Anexo 7. Algoritmos de búsqueda y ordenación

A. J. Pérez

Algoritmos de búsqueda

En este apartado se van a ver una serie de algoritmos clásicos y cuyo conocimiento es fundamental para cualquier programador:

- Algoritmos de búsqueda:
  - ♦ Para datos no ordenados:
    - Búsqueda lineal o secuencial
  - ♦ Para datos ordenados:
    - Búsqueda lineal o secuencial optimizada con ruptura
    - Búsqueda binaria o dicotómica
    - Búsqueda binaria recursiva
- Algoritmos de ordenación:
  - Método de selección directa
  - Método de inserción directa (baraja)
  - Método de intercambio directo (burbuja)
  - Método de intercambio directo con test (switch)
  - Método de intercambio directo (sacudida)
  - Método shell
  - Método quick sort (recursivo)

Búsqueda lineal

Dado un array y un valor del mismo tipo que sus elementos, la búsqueda consiste en determinar si ese valor está en el array y qué posición ocupa.

Para los distintos métodos, aplicaremos los algoritmos a un vector *V* de *n* elementos. X será la variable que contendrá el valor a buscar en el vector V, y utilizaremos la variable IND como contador asociado a los bucles.

Búsqueda lineal en un vector

Se recorre el vector de izquierda a derecha hasta encontrar un componente cuyo valor coincida con el buscado o hasta que se acabe el vector. En este último caso, el algoritmo debe indicar la no existencia de dicho valor.

```
IND ← 1
Mientras V[IND] <> X y IND < n hacer
        IND ← IND + 1
        Si V[IND] = X entonces
                Escribir “Encontrado en la posición “, IND
        Si no
                escribir “No existe el valor buscado”
        finSi
FinMientras
```

Búsqueda lineal en un vector ordenado

Cuando el vector de búsqueda está ordenado se consigue un algoritmo más eficiente con sólo modificar la condición de terminación en el algoritmo anterior: una vez sobrepasado el valor buscado, no es necesario recorrer el resto del vector para saber que el valor no existe.

```
IND ← 1
Mientras V[IND] < X y IND < n hacer
        IND ← IND + 1
        Si V[IND] = X entonces
                Escribir “Encontrado en la posición “, IND
        Si no
                escribir “No existe el valor buscado”
        finSi
FinMientras
```

Búsqueda lineal en una tabla

Se realiza mediante el anidamiento de dos bucles tipo *hasta,* cuya finalización vendrá dada por la aparición del valor buscado o la terminación de la Tabla.

Sea una Tabla A de *m* filas y *n* columnas y un valor *X* a buscar en la Tabla *A*. Suponemos la Tabla *A* y la variable *X* ya cargadas previamente. *F* y *C* son dos contadores para el direccionamiento de las filas y las columnas.

```
F ← 0
repetir
        F ← F + 1
        C ← 0
        repetir
                C ← C + 1
        hasta A[F,C] = X o C = n
        si A[F,C] = X
                entonces escribir “Encontrado en fila ”, F, “ columna “, C
        sino
                escribir “No existe el valor buscado”
        finSi
hasta A[F,C] = X o F = m
```

Búsqueda binaria o dicotómica

La búsqueda binaria es un algoritmo que se aplica a una serie ordenada de datos. El proceso de búsqueda se delimita por dos posiciones: el límite inferior y el límite superior. El algoritmo empieza la búsqueda por el elemento que está almacenado en la mitad del intervalo de búsqueda. Si el elemento almacenado en la mitad del intervalo es mayor que el valor que se busca, entonces continúa la búsqueda en la primera mitad. Si el elemento almacenado en la mitad del intervalo es menor que el valor que se busca, entonces continúa la búsqueda en la segunda mitad. Si el elemento almacenado en la mitad del intervalo es igual que el valor que se busca, finaliza el proceso. En cada comparación, el algoritmo reduce el intervalo de búsqueda a la mitad. Si durante las sucesivas reducciones del intervalo de búsqueda el límite inferior es mayor que el límite superior, entonces el valor que se busca no está almacenado y finaliza el proceso.

Utilizaremos tres variables que nos indican en qué zona del vector nos encontramos: INF, SUP y MED. Suponemos el vector *V* ordenado ascendentemente.

```
INF ← 1
SUP ← n
MED ← (n + 1) DIV 2
```

```
mientras V[MED] <> X y INF < SUP hacer
        si V[MED] > X entonces
                SUP ← MED – 1
        sino
                INF ← MED + 1
        finSi
        MED ← (INF + SUP) DIV 2
finMientras
si V[CEN] = X entonces
        escribir “Encontrado en la posición “, MED
sino
        escribir “No existe el valor buscado”
finSi
```

Búsqueda binaria recursiva

Una versión recursiva, alternativa para la búsqueda binaria podría ser:

```
subprograma BUSCA_RECURSIVA (V, INF, SUP, BUSCADO)
Recibe:
        V vector de datos
        INF  posición elemento izquierdo
        SUP  posición elemento derecho
        BUSCADO
Devuelve:
        POSICION del elemento BUSCADO
Entorno:
        MED
Algoritmo:
        MED ← (INF  + SUP) DIV 2
        si INF > SUP  entonces
                devuelve -1
        sino
                if BUSCADO = V[MED] entonces
                        devuelve MED
                sino
                        if buscado < V[MED]
```

```
                    devuelve BUSCA_RECURSIVA (V, INF, MED-1, BUSCADO)
                sino
                    devuelve BUSCA_RECURSIVA (V, MED+1, SUP, BUSCADO)
                finSi
            finSi
        finSi

finSubprograma
```

Ordenación de vectores

La ordenación es la reagrupación de un conjunto de elementos en una secuencia específica.

Ordenación por selección directa

Consiste en repetir el siguiente proceso desde el primer elemento hasta el penúltimo: se selecciona la componente de menor valor de todas las situadas a la derecha de la tratada y se intercambia con ésta si es necesario.

En realidad se trata de sucesivas búsquedas del menor de los elementos que quedan por ordenar.

Ejemplo:

Secuencia inicial	3	2	4	1	2
Primera búsqueda	1	2	4	3	2
Segunda búsqueda	1	2	4	3	2
Tercera búsqueda	1	2	2	3	4
Cuarta búsqueda	1	2	2	3	4
Vector ordenado	1	2	2	3	4

Utilizamos K y AUX que guardan la posición y el valor del menor en cada búsqueda.

```
. . .

para I de 1 a n-1 hacer
        K ← I
        AUX ← V[I]
        para J de I + 1 a n hacer
            si V[J] < AUX entonces
                K ← J
```

```
                AUX ← V[J]
            finSi
        finPara
        V[K] ← V[I]
        V[I] ← AUX
finPara
...
```

Ordenación por inserción directa (método de la baraja)

Consiste en tomar los elementos del vector desde el segundo hasta el último y con cada uno de ellos repetir las siguientes operaciones:

1. Se saca del vector el elemento V[IND] (se almacena en la variable AUX)
2. Desde el anterior al que estamos tratando y hasta el primero, desplazamos un lugar a la derecha todos los que sean mayores, para buscar su hueco
3. Encontrado el hueco, lo insertamos en él

Hay que tener en cuenta que cuando tratamos un elemento, todos los anteriores se encuentran ordenados.

Ejemplo:

Secuencia inicial	3	2	4	1	2
Primer paso	3	2	4	1	2
Segundo paso	2	3	4	1	2
Tercer paso	2	3	4	1	2
Cuarto paso	1	2	3	4	2
Vector ordenado	1	2	2	3	4

AUX es una variable auxiliar del mismo tipo que los elementos del vector. I direcciona cada elemento a insertar. J direcciona los anteriores.

```
...
para I de 2 a n hacer
    AUX ← V[I]
    J ← I – 1
    mientras V[J] > AUX y J > 1 hacer
```

```
            V[J + 1] ← V[J]
            J ← J – 1
        finMientras
        si V[J] > AUX entonces
            V[J+1] ← V[J]
            V[J] ← AUX
        sino
            V[J + 1] ← AUX
        finSi
finPara
...
```

Ordenación por intercambio directo (método de la burbuja)

Consiste en recorrer sucesivamente el vector, comparando los elementos consecutivos e intercambiándose cuando están descolocados.

Ejemplo:

Secuencia inicial	3	2	4	1	2
Primera pasada2	3	1	2	4	
Segunda pasada	2	1	2	3	4
Tercera pasada	1	2	2	3	4
Cuarta pasada	1	2	2	3	4

Cada pasada asegura el posicionamiento de un elemento por la derecha; por tanto serán necesarias *n-1* pasadas para asegurar la ordenación.

P es un contador de pasadas, I direcciona los elementos y AUX se utiliza para los intercambios.

```
...
para P de 1 a n-1 hacer
        para I de 1 a n – P hacer
            si V[I] > V[I + 1] entonces
                AUX ← V[I]
```

```
                    V[I] ← V[I + 1]
                    V[I + 1] ← AUX
                finSi
        finPara
finPara
...
```

Ordenación por intercambio directo con test de comprobación (switch)

Es una mejora de los métodos de intercambio directo en la que se comprueba mediante un switch (interruptor) si el vector está totalmente ordenado después de cada pasada (es decir, si se han producido intercambios), terminando la ejecución en caso afirmativo.

```
...
P ← 1
SW ← CIERTO
mientras SW y P < n hacer
        SW ← FALSO
        para I de 1 a n – P hacer
                si V[I] > V[I + 1] entonces
                        AUX ← V[I]
                        V[I] ← V[I + 1]
                        V[I + 1] ← AUX
                        SW ← CIERTO
                finSi
        finPara
        P ← P + 1
finMientras
...
```

Ordenación por intercambio directo (método de la sacudida)

Consiste en mezclar las dos versiones del método de la burbuja alternativamente, de tal forma que se realiza una pasada de izquierda a derecha y a continuación otra de derecha a izquierda, recortándose los elementos a tratar por ambos lados del vector.

Se utilizan las variables IZQ y DER, que nos indican en cada momento la zona del vector que queda por ordenar. En cada pasada de izquierda a derecha recortamos una comparación por la derecha (DER ← DER – 1) y en cada pasada de derecha a izquierda lo hacemos por la izquierda (IZQ ← IZQ + 1).

Ejemplo:

Secuencia inicial		3	2	4	1	2
Primera pasada	2	3	1	2	4	
Segunda pasada		1	2	3	2	4
Tercera pasada	1	2	2	3	4	
Cuarta pasada	1	2	2	3	4	

• • •

```
IZQ ← 1
DER ← n
mientras IZQ <> DER hacer
        ** pasada de izquierda a derecha
        para I de IZQ a DER – 1 hacer
                si V[I] > V[I+1] entonces
                        AUX ← V[I]
                        V[I] ← V[I + 1]
                        V[I + 1] ← AUX
                finSi
        finPara
        DER ← DER – 1
        si IZQ <> DER entonces
                ** pasada de derecha a izquierda
                para I de DER a IZQ + 1 con incremento – 1 hacer
```

```
                si V[I – 1] > V[I] entonces
                        AUX ← V[I – 1]
                        V[I – 1] ← V[I]
                        V[I] ← AUX
                finSi
        finPara
        IZQ ← IZQ + 1
    finSi
finMientras
```

• • •

Este método aumenta su efectividad si se le incluye un switch para detectar si en una pasada, se encuentran todos los elementos ordenados.

• • •

```
IZQ ← 1
DER ← n
SW ← CIERTO
mientras IZQ <> DER y SW hacer
    SW ← FALSO
    ** pasada de izquierda a derecha
    para I de IZQ a DER – 1 hacer
        si V[I] > V[I+1] entonces
            AUX ← V[I]
            V[I] ← V[I + 1]
            V[I + 1] ← AUX
            SW ← CIERTO
        finSi
    finPara
    DER ← DER – 1
    si IZQ <> DER y SW entonces
        SW ← FALSO
        ** pasada de derecha a izquierda
```

```
                para I de DER a IZQ + 1 con incremento – 1 hacer
                        si V[I – 1] > V[I] entonces
                                AUX ← V[I – 1]
                                V[I – 1] ← V[I]
                                V[I] ← AUX
                                SW ← CIERTO
                        finSi
                finPara
                IZQ ← IZQ + 1
        finSi
finMientras
. . .
```

Ordenación por el método shell

La ordenación Shell debe el nombre a su inventor, D. L. Shell. Se suele denominar también ordenación por inserción con incrementos decrecientes. Se considera que el método Shell es una mejora de los métodos de inserción directa. En el algoritmo de inserción, cada elemento se compara con los elementos contiguos de su izquierda, uno tras otro. Si el elemento a insertar es el más pequeño hay que realizar muchas comparaciones antes de colocarlo en su lugar definitivo.

El algoritmo de Shell modifica los saltos contiguos resultantes de las comparaciones por saltos de mayor tamaño y con ello se consigue que la ordenación sea más rápida. Generalmente se toma como salto inicial n/2 (siendo n el número de elementos), luego se reduce el salto a la mitad en cada repetición hasta que el salto es de 1

Ejemplo:

6 1 5 2 3 4 0

El número de elementos que tiene el vector es 7, por lo que el salto inicial es 7/2 = 3. La siguiente tabla muestra el número de recorridos realizados en el vector con los saltos correspondiente.

Recorrido	Salto	Intercambios	Verctor
1	3	(6,2), (5,4), (6,0)	2 1 4 0 3 5 6
2	3	(2, 0)	0 1 4 2 3 5 6

3	3	ninguno	0 1 4 2 3 5 6
salto 3/2 = 1			
4	1	(4,2), (4,3)	0 1 2 3 4 5 6
5	1	ninguno	0 1 2 3 4 5 6

Los pasos a seguir por el algoritmo para una lista de n elementos son:

1. Dividir la lista original en n/2 grupos de dos, considerando un incremento o salto entre los elementos de n/2.

2. Clarificar cada grupo por separado, comparando las parejas de elementos, y si no están ordenados, se intercambian.

3. Se divide ahora la lista en la mitad de grupos (n/4), con un incremento o salto entre los elementos también mitad (n/4), y nuevamente se clasifica cada grupo por separado.

4. Así sucesivamente, se sigue dividiendo la lista en la mitad de grupos que en el recorrido anterior con un incremento o salto decreciente en la mitad que el salto anterior, y luego clasificando cada grupo por separado.

5. El algoritmo termina cuando se consigue que el tamaño del salto es 1.

Los recorridos por el vector está condicionados por el bucle:

```
INTERVALO ← N / 2
mientras INTERVALO > 0  hacer
        para I ← INTERVALO + 1 a N hacer
                J ← i - INTERVALO
                mientras J > 0 hacer
                        K ← J + intervalo
                        si  V[J] <= V[K] entonces
                                J ← − 1
                        sino
                                AUX ← V[J]
```

```
                    V[J]← V[K]
                    V[K] ← AUX
                    J ← J - INTERVALO
                finSi
            finMientras
        finPara
finMientras
```

Se comparan pares de elementos indexados por J y K, (V[J], V[K]), separados por un salto, INTERVALO. Así, si n = 8 el primer valor de intervalo = 4, y los índices I =5, J =1, K =6. Los siguientes valores de los índices son I = 6, J = 2, K = 7, y así hasta recorrer el vector.

Para realizar un nuevo recorrido de la lista con la mitad de grupos, el intervalo se hace la mitad: INTERVALO ←INTERVALO / 2 así se repiten los recorridos por la lista, mientras intervalo > 0.

Ordenación por el método mergeSort

Fue desarrollado por John Von Neumann. El algoritmo se basa en separar en dos partes el vector a ordenar, aplica la *técnica divide y vencerás*. El método trocea el vector original en dos partes de aproximadamente la mitad del tamaño; ordena cada parte por el mismo procedimiento y vuelve a juntar las dos partes ya ordenadas; mezclandolas manteniendo el orden.

```
subprograma MERGESORT (V, LENGTH)
Recibe:
        V vector de datos
        LENGTH longitud del vector
Devuelve:
        V vector de datos ordenado
Entorno:
        CENTRAL ← LENGTH DIV 2
        si LENGTH > 1
                V1 ← MERGESORT (SubV[1 a CENTRAL])          ordena subvector izquierdo
                V2 ← MERGESORT(SubV[CENTRAL a LENGTH])      ordena subvector izquierdo
                V ← MERGE(V1, V2, LENGTH1 , LENGTH2)        combina
        finSi
        devuelve V
finSubprograma

subprograma MERGE (V1, V2, LENGTH1, LENGTH2 )
```

```
Recibe:
        V1 vector de datos
        V2 vector de datos
        LENGTH longitud del vector
Devuelve:
        V3 vector combinado
Entorno:
        I ← 1
        J ← 1
        K ← 1
        mientras I <= LENGTH1 AND J <= LENGTH2 hacer
                si V1[I] < V2[J] entonces
                        V3[K] ← V1[I]
                        K ← K + 1
                        I ←I + 1
                sino
                        V3[K] ← V2[J]
                        K ← K + 1
                        J ←J + 1
                finSi
        finMientras
        mientras  I <= LENGTH1 hacer            restantes elementos de V1
                V3[K] ← V1[I]
                K ← K + 1
                I ←I +1
        finMientras
        mientras  I <= LENGTH1 hacer            restantes elementos de V2
                V3[K] ← V1[J]
                K ← K + 1
                J ←J +1
        finMientras
```

Ordenación por el método quickSort

El algoritmo conocido como quicksort (ordenación rápida) recibe el nombre de su autor, Tony Hoare. La idea del algoritmo es simple, se basa en la división en particiones de la vector a ordenar, por lo que se puede considerar que aplica la técnica divide y vencerás. El método es, posiblemente, el más pequeño de código, más rápido, más elegante y eficiente de los algoritmos de ordenación conocidos.

El método se basa en dividir los n elementos del vector a ordenar en dos partes o particiones separadas por un elemento: una partición izquierda, un elemento central denominado pivote o elemento de partición, y una partición derecha. La partición o división se hace de tal forma que todos los elementos del primer subvector (partición izquierda) son menores que todos los elementos del segundo subvector (partición derecha). Los dos subvectores se ordenan entonces independientemente.

Para dividir el vector en particiones (subvectores) se elige uno de los elementos de la lista y se utiliza como pivote o elemento de partición. Si se elige una lista cualquiera con los elementos en orden aleatorio, se puede seleccionar cualquier elemento de la lista como pivote, por ejemplo, el primer elemento de la lista. Si la lista tiene algún orden parcial conocido, se puede tomar otra decisión para el pivote. Idealmente, el pivote se debe elegir de modo que se divida la lista aproximadamente por la mitad, de acuerdo al tamaño relativo de las claves. Por ejemplo, si se tiene un vector de enteros de 1 a 10, 5 o 6 serían pivotes ideales, mientras que 1 o 10 serían elecciones peores.

Una vez que el pivote ha sido elegido, se utiliza para ordenar el resto del vector en dos subvectores: Una tiene todas las claves menores que el pivote y la otra, todos los elementos (claves) mayores que o iguales que el pivote (o al revés). Estas dos listas parciales se ordenan recursivamente utilizando el mismo algoritmo; es decir, se llama sucesivamente al propio algoritmo quicksort. El vector final ordenado se consigue concatenando el primer subvector, el pivote y el segundo vector, en ese orden, en uno solo. La primera etapa de quicksort es la división o «particionado» recursivo del vector hasta que todos los subvectores constan de sólo un elemento.

1. Vector original:	5 2 1 9 3 8 7
Pivote:	5
Subvector izquierdo 1 (menores a 5):	2 1 3
Subvector derecho 1 (mayores o iguales a 5):	9 8 7
2. El algoritmo se aplica al subvector izquierdo1:	2 1 3
Pivote:	2

Subvector Izquierdo 1.1 (menores a 2): 1

Subvector derecho 1.1(mayores o iguales a 2): 3

Subvector derecho 1.1 ordenado: 1 2 3

3. El algoritmo se aplica al subvector derecho 1: 9 8 7

Pivote: 9

Subvector izquierdo 1.2 (menores a 9): 7 8

Subvector derecho 1.2 (mayores o iguales a 9):

Subvector derecho 1.1 ordenado: 7 8 9

4. Vector ordenado subvector izquierda 1 pivote subvector derecha 1: 1 2 3 5 7 8 9

La primera etapa en el algoritmo de partición es obtener el elemento pivote; una vez que se ha seleccionado se ha de buscar la manera de situar en el subvector izquierdo todos los elementos menores.

Los pasos que sigue el algoritmo quicksort son:

1. Seleccionar el elemento central como pivote.
2. Dividir los elementos restantes en particiones izquierda y derecha, de modo que ningún elemento de la izquierda tenga una clave (valor) mayor que el pivote y que ningún elemento a la derecha tenga una clave más pequeña que la del pivote.
3. Ordenar la partición izquierda utilizando quicksort recursivamente.
4. Ordenar la partición derecha utilizando quicksort recursivamente.
5. La solución es: partición izquierda seguida por el pivote y a continuación partición derecha.

```
subprograma QUICKSORT (V, IZQ, DER)
Recibe:
        V vector de datos
        IZQ posición elemento izquierdo
        DER posición elemento derecho
Entorno:
        PIVOTE ← (IZQ + DER) DIV  2
        I ← IZQ
        J ← DER
        AUX
        repetir
                mietras V[I] < V[PIVOTE]
                        I ← I + 1
```

```
                finMientras
                mietras V[J] > V[PIVOTE]
                        J ← J – 1
                finMientras
                si I <= J entonces
                        AUX ← V[I]
                        V[I] ← V[J]
                        V[J] ← AUX
                        I ← I + 1
                        J ← J - 1
                finSi
        mientras I <= J
        si IZQ < J entonces
                QUICKSORT(V, IZQ, J)    repetir proceso con subvector izquierdo
        si I < DER entonces
                QUICKSORT(V, I, DER)    repetir proceso con subvector derecho
finSubprograma
```

Ejercicios

Fuentes y bibliografía