Estructuras de Datos

Práctica 1 Eficiencia

Juan Antonio Fernández Sánchez

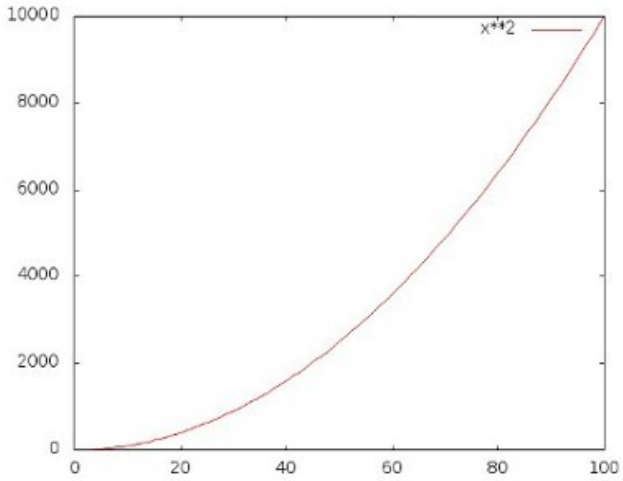
Ejercicio 1: Algoritmo de ordenación Burbuja

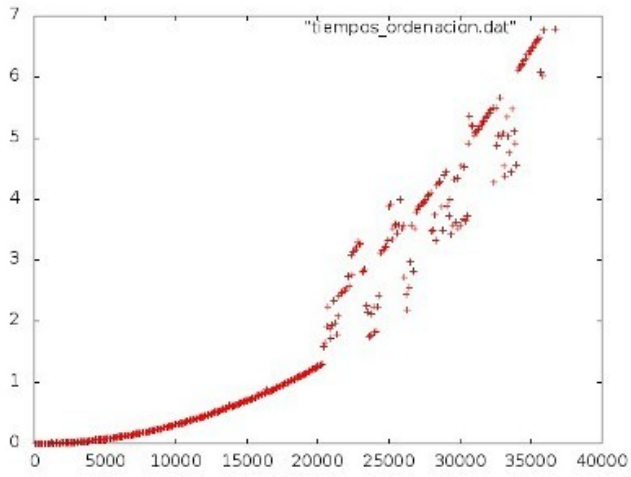
Desarrollo completo del cálculo de la eficiencia teórica y gráfica

Para calcular la eficiencia teórica realizamos el cálculo de las operaciones elementales que hay dentro del código principal del algoritmo de ordenación 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;
            }
}
```

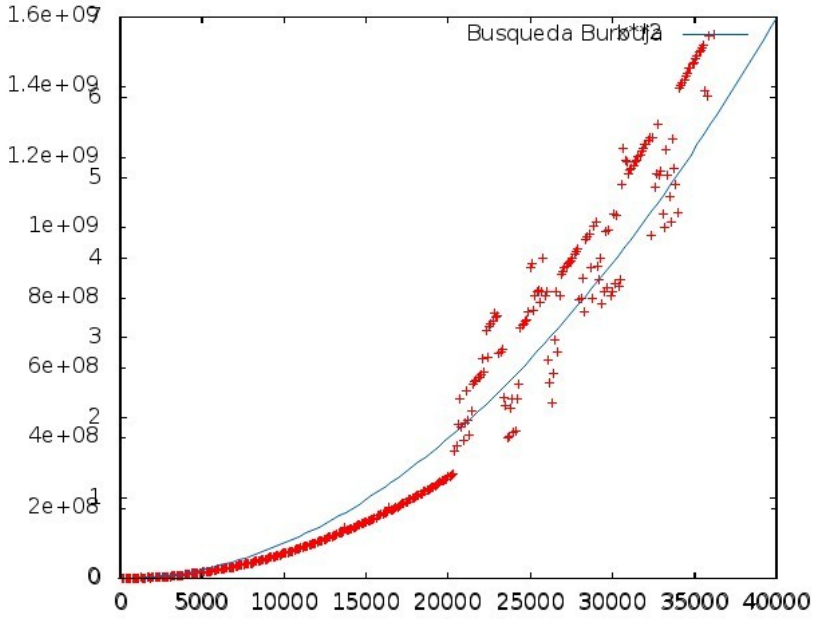
Haciendo el calculo de forma simplificada y quedándonos con el orden podemos decir que este algoritmo tiene una eficiencia de 0(n^2), cuadrática. En la gráfica de la izquierda podemos ver la curva de n cuadrado y a la derecha los tiempos que empíricamente se han obtenido.





Para el cálculo de los tiempos ejecutamos el script ejecuciones_ordenacion.csh que ejecuta a su vez de forma controlada el ejecutable a partir de Ejercicio1.cpp que previamente hemos compilado. Durante la ejecución del script los datos resultado del ejecutable se van escribiendo en tiempos_ordenacion.dat que después usamos con gnuplot para crear la gráfica. En concreto hemos usado ordenacionBurbuja.gp para que gnuplot nos cree la gráfica como nos interesa, superpuesta encima de la de n cuadrado.

Si superponemos ambas gráficas podemos comparar la eficiencia teórica que hemos calculado que tendrá nuestro algoritmo con el resultado empírico de la ejecución:



Como vemos la eficiencia teórica calculada se ajusta bastante a la empírica, pese que al principio los resultados empíricos nos muestras un crecimiento menor al de la curva cuadrada a partir de las 20000 unidades de vector el tiempo empieza a ser distinto pero siguiendo la misma tendencia de la curva, siendo en su mayoría los tiempos superiores a los de esta.

Detalles:
En linux para conocer el modelo de nuestra CPU además de otros muchos datos de los núcleos de la misma podemos ejecutar: cat /cpu/info .

Para conocer la versión del S.O. cat /etc/issue y para conocer la arquitectura (32 o 64 bits) uname -m donde veremos x86_64 para 64 bits ó i686 para las de 32.

Para conocer la versión de nuestro compilador podemos hacer g++ -v .

El script que nos dan de ejemplo está escrito en C-Shell (csh), cuya sintaxis es similar a C, estos ficheros no pueden ejecutarse bajo bash (el shell por defecto de Ubuntu) y por tanto tendremos que instalarlo si queremos ejecutarlo sudo apt-get install csh .

Para la creación de las gráficas con gnuplot usamos los script .gp asi: gnuplot ordenacionBurbuja.gp, que en este caso nos dejará la gráfica en jpeg donde se haya ejcutado gnuplot.

Ejercicio 2: Ajuste en la ordenación de la burbuja

En el ejercicio anterior vimos como obtener las gráficas tanto de la función que corresponde al orden del algoritmo como de los datos que el algoritmo generó.

Superponer estas dos funciones parece que es la única forma de ver como se ajusta el resultado empírico

al teórico pero gnuplot nos puede ayudar a esto de forma muy sencilla.

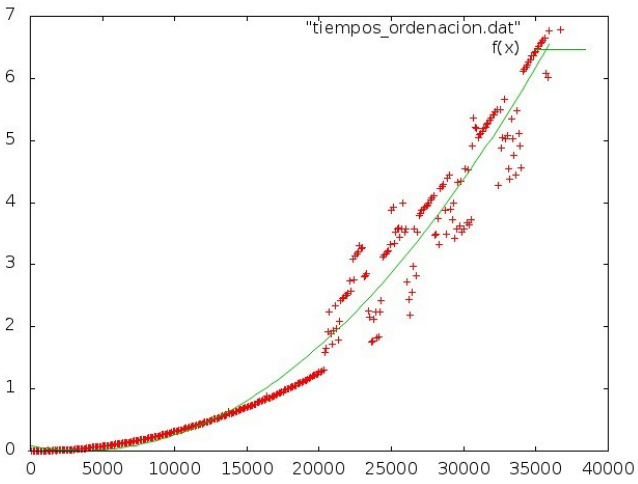
Lo único que tenemos que hacer es definir la función que corresponde al orden, en nuestro caso O(n²) y después asignare un fichero de datos para que realice el ajuste, así:

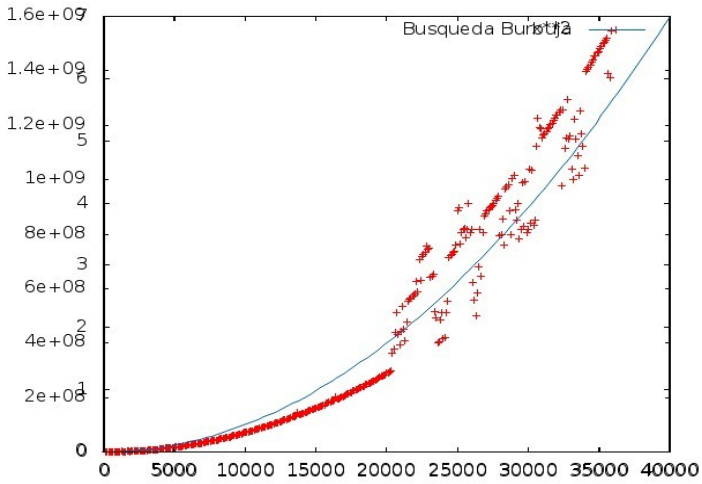
```
f(x)=a*x**2 + b*x + c
fit f(x) "tiempos_ordenacion.dat" via a, b, c, d
set terminal jpeg
set output "graficaAjuste.jpeg"
plot "tiempos_ordenacion.dat", f(x)
```

Después ejecutamos gnuplot consiguiendo la gráfica del ajuste:

```
gnuplot ajuste.gp
```

Que es mejor solución a la tediosa tarea de ajustar a mano ambas gráficas como hacíamos antes, aunque vemos que el resultado es muy similar:



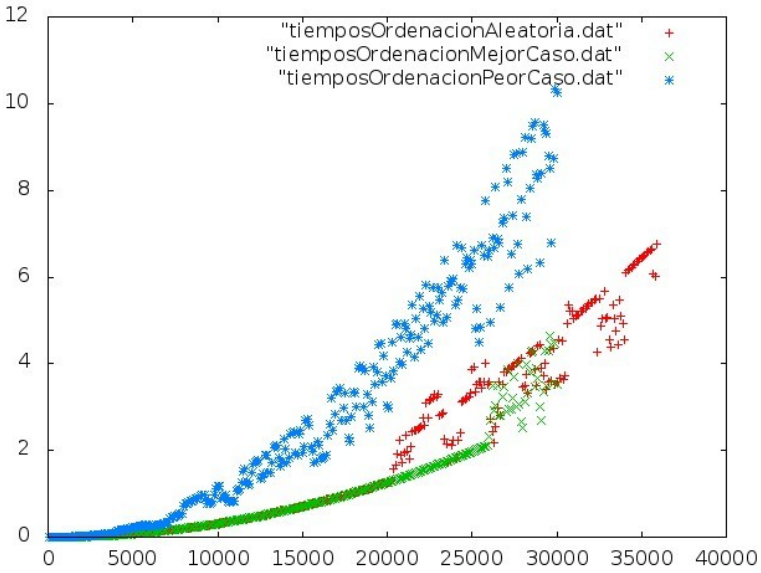


Ejercicio 4: Mejor y peor caso

Para calcular la eficiencia teórica realizamos el cálculo de las operaciones elementales que hay dentro del código principal del algoritmo de ordenación 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;
            }
}
```

Haciendo el calculo de forma simplificada y quedandonos con el orden podemos decir que este algoritmo tiene una eficiencia de 0(n^2), cuadrática.



Para el cálculo de los tiempos ejecutamos el script ejecucionComparacion.sh que realiza todo el proceso para ver la comparativa. Compila las dos versiones del programa, ejecuta a su vez de forma controlada los script ejecucionesOrdenacionMejorCaso.csh y ./ejecucionesOrdenacionPeorCaso.csh que a su vez ejecutan los programas en bucle de forma controlada enviando los datos a unos ficheros .dat que luego son leidos por gnuplot mediante otro script para crear una gráfica como la que vemos abajo.

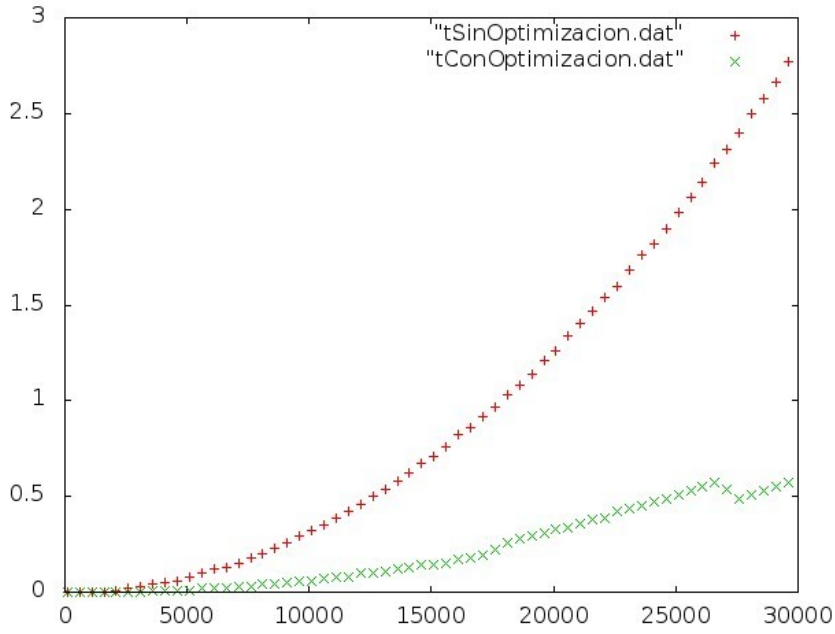
Todos los fuentes se encuentran disponibles aquí para sólo ser necesario bajarlos y ejecutar el .sh.

Si comparamos los resultados (mejor y peor caso) con los resultados obtenida en el caso medio para este algoritmo vemos que como esperábamos los resultados del caso aleatorio en la mayoría de los casos se alojan entre los peores y mejores.

Ejercicio 6: Influencia del proceso de compilación

Vamos a retomar el ejercicio donde tratamos el algoritmo de ordenación de la burbuja. Replicando el experimento esta vez compararemos los tiempos que se obtienen aplicando la optimización de mayor nivel del compilador, -O3.

Ejecutando el script que compila con y sin optimización y ejecutando ambas versiones del programa para vectores de tamaño de 100 a 30000 elemntos obtenemos la siguiente gráfica:



Vemos la enorme diferencia en el tiempo de ejecución incluso para los tamaños más grandes y la eficacia de la optimización que el propio compilador es capaz de hacer.

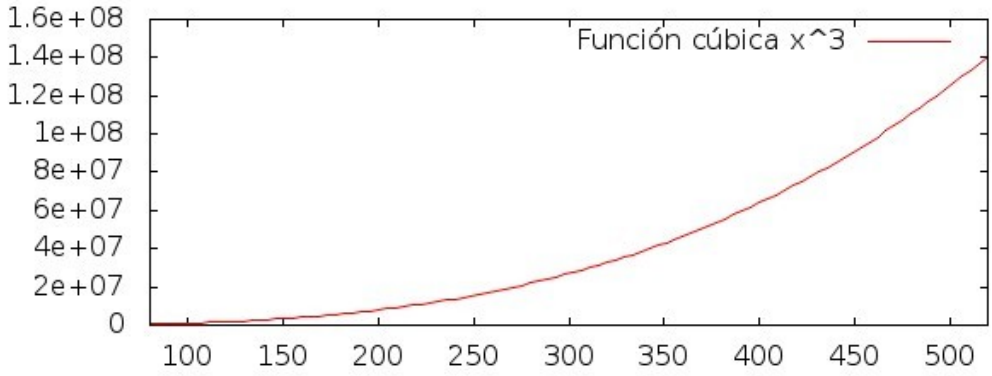
Ejercicio 7: Multiplicación matricial

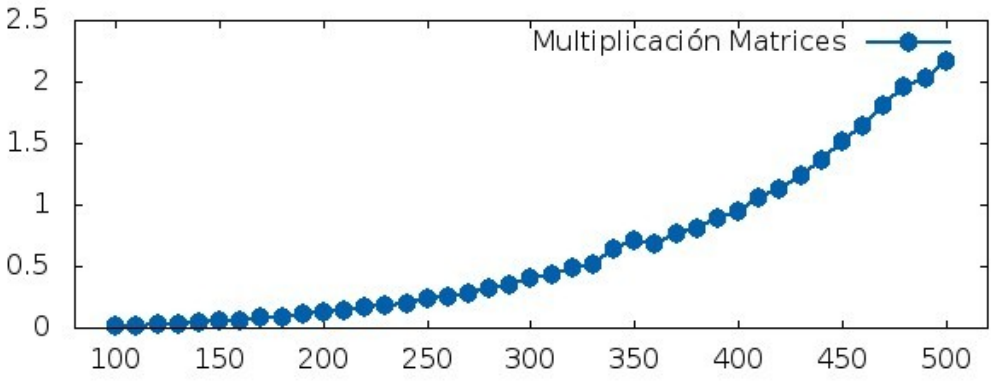
Para calcular la eficiencia teórica realizamos el cálculo de las operaciones elementales que hay dentro del código principal del sencillo algoritmo de multiplicación de matrices.

```
void multiplicaMatrices (int **matrizA, int **matrizB, int **matrizResultado, int tam){

for(int k=0; k<tam; k++)
  for(int i=0; i<tam; i++)
    for(int j=0; j<tam; j++)
      matrizResultado[k][i]+=matrizA[k][j]*matrizB[j][i];

}
```

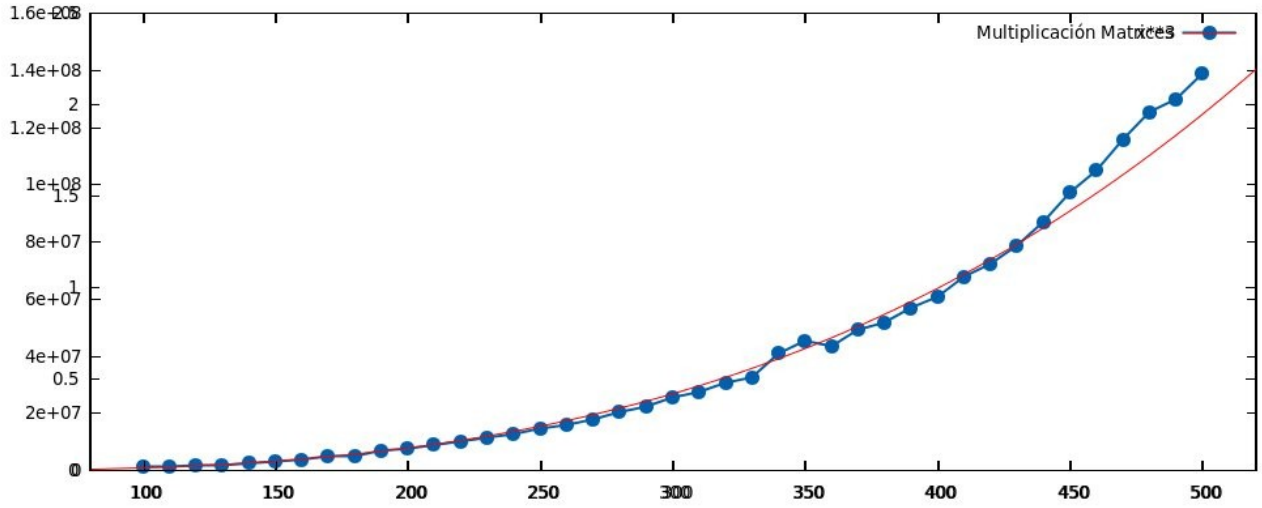
Haciendo el calculo de forma simplificada y quedandonos con el orden al que pertenece podemos decir que este algoritmo tiene una eficiencia de 0(n^3), cúbica.

Más abajo podemos ver la forma en que crece la función cúbica x^3 y justo debajo como ha crecido el tiempo de ejecución de nuestro algoritmo en función del tamaño de la matriz (para ello hemos montado un pequeño script que agiliza el trabajo).

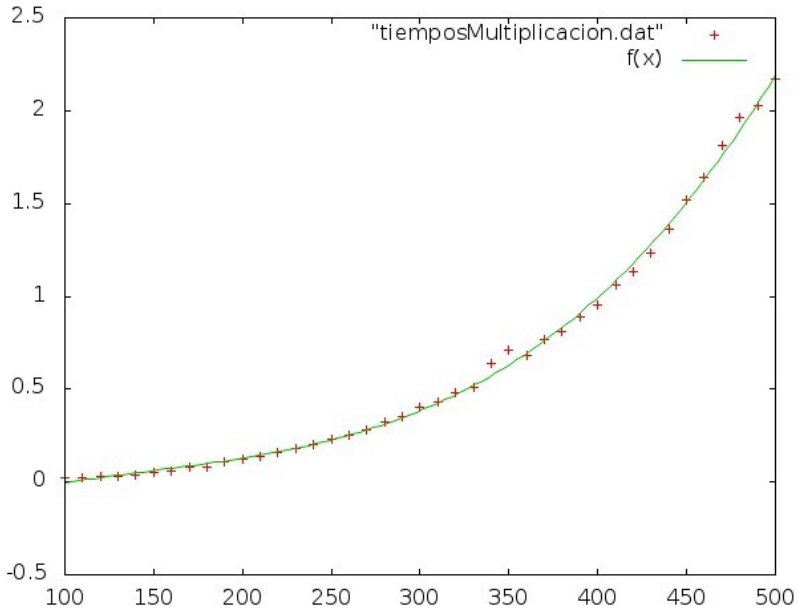




Ya se puede ver que los resultados empíricos se ajustan a los teóricos, es más, si superponemos ambas gráficas aunque las escalas en el eje Y no son las mismas podemos ver como realmente la eficiencia teórica que le calculábamos a este algoritmo responde fielmente a la empírica, al menos en su mayor parte.



Pero para hacer esto mejor y no superponiendo las gráficas a mano la mejor opción es *ajustar la función del orden de eficiencia del algoritmo O(n³) a los datos contenidos en el fichero .dat* así obtenemos la siguiente gráfica donde vemos realmente como se ajustan los datos empíricos a los teóricos.



La forma de realizar el ajuste con gnuplot puede verse en el fichero ajuste.gp y los datos de este ajuste se graban en un fichero de control fit.log

a. Detalles del equipo:

Hardware Usado
El equipo donde se ha realizado la compilación y ejecución del programa es un Portátil Acer Aspire 5920 con las siguientes características:

- Procesador de doble núcleo: Intel(R) Core(TM)2 Duo CPU T5450 a 1.66GHz
- 4GB de Ram

Sistema Operativo
Ubuntu 12.04.4 LTS 32 bits

Compilador utilizado y opciones de compilación
Se ha usado gcc versión 4.6.3 (Ubuntu/Linaro 4.6.3-1ubuntu5)

b. Publicación

Todo el contenido de esta práctica se encuentra publicado en Git Hub en el repositorio:

https://github.com/juanAFernandez/EstructurasDeDatos/