo:** Nodo sucesor directo.
- **Nivel de un árbol:** A menudo también se le denomina generación y es el nivel del nodo antecesor directo (padre) más uno. El nivel de la raíz es uno.
- **Nodos hermanos:** Nodos con el mismo nivel jerárquico y un padre común.
- **Altura de un árbol:** La altura o profundidad de un árbol es el número de nodos que constituyen el camino más largo desde la raíz a una hoja; la altura se encuentra dada por el nodo que posee el nivel más grande.
- **Bosque:** Conjunto de árboles. Si la raíz de un árbol se borra se genera un bosque y si al contrario las raíces de un bosque se unen se genera un sólo árbol.

El siguiente esquema muestra algunos de estos conceptos:

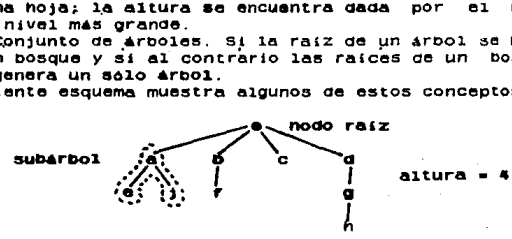


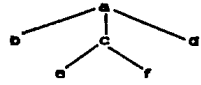
FIG 6.17

a es un nodo padre de e y j; e es un nodo hijo de a, y, e y j son nodos hermanos.

FORMAS DE REPRESENTAR UN ARBOL.

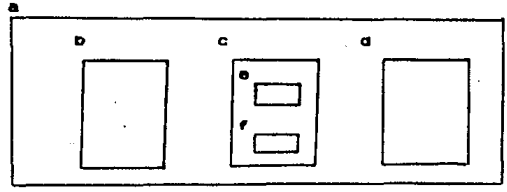
Existen distintas formas de representar un árbol, pero no todas muestran con claridad la relación que existe entre los nodos y ésto es la principal consecuencia de su poco empleo. Algunas de las representaciones son:

- **Gráfica:** Circulos o puntos unidos por lineas.



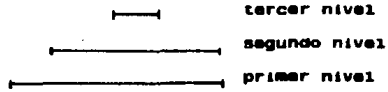
- **Barras:** Barras o segmentos de distinta longitud para representar los distintos niveles.

- **Conjuntos:**



- **Paréntesis:**

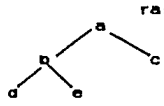
(a (b,c (e,f), d))



Las formas empleadas en la computadora para representar un árbol son también diversas; la selección de una de ellas para la solución de un problema especifico depende del tipo de operaciones que se desean realizar con la estructura. Algunas de ellas son:

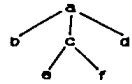
1) Por medio de una matriz de adyacencia, empleando los elementos de la diagonal principal para representar la raíz.

Ejemplo



	a	b	c	d	e
a	•	1	1	0	0
b	0	0	0	1	1
c	0	0	0	0	0
d	0	0	0	0	0
e	0	0	0	0	0

2) Por medio de un arreglo unidimensional donde cada localidad del arreglo representa cada nodos de árbol y su contenido a su antecesor directo.



a	b	c	d	e	f
e	a	a	a	c	c

3) Para representar un árbol en forma dinámica o ligada se deben tener nodos con el número de campos liga igual al número máximo de arcos que tenga un nodo que pertenezca al árbol:

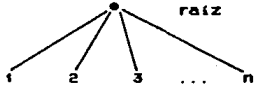
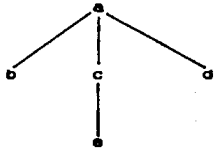


FIG 6.18

Ejemplo



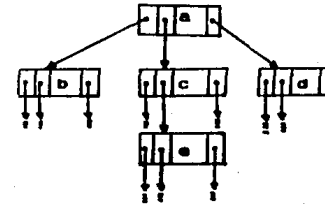


FIG 6.19

Este tipo de representación tiene muchos inconvenientes ya que contiene muchos enlaces **NIL** (↳) y además su recorrido necesita una pila, lo que consume tiempo y espacio en memoria; sin embargo esta forma es la más adecuada de representar a los árboles para determinadas aplicaciones .

RECORRIDO DE LOS ARBOLES.

Recorrer un árbol es realizar visitas a los nodos para recuperar la información almacenada en ellos.

Las formas más simples de realizar éstas son:

1) **TOP-DOWN (Recorrer el árbol de arriba hacia abajo):**

Consiste en visitar la raiz y continuar con los niveles consecutivos recorriendo los nodos de izquierda a derecha.

2) **BOTTOM-UP (Recorrer el árbol de abajo hacia arriba):**

Consiste en recorrer el árbol del n-ésimo al primer nivel, cada nivel de derecha a izquierda.

Ejemplo

RECORRIDO

TOP-DOWN : 1,2,3,4,5
BOTTOM-UP : 5,4,3,2,1

FIG 6.20

TIPOS DE ARBOLES.

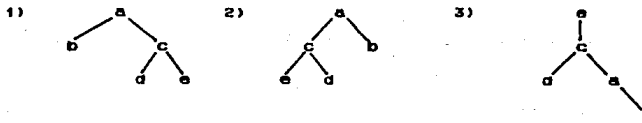
La clasificación de árboles que se muestra a continuación se basa en el hecho de considerar o no la dirección y orden de los subárboles:

- **ARBOL ORDENADO:** El orden de los subárboles es importante, los vértices o nodos se encuentran alineados de izquierda a derecha.

- **ARBOL ORIENTADO:** No se considera un orden para los subárboles.

- **ARBOL LIBRE:** La dirección de los arcos que pertenecen al árbol se ignora.

Ejemplo



Si los árboles son ordenados : 1 <> 2 <> 3
Si los árboles son orientados: 1 = 2 <> 3
Si los árboles son libres : 1 = 2 = 3

FIG 6.21

ARBOL BINARIO O ARBOL DE KNUTH.

Un árbol binario es un árbol cuyos nodos poseen cero, uno o dos subárboles; cuando cada nodo del árbol tiene exactamente cero o dos subárboles se trata de un árbol extrictamente binario.

Un árbol binario tiene tres características principales:

1) Cada nodo puede tener a lo más dos subárboles.
2) Cada subárbol se identifica como hijo derecho o hijo izquierdo.
3) Puede ser vacio (denominado árbol nulo o vacio).

Dos árboles binarios T y T son **SIMILARES** si tienen la misma estructura; formalmente esto significa que ambos son vacios y sus subárboles izquierdo y derecho son similares respectivamente. Similar significa informalmente que los diagramas de T y T son iguales.

Dos árboles binarios T y T se dice que son **EQUIVALENTES** si son similares y los nodos correspondientes contienen la misma información.

TRANSFORMACION DE ARBOLES A ARBOLES BINARIOS.

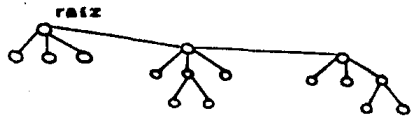
El convertir un bosque a un árbol binario es una operación extremadamente importante y se llama correspondencia natural entre los bosques y los árboles binarios. Para convertir un bosque a un árbol de Knuth se aplica el siguiente algoritmo:

PASO 1) Ligar las raices de los árboles que pertenecen al bosque.
PASO 2) La raiz del nuevo árbol será la raiz del árbol de la izquierda.
PASO 3) Ligar todos los nodos hermanos.
PASO 4) Retirar todas las ligas de un padre a sus hijos, excepto la del hijo que se encuentra más a la izquierda.

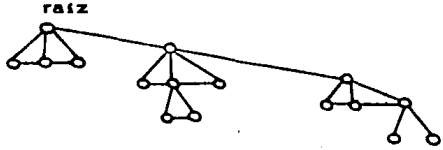
Ejemplo

Convertir el siguiente bosque a un arbol de Knuth, empleando el algoritmo anterior:

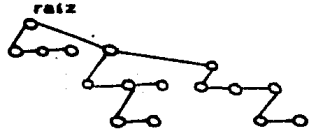
PASOS 1 y 2:



PASO 3:



PASO 4:



ORDENANDOLO:

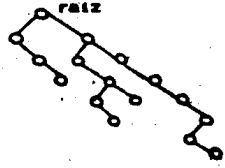


FIG 6.22

ARBOL BINARIO EXPANDIDO O ARBOL-2.

Un árbol binario es un árbol-2 o árbol expandido si cada nodo tiene 0 o 2 hijos denominando nodos internos y nodos externos a los que no tienen hijos . Para mayor claridad en la representación gráfica los nodos internos se representan con círculos y los nodos externos se representan con cuadros. Si en un árbol binario expandido existen n nodos circulares y s nodos cuadrados se tienen **n+s-1** arcos donde **s=n+1**, por lo que el número de nodos externos es uno más que el número de nodos internos.

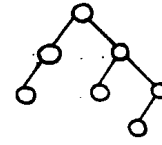
La **LONGITUD DE TRAYECTORIA EXTERNA** de un árbol E se define como la suma (tomando todos los nodos externos) de las longitudes de todos los caminos desde la raíz a cada nodo externo.

La **LONGITUD DE TRAYECTORIA INTERNA** I, es la suma de las longitudes de todos los caminos desde la raíz a cada nodo interno.

Para construir un árbol binario expandido se reemplaza cada subárbol vacío por uno o dos nuevos nodos según corresponda, de tal forma que los nodos originales del árbol son los nodos internos del árbol expandido y los nuevos nodos son los nodos externos.

Ejemplo

Obtener el árbol expandido del siguiente árbol y calcular su longitud de trayectoria interna y externa.



n = 7

El árbol expandido correspondiente es:

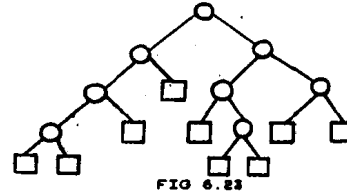


FIG 6.23

Longitud de trayectoria:

E = 3+3+2+3+4+4+3+3 = 25 I = 2+1+0+2+3+2+1 = 11

en donde E = I + 2n

El concepto de longitud de trayectoria de un árbol en el análisis de algoritmos, es de gran importancia, ya que ésta cantidad está directamente relacionada con el tiempo de ejecución de un proceso.

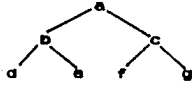
REPRESENTACION DE LOS ARBOLES BINARIOS EN COMPUTADORA.

Es importante tener presente que las formas de implementar un árbol general en una computadora, no permiten distinguir si un nodo es hijo izquierdo o hijo derecho, en el caso de estar tratando un árbol binario; por lo que no pueden ser empleadas para representar este tipo de árbol.

Los árboles binarios también pueden ser implementados en forma contigua y en forma ligada.

ORGANIZACION CONTIGUA: Se emplean arreglos unidimensionales, en los cuales los hijos de cualquier nodo k se encuentran en la localidad 2k (hijo izquierdo) y 2k+1 (hijo derecho); lo que implica que la longitud del arreglo es 2**n-1; donde n es el número de niveles del árbol.

Ejemplo



1	2	3	4	5	6	7
a	b	c	d	e	f	g

Dado que n = 3, la longitud del arreglo es 2**n-1 = 7

FIG 6.24

ORGANIZACION LIGADA: Se emplean nodos que contienen tres campos mínimamente; uno o más de ellos para almacenar la información, y los otros dos para los hijos derecho e izquierdo.

LIGA IZQUIERDA	INFORMACION	LIGA DERECHA

Se requiere también de dos apuntadores básicos :

RAIZ: Apuntador estático ya que apunta siempre al primer elemento.

ACTUAL: Apuntador dinámico dado que cambia su posición de acuerdo a la última operación realizada (indica el último elemento que se insertó, consultó o el antecesor del último nodo que se borró).

RECORRIDO DE UN ARBOL BINARIO.

Existen tres formas básicas de recuperar la información de los nodos de un árbol binario denominadas: **PREORDEN, ENORDEN** y **POSTORDEN**; estos nombres denotan la posición de la visita realizada a la raíz respecto a sus subárboles. Dado que la mayoría de las funciones que utilizan árboles son recursivas, ya que el árbol por sí mismo es una estructura recursiva (cada subárbol es un árbol), las rutinas que se desarrollan en esta sección para recorrer el árbol son recursivas y pueden ser implementadas en forma no recursiva pero, el código generado, puede ser más difícil de comprender.

- **PREORDEN: Recupera la información antes de visitar sus subárboles.**

PASO 1) Procesa la raíz.
PASO 2) Recorre el subárbol izquierdo en preorden.
PASO 3) Recorre el subárbol derecho en preorden.

CODIGO.

```
PROCEDURE PREORDEN (p : apuntador);
BEGIN
     IF  p <> NIL  THEN
     BEGIN
         PROCESAR(p);
         PREORDEN(p^.HI);
         PREORDEN(p^.HD);
     END;
END;
```

- **ENORDEN: La información es recuperada después de visitar alguno de sus subárboles.**

PASO 1) Recorre el subárbol izquierdo en enorden.
PASO 2) Procesa la raíz.
PASO 3) Recorre el subárbol derecho en enorden.

CODIGO.

```
PROCEDURE ENORDEN (p:apuntador);
BEGIN
     IF  p <> NIL  THEN
     BEGIN
         ENORDEN (p^.HI);
         PROCESAR(p);
         ENORDEN (p^.HD);
```

```
        END;
    END;
```

- **POSTORDEN : La información se recupera después de visitar los dos subárboles.**

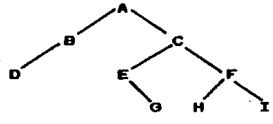
PASO 1) Recorre el subárbol izquierdo en postorden.
PASO 2) Recorre el subárbol derecho en postorden.
PASO 3) Procesa la raiz.

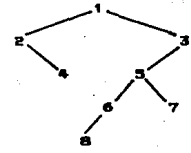
CODIGO.

```
PROCEDURE POSTORDEN (p:apuntador);
BEGIN
     IF  p <> NIL  THEN
     BEGIN
          POSTORDEN(p^.HI);
          POSTORDEN(p^.HD);
          PROCESAR(p);
     END;
END;
```

Ejemplo

Recorrer los siguientes árboles en PREORDEN, ENORDEN y POSTORDEN.





Preorden : ABDCEGFHI
Enorden : DBAEGCHFI
Postorden: DBGEHIFCA

Preorden : 12435687
Enorden : 24186573
Postorden: 42867531

FIG 6.25

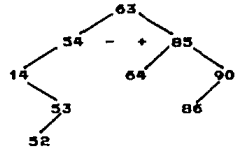
ALGUNOS EJEMPLOS DE ARBOLES BINARIOS.

- ARBOL BINARIO DE BUSQUEDA.

Un árbol binario de búsqueda es aquel en el que el valor que contiene un nodo es mayor que cualquier valor de su subárbol

izquierdo y menor (o igual si permite duplicaciones) que cualquier valor de su subárbol derecho; lo cual garantiza que el recorrido enorden de un árbol binario de búsqueda generá una lista ordenada.

Ejemplo



Recorrido en enorden:

14,52,53,54,63,64,85,86,90

FIG 6.26

Si se desea una lista en orden descendente, el recorrido sería:

PASO 1) Visitar subárbol derecho en enorden.
PASO 2) Procesar la raíz.
PASO 3) Visitar subárbol izquierdo en enorden.

Este tipo de árbol nos permite minimizar los tiempos de búsqueda, ya que se puede determinar si un elemento pertenece o no a la estructura en forma más eficiente (sin tener que recorrer todo el árbol).

- ARBOL B.

En este tipo de árbol siempre se considera un orden para los subárboles (incluso cuando sólo existe uno de ellos); cada arco tiene asociado un peso cuyo valor es cero si el arco está orientado a la izquierda y uno cuando se encuentra orientado a la derecha (dos arcos de un mismo nodo no pueden tener la misma orientación).

Ejemplo

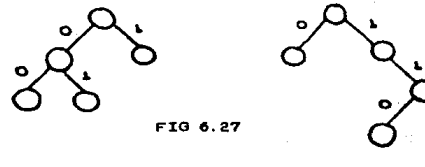


FIG 6.27

- ARBOLES BALANCEADOS.

Un árbol balanceado es aquel donde existe una diferencia de altura igual a cero entre el subárbol izquierdo y derecho.

Ejemplo

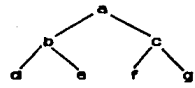


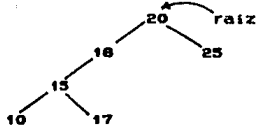
FIG 6.28

- ARBOL AVL.

Un árbol AVL es un árbol en el cual la diferencia de altura máxima en los subárboles izquierdo y derecho es uno. Por lo que todo árbol balanceado es un árbol AVL.

Un árbol que no es AVL puede ser convertido a este tipo de árbol realizando una rotación de sus nodos, lo cual se realiza con el propósito de estandarizar los tiempos de búsqueda.

Ejemplo



No es un árbol AVL.

FIG 6.29

realizando una rotación para convertirlo en AVL, tenemos:

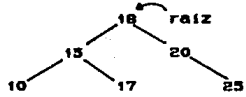


FIG 6. 30

- ARBOL B-TREE.

En este árbol cada nodo contiene un rango de valores que debe ser finito, es decir, es necesario establecer un límite máximo de elementos en cada nodo.

Ejemplo

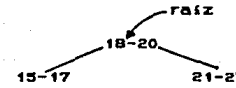


FIG 6.31

- **ARBOL DEGENERADO.**

Cuando un árbol se comporta como una lista, recibe el nombre de árbol degenerado.

Ejemplo

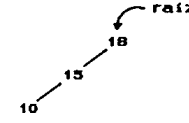


FIG 6.32

APLICACIONES DE LOS ARBOLES.

- Las estructuras de árbol se emplean principalmente para representar cualquier diversificación de las relaciones entre los datos tales como la representación jerárquica; por ejemplo, el diagrama de organización de una empresa o de un conjunto que incluye conjuntos que en su momento pueden incluir sus propios conjuntos.

- El análisis de decisiones es el área primordial de aplicación de la estructura árbol debido a que cada nodo tiene un valor determinado, que permite la agilización de la información.

- Cuando se emplean programas de administración de base de datos, los árboles ofrecen potencia, flexibilidad y eficiencia ya que la información que contiene la base de datos, reside en disco y el tiempo de acceso es importante. Si se emplea un árbol binario equilibrado se realizan, en el peor caso **log n** comparaciones en una búsqueda, si se utiliza una lista enlazada, la búsqueda tiene que ser secuencial; siendo esta última menos eficiente.

- Los árboles también se emplean para obtener la notación deseada de una expresión aritmética. Si el árbol se recorre en preorden se obtiene la notación polaca prefija; si se recorre en enorden se obtiene la notación infija y si se recorre en postorden se obtiene la notación polaca postfija.

- En organización de archivos (por ejemplo los archivos del MS-DOS)

- En redes de computadoras (cada computadora tiene diferente nivel), y

- En la organización de una RED DESCENTRALIZADA donde cada nodo es una red.

6.1.4 GRAFICAS PESADAS.

Una gráfica pesada es una tripleta (V,E,W) donde (V,E) es una gráfica y W es una función de E en los enteros positivos. Si e pertenece a E entonces W(e) es llamado peso del arco e.

Ejemplo

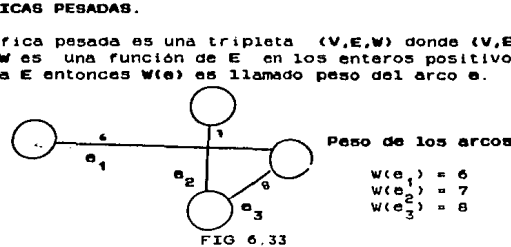


FIG 6.33

Los pesos de los arcos en algunas aplicaciones podrían corresponder a costos o aspectos deseados de los arcos.

Las formas de representar este tipo de gráficas en computadora es semejante al de representar una gráfica general; pero ahora es necesario considerar el peso de los arcos.

a) FORMA CONTIGUA.

Por medio de la matriz de adyacencia A(G) donde:

$$a = \begin{cases} W(e) & \text{si } e \text{ es la línea incidente a Vi y Vj} \\ 0 & \text{si Vi no es adyacente a Vj} \end{cases}$$

b) FORMA LIGADA.

En esta representación se anexa un nuevo campo, que contendrá el peso del arco.

Ejemplo

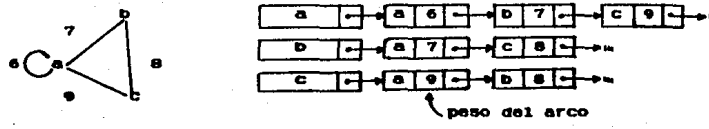


FIG 6.34

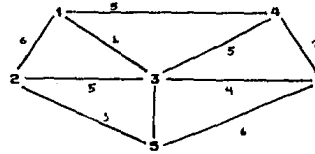
6.1.5 GRAFICAS NO DIRIGIDAS.

Una gráfica no dirigida G=(V,E) consta de un conjunto finito de vértices V y de un conjunto de arcos E, en la cual cada arco que pertenece a E es un par no ordenado de vértices, es decir si (v,w) es un arco no dirigido entonces (v,w)=(w,v).

ARBOL EXPANDIDO DE COSTO MINIMO.

Ahora G(V,E) es una gráfica conectada y no dirigida de la cual es necesario obtener un árbol expandido de costo mínimo (árbol que contenga todos los vértices de la gráfica cuyo peso en los arcos sea mínimo). Una aplicación típica de esto es el diseño de redes de comunicación: los vértices de la gráfica representan ciudades, y los arcos representan el costo de seleccionar ese canal de comunicación. El árbol expandido de costo mínimo de esa gráfica representa la red que comunica todas las ciudades de costo mínimo.

Ejemplo



El árbol expandido de costo mínimo de G es:

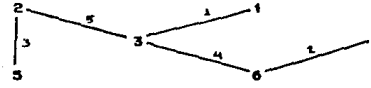


FIG 6.35

Existen diversas formas de construir un árbol expandido de costo mínimo, la mayoría de ellas emplean la propiedad denominada **ADN**:

Sea G(V,E) una gráfica conectada con una función de costo definida en los arcos; sea U un subconjunto propio del conjunto de vértices V. Si (U,V) es un arco de costo mínimo tal que u pertenece a U y v pertenece a V-U, existe un árbol de extensión de costo mímino que incluye (u,v) entre sus arcos.

ALGORITMOS PARA CONSTRUIR UN ARBOL EXPANDIDO DE COSTO MINIMO.

Los algoritmos más populares, basados en la propiedad ADM son dos; uno de ellos se conoce con el nombre de **PRIM** y el otro con el nombre de **KRUSKAL**.

a) ALGORITMO PRIM.

Supóngase que V = { 1,2,3, ...,n } ; el algoritmo PRIM inicia, asignando a un conjunto **U** el valor 1. Posteriormente localiza el arco con menor **peso (u,v)** que conecte U y V-U, agrega el vértice v que pertenece a V-U a U y repite este paso hasta que U sea igual a V.

Para encontrar el arco de menor costo entre U y V-U en cada paso, el algoritmo emplea dos vectores llamados: **mas_cercano** c i), el cual contiene el vértice que pertenece a U que está unido por el arco de menor peso a(i) que pertenece a V-U, y **menor_costo** que contiene el peso del **arco (i,mas_cercano(i))**.

En cada paso se revisa **menor_costo** para encontrar algún vértice k en V-U que se encuentre unido por el arco de menor costo a U; se imprime el **arco (k, mas_cercano k)**. Entonces se actualizan los vectores **menor_costo** y **mas_cercano**; teniendo en cuenta que **k** debe ser agregado a U.

En este algoritmo se supone que C es una matriz de nxn tal que c(i,j) es el costo del arco (i,j), si este arco no existe c(i,j) contiene un valor muy grande.

Si se encuentra un vértice k para el árbol expandido, **menor_costo(k)** debe actualizarse, con un valor muy grande, un valor mayor que el costo de cualquier arco o que el costo asociado a un arco no existente, de modo que este vértice ya no se considere en los recorridos siguientes.

CODIFICACION DEL ALGORITMO PRIM.

```
PROCEDURE PRIM (C: ARRAY  1..n  OF REAL);
 { imprime los arcos de un árbol expandido de costo mínimo para
   una gráfica de n vértices y una matriz de costos c }
VAR
    menor_costo : ARRAY [1..n] OF REAL;
    mas_cercano : ARRAY [1..n] OF INTEGER;
    i,j,k,min   : INTEGER;
    { i y j        son los indices,
      k            es el indice del vértice encontrado hasta
                   ese punto cuyo costo es mínimo, y
      min          es el valor del costo menor_costo k }
BEGIN
1)    FOR  i:=2  TO  n  DO
      BEGIN
            Asigna el vértice 1 al conjunto U
2)        menor_costo(i) := c[1,i];
3)        mas_cercano    := 1;
      END;
```

```
 4)      FOR  i:=2  TO  n  DO
         BEGIN
              { Encuentra el vértice k en V-U unido por un costo
                mínimo a U }
 5)           min := menor_costo[2];
 6)           k   := 2;
 7)           FOR  j:=3  To  n  DO
 8)                IF menor_costo[j] < min  THEN
                  BEGIN
 9)                     min := menor_costo[j];
10)                     k   := j;
                  END;
11)           WRITELN (k,mas_cercano k);  { imprime arco }
12)           menor_costo[k] := infinito; { se anexa k a u }
13)           FOR  j:=2  TO  n  DO
                 { ajusta los costos de U }
14)              IF (C[k,j] < menor_costo[j]) AND
                    (menor_costo[j] < INFINITO) THEN
                 BEGIN
15)                   menor_costo[j] := C[k,j];
16)                   mas_cercano[j] := k;
                 END;
         END;
   END;
```

La complejidad de este algoritmo es $O(n^2)$, ya que se efectúan n-1 iteraciones del ciclo comprendido en las líneas 4-16 y cada iteración del ciclo lleva un tiempo O(n), debido a los ciclos más internos (líneas 7-10 y 13-6). Conforme n crece el algoritmo puede dejar de ser satisfactorio.

b) ALGORITMO DE KRUSKAL.

Este algoritmo supone que se tiene una gráfica conexa G(V,E) con V = { 1,2,3, ... } y una función de costo C definida en los arcos E. Para construir el árbol expandido de costo mínimo para G, el algoritmo inicia con una gráfica T(V,0) constituida por los vértices de G pero sin los arcos. Por tanto, cada vértice es un componente conectado por sí mismo; conforme el algoritmo se ejecuta, siempre existe una colección de componentes conectados y para cada componente se deben seleccionar los arcos que van formando el árbol expandido. Para construir componentes cada vez más grandes, se examinan los arcos a partir de E en orden creciente de acuerdo con el costo. Si el arco conecta dos vértices que se encuentran en dos componentes conectados distintos, entonces se agrega el arco a T. El arco se descartará si conecta dos vértices contenidos en el mismo componente ya que puede provocar un ciclo si se anexa al árbol expandido. Cuando todos los vértices están en un sólo componente, T es un árbol expandido de costo mínimo para G.

CODIFICACION DEL ALGORITMO DE KRUSKAL.

```
PROCEDURE KRUSKAL (v : conjunto_de_vértices; a : conjunto de
```

```
                    arcos; VAR t : conjunto_de_arcos);
VAR
     comp_n : INTEGER;              {número actual de componentes}
     arcos  : Cola_de_prioridad;    {conjunto de arcos  }
     componentes : Conjunto_de      {conjunto v agrupado en un
                   conjunto de componentes combina_encuentra   }
     u,v    : vértice;
     a      : arco;
     comp_siguiente : INTEGER       {nombre para el nuevo
                                     componente }
     comp_u, comp_v : INTEGER       {nombres de los componentes }
BEGIN
     ANULA(T);
     ANULA(ARCOS);
     comp_siguiente := 0;
     comp_n         := número de miembros de V;
     FOR  v en V  DO
     { Asigna un valor inicial a un componente para que
       contenga un vértice de V  }
     BEGIN
          comp_siguiente := comp_siguiente + 1;
          INICIAL (comp_siguiente,v,componente);
     END;
     FOR  a  en  A  DO
     {Asigna un valor inicial a la cola de prioridad de arcos }
          INSERTA(a,arcos);
     WHILE comp_n > 1 DO
       Considera el siguiente arco
     BEGIN
          a         := SUPRIME_MIN(arcos);
          sea  a    := (u,v);
          comp_u    := ENCUENTRA(u,componentes);
          comp_v    := ENCUENTRA(v,componentes);
          IF comp_u <> comp_v THEN
            Conecta dos componentes diferentes
          BEGIN
               COMBINA(comp_u, comp_v,componente);
               comp_n := comp_n -1;
               INSERTA (a,T);
          END;
     END;
END;
```

El rendimiento máximo de este algoritmo es O(alogа) donde a es el número de arcos de la gráfica dada. Se recomienda el empleo de este algoritmo cuando a es mucho menor a n^2; pero si es cercano a n^2, se debe optar por el algoritmo **PRIM**.

RECORRIDOS.

En la mayoría de los problemas involucrados con gráficas es necesario visitar sistemáticamente sus vértices; la **BUSQUEDA EN PROFUNDIDAD** y la **BUSQUEDA EN AMPLITUD**, son dos técnicas importantes para hacerlo, ambas pueden emplearse para determinar

cuales son los vértices que se encuentran conectados a un vértice dado.

a) BUSQUEDA EN PROFUNDIDAD.

Para efectuar la búsqueda en profundidad, se debe tener presente primero, que cada árbol de un bosque es un componente conectado de la gráfica y que si la gráfica es conectada tiene un sólo árbol en su bosque; segundo, que en una gráfica no dirigida sólo existen dos clases de arcos: arcos de árbol y arcos de retroceso (comprenden los arcos de avance y los arcos de retroceso por no existir distinción de ellos en las gráficas no dirigidas); Así mismo en una gráfica no dirigida no existen arcos cruzados, esto es, arcos (v,w) donde v no es antecesor ni antecedente de w en el árbol de extensión; si los hubiera, entonces, si v es un vértice alcanzado antes que w en la búsqueda, la llamada a **bpf(v)** no puede terminar hasta haber buscado w; así que w se introduce en el árbol como descendiente de v. De modo semejante, si **bpf(w)** se llama antes que **bpf(v)**, v se convierte en descendiente de w Por lo tanto durante una búsqueda en profundidad de una gráfica no dirigida G, los arcos sólo pueden ser :

1) **arcos de árbol** : arcos (v,w) tales que **bpf(v)** llama directamente a **bpf(w)** o viseversa, o bien

2) **arcos de retroceso** : arcos (v,w) tales que ni **bpf(v)**, ni **bpf(w)** se llaman directamente, pero uno llama indirectamente a a otro (es decir, **bpf(w)** llama **bpf(x)**, quien a su vez llama a **bpf(v)**, de modo que **w** es antecesor de v.

Ejemplo

Visitar la siguiente gráfica por el método de búsqueda en profundidad, iniciando la visita en el vértice **a**:

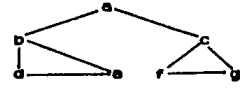


FIG 6.36

- El procedimiento **bpf(a)** llama a **bpf(b)** y anexa el arco (a,b) a T, ya que **b** no ha sido visitado.

- En **b**; **bpf** llama a **bpf(d)** y anexa el arco (d,e) a T.

- En **e**, los vértices a,b y d ya estan marcados como visitados, de modo que **bpf(e)** regresa sin incorporar ningún arco a T.

- En **d** **bpf** encuentra los vértices a y b marcados como visitados, así que **bpf(d)** regresa también sin anexar más arcos a T.

- En **b**, **bpf** encuentra los vértices adyacentes restantes a y a marcados como visitados, así que bpf(b) regresa. La búsqueda continúa después con c, f y g.

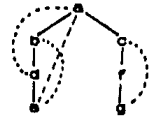


FIG 6.37

b) BUSQUEDA EN AMPLITUD.

Otra forma de visitar los vértices en forma sistemática es conocida como búsqueda en amplitud, denominada así porque desde cada vértice v que se visita se busca en forma tan amplia como sea posible, visitando todos los vértices adyacentes a v.

Al realizar una búsqueda en amplitud se puede construir un bosque de extensión; en este caso, se considera la arista (x,y) como el arco del árbol, si el vértice y es el que se visitó primero partiendo del vértice x.

El procedimiento para este método es el siguiente.

```
PROCEDURE bea(v);
{ Procedimiento que visita todos los vértices conectados a v
  empleando la búsqueda en profundidad }
VAR
    c    : cola_de_vértices;
    x,y :vértices;
BEGIN
    marca(v) := visitado;
    PONE_EN_COLA (v,c);
    WHILE  NOT VACIA(c)  DO
    BEGIN
         x := FRENTE(c);
         QUITA_DE_COLA(c);
         FOR cada vértice y adyacente a x DO
            IF  marca(y)  =  no_visitado  THEN
            BEGIN
                marca(y) := visitado;
                PONE_EN_COLA (v,c);
                INSERTA((x,y),T);
            END;
    END;
END;
```

Resulta que para este tipo de búsqueda, todo arco que no es un arco árbol, es un arco cruzado; esto es, conecta dos vértices

ninguno de los cuales es antecesor de otro. Si la gráfica no es conectada, este procedimiento debe llamarse desde un vértice de cada componente. Observe que en el algoritmo de búsqueda en amplitud se debe marcar cada vértice como visitado, antes de que se introduzca a la cola, para evitar con ello que el vértice se coloque en la cola más de una vez.

Ejemplo

Recorrer la siguiente gráfica, iniciando en el vértice a, por el método de búsqueda en amplitud.

FIG 6.38

La complejidad de tiempo de la búsqueda en amplitud es la misma que para la búsqueda en profundidad. Cada vértice visitado se coloca en la cola sólo una vez, así el ciclo WHILE se ejecuta sólo una vez para cada vértice. Cada arco (x,y) se examina dos veces, desde x y desde y. Si la gráfica tiene n vértices y a arcos, el tiempo de ejecución de este procedimiento es O(máx(n,a)) si se emplea una lista de adyacencia para los arcos.

PUNTOS DE ARTICULACION Y COMPONENTES BICONEXOS.

Una gráfica sin puntos de articulación (vértices que al ser eliminados desconectan la gráfica) se denomina gráfica biconexa; una gráfica G tiene conectividad K, si la eliminación de K-1 vértices cualesquiera no la desconecta. La búsqueda en profundidad, es una técnica muy útil para encontrar los componentes biconexos de una gráfica.

ALGORITMO DE BUSQUEDA EN PROFUNDIDAD PARA ENCONTRAR TODOS LOS PUNTOS DE ARTICULACION DE UNA GRAFICA Y PROBAR POR MEDIO DE SU AUSENCIA, SI LA GRAFICA ES BICONEXA.

1) Realizar una búsqueda en profundidad de la gráfica calculando **número_bp** v para todo vértice v. En esencia **número_bp** ordena los vértices como en un recorrido en orden previo del árbol de extensión en profundidad.

2) Para cada vértice v, obtener **bajo**[v] que es el número más pequeño de v o cualquier otro vértice w accesible desde v, siguiendo cero o más arcos de árbol hasta un descendiente x de v (x puede ser v), y después seguir un arco de retroceso (x,w). Se calcula **bajo**(v) para todos los vértices v, visitándolos en un recorrido posterior. Cuando se procesa v, se ha calculado **bajo**(y) para todo hijo y de v; se toma **bajo** v como el mínimo de:

a) número_bp[v]
b) número_bp[z] para cualquier vértice z, para el cual exista un arco de retroceso (v,z), y
c) bajo[y] para cualquier hijo y de v.

3) Encontrar los puntos de articulación como sigue:

a) La raíz es un punto de articulación si y sólo si tiene dos o más hijos. Puesto que no existen arcos cruzados, la eliminación de la raíz debe desconectar los subárboles cuyas raíces se encuentran en sus hijos.
b) Un vértice v distinto de la raíz es un punto de articulación si y sólo si, hay un hijo w de v tal que bajo [w] >= bp[v]. En este caso, v desconecta a w y sus descendientes del resto de la gráfica.

El tiempo que consume este algoritmo en una gráfica de a arcos y n <= a vértices es O(a); ya que el tiempo empleado en cada una de las tres fases puede atribuirse al vértice visitado o a un arco que parte de ese vértice, y sólo se le puede atribuir una cantidad constante de tiempo a cualquier arco o vértice en cualquier paso. Así, el tiempo total es O(n+(a)), el cual es O(a) si y sólo si n<=a.

EMPAREJAMIENTO DE GRAFICAS.

El problema del emparejamiento se puede formular en términos generales como sigue: dada una gráfica G=(V,E), el subconjunto de arcos de A, en el cual ningún par de arcos es incidente sobre un mismo vértice de V se conoce como emparejamiento. La tarea de formar subconjuntos máximos de tales arcos se denomina problema de emparejamiento maximal; un ejemplo de ello son los arcos punteados de la siguiente gráfica:

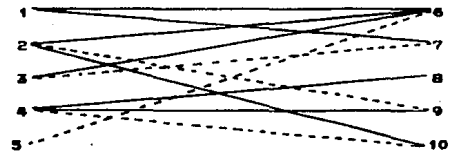


FIG 6.39

Un emparejamiento completo es aquel en el cual todo vértice es un punto final de algún arco que pertenece al conjunto de emparejamiento; claramente todo emparejamiento completo es maximal.

Un modo directo de encontrar emparejamientos maximales, es generar en forma sistemática todos los emparejamientos y luego marcar aquél que tenga el mayor número de arcos. La dificultad de

este método radica en su tiempo de ejecución, que es una función exponencial del número de aristas.

Existen algoritmos más eficientes para obtener emparejamientos maximales, basados en una técnica conocida como **"Técnica de caminos aumentados"**. Sea C un emparejamiento en una gráfica G; un vértice v esta emparejado si es adyacente a un arco de C. Un camino que desconecte dos vértices no emparejados cuyas aristas alternas estén en C, se conoce como camino aumentado relativo a C.

Un camino aumentado (A) debe tener longitud impar, empezar y terminar con arcos que no pertenecen a C; a partir de este tipo de caminos siempre es posible encontrar un emparejamiento más amplio, si se suprimen de C aquellos arcos que estén en A y se anexen después a C los arcos de A que no estaban inicialmente en C. este emparejamiento nuevo es C ⊕ A, donde ⊕ denota un "or exclusivo" en conjuntos, esto es, un nuevo emparejamiento que consta de todos los arcos que están en C o en A, pero no en ambos. Por lo tanto C es un emparejamiento maximal, si y sólo si, no existe un camino aumentado relativo a C.

Supóngase que C y D son emparejamientos con /C/ < /D/ (/C/ representa el número de arcos en C). Para demostrar que C ⊕ D contiene un camino aumentado relativo a C, considérese la gráfica G = (V, C⊕D), dado que C y D son emparejamientos, cada vértice de V es un punto final de sólo un arco de C y también es un punto final de sólo un arco de D. Así, cada componente de G forma un camino simple (tal vez un ciclo) con arcos alternando entre C y D. Cada camino que no sea un ciclo puede ser un camino aumentado relativo a C o un camino aumentado relativo a D, según quién tenga menos arcos. Ya que /C/ < /D/; C ⊕ D posee más arcos de D que de C, por lo que tiene por lo menos un camino aumentado relativo a C.

De todo esto se puede plantear el procedimiento para encontrar un emparejamiento maximal C para la grafica G=(V,A):

1) Iniciar con C=0;
2) Encontrar un camino aumentado A relativo a C y remplazar C por C⊕A.
3) Repetir el paso 2 hasta que ya no existan más caminos aumentados dado que C es ya un emparejamiento maximal.

Sólo falta demostrar como encontrar un camino aumentado relativo a un emparejamiento C; esto se hará para el caso más simple, en el que G es una gráfica bipartita (Gráfica en la cual el conjunto de vértices V, se puede particionar en dos conjuntos V', V'' de tal forma que a cada línea de la gráfica le corresponda un vértice terminal de V' y el otro vértice terminal de V'' . Donde V' y V'' son conjuntos disjuntos no vacíos) se construirá una gráfica de caminos aumentados para G en los niveles i=0,1,2,... por medio de un proceso similar al de la búsqueda en amplitud. El nivel i=0 se inicia en algún vértice sin

emparejamiento. En un nivel impar i, se anexan vértices nuevos adyacentes a un vértice en un nivel i-1, gracias a un arco del emparejamiento c, junto con ese arco.

Se continua construyendo la gráfica de caminos aumentados nivel a nivel hasta que se anexe un vértice sin emparejamiento en el nivel impar, o hasta que no se puedan anexar más vértices. Si un vértice sin emparejamiento v se anexa a un nivel impar, el camino existente entre v y el vértice inicial del nivel cero, será un camino aumentado empezando en un vértice inicial sin emparejamiento. Si no existen más vértices nuevos sin emparejamiento, entonces no existirá camino aumentado relativo a C. Si hay un camino aumentado, tarde o temprano se construirá una gráfica de caminos aumentados que termine en un vértice sin emparejamiento en el nivel impar.

Supóngase que G tiene n vértices y a arcos. La construcción de las gráficas de caminos aumentados para un emparejamiento dado, consume un tiempo O(a) si se representan los arcos por medio de una lista de adyacencia. Así, encontrar cada nuevo camino aumentado lleva un tiempo O(a). Para encontrar un emparejamiento maximal se construyen hasta n/2 caminos aumentados, ya que cada uno incrementa por lo menos con un arco el emparejamiento actual. Por tanto, puede encontrarse un emparejamiento maximal en un tiempo O(na) para una gráfica bipartida G.

6.1.6 GRAFICAS DIRIGIDAS.

Una gráfica dirigida (digráfica) G consiste de un conjunto de vértices V y un conjunto arcos dirigidos E. Un arco dirigido es, un par ordenado de vértices (v,w); v es el vértice inicial y w el vértice final. El arco dirigido (v,w) se expresa comúnmente como v -> w y se representa como:

Se dice que el arco v -> w va de v a w y que w es adyacente a v.

OPERACIONES CON GRAFICAS DIRIGIDAS.

Las operaciones más comunes en gráficas dirigidas incluyen desde la lectura del peso de un arco, la inserción y supresión de vértices y arcos, y el recorrido de los arcos desde un vértice inicial a un vértice final.

Si se emplea una matriz de adyacencia para representar una gráfica, un índice es un entero que representa un vértice adyacente y se requerirán de las tres operaciones siguientes para manipular la gráfica:

1) **PRIMERO(v)** : Devuelve el índice del primer vértice adyacente a v; si devuelve un vértice nulo, significa que no existe ningún vértice adyacente a v.
2) **SIGUIENTE(v,i)** : Devuelve el índice posterior al índice i

de los vértices adyacentes a v, un vértice nulo significa que i es el último de los vértices adyacentes a v.

3) **VERTICE (v,i)** : Devuelve el vértice i; el cual es un vértice adyacente a v.

PROBLEMA DE LOS CAMINOS MAS CORTOS CON UN SOLO ORIGEN.

Supóngase que G es una gráfica pesada dirigida G(V,E,W) y sea vi un vértice considerado como origen. El problema es determinar el costo del camino más corto del origen a todos los demás vértices de V donde la longitud de un camino es la suma de los costos de los arcos de un camino.

Para resolver este problema se manejará un proceso conocido como algoritmo de **DIJKSTRA**, que opera a partir de un conjunto S de vértices, cuya distancia más corta desde el origen ya es conocida Al iniciar el proceso S contiene sólo el vértice origen; en cada paso se anexa algún vértice restante v a S, cuya distancia desde el origen es la más corta posible.

En cada paso del algoritmo se registra la longitud del camino en una matriz D; Una vez que S incluye todos los vértices de G, D contendrá la distancia más corta del origen a cada vértice.

ALGORITMO DE DIJKSTRA.

```
PROCEDURE DIJKSTRA;

  {Este procedimiento calcula el costo de los caminos más
   cortos entre el vértice 1 y todos los demás vértices de
   una gráfica dirigida}
BEGIN
1)     S := 1 ;
2)     FOR i:=2 TO n DO
3)         d[i] := c[1,j]; {Asigna el valor inicial a D}
4)     FOR i:=1 TO n-1 DO
       BEGIN
5)         Elegir un vértice w en V-S tal que d w sea minimo;
6)         Agregar w a S;
7)         FOR cada vértice v en V-S DO
8)             d[v] := min(d[v], d[w]+c[w,v]);
       END;
END;
```

Ejemplo

Calculos obtenidos por el algoritmo DIJKSTRA:

ITERACION	S	W	d 2	d 3	d 4	d 5
inicio	{ 1 }	-	10	infinito	30	100
1	{ 1,2 }	2	10	60	30	100
2	{ 1,2,4 }	4	10	50	30	90
3	{ 1,2,4,3 }	3	10	50	30	60
4	{ 1,2,4,3,5 }	5	10	50	30	60

FIG 6.40

ANALISIS DEL ALGORITMO DIJKSTRA

Si se emplea una matriz de adyacencia para representar la gráfica pesada dirigida, el ciclo de las líneas 7 y 8 se ejecuta en un tiempo $O(n)$, y se ejecuta n-1 veces, consumiendo un tiempo total de $O(n^2)$.

Si el número de arcos es mucho menor que n^2, es conveniente emplear una representación con lista de adyacencia y una cola de prioridades aplicada a manera de árbol parcialmente ordenado para organizar los vértices de V-S. El ciclo de las líneas 7 y 8 se puede aplicar recorriendo la lista de adyacencia para w y actualizando las distancias en la cola de prioridad, se harán un total de a (número de arcos) actualizaciones, cada una con un costo de tiempo $O(\log n)$, por lo que el tiempo total consumido en las líneas 7 y 8, es ahora $O(a \log n)$.

Las líneas 1-3 consumen un tiempo $O(n)$, lo mismo que las líneas 4-6. Al manejar la cola de prioridad para representar V-S, las líneas 5 y 6 (la operación **SUPRIME-MIN** y cada una de las n-1 iteraciones de esas líneas) requieren un tiempo $O(\log n)$, como resultado, el tiempo total consumido por esta versión del algoritmo es $O(a \log n)$, el cual es mucho mejor que $O(n^2)$ si a es muy pequeña comparada con n^2.

PROBLEMA DE LOS CAMINOS MAS CORTOS ENTRE TODOS LOS PARES DE VERTICES.

Supóngase que se tiene una gráfica pesada dirigida cuyo peso representa el tiempo de vuelo para ciertas rutas entre ciudades, y se desea construir una tabla que contenga el menor tiempo requerido para volar entre dos ciudades cualesquiera. Esto es sólo un ejemplo del problema de los caminos más cortos entre todos los pares de vértices de una gráfica.

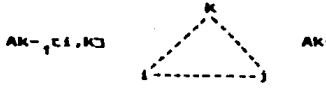
Para plantear el problema con precisión, se requiere de una gráfica dirigida $G=(V,E)$ en la que cada arco $v \rightarrow w$ tiene un costo no negativo $c(v,w)$; el problema consiste en encontrar el camino de longitud más corta entre v y w para cada par ordenado de vértices (v,w).

Inicialmente este problema podría resolverse por medio del algoritmo de **DIJKSTRA**, tomando por turno cada vértice origen; pero existe un método más directo creado por R.W Floyd denominado algoritmo de **FLOYD**; que supone que los vértices en V se encuentran numerados 1,2,...,n y emplea una matriz A de nxn en la que calcula las longitudes de los caminos más cortos. Inicialmente se hace A[i,j] = C[i,j] para toda i<>j. Si no existe un arco del vértice i al vértice j, se supone que c [i,j] = infinito. Cada elemento de la diagonal es igual a cero después, se realizan n iteraciones de tal forma que al final de la k-ésima iteración A[i,j] contendrá el valor de la longitud más pequeña de cualquier camino que vaya desde el vértice i hasta el vértice j y que no pase por un vértice etiquetado con un número mayor que k ; esto es i y j son los vértices extremos del camino y pueden ser cualquier vértice; pero todo vértice intermedio debe ser menor o igual a k.

Para calcular A, en la k-ésima iteración se aplica la siguiente fórmula:

$$A_k(i,j) = \min \begin{cases} A_{k-1}(i,j) \\ A_{k-1}(i,K) + A_{k-1}(K,j) \end{cases}$$

Para obtener Ak[i,j], se compara Ak_{-1}[i,j] (el costo de ir de i a j sin pasar por k o cualquier otro vértice con numeración mayor), con Ak_{-1}[i,k] + Ak_{-1}[k,j] (costo de ir primero a k y después de k a j, sin pasar a través de un vértice con número mayor a k). Si el paso por el vértice k produce un camino más económico que el de Ak_{-1} [i,j] se elige ese costo para Ak[i,j] . Gráficamente:



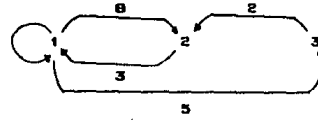
Inclusión de K entre los vértices que van de i a j.

Dado que Ak[i,k] = Ak_{-1}[i,k] y Ak[k,j] = Ak_{-1}[k,j] ninguna entrada con un subíndice igual a k cambia durante la k-ésima iteración.

```
PROCEDURE FLOYD (VAR a: ARRAY [1..n,1..n] OF REAL;
                 c: ARRAY [1..n,1..n] OF REAL);
  FLOYD calcula la matriz A de caminos más cortos dada la
  matriz de costos de arcos C
VAR
   i,j,k : INTEGER;
BEGIN
     FOR  i:=1  TO  n  DO
        FOR  j:=1  TO  n  DO
           a [i,j] :=c[i,j];
```

```
FOR  i:=1  TO  n  DO
     a[i,j]:=0
FOR  k:=1  TO  n  DO
     FOR  j:=1  TO  n  DO
          IF a[i,k] + a[k,j] < a[i,j] THEN
               a[i,j] := a[i,k] + a[k,j];
END;
```

Ejemplo



Valores de la matriz A en cada iteración:

$$A_0[i,j] = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & \infty \\ \infty & 2 & 0 \end{pmatrix} \qquad A_1[i,j] = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ \infty & 2 & 0 \end{pmatrix} \qquad A_2[i,j] = \begin{pmatrix} 0 & 8 & 5 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{pmatrix} \qquad A_3[i,j] = \begin{pmatrix} 0 & 7 & 8 \\ 3 & 0 & 8 \\ 5 & 2 & 0 \end{pmatrix}$$

FIG 6.41

Es evidente que el tiempo de ejecución de este método es $O(n^3)$, ya que se encuentra formado por tres ciclos **FOR** anidados.

COMPARACION DE LOS ALGORITMOS DE FLOYD Y DIJKSTRA.

Dado que el algoritmo de **DIJKSTRA** y el de **FLOYD** pueden encontrar los caminos más cortos desde un vértice, con un tiempo $O(n^2)$ y $O(n^3)$ respectivamente, la experimentación y medición del compilador, la máquina y los detalles de aplicación (Factor constante), es la manera más fácil de determinar cual es el mejor algoritmo para la aplicación en cuestión.

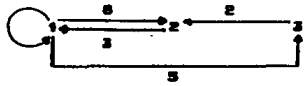
Si el número de arcos (a) es mucho menor que n^2 aún con un factor constante relativamente pequeño en el tiempo de ejecución de **FLOYD**, la versión de **DIJKSTRA** con tiempo **O(nalogn)** será superior, al menos para gráficos grandes y poco densos.

CAMINOS DE LONGITUD MINIMA ENTRE DOS VERTICES.

En algunas aplicaciones se desea conocer el camino de costo mínimo entre dos vértices. Un modo de lograrlo es empleando una matriz P, donde **P[i,j]** tiene el vértice k que permitió por medio del algoritmo de FLOYD, encontrar el valor más pequeño **A[i,j]**. Si **P[i,j] = 0**, el camino de costo mínimo de **i** a **j** es directo, siguiendo el arco entre ambos.

```
PROCEDURE CAMINO_MAS_CORTO(VAR a:ARRAY[1..n,1..n] OF REAL;
                               c:ARRAY[1..n,1..n] OF INTEGER);
{Procedimiento que dada una matriz de costos C de  nxn; produce
una matriz A de nxn de longitudes de caminos más cortos mínimos y
una matriz P de nxn que contiene un vértice intermedio del camino
de costo mínimo entre un par de vértices}
  VAR
     i,j,k : INTEGER;
  BEGIN
       FOR  i:=1  TO  n  DO
           FOR  j:=1  TO  n  DO
           BEGIN
                a[i,j] := c[i,j];
                p[i,j] := 0;
           END;
       FOR  i:=1  TO  n  DO
           a[i,i] := 0;
       FOR  k:=1  TO  n  DO
           FOR  i:=1  TO  n  DO
               FOR  j:=1  TO  n  DO
                   IF a[i,k] + a[k,j] < a[i,j]  THEN
                   BEGIN
                        a[i,j] := a[i,k] + a[k,j];
                        p[i,j] := k;
                   END;
  END;
```

Ejemplo



$$P = \begin{bmatrix} 0 & 3 & 0 \\ 0 & 0 & 1 \\ 2 & 0 & 0 \end{bmatrix}$$

P 1,2 = 3 Indica que 7 es el costo mínimo del camino del vértice 1 a 2 y que 3 es un vértice intermedio en ese camino.

FIG 6.42

```
  PROCEDURE CAMINO(i,j : INTEGER);
  {Imprime los vértices intermedios del camino de costo mínimo
    del vértice i al vértice j}
  VAR
      k: INTEGER;
  BEGIN
       k := p[i,j];
       IF  k:=0  THEN
           RETURN;
       CAMINO(i,k);
       WRITELN(k);
       CAMINO(k,j);
  END;
```

CIERRE TRANSITIVO.

En algunos problemas podría resultar interesante saber si existe un camino de longitud igual o mayor que el camino de costo mínimo (longitud mínima), que va del vértice i al vértice j. este proceso se conoce como algoritmo de Warshall y fue diseñado en el año de 1962.

Supóngase que la matriz de costo C es sólo la matriz de adyacencia para la gráfica dirigida dada. Esto es C[i,j]=1, si existe un arco de i a j, y 0 si no lo existe. Se desea obtener una matriz A tal que a c i,j=1 si existe un camino de longitud igual o mayor que el camino de longitud menor de i a j, y 0 en otro caso. A se conoce a menudo como **CIERRE TRANSITIVO** de la matriz de adyacencia.

El cierre transitivo puede obtenerse por medio del procedimiento **FLOYD** si se aplica la siguiente fórmula en el k-ésimo paso:

$$A^k[i,j] = (A^{k-1}[i,j]) \text{ OR } (A^{k-1}[i,k] \text{ AND } A^{k-1}[k,j])$$

Esta fórmula establece que existe un camino de i a j que no pasa por un vértice con un número mayor que k si

a) Ya existe un camino de i a j que no pasa por un vértice con número mayor que k-1, o si
b) Existe un camino de i a k que no pasa por un vértice con número mayor que k-1, y un camino de k a j que no pasa por un vértice con número mayor que k-1.

Al igual que el algoritmo de **FLOYD** $A^k[i,k] = A^{k-1}[i,k]$ y $A^k[k,j] = A^{k-1}[k,j]$, así que se puede realizar el cálculo con sólo una copia de la matriz A.

```
PROCEDURE WARSHALL(VAR a:ARRAY[1..n,1..n] OF BOOLEAN;
                       c:ARRAY[1..n,1..n] OF BOOLEAN);
{Este procedimiento convierte a A en el cierre transitivo de C}
 VAR
     i,j,k : INTEGER;
 BEGIN
      FOR  i:=1  TO  n  DO
          FOR  j:=1  TO  n  DO
               a[i,j] := c i,j ;
      FOR  k:=1   TO  n  DO
          FOR  j:=1  TO  n  DO
               IF a[i,j] = FALSE  THEN
                  a[i,j] := a[i,k] AND a[k,j];
END;
```

EXCENTRICIDAD DE UNA GRAFICA DIRIGIDA.

Sea v un vértice de una gráfica dirigida G=(V,E); se define la excentricidad de v como:

máx | longitud mínima de un camino de w a v | para w = 1 -> n

w en v

El centro de G es el vértice de mínima excentricidad; Así, el centro de una gráfica dirigida es el vértice más cercano al vértice más distante.

Para encontrar el centro de una gráfica dirigida, suponiendo que C es la matriz de costos de G, se aplica el siguiente proceso:

a) Aplicar el algoritmo FLOYD a C para obtener la matriz A de los caminos más cortos entre todos los pares.
b) Encontrar el costo máximo en cada columna i, es decir determinar la excentricidad de los vértices.
c) Determinar el centro de G, es decir el vértice cuya excentricidad sea mínima.

Los tiempos de ejecución de cada paso de este proceso son los siguientes : primer paso $O(n^3)$; segundo paso $O(n^2)$ y tercer paso $O(n)$ por lo que el algoritmo consume un tiempo proporcional a $O(n^3)$.

Ejemplo

Determinar el centro de la siguiente gráfica:

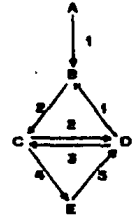


FIG 6.43

a)

	a	b	c	d	e
a	0	1	3	5	7
b	∞	0	2	4	6
c	∞	3	0	2	4
d	∞	1	3	0	7
e	∞	6	8	5	0

Matriz de los caminos de costos mínimos para todo par de vértices de G.

b) La excentricidad de los vértices es:

VERTICES	EXCENTRICIDAD
a	
b	6
c	8
d	5
e	7

c) El vértice centro de la gráfica es d.

RECORRIDO DE LAS GRAFICAS DIRIGIDAS.

Para resolver con eficiencia algunos problemas relacionados con gráficas dirigidas, se requiere muchas veces de visitar los vértices y los arcos de forma sistemática, con el objeto de recuperar o procesar la información que contienen; una técnica importante para lograrlo es la búsqueda en profundidad, que es una generalización del recorrido en orden previo de un árbol, y puede servir como base para construir algoritmos más eficientes.

Supóngase que se tiene una gráfica dirigida G en la que todos los vértices están marcados inicialmente como no visitados. La búsqueda en profundidad selecciona un vértice v de G como vértice de partida; lo marca como visitado y, después recorre cada vértice adyacente a v no visitado, emplea en forma recursiva la búsqueda en profundidad. Una vez que se han visitado todos los vértices que se pueden alcanzar desde v, la búsqueda de v está completa. Si algunos vértices quedan sin visitar se selecciona alguno de ellos como un nuevo vértice de partida y se repite este proceso hasta que todos los vértices de G se hayan visitado.

El método debe su nombre a la forma en que realiza su proceso ya que busca los nodos en dirección hacia adelante (tan profundo como sea posible).

Para implementar este algoritmo ; ede usarse una lista de adyacencia L v para representar los vértices adyacentes al vértice v, y una matriz marca cuyos elementos sean del tipo visitado o no visitado para determinar si un vértice ya fue visitado antes.

Para poder invocar al procedimiento recursivo BPF es necesario iniciar marca con el valor no visitado; posteriormente iniciar el método de búsqueda en profundidad para cada vértice.

```
FOR  v:=1  TO  n  DO
     marca[v] := no_visitado;
FOR  v:=1  TO  n  DO
     IF  marca[v] = no_visitado  THEN
         BPF (v);

PROCEDURE BPF(v: vértice);
```

```
   VAR
       w : vértice;
   BEGIN
1)      marca[v] := visitado;
2)      FOR  cada vértice w en l v   DO
3)           IF  marca[w]= no_visitado  THEN
4)               BPF(w);
   END;
```

Todas las llamadas realizadas al procedimiento **BPF** en una gráfica con a arcos y n<=a vértices consume un tiempo O(a) ya que ningún vértice llama a **BPF** más de una vez dado que se actualiza marca v con el valor visitado, y nunca un vértice que tiene su marca igual a visitado llama **BPF**. Así el tiempo total que se emplea en 2 y 3 recorriendo las listas de adyacencia es proporcional a la suma de las longitudes de tales listas, esto es O(a). Suponiendo que n<=a, el tiempo total de ejecución del método de búsqueda en profundidad de una gráfica completa es O(a), lo cual es, simplemente el tiempo necesario para recorrer cada arco.

BOSQUE EXPANDIDO EN PROFUNDIDAD.

Durante el recorrido en profundidad de una gráfica dirigida, ciertos arcos llevan a vértices sin visitar; estos arcos se conocen como **ARCOS DE ARBOL** y juntos forman un bosque expandido en profundidad para la gráfica en cuestión.

Además de los arcos de árbol, existen otros tres tipos de arcos definidos por una búsqueda en profundidad de una gráfica dirigida: **ARCOS DE RETROCESO, ARCOS DE AVANCE y ARCOS CRUZADOS**; los cuales definiremos con la ayuda del siguiente esquema:

Gráfica **Bosque expandido en profundidad**

FIG 6.44

Un arco como C -> A se denomina arco de retroceso, porque va de un vértice descendiente a un vértice antecesor del bosque expandido (posiblemente así mismo); B -> C es un arco de avance ya que el arco va de un antecesor a un descendiente propio, pero no es arco de árbol.

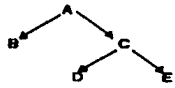
Los arcos como D -> C o G -> D que van de un vértice a otro que no es antecesor ni descendiente se conocen como arcos cruzados.

La búsqueda en profundidad debe asignar a todos los descendientes de un vértice v, números mayores o iguales al número asignado a v. De hecho w será descendiente de v, si y sólo si, **número p(v) <= número(w) < el número de descendientes de v.** Así las aristas de avance irán de los vértices de baja numeración a los de alta numeración; los arcos de retroceso irán de los vértices de alta numeración a los de baja numeración.

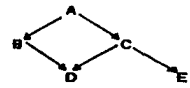
Todos los arcos cruzados irán de los vértices de alta numeración a los de baja numeración. Para ejemplificar esto, supóngase que **x -> y** es un arco y **número p(x) <= número p(y)**. Así **x** se visitó antes que a **y**. Todo vértice visitado entre la invocación por primera vez a **BPF(x)** y el momento en que **BPF(x)** termina, se convierte en descendiente de **x** en el bosque expandido. Si y permanece sin visitar cuando se explora el arco **x -> y**, se vuelve un arco de árbol; de otra forma, **x -> y** no puede ser un arco cruzado con número p(y) <= número p(y).

GRAFICAS DIRIGIDAS ACICLICAS.

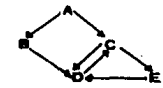
Una gráfica dirigida acíclica o GDA es una gráfica dirigida sin ciclos; son gráficas más generales que los árboles, pero menos que las gráficas dirigidas arbitrarias; es decir todo árbol es un GDA y todo GDA es una gráfica, pero no toda gráfica es un GDA, ni todo GDA es un árbol.



Árbol



Gráfica dirigida
Acíclica (GDA)



Gráfica dirigida

FIG 6.45

Los GDA son útiles para la representación de órdenes parciales. Un orden parcial R en un conjunto S es una relación binaria, tal que :

1) Para toda **a** que pertenezca a S, **aRa** es falso (irreflexiva), y
2) Para toda **a,b,c** en **S**, si **aRb** y **bRc** entonces **aRc** (R es transitiva)

Dos ejemplos naturales de órdenes parciales son la relación **"menor que"** (<) en enteros, y la relación de inclusión en conjuntos (C).

Ejemplo

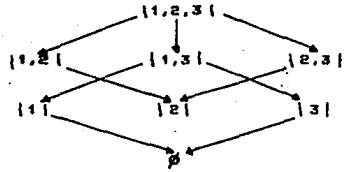
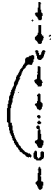


FIG 6.46 GDA para inclusiones propias.

PRUEBA DE ACICLICIDAD.

Para determinar si una gráfica es acíclica, esto es, si G no contiene ciclos puede emplearse la búsqueda en profundidad, ya que si se encuentra un arco de retroceso durante la búsqueda en profundidad, la gráfica G tiene un ciclo.

Supóngase que G es acíclica, si se efectúa la búsqueda en profundidad se debe encontrar un vértice v que tenga el número de búsqueda en profundidad menor que un vértice u en el ciclo; es decir, sea el arco u ->v en algún ciclo que contenga a v, ya que u está en el ciclo, debe ser descendiente de v en el bosque expandido en profundidad. Así u->v no puede ser un arco cruzado, puesto que el número en profundidad de u es mayor que el de v, por lo mismo u->v tampoco puede ser un arco de árbol ni un arco de avance; Así que u->v debe ser un arco de retroceso.



Todo ciclo tiene un arco de retroceso.

FIG 6.47

ORDEN TOPOLOGICO.

Una operación fundamental en las gráficas es procesar sus vértices en un orden tal que algunos de ellos, no puedan ser procesados si antes no fueron procesados otros; por ejemplo, si se tiene un proyecto grande éste suele dividirse en un conjunto de tareas pequeñas, algunas de las cuales se deben realizar en cierto orden específico, de modo que se pueda culminar el proyecto total. Los GDA es la estructura que suele emplearse para representar de forma natural esas situaciones.

Así el orden topológico es un proceso de asignación de un orden lineal a los vértices de un GDA, tal que si existe un arco del vértice i al vértice j, i aparece antes que j en el ordenamiento lineal.

El siguiente procedimiento genera el orden topológico de un GDA en orden inverso:

```
PROCEDURE CLASIFICACION_TOPOLOGICA (v: vértice);
  Imprime los vértices accesibles desde v en orden topológico
  invertido
VAR
   w : vértice;
BEGIN
     marca v  := visitado;
     FOR cada vértice w en l v    DO
        If marca w  = no_visitado THEN
           CLASIFICACION_TOPOLOGICA(w);
        WRITELN(v);
END;
```

El tiempo de ejecución del algoritmo es O(a) donde a es el número de arcos, si y sólo si a>=n.

COMPONENTES FUERTEMENTE CONECTADOS EN UNA GRAFICA DIRIGIDA.

Un componente fuertemente conectado de una gráfica dirigida es un conjunto maximal de vértices en el cual existe un conjunto que va desde cualquier vértice del conjunto hasta cualquier otro vértice también del conjunto. La búsqueda en profundidad puede emplearse para determinar con eficiencia los componentes fuertemente conexos de una gráfica dirigida.

Sea G=(V,E) una gráfica dirigida; V se puede dividir en clases de equivalencia Vi, 1<=i<=r, tal que los vértices u y w son equivalentes, si y sólo si, existe un camino de u a w y otro de w a u. Sea Ai, 1<0i<=r, el conjunto de arcos u->w tal que u y w pertenecen a Vi. Las gráficas (Vi,Ai) se denominan componentes fuertemente conectados (o sólo componentes fuertes) de G. Una gráfica constituida por sólo un componente fuerte se dice que es una gráfica fuertemente conectada.

Ejemplo

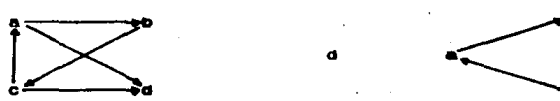


FIG 6.48

Todos los componentes de una gráfica dirigida G, pertenecen a algún componente fuertemente conectado de la gráfica, pero

ciertos arcos, llamados arcos de cruce de componentes, no pertenecen a ningún componente, ya que son arcos que van de un vértice que pertenece a un componente a otro vértice de un componente distinto. Para representar las interconexiones entre los componentes fuertemente conectados de una gráfica se puede construir una gráfica dirigida de G, cuyos vértices (Ci) son los componentes fuertemente conectados de G. En este tipo de gráfica existe un arco Ci->Cj, si existe un arco en G que vaya de algún vértice del componente Ci a algún otro del componente Cj.

La gráfica reducida siempre es un GDA, porque si existiera un ciclo, todos los componentes del ciclo serán en realidad un sólo componente fuerte, lo cual significará que no se calcularon en forma correcta los componentes fuertemente conectados.

La gráfica reducida del ejemplo anterior es:

FIG 6.49

Un posible proceso para encontrar los componentes fuertemente conectados de una gráfica dirigida es:

1) Efectuar una búsqueda en profundidad de G y numerar los vértices en el orden de terminación de las llamadas recursivas; esto es, asígnese un número al vértice v después de la línea 4 del procedimiento BPF.

2) Constrúyase una gráfica dirigida nueva Gr, invirtiendo las direcciones de todos los arcos de G.

3) Realícese una búsqueda en profundidad en Gr, partiendo del vértice con numeración más alta de acuerdo a la numeración asignada en el paso 1, si la búsqueda en profundidad no llega a todos los vértices, iníciese la siguiente búsqueda a partir del vértice restante con la numeración más alta.

4) Cada árbol del bosque de extención resultante es un componente fuertemente conectado de G.

Ejemplo

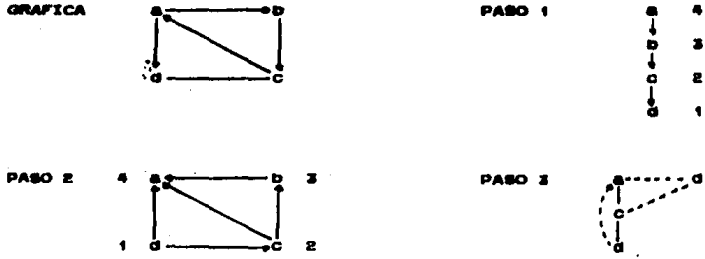


FIG 6.50

FLUJO DE REDES.

Las gráficas pesadas dirigidas son usadas como modelos de diferentes tipos de aplicaciones: una de estas aplicaciones son las **REDES.**

Se define una red como una gráfica pesada dirigida G(V,E) con dos vértices especiales: Uno llamado vértice origen y otro denominado vértice destino. El peso de los arcos es positivo y es llamado capacidad del arco.

Dada la red N, se define el flujo en N como el conjunto de pesos en los arcos tal que:

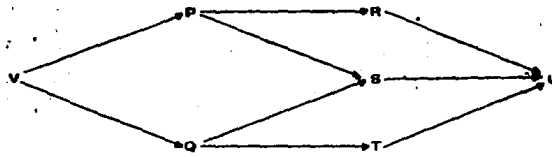
1) El flujo en cada arco debe ser mayor o igual a 0.

2) El flujo que entra a cada vértice es igual al flujo que sale de ese vértice (propiedad conservativa).

3) El flujo de cada arco debe ser menor o igual a la capacidad del arco.

El vértice origen (v) de la red N es aquel vértice que no tiene líneas incidentes hacia adentro y el vértice destino (u) es el vértice que no tiene líneas incidentes hacia afuera, para estos vértices el flujo que sale de v debe ser igual al flujo que llega a u, es decir, la suma del flujo de los arcos que salen de v es igual a la suma del flujo de los arcos que llegan a u; a esta suma se le denomina valor del flujo.

Ejemplo



	CAPACIDAD DE LOS ARCOS:	
vértice origen v	v -> p = 3	q -> t = 3
vértice destino u	v -> q = 6	r -> u = 5
	p -> r = 5	s -> u = 5
	p -> s = 4	t -> u = 1
valor del flujo : 6	q -> s = 2	

FIG 6.51

El problema del flujo consiste en encontrar un flujo con valor máximo para una red dada y se encuentra relacionado con el concepto de **CORTE**, el cual es un conjunto de líneas A que pertenecen a E con la propiedad de que cada trayectoria del vértice origen al vértice destino incluye una línea del conjunto A; en otras palabras un corte es un conjunto desconectador en la gráfica correspondiente.

La capacidad del corte se define como la suma de las capacidades de cada línea que pertenece al corte; y por el teorema del **FLUJO MAXIMO-CORTE MINIMO**, se tiene que el valor del flujo máximo es igual a la capacidad del corte mínimo.

En el ejemplo anterior

CORTES	CAPACIDAD DEL CORTE
(v->p,v->q)	9
(p->r,p->s,q->s,q->t)	14
(r->w,s->w,t->w)	11
(v->p,q->s,t->w)	6

Por lo tanto el flujo máximo de la red es 6.

El desarrollo clásico del problema del flujo de redes fue propuesto por L. R. Ford y D. R. Fulkerson en 1962. Su método inicia con un flujo cero, y se aplica repetidamente tantas veces como pueda ser aplicado, produciendo un incremento de flujo en cada iteración; si ya no se puede incrementar éste, el flujo máximo ha sido encontrado.

En el manejo de redes encontramos dos tipos de arcos: **ARCOS ADELANTADOS** : que son arcos con flujo del vértice origen, al vértice destino a lo largo de un camino particular y **ARCOS DE**

RETROCESO que son arcos con flujo desde el vértice destino al vértice origen. El método Ford-Fulkerson se basa en la manipulación de estos arcos, es decir, inicia con un flujo cero e incrementa el flujo a lo largo de un camino del origen al destino, con arcos adelantados no llenos o arcos vacios de retroceso, se continúa aplicando este criterio hasta que no existan caminos en la **RED**, pero esto no es un algoritmo en el sentido de que el método para encontrar los caminos no esta descrito.

Edmonds y Karp emplearon la búsqueda en profundidad para encontrar los caminos, entonces, el número de caminos encontrados antes de obtener el flujo máximo en una red con v vértices y e arcos puede ser menor que v.

El problema de flujo de redes puede ser extendido a varios tópicos y muchas variaciones de este problema han sido estudiadas porque son importantes en aplicaciones actuales.

6.2 ALGORITMOS PARA ESTADISTICA Y PROBABILIDAD.

6.2.1 ALGORITMOS PARA ESTADISTICA.

La estadística es una mina de oro, desde el punto de vista de la complejidad computacional ya que proporciona una rica y extensa fuente de algoritmos y procedimientos computacionales aún no analizados. Sin embargo, para los Estadísticos la búsqueda de algoritmos eficientes no ha sido de principal interés debido a varias razones:

1o. El diseño de algoritmos rápidos es un arte nuevo el cual se esta desarrollando.

2o. Hasta recientemente, el costo de la obtención de los datos ha sido mucho mayor que el costo de analizar estos, no obstante, que el procesamiento hablado y de imagen proporciona información a programas de análisis estadístico en una forma rápida y barata.

3o. Los estadísticos están propiamente interesados en la significancia y la efectividad de las pruebas que realizan, antes que con su costo.

Por todo lo anterior, el resultado es que el análisis de algoritmos estadísticos permanece en su mayor parte ignorado.

En esta sección se presenta un estudio sistemático de algunos algoritmos de estadística básica comúnmente denominados "recetas estadísticas".

NOCIONES ELEMENTALES DE PROBABIBLIDAD Y ESTADISTICA.

PROBABILIDAD: Es la propiedad que tiene todo aquello en lo que hay buenas razones para creer que se efectuará o sucederá.

Sea S = $(s_1, s_2, s_3, \ldots, s_n)$ el espacio muestral de un experimento, se supone que S contiene un número finito o infinitamente contable de elementos. Un evento es algún subconjunto del espacio muestral, usualmente es expresado en forma verbal y transladado entonces a un subconjunto de S.

Una ejecución sencilla de un experimento es conocida como una prueba E, que es un evento definido en el espacio muestral S. Si el resultado de una prueba o ensayo del experimento está en E, entonces se dice que el evento E ha ocurrido. Solamente un resultado s en S puede ocurrir en alguna prueba; pero todo evento que esté incluido en S ocurrirá.

Tales eventos pueden estar formados de otros eventos, que fueron formados al emplear el conjunto estandar de operaciones de unión, intersección y complementación.

Verbalmente, la unión U se lee como "AND", la intersección como "OR", y la complementación como "NOT".

Si E_1 y E_2 son dos eventos disjuntos cualesquiera (esto es, $E_1 \cap E_2 = 0$), se llaman **MUTUAMENTE EXCLUSIVOS**. Si E_1 y E_2 son mútuamente exclusivos, no es posible que ambos eventos ocurran en la misma prueba o ensayo.

Ahora bien tratemos de formular el concepto intuitivo estandar de probabilidad con mencionado anteriormente: se esta interesado en la probabilidad de que un evento dado suceda como un resultado de un experimento $\mathcal{E}$. El experimento es repetido N veces, donde N es un número grande. Cada prueba produce un resultado en S. El resultado está en E, en cuyo caso este evento ha ocurrido en n de las N pruebas. Entonces definiremos la probabilidad del evento E como P(E)=n/N.

Este número puede ser interpretado como la fracción de veces en que el evento E ocurrirá cuando el experimento es ejecutado. Esta es la esencia de la interpretación de frecuencia de probabilidad.

La teoría axiomática define todas las reglas computacionales empleadas para calcular la probabilidad. para lo cual, S será un espacio muestral para un experimento $\mathcal{E}$. P será una medida de probabilidad asociada con S, la cual asigna a ciertos eventos E S un número real P(E), el cual es llamado la probabilidad del evento E. Estas probabilidades deben satisfacer tres reglas básicas (axiomas):

I. $P(E) \geq 0$ para alguna $E \subset S$.

II. $P(S) = 1$

III. $P(E_1 \cup E_2) = P(E_1) + P(E_2)$, si E_1 y E_2 son mutuamente exclusivos, o sea, si $E_1 \cap E_2 = 0$.

Una vez que a estos "ciertos" eventos se les asignan probabilidades consistentes con estos axiomas, entonces es posible calcular la probabilidad de algún otro evento definido en S en base a las asignaciones realizadas y con la ayuda de los tres axiomas.

Probablemente, el aspecto más difícil de la construcción de modelos probabilísticos es asignar probabilidades a los eventos. Debido a que la Teoría General no nos dice cómo mirar un problema específico y decidir que eventos usar y qué probabilidad asignarles, estas elecciones se dejan por completo al analista.

En resumen,

1. Identificar el espacio muestral S, S a menudo contendrá un número finito de elementos. Elegir S de modo que todos sus elementos sean mutuamente exclusivos y exhaustivos colectivamente, esto es, dos elementos no pueden ocurrir simultáneamente y sólo un elemento ocurrirá en cualquier prueba. Formalmente, un conjunto de subconjuntos $A_1, A_2, \ldots An$ de S es mutuamente exclusivo y exhaustivo colectivamente si $Ai \cap Aj = 0$ para alguna $i \neq j$, y $\bigcup_{i=1}^{n} Ai = S$.

2. Asignación de probabilidades a los elementos en S. Esta asignación debe ser consistente con los axiomas I, II y III. Cumpliendo con el apartado 1, las probabilidades deberán ser asignadas a los elementos en S tal que la suma de todas las asignaciones sumen 1.

3. Identificar los elementos solución. Transladar o traducir la solución deseada en eventos en S.

4. Calcular las probabilidades deseadas, empleando las reglas computacionales (axiomas I, II, y III).

Por otra parte, dentro de las fórmulas probabilísticas, existe una que es muy importante, conocida como la fórmula de probabilidad condicional: sean E y F dos eventos definidos en un espacio muestral S, y P(E) y P(F) las probabilidades asociadas con estos eventos. P(E\F) denotará la probabilidad del evento E dado que el evento F ha ocurrido.

Usualmente, $P(E) \neq P(E \setminus F)$, una vez que F ha ocurrido, el cálculo de la probabilidad de E no es tan grande, ya que sólo es necesario considerar los elementos de S que impliquen la ocurrencia de F. Para esto se emplea la fórmula

$$P(E\backslash F)=P(E\cap F)/P(F) \qquad \text{definida sólo si } P(F) <> 0$$

Pero, ¿Qué sucede si la ocurrencia de F no tiene ninguna relación o referencia con la probabilidad de E? Entonces se tiene que

$$P(E)=P(E\backslash F) \text{ pero } P(E\backslash F)=P(E\cap F)/P(F)$$

por lo tanto, $P(E) = P(E\cap F)/P(F)$ o $P(E\cap F)=P(E)P(F)$

Si los eventos E y F son tales que la probabilidad de ocurrencia de ambos es igual al producto de sus probabilidades de ocurrencia por separado, se dice que estos eventos son independientes. Esta fórmula es extendé al caso de tres o más eventos mutuamente independientes; esto es, si los conjuntos A_1, A_2, ..., An son tales que $P(Ai\cap Aj)=P(Ai)P(Aj)$ para alguna $i<>j$, entonces $P(A_1\cap A_2\cap A_3\cap \ldots \cap An) = P(A_1)P(A_2)P(A_3)\ldots P(An)$.

En muchos problemas es posible identificar una o más cantidades valoradas reales que proveen información más conveniente que las descripciones explícitas de eventos Cualquier función del espacio muestral (y su medida de probabilidad asignada) de un problema a un conjunto de números reales es llamada variable **ALEATORIA**. Todos los cálculos probabilísticos serios son hechos en términos de variables aleatorias.

Una asignación de probabilidad al rango de una variable aleatoria se llama distribución. Una variable aleatoria cuyo rango consiste de un número contable de posibles valores se denomina discreta. Si una variable aleatoria Y asume uno de los valores $y_1, y_2, y_3, \ldots yn$ entonces

$$P(Y=yi) > 0 \quad \text{por lo tanto} \quad i=1,2,..n$$

$$P(Y=y) = 0 \quad \text{para todos los valores de y.}$$

A las distribuciones discretas de este tipo se les llama **funciones de masa de probabilidad**; ya que Y podría tomar uno de los valores yi en el experimento, la función masa podría satisfacer la relación

$$\sum_{i=1}^{n} P(Y=yi) = 1$$

Es decir, la probabilidad de que Y iguale a uno de sus valores permitidos es 1 (note que Y no puede igualar más que uno simultáneamente).

Ahora, el valor esperado, o valor promedio, o medio, de una variable aleatoria Y es definida como:

$$E[Y]=\sum_{i=1}^{n} yiP(Y=yi) \qquad (1)$$

En términos de la interpretación de frecuencias de probabilidad, la media es el valor promedio de la variable

aleatoria observada sobre un gran número de pruebas, cuya media no se encuentra necesariamente en el rango de la variable aleatoria.

La utilidad del valor esperado se incrementa cuando el valor actual tomado por la variable aleatoria es improbable, o sea, se desvía mucho de la media.

Una medida de esta desviación es conocida como la varianza de la variable aleatoria. Si V es una variable aleatoria con media $\mu = E(V)$. Entonces la varianza de V, denotada Var(V), se define como

$$Var(V) = E(V-\mu)^2 = \sum_{i=1}^{n} (y_i-\mu)^2 p(V=y_i) \qquad (2)$$

en la cual se ha usado la fórmula para el valor esperado de la función de una variable aleatoria. La desviación estándar de una variable aleatoria se define como:

$$\sigma(V) = (Var(V))^{1/2}$$

Por otra parte, los problemas estadísticos en su mayoría son problemas "reservados" de probabilidad. Donde primero se estudia el comportamiento o funcionamiento del sistema y posteriormente, se trata de bosquejar inferencias (suposiciones) acerca de las distribuciones de las variables aleatorias que describen la actividad del sistema.

Por ejemplo, si tomamos el algoritmo de ordenamiento por inserción directa, podemos definir una variable aleatoria Y la cual iguala, para alguna secuencia aleatoria de N enteros, el número de comparaciones requeridas por este algoritmo para ordenar la secuencia. Toda aplicación del algoritmo de ordenamiento por inserción directa es una secuencia aleatoria de enteros por tanto produce una muestra de Y. Aunque no se sabe o no se conoce la distribución de Y, se podría tomar un número de muestras, calcular el promedio $\bar{Y}$ de esos números experimentales, y utilizar $\bar{Y}$ como un estimador de la media de Y (es decir, utilizar $\bar{Y}$ para aproximar E(Y)). Así, los valores observados son usados para hacer una declaración acerca de la variable aleatoria.

El procedimiento general para estimar una propiedad Θ de la distribución es el siguiente. Tomar un número de observaciones $(x_1, x_2, x_3, \ldots x_n)$ de la variable aleatoria X de interés. Formar una función $F(x_1, x_2, x_3, \ldots x_n)$ cuyo dominio es el conjunto de observaciones y cuyo rango es el conjunto de posibles valores de propiedad Θ. Tal función se llama estimador de Θ y es una variable aleatoria.

En este momento, la pregunta es: ¿Cómo elegir una función específica para usarla como estimador? Puesto que existen muchas posibles elecciones, algún criterio debe permitirse para guiar la elección de un buen estimador. El criterio más obvio

intuitivamente, es elegir una función estimador F (recuérdese que F es una función aleatoria) cuya propia distribución esta altamente concentrada alrededor del valor verdadero de Θ. Consideraremos tres características específicas de buenos estimadores, los cuales reflejan este requerimiento intuitivo.

Una variable aleatoria F es un estimador imparcial de Θ si $E[F] = \Theta$. Existen muchos estimadores sencillos imparciales para la media y la varianza de alguna variable aleatoria. El estimador

$$\bar{X}(x_1, x_2, x_3, \ldots x_n) = 1/n \sum_{i=1}^{n} x_i$$

es un estimador imparcial de la media E X. Esto sigue de

$$\begin{aligned} E[\bar{X}] &= E[1/n \sum_{i=1}^{n} x_i] \\ &= 1/n \sum_{i=1}^{n} E[x_i] \\ &= (1/n) n (E[X]) \\ &= E[X] \end{aligned}$$

Puesto que Var [X] está definida en términos de E [X], el estimador elegido para Var[X] dependerá de que si E[X] es conocido o no será estimado. Si E[X] es conocido, entonces el estimador

$$G(x_1, x_2, x_3, \ldots x_n) = 1/n \sum_{i=1}^{n} (x_i - E[X])^2$$

sugiere a sí mismo debido a la definición de Var X, y el decir que G es imparcial se establece fácilmente:

$$\begin{aligned} E[G] &= E[1/n \sum_{i=1}^{n} (x_i - E[X])^2] \\ &= 1/n \sum_{i=1}^{n} E[(x_i - E[X])^2] \\ &= 1/n \sum_{i=1}^{n} (E[x_i^2] - 2E[x_i]E[X] + (E[X])^2) \\ &= (1/n) n E[X^2] - n(E[X])^2 \\ &= Var[X] \end{aligned}$$

Si E[X] no es conocida, como es el caso usualmente, entonces el estimador

$$S^2(x_1, x_2, x_3, \ldots x_n) = 1/(n-1) \sum_{i=1}^{n} (x_i - \bar{X})^2$$

es un estimador imparcial de Var[X].

Un estimador consistente $G(x_1, x_2, x_3, \ldots x_n)$ de una propiedad es uno cuyo valor será arbitrariamente cerrado a Θ con probabilidad acercándose a 1 como $n \to \infty$. Más formalmente, $G(x_1, x_2, x3, \ldots x_n)$ es un estimador consistente para Θ si

$$\lim_{n \to \infty} P\{|G(X_1, X_2, X_3, \ldots Xn) - \Theta| < E\} = 1 \text{ para toda } E > 0$$

La característica de consistencia refleja la sensación intuitiva de que muestras más grandes deben ser más confiables. Esta sensación es matemáticamente justificada por uno de los más importantes resultados de la teoría de probabilidad.

TEOREMA (Ley debil de los grandes números): $X_1, X_2, X_3, \ldots Xn$ serán variables aleatorias independientes con (a) $E[Xi] = m$, y (b) $\text{var}[Xi] = q^2$ para $i = 1,2,3,\ldots,n$.

Si X está definida como

$$X(X_1, X_2, X_3, \ldots Xn) = 1/n \sum_{i=1}^{n} Xi$$

entonces para alguna $E>0$,

$$\lim_{n \to \infty} P\{|X-m| < E\} = 1$$

Una versión firme de este teorema permite que la condición de varianza finita termine. Una consecuencia inmediata de la ley de los Grandes Números es que X un estimador consistente de la media de alguna variable aleatoria.

Un estimador imparcial varianza-mínima es aquél cuya varianza es la más pequeña entre todos los estimadores imparciales, o quizá entre todos los estimadores imparciales de cierto tipo. La varianza puede ser muchas veces considerada como una media de la precisión de un estimador. El valor más pequeño de la varianza tiene mejores oportunidades de que el valor del estimador se aproxime a la propiedad que éste estima. Así, la varianza más pequeña es una característica deseable de un estimador.

ESTADISTICA DESCRIPTIVA.

Existen, clases muy importantes de algoritmos cuyo único propósito es la descripción de los datos actuales y la reducción de las grandes cantidades de datos. En estos algoritmos se encuentran la mayor parte de los procedimientos y métodos de Estadística.

La estadística descriptiva se emplea para analizar un gran número de datos por medio de algunos parámetros cuidadosamente seleccionados, dichos parámetros deben captar aspectos vitales de los datos. Las dos estadísticas más usadas para analizar un conjunto de datos son la media y la desviación estándar. La primera también se llama media aritmética y es un valor numérico en el cual los valores de experimento se acercan (Tendencia central), y la segunda es una medida de dispersión de los datos alrededor de la media que toman como punto de agrupación.

La media se define formalmente como

$$\mu = \left(\sum_{i=1}^{n} Xi\right)/n \qquad (1)$$

es decir, la sumatoria de todos los elementos de la muestra dividida entre el número de elementos; y la desviación estándar como

$$\sigma = \sum_{i=1}^{n} (Xi-\mu)^2/n \qquad (2)$$

donde las Xi son los datos y n es el número de datos.

La mediana **M** es el valor menor que la mitad de los puntos de datos y mayor que la otra mitad; es decir, es un valor medio basado en el orden de magnitud. También existe la moda que es el valor del elemento que aparece más frecuentemente.

Una forma de representar gráficamente el conjunto de datos junto con su estadística descriptiva es trazando un histograma, que se obtiene particionando el rango de los datos en subrangos de igual tamaño y trazando el número de partidas de datos en cada subrango. Si en el experimento se tiene una gran muestra, se puede normalizar el histograma dividiendo el número de puntos en cada subrango para un tamaño de muestra.

El cálculo de la estadística descriptiva juega un papel muy importante en el procesamiento de datos, y el desarrollo de algoritmos eficientes estables (insensitivos a errores de redondeo) para su cálculo es vital.

Si los datos son dados o representados todos a la vez como una **n-dupla**, las definiciones (1) y (2) se pueden ver como **algoritmos "naturales"** (note que éstos son ejecutados en forma secuencial).

Si fijamos C, para el análisis de costo a cero y dejamos **np** ser el número de sumas (restas) de datos y **nm** el número de multiplicaciones (divisiones), la complejidad del algoritmo es:

$$C(n) = w_1((n-1)Cp+Cm+(n+(n-1)+1)Cp+(n+1)Cm)+w_2nCs$$

$$= w_1((3n-1)Cp+(n+2)Cm)+w_2nCs$$

más la complejidad de la operación de la raíz cuadrada.

Por otra parte podemos reducir la complejidad de este algoritmo notando que (2) puede ser formulado como.

$$\sigma^2 = \left(\sum_{i=1}^{n}(Xi-\mu)^2\right)/(n-1)$$

$$= \left(\sum_{i=1}^{n} Xi^2 - 2\mu\sum_{i=1}^{n} Xi + \sum_{i=1}^{n}\mu^2\right)/(n-1) \qquad (3)$$

Usando la formulación (3) podemos escribir un algoritmo, que calcula σ y μ concurrentemente. Si se observa lo anterior el término que domina el límite es:

$$r \sum_{i=1}^{n} |Si|$$

que indica que el límite es proporcional a la suma de las sumas parciales.

En ambos algoritmos los términos en las sumas parciales de la desviasión estándar son positivos, pero en el segundo esos términos serán mucho más grandes. La razón es obvia, dado que en el algoritmo "natural" la media se substrae primero de los datos.

6.2.2 ALGORITMOS PARA PROBABILIDAD

En la actualidad existe un gran número de problemas importantes que envuelven algún grado de incertidumbre, donde las cantidades de mayor interés no son predecibles de antemano, pero exhiben una inherente aleatoriedad que se considera en algún modelo útil. A los modelos que pertenecen a esta especie se les denomina modelos de probabilidad.

Dentro del área del procesamiento de datos y en el área de juegos, los modelos de probabilidad tienen una amplia aplicación en la solución de diversos problemas. Numerosos problemas requieren una muestra imparcial de exactamente **M** elementos seleccionados de un conjunto de **N** elementos; por ejemplo, si se desea simular el turno de un jugador en una partida de poker o tomar una muestra limitada de un conjunto de registros.

La primera proposición algo ingenua, para la construcción de un procedimiento de muestreo es seleccionar cada registro con probabilidad **M/N**. Este método tiene la característica de que cada elemento se rechaza o acepta, éste paso debe realizarse a través de todo el archivo de elementos. Desafortunadamente, este método no garantiza que exactamente M elementos serán seleccionados.

Para ver esto, se definen las variables aleatorias para i=1,2,3,...n; como

$$Xi \begin{cases} 1 & \text{Si el elemento i es elegido.} \\ 0 & \text{Si el elemento i no es elegido.} \end{cases}$$

Donde la función de probabilidad (el nombre completo de esta función es el de: Función masa de probabilidad de una variable aleatoria) para cada X, es

$$P(Xi=0) = 1-M/N$$

$$P(Xi=1) = M/N$$

las Xi son variables aleatorias independientes distribuidas uniformemente. Cada una tiene media **M/N** y varianza **(M/N)(1-(M/N))**. La variable aleatoria

$$X = X_1 + X_2 + X_3 + \ldots + Xn$$

denota los números de elementos elegidos de alguna prueba.

Entonces,

$$E[X] = E[X_1] + \ldots + E[Xn] = N(M/N) = M$$

y

$$Var[X] = Var[X_1] + \ldots + Var[Xn]$$

$$= N(M/N)(1-M/N)$$

$$= M(1-M/N)$$

Asi el número promedio de elementos elegidos es **M**, y la gran diferencia o variación implica que el número escogido en cualquier prueba puede ser considerablemente mayor o menor que **M**.

La segunda proposición intuitiva para seleccionar **M** enteros de una muestra de **N** elementos es generar repetidamente números enteros aleatorios hasta que **M** diferentes números que pertenecen a la muestra hayan aparecido..

Este procedimiento puede ser planteado algoritmicamente como:

ALGORITMO TRYAGAIN.

Seleccionar una muestra imparcial de números **1<=M<=N** del conjunto de enteros 1,2,3,...,N. Los números seleccionados en un arreglo se almacenarán en el arreglo **SELECT**.

```
1)  INICIALIZA
    L := 0  } La variable l es usada para grabar el número de
              elementos seleccionados hasta ese momento. {
    Etiquetar todos los N enteros como "no seleccionados".
2)  Mientras L < M
    BEGIN
       Generar un número aleatorio uniformemente distribuido K
       Si K esta etiquetado como "no seleccionado"
       BEGIN
          L := L + 1
          SELECT(L) := K
          Etiquetar K como "seleccionado"
       END
    END
```

Aunque el algoritmo de **TRYAGAIN** es intuitivamente atrayente y simple, este puede ser muy lento e ineficiente, debido a que se puede generar una gran cantidad de números aleatorios antes de que ocurra el siguiente entero "no seleccionado". Esto se explica en el siguiente análisis:

Sea $p = (N-L)/N$ la probabilidad de generación de un entero "no seleccionado" 1,2,...,N; suponiendo que L enteros ya han sido seleccionados. Siendo $q_L = 1-p_L = L/N$ la probabilidad de generación de uno de los L enteros previamente seleccionados.

Si X es la variable aleatoria igual al número de enteros generados antes de que ocurra un entero "no seleccionado", y si se supone que n enteros ya han sido seleccionados, entonces

$$P(X_L = K) = q_L^{K-1} p_L$$

Así observamos que X_L tiene una distribución geométrica con

$$E(X_L) = \sum_{K=1}^{\infty} K q_L^{K-1} p_L = 1/p_L$$

Ahora, si T es una variable aleatoria, igual al número de enteros generados, si seleccionamos M de N enteros, entonces

$$E(T) = \sum_{L=0}^{M-1} E(X_L)$$

$$E(T) = 1 + N/(N-1) + N/(N-2) + N/(N-3) + \ldots + N/(N-(M-1))$$

$$= N \sum_{K=N-M+1}^{N} 1/K$$

$$= N(H_N - H_{N-M})$$

donde

$$H_N = \sum_{K=1}^{N} 1/K \text{ (es el número de n-ésimo armónico).}$$

Por otra parte, se desea un algoritmo que garantice escoger exactamente M elementos en toda la prueba, que cuente también con la característica de aceptación/rechazo como en el paso 1 del primer procedimiento descrito.

Suponiendo que tratamos de inspeccionar el elemento (K+1), y que q elementos ya han sido seleccionados, la pregunta es ¿Qué probabilidad será usada para aceptar a ese elemento? Observe que existen $\binom{K}{M-q}$ maneras de escoger M-q elementos de los restantes N-K. Cada manera se considera igualmente probable, y debe ocurrir una si estamos seleccionando eventualmente M elementos. Similarmente, si el elemento (K+1) es elegido, existen $\binom{K}{M-q-1}$ maneras de escoger M-q-1 elementos de los N-K-1 que restan. Las formas para elegir M de N elementos, tal que q son elegidos entre los K primeros son:

$$\frac{\binom{N-K-1}{M-q-1}}{\binom{N-K}{M-q}} = \frac{M-q}{N-K}$$

de esos eligiremos a el elemento (K+1).

ALGORITMO SS (SELECCION DE UNA MUESTRA IMPARCIAL).

Seleccionar una muestra imparcial de 1 <= M <= N elementos de un conjunto de N elementos.

```
1. k := 0; |indica que elemento será inspeccionado    |
   q := 0; |denota el número de elementos seleccionados
           actualmente |
2. MIENTRAS q<m
     a)|Generar Y   un número aleatorio entre 0 y 1 |
     b)| Prueba de aceptación o rechazo |
        SI (n-k)Y < (M-q) ENTONCES
           selecciona k+1;
           q := q + 1;
     c) k := k+1;
```

A primera vista, no es del todo claro que el algoritmo SS siempre selecciona exactamente M elementos y que la muestra es imparcial, esto es, la probabilidad de aceptación usada en el paso 3 da a cada elemento una oportunidad igual de adaptación.

Se presentan dos teoremas relacionados con este algoritmo.

TEOREMA 1. El algoritmo SS selecciona M elementos.

PRUEBA: Claramente se ve que no más de M elementos se elegen, puesto que el algoritmo termina tan pronto como el M-ésimo elemento se selecciona.

En el paso b, se observa que si q elementos han sido seleccionados y que solamente los M-q elementos restantes serán inspeccionados, entonces todos los elementos restantes serán elegidos. Así, al menos M elementos son seleccionados.

TEOREMA 2. El algoritmo SS selecciona cada elemento de una manera imparcial.

Considere el elemento (k-1)-ésimo. En cualquier prueba la probabilidad de que aceptemos este elemento es (M-q)/(n-k), donde q es el número de elementos ya seleccionados. El valor de q es una variable aleatoria que depende de las selecciones hechas durante las primeras k inspecciones.

Dado que, para alguna k,M y n, el valor promedio de (M-q)/(n-k) es exactamente M/N, el algoritmo SS es imparcial.

Otro método para seleccionar M enteros de N es el que a continuación se presenta:

ALGORITMO SELECT

Para seleccionar una muestra imparcial de 1 <= M <= N enteros del conjunto de enteros 1, 2, ..., N. Los enteros seleccionados serán almacenados en un arreglo **SELECT**.

```
1. PARA  i:=1  HASTA n HACER
     ITEM(i) := i;
2. ultimo := n;
   l       := 1;
3. MIENTRAS  l <= M
      a) Generar k { un número aleatorio entre 1 y último};
      b) SELECT(l) := ITEM(k);
         l         := l + 1;
      c) ITEM(k) := ITEM (ultimo);
         ultimo := ultimo - 1;
```

Es fácil probar que el algoritmo **SELECT** es imparcial. Si **pi(k)** denota la probabilidad de que el entero **i-ésimo** es seleccionado en la iteración **k-ésima** del algoritmo **SELECT**. La probabilidad pi de seleccionar el entero i es:

$$pi = \sum_{k=1}^{M} pi(k)$$

donde

$$pi(k) = ((N-1)/N)((N-2)/(N-1))((N-3)/(N-2))\ldots((N-(K-1))/(N-(K-2)))(1/(N-(K-1))) = 1/N$$

Así,

$$pi = \sum_{k=1}^{M} 1/N = M/N$$

Note que solamente un número fijo de M enteros aleatorios se genera a fin de seleccionar M enteros distintos obtenidos de N usando el algoritmo **SELECT**. De hecho, la cantidad de trabajo requerido es **O(NMM)**, esto es, el algoritmo **SELECT** ejecutará:

```
M+2     declaraciones de asignación del paso 1 y 2.
4M      declaraciones de asignación del paso 3.
M       generaciones de números aleatorios en el paso 3-b.
```

en comparación con el algoritmo **SS** el cual ejecuta:

```
2       declaraciones de asignación en el paso 1.
N       generaciones de números aleatorios en el paso 2-a.
N       pruebas en el paso 2-b.
3M      declaraciones de asignación en el paso 3-b.
n       incrementos en el paso 3-c.
```

Note también que el algoritmo **SELECT** produce una permutación de los enteros 1,2,3,4,...N cuando M = N, de tal modo que aumenta su total utilidad.

NUMEROS ALEATORIOS.

Los Generadores de Números Aleatorios son [illegible] modelos probabilísticos muy importante dentro del Area de Juegos, por otro lado, para entender esto es necesario saber lo siguiente:

Las variables aleatorias discretas, son variables aleatorias cuyos rangos son finitos o conjuntos contables infinitos. Por lo tanto, las variables aleatorias cuyos rangos son infinitos no contables son llamadas variables aleatorias continuas. Aunque muchas aplicaciones elementales en la Ciencia de la Computación de Probabilidad, y Estadística usan solamente variables aleatorias discretas, existen funciones de variables aleatorias continuas muy importantes: **La función de densidad y la función** continua

Una variable aleatoria X es continua si existe una función no negativa f(x), definida en la línea real entera, teniendo la propiedad de que si S es algún conjunto de números reales, entonces:

$$P(X \in S) = \int_S f(x)dx$$

La función f(x) es llamada **La función densidad de X.**

En particular si S es la línea entera real, entonces, después X puede tomar algún valor,

$$1 = P(X \in (-\infty,\infty)) = \int_{-\infty}^{\infty} f(x)dx.$$

También, si S es algún intervalo [a, b] es la línea real, entonces

$$P(a<=x<=b) = \int_a^b f(x)dx$$

La Contraparte de una variable aleatoria discreta con una distribución probablemente igual es la Variable Aleatoria Uniforme. Si X es una variable aleatoria, es probable que tome algún valor en el rango [0, 1], y su función densidad debe ser

$$f(x) = \begin{cases} 1, \text{ si } 0 <= X <= 1 \\ 0, \text{ de otra manera.} \end{cases}$$

Entonces, por ejemplo

$$P(1/4 <= X <= 3/4) = \int_{1/4}^{3/4} 1dx = 3/4 - 1/4 = 1/2$$

y

$$P(X = 1/2) = \int_{1/2}^{1/2} 1dx = 1/2 - 1/2 = 0$$

En donde, el último resultado es esencialmente importante.

La probabilidad de que una variable aleatoria continua sea igual a algún valor particular de X en su rango continuo es siempre 0, no f(x). Sin embargo, no podemos asignar probabilidades finitas a una infinidad continua de valores, ya que la "suma" de todas estas probabilidades debería ser infinita (no la unidad). Es necesario pensar cuidadosamente acerca de esto. La distribución uniforme en el intervalo (0, 1) ó (0, 1); ¿es importante? Si, ya que ésta es esencialmente la distribución simulada por los generadores de números aleatorios en computadora. Esto es, un número aleatorio generado por una computadora es básicamente una muestra de una variable aleatoria uniforme.

El valor esperado y la varianza de una variable aleatoria continua es similar al de una variable aleatoria discreta, con la excepción de que las sumatorias son reemplazadas por integrales. Así,

$$E[X] = \int_{-\infty}^{\infty} x f(x)\,dx$$

$$VAR[X] = \int_{-\infty}^{\infty} (x - E[X])^2 f(x)\,dx.$$

Para una variable aleatoria uniforme U en [0, 1], tenemos

$$E[U] = \int_{-\infty}^{\infty} x f(x)\,dx = \int_0^1 x\,dx = x^2/2 \Big|_0^1 = 1/2$$

$$VAR[U] = \int_{-\infty}^{\infty} (x - E[X])^2 f(x)\,dx = \int_0^1 (x - 1/2)^2\,dx = 1/12$$

Otra distribución continua de gran importancia es la Distribución Normal. La función densidad para una variable aleatoria X con una distribución normal con parámetros μ y σ^2 está dada por

$$f(x) = 1/\sqrt{2\pi\sigma^2}\; e^{[-(x-\mu)^2]/2\sigma^2} \quad \text{con} \quad -\infty < X < \infty.$$

Esta es la famosa curva "acampanada de la Teoría de la Probabilidad".

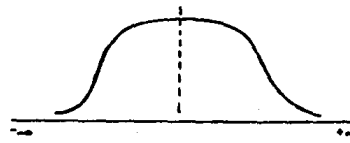


FIG 6.52

La curva es simétrica con respecto al valor , la cresta pronunciada para valores pequeños de , y más extendida para valores grandes de . Una distribución normal con parámetros μ y σ^2 es denotada por $N(\mu, \sigma^2)$. La distribución N(0, 1) es llamada Normal Estándar.

Un número sorprendentemente grande de variables aleatorias tienen distribucion Normal y cerca de la Normal. Además, muchas de estas variables aleatorias tienden a exhibir la propiedad acumulativa, o aditiva, que son efectos de otras variables aleatorias. Teóricamente, este sigue a un número de importantes teoremas, colectivamente conocidos como Teoremas de Límite Central.

Extrictamente hablando, estos teoremas afirman que una variable aleatoria o mejor dicho que la suma de muchas variables aleatorias tienden a tener una distribución normal.

Ahora los estimadores $\bar{X}$ y S^2 pueden ser presentados como el estimador imparcial y el estimador consistente respectivamente de varianza mínima para μ y σ^2 para alguna variable aleatoria distribuida normalmente. En casos donde una variable aleatoria está pensada para reflejar los efectos aditivos de otras variables aleatorias, una distribución normal con $\bar{X}$ y S^2 (basado en observaciones estadísticas) puede ser un buen estimador para la distribución entera no conocida de la variable aleatoria.

Por consiguiente, los generadores de números aleatorios en una computadora tratan de funcionar como generadores de valores muestrales de una variable aleatoria que es uniforme en el intervalo de 0 a 1. Por lo tanto, otras distribuciones de variables aleatorias pueden ser fácilmente simuladas usando esta distribución básica. Así un algoritmo de generación de números aleatorios intenta elegir un número entre 0 y 1, tal que la probabilidad de que el número esté situado en algún subintervalo particular del intervalo unidad es igual a la longitud del subintervalo particular.

Uno de los medios mecánicos obvios para realizar esto, es usar una rueda perfectamente balanceada con circunferencia igual a la unidad. La rueda hila bajo una fina aguja, y la lectura bajo la aguja se considera el siguiente número generado aleatoriamente. Esto da pie a una pregunta ¿Por qué no construir tal dispositivo para el **HARDWARE** de la computadora? la razón por la cual la rueda de sorteo no es usada es que los números aleatorios pueden ser generados más rápidamente, en forma exacta y barata y casi tan confiable, usando ciertos algoritmos simples.

Actualmente estos algoritmos no generan números aleatorios con la misma incertibumbre como la rueda girando. De hecho, ellos usan operaciones aritméticas simples para producir determinísticamente una secuencia de números que solamente parecen aleatorios, ya que pueden ser predecidos antes de ser generados. Sin embargo las secuencias de números aleatorios generados por estos algoritmos satisfacen pruebas estadísticas sofisticadas dando a estos la licencia de razonables (Ver Capitulo II).

ALGUNAS APLICACIONES.

Como ya se había observado este subcapítulo está enfocado a temas pertenecientes a la Teoría de Probabilidad y Estadística, por consiguiente se describirán a continuación y en forma breve, algunas de las aplicaciones enfocadas al diseño y análisis de algoritmos de estas áreas:

1. Análisis del funcionamiento promedio de un algoritmo.

Supóngase que se tiene un algoritmo óptimo (algoritmo S) para tareas de carácter sencillo dentro de la computación. Sabemos que los factores importantes que determinan el tiempo de ejecución del algoritmo son el número **N** (número de tareas) y los tipos de tareas que esperan ser realizadas. Si todas las tareas son de un tipo simple, el algoritmo S puede construir un proyecto en un tiempo **O(N)**. Sin embargo, si todas las tareas son de un tipo complicado, entonces el algoritmo S requiere un tiempo digamos $O(N^4)$.

Si por ejemplo, existen 12 diferentes tipos de tareas entre estos dos extremos, y existe también un número grande de tareas que contienen una mezcla de varios tipos. Surge la siguiente pregunta ¿Qué tan bueno es el algoritmo? a fin de responder esta pregunta, se necesita saber qué conjunto de tareas amenazan, para que se decida qué tan bien funciona el algoritmo en el caso promedio. Todo esto implica la necesidad de alguna definición precisa de **"promedio o común"**. Quizá el peor caso $O(N^4)$ es inaceptablemente lento para el sistema operativo; por ejemplo, el sistema completo podría encontrarse haciendo nada pero, esperando a que la subrutina $O(N^4)$ se presente con un proyecto. Si el comportamiento del peor caso no ocurre frecuentemente y el comportamiento promedio de algoritmo es digamos **O(N)**, puede ser posible usar el algoritmo S en el sistema operativo. De otra manera, si el comportamiento promedio es $O(N^4)$, se podría obligar a usar un rápido y codicioso algoritmo heurístico, que no puede garantizar proyectos óptimos.

Este ejemplo, es representativo de un número de importantes preguntas, que aparecen en el análisis de algoritmos.

2. Simulación.

Uno de los usos más importantes de una computadora es proveer un vehículo para la conducción de experimentos controlados en complicados sistemas reales de costo relativamente bajo y sobre intervalos de tiempo breves. Esto es hecho usando un algoritmo que modela y simula el sistema real.

Esta es una técnica de diseño y análisis de último recurso ya que los modelos de simulación son generalmente usados para problemas que están más allá de nuestro poder analítico, porque combinan un gran tamaño e insertidumbre. Además, virtualmente todos los modelos serios de simulación son probabilísticos.

3. Estadística Computacional.

La computadora es a menudo usada para procesar y analizar la información estadística. Los algoritmos deben ser diseñados y analizados para ejecutar cálculos estadísticos exacta y eficientemente.

4. Algoritmos para la toma de desiciones.

La computadora se está haciendo cada vez mas popular en el mundo de los negocios y en el gobierno como una ayuda en la toma de decisiones. Básicamente todos los problemas en esta área de aplicaciones envuelven la toma de decisiones bajo insertidumbre.

Uno de los más prometedores métodos cuantitativos para tratar con tales problemas es conocido como la Teoría de decisión estadística Bayesiana. Desafortunadamente, esta teoría esta en necesidad grave de un buen algoritmo computacional, de hecho su utilidad actualmente esta restringida sólo para problemas relativamente simples o sencillos.

5. Generación de números aleatorios.

Muchas de las aplicaciones precedentes de probabilidad y estadística requieren de un modo de introducir incertibumbre en los datos. Si es posible, se quiere hacer esto por medio de algún mecanismo interno en la computadora misma, evitando así consumir tiempo y actividad externa. El diseño y análisis de buenos algoritmos para la generación de números aleatorios es un problema significante.

CAPITULO VII

EFICIENCIA DE LOS ALGORITMOS.

Aún cuando lo más importante es que los programas sean correctos, en la práctica no queremos algún algoritmo, deseamos buenos algoritmos, pero ¿Qué entendemos por bueno? ¿Cómo podemos comparar dos algoritmos que realizan la misma tarea? El análisis de algoritmos es un área importante de la Teória de Computación que se encarga de hacer tal comparación, definiendo para ello un conjunto de medidas objetivas que pueden aplicarse a un algoritmo.

Es importante determinar cuál de dos algoritmos requiere menos trabajo para realizar una tarea particular; una primera solución es sencillamente codificar los algoritmos y luego comparar sus tiempos de ejecución. El que tenga el mejor tiempo será evidentemente el mejor algoritmo ¿o no lo es? Sólo se puede decir con ello que un programa es más eficiente que otro, pero esto es en una computadora en particular. Por supuesto, podemos probar los algoritmos sobre todas las computadoras posibles, pero lo que se desea es una medida más general.

Una segunda posibilidad es contar el número de instrucciones o sentencias ejecutadas, sin embargo, esta medida varía con el lenguaje de programación empleado. Para estandarizar de alguna forma esta medida, podríamos contar el número de pasos de un bucle crítico del algoritmo. Si cada iteración conlleva una cantidad constante de trabajo, esta medida nos dará un criterio con sentido de eficiencia.

La eficiencia es simplemente una cuestión de practicabilidad, sin embargo, es uno de los aspectos más importantes, ya que se refiere a la velocidad de ejecución del programa, a su uso de los recursos del sistema o a ambas cosas. Los recursos del sistema incluyen la Memoria RAM, el espacio en disco, el papel de impresora y básicamente cualquier cosa que se pueda emplear y se pueda distribuir.

Hay que tener presente que la eficiencia de un programa, es algo subjetivo que depende de cada situación.

Otra consideración en lo que concierne a la eficiencia es que generalmente al optimizar un aspecto del programa se degrada el otro. Por ejemplo, el hacer que un programa se ejecute más rápido, puede implicar el uso de mayor código, o el hacer un uso más eficiente de un disco compactando los datos, puede implicar el disminuir la velocidad de acceso a este dispositivo.

Los diseñadores y planeadores de sistemas para maximizar su propia eficiencia, deben conocer un buen número de técnicas de análisis y diseño de algoritmos, y reconocer las ventajas y desventajas de cada una de ellas, de tal manera, que cuando se presente la necesidad de su empleo, puedan suministrar aquella que sea la más apropiada para una situación en particular.

7.1 COMPLEJIDAD DE UN ALGORITMO.

Tan pronto como se tiene un algoritmo es necesario realizar un análisis de éste, con la intención de mejorarlo si es posible, o para determinar si es el óptimo entre los disponibles para resolver un problema dado.

Algunos de los criterios empleados para analizar un algoritmo son:

1) FUNCIONAMIENTO CORRECTO.

El saber si un algoritmo es correcto o no, es el primer tipo de análisis que podemos realizar; antes de determinar cuando un algoritmo es correcto, se debe tener claro qué significa esa palabra; se dice que un algoritmo cumple ésta condición si cuando se da una entrada válida, esta se procesa en una cantidad finita de tiempo y produce una respuesta. En este proceso existen dos aspectos importantes:

a) Método de solución del problema.

Establecer que el método y/o las fórmulas empleadas sean correctos requiere una secuencia grande de lemas y teoremas acerca de los elementos con los que trabaja el algoritmo (por ejemplo: gráficas, permutaciones, matrices).

b) Secuencia de instrucciones para obtener una salida.

Si un algoritmo es pequeño e informal, generalmente se emplean significados de autoconvencimiento (las partes del programa hacen lo que nosotros deseamos que hagan). En este caso basta con checar algunos detalles cuidadosamente y probar el algoritmo con pequeños ejemplos, teniendo presente que nada de esto prueba que el algoritmo es correcto, en forma válida. Estas técnicas informales pueden ser suficientes para determinados procesos, no así para programas grandes y complejos.

El probar que un programa grande sea correcto puede facilitarse si éste se divide en segmentos, y se demuestra que cada segmento realiza su trabajo en forma correcta.

Una de las técnicas mas empleadas para pruebas rigurosas de funcionamiento correcto, es la inducción matemática, que se emplea para demostrar que los ciclos en un algoritmo realizan lo que intentan hacer; para cada ciclo se establecen condiciones y relaciones que se cree, son satisfechas por las variables y estructuras de datos empleadas, entonces se verifica que esas condiciones son válidas en el número de pasos a través del ciclo. Los detalles de la prueba requieren seguir con cuidado las instrucciones en el algoritmo.

En general, el saber si un algoritmo es correcto o no, es dado por el propio algoritmo y se lleva a cabo verificando si la lógica, la implementación y las técnicas matemáticas que emplea son las adecuadas.

2) SIMPLICIDAD.

Otro tipo de análisis es la simplicidad del algoritmo. Sin embargo, éste no es mejor camino ya que puede implicar mucho tiempo y espacio computable, pero es una característica deseable.

3) CANTIDAD DE TRABAJO EMPLEADO.

La cantidad de tiempo usado por la computadora no se encuentra en función del trabajo empleado, éste es la eficiencia del método usado por un algoritmo independiente, no sólo de la computadora, el lenguaje de programación o el programador; sino también de muchos detalles de implementación, o de la contabilidad de operaciones tales como incrementos de los índices, calculo de los arreglos indexados o colocación de los contadores en las estructuras de datos.

La cantidad de trabajo no puede ser descrita por un sólo número, porque el número de operaciones básicas, no es el mismo para todas las entradas. Es importante realizar la observación que aún si nosotros consideramos sólo entradas de un tamaño, el número de operaciones desarrolladas por un algoritmo puede depender de una entrada en particular.

Dada la implementación del algoritmo en un lenguaje, es posible identificar todos los operandos definidos como variables o constantes; similarmente, es también posible identificar todos los operadores definidos como símbolos o combinaciones de símbolos que afecta el valor u orden de una operación.

Una vez identificados los operandos y operadores se puede definir un número contable de elementos que se presentan en una versión de un algoritmo.

Las propiedades de cada expresión de un algoritmo que es capaz de ser contabilizado o medido incluye lo siguiente:

n_1 : número de operadores distintos que aparecen en la implementación.

n_2 : número de operandos diferentes que aparecen en la implementación.
N_1 : uso total de los operadores que aparecen en la implementación.
N_2 : uso total de los operandos que aparecen en la implementación.
f_{1j} : número de ocurrencia del j-ésimo operador.
f_{2j} : número de ocurrencia del j-ésimo operando.

Por lo tanto, el vocabulario n queda definido como $n = n_1 + n_2$ y la longitud de la implantación $N = N_1 + N_2$.

Donde

$$N_1 = \sum_{j=1}^{n_1} f_{1j}$$

$$N_2 = \sum_{j=1}^{n_2} f_{2j} \quad \longrightarrow \quad N = \sum_{i=1}^{i} \sum_{j=1}^{n} f_{ij}$$

4) **MEMORIA USADA.**

Un programa, requiere espacio de memoria para las instrucciones, las constantes usadas y los datos de entrada; también usa espacio para manipular los datos que pueden ser representados en distintas formas, algunas de ellas requieren más espacio que otras, por lo que su elección y manipulación adecuada puede marcar la diferencia entre algoritmos que realizan la misma función.

Estos son sólo algunos de los criterios que en un momento dado son válidos para analizar un algoritmo; pero el criterio primordial y de mayor importancia es el del análisis del tiempo y espacio requerido por un programa conocido con el nombre de **COMPLEJIDAD COMPUTACIONAL**.

7.1.1 Definición del término COMPLEJIDAD.

N. Klaus Wirth en el libro **ALGORITHMS + DATA STRUCTURES = PROGRAMS**, define la complejidad o costo de un algoritmo como una función **F(n)**, que puede indicar el tiempo o la cantidad de memoria requerida al ejecutarse el algoritmo correspondiente a un problema de tamaño n; en otras palabras es un término empleado para estimar el poder computacional requerido para resolver un problema. Los axiomas empleados para calcularla se basan ya sea en el número de instrucciones ejecutadas (operaciones aritméticas o lógicas), o en la cantidad de bits empleados en el almacenamiento del programa; por ejemplo,

- El número de comparaciones en un algoritmo de ordenamiento,
- El número de multiplicaciones y sumas requeridas para evaluar un polinomio

De ahí que la obtención de la complejidad implique el conocer los límites extremos de las entradas para un algoritmo; si para un tamaño dado se considera a la complejidad máxima sobre todas las entradas de ese tamaño, se denomina **Complejidad del peor caso**, pero si se toma a la complejidad promedio, entonces es llamada **Complejidad esperada**.

La complejidad esperada de un algoritmo es más difícil de calcular, ya que se debe realizar una suposición probabilística respecto a la distribución de las entradas.

7.1.2 Eficiencia del tiempo de ejecución de los algoritmos.

Una forma de calcular la eficiencia del tiempo de ejecución de un algoritmo es programarlo y medir el tiempo que tarda esa versión en particular, en una computadora específica para un conjunto seleccionado de entradas. Aunque este enfoque es popular y útil, presenta algunos problemas ya que los tiempos de ejecución no sólo dependen del algoritmo base, sino también del conjunto de instrucciones de la computadora, de la calidad del compilador y la destreza del programador.

Para superar estos inconvenientes, los expertos en computación han adoptado la complejidad de tiempo asintótica como una medida fundamental del rendimiento de un algoritmo, así el término eficiencia se referirá a esta medida, y en especial, a la complejidad del tiempo en el peor caso.

La eficiencia o complejidad en el peor caso de un algoritmo es **O(f(n))** (f(n) para abreviar) que es la función de n que da el máximo, en todas las entradas de longitud n del número de pasos dados por el algoritmo en esas entradas. En otras palabras, existe alguna constante c tal que para una n suficientemente grande, **cf(n)** es una cota superior del número de pasos dados por el algoritmo con cualquier entrada de longitud n.

Si se afirma que la eficiencia de un algoritmo dado es f(n), existe la implicación de que la eficiencia también es **OMEGA(f(n))** de tal forma que f(n) es la función de crecimiento más lenta de n que acota el tiempo de ejecución en el peor caso por arriba. Sin embargo, este último requisito no es parte de la definición de **O(f(n))**, y algunas veces no es posible asegurar que se tenga la cota de crecimiento superior más lenta.

La definición dada de eficiencia ignora factores constantes en el tiempo de ejecución y existen varias razones para ello:

1) Dado que la mayoría de los algoritmos están escritos en lenguajes de alto nivel, se deben describir en función de pasos que emplean una cantidad constante de tiempo cuando se traducen al lenguaje de máquina de cualquier computadora, sin embargo, el tiempo exacto que requiere un paso depende no sólo del paso mismo, sino del proceso de traducción y del conjunto de instrucciones de la

máquina. Así intentar tener más precisión al decir que el tiempo de ejecución de un algoritmo es del orden f(n) podría significar inducir al usuario en detalles de máquinas específicas y sólo sería aplicable a esas máquinas.

2) Hay que tener presente que más que los factores constantes, la complejidad asintótica determina para qué tamaño de entradas puede usarse el algoritmo para producir soluciones en una computadora.

Sin embargo, es conveniente desviarse de esta noción de eficiencia en el peor caso, cuando se conoce la distribución esperada de las entradas de un algoritmo. En esta situación, el análisis del caso promedio puede ser más significativo que el análisis del peor caso, por ejemplo, cuando todas las permutaciones de un conjunto de datos a ordenar tienen la misma probabilidad de presentarse como entradas, es conveniente analizar la ejecución del algoritmo en el caso promedio

7.1.3 Notación asintótica (O Grande y OMEGA Grande).

Para hacer referencia a la velocidad de crecimiento de los valores de una función se usará la notación conocida como notación asintótica **(O Grande)**. Por ejemplo, decir que el tiempo de ejecución **T(n)** de un programa es $O(n^2)$ (O Grande de n al **cuadrado o sólo O de n cuadrada)**, significa que existen constantes enteras positivas **C** y n_0 tales que para **n** mayor o igual a n_0, se tiene que **T(n) <= Cn²**.

Dado que todas las funciones del tiempo de ejecución están definidas en los enteros no negativos, sus valores son siempre no negativos, pero no necesariamente enteros, entonces se dice que T(n) es O(f(n)) si existen constantes positivas C y n_0 tales que T(n) <= Cf(n) cuando n >= n_0. Si el tiempo de ejecución de un programa es **O(f(n))**, se dice que dicho programa tiene una velocidad de crecimiento de **f(n)**.

Cuando se dice que **T(n)** es **O(f(n))**, **f(n)** es una cota superior para la velocidad de crecimiento de **T(n)**. Para especificar su cota superior se emplea la notación **OMEGA(g(n))** (**OMEGA Grande de g(n)** u **OMEGA de g(n)**), lo cual significa que existe una constante **C** tal que **T(n) >= Cg(n)** para un número infinito de valores de **n**.

En estas dos notaciones empleadas existe una asimetría que es útil, ya que muchas veces un algoritmo es rápido para muchas entradas pero no con todas. Por ejemplo, existen algoritmos rápidos para entradas de longitud prima, no así para entradas de longitud par, de modo que no es posible obtener una buena cota inferior que sea válida para toda **n >= 0**.

7.1.4 ¿Cómo medir el tiempo de ejecución de un programa?

Desafortunadamente no es fácil medir el tiempo de ejecución de un programa ya que éste depende de diferentes factores:

1) Datos de entrada al programa.

2) Calidad del código generado por el compilador empleado para crear el programa objeto.

3) Naturaleza y rapidez de las instrucciones de máquina empleadas en la ejecución del programa.

4) Complejidad de tiempo del algoritmo base del programa.

El hecho de que el tiempo de ejecución dependa de los datos de entrada, indica que debe definirse como una función de la entrada. Pero con frecuencia el tiempo de ejecución no depende de la entrada exacta, sino sólo de su tamaño por lo que se acostumbra denominar **T(n)** al tiempo de ejecución de un programa con una entrada de tamaño **n**.

Para muchos programas, el tiempo de ejecución, es en realidad una función de la entrada específica, y no sólo del tamaño de ella; en este caso **T(n)** se define como el tiempo de ejecución del peor caso, es decir, el máximo valor del tiempo de ejecución para entradas de tamaño **n**. También suele considerarse **Tprom(n)**, que es el valor medio del tiempo de ejecución de todas las entradas de tamaño **n**, aún cuando esta medida parece más razonable, es engañoso suponer que todas las entradas son igualmente probables. En la práctica, casi siempre es más difícil determinar el tiempo de ejecución promedio que el del peor caso, ya que la noción de entrada "**promedio**" puede carecer de un significado claro.

Si se mide el tiempo en función a su implementación y se especifica el lenguaje y la máquina a emplear, es necesario conocer las instrucciones exactas ejecutadas por el hardware y el tiempo requerido para cada instrucción, además, no es posible expresar **T(n)** en unidades estándar de tiempo y las observaciones realizadas se limitarán a expresiones como "**el tiempo de ejecución de tal algoritmo es y**", sin especificar la constante de proporcionalidad.

7.1.5 Reglas generales de análisis en relación al tiempo de ejecución.

El tiempo de ejecución de una proposición o un grupo de ellas puede tener como parámetros el tamaño de la entrada, una o más variables, o ambas cosas, pero el único parámetro permisible para calcular el tiempo de ejecución del programa completo es el tamaño de la entrada **n**. En base a esto los tiempos de ejecución de una proposición son:

- PROPOSICION DE ASIGNACION, DE LECTURA O ESCRITURA:

Por lo común el tiempo de ejecución empleado por estas proposiciones puede tomarse como **O(1)**, pero hay que tener presente que existen lenguajes como **PL/I**, donde una asignación puede implicar matrices arbitrariamente grandes o lenguajes como **PASCAL** que permiten llamadas a funciones en las proposiciones de asignación.

- SECUENCIA DE PROPOSICIONES:

Se determina por la regla de la suma. El tiempo de ejecución de una secuencia de proposiciones, dentro de un factor constante, es el máximo tiempo de la ejecución de una proposición de la secuencia.

- PROPOSICION CONDICIONAL:

El tiempo de ejecución de una proposición condicional (IF) es el costo de las proposiciones que se ejecutan condicionalmente, más el tiempo para evaluar la condición que por lo general es **O(1)**.

El tiempo para una construcción (IF-THEN-ELSE) es la suma del tiempo requerido para evaluar la condición más el mayor entre los tiempos necesarios para ejecutar las proposiciones cuando la condición es verdadera y el tiempo de ejecución de las proposiciones cuando la ejecución es falsa.

- CICLO:

El tiempo para ejecutar un ciclo es la suma, sobre todas las iteraciones del ciclo, del tiempo de ejecución del cuerpo y del tiempo empleado para evaluar la condición de terminación (este último suele ser **O(1)**). A menudo este tiempo (despreciando factores constantes) es el producto del número de iteraciones del ciclo y el mayor tiempo posible para una ejecución del cuerpo, pero por seguridad, debe considerarse cada iteración por separado. Por lo común se conoce con certeza el número de iteraciones, pero en ocasiones no es posible determinarlo con precisión. Incluso puede ocurrir que un programa no sea un algoritmo y no exista un límite al número de iteraciones de ciertos ciclos.

- PROCEDIMIENTOS:

Se evalúan como un programa independiente, si realiza llamadas, primero debé evaluarse el tiempo de ejecución de los procedimientos que llama.

Si hay procedimientos recursivos, no es posible ordenar evaluaciones de modo que cada uno utilice sólo estimaciones ya

realizadas. Lo que se hace es asociar una función de tiempo desconocido T(n), donde n mide el tamaño de los argumentos del procedimiento. Luego se puede obtener una recurrencia para T(n), es decir, una ecuación para T(n) en función de T(k) para varios valores de k.

Ejemplo:

Sea el programa recursivo para calcular n!

```
        FUNCTION FACTORIAL (n: INTEGER): INTEGER;
        BEGIN
1)          IF n <= 1 THEN
2)              FACT := 1
            ELSE
3)              FACT := n * FACTORIAL(n-1)
        END;
```

Una medida de tamaño apropiado para esta función es el valor de n. Sea T(n) el tiempo de ejecución para FACTORIAL(n). El tiempo de ejecución para las proposiciones 1 y 2 es O(1) y para la 3 es O(1) + T(n-1); por tanto para ciertas constantes c y d:

$$T(n) = \begin{cases} c + T(n-1) & \text{si } n > 1 \\ d & \text{si } n <= 1 \end{cases}$$

Suponiendo que n > 2, se puede desarrollar T(n-1), obteniendo:

$$T(n) = c + T(n-1)$$

$$T(n-1) = c + T(n-2)$$

por lo tanto:

$$T(n) = 2c + T(n-2) \text{ si } n > 2.$$

Así pues, es posible reemplazar T(n-1) con c + T(n-2) en la ecuación T(n) = c + T(n-1). Después se puede desarrollar T(n-2) con lo que se obtiene T(n) = 3c + T(n-3) si n > 3 y así sucesivamente.

En general:

$$T(n) = ic + T(n-i) \text{ si } n > i$$

por último cuando i = n-1 se obtiene

$$T(n) = c(n-1) + T(1) = c(n-1) + d.$$

Por lo que se concluye que T(n) es O(n).

Es importante observar que se ha supuesto que la multiplicación de dos enteros es una operación de tiempo O(1).

- PROPOSICION DE TRANSFERENCIA INCONDICIONAL DE CONTROL (GO TO):

Las proposiciones **GO TO** hacen más complejo el agrupamiento lógico de las sentecias de un programa; por lo que se recomienda el no emplearlas; sin embargo, existen lenguajes de programación (PASCAL) que carecen de proposiciones para salir de un ciclo o terminarlo en forma anormal (**BREAK** o **CONTINUE**), por lo que con frecuencia se utiliza **GO TO** para estos fines.

Si el GO TO transfiere el control a una proposición que se ejecutará después de terminado el ciclo podemos suponer que nunca se efectuará y presuponer que el ciclo se ejecutará por completo; pero si se presentará un **GO TO** que transfiera el control a código ejecutado con anterioridad, no sería posible ignorarlo, ya que ese **GO TO** podría crear un ciclo que consumiera la mayor parte del tiempo de ejecución.

- SEUDOPROGRAMA:

Si se conoce la velocidad de crecimiento del tiempo necesario para ejecutar proposiciones informales en español, es posible analizar seudoprogramas, como si fueran programas reales. Sin embargo, muchas veces no se conoce el tiempo que demorarán las partes no completamente terminadas de un seudoprograma. Por lo que para poder analizar seudoprogramas que contienen proposiciones en algún lenguaje de programación y llamadas a procedimientos aún no aplicados, se calcula el tiempo de ejecución como una función de los tiempos de ejecución no especificados de esos procedimientos. Hay que tener presente que, el tiempo de ejecución de un procedimiento obtiene sus parámetros mediante el tamaño de su argumento (o argumentos), y así como sucede con el tamaño de la entrada, la medida apropiada para el tamaño de un argumento es decisión de quien hace el análisis del problema.

7.1.6 Cálculo del tiempo de ejecución de un programa.

El calcular el tiempo de ejecución de un programa arbitrario (o una aproximación a un factor constante) puede ser un problema matemático complejo, pero en la práctica suele ser sencillo si se aplican algunos principios básicos, para lo cual es necesario saber sumar y multiplicar en notación asintótica.

REGLA DE LA SUMA.

Supóngase que $T_1(n)$ y $T_2(n)$ son los tiempos de ejecución de dos fragmentos de los programas P_1 y P_2, y que $T_1(n)$ es $O(f(n))$ y $T_2(n)$ es $O(g(n))$, entonces $T_1(n) + T_2(n)$, es el tiempo de ejecución de P_1 seguido de P_2, o sea $O(\max(f(n),g(n)))$. Para saber porqué, obsérvese que para algunas constantes C_1, C_2, n_1 y n_2, si $n >= n_1$, entonces $T_1(n) <= C_1 f(n)$, y si $n >= n_2$ entonces

$T_2(n) <= C_2 g(n)$. Sea $n_0 = max(n_1, n_2)$. Si $n >= n_0$ entonces $T_1(n)+T_2(n) <= C_1 f(n)+C_2 g(n)$. De esto se concluye que si $n >= n_0$, entonces $T_1(n) + T_2(n) <= (C_1+C_2)$ max $(f(n), g(n))$, por lo tanto $T_1(n) + T_2(n)$ es $O(max(f(n), g(n)))$.

En general, el tiempo de ejecución de una secuencia fija de pasos, dentro de un factor constante, es igual al tiempo de ejecución del paso con mayor tiempo de ejecución. En raras ocasiones dos pasos pueden tener tiempos de ejecución incommensurables (ninguno es mayor que el otro, ni son iguales) por ejemplo:

$$f(n) = \begin{cases} n^4 & \text{si n es par} \\ n^2 & \text{si n es impar} \end{cases} \qquad g(n) = \begin{cases} n^2 & \text{si n es par} \\ n^3 & \text{si n es impar} \end{cases}$$

En tales casos, la regla de la suma se aplica directamente:

$$O(max(f(n),g(n))) \begin{cases} n^4 & \text{si n es par} \\ n^3 & \text{si n es impar} \end{cases}$$

REGLA DEL PRODUCTO.

Si $T_1(n)$ y $T_2(n)$ son $O(f(n))$ y $O(g(n))$ respectivamente, entonces $T_1(n)T_2(n)$ es $O(f(n)g(n))$. Hay que tener presente que al aplicar esta regla $O(Cf(n))$ significa lo mismo que $O(g(n))$ si C es una constante positiva cualquiera. Por ejemplo, $O(n^2/2)$ es lo mismo que $O(n^2)$.

Ejemplo:

Si tenemos el siguiente programa:

```
PROCEDURE BURBUJA (VAR A:ARRAY 1..N OF INTEGER);
{Procedimiento que ordena el vector de menor a mayor}
VAR I, J, Temp: INTEGER;
BEGIN
     FOR I := 1 TO n-1 DO
          FOR J:=n DOWNTO I+1 DO
              IF A[J-1] > A[J] THEN
              BEGIN
                   Temp   := A[J-1];
                   A[J-1] := A[J];
                   A[J]   := Temp;
              END;
END;
```

El número n de elementos que se van ordenar es la medida apropiada del tamaño de entrada. Independientemente del tamaño de

la entrada cada proposición toma cierta cantidad constante de tiempo, sea un tiempo O(1); por regla de la suma, el tiempo de ejecución combinado de este grupo de proposiciones es O(max(1,1,1)) = O(1).

En cuanto a la proposición IF, la prueba de la condición requiere un tiempo O(1), no se sabe si esta proposición se ejecutará, pero dado que se busca el tiempo del peor caso, esto es, que se ejecute. Este grupo de instrucciones requiere un tiempo O(1).

El tiempo de ejecución total de un ciclo resulta de sumar, en cada iteración, los tiempos empleados en ejecutar el cuerpo del ciclo en esa iteración. En cada iteración se acumula al menos O(1) ya que el índice sufre un incremento o decremento para verificar después si se alcanzó el límite o se continúa ejecutando el ciclo.

Para el ciclo interno el cuerpo tarda un tiempo O(1) en cada iteración, como el número de iteraciones es n-i, por regla del producto el total de tiempo invertido en este ciclo es O((n-i)*1), o sea O(n-i).

Dado que el ciclo externo se ejecuta n-1 veces, el tiempo de ejecución del programa tiene como cota superior una constante multiplicada por:

$$\sum_{i=1}^{n-1}(n-i) = n(n-1)/2 = n^2/2 - n/2$$

que es $O(n^2)$, por tanto para ejecutar ese programa se requiere un tiempo proporcional al cuadrado del número de elementos que se desean ordenar.

7.1.7 Velocidad de crecimiento del tiempo de ejecución de un programa.

Supóngase que es posible evaluar programas comparando sus funciones de tiempo de ejecución sin considerar las constantes de proporcionalidad, entonces un programa con tiempo de ejecución $O(n^2)$ es mejor que uno con un tiempo $O(n^3)$. Sin embargo, además de los factores constantes debidos al compilador y a la máquina, existe un factor constante debido a la naturaleza del programa mismo. Es posible, por ejemplo, que con una combinación determinada de compilador y máquina, el primer programa tarde $100n^2$ milisegundos, mientras que el segundo tarda $5n^3$ milisegundos. En este caso ¿No es preferible el segundo programa al primero?

La respuesta a esto depende del tamaño de las entradas que se espera procesen los programas. Para entradas de tamaño $n < 20$ el

programa con tiempo de ejecución $5n^3$ será más rápido que el de tiempo de ejecución $100n^2$.

De esta forma si el programa emplea entradas pequeñas es preferible aquel cuyo tiempo de ejecución es $O(n^3)$. No obstante conforme n crece, la razón de los tiempos de ejecución, que es $5n^3/100n^2 = n/20$ se hace arbitrariamente grande. Así, a medida que crece el tamaño de la entrada, el programa $O(n^3)$ requiere un tiempo significativamente mayor que el programa $O(n^2)$. Esto se muestra en la Figura 7.1.

Si existen algunas entradas grandes en los problemas para cuya solución se hayan diseñado estos programas, será mejor optar por el programa cuyo tiempo de ejecución tiene la menor **VELOCIDAD DE CRECIMIENTO.**

La velocidad de crecimiento de un programa es quien determina el tamaño del problema que se puede resolver en una computadora, teniendo presente que conforme las computadoras se hacen más veloces; también aumenta el deseo de los usuarios de resolver problemas más grandes.

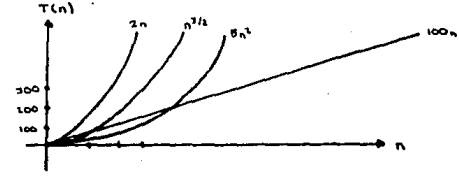


FIG 7.1 Tiempo de ejecución de un programa.

7.1.8 Algunas funciones comunes del tiempo de ejecución de un programa.

Las funciones respecto al tiempo de ejecución de un proceso, que se presentan más frecuentemente son:

- **Log N** Cuando el tiempo del programa es logarítmico comúnmente los procesos resuelven problemas grandes, transformándolos en problemas más pequeños, reduciendo para ello su tamaño a una fracción constante.

- **N** Generalmente un programa tiene una función de tiempo lineal cuando una pequeña cantidad del proceso es dada por cada elemento de entrada.

- **N Log N** El tiempo se eleva en los programas que resuelven un

problema por medio de pequeños subproblemas resolviendo estos independientemente y combinando la solución.

- N^2 El tiempo de un algoritmo es cuadrático cuando generalmente se emplean ciclos doblemente anidados.

- N^3 Los procesos que emplean ciclos triplemente anidados comúnmente emplean un tiempo cúbico en su ejecución.

- 2^n Existen muy pocos algoritmos con tiempo exponencial que son apropiados para ser empleados.

- 1 Se presenta cuando el tiempo de cálculo es constante.

Por ejemplo:

Si se tiene la siguiente función

$$F(n) = N^4 + 100N^2 + 10N + 50$$

El tiempo de la función F(n) es del orden N^4 ya que este exponente domina la función.

7.2 TECNICAS DE ANALISIS DE ALGORITMOS.

Es relativamente fácil inventar algoritmos, sin embargo, no sólo se desean algoritmos, sino buenos algoritmos que den solución en forma eficiente al problema dado.

Por consiguiente, para realizar un buen algoritmo es preciso estudiar los métodos de análisis que nos conducen a una o más soluciones de un problema.

En el Subcapitulo 7.1 (Complejidad de un Algoritmo), se abordaron las técnicas básicas para establecer y analizar el tiempo de ejecución de programas simples, es decir, de programas no recursivos (procesos que no se llaman a sí mismos); sin embargo, cuando los programas son recursivos, se requiere de técnicas nuevas para llevar a cabo su análisis.

7.2.1 Algoritmos Recursivos.

Un objeto es recursivo, si está definido parcialmente en términos de sí mismo.

En Computación, la herramienta necesaria y suficiente mediante la cual podemos expresar los programas en forma recursiva es el

procedimiento o la subrutina, que permiten dar a una proposición un nombre por medio del cual puede ser llamado.

Ahora, si un procedimiento P contiene una referencia explícita a si mismo, entonces, se dira que P es directamente recursivo; si P contiene una referencia a otro procedimiento Q que contiene una referencia (directa o indirecta) a P, entonces se dira que P es indirectamente recursivo. Por lo que se ve que el uso de la recursión puede no ser inmediatamente visible en el texto (o código) del programa.

Por lo tanto, un procedimiento que se llama a si mismo directa o indirectamente, es llamado recursivo.

Un procedimiento recursivo bien definido debe cumplir con las siguientes caracteristicas:

1.- Debe existir un criterio (denominado criterio base) por el cual el procedimiento no se llame a si mismo

2.- Cada vez que el procedimiento se llame a si mismo (directa o indirectamente), debe estar más cerca del criterio base.

A su vez, una función recursiva bien definida cumple con:

1.- Tener ciertos argumentos (valores base) para los que la función no se refiera a si misma.

2.- Cada vez que la función se refiera a si misma el argumento de la función debe acercarse más al valor base.

Ejemplo I: Procedimiento para Calcular el Factorial de un Número.

```
PROCEDURE FACTORIAL (Fact, N);
 Este procedimiento calcula N! y devuelve el valor en la variable
Fact
BEGIN
     IF N = 0 THEN Fact := 1
     ELSE
          BEGIN
               FACTORIAL(Fact,N-1);
               Fact := N * Fact;
          END;
END;
```

Es importante saber que cuando cada procedimiento se activa de un modo recursivo se crea un nuevo conjunto de variables locales acotadas con valores específicos, esto es difícil de comprender, pero es explicable si consideramos que:

En el corazón de la implementación de procedimientos recursivos se encuentra una pila (stack) en donde se almacenan los datos usados por cada llamada de un procedimiento que todavía no ha terminado. Esto es, todos los datos no globales (locales) están en una pila. La pila es dividida en marcos (stack frames), los cuales son bloques de localidades (registros) consecutivos.

Cada llamada de un procedimiento usa un marco de pila cuya longitud depende de la llamada al procedimiento en particular.

Suponiendo que el procedimiento A ésta actualmente en ejecución. La pila aparecería como

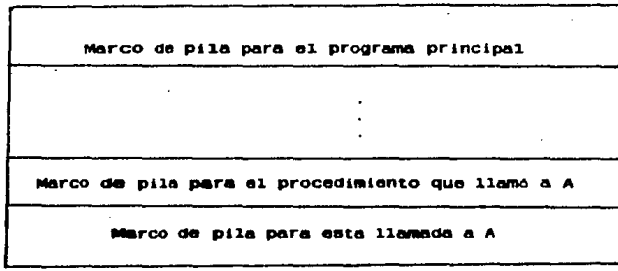


FIG 7.2 Pila para llamadas a procedimientos recursivos.

Si A llama al procedimiento B, se hace lo siguiente:

1. Se coloca un marco de pila del tamaño adecuado en el tope, en un orden conocido para B:

   a) Los apuntadores a los parámetros actuales para esta llamada a B.
   b) Espacio vacío para las variables locales usadas por B.
   c) La dirección de la siguiente instrucción en la rutina A que debe ser ejecutada después de que la llamada a B termina (la dirección de regreso). Si B es una función que regresa un valor, también se coloca en el marco para B un apuntador a la localidad en el

marco de A en la cual se encontrará el valor de la función (la dirección del valor).

2. Pasa control a la primera instrucción de B. La dirección del valor de algún parámetro o identificador local pertenecientes a B se encuentran por medio de índices en el marco de pila para B.

3. Cuando B termina, se regresa el control a A mediante la siguiente secuencia de pasos:

   a) La dirección de regreso se obtiene del tope de la pila.
   b) Si B es una función, el valor denotado por la porción de la expresión de la instrucción **RETURN** (regreso) es almacenada en la localidad prescrita por la dirección del valor en la pila.
   c) El marco de pila para el procedimiento B se borra de la pila, que deja el marco para el procedimiento A situado en el tope de la pila.
   d) La ejecución de A se reanuda en la localidad dada por la dirección de regreso.

Por último, el tiempo requerido por una llamada a un procedimiento es proporcional al tiempo requerido para evaluar los parámetros actuales y almacenar los apuntadores a sus valores en la pila, el tiempo para un regreso no más grande que éste.

En la contabilidad para el tiempo gastado por una colección de procedimientos recursivos, es más fácil cargar el costo de una llamada al procedimiento haciendo la llamada, entonces se puede limitar, como una función del tamaño de entrada, el tiempo gastado por una llamada de cada procedimiento, con exclusión del tiempo gastado por los procedimientos que éste llamó. Sumando este límite total de llamadas de procedimientos da un límite superior en el tiempo total gastado.

Para calcular la complejidad del tiempo de un algoritmo recursivo, hacemos uso de ecuaciones de recurrencia. Una función $T_i(n)$ es asociada con el *i-ésimo* procedimiento y denota el tiempo de ejecución del *i-ésimo* proceso como una función de algún parámetro n de la entrada. Usualmente uno puede expresar una ecuación de recurrencia para $T_i(n)$ en términos de los tiempos de ejecución de los procesos llamados por el procedimiento i. El conjunto resultante de ecuaciones de recurrencia simultáneas debe resolverse y con ello se obtiene la complejidad del tiempo total. Muchas veces sólo uno de los procedimientos está envuelto, y $T(n)$ depende de los valores de $T(m)$ para un conjunto finito de m menor que n.

Hay que tener en cuenta que aquí, y en otro lado, todos los análisis de costo asumen la función de costo uniforme. Si usamos la función de costo logarítmica, la longitud de la pila usada para implementar procedimientos recursivos puede afectar el análisis de la complejidad de tiempo.

Finalmente, el uso de la recursión nos presenta varias ventajas. Primero, permite descripciones más claras y concisas de los algoritmos que las que pudieran ser posibles sin recursión, además de que es muchas veces más fácil entender programas recursivos.

Pero, **¿Cuándo usar la recursión?**

Los algoritmos recursivos son particularmente apropiados cuando el problema o los datos a ser tratados son definidos en términos recursivos. Sin embargo, esto no significa que tal definición recursiva garantice que un algoritmo recursivo es la mejor manera para resolver el problema.

Las malas explicaciones del concepto de recursividad, mediante ejemplos inapropiados, son la causa principal de una mala aplicación de dicho concepto, creando incomprensión y antipatía hacia el uso de la recursión en la programación y en algunas ocasiones el hecho de que se considere equivalente la recursión con la ineficiencia.

Hay también esquemas de composición recursiva muy complicados, que pueden y deberían ser traducidos en una forma iterativa; el hecho de que la implementación de procedimientos recursivos en máquinas esencialmente no-recursivas se pueda dar, prevee que para propósitos prácticos todo programa recursivo puede ser transformado en uno puramente iterativo. Este, sin embargo, envuelve el manejo explícito de una pila de recursión, y estas operaciones a menudo obscurecen la esencia de un programa, de tal forma que llega a ser más difícil de comprender.

En conclusión los algoritmos, que por su naturaleza son recursivos antes que iterativos, deben ser formulados como procedimientos recursivos.

Anteriormente se dio un ejemplo de un procedimiento recursivo, a continuación se presenta el mismo procedimiento, pero en forma iterativa, como un ejemplo de que todo programa recursivo puede ser traducido a un programa iterativo:

```
PROCEDURE FACTORIAL (Fact, N);
{Este procedimiento calcula N! y devuelve el valor en la variable}
Fact
BEGIN
      Fact := 1;
      FOR I := 1 TO N DO
          Fact := Fact * I;
END;
```

7.2.2 Análisis de programas recursivos.

El análisis de un programa recursivo suele implicar la solución de una ecuación de diferencias (o recurrencia). Las técnicas para la solución de ecuaciones de diferencias algunas

veces son sutiles, y tienen una considerable semejanza con los métodos de solución de ecuaciones diferenciales.

Para ejemplificar el análisis de programas recursivos, nos basaremos en el procedimiento MERGESORT, cuyo algoritmo es el siguiente:

```
FUNCTION MERGESORT (L:Lista; n:Entero); Lista;
{L es una lista de longitud n. Se devuelve una versión ordenada
de L.}
{Se supone que n es una potencia de 2.}
VAR
   L1, L2 : Lista;
BEGIN
     IF n = 1 THEN
        RETURN (L)
     ELSE
     BEGIN
          {Particionar L en dos listas, L1 y L2, cada una de
           longitud n/2          }
          RETURN  (COMBINA (MERGESORT(L1,n/2),
                            MERGESORT(L2,n/2));
     END;
END; {MERGESORT}
```

Procedimiento Recursivo MERGESORT.

El procedimiento **COMBINA** (L1,L2) toma como entrada las listas ordenadas L1 y L2, y recorre cada una, elemento por elemento, desde el inicio. En cada paso, el mayor de los dos elementos delanteros se borra de esta lista y se emite como salida. El resultado es una sola lista clasificada con los elementos L1 y L2. Los detalles de **COMBINA** no son importantes en este momento, lo importante es que el tiempo empleado por **COMBINA** en listas de longitud n/2 es O(n).

Sea T(n) el tiempo de ejecución en el peor caso del procedimiento **MERGESORT**. se escribe una ecuación de recurrencia (o diferencias) que acote T(n) por arriba, como sigue

$$T(n) <= \begin{cases} c_1 & \text{si } n = 1 \\ 2T(n/2) + c_2 n & \text{si } n > 1 \end{cases} \qquad (a)$$

donde c_1 representa el número constante de pasos dados cuando L tiene longitud 1.

En el caso de que n > 1, el tiempo requerido por **MERGESORT** puede dividirse en dos partes. Las llamadas recursivas a

MERGESORT con listas de longitud **n/2** con un tiempo **T(n/2)**, de aquí el término **2T(n/2)**. La segunda parte consiste en la prueba para descubrir que **n <> 1**, la división de la lista **L** en dos partes iguales y el procedimiento **COMBINA**. Esas tres operaciones requieren un tiempo que puede ser una constante, en el caso de la prueba, o ser proporcional a **n** al dividir y combinar. Así, la constante C_2 puede escogerse de modo que el término C_2n sea una cota superior del tiempo requerido por **MERGESORT** para hacer todo excepto las llamadas recursivas **(a)**.

Se observa que **(a)** sólo se aplica cuando **n** es par, por lo que generará una cota superior en forma cerrada (esto es, como una fórmula para **T(n)** que no implica ningún **T(m)** para **m < n** sólo cuando **n** es una potencia de 2. Sin embargo, aunque sólo se conozca **T(n)** cuando **n** es una potencia de 2, se tiene una buena idea de **T(n)** para toda **n**. En particular, para casi todos los algoritmos, se puede suponer que **T(n)** está entre **T(2 ** i)** y **T(2 ** i+1)** si **n** está entre **2 ** i** y **2 ** i+1**. Y con un poco más de esfuerzo para encontrar la solución, se puede reemplazar el término **2T(n/2)** de **(a)** por **T((n+1)/2) + T((n-1)/2)** para **n > 1** impar. Después, se puede resolver la nueva ecuación de diferencias para obtener una solución cerrada para toda **n**.

7.2.3 Resolución de ecuaciones de recurrencia.

Existen dos enfoques distintos para resolver una ecuación de recurrencia.

1. Conjeturar una solución **f(n)** y usar la recurrencia para mostrar que **T(n) <= f(n)**. Algunas veces sólo se conjetura la forma de **f(n)**, dejando algunos parámetros sin especificar (por ejemplo, supóngase **f(n) = an^2** para alguna **a** y dedúzcase valores adecuados para los parámetros al intentar demostrar que **T(n) <= f(n)** para toda **n**.

2. Utilizar la recurrencia misma para sustituir **m < n** por cualquier **T(m)** para **m > 1** se hayan reemplazado por fórmulas que impliquen sólo **T(1)**. Como **T(1)** siempre es constante, se tiene una fórmula para **T(n)** en función de **n** y de algunas constantes. A esta fórmula se le ha denominado **<<forma cerrada>>** para **T(n)**.

CONJETURA DE UNA SOLUCION.

Ejemplo I:
considérese el método **1** aplicado a la ecuación **(a)**

Supóngase que para alguna **a**, se observa que esta conjetura no funcionará, porque **a*n Log n** tiene valor **0**, independientemente del valor de **a**. Así, se intenta a continuación **T(n)=ab Log n + b**. Ahora, $n_1 = 1$ requiere que **b >=** C_1.

Para emplear la inducción, se supone que

$$T(K) <= aK \text{ Log } K + b \qquad (b)$$

para toda K<n y se intenta establecer que T(n)<= an Log n + b.

Para realizar la demostración, se supone que n >= 2.

Sustituyendo en (a) tenemos:

$$T(n) <= 2T(n/2) + C_2 n$$

Sustituyendo en (b), con K = n/2, se tiene

$$T(n) <= 2 \quad a * n/2 * \text{Log } n/2 + b \quad + C_2 n \qquad (c)$$

$$<= an\text{Log}n - an + C_2 n + 2b$$

$$<= an \text{ Log } n + b$$

siempre que a >= C_2 + b.

Así, T(n) <= anLog n + b siempre y cuando se satisfagan dos restricciones: b >= C_1 y a >= C_2 + b. Por suerte, existen valores que se le pueden dar a a que satisfacen ambas restricciones. Por ejemplo, al elegir b = C_1 y a = C_1 + C_2, por inducción sobre n, se concluye que para toda n >= 1.

$$T(n) <= (C_1 + C_2)n \text{ Log } n + C_1 \qquad (d)$$

Es decir, T(n) es O(n Log n).

Son útiles dos observaciones sobre el ejemplo I. Si se supone que T(n) es O(f(n)), y si el intento de probar que T(n) <= Cf(n) por inducción falla, se sigue que T(n) no sea O(f(n)).

En segundo lugar, aún no se ha determinado la tasa exacta de crecimiento asintótica para f(n), aunque se ha demostrado que no es peor que O(n Log n). Si se conjetura una solución más lenta, como f(n) = an, o f(n) = an Log n, no se puede demostrar la validez de T(n) <= f(n).

El ejemplo I expone una técnica general para demostrar que alguna función es una cota superior en el tiempo de ejecución de un algoritmo.

Supóngase la ecuación de recurrencia

$$T(1) = C$$

$$T(n) <= g(T(n/2),n), \text{ para } n>1 \qquad (e)$$

Obsérvese que (e) generaliza (a), donde g(x,y) es 2x + C_2y. También obsérvese que podrían imaginarse ecuaciones más generales que (e). Por ejemplo, la fórmula g puede comprender todos los T(n-1), T(n-2),..., T(1), y sólo T(n/2). También, es posible

tener valores para T(1), T(2),...,T(k), y la recurrencia sólo sería aplicable para n > k.

Considérese ahora (e) en lugar de sus generalizaciones. Supóngase una función $f(a_1,\ldots,aj, n)$, donde $a_1,\ldots,aj$ son parámetros e intentese demostrar por inducción en n que $T(n) <= f(a_1,\ldots,aj,n)$. Para probar que para algunos valores de $a_1,\ldots,aj$ se tiene $T(n) <= f(a_1,\ldots, aj, n)$ para toda $n >= 1$, se debe cumplir

$$f(a_1, \ldots, aj, 1) >= C \qquad (f)$$

$$f(a_1, \ldots, aj, n) >= g(f(a_1, \ldots, aj, n/2), n)$$

Esto es, por la hipótesis inductiva, se puede sustituir f por T en el lado derecho de la recurrencia (e) para obtener

$$T(n) <= g(f(a_1, \ldots, a,n/2), n) \qquad (g)$$

cuando se cumple la segunda línea de (f), se combina con (g) para demostrar que $T(n) <= f(a_1, \ldots,an, n)$, que es lo que se deseaba demostrar por inducción en n.

En el ejemplo I, $g(x,y) = 2x + C_2y$ y $f(a_1,a_2,n) = a_1n \text{ Log } n + a_2$. Aquí se intenta satisfacer $f(a_1,a_2,1) = a_2 >= C_1$, $f(a_1,a_2,n) = 2a_1n \text{ Log } n + a_2 >= 2(a_1(n/2) + a_2) + C_2n$. Como se analizó, $a_2 = C_1$ y $a_1 = C_1 + C_2$ son una elección satisfactoria.

EXPANSION DE RECURRENCIAS.

Si no se puede conjeturar una solución, o no existe la seguridad de tener la mejor cota en T(n), se usa un método que, en principio, siempre es adecuado para la solución exacta de T(n), aunque en la práctica muchas veces surgen problemas al sumar series y hay que recurrir al cálculo de una cota superior en la suma. La idea general es tomar una recurrencia como (a), que indica que $T(n) <= 2T(n/2) + C_2n$, y emplearla para obtener una cota en T(n/2) sustituyendo n/2 por n. Esto es,

$$T(n/2) <= 2T(n/4) + C_2n/2 \qquad (h)$$

Al sustituir el lado derecho de (h) por T(n/2) en (a), se obtiene

$$T(n) <= 2\,(2T(n/4) + C_2n/2) + C_2n = 4T(n/4) + 2C_2n \qquad (i)$$

Del mismo modo, se sustituye n/4 por n en (a) y se emplea para obtener una cota superior en T(n/4). Esa cota, $2T(n/8) + C_2n/4$, se sustituye en el lado derecho de (i) para obtener

$$T(n) <= 8T(n/8) + 3C_2n \qquad (j)$$

Ahora es necesario establecer un patrón. Por inducción en (i) se obtiene la relación

$$T(n) <= 2\,T(n/(2**i)) + iC_2n \qquad (k)$$

para cualquier i. Si se supone que n es una potencia de 2, como $2**k$, este proceso de expansión termina tan pronto como se alcance T(1) en el lado derecho de (k), lo que ocurre cuando i=k, quedando (k) como

$$T(n) <= 2**k\,T(1) + kC_2n \qquad (1)$$

Entonces, como $2**k = n$, se sabe que $k = \text{Log } n$. Como $T(1) <= C_2(1)$ es

$$T(n) <= C_1n + C_2n \text{ Log } n \qquad (m)$$

La ecuación (m) es en realidad la mejor cota que se pudo alcanzar en T(n), y demuestra que T(n) es O(n Log n).

7.2.4 Solución general para una clase grande de recurrencias.

Considérese la recurrencia que surge al dividir un problema de tamaño n en a subproblemas de tamaño n/b. Por conveniencia, se supone que un problema de tamaño 1 requiere una unidad de tiempo y que el tiempo para reunir las soluciones de los problemas y obtener una solución del problema de tamaño n es d(n), en las mismas unidades de tiempo.

Entonces, si T(n) es el tiempo para resolver un problema de tamaño n, se tiene

$$T(1) = 1$$

$$T(n) = aT(n/b) + d(n) \qquad (n)$$

Sólo se aplica (n) a las n que sean una potencia de b, pero si se presume que T(n) es continua, tomar una cota superior sobre T(n) para aquellos valores de n, nos indica como crece T(n) en general.

Obsérvese también que se utiliza la igualdad en (n), mientras que en (a) hay desigualdades. La razón es que aquí d(n) puede ser arbitraria y, por lo tanto, exacta, mientras que en (a) la suposición de que C_2n fue el peor caso en tiempo de combinación, para una constante C_2 y toda n, fue sólo una cota superior; el peor caso real del tiempo de ejecución con entradas de tamaño n pudo haber sido menor que $2T(n/2) + C_2n$.

En realidad, hay muy poca diferencia entre utilizar = ó <= en la recurrencia, ya que de cualquier modo se obtiene una cota superior para el peor caso del tiempo de ejecución.

Para resolver (n) se aplica la técnica de sustituciones repetidas para T en el lado derecho, igual que se hizo para un ejemplo específico en la exposición anterior de la expansión de recurrencias. Esto es, la sustitución $n/(b**i)$ por n en la segunda línea de (n) da

$$T(n/(b^{**}i)) = aT(n/(b^{**}(i+1))) + d(n/(b^{**}i)) \qquad (A)$$

Así, al empezar con (n) y sustituir (A) para i=1, 2, ..., se tiene

$$\begin{aligned} T(n) &= aT(n/b) + d(n) = a\,[\,aT(n/b^{**}2) + d(n/b)\,] + d(n) \\ &= a^{**}2\,T(n/b^{**}2) + ad(n/b) + d(n) \\ &= a^{**}2\,[\,aT(n/b^{**}3) + d(n/b^{**}2)\,] \\ &\quad + ad(n/b) + d(n) \\ &= a^{**}3\,T(n/b^{**}3) + (a^{**}2)(d(n/b^{**}2)) \\ &\quad + ad(n/b) + d(n) \\ &= \ldots \\ &= a^{**}i\,T(n/b^{**}i) + \sum_{j=0}^{i-1} a^{**}j\,d(n/b^{**}j) \end{aligned}$$

Ahora, suponiendo que n = b**k, se puede utilizar el hecho de que T(n/b**k) = T(1) = 1, para tener de lo anterior, con i = k, la fórmula

$$T(n) = a^{**}k + \sum_{j=0}^{k-1} a^{**}j\;d(b^{**}(k-j)) \qquad (o)$$

Si se aplica el hecho de que k = Log de base b de n, el primer término de (o) puede escribirse como a**Log base b de n o, de forma equivalente, n**Log de base b de a (se toman logaritmos de base b de ambas expresiones para ver que son lo mismo). Esta expresión es n a una potencia constante. Por ejemplo, en el caso del **MERGESORT**, donde a=b=2, el primer término es n. En general, cuanto mayor sea a (cuantos más subproblemas haya que resolver), tanto mayor será el exponente; cuanto mayor sea b (cuanto menor sea cada subproblema), tanto menor será el exponente.

SOLUCIONES HOMOGÉNEAS Y PARTICULARES

Los diferentes papeles que desempeñan los dos términos de (o), son:

Primero: a**k ó n**Log de base b de a, se conoce como solución homogénea, en analogía con la terminología de las ecuaciones diferenciales. La solución homogénea es la solución exacta cuando d(n), conocida como función motriz, es 0 para toda n. En otras palabras, la solución homogénea representa el costo de resolver todos los subproblemas, aunque puedan combinarse sin costo.

Por otra parte: el segundo término de (o) comprende el costo de creación de los subproblemas y la combinación de sus resultados. Este término se denomina solución particular, y está afectada tanto por la función motriz como por el número y el tamaño de los subproblemas. Como regla general, si la solución homogénea es mayor que la función motriz, la solución particular tendrá la misma tasa de crecimiento que la solución homogénea. Si

la función motriz crece más rápido que la solución homogénea en más que n^{ε} para alguna $\varepsilon>0$, la solución particular tendrá la misma tasa de crecimiento que la función motriz. Si la función motriz tiene la misma tasa de crecimiento que la solución homogénea, o crece más rápido que $\log^{k} n$ de n como mucho para alguna k, entonces la solución particular crecerá $\log n$ veces la función motriz.

Es importante reconocer que cuando se buscan mejoras en un algoritmo es necesario saber si la solución homogénea es mayor que la función motriz. Por ejemplo, si la solución homogénea es mayor, prácticamente no tendrá efecto encontrar una forma más rápida de combinar los subproblemas, en la eficiencia de todo el algoritmo . Lo mejor, en este caso, es encontrar una forma de dividir el problema en menos o menores subproblemas. Eso afectará a la solución homogénea y reducirá el tiempo total de ejecución.

Si la función motriz excede a la solución homogénea, entonces es necesario tratar de reducir la función motriz. Por ejemplo, en el caso del MERGESORT, donde $a = b = 2$, y $d(n) = Cn$, la solución particular es $O(n \log n)$. Sin embargo, reducir $d(n)$ a una función ligeramente sublineal, por ejemplo, $n/\log 0.9$, hará que la solución particular sea también menos que lineal y que se reduzca el tiempo de ejecución total a $O(n)$, que es la solución homogénea.

FUNCIONES MULTIPLICATIVAS.

La solución particular de (o) es difícil de evaluar, aún sabiendo lo que es d(n). Sin embargo para ciertas funciones d(n) comunes, se puede resolver (o) con exactitud, y hay otras para las cuales se puede conseguir una buena cota superior. Se dice que una función f en enteros es multiplicativa si $f(xy)=f(x)f(y)$ para todos los enteros positivos x E y.

Ejemplo II:
Las funciones multiplicativas de mayor interés son de la forma n^{a} para cualquier a positiva. Para demostrar que $f(n) = n^{a}$ es multiplicativa, sólo hay que observar que $(xy)^{a}=x^{a}y^{a}$.

Ahora, si d(n) de (o) es multiplicativa, entonces :

$$d(b^{a}(k-j)) = (d(b))^{a(k-j)},$$

y la solución particular de (o) es

$$\sum_{j=0}^{k-1} a^{j}(d(b))^{a(k-j)} = d(b)^{ak}\sum_{j=0}^{k-1}(a/d(b))$$

$$= d(b)^{ak}((a/d(b))^{k}-1)/(a/d(b)-1)$$

$$= (a^{k}-d(b)^{ak})/(a/d(b)-1) \qquad (p)$$

Se consideran 3 casos, dependiendo de si a es mayor, menor o igual que d(b).

1. Si $a > d(b)$, la fórmula (p) es $O(a^{\log_b a})$, que como se recuerda es $n^{\log_b a}$ de base b de a, ya que $k = \log$ base b de n. En

este caso, las soluciones particular y homogénea son iguales, y sólo dependen de a y b, y no de la función motriz d. Así, las mejoras en el tiempo de ejecución deben proceder de disminuir a o aumentar b; la disminución de d(n) no es muy útil.

2. Si a < d(b), (p) es O(d(b)**k) o, en forma equivalente, O(n). La solución particular en este caso, excede a la homogénea, y se puede dirigir la atención también hacia la función motriz d(n), además de a y b, para obtener mejoras. Obsérvese el importante caso especial en que d(b) = n**a. Entonces, d(b)=b**a y log (b**a) = a. Así la solución particular es O(n**a) o bien O(d(n)).

3. Si a = d(b), se reconsideran los cálculos implicados en (p), pues la fórmula para la suma de una serie geométrica no es apropiada.

En este caso,

$$\sum_{j=0}^{k-1} a^j (d(b))^{k-j} = d(b)^k \sum_{j=0}^{k-1} (a/d(b))^j$$

$$= d(b)^k \sum_{j=0}^{k-1} I = d(b)^k k = n^{\log_b a} \log_b n \qquad (q)$$

Como a = d(b), la solución particular dada por (q) es Log n veces la solución homogénea, y de nuevo la solución particular excede a la homogénea. En el caso especial d(n) = n**a, (q) se reduce O(n**a Log n), por observaciones similares a las del caso (2).

Ejemplo III.
Considérense las siguientes recurrencias, con T(1)=1.

1. $T(n) = 4T(n/2) + n$
2. $T(n) = 4T(n/2) + n^2$
3. $T(n) = 4T(n/2) + n^3$

En cada caso, a=4, b=2, y la solución homogénea en n^2. En la ecuación (1), con d(n)=n, se tiene d(b)=2. Como a=4 > d(b), la solución particular también es n^2, y T(n) es $O(n^2)$ en (1).

En la ecuación (3), $d(n) = n^3$, d(b) = 8, y a < d(b). Así, la solución particular es $O(n^{\log_2 8}) = O(n^3)$, y T(n) de la ecuación (3) es $O(n^3)$. Se puede deducir que la solución particular es del mismo orden que $d(n) = n^3$, aplicando las observaciones anteriores acerca de los d(n) de la forma n**a y analizando el caso a <d(b) de (p). En la ecuación (2), se tiene d(b) = 4 = a, con lo que se aplica (q). Como d(n) es de la forma n**a, la solución particular y, por tanto, T(n), es $O(n^2 \text{ Log } n)$.

OTRAS FUNCIONES MOTRICES.

Existen otras funciones motrices no multiplicativas, por medio de las cuales se obtienen soluciones para (o) e incluso para (p). Se considerarán dos ejemplos: El primero generaliza cualquier función que sea el producto de una función multiplicativa y una constante mayor o igual que uno. El segundo es típico de un caso donde es preciso examinar (o) con detalle y obtener una cota superior ajustada a la solución particular.

Ejemplo IV:
Considérese

$$T(1) = 1$$

$$T(n) = 3T(n/2) + 2n^{1.5}$$

Ahora, $2n^{1.5}$ no es multiplicativa, pero $n^{1.5}$ sí lo es. Sea $T(n) = T(n)/2$ para toda n. Entonces,

$$U(1) = 1/2$$

$$U(n) = 3U(n/2) + n^{1.5}$$

La solución homogénea, si U(1) fuera 1, sería $n^{\log_2 3} = n^{1.59}$; como $U(1) = 1/2$, se demuestra con facilidad que la solución homogénea es $n^{1.59}/2$; y es $O(n^{1.59})$. Para la solución particular se ignora el hecho de que $U(1) \neq 1$, dada que al incrementar U(1) seguramente no se reducirá la solución particular. Como $a = 3$, $b=2$ y $b^{1.5} = 2.82 < a$, la solución particular también es $O(n^{1.59})$, que es la tasa de crecimiento de U(n). Como T(n) = 2U(n), T(n) también es $O(n^{1.59})$ o $O(n)^{\log_2 3}$.

Ejemplo V:
Considérese

$$T(1) = 1$$

$$T(n) = 2T(n/2) + n \text{ Log } n$$

Es fácil ver que la solución homogénea es n, pues a=b=2. Sin embargo,
d(n) = n Log n no es multiplicativa, y se debe sumar la fórmula de la solución particular de (o) por medios apropiados. Esto es, se desea evaluar

$$\sum_{j=0}^{k-1} 2^j \, 2^{k-j} \text{Log } 2 = 2^k \sum_{j=0}^{k-1} (k-j)$$

$$= 2^{k-1} k(k+1)$$

Como $k = \text{Log } n$, la solución particular es $O(n \text{ Log}^2 n)$, y esta solución, al ser mayor que la homogénea, también es el valor que se obtiene de T(n).

7.3 TECNICAS DE DISEÑO DE ALGORITMOS.

El término algoritmo es universalmente empleado en la computación para describir métodos para resolver problemas, que serán implementados como programas ejecutables en una computadora. Por lo que, el diseñador de un algoritmo debe tener mucho cuidado en describir con precisión el proceso que desea se realice y tomar en cuenta todas las posibilidades, y todas las posibles circunstancias. Sin embargo, un diseñador tiene pocas posibilidades de éxito a menos que adopte un enfoque rigurosamente metódico.

Ahora cuando nosotros queremos resolver un problema dado, analizamos una serie de algoritmos y dependiendo de los resultados, seleccionamos el que mejor trabaja en un conjunto dado de circunstancias.

A través de los años los científicos de la computación han desarrollado diversas técnicas, las que son aplicables en ciertas circunstancias bien definidas, y con frecuencia producen algoritmos eficientes para la resolución de muchos tipos de problemas. Así que, existe un número de técnicas básicas e ideas que deben ser parte del conocimiento del trabajo de cualquiera que diseñe algoritmos.

El objetivo de este subcapítulo es presentar algunas de las técnicas de diseño de algoritmos conocidas, en forma general.

7.3.1 Técnica de inducción.

El concepto de inducción matemática podría ser distinguido de lo que usualmente conocemos como razonamiento inductivo en la Ciencia, o sea, un científico toma observaciones y por "inducción" crea una teoría general o hipótesis que funciona para esos hechos. La "inducción" no es más que una suposición acerca de una situación.

Inducción como un término matemático se refiere a una técnica de prueba para teoremas acerca de enteros. Por ejemplo, si T(n) es un teorema acerca de los enteros

$$T(n): \sum_{i=1}^{n} i = n(n+1)/2$$

Entonces, una prueba por inducción consiste de los dos siguientes pasos:

1. Probar que T(1) es verdadera.

2. Probar que si T(1), ..., T(n-1), T(n) son verdaderas, entonces
   T(n+1) también es verdadera.

Ahora, es posible usar la inducción matemática para proveer cosas acerca de algoritmos.

Si se desea obtener un algoritmo A que resuelva un problema con entrada x de tamaño n, y se postula que la entrada puede ser representada como n elementos individuales, es decir, x = (x$_1$, ..., xn); entonces la técnica de inducción consiste de los siguientes pasos:

1. Encontrar un algoritmo B el cual resuelva el problema para n=1.

2. Encontrar un algoritmo C el cual toma como entrada $\bar{x}_i$, la solución para x_1, ..., x_i y x_{i+1} y producir la solución $\bar{x}_{i+1}$.

Dados esos dos algoritmos A y B, la solución $\bar{x}_n$ del problema original es encontrada por

```
Algoritmo A
BEGIN
     x̄1 := B(X1);
     i := 1;
     WHILE (i<n)[x̄i+1 := C(xi, x̄i, xi+1), i := i+1]
END
```

7.3.2 Técnicas básicas.

A continuación se da una breve descripción de tres técnicas o métodos básicos que han sido encontrados útiles en el diseño de algoritmos.

I. METODO DE SUBMETAS.

Este primer método involucra la reducción de un problema difícil en una secuencia de problemas simples, esperando que los problemas más simples sean más tratables que el problema original, y que las soluciones a los problemas simples puedan ser organizados en una solución para el problema inicial.

Este método suena razonable, pero muchas veces la elección juiciosa de los subproblemas, es una tarea difícil de realizar, sin embargo, las respuestas a las siguientes preguntas nos pueden ayudar en esta elección:

1. ¿Podemos resolver parte del problema? ¿Es posible ignorar algunas condiciones y resolver el resto del problema?

2. ¿Podemos resolver el problema para casos especiales? ¿Es posible diseñar un algoritmo que produzca una solución que satisfaga todas las condiciones del problema, pero cuyas entradas sean restringidas a un subconjunto de todas las entradas?

3. ¿Existe algún aspecto del problema, que no se entendió completamente? ¿Si nos esforzamos para entender mejor algunas características del problema, es posible aprender algo que nos ayude a acercarnos a la solución?

4. ¿Existe algún problema similar cuya solución se conoce? ¿Podría esta solución ser modificada para resolver el problema? ¿Es posible que el problema sea equivalente a un problema insoluble conocido?

II. METODO HILL - CLIMBING (Escalando la colina).

Un algoritmo hill-climbing realiza una suposición inicial o cálculo inicial del problema. Procediendo a moverse "cuesta arriba" desde esta solución hacia mejores soluciones tan rápido como sea posible. Cuando el algoritmo alcanza un punto en el que ya no puede moverse ascendentemente, el algoritmo se detiene.

Desafortunadamente, no siempre se puede garantizar que la solución final producida, aplicando el método hill-climbing sea la óptima.

Escalando la colina, toma su nombre en parte de los algoritmos empleados para encontrar el máximo de funciones de varias variables.

Generalmente, los métodos hill-climbing son "ambiciosos" ya que aún cuando tienen una cierta meta en mente, tratan de hacer cualquier cosa, si es posible, para obtener una meta mejor.

III. METODO WORKING BACKWARD (Trabajando hacia atrás).

Este método inicia en la meta o solución final y camina hacia atrás en busca de la proposición inicial del problema. Entonces, si estos pasos son reversibles, se puede ir de la proposición del problema a la solución. Muchos de nosotros hemos hecho esto mientras resolvemos el enigma del laberinto que aparece en la sección de tiras cómicas en el periódico del domingo.

7.3.3 Técnica de clasificación por resultado por división.

Tal vez la técnica más importante y aplicada para el diseño de algoritmos eficientes sea la estrategia llamada de clasificación por resultado por división, que consiste en la descomposición de un problema de tamaño n en problemas más pequeños, de modo que a partir de la solución de dichos problemas sea posible construir con facilidad una solución al problema completo; la clasificación por intercalación a los árboles binarios de búsqueda, el acertijo de las Torres de Hanoi, el problema de la multiplicación de enteros grandes y la programación de Torneos de Tenis son algunas de las aplicaciones más conocidas de esta técnica.

Algunos métodos para el diseño de algoritmos tienen como principio fundamental la aplicación de esta técnica, los más conocidos son:

A. DIVIDE Y VENCERAS.

El método de dividir y vencer es la vena principal de la metodología de programación estructurada, nos interesó principalmente como un método para dividir una tarea en subtareas ordenadas para obtener un flujo lineal de control, pero ahora, queremos dividir el problema en términos de tamaño de la entrada. Esta estrategia está basada en la observación de que dos veces la mitad no es siempre el todo o el todo algunas veces es mayor que la suma de las partes.

La interpretación de esta observación nos conduce a la siguiente proposición Asumir que tenemos ya un algoritmo **A**, para el problema T y además suponemos que podemos particionar la entrada $X = (X_1, \ldots, Xn)$ en dos partes iguales $Y = (X_1, \ldots, Xn/2)$ y $Z = (X^1, \ldots, Xn)$, aplicando A a Y para obtener $\bar{Y}$, y aplicando **A** a Z para obtener $\bar{Z}$, donde $\bar{Y}$ y $\bar{Z}$ es la salida de la computación anterior respectivamente.

Desafortunadamente, en muchos casos teniendo $\bar{Y}$ y $\bar{Z}$ no implica necesariamente que tenemos $\bar{X}$ la salida de A. y se requiere un proceso de fusionamiento **(merging)** adicional para producir la salida del problema original.

Formalmente planteado, obtenemos el algoritmo **B** de aplicar al algoritmo **A** el método divide-y-venceras como sigue:

```
BEGIN    divide-y-venceras para A en X=(Y,Z) donde X
         es la entrada para el problema
Ȳ := A(Y); Z̄ := A(Z); X̄ := M(Ȳ,Z̄);
END.
```

En general, esto no es fácil y muchas veces requiere de esfuerzo considerable para estructurar un problema de tal manera que la solución de dos problemas idénticos de tamaño n/2 implique una solución para el problema original.

B. REFINAMIENTO DE PASOS (TOP-DOWN).

Es una variación reciente del método Divide y Vencerás. El método consiste en particionar el proceso que va a realizarse en un cierto número de pasos, cada uno de los cuales pueda describirse por medio de un algoritmo que sea más pequeño y más simple que el correspondiente al proceso completo.

Debido a que cada debe ser más simple que el todo, por lo general el diseñador tiene una idea más clara de cómo construirlo, por lo que puede bosquejarlo con más detalle que si intentara manejar todo el algoritmo de una sola vez. Los subalgoritmos pueden a su vez descomponerse en porciones más pequeñas, que a causa de su mayor simplicidad pueden expresarse con más detalle y precisión. El refinamiento del algoritmo continúa de esta forma hasta que cada uno de sus pasos sea lo suficientemente detallado y preciso para permitir que lo ejecute el procesador que se emplee.

C. RECURSION.

En matemáticas y programación de computadoras "recursión" es el nombre dado al método de definición o expresión de una función, procedimiento, constructor de lenguaje, o la solución de un problema en términos de sí mismo.

La recursión es una extensión del método Divide-y-Vencerás, más específicamente, es una manera de representación de la aplicación repetida de Divide-y-Vencerás.

La recursión como una técnica de especificación de algoritmos nos permite describir el proceso bastante intrincado de división de la entrada y de recombinación de las soluciones parciales muy concisas. No obstante, el conjunto subordinado de convenciones debe hacerse explícito para definir la ejecución de un algoritmo recursivo.

La técnica general de recursión no consiste en una división balanceada de la entrada; frecuentemente la división se realiza con partes de tamaño n-1 y 1 respectivamente. Los pasos a seguir para emplear esta técnica son:

1. Especificar el problema y dar un nombre al procedimiento el cual resolverá el problema.

2. Identificar el parámetro de la entrada a ser dividido

3. Proveer una solución al problema con entrada de tamaño 1 y una prueba para detectar que ocurre en tal situación.

4. Encontrar una relación entre soluciones para los problemas de tamaño pequeño y el problema original, y expresar esta relación algorítmicamente.

7.3.4 TECNICAS CON ALTO GRADO DE ELABORACION.

Las siguientes técnicas se consideran más sofisticadas que las vistas anteriormente, y por lo general conducen a algoritmos elegantes y naturales.

A. PROGRAMACION DINAMICA.

A menudo sucede que no hay manera de dividir un problema en un pequeño número de subproblemas cuya solución pueda combinarse para resolver el problema original. En tales casos, se puede intentar dividir el problema en tantos subproblemas como sea necesario, dividir cada subproblema en subproblemas más pequeños, y así sucesivamente; lo cual se hace quizá que se produzca un algoritmo de tiempo exponencial.

No obstante, con frecuencia, sólo hay un número polinomial de subproblemas, de aquí que se deba resolver algún subproblema

muchas veces. Si, en cambio, se conserva la solución a cada subproblema resuelto, y tan sólo se toma la respuesta cuando se requiere, se obtiene un algoritmo de tiempo polinomial. En este caso de una técnica tabular llamada programación dinámica muchas veces resulta un algoritmo más eficiente.

En esencia, la programación dinámica calcula la solución de todos los subproblemas. El cálculo procede de los pequeños subproblemas a los subproblemas más grandes, almacenando las respuestas en una tabla. La ventaja del método está basado en el hecho de quecuando se resuelve un subproblema , la respuesta se almacenada y nunca se recalcula.

La forma de un algoritmo de programación dinámica puede variar, pero hay un esquema común: una tabla para valores y un orden en el que se hacen las entradas.

Un problema frecuente usado para ilustrar esta técnica, es encontrar la ruta más barata de A a B (2 ciudades) con varios puntos de parada seleccionados de varios conjuntos de ciudades.

B. PREPROCESAMIENTO.

La técnica de preprocesamiento se aplica a problemas que ya se han resuelto para varias entradas. En particular, si una parte de los valores de entrada permanece igual sobre un número de corridas, tal vez nos permitamos preprocesar esta parte constante y obtener un algoritmo de menor complejidad.

C. ALGORITMOS EXHAUSTIVOS.

En cualquier etapa individual, un algoritmo exhaustivo es aquel que selecciona la lo que lo hace lo cual es localmente óptimo en algún sentido particular; es necesario subrayar el hecho de que no todos los enfoques exhaustivos llegan a dar el mejor resultado. Como sucede en la vida real, una estrategia exhaustiva puede dar un buen resultado durante un tiempo, pero el resultado general puede ser pobre.

Para algunos problemas, no existen algoritmos exhaustivos conocidos que produzcan soluciones óptimas. En otros casos, un algoritmo exhaustivo con frecuencia brinda la forma más rápida para llegar a una buena solución . De hecho, existen problemas en los cuales la única forma de alcanzar una solución óptima es mediante el uso de una técnica de búsqueda exhaustiva.

En seguida se describen brevemente algunas técnicas de búsqueda exhaustivas.

- BACKTRACKING (Método de Retroceso).

La técnica de diseño de algoritmos conocida como Backtracking puede ser descrita como una búsqueda exhaustiva organizada, que muchas veces evita la búsqueda de todas las posibilidades. Esta

técnica generalmente es apropiada para resolver problemas donde un número de soluciones potencialmente grande, pero finito, han sido inspeccionadas.

La técnica de Backtracking está relacionada con la técnica de inducción al menos con respecto a la naturaleza iterativa de la solución. La técnica llega a ser aplicable siempre que la solución a un problema requiera de encontrar uno o más elementos entre muchos "elementos" que satisfacen un número de condiciones y en los cuales el elemento mismo consiste de varios subelementos que pueden ser calculados separadamente. Si el elemento solución X tiene n componentes, entonces construimos los componentes X(1), X(2), ..., X(i-1) iterativamente. Esos componentes forman una solución parcial si cumplen las condiciones dadas del problema. La suposición básica de Backtracking es que existen muchas posibilidades entre las cuales se puede seleccionar X(i) pero no se debe violar ninguna condición. Este proceso se repite hasta determinar que no existe la manera de seleccionar X(i), por lo que no puede ser parte de la solución: en este caso, regresamos al paso anterior y seleccionamos otra posibilidad para X(i-1) (si ésta existe) y una vez más tratamos para X(i). Si no existe la posibilidad para X(i-1), y repetimos el paso de retroceso para X(i-2), etc.

Esta técnica de retroceso ha sido llamada a menudo **"Método de último recurso"**; sin embargo, es una herramienta importante que todo programador y todo diseñador de algoritmos debe saber como utilizar.

- RAMIFICACION Y ACOTAMIENTO.

La técnica conocida como Ramificación y Acotamiento es similar a la de Backtracking ya que busca un modelo de árbol del espacio de soluciones y es aplicable a una gran variedad de problemas combinatoriales discretos. Los algoritmos backtracking tratan de encontrar una o todas las soluciones representadas como **N-tuplas**, que satisfagan ciertas propiedades. Los algoritmos ramificar y acotar están orientados más hacia la optimización. El problema

a ser resuelto especifica una función costo valor-real para cada uno de los vértices (elementos de soluciones parciales) que aparecen en el árbol de búsqueda. Y la meta es encontrar una solución para la cual la función costo es maximizada o minimizada. El problema del agente viajero es una buena aplicación de esta técnica.

7.3.4 Técnicas heurísticas y de simulación.

Dos de las técnicas de diseño de algoritmos comúnmente usadas en la **"vida-real"** son: **la técnica heurística** y **la técnica de simulación.**

A. TECNICA HEURISTICA.

Un algoritmo heurístico, es un proceso que tiene las siguientes propiedades:

1. Usualmente encuentra buenas soluciones, aunque no necesariamente soluciones óptimas.

2. Es más rápido y fácil de implementar que un algoritmo exacto conocido (uno que garantice una solución óptima).

Una proposición general para el diseño de un algoritmo heurístico es listar todos los requerimientos de una solución exacta y dividirlos en dos clases, por ejemplo,

1. Aquellos que se deben satisfacer (obligatorios).

2. Aquellos que deseamos satisfacer (voluntarios).

El objetivo del diseño, es entonces construir un algoritmo que garantice los requerimientos en la clase 1 pero no necesariamente aquellos de la clase 2. Lo cuál no implica que no se haga ningún esfuerzo para satisfacer los requerimientos de la clase 2

B. TECNICA DE SIMULACION.

Los problemas simples son fáciles de establecer y modelar mediante la simulación, ya que no involucran muchos parametros, usualmente pueden ser resueltos analiticamente, y además, los algoritmos resultantes son razonablemente cortos.

La simulación en computadora es el proceso de conducción de experimentos de un modelo en computadora por medio de un sistema dinámico. El propósito inmediato de esos experimentos es observar el comportamiento de un sistema bajo un conjunto dado de suposiciones, condiciones, y valores de parámetros. El último propósito podría ser (1) formular políticas de manejo, (2) determinar óptimo o configuración factible del sistema, (3) establecer catálogos (programas) de producción realista, ó (4) decidir sobre estrategias económicas óptimas.

Las ventajas de simulaciones en computadora son numerosas:

- Se facilita el estudiar todas las partes de un sistema completamente integrado, tan detalladamente como se desee.

- Las partes individuales de un sistema pueden ser estudiadas analíticamente.

- Todas las variables pueden ser controladas y medidas, facilitando obtener información acerca de un sistema real cuando la experimentación directa en el sistema es impracticable.

- La simulacion hace posible probar un sistema despues de que se le haya invertido tiempo y dinero al sistema real correspondiente.

7.3.5 Técnica de análisis ascendente.

Por último, presentamos una técnica diferente de diseño de algoritmos, llamada **"Análisis Ascendente"**.

El análisis descendente es una forma de reflexión orientada a construir los algoritmos partiendo de un nivel muy general y detallando progresivamente cada tratamiento, hasta llegar al nivel de descripción exigido por la máquina (o el usuario).

Las técnicas de análisis descendente procuran ofrecer un método que permita abordar un problema, reconocer su estructura y, en consecuencia, los tratamientos que habrá de aplicar para resolverlo. Los elementos principales de esta técnica son:

- Utilización del esquema de tratamiento, que permita aplicar con seguridad algoritmos que responden a especificaciones precisas.

- El intento de reconocer en un problema ciertos rasgos que permitan reducirlo a un problema conocido. Ello exige, por lo tanto, que sea capaz de hacer abstracción de ciertos niveles de detalle, con el fin de juzgar globalmente sus características principales.

- Aplicación reiterada de los métodos empleados a cada nivel de descripción, de modo que conduzcan a un análisis, desde el nivel más alto hasta el nivel más bajo.

La construcción de un algoritmo por un método de análisis descendente consiste en definir máquinas abstractas cuyo funcionamiento es descrito por algoritmos adaptados a cada nivel de máquina, se parte de una máquina que permita tratar información de muy alto nivel. Una vez explicito y verificado el tratamiento, se toma cada una de las partes y se describen de nuevo en función de máquinas abstractas de nivel más bajo (esto es, que sus posibilidades de tratamiento son más débiles, o que tratan elementos de información menos complejos). Cuando una máquina abstracta coincide con la máquina real, el análisis ha concluido.

Ciertamente, las técnicas de diseño de algoritmos presentadas en este subcapitulo, no son las únicas herramientas válidas, pero se encuentran entre las más importantes, y por ende entre las más empleadas.

Tal vez el principio más importante para el diseñador de buenos algoritmos, es el no conformarse con lo realizado y continuar examinando el problema desde diferentes puntos de vista hasta estar convencido de que se tiene el algoritmo más apropiado para sus necesidades.

7.4 RECOMENDACIONES AL SELECCIONAR Y PROGRAMAR UN ALGORITMO.

La programación es tanto una ciencia como un arte; es ciencia porque se debe conocer la lógica y saber cómo y porqué funcionan los algoritmos, y es un arte porque se crea la entidad total que es el programa, teniendo presente que el arte frente a la ciencia es un conocimiento verificable, racional y práctico, a través de una técnica, el arte constituye un orden que busca la distracción y goce estético.

Un programa de computadora es un conjunto de instrucciones para dirigir la operación de un sistema computacional. Debido a que es un conjunto de instrucciones, engendra una semejanza con otros tipos de instrucciones empleadas en organizaciones, sin embargo, su diferencia más importante es la precisión y el grado de detalle requerido en un programa de computadora (Cuando se escribe un grupo de instrucciones para dirigir a una determinada persona, se asume que ésta tiene un cierto antecedente cognositivo, y por lo tanto no es necesario especificar los pasos con demasiado detalle).

Para poder obtener un buen rendimiento de las computadoras, es indispensable tener una programación eficiente y adecuada, que reduzca los tiempos de proceso y operación, para lo que es necesario conocer los métodos que resuelvan los principales problemas que se presentan y nunca olvidar que una computadora solamente puede resolver un problema, después de que se le ha indicado cómo hacerlo, los errores que se produzcan en el proceso, serán errores humanos, no de la máquina.

Hay un buen número de ideas que se deben tener presentes para lograr una programación eficiente y adecuada:

- PROBLEMA.

La solución del problema no surge en forma espontánea, ésta se complementa en cada detalle, desde la concepción del problema mismo.

- SELECCION DE UN ALGORITMO.

Dos o más algoritmos pueden ser correctos, pero puede existir entre ellos una diferencia tremenda en cuestiones de efectividad, por lo que un programador debe estar al tanto de las diferentes técnicas que existen para analizarlos; y poder escoger el algoritmo más eficiente; Así mismo, debe saber cuando aplicar estas técnicas y cuando ignorarlas.

- ALMACENAMIENTO.

Si se requiere escoger entre almacenamiento secuencial y ligado para una aplicación en particular, es necesario examinar las operaciones que van a ser desarrolladas en la estructura para poder realizar una decisión inteligente.

- **LENGUAJE.**

Si se tiene la opción de escoger un lenguaje de programación se recomienda aquél que manipule de manera más eficiente y transparente los tipos de datos inherentes al problema.

- **ENTRADAS.**

Las entradas de un algoritmo se encuentran restringidas por las condiciones del problema y la naturaleza de la computadora, se recomienda que parte del programa determine si la entrada es aceptable; además para determinados datos de entrada se puede requerir de una cantidad enormemente grande de pasos; para ello se puede prever una condición de paro.

- **SALIDAS.**

Dado que un algoritmo produce mínimamente una salida, desarrollar un programa sin ellas, puede significar una actividad sin sentido.

- **PLANIFICACION DEL DISEÑO DEL PROGRAMA.**

Para diseñar un buen programa se debe partir de un esbozo informal del algoritmo, con el diseño de un pseudoprograma y luego con su refinamiento gradual, hasta convertirlo en ejecutable. Esta estrategia de esbozar y después detallar, tiende a la producción de un programa más organizado.

- **ORGANIZACION DEL CODIGO.**

Se debe tratar de que el código asociado a cada operación importante y a cada tipo de datos quede colocado en un sólo lugar del programa, lo cual implica facilidad para localizar determinado código sujeto a revisión o modificación.

- **EMPLEO DE SUBRUTINAS.**

Cuando cierta tarea se desarrolla en diferentes partes de un programa, no es recomendable repetir el código; para solucionar esta situación se puede recurrir al empleo de subrutinas, en las cuales el código en cuestión se escribe una sola vez; y se invoca en la parte del programa que se requiera con sólo escribir el nombre de la subrutina.

Las subrutinas tienen varias ventajas, ya que con su empleo es más fácil visualizar la estructura de programas grandes y complejos; y facilitan también, la depuración de los programas. Fueron inventadas fundamentalmente para hacer un uso más eficiente de la memoria (ya que sólo existen cuando son llamadas por la rutina principal), sin embargo, hay que considerar el tiempo empleado por el procesador al crearla; una regla que casi siempre se cumple, es que al acortar un programa (empleando subrutinas), su tiempo de ejecución se agranda; por lo que hay

que tener presente que sólo tiene sentido emplear código directo, en lugar de llamadas a subrutinas cuando la velocidad del proceso tiene absoluta prioridad.

- USAR Y MODIFICAR UN PROGRAMA YA EXISTENTE.

Una de las principales causas de ineficiencia en el proceso de programación es que muchas veces un proyecto se aborda como si fuera el primer programa que se hubiera escrito. Se recomienda buscar primero un programa ya elaborado que realice la tarea completa o parte de ella; a la inversa, cuando se escribe un programa, se debe pensar en ponerlo a la disposición de otras personas.

- CONFECCIONAR HERRAMIENTAS.

Cuando se escribe un programa, debe pensarse en la posibilidad de escribirlo de un modo más general, sin mayor esfuerzo adicional; es decir, pensar en crear una herramienta (programa con gran variedad de usos).

- PROGRAMAS A NIVEL MANDATOS.

En ocasiones, es imposible encontrar en una biblioteca el programa que se requiere para realizar cierto trabajo, ni adoptar una herramienta para tal efecto, pero si se cuenta con un sistema operativo bien diseñado que permita conectar entre sí una red de programas disponibles sin que sea necesario escribir un programa nuevo, sino sólo una lista de mandatos (comandos) del sistema operativo, se debe emplear esta facilidad, teniendo presente que los mandatos serán pertinentes si se comportan como un filtro (archivo de entrada, archivo de salida). Siempre es posible componer un número arbitrario de filtros, y si el sistema operativo está diseñado de forma inteligente, una simple lista de mandatos (dispuestos como han de ejecutarse) será suficiente como programa.

La programación a nivel mandatos requiere disciplina por parte de los programadores, ya que deben escribir programas como filtros, y escribir herramientas en lugar de programas de propósito especial siempre que les sea posible. La recompensa de esto, es substancial y se encuentra en función de la razón global entre trabajo y resultados.

Ejemplo,

```
PROGRAMA SPELL
Spell : translate [A-Z] - [a-z],espacio → nueva_linea
        sort
        unique
        diff diccionario
```

Este programa fue escrito por S. C. Johnson a partir de mandatos de **UNIX**, toma como entrada un archivo a, que contiene un

texto en inglés, y produce como salida todas aquellas palabras en a, que no se encuentren en un pequeño diccionario. Spell tiende a listar los nombres propios y las palabras que no figuren en el diccionario como falta de ortografía; los comandos empleados son:

TRANSLATE : Reemplaza las letras mayúsculas por minúsculas y los espacios por saltos de línea.

SORT : Clasifica las líneas duplicadas y produce un archivo que contiene las palabras del archivo original, sin mayúsculas ni duplicaciones, en orden alfabético.

DIFF : Produce un archivo que contiene todas las palabras que no se encuentran en el diccionario.

- PROGRAMACION PORTATIL.

La capacidad de escribir programas que usen los recursos del sistema eficientemente, libres de errores y fácilmente transportables a otras computadoras, es la aspiración de cualquier programador profesional.

Es normal que un programa escrito en una máquina requiera de ser transportado a otra con un procesador distinto, u otro sistema operativo o ambas cosas, este proceso puede ser desde muy sencillo hasta extremadamente difícil; ello depende de cómo se haya escrito originalmente el programa.

Un programa no es fácil de transportar si contiene muchas dependencias con la máquina (fragmentos de código que sólo funcionan en un procesador o sistema operativo determinado), es por ello que cuando haya que usar llamadas al sistema operativo, éstas se realicen siempre mediante una función maestra de forma tal que sólo haya que cambiar la misma, para ajustarla a un nuevo sistema operativo dejando el resto del código intacto.

- DEPURACION.

Los buenos programadores normalmente son buenos depuradores. Aún cuando existen varios métodos de depuración, la prueba incremental se considera como el método de menor costo y de mayor efectividad en tiempo, incluso aunque parezca que retrasa el proceso de desarrollo.

La prueba incremental consiste en tener un código que funcione, a medida que se vaya anexando código al programa, hay que continuar probando las nuevas secciones, así como la relación que guardan con el código original. De este modo, se puede estar seguro que cualquier error que aparezca pertenece a la nueva zona del código.

- EMPLEAR RUTINAS INTERPRETATIVAS.

Una rutina interpretativa es un programa que desarrolla las instrucciones de otro programa que se encuentra escrito en un lenguaje de máquina. Este tipo de programas se recomienda cuando se desea velocidad en el proceso, representar una secuencia de acciones y decisiones de forma compacta y eficiente, o para comunicar las acciones de un programa de pasos múltiples.

- MANEJO DE RUTINAS DE SEÑAL.

En un programa siempre es recomendable el empleo de rutinas de señal, que nos indiquen el porqué de un determinado error en el proceso.

CONCLUSIONES Y RECOMENDACIONES

El haber desarrollado este tema, fue una tarea determinante en nuestra área de conocimiento, ya que nos permitio estudiar diversos aspectos teóricos que nos ayudaron a reafirmar conocimientos que son básicos en nuestra carrera. Pensamos que es un documento interesante para cualquier alumno de **Matemáticas Aplicadas y Computación** y en general para toda persona que tenga alguna relación con el área de la computación. Algunas de las recomendaciones que nos permitimos hacer son:

1. Se debe tener presente que el objetivo de la tesis es resaltar la importancia que tiene el **ANALISIS DE ALGORITMOS** para lograr una mayor eficiencia en los procesos que se realicen, y no el exponer los algoritmos más eficientes de un tema en particular, ya que esto en realidad depende de las necesidades y recursos con los que cuente el usuario.

2. Si el lector considera un tema muy ambigüo, debe recurrir a la referencia bibliográfica, dado que por ser temas muy amplios no nos fue posible tratarlos con más detalle, pero los consideramos en general temas muy interesantes cuyo conocimiento es importante.

3. Siempre que se analice un algoritmo se debe tener presente que su eficiencia esta dada en base a diferentes factores y es medible sólo en función a los beneficios que reporte al problema en cuestión y no a la eficiencia del algoritmo en comparación con otros métodos, lo cual puede implicar en un momento dado que el peor algoritmo para un problema determinado puede ser el que mejor resuelva el nuestro.

4. Un análisis del problema en cuestión repercute directamente en el diseño y selección correcta del algoritmo; por lo que debemos tener presente las diferentes técnicas que existen para solucionar un problema y que la solución de éste, no surgirá en forma espontánea sino será complementada con cada detalle que se tenga en cuenta.

5. No se debe olvidar nunca que la computadora realiza estrictamente lo que nosotros le pedimos que ejecute: resultados erróneos implican procedimientos mal formulados. Para poder obtener un buen rendimiento de ellos, es indispensable tener una programación eficiente, que se logra profundizando en la escritura y en la preparación de los programas, en otras palabras, se debe programar con "estilo" que es la combinación de los siguientes tres

principios: **CLARIDAD DE EXPRESION, CONSISTENCIA** y una buena **PLANEACION** que implique la facilidad para entenderlo y darle mantenimiento; es recomendable escribir en pocas líneas el nombre y el objetivo del programa así como el nombre del programador, y de la computadora para la cual fue hecho y otros comentarios que ayuden más tarde al usuario.

6. Por último y con respecto a los algoritmos que se tratan en este documento, su éxito de implementación no depende de la copia idéntica de ellos, sino de el saberlos adaptar al equipo y versión del lenguaje empleado así como de la comprensión adecuada de su proceso para poder evaluar los resultados que dichos algoritmos arrojen.

REFERENCIAS BIBLIOGRAFICAS

- **ALGORITHMS**
  Robert Sedgewick
  United States of America, 1984
  pp. 551

- **ALGORITHMS**
  Their Complexity and Efficiency
  Lydia I. Kronsjo
  Great Britain
  Copyright, 1979
  pp 361

- **ALGORITHMS AND AUTOMATIC COMPUTING MACHINES**
  (Topics in mathematics translated from de Russian)
  B. A. Trakhtenbrot, D.C. Heath and Company, Lexintong
  Massachusetts, 1960
  pp. 101

- **ALGORITHMS AND COMPLEXITY**
  Symposium on New Directions and Recent Results in Algorithms
  and Complexity
  Carnegie-Mellon, University
  Edited by J.F. Traub
  New York, 1979
  pp. 523

- **ALGORITHMS AND THEIR CONSTRUCTION**
  Computer Hardware
  Prepared by the course team the open University Press,1973
  pp. 304

- **ALGORITHMS + DATA STRUCTURES = PROGRAMS**
  Niklaus Wirth
  United States of America
  Prentice Hall, INC.
  pp. 365

- **ALGORITMOS Y ESTRUCTURAS DE DATOS**
  Niklaus Wirth
  Traducción y adaptación: Juan Carlos Vega Fagoaga
  México, 1988
  Prentice Hall Hispanoamericana, S.A.
  pp. 305

- **ALGORITHMIC GRAPH THEORY**
  Alan Gibbons
  Cambridge University Peress, 1985
  pp. 259

- **ALGORITHMIC LANGUAGE AND PROGRAM DEVELOPMENT**
  Friedrich L. Bauer
  Hans Wossner
  Germany
  Springer Verlag Berlin Heidelberg, 1982
  pp. 495

- **THE ART OF COMPUTER PROGRAMMING**
  (FUNDAMENTAL ALGORITHMS)
  Donald E. Knuth
  E.U.A.
  Vol I
  Stanford University
  Second Edition
  Copyright, 1973
  pp. 634

- **THE ART OF COMPUTER PROGRAMMING**
  (SEMINUMERICAL ALGORITHMS)
  Donald E. Knuth
  E.U.A.
  Vol II
  Stanford University
  Addison, Wesley Publishing Companing, 1969
  Second Edition, Noviembre 1971
  pp. 624

- **THE ART OF COMPUTER PROGRAMMING**
  (SORTING AND SEARCHING)
  Donald E. Knuth
  E.U.A.
  Vol III
  Stanford University
  Addison, Wesley Publishing Companing, 1969
  pp. 634

- **COMPUTER ALGORITHMS**
  Introduction to design and analysis
  Sara Baase
  Addison, Wesley Publishing Companing
  E.U.A., 1978

- **DATA STRUCTURES AND ALGORITHMS**
  Alfred V. Aho, John E. Hopcroft, Jeffrey D. Ullman
  E.U.A.
  Copyright, 1983
  pp. 427

- **DATA STRUCTURES AND PROGRAMMING TECHNIQUES**
  Herrmann A. Maurer
  Prentice Hall
  Englewood Cliffs, N.J. 1977
  pp. 228

- **THE DESING AND ANALYSIS OF COMPUTER ALGORITHMS**
  Alfred V. Aho, John E. Hopcropt, Jeffrey D. Ullman
  Addison Wesley Publishing Company
  United States of America, 1974
  pp. 470

- **FUNDAMENTALS OF THE COMPUTING SCIENCES**
  Kurt Maly
  Prentice Hall, Inc; 1987
  Englswood Cliffs, New Jersey
  pp. 488

- **INTRODUCTION TO THE DESIGN AND ANALYSIS OF ALGORITHMS**
  Goodman, Hedetniemi
  University of Virginia
  Mc Graw Hill Book Company
  Computer Science Series, 1977
  The United States of America
  pp 371

- **PROCESS ANALYSIS AND SIMULATION DETERMINISTIC SYSTEMS**
  David M. Himmelblay Kenneth B. Bischoff
  John Wiley & Sons, Inc.
  New York, 1968
  pp. 347

APENDICE A

LA ESTADISTICA Y GRAFICACION COMO HERRAMIENTAS AUXILIARES EN EL ESTUDIO DE ALGORITMOS.

La estadistica es una herramienta muy poderosa que en un momento dado nos ayuda a seleccionar con criterios más valiosos un algoritmo dado, por ejemplo si contamos con varios métodos o procesos que resuelven un problema dado ¿Por qué no realizar una serie de corridas de ellos con diferentes entradas y por medio de un análisis estadistico evaluar su comportamiento y, realizar un estimado de cual es el mejor algoritmo para las entradas dadas? y, ¡Claro! ¿Cuál es la probabilidad de que dichas entradas se presenten en el problema en cuestión?. Para un análisis más detallado podemos graficar estos comportamientos para tener bases más sólidas de argumentación al por qué nos inclinamos por un algoritmo dado, logrando con ello el auxilio de dos herramientas poderosas: la **ESTADISTICA** y la **GRAFICACION**.

El propósito de este apéndice no es más que realizar una reflexión respecto a la gran cantidad de recursos que existen y que nos pueden ayudar o facilitar la tarea de seleccionar un algoritmo, pero hay que tener presente también el costo adicional que esto puede implicar.

APENDICE B

DESCRIPCION DEL EQUIPO Y LENGUAJE EMPLEADO.

En general los programas o procedimientos pueden ser implementados en cualquier PC, que puede tener las siguientes características:

- Procesador 8088 o mayor.
- Monitor monocromático o color.
- Sistema operativo MS-DOS versión 3.30
- Mínimo de memoria RAM para ejecutarlos 3 K
- Compilador del lenguaje PASCAL.
- Compilador del lenguaje C.

También pueden ser implementados en distintos equipos que manejen dichos lenguajes realizando tan sólo las modificaciones requeridas (adaptando los programas al equipo que se va a emplear).

CARACTERISTICAS DEL LENGUAJE C.

El lenguaje C fue inventado por Dennis Ritchie quién lo implementó en un DEC PDP-11, utilizando el sistema operativo UNIX; C es el resultado de un proceso de desarrollo que comenzó con el lenguaje BCPL, que aún se emplea en Europa. El BCPL fue desarrollado por Martin Richard y tuvo una marcada influencia en el lenguaje B (inventado por Ken Thompson), lenguaje que llevó al desarrollo de C.

Algunas de las principales características de este lenguaje son:

- Posee siete tipos de datos incorporados, permite casi todas las conversiones de tipos y, los tipos carácter y entero pueden ser mezclados libremente en la mayoría de las expresiones.
- No realiza ningún tipo de chequeo de errores en tiempo de ejecución, por ejemplo checar los límites de un arreglo.
- El lenguaje C permite la manipulación directa de bits, bytes, palabras y apuntadores, esto lo hace muy apropiado para la

programación de sistemas en lo cual estas operaciones son muy comunes.

- Sólo posee 25 palabras clave que son las que conforman el lenguaje; por lo que se dice que es un lenguaje pequeño que puede aprenderse rápido.

- C tiene un poderoso conjunto de operadores, cuyo empleo correcto aumentan la expresión y eficiencia del lenguaje, pero si se emplean equivocadamente conducen a expresiones difíciles de leer y a resultados erróneos.

- Aunque inicialmente fue desarrollado para correr bajo el sistema operativo UNIX, el lenguaje C se ha hecho tan popular que los compiladores están disponibles para todas las computadoras y sistemas operativos, lo que lo hace un lenguaje portátil, haciendo posible poder utilizar el mismo programa en cualquier equipo con sólo unas pequeñas modificaciones.

- C es un lenguaje de programador que impone muy pocas restricciones en lo que se pueda hacer con él. Si el programador emplea este lenguaje puede evitar el uso de código ensamblador en la mayoría de las situaciones en que lo requiera. De hecho uno de los motivos para su diseño fue el proporcionar una alternativa a la programación en el lenguaje ensamblador.

- Inicialmente el lenguaje C se utilizaba para la programación de sistemas (Un programa de sistemas es parte de los programas que conforman el sistema operativo de una computadora o de sus utilerías de soporte) por ejemplo: sistemas operativos, intérpretes, editores, compiladores, administradores de Bases de Datos; pero, cuando el lenguaje C creció en popularidad muchos programadores lo empezaron a utilizar para programar todas sus tareas debido a su transportabilidad y eficiencia, lo que repercute en un ahorro de tiempo y dinero.

- El componente estructural de C es la función; en C las funciones son bloques constitutivos en los que se desarrolla toda la actividad de los programas, esto hace de C un lenguaje modular.

- Posee la velocidad del ensamblador.

Algunas de las desventajas de C son:

C al igual que todos los lenguajes presenta ciertas desventajas, algunas de ellas son:

- C no contiene sentencias (operaciones) para trabajar directamente con objetos compuestos tales como cadenas, conjuntos, listas de arreglos o vectores.

- No cuenta con operaciones de ENTRADA/SALIDA, ni existen métodos propios para el acceso a archivos. Todos estos mecanismos de alto nivel deben ser aportados por funciones llamadas explícitamente.

- Sólo ofrece proposiciones (sentencias) de control de flujo sencillas, secuenciales, de selección, de iteración; pero no multiprogramación, paralelismo, sincronización o corrutinas.

- No es un lenguaje fuertemente estructurado, lo cual permite al usuario realizar pensando el programa, llevando esto muchas veces a un mal hábito al programar

CARACTERISTICAS DEL LENGUAJE PASCAL

PASCAL es un lenguaje desarrollado en 1971, fue diseñado por el profesor Niklaus Wirth; es un lenguaje sistemático y lógicamente consistente. Sus objetivos principales son:

Ser eficiente al implementarlo y correrlo.

Permitir desarrollar programas organizados y estructurados.

Servir como vehículo para la enseñanza de los más importantes conceptos de programación.

PASCAL presenta varias características que lo distinguen de sus predecesores ya que no sólo ofrece un fuerte repertorio de estructuras de control y un rico conjunto de mecanismos para ciclos de control y ejecuciones condicionadas, sino también facilita la escritura de programas al ofrecer un amplio conjunto de tipos de datos definidos por el lenguaje y predefinidos por el programador, así mismo es un lenguaje que da un enfoque ordenado y disciplinado de la programación, lo que conduce a programas claros.

Este lenguaje no sólo ha sido empleado en forma extensiva en la comunidad académica, donde es muy común ver los algoritmos de ensayos técnicos escritos en lenguaje PASCAL, sino también se emplea para experimentos en lenguajes de extensión y para sistemas de aplicación general tal como aplicaciones comerciales implementadas en microcomputadoras.

Al ser PASCAL un lenguaje de propósito general no está restringido a desarrollar una sola actividad, además permite realizar programas generales ya que maneja parámetros en lugar de valores fijos, lo que implica el manejo de la misma función y procedimiento con distintos resultados.

Algunas de las desventajas de PASCAL son:

- No inicializa variables y también no permite crear constantes de tipo estructurado haciendo difícil escribir ciertos tipos de programas eficientes.

- Permite declarar variables locales y globales con el mismo nombre lo cual muchas veces se presta a confusiones.

- El lenguaje **PASCAL** no permite la compilación por separado es decir, los procedimientos y funciones no pueden ser compilados en forma independiente del programa principal.

- Existen varios factores a considerar cuando se decide emplear **PASCAL** para una aplicación particular. El primero, es verificar si el lenguaje soporta esa aplicación y si ésta requiere soporte específico tal como la aritmética decimal o el manejo de Base de Datos

COMPARACION ENTRE PASCAL Y C

PASCAL y **C** tienen muchas semejanzas, especialmente en sus estructuras de control y en el uso de subrutinas independientes con variables locales. Esto hace posible hacer muchas traducciones instrucción por instrucción; a menudo se puede simplemente sustituir las funciones o palabras claves por su equivalente en **C**.

Aún cuando **PASCAL** y **C** son similares, existen tres diferencias principales entre ellos:

1.- **PASCAL** es más restrictivo y en cierto sentido más limitado que **C**; por ejemplo, el **PASCAL** estándar no sólo hace difícil el escribir programas de sistemas (ya que no puede cargar directamente las direcciones de memoria), sino que además no realiza conversión de tipos.

2.- **PASCAL** está formalmente estructurado en bloques mientras que **C** no. El término estructurado se refiere a la capacidad de un lenguaje de crear unidades de código lógicamente interconectadas que se pueden referenciar juntas; el término también significa que los procedimientos pueden tener procedimientos conocidos sólo por el procedimiento que los engloba. A **C** normalmente se le define como lenguaje estructurado porque permite fácilmente crear bloques de código pero no permite definir funciones dentro de otras.

3.- En **PASCAL** todas las variables, funciones y procedimientos deben ser declaradas antes de emplearlas; lo cual significa que no se permiten referencias anticipadas sin la sentencia **FORWARD**. En **C** todas las variables deben ser declaradas antes de ser usadas; pero no hay restricciones en las referencias a las funciones.

CUADRO COMPARATIVO ENTRE PASCAL Y C (RESUMEN):

LENGUAJE C

VENTAJAS	DESVENTAJAS
- Código portable. - Manejo de bloques estructurados - Acceso directo al hardware. - Funciones únicas ya definidas - Manipulación eficiente de tipos., - Lenguaje portable. - Código compacto. - Manipulación directa de bits. - Velocidad del ensamblador - Permite la conversión de tipos.	- No es fácil de entender. - Sólo ofrece proposiciones de control de flujo sencillas, secuenciales, de iteración, bloques y subprogramas; pero no multiprogramación, paralelismo,sincronización o corrutinas. - Los mecanismos de alto nivel (operaciones I/O y acceso a archivos) deben ser aportados por funciones llamadas en forma explícita. - No contiene operaciones (sentencias) para trabajar directamente con objetos compuestos (conjuntos, arreglos, listas) considerados como un todo.

LENGUAJE PASCAL

VENTAJAS	DESVENTAJAS
- Código más entendible. - No es un lenguaje muy simbólico - Permite anidamiento de funciones. - Permite declarar distintos tipos. - Es una excelente herramienta de enseñanza y sus programas son fáciles de entender.	- Código fuente extenso. - Requiere más recursos. - No tiene acceso directo al sistema operativo. - No inicializa variables. - Poco portable. - No permite compilación por separado. - No realiza conversión de tipos.