Práctica de eficiencia

Yábir García Benchakhtir

5 de octubre de 2017

Especificaciones técnicas

Los ejercicios de esta práctica se han realizado en un ordenador de sobremesa con las siguientes características:

- CPU: Intel® Pentium(R) CPU G3258 @ 3.20GHz x 2
- RAM: 7,7 GiB DDR3
- SO: Ubuntu Linux 16.04.4 LTS 64-bits

Como compilador se ha usado *g++* en la version `g++ (Ubuntu 5.4.0-6ubuntu1 16.04.4) 5.4.0 20160609`
Los archivos .fit y los programas cpp utilizados se adjuntan en un comprimido clasificados por ejercicio.

Ejercicio 1

Crear un fichero ordenacion.cpp con el programa completo para realizar una ejecución del algoritmo.

La eficiencia teórica de la función es $O(n^2)$. A modo de práctica se intentó calcular el polinomio que da dicha complejitud obteniendo como resultado:

$$8{,}5n^2 - 6n + 4{,}5$$

Dicha aproximación teórica nos da una idea de la forma que pueden tener nuestros datos cuando los representemos enfrentando tamaño del problema contra tiempo de ejecución, aunque lo que más nos interesa conocer es que es un polinomio de segundo grado.

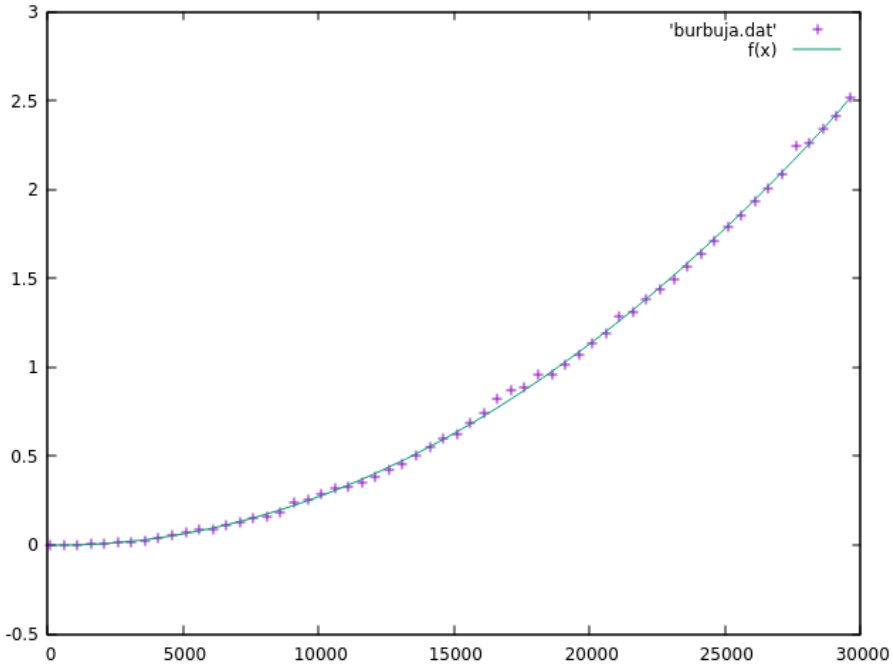
Crear un script en C-Shell que permite ejecutar varias veces el programa anterior y generar un fichero con los datos obtenidos.

El script de bash utilizado para realizar la recogida de datos ha sido el siguiene:

```
#!/bin/bash
inicio=100
fin="$4"
incremento="$3"
ejecutable="$1"
salida="$2"

echo "ejecutable: $ejecutable"
echo "salida: $salida"
echo "incremento: $incremento"
echo "fin: $fin"

i=$inicio

while [ $i -lt $fin ]
do
    echo "Ejecutando tam = $i"
    echo `./$ejecutable $i $fin` >> $salida
    ((i+=incremento))
done
```

Usar gnuplot para dibujar los datos obtenidos en el apartado previo.

Figura 1: Eficiencia empírica



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

```
gnuplot> f(x) = a*x**2 + b*x + c
gnuplot> fit f(x) 'burbuja.dat' via a,b,c
iter      chisq       delta/lim  lambda   a             b             c
   0 9.4779953800e+18   0.00e+00  2.29e+08    1.000000e+00   1.000000e+00   1.000000e+00
   1 2.8930837514e+14  -3.28e+09  2.29e+07    5.483216e-03   9.999583e-01   1.000000e+00
   2 1.1088359994e+09  -2.61e+10  2.29e+06   -4.157021e-05   9.999560e-01   1.000000e+00
   3 1.1074771167e+09  -1.23e+02  2.29e+05   -4.186834e-05   9.997456e-01   1.000000e+00
   4 1.0623066684e+09  -4.25e+03  2.29e+04   -4.100547e-05   9.791424e-01   9.999972e-01
   5 1.1026258621e+08  -8.63e+05  2.29e+03   -1.320633e-05   3.153609e-01   9.999082e-01
   6 2.4744107522e+03  -4.46e+09  2.29e+02   -5.582392e-08   1.356232e-03   9.998648e-01
   7 6.9496420833e+00  -3.55e+07  2.29e+01    6.677312e-09  -1.361516e-04   9.997330e-01
   8 6.7706268175e+00  -2.64e+03  2.29e+00    6.631652e-09  -1.344808e-04   9.867374e-01
   9 1.2756014394e+00  -4.31e+05  2.29e-01    4.532482e-09  -5.929887e-05   4.257803e-01
  10 1.7360191103e-02  -7.25e+06  2.29e-02    2.950202e-09  -2.629377e-06   2.950736e-03
  11 1.7288698672e-02  -4.14e+02  2.29e-03    2.938185e-09  -2.199011e-06  -2.603657e-04
  12 1.7288698672e-02  -2.38e-06  2.29e-04    2.938184e-09  -2.198978e-06  -2.606096e-04
iter      chisq       delta/lim  lambda   a             b             c

After 12 iterations the fit converged.
final sum of squares of residuals : 0.0172887
rel. change during last iteration : -2.38461e-11

degrees of freedom    (FIT_NDF)                        : 57
rms of residuals      (FIT_STDFIT) = sqrt(WSSR/ndf)    : 0.0174158
variance of residuals (reduced chisquare) = WSSR/ndf   : 0.000303311

Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a               = 2.93818e-09      +/- 3.354e-11    (1.142%)
b               = -2.19898e-06     +/- 1.029e-06    (46.81%)
c               = -0.00026061      +/- 0.006615     (2538%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.968  1.000
c               0.738 -0.861  1.000
gnuplot> a = 2.93818e-09
gnuplot> b = -2.19898e-06
gnuplot> c = -0.00026061
gnuplot> plot 'burbuja.dat', f(x)
```

Ejercicio 3

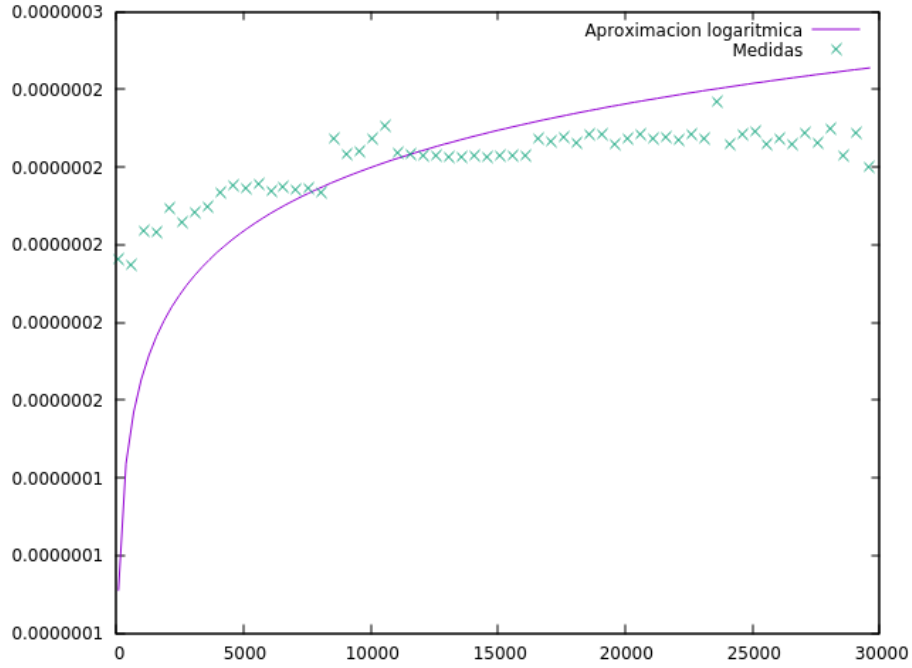
Junto con este guión se le ha suministrado un fichero *ejercicio-desc.cpp*. En él se ha implementado un algoritmo. Se pide que:

1. **Explique qué hace este algoritmo.** Se trata de un algoritmo de busqueda binaria. Se parte de un vector ordenado y se eligen dos variables que representan los extremos del vector. Una vez fijadas las variables *sup* e *inf* seleccionamos la posición que está en el *'centro'* respecto a ambas. Si coincide con lo que buscamos hemos terminado, si es más granda cambiamos el extremo infererior a este valor del *'centro'* y si es menor cambiamos el extremo superior.

2. **Calcule su eficiencia teórica.** La complejidad teórica es $log_2(n)$ ya que fuera del bucle *while* todo tiene complejidad $O(1)$ y dentro, en cada iteración, reducimos la condición en un orden de dos. Por tanto el algoritmo tiene la mayor de las complejidades, $log_2(n)$.

3. **Calcule su eficiencia empírica.**

Figura 2: Eficiencia empírica



Se presentó el inconveniente de que las medidas en tiempo eran muy pequeñas y por lo tanto muchos datos habían tomado como tiempo de ejecución el valor 0. Para solucionar este problema se procedió a calcular la media aritmética del tiempo empleado en ejecutar la función con los mismos datos un número determinado de veces.

Por algún motivo la nube de puntos obtenida no se ajusta a una función logarítimica de manera óptima. Se ha probado a tomar más datos y a cambiar los intervalos en los que se toma pero no se obtuvieron medidas mejores.

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.

Para crear un vector ordenado hemos usado la siguiente modificación (se han usado numero aleatorios aunque esto no afecta de manera alguna):

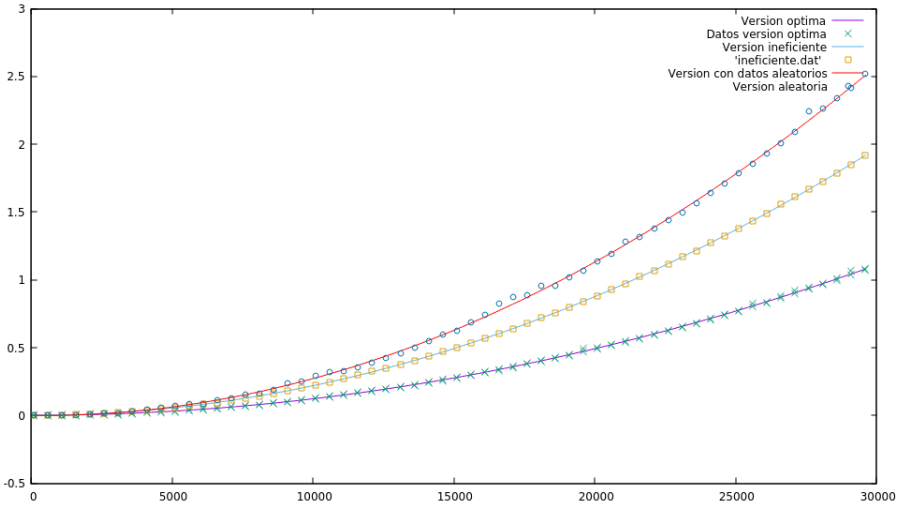
```
v[0] = 0;
for (int i=1; i<tam; i++){  // Recorrer vector
  v[i] = v[i-1] + rand() % 50;    // Generar aleatorio [0,vmax[
}
```

y para crear un vector decreciente:

```
v[0] = vmax;
for (int i=1; i<tam; i++){  // Recorrer vector
  v[i] = v[i-1] - rand() % 100;    // Generar aleatorio [0,vmax[
}
```

El resultado ha sido el siguiente:

Figura 3: Comparación de modificaciones en el algoritmo bubble sort

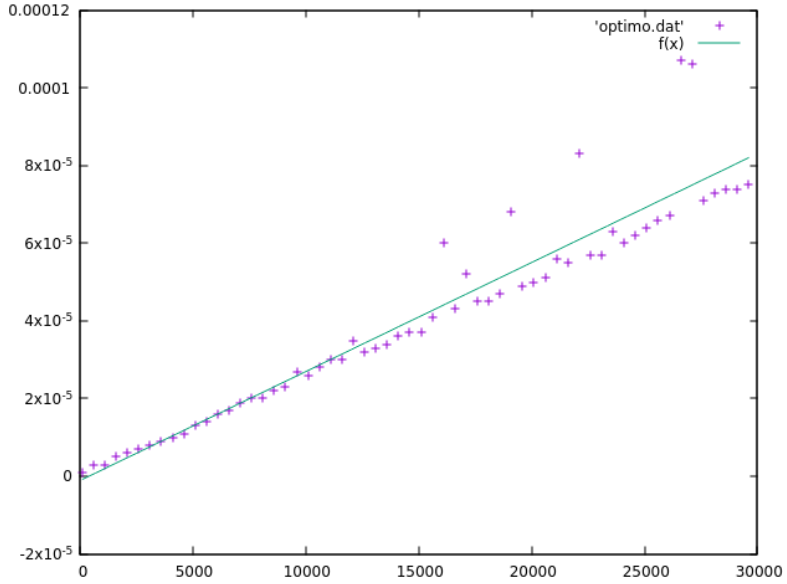


Por algún motivo, la gráfica cuando la función toma valores aleatorios está por encima de cuando toma valores en el caso más desfavorable. No se ha debido a un error en los vectores ya que estos se generan correctamente.

Ejercicio 5

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.

El caso en el cual todos los elementos del vector estan ordenados el programa solo realiza operaciones elementales (comparaciones y operaciones) n veces, luego la complejitud del problema será lineal, $O(n)$.



Ejercicio 6

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

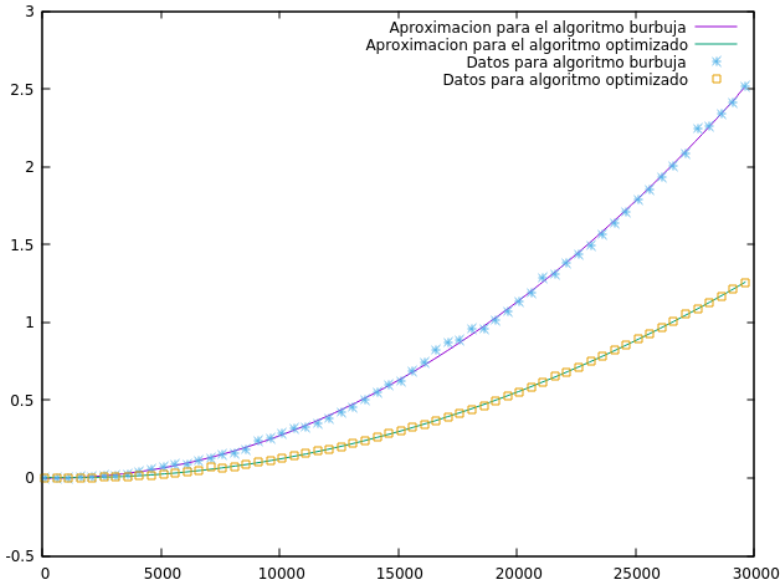
Hemos compilado el programa indicado de dos maneras distintas:

g++ -O3 ordenacion.cpp -o burbuja

g++ odernacion.cpp -o optimizado

Comprobamos al representar la nube de datos como se reduce de manera considerable el tiempo de ejecución de la version *optimizada* respecto de la *no optimizada*. Ambas curvas se ajustan a una curva del tipo $ax^2 + bx + c$

Figura 4: Algoritmo bubble sort optimizado por el compilador



Ejercicio 7

Implemente un programa que realice la multiplicación de dos matrices bidimensionales. Realice un análisis completo de la eficiencia tal y como ha hecho en ejercicios anteriores de este guión.

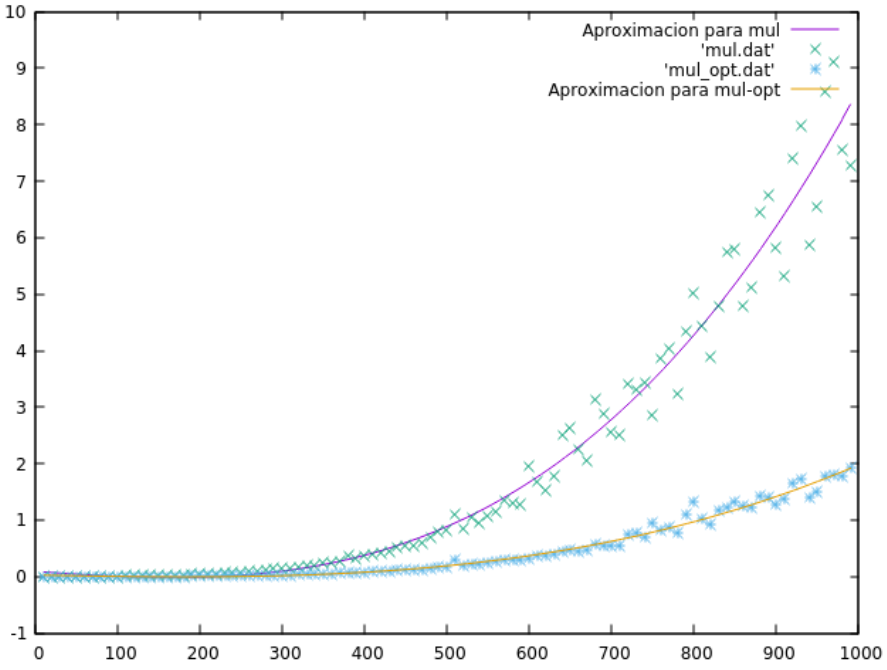
En este ejercicio se ha implementado un programa que utiliza el método tradicional para multiplicar matrices y se ha diseñado un test donde se expone este programa a la multiplicación de dos matrices de iguales dimensiones. El test comienza multiplicando matrices de 10x10 con valores aleatorios y realiza en cada test un incremento de 10 unidades.

A la hora de compilar el código se han generado dos programas con las ordenes siguientes:

```
g++ -o prod_opt mult.cpp -O3
g++ -o prod mult.cpp
```

Se ha obtenido la siguiente gráfica que representa los datos obtenidos para ambos tests:

Figura 5: Multiplicacion de matrices n x n



Código implementado: Disponible en la carpeta *ejer7/mult.cpp*

Ejercicio 8

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 UMBRALMS a la eficiencia del algoritmo. Para ello, pruebe distintos valores del mismo y analice los resultados obtenidos.