Universidad de Granada

Algorítmica

Memoria de Prácticas

Práctica I: Eficiencia

Autor:
Antonio Gámiz Delgado

13 de Marzo

1 Análisis de la Eficiencia

1.1 Algoritmo de Inserción

```
1 inline static void insercion(int T[], int num_elem)
2 {
3   insercion_lims(T, 0, num_elem);
4 }
5 
6 static void insercion_lims(int T[], int inicial, int final)
7 {
8   int i, j;
9   int aux;
10   for (i = inicial + 1; i < final; i++) {
11     j = i;
12     while ((T[j] < T[j-1]) && (j > 0)) {
13       aux = T[j];
14       T[j] = T[j-1];
15       T[j-1] = aux;
16       j--;
17     };
18   };
19 }
```

Vamos a estudiar el peor caso que se le podría presentar al algoritmo de inserción, es decir, que el vector estuviera ordenado en orden inverso (de mayor a menor).

Como vemos en la línea 5, siempre se llama al método con los argumentos 0 y *num_elem*, por lo que a partir de ahora para nosotros, *inicial* será 0 y $final$ será n.

La mayor parte del tiempo de ejecución se emplea en el cuerpo del bucle *while* interno. Ese trozo de código se puede acotar por una constante a. Por lo tanto, las líneas 15-19 se ejecutan un número de veces dependiente del bucle externo, exactamente i veces (ya que estamos suponiendo que nos encontramos en el peor caso). El bucle externo se ejecuta exactamente n veces, por lo que nos queda:

$$T(n) = \sum_{i=1}^{n-1} \sum_{j=1}^{i} a = a \sum_{i=0}^{n-1} \sum_{j=1}^{i} 1 = a \sum_{i=0}^{n-1} i = a \frac{n(n-1)}{2}$$

Por lo que vemos que $T(n) \in O(n^2)$ o cuadrático.

1.2 Algoritmo QuickSort

```
1
2 static void heapsort(int T[], int num_elem)
3 {
4   int i;
5   for (i = num_elem/2; i >= 0; i--)
6     reajustar(T, num_elem, i);
7   for (i = num_elem - 1; i >= 1; i--)
8     {
9       int aux = T[0];
10      T[0] = T[i];
11      T[i] = aux;
12      reajustar(T, i, 0);
13    }
14 }
15
16 static void reajustar(int T[], int num_elem, int k)
17 {
18   int j;
19   int v;
20   v = T[k];
21   bool esAPO = false;
22   while ((k < num_elem/2) && !esAPO)
23     {
24       j = k + k + 1;
25
26       if ((j < (num_elem - 1)) && (T[j] < T[j+1])) j++;
27       if (v >= T[j]) esAPO = true;
28
29      T[k] = T[j];
30       k = j;
31     }
32   T[k] = v;
33 }
```

Vemos que en el primer bucle de la función *heapsort* aparece la función *reajustar*, por lo que vamos a calcular su eficiencia primero.

Vemos que el cuerpo del bucle *while* consume la mayor parte del tiempo de ejecución. El cuerpo de bucle se puede acotar por una constante b. Como vemos en la línea 22, el bucle empieza en k y termina en $\frac{n}{2}$, pero k no avanza de 1 en 1, sino de $2k+1$ en $2k+1$, por lo que como mucho se ejecutará $\log(n/2)$ veces. Por lo que $R(n) \in O(\log(n))$, siendo $K(n)$ la función de eficiencia de la función *reajustar*. Una vez conocida la eficiencia de la función *reajustar*, pasamos a estudiar el primer bucle de la función *heapsort*. Va desde $n/2$ hasta 0, así que se ejecuta $n/2$ veces, y en cada una de esas veces ejecuta la función *reajustar*, por lo que encontes su eficiencia es $\frac{nlog(n)}{2}$. El segundo bucle se ejecuta $n-1$ veces, por lo que nos queda:

$$T(n) = \frac{nlog(n)}{2} + (n-1)$$

Por lo que $T(n) \in O(n \log(n))$.

1.3 Algoritmo de Floyd

```
void Floyd(int **M, int dim)
{
  for (int k = 0; k < dim; k++)
    for (int i = 0; i < dim;i++)
      for (int j = 0; j < dim;j++)
        {
        int sum = M[i][k] + M[k][j];
       M[i][j] = (M[i][j] > sum) ? sum : M[i][j];
        }
}
```

Vemos que el cuerpo del tercer bucle *while* anidado consume la mayor parte del tiempo de ejecución, por lo que lo acotamos por una constante a. Fácilmente vemos que el resto de bucles va desde 0 hasta *dim*, que podemos denominar n para mayor facilidad. Por lo que evidentemente tenemos que la eficiencia de este algoritmo es an^3, es decir, $T(n) \in O(n^3)$.