Estudio y optimización del algoritmo de ordenamiento Shellsort

Benjamin Bustos
Departamento de Ciencias de la Computación, Universidad de Chile
bebustos@dcc.uchile.cl

Resumen

Este estudio analiza, en forma empírica, el desempeño del algoritmo de ordenamiento Shellsort con diferentes series de pasos. Se estudian optimizaciones al algoritmo para evitar los peores casos y se compara su rendimiento con algoritmos de ordenamiento eficientes (Quicksort, Mergesort y Heapsort), discutiéndose la utilidad del algoritmo para resolver el problema de ordenamiento de conjuntos de tamaño medio.

1. Introducción

El estudio de algoritmos de ordenamiento tiene una gran importancia dentro de la Ciencia de la Computación, pues una buena cantidad de los procesos realizados por medios computacionales requieren que sus datos estén ordenados. Además, el hecho de almacenar los datos de manera ordenada permite implementar algoritmos de búsqueda muy rápidos (por ejemplo: búsqueda binaria). Esta y muchas otras razones de fin práctico impulsaron el estudio y la búsqueda de algoritmos de ordenamiento eficientes.

Desde los comienzos del uso de computadores se conocían algoritmos que resolvían el problema en tiempo cuadrático respecto del tamaño del problema, pero eran rutinas muy sencillas y lentas. El algoritmo de ordenamiento Shellsort fue publicado en 1959 por Donald L. Shell, y fue uno de los primeros en romper la barrera del orden cuadrático, aunque esto en realidad se probó un par de años después de su publicación.

El objetivo de este estudio es demostrar empíricamente que implementar Shellsort con series de pasos dependientes del tamaño del arreglo puede llegar a ser mucho más eficiente que con las series clásicas, las cuales son independientes del tamaño del arreglo, pero hay que aplicar una optimización sencilla para obtener buenos resultados: todos los pasos de la serie deben ser números impares. Además,

este estudio muestra que dada la simplicidad de programación del algoritmo, su buen peor caso y caso promedio, y su ejecución "in place", es decir, no necesita espacio adicional para realizar el ordenamiento del arreglo, lo hacen un buen candidato para resolver el problema de ordenamiento cuando la cantidad de elementos a ordenar no es muy grande (menos de 100.000 elementos).

2. Descripción del algoritmo de ordenamiento Shellsort

El problema consiste esencialmente en ordenar un número finito de elementos en un tiempo razonable. Para estos efectos, diremos que cada elemento ocupa una celda dentro de un arreglo previamente definido. Shellsort trabaja mediante una serie de iteraciones, utilizando un algoritmo de ordenación simple (Insert Sort) entre elementos que se encuentran a determinada distancia dentro del arreglo, distancia que disminuye a medida que avanzamos en iteraciones, con lo que la última iteración corresponde al algoritmo de ordenación tradicional de Insert Sort (cuando la distancia es 1).

En primer lugar analicemos el algoritmo de Insert Sort utilizando pseudocódigo:

```
for j ← 2 to n do
    KEY ← L[j]; i ← j − 1
    while i>0 and L[i]>KEY do
        L[i+1] ← L[i]; i ← i − 1
    end
    L[i+1] ← KEY
end
```

donde **n** corresponde al número total de elementos a ordenar y **L** es originalmente el arreglo desordenado. El algoritmo actúa tomando cada elemento desde el segundo en adelante, y se va intercambiando con los elementos anteriores a él mientras encuentre que el elemento a su izquierda en el arreglo es mayor que él. De esta forma, cuando vamos a comparar el elemento i-esimo, todos los elementos anteriores (hasta el (i-1)ésimo) se encuentran ya ordenados. Cuando termina la ejecución, **L** presenta los **n** elementos ordenados de menor a mayor. Se puede demostrar que el tiempo promedio que demora el algoritmo Insert

Sort en ordenar un arreglo es de $O(n^2)$, y que el algoritmo es muy rápido si el arreglo está semi-ordenado. En particular, si el arreglo está ordenado el algoritmo de inserción demora $O(n)$.

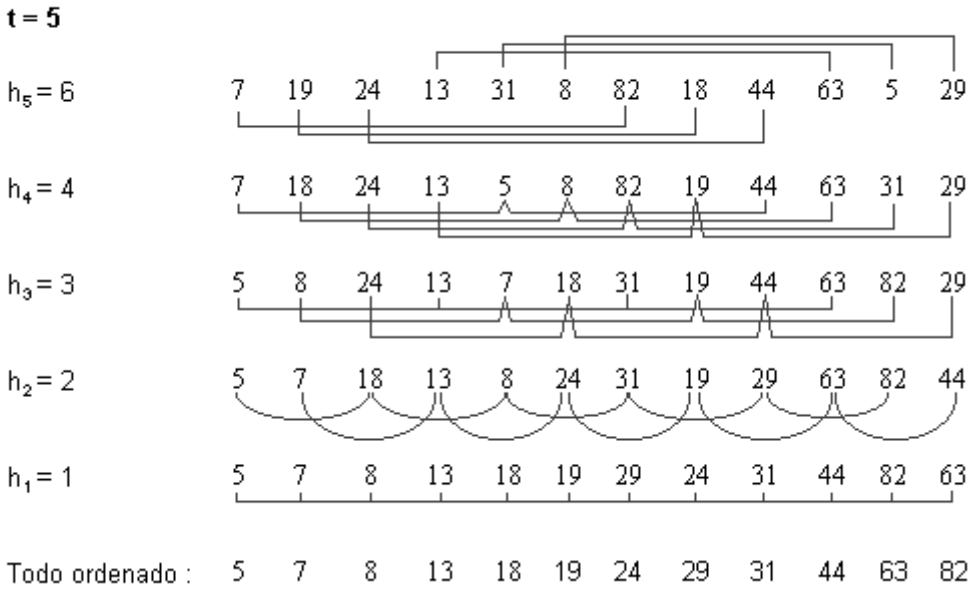
El algoritmo de Shellsort actúa de manera similar, pero entre elementos separados a una distancia que va disminuyendo en cada iteración. Lo que obtenemos con esto es que cuando se emplee el algoritmo tradicional de Insert Sort, los elementos ya están ordenados relativamente, y así la cantidad de comparaciones que tiene que hacer es mucho menor. Eligiendo las distancias adecuadas, el algoritmo de Shellsort presenta un mejor orden promedio que Insert Sort.

A continuación se presenta el algoritmo de Shellsort en pseudocódigo:

```
for s ← t to 1 by – 1   {s es el índice del incremento o distancia}
    h ← dist[s]         {h es el incremento o distancia entre elementos a comparar}
    for j ← h + 1 to n  {j empieza en el segundo elemento del arreglo original}
        KEY ← L[j]; i ← j – h
        while i>0 and L[i]>KEY do
            L[i + h] ← L[i]; i ← j – h
        end
        L[i + h] ← KEY
    end
end
```

donde **t** corresponde al número de iteraciones, **dist** es un arreglo que contiene la distancia entre elementos a comparar para cada iteración (la primera en el arreglo siempre es igual a 1 y corresponde a la última distancia ocupada por el algoritmo, momento en el cual es equivalente a Insert Sort), **n** es el número de elementos a ordenar y **L** es originalmente el arreglo desordenado. Luego de su ejecución, en **L** están ordenados los **n** elementos de menor a mayor.

Para que esto quede más claro conviene ilustrar el problema y su solución con el siguiente ejemplo:



La razón para usar el método de inserción para ordenar los subarreglos de las etapas sucesivas de Shellsort, es por su sencillez y por el hecho que el trabajo realizado en etapas previas se mantiene al ordenar la etapa actual. En efecto, si se define como arreglo **t-ordenado** aquel en el cual los elementos que se encuentran a distancia **t**, dentro del arreglo, están ordenados, se puede demostrar que si un arreglo **h-ordenado** es transformado a **k-ordenado** (con **k<h**), se mantiene **h-ordenado**,. Esto permite que a medida que el paso se va haciendo más pequeño, los elementos ya están bastante ordenados globalmente y al realizar la última etapa con paso **h=1** (que es una ordenación por Inserción), prácticamente se lleva a cabo en una sola pasada ($O(n)$), dado todo el trabajo realizado en las iteraciones previas. Este algoritmo es un claro ejemplo de como una pequeña modificación a un algoritmo lento lo puede convertir en uno bastante más rápido.

El número de comparaciones efectuado por Shellsort es una función de las secuencias de incremento o distancias que utiliza. Su análisis es extremadamente difícil y requiere respuestas a varios problemas matemáticos todavía no resueltos. Por lo tanto, la mejor secuencia posible de incrementos aún no ha sido determinada, aunque algunos casos específicos han sido estudiados. Por ejemplo, para la serie

$h_2 = 1.72\sqrt[3]{n}$ y $h_1 = 1$ se ha demostrado que el tiempo de ejecución es $O(n^{\frac{5}{3}})$, lo que puede parecer sorprendente: haciendo una sola pasada previa se mejora el algoritmo de inserción que en promedio es $O(n^2)$.

Las serie de pasos almacenados en el arreglo **dist** tiene la característica de ser **independiente** del tamaño del arreglo al cual se aplica durante el desarrollo del algoritmo, por lo que sus valores pueden ser pre-calculados antes de ejecutar el ordenamiento. En general, al programar el algoritmo esto no es así, y los valores de los pasos se calculan cada vez que una iteración termina, lo cual no influye en nada en la eficiencia del algoritmo: durante las pruebas realizadas en el estudio, las diferencias de tiempo entre tener pre-calculados los pasos e irlos calculando en cada iteración no superaron el margen de error de las pruebas (menos del 0.01% del tiempo promedio obtenido).

Sin perjuicio de lo anterior, existen otras series de pasos que son **dependientes** del tamaño del arreglo al cual se aplican, como por ejemplo $h_i = \left\lfloor \frac{h_{i+1}}{2} \right\rfloor$, partiendo con $h_t = n$ y terminando cuando $h_1 = 1$. El comportamiento de ésta serie no es muy diferente a la de $h_i = 2^{i-1}$, pero si se cambia el factor de división 2 por uno ligeramente superior, digamos 2.2, el algoritmo se acelera notablemente. En la literatura se recomienda **no** utilizar este tipo de series, pues si antes el análisis matemático era muy complicado, con este tipo de series dependientes es imposible realizarlo. Sin embargo, este estudio demuestra empíricamente que algunas series de este tipo pueden llegar a ser muy eficientes si se le aplica al algoritmo una pequeña optimización, la cual se analiza a continuación.

3. Evitando el peor caso

Durante el desarrollo del estudio se probó el rendimiento de la siguiente serie: $h_i = \left\lfloor \frac{h_{i+1}}{2.4} \right\rfloor$, partiendo con $h_t = \left\lfloor \frac{n}{4.8} \right\rfloor$ y terminando cuando $h_1 = 1$. Para esto, se generaron arreglos con elementos

al azar, partiendo desde un tamaño de 100 hasta un tamaño de 100.000 y agregando 100 elementos en cada iteración del experimento (¡un test bastante exhaustivo!). El algoritmo funcionaba bastante bien, pero resaltaron dos problemas: las diferencias en el tiempo utilizado entre cantidades de elementos muy similares eran grandes, y el algoritmo se comportaba estrepitosamente mal con arreglos de exactamente 64.700 elementos (un orden de magnitud de diferencia con respecto a arreglos de 64.600 y 64.800 elementos), lo cual era bastante extraño. La raíz del problema radicaba, curiosamente, en los mismos inicios del algoritmo:

Se ha demostrado que el peor caso de Shellsort ocupando los incrementos de Shell, es decir $h_i = 2^{i-1}$ (h es potencia de 2), ejecuta un número de comparaciones $O(n^2)$, si los elementos están distribuidos originalmente en el arreglo de tal manera que la mitad mayor se encuentre en celdas pares y la mitad menor en celdas impares. Dado que todos los incrementos son pares exceptuando el último, cuando se ha llegado a la última iteración, con el único incremento impar igual a 1, continúan estando todos los elementos mayores en las celdas pares y los menores en las celdas impares. De este modo en el último paso (equivalente al algoritmo de Insert Sort) se deben realizar una gran cantidad de comparaciones (recordemos que Insert Sort es, en promedio, $O(n^2)$).

Efectivamente esto era lo que ocurría con la serie de pasos estudiada. Todas las aproximaciones de $h_i = \left\lfloor \frac{h_{i+1}}{2.4} \right\rfloor$ partiendo con $h_t = \left\lfloor \frac{64.700}{4.8} \right\rfloor$ dan números pares (y es el único valor entre 100 y 100.000, múltiplos de 100, en el que ocurre este fenómeno), con lo que el último paso, igual a un Insert Sort, debía realizar una gran cantidad de trabajo al no haberse comparado nunca las posiciones impares del arreglo con las pares, lo que degradaba notablemente el rendimiento del algoritmo.

Realizando la pequeña optimización de restarle uno al paso si éste resulta par, se obtiene un algoritmo mucho más homogéneo y eficiente en su comportamiento. La aplicación de esta optimización debiera realizarse a todas las series de pasos, sean éstas dependientes del tamaño del arreglo o no.

4. Comparación del rendimiento de distintas series de pasos

Las series implementadas para realizar el estudio (y que fueron pre-seleccionadas por ser las más eficientes), son las siguientes:

1. $h_i = 3 * h_{i-1} + 1$ (h = 1, 4, 13, 40, 121, 364, 1093, …), en donde el paso mayor es el número de la serie más cercano a $\left\lfloor \frac{n}{9} \right\rfloor$.

2. $h_i = 2^i - 1$, en donde el paso mayor es el número de la serie más cercano a n.

3. $h_i = \frac{3^i - 1}{2}$, en donde el paso mayor es el número de la serie que se encuentra dos posiciones antes del más cercano a n .

4. $h_i = \left\lfloor \frac{h_{i+1}}{k} \right\rfloor$ (restándole 1 si h_i resulta par), donde: (a) $k = 2.2$, (b) $k = 2.4$, y el paso mayor es $h_t = \left\lfloor \frac{n}{2 * k} \right\rfloor$.

Se generaron arreglos de números enteros con valores aleatorios, cuya cantidad de elementos varió entre los 10.000 y 100.000 elementos (múltiplos de 5.000). Cada iteración del test consistió en repetir 30 veces el algoritmo con arreglos distintos pero con la misma cantidad de elementos, calculando luego el tiempo promedio que tomaba el algoritmo en ordenar el arreglo. Para verificar la validez de los datos, se calculó el intervalo de confianza **I** en el cual se encontraría la media real de los tiempos obtenidos, con un nivel de confianza del 95%:

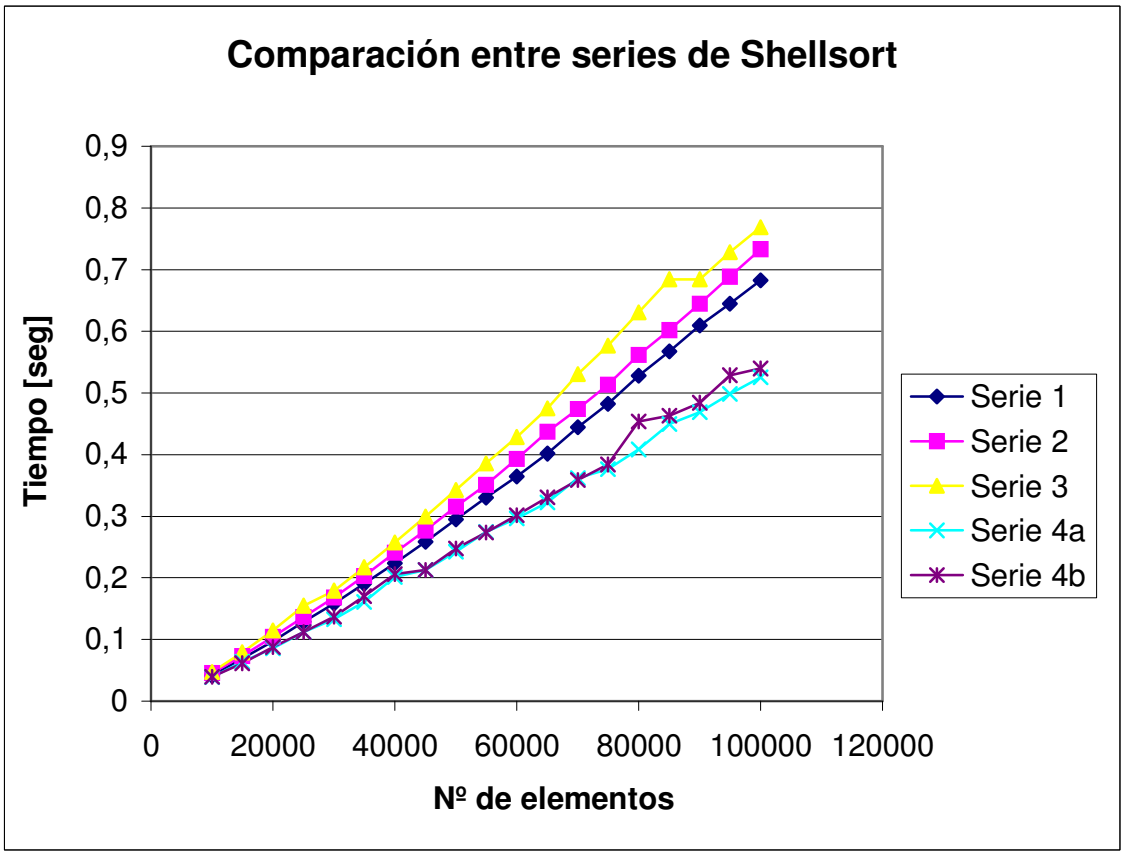
$$I = [\overline{X_n} - \frac{S_{n-1}}{\sqrt{n}} * 2.262, \overline{X_n} + \frac{S_{n-1}}{\sqrt{n}} * 2.262] \ , \ n = 30 \text{ para cada iteración.}$$

$$\overline{X_n} = \frac{1}{n} * \sum_{i=1}^{n} X_i \qquad \text{(media empírica)}$$

$$S_{n-1}^2 = \frac{1}{n-1} * \sum_{i=1}^{n} (X_i - \overline{X_n})^2 \quad \text{(varianza empírica)}$$

El test fue realizado en un PC con procesador Pentium 200 MMX y sistema operativo Linux (kernel 2.0.34). Los algoritmos fueron implementados en lenguaje C.

Los resultados obtenidos se muestran en el siguiente gráfico:

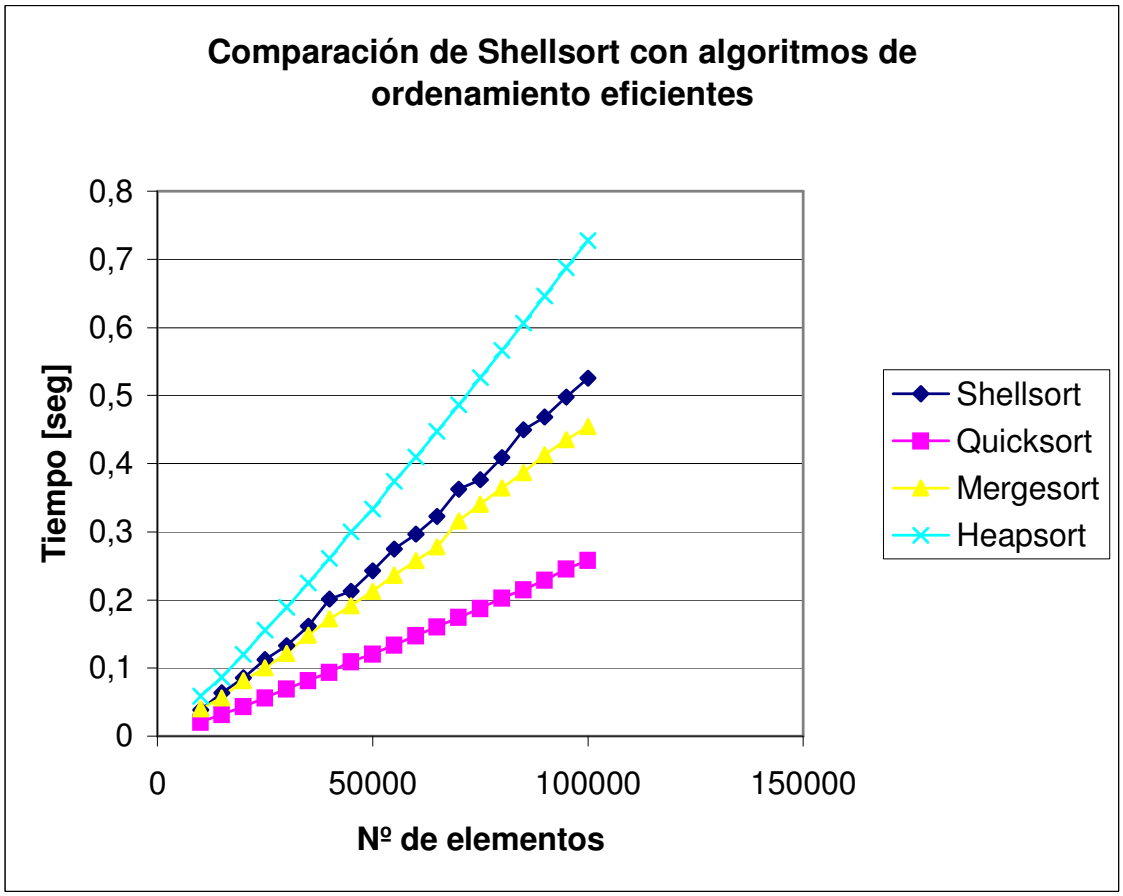


Los puntos representan el tiempo promedio obtenido por cada algoritmo. Los intervalos de confianza calculados poseen un ancho promedio del 2% del tiempo promedio obtenido, por lo que dichos tiempos se acercan bastante a la media real.

Se observa en el gráfico que las series dependientes del tamaño del arreglo superan ampliamente a la mejor serie independiente del arreglo. En particular, la serie de pasos $h_i = \left\lfloor \frac{h_{i+1}}{2.2} \right\rfloor$ reduce en un 24% el tiempo de la mejor serie clásica ($h_i = 3 * h_{i-1} + 1$) en arreglos con 100.000 elementos. La curva que

mejor aproxima a los datos obtenidos utilizando esta serie es $Y = e^{-13.6239043} * X^{1,1283644}$, es decir, el tiempo promedio que toma el algoritmo en ordenar es de $O(n^{\frac{9}{8}})$ aproximadamente.

Se realizó el mismo test pero ahora comparando Quicksort, Mergesort y Heapsort con la mejor serie de Shellsort, $h_i = \left\lfloor \frac{h_{i+1}}{2.2} \right\rfloor$. El resultado obtenido fue el siguiente:



Cabe destacar que Mergesort se demora lo mismo en ordenar un arreglo desordenado de 10.000 elementos que Shellsort, pero éste no ocupa espacio extra para realizar el ordenamiento. Quicksort sigue siendo el algoritmo más eficiente, pero hasta los 100.000 Shellsort sólo toma el doble de tiempo que Quicksort para ordenar el arreglo.

Resumiendo: para arreglos de menos de 100.000 elementos Shellsort se comporta **mejor** que Heapsort, que tiene complejidad en tiempo $O(n*\ln(n))$ y opera "in place"; Mergesort se comporta un poco mejor que Shellsort en tiempo de ejecución, pero tiene la desventaja que no trabaja "in place"; Quicksort se comporta mucho mejor que Shellsort y también opera "in place", pero tiene la desventaja de tener un peor caso $O(n^2)$.

5. Conclusiones

Si bien Shellsort no es el algoritmo más eficiente para ordenar arreglos, comparado con la complejidad $O(n*\ln(n))$ de los algoritmos Quicksort, Mergesort y Heapsort, es un algoritmo mucho más fácil de programar. Su simplicidad radica en que deriva del algoritmo más simple para ordenar, Insert Sort. Además, su complejidad promedio en tiempo de $O(n^{\frac{9}{8}})$ ocupando la serie $h_i = \left\lfloor \frac{h_{i+1}}{2.2} \right\rfloor$, y su complejidad en espacio de $O(n)$, debido a que opera “in place”, lo hacen un buen candidato para resolver el problema de ordenamiento en conjuntos de menos de 100.000 elementos.

Es vital para la eficiencia del algoritmo que todos los elementos de la serie de pasos sean números impares, para lo cual basta con restarle 1 al paso si éste es par. Con esta pequeña modificación se reduce el tiempo promedio de ejecución y su varianza. Además, el estudio demuestra empíricamente que algunas series dependientes del tamaño del arreglo reducen el tiempo de ejecución del algoritmo con respecto a las series clásicas. Sin embargo, aún no se sabe con certeza cuál es la eficiencia real del algoritmo, y es muy posible que existan series de pasos que reduzcan los tiempos de ejecución obtenidos en los tests descritos.

6. Referencias

- Mark Allen Weiss, *Data Structures and Algorithm Analysis*, Benjamin/Cummings, 2ª ed., 1995, pág. 216-220.

- Micha Hofri, *Analysis of Algorithms: Computational Methods and Mathematical Tools*, Oxford University Press, 1995, pág. 431-437.
- Robert Sedgewick, *Algorithms in C++*, Addison-Wesley, 1992, pág. 107-112.
- Sara Baase, *Computer Algorithms: Introduction to Design and Analysis*, Addison-Wesley, 1978, pág. 73-76.