1. PRACTICA 0: CALCULO DE EFICIENCIA

Grupo C (J. Huete)

El objetivo de estas sesiones es que el alumno/a comprenda la importancia de analizar la eficiencia de los algoritmos y se familiarice con las formas de llevarlo a cabo. Para ello se mostrará como realizar un estudio teórico y empírico de un algoritmo.

Cálculo del tiempo teórico

A partir de la expresión del algoritmo, se aplicará las reglas conocidas para contar el número de operaciones que realiza un algoritmo. Este valor será expresado como una función de $T(n)$ que dará el número de operaciones requeridas para un caso concreto del problema caracterizado por tener un tamaño n. En los casos de algoritmos recursivos aparecerá una expresión del tiempo de ejecución con forma recursiva, que habrá de resolver con las técnicas estudiadas (ej: resolución de recurrencias por el método de la ecuación característica).

El análisis que nos interesa será el del peor caso. Así, tras obtener la expresión analítica de $T(n)$, calcularemos el orden de eficiencia del algoritmo empleando la notación $O(\cdot)$.

Ejemplo 1: Algoritmo de Ordenación Burbuja

Vamos a obtener la eficiencia teórica del algoritmo de ordenación burbuja. Para ello vamos a considerar el siguiente código que implementa la ordenación de un vector, desde la posición inicial a final de éste, de enteros mediante el método burbuja.

```
1. void burbuja(int T[], int inicial, int final)
2. {
3. int i, j;
4. int aux;
```

```
5.  for (i = inicial; i <final - 1; i++)
6.   for (j = final - 1; j >i; j–)
7.    if (T[j] <T[j-1])
8.    {
9.     aux = T[j];
10.    T[j] = T[j-1];
11.    T[j-1] = aux;
12.   }
13. }
```

La mayor parte del tiempo de ejecución se emplea en el cuerpo del bucle interno. Esta porción de código lo podemos acotar por una constante a. Por lo tanto las líneas de 7-12 se ejecutan exactamente un número de veces de (final-1)-(i+1)+1, es decir, final-i-1. A su vez el bucle interno se ejecuta una serie de veces indicado por el bucle externo. En definitiva tendríamos una fórmula como la siguiente:

$$\sum_{i=inicial}^{final-2} \sum_{j=i+1}^{final-1} a \tag{1}$$

Renombrando en la ecuación (1) $final$ como n e $inicial$ como 1, pasemos a resolver la siguiente ecuación:

$$\sum_{i=1}^{n-2} \sum_{j=i+1}^{n-1} a \tag{2}$$

Realizando la sumatoria interior en (2) obtenemos:

$$\sum_{i=1}^{n-2} a(n-i-1) \tag{3}$$

Y finalmente tenemos:

$$\frac{a}{2}n^2 - \frac{3a}{2}n + a \tag{4}$$

Claramente $\frac{a}{2}n^2 - \frac{3a}{2}n + a \in O(n^2)$

Diremos por tanto que el método de ordenación es de orden $O(n^2)$ o cuadrático.

2. Otros ejemplos

2.1. Fusión

La Tabla 1 muestra el código del algoritmo que permite fusionar dos vectores ordenados U,V en uno. Calcular la eficiencia del algoritmo.

2.2. Inserción

La Tabla 2 muestra el código del algoritmo de ordenación por inserción. Calcular la eficiencia del algoritmo.

2.3. Selección

La Tabla 3 muestra el código del algoritmo de ordenación por inserción. Calcular la eficiencia del algoritmo.

2.4. Ajuste

Calcular la eficiencia del algoritmo de Tabla 4.

Cuadro 1: Algoritmo de Fusión

Calcular la eficiencia del siguiente algoritmo.

```
/**
   @brief Mezcla dos vectores ordenados sobre otro.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a escribir.
   @param final: Posición detrás de la última de la parte del
                  vector a escribir
   inicial < final.
   @param U: Vector con los elementos ordenados.
   @param V: Vector con los elementos ordenados.
             El número de elementos de U y V sumados debe coincidir
             con final - inicial.

   En los elementos de T entre las posiciones inicial y final - 1
   pone ordenados en sentido creciente, de menor a mayor, los
   elementos de los vectores U y V.
*/

static void fusion(int T[], int inicial, int final, int U[], int V[])
{
  int j = 0;
  int k = 0;
  for (int i = inicial; i < final; i++)
    {
      if (U[j] < V[k]) {
T[i] = U[j];
j++;
      } else{
T[i] = V[k];
k++;
      };
    };
}
```

Cuadro 2: Algoritmo de Inserción

```
/**
   @brief Ordena parte de un vector por el método de inserción.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                  vector a ordenar.
   inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de inserción.
*/

static void insercion_lims(int T[], int inicial, int final)
{
  int i, j;
  int aux;
  for (i = inicial + 1; i < final; i++) {
    j = i;
    while ((T[j] < T[j-1]) && (j > 0)) {
      aux = T[j];
      T[j] = T[j-1];
      T[j-1] = aux;
      j--;
    };
  };
}
```

Cuadro 3: Algoritmo de ordenación por selección

```
/**
   @brief Ordena parte de un vector por el método de selección.

   @param T: vector de elementos. Tiene un número de elementos
                  mayor o igual a final. Es MODIFICADO.
   @param inicial: Posición que marca el incio de la parte del
                  vector a ordenar.
   @param final: Posición detrás de la última de la parte del
                  vector a ordenar.
   inicial < final.

   Cambia el orden de los elementos de T entre las posiciones
   inicial y final - 1de forma que los dispone en sentido creciente
   de menor a mayor.
   Aplica el algoritmo de selección.
*/
static void seleccion_lims(int T[], int inicial, int final);


static void seleccion_lims(int T[], int inicial, int final)
{
  int i, j, indice_menor;
  int menor, aux;
  for (i = inicial; i < final - 1; i++) {
    indice_menor = i;
    menor = T[i];
    for (j = i; j < final; j++)
      if (T[j] < menor) {
indice_menor = j;
menor = T[j];
      }
    aux = T[i];
    T[i] = T[indice_menor];
    T[indice_menor] = aux;
  };
}
```

Cuadro 4: Algoritmo de ajuste

```
/**
   @brief Reajusta parte de un vector para que sea un heap.

   @param T: vector de elementos. Debe tener num_elem elementos.
             Es MODIFICADO.
   @param num_elem: número de elementos. num_elem > 0.
   @param k: índice del elemento que se toma com raíz

   Reajusta los elementos entre las posiciones k y num_elem - 1
   de T para que cumpla la propiedad de un montón (APO),
   considerando al elemento en la posición k como la raíz.
*/
static void reajustar(int T[], int num_elem, int k)
{
  int j;
  int v;
  v = T[k];
  bool esAPO = false;
  while ((k < num_elem/2) && !esAPO)
    {
      j = k + k + 1;
      if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
j++;
      if (v >= T[j])
esAPO = true;
      T[k] = T[j];
      k = j;
    }
  T[k] = v;
}
```