Práctica Eficiencia

Estructura de Datos

Realizado por: Daniel Díaz Pareja
Fecha: 10/10/2017
Universidad de Granada

Índice

Hardware, sistema operativo, compilador y opciones de compilación.

Hardware:

- Procesador Intel Core i7-4770K
- Velocidad de reloj de 3.5 Ghz
- 16 GB RAM

SO: Windows 8.1

Compilador: TDM-GCC 4.8.1 64-bit Debug

Opciones de compilación: Ninguna, se presume que se compila sin optimizar ya que se está usando el compilador Debug, que por defecto no utiliza optimización.

Ejercicio 1: Ordenación de la burbuja

1.1 Calcule la eficiencia teórica de este algoritmo.

Código del algoritmo:

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

Como en el peor caso los dos bucles for son de orden O(n), y están anidados, el algoritmo es de orden $O(n^2)$.

1.2 replique el experimento que se ha hecho antes (búsqueda lineal) con este nuevo código. Es decir, dibuje los datos con GNUPlot.

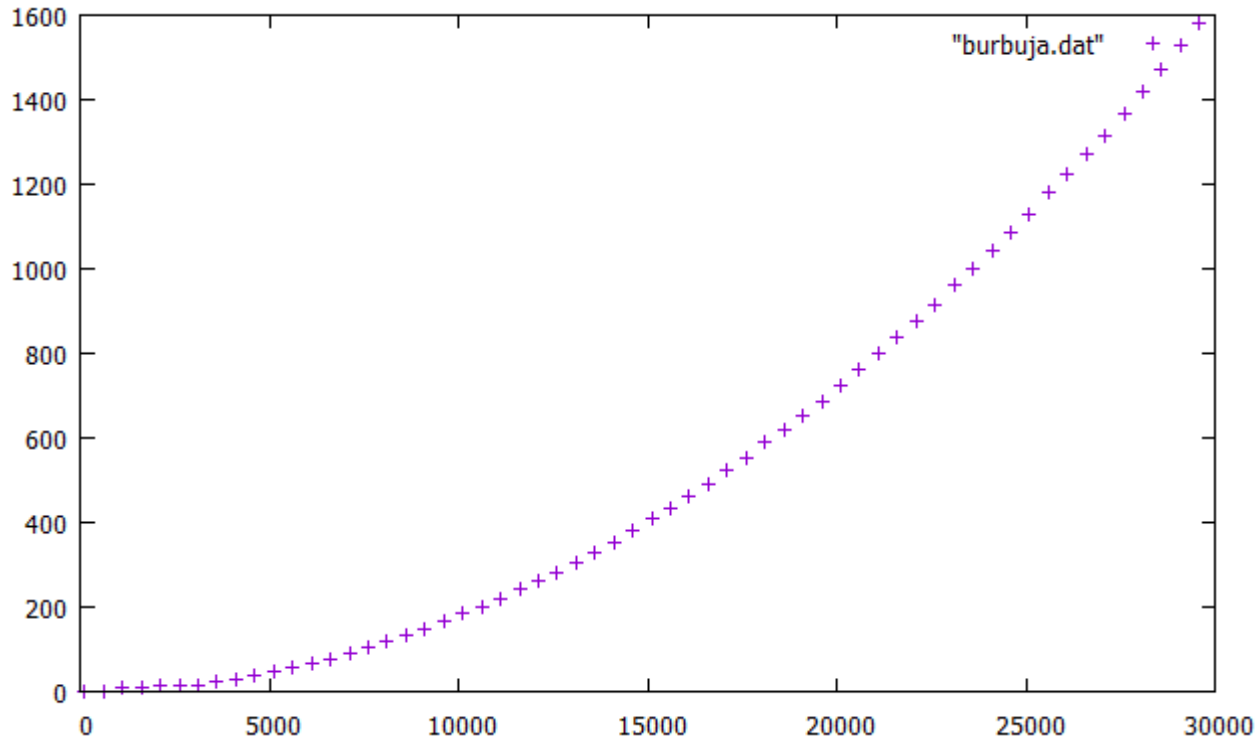
1.2.1 Código fuente:

```
#include <iostream>
```

```
#include <ctime>

using namespace std;

// Creamos un vector totalmente desordenado (peor caso)
void crear_vector(int *v, int n_datos){

  for (int i=0, j=n_datos-1; i<n_datos; i++,j--)
     v[i]=j;
}

// _Ordenación Burbuja. O(n²) peor caso
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;
            }
}

int main(void){
  clock_t ti, tf;

  for(int n_datos=100; n_datos<30000; n_datos+= 500){

    int v[n_datos];
    crear_vector(v, n_datos);

    ti = clock();
    ordenar(v,n_datos);
    tf = clock();
    cout << n_datos << "\t" << tf - ti << endl;

  }
}
```

1.2.2 Representación gráfica de los datos.

Eje x: tamaño del vector
Eje y: tiempo



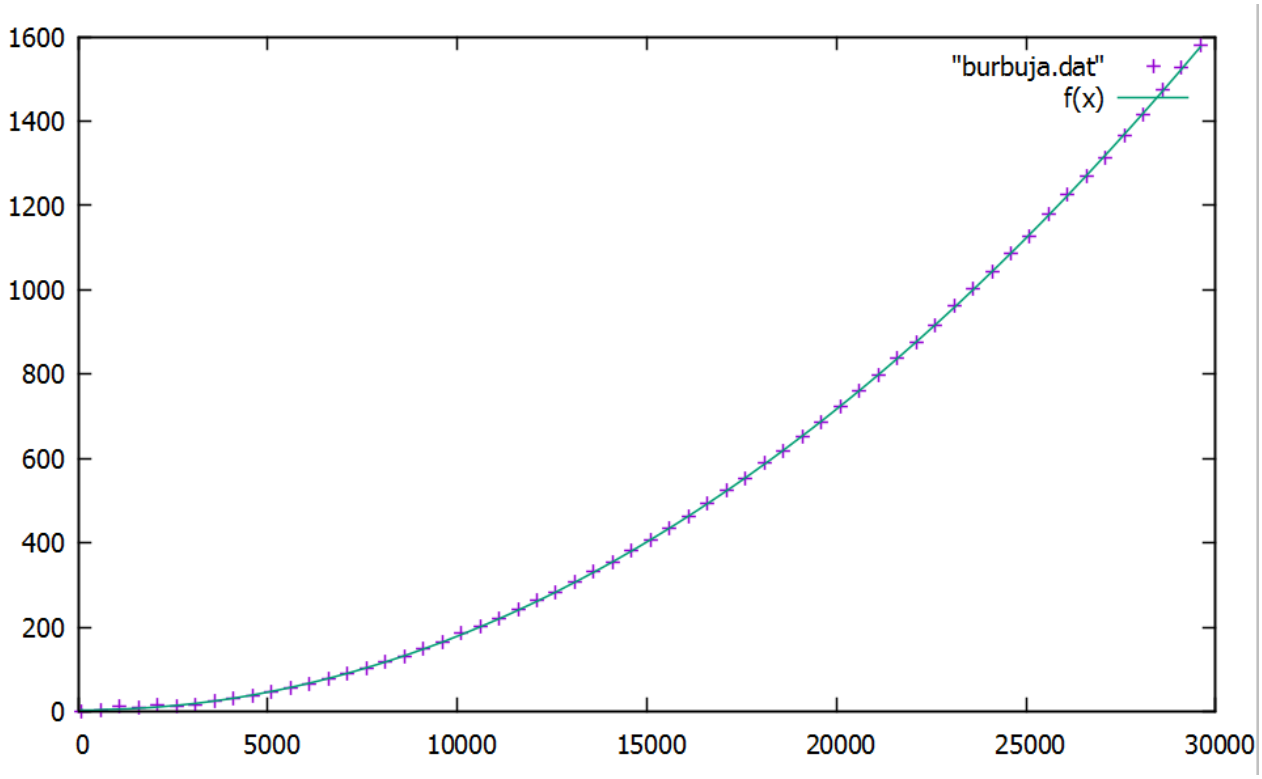
Ejercicio 2: Ajuste en la ordenación de la burbuja.

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 ax2+bx+c.

Resultados:

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                = 1.82031e-006    +/- 4.509e-009   (0.2477%)
b                = -0.000779296    +/- 0.0001384    (17.76%)
c                = 4.41734         +/- 0.8892       (20.13%)

correlation matrix of the fit parameters:
                a      b      c
a               1.000
b              -0.968  1.000
c               0.738 -0.861  1.000
```

A continuación mostramos el dibujo de la curva obtenida y los resultados obtenidos. Vemos que el ajuste se cumple casi a la perfección:



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:

3.1. Explique que hace este algoritmo.

Este algoritmo es lo que haría la búsqueda binaria, es decir, coge el elemento de la mitad del vector y comprueba si es el valor buscado. Si lo es, termina. Si no, vuelve a hacer este proceso pero con la mitad del vector donde se encuentre el valor buscado. Esto se sabe porque si el elemento medio es menor que el buscado, estará en la mitad inferior. Ocurrirá lo contrario si el elemento medio es mayor que el buscado, que estará en la mitad superior.

Para que este algoritmo funcione hay que establecer que el vector utilizado este **ordenado** y en el código fuente se están generando vectores aleatorios, sin orden aparente. Además, los elementos de dicho vector se están generando con la operación “rand() % tam”, por lo que siempre saldrá un número < tam, ya que al dividir cualquier número entre otro nunca te puede dar como resto el divisor.

Dicho esto, en el código fuente el valor a buscar es el propio “tam”, por lo que, además de que el vector no esta ordenado y esto hace improbable el correcto funcionamiento de la

búsqueda binaria, nunca va a poder encontrarse el valor “tam” porque nunca va a existir en el vector tal y como se está creando en el código.

3.2. Calcule su eficiencia teórica.

La eficiencia teórica es O(log n) en el peor de los casos, ya que siempre se está dividiendo el vector por la mitad, y una de las siguientes mitades a la mitad, por lo que el número máximo de iteraciones del bucle while será de orden logarítmico en base al tamaño del vector.

3.3. Calcule su eficiencia empírica.

Tal y como está el código, no puedo calcular su eficiencia empírica. El valor obtenido para tamaños grandísimos de n es siempre de 0, ya que el orden logarítmico crece muy lentamente. Llega un punto en el que el programa se queda sin memoria para almacenar un vector de un tamaño de n tan grande antes de que salga algún tiempo distinto de 0.

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.

Como ya se ha comentado anteriormente, la eficiencia empírica siempre da tiempos de 0. Esto es porque el orden logarítmico crece muy lentamente, y además porque al hacer:

```
cout << tam << "\t" << ((tfin-tini)/num_ejecuciones)/(double)CLOCKS_PER_SEC << endl;
```

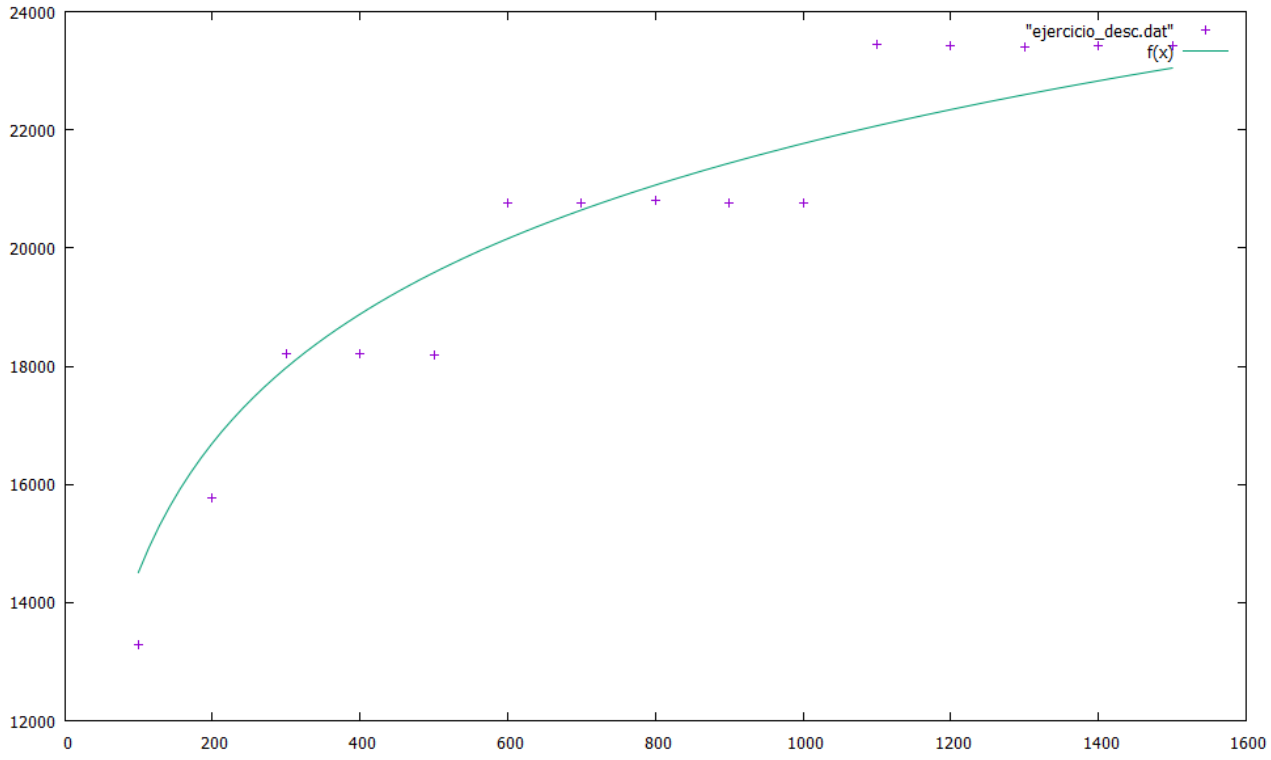
estamos dividiendo tfin-tini (que ya de por sí es muy pequeño) entre la constante CLOCKS_PER_SEC del sistema, el número de ciclos de reloj por segundo del procesador, que es un número muy grande. Por lo tanto, no tenemos tanta precisión y el valor siempre se aproxima a 0.

Para arreglarlo he intentado hacer varias ejecuciones del algoritmo y dividir el tiempo total entre el número de ejecuciones pero, de nuevo, salían tiempos de 0 para tamaños muy grandes (se consume más tiempo y memoria en crear un vector tan grande que en ejecutar el algoritmo miles de veces...).

Así que al final he optado por ejecutar el algoritmo 500 millones de veces y medir el tiempo del millón de ejecuciones, en ciclos de reloj (ya que en milisegundos se pierde demasiada precisión), para tamaños distintos del vector, empezando desde 100, y sumando 100 en cada iteración hasta 15 veces. Estos sería el trozo de código de esta parte:

```
clock_t tini;     // Anotamos el tiempo de inicio
int num_ejecuciones = 5000000000;
tini=clock();

// Algoritmo a evaluar
for (int i = 0; i < num_ejecuciones; i++)
      operacion(v,tam,tam+1,0,tam-1);

clock_t tfin;     // Anotamos el tiempo de finalizaciÃ³n
tfin=clock();

// Mostramos resultados
cout << tam << "\t" << tfin-tini<< endl;
```

Los resultados obtenidos son los siguientes:



Como vemos, el ajuste es bastante malo, pero he estado probando con distintos valores del nº de ejecuciones del algoritmo y con los tamaños del vector y no consigo que me salga nada muy parecido a una función logarítmica en los resultados. Simplemente los tiempos de ejecución son tan rápidos que no hay apenas diferencia entre cambiar el tamaño del vector o el número de ejecuciones del algoritmo.

Ejercicio 4: Mejor y peor caso.

Mejor caso, código para crear el vector:

```
void crear_vector(int *v, int n_datos){
        for (int i=0; i<n_datos; i++)
                v[i]=i;
}
```

Peor caso, código para crear el vector:

```
void crear_vector(int *v, int n_datos){
  for (int i=0, j=n_datos-1; i<n_datos; i++,j--)
          v[i]=j;
}
```

Resultados del ajuste:

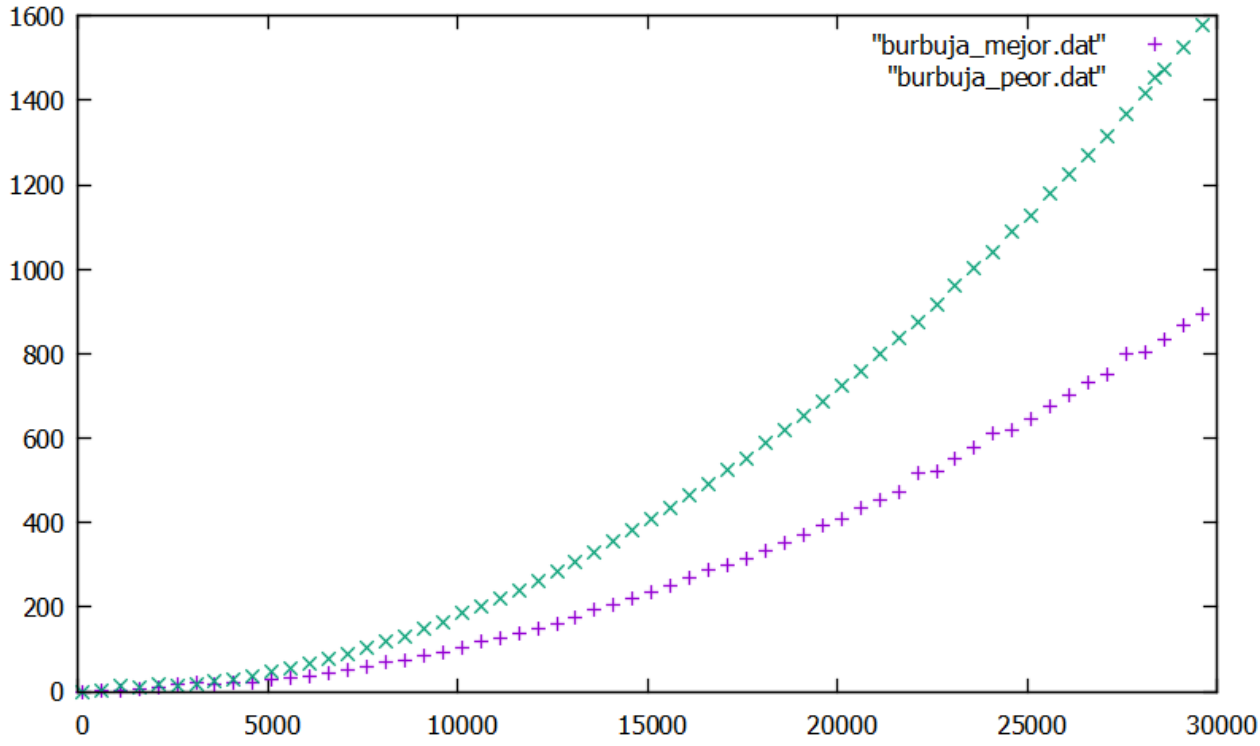
- Mejor caso:

Final set of parameters Asymptotic Standard Error
========== ==========
a = 1.82031e-006 +/- 4.509e-009 (0.2477%)
b = -0.000779296 +/- 0.0001384 (17.76%)
c = 4.41734 +/- 0.8892 (20.13%)

- Peor caso:

Final set of parameters Asymptotic Standard Error
========== ==========
a = 1.03511e-006 +/- 9.173e-009 (0.8862%)
b = -0.000330288 +/- 0.0002815 (85.24%)
c = 3.49613 +/- 1.809 (51.74%)

Comparación de resultados:

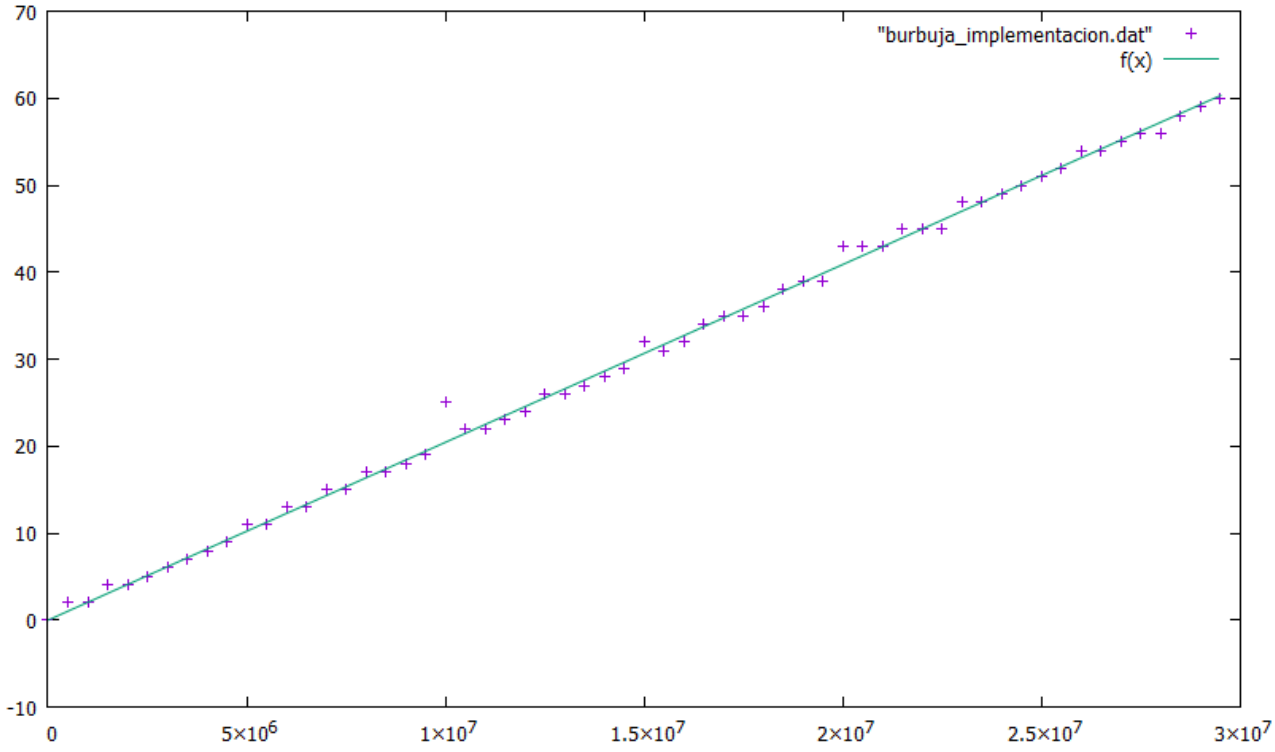


Ejercicio 5: Dependencia de la implementación

La eficiencia teórica de este nuevo código en el mejor caso es de O(n), ya que si el vector está ordenado, en la primera iteración del bucle interno no se entrará al "if" y la variable cambio permanecerá en "false", por lo que no se seguirá iterando por el resto del vector.

```
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;
			}
	}
}
```

Eficiencia empírica:



Valores para f(x) = a*x + b:

Final set of parameters		Asymptotic Standard Error	
a	= 2.04432e-006	+/- 1.307e-008	(0.6393%)
b	= -0.000731118	+/- 0.2235	(3.057e+004%)