UNIVERSIDAD DE GRANADA

Algorítmica

2º Grado en Ingeniería Informática

Guión de prácticas
Práctica 1

Cálculo de la eficiencia de algoritmos

© Prof. Manuel Pegalajar Cuéllar
Dpto. Ciencias de la Computación e I. A.
Universidad de Granada

DECSAI

Departamento de Ciencias de la Computación e Inteligencia Artificial

Cálculo de la eficiencia de algoritmos

1. Objetivo

El objetivo de la práctica consiste en que el alumno sea capaz de analizar y calcular la eficiencia teórica, práctica e híbrida de algoritmos, tanto iterativos como recursivos. Para ello, se expondrá un problema que será resuelto en clase por el profesor, para cada apartado. Posteriormente, se expone un conjunto de problemas que deberán ser resueltos por el estudiante.

2. Cálculo de la eficiencia teórica

2.1. Algoritmos iterativos

Se debe calcular la eficiencia teórica de los siguientes algoritmos. Utilícese la explicación dada por el profesor para resolver el ejemplo siguiente como plantilla para solucionar cada problema.

2.1.1. Ejercicio guiado

Calcule la eficiencia, en el peor caso, para el siguiente algoritmo:

```
void selección(double *v, int inicio, int final) {

   int i, j, min;

   for (i= inicial; i<final; i++) {
      min= i;
      for (j= i+1; j<final; j++)
         if (v[j] < v[min])
            min= j;
      Intercambia(v[i], v[min]);
   }
}
void Intercambia(int &a, int &b) {
   int aux= a; a= b; b= aux;
}
```

Solución:

El algoritmo resuelve el problema de ordenación de un subvector, desde una posición "inicial" a una posición "final", (final no inclusive). Se trata del algoritmo de ordenación por selección para ordenar vectores, particularizado para la ordenación de un subvector.

El tamaño del problema depende de un único parámetro ***n****, que es la longitud del subvector a ordenar. Por tanto,*

$$n=final-inicial$$

Comenzamos analizando la función externa Intercambia. Está formada por 3 sentencias simples, consideradas como operaciones elementales. La eficiencia de la secuencia de estas 3 sentencias simples se calcula como el max{O(1), O(1), O(1)}= O(1), por lo que la función Intercambia tiene eficiencia O(1).

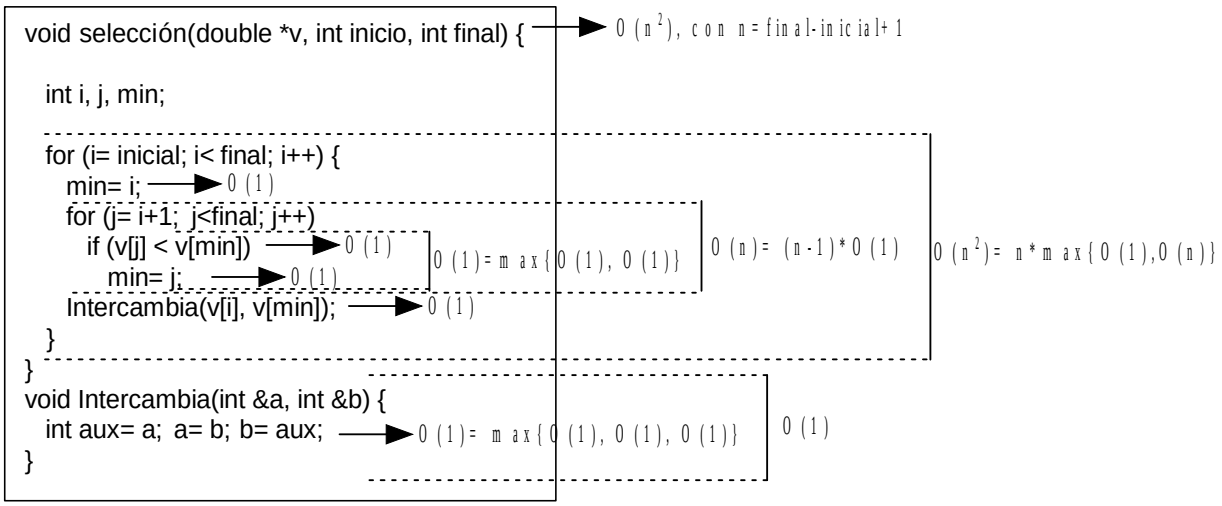
Para analizar el algoritmo selección, comenzaremos desde el código más interno al algoritmo y finalizaremos por las sentencias del primer nivel. Comenzamos por la parte del "if". En esta sentencia condicional, tanto la evaluación de la condición como la sentencia que se ejecuta cuando esta es verdadera son O(1) porque son sentencias simples. Por tanto, la sentencia "if" es de eficiencia max{O(1), O(1)}= O(1). Ahora pasemos a analizar el siguiente nivel, el bucle "for j":

Llamemos "k" al tiempo constante que tarda en ejecutarse la sentencia condicional interna al bucle. El peor caso que pueda darse para que el bucle se ejecute el máximo número de veces posible es cuando j=i+1 e i=inicial. En este caso, el bucle se ejecutará final-inicial-1 veces; es decir, n-1 veces, y el tiempo que tardará en ejecutarse todo el bucle será k(n-1)= k*n-k<=k*n. Por tanto, la eficiencia del bucle interno "For j" es O(n).*

Antes de analizar el bucle externo, tenemos que analizar la secuencia de instrucciones que contiene. La asignación "min=i" y la llamada a "intercambia" son O(1) porque se consideran operaciones elementales. Como el bucle interno "for j" es O(n), la eficiencia de esta secuencia de instrucciones es max{O(1), O(n), O(1)}= O(n).

*Analicemos el bucle externo. Este se ejecutará final-inicial veces; es decir, n veces. Por tanto, la eficiencia del mismo será n*O(n); es decir, $O(n^2)$.*

Para finalizar el análisis, las sentencias que quedan en el algoritmo son definiciones de variables de tipos básicos "int i, j, min;", que son operaciones elementales cuya eficiencia es O(1). Por tanto, la eficiencia de la función "selección" es el max{O(1), $O(n^2)$}= $O(n^2)$.



2.1.2. Ejercicios propuestos

Calcule la eficiencia en el peor y en el mejor caso para los siguientes algoritmos. En caso de que el algoritmo tenga orden exacto, indíquelo.

a)

```
6   int pivotar(double *v, const int ini, const int fin) {
7
8       double pivote= v[ini], aux;
9       int i= ini+1, j= fin;
10
11
12      while (i<=j) {
13          while (v[i]<pivote && i<=j) i++;
14          while (v[j]>=pivote && j>=i) j--;
15
16          if (i<j) {
17              aux= v[i]; v[i]= v[j]; v[j]= aux;
18          }
19      }
20
21
22      if (j>ini) {
23          v[ini]= v[j];
24          v[j]= pivote;
25      }
26      return j;
27  }
```

---

b)

```
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;
}
```

---

c)

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

  int i, j, k;

  // Pasamos por cada componente de original
  for (i= 0; i<nOriginal; i++) {

            // Buscamos valor repetido de original[i]
            // desde original[i+1] hasta el final
```

```
        j= i+1;
        do {

                if (original[j] == original[i]) {

                  // Desplazamos todas las componentes desde j+1
                  // hasta el final, una componente a la izquierda
                  for (k= j+1; k<nOriginal; k++)
                          original[k-1]= original[k];

                  // Como hemos eliminado una componente, reducimos
                  // el numero de componentes utiles
                  nOriginal--;
                } else // Si el valor no esta repetido, pasamos al siguiente j
                  j++;
        } while (j<nOriginal);

  } // FIN del primer for
}
```

2.2. Algoritmos recursivos

Se debe analizar la eficiencia de los siguientes códigos recursivos. Para ello, siga la explicación del profesor hecha en clase para los ejercicios guiados. Posteriormente, calcule la ecuación en recurrencias de cada código recursivo propuesto y resuélvala.

2.2.1. Ejercicios guiados

a)

```
16  double ElevarDyV(double r, int n) {
17
18      if (n==0) return 1;
19      else if (n==1) return r;
20      else {
21          double subsolucion= ElevarDyV(r, n/2);
22          if (n%2 == 0)
23              return subsolucion*subsolucion;
24          else return r*subsolucion*subsolucion;
25      }
26  }
```

---

b)

```
void MergeSort(int *v, int ini, int fin) {

    // Cuando ini>=fin es el caso base, donde v ya esta ordenado
    // Solo dividimos vector cuando tiene más de 1 componente
    if (ini < fin) {
        int med = (ini + fin)/2; // Partimos vector en 2
        MergeSort(v, ini, med); // Ordenamos parte izq
        MergeSort(v, med + 1, fin); // Ordenamos parte dcha
        combina(v, ini, med, fin); // Combinamos soluciones
    }
}
```

```
void combina(int *v, int ini, int med, int fin) {
    int vtam= fin - ini + 1;
    int *aux= new int[vtam];
    int i = ini, j= med+1, k=0;

     // Copiar el mínimo elem de los vectores
     while (i <= med && j <= fin) {
        if (v[i] < v[j]) {  aux[k]= v[i]; i++; }
        else { aux[k]= v[j]; j++; }
        k++;
    }

    // Si subvector 1 más grande que el 2, copiamos 1 en aux
    while (i <= med) { aux[k] = v[i]; i++; k++; }
    // Si subvector 2 más grande que el 1, copiamos 2 en aux
    while (j <= fin) { aux[k] = v[j]; j++; k++; }

    // Devolvemos aux en v
    for (int n = 0; n < vtam; ++n) v[ini+n]= aux[n];
    delete [] aux;
}
```

2.2.2. Ejercicios propuestos

a)

```
int BuscarBinario(double *v, const int ini, const int fin,
                  const double x) {
    int centro;
    if (ini>fin) return -1;

    centro= (ini+fin)/2;
    if (v[centro] == x) return centro;
    if (v[centro]>x) return BuscarBinario(v, ini, centro-1, x);
    return BuscarBinario(v, centro+1, fin, x);
}
```

---

b)

```
void HeapSort(int *v, int n){

	double *apo=new double [n];
	int tamapo=0;

	for (int i=0; i<n; i++){
		insertar(apo,tamapo,v[i]);
	}
	for (int i=0; i<n; i++) {
		v[i]=apo[0];
		borrarRaiz(apo,tamapo);
	}
	delete [] apo;

}

void insertar(double *apo, int &tamapo, double valor){

	apo[tamapo]=valor;
	tamapo++;
	int aux =tamapo-1;
	bool fin =false;
	while (!fin) {
		int padre;
		if (aux==0) {
			fin=true;
		}else{

			if (aux%2==0) {
				padre=(aux-2)/2;
			}else{
				padre=(aux-1)/2;
			}

			if (apo[padre] > apo[aux]) {
				double tmp=apo[aux];
				apo[aux]=apo[padre];
				apo[padre]=tmp;
				aux=padre;
```

```
                    }else{
                            fin=true;
                    }

            }
        }
}
```

3. Cálculo de la eficiencia práctica

Para medir la eficiencia práctica de los algoritmos, tenemos que ejecutar el mismo algoritmo para diferentes tamaños de casos, y calcular su tiempo de ejecución. Para ello, y dado que los ordenadores modernos son muy veloces, haremos uso de la biblioteca **chrono** incluida en el estáncar **C++11**. No olvide compilar cada programa con el compilador **g++** incluyendo este estándar antes de compilar cada programa.

3.1. Ejercicio guiado

El tiempo de ejecución lo mediremos como la diferencia entre el instante de tiempo justo anterior al inicio del algoritmo, y el instante justamente posterior. Para ello, necesitaremos declarar las siguientes variables en nuestro programa:

```
#include<chrono>
...

chrono::time_point<std::chrono::high_resolution_clock> t0, tf; // Para medir el tiempo de ejecución
```

Los instantes de tiempo, y la duración, los obtendremos de la siguiente forma:

```
// Comenzamos a medir el tiempo
t0= std::chrono::high_resolution_clock::now();

// Aquí vendría la ejecución del algoritmo

// Terminamos de medir el tiempo
tf= std::chrono::high_resolution_clock::now();
```

```
// Medimos la duración, en MICROSEGUNDOS
unsigned long duration;
duration= std::chrono::duration_cast<std::chrono::microseconds>(tf - t0).count();
```

Ejecute el código del ejemplo **Practica1.cpp** proporcionado por el profesor para medir el tiempo de ejecución del algoritmo de ordenación por Burbuja con diversos tiempos de ejecución.

3.2. Ejercicios propuestos

Mida el tiempo de ejecución para diversos tamaños de caso (1000, 2000, 3000, ..., 10000, 20000, 30000, ...), para los algoritmos b) del apartado 2.2.1 (MergeSort) y b) del apartado 2.2.2 (HeapSort).

4. Eficiencia híbrida

En este apartado comprobaremos cómo la eficiencia práctica medida en el apartado 3 se corresponde con la eficiencia teórica calculada en el apartado 2, para los algoritmos b) del apartado 2.2.1 (MergeSort) y b) del apartado 2.2.2 (HeapSort). Incluya también el cálculo para el algoritmo de ordenación por burbuja proporcionado por el profesor.

Para ello, ejecute los algoritmos de ordenación bajo las mismas condiciones (mismas entradas, mismo tamaño de caso). Mida el tiempo de ejecución para múltiples tamaños de caso diferentes, y calcule el valor de la constante oculta tal y como se explica a continuación.

4.1. Cálculo de la constante oculta

En apartados previos hemos calculado el orden O(f(n)) de cada algoritmo propuesto. Este orden O(f(n)) quiere decir que existe una constante **K**, para cada algoritmo, tal que el tiempo **T(n)** de ejecución del mismo para un tamaño de caso **n** es:

$$T(n) <= K*f(n)$$

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

$$K= T(n)/f(n)$$

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. La siguiente gráfica muestra un ejemplo del cálculo de este valor en LibreOffice Calc, para los resultados del algoritmo de ordenación por burbuja ejecutado para los casos 1000, 2000, 3000, 4000, 5000, utilizando una semilla de inicialización de

números aleatorios igual a 123456.

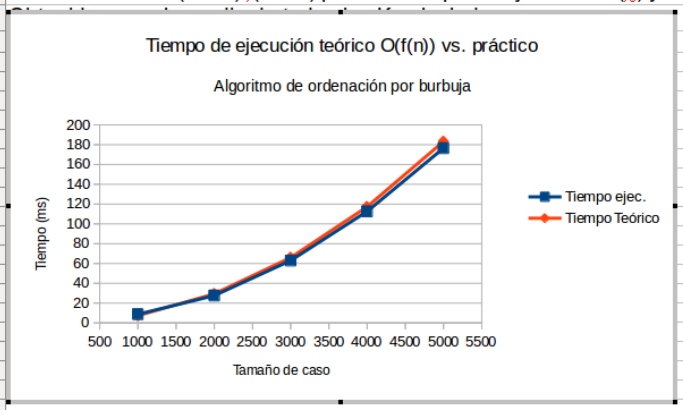
	B	C	D
1	**Tiempo ejec.**	**K**	
2	8,663	0,000008663	Obtenido como T(1000)/f(1000) para un tiempo de ejecución T(n) y orden O(f(n))
3	27,584	0,000006896	Obtenido como T(2000)/f(2000) para un tiempo de ejecución T(n) y orden O(f(n))
4	62,921	6,99122222222222E-006	…
5	112,555	7,0346875E-006	
6	176,439	7,05756E-006	Obtenido como T(5000)/f(5000) para un tiempo de ejecución T(n) y orden O(f(n))
7	**K media=**	7,32849394444445E-006	Obtenido como la media de todas las K calculadas
8			
9			
10			

NOTA: Para importar los datos de los ficheros de salida en LibreOffice Calc o Excel, es necesario reemplazar todos los puntos (.) por comas (,) en el fichero de datos.

4.2. Comparación gráfica de órdenes de eficiencia

Asumiendo que hemos calculado el orden de eficiencia del algoritmo de ordenación por burbuja como $O(n^2)$, y que el valor de la constante oculta en promedio vale $K=7.33 \cdot 10^{-6}$, a continuación, comprobaremos experimentalmente que el orden O(f(n)) calculado con una constante oculta superior siempre será mayor que el tiempo de ejecución real del algoritmo, por lo que hemos conseguido acotar el tiempo de ejecución del mismo y estimar, en el futuro, cuánto tardará en ejecutarse para otros tamaños de casos. Se muestra la siguiente gráfica generada por LibreOffice Calc para el tiempo de ejecución real y teórico con la constante oculta calculada:

	D	E
1		**Tiempo Teórico**
2	Obtenido como T(1000)/f(1000) para un tiempo de ejecución T(n) y orden O(f(n))	7,3284939444
3	Obtenido como T(2000)/f(2000) para un tiempo de ejecución T(n) y orden O(f(n))	29,3139757778
4	…	65,9564455
5		117,2559031111
6	Obtenido como T(5000)/f(5000) para un tiempo de ejecución T(n) y orden O(f(n))	183,2123486111



5. Entrega de la práctica

Se deberá entregar un documento individual o por parejas que contenga los siguientes apartados:

1. Solución a los problemas de eficiencia teórica.
2. Solución a los problemas de eficiencia práctica.
3. Solución a los problemas de eficiencia híbrida: cálculo de constantes y comparación gráfica.
4. Análisis de resultados del apartado 3. Indique, atendiendo a los resultados obtenidos, qué algoritmo es mejor entre los algoritmos de ordenación por burbuja (proporcionado por el profesor en el fichero **Practica1.cpp**, Mergesort y QuickSort).

Estos apartados deberán contener las soluciones en el formato explicado en clase por el profesor.

La práctica deberá ser entregada por PRADO, en la fecha y hora límite explicada en clase por el profesor. No se aceptarán, bajo ningún concepto, prácticas entregadas con posterioridad a la fecha límite indicada. La entrega de PRADO permanecerá abierta con, al menos, una semana de antelación antes de la fecha límite, por lo que todo alumno tendrá tiempo suficiente para entregarla.

La práctica se valorará de 0 a 10. Cada apartado se valorará con 2,5 puntos. Se valorará no sólo la exactitud y bondad de los resultados obtenidos, sino principalmente la explicación que realice el alumno sobre cómo ha realizado los cálculos en cada apartado.

La práctica contribuirá con 1 punto sobre 10, ponderado al total de la puntuación de prácticas expuesto en la guía docente de la asignatura.

El profesor, en clase de prácticas, realizará defensas de las prácticas a discreción, con el fin de asegurar de que los estudiantes alcanzan las competencias deseadas. Por este motivo, una vez finalizada la entrega de prácticas por PRADO, es recomendable repasar los ejercicios entregados para poder responder a las preguntas del profesor, llegado el caso de su defensa. La no superación de la defensa de prácticas supondrá una calificación de 0 en esta práctica. La superación de la defensa supondrá mantener la calificación obtenida.