Práctica 1: Eficiencia

Dpto. Ciencias de la Computación e Inteligencia Artificial
E.T.S. de Ingenierías Informática y de Telecomunicación
Universidad de Granada

DECSAI

Algorítmica

Grado en Ingeniería Informática

Índice de contenido

1. Introducción

Los objetivos de este guión de prácticas son los siguientes:

1. Profundizar en el cálculo de la eficiencia teórica de un código junto con su orden de eficiencia en el peor, promedio y mejor de los casos.
2. Practicar el cálculo de la eficiencia empírica de un código.
3. Ajustar los datos obtenidos empíricamente usando la curva de eficiencia teórica

2. Eficiencia teórica: Caso Peor, Caso mejor, Caso Promedio

La eficiencia teórica viene dada por una función que da una cota superior, cota inferior o cota exacta del tiempo de ejecución de un algoritmo en función del tamaño de los datos de entrada. Esta medida debe ser independiente del hardware en el que se ejecuta el algoritmo, del lenguaje en el que ha sido implementado y de las bibliotecas que utiliza. Además, no se necesita ejecutar el programa para conocer la eficiencia.

Esta medida se establece contando el número de operaciones elementales (OE) que realiza el algoritmo para un volumen determinado de datos entrada.

A continuación vemos un ejemplo de implementación de un algoritmo de búsqueda lineal y vamos a obtener para el la diferentes cotas. Tenemos un vector v que contiene n elementos y se desea buscar un cierto elemento x. La función recibe esos datos como entrada y:

- Si el elemento está devuelve la posición en donde se ha encontrado.
- Si el elemento no está devuelve el valor -1.

```
1. int buscar(const int *v, int n, int x) {
2.    int i=0;
3.    while (i<n && v[i]!=x)
4.       i=i+1;
5.    if (i<n)
6.       return i;
7.    else
8.       return -1;
9. }
```

Para determinar el tiempo de ejecución, calcularemos el número de OE que realiza esta implementación:

- Línea 2: 1 OE (asignación)
- Línea 3: 4 OE (acceso al elemento v[i], comparación i<n, comparación v[i]!=x, operación &&).
- Línea 4: 2 OE (incremento, asignación).
- Línea 5: 1 OE (comparación).
- Línea 6: 1 OE (devolución).
- Línea 8: 1 OE (devolución).

Por lo tanto el tiempo de ejecución en el peor de los casos se puede formular con la siguiente ecuación:

$$T_p(n) = \underset{(2)}{1} + \underset{(3)}{4} + \Big(\sum_{i=0}^{n-1} \underset{(3)}{2} + \underset{(4)}{4}\Big) + \underset{(5)}{1} + \underset{(6)}{max}\underset{(8)}{(1,1)} = 6n+7$$

Observe que el número de OE que se realizan depende de n que es el número de elementos del vector. En este algoritmo, el peor de los casos posibles es que no se encuentre el elemento x, caso en el que se recorre el vector completo.

Puesto que $T_p(n)=6n+7 \in O(n)$ podemos afirmar que el orden de eficiencia del algoritmo de búsqueda secuencial es O(n).

Por otro lado el mejor de los casos es cuando para un determinado vector de tamaño n encontramos el elemento x en la primera posición. En este caso el número de OE elementales que se realizan son las siguientes:

- Línea 2: 1 OE (asignación)
- Línea 3: 4 OE (acceso al elemento v[i], comparación i<n, comparación v[i]!=x, operación &&).
- Línea 5: 1 OE (comparación).
- Línea 6: 1 OE (devolución).

Con esta secuencia de intrucciones podemos formular el tiempo de ejecución en el mejor de los casos con la siguiente ecuación:

$$T_m(n)=1+4+1+1=7$$

Así podemos afirmar que el tiempo en el mejor de los caso $T_m(n)=7 \in \Omega(1)$.

Y finalmente para poder estudiar el tiempo promedio $T_{1/2}(n)$ debemos de establecer la probabilidad de que el elemento se encuentre en cada una de las posiciones del vector. Para ello tenemos que suponer que esta probabilidad sigue una determinada distribución.

Suponiendo que es una distribución uniforme de la siguiente forma

$$p(x=v[i])=\frac{1}{n} \forall i=0,1,2,\ldots,n-1$$

tenemos que el numero medio de OE que se ejecutarán en el while es

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

Una vez que conocemos este dato podemos calcular nuestr tiempo de ejecución promedio de la siguiente forma:

$$T_{1/2}(n)=1+4+(\sum_{i=0}^{\frac{n-1}{2}} 2+4)+1+max(1,1)=5+3n+3+2=10+3n$$

Así por lo tanto podemos afirmar que $T_{1/2}(n) \in \Theta(n)$

3. Eficiencia empírica

En esta sección vamos a ver cómo se puede medir la eficiencia de forma empírica, es decir, vamos a estudiar de forma experimental el comportamiento del algoritmo. Para ello vamos a medir los recursos empleados por el mismo. Puesto que estamos hablando de eficiencia temporal, mediremos tiempos de ejecución de la implementación que tenemos para diferentes entradas de datos. Esta medida depende del sistema en el que vamos a realizar las ejecuciones (hardware y software).

Para hacer las mediciones usaremos la biblioteca **`chrono.`** `Esta librería premite tratar con el tiempo.` Los elementos para nuestro objetivo de medir el tiempo que necesitamos son:

- **high_resolution_clock::time_point** es un nuevo tipo de dato que representa un punto en el tiempo relativo al reloj del sistema.
- **duration:** permite establecer el tiempo transcurrido desde dos puntos de tiempo.

El esquema que usaremos para medir tiempos con estos recursos es el siguiente:

```
#include <chrono>


using namespace std;
using namespace std::chrono;
….
high_resolution_clock::time_point start,//punto de inicio
                                  end; //punto de fin
duration<double> tiempo_transcurrido;  //objeto para medir la duracion de end
                                       // y start

start = high_resolution_clock::now(); //iniciamos el punto de inicio

// Ejecutamos nuestro  Algoritmo
// …
end = high_resolution_clock::now(); //anotamos el punto de de fin
//el tiempo transcurrido es
tiempo_transcurrido  = duration_cast<duration<double> >(end - start);

// Mostramos resultados
cout << "Segundos       : " << tiempo_transcurrido.count() << endl;
```

A continuación vemos un ejemplo completo para medir el tiempo de ejecución del algoritmo de búsqueda secuencial. Este programa genera un vector de números aleatorios (todos ellos en un intervalo [0,VMAX]) y, a continuación, busca un elemento que sabemos que no va a estar incluido en ese vector, provocando de esta forma que se dé el peor caso posible. El programa tiene dos argumentos que se le suministran en la línea de órdenes. El primero es el tamaño del vector y el segundo es VMAX.

Recuerde que para crear el ejecutable en un sistema GNU/Linux desde la línea de órdenes, y asumiendo que el programa está almacenado en un fichero `busqueda_lineal.cpp`, debe ejecutar:

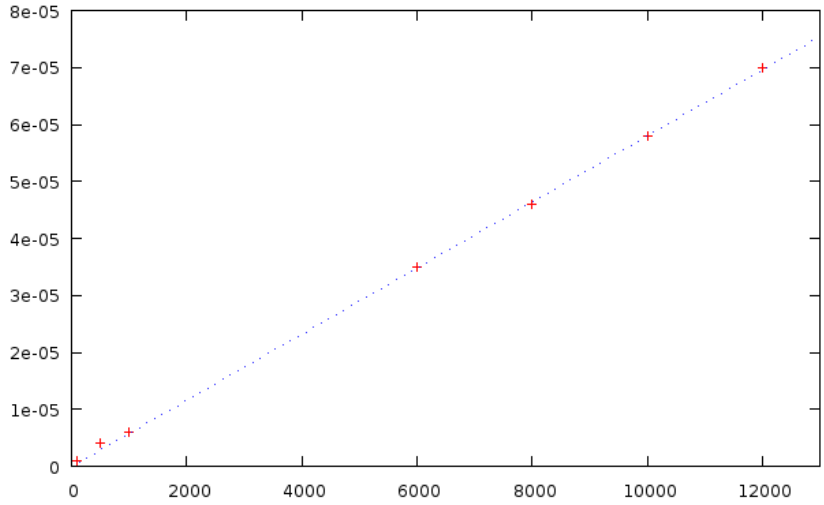
```
g++ busqueda_lineal.cpp -o busqueda_lineal
```

```
#include <iostream>
#include <cstdlib>  // Para generación de números pseudoaleatorios
#include <chrono> // Recursos para medir tiempos
using namespace std;
using namespace std::chrono;
int buscar(const int *v, int n, int x) {
  int i=0;
  while (i<n && v[i]!=x)
    i=i+1;
if  if (i<n)
    return i;
  else
    return -1;
}
void sintaxis() {
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "  VMAX: Valor máximo (>0)" << endl;
  cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[]) {
  if (argc!=3)                // Lectura de parámetros
    sintaxis();
  int tam=atoi(argv[1]);      // Tamaño del vector
  int vmax=atoi(argv[2]);     // Valor máximo
  if (tam<=0 || vmax<=0)
    sintaxis();

  // Generación del vector aleatorio
  int *v=new int[tam];    // Reserva de memoria
  srand(time(0));         // Inicialización generador números pseudoaleatorios
  for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = rand() % vmax;    // Generar aleatorio [0,vmax[

  high_resolution_clock::time_point start,//punto de inicio
                                    end; //punto de fin
  duration<double> tiempo_transcurrido;  //objeto para medir la duracion de
                                        // end       y start

 start = high_resolution_clock::now(); //iniciamos el punto de inicio

   int x = vmax+1;  // Buscamos un valor que no está en el vector
   buscar(v,tam,x); // de esta forma forzamos el peor caso

  end = high_resolution_clock::now(); //anotamos el punto de de fin

 //el tiempo transcurrido es
 tiempo_transcurrido  = duration_cast<duration<double> >(end - start);

 // Mostramos resultados (Tamaño del vector y tiempo de ejecución en seg.)
  cout << tam << "\t" <<tiempo_transcurrido.count() << endl;

  delete [] v;      // Liberamos memoria dinámica
}
```

Una vez que sabemos como calcular el tiempo de ejecución de un programa, para obtener la eficiencia empírica debemos ejecutarlo para muchos tamaños diferentes del problema. Así, por ejemplo, podemos hacer las siguientes ejecuciones del programa anterior:

Tamaño del vector	*Tiempo de ejecución*
100	1x10^-6
500	4x10^-6
1000	6x10^-6
6000	3.5x10^-5
8000	4.6x10^-5
10000	5.8x10^-5
12000	7x10^-5

Si ahora representamos esos datos gráficamente tenemos esto:



en donde se puede apreciar que, efectivamente, los datos se ajustan aproximadamente a una función lineal (superpuesta con línea discontinua).

3.1. Visualización de datos con gnuplot

En este ejemplo hemos hecho siete ejecuciones pero para que el experimento sea más fiable debemos hacer muchas más. Y, además, podemos hacerlo de una forma más sistemática. Para ello vamos a almacenar los datos de muchas ejecuciones en un fichero que posteriormente

```
#!/bin/csh
@ inicio = 100
@ fin = 1000000
@ incremento = 100

@ i = $inicio
echo > tiempos.dat
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./busqueda_lineal $i 10000` >> tiempos.dat
  @ i += $incremento
end
```

vamos a dibujar con el programa `gnuplot`. A modo de ejemplo, el siguiente script de C-Shell calcula el tiempo de ejecución con tamaños 100, 200, 300, ..., 1000000:

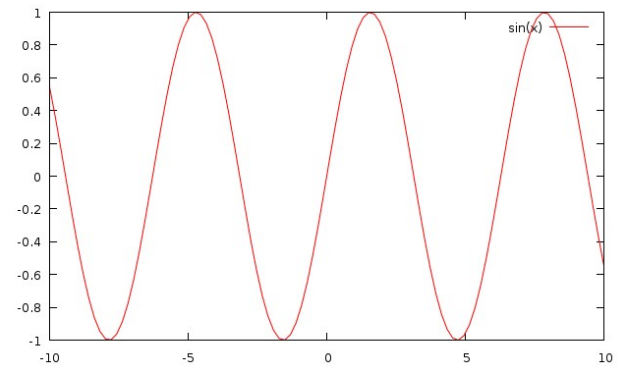
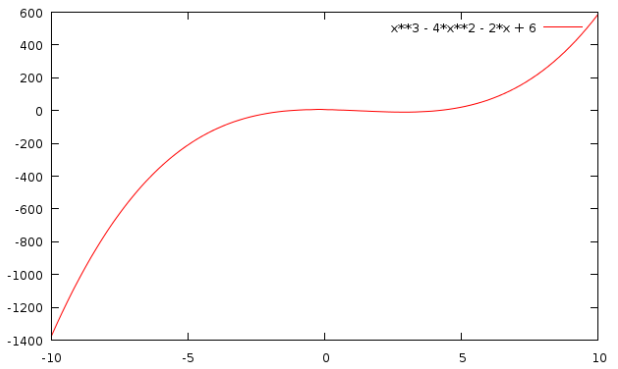
El resultado de todas las ejecuciones se guarda en un fichero de texto llamado `tiempos.dat`. Recuerde que para poder ejecutar este script debe tener permisos de ejecución. Si el nombre del mismo es, por ejemplo, `ejecuciones.csh` deberá ejecutar esta orden para darle dichos permisos:

```
chmod a+x ejecuciones.csh
```

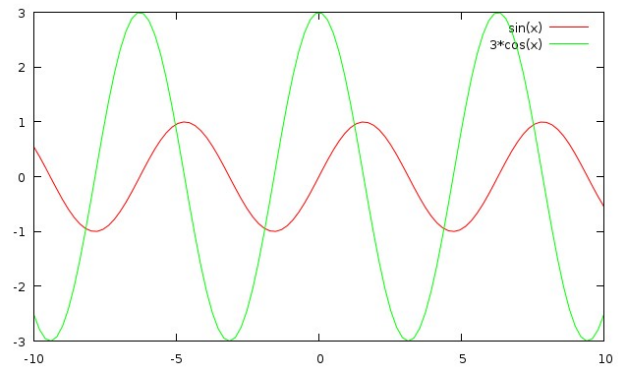
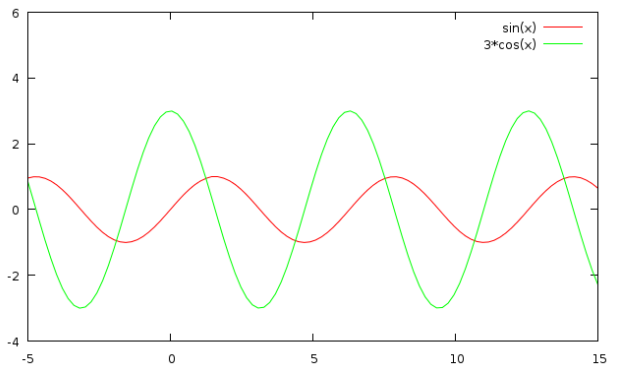
Observe también que los tiempos de ejecución obtenidos dependen del sistema que esté usando por lo que deberá acompañar sus informes siempre de datos que sean relevantes a este efecto tales como tipo de ordenador, compilador usado, opciones que se han usado para compilar, etc.

Para generar una gráfica como la anterior usaremos el programa `gnuplot`[1]. Este nos permite, entre otras cosas, dibujar tanto funciones como gráficas a partir de datos almacenados en ficheros. De esa forma podemos ver simultáneamente nuestros datos empíricos y alguna función teórica a la que deberían ajustarse. Al ejecutarlo nos aparecerá un prompt esperando instrucciones.

La instrucción `plot` permite realizar casi todo lo que a nosotros nos interesa. Por ejemplo, para dibujar una función en la variable x simplemente escribiremos `plot` seguido de la expresión de la función. Por ejemplo:

`plot sin(x)`	`plot x**3 - 4*x**2 - 2*x + 6`
	

También es posible dibujar varias funciones de forma simultánea o cambiar los rangos de valores representados (que como se ha visto antes, por defecto se determinan de forma automática):

`plot sin(x), 3*cos(x)`	`plot [-5:15][-4:6] sin(x), 3*cos(x)`
	

Además de funciones definidas de forma analítica, también se pueden dibujar funciones especificadas como parejas de números (abscisa y ordenada) y almacenadas en un fichero de texto plano. Para ello simplemente se debe ejecutar `plot` seguido del nombre del fichero entre

1 *Información sobre gnuplot: http://www.gnuplot.info/*

comillas dobles.

Continuando con el ejemplo anterior, tras ejecutar el script `ejecuciones.csh`, disponemos de un fichero `tiempos.dat` que contiene parejas de números indicando el tamaño del vector junto con el tiempo de ejecución. Dicho fichero se dibuja con la instrucción:

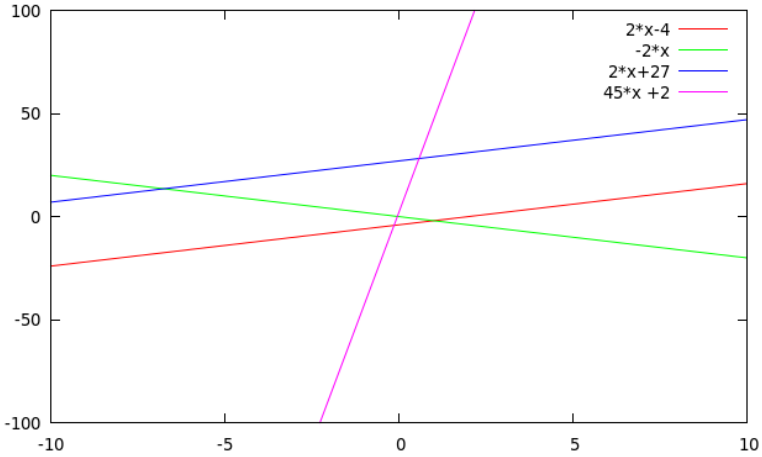
```
plot “tiempos.dat”
```

4. Ajuste de la eficiencia teórica

Cuando decimos que un algoritmo es de un determinado orden de eficiencia (notación O o Θ) lo que estamos es decir que hemos encontrado una clase de equivalencia para la función del tiempo de ejecución. Sin embargo, dentro de cada clase de equivalencia hay infinitas funciones. Por ejemplo, las funciones f(x)=3x y g(x)=500x-30 son ambas O(n).

El estudio empírico de la eficiencia puede servirnos para ajustar mejor cuál es la eficiencia teórica. Puesto que conocemos la forma teórica de la función y tenemos datos medidos de forma empírica podemos aplicar algún método de regresión para ajustar la curva a los datos.

Con `gnuplot` podemos hacer esto mediante la orden `fit`. Pongámonos en el ejemplo de la búsqueda lineal que se ha mostrado antes. Este algoritmo es O(n), es decir que la función que mide su eficiencia temporal es de la forma f(x)=a*x+b, sin embargo, el cálculo teórico que hemos hecho no puede concretar los valores a y b. A continuación vemos superpuestas 4 funciones de este tipo:



Como se puede ver, tenemos infinitas posibilidades. ¿Cuál de ellas se parece más a los datos empíricos de los que disponemos?

Para calcular a y b en nuestro ejemplo ejecutaremos esto en `gnuplot`:

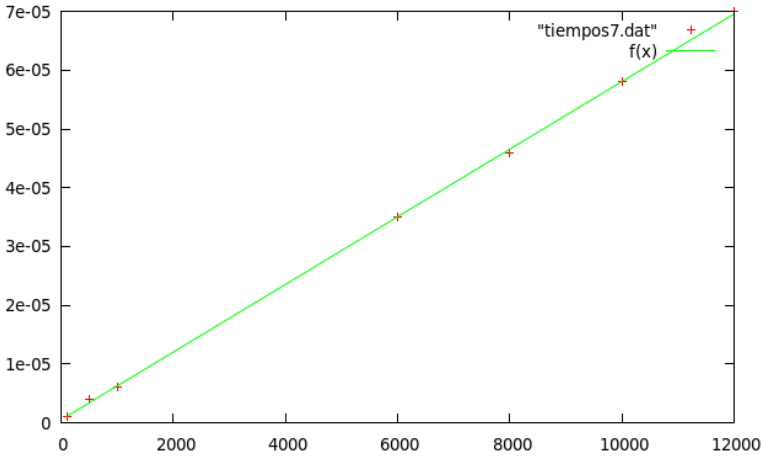
```
f(x) = a*x + b
fit f(x) “tiempos7.dat” via a, b
```

es decir, definimos la ecuación que define f(x) y a continuación le pedimos que haga el mejor ajuste entre esa función y los datos contenidos en el fichero `tiempos7.dat` (en donde están nuestros 7 resultados de ejecución). Como resultado obtendremos los valores a y b que producen un mejor ajuste entre la curva teórica y la empírica:

```
a = 5.7531e-09
b = 5.2623e-07
```

y acto seguido dibujaremos ambas funciones superpuestas:

```
plot "tiempos7.dat", f(x)
```



5. Ejercicios

5.1. Algoritmos de Ordenación Básicos.

Para los algoritmos de ordenación Inserción, Selección y Burbuja implementadlos y obtener su orden de eficiencia en el peor, mejor y caso promedio. Así como tiempos empíricos y ajustar a estos datos la eficiencia teórica en el peor caso con objeto de ajustar las constantes.

5.2. Algoritmos de Ordenación con estructuras jerárquicas.

Dados los T.D.A ABB (árbol binario de búsqueda) y APO (árbol parcialmente ordenado) generar un algoritmo de ordenación insertando las claves en el árbol considerado. En el caso que sea ABB si lo recorremos en inorden obtendremos los datos ordenados y si es un APO consultando y borrando el mínimo. Implementar tales algoritmos y obtener igualmente la eficiencia en el peor, mejor y caso promedio.

5.3. Algoritmo de Ordenación: fuerza bruta.

Dado el código que obtiene las permutaciones de 1 a n, permutaciones.cpp, establecer un algoritmo de ordenación tal que cada permutación determina una ordenación de los elementos de entrada. Para cada ordenación de elementos según la permutación considerada se comprobará si los elementos ya están ordenados. Por ejemplo si nuestros elementos son {7,5,4} tendríamos las siguientes posibilidades:

Permutación	**Vector Permutado**
1 2 3	7 5 4
1 3 2	7 4 5
2 1 3	5 7 4
2 3 1	5 4 7
3 1 2	4 7 5
3 2 1	***4 5 7***

Como se puede observar la única permutación que obtiene los elementos ordenados del vector original es 3 2 1 dando lugar al vector {4 ,5 ,7}. Implementar tal algoritmo y obtener igualmente la eficiencia en el peor, mejor y caso promedio. Igualmente obtener los tiempos empíricos y ajustar a estos datos la eficiencia teórica en el peor caso.

5.4. Algoritmos de Ordenación por Mezcla

Estudie el código del algoritmo recursivo disponible en el fichero *mergesort.cpp*. En él, se integran dos algoritmos de ordenación: inserción y mezcla (o mergesort). El parámetro *UMBRAL_MS* condiciona el tamaño mínimo del vector para utilizar el algoritmo de inserción en vez de seguir aplicando de forma recursiva el mergesort. Obtener la eficiencia teórica en el mejor, peor y caso promedio.

Realice un análisis de la eficiencia empírica y haga el ajuste de ambas curvas. Incluya también, para este caso, un pequeño estudio de cómo afecta el parámetro *UMBRAL_MS* a la eficiencia del algoritmo. Para ello, pruebe distintos valores del mismo y analice los resultados obtenidos.

Cambiar el algoritmo básico por los algoritmos jerárquicos desarrollados en el apartado 5.2 y

el algoritmo de ordenación basado en la fuerza bruta. Estudiar UMBRAL_MS en cada uno de los casos.

6. Informe de la eficiencia

En cada ejercicio, debe generar un fichero pdf indicando de forma clara:

- Código fuente
- Hardware usado (CPU, velocidad de reloj, memoria RAM, ...)
- Sistema operativo
- Compilador utilizado y opciones de compilación
- Desarrollo completo del cálculo de la eficiencia teórica y gráfica.
- Parámetros usados para el cálculo de la eficiencia empírica y gráfica.
- Ajuste de la curva teórica a la empírica: mostrar resultados del ajuste y gráfica.

7. Referencias

[GNUPLOT] Manual de gnuplot. http://www.gnuplot.info/