Análisis de complejidad algorítmica

Roberto Emilio Salas Ruiz[*]

Jorge Enrique Rodríguez Rodríguez[**]

Resumen

En este artículo se plasma la base del análisis de complejidad algorítmica, para lo que se toma un problema clásico en las ciencias de la computación, como es el ordenamiento de datos. En éste se comparan tres métodos de ordenamiento, a saber, por unión, burbuja e inserción; a cada uno de éstos se le halla el orden de complejidad y se implementan en un lenguaje de programación con el fin de comprobar la teoría frente a la práctica. Finalmente, los autores concluyen los resultados obtenidos.

Palabras clave: análisis de complejidad, tiempo de ejecución, algoritmo de ordenamiento, orden de complejidad.

Abstract

This paper provides a comprehensive introduction to the algorithms analysis. We have one of the most fundamental problems in the computer sciences, the data sorting. We compare three sorting algorithms: merge, bubble and insertion sort. For each algorithm is made an analysis of complexity that is worst-case running time. They were implemented an programming languaje with the objective to contrast theory and practice. Finally, the authors give conclusions about results obtained.

[*] Ingeniero de sistemas, candidato a Magíster en Ingeniería de Sistemas de la Universidad Nacional de Colombia, docente Universidad Distrital Francisco José de Caldas, adscrito a la Facultad Tecnológica. Correo electrónico: rsalas72@hotmail.com.

[**] Ingeniero de sistemas, especialista en Diseño y Construcción de Soluciones Telemáticas, especialista en Ingeniería del Software, candidato a Magíster en Ingeniería de Sistemas de la Universidad Nacional de Colombia, docente tiempo completo de la Universidad Distrital Francisco José de Caldas, adscrito a la Facultad Tecnológica. Correo electrónico: jrodri@udistrital.edu.co.

Words Keys: analysis of complexity, running time, sorting algorithm, order of complexity.

Introducción

Informalmente, un algoritmo es cualquier procedimiento computacional bien definido que toma algún valor o conjunto de valores como entrada y produce algún valor o conjunto de valores como salida. Así, un *algoritmo* es una secuencia de pasos computacionales que transforma la entrada en salida.

Se puede ver un algoritmo como una herramienta para resolver un problema computacional bien especificado. La declaración del problema especifica, en términos generales, las relaciones deseadas entrada/salida; es decir:

Entrada: una secuencia de n números $<a_1, a_2, \ldots, a_n>$

Salida: una permutación $<a_{1'}, a_{2'}, \ldots, a_{n'}>$ de la secuencia de entrada tal que $a_{1'} \leq a_{2'} \leq \ldots \leq a_{n'}$.

Por ejemplo, dada la secuencia de entrada <30, 50, 37, 22, 48, 29>, un algoritmo de ordenamiento retorna como salida la secuencia <22, 29, 30, 37, 48, 50>.

El ordenamiento es una operación fundamental en las ciencias de la computación y, como resultado de esto, un gran número de algoritmos de ordenamiento han sido desarrollados. Seleccionar cuál algoritmo es mejor depende, entre otros factores, del número de elementos a ser ordenados, en que grado están los elementos parcialmente ordenados, posibles restricciones en los valores de los elementos y el tipo de dispositivo de almacenamiento a ser utilizado (memoria principal, discos, CD-ROM).

Teniendo en cuenta lo anterior, es fundamental que los ingenieros del *software* realicen el análisis de complejidad algorítmica en el diseño de *software*, ya que, dependiendo del problema a resolver algorítmicamente, éste va a permitir seleccionar el algoritmo más eficiente para tal fin. Del mismo modo, este análisis le permite al ingeniero del *software* conocer el recurso de máquina (tiempo de ejecución y

memoria ocupada) utilizado en la ejecución de un algoritmo, y, de esta forma, minimizar costos de implementación del mismo y optimizar el recurso de máquina.

El objetivo principal de este artículo es dar algunos lineamientos para diseñar algoritmos eficientes. Sin embargo, cuando uno se enfrenta con varios algoritmos distintos para resolver el mismo problema, es preciso decidir cuál de ellos es el más adecuado para la aplicación considerada. Una herramienta esencial para este propósito es el análisis de algoritmos.

El análisis de algoritmos planteado en este artículo es válido para computadoras personales empleadas en nuestro medio.

1. Análisis de algoritmos

Analizar un algoritmo significa predecir los recursos que el algoritmo requiere. Ocasionalmente, recursos tales como memoria, ancho de banda de comunicación o *hardware* son de interés primario, pero, más a menudo, el tiempo computacional es el que se desea medir. Por lo general, al analizar varios algoritmos candidatos para un problema, el más eficiente puede ser fácilmente identificado [1].

Antes de iniciar con el análisis de un algoritmo, se debe tener un modelo de la tecnología de implementación que será utilizada, incluyendo un modelo para los recursos de esta tecnología y sus costos.

En este artículo, se asumirá un sólo procesador genérico de computadora personal, en el cual las instrucciones son ejecutadas una seguida de otra y no concurrentemente, y contienen las instrucciones más comunes: aritméticas (suma, resta, multiplicación, división, residuo...), movimiento de datos (cargar, almacenar, copiar) y control (ramificación condicional e incondicional, llamado a funciones). Cada una de estas instrucciones toma una cantidad constante de tiempo.

Analizar el algoritmo más simple en la computadora más sencilla de hoy puede ser todo un desafío. Las herramientas matemáticas requeridas podrían incluir combinatorias, teoría de la

probabilidad, destreza algebraica y habilidad para identificar los más significantes términos en una fórmula. Debido a que el comportamiento de un algoritmo puede ser diferente para cada una de las posibles entradas, se necesitan medios para resumir este comportamiento en fórmulas simples y fáciles de entender. Pero, no hay una fórmula mágica para analizar los algoritmos. En su mayor parte, es una cuestión de juicio, intuición y experiencia. Sin embargo, existen algunas técnicas que suelen resultar útiles, tales como saber la forma de enfrentarse a estructuras de control y a ecuaciones de recurrencia. [1].

Con el fin de introducir al lector en el tema, a continuación se darán algunas definiciones básicas para el análisis de algoritmos.

1.1 Notación asintótica. La notación asintótica es utilizada para describir el tiempo de ejecución (T(n)) asintótico de un algoritmo definido en términos de funciones, cuyo dominio puede ser el conjunto de los números naturales o los reales. Se denomina asintótica porque trata acerca del comportamiento de funciones en el límite, esto es, para valores suficientemente grandes de su parámetro. En consecuencia, los argumentos basados en esta notación pueden no llegar a tener un valor práctico cuando el parámetro adopta valores *de la vida real*.

a. Notación Θ. Para una función dada $g(n)$, se denota por $\Theta(g(n))$ al conjunto de funciones $\Theta(g(n)) = \{f(n)$: si existen constantes positivas, c_1, c_2 y n_0 tal que $0 \leq c_1 g(n) \leq f(n) \leq c_2 g(n)$ para todo $n \geq n_0\}$. Se dice que $g(n)$ está acotada tanto por encima como por debajo por $f(n)$; por ejemplo, $3n^2 - 4n = \Theta(n^2)$.

b. Notación O. Para una función dada $g(n)$, se denota por $O(g(n))$ al conjunto de funciones $O(g(n)) = \{f(n)$: si existen constantes positivas, c y n_0 tal que $0 \leq f(n) \leq cg(n)$ para todo $n \geq n_0\}$ [2]. Se dice que $g(n)$ está acotada por encima por $f(n)$; por ejemplo, $3n - 5 = O(n)$.

c. Notación Ω. Para una función dada $g(n)$, se denota por $\Omega(g(n))$ al conjunto de funciones $\Omega(g(n)) = \{f(n)$: si existen constantes positivas, c y n_0 tal que $0 \leq cg(n) \leq f(n)$ para todo $n \geq n_0\}$. Se dice que $g(n)$ está acotada por debajo por $f(n)$; por ejemplo, $3n^2 + 5 = \Omega(n^2)$.

Adicionalmente, $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$.

Suponiendo que se tiene un algoritmo con n entradas y su tiempo de ejecución, en el peor de los casos, es $T(n) = an^2 + bn + c$, se dice que el tiempo de ejecución de este algoritmo es de orden $\Theta(n^2)$. Aunque algunas veces se puede determinar el tiempo de ejecución exacto de un algoritmo, esta precisión extra no es fundamental ya que, para grandes entradas, las constantes multiplicativas y los términos de orden bajo son dominados por los efectos del tamaño de la entrada, por esta razón se estudia el comportamiento asintótico de los algoritmos.

1.2 Análisis de las estructuras básicas de control. La idea fundamental en el análisis de algoritmos es determinar el tiempo que tardan en ejecutarse cada una de las instrucciones que lo componen. En primer lugar, se determina el tiempo requerido por las estructuras secuenciales las cuales son sencillas de analizar, ya que este tiempo suele estar acotado por una constante [3].

a. Estructuras secuenciales. Sean P_1 y P_2 dos fragmentos de un algoritmo, y t_1 y t_2 los tiempos requeridos por P_1 y P_2 respectivamente, el tiempo máximo de ejecución de estos fragmentos está en el orden de $\Theta(máx(t_1, t_2))$ [4].

b. Estructuras de repetición. Sólo se examinará el tiempo de ejecución de la instrucción *PARA (for)*, las otras estructuras de repetición se dejan a interés del lector. La estructura *PARA* es la más sencilla de analizar dentro de este grupo.

Considere el siguiente ciclo:

(1) PARA $i \leftarrow 1$ *HASTA* n *HACER*

(2) $p(i)$

La instrucción (1) tiene una duración (tiempo de ejecución) de t_1 y se ejecuta $n + 1$ veces, la instrucción (2) tiene una duración de t_2 y se ejecuta n veces. Para calcular $T(n)$ se suman los productos del tiempo de ejecución de cada instrucción por el número de veces que se realiza; lo cual da como resultado $T(n) = t_1 * (n+1) + t_2 n$, al agruparlo esto es $T(n) = (t_1 + t_2) * n + t_1$.

Por la notación anterior $T(n) = \Theta(n)$. De manera general, se puede afirmar que una estructura *PARA* que se ejecute n veces es del orden $\Theta(n)$.

2. ¿Cómo hallar la complejidad de un algoritmo?

El análisis de complejidad de un algoritmo consiste en estimar el tiempo de ejecución que gastará para una entrada de tamaño n. Ordenar 64.000 datos tomará más tiempo que ordenar diez datos; además, el algoritmo de ordenamiento utilizado puede tomar diferentes tiempos para ordenar dos secuencias de entradas del mismo tamaño, dependiendo de cuan parcialmente ordenados se encuentren. En general, el tiempo tomado por un algoritmo crece con el tamaño de la entrada, de modo que es común describir el tiempo de ejecución de un algoritmo como una función del tamaño de su entrada.

El tiempo de ejecución de un algoritmo para una entrada en particular es el número de operaciones primitivas o pasos que son ejecutados. Una instrucción podría tomar una cantidad de tiempo diferente que otra instrucción, pero se asumirá que cada ejecución de la *i-ésima* instrucción toma un tiempo c_i, donde c_i es una constante.

Para el análisis de los algoritmos tratados en este artículo nos centraremos en el tiempo de ejecución para el peor caso, ya que éste ofrece un límite superior en tiempo de ejecución para cualquier entrada y nunca tomará un tiempo mayor.

2.1 Ordenamiento por inserción. El ordenamiento por inserción funciona de la misma forma como las personas ordenan a mano un juego de cartas; se inicia con la mano izquierda vacía y las cartas se colocan cara abajo sobre la mesa. Se remueve una carta a la vez de la mesa y se inserta en la posición

correcta de la mano izquierda. Para hallar la correcta posición de una carta se compara con cada una de las cartas que ya están en la mano, de derecha a izquierda. En todo momento, las cartas mantenidas en la mano izquierda se encuentran ordenadas, y éstas, originalmente, estuvieron en el tope de la pila de cartas en la mesa.

A continuación se muestra el seudocódigo de este método de ordenamiento, junto con el análisis de complejidad:

ordenamientoInserción (v)

(1) para i ← 2 hasta n hacer

(2) temporal ← v[i]

(3) j ← i - 1

(4) mientras v[j] > 0 ∧ v[j] > temporal haga

(5) v[j + 1] ← v[j]

(6) j ← j – 1

(7) fin-mientras

(8) v[j + 1] ← temporal

(9) fin-para

(10) fin-ordenamientoInserción

Para realizar el análisis del algoritmo de ordenamiento por inserción, se presenta el procedimiento de ordenamiento con el *costo* de tiempo c_i de cada instrucción, y el número de veces que cada instrucción es ejecutada. Para cada $i = 2, 3..., n,$ se denota t_i el número de veces que la instrucción *mientras* en la línea (4) es ejecutada para un valor de i; de la misma forma, se asume que las instrucciones fin-para, fin-mientras y fin-ordenamientoInserción no toman tiempo de ejecución. (Véase Tabla 1)

Tabla 1. Algoritmo de ordenamiento por inserción (Fuente tomada y adaptada de Cormen 2001)

Algoritmo	**Costo**	**Número de veces**
ordenamientoInserción (v)		
(1) para i ← 2 hasta n hacer	c_1	n
(2) temporal ← v[i]	c_2	n - 1
(3) j ← i - 1	c_3	n - 1
(4) mientras v[j] > 0 ∧ v[j] > temporal haga	c_4	$\sum_{i=2}^{n} t_i$
(5) v[j + 1] ← v[j]	c_5	$\sum_{i=2}^{n} t_i - 1$
(6) j ← j – 1	c_6	$\sum_{i=2}^{n} t_i - 1$
(7) fin-mientras		
(8) v[j + 1] ← temporal	c_7	n - 1
(9) fin-para		
(10) fin-ordenamientoInserción		

El tiempo de ejecución del algoritmo es la suma de los tiempos de ejecución de cada instrucción. Para calcular el *T(n)* se suman los productos de las columnas de costo y el número de veces para obtener:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4 \sum_{i=2}^{n} t_i + c_5 \sum_{i=2}^{n} (t_i - 1) + c_6 \sum_{i=2}^{n} (t_i - 1) + c_7(n-1) \quad \text{(f1)}$$

Para el algoritmo, el mejor caso ocurre cuando el vector de entrada se encuentra ya ordenado. Para cada $i = 2, 3, \ldots, n$, se cumple que $v[i] \leq$ temporal en la instrucción (4) cuando j tiene el valor inicial de $i - 1$. Así, $t_i = 1$ para $i = 2, 3, \ldots, n$, y el tiempo de ejecución en el mejor caso es: $T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1)$.

Agrupando términos $T(n) = (c_1 + c_2 + c_3 + c_4 + c_7)n - (c_1 + c_2 + c_3 + c_4 + c_7)$.

Este tiempo de ejecución puede ser expresado como an + b para constantes a y b que dependen de los costos c_i de las instrucciones. En la notación asintótica este algoritmo, para el mejor caso, es $\Theta(n)$, es decir, es una función lineal de n.

En otro caso, si el vector de entrada se encuentra completamente en orden inverso resulta el peor caso. Se debe comparar cada elemento *v[i]* con cada elemento en el subvector ordenado *v[1….i-1]*, y así, $t_i = i$, para $i = 2, 3, \ldots, n$. Teniendo que $\sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1$ y

$$\sum_{i=2}^{n} (i-1) = \frac{n(n-1)}{2}.$$

Reemplazando estos valores en *(f1)* se obtiene el tiempo de ejecución del algoritmo de ordenamiento por inserción para el peor caso:

$$T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5\left(\frac{n(n-1)}{2}\right) + c_6\left(\frac{n(n-1)}{2}\right) + c_7(n-1)$$

Agrupando términos, se obtiene:

$$T(n) = \left(\frac{c_4 + c_5 + c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4 - c_5 - c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7)$$

Así, el tiempo de ejecución para el peor caso puede ser expresado como an^2 + bn + c. Para constantes a, b, y c que dependen de los costos c_i de las instrucciones. En la notación asintótica, este algoritmo, para el peor caso, es $\Theta(n^2)$, es decir, es una función cuadrática de n.

2.2 Ordenamiento burbuja. Éste es uno de los algoritmos de ordenamiento más populares en el campo de la computación, por su facilidad de construcción. Funciona intercambiando repetidamente elementos adyacentes que no se encuentran ordenados; es decir, inicia comparando el último elemento con el penúltimo, luego el penúltimo con antepenúltimo, y así sucesivamente hasta llegar a comparar el segundo elemento con el primero. Cada vez que hace una comparación toma la decisión de intercambiarlos de acuerdo con la política de ordenamiento (ascendente o descendentemente).

El algoritmo es:

ordenamientoBurbuja(v)

(1) para i ← 1 hasta n hacer

(2) para j ← n hasta i +1 hacer

(3) si (v[j] < v[j – 1]) entonces

(4) intercambiar v[j] con v[j - 1]

(5) fin-si

(6) fin-para

(7) fin-para

(8) fin-ordenamientoBurbuja

Ahora se hará el análisis de complejidad (véase Tabla 2):

Tabla 2. Algoritmo de ordenamiento burbuja

Algoritmo	**Costo**	**Número de veces**
ordenamientoBburbuja(v)		
(1) para i ← 1 hasta n hacer	c1	n + 1
(2) para j ← n hasta i +1 hacer	c2	$\sum_{i=1}^{n} t_i$
(3) si (v[j] < v[j – 1]) entonces	c3	$\sum_{i=1}^{n} (t_i - 1)$
(4) intercambiar v[j] con v[j - 1]	c4	$\sum_{i=1}^{n} (s_i - 1)$
(5) fin-si		

(6) fin-para		
(7) fin-para		
(8) fin-ordenamientoBurbuja		

Donde $T(n) = c_1(n+1) + c_2\sum_{i=1}^{n} t_i + c_3\sum_{i=1}^{n}(t_i - 1) + c_4\sum_{i=1}^{n}(s_i - 1)$. (f2)

El mejor caso ocurre cuando el vector de entrada ya se encuentra completamente ordenado; para este caso en la cuarta (4) instrucción s_i es igual a *1*, ya que esta instrucción no se ejecuta. La segunda y tercera línea siempre se ejecutan, ya sea en el mejor o en el peor caso; así, t_i es igual a $n - i + 1$, esto se debe a que se tiene hacer un recorrido desde la posición *n* hasta la posición i + *1*. Teniendo que la $\sum_{i=1}^{n}(n-i+1) = \frac{n(n+1)}{2}$ y $\sum_{i=1}^{n}(n-i) = \frac{n^2 - n}{2}$. Reemplazando estos valores en *(f2)* se obtiene

$T(n) = c_1(n+1) + c_2\frac{n(n+1)}{2} + c_3\frac{n(n-1)}{2}$, al agrupar los términos, se obtiene:

$T(n) = \left(\frac{c_2}{2} + \frac{c_3}{2}\right)n^2 + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2}\right)n + c_1$, este tiempo se puede expresar de la forma $an^2 + bn + c$. Con lo anterior, se deduce que este método de ordenamiento para el mejor caso tiene un orden de complejidad $\Theta(n^2)$.

El peor caso para este algoritmo ocurre cuando el vector de entrada se encuentra en orden inverso. En esta situación, s_i es igual a $n - i + 1$, por consiguiente, en *(f2)* se obtendría: $T(n) = c_1(n+1) + c_2\frac{n(n+1)}{2} + c_3\frac{n(n-1)}{2} + c4\frac{n(n-1)}{2}$, al agrupar los términos

$T(n) = \left(\frac{c_2}{2} + \frac{c_3}{2} + \frac{c_4}{2}\right)n^2 + \left(c_1 + \frac{c_2}{2} - \frac{c_3}{2} - \frac{c_4}{2}\right)n + c_1$, teniendo, en el peor de los casos, un orden de complejidad de $\Theta(n^2)$.

2.3 Ordenamiento por unión (*merge sort*). Este método de ordenamiento sigue de manera similar el paradigma de *dividir* y *conquistar*. Intuitivamente funciona de la siguiente forma:

- Dividir. Divide la secuencia de *n* elementos a ser ordenada en dos subsecuencias de *n/2* elementos cada uno.
- Conquistar. Ordena las dos subsecuencias recursivamente utilizando el procedimiento *merge sort*.
- Combinar. Une las dos subsecuencias ordenadas para producir el ordenamiento completo.

La recursividad termina cuando la secuencia a ser ordenada tiene una longitud de uno (1), en este caso no se realiza ninguna tarea de ordenamiento dado que cualquier secuencia de longitud uno (1) se encuentra ordenada. La operación fundamental de este algoritmo es la unión de las dos secuencias ordenadas en el paso *combinar*. Para tal fin, se utiliza un algoritmo auxiliar *unir(v, ip, iq, ir)*, donde *v* es el vector de entrada, e *ip*, *iq*, e *ir*, son índices de elementos del vector, tal que, $ip \leq iq < ir$. El procedimiento asume que los subvectores *v[ip..iq]* y *v[iq+1..ir]* están ordenados. El procedimiento los une formando un sólo subvector que reemplaza el actual subvector *v[ip..ir]*.

Retomando el tema de juego de cartas, por ejemplo, se tienen dos pilas de cartas de cara arriba en una mesa. Cada pila está ordenada, con las cartas más pequeñas en la parte de arriba. Lo que se desea es unir las dos pilas en una sola, la cual se coloca cara abajo en la mesa. El paso básico consiste en la selección de las dos cartas más pequeñas de las dos pilas que se encuentran cara arriba, removiéndolas de su pila y colocándolas cara abajo en la pila que se está formando. Este paso se repite hasta que una de las dos pilas cara arriba quede vacía. En ese momento, sólo se toman las cartas que sobraron en la pila que no quedó vacía y se colocan cara abajo en la pila que se está formando, obteniendo una nueva pila ordenada. Computacionalmente, cada paso toma un tiempo constante, ya que sólo se revisan las dos cartas de arriba. Dado que se ejecutan a lo sumo *n* pasos básicos, la unión toma un tiempo de $\Theta(n)$.

A continuación se muestra el procedimiento unir (Fuente tomada y adaptada de Cormen, 2001):

```
unir(v, ip, iq, ir)
(1)     n1 ← iq – ip + 1
(2)     n2 ← ir - iq
(3)     para i ← 1 hasta n1 hacer
(4)       l[i] ← v[ip+i-1]
(5)     fin-para
(6)     para i ← 1 hasta n2 hacer
(7)       r[i] ← v[iq+i]
(8)     fin-para
(9)     l[n1+1] ← ∝[1]
(10)    r[n2+1] ← ∝
(11)    i ← 1
(12)    j ← 1
(13)    para k ← ip hasta ir hacer
(14)      si l[i] ≤ r[j] entonces
(15)        v[k] ← l[i]
(16)        i ← i+1
(17)      sino
(18)        v[k] ← r[j]
(19)        j ← j+1
(20)      fin-si
```

[1] El símbolo $\propto$ se utiliza para identificar el fin de las dos pilas y, de esta forma, evitar tener que verificar cuál de las dos pilas ya se encuentra vacía. El $\propto$ es un valor que nunca va a ser más pequeño que cualquier elemento de las dos pilas.

(21) fin-para

(22) fin-unir

Para demostrar que el procedimiento unir opera en un tiempo de $\Theta(n)$, donde $n = ir - ip + 1$, se debe observar que las instrucciones (1), (2), (9), (10), (11), y (12) toman un tiempo constante. Los ciclos *para* de las instrucciones de la (3) hasta la (8) toman un tiempo de $\Theta(n1 + n2) = \Theta(n)$, y hay n iteraciones para el ciclo *para* de las instrucciones (13) hasta (21), cada una de las cuales toma un tiempo constante.

Ahora, se puede utilizar el procedimiento unir como una subrutina en el algoritmo *ordenamientoUnión(v, ip, ir)*. Este algoritmo ordena los elementos en el subvector *v[ip..ir]*. Si $ip \geq ir$, el subvector tiene, a lo sumo, un elemento que se encuentra ya ordenado. De otro modo, el paso *dividir* utiliza el índice *iq* para dividir el vector *v[ip..ir]* en dos subvectores: *v[ip..iq]* que contiene $\lceil n/2 \rceil$ elementos, y *v[iq+1..ir]* que contiene $\lfloor n/2 \rfloor$ elementos. A continuación se muestra el algoritmo ordenamiento por unión:

ordenamientoUnir(v, ip, ir)

(1) si ip < ir entonces

(2) iq $\leftarrow \lfloor (ip+ir)/2 \rfloor$

(3) ordenamientoUnir(v, ip, iq)

(4) ordenamientoUnir(v, iq+1, ir)

(5) unir(v,ip,iq,ir)

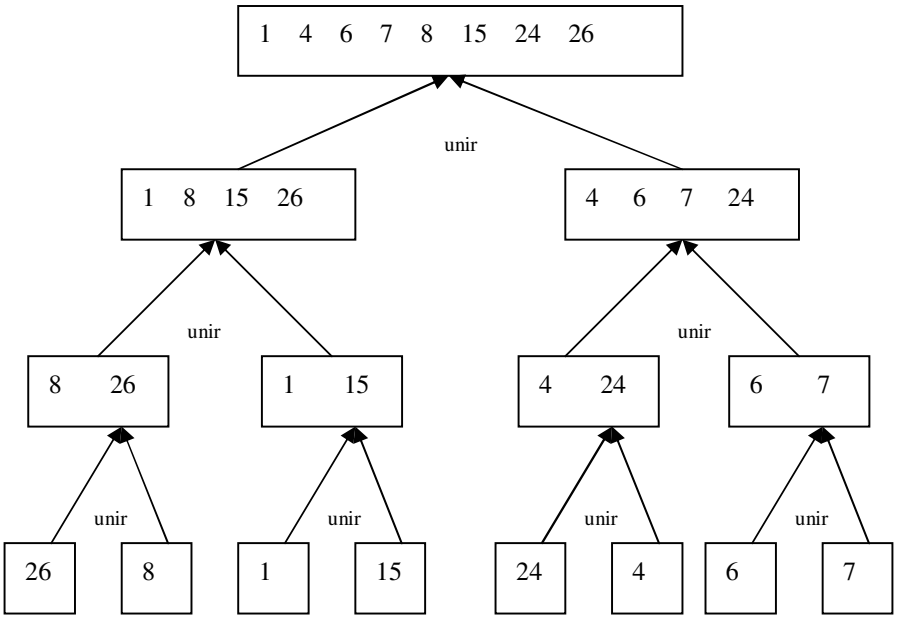
(6) fin-si

(7) fin-ordenamientoUnir

La Figura 1 ilustra la operación del ordenamiento por unión de abajo a arriba cuando n es una potencia de dos. El algoritmo consiste en unir pares de secuencias de un elemento para formar secuencias ordenadas de longitud 2; luego, unir pares de secuencias de longitud 2 para formar

secuencias ordenadas de longitud 4, y así sucesivamente, hasta que dos secuencias de longitud *n/2* son unidas para formar la secuencia final ordenada de longitud *n*.

Figura 1. Operación del algoritmo *ordenamientoUnir* para el vector *v = <26, 8, 1, 15, 24, 4, 6, 7>*



Fuente:

Análisis de complejidad para el algoritmo de ordenamiento por unión. Aunque el seudocódigo del algoritmo *ordenamientoUnir* funciona correctamente cuando el número de elementos no es par, el análisis basado en recurrencias[2] es simplificado si se asume que el tamaño del problema original es una potencia de 2. Cada paso *dividir* produce dos subsecuencias de tamaño *n/2*.

El ordenamiento por unión sobre un sólo elemento toma un tiempo constante. Cuando se tiene *n > 1* elementos, el análisis de tiempo de ejecución se realiza como sigue:

- Dividir. Este paso sólo divide en dos el subvector, lo cual asume un tiempo constante, es decir, *c'*.

- Conquistar. Recursivamente se resuelven dos subproblemas, cada uno de tamaño *n/2*, los cuales contribuyen *2T(n/2)* al tiempo de ejecución.

- Combinar. Como se dijo el procedimiento unión en un subvector de *n* elementos toma un tiempo de ejecución *Θ(n)*, entonces, este paso toma un tiempo de *c''n*.

Lo anterior, permite deducir las siguientes ecuaciones de recurrencia:

$T(n) = c$ si $n = 1$

$T(n) = 2T(n/2) + c''n + c'$ si $n > 1$

Esta recurrencia resuelta por el teorema maestro[3] da como resultado $T(n) = \Theta(n\log(n))$, el logaritmo se da en base 2. Debido a que la función logaritmo crece más lentamente que cualquier función lineal, para entradas suficientemente grandes el ordenamiento por unión con su tiempo de ejecución *Θ(nlogn)* vence tanto al ordenamiento por inserción como al burbuja, cuyos tiempos de ejecución son $\Theta(n^2)$ en el peor de los casos.

Conclusiones

Una vez implementados estos tres métodos de ordenamiento, en C++, se comprobó el tiempo de ejecución que utilizó cada uno para ordenar 16.384 datos (véase Tabla 3) para el peor caso, es decir, con los datos completamente invertidos. La máquina utilizada para la prueba de los algoritmos tiene las siguientes características generales: procesador Pentium MMX y 32 megabytes en memoria RAM[4].

---

[2] Una recurrencia es una ecuación o desigualdad que describe una función en términos de la misma función, pero, para valores de entradas más pequeños.

[3] Sea $a \geq 1$ y $b > 1$ constantes, *f(n)* una función, y *T(n)* definida sobre los enteros no negativos por la recurrencia *T(n)=aT(n/b)+f(n). T(n)* puede ser limitada asintóticamente por $T(n) = \Theta\left(n^{\log_b a}\log(n)\right)$, si $f(n) = \Theta\left(n^{\log_b a}\right)$.

[4] Memoria de acceso aleatorio.

Tabla 3. Tiempo que utiliza cada algoritmo en ordenar 16.384 datos

Método de ordenamiento	**Tiempo de ejecución (milisegundos)**	**Orden de complejidad**
Ordenamiento burbuja	8.070	$\Theta(n^2)$
Ordenamiento por inserción	9.560	$\Theta(n^2)$
Ordenamiento por unión	60	$\Theta(nlog(n))$

Fuente:

- Con la implementación de los algoritmos se comprueba que el ordenamiento por unión es más eficiente, en cuanto a tiempo de ejecución se refiere; sin embargo, su implementación es más difícil que los otros dos métodos de ordenamiento. Del mismo modo, si bien su tiempo de ejecución para el peor caso es menor, la invocación recursiva de rutinas que utiliza hace que requiera más recurso de máquina en la escala de memoria RAM.

- Se observa la enorme diferencia que existe en cuanto al tiempo de ejecución de un algoritmo con complejidad $\Theta(nlog(n))$ frente a los que utilizan un orden de complejidad de $\Theta(n^2)$. En el ejemplo planteado, se muestra una diferencia de aproximadamente ocho segundos entre estos dos tipos de algoritmos; lo cual implica que esta diferencia crece exponencialmente a medida que se incrementan los datos.

- Como se observa en la Tabla 3, el algoritmo de ordenamiento burbuja utiliza menos tiempo que el método por inserción en ordenar los 16.384 datos; esto se debe a las constantes de

tiempo que utiliza cada método. Pues, el método burbuja emplea menos instrucciones de máquina para hacer el ordenamiento.

- Con lo anterior, queda demostrado la importancia de que los ingenieros del *software* dediquen parte del tiempo dentro del desarrollo de *software* a realizar un buen diseño del algoritmo basado en el análisis de complejidad; pues, como se observa, un buen análisis conlleva a la minimización del recurso de máquina, específicamente el tiempo de ejecución.

Referencias bibliográficas

[1] Cormen TH, Leiserson CE., Rivest RL, Stein C. Introduction to Algorithms, 2nd ed. MIT Press; 2001.

[2] Torres C. Diseño y análisis de algoritmos. Paraninfo; 1992.

[3] Brassard G y Bratley P. Fundamentos de algoritmia. Prentice Hall; 1997.

[4] Aho AV, Hopcroft JE y Ullman JD, Estructuras de datos y algoritmos. Addison Wesley Iberoamericana; 1988.