UNIVERSIDAD NACIONAL AUTONOMA DE MEXICO

FACULTAD DE CIENCIAS

ADAPTABILIDAD EN EL PROBLEMA DE ORDENAMIENTO

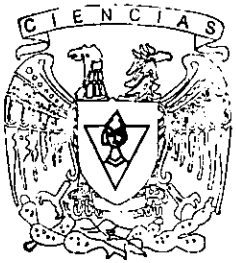
T E S I S

Que para obtener el título de

M A T E M A T I C O

p r e s e n t a

EFRAIN URIOSTEGUI ATANACIO



FACULTAD DE CIENCIAS
UNAM

Director de Tesis:
M. en I. María de la Luz Gasca Soto

1999
FACULTAD DE CIENCIAS
SECCION ESCOLAR

VNIVERSIDAD NACIONAL
AVFNOMA DE
MEXICO

MAT. MARGARITA ELVIRA CHÁVEZ CANO
Jefa de la División de Estudios Profesionales de la
Facultad de Ciencias
Presente

Comunicamos a usted que hemos revisado el trabajo de Tesis:

Adaptabilidad en el Problema de Ordenamiento

realizado por Efraín Urióstegui Atanacio

con número de cuenta 8638989-9 , pasante de la carrera de Matemáticas

Dicho trabajo cuenta con nuestro voto aprobatorio.

Atentamente

Director de Tesis Propietario	M. en I. María de Luz Gasca Soto
Propietario	M. en C. Elisa Viso Gurovich
Propietario	Dra. Amparo López Gaona
Suplente	M. en C. José de Jesús Galaviz Casas
Suplente	M. en C. María Guadalupe Elena Ibargüengoitia González

Consejo Departamental de Matemáticas
Mat. César Guevara Bravo

AGRADECIMIENTOS Y DEDICATORIA

"En el principio creó Dios los cielos y la Tierra."

Génesis 1:1

Doy Gracias a Dios por ayudarme y guiarme continuamente en mi vida espiritual y profesional y por ayudarme a concluir este trabajo a quien presento y dedico esta tesis.

A Dios

"Y dijo Jehová Dios: No es bueno que el hombre esté solo; le haré ayuda idónea para él."

Génesis 2:18

Doy gracias a mi esposa por estar a mi lado siempre y animarme y alentarme por su oración en la realización de esta tesis a quien se la dedico con amor en Cristo.

Con cariño a mi ayuda idónea Eugenia

"Porque de tal manera amó Dios al mundo, que ha dado a su Hijo unigénito, para que todo aquel que en él cree, no se pierda, mas tenga vida eterna."

Juan 3:16

Dedico esta tesis

A mis padres

A mis hermanos

Y en memoria de mi abuelo.

A todos ellos gracias por su apoyo, sacrificio y ánimo que me impulsaron para la realización de esta tesis.

"Es, pues, la fe la certeza de lo
que se espera, la convicción
de lo que no se ve."

Hebreos 11:1

Especial agradecimiento a mi Directora de Tesis por su apoyo en todo momento, gracias por su paciencia, dedicación e impulso para hacer posible este trabajo.

M. en I. María de Luz Gasca Soto

"Jesús le dijo: Yo soy el camino, y
la verdad, y la vida; nadie viene
al Padre, sino por mí."

Juan 14:6

Gracias a los colaboradores y hermanos en Cristo de la Universidad Bíblica Fraternal (UBF) por su oración, a quienes dedico esta tesis.

Gracias a todos aquellos que de una manera directa o indirectamente me ayudaron y apoyaron para poder concluir esta tesis, tanto de mi trabajo(SESIC), amigos y familiares a quienes dedico esta tesis.

ADAPTABILIDAD EN EL PROBLEMA DE ORDENAMIENTO

ÍNDICE

Página

Introducción

El ordenamiento es el proceso computacional que consiste en reorganizar una secuencia dada de elementos en orden ascendente o descendente [7]. En un modelo de cómputo basado en comparaciones, se llega a obtener, para algoritmos de ordenamiento, una complejidad de $\Omega(n\log n)$, en el peor de los casos. Cuando el algoritmo de ordenamiento toma ventaja del orden existente en la entrada, el desempeño computacional del algoritmo puede verse como una función creciente con respecto al tamaño de la secuencia y al desorden en la secuencia. Si un algoritmo usa el hecho de que los datos de entrada están casi ordenados, para obtener un mejor desempeño computacional, decimos que tal algoritmo es adaptivo[1] [7].

A medida que existe desorden en la secuencia de datos, el desempeño computacional del algoritmo adaptivo va incrementándose y, en el peor de los casos, resulta ser similar al desempeño computacional de los algoritmos clásicos.

Los algoritmos de ordenamiento adaptivos son atractivos porque secuencias casi ordenadas son comunes en la práctica [7]. El análisis y diseño de algoritmos adaptivos nos da la posibilidad de mejorar algoritmos que no consideran el orden existente en la entrada.

El objetivo de este trabajo es introducir los conceptos formales sobre adaptabilidad de algoritmos desarrollando el análisis de adaptabilidad en el problema de ordenamiento.

En el Capítulo 1 presentamos el Análisis de Algoritmos, que trata la complejidad computacional y medidas de complejidad; en el Capítulo 2 ofrecemos una introducción a la adaptabilidad de algoritmos; en el Capítulo 3 describimos los algoritmos de ordenamiento clásicos y su desempeño computacional; en el Capítulo 4 presentamos las Medidas de Desorden que se usan en la adaptabilidad de algoritmos; en el Capítulo 5 desarrollamos el análisis de la adaptabilidad para algoritmos de ordenamiento y sus variantes adaptivas, en el Capítulo 6 presentaremos algunas evidencias empíricas de nuestros resultados y finalizamos presentando las conclusiones sobre este trabajo.

[1] Usaremos el término Algoritmo adaptivo como traducción literal de *Adaptive Algorithm*

Capítulo 1. Análisis de Algoritmos

INTRODUCIÓN

Los algoritmos son la parte medular de las Ciencias de la Computación. Muchos de los trabajos iniciales en este campo estaban dirigidos hacia la identificación de tipos y clases de problemas que podían ser resueltos algorítmicamente.

El análisis de un algoritmo lo podemos dividir en tres pasos.

- **Primero**, debemos determinar si nuestra solución parece factible con respecto a requerimientos de cómputo como capacidad de memoria, facilidad de uso, etcétera.
- **Segundo**, debemos revisar y validar la descripción de pseudo-código del algoritmo, esto es, demostrar formalmente que el algoritmo es correcto.
- **Tercero**, requerimos desarrollar un análisis de la complejidad del algoritmo. La complejidad en este sentido no se refiere a la dificultad de entender el programa; más bien, es una medida sobre la cantidad de trabajo hecho por la función que se ejecuta.

Este tipo de análisis es especialmente útil cuando hay dos o más soluciones disponibles y deseamos seleccionar una, la mejor, para su implementación de acuerdo a las necesidades existentes.

1.1 COMPLEJIDAD COMPUTACIONAL.

En general, en la literatura de computación, no se ha discutido mucho del esfuerzo computacional involucrado en la ejecución de algoritmos que trabajan con secuencias. Este esfuerzo puede ser medido en términos del número de operaciones que una máquina tiene que realizar para construir la solución conforme a un algoritmo dado. Esto es usualmente llamado ***desempeño computacional*** del algoritmo.

Para discutir el desempeño computacional de un algoritmo es necesario establecer qué constituye un paso computacional u operación básica. Esto puede ser definido solamente en términos de una máquina (abstracta o real) la cual es asumida para realizar el cálculo; en otras palabras, requerimos un *modelo computacional.*

La máquina de Turing es usualmente usada como modelo para estudiar el desempeño computacional de los algoritmos. Todos los algoritmos deben ser formulados conforme al modelo. Esto es usualmente muy tedioso, puesto que todos los datos deben ser representados como cadenas de ceros y unos. Una mejor opción, que trabajar todo a ese nivel bajo, es usar un modelo el cual permita esta discusión a un nivel más fácil de entender, involucrando operaciones de un tipo de datos abstracto.

Esto puede ser justificado en la práctica porque una operación particular o paso computacional puede usualmente ser calculado en un número bien definido de operaciones de nivel bajo en una máquina de Turing o bien en una máquina real con memoria de acceso aleatorio y una unidad central de procesamiento. De cualquier manera, es importante tener cuidado de que el tamaño del problema no afecte el desempeño de las operaciones elementales que se asumen para constituir los pasos computacionales básicos.

1.2 MEDIDAS DE COMPLEJIDAD

Frecuentemente deseamos evaluar la eficiencia de programas, esto es queremos descubrir el tiempo de cómputo y espacio de almacenamiento que está siendo usado para ejecutar un algoritmo. Este problema puede ser dividido en dos partes:

- **Primero**.- Evaluar la eficiencia del algoritmo; y
- **Segundo**.- Evaluar la calidad de la implementación.

Para evaluar la calidad de un algoritmo, necesitamos una medida de eficiencia que nos permita compararlo con otros. En la búsqueda para tal medida, confrontamos dos problemas, los cuales

explicaremos en el contexto de estimar el tiempo eficiente para un algoritmo de búsqueda. Los problemas que vamos a encontrar son:

- **Dificultad**, sin embargo no es imposible asociar un valor de tiempo actual con el periodo de ejecución de un algoritmo; y

- **Relación** entre el tiempo de ejecución y el tamaño del problema. El tiempo necesario para completar una tarea usando un algoritmo particular, parece depender (usualmente) del tamaño del problema, esto es, de algún valor derivado del número o tamaño de los datos de entrada.

Para ver como el tiempo necesario para un cálculo depende del tamaño de la entrada, considérese un programa que busca un valor en una tabla de números. Supóngase que estos números no están guardados en algún orden particular. Así, un programa de búsqueda tiene que recorrer la tabla observando en cada entrada hasta que un elemento sea igual al buscado o hasta que el fin de la tabla sea alcanzado. Si el número buscado está en el comienzo de la tabla entonces podremos encontrarlo rápidamente o tardaremos más si está al final. Cada búsqueda de un número que no está en la tabla causaría recorrer todos los n elementos. En cualquier evento, conocemos que en el peor de los casos tenemos que observar a n elementos. Así, el tiempo necesario para buscar a un elemento depende de n, el tiempo para encontrar un elemento crece en proporción con el número n de elementos. Podemos decir que, para valores grandes de n, el tiempo necesario para nuestro programa de búsqueda, es proporcional a n.

El *desempeño computacional* es una propiedad intrínseca a un algoritmo. Esta no depende de un método particular de implementación, ni de la velocidad de la computadora en la cual está ejecutándose el programa. Por lo tanto, es una excelente medida de la eficiencia del algoritmo. Para nuestro ejemplo el desempeño computacional es asintóticamente proporcional a n o simplemente "el desempeño es orden n", denotado por $O(n)$.

El término "Asintótico" expresa la noción que la aserción es aproximadamente verdad y llega a ser más exacta cuando n crece.

El propósito de análisis de algoritmos es predecir el comportamiento, especialmente el tiempo de ejecución, de un algoritmo sin implementarlo en una computadora específica. Las ventajas de hacerlo así son claras. Es mucho más conveniente tener medidas simples para la eficiencia de un algoritmo que implementar el algoritmo y probar la eficiencia cada vez que un cierto parámetro en el sistema de la computadora cambia. Además un programa complicado usualmente incluye muchos algoritmos "pequeños". Sería bastante trabajo probar completamente todas las alternativas diferentes para cada parte del programa [6].

Desafortunadamente, es imposible predecir el comportamiento exacto de un algoritmo. Hay muchos factores que influyen. En lugar de esto, tratamos de extraer las características principales del algoritmo. Definimos ciertos parámetros y ciertas medidas que son las más importantes para el análisis. Muchos detalles concernientes a la implementación exacta se ignoran. El análisis es así solamente una *aproximación*. Por otro lado, aún una áspera aproximación puede producir información significante acerca del algoritmo. Lo más importante es que, usando este análisis, podemos comparar diferentes algoritmos para determinar el mejor de acuerdo a nuestros propósitos.

Describiremos una metodología para pronosticar la aproximación de tiempo de ejecución de algoritmos y para comparar diferentes algoritmos. La principal característica de este acercamiento es que ignoramos factores constantes y nos concentramos en el comportamiento del algoritmo como el tamaño de la entrada que puede ir creciendo.

Ejemplo 1.- Si la entrada es un arreglo de tamaño n, y si el algoritmo consiste de $100n$ pasos, entonces ignoramos la constante 100 y decimos que el tiempo de ejecución es aproximadamente n, o es de orden n.

Ejemplo 2.-Si el número de pasos es $2n^2+50$, entonces ignoramos las constantes 2 y 50 y decimos que el tiempo de ejecución es aproximadamente n^2, o es de orden cuadrático.

Dado que n^2 es más grande que n, decimos que el segundo algoritmo es más lento; aún sin embargo para $n=5$, por ejemplo, el primer algoritmo requiere 500 pasos, mientras que el segundo

requiere solamente 100 pasos. Esta aproximación es válida, si n es bastante grande. El segundo algoritmo es verdaderamente más lento que el primero para todo $n \geq 50$.

Por otro lado, supóngase que el tiempo de ejecución del primer algoritmo es $100n^{1.8}$. Otra vez, el primer algoritmo parece mejor, donde $n^{1.8}$ es más pequeño que n^2. En este caso, sin embargo, n tendrá que ser aproximadamente $300,000,000$ para que $100n^{1.8}$ sea más pequeño que $2n^2+50$. Afortunadamente, la mayoría de los algoritmos tienen constantes pequeñas en la expresión de sus tiempos de ejecución. Sin embargo, el acercamiento asintótico puede ser extraviado algunas veces, trabaja bien en la práctica. En la mayoría de los casos, observando solamente el comportamiento asintótico es suficiente como una primera aproximación e indicación de eficiencia.

El número de posibilidades de entradas es enorme, y la mayoría de los algoritmos se comporta diferente para diversas entradas. En lugar de eso, definimos una medida para la entrada, llamada el *tamaño* de la entrada, y el análisis es relativo a ese tamaño. El tamaño es definido usualmente como una medida de la cantidad de espacio requerido para guardar la entrada. El tamaño será denotado por n.

Dado un problema y una definición de tamaño, queremos encontrar una expresión que proporcione el tiempo de ejecución del algoritmo relativo al tamaño.

El análisis asintótico y el análisis del peor caso son solamente aproximaciones del tiempo de ejecución de un algoritmo en particular bajo una entrada particular.

1.3 LA NOTACIÓN *O*

En esta sección definiremos la notación O según Manber[6].

Como ya hemos dicho, se ignoran los factores constantes cuando se trata de evaluar el tiempo de ejecución de un algoritmo particular. Para hacer eso eficazmente necesitamos una notación especial.

Definición. Se dice que la función $g(n)$ es $O(f(n))$ para alguna función $f(n)$ si existen un par de constantes c y N, tales que, para toda $n \geq N$, tenemos $g(n) \leq cf(n)$. $O(f(n))$ se lee como "O de $f(n)$" y algunas veces como "O - grande de $f(n)$".

Ejemplo. $5n^2 + 15 = O(n^2)$, donde $5n^2 + 15 \leq 6n^2$ para $n \geq 4$. Al mismo tiempo, $5n^2 + 15 = O(n^3)$, donde $5n^2 + 15 \leq n^3$ para todo $n \geq 6$.

La notación O nos permite ignorar constantes convenientemente. Aunque podemos incluir constantes dentro de la notación O, no hay razón para hacerlo. Siempre escribimos $O(n)$ en lugar de decir $O(5n+4)$. Similarmente escribimos $O(\log n)$ sin especificar la base del logaritmo, porque cambiando las bases cambia el algoritmo solamente por una constante. Escribimos $O(1)$ para denotar una constante. Podemos también usar la notación O si queremos especificar las constantes solamente en partes de la expresión. Por ejemplo, podemos escribir $T(n) = 3n^2 + O(n)$ o $S(n) = 2n \log_2 n + 5n + O(1)$.

En general, determinar si una cierta función $g(n)$ es $O(f(n))$ puede no ser fácil. Con algunas reglas simples, podremos cubrir la mayoría de los casos. La regla más útil es la siguiente:

Definición. Se dice que una función $f(n)$ es monótonamente creciente si $n_1 \geq n_2$ implica que $f(n_1) \geq f(n_2)$.

Teorema 1

Para cualesquiera constantes c y a, con $c > 0$ y $a > 1$, y para toda función $f(n)$ monótonamente creciente se tiene :

$$(f(n))^c = O(a^{f(n)})$$

es decir, una función exponencial crece más rápido que una función polinomial [6].

Esta regla puede ser usada para comparar diferentes funciones. Por ejemplo, si $f(n)=n$, $\forall n$ en el Teorema 1, obtenemos que:

$$n^c = O(a^n),\ \forall\ c>0 \text{ y } a>1,$$

Otro ejemplo se obtiene substituyendo la función $f(n)$ por $\log_a n$:

$(\log_a n)^c = O(a^{\log_a n}) = O(n),\ \forall\ c>0$ y $a>1$, a y c constantes.

Podemos sumar y multiplicar con la notación O usando las siguientes reglas.

Lema 1

1. Si $f(n) = O(s(n))$ y $g(n) = O(r(n))$ entonces $f(n) + g(n) = O(s(n) + r(n))$

2. Si $f(n) = O(s(n))$ y $g(n) = O(r(n))$ entonces $f(n)\cdot g(n) = O(\ s(n)\cdot r(n)\ )$

Prueba:

Por definición, existen constantes c_1 , N_1 , c_2 , y N_2 , tales que

$f(n) \leq c_1 s(n)$ para $n \geq N_1$, y

$g(n) \leq c_2 r(n)$ para $n \geq N_2$.

La más grande de c_1 y c_2 y la más grande de N_1 y N_2 puede ser usada para ambos resultados [6].

Para la notación O corresponde la relación "$\leq$", sin embargo no es posible restar o dividir. Esto es, no es verdad en general que:

$f(n)=O(s(n))$ y $g(n)=O(r(n))$ implique que $f(n)-g(n) = O(s(n)-r(n))$ o que $f(n)/g(n)=O(s(n)/r(n))$.

La importancia de concentrarnos en el comportamiento asintótico es ilustrado en la Tabla 1.1. La cual contiene varios tiempos de ejecución típicos y el tiempo correspondiente que consume el algoritmo para un problema de tamaño $n=800$ para diferentes velocidades de computadora. Las velocidades difieren por una constante de *2* de columna a columna, de *1000* pasos por segundo a *8000* pasos por segundo. Podemos ver el mejoramiento que logramos por velocidades altas de la computadora (o el algoritmo) por un factor constante contra el mejoramiento que logramos al cambiar a un algoritmo asintótico más rápido, ver la parte superior de la Tabla 1.1. Un algoritmo exponencial requerirá tiempo astronómico (billones y billones de años) para manejar $n = 1000$, a menos que la base sea casi 1.

tiempo de ejecución	$tiempo_1$ 1000 pasos/seg	$tiempo_2$ 2000 pasos/seg	$tiempo_3$ 4000 pasos/seg	$tiempo_4$ 8000 pasos/seg
$\log_2 n$	0.010	0.005	0.003	0.001
n	1	0.5	0.25	0.125
$n\log_2 n$	10	5	2.5	1.25
$n^{1.5}$	32	16	8	4
n^2	1,000	500	250	125
n^3	1,000,000	5000,000	250,000	125,000
1.1^n	10^{39}	10^{39}	10^{38}	10^{38}

Tabla 1.1 *Tiempos de ejecución, en segundos* [6].

La notación *O* es usada para denotar límites superiores en el tiempo de ejecución de algoritmos, sin embargo, usar solamente límites superiores no es suficiente. Algunos algoritmos tienen tiempo de ejecución de $O(2^n)$. Esto es, no requieren más que tiempo exponencial. Sin embargo, $O(2^n)$ es un límite superior malo para la mayoría de estos algoritmos, ellos son mucho más rápidos que esta cota. Estamos interesados no solamente en los límites superiores sino en una expresión que esté cerca del tiempo de ejecución actual como sea posible. En casos donde es difícil encontrar la expresión exacta, nos gustaría encontrar al menos la menor cota inferior para esto. Obtener cotas bajas es más difícil que obtener límites superiores. Una cota superior, en el

tiempo de ejecución de un algoritmo implica solamente que existe algún algoritmo que no usa más tiempo que el indicado. Un límite inferior, para el peor de los casos, puede implicar que ningún algoritmo puede lograr una mejor cota para el problema, en el peor de los casos. Es imposible, por supuesto, considerar todos los algoritmos posibles uno por uno. Necesitamos mecanismos para modelar problemas y algoritmos en una forma para capacitarnos para comprobar cotas inferiores. Hay una notación similar para manejar cotas inferiores ignorando constantes.

Deficinión 1.-Si existen constantes c y N, tales que para toda $n \geq N$ el número de pasos $T(n)$ requiere para resolver el problema para una entrada de tamaño n es al menos $c \cdot g(n)$, entonces decimos que $T(n)=\Omega(g(n))$.

Ejemplo.- Sea $n^2=\Omega(n^2-100)$, y también $n=\Omega(n^{0.9})$.

La notación Ω corresponde a la relación "≥" y representa una cota inferior.

Definición 2.-Si una cierta función $f(n)$ satisface que $f(n) = O(g(n))$ y $f(n) = \Omega(g(n))$, entonces decimos que $f(n) = \Theta(g(n))$. Representa una categoría de orden para la función.

Ejemplo.- $5n \cdot log_2 n - 10 = \Theta(n \cdot log n)$.

La base del logaritmo puede ser omitida en la expresión $\Theta(n \cdot log n)$, de bases diferentes cambia el algoritmo solamente por un factor constante. Las constantes usadas para O y Ω no necesitan ser las mismas.

Las notaciones, O, Ω y Θ corresponden a las relaciones "≤" una cota superior, "≥" una cota inferior, y una equivalencia "=".

Algunas veces necesitamos notación correspondiente a "<" y ">".

Definición 3.-Decimos que $f(n) = o(g(n))$, léase "$f(n)$ es o-pequeña de $g(n)$" si

$$lim_{n \to \infty} \frac{f(n)}{g(n)} = 0$$

Ejemplo.- $n/log_2 n = o(n)$, pero $n/10 \neq o(n)$.

Definición 4.- Similarmente, decimos que $f(n) = \omega\ (g(n))$ si $g(n) = o(f(n))$.

En la Tabla 1.2 se engloban todas las definiciones anteriores y su tipo de relación.

Se puede reforzar el Teorema 1 reemplazando la O grande con la o pequeña y se tiene:

Teorema 2

Para toda constante $c>0$ y $a>1$, y para toda función $f(n)$ monótonamente creciente:
$(f(n))^c = o(a^{f(n)})$

En otras palabras, una función exponencial crece más rápido que una función polinomial [6].

Símbolo	O	Ω	Θ	o	ω
Relación	$\leq$	$\geq$	=	<	>

Tabla 1.2 *Notaciones y su tipo de relación*

Capítulo 2. Adaptabilidad de Algoritmos

INTRODUCCIÓN

En este capítulo presentamos las nociones básicas y conceptos de ordenamiento adaptivo. La motivación inicial para los algoritmos de ordenamiento adaptivos es la alta frecuencia con la cual se presentan entradas casi ordenadas en aplicaciones prácticas. Ordenar secuencias casi ordenadas debería requerir menos trabajo que ordenar una secuencia aleatoriamente permutada.

Diremos informalmente que un algoritmo es adaptivo si resulta ser capaz de tomar ventaja de la forma como están organizados o representados los datos de entrada, para mejorar su desempeño computacional. Esto significa que el algoritmo es capaz de aprovechar las características del ejemplar[1] del problema a solucionar.

Estivill-Castro[9] nos indica que un algoritmo es adaptivo si para cada ejemplar dado, la realización de operaciones elementales resulta ser una función no decreciente que depende del tamaño y dificultad del problema.

Un ejemplar de un problema son los valores específicos que toman los parámetros de un problema. De manera informal diremos que un ejemplar son los datos de entrada para un problema.

Más formalmente, un algoritmo es adaptivo si la información sobre el desempeño computacional de un algoritmo A, aplicado a un ejemplar E, no sólo se expresa como $T_A(/E/)$ sino como una función $T_A(/E/,dif(E))$ donde $dif(E)$ es una función que mide la dificultad del ejemplar E, y $/E/$representa el tamaño del ejemplar [7].

La adaptabilidad es una propiedad que poseen algunos algoritmos. Hay algoritmos adaptivos en diferentes áreas, a continuación damos algunos ejemplos.

2.1 ADAPTABILIDAD EN GEOMETRIA COMPUTACIONAL

La triangulación de polígonos es un problema clásico de la Geometría Computacional y consiste en triangular un polígono simple. La Figura 2.1 ilustra algunos polígonos y sus triangulaciones.

Este problema tiene diversas aplicaciones prácticas, la más conocida es la distribución de guardias en una galería de arte.

Si un polígono es convexo, o casi convexo (Figura 2.1 b) resulta muy fácil triangularlo, pero si contiene una gran cantidad de sinuosidades (Figura 2.1 a o c) resulta más difícil obtener una triangulación.

En Geometría computacional, existen algoritmos adaptivos para realizar la triangulación de polígonos. Chazell e Incerpi[10], diseñaron un algoritmo que requiere tiempo $O(n \log s)$ donde s "mide" la sinuosidad del polígono. En este campo, se tiene otro algoritmo adaptivo creado por Hertel y Mehlhorn [11], el cual requiere tiempo $O(n + r \log r)$, donde r es el número de vértices cuyas aristas forman una región convexa. G. Towssain diseñó un algoritmo, para triangulación de polígonos, cuyo desempeño computacional es $O(n \cdot \lceil 1+f \rceil)$ donde f representa la complejidad de la triangulación derivada[9].

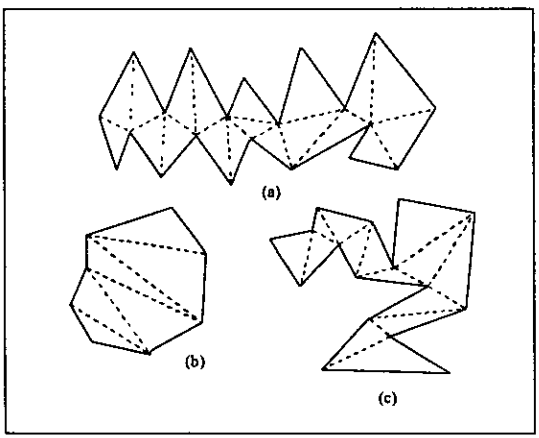


Figura 2.1 *Triangulación de polígonos.*

[1] En este trabajo usamos el término "ejemplar de un problema" como traducción literal de *instance of a problem.*

2.2 ADAPTABILIDAD EN OPTIMIZACION COMBINATORIA

En el campo de la optimización combinatoria se tiene un algoritmo clásico para resolver un problema de n variables con m restricciones: el método simplex. Un análisis del peor de los casos, nos indica que el método Simplex requeriría de una revisión exhaustiva sobre los puntos extremos de la región convexa, esto es: $\binom{n}{m} = \frac{n!}{m!(m-n)!}$ pruebas individuales del problema.

Sin embargo el Método Simplex, en la práctica, no revisa cada punto extremo, resulta ser más eficiente que en el peor de los casos y para una gran cantidad de ejemplares su desempeño es aceptable por lo tanto este es un algoritmo adaptivo.

La Figura 2.2 nos ilustra una región convexa, cada vértice en ella representa una posible solución, f es la función a optimizar.

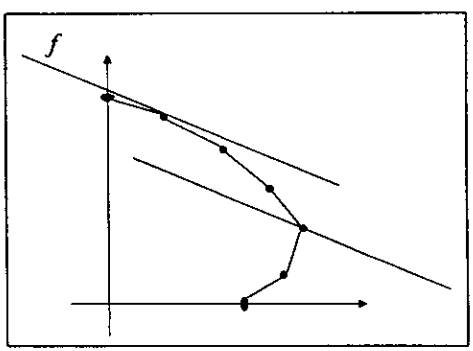


Figura 2.2 *Región convexa.*

2.3 ADAPTABILIDAD EN TEORIA DE REDES

- **Problema de los árboles generadores de peso mínimo.**

Un árbol generador o de expansión de una gráfica G, es aquel que contiene todos los vértices de G. Dada una gráfica G con pesos en las aristas un árbol generador de peso mínimo de G es un árbol generador de peso mínimo de G cuya suma de pesos en sus aristas es menor o igual que la

de cualquier otro árbol generador de G. La Figura 2.3 ilustra un ejemplo de árbol generador de peso mínimo para la gráfica G.

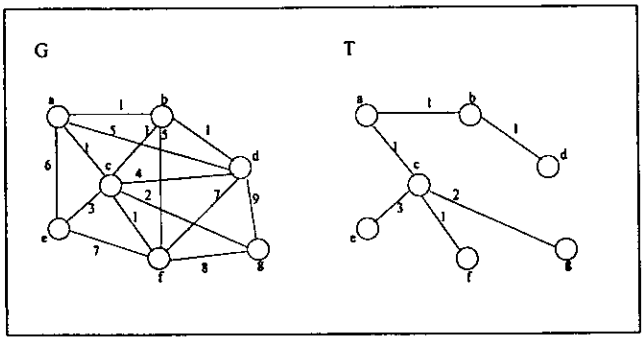


Figura 2.3 *T es un árbol generador de peso mínimo para G.*

Para este problema se tienen 2 algoritmos clásicos: algoritmo de Prim y el algoritmo de Kruskal. El algoritmo de Kruskal ordena las aristas y va tomando aristas de mínimo peso y las incluye en el árbol generador mientras no formen ciclos. El algoritmo de Prim, a partir de un vértice va tomando las aristas adyacentes de mínimo peso. La Figura 2.4 y la Figura 2.5 muestra la idea general de ambos algoritmos aplicados a la gráfica G, ahí presentada.

En términos prácticos si al algoritmo de Kruskal le damos como entrada una gráfica que ya sea un árbol va a gastar tiempo ordenando las aristas para después reconstruir el árbol. En cambio el algoritmo de Prim al tomar un vértice y sus aristas irá construyendo el árbol sin tener que reorganizar todas las aristas. Por lo tanto podemos concluir que el algoritmo de Prim resulta ser adaptivo y el algoritmo de Kruskal no. Sin embargo Estivill y Gasca [12] obtuvieron versiones adaptivas del algoritmo de Kruskal para el problema de árboles generadores de peso mínimo, apoyándose en el uso de algoritmos de ordenamiento adaptivos y mejorando la implantación de estructuras de datos.

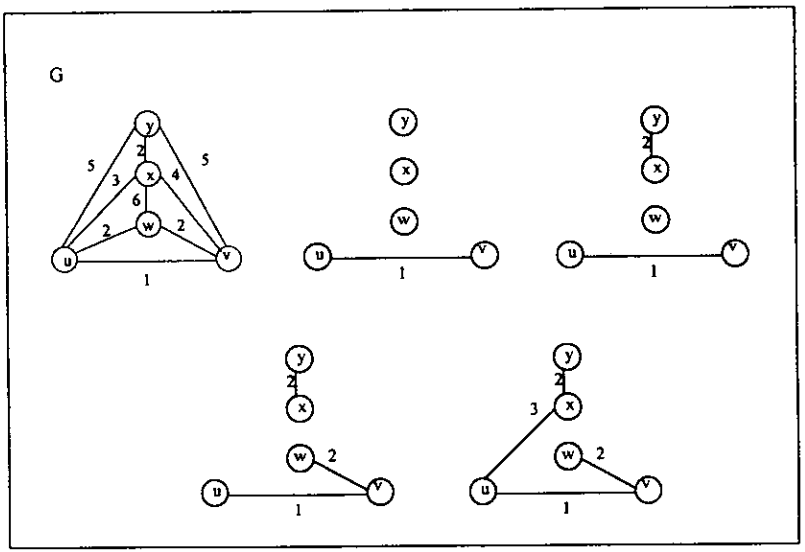


Figura 2.4 *Construcción del árbol de peso Mínimo usando Kruskal.*

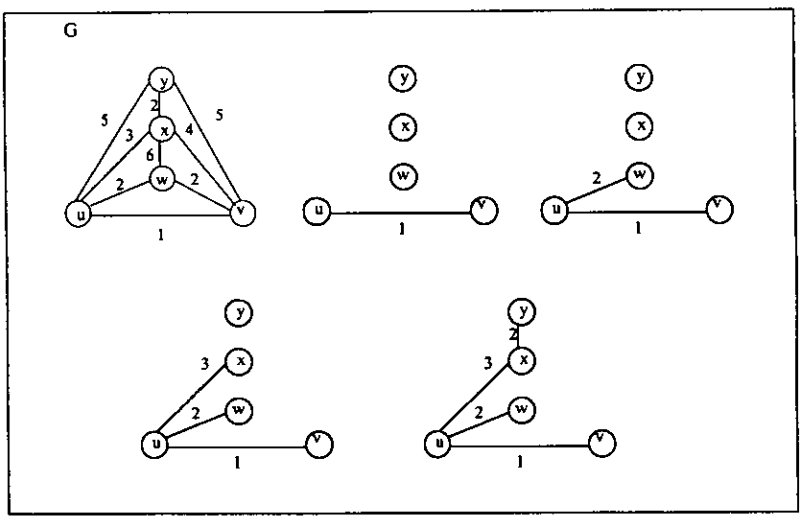


Figura 2.5 *Construcción del árbol de peso Mínimo usando Prim(u).*

- **Adaptabilidad en el problema de la Ruta Más Corta (RMC)**

Dada una Gráfica con costos en los arcos, y con dos vértices distinguidos *s* y *t*, llamados vértice fuente y destino, respectivamente, el problema de RMC consiste en determinar la ruta de menor costo entre los vértices *s* y *t*.

El problema de Adaptabilidad en RMC fue tratado por Gasca [13] donde logró obtener versiones adaptivas del algoritmo de Dijkstra para este problema, basándose en estructuras de datos sofisticadas y el algoritmo de ordenamiento de inserción local, el cual es adaptivo. Se observó que la adaptabilidad del algoritmo de Dijkstra depende de los ciclos que tenga la gráfica, el grado de los vértices, de la manera en que le llegan los costos de las aristas al algoritmo, así como de la forma como se manipulan los costos de las aristas en el algoritmo.

2.4 ADAPTABILIDAD EN EL PROBLEMA DE ORDENAMIENTO

Nuestro enfoque en este trabajo está en atención sobre algoritmos de ordenamiento basados en comparaciones para modelos secuenciales de computación.

Un ejemplo clásico de un algoritmo de ordenamiento adaptivo es *Straigth Insertion Sort,* ver Figura 2.6. En el algoritmo *Straigh_Insertion_Sort* se examina toda la entrada una vez, de izquierda a derecha, repetidamente encontrando la posición correcta del actual elemento, insertándolo en un arreglo de elementos ordenados anteriormente.

```
Procedure Straight_Insertion_Sort(X,n);
  X[0] = - ;
  for j:=2 to n do
  begin   i:= j-1;
          t:=X[j];
          while t<X[i] do
            begin X[i+1] :=X[i];
            i:=i-1;
          end
          X[i+1]:=t;
  end
```

Figura 2.6 *Algoritmo Adaptivo de Ordenamiento*

Definimos ***Inv(X)*** como el número que denota la cantidad de inversiones en una secuencia $X=<x_1,x_2,...,x_n>$, donde *(i,j)* es una inversión si $i<j$ y $x_i > x_j$. Intuitivamente, *Inv(X)* mide el desorden, puede alcanzarse el valor mínimo cuando X está ordenado y su valor depende solamente del orden relativo de los elementos en X.

El desempeño computacional del algoritmo *Straight_Insertion_Sort* se puede describir en términos del tamaño de la entrada y el número de inversiones en la entrada. Para una secuencia X con n elementos, *Straight_Insertion_Sort* hace exactamente $Inv(X)+n-1$ comparaciones y $Inv(X)+2n-1$ movimientos de datos y usa espacio adicional constante. Decimos que *Straight_Insertion_Sort* es un algoritmo adaptivo con respecto a la medida *INV*. Empíricamente las evidencias confirman que *Straight_Insertion_Sort* es eficiente para secuencias pequeñas y secuencias casi ordenadas [7]. En las Figuras 2.7 a 2.9 se muestran algunos comportamientos del algoritmo *Straight_Insertion_Sort* para diferentes secuencias.

```
Lista ordenada
1,2,3,4,5,6,7,8,9,10

Iteraciones de Straight_Insertion_Sort:
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
1,2,3,4,5,6,7,8,9,10
Total de operaciones elementales:27
```

Figura 2.7 *Lista ordenada*

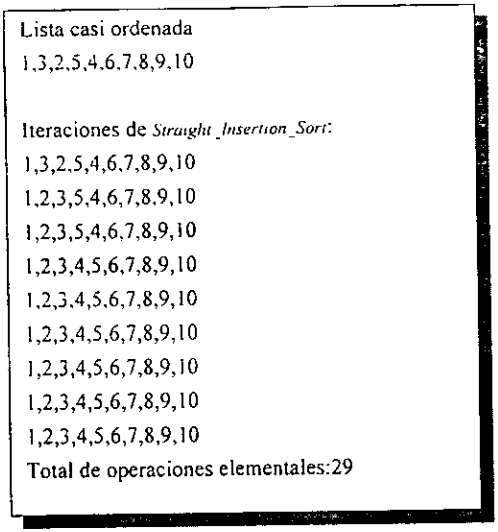


Figura 2.8 *Lista casi ordenada*

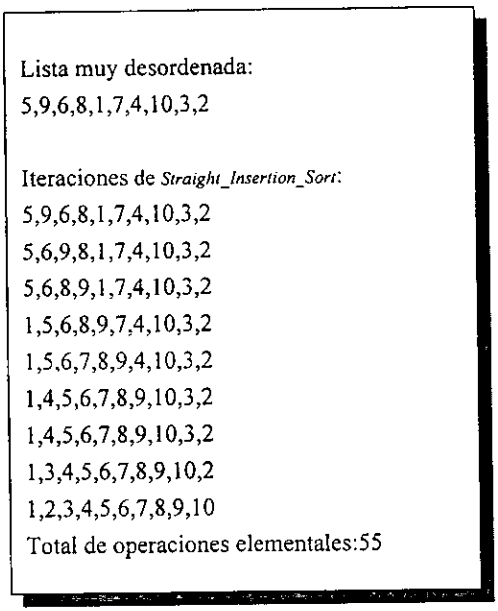


Figura 2.9 *Lista muy desordenada*

Capítulo 3. El Problema de Ordenamiento

INTRODUCCIÓN

El ordenamiento es el proceso por medio del cual organizamos datos basado en algún criterio de orden. Los criterios de ordenamiento pueden ser los clásicos como: alfabéticos y numéricos, y algunos no tan obvios, como controladores de disco de prioridad I/O que están basados en la proximidad de los bloques de datos con respecto a la posición actual de la cabeza de lectura/escritura del *dispositivo de entrada.*

Debemos considerar lo siguiente cuando discutimos algoritmos de ordenamiento:

Tiempo de ejecución. Determina una complejidad del algoritmo y lo compara con el desempeño de otro algoritmo de ordenamiento. Además, determina si la ejecución del algoritmo es afectada por la composición, el orden relativo, de sus datos. Por ejemplo, algunas rutinas de ordenamiento son eficientes cuando los datos están ordenados o casi ordenados.

Requerimiento de espacio. Para ordenar el algoritmo se evalúa si se requiere almacenamiento adicional. Es deseable un algoritmo eficiente que no requiera espacio adicional.

3.1 DEFINICIÓN DE ORDENAMIENTO

Para saber que una secuencia está correctamente ordenada en forma ascendente, se necesita una definición precisa.

Definición: Una (sub)secuencia finita $E[m],E[m+1],...,E[n]$ esta ordenada si para todo índice $i,j \in N$, $m<i<j<n$ implica $E[i]<E[j]$.

De esta definición se sigue directamente que una secuencia con solamente un elemento está trivialmente ordenada, y ningún elemento es más pequeño que el primero. También si quitamos el primer elemento de la secuencia ordenada, el resto de la tabla queda ordenada, a menos que sea vacía.

3.2 ALGORITMOS DE ORDENAMIENTO

Para solucionar el problema de ordenamiento, describiremos varios métodos de ordenamiento y determinaremos su desempeño computacional. Los algoritmos de ordenamiento que describimos en esta sección son conocidos como algoritmos clásicos de ordenamiento. Se describe su desempeño computacional en el peor caso, mejor caso y caso promedio.

3.2.1 BUBLESORT: Método de la Burbuja.

Uno de los métodos más directos de ordenamiento es un ordenamiento burbuja, *Bubble Sort*. La idea general del método es:

- Pasar a través en un arreglo de elementos desordenados, comparando celdas adyacentes.
- Si están fuera de orden, intercambiarlos.
- Cuando se complete una revisión sin intercambiar ningún elemento, los datos están ordenados y el proceso termina.

La Figura 3.1 ilustra un ordenamiento ascendente burbuja haciendo varias iteraciones sobre el conjunto de datos.

Estado inicial:	5 4 1 3 2
1ª iteración :	[5 4] 1 3 2
	4 [5 1] 3 2
	4 1 [5 3] 2
	4 1 3 [5 2]
	4 1 3 2 5
Después de la primera iteración:	4 1 3 2 5
Después del segunda iteración:	1 3 2 4 5
Después de la tercera iteración:	1 2 3 4 5

Figura 3.1 *Ejecución de Ordenamiento Burbuja*

La función comienza por comparar el valor en la posición 1 con *la posición 2*, luego *la posición 2* con *la posición 3* y así sucesivamente. Después de completar el primer paso, el elemento más grande queda en su posición correcta, al terminar el segundo paso, el segundo elemento más

grande está en su posición final. Este proceso continúa de esta manera hasta que todos los elementos han sido movidos a su posición correcta.

Este algoritmo compara elementos adyacentes, si se encuentran en desorden los intercambia y de esta manera los ordena.

Una versión de esta técnica de ordenamiento aparece en el Listado 1.

```
void bbl_sort(int data[], int int no_elems)
{  int top, flag, tmp, i;
 top = no_elems;
 do {
  flag = 0;
  top--;
  for( i=0; i<top; i++){                    /* recorre todo el arreglo y              */
   if (data[i] > data[i+1]) {               /* compara elementos adyacentes           */
     tmp = data[i];                         /* si el i-ésimo es mayor que el          */
     data[i] = data[i+1];                   /* i-esimo+1 los intercambia              */
     data[i+1] = tmp;
     flag++;                                /* como hubo intercambio incrementa flag  */
   } /* fin de if */
  } /* fin de for */
 } while (flag >0);                         /* flag=0 indica que no hizo intercambios */
}                                           /* fin de bbl_sort()                      */
```

Listado 1. *Algoritmo BubleSort()*

La función *bbl_sort()* requiere dos argumentos: el arreglo a ordenar y su tamaño. El ciclo exterior *do* controla la ejecución. Eso es, la función se repetirá hasta que el ciclo interior haga una iteración sin intercambiar ningún elemento. Esto es indicado por el valor asignado en la variable *flag*. El ciclo interior hace el mayor trabajo; pasa a través de cada celda del arreglo, intercambiando los elementos adyacentes como sea requerido.

Análisis

El ciclo interior *for* se ejecuta *n* veces, una vez por cada elemento de el arreglo. En el peor de los casos, el ciclo exterior *do* iterará también una vez por cada elemento. Esto produce una complejidad de $O(n^2)$. En el caso promedio el comportamiento de *bbl_sort()* es predecir en base

de la posición de sus datos de entrada. Por ejemplo, si los datos están ordenados, entonces solo una pasada es requerida. Sin embargo este cambio del comportamiento del caso promedio de este algoritmo es solamente un poco mejor que el comportamiento del peor caso y aún produce una complejidad de $O(n^2)$.

3.2.2 SELECTION SORT: Ordenamiento por selección.

Otro método simple de ordenamiento es el ordenamiento por selección (*selection_sort*). La idea de esta técnica es:

- Buscar en el arreglo de datos al elemento más pequeño.
- Intercambiar la posición de este elemento con la del elemento en la localidad 1.
- Localizar el segundo elemento más pequeño y cambiar su posición con la del elemento en la localidad 2.
- Continuar de esta manera, buscando cada elemento sucesivo, hasta que todo el arreglo quede ordenado.

El algoritmo deriva su nombre del hecho de que selecciona el mínimo elemento y lo posiciona en su lugar y esto a cada paso a través de el arreglo. La Figura 3.2 ilustra varias iteraciones del algoritmo en un conjunto de datos como muestra.

```
Estado inicial: 4 2 5 3 1
1ra. iteración: 1 2 5 3 4
2a. iteración:  1 2 5 3 4
3a. iteración:  1 2 3 5 4
4a. iteración:  1 2 3 4 5
```

Figura 3.2 *Ejecución del algoritmo Selection Sort*

Durante el primer paso, la función identifica el elemento 1 como el más pequeño e intercambia su posición con la del elemento 4. No hay cambios ocurridos durante el segundo paso porque el elemento 2 estaba ya en su posición correcta.

Nótese que, por virtud del diseño, *selection_sort* puede mover el mismo elemento varias veces. Esto es muy claro en las iteraciones 1 y 4, donde la función reubica el elemento 4 durante ambas iteraciones. Sin embargo, el algoritmo realizará, a lo más, un intercambio durante cada paso.

La idea general: Obtenemos el elemento más pequeño dentro de la posición 1, entonces el segundo más pequeño dentro de la posición 2, y así sucesivamente. En otras palabras para cada valor de i entre 1 y n -1, construimos A[1..i] ordenado y menor o igual a A[i+1..N].

El Listado 2 contiene el código para la función sel_sort(). Sus dos argumentos indican el arreglo de datos y su tamaño.

```
void sel_sort(int data[], int no_elems)
{   int i,j,min,tmp;
for (i=0; i<no_elems; i++){                  /* Recorre todo el arreglo de datos        */
   min=i;                                    /* asume i como indice del minimo          */
   for( j=i+1; j<no_elems; j++)              /* compara con los i+1 en adelante         */
     if (data[j] < data[min])                /* si hay uno menor que el de indice min   */
       min=j;                                /* entonces este es el nuevo indice min    */
   tmp = data[i];                            /* Deja en tmp el valor actual             */
   data[i]=data[min];                        /* En la localidad actual pone el minimo   */
   data[min]=tmp;                            /* El valor que era actual lo pone en min  */
 }                                           /* Fin del ciclo for exterior              */
}                                            /* fin de sel_sort()                       */
```

Listado 2. *Selection Sort*

Durante cada iteración del ciclo exterior, el ciclo interior localiza el elemento más pequeño y guarda su índice en la variable *min*. El cambio real ocurre cuando termina el ciclo interior. El ciclo exterior recorre todo el arreglo colocando los elementos de menor orden en forma ordenada. Cuando termina el ciclo exterior, los elementos quedan ordenados.

Análisis

El ciclo exterior itera n veces; con cada iteración del ciclo exterior el ciclo interior desarrolla una comparación por cada elemento desordenado. Esto produce una complejidad de $O(n^2)$. Debido a su diseño, el comportamiento de la función queda constante, independiente de la composición de su conjunto de datos. Así, la complejidad del caso promedio es también $O(n^2)$.

El tiempo para selection sort está determinado por el número de iteraciones del ciclo interior:

$$(n-1) + (n-2) + (n-3) + \ldots + 1 = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2}$$

el cual es $O(n^2)$.

Así que, el desempeño computacional para el mejor caso, peor caso o caso promedio es siempre $O(n^2)$. Esto significa que el desempeño computacional del algoritmo *selection_sort* es $\Theta(n^2)$.

3.2.3 INSERTION SORT: Ordenamiento por Inserción.

Otro método de ordenamiento es por inserción, *insertion sort*. Este método de ordenamiento puede ser comparado a la forma de una mano de juego de cartas. Para iniciar, la primera carta es colocada en la mano. Entonces, como cada carta sucesiva es recibida, esta es insertada en la mano en orden. El jugador hace espacio para cada nueva carta cambiando las cartas de más alto valor a la derecha. Ver Figura 3.3 para un ejemplo. El elemento en la localidad 1 del arreglo servirá como la primera carta. Los nuevos elementos son distribuidos examinando el arreglo de las localidades *2* a *n*. Entonces determinamos en dónde pertenece el nuevo elemento y lo insertamos en la mano; estos es, la porción de bajo orden del arreglo.

```
Estado inicial:  4 2 3 1 5
1ra. iteración:  2 4 3 1 5
2a. iteración:   2 3 4 1 5
3a. iteración:   1 2 3 4 5
```

Figura 3.3 *Ejemplo de Insertion Sort*

Uno de los algoritmos más simples de ordenamiento es el algoritmo *insertion_sort* el cual ejecuta a lo más *n -1* iteraciones. Inicia del paso *i = 2* a *n*, El algoritmo *insertion_sort* asegura que los elementos en las posiciones *1* a *i* están en forma ordenada. El algoritmo *insertion_sort* utiliza el hecho de que los elementos en las posiciones *1* a *i-1* están en forma ordenada.
El Listado 3 contiene el código para la función ins_sort().

Su ciclo exterior, el cual selecciona los elementos para inserción, recorre de *1* a *no_elemns-1*. Nótese que se comienza por dar a *i* el valor *1*; así el elemento en la localidad *0* sirve para la primera comparación como el elemento actual. La inserción toma lugar en el ciclo interior. Esta sección de código examina la porción ya ordenada del arreglo. Es decir los índices de orden menor en orden inverso, cambiando los elementos a la derecha como sea requerido. Determina la posición correcta del nuevo elemento y le hace espacio. Cuando el ciclo interior termina, la

función guarda el nuevo elemento en la localidad desocupada. Nótese que al igual que el algoritmo *sel_sort()*, los elementos de bajo orden están ordenados, cuando termina el ciclo *for* el arreglo queda ordenado.

```
void ins_sort(int data[], int no_elems)
{  int i,j,tmp;
for ( i=1;i<no_elems; i++){          /* Inicia con la segunda posición ya que compara hacia la izquierda.   */
  tmp = data[i];                     /* Es el elemento actual                                               */
  j=i-1;                             /* Inicia comparando desde el indice i-1 posición hacia la izquierda   */
  while ( (data[j] > tmp) && (j>=0) ){ /* El elemento de posición j es mayor que el elemento actual tmp.     */
    data[j+1] = data[j];             /* Entonces se recorre una posición hacia la derecha                   */
    j--;                             /* Se decrementa j para comparar con el elemento mas a la izquierda    */
  }                                  /* fin de ciclo while                                                  */
  data[j+1] = tmp;                   /* El ciclo while termina cuando el elemento j ya no es mayor que tmp, */
                                     /* entonces tem lo coloca en la posición j+1 y queda ordenado          */
}                                    /* fin de ciclo for                                                    */
}                                    /* fin de ins_sort()                                                   */
```

Listado 3. *Insertion Sort*

Análisis

Para cada valor de *i* entre *2* y *n*, iniciamos con el subarreglo *A[1...i-1]* ya ordenado, y colocamos *A[i]* es su posición correcta, así dejamos *A[1..i]* ordenado. La elección es llevada a cabo de derecha a izquierda iniciando en la posición i.

El ciclo *for* será ejecutado *n-1* veces. Si la lista inicial estuviera en orden creciente, el ciclo *while* no sería ejecutado, así el número total de iteraciones será n-1. Por lo que en el mejor caso su desempeño computacional es $O(n)$.

Si la lista inicial estuviese en orden decreciente, el número de iteraciones del ciclo while será: $1+2+3+4...+n-2+n-1$. Así, en el peor caso, el desempeño computacional es $O(n^2)$.

En el caso promedio será aproximadamente $n^2/4$ iteraciones del ciclo while, por lo tanto, para este caso, el tiempo es $O(n^2)$.

Por lo tanto el algoritmo es $\Theta(n^2)$.

3.2.4 QUICKSORT

Iniciaremos la presentación de técnicas de ordenamiento avanzadas con uno de los más populares algoritmos de ordenamiento llamado *quicksort*, también llamado *partition sort*. Quicksort fue creado en 1960 por C.A.R. Hoare y ha sido estudiado y analizado desde entonces.

Iniciamos con una breve descripción del algoritmo. Asumimos que n es el tamaño de nuestro arreglo de datos a ordenar; al cual llamaremos *data[]*.

- Seleccionar un elemento x del arreglo. Nos referiremos a este elemento como el elemento de partición (*pivote*). Inicialmente, la elección del elemento partición será arbitraria.
- Determinar la posición final de x en el arreglo ordenado. Por ahora se asume que esta es alguna localidad *data[i]*.
- Reorganizar todos los otros elementos del arreglo de tal forma que todos los elementos en las localidades *data[0]* hasta *data[i-1]* sean menores o iguales a x, y todos los elementos en las posiciones *data[i+1]* a *data[n]* sean mayores o iguales a x.
- Recursivamente aplicar el algoritmo en los dos subarreglos *data[0,...,i-1]* y *data[i+1,..,n]* hasta que todos los elementos queden ordenados.

La Figura 3.4 ilustra de manera intuitiva, con un ejemplo, el comportamiento general del algoritmo. Considere ahora la Figura 3.5, durante el paso inicial, la función arbitrariamente selecciona el elemento en la localidad *0* del arreglo (valor *4*) como el elemento de partición. Cuando el primer paso se completa, este elemento está en su posición correcta y la función puede proceder con la llamada recursiva en los dos subarreglos.

Si se considera el problema en cada iteración. Resulta claro que la mayor dificultad de la tarea es determinar la posición final del elemento de partición. Finalmente, no necesitamos ordenar el arreglo para determinar la posición final de x. Todo lo que necesitamos saber es el número de los otros elementos que sean posicionados en cualquiera de dos lugares, a la izquierda de x o a la derecha de x en el arreglo, se llegará entonces a un simple cálculo para determinar la localidad correcta de x.

A continuación se describe de manera general el algoritmo:

- Seleccionar un elemento de partición x.
- Contar el número de elementos menores que x.

- Mover x en su posición final.
- Recursivamente procesar los subarreglos en cualquiera de los dos lados.

Considere la siguiente situación: Seleccione dos índices, digamos i y j. Simultáneamente, mueva el índice i a través del arreglo de izquierda a derecha, *es decir de 0 a n*, y mueva el índice j a través del arreglo de derecha a izquierda (*es decir de n a 0*). Cuando i encuentra una condición donde $data[i] > x$ y j encuentra una condición donde $data[j] < x$, intercambie elementos. La función continúa de esta manera hasta que los índices se cruzan, *es decir cuando* $j \leq i$. Esto asegura que todos los elementos son particionados correctamente. Así, cuando x es finalmente ubicado, todos los elementos menores a x serán reubicados a la izquierda de x en el arreglo, y todos los elementos mayores a x serán reubicados a la derecha de x en el arreglo. La Figura 3.6 muestra un ejemplo de ejecución del algoritmo *QuickSort*.

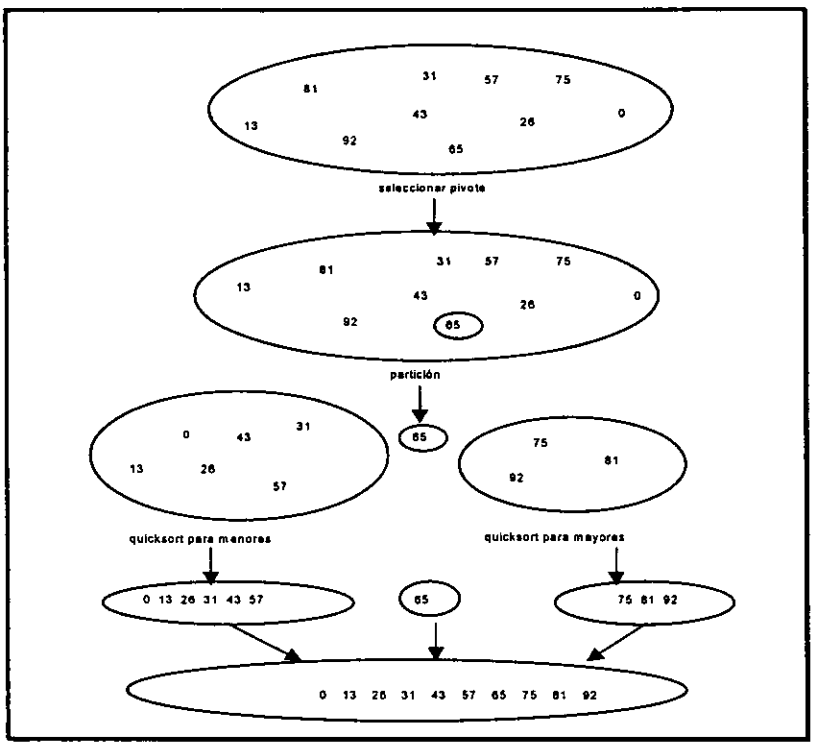


Figura 3.4 Pasos de quicksort ilustrados por un ejemplo

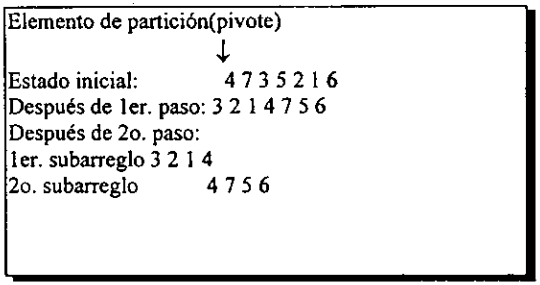


Figura 3.4 *Quicksort*

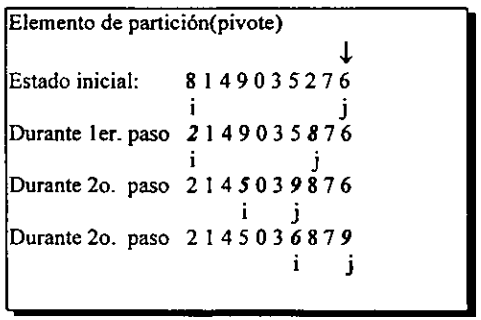


Figura 3.6 *Ejemplo de ejecución del QuickSort*

Aplicando recursivamente el algoritmo en todos los subarreglos el arreglo quedará ordenado.

Como su nombre lo indica *quicksort* es uno de los algoritmo de ordenamiento más rápidos. La descripción de este análisis está basada en[8]. El Listado 4 especifica una versión de Quicksort.

La función *qck_sort()* requiere 3 argumentos: *data[],* arreglo de datos a ordenar; *lo,* límite inferior; y *hi,* límite superior.

```
void qck_sort(int
 data[],                          /* arreglo de datos a ordenar    */
   lo,                            /*  limite inferior              */
   hi )                           /*  limite superior              */
{  int i,j,tmp,part_elem;
 if (hi  > lo ) {                  /* Para terminar la llamada recursiva                                  */
   part_elem = data[hi];           /* Toma el Elemento de Partición o PIVOTE, elemento mas a la derecha   */
   i = lo -1;                      /* i y j son índices para recorrer el arreglo                          */
   j = hi;
   while (1) {                     /* En este ciclo de while() se particiona el arreglo                   */
                                   /* de datos de acuerdo al elemento particion part_elem.                */
     while (data[++i] < part_elem); /* Se desplaza i desde el limite inferior hacia la derecha del arreglo */
                                   /* mientras se cumpla que el elemento del del arreglo sea menor que el pivote */
     while (data[--j]> part_elem); /* Se desplaza j desde el limite superior hacia la izquierda del arreglo */
                                   /* mientras se cumpla que el elemento del del arreglo sea mayor que el pivote. */
      if (i >= j)                  /* Condición para terminar el ciclo while()                            */
        break;
 tmp = data[i];                   /* El elemento mayor que el PIVOTE y que se va a intercambiar se guarda. */
 data[i]=data[j];                 /* En la posición i se coloca el elemento que fue  menor que el PIVOTE   */
 data[j] = tmp;                   /* En la posición j se coloca el elemento que fue mayor que el PIVOTE    */
}
   tmp = data[i];                 /* Se guarda el elemento que fue mayor que el  elemento PIVOTE          */
   data[i] = data[hi];            /* En la posición (donde se finalizo en el ciclo anterior while) del elemento */
                                  /* mayor que el PIVOTE se coloca ahí el elemento de partición  PIVOTE.  */
   data[hi] = tmp;                /* En la posición que ocupaba el elemento de partición,
                                     se coloca ahí el que fue mayor que el.                               */
   qck_sort( data, lo, i-1);      /* Llamada recursiva para la parte izquierda                            */
   qck_sort( data, i+1, hi);      /* Llamada recursiva para la parte derecha                              */
  }                               /*  fin de if                                                           */
}                                 /* fin de qck_sort()                                                    */
```

Listado 4. *QuickSort()*

La función selecciona el elemento de partición *data[hi]* y da un valor inicial a su variable índice *i*. La declaración de el ciclo exterior *while(1)* es un ciclo que maneja el cuerpo principal de la función. Contenidos en ese ciclo están dos ciclos más anidados. Su propósito es recorrer sus variables índices a través del arreglo de datos buscando los elementos que necesitan reubicarse. Cuando el ciclo interior termina, la función examina si *i* y *j* se han cruzado. Si es así, el ciclo exterior finaliza; *qck_sort()* entonces reubica el elemento partición y se invoca a si mismo

recursivamente en los dos nuevos subarreglos creados. Si *i* y *j* no se han cruzado, la función cambia elementos en las posiciones *data[i]* y *data[j]*, continuando con la siguiente iteración del ciclo exterior *while*.

Análisis del algoritmo.

Quicksort es recursivo, y por lo tanto, su análisis requiere resolver una fórmula de recurrencia. Se hará el análisis para *quicksort*, suponiendo un pivote aleatorio. Tomaremos T(0) = T(1)=1. El tiempo de ejecución de *quicksort* es igual al tiempo de ejecución de dos llamadas recursivas más el tiempo lineal gastado en la partición, la selección del pivote toma solamente tiempo constante. Esto produce la relación básica para *quicksort:*

$T(n) = T(i) + T(n-i-1) + cn.$

donde $i = |S_1|$ es el número de elementos en S_1 y S_1 el conjunto de datos obtenidos. Analizaremos tres casos: el peor caso, el caso promedio y el mejor caso.

Análisis del peor caso. El pivote es el más elemento pequeño, siempre. Entonces $i=0$ y si ignoramos $T(0) = 1$, lo cual es insignificante, la recurrencia es $T(n) = T(n-1) + cn, \quad n>1$.

Usando la ecuación anterior recursivamente, obtenemos:

$T(n-1) = T(n-2) + c(n-1), \quad T(n-2) = T(n-3) + c(n-2).$

Aplicando la función sucesivamente, obtenemos: $T(2) = T(1) + c(2)$.

Sumando todas estas ecuaciones se obtiene:

$$T(n) = T(1) + c \sum_{i=2}^{n} i \quad = O(n^2).$$

Análisis del mejor caso. En el mejor caso, el pivote está en la mitad. Asumimos que los dos subarchivos son exactamente la mitad del tamaño original, ya que estamos interesados en la respuesta de la O grande $O(n)$.

Entonces,

$T(n) = 2T(n/2) + cn.$

Dividiendo ambos lados de la ecuación por n, $\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + c.$

Por sumas telescópicas y aplicando la función recursivamente

$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + c,\quad \frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + c, \ldots, \frac{T(2)}{2} = \frac{T(1)}{1} + c.$

Sumando todas las ecuaciones y notando que hay logn ecuaciones, se obtiene

$\frac{T(n)}{n} = \frac{T(1)}{1} + c \cdot \log n \Rightarrow T(n) = cn\log n + n$

por lo tanto el algoritmo es $O(n \log n)$

Análisis del caso Promedio. Para el análisis del caso promedio, asumiremos que cada tamaño de archivo S_1 es igual y por lo tanto tiene probabilidad 1/n. Esta suposición es válida para nuestro pivote y estrategia de partición, pero no es válida para otras. La estrategia de partición que no conserva la forma aleatoria de los subarchivos no puede usar este análisis.
Con esta suposición, el valor promedio de T(i), y por lo tanto T(n-i-1), es

$\frac{1}{n}\sum_{j=0}^{n-1} T(j)$

entonces la ecuación $T(n) = T(i) + T(n-i-1) + cn$, *se convierte en*

$nT(n) = 2\left[\sum_{j=0}^{n-1} T(j)\right] + cn^2$

por suma telescópica tenemos:

$(n-1)T(n-1) = 2\left[\sum_{j=0}^{n-2} T(j)\right] + c(n-1)^2$ restándola a la ecuación anterior tenemos:

$nT(n) - (n-1)T(n-1) = 2T(n-1) + 2cn - c$

removiendo -1, tenemos : $nT(n) = (n+1)T(n-1) + 2cn.$

Ahora la fórmula de $T(n)$ está en términos de $T(n-1)$ solamente. Otra vez por sumas telescópicas en esta ecuación dividiendo por $n(n+1)$:

$\frac{T(n)}{n+1} = \frac{T(n-1)}{n} + \frac{2c}{n+1}.$ (Ecuación 1)

Se aplica suma telescópica y se aplica recursivamente:

$$\frac{T(n-1)}{n}=\frac{T(n-2)}{n-1}+\frac{2c}{n},\ \frac{T(n-2)}{n-1}=\frac{T(n-3)}{n-2}+\frac{2c}{n-1},\ldots,\frac{T(2)}{3}=\frac{T(1)}{2}+\frac{2c}{3}.$$

Sumándola a la Ecuación 1 se obtiene: $\frac{T(n)}{n+1}=\frac{T(1)}{2}+2c\sum_{i=3}^{n+1}\frac{1}{i}$

La suma es aproximada a $\log_e(n+1)+\gamma-\frac{3}{2}$ donde $\gamma \approx 0.577$ que es conocida como la constante de Euler, por lo que finalmente:

$\frac{T(n)}{n+1}+O(logn)$ Y $T(n) = O(n\log n)$.

3.2.5 HEAPSORT

Ahora presentamos el algoritmo *Heapsort*. Este algoritmo deriva su nombre de la estructura de datos que emplea. Antes de explicar la técnica de ordenamiento en sí misma, revisaremos la estructura de datos.

Un ***heap*** es un árbol binario completo con la propiedad de que la llave asociada con cualquier nodo dado *n* es más grande que las llaves de sus hijos. La Figura 3.7 da un ejemplo.

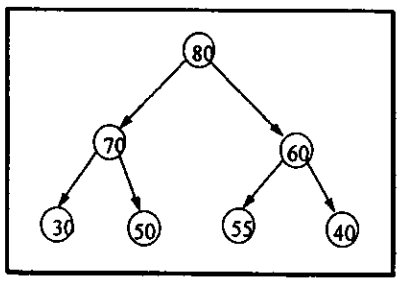


Figura 3.7 *Ejemplo de Heap*

Un *heap* tiene diversos usos; uno de los más comunes es implementar colas de prioridades. Refiriéndonos a la Figura 3.7, vemos que el elemento posicionado en la raíz siempre tiene la más alta prioridad. Esto puede ser una forma conveniente de aplicaciones para fijar tareas prioritarias.

Cuando quitamos un elemento de un *heap*, debemos reorganizar el árbol. El hijo del nodo borrado con la mayor prioridad llegará a ser el nuevo padre; uno de los hijos de este nodo lo reemplazará; y así sucesivamente. De esta manera, la implementación de un *heap* es un proceso de dos etapas. Primero, debemos trasformar un árbol binario completo en un *heap*. Entonces cuando los elementos son insertados y removidos, debemos mantener la integridad del *heap*.

Analicemos el proceso de transformar un árbol binario completo en un heap. Considere el árbol mostrado en la Figura 3.8.

Para transformarlo en un heap, tenemos que cambiar el nodo *D* con el nodo *B*; una vez movido, necesitamos intercambiar el nodo *B* con el nodo *A*. Aunque simple teoría, esta técnica tiene un defecto en que un hijo no puede fácilmente accesar a su padre. Una solución es agregar apuntadores a cada nodo que lo liguen con su padre. Sin embargo, esto trata un caso particular, no el problema. Una mejor solución es usar un arreglo. Usando una implementación de arreglo de un árbol binario, los hijos de cualquier nodo *i* son localizados en *2i* y *2i +1*; su padre es localizado en *i/2*. Así, vía fórmulas, es fácil referirnos a los elementos del árbol.

Hay un aspecto negativo de usar arreglos para implementar árboles. El problema está en extender árboles y en la dificultad asociada en la programación con localidades de arreglo vacías. Sin embargo, por definición, un heap está basado en un árbol binario balanceado, el cual garantiza que no habrá localidades vacías dentro de el arreglo.

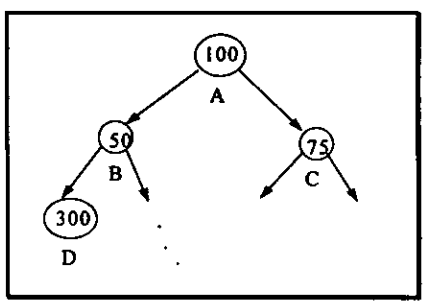


Figura 3.8 *Arbol para construir un Heap*

En este algoritmo la estrategia básica es construir un heap binario de *n* elementos. El problema principal con este algoritmo es que usa un arreglo extra. Así, el requerimiento de memoria es doble.

Para transformar un árbol binario en un heap, iniciemos en el final del arreglo y movamos hacia arriba la raíz, cambiando elementos como sea requerido. El código aparece en el Listado 5 y usa dos funciones *form_heap()* y *buildheap()*.

La función *form_heap()* toma tres argumentos: un apuntador al arreglo de datos y dos variables enteras que describen su límite inferior y superior. Su tarea es formar un heap iniciando en la posición *lo*, límite inferior. El primer *if* determina si tiene algún hijo; la función indica inmediatamente si no tiene hijos. La función entonces decide cual hijo procesar -el más grande de los dos- y asigna su índice a la posición *desc*. Entonces, si el hijo es más grande que su padre, cambia los dos elementos y se invoca a sí misma recursivamente para continuar el proceso en el próximo nivel en el árbol. Nótese que *form_heap()* asume que ningún cambio es requerido, el resto del árbol bajo este punto está ya en forma heap.

La función *buildheap()* del Listado 5 es la rutina que invoca *form_heap()*. Requiere dos argumentos: el arreglo y su tamaño. Su único ciclo inicia por calcular la mitad del arreglo. Entonces, mientras decrementa su variable de control, la función invoca iterativamente *form_heap()* con *i* como parámetro de mitad, argumento *lo* de *form_heap()*. Esto significa que del nodo *i* hacia todos sus descendientes el árbol será trasformado en un heap. Otra vez teniendo en cuenta que *form_heap()* terminará tan pronto como identifique un caso donde el padre es más grande que ambos de sus hijos. El arreglo entero queda en forma de heap cuando *buildheap()* termina.

Estas dos funciones pueden ahora servir como fundamento para un proceso heapsort. Consideremos que después del heap inicial del arreglo, el elemento más grande está en la posición raíz. Si removieramos ese elemento y reorganizaramos el heap, el segundo elemento más grande estaría en la posición raíz. Podríamos proceder de esta manera hasta que hubiéramos procesado todos los elementos.

Nótese que el proceso descrito ordena elementos en orden inverso. Podríamos hacer un trabajo rápido de este problema invirtiendo simplemente el heap. Sin embargo, consideraremos que cuando se quita el nodo raíz del heap, el árbol tiene un elemento menos. Después reorganizamos el heap y podemos reusar esa localidad vacía para guardar el elemento removido. Continuamos

en esta manera con cada elemento sucesivo; cuando el proceso termine, el arreglo entero habrá sido ordenado.

```
void form_heap(int data[], int lo, int hi)
{  int tmp, desc;
 if (2*(lo+1)-1 > hi)                    /* Determina si tiene hijo si no, nada que hacer */
   return;
                                         /* Decide cual hijo procesar -el mas grande de los  dos-
                                            y   asigna su indice a la variable desc   */
 if ( (2*(lo+1)) <= hi && data[2*(lo+1)]
        > data[2*(lo+1)-1])
   desc = 2*(lo+1);                      /* hijo derecho     */
 else
   desc = 2*(lo+1)-1;                    /* hijo izquierdo   */
 if (data[lo] < data[desc]) {            /* Si el hijo es mas grande que su padre */
    tmp = data[lo];                      /* cambia los dos elementos   */
    data[lo]=data[desc];
    data[desc]=tmp;
    form_heap(data,desc,hi);             /* el siguiente nivel en el árbol    */
  }
}                                        /*  fin de form_heap()*/
void buildheap(int data[], int size) {
  int j;                                 /* Para construir el heap inicia de la mitad del  arreglo */
  for (j = size/2; j>=0; j--)            /* Del nodo j hacia todos sus descendientes                */
    form_heap(data,j,size);              /* el árbol ser  transformado en un heap.                  */
}                                        /* fin de buildheap()                                       */
```

Listado 5. *HeapSort*

Podemos ahora formalizar la presentación del algoritmo:

1. Construir el heap inicial.

2. Cambiar el nodo raíz con el último nodo de el arreglo.

3. Reorganizar el árbol con la función *biuldheap()*.

4. Repetir los pasos 2 y 3 hasta que todos los elementos hayan sido procesados.

El Listado 6 contiene el código para la función *heap_sort()*.

```
void heap_sort( int data[], int size)
{ int aux,i;
 buildheap( data, size-1);          /* Construye el heap.                                    */
                                    /* En el siguiente ciclo for intercambia la posicion
                                       final i con  con el valor de la posición 0 y          */
                                    /* reordena el heap de la posición 0 a la  posición i-1. */
 for (i = size-1; i>=0; i--){       /* Comienza del extremo derecho del arreglo              */
  aux = data[0];                    /* El elemento de la posición 0 lo guarda  este elemento
                                       es el nodo raíz del heap                              */
  data[0] = data[i];                /* En la posición cero coloca el i-ésimo  elemento.      */
  data[i] = aux;                    /* Y en la posición i coloca el nodo raíz  que es el mayor. */
  form_heap( data, 0, i-1);         /* reordena el hep de la posición 0 a la  posición i-1 puesto que */
                                    /* de la posición i a size el arreglo queda ordenado en  forma ascendente. */
 }                                  /* Fin de for. Queda el arreglo ordenado ascendentemente.  */
}                                   /* Fin de heap_sort()                                    */
```

Listado 6. ***Función heap_sort()***

La función *Heap_Sort()* es la que implementa el algoritmo *heapsort*. Usa las funciones *builheap()* y *form_heap()* para crear y mantener el heap; revisa el fin actual del arreglo vía su contador de ciclo *i*.

Análisis

Inicialmente vía *buildheap()*, la rutina *form_heap()* es llamada una vez por cada nodo que tiene un hijo; esto requiere tiempo $O(n)$. En *heap_sort()*, *form_heap()* es llamado $n-1$ veces con una máxima profundidad de $log_2(n+1)$. Como resultado, la complejidad completa llega a ser $O(n \cdot log_2 n)$.

3.2.6 MERGESORT

En esta sección presentamos el método de ordenamiento llamado *MergeSort*. Como su nombre lo indica, mezclar (*merge*) juega el papel principal en este algoritmo de ordenamiento. *Merging* es el proceso por el cual combinamos dos o más conjuntos de datos en uno. Por ejemplo, considere dos arreglos ordenados: A de tamaño m, y B de tamaño n. Unir estos dos conjuntos de datos crearía un tercer arreglo ordenado C de tamaño $m+n$ que contiene todos los elementos de ambos arreglos. El listado de la Figura 3.9 presenta una versión en pseudo-código con la descripción de tal algoritmo.

La función *merge()* comienza dando valores iniciales a sus variables de control. Con cada iteración del ciclo inicial *while*, la función selecciona y guarda en el arreglo *C* el siguiente elemento más grande de los arreglos *A* y *B*; entonces avanza las variables de control apropiadamente. Nótese que el primer ciclo termina cuando una de las variables de control alcanza el fin de su arreglo correspondiente. Por lo tanto, *merge()* debe determinar cual arreglo no ha sido agotado y entonces copia sus elementos restantes en el arreglo *C*.

```
merge (C,A,B,m,n)
{

 i=1;                          // índice en arreglo A
 j=1;                          // índice en arreglo B
 k=1;                          // índice en arreglo C

 while (i<=m and j<=n) {
   if (A[i]) <= B[j]
     C[k++]=A[i++];
   else
     C[k++]=B[j++];
 }
 if (i<=m)                     // procesa elementos restantes
   while (i<=m)
     C[k++]=A[i++];
 else
   while (j<=m)
   C[k++]=B[j++];
 }
```

Figura 3.9 *Pseudocódigo de Merge*

Para entender este proceso, necesitamos alterar nuestra idea de almacenamiento de arreglo temporalmente, imaginemos un arreglo no como un conjunto de elementos, sino como un conjunto de subarreglos adyacentes. Por ejemplo, podríamos ver un arreglo de tamaño *n* como *n* arreglos adyacentes de tamaño *1*. Obviamente, si los subarreglos son de tamaño *1*, están ordenados. Ahora considere que sucedería si nosotros fuéramos a combinar pares adyacentes de subarreglos. Esto crearía subarreglos adyacentes de tamaño *2* también ordenados. Podríamos

repetir este proceso para crear subarreglos adyacentes de tamaño *4,8,...* y así sucesivamente. Eventualmente, alcanzaríamos el caso donde solamente dos subarreglos permanecen; cuando mezclemos estos, el arreglo entero es ordenado. La Figura 3.10 ilustra este proceso.

Estado inicial.	[10 5 6 3 4 1 9 2 8 7]
Después de 1ª. iteración	[5 10] [3 6] [1 4] [2 9] [7 8]
2ª. iteración	[3 5 6 10] [1 2 4 9] [7 8]
3ª. iteración	[1 2 3 4 5 6 9 10] [7 8]
Iteración final	[1 2 3 4 5 6 7 8 9 10]

Figura 3.10 *Ejecución del MergeSort()*

La operación fundamental en este algoritmo es mezclar dos listas ordenadas. Dado que las listas están ordenadas, esto puede ser hecho en un paso desde la entrada de las listas colocándolas en una tercer lista. Este algoritmo toma dos arreglos de entrada y un arreglo de salida.

El Listado 7 contiene el código para el algoritmo de mergesort:

Nótese que en esta versión, el proceso *merge()* requiere cinco argumentos: Los primeros dos son el arreglo destino y el fuente; los siguientes tres son variables índices que denotan los subarreglos adyacentes que se unirán en el arreglo fuente. El Listado 8 contiene otras dos funciones que completan la implementación de el algoritmo *mergesort()*.

Función *mrg_pass()*.- Esta función es invocada con cuatro argumentos: Los primeros dos son los arreglos destino y fuente, *size* es el tamaño de el arreglo y *len* es la longitud del subarreglo para cada paso. La función divide el arreglo *from[]* en subarreglos de tamaño *len* e invoca *merge()* una vez por cada par adyacente.

```
void merge ( int low, int mid, int high)
int to[]                                  /*  Arreglo destino   */
int from[]                                /*  Arreglo fuente */
{  int ilow,                              /* indice del limite inferior del subarreglo izquierdo      */
     ihigh,                               /* indice del limite inferior del subarreglo derecho        */
     ito;                                 /* Indice para la posición del arreglo destino              */
  ilow = ito = low;
  ihigh = mid +1;
  while (ilow <= mid && ihigh <= high) {  /* Mientras los índices de los  subarreglos no se crucen.   */
    if ( from[ilow] < from[ihigh]){       /* Si el elemento del subarreglo  izquierdo es menor.       */
      to[ito] = from[ilow];               /* Colocarlo en el arreglo destino.                         */
      ilow++;
    } else {                              /* Si el elemento del segundo subarreglo fue el mayor. Entonces este */
       to[ito]=from[ihigh];               /* colocarlo en el arreglo destino.                         */
       ihigh++;   }
    ito++;                                /* Avanza una posición el indice del  arreglo destino       */
  }
  while ( ilow <= mid )                   /* Copia los elementos restantes                            */
    to[ito++] = from[ilow++];             /* del subarreglo izquierdo y del                           */
  while ( ihigh <= high )
    to[ito++] = from[ihigh++];            /* subarreglo derecho en el arreglo destino                 */
}
```

Listado 7. *Merge() modificado*

Función *mrg_sort()*.-maneja el algoritmo entero de mergesort. Es invocada con dos argumentos: el arreglo a ordenar, y su tamaño. Su ciclo que maneja calcula la longitud de el subarreglo y llama *mrg_pass()*.

Nótese que durante cada iteración del ciclo *while*, *mrg_sort()* llama *mrg_pass()* dos veces, alternando los primeros dos argumentos. Es decir, durante la primera llamada *mrg_pass()* ordena de *data[]* en *tmp[]*; la segunda llamada invierte ese orden. Esto salva el tiempo que nosotros de otra manera gastamos copiando los elementos de *tmp[]* regresando a *data[]* después de cada paso.

Análisis

Mergesort es un ejemplo clásico de técnicas usadas para analizar rutinas recursivas. No es obvio que mergesort puede fácilmente ser reescrita sin recursión, así tenemos que escribir una relación

de recurrencia para el tiempo de ejecución. Asumiremos que *n* es una potencia de *2*, así dividiremos en mitades. Para *n=1*, el tiempo para mergesort es constante, el cual denotaremos por 1. El tiempo para mergesort es igual al tiempo para hacer mergesorts recursivos de tamaño *n/2*, más el tiempo para mezclar el cual es lineal. La ecuación siguiente expresa lo dicho:

$T(1) = 1$ y $T(n) = 2T(n/2) + n$.

```
void mrg_pass(
int to[],                                   /* Arreglo destino              */
int from[],                                 /* Arreglo fuente               */
int size,                                   /* Tamaño del arreglo           */
int len )                                   /* Tamaño de los subarreglos    */
{ int low = 0;
 while (low < size - 2*len){                /* Caso en el que se divide     */
  merge (to,from,low,low+len-1,low+2*len-1); /* el arreglo en subarreglos    */
  low+=2*len;                               /* adyacentes de tamaño len.*/
 }
 if (low+len-1 < size) {                    /* Queda solo un subarreglo     */
  merge(to,from,low,low+len-1,size);        /* de tamaño len.               */
 } else {
    while (low <= size) {                   /* Ya no se puede particionar en */
     to[low] = from[low];                   /* subarreglos, entonces se agregan los */
     low++;                                 /* elementos restantes al arreglo destino. */
    }
 }
}
void mrg_sort(int data[], int size)
{ int tmp[2048];                            /* malloc                       */
 int len =1;                                /* longitud de subarchivo */
 while (len < size) {
  mrg_pass(tmp,data,size-1,len);            /* Mezcla de data[] en tmp[]    */
  len *=2;
  despliega(tmp, size);
  mrg_pass(data,tmp,size-1,len);            /* Mezcla de tmp[] y guarda en data[] */
  len*=2; }
}
```

Listado 8. *MergeSort()*

Para resolver esta relación de recurrencia, primero la dividiremos por *n*. Obtenemos:

$$\frac{T(n)}{n} = \frac{T(n/2)}{n/2} + 1.$$

Esta ecuación es válida para cualquier n que sea una potencia de 2, así podemos también escribir

$$\frac{T(n/2)}{n/2} = \frac{T(n/4)}{n/4} + 1 \quad \text{y} \quad \frac{T(n/4)}{n/4} = \frac{T(n/8)}{n/8} + 1 \ldots$$

sucesivamente obtenemos: $\frac{T(2)}{2} = \frac{T(1)}{1} + 1.$

Ahora sumamos todas las ecuaciones. Es decir, sumamos todos los términos del lado izquierdo y los igualamos a la suma de todos los términos del lado derecho. Observe que el término *T(n/2)/(n/2)* aparece en ambos lados por lo que se cancela. De hecho, virtualmente todos los términos que aparecen en ambos lados son cancelados, pues es una suma telescópica. Después de que todo es sumado, el resultado final es:

$$\frac{T(n)}{n} = \frac{T(1)}{1} + \log n,$$

porque todos los otros términos son cancelados y quedan log*n* ecuaciones y así todos los unos (1s) en el final de estas ecuaciones se suman a log*n*. Multiplicando por *n* se obtiene el resultado final[1]

$T(n) = n \cdot \log n + n = O(n \cdot \log n)$.

[1] Mark Allen Weiss, Data Structures and Algorithm Analysis,1992.Pág.227

Capítulo 4. Medidas del Desorden

INTRODUCCION

El problema de Ordenamiento consiste en organizar una secuencia de elementos en orden ascendente o descendente. Pero, ¿qué significa que exista adaptabilidad en el problema de ordenamiento? o más específicamente: ¿qué significa que un algoritmo sea adaptivo? Una respuesta intuitiva la dá Mehlhorn[15]:

> "Cuando un algoritmo de ordenamiento toma ventaja del orden existente en los datos de entrada, el tiempo que toma el algoritmo para ordenar es una función suavemente creciente que depende del tamaño de la secuencia y del desorden en ella. En este caso decimos que el algoritmo es adaptivo"

Una panorámica sobre la importancia de la adaptabilidad en el problema de ordenamiento la dan Estivill-Castro y D. Wood[7]:

> "El diseño y análisis de algoritmos de ordenamiento adaptivos, ha hecho importantes contribuciones, tanto para la teoría como para la práctica. Desde el punto de vista teórico, las contribuciones son:
>
> - La descripción del desempeño computacional del algoritmo de ordenamiento no solo depende del tamaño de un ejemplar del problema sino, también, del desorden que hay en el ejemplar;
> - El establecimiento de nuevas relaciones entre las medidas del desorden;
> - La introducción de nuevos algoritmos de ordenamiento que toman ventaja del orden existente en la secuencia de entrada;
> - La prueba de que varios de los nuevos algoritmos de ordenamiento son adaptivamente óptimos con respecto a múltiples medidas del desorden.
>
> Las principales contribuciones desde el punto de vista práctico son:
>
> - La demostración de que múltiples algoritmos generalmente en uso son adaptivos.

- El desarrollo de nuevos algoritmos, similares a los comúnmente usados que se ejecutan competitivamente sobre secuencias aleatorias y son significativamente más rápidos sobre secuencias casi ordenadas.

Los algoritmos de ordenamiento adaptivos son atractivos, porque las secuencias casi ordenadas son muy comunes en la práctica."

4.1 MEDIDAS DEL DESORDEN

Mehlhorn [15], justifica las medidas del desorden de la siguiente manera:

> "Identificar en algún sentido casos fáciles de un problema computacional y utilizar esa facilidad tiene interés esencial. En ordenamiento, tal facilidad puede ser identificada como un orden existente."

El número de operaciones que un algoritmo de ordenamiento ejecuta es definido como la medida de eficiencia. El número de comparaciones proporciona no sólo una estimación razonable del tiempo relativo requerido para todas las implantaciones, sino que también permite cotas mínimas bajo el modelo computacional de árboles de decisión.

A continuación describimos algunas medidas para cuantificar el orden existente en una secuencia, las cuales son usadas en el estudio de adaptabilidad y las propiedades generales que deben satisfacer, de acuerdo a H. Mannila [2], las medidas del desorden.

4.1.1 Medidas Naturales.

Definiciones básicas.

i) Sea $X= <x_1,...,x_n>$ una secuencia de n elementos x_i de algún conjunto totalmente ordenado. Por simplicidad, se asume que cada x_i es distinta. Para simplificar, supongamos que los elementos x_i son enteros distintos. Consideremos que el orden ascendente es el correcto, esto es, se desea ordenar a la secuencia X en forma ascendente.

ii) Sean dos secuencias $X = <x_1,...,x_n>$ y $Y=<y_1,...,y_n>$, se define su concatenación XY como la secuencia $<x_1,...,x_n,y_1,...,y_n>$.

iii) Una secuencia obtenida por borrar cero o más elementos de X es llamada un subsecuencia de X.

iv) Sea $|X|$ la longitud de la secuencia X

v) Sea $||S||$ la cardinalidad de un conjunto S.

vi) Sea S_n el conjunto de todas las permutaciones de $\{1,...,n\}$, y $logx=log_2(max\{2,x\})$.

vii) Para un elemento x, se define su rango en una secuencia Y como el número de elementos en Y que son más pequeños que x, esto es, $rango(x,Y)=||\{y_j | 1<j<|Y| \; y \; x>y_j\}||$.

4.1.1.1 **Inv(X). Inversiones.**

Inversiones. Dada una secuencia X, las inversiones(X) representan el número de parejas que se encuentran en orden incorrecto en X o que están en posición invertida. Formalmente, se define como:

$$Inv(X) = || \{(i,j) / \; 1 \leq i < j \leq n \; y \; x_i > x_j\} ||.$$

El valor de $Inv(X)$ indica cuántos cambios entre cada par de elementos son necesarios para ordenar X. Se tiene que $0 \leq Inv(X) \leq n(n-1)/2$ para todo secuencia X, de donde se obtiene, más específicamente,

$$Inv\,(X) = \begin{cases} 0 & X \;\; \textit{está ordenada en forma ascendente.} \\ \dfrac{n(n-1)}{2} & X \;\; \textit{está ordenada en forma descendente.} \end{cases}$$

Ejemplo 1. Sea $Y = <3,1,9,4,2,10,5,7,6,8>$. El elemento en la posición uno de Y es mayor que el elemento de la posición dos, entonces la pareja $(1,2)$ representa una inversión de$Y$. El conjunto de inversiones o parejas invertidas, para Y es:

$$\{(1,2),(1,5),(3,4),(3,5),(3,7),(3,8),(3,9),(3,10),(4,5),(6,7),(6,8),(6,9),(6,10),(8,9)\}$$

Así que el número de inversiones de Y es $Inv(X) = 14$.

Ejemplo 2. Sea $W = <n+1,n+2,...,2n,1,2,...,n>$. Esta secuencia tiene un número cuadrático de inversiones.

4.1.1.2 **Runs(X). Corridas.**

Corridas. Dada una secuencia X, el valor de *Runs(X)* representa el número de subsecuencias ascendentes de X y se define formalmente como:

$$Runs(X) = 1 + || \{ i / \ 1 \leq i < n \ y \ x_{i+1} < x_i \} ||.$$

Run se define como el número de "separaciones" entre las corridas[7]. Las subsecuencias ascendentes de X están formadas por elementos en posiciones contiguas de X, por ello se les llama *corridas (runs)*, se tiene que:

$$Runs(X) = \begin{cases} 1 & X \text{ está ordenada en forma ascendente.} \\ n & X \text{ está ordenada en forma descendente.} \end{cases}$$

Cuando la secuencia se encuentra casi ordenada, el valor de *Runs(X)* es un número pequeño, pero cuando existe desorden local el número de corridas es grande.

Ejemplo 3. Para la secuencia Y del Ejemplo 1, $Runs(Y) = 5$, pues podemos ver a Y dividida de la siguiente forma: $Y = <3, \downarrow 1,9, \downarrow 4, \downarrow 2,10, \downarrow 5,7, \downarrow 6,8>$, es decir tiene 5 separaciones.

Ejemplo 4. Para la secuencia W del Ejemplo 2, $Runs(W) = 1$.

4.1.1.3 **Las(X).**

Las(X). Dada una secuencia X, el valor de *las(X)* representa el tamaño de la subsecuencia ascendente más larga de X, y se define formalmente como:

$$las(X) = max\{ t / \exists \ i(1),...,i(t) \ tales \ que \ \ 1 \leq i(1) < ... < i(t) \leq n \ y \ x_{i(1)} < ... < x_{i(t)} \}.$$

Como el valor de ***las(X)*** representa la longitud más larga de una corrida ascendente, se tiene entonces que $1 \leq las(X) \leq n$.

Ejemplo 5. Para la secuencia Y del Ejemplo 1, $las(Y) = 2$.

Ejemplo 6. Para la secuencia W del Ejemplo 2, $las(W) = n$.

4.1.1.4 **Rem(X).**

Rem. Dada una secuencia X, el valor de $rem(X)$ indica cuantos elementos tienen que ser removidos de la secuencia para dejar la lista ordenada. Se define formalmente como:

$$rem(X) = n - las(X)$$

Se tiene que : $rem(X) = \begin{cases} 0 & X \text{ está ordenada en forma ascendente.} \\ n-1 & X \text{ está ordenada en forma descendente.} \end{cases}$

Ejemplo 7. Para la secuencia Y del Ejemplo 1, $rem(Y) = 10 - 2 = 8$.

Ejemplo 8. Para la secuencia W del Ejemplo 2, $rem(W) = 2n - n = n$.

4.1.1.5 **Exc(X). Intercambios.**

Intercambios. Dada una secuencia X, el valor de ***Exc(X)*** indica el número más pequeño de cambios necesarios de elementos arbitrarios para ordenar la secuencia X en forma ascendente.

Se tiene que: $0 \leq Exc(X) \leq Inv(X) \quad \forall X$.

Ejemplo 9. Para la secuencia Y del Ejemplo 1, $Exc(Y) = 8$.

Ejemplo 10. Sea $W1 = \langle n,1,2,\ldots,(n-1)\rangle$. El número de cambios para esta secuencia es, $Exc(X) = n - 1$.

Mannila [2], comenta algunas observaciones sobre estas medidas:

"Comparar estas medidas no es fácil. *Inv* y *Runs* son medidas muy bien conocidas y su comportamiento es fácil de comprender; las inversiones, *Inv*, cuantifican el preorden global de una secuencia, mientras que la medida *Runs* cuantifica el preorden local. Al parecer el valor de *Rem* combina estas dos propiedades, mientras que la medida *Exc* parece diferir drásticamente de las otras, a pesar de su definición natural. Estas diferencias muestran que el pre-orden puede ser medido en muchas formas."

4.1.2 **Otras Medidas.**

En esta subsección se definen otras medidas del desorden de las cuales algunas se basan en las anteriores.

4.1.2.1 **Dis(X). Distancia.**

Distancia. Dada una secuencia *X*, se define *Dis(X)* como la distancia más grande determinada por una inversión o una pareja invertida.

Esto hace énfasis en que "...una inversión, para la cual los elementos están muy apartados el desorden es más significativo, que una inversión en la cual los elementos están muy cerca el uno del otro" [7].

Ejemplo 11. Para la secuencia del Ejemplo 1, *Dis* = 7, ya que la *inv(3,10)* tiene la distancia más grande entre las *Inv(x)*. Viéndolo de otra manera: el número nueve, elemento de la posición *3*, está siete posiciones del número ocho, elemento de la posición 10; y son los elementos más apartados de la secuencia *Y*.

4.1.2.2 **Max(X). Máxima Distancia.**

Máxima Distancia. Dada una secuencia *X*, se define *Max(X)* como la máxima distancia que un elemento debe viajar hasta encontrar su posición correcta.

Una definición formal de esta medida:

$$Max(X) = \max_{1 \le i \le n} / i - \pi(i) /$$

donde $\pi(x)=rango(x_i,X)+1$, la posición final de x_i, para $1 \leq i \leq n$. Es decir, *Max* dice la distancia máxima que un elemento esta de su posición correcta.

Esta medida considera que el desorden local no es tan importante como el global.

Ejemplo 12. Para la secuencia *Y* del Ejemplo 1, $Max(Y) = 6$, pues el elemento de la posición *3*, el número nueve, debe viajar seis lugares para llegar a su posición correcta que es la nueve.

4.1.2.3 **SUS(X).**

SUS. Dada una secuencia *X*, el valor de *SUS(X)* indica el mínimo número de subsecuencias ascendentes en las que podemos particionar *X*. La medida *SUS* es una generalización natural de las corridas, su nombre proviene de *Shufflet Up-Sequences*.

Ejemplo 13. Para la secuencia *Y* del Ejemplo 1,

$SUS(Y) = ||\{<3,9,10>; <1,4,5,7,8>;<2,6>\}|| = 3.$

4.1.2.4 **SMS(X).**

SMS. Dada una secuencia *X*, el valor de *SMS(X)* indica el mínimo número de subsecuencias, ascendentes o descendentes en que podemos particionar la secuencia dada. Esta medida del desorden, es una generalización de la medida *SUS(X)*. *SMS* es abreviación de *Shuffed Monotone Subsequence*.

Ejemplo 14. Para la secuencia *Y* del Ejemplo 1, $SMS(Y) = 3$.

Ejemplo 15. Sea $W' = <6,5,8,7,10,9,12,11,4,3,2>$.

Para esta secuencia de datos *W'* se tiene que:

$Runs(W') = 7.$

$SUS(W') = ||\{<6,8,10,12>;<5,7,9,11>;<4>;<3>;<2>\}|| = 5.$
$SMS(W') = ||\{<6,8,10,12>;<5,7,9,11>;<4,3,2>\}|| = 3.$

4.1.2.5 **Enc(X).**

Enc Dada una secuencia *X*, el valor de *Enc(X)* se define como el número de listas ordenadas construidas por *MELSORT* al aplicarlo a la secuencia dada. Esta medida del desorden fue propuesta por *Skiena*[16].

4.1.2.6 **Osc(X). Oscilaciones.**

Osc. La Medida *Osc(X)* evalúa, en algunos casos la Oscilación entre el elemento más grande y el más pequeño de la secuencia *X*. Esta medida del desorden fue propuesta por *Levcopoulos y Petersson*[4]:

Una definición formal de esta medida:

$$Osc(X) = \sum_{i=1}^{n} \| \{j \mid 1 \le j < n \ y \ min\{x_j, x_{j+1}\} < x_i < max\{x_j, x_{j+1}\} \| .$$

La motivación de *Osc* viene de un interpretación geométrica de una secuencia *X*. Si graficamos cada elemento x_i, sobre el punto (i,x_i) en el plano y dibujamos segmentos de línea entre los puntos que corresponden a los elementos consecutivos en *X*, *Osc(X)* dice cuantos polígonos resultan de unir las oscilaciones. Más precisamente, *Osc(X)* es el número de intersecciones entre líneas horizontales para las x_i's y segmentos de línea definidos por x_i's consecutivos.

4.1.2.7 **Reg(X).**

Reg. Esta medida es definida por *Moffat y Peterson*[17,18]; y nos indican que cualquier algoritmo de ordenamiento Reg-óptimo es óptimamente adaptivo con respecto a las medidas *Inv, Dis, Max, Exc, Rem, Runs, SUS, SMS, Enc y Osc.*

4.1.2.8 **Block(X). Bloques.**

Block. Esta medida dice el número de subsecuencias consecutivas en la entrada que quedan en la salida ordenadas:

$$Block(X) = 1 + ||\{ i \mid 1 \leq i < n \ \ y \ \ \pi(i)+1 \neq \pi(i+1) \}||$$

donde $\pi(i)=rango(x_i,X)+1$, la posición final de x_i, para $1 \leq i \leq n$. Los elementos de *Block* son aquellos que reciben un nuevo sucesor en la secuencia ordenada. Estos elementos dividen la secuencia de entrada en subsecuencias consecutivas *Block(X)*, las cuales en lo sucesivo serán llamadas *blocks*. Otra interpretación es que si X es implementada por una lista ligada, entonces *Block(X)-1* dice cuantos apuntadores tienen que ser cambiados en orden para que la lista ligada represente la permutación ordenada de X.

4.2. PROPIEDADES GENERALES.

En esta sección se dan algunas de las condiciones generales que deben satisfacer todas las medidas del pre-orden.

Definición. Una Medida m es una función real, $m : N^{<N} \rightarrow N$, donde $N^{<N}$ denota al conjunto de todas las secuencias finitas de enteros distintos la cual cumple las siguientes condiciones:

1. $m(X) = 0$, si la secuencia X está ordenada en forma ascendente.
2. Sean las secuencias $X = <x_1,x_2,...,x_n>$ y $Y = <y_1,y_2,...,y_n>$. Si $x_i<x_j$ si y sólo si $y_i < y_j$ para todo i y j, entonces, $m(X) = m(Y)$.
3. Si X es una subsecuencia de Y entonces, $m(X) < m(Y)$.
4. Dadas las secuencias X y Y. Si $X < Y$, esto es, que cada elemento de X es más pequeño que cada elemento de Y, entonces, $m(XY) \leq m(XY) + m(Y)$.
5. Sea $a \in N$, entonces $m(<a>X) \leq |X| + m(X)$.

Las primeras dos condiciones indican que una lista ordenada tiene desorden igual a cero y que el valor de la medida m depende únicamente del orden de sus argumentos. Las siguientes tres, son condiciones más fuertes, de hecho, de acuerdo con *H. Mannila*, son necesarias para toda medida del desorden.

H. Mannila [2], da dos puntos de vista sobre el concepto de pre-orden o desorden:

a) El desorden es cuantificado por el número de operaciones de un tipo dado, el cual es necesario para ordenar la secuencia de entrada.

b) El desorden es cuantificado por la cantidad de información de la forma $x_i < x_j$, que se requiere para identificar la secuencia usando una forma dada de coleccionar la información, aproximación de información teórica.

Estos puntos de vista obligan a las medidas del desorden a satisfacer las últimas tres condiciones. El punto de vista dado por la observación **a)**, asume que el conjunto de operaciones permitidas es cerrado bajo las subsecuencias. Si podemos ordenar una secuencia de X con un cierto número de operaciones, entonces las restricciones de tales operaciones para X ordenarán X; si $X<Y$, entonces XY (la concatenación de X con Y) podrá ser ordenada si primero ordenamos X y luego Y, con un total de $m(X) + m(Y)$ operaciones; finalmente ordenar $<a>X$, puede hacerse ordenando X con $m(X)$ operaciones e insertando a en su lugar correspondiente en la secuencia, por lo cual a tiene $/X/$ diferentes posibilidades de ubicarse en la secuencia. De acuerdo con **b)**, que si tenemos suficiente información para identificar una secuencia de X, entonces esa misma información identifica a X; si $X<Y$, entonces XY es identificada por X y Y; y $<a>X$, es identificada al identificar X y después localizar el lugar de a.

4.3 CLASIFICACIÓN ENTRE MEDIDAS.

La medida Reg, nos induce a las siguientes preguntas: ¿son estas medidas diferentes? o ¿Cómo pueden relacionarse entre sí? Estivill-Castro y Wood [7] nos indican que: "Desde un punto de vista algorítmico, si dos medidas M_1 y M_2 particionan el conjunto de permutaciones en exactamente las mismas clases de conjuntos *below*,[1] entonces cualquier algoritmo que es M_1-*óptimo* es también M_2-*óptimo* y viceversa." Además, nos proporcionan las siguientes definiciones:

Definición. Sean M_1 y M_2 dos medidas del desorden. Decimos que:

[1] Los conjuntos *below* se definen en el siguiente capítulo.

a) M_1 es algorítmicamente más fina que (*algorithmicaly finer*) M_2, denotado *por* $M_1 \leq_{alg} M_2$, si y sólo si cualquier algoritmo M_1-*óptimo* es también M_2-*óptimo*.

b) M_1 y M_2 son algorítmicamente equivalentes si y sólo si $M_1 \leq_{alg} M_2$ y $M_2 \leq_{alg} M_1$, se denota como $M_1 =_{alg} M_2$.

La Figura 4.1 es dada por Estivill-Castro y D. Wood [7] para ilustrar el orden parcial entre las medias del desorden, con respecto a $\leq_{alg}$, la forma de "leer" el diagrama es de abajo hacia arriba; por ejemplo si un algoritmo es *Inv-óptimo* implica que es *Dis-óptimo*.

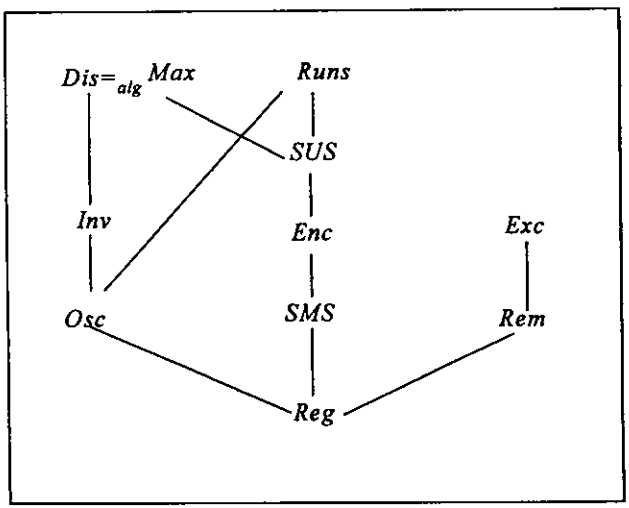


FIGURA 4.1 *Orden parcial entre las Medidas del Desorden[7].*

4.4 ALGORITMOS ÓPTIMOS

La medida de eficiencia de un algoritmo de ordenamiento, es el número de comparaciones que ejecuta, el cual no sólo proporciona una estimación razonable del tiempo relativo requerido para todas las implantaciones, sino que también permite cotas mínimas para ser obtenidas bajo el modelo computacional de Árboles de Decisión.

4.4.1 Ejemplo de Algoritmo Óptimo

Ordenamiento de Inserción Local.

Suponga que se tiene una estructura de datos para representar listas ordenadas, y que tal estructura es capaz de hacer inserciones en tiempo $O(1+log(d+1))$, donde d es la distancia del previo elemento insertado. Un algoritmo de ordenamiento que se implanta usando dicha estructura de datos es llamado ORDENAMIENTO DE INSERCION LOCAL (*Local Insertion Sort*). El tiempo de ejecución de este algoritmo es $\sum_j c(1+log(d_j+1))$ donde d_j es la distancia desde el elemento anterior:

$d_j = ||\{i/1<i<j \ y \ (x_{j-1} < x_i \ o \ x_j < x_i < x_{j+1})\}||$.

Teorema 2 El algoritmo de INSERCION LOCAL es óptimo con respecto al número de inversiones.

Demostración

La eficiencia del algoritmo de INSERCION LOCAL depende de la distancia entre inserciones sucesivas. En un ejemplar que contiene pocas inversiones, la mayoría de las inserciones ocurren cerca del final de la lista. En cada secuencia la distancia entre inserciones sucesivas debe ser pequeña, ya que es acotada por la mayor distancia desde el final de la lista de elementos a insertar y los elementos previos.

Si h_j es la cantidad que describe la distancia del elemento x_j desde el final de la lista, entonces se tiene que:

$d_j+1 < max\{h_j, h_{j-1}\} + 1 < h + j + h_{j+1} + 1 < (h_j+1)(h_{j-1}+1)$

Así que, $log(d_{j+1}) < log(h_{j+1}) + log(h_{j-1} + 1)$

y por tanto:

$$\sum_j c(1+\log(d_{j+1})) < cn + c\left(\sum_j \log(d_{j+1})\right) < cn + 2c\left(\sum_j \log(h_{j+1})\right) < cn + 2cn\log$$

$$\left(\prod_j \log(h_{j+1})1/2\right) < cn + 2cn\log\left(\sum_j \log(h_{j+1})/n\right) < cn + 2cn\log(1 + Inv(X)/n).$$

Finalmente, tenemos que:

$$\log |below(X,Inv)| = \Theta(n \log(1+Inv(X)/n))$$

lo cual demuestra que el algoritmo es óptimo con respecto a la medida del desorden denominada inversiones.

Capítulo 5. Algoritmos de Ordenamiento Adaptivos

En este capítulo revisaremos algunos de los algoritmos de ordenamiento adaptivos basados en selección. Este material esta basado en los artículos de O. Peterson[4], D.Wood & Estivill[7] y H. Mannila[2].

INTRODUCCION

Un algoritmo de ordenamiento es adaptivo si ordena secuencias que están casi ordenadas más rápido que secuencias aleatorias, donde la distancia es determinada por alguna medida de preordenamiento. Las investigaciones han apoyado los paradigmas de *inserción* y *divide y vencerás* para diseñar algoritmos de ordenamiento adaptivo. Aquí se muestra que el ordenamiento por selección es un paradigma que conduce a algoritmos adaptivos eficientes. Además, se presenta la prueba de un lema general que simplifica el diseño de algoritmos de ordenamiento adaptivos óptimos para la mayoría de las medidas de preordenamiento. El lema declara que, sin pérdida de generalidad, uno puede asumir el conocimiento de la cantidad de preordenamiento previo para el orden.

5.1 ORDENAMIENTO POR SELECCION

Es bien conocido que el tiempo $\Omega(n \log n)$ es necesario para ordenar n elementos en el peor caso y en el caso promedio en modelos de cómputo basados en comparación. A pesar de este hecho, en ocasiones algunas secuencias parecen ser más fáciles de ordenar, más rápido que otras como las que son indicadas por este límite inferior. Por ejemplo, una secuencia ya ordenada o la concatenación de dos secuencias ordenadas. Estas no deberían requerir la misma cantidad de recursos que una secuencia permutada aleatoriamente. El concepto de preordenamiento fue eventualmente formalizado por Mannila[2]. Mannila estudió el cuestionamiento de cómo el preordenamiento de una secuencia puede ser medido. Ejemplos de medidas de preordenamiento que él consideró son: el número de corridas y el número de inversiones. Mannila también estudió el problema de cómo un algoritmo de ordenamiento puede tomar ventaja y de tal modo adaptarse al orden existente.

Un algoritmo de ordenamiento se dice que se adapta con respecto a una medida de preordenamiento si ordena toda la secuencia, pero se desarrolla particularmente bien en las que tienen un alto grado de preordenamiento conforme a la medida. A mayor preorden en la entrada, más rápido debería ser ordenado. Sin embargo, el algoritmo debería adaptarse sin conocimiento previo de la cantidad de preordenamiento. La mayoría de los algoritmos de ordenamiento óptimos en el peor de los casos, por ejemplo Heapsort y Mergesort, no toman en cuenta el orden existente dentro de su entrada.

Después del trabajo de Mannila se ha investigado intensamente el área de ordenamiento adaptivo. La estrategia sostenida en todos estos algoritmos es divide y vencerás. La ventaja del ordenamiento por el paradigma de inserción en un ordenamiento adaptivo ha sido también establecida e implementaciones diferentes producen diferentes tipos de adaptabilidad.

En este capítulo se muestra que el tercer paradigma básico de ordenamiento, es decir, el *ordenamiento por selección*, es también útil cuando se diseñan algoritmos de ordenamiento adaptivos eficientes. Se implementa primero una cola de prioridades que consta de todos los elementos de la secuencia a ordenar de la cual repetidamente se extrae el elemento máximo y entonces se diseña un *selection sort* adaptivo genérico. La idea principal es que en lugar de ordenar todos los elementos de la cola de prioridad solamente se almacenan los que podrían ser los máximos de los elementos restantes. Después se muestra que varias versiones adaptivas de *selection sort* son ejemplares de este algoritmo genérico.

Se proporciona una técnica simple para diseñar algoritmos del tipo *selection sorts* adaptivos óptimamente con respecto a la medida *Block*, la cual determina el número de elementos que reciben nuevos sucesores durante el ordenamiento. El método está basado en la observación de la adaptabilidad inherente en un heap.

Se presenta un lema que declara que para la mayoría de las medidas de preordenamiento uno puede asumir conocimiento previo del valor de la medida cuando se diseña un algoritmo óptimo.

Se evalúan ejemplares fáciles en el problema de ordenamiento por una medida de preordenamiento, una función entera no negativa en una secuencia X que refleja cuanto difiere la secuencia de la permutación ordenada de X. Una medida de preordenamiento crece bastante con

la cantidad de desorden que con la cantidad de orden existente. El valor de la medida depende solamente en el orden relativo de los elementos.

Como se ha mencionado, un algoritmo de ordenamiento adaptivo es un algoritmo que se adapta al preordenamiento existente en la entrada, medido de alguna forma. El mayor preorden de una secuencia debería ser lo más rápido para ordenarla. Además, el algoritmo se debería adaptar sin ningún conocimiento previo de la cantidad de preordenamiento, pero cualquier tiempo gastado aquí es contado como parte del tiempo de ejecución.

El concepto de un algoritmo óptimo con respecto a una medida de preordenamiento fue dado en forma general por Mannila. Usamos la siguiente definición equivalente:

Definición 1. Sea M una medida de preordenamiento, y T_n el conjunto de árboles de comparación de los conjuntos S_n. Entonces para cualquier $k \geq 0$ y $n \geq 1$,

$$C_M(n,k) = \min_{T \in T_n} \max_{\pi \in below_M(n,k)} \{\text{el número de comparaciones gastadas por } T \text{ para ordenar } \pi\}$$

donde $below_M(n,k) = \{ \pi \mid \pi \in S_n \text{ y } M(\pi) \leq k \}$.

Definición 2. Sea M una medida de preordenamiento y S un algoritmo de ordenamiento basado en comparación que usa $T_S(X)$ pasos en la entrada X. Se dice que S es *M-optimal* u óptimo con respecto a M, si $T_S(X) = O(C_M(|X|, M(X)))$.

Cuando se prueba optimalidad de un algoritmo de ordenamiento adaptivo el siguiente teorema es útil.

Teorema 1. Sea M una medida de preordenamiento. Entonces

$C_M(n,k) = \Theta(n + \log||below_M(n,k)||)$ nos da una cota mínima y máxima para el número de comparaciones empleadas para ordenar S con respecto a la medida M.

5.2 ORDENAMIENTO ADAPTIVO POR SELECCION

Uno de los paradigmas de ordenamiento básico es la ordenación por selección. La representación más simple entre tales algoritmos es *Linear Selection Sort*. Presentado con una secuencia, este algoritmo selecciona el elemento máximo en una revisión lineal, y cambia el elemento encontrado con el último elemento de la secuencia. El mismo procedimiento entonces es aplicado para los elementos restantes de la secuencia hasta que la secuencia entera es ordenada. Como todos los elementos restantes deben ser examinados durante cada selección, *Selection_Sort* se beneficia de algún tipo de preordenamiento, y corre en tiempo $\Theta(n^2)$. En particular, usa tiempo cuadrático aún si la entrada está ya ordenada. Por contraste, el algoritmo *Linear Insertion Sort* ordena tal secuencia en tiempo lineal.

Aparentemente, la debilidad de *Selection_Sort* consiste de que usa un arreglo sin orden para implementar una cola de prioridades, la cual es un tipo de datos abstracto que mantiene la extracción e inserción del elemento de máxima prioridad. Una estructura de datos más eficiente para este propósito es el heap, el cual mantiene sus operaciones en tiempo logarítmico. El algoritmo de ordenamiento asociado, Heapsort, inicia insertando todos los elementos en un heap, y entonces repetidamente extrae el elemento máximo. Así se ejecuta en tiempo $O(nlogn)$; sin embargo, su eficiencia no es afectada significantemente por algún preordenamiento. Es decir, no es adaptivo.

Esencialmente hay dos diferentes acercamientos para construir un *Selection_Sort* que tome ventaja del orden existente dentro de la entrada. La primera posibilidad es inventar una estructura de datos adaptiva para implementar la cola de prioridades. Esto es, una estructura de datos que no siempre logre su límite en el peor caso, pero para el cual el tiempo de complejidad de las operaciones dependa en algún ordenamiento de la secuencia original. El primer algoritmo en el cual esta idea fue adoptada es el *Smoothsort* de Dijkstra[4]. *Smoothsort* implementa una cola de prioridades por un bosque ordenado de heaps completos, el cual es calculado en tiempo lineal, donde las raíces están en orden ascendente y los tamaños disminuyen exponencialmente. Dijkstra declaró que Smoothsort es adaptivo sin mencionar con respecto a que medida[4]. La adaptividad de Smoothsort fue después investigada por Hertel[4], quien probó que no era óptimo con respecto al número de inversiones. De hecho, hasta la fecha no se sabe si Smoothsort es óptimo con respecto a alguna medida de ordenamiento conocida. Recientemente, Chen y Carlsson[4],

demostraron que la optimalidad para la medida *inv* puede ser obtenida gastando tiempo lineal reordenando la entrada antes de construir el árbol de heaps.

La segunda forma de lograr adaptabilidad es motivada por la observación de que en lugar de ordenar todos los elementos en la cola de prioridades, solamente se ordenan los que pueden posiblemente ser los máximos de los elementos restantes, que llamamos los candidatos máximos, que necesitan ser ordenados. Para sostener este proyecto, se emplea alguna estructura de datos que proporcione candidatos máximos e interactúe con la cola de prioridades. La Figura 5.1 describe un algoritmo *Selection Sort* adaptivo genérico basado en este acercamiento.

```
procedure Adaptive Selection Sort (X:secuencia)
  Construir una estructura de datos S(X) para manipular la cola de prioridades
   Insertar algún elemento de S(X) en una cola de prioridad vacía
   for i:=1 to n do
       Extraer el elemento máximo de la cola de prioridades
       if (nuevos candidatos máximos son necesitados) then
         Recuperar elementos máximos de S(X)
         Insertar candidatos máximos en la cola de prioridades
       endif
   endfor
end
```

Figura 5.1 *Adaptive Selection Sort*

Supóngase que la cola de prioridades es implementada por una estructura de datos que soporta las operaciones en tiempo logarítmico, por ejemplo un heap. Si la entrada está casi ordenada, la estructura de datos economizará en el número de elementos proporcionados. Por lo tanto, la cola de prioridad contendrá algunos elementos durante la mayoría de las operaciones, y el algoritmo se completa en tiempo $O(nlogn)$. Por otro lado, en el peor caso la cola de prioridad consiste de un número lineal de elementos durante la mayoría de las operaciones, y el algoritmo tiene desempeño computacional de $\Theta(nlogn)$.

En las siguientes subsecciones se presentan varias versiones del algoritmo *selection sort* adaptivas que pueden ser vistas como ejemplos del algoritmo genérico antes mencionado.

5.2.1 **Rheapsort**

Igarashi y Wood[4] estudiaron la adaptabilidad en algoritmos de ordenamiento para la medida *Max*. Por ahora, supóngase que $Max(X)=k$ es conocido para ser ordenado. Es fácil ver que para seleccionar el elemento más grande en la secuencia X basta examinar los últimos $k+1$ elementos. Similarmente, el segundo elemento más grande puede ser encontrado entre los últimos $k+2$ elementos y así sucesivamente. Esta observación demuestra cuales elementos son los *max-candidatos*, y esto llevó a Igarashi y Wood al algoritmo *Rheapsort* para ordenar un arreglo X de longitud n, Figura 5.2.

```
Procedure Rheapsort(X:sequence;k:integer)
 Construir una heap para (x n-k,x n-k+1,...x n)
 for i:= 1 to n do
   Extraer el elemento máximo del heap
   if (X no es agotado) then
     Recuperar x n-k-i del arreglo.
     Insertar x n-k-i en el heap
   endif
 endfor
end
```

Figura 5.2 *Rheapsort*

El algoritmo Rheapsort no requiere ninguna estructura de datos, sólo el arreglo de entrada para mantener el heap. Recordemos que un heap puede ser construido en tiempo lineal. Como el heap inicialmente contiene $k+1$ elementos y cada inserción es precedida por una extracción, cada operación heap en el ciclo toma tiempo $O(\log k)$. Por lo tanto, el algoritmo se ejecuta en tiempo $O(n \log k)$. Estivill-Castro y Wood[4] probaron que $C_{Max}(n,k) = \Omega(n \log k)$, y por lo tanto Rheapsort es óptimo con respecto a *Max*, aplicando la Definición 2. Sin embargo, parece que el conocimiento del valor de la medida es crucial. En la práctica esto no es realista. En seguida se

presenta la prueba de un lema que muestra que en la mayoría de los casos no obstante es razonable hacer esta suposición cuando buscamos un algoritmo óptimo.

LEMA 2. Sea M una medida de preordenamiento para la cual $C_M(n,k) = \Omega(f(n,k))$, donde $f(n,k)=n\log k$ o $f(n,k)=n\log(k/n)$. Además, sea S un algoritmo de ordenamiento basado en comparación con la propiedad de que dado un número k arbitrario, ordena alguna secuencia X de longitud n con $M(X) \leq k$ en $T_S(X)=O(f(n,k))$ pasos. Entonces el algoritmo S puede ser aplicado para diseñar un algoritmo de ordenamiento *M-óptimo*[4].

Prueba. Supóngase, sin pérdida de generalidad, que $f(n,k)=n\log k$. El otro caso se prueba similarmente. Sea $|X|=n$ y $T_S(X) \leq c \cdot n\log k$, para alguna constante $c>0$, dado que $M(X) \leq k$. La idea es simple, se supone el valor de $M(X)$ y se aplica S repetidamente, elevando al cuadrado el valor antes de cada nueva llamada, hasta que X quede ordenado, la Figura 5.3 describe este proceso.

```
G:=2
repeat
  Run c·nlogG pasos de S en X, asumiendo M(X) ≤ G
  G:= G^2
until: X esté ordenado
```

Figura 5.3 *Proceso de ordenamiento.*

Por la suposición, el algoritmo S logra ordenar la secuencia X cuando $G \geq M(X)$. Por lo tanto, el tiempo total gastado por el algoritmo está dado por:

$$\sum_{i=0}^{\lceil \log\log M(X) \rceil} c \cdot n \log 2^{2^i} = c \cdot n \sum_{i=0}^{\lceil \log\log M(X) \rceil} 2^{2^i} \leq 4c \cdot n \log M(X)$$

lo cual es $O(C_M(n,M(X)))$, y por lo tanto es *M-óptimo* por la Definición 2.

Observación: La mayoría de las medidas de preordenamiento que aparecen en la literatura como Inv y Runs, satisfacen la suposición acerca de la función *f(n,k)* hecha en el Lema 2.

Se nota que el método de raíz cuadrada repetida, aplicada en la prueba del Lema 2, no es nuevo cuando se diseñan algoritmos de ordenamiento adaptivos. También, el algoritmo resultante es principalmente de interés teórico como el factor constante implícito que se obtiene multiplicando por cuatro en el peor caso. La contribución del lema está en simplificar la prueba de la existencia de un algoritmo óptimo.

5.2.2 **Merge Sort Multiforma.**

El algoritmo *Merge Sort Multiforma* es adaptivo con respecto a la medida *Runs*. Dada una secuencia *X* de tamaño *n*, inicia encontrando una corrida ascendente en *X* en tiempo lineal. Estas corridas entonces constituyen la estructura de datos *S(X)*. Una versión del algoritmo se describe en la Figura 5.4.

```
procedure Multiway Merge Sort(X:sequence)
  Encontrar las corridas en X
  Construir un heap que consista del elemento máximo de cada corrida
  for i:=1 to n do
    Extraer el elemento máximo del heap
    if  (la corrida del elemento extraído no es vacía) then
      Recuperar el siguiente elemento de la corrida
      Incertar el elemento en el heap
    endif
  endfor
end
```

Figura 5.4 *Multiway Merge Sort*

Para el análisis, sea *Runs(X)* = *r*. El heap inicialmente contiene *r* elementos y cada inserción es precedida por una extracción. Cada operación heap toma tiempo *O(log r)*, por lo que finalmente el algoritmo corre en tiempo *O(nlogr)*. Además, Mannila probó que este algoritmo es óptimo con respecto a *Runs*.

Aunque no se observó por otras investigaciones, el algoritmo *Multiway Merge Sort* es óptimo con respecto a varias otras medidas de preordenamiento. La observación clave es que el algoritmo es capaz de explotar varias formas adaptivas inherentes de la estructura de datos heap, lo cual es formalizado de manera explícita en el siguiente lema.

LEMA 3. Sea H un heap con elementos tomados de una secuencia X, y sea x_i el elemento máximo en H. Entonces, una extracción de x_i seguida por una inserción de x_{i-1} en H puede ser desarrollada en tiempo constante si x_i y x_{i-1} pertenecen al mismo bloque en X, y en tiempo $O(\log||H||)$ en otro caso[4].

Prueba. Considere una extracción del elemento máximo, seguido por una inserción. Más que reordenar el heap después de la raíz borrada e insertando el nuevo elemento en forma ascendente, hacemos la inserción descendente, iniciando de la raíz vacía y separándola hacia abajo a la posición correcta.

Sea x_i y x_{i-1} pertenecientes al mismo bloque. Entonces, por definición de bloque, cuando x_i ha sido extraído de H, x_{i-1} es el mayor de los elementos presentes en H. Consecuentemente, cuando insertamos x_{i-1} quedará en la raíz de H, y así, la extracción e inserción toman tiempo constante en total. Si x_i y x_{i+1} pertenecen a bloques diferentes, entonces x_{i+1} puede ser separado abajo a una hoja, lo cual toma tiempo $O(\log ||H||)$.

Se aplica el Lema 3 para derivar un límite superior en el algoritmo *Merge Sort Multiforma* en términos de $Block(X)$. Primero se observa que ningún bloque en la secuencia X es dividido entre dos o mas corridas, y así, poniendo $b=Block(X)$, tenemos $r \leq b$. Por el Lema 3, solamente el primero y último elemento en un bloque puede causar más que tiempo constante en el algoritmo *Multiway Merge Sort*. Por lo tanto, el tiempo de complejidad del algoritmo *Multiway Merge Sort* es $O(n+blogr)=O(n+blogb)$. Carlsson, Levcopoulos y Petersson[4] probaron que $C_{Block}(n,k)=\Omega(n+klogk)$. Por lo tanto se ha probado que el algoritmo *Multiway Merge Sort* es óptimo con respecto a las medidas *Runs* y *Block*.

Para apreciar la adaptabilidad del algoritmo *Multiway Merge Sort* se consideran otras dos medidas de preordenamiento comúnmente usadas: *Rem* y *Exc*. La medida de preordenamiento *Rem* indica que el número mínimo de elementos que tienen que ser removidos de una secuencia para dejar una secuencia ordenada. La medida *Exc* indica el número mínimo de intercambios de

elementos arbitrarios necesitados para traer una secuencia en forma ordenada. Puede ser mostrado que cada algoritmo de ordenamiento *Block-óptimo* es *Rem-óptimo*, y similarmente, que cada algoritmo de ordenamiento *Rem-optimal* es *Exc-óptimo*, los resultados no son verdad a la inversa.

Se resume la discusión anterior en el siguiente teorema:

TEOREMA 4. El algoritmo *Multiway Merge Sort* es óptimo con respecto a las medidas de preordenamiento *Runs, Block, Rem* y *Exc*[4].

El algoritmo *Multiway Merge Sort* es notablemente simple comparado con otros algoritmos de ordenamiento *Block-óptimo*, como los algoritmos *Adaptive Merge Sort* y *Local Insertion Sort*.

Finalmente, se enfatiza que la técnica de utilizar la adaptabilidad de un heap puede ser aplicada para hacer también otros algoritmos de ordenamiento adaptivos *Block-óptimo*. Por ejemplo *Melsort* es un algoritmo que primero revisa la entrada y construye un conjunto ordenado de listas ordenadas. Las listas son llamadas listas sobrepasadas, y cada bloque en la secuencia de entrada es también un bloque en una lista única. Las listas son entonces mezcladas para formar la salida ordenada. El primer elemento en cada bloque cuesta más que tiempo constante para procesarlo durante la fase de construcción. Aplicando *Multiway Merge Sort* para mezclar las listas sobrepasadas entonces resulta un *Melsort Block-óptimo*.

5.2.3 **Heapsort Adaptivo.**

El algoritmo *Heapsort Adaptivo* es, como su nombre lo indica, otro algoritmo de ordenamiento basado en heaps. Es adaptivo con respecto a la oscilación dentro de la entrada, es decir con respecto a la medida *Osc(X)*, oscilaciones.

El Heapsort Adaptivo comienza construyendo el árbol Cartesiano para la secuencia de entrada y gasta tiempo lineal. El árbol cartesiano para la secuencia X de longitud n, denotada $C(X)$, es el árbol binario con raíz $x_i = max\{x_1,...,x_n\}$. Su subárbol izquierdo es el árbol Cartesiano para $<x_1,...,x_{i-1}>$ y su subárbol derecho es al árbol Cartesiano para $<x_{i+1},...x_n>$. El árbol cartesiano para la secuencia de longitud cero es el árbol binario vacío. El árbol Cartesiano es entonces usado para mantener el heap durante el ordenamiento, Figura 5.5.

Sea $Osc_i(X) = ||\{j \mid 1 \leq j < n \ \ y \ min\{x_j, x_{j+1}\} < x_i < max\{x_j, x_{j+1}\}\}||$, es decir las x_i's contribuyen a $Osc(X)$. Levcopoulos y Peterson probaron que, en el momento que x_i es extraído del heap, este contiene $O(Osc_i(X))$ elementos. Además, el nuevo max-candidato puede ser encontrado en tiempo constante, siguiendo los apuntadores del elemento extraído en $C(X)$. El árbol Cartesiano puede ser calculado en tiempo lineal, tenemos así que el algoritmo *Heapsort Adaptivo* corre en tiempo $O(n + \sum_{i=1}^{n} logOsc_i(X)) = O(nlog(Osc(X)/n))$. Levcopoulos y Peterson[4] probaron que $C_{Osc}(n,k) = \Omega(nlog(k/n))$, y por lo tanto, *Heapsort Adaptivo* es *Osc-óptimo.*

```
Procedure Heapsort Adaptivo (X:secuencia )
 Construir el árbol Cartesiano C(X)
 Insertar la raiz de C(X) en un heap
 For i=1 to n do
   Extraer el elemeto máximo del heap
   If (el elemento extraído tiene al menos un hijo en C(X)) then
    Recuperar los hijos de C(X)
    Insertar los hijos en el heap
   Endif
 EndFor
End
```

Figura 5.5 *Heapsort Adaptivo*

Para entender la adaptabilidad de *Heapsort Adaptivo*, Levcopoulos y Peterson[4] investigaron como la medida *Osc* se relaciona con otras medidas de preordenamiento. Ellos probaron que cada algoritmo de ordenamiento *Osc-óptimo* es también óptimo con respecto a las medidas *Inv* y *Runs*, y similarmente que cada algoritmo de ordenamiento Inv-optimo es *Max-optimo*. Los resultados no son verdad en sentido contrario.

Finalmente, se esboza a continuación una prueba de que *Heapsort Adaptivo* puede ser implementado para ser *Block-óptimo* para esto se utiliza el Lema 3. Por la definición de árbol Cartesiano, cada Bloque en la entrada da lugar a una unión en el árbol, en el cual todos los nodos, excepto posiblemente los correspondientes al primero y último elemento en el bloque, solamente tienen un hijo izquierdo. Por lo tanto, por el Lema 3, solamente dos elementos por bloque toman

tiempo constante para procesar el ordenamiento. Además, dos elementos del mismo bloque no pueden estar en el heap simultáneamente, porque el más pequeño de ellos no es insertado en el heap antes de que el más grande ha sido borrado, esto por definición de árbol Cartesiano. Por lo tanto, la cardinalidad del heap no puede exceder al número de bloques en la secuencia de entrada. Así *Heapsort Adaptivo* corre en tiempo $O(n+blogb)$, el cual es óptimo con respecto a *Block*.

Lo anterior se resume en el siguiente resultado:

TEOREMA 5. El algoritmo *Heapsort Adaptivo* es óptimo con respecto a las medidas de preorden *Osc*, *Inv*, *Runs*, *Max*, *Block*, *Rem* y *Exc*.[4]

CAPÍTULO 6. EVIDENCIAS EMPÍRICAS

INTRODUCCIÓN

Se implantaron los algoritmos clásicos y los adaptivos para ilustrar con evidencias empíricas los resultados teóricos presentados en este trabajo.

Ya que nos interesa ver el comportamiento de los algoritmos cuando la lista está ordenada o no, para saber si se toma ventaja de este hecho, hemos realizado experimentos con los diferentes tipos de secuencia:

1. Listas ordenadas ascendentemente. Sea X una secuencia entonces $x_i<x_j$ para $i<j$ y $x \in X$.
2. Listas ordenadas descendentemente Sea X una secuencia entonces $x_i>x_j$ para $i>j$ y $x \in X$.
3. Listas aleatorias obtenidas bajo distribución uniforme.
4. Listas en forma *zig-zag-d.*
5. Listas en forma de bloques invertidos.

Antes de empezar a mostrar los resultados empíricos definiremos las secuencias en forma zig-zag-d y las secuencias en forma de bloques invertidos.

6.1 LISTAS EN FORMA zig-zag-*d*

Definición. Sea L una secuencia ordenada en forma ascendente, entonces: Dividimos a L en bloques bi,Bi cuyo tamaño es exactamente d

$L: b_1, b_2, \ldots b_k, V, B_k, \ldots, B_2, B_1$

Se tiene que $k = n \; div \; (2d)$ donde $|V| = n - 2*d*k$

Se reorganiza L de la siguiente forma: $L': V, b_k B_k \ldots b_2 B_2 b_1 B_1$

Diremos que L' esta organizada, en bloques ordenados, en la forma *zig-zag-d*, Figura 6.1 y Figura 6.2.

Ejemplo: n=20

d=1 k=10

L :: 1.2.3.4.5.6.7.8.9.10.:11:12:13:14:15:16:17:18:19:20

L':: 10 11 9 12 8 13 7 14 6 15 5 16 4 17 3 18 2 19 1 20

d=2 k=5

L :: 1 2.3 4.5 6.7 8.9 10.:11 12:13 14:15 16:17 18:19 20

L':: 9 10-11 12-7 8-13 14-5 6-15 16-3 4-17 18-1 2-19 20

d=3 k=3

L :: 1 2 3. 4 5 6. 7 8 9. 10 11 :12 13 14 :15 16 17 :18 19 20

L':: 10 11- 7 8 9- 12 13 14- 4 5 6- 15 16 17- 1 2 3- 18 19 20

d=4 k=2

L :: 1 2 3 4. 5 6 7 8. 9 10 11 12 :13 14 15 16 :17 18 19 20

L':: 9 10 11 12- 5 6 7 8- 13 14 15 16- 1 2 3 4- 17 18 19 20

d=5 k=2

L :: 1 2 3 4 5. 6 7 8 9 10. :11 12 13 14 15 :16 17 18 19 20

L':: 6 7 8 9 10- 11 12 13 14 15- 1 2 3 4 5- 16 17 18 19 20

d=6 k=1

L :: 1 2 3 4 5 6. 7 8 9 10 11 12 13 14 :15 16 17 18 19 20

L':: 7 8 9 10 11 12 13 14- 1 2 3 4 5 6- 15 16 17 18 19 20

d=8 k=1

L :: 1 2 3 4 5 6 7 8. 9 10 11 12 :13 14 15 16 17 18 19 20

L':: 9 10 11 12- 1 2 3 4 5 6 7 8- 13 14 15 16 17 18 19 20

d=10 k=1

L: 1 2 3 4 5 6 7 8 9 10.:11 12 13 14 15 16 17 18 19 20

L: 1 2 3 4 5 6 7 8 9 10- 11 12 13 14 15 16 17 18 19 20

Nótese que para las secuencias zig-zag-d con d=1 representa un gran desorden y d=$n/2$ representa la secuencia ordenada.

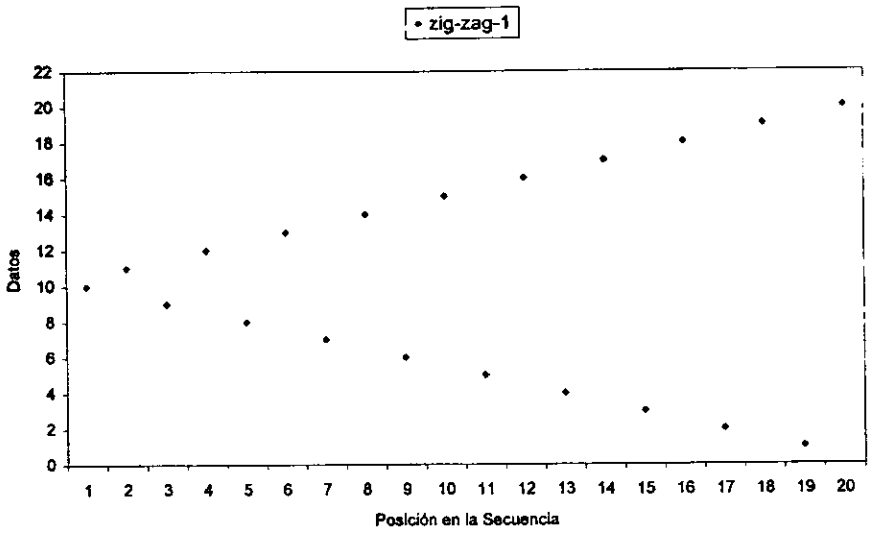


Figura 6.1 *Zig-zag-1 para n=20*

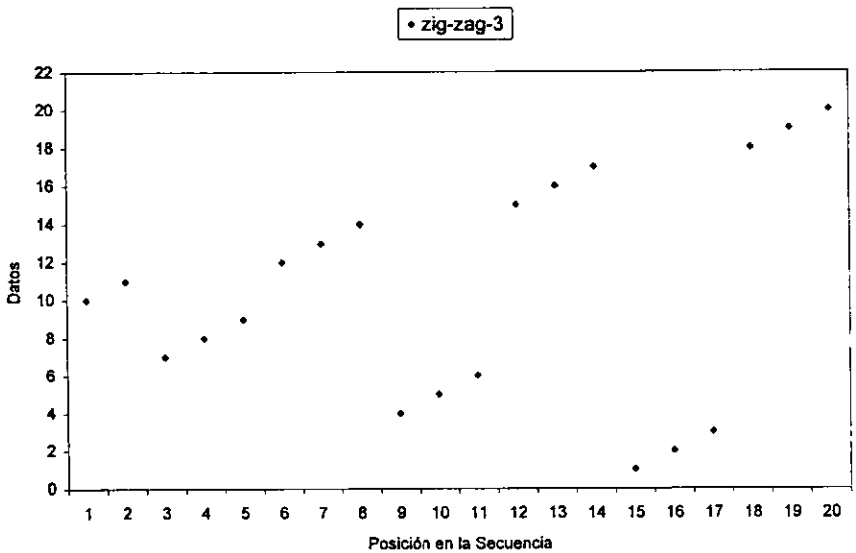


Figura 6.2 *Zig-zag-3 para n=20*

Entonces a medida que d se incrementa el desorden va desapareciendo.

Estas secuencias nos serán de gran utilidad para observar detalladamente el comportamiento de los algoritmos, sobre todo los algoritmos que son adaptivos con respecto a la medida *INV*.

6.2 LISTAS EN FORMA DE BLOQUES INVERTIDOS

Las listas de datos organizadas en forma de bloques invertidos las definimos de la siguiente manera.

Definición. Considere la secuencia $X=\{x_n,x_{n-1},...x_2,x_1\}$ con $x_n>x_{n-1}>...>x_1$

Una secuencia X_k está organizada en bloques invertidos si reacomodamos los elementos de X en la siguiente forma

$$\underset{B_1}{x_1,x_2,\ldots,x_k} \quad \underset{B_2}{x_{n-k},x_{n-(k+1)},\ldots,x_{k+1}} \quad \underset{B_3}{x_{n-(k-1)},x_{n-(k-2)},\ldots,x_n}$$

El primer bloque y el tercero tienen k elementos, el segundo bloque tiene $(n-2k)$ elementos, Figura 6.3 y Figura 6.4.

Ejemplo: Sea una secuencia X={60,59,58...,3,2,1}

Se tiene que:

X_1={1,59,58,...3,2,60}
X_2={1,2,58,57,...4,3,59,60}
X_3={1,2,3,57,56,...5,4,58,59,60}
X_4={1,2,3,4,56,...6,5,57,58,59,60}
X_5={1,2,3,4,5,55,...7,6,56,57,58,59,60}

Este tipo de secuencias la usamos para observar el comportamiento del algoritmo Rheapsort.

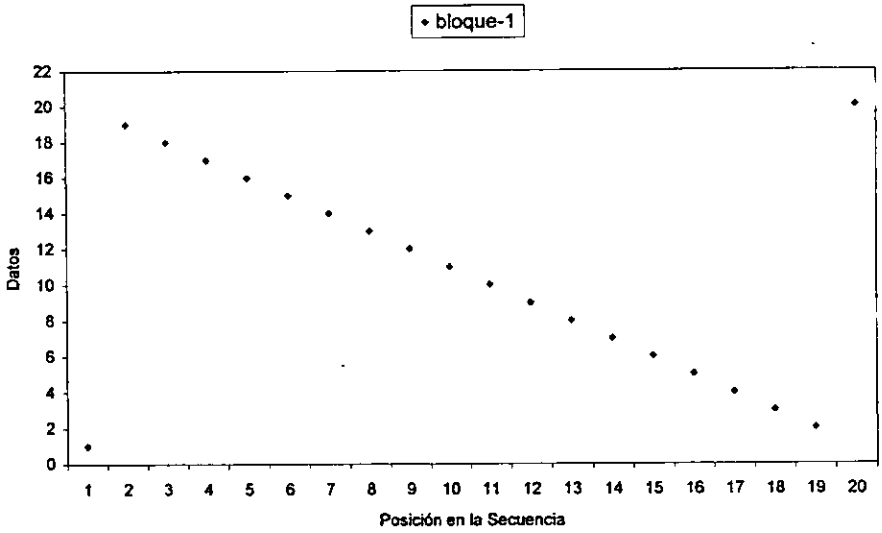


Figura 6.3 *bloque-1 para n=20*

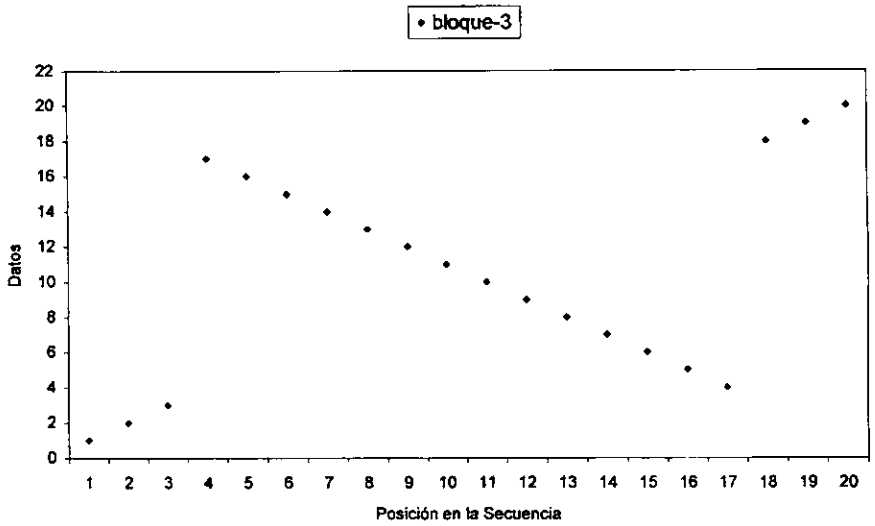


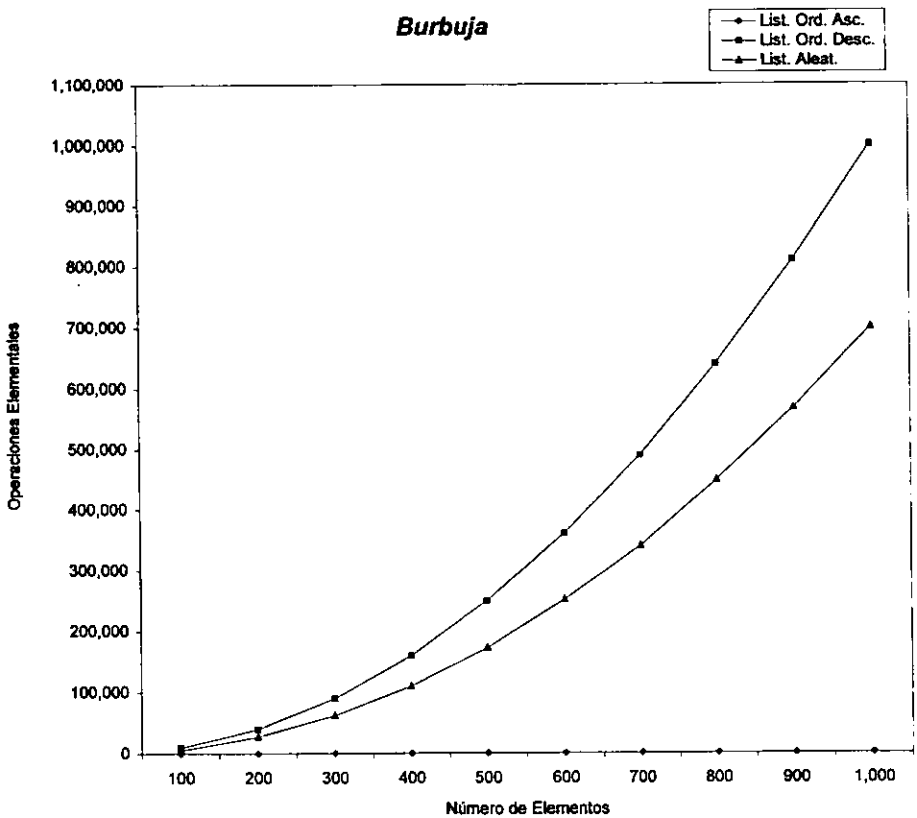
Figura 6.4 *bloque-3 para n=20*

6.3 RESULTADOS EMPÍRICOS

Presentamos los resultados de los experimentos en 2 partes. Primero una general donde observamos el comportamiento de los algoritmos para listas ordenadas en forma ascendente y descendente, y para listas obtenidas aleatoriamente bajo una distribución uniforme. En la segunda parte, mostramos los resultados de ejecutar los algoritmos con secuencias organizadas en forma zig-zag-d. Supondremos, siempre, que queremos ordenar una secuencia en forma ascendente.

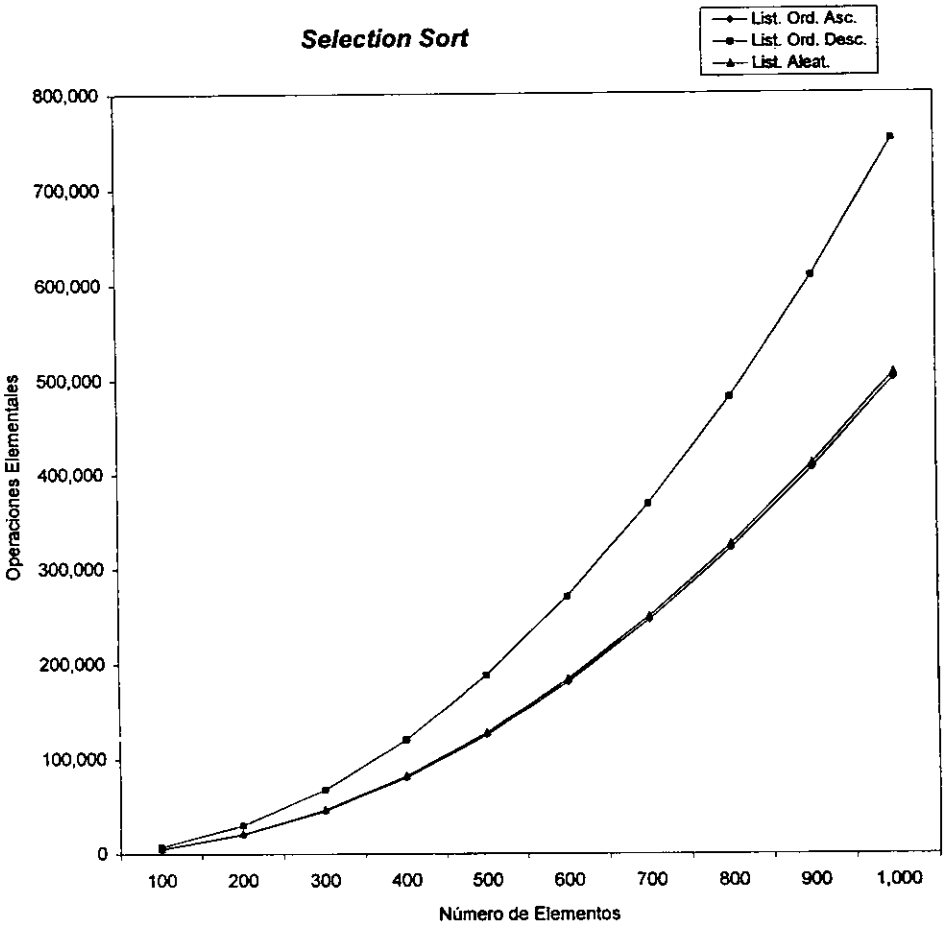
6.3.1 Comportamiento general.

Podemos observar que para los algoritmos clásicos el comportamiento en el peor caso y en el caso promedio coincide con los resultados teóricos presentados.



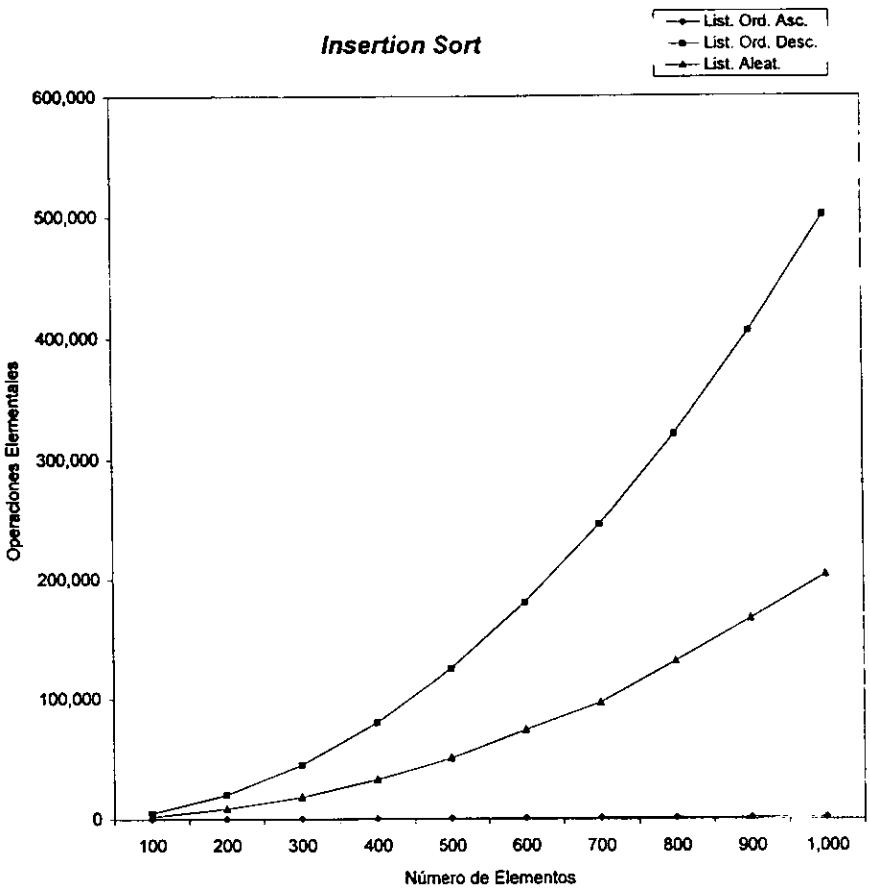
Gráfica 1. *Burbuja*

La Gráfica 1 nos indica que el algoritmo Burbuja es cuadrático en el peor caso y en el caso promedio. La versión implantada del método Burbuja resulta ser lineal sólo cuando la secuencia está ordenada



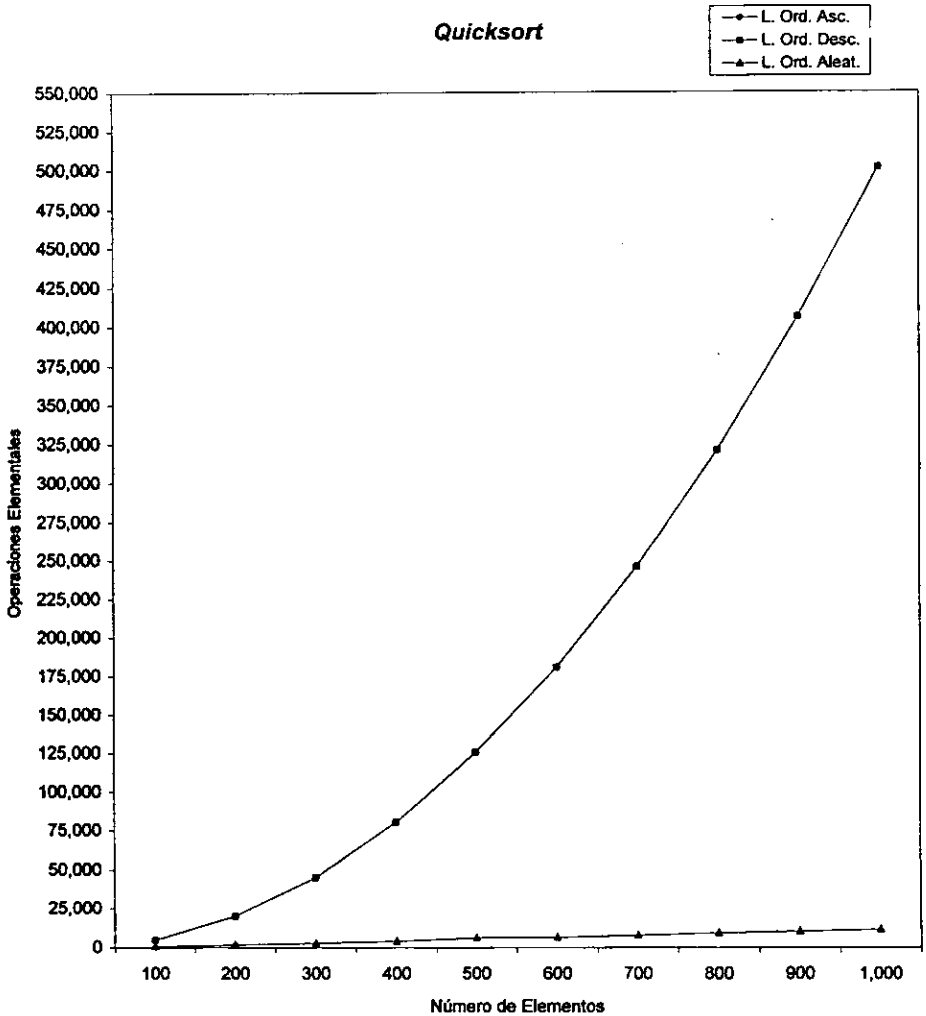
Gráfica 2. *Selection Sort*

La Gráfica 2 nos muestra que Selection Sort siempre es cuadrático.



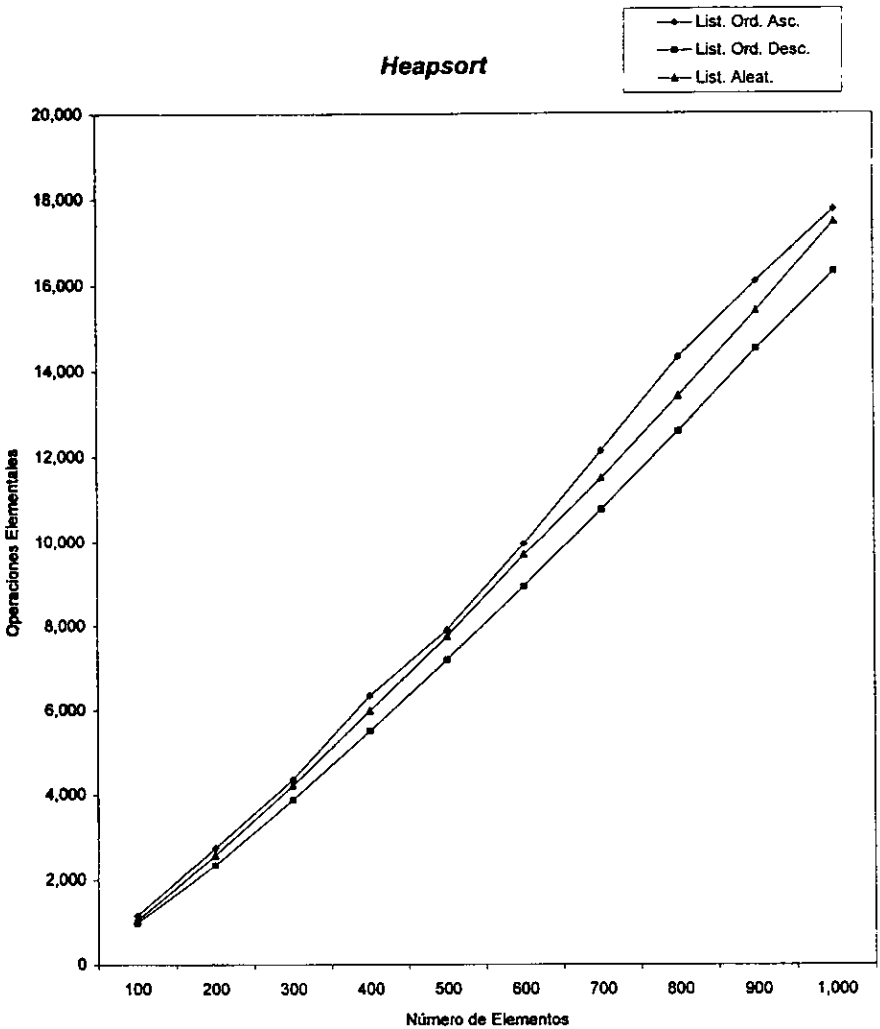
Gráfica 3. *Insertion Sort*

En la Gráfica 3 observamos que la versión implantada de Insertion Sort es cuadrática tanto en el peor caso como en el caso promedio y tiene un desempeño lineal cuando la lista ya está ordenada.



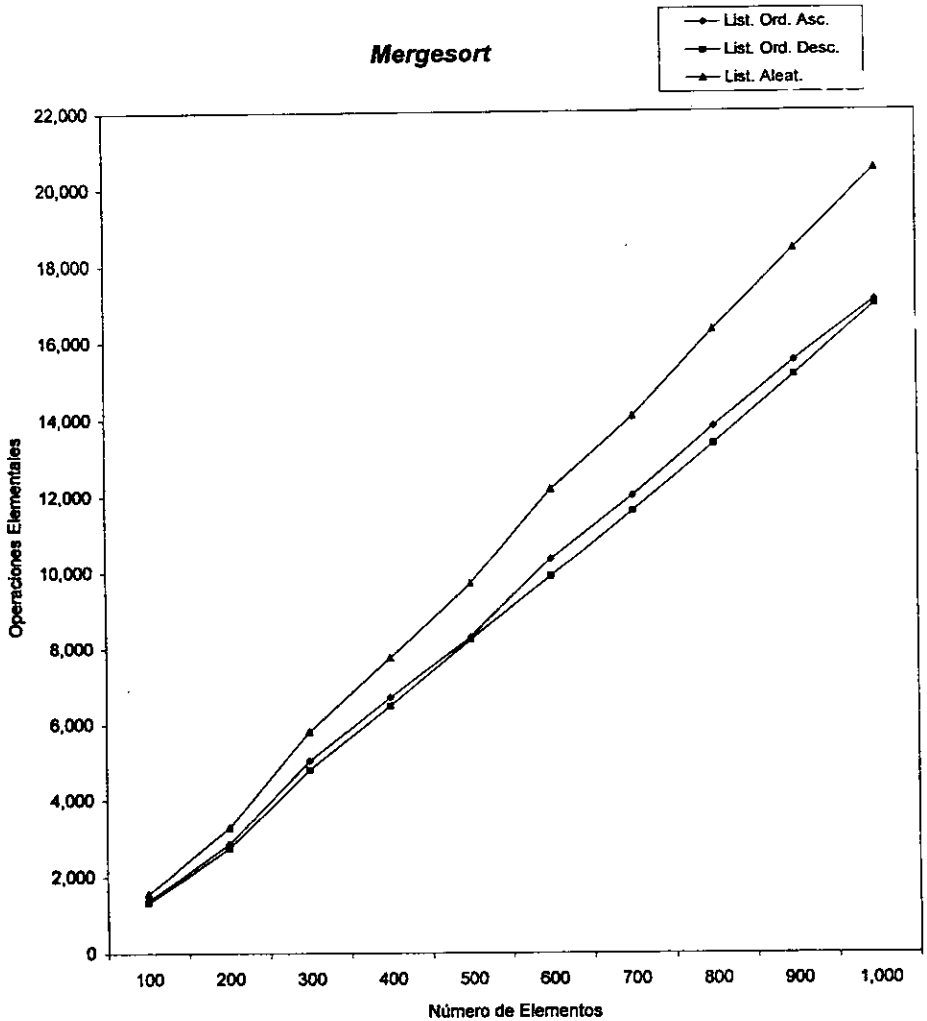
Gráfica 4. *Quicksort*

La implantación concreta de Quick Sort, que ejecutamos, elige como pivote al elemento "más a la derecha" del arreglo. Por lo que las listas ordenadas, en forma ascendente o descendente resultan ser el peor caso para esta versión de Quick Sort. Por tanto en estas su desempeño es cuadrático. Las listas obtenidas aleatoriamente, resultan tener un comportamiento aceptable, para esta versión, requieren desempeño $O(nlogn)$.



Gráfica 5. *Heapsort*

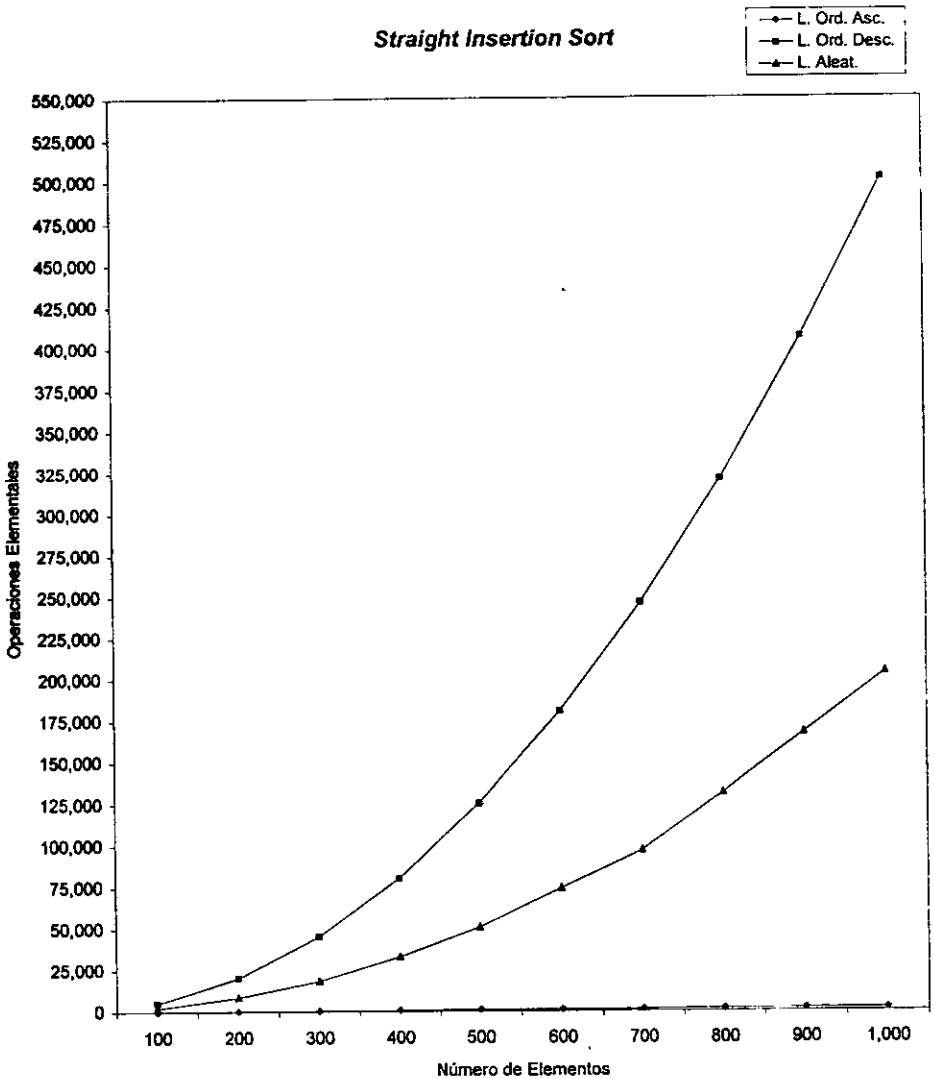
La Gráfica 5 nos ilustra que Heap Sort es siempre $O(nlogn)$.



Gráfica 6. *Mergesort*

La Gráfica 6 nos ilustra que Merge Sort es siempre $O(n\log n)$.

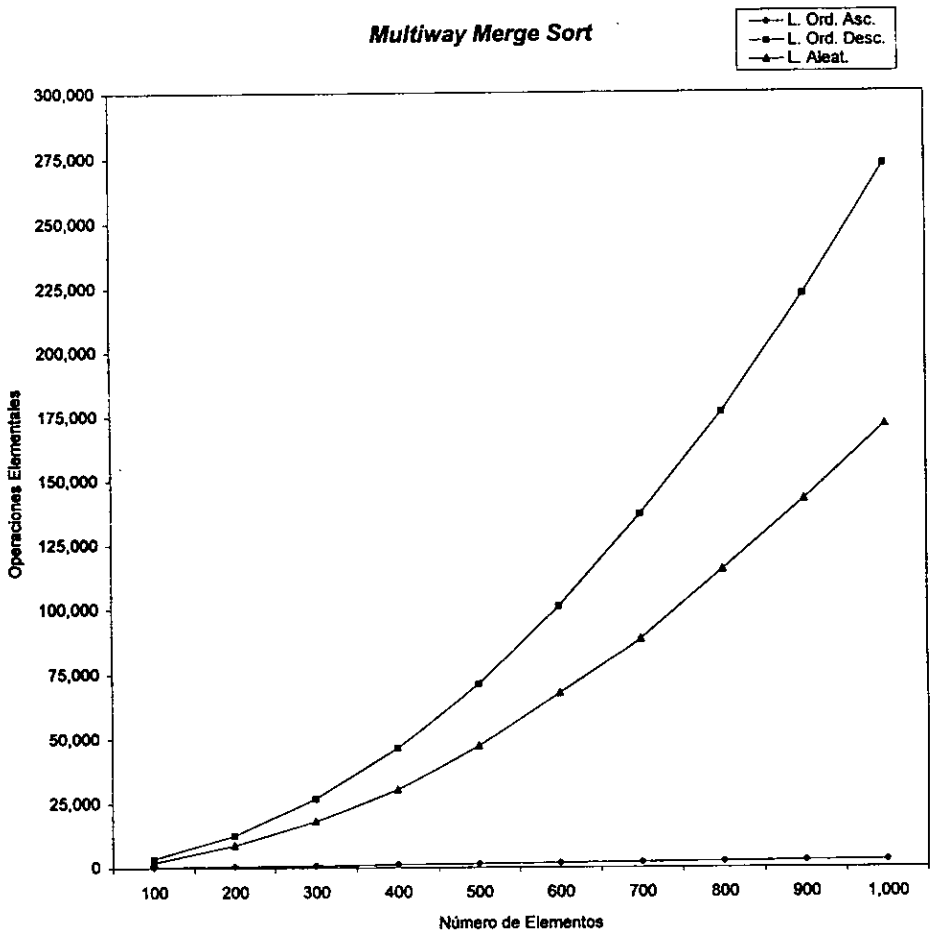
Para los algoritmos adaptivos el comportamiento es más drástico.



Gráfica 7. *Straight Insertion Sort*

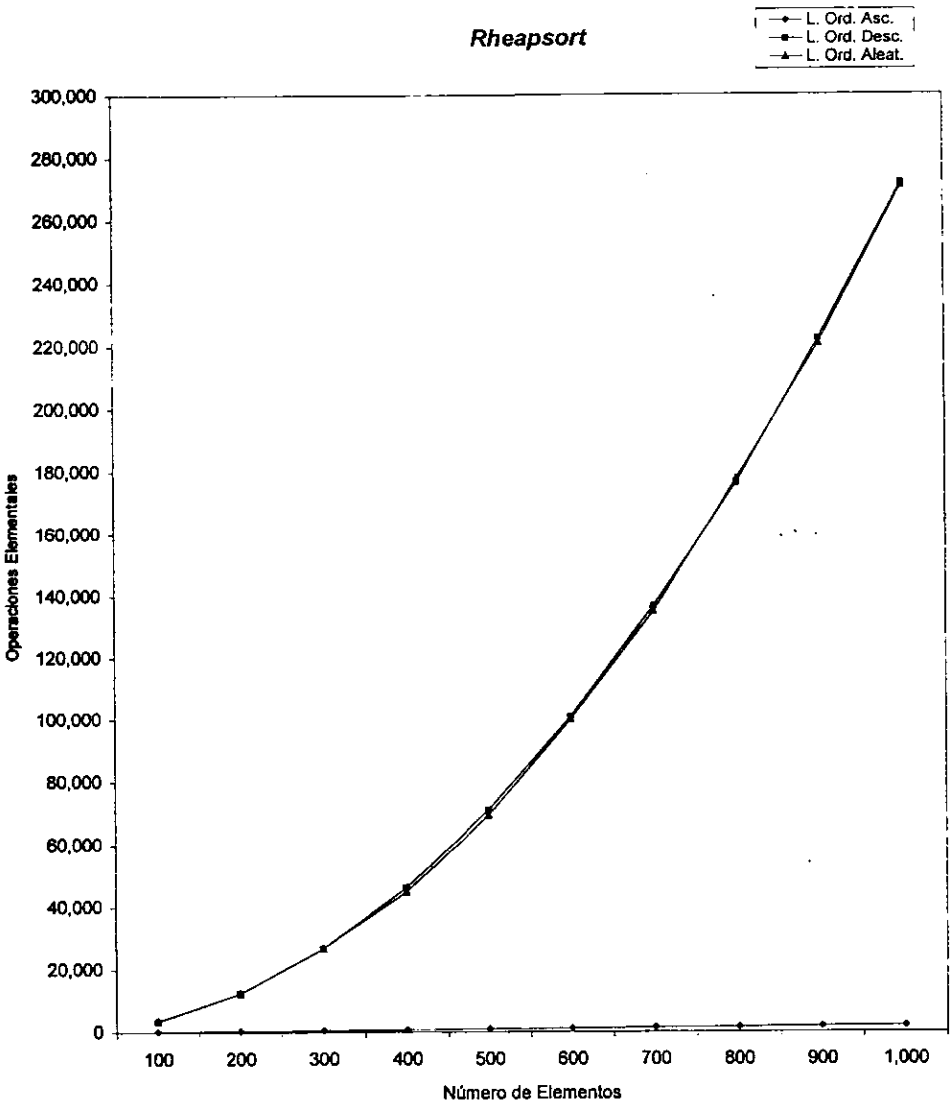
El algoritmo Straight Insertion Sort es adaptivo con respecto al número de inversiones .En una lista ordenada ascendentemente no hay inversiones, el algoritmo es lineal: $O(n)$.

En una lista ordenada en forma descendente el número de inversiones es cuadrático, igual que el desempeño del algoritmo $O(n^2+n)=O(n^2)$, Para una secuencia obtenida aleatoriamente el número de inversiones varia entre 0 y n^2. En la Gráfica 7 se ilustra el comportamiento del Algoritmo. Se observa que cuando la secuencia es obtenida aleatoriamente su desempeño está entre el peor caso y el mejor.



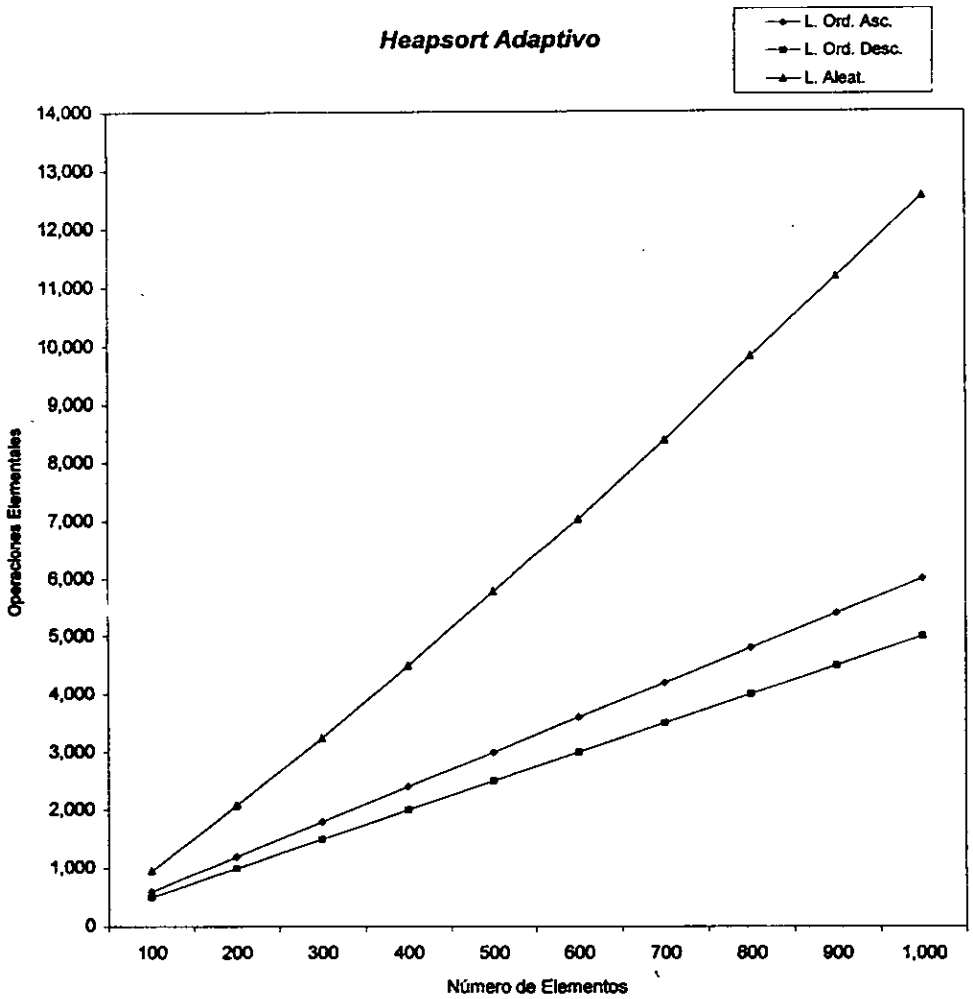
Gráfica 8. *Multiway Merge Sort*

Para el algoritmo Multiway Merge Sort el comportamiento es similar al anterior, lo podemos observar en la Gráfica 8.



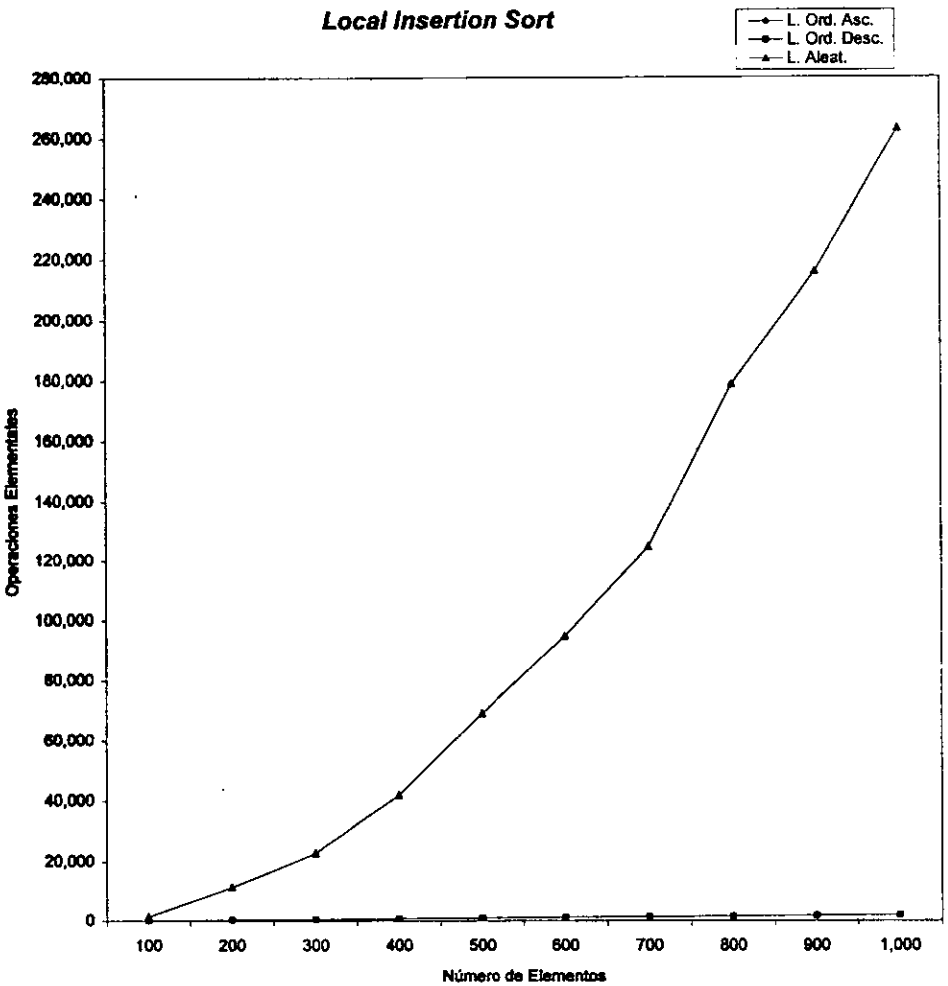
Gráfica 9. *Rheapsort*

El Algoritmo Rheapsort es adaptivo con respecto a la medida Max. La lista ordenada en forma descendente resulta ser un peor caso para este algoritmo y las listas obtenidas aleatoriamente resultaron ser ejemplares donde la medida Max resulta alta. Pero el algoritmo es lineal cuando la lista está ya ordenada. La Gráfica 9 nos muestra los resultados obtenidos.



Gráfica 10. *Heapsort Adaptivo*

Tenemos que el HeapSort Adaptivo tiene desempeño computacional $O(n+blog_2 b)$ donde b es el número de bloques. Para las secuencias ordenadas tanto ascendente como descendentemente b=1. Por lo tanto el algoritmo es lineal, esto lo podemos observar en la Gráfica 10.

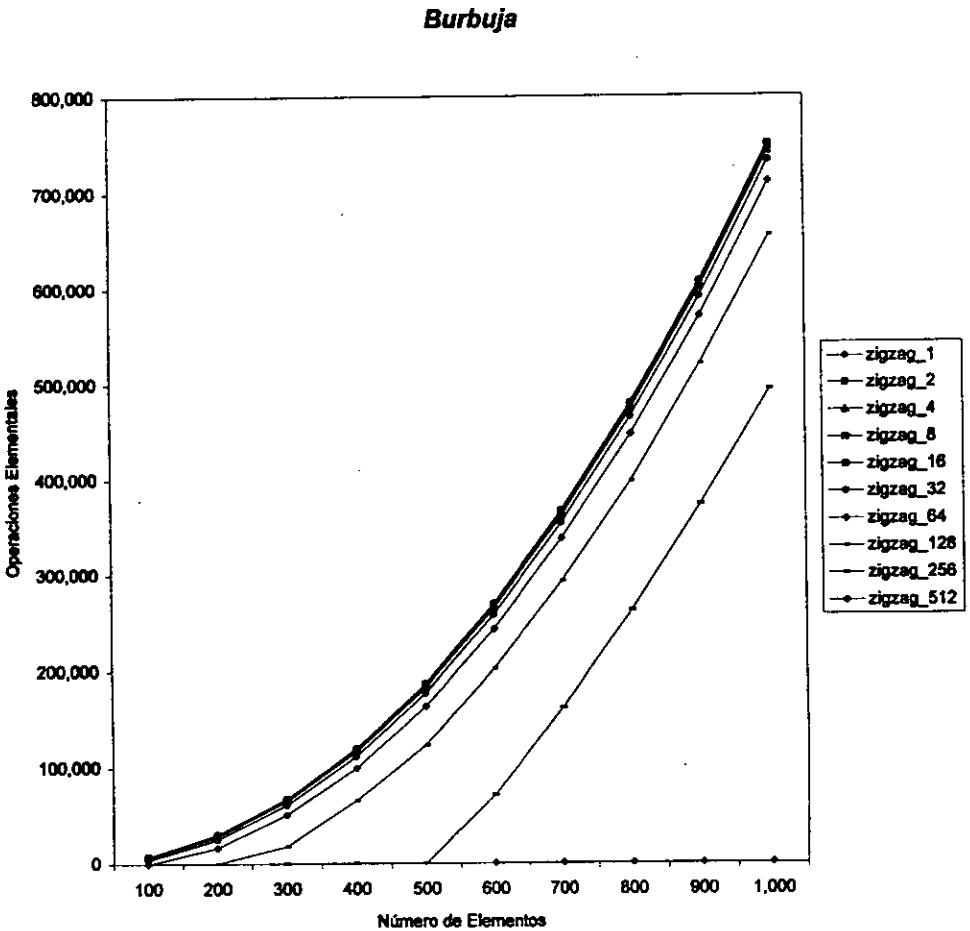


Gráfica 11. *Local Insertion Sort*

El algoritmo Local Insertion Sort es adaptivo con respecto a inversiones en ambos sentidos. Entonces para listas ordenadas en forma tanto ascendente como descendente el número de *inv* es cero, por lo cual Local Insertion Sort es lineal en estos casos. Para secuencias obtenidas aleatoriamente, el algoritmo tiene un desempeño cuadrático. La Gráfica 11 nos muestra los resultados obtenidos.

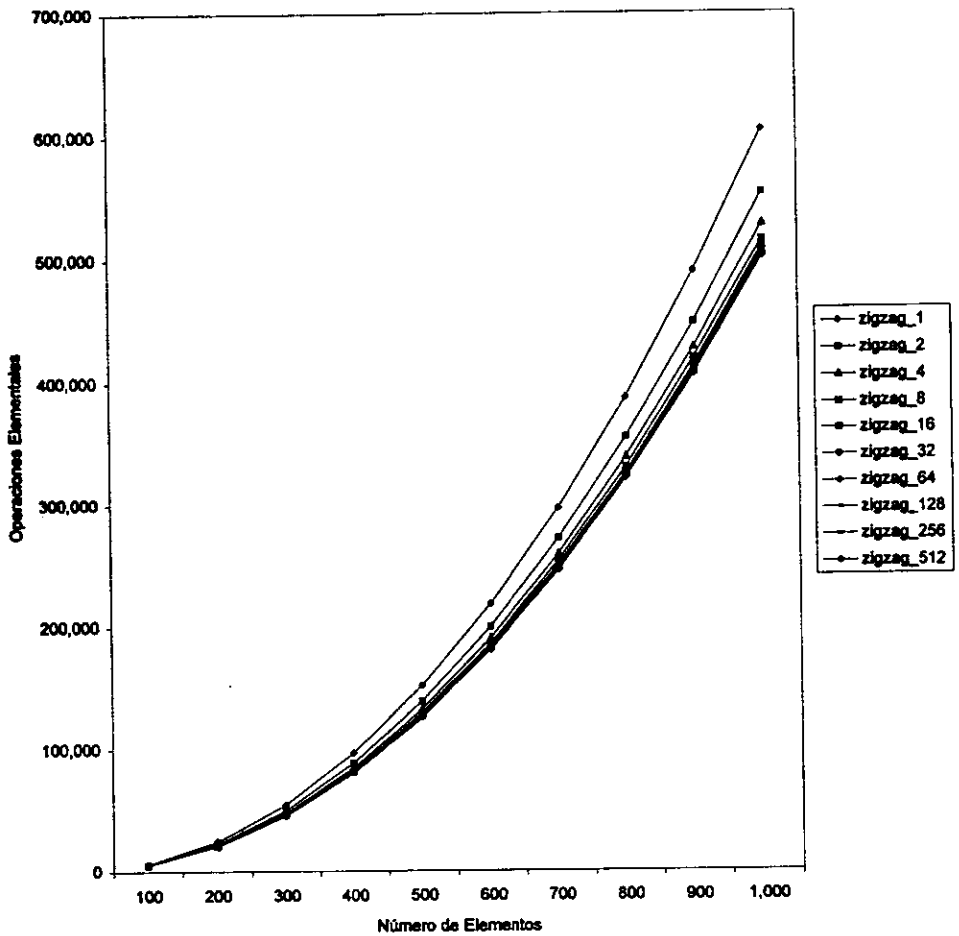
6.3.2 Comportamiento en listas zig-zag-d y bloques invertidos.

En esta sección veremos los resultados obtenidos para los algoritmos clásicos y adaptivos utilizando listas zig-zag-d y listas en forma de bloques invertidos los cuales coinciden con los resultados teóricos presentados.



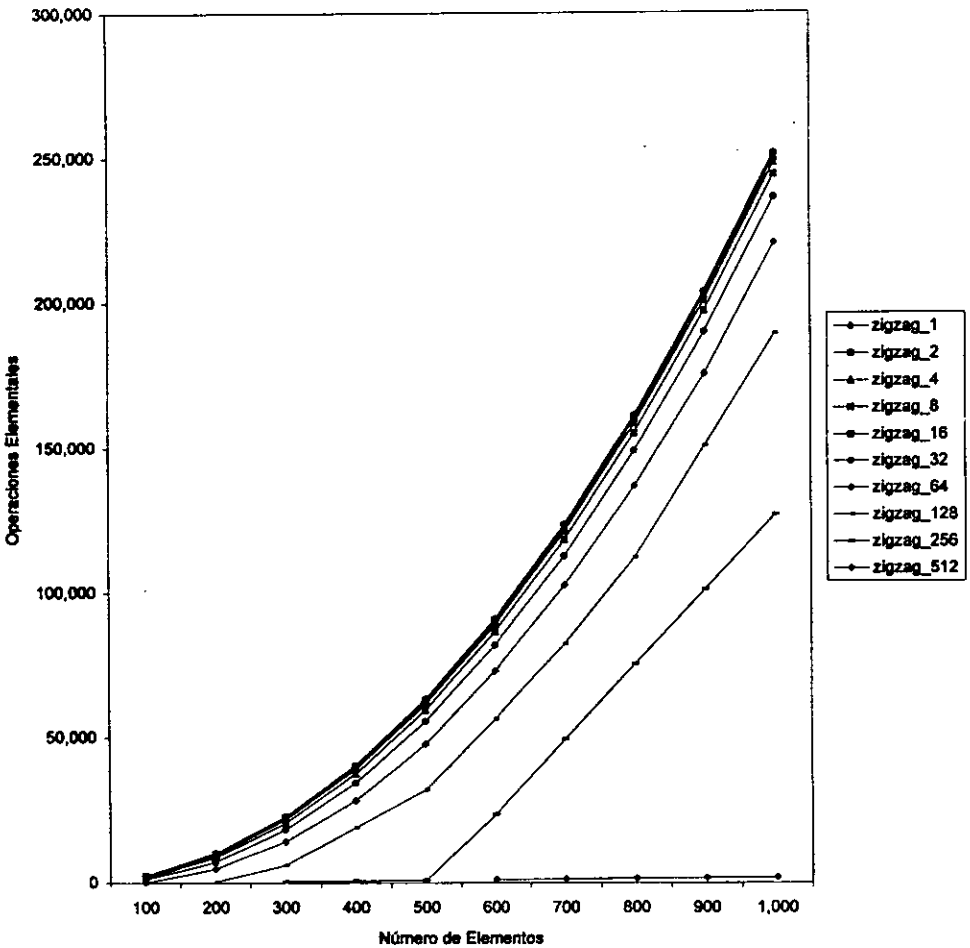
Gráfica 12. *Burbuja*

La Gráfica 12 nos indica que el algoritmo Burbuja es cuadrático independientemente del grado de desorden de las listas organizadas en forma zig-zag-d y en el caso de listas ordenadas ascendentemente y descendentemente tiene comportamiento lineal solo cuando la secuencia está ordenada.



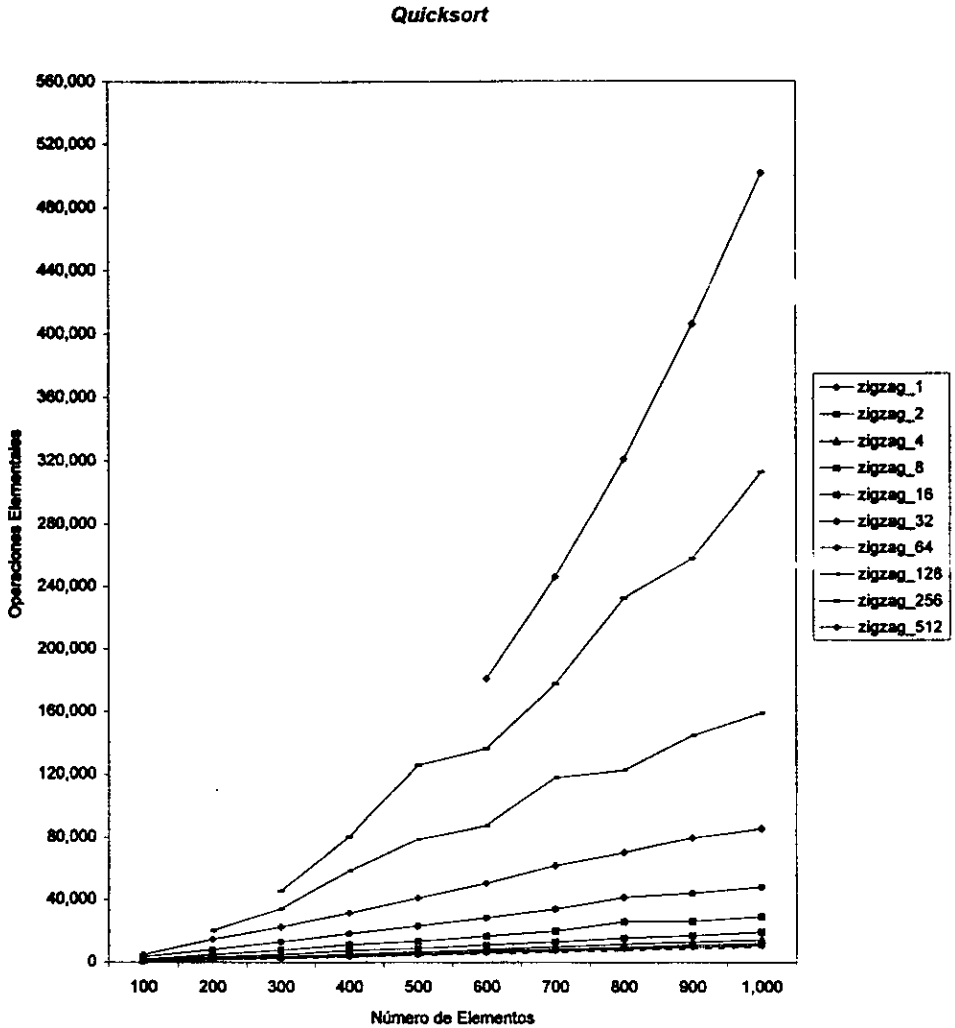
Gráfica 13. *Selection Sort*

La Gráfica 13 nos muestra que Selection Sort siempre es cuadrático.



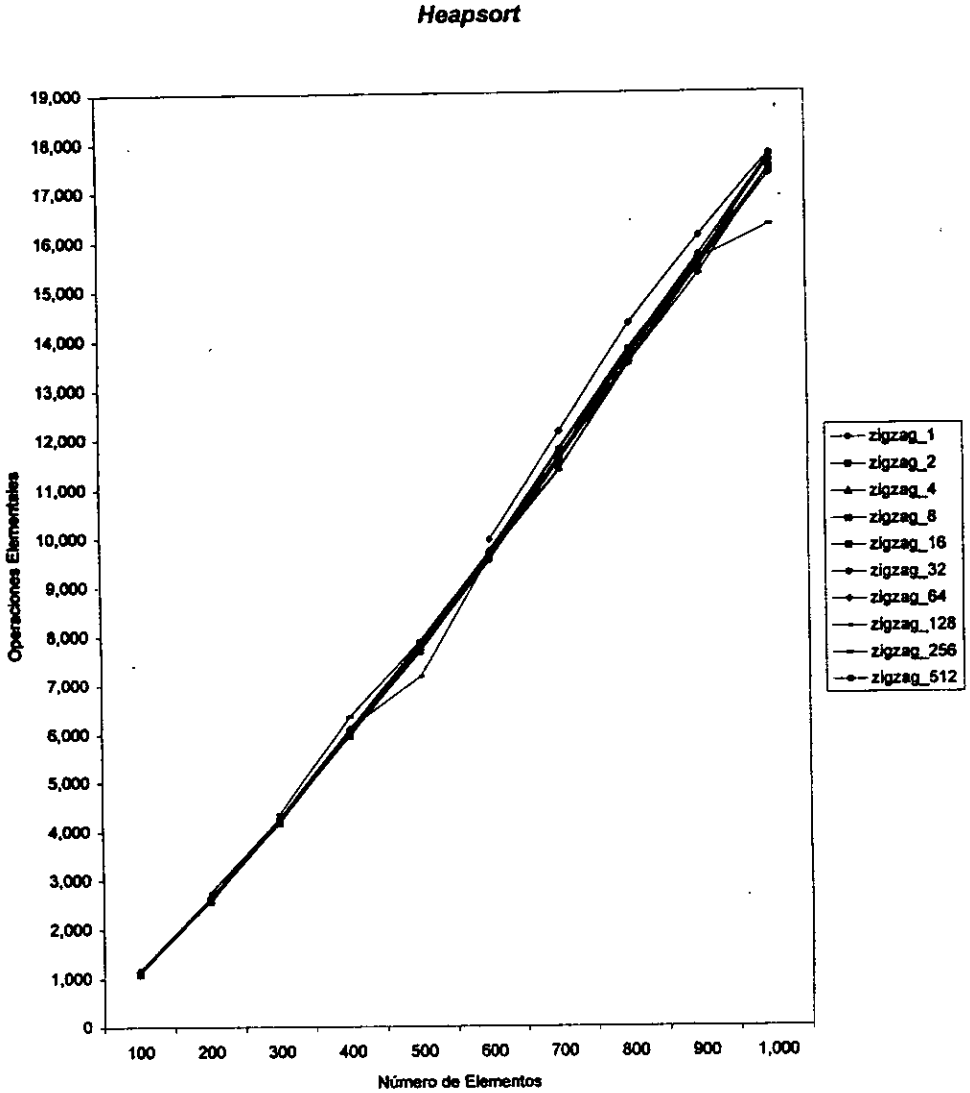
Gráfica 14. *Insertion Sort*

En la Gráfica 14 observamos que la versión implantada de Insertion Sort es cuadrática y tiene un desempeño lineal sólo cuando la lista ya está ordenada.



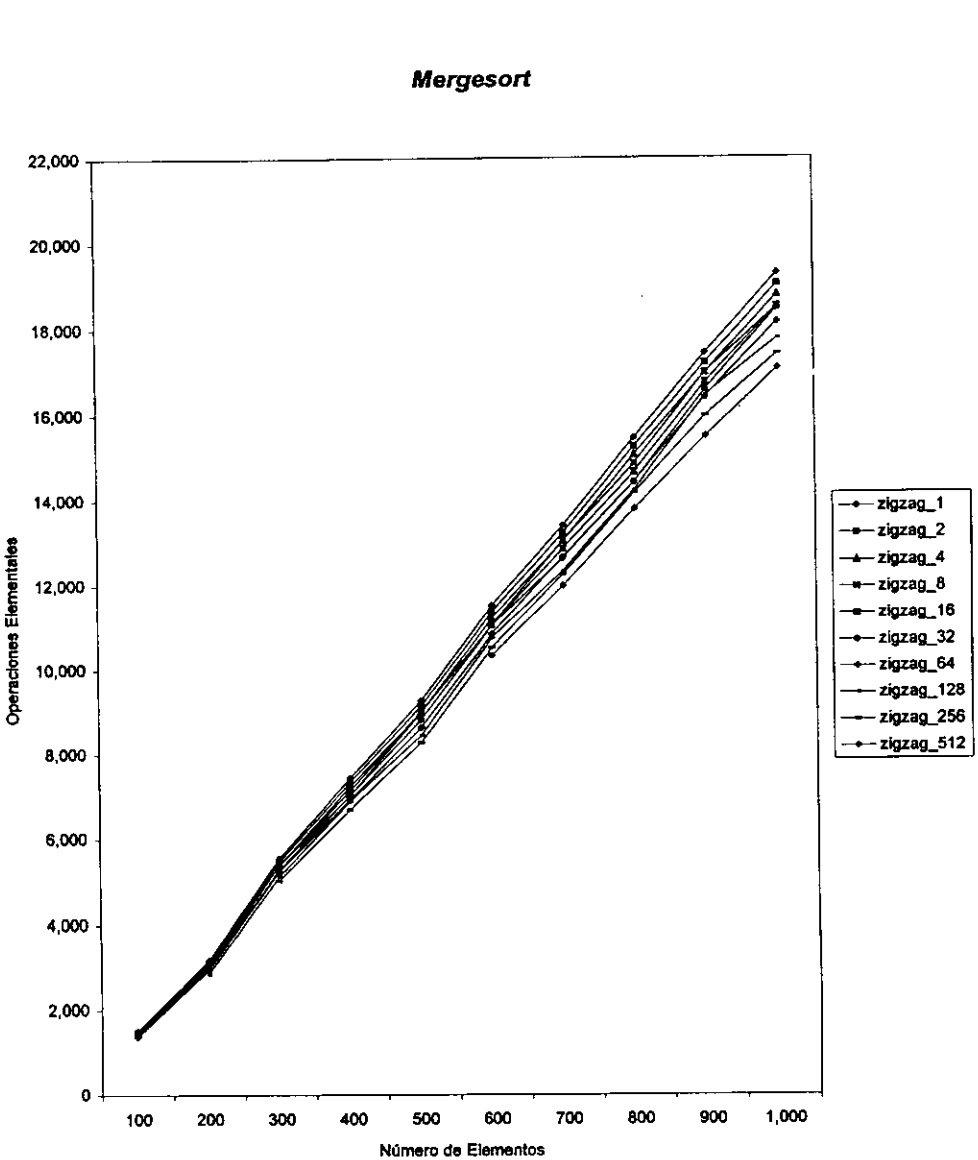
Gráfica 15. *Quicksort*

La implantación concreta de Quick Sort, que ejecutamos, elige como pivote al elemento "más a la derecha" del arreglo. Por lo que cuando las listas tienden a estar ordenadas estas resultan ser el peor caso para Quick Sort. Por tanto su desempeño es cuadrático. Las listas obtenidas aleatoriamente, resultan tener un comportamiento aceptable, y para esta versión, requieren desempeño *O(nlogn)*, ver Gráfica 15.



Gráfica 16. *Heapsort*

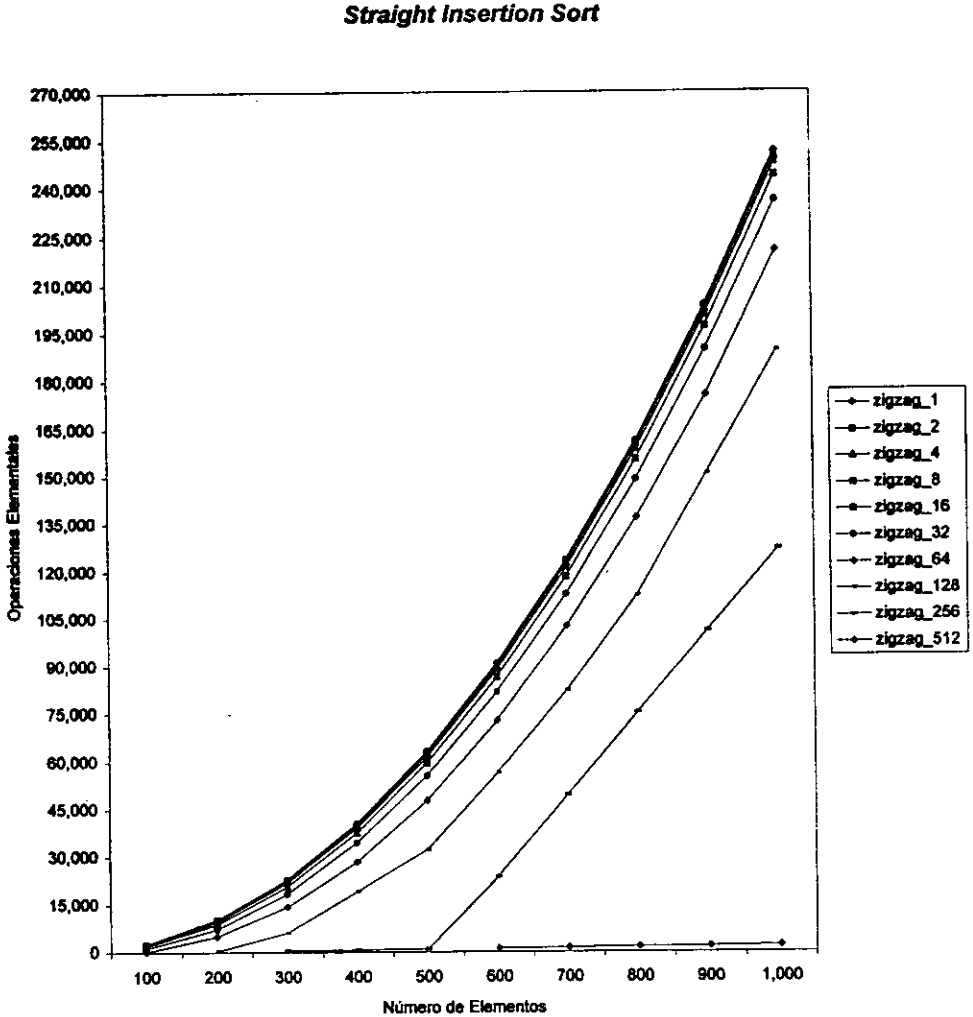
La Gráfica 16 nos ilustra más claramente que el algoritmo Heap Sort es siempre $O(n\log n)$.



Gráfica 17. *Mergesort*

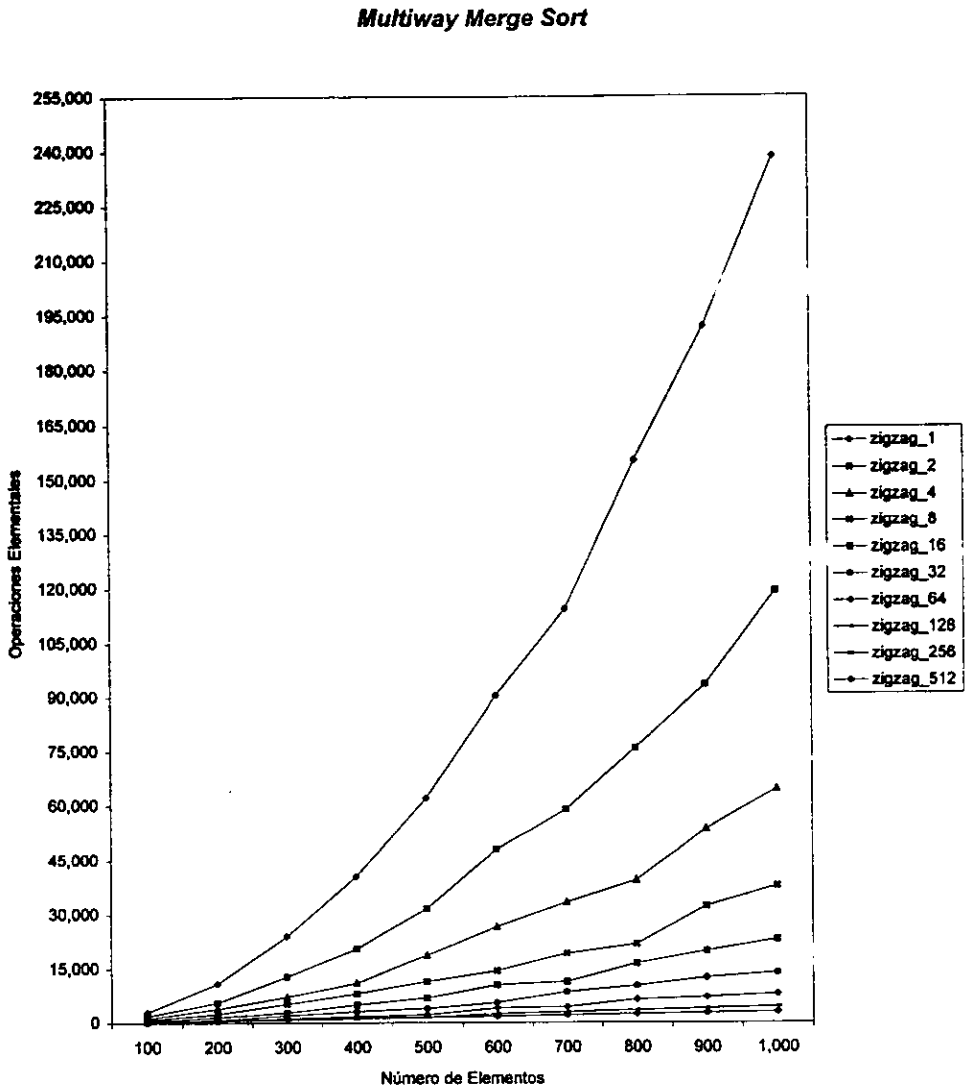
La Gráfica 17 nos ilustra más claramente que el algoritmo Merge Sort es siempre $O(nlogn)$.

Para los algoritmos adaptivos utilizando listas zig-zag-d podemos ver mas claramente como su comportamiento es más drástico.



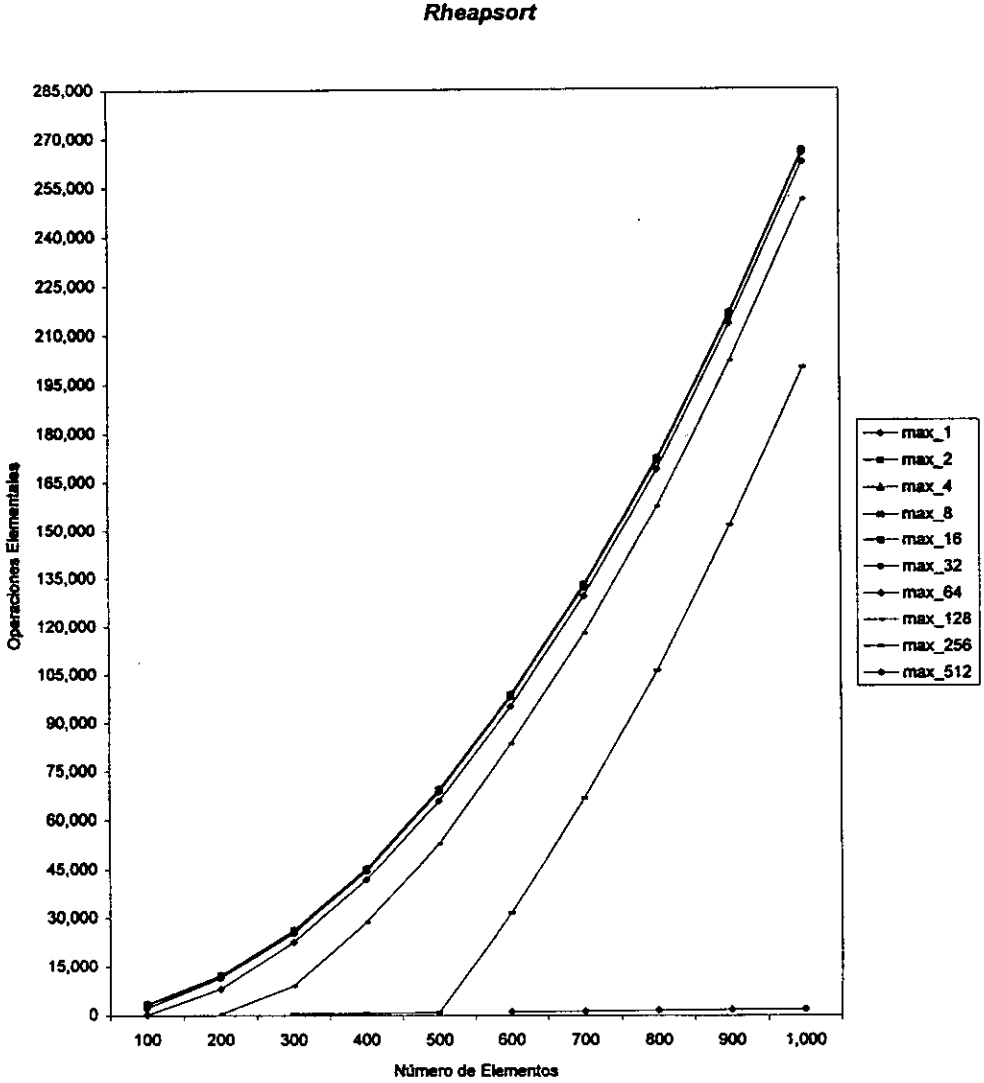
Gráfica 18. *Straight Insertion Sort*

El algoritmo Straight Insertion Sort es adaptivo con respecto al número de inversiones.Vemos que a mayor orden el comportamiento del algoritmo es lineal. Y a mayor desorden su comportamiento es cuadrático. En la Gráfica 18 se ilustra el comportamiento del algoritmo.



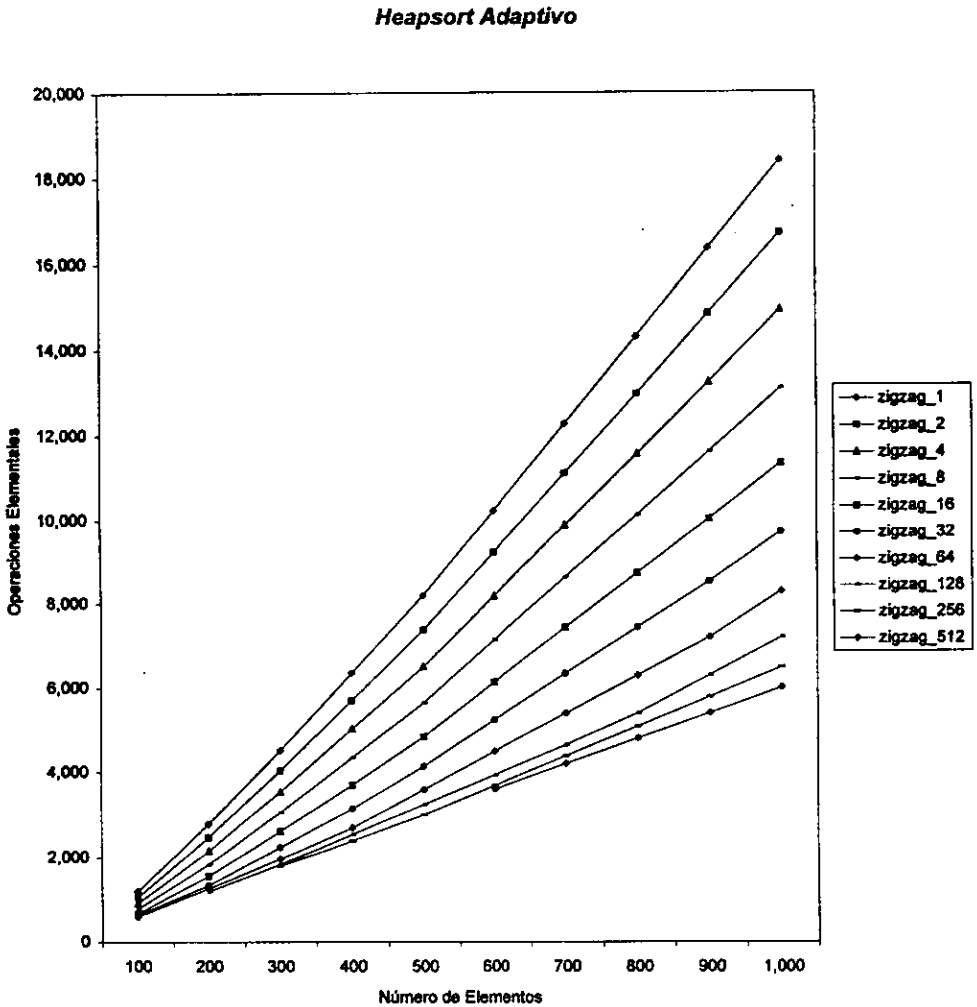
Gráfica 19. *Multiway Merge Sort*

Para el algoritmo Multiway Merge Sort utilizando listas zig-zag-d el comportamiento es similar al anterior, lo podemos observar en la Gráfica 19.



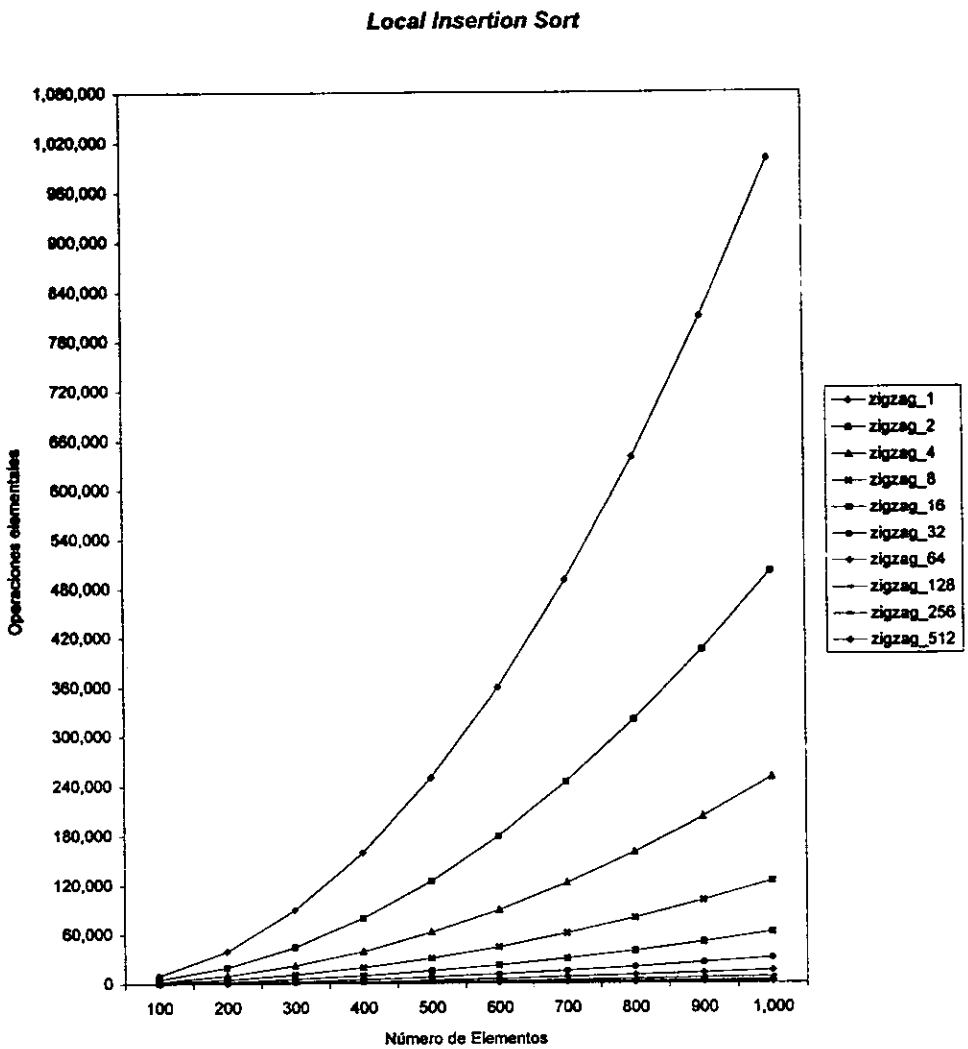
Gráfica 20. *Rheapsort*

El Algoritmo Rheapsort es adaptivo con respecto a la medida Max. A mayor desorden el algoritmo presenta un comportamiento cuadrático y el algoritmo es lineal cuando la lista está ya ordenada. La Gráfica 20 nos muestra los resultados obtenidos.



Gráfica 21. *Heapsort Adaptivo*

El algoritmo HeapSort Adaptivo tiene un comportamiento *O(nlogn)*, como se observa en la Gráfica 21.

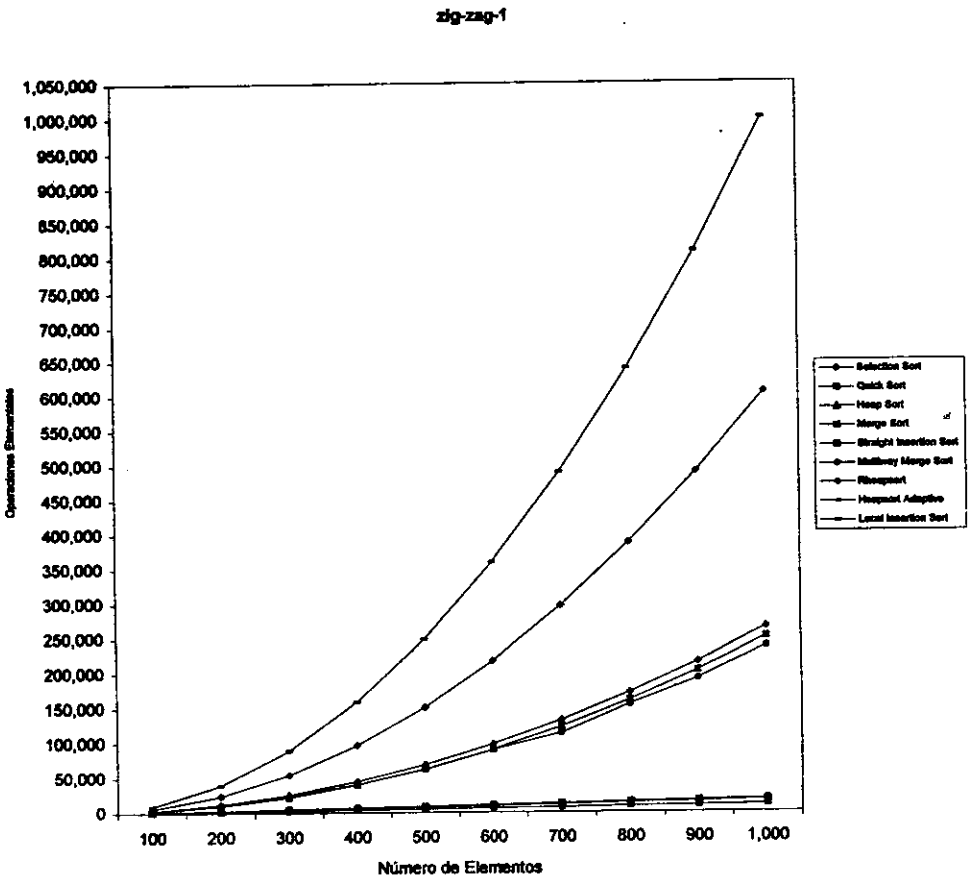


Gráfica 22. *Local Insertion Sort*

El algoritmo Local Insertion Sort tiene un comportamiento cuadrático a mayor desorden y cuando presentan orden las listas el algoritmo es lineal. La Gráfica 22 nos muestra los resultados obtenidos.

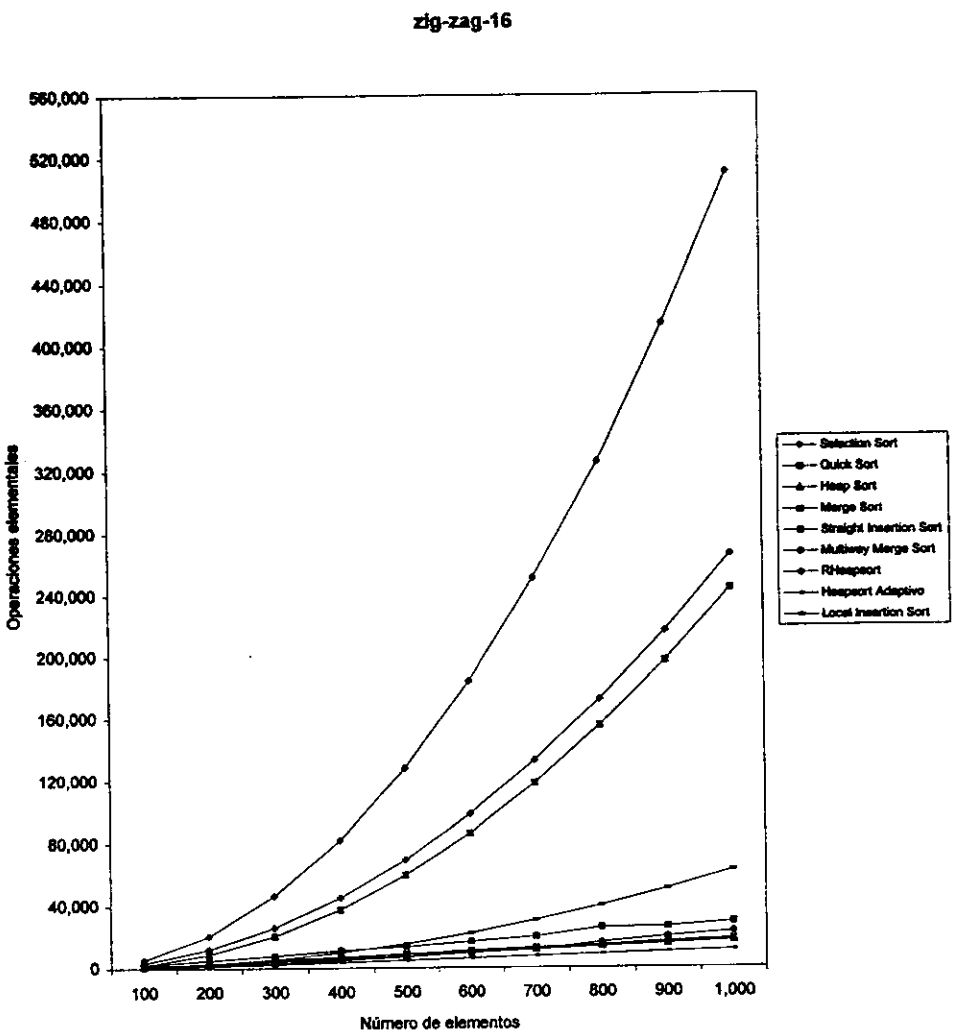
6.3.3 Comparación entre algoritmos clásicos y adaptivos.

En esta sección presentamos una comparación gráfica entre los algoritmos clásicos de ordenamiento y los algoritmos adaptivos de ordenamiento presentados en este trabajo.



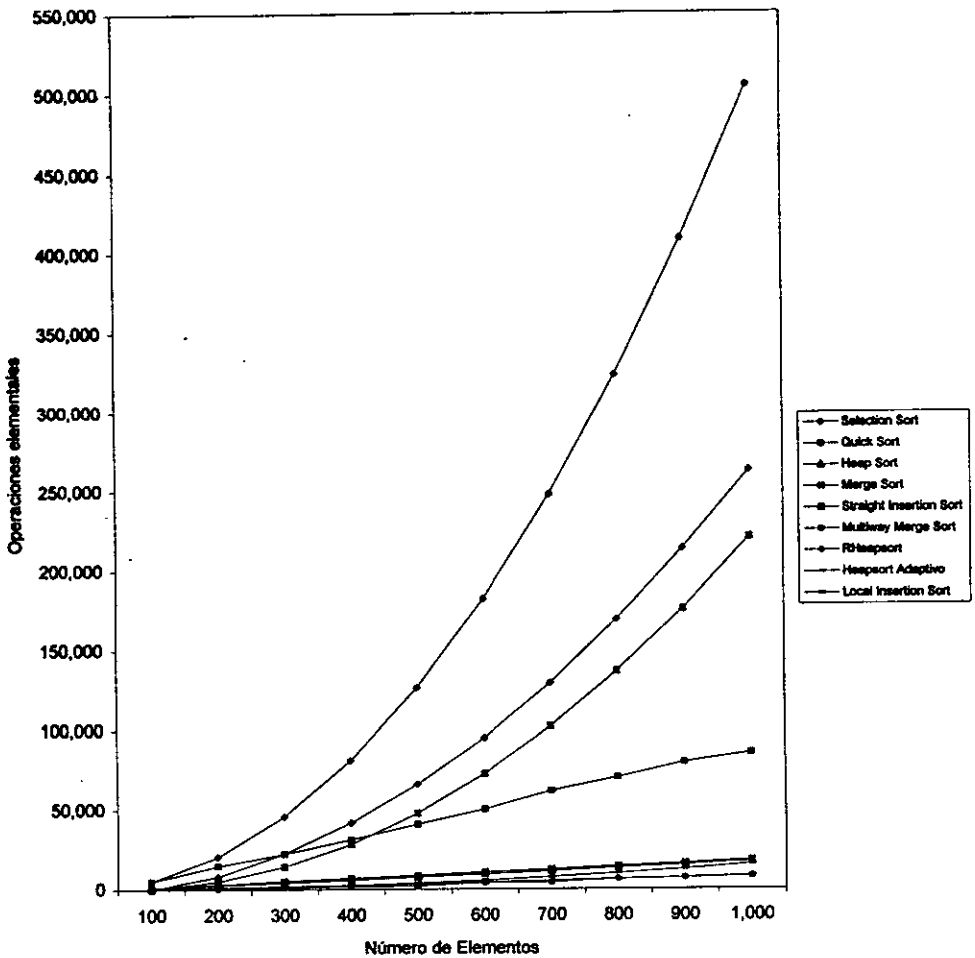
Gráfica 23. *zig-zag-1*

La Gráfica 23 nos muestra como para listas que presentan un mayor desorden, los algoritmos adaptivos de ordenamiento tienen un comportamiento como los algoritmos clásicos, aún de tiempo cuadrático como es el caso de Local Insertion Sort.



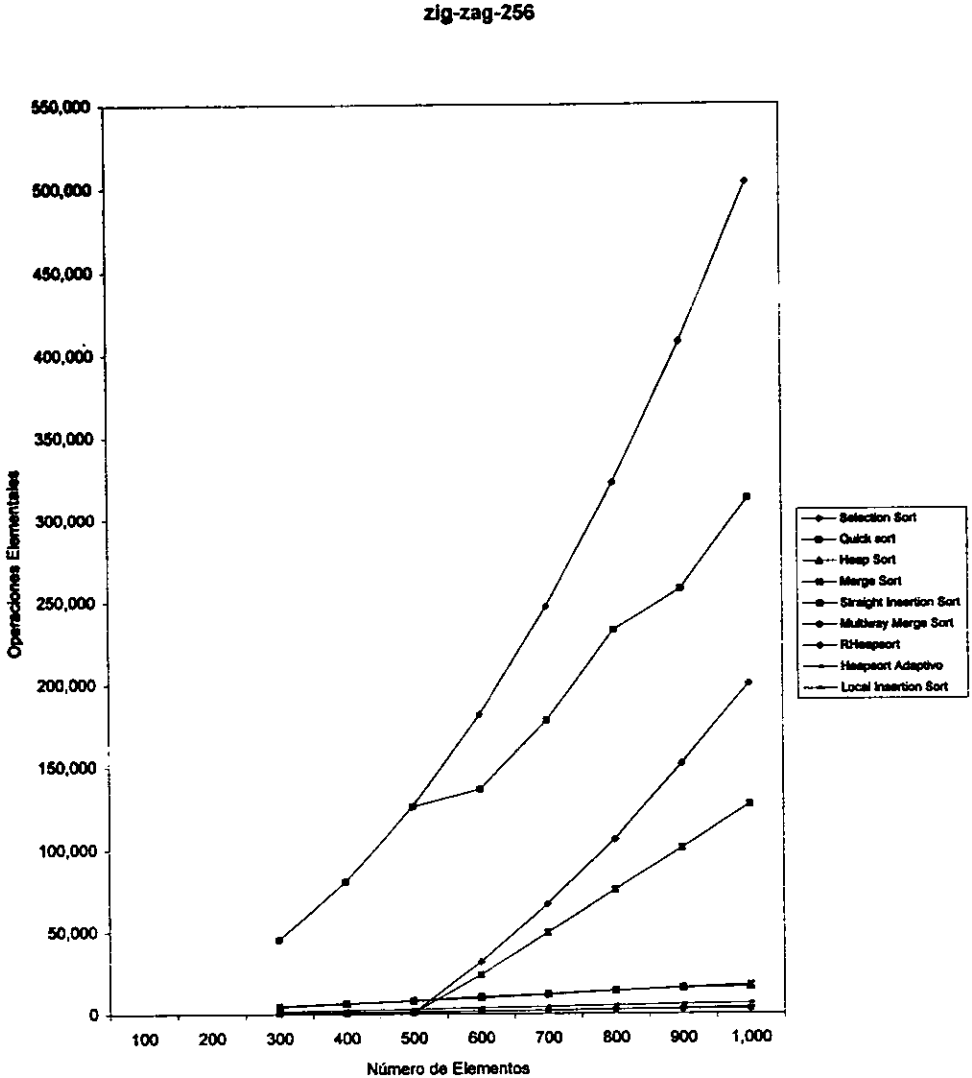
Gráfica 24. *zig-zag-16*

Cuando las listas comienzan a presentar cierto orden, el comportamiento de los algoritmos adaptivos mejora notablemente como lo vemos en el algoritmo Multiway Merge Sort y Local Insertion Sort, Gráfica 24.



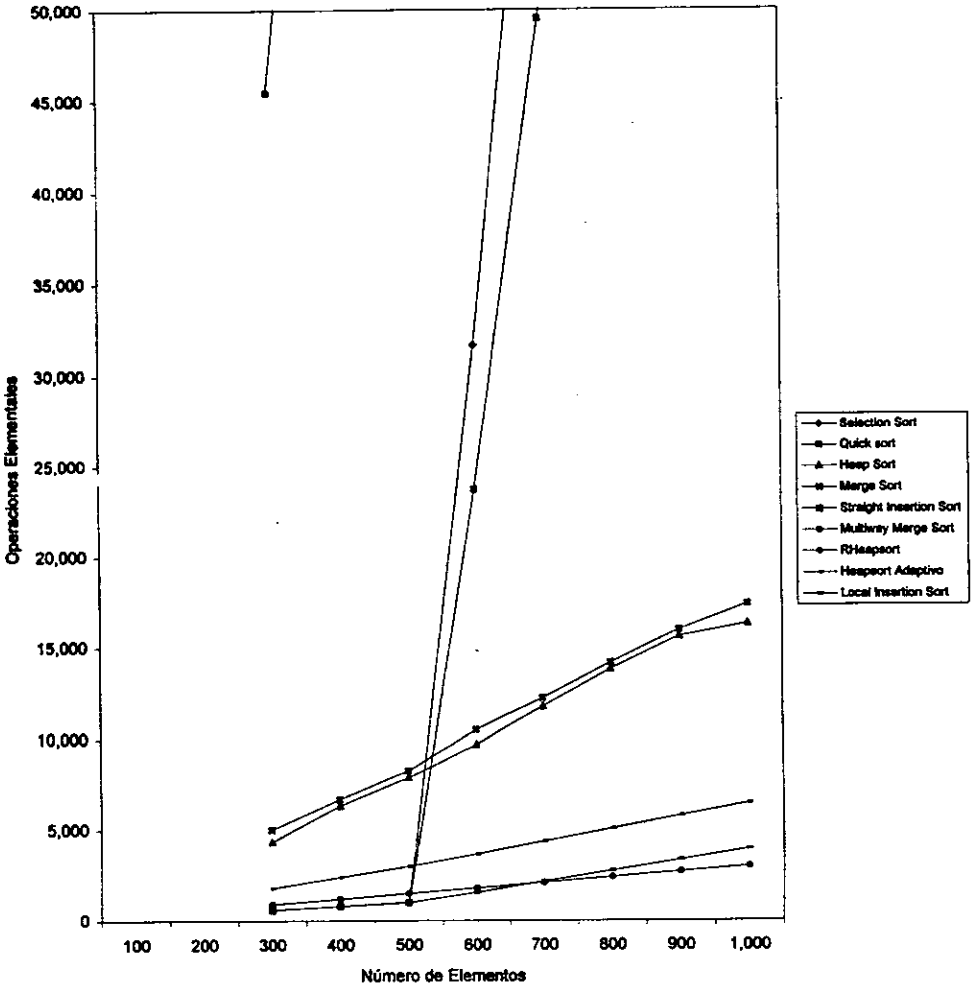
Gráfica 25. *zig-zag-64*

Los algoritmos adaptivos van teniendo su mejor desempeño cuando las listas estan ordenadas. En la Gráfica 25 observamos que para la versión implantada de Quick Sort cuando las listas estan ordenadas este va siendo el peor caso para Quick Sort.



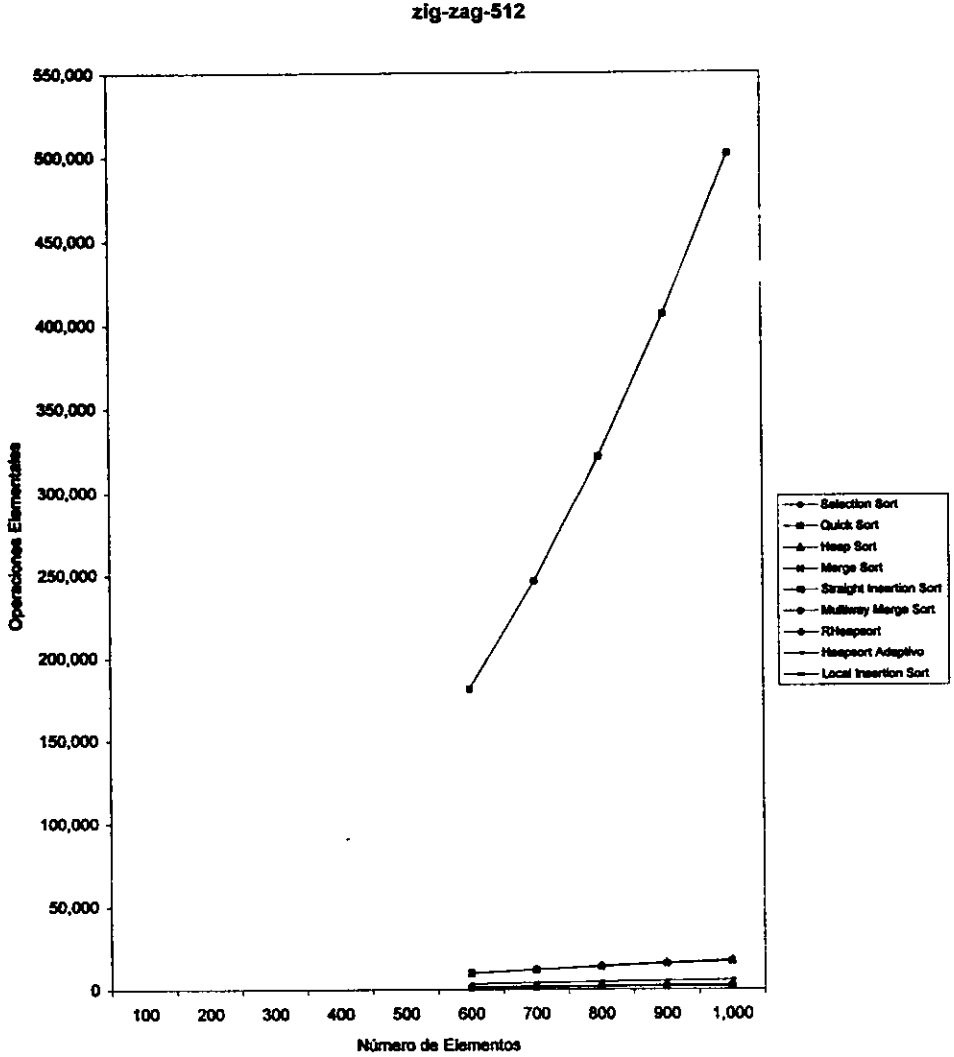
Gráfica 26. *zig-zag-256*

El desempeño de los algoritmos adaptivos Heapsort Adaptivo, Local Insertion Sort y Multiway Merge Sort ya mejoró el desempeño *nlog(n)* de los algoritmos clasicos de ordenamiento, como lo es de los algoritmos Merge Sort y Heap Sort, Gráficas 26 y 27.



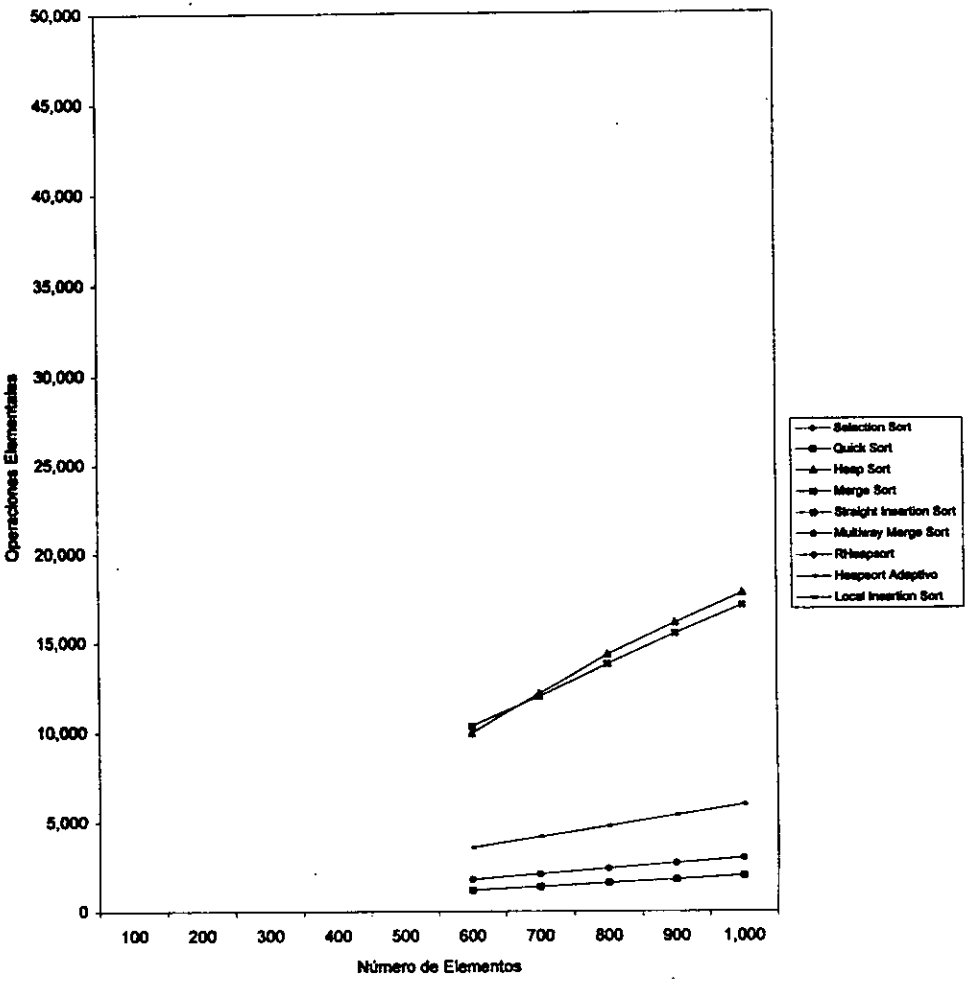
Gráfica 27. *zig-zag-256*

Aquí se presenta un acercamiento para apreciar las observaciones de la Gráfica 26.



Gráfica 28. *zig-zag-512*

Cuando las listas están ordenadas, el desempeño de los cinco algoritmos de ordenamiento adaptivos que presentamos en este trabajo, esta por debajo del tiempo *nlog(n)* de los algoritmos clásicos de ordenamiento, Gráfica 28 y 29.



Gráfica 29. *zig-zag-512*

Aquí se presenta un acercamiento para apreciar las observaciones de la Gráfica 28

CONCLUSIONES

El objetivo de este trabajo fué presentar los conceptos formales de Adaptabilidad de Algoritmos desarrollando el análisis de adaptabilidad en el problema de ordenamiento. La Adaptabilidad llega a ser de interés al observar en la práctica que secuencias de números presentan un cierto grado de orden. Para el objetivo de este trabajo se analizaron los algoritmos clásicos de ordenamiento y comparamos su comportamiento con los algoritmos de ordenamiento adaptivos. Pudimos observar que las medidas del desorden son importantes para cuantificar el orden existente en una secuencia. De esta manera, podemos concluir que para poder ver la ventaja de los algoritmos adaptivos con los de ordenamiento clásicos es importante el concepto de las medidas del desorden. Logramos comprobar con resultados empíricos la teoría acerca de la adaptabilidad de los algoritmos de ordenamiento adaptivos. Observamos también que unas implementaciones requieren más recursos físicos de la máquina como memoria, para observar los resultados buscados, es el caso concreto del algoritmo Heapsort Adaptivo y Local Insertion Sort, por el uso de estructuras de datos dinámicas.

Para la solución de un problema que requiera la utilización de un algoritmo de ordenamiento siempre es importante elegir el adecuado de acuerdo a las necesidades que se tienen en la tarea específica a resolver. Ahora con la introducción de los algoritmos adaptivos nuestro universo crece encontrando mejores soluciones para nuestros problemas de ordenamiento a resolver. Por la teoría utilizada y estructuras de datos empleadas para la comprobación empírica de nuestros resultados este puede ser considerado como material de apoyo para un curso de Estructuras de Datos o Análisis de Algoritmos. El trabajo que hemos presentado da pauta a la investigación de algoritmos adaptivos en otras medidas de desorden, dando lugar a un trabajo de investigación futuro.

BIBLIOGRAFÍA

[1] Charless F. Bowman
Algorithms y Data Structures An Aproach in C
Saunders College Publishing, USA, 1994.

[2] Heikki Mannila
Measures of presortedness and optimal sorting algorithms
IEEE Transactions on Computer, 34:318-325, April 1985.

[3] Johannes J. Martin.
Data Types and Data Structures.
Prentice-Hall International. Great Britain, 1986.

[4] Ola Peterson
Adaptivr Selection Sort,
Lund University, Departament of Computer Science, Box 118,S-221 00 LUND, Sweden, 1991.

[5] Reinhold Friedrich Hille
Data Abstraction and Program Development using Pascal.
Prentice Hall. Australia 1988.

[6] Udi Manber
Introduction to algorithms, A Creative Approach,
Addison-Wesley Publishing Company Inc. 1989.

[7] Vladimir Estivill-Castro, Derick Wood
A Survey of Adaptive Sorting Algorithms.
ACM Computing Survey. Vol. 24, No. 4, December 1992.

[8] Weis, Mark Allen
Data Structures and Algorithm Analysis,
The Benjamin/Cummings Publishing Company, Inc. 1992 California.

[9] Vladimir Estivill-Castro
Sorting and Measures of Disorder
PhDThesis, Waterloo, Ontario Canada, 1991.

[10] B. Chazelle and J. Incerpi.
Triangulation and shape complexity.
ACM Trans. On Graphics, 3:135-152, 84.

[11] S. Hertel and K. Mehlhorn.
Fast triangulation of simple polygons.
In Proceedings Conf. Foundations of Computer Theory, pages 207-218.
New York, 1983. Springer-Verlag.

[12] Vladimir Estivill-Castro y Luz Gasca Soto
El Problema de Los Árboles generadores de Peso mínimo y su Adaptabilidad. Un primer acercamiento. Reporte Técnico LANIA, RD-96-2, 1996.

[13] Maria de Luz Gasca Soto
Adaptabilidad en el problema de la ruta más corta.
Tesis de Maestria, Facultad de Ingenieria. México, D.F. 1994

[14] Vladimir Estivill-Castro y Luz Gasca Soto
Algoritmo Adaptivo para flujo en Redes.
Reporte Técnico LANIA, RI-96-5, 1996.

[15] K Mehlhorn
Data Structures and Algorithms, Vol I Sorting and Searching. Monographs on Theorical Computer Sciencie, Springer-Verlag,Berlin, 1984.

[16] S.S. SKIENA.
Encroaching list as a mesure of preorde.
BIT, 28(28):755-755,1988.

[17] A. MOFFAT and OLA PETERSON
Historial searching and sorting, In Second Annual International Simposium on Algorithms Lecture Notes in Computer Science. Springe-Varlag, 1991.

[18] A. MOFFAT and OLA PETERSON
A framework for adaptive sorting. In 3rd Scandinavian Workshop on Algorithms Theory, Lecture Notes in Computer Science. Springe-Varlag, 1991.