Estructura de datos. Práctica 1: Informe de eficiencia.

Francisco Navarro Morales

12 de octubre de 2016

Hardware utilizado y Sistema Operativo.

Sudo lscpu:

```
Architecture: x86_64
CPU op-mode(s): 32-bit, 64-bit
Byte Order: Little Endian
CPU(s): 8
On-line CPU(s) list: 0-7
Thread(s) per core: 2
Core(s) per socket: 4
Socket(s): 1
NUMA node(s): 1
Vendor ID: GenuineIntel
CPU family: 6
Model: 60
Model name: Intel(R) Core(TM) i7-4712MQ CPU @ 2.30GHz
Stepping: 3
CPU MHz: 2301.000
CPU max MHz: 2301.0000
CPU min MHz: 800.0000
BogoMIPS: 4589.37
Virtualization: VT-x
L1d cache: 32K
L1i cache: 32K
L2 cache: 256K
L3 cache: 6144K
NUMA node0 CPU(s): 0-7
```

Compilador:

gcc -v :

```
Using built-in specs.
COLLECT_GCC=/usr/x86_64-pc-linux-gnu/gcc-bin/4.9.3/gcc
COLLECT_LTO_WRAPPER=
/usr/libexec/gcc/x86_64-pc-linux-gnu/4.9.3/lto-wrapper
Target: x86_64-pc-linux-gnu
```

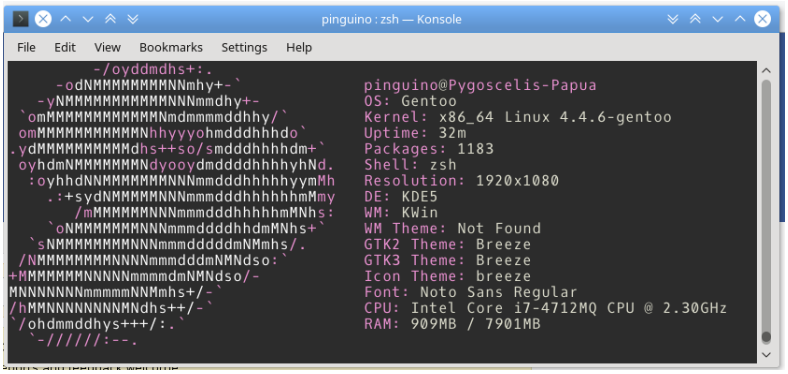


Figura 1: Información sobre el sistema operativo

Ejercicio 1.

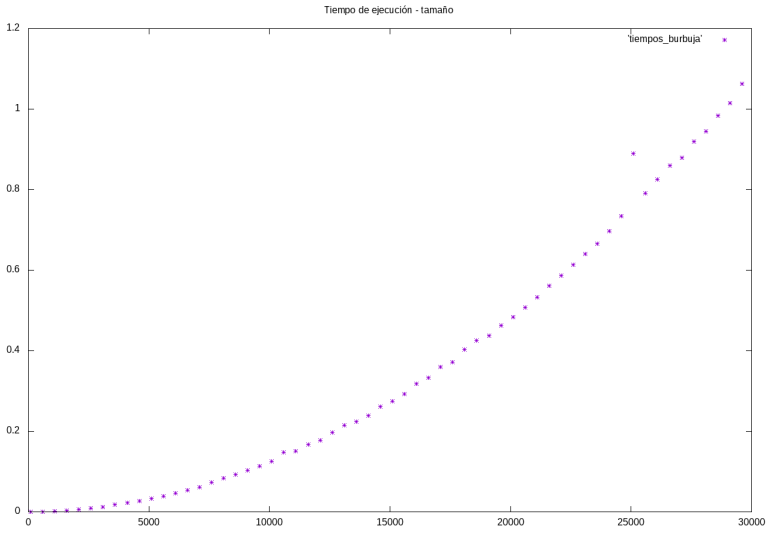
El codigo que utilizamos es:

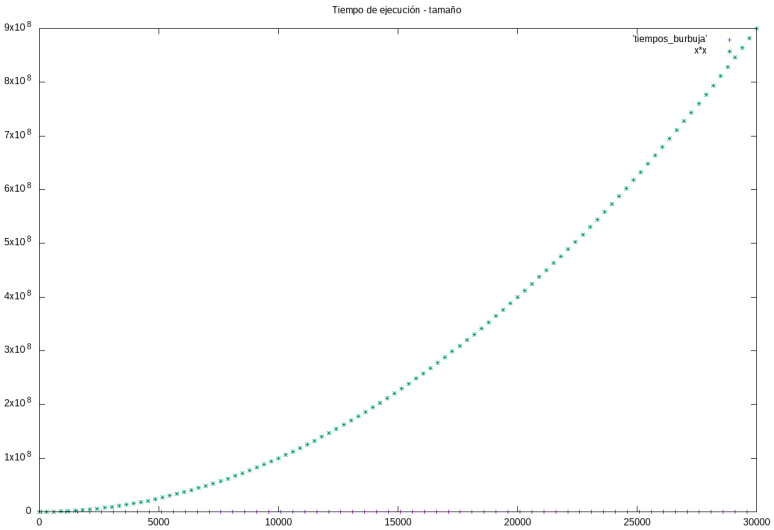
```
void ordenar(int *v, int n){
    for (int i=0; i<n-1; i++)
        for (int j=0; j<n-i-1; j++)
            if (v[j]>v[j+1]) {
                int aux = v[j];
                v[j] = v[j+1];
                v[j+1] = aux;
            }
 }
```

Tenemos dos for anidados. El orden del if es O(1). Dado que en el segundo for tenemos:
$\sum_{j=0}^{n-i-1} 1 = n - i$, aplicando la regla del producto con el primer for nos queda:
$\sum_{i=0}^{n-1} n - i = \sum_{i=0}^{n-1} n - \sum_{i=0}^{n-1} i = n \sum_{i=0}^{n-1} 1 - \sum_{i=0}^{n-1} i = n^2 - \frac{n^2}{2} = (2n^2 - n^2)/2 = \frac{n^2}{2} \in O(n^2)$
Es decir, que la eficiencia teórica del algoritmo es del orden de n^2.

El fichero ordenación.cpp y el script ejecuciones_ordenacion.csh se encuentran en este mismo directorio junto con los gráficos generados y un script más para generarlos de nuevo.

Si se dibujan superpuestas la función de eficiencia teórica y la empírica, estas quedan prácticamente superpuestas, lo que indica que la eficiencia teórica se ha calculado correctamente (ver grafico1 y grafico2).


Tiempo de ejecución - tamaño
'tiempos_burbuja'
1.2
1
0.8
0.6
0.4
0.2
0
0
5000
10000
15000
20000
25000
30000


Tiempo de ejecución - tamaño
'tiempos_burbuja'
x*x
9x10^8
8x10^8
7x10^8
6x10^8
5x10^8
4x10^8
3x10^8
2x10^8
1x10^8
0
0
5000
10000
15000
20000
25000
30000

Ejercicio2.

Vamos a calcular los parámetros a, b y c de $f(x) = an^2 + bn + c$ que serían los reales para la función que representa la eficiencia (teóricamente habíamos afirmado que era n^2).

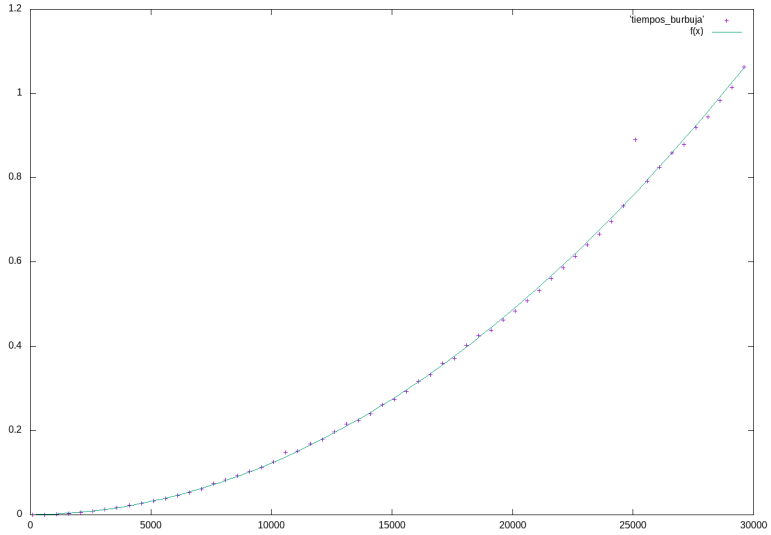
Con gnuplot escribo:

- $f(x) = a * x * *2 + b * x + c$
- fit $f(x)$ "tiempos_burbuja" via a,b,c

y obtengo:

$a = 1{,}20418e^{-9}$ $b = 2{,}19891e^{-7}$ $c = 0{,}000441629$

Volvemos a dibujar la función con este ajuste:



Ejercicio3.

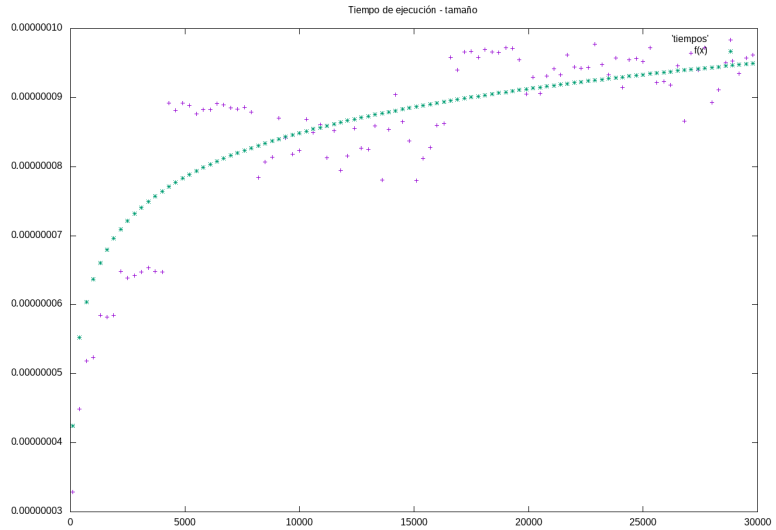
Dado el código de ejercicio_desc.cpp:

- ¿qué hace el algoritmo? El algoritmo es el método de busqueda binaria. El programa lee unos parámetros de entrada, genera un vector aleatorio y calcula el tiempo empleado en ordenar dicho vector con este algoritmo. Finalmente imprime en pantalla los resultados.

- ¿cuál es su eficiencia teórica? El mejor caso se presenta cuando el elemento buscado está en el centro de la lista. En este caso la complejidad es $O(1)$, dado que sólo se realiza una prueba de comparación de la igualdad. En el peor cas ola complejidad podría deducirse teniendo en cuenta que en cada iteración del bucle while se obtiene un número de elementos $\frac{n}{2^i}$ donde i es el número de iteraciones; de forma que $n/2^m = 1$ (m es el total de iteraciones) y, por tanto, $2^m = n$, $m = log_2 n$; por esa razón, la complejidad en el peor de los casos pertenece a $O(log_2 n)$, es del orden de $log_2 n$.

- ¿cuál es su eficiencia empírica? La calculamos como en los ejercicios anteriores y obtenemos el siguiente gráfico:

  Como se puede observar, los valores obtenidos son o 0, o minúsculos y la gráfica obtenida no aporta mucha información. Para solucionarlo, modifico el códico de ejercicio_desc.cpp para que ejecute la búsqueda un determinado número de veces y luego divida el tiempo entre dicho número y devuelva ese resultado. Finalmente, realizamos el ajuste con:
  gnuplot -e "set terminal png size 1300, 900 ;
  set title 'Tiempo de ejecución - tamaño' ;
  f(x) = a*log(x)/log(2) ; fit f(x) 'tiempos' via a ;
  set output 'grafico2.png' ;
  plot 'tiempos', f(x) with points pointtype 3"
  Obtenemos que a = 6.38588e-09
  La gráfica obtenida es:



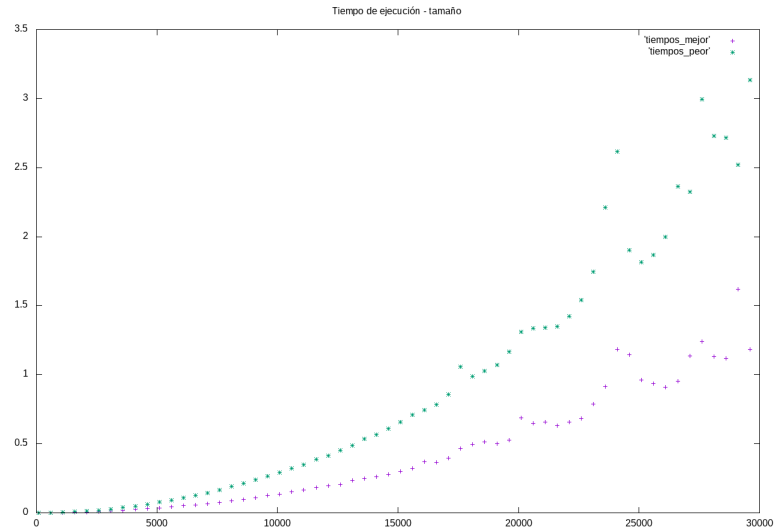
Ejercicio 4.

Simplemente he generado los métodos:

```
void Inicializar_mejor(){
   for(int i = 0 ; i < total_utilizados ; i++)
     v[i] = i ;
}

 void Inicializar_peor(){
   for(int i = 0 ; i < total_utilizados ; i++)
     v[i] = total_utilizados - i ;
}
```

Los resultados obtenidos quedan reflejados en el siguiente gráfico:



Como se puede ver, ocurre que en este caso tanto el mejor caso posible como el peor caso tiene orden de n^2, sin embargo, la eficiencia en los mejores casos es mucho mejor que en el peor de los casos, lo cual queda reflejado en la gráfica.

Ejercicio 5.

En este caso, la eficiencia teórica debería ser del orden de n, ya que solo se tienen que hacer las n primeras comparaciones.

Es más, dado que solo se hacen n comparaciones la diferencia de tiempos entre el mejor y el peor caso se hace tan grande que no se pueden apreciar corréctamente en una misma gráfica.

Una comprobación de que el mejor caso tiene una eficiencia del orden de n: