Ampliación de Programación

Práctica 1

Algoritmos de ordenación

Sergio García Mondaray

Escuela Superior de Informática de Ciudad Real
Universidad de Castilla-La Mancha

Índice general

Algoritmo 1

Ordenación por inserción directa

El algoritmo de ordenación por inserción directa (*InsertionSort*) es un algoritmo relativamente sencillo, que se comporta razonablemente bien en gran cantidad de situaciones. Es uno de los algoritmos de ordenación más básicos, junto con el algoritmo de la burbuja (*BubbleSort*) y el de selección directa (*SelectionSort*). La cantidad de comparaciones que realiza está bastante equilibrada con respecto a los intercambios, es por ello que es considerado "el mejor", a efectos prácticos, de entre los mencionados algoritmos básicos.

Una diferencia importante con los algoritmos *BubbleSort* y *SelectionSort* es que, a diferencia de cómo sucede en éstos, con el algoritmo de inserción directa el array no está totalmente ordenado hasta que no finaliza todo el proceso. Además es importante resaltar que en cada pasada no se recorre todo el array (sino sólo los elementos analizados hasta ese instante), por lo que el algoritmo de InsertionSort es considerado un algoritmo on-line; es decir, que no necesita disponer de todos los elementos a ordenar desde el principio, sino que puede aceptarlos de uno en uno y procesarlos a medida que los recibe.

1.1 Cómo funciona

El mecanismo de trabajo de este algoritmo es realizar varias pasadas sobre el array a ordenar. En cada pasada se analiza un elemento y se intenta encontrar su orden relativo entre los analizados en pasos anteriores. De esta manera se mantiene en todo momento una lista ordenada, y cada elemento a analizar se desplaza a la posición que le corresponda en esa lista.

En total se realizan `n-1` pasadas (considerando que `n` es el número de elementos del vector a ordenar). El motivo es que el algoritmo empieza a trabajar con el segundo elemento del vector (puesto que el primero se considera ordenado, al ser una lista de un único elemento).

En cada pasada escogemos un elemento, por orden (en la primera pasada el segundo elemento, en la segunda pasada el tercero, y así sucesivamente, hasta que en la pasada `n-1` escogemos el elemento `n`) y recorremos el array hacia atrás, buscando el orden relativo del elemento en cuestión entre los anteriores (que ya fueron analizados en pasos anteriores, por lo que forman una lista ordenada). Cada vez que encontremos un elemento mayor que él realizaremos un intercambio.

1.2 Pseudocódigo e implementación

1.2.1 Pseudocódigo del algoritmo

Llamemos `A` al vector a ordenar.

```
Inicio
    n <- longitud del vector
    Para i <- 2 mientras que i <= n
        v <- A[i]
        j <- i-1
        Mientras j >= 0 y A[j] > v
            A[j+1] <- A[j]
            j <- j-1
        Fin mientras
        A[j+1] <- v
    Fin para
Fin
```

1.2.2 Implementación (en lenguaje C)

```
1  void insercionDirecta(int vector[], int n){
2      int i,j,v;
3      for(i = 2; i <= n; i++){
4          v = *(vector + i);
5          j = i - 1;
6          while((j >= 0) && *(vector + j) > v){
7              *(vector + (j+1)) = *(vector + j);
8              j--;
9          }
10         *(vector + (j+1)) = v;
11     }
12 }
```

1.3 Análisis del algoritmo

Para estudiar el algoritmo llevaremos a cabo dos tipos de análisis. En primer lugar hallaremos la complejidad del mismo mediante métodos numéricos, después estudiaremos los resultados de ordenar vectores de diferentes longitudes en varias máquinas, para ver cómo se comporta.

1.3.1 Complejidad analítica

Para estudiar la complejidad analítica de nuestro algoritmo contaremos el número de operaciones básicas (asignación, comparación, suma, multiplicación...) que lleva a cabo en el peor de los casos, es decir, hallaremos su función de complejidad O. Trabajaremos sobre el algoritmo:

```
1  Inicio
2      n <- longitud del vector
3      Para i <- 2 mientras que i <= n
4          v <- A[i]
5          j <- i-1
6          Mientras j >= 0 y A[j] > v
7              A[j+1] <- A[j]
8              j <- j-1
9          Fin mientras
10         A[j+1] <- v
11     Fin para
12 Fin
```

Analicemos el número de operaciones básicas que se realizan en el peor caso, teniendo en cuenta que el peor caso se dará cuando el vector esté ordenado de manera inversa, es decir, descendentemente:

$$\sum_{i=2}^{n}\left(1+1+\sum_{j=i-1}^{A[j]>v}(1+1)+1\right)=3\cdot\sum_{i=2}^{n}1+\sum_{i=2}^{n}\left(2\cdot\sum_{j=i-1}^{A[j]>v}1\right)=3(n-1)+2\cdot\sum_{i=2}^{n}(i-1)=*$$

$$*=3(n-1)+2\frac{1+(n-1)}{2}(n-1)=3(n-1)+n(n-1)=n^2+2n-3\in O(n^2)$$

Por lo tanto, la complejidad del algoritmo de ordenación por inserción directa es de $\boxed{O(n^2)}$.

1.3.2 Complejidad experimental

Hemos cronometrado el tiempo empleado en ordenar varios vectores en 3 máquinas diferentes. Los vectores objeto de estudio han sido generados aleatoriamente, y son de 10000, 100000 y 1000000 elementos, respectivamente. Las computadoras bajo las que se ha llevado a cabo el experimento son las siguientes:

- HP Compaq V6355eu. Procesador: AMD Turion 64 x2 TL-56 (1.6 GHz). RAM: 2 GB.
- HP Compaq V6515es. Procesador: AMD Athlon 64 x2 (1.6 GHz). RAM: 2 GB.
- HP DV5 1113es. Procesador Intel Core 2 Duo (2.2 GHz). RAM 3 GB.

La hoja de resultados es la siguiente:

```
HP Compaq V6355eu. Procesador AMD Turion 64 x2 TL-56 (1.8 GHz). RAM 2 GB
------------------------------------------------------------------------
Insercion directa (10000 elementos):   0.35000 segundos (0.00583 minutos)
Insercion directa (100000 elementos):  28.3500 segundos (0.47250 minutos)
Insercion directa (1000000 elementos): 2965.90 segundos (49.4316 minutos)

HP Compaq V6515es. Procesador AMD Athlon 64 x2 1.6 GHz. RAM 2 GB
------------------------------------------------------------------------
Insercion directa (10000 elementos):   0.63000 segundos (0.01050 minutos)
Insercion directa (100000 elementos):  31.7500 segundos (0.52917 minutos)
Insercion directa (1000000 elementos): 3228.62 segundos (53.8103 minutos)

HP DV5 1113es. Procesador Intel Core 2 Duo 2.2 GHz. RAM 3 GB
------------------------------------------------------------------------
Insercion directa (10000 elementos):   0.17000 segundos (0.00283 minutos)
Insercion directa (100000 elementos):  16.7900 segundos (0.27983 minutos)
Insercion directa (1000000 elementos): 1694.59 segundos (28.2431 minutos)
```

Algoritmo 2

Ordenación por el método de Shell

Este algoritmo debe su nombre al ingeniero y matemático estadounidense Donald Shell, y fue publicado en la revista *Communications of the ACM* en 1959. Es un algoritmo de ordenación interna muy sencillo e ingenioso, basado en comparaciones e intercambios, y con resultados radicalmente mejores que los que se pueden conseguir con el triplete de algoritmos básicos: burbuja, selección directa e inserción directa.

El método de Shell es el mejor algoritmo de ordenación *in-situ*[1].

2.1 Cómo funciona

Consideremos la necesidad de ordenar el vector [74, 14, 21, 44, 38, 97, 11, 78, 65, 88, 30].

Si recordamos, en el algoritmo de inserción directa recorríamos uno por uno todos los elementos, desde el segundo al último, desplazando hacia la izquierda cada elemento hasta su posición entre los anteriores. Pues bien, si nos fijamos en que los elementos están comprendidos entre el 11 y el 97 y que parecen más o menos uniformemente distribuídos, intuitivamente podemos estar casi seguros de que el 97 estará por el final, el 88 también, el 14 y el 11 estarán por el principio, y el 44 y 66 más o menos por la mitad.

Si fuésemos capaces de "mover" los elementos más grandes hacia el final del array de un gran salto (con tan poco esfuerzo como precisión), los más pequeños al principio y dejar los medianos por el medio, no habríamos ordenado el array, pero habríamos conseguido situar cada elemento "cerca" de su ubicación definitiva.

Esta es la idea del método de ordenación de Shell: si conseguimos desplazar *a grosso modo* los elementos más grandes hacia el final y los más pequeños hacia el principio con poco esfuerzo, podremos aplicar después el algoritmo de Inserción Directa de manera más eficiente (puesto que cada elemento tendrá que desplazarse pocos lugares, al estar "cerca" de su posición final).

Para que un ordenador pueda llevar a cabo esta labor, sustituiremos la idea de "intuitivo" por un procedimiento mecánico algo ingenioso: dividimos el array original en varios sub-arrays tales que cada elemento esté separado k elementos del anterior. Se debe empezar por $k = \frac{n}{2}$, siendo n el número de elementos del array, y utilizando siempre la división entera, variando k por sucesivas divisiones por 2 hasta llegar a $k = 1$. Los elementos de cada sub-array son

[1] Se consideran algoritmos de ordenación *in-situ* a aquellos en los que la cantidad de memoria adicional –sin contar la memoria ocupada por los propios datos– que necesita es constante, sea cual sea la cantidad de datos a ordenar. Los 3 algoritmos de ordenación básicos (burbuja, insercción y selección directa) son *in-situ*, por otro lado, los algoritmos como QuickSort, BinSort, HeapSort, etc no lo son, ya que cuantos más datos haya que ordenar más memoria adicional necesitan.

ordenados por Inserción Directa entre ellos. De esta manera habremos conseguido terminar la primera fase: acercar cada elemento a su posición definitiva, y podremos decir que el vector está *k-ordenado*.

Para continuar con el algoritmo, debemos ir reduciendo progresivamente k, dividiéndolo por 2 y k-ordenando os sub-arrays que nos salgan (saldrán k subarrays). Al llegar a $k = 1$ habremos terminado y tendremos el vector perfectamente ordenado.

2.2 Pseudocódigo e implementación

2.2.1 Pseudocódigo del algoritmo

El siguiente pseudocódigo es deducido de la explicación del apartado 2.1. No es la versión en pseudocódigo más conocida, puesto que en la mayoría de libros el algoritmo se realiza con 3 bucles en lugar de 4. Sin embargo, puesto que computacionalmente no hay diferencia en el rendimiento (lo único es que el código queda más compacto) explicaremos la versión de 4 bucles, que es más intuitiva:

Consideramos que el vector a ordenar es `A`, y que consta de `n` elementos.

```
Inicio
  k <- n/2
  Mientras(k >= 1) hacer:
      Para subarray <- 0, mientras que subarray < k, subarray++:
          Para i <- k + subarray, mientras i < n, i = i + k:
            v <- A[i]
            j <- i - k
            Mientras(j >= 0 y A[j] > v) hacer:
              A[j + k] <- A[j]
              j = j - k
            Fin mientras
            A[j + k] <- v
          Fin Para
      Fin Para
  Fin mientras
Fin
```

2.2.2 Implementación (en lenguaje C)

Como hemos mencionado, esta no es la versión más común de este método de ordenación, pero sí la más intuitiva. Es por ello que lo he implementado así (siguiendo el pseudocódigo anterior), porque es mucho más intuitivo que otras versiones del algoritmo:

```
void shell(int vector[], int n){
    k = n/2;
    while(k >= 1){
        for(subarray = 0; subarray < k; subarray++){
            for(i = k + subarray; i < n; i += k){
                v = *(vector + i);
                j = i - k;
                while(j >= 0 && *(vector + j) > v){
                    *(vector + j + k) = *(vector + j);
                    j -= k;
                }
```

```
                *(vector + j + k) = v;
            }
        }
        k = k/2;
    }
}
```

2.3 Análisis del algoritmo

Para estudiar el algoritmo llevaremos a cabo dos tipos de análisis. En primer lugar hallaremos la complejidad del mismo mediante métodos numéricos, después estudiaremos los resultados de ordenar vectores de diferentes longitudes en varias máquinas, para ver cómo se comporta.

2.3.1 Complejidad analítica

Para estudiar la complejidad analítica de nuestro algoritmo contaremos el número de operaciones básicas (asignación, comparación, suma, multiplicación...) que lleva a cabo en el peor de los casos, es decir, hallaremos su función de complejidad O. Trabajaremos sobre el algoritmo:

```
Inicio
  k <- n/2
  Mientras(k >= 1) hacer:
      Para subarray <- 0, mientras que subarray < k, subarray++:
          Para i <- k + subarray, mientras i < n, i = i + k:
            v <- A[i]
            j <- i - k
            Mientras(j >= 0 y A[j] > v) hacer:
              A[j + k] <- A[j]
              j = j - k
            Fin mientras
            A[j + k] <- v
          Fin Para
      Fin Para
  Fin mientras
Fin
```

Analicemos el número de operaciones básicas que se realizan en el peor caso:

$$\sum_{k=n/2}^{k<1} \left(\sum_{s=0}^{s\geq k} \left(\sum_{i=k+s}^{n} (2) + 1 \right) \right) \in O(n^2)$$

Por lo tanto, la complejidad del algoritmo de ordenación por el método de Shell es de $\boxed{O(n^2)}$.

2.3.2 Complejidad experimental

Hemos cronometrado el tiempo empleado en ordenar varios vectores en 3 máquinas diferentes. Los vectores objeto de estudio han sido generados aleatoriamente, y son de 10000, 100000 y 1000000 elementos, respectivamente. Las computadoras bajo las que se ha llevado a cabo el experimento son las siguientes:

- HP Compaq V6355eu. Procesador: AMD Turion 64 x2 TL-56 (1.6 GHz). RAM: 2 GB.
- HP Compaq V6515es. Procesador: AMD Athlon 64 x2 (1.6 GHz). RAM: 2 GB.
- HP DV5 1113es. Procesador Intel Core 2 Duo (2.2 GHz). RAM 3 GB.

La hoja de resultados es la siguiente:

```
HP Compaq V6355eu. Procesador AMD Turion 64 x2 TL-56 (1.8 GHz). RAM 2 GB
------------------------------------------------------------------------
Metodo Shell (10000 elementos):        0.01000 segundos (0.00017 minutos)
Metodo Shell (100000 elementos):       0.07000 segundos (0.00117 minutos)
Metodo Shell (1000000 elementos):      1.36000 segundos (0.02267 minutos)

HP Compaq V6515es. Procesador AMD Athlon 64 x2 1.6 GHz. RAM 2 GB
-------------------------------------------------------------------------
Metodo Shell (10000 elementos):        0.01000 segundos (0.00017 minutos)
Metodo Shell (100000 elementos):       0.10000 segundos (0.00167 minutos)
Metodo Shell (1000000 elementos):      1.54000 segundos (0.02567 minutos)

HP DV5 1113es. Procesador Intel Core 2 Duo 2.2 GHz. RAM 3 GB
-------------------------------------------------------------------------
Metodo Shell (10000 elementos):        0.00000 segundos (0.00000 minutos)
Metodo Shell (100000 elementos):       0.06000 segundos (0.00100 minutos)
Metodo Shell (1000000 elementos):      0.90000 segundos (0.01500 minutos)
```

Algoritmo 3

Ordenación rápida: Quicksort

El método de QuickSort (ordenamiento rápido) es un algoritmo basado en la técnica de divide y vencerás. Es la técnica de ordenamiento más rápida conocida, y fue desarrollada por C. Antony R. Hoare en 1960. El algoritmo original es recursivo, pero se utilizan versiones iterativas para mejorar su rendimiento.

3.1 Cómo funciona

El fundamento, como se ha mencionado, es la técnica de *divide y vencerás*, y el algoritmo funciona de la siguiente manera:

1. Se elige un elemento de la lista a ordenar, al que llamamos *pivote*. La elección de este elemento se puede realizar de diferentes formas, pero aquí tomaremos el primer elemento de la lista.

2. Recolocamos todos los demás elementos de la lista según les corresponda, antes (si son menores) o después (si son mayores) que el pivote elegido. De esta manera el pivote queda en su posición definitiva.

3. Ahora la lista queda separada en dos sublistas (la anterior al pivote y la posterior). Repetimos el proceso para cada una de estas sublistas de manera recursiva, mientras que tengan más de un elemento.

4. Terminado el proceso, todos los elementos estarán ordenados.

Podemos apreciar que la eficiencia del algoritmo dependerá de la posición en la que termine el pivote elegido. En el mejor caso, el pivote quedará colocado en el centro de la lista, dividiéndo la lista en 2 sublistas de igual tamaño. Por el contrario, en el peor caso el pivote terminará en un extremo de la lista (se suele dar este caso en listas ya ordenadas, o casi ordenadas, aunque depende de cómo seleccionemos nuestro pivote).

Sobre cómo elegir el elemento pivote existen discusiones: Una de las propuestas es elegir un elemento al azar, lo cual ofrece la ventaja de no necesitar ningún cálculo adicional. Como alternativa está la propuesta de recorrer la lista de antemano para conocer qué elemento ocupará la posición central y seleccionarlo. Una solución intermedia es la que propone tomar 3 elementos cualesquiera y compararlos entre ellos, eligiendo el valor del medio como pivote. Nuestra alternativa será más sencilla, y tomaremos siempre el primer elemento del array como pivote.

3.2 Pseudocódigo e implementación

3.2.1 Pseudocódigo del algoritmo

Destaquemos que `izq` y `der` son punteros a elementos del array. Cuando escribimos `$izq` o `$der` hacemos referencia al contenido de la zona de memoria donde `izq` o `der` apuntan, respectivamente.

```
Inicio(izq, der)
    Si der>izq
        pivote <- $izq
        ult <- der
        pri <- izq
        Mientras (izq < der) haz
          Mientras($izq <= pivote y izq < der+1) haz izq = izq+1
          Mientras($der > pivote y der > izq-1) haz der = der-1
          Si(izq < der) intercambiar(izq,der)
        Fin mientras
        Intercambiar(pri,der)
        Llamada recursiva con (pri, der-1)
        Llamada recursiva con (der+1, ult)
    Fin si
Fin
```

3.2.2 Implementación (en lenguaje C)

```
void quicksort(int* izq, int* der){

    if(der<izq) return;
        int pivot=*izq;
        int* ult=der;
        int* pri=izq;

        while(izq<der)
        {
                while(*izq<=pivot && izq<der+1) izq++;
                while(*der>pivot && der>izq-1) der--;
                if(izq<der) swap(izq,der);
        }
        swap(pri,der);
        quicksort(pri,der-1);
        quicksort(der+1,ult);

}

void swap(int* a, int* b){
        int temp=*a;
        *a=*b;
        *b=temp;
}
```

3.3 Análisis del algoritmo

3.3.1 Complejidad analítica

Para calcular la complejidad del algoritmo QuickSort analíticamente, estudiemos el número de operaciones básicas que realiza fijándonos en su pseudocódigo:

```
Inicio(izq, der)
    Si der>izq
        pivote <- $izq
        ult <- der
        pri <- izq
        Mientras (izq < der) haz
          Mientras($izq <= pivote y izq < der+1) haz izq = izq+1
          Mientras($der > pivote y der > izq-1) haz der = der-1
          Si(izq < der) intercambiar(izq,der)
        Fin mientras
        Intercambiar(pri,der)
        Llamada recursiva con (pri, der-1)
        Llamada recursiva con (der+1, ult)
    Fin si
Fin
```

En líneas generales, atendiendo a la recursión y al número de iteraciones del bucle Mientras, tendremos:

$$T_n = \underbrace{n}_{\text{Por recorrer el vector}} + \underbrace{2 \cdot T_{n/2}}_{\text{Por las dos llamadas recursivas}} + \underbrace{c}_{\text{Gasto adicional}}$$

Ahora podemos calcular la complejidad:

$$T_n - 2T_{n/2} = n + c \overset{n=2k}{\Longrightarrow} T_{2^k} - 2T_{2^{k-1}} = 2^k + c \Longrightarrow (x-2)(x-2)(x-1) = 0$$

$$T_{2^k} = c_{11}k^0 2^k + c_{12}k^1 2^k + c_2 k^0 1^k = c_{11}n + c_{12}\log_2(n) \cdot n + c_2 \in O(n \cdot \log_2(n))$$

Por lo tanto, la complejidad del algoritmo de ordenación rápida, o QuickSort, es de $\boxed{O(n \cdot \log_2(n))}$.

3.3.2 Complejidad experimental

Hemos cronometrado el tiempo empleado en ordenar varios vectores en 3 máquinas diferentes. Los vectores objeto de estudio han sido generados aleatoriamente, y son de 10000, 100000 y 1000000 elementos, respectivamente. Las computadoras bajo las que se ha llevado a cabo el experimento son las siguientes:

- HP Compaq V6355eu. Procesador: AMD Turion 64 x2 TL-56 (1.6 GHz). RAM: 2 GB.
- HP Compaq V6515es. Procesador: AMD Athlon 64 x2 (1.6 GHz). RAM: 2 GB.
- HP DV5 1113es. Procesador Intel Core 2 Duo (2.2 GHz). RAM 3 GB.

La hoja de resultados es la siguiente:

```
HP Compaq V6355eu. Procesador AMD Turion 64 x2 TL-56 (1.8 GHz). RAM 2 GB
-------------------------------------------------------------------------
Quicksort (10000 elementos):           0.00000 segundos (0.00000 minutos)
Quicksort (100000 elementos):          0.03000 segundos (0.00050 minutos)
Quicksort (1000000 elementos):         0.34000 segundos (0.00567 minutos)

HP Compaq V6515es. Procesador AMD Athlon 64 x2 1.6 GHz. RAM 2 GB
-------------------------------------------------------------------------
Quicksort (10000 elementos):           0.01000 segundos (0.00017 minutos)
Quicksort (100000 elementos):          0.03000 segundos (0.00050 minutos)
Quicksort (1000000 elementos):         0.38000 segundos (0.00633 minutos)

HP DV5 1113es. Procesador Intel Core 2 Duo 2.2 GHz. RAM 3 GB
-------------------------------------------------------------------------
Quicksort (10000 elementos):           0.01000 segundos (0.00017 minutos)
Quicksort (100000 elementos):          0.02000 segundos (0.00033 minutos)
Quicksort (1000000 elementos):         0.28000 segundos (0.00467 minutos)
```