UNIVERSIDAD DE GRANADA

Departamento de Ciencias de la Computación e Inteligencia Artificial

Eficiencia teórica vs empírica

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

Estructuras de Datos

Grado en Ingeniería Informática
Doble Grado en Ingeniería Informática y Matemáticas
Doble Grado en Ingeniería Informática y ADE

1.- Introducción

Los objetivos de este ejercicio son los siguientes:

1. Aprender a calcular la eficiencia teórica de un código junto con su orden de eficiencia en el peor de los casos.
2. Aprender a obtener la eficiencia empírica de un código.
3. Aprender a realizar un ajuste de la curva de eficiencia teórica a la empírica.

2.- Eficiencia teórica

La eficiencia teórica viene dada por una función que da una cota superior 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. 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:

$$T(n)=1+4+(\sum_{i=0}^{n-1} 2+4)+1+max(1,1)=6n+7$$

(2) (3) (3) (4) (5) (6) (8)

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(n)=6n+7 ∈ O(n) podemos afirmar que el orden de eficiencia del algoritmo de búsqueda secuencial es O(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 `ctime` que incluye los siguientes recursos:

- `clock_t` es un nuevo tipo de dato que representa cantidades de ticks de reloj.
- `CLOCKS_PER_SEC` es una macro que vale el número de ticks por segundo que es capaz de realizar nuestro sistema.
- `clock()` es una función que devuelve el número de ticks que han transcurrido desde un momento determinado. Generalmente estos ticks son referidos al comienzo de la ejecución pero esto puede variar.

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

```
clock_t tini;    // Anotamos el tiempo de inicio
tini=clock();

// Algoritmo
// …

clock_t tfin;    // Anotamos el tiempo de finalización
tfin=clock();

// Mostramos resultados
cout << "Ticks de reloj : " << tfin-tini << endl;
cout << "Segundos       : " << (tfin-tini)/(double)CLOCKS_PER_SEC
<< 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.

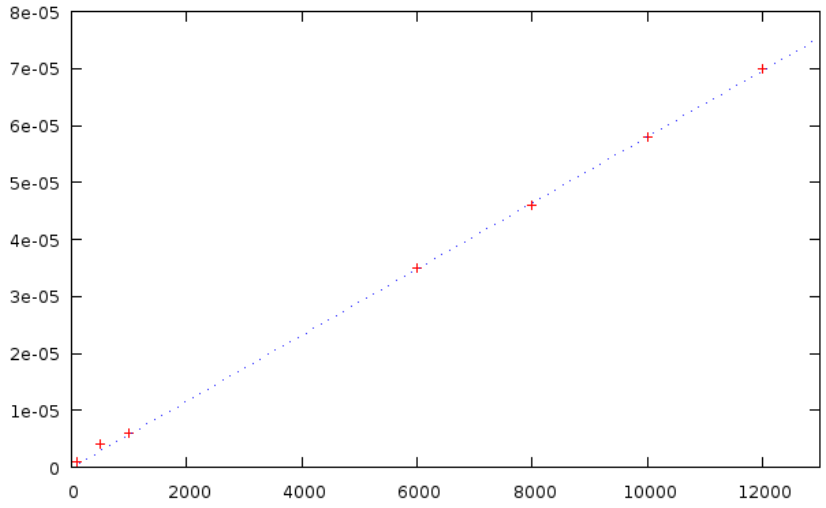
Recuerde que para crear el ejecutable en un sistema GNU/Linux desde la línea de órdenes, y asumiendo que el programa está almacenado en un fichero `busqueda_lineal.cpp`, debe ejecutar: `g++ busqueda_lineal.cpp -o busqueda_lineal`

```
#include <iostream>
#include <ctime>    // Recursos para medir tiempos
#include <cstdlib>  // Para generación de números pseudoaleatorios
using namespace std;

int buscar(const int *v, int n, int x) {
  int i=0;
  while (i<n && v[i]!=x)
    i=i+1;
  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[

  clock_t tini;    // Anotamos el tiempo de inicio
  tini=clock();

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

  clock_t tfin;    // Anotamos el tiempo de finalización
  tfin=clock();

  // Mostramos resultados (Tamaño del vector y tiempo de ejecución en seg.)
  cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << 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 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:

```
#!/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
```

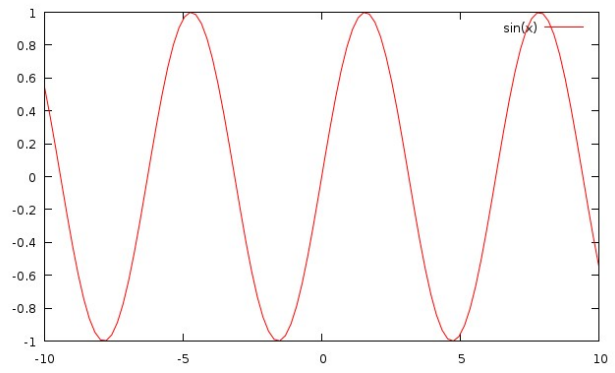
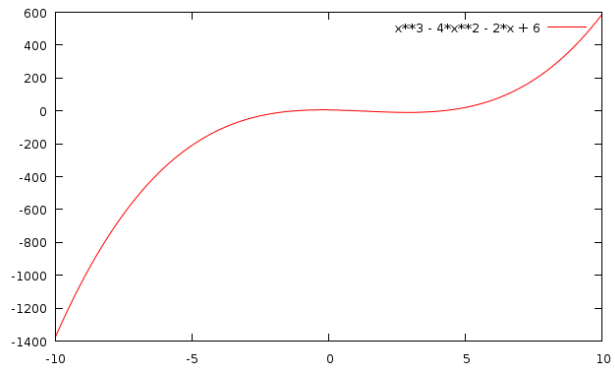
El resultado de todas las ejecuciones se guarda en un fichero de texto llamado `tiempos.dat`. Recuerde que para poder ejecutar este script debe tener permisos de ejecución. Si el nombre del mismo es, por ejemplo, `ejecuciones.csh` deberá ejecutar esta orden para darle dichos permisos:

```
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.

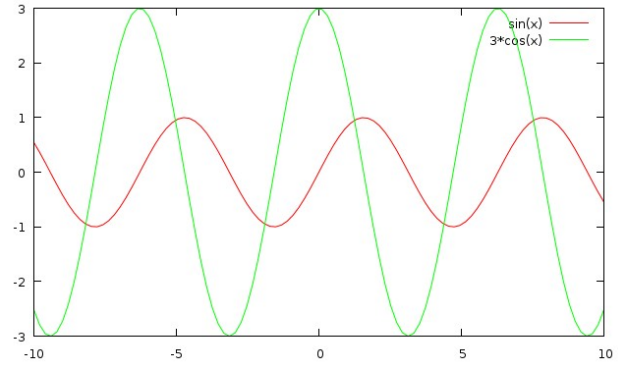
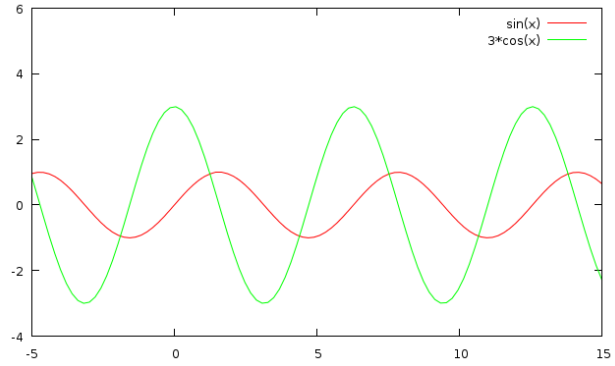
Para generar una gráfica como la anterior usaremos el programa `gnuplot`[1]. Este nos permite, entre otras cosas, dibujar tanto funciones como gráficas a partir de datos almacenados en ficheros. De esa forma podemos ver simultáneamente nuestros datos empíricos y alguna función teórica a la que deberían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

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

`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 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”
```

Ejercicio 1: ***Ordenación de la burbuja***

El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:

```
void ordenar(int *v, int n) {
  for (int i=0; i<n-1; i++)
    for (int j=0; j<n-i-1; j++)
      if (v[j]>v[j+1]) {
        int aux = v[j];
        v[j] = v[j+1];
        v[j+1] = aux;
      }
}
```

Calcule la eficiencia teórica de este algoritmo. A continuación replique el experimento que se ha hecho antes (búsqueda lineal) con este nuevo código. Debe:

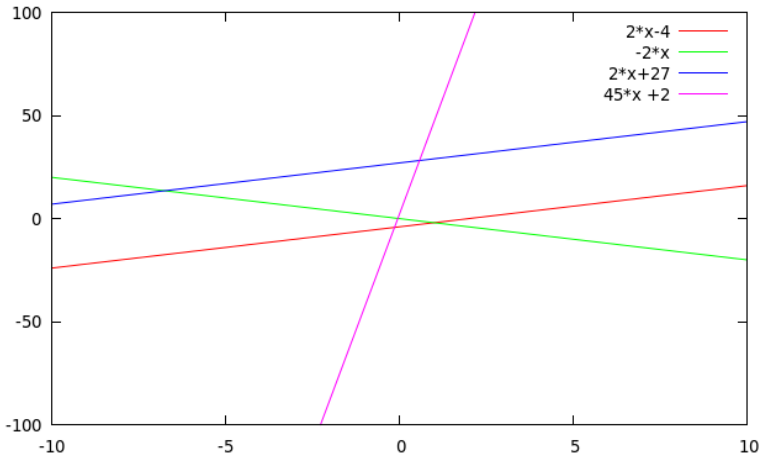
- Crear un fichero `ordenacion.cpp` con el programa completo para realizar una ejecución del algoritmo.
- Crear un script `ejecuciones_ordenacion.csh` en C-Shell que permite ejecutar varias veces el programa anterior y generar un fichero con los datos obtenidos.
- Usar `gnuplot` para dibujar los datos obtenidos en el apartado previo.

Los datos deben contener tiempos de ejecución para tamaños del vector 100, 600, 1100, …, 30000. Pruebe a dibujar superpuestas la función con la eficiencia teórica y la empírica. ¿Qué sucede?

4.- Ajuste de la eficiencia teórica

Cuando decimos que un algoritmo es de un determinado orden de eficiencia (notación O) lo que estamos haciéndolo es diciendo es 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?

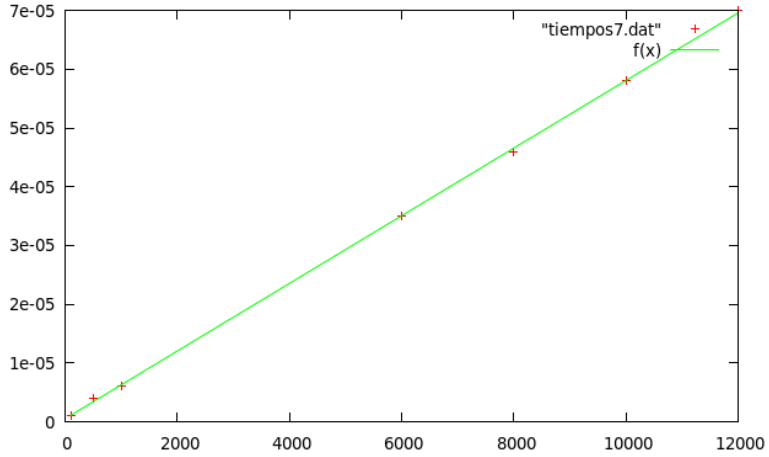
Para calcular a y b en nuestro ejemplo ejecutaremos esto en `gnuplot`:

```
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)
```

Ejercicio 2: ***Ajuste en la ordenación de la burbuja***

Replique el experimento de ajuste por regresión a los resultados obtenidos en el ejercicio 1 que calculaba la eficiencia del algoritmo de ordenación de la burbuja. Para ello considere que f(x) es de la forma ax^2+bx+c

5.- Falta de precisión en la medición

Observe que si el algoritmo que estamos cronometrando es muy rápido o se aplica sobre problemas de tamaño muy pequeño, el tiempo en segundos podría ser 0 ya que los recursos de `ctime` no tienen tanta precisión como sería deseable para estas tareas. En estas situaciones podemos optar por ejecutar el mismo algoritmo muchas veces y dividir el tiempo total de ejecución por el número de ejecuciones.

Ejercicio 3: ***Problemas de precisión***

Junto con este guión se le ha suministrado un fichero ejercicio_desc.cpp. En él se ha implementado un algoritmo. Se pide que:

- Explique qué hace este algoritmo.
- Calcule su eficiencia teórica.
- Calcule su eficiencia empírica.

Si visualiza la eficiencia empírica debería notar algo anormal. Explíquelo y proponga una solución. Compruebe que su solución es correcta. Una vez resuelto el problema realice la regresión para ajustar la curva teórica a la empírica.

6.- Mejor y peor caso

Cabe esperar que la mayoría de la veces que ejecutamos nuestros programas estos trabajan sobre datos “normales” en el sentido de que no son ni los mejores posibles ni los peores posibles. Por ejemplo, si hacemos un programa para ordenar números, lo que cabe esperar es que reciba números desordenados. Sin embargo, puede ocurrir que en determinadas ocasiones los datos tengan alguna peculiaridad que afecte considerablemente al tiempo de ejecución. Por ejemplo, si al algoritmo de búsqueda lineal le pedimos que busque un elemento que se encuentra en la primera posición del vector nos encontraríamos en el mejor caso posible desde el punto de la eficiencia temporal, mientras que si le pedimos que busque un elemento que no está estaríamos en el peor caso posible.

Ejercicio** 4: **Mejor y peor caso

Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Debe modificar el código que genera los datos de entrada para situarnos en dos escenarios diferentes:

- El mejor caso posible. Para este algoritmo, si la entrada es un vector que ya está ordenado el tiempo de cómputo es menor ya que no tiene que intercambiar ningún elemento.
- El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada iteración del bucle interno hay que hacer un intercambio.

Calcule la eficiencia empírica en ambos escenarios y compárela con el resultado del ejercicio 1. Aunque lo más frecuente será preguntar por números que estén en posiciones arbitrarias del vector (casos promedio).

***Ejercicio** 5: Dependencia de la implementación*

Considere esta otra implementación del algoritmo de la burbuja:

```
void ordenar(int *v, int n) {
  bool cambio=true;
  for (int i=0; i<n-1 && cambio; i++) {
    cambio=false;
    for (int j=0; j<n-i-1; j++)
      if (v[j]>v[j+1]) {
        cambio=true;
        int aux = v[j];
        v[j] = v[j+1];
        v[j+1] = aux;
      }
  }
}
```

En ella se ha introducido una variable que permite saber si, en una de las iteraciones del bucle externo no se ha modificado el vector. Si esto ocurre significa que ya está ordenado y no hay que continuar.

Considere ahora la situación del mejor caso posible en la que el vector de entrada ya está ordenado. ¿Cuál sería la eficiencia teórica en ese mejor caso? Muestre la gráfica con la eficiencia empírica y compruebe si se ajusta a la previsión.

6.1 Dependencia del entorno

No cabe duda de que el hardware en el que ejecutamos el algoritmo puede influir bastante en los tiempos de ejecución. También influye el software, en particular puede influir el proceso de

Ejercicio 6: ***Influencia del proceso de compilación***

Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Ahora replique dicho ejercicio pero previamente deberá compilar el programa indicándole al compilador que optimice el código. Esto se consigue así:

```
g++ -O3 ordenacion.cpp -o ordenacion_optimizado
```

Compare las curvas de eficiencia empírica para ver cómo mejora esto la eficiencia del programa.
compilación del programa.

9.- Referencias

[GAR06b] Garrido, A. Fdez-Valdivia, J. "*Abstracción y estructuras de datos en C++*". Delta publicaciones, 2006.

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