II Doble Grado Ingieniería Infomática y Matemáticas

Estructura de Datos

Práctica 1 - Eficiencia

Pedro Bonilla Nadal, Sofía Almeida Bruno,
Jesús Sánchez de Lechina Tejada

Características

El sistema operativo usado ha sido **Ubuntu 16.04**
Con una memoria **1,8 GiB RAM**
Y con las especificaciones: **Intel® Core™ i7-4700MQ CPU @ 2.40GHz**

Opciones de compilación:

Salvo en los ejercicios expresamente indicados (ejercicio 6, que le pide una optimización expresa al compilador) todos los ejecutables han sido generados usando g++ generando directamente el ejecutable a partir del único archivo de código fuente con la opción -o

$$g++ \quad -o\ ejecutable\ fuente.cpp$$

Dibujo de gráficas:

Para esta representación se ha hecho uso exclusivamente de gnuplot, en concreto su orden plot, y se ha hecho una exportación directa a este documento

Ejercicio 1. El siguiente código realiza la ordenación mediante el algoritmo de la burbuja:

```
1. void ordenar(int *v, int n) {
2.       for (int i=0; i<n-1; i++)
3.            for (int j=0; j < n-i-1; j++) {
4.                 if (v[j] > v[j+1]) {
5.                      int aux = v[j];
6.                      v[j] = v[j+1];
7.                      v[j+1] = aux;
8.                 }
9.            }
10. }
```

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 fichero ejecuciones_ordenación.csh 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?

Solución.
El archivo ordenación.cpp y ejecuciones_ordenacion.csh se encuentran adjuntos en la práctica.

Eficiencia teórica:
Línea 2: 5 OE (2 asignaciones, 3 operaciones aritmético-lógica).
Línea 3: 6 OE (2 asignaciones, 4 operaciones aritmético-lógica).
Línea 4-7: 13 OE (6 accesos a vector, 4 operaciones artimético-lógicas, 3 asignaciones).

Entonces:$= 1 + \sum_{i=0}^{n-2}(5 + \sum_{j=0}^{n-i-2}(5+13)) = 1 + \sum_{i=0}^{n-2}(5 + ((n-i-2)*18)) =$
$1 + \sum_{i=0}^{n-2}(5) + \sum_{i=0}^{n-2}(18n) + \sum_{i=0}^{n-2}(18n) + \sum_{i=0}^{n-2}(18i) + \sum_{i=0}^{n-2}(18) = -4 + 4n + 9n^2.$

Nota: debemos considerar que para ordenar un vector debe tener al menos 2 elementos.

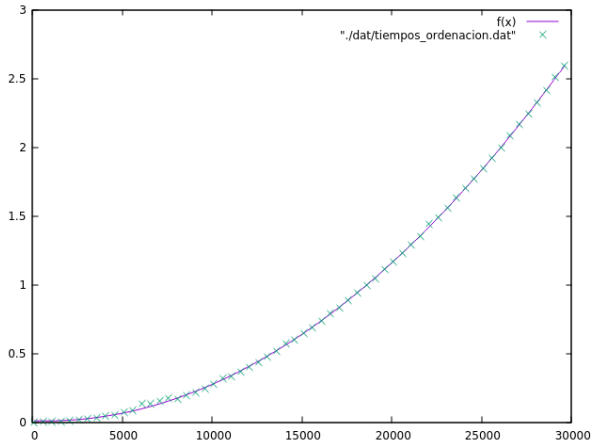
Ejercicio 2. Replique el experimento de ajuste por regresión a los resultados obtenidos en el ejercicio 1 que calculaba la eficiencia del algoritmo de ordenación de la burbuja. Para ello considere que f(x) es de la forma ax2+bx+c.

Solución:
Para la resolución de este ejercicio, utilizaremos la orden fit, con los datos previamente obtenidos, para hacer la regresión, en este caso cuadrática.

Datos obtenidos:
Teniendo la gráfica $f(x) = ax^2 + bx + c$. la regresión se obtiene mediante los comandos:
gnuplot> f(x) = a*x*x + b*x + c
gnuplot> fit f(x) "tiempos_ordenacion.dat"via a, b, c

Queda como resultado:
a = 3.108e-09 +/- 1.64e-11 (0.5277 %)
b = -4.758e-06 +/- 5.035e-07 (10.58 %)
c = 0.0107285 +/- 0.003235 (30.15 %)

Figura 1: Comparación eficiencia teórica(f(x)), eficiencia práctica



Ejercicio 3. 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.

Solución:
El algortimo *operacion* busca en un vector la posición de un elemento concreto. Para ello, utiliza tres índices: inf, sup y med; además de un booleano que indica si se ha encontrado o no. Al principio, inf apunta al primer elemento del vector, sup al último y med al elemento central. Mientras inf sea menor que sup y no hayamos encontrado al elemento, entraremos en un bucle que asigna a med el elemento medio entre inf y sup y compara el elemento que se encuentra en dicha posición con el elemento buscado para, en caso de no encontrarlo, asignar a inf el elemento siguiente a med (si es menor que el elemento buscado) o a sup el anterior a med. Así, en las sucesivas iteraciones del bucle vamos comparando con el elemento que se encuentra en la mitad, en la mitad de la mitad, en la mitad de la mitad de la mitad,... Este es el algoritmo conocido como búsqueda binaria.

```
1. int operacion(int *v, int n, int x, int inf, int sup) {
2.       int med;
3.       bool enc=false; {
4.       while ((inf<sup) && (!enc))
5.            med = (inf+sup)/2;
6.            if (v[med]==x)
7.                 enc = true;
8.            else if (v[med] < x)
9.                 inf = med+1;
10.           else
11.                sup = med-1;
12.      }
13.      if (enc)
14.           return med;
15.      else
16.           return -1;
17. }
```

Eficiencia teórica:
Línea 2 y 3: 1 OE (declaración de variable).
Línea 4: 3 OE (3 operaciones aritmético-lógicas).
Línea 5: 3 OE (2 operaciones artimético-lógicas, 1 asignación).
Líneas 6-11: 4 OE (aplicando la regla del máximo: 1 acceso a vector, 2 operaciones aritmético-lógicas, 1 asignación
Líneas 13-17: 2 OE (1 operación aritmético-lógica, 1(return?))

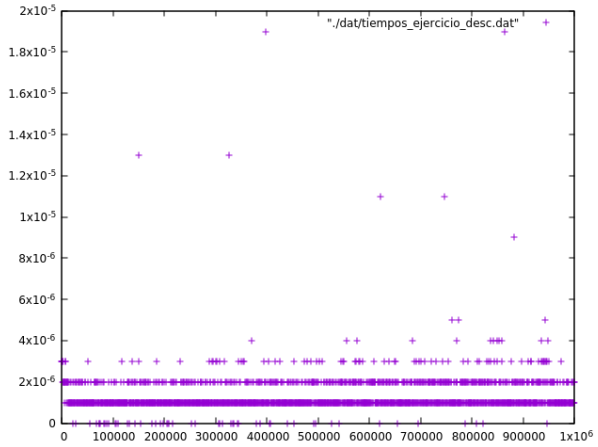
Al ser los otros bloques O(1) es, por tanto, el bucle de la línea 4 el que determina la eficiencia del algoritmo. Como los índices involucrados en la condición del mismo varían dividiendose entre dos, vamos quedándonos cada vez con menos elementos, de forma que la eficiencia es O(log2n).

Concluimos, aplicando la regla de la suma, que la eficiencia de este algoritmo es O(log2n).

Eficiencia empírica

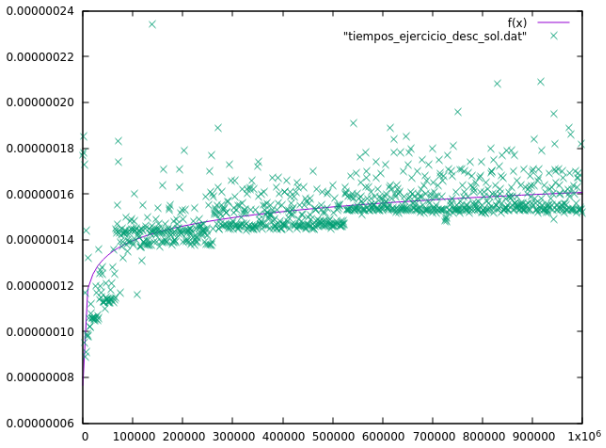
Para calcular la eficiencia empírica creamos mediante /src/ejercicio_desc.cpp el ejecutable /bin/ejercicio_desc. El guión /csh/ejercicio_desc.csh guarda en /dat/tiempos_ejercicio_desc.dat los tiempos de ejecución del programa con datos entre 100 y 1000000. Usando gnuplot, generamos la siguiente gráfica:

Figura 2: Tiempos operacion



Observamos que esta gráfica no es logarítmica, como habíamos predicho de forma teórica, sino que los valores se mantienen en tiempos relativamente constantes. Esto se debe a que la precisión con que medimos los tiempos no es suficiente. Como solución se aporta el archivo /src/ejercicio_desc_sol.cpp. En él se realiza, para cada tamaño, 1000 veces *operacion* y el tiempo tardado se divide entre 1000. Una vez realizada la regresión, observamos en la siguiente gráfica cómo queda la eficiencia teórica frente a la empírica:

Figura 3: Comparación eficiencia teórica(f(x)), eficiencia práctica



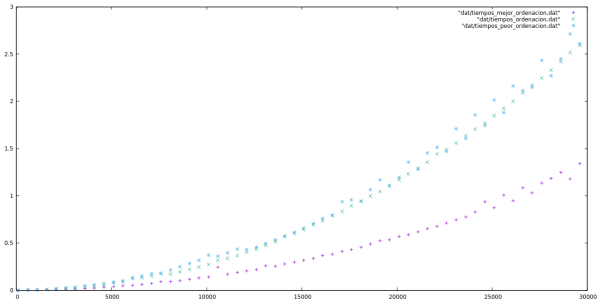
Ejercicio 4. Retome el ejercicio de ordenación mediante el algoritmo de la burbuja. Debe modificar el código que genera los datos de entrada para situarnos en dos escenarios diferentes:

- 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.

Solución:
Como solución a este ejercicio creamos dos nuevos archivos .cpp que recogen ambos casos. A continuación, ofrecemos la gráfica en que se comparan los tiempos de ejecución de los tres casos:

Figura 4: Comparación ordenación normal/mejor caso/peor caso



Se aprecia un menor tiempo de ejecución cuando el vector ya está ordenado frente a cuando está ordenado de forma inversa o desordenado de forma aleatoria.

Ejercicio 5. Dependencia de la implementación:

```
1. void ordenar(int *v, int n) {
2.       bool cambio=true;
3.       for (int i=0; i<n-1 && cambio; i++) {
4.            cambio=false; {
5.            for (int j=0; j<n-i-1; j++){
6.                 if (v[j]>v[j+1]) {
7.                      cambio=true;
8.                      int aux = v[j];
9.                      v[j] = v[j+1];
10.                     v[j+1] = aux;
11.                }
12.           }
13.      }
14. }
```

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.

Cálculo eficiencia teórica:

En cada línea de código se producen las siguientes operaciones elementales:
Línea 2: 1 OE inicialización booleana
Línea 3: 4 OE inicial. " i ", comparación "<", resta "n-1", incremento " i++"
Línea 4: 1 OE asignación booleana
Línea 5: 5 OE inicial. "j", comp. "<", doble resta "n-i-1", incremento "j++"

Línea 6: 4 OE accesos a v[j] y v[j+1], suma "j+1", comp. ">"
Línea 7: 1 OE asign. bool.
Línea 8: 2 OE inicial. " aux", acceso a v[j]
Línea 9: 4 OE acceso a v[j] y v[j+1], suma "j+1", asignación
Línea 10: 3 OE acceso a v[j+1], suma "j+1", asignación

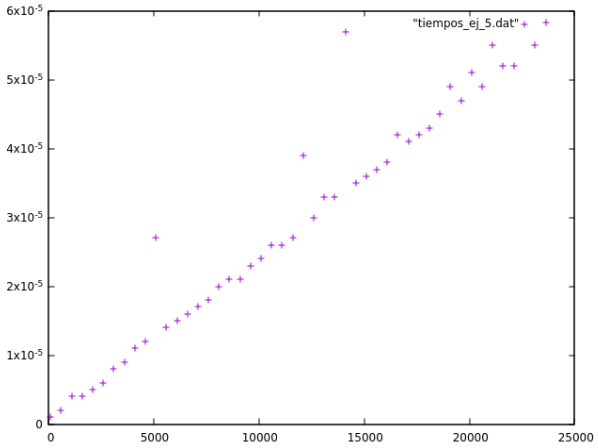
*Las líneas 7-10 no sucederían en el mejor de los casos, cuando el vector está ordenado

Por tanto, en el mejor de los casos, donde el primer bucle sólo produce una iteración:

$$T(n) = 1 \ + \ 1 \ + \sum_{j=0}^{n-1} 4 = 1 + 1 + (\frac{4+4}{2}) \cdot (n-1) = 4n - 2$$

Eficiencia práctica:

Figura 5: Gráfica eficiencia práctica



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.

Figura 6: Comparación ordenación normal/optimizado