iRAT #008 UT9

Aho, $Hopcroft$, Ullman

Clasificación

El proceso de clasificación u ordenamiento de una lista de objetos de acuerdo a un orden lineal. Se divide en dos partes: interna y externa. **La clasificación interna se da memoria principal de la computadora**, donde se puede aprovechar el acceso aleatorio de varias formas. La clasificación externa se da cuando el número de objetos a ordenar sobrepasa la capacidad de la memoria principal, por lo que se dan diferentes transferencias de bloques de objetos entre la memoria principal y la secundaria.

8.1 El modelo de clasificación interna

Los algoritmos de ordenación más simples requieren un tiempo $O(n^2)$ para ordenar n elementos. Solo funciona para listas muy pequeñas.

El $quicksort$ lleva un tiempo de $O(n * log(n))$, en el peor caso requiere un tiempo $O(n^2)$.

El $heapsort$ y el $intersort$ llevan un tiempo $O(n * log(n))$ en el peor caso, siendo el segundo caso una buena elección para la clasificación externa. Sin embargo, en los casos promedios, es mejor el rendimiento del $quicksort$.

Por otro lado, tenemos el caso del "$Ordenamiento\ por\ cabezas$" que funcionan muy bien en casos muy específicos, por ejemplo, de enteros elegidos en cierto rango. Cuando son aplicables solo requieren $O(n)$ en el peor caso.

Suponemos que los objetos a ordenar son registros que contienen uno o más campos, dentro de los cuales, llamamos **clave**, que es **de un tipo para el cual está habilitado la operación de ordenamiento**; este tipo puede ser enteros o arreglos de caracteres y, en general, cualquier tipo para el cual se pueda utilizar la operación "**menor o igual que"** o "**menor que**".

Definición del problema

El problema de la clasificación consiste en ordenar una secuencia de registros de forma que sus claves formen una secuencia no decreciente. Es decir, si tenemos una lista de registros

$r_1, r_2, r_3, \ldots, r_n$ con sus respectivas claves, $k_1, k_2, k_3, \ldots, k_n$, deben quedar ordenados de forma que $k_1 \leq k_2 \leq k_3 \leq \ldots \leq k_n$. No es necesario que los registros tengan valores distintos, ni que los registros con claves iguales aparezcan en un orden particular.

Criterios de evaluación del tiempo de ejecución

8.2 Esquemas de clasificación

$Bubblesort$

Uno de los métodos de clasificación más simples es el $bubblesort$. **La idea es imaginar que los registros están almacenados en un arreglo vertical, entonces, los registros con claves más livianas "flotan" hacia arriba**. Es decir, en el primer recorrido la clave más liviana sube hacia arriba del todo, en el segundo recorrido la segunda clave más liviana sube a la segunda posición, y así sucesivamente. Mientras se acumulan más recorridos, cada vez tenemos que recorrer menos celdas del arreglo (*la posición i no intenta pasar más allá de la posición i*).

```
procedure bubblesort
for i = 1 to n-1 do
   for j = 1 to i+1 do
      if A[j].clave < A[j-1].clave then
         intercambia(A[j], A[j-1])
```

```
procedure intercambia (var x, x: registro)
   {intercambia valores de x e y}
   var
      temp: registro
   begin
      temp = x
      x = y
      y = temp
   end {intercambia}
```

Clasificación por inserción

En el $i - ésimo$ recorrido se mueve el elemento $A[i]$ en el lugar correcto entre $A[1]$ a $A[i-1]$, los cuales fueron ordenados previamente.

```
procedure insertion sort
    A[0].clave = -∞
    for i = 2 to n do begin
        j = i
        while A[j] < A[j-1] do begin
            intercambia(A[j], A[j-1])
            j = j - 1
        end
    end
```

Clasificación por selección

En el $i-ésimo$ recorrido se selección el registro con la clave más pequeña entre $A[j]$ y $A[n]$, y se intercambia con $A[i]$. Como resultado, después de i recorridos, los i registros menores ocuparán $A[1]$ hasta $A[i]$.

```
procedure selection sort
    for i = 1 to n-1 do
        seleccionar el más pequeño entre A[i], ..., A[n]
        intercambia con A[i]
    end
```

Complejidad de tiempo de los métodos

Tanto $bubblesort$, como $insertion\ sort$ y $selection\ sort$ llevan $O(n^2)$ y van a tomar $\Omega(n^2)$ en la mayoría de secuencias de n elementos.

Cuenta de intercambios

Si el tamaño de los registros es grande, el procedimiento *intercambia* en donde se copian registros, llevará más tiempo que cualquier otro paso, ya sea comparación de claves o cálculos de índices en un arreglo. Así, aunque los tres algoritmos lleven tiempo $O(n^2)$, **se puede comparar con más detalle dependiendo de cuántas veces se usa *intercambia*.**

La **clasificación burbuja** intercambia elementos aproximadamente unas $n^2/4$ veces, al igual que en la **clasificación por inserción**. En el **intercambio por selección**, la operación intercambia se ejecuta exactamente $n-1$ veces en un arreglo de longitud n.

Cuando los registros sean grandes, una estrategia puede ser **mantener punteros a registros**, pudiendo intercambiar punteros en lugar de registros. De este modo, una vez los punteros se

encuentren en el orden deseado, podemos simplemente ordenar los registros en un tiempo $O(n)$.

Limitaciones de los algoritmos simples

El valor de n para el cual algoritmos más complejos se vuelven mejores que los simples depende del tamaño de los registros que debemos intercambiar, así como la máquina en la que ejecutemos el programa. $A\ rule\ of\ thumb$, si n es aproximadamente 100, puede ser una pérdida de tiempo implementar un algoritmo más complicado.

8.3 Clasificación rápida ($quicksort$)

En el caso promedio es $O(n * log(n))$ y en el peor caso es $O(n^2)$.

La esencia del método consiste en clasificar un arreglo $A[1], \ldots, A[n]$ tomando dentro de este un valor clave v como elemento pivote, alrededor del cual reorganizar los valores del arreglo. Es de esperar que este valor esté cerca del elemento medio del arreglo, de modo que una mitad de las claves estén antes que v y otra mitad después. Una vez tengamos los elementos más pequeños $A[1], \ldots, A[j]$ y luego lo valores más grandes $A[j + 1], \ldots, A[n]$ es que podemos llamar recursivamente al $quicksort$ en estos dos arreglos de forma separada.

```
(1)   if de A[i] a A[j] existen al menos dos claves distintas then begin
(2)       sea v la mayor de las dos claves distintas encontradas;
(3)       permutar A[i],...,A[j] de manera que para alguna k entre
              i+1 y j, A[i],..., A[k – 1] tengan claves menores que
              v y los elementos A[k],...,A[j] tengan claves ≥ v
(4)       quicksort(i, k – 1);
(5)       quicksort(k, j)
      end
```

Fig. 8.10. Esbozo de la clasificación rápida.

```
function encuentra_pivote ( i, j: integer ) : integer;
    { devuelve 0 si A[i],...,A[j] tienen claves idénticas; de otra forma, devuel-
    ve el índice de la mayor de las dos claves diferentes de más a la izquierda }
    var
        primera_clave: tipo_clave; { valor de la primera clave encontrada,
            es decir, A[i].clave }
        k: integer; { va de izquierda a derecha buscando una clave diferente }
    begin
        primera_clave := A[i].clave;
        for k := i + 1 to j do { rastrea en busca de una clave distinta }
            if A[k].clave > primera_clave then { selecciona la clave mayor }
                return (k)
            else if A[k].clave > primera_clave then
                return (i);
        return (0) { nunca se encontraron dos claves diferentes }
    end; { encuentra_pivote }
```

Fig. 8.11. Procedimiento *encuentra_pivote*.

```
    procedure quicksort ( i, j: integer );
        { clasifica los elementos A[i],...,A[j] del arreglo externo A }
        var
            pivote: tipo_clave; { el valor del pivote }
            índice_pivote: integer; { el índice de un elemento de A donde clave es
                el pivote }
            k: integer; { índice al inicio del grupo de elementos ≥ pivote }
        begin
(1)         índice_pivote := encuentra_pivote(i,j);
(2)         if índice_pivote <> 0 then begin { no hacer nada si todas las claves
                son iguales }
(3)             pivote := A[índice_pivote].clave;
(4)             k := partición(i, j, pivote);
(5)             quicksort(i,k - 1);
(6)             quicksort(k,j)
            end
        end; { quicksort }
```

Fig. 8.14. El procedimiento *quicksort*.

Mejoras a la clasificación rápida

- **Dividir cada sub-arreglo en partes similares.**
- **Tomar sub-arreglos pequeños:** los algoritmos simples son mejores que los complejos para n pequeñas, se recomienda llamar a algoritmos más simples una vez los arreglos sean de tamaño 9 y menores.

- **Crear un arreglo de punteros a los registros del arreglo A**, si se cuenta con el espacio suficiente: cambiamos $O(n * log(n))$ por $O(n)$, lo cual es sustancial si n es grande. Sin embargo, el acceso a las claves es más lento que antes ya que primero tenemos que pasar por el puntero y luego el registro.

8.4 Clasificación por montículos ($heapsort$)

Cuyo caso promedio y peor caso son $O(n * log(n))$. Esta clasificación puede expresarse por medio de las cuatro operaciones de conjuntos. Supóngase que L es la lista de elementos a clasificar, y S el conjunto de elementos de $tipoRegistro$ que se usará para guardar los elementos conforme se clasifican.

```
(1)    for x en la lista L do
(2)        INSERTA(x, S);
(3)    while not VACIA(S) do begin
(4)        y := MIN(S);
(5)        writeln(y);
(6)        SUPRIME(y, S)
       end
```

Fig. 8.16. Algoritmo abstracto de clasificación.

Un árbol parcialmente ordenado puede representarse por medio de un montículo, un arreglo A[1], …, A[n] cuyos elementos cumplen la propiedad: $A[i].clave \leq A[2 * i].clave \leq A[2 * i + 1].clave$. Al considerar los elementos $2i$ y $2i + 1$ como los hijos de i, el arreglo forma un árbol binario equilibrado.

A pesar de su tiempo $O(n * log(n))$ en el peor caso, $heapsort$ llevará en promedio más tiempo que $quicksort$, en un pequeño factor constante. Es de utilidad práctica cuando no se desea clasificar los n elementos, sino solo las k menores de entre ellos, con k mucho menor que n.

8.5 Clasificación por urnas ($binsort$)

En muchas ocasiones es posible ordenar en tiempos menores a $O(n * log(n))$ siempre que se conozca algo especial acerca de las claves que se están clasificando.

Si suponemos que las claves son de tipo entero, y sabemos que se encuentran en un intervalo de 1 a n sin duplicados (un segundo registros que tenga valor v podría sobrescribir $B[v]$), podemos ordenar los elementos de la lista A en la lista B de la siguiente manera:

```
for i = 1 to n do
   B[A[i].clave] = A[i];
```

En un caso similar podemos implementar este código:

```
for i = 1 to n do
   while A[i].clave <> i do
      intercambia(A[i], A[A[i].clave]);
```

El primer programa es un caso simple de una clasificación por urnas, donde se crea una urna para contener todos los registros con cierta clave. **Se examina cada registro a clasificar y se coloca en la urna de acuerdo con el valor de la clave del registro.**

O sea, las urnas son los elementos en B[1], …, B[n], y B[i] es la urna para el registro con clave i. En este caso se pueden usar las casillas del arreglo porque sabemos que solo existe un solo elemento que se corresponde con una única urna.

En el caso general, a veces es necesario enlazar más de un registro en una urna.