UNIVERSIDAD DE GRANADA

E.T.S. DE INGENIERÍAS INFORMÁTICA y DE TELECOMUNICACIÓN

Departamento de Ciencias de la Computación e Inteligencia Artificial

Algorítmica

Guión de Prácticas

Práctica 1: Análisis de Eficiencia de Algoritmos

Curso 2018-2019

Grado en Ingeniería Informática

1 Objetivo

El objetivo de esta práctica es que el estudiante comprenda la importancia del análisis de la eficiencia de los algoritmos y sea capaz de llevarlo a cabo con solvencia. Para ello se organizarán equipos de prácticas, los cuales tendrán que preparar una memoria en la que se exponga detalladamente el proceso seguido para el análisis de la eficiencia teórica, empírica e híbrida de los algoritmos iterativos y recursivos que se proponen en la presente práctica.

2 Cálculo del tiempo teórico

A partir de la expresión del algoritmo, se aplicarán las reglas conocidas para contar el número de operaciones que realiza un algoritmo. Este valor será expresado como una función $T(n)$ que dará el número de operaciones requeridas para un caso concreto del problema caracterizado por tener un tamaño n.

En el caso de los algoritmos recursivos aparecerá una expresión del tiempo de ejecución con forma recursiva, que habrá que resolver con las técnicas estudiadas (p.e. 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.

A continuación, desarrollaremos el estudio teórico sobre dos algoritmos de ejemplo.

2.1 Ejemplo Algoritmo Iterativo: 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 de enteros, desde la posición inicial a la final de éste, mediante el método burbuja.

```
1 void burbuja(int T[], int inicial, int final){
2   int i, j;
3   int aux;
4   for (i = inicial; i < final - 1; i++){
5     for (j = final - 1; j > i; j--){
6       if (T[j] < T[j-1]){
7         aux = T[j];
8         T[j] = T[j-1];
9         T[j-1] = aux;
10      }
11    }
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 6-11 se ejecutan exactamente un número de veces igual a $(final - 1) - (i + 1) + 1$, es decir, $final - i - 1$. A su vez el bucle interno se ejecuta un número 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, pasamos 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 $\mathbf{O(n^2)}$ o cuadrático.

2.2 Ejemplo Algoritmo Recursivo: Ordenación Mergesort

En este ejemplo vamos a calcular la eficiencia del algoritmo de ordenación `Mergesort`. Este algoritmo divide el vector en dos partes iguales y se vuelve a aplicar de forma recurrente a cada una de ellas. Una vez hecho esto, fusiona los dos vectores sobre el vector original, de manera que esta parte ya queda ordenada. Si el número de elementos del vector que se está tratando en cada momento de la recursión es menor que una constante `UMBRALMS`, entonces se ordenará mediante el algoritmo burbuja.

El código del algoritmo `MergeSort` es el siguiente:

```
1 void fusion (int T[], int inicial, int final, int U[], int V[]){
2   int j = 0 ;
3   int k = 0 ;
4   for (int i = inicial; i < final; i++){
5     if (U[j] < V[k]) {
6       T[i] = U[j];
7       j++;
8     } else {
9       T[i] = V[k];
10      k++;
11    }
12  }
13 }
14
15 void mergesort (int T[], int inicial, int final) {
16   if (final - inicial < UMBRALMS){
17     burbuja (T, inicial, final);
18   } else {
19     int k = (final - inicial)/2;
20     int *U = new int [k - inicial + 1];
21     assert (U==0);
22     int l, l2;
23     for (l = 0, l2 = inicial; l < k; l++, l2++){
24       U[l] = T[l2];
25     }
26     U[l] = INT_MAX;
27     int *V = new int [final - k + 1];
28     assert (V==0);
29     for (l = 0, l2 = k; l < final - k; l++, l2++){
30       V[l] = T[l2];
31     }
32     V[l] = INT_MAX;
33     mergesort(U, 0, k);
34     mergesort(V, 0, final - k);
35     fusion(T, inicial, final, U, V);
36     delete []U;
37     delete []V;
38   }
39 }
```

Una vez entendido el procedimiento `Mergesort` se va a obtener su eficiencia teórica. Suponiendo que el tamaño del vector de entrada es superior a `UMBRALMS`, la secuencia de instrucciones de las líneas 19-22 se puede acotar por una constante. Sin perder generalidad, se tomará c como constante. El bucle `for` de la línea 23 se ejecuta un número de veces igual a $n/2$ (en la primera llamada de `mergesort` se tomará *inicial* como 0 y $final$ como n), por tanto la eficiencia hasta aquí sería $O(n)$. Este mismo razonamiento lo tenemos para las instrucciones desde la línea 26 hasta 32. Aplicando la regla del máximo se obtendría un orden $O(n)$ hasta la línea 32.

A continuación, se hacen dos llamadas de forma recursiva a `mergesort` (líneas 33 y 34) con los dos nuevos vectores construidos para ordenar en cada uno de ellos $n/2$ elementos. En la línea 35 se llama al procedimiento fusión que como se puede observar tiene una eficiencia de $O(n)$.

Por lo tanto, para averiguar la eficiencia de `mergesort` se debe formular la siguiente ecuación:

$$T(n) = \begin{cases} 2T(\frac{n}{2}) + n & \text{si } n \geq UMBRALMS \\ n^2 & \text{si } n < UMBRALMS \end{cases} \tag{5}$$

Se resuelve la recurrencia en la ec.(5):

$$T(n) = 2T(\frac{n}{2}) + n \text{ si } n \geq UMBRALMS \tag{6}$$

Haciendo en la ec.(6) el cambio de variable $n = 2^m$ se obtiene:

$$T(2^m) = 2T(2^{m-1}) + 2^m \text{ si } m \geq \log_2(UMBRALMS) \tag{7}$$

$$T(2^m) - 2T(2^{m-1}) = 2^m \tag{8}$$

Renombrando $T(2^m) = t_m$ se obtiene:

$$t_m - 2t_{m-1} = 2^m \tag{9}$$

La ecuación de recurrencia que se deduce de la ec.(9) es:

$$(x-2)^2 = 0 \tag{10}$$

Por tanto la solución general será:

$$t_m = c_1 2^m + c_2 m 2^m \tag{11}$$

Deshaciendo el cambio de variable obtenemos:

$$T(n) = c_1 n + c_2 n \log_2(n) \tag{12}$$

Por tanto $T(n) \in O(n \log_2(n))$.

2.2.1 Comentarios adicionales

En este ejemplo cabe preguntarse por qué hemos hecho uso de un algoritmo de ordenación de orden cuadrático (línea 17), el algoritmo burbuja de ordenación para el caso base, cuando el algoritmo `Mergesort` es más eficiente.

La respuesta se verá cuando se estudie la técnica de resolución de problemas "Divide y Vencerás". Como idea básica, basta saber que el algoritmo de ordenación burbuja a pesar de ser cuadrático, es lo suficientemente eficiente para aplicarlo sobre un vector con pocos elementos, como se da en este caso con un valor de `UMBRALMS` lo suficientemente pequeño.

También se verá más adelante que los algoritmos recursivos, a igual orden de eficiencia, son menos eficientes que los algoritmos iterativos (hacen uso de llamadas recursivas donde internamente se debe gestionar una Pila donde guardar los valores de las variables en cada recursión).

3 Cálculo de la eficiencia empírica

Veremos ahora como llevar a cabo un estudio puramente empírico del comportamiento de los algoritmos analizados. Para ello mediremos los recursos empleados (tiempo) para cada tamaño dado de las entradas.

En el caso de los algoritmos de ordenación, el tamaño viene dado por el número de componentes del vector a ordenar. En otro tipo de problemas, como es el caso del algoritmo para resolver el problema de las torres de Hanoi, el tamaño se corresponde con el valor del entero que representa el número de discos. Para el caso del algoritmo de Floyd, que calcula los caminos mínimos entre todos los pares de nodos en un grafo dirigido, el tamaño es el número de nodos del grafo.

3.1 Cálculo del tiempo de ejecución

La obtención del tiempo empírico de un algorítmo require del cálculo de su tiempo de ejecución. Para ello se deben definir dos momentos temporales: (1) antes del algoritmo (`t_antes`) y (2) después (`t_despues`) del algoritmo. En la variable `t_antes` se capturará el valor del reloj antes de la ejecución, y en la variable `t_despues` el valor del reloj después de la ejecución. Para el cálculo del tiemp de ejecución, vamos a usar la librería `chrono`[1] de la biblioteca `STL`, la cual permite medir el tiempo con una precisión de hasta nanosegundos.

En el siguiente bloque de código se muestra el código para calcular el tiempo de ejecución

```
#include <chrono>
using namespace std;

//Captura el valor del reloj antes de la llamada a burbuja
t_antes = chrono::high_resolution_clock::now();
// Llama al algoritmo de ordenación burbuja
burbuja(T,0,n);
//Captura el valor del reloj después de la ejecución de burbuja
t_depues = chrono::high_resolution_clock::now();

unsigned long t_ejecucion =
   chrono::duration_cast<chrono::microseconds>(t_despues - t_antes).count();
```

Por último, se debe guardar (en fichero) o mostrar por pantalla el tiempo de ejecución, como muestra el fragmento de código siguiente.

```
cout << t_ejecucion << "(us)" << endl;
```

Se debe usar la directiva `-std=gnu++0x` para compilar un código que usa la librería `chrono`.

3.2 Cómo proceder

Para realizar un análisis de eficiencia empírica deberemos ejecutar el mismo algoritmo para diferentes tamaños de entrada.

Así, para un algoritmo de ordenación, por ejemplo, lo ejecutaremos para diferentes tamaños del vector a ordenar y obtendremos el tiempo (preferiblemente varias veces para cada tamaño, en cuyo caso obtendremos el tiempo medio por tamaño[2]). Estos tiempos los almacenaremos en un fichero. Los dos ejemplos de código (`Burbuja` y `Mergesort`) que forman parte del material de prácticas están preparados para recibir como primer argumento la ruta de un fichero de texto, y como resto de argumentos los tamaños de entrada que se quieran evaluar.

3.3 Cómo mostrar los resultados

Para mostrar la eficiencia empírica haremos uso de tablas que recojan el tiempo invertido para cada caso y también de gráficas. Para mostrar los datos en gráfica pondremos en el eje X (abscisa) el tamaño de los casos

[1] `https://en.cppreference.com/w/cpp/chrono/`

[2] Esto tiene sentido en tanto en cuanto los tiempos de ejecución del algoritmo puedan variar considerablemente para entradas del mismo tamaño.

y en el eje Y (ordenada) el tiempo, medido en microsegundos, requerido por la implementación del algoritmo. Para hacer esta representación de los resultados se puede usar cualquier programa que permite la generación de gráficas (Microsoft Word, Google Docs, LibreOffice, Matplotlib (python)).

Partimos de un conjunto de datos, por ejemplo `salida.dat` que contiene en cada fila pares de elementos (x, y) separados por espacios en blanco, como se indica a continuación. El primer elemento del par se corresponde con el tamaño del problema y el segundo elemento se corresponde con el tiempo.

```
100 56
1000 738
10000 7147
100000 19080
500000 88798
1000000 180706
```

La salida anterior se puede usar para generar la gráfica de tiempos usando la herramienta que más nos guste. Por ejemplo, usando las hojas de cálculo de Google Drive la gráfica de tiempos asociada a salida anterior se muestra en la Figura 3.3.

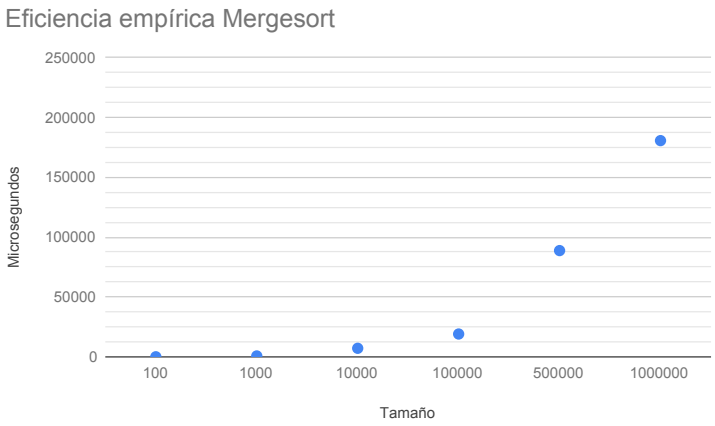


Figure 1: Eficiencia empírica del algoritmo `Mergesort`.

4 Cálculo de la eficiencia híbrida

El cálculo teórico del tiempo de ejecución de un algoritmo nos da mucha información. Es suficiente para comparar dos algoritmos cuando los suponemos aplicados a casos de tamaño arbitrariamente grande. Sin embargo, cuando se va a aplicar el algoritmo en una situación concreta, es decir, especificadas la implementación, el compilador utilizado, el ordenador sobre el que se ejecuta, etc., nos interesa conocer de la forma más exacta posible la ecuación del tiempo. Así, el cálculo teórico nos da la expresión general, pero asociada a cada término de esta expresión aparece una constante de valor desconocido.

Para describir completamente la ecuación del tiempo, necesitamos conocer el valor de esas constantes. La forma de averiguar esos valores es ajustar la función a un conjunto de puntos. El ajuste de una función a un conjunto de puntos se puede realizar de diversas maneras, como por ejemplo alguna técnica de regresión, y más concretamente regresión por mínimos cuadrados (ver Ecuación 13).

$$T(n) = a_0 \times n^2 + a_1 \times n + a_2 \tag{13}$$

Pero dado que de el análisis de un algoritmo requiere de su ejecución con distintos tamaños, también se puede aproximar la función de tiempos atendiendo a la definición de la notación O, la cual indica que si un

algoritmo es de orden $O(f(n))$ entonces existe una constante positiva K tal que, para valores muy grandes del tamaño del caso n, el tiempo de ejecución del algoritmo siempre será inferior o igual a K multiplicando a $f(n)$ (ver Ecuación 15).

$$T(n) \leq K \times f(n) \tag{14}$$

El cálculo de la constante K se realiza despejando e igualando la fórmula anterior:

$$K = \frac{T(n)}{f(n)} \tag{15}$$

Este valor de K se calculará para todas las ejecuciones del mismo algoritmo, produciendo valores aproximados para K. Por tanto, calcularemos el valor final de K como la media de todos estos valores. Una vez que se tiene el valor de K ya es posible usar la función de tiempo teórica para determinar el tiempo máximo (cota superior) que tardará el algoritmo. Ese cálculo se hará sustituyendo n (tamaño del problema) en la función teórica de tiempos. La siguiente gráfica (Figura 4) muestra un ejemplo del cálculo de este valor en Google Docs para los resultados del algoritmo de ordenación por `Mergesort` ejecutado para los casos 100, 1000, 10000, 100000, 500000, 1000000, utilizando una semilla de inicialización de números aleatorios igual a 7.

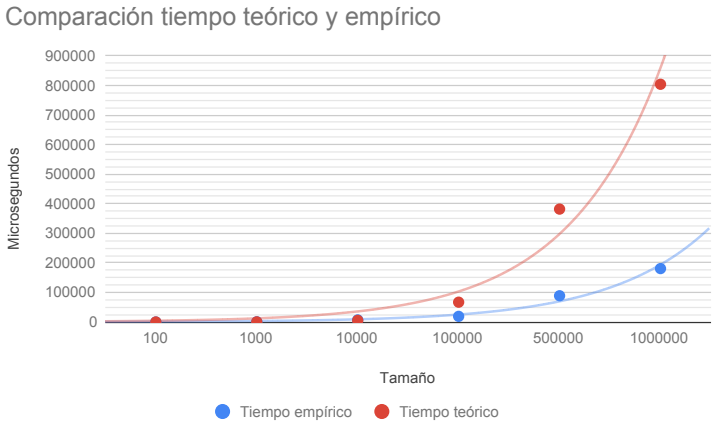


Figure 2: Comparación función de tiempos de la eficiencia empírica y de la función de tiempo teórica.

La Figura 4 muestra como la función correspondiente a la eficiencia teórica (notación O) es una cota superior de la función de tiempos del algoritmo, de manera que la función del tiempo empírico de una misma implementación de un algoritmo no superará a la función teórica independientemente del ordenador en el que se ejecute, como se ha visto en teoría.

5 Elaboración de la práctica

La práctica tendrá tres partes: (1) individual; (2) grupo y (3) defensa del trabajo en grupo.

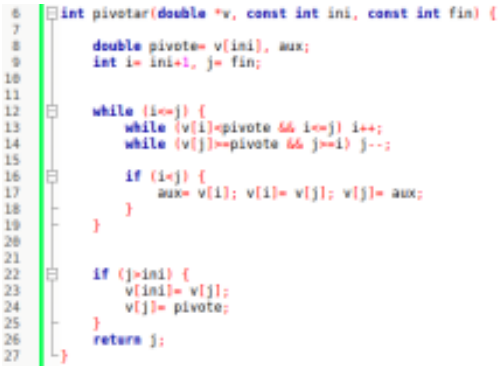
Parte individual. Consistirá en la elaboración del análisis de la eficiencia teórica, empírica e híbrida (cálculo constante K y comparación entre función teórica y empírica) de los algoritmos 1, 2, 3 (iterativos) que se muestran a continuación. Además, se deberá realizar el mismo análisis al algoritmo ordenación por burbuja cuyo código se encuentra dentro del material de prácticas.

Grupo Unificación y comparación de la parte individual. Así mismo se deberá realizar el análisis de eficiencia teórica, empírica e híbrida de los algoritmos 4, y 5, de los algoritmos `Mergesort` y el que resuelve el problema de las Torres de Hanoi que se aporta como parte del material de prácticas. Además, el grupo deberá preparar una memoria de la práctica en la que se describan todos los análisis efectuados, se recoja la parte individual y se realicen las comparaciones oportunas entre los resultados empíricos e híbridos obtenidos por cada integrante del grupo en la parte individual.

Defensa El grupo deberá preparar una defensa de la memoria de la práctica. Es obligatorio realizar la defensa.

5.1 Algoritmos

- Algoritmo 1.



```
int pivotar(double *v, const int ini, const int fin) {

    double pivote= v[ini], aux;
    int i= ini+1, j= fin;


    while (i<=j) {
        while (v[i]<pivote && i<=j) i++;
        while (v[j]>=pivote && j>=i) j--;

        if (i<j) {
            aux= v[i]; v[i]= v[j]; v[j]= aux;
        }
    }


    if (j>ini) {
        v[ini]= v[j];
        v[j]= pivote;
    }
    return j;
}
```

- Algoritmo 2.

```
int Busqueda (int *v, int n, int elem) {

  int inicio, fin, centro;

  inicio= 0;
  fin= n-1;
  centro= (inicio+fin)/2;
  while ((inicio<=fin) && (v[centro] != elem)) {

   if (elem<v[centro])
    fin= centro-1;
   else
    inicio= centro+1;
    centro= (inicio+fin)/2;
}

  if (inicio>fin)
   return -1;

  return centro;
}
```

- Algoritmo 3.

```

void EliminaRepetidos(double original[], int & nOriginal) {
```

```
3
4   int i, j, k;
5
6   // Pasamos por cada componente de original
7   for (i= 0; i<nOriginal; i++) {
8
9    // Buscamos valor repetido de original[i]
10   // desde original[i+1] hasta el final
11   j= i+1;
12   do {
13
14    if (original[j] == original[i]) {
15
16      // Desplazamos todas las componentes desde j+1
17      // hasta el final, una componente a la izquierda
18      for (k= j+1; k<nOriginal; k++)
19       original[k-1]= original[k];
20
21      // Como hemos eliminado una componente, reducimos
22      // el numero de componentes utiles
23      nOriginal--;
24    } else // Si el valor no esta repetido, pasamos al siguiente j
25      j++;
26   } while (j<nOriginal);
27
28  } // FIN del primer for
```

- Algoritmo 4.

```
47  int BuscarBinario(double *v, const int ini, const int fin,
48                    const double x) {
49      int centro;
50      if (ini>fin) return -1;
51
52      centro= (ini+fin)/2;
53      if (v[centro] == x) return centro;
54      if (v[centro]>x) return BuscarBinario(v, ini, centro-1, x);
55      return BuscarBinario(v, centro+1, fin, x);
56  }
```

- Algoritmo 5.

```
1 void heapsort(int T[], int num_elem)
2 {
3   int i;
4   for (i = num_elem/2; i >= 0; i--)
5     reajustar(T, num_elem, i);
6   for (i = num_elem - 1; i >= 1; i--)
7     {
8       int aux = T[0];
9       T[0] = T[i];
10      T[i] = aux;
11      reajustar(T, i, 0);
12    }
13 }
```

```
1
```

```
2 void reajustar(int T[], int num_elem, int k)
3 {
4   int j;
5   int v;
6   v = T[k];
7   bool esAPO = false;
8   while ((k < num_elem/2) && !esAPO)
9     {
10      j = k + k + 1;
11      if ((j < (num_elem - 1)) && (T[j] < T[j+1]))
12  j++;
13      if (v >= T[j])
14  esAPO = true;
15      T[k] = T[j];
16      k = j;
17    }
18  T[k] = v;
19 }
```

NOTA: Recuerde que la nota de la Práctica 1 es la media geométrica de la nota sobre 10 de las partes individual, de grupo y de la defensa.