PRACTICA 1: Eficiencia

Estructura de datos

Francisco José González García Grupo: A2

EJERCICIO 1

Código fuente: ordenacion.cpp

```
#include <iostream>
#include <ctime>   // Recursos para medir tiempos
#include <cstdlib> // Para generación de números pseudoaleatorios
using namespace std;


int buscar(const int *v, int n, int x)
{
   int i = 0;
   while (i < n && v[i] != x)
      i = i + 1;
   if (i < n)
      return i;
   else
      return -1;
}


void burbuja(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;
         }
}

void sintaxis()
{
   cerr << "Sintaxis:" << endl;
   cerr << " TAM: Tamaño del vector (>0)" << endl;
   cerr << " VMAX: Valor máximo (>0)" << endl;
   cerr << "Genera un vector de TAM números aleatorios en [0,VMAX[" << endl;
   exit(EXIT_FAILURE);
}
int main(int argc, char *argv[])
{
   if (argc != 3) // Lectura de parámetros
      sintaxis();
   int tam = atoi(argv[1]);  // Tamaño del vector
```

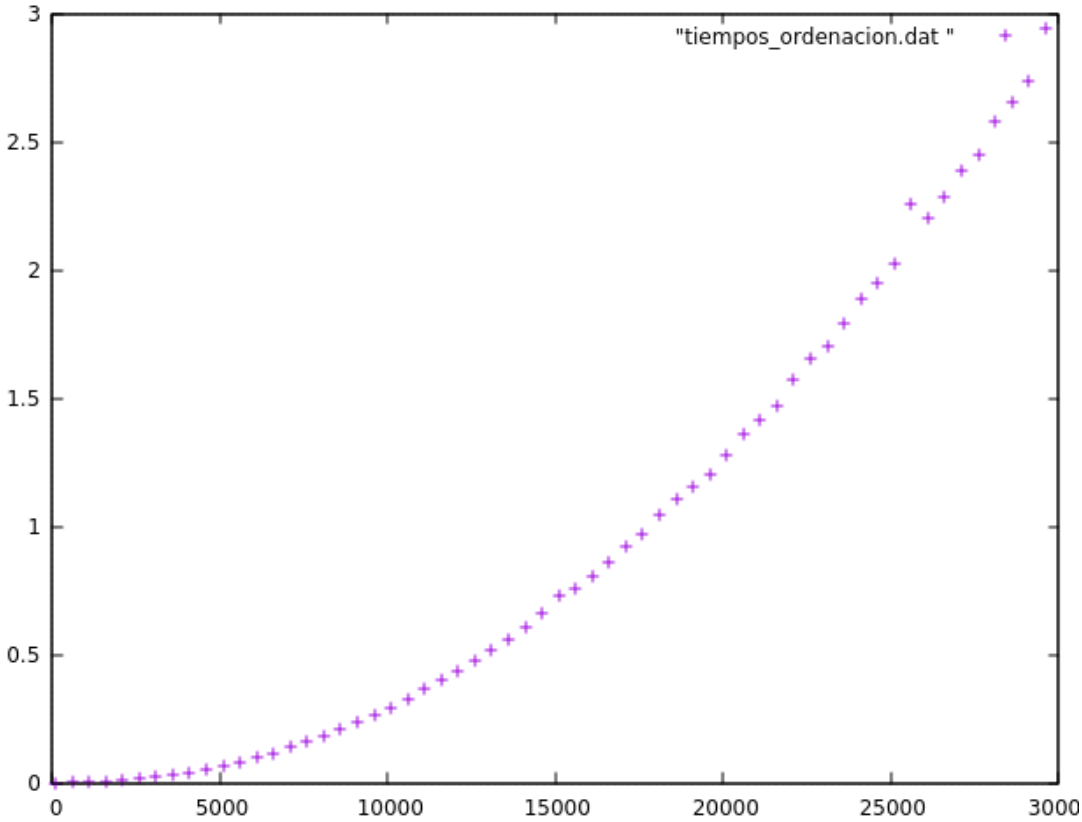
```
   int vmax = atoi(argv[2]); // Valor máximo
   if (tam <= 0 || vmax <= 0)
      sintaxis();

   // Generación del vector aleatorio

   int *v = new int[tam];       // Reserva de memoria
   srand(time(0));              // Inicialización generador números pseudoaleatorios
   for (int i = 0; i < tam; i++) // Recorrer vector
      v[i] = rand() % vmax;     // Generar aleatorio [0,vmax[

   clock_t tini; // Anotamos el tiempo de inicio
   tini = clock();
   burbuja(v, tam);
   clock_t tfin; // Anotamos el tiempo de finalización
   tfin = clock();

   // Mostramos resultados (Tamaño del vector y tiempo de ejecución en seg.)
   cout << tam << "\t" << (tfin - tini) / (double)CLOCKS_PER_SEC << endl;
   delete[] v; // Liberamos memoria dinámica
}
```

Script: ejecuciones_ordenacion.csh

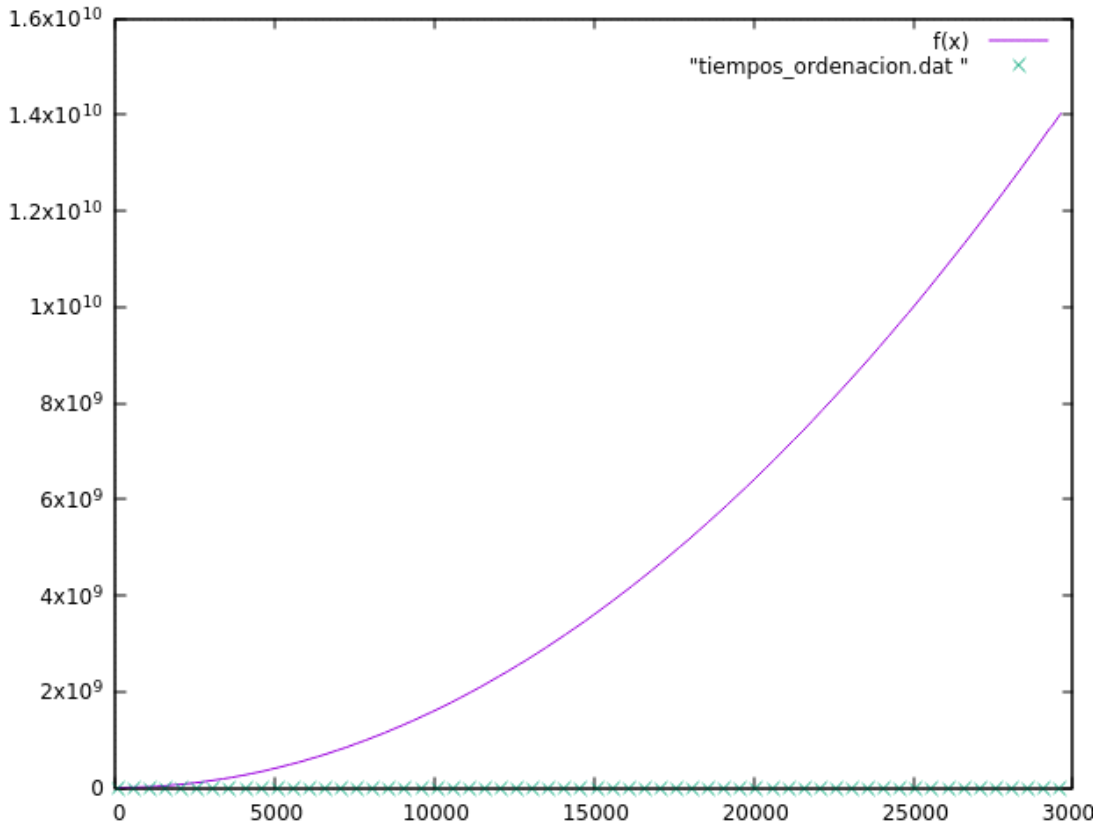
```
#!/bin/csh
```

```
@ inicio = 100

@ fin = 30000

@ incremento = 500

set ejecutable = ordp

set salida = tiempos_ordenacionp.dat


@ i = $inicio

echo > $salida

while ( $i <= $fin )

  echo Ejecución tam = $i

  echo `./{$ejecutable} $i 10000` >> $salida

  @ i += $incremento

end
```

La eficiencia teórica calculada da como resultado $16*n^2$, por lo que la ordenación se ejecuta en tiempo $O(n^2)$.



Al dibujar superpuestas la eficiencia teórica y práctica podemos observar que la gráfica correspondiente a la eficiencia empírica no se puede apreciar. Esto se debe a que los tiempos usados en cada función no tienen la misma medida.



EJERCICIO 2

Ajuste de regresión para el algoritmo de ordenación burbuja:

FIT: data read from "tiempos_ordenacion.dat "

format = z

#datapoints = 60

residuals are weighted equally (unit weight)

function used for fitting: f(x)

f(x)=a*x*x+b*x+c

fitted parameters initialized with current variable values

iter chisq delta/lim lambda a b c

0 9.4779953704e+18 0.00e+00 2.29e+08 1.000000e+00 1.000000e+00 1.000000e+00

12 4.0000637814e-01 -9.49e-08 2.29e-04 3.926555e-09 -1.558116e-05 4.055974e-02

After 12 iterations the fit converged.

final sum of squares of residuals : 0.400006

rel. change during last iteration : -9.48948e-13

degrees of freedom (FIT_NDF) : 57

rms of residuals (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0837714

variance of residuals (reduced chisquare) = WSSR/ndf : 0.00701766

Final set of parameters Asymptotic Standard Error

========== ==========

a = 3.92656e-09 +/- 1.613e-10 (4.109%)

b = -1.55812e-05 +/- 4.952e-06 (31.78%)

c = 0.0405597 +/- 0.03182 (78.44%)

correlation matrix of the fit parameters:

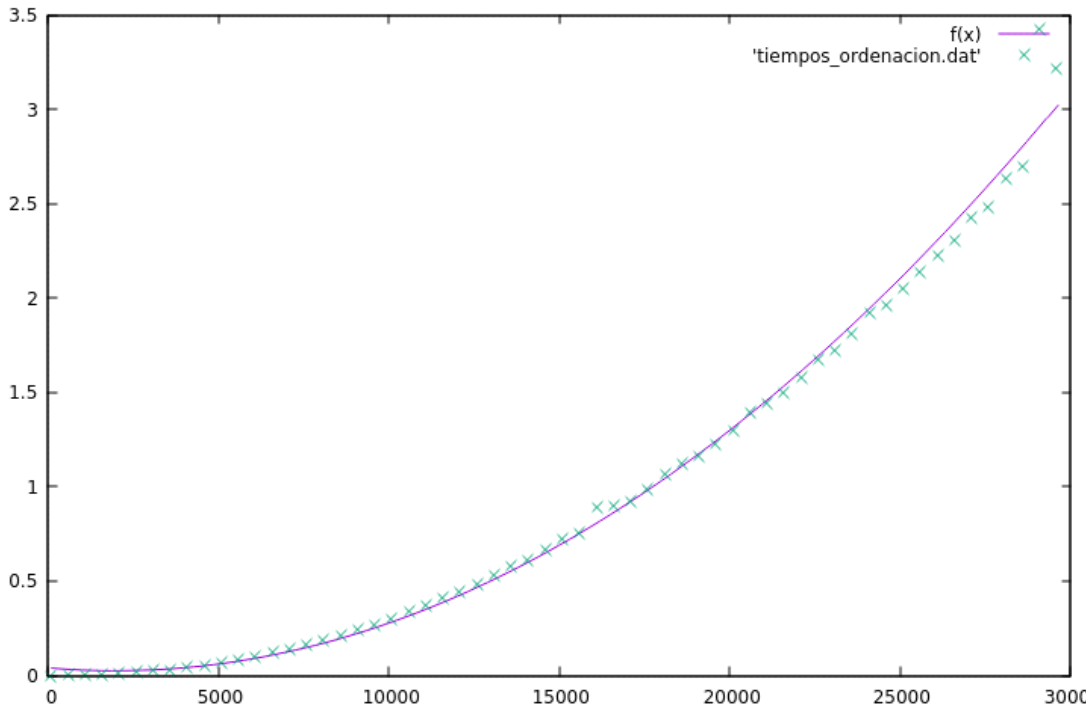
a b c

a 1.000

b -0.968 1.000

c 0.738 -0.861 1.000

La gráfica asociada es:



EJERCICIO 3:

Código fuente: ejercicio_desc.cpp

```
#include <iostream>
#include <ctime>   // Recursos para medir tiempos
#include <cstdlib> // Para generación de números pseudoaleatorios

using namespace std;
void burbuja(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 operacion(int *v, int n, int x, int inf, int sup)
{
  int med;
  bool enc = false;
  while ((inf < sup) && (!enc))
  {
    med = (inf + sup) / 2;
    if (v[med] == x)
      enc = true;
    else if (v[med] < x)
      inf = med + 1;
    else
      sup = med - 1;
  }
  if (enc)
    return med;
  else
    return -1;
}

void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "Se genera un vector de tamaño TAM con elementos aleatorios" << endl;
  exit(EXIT_FAILURE);
```

```
}

int main(int argc, char *argv[])
{
  // Lectura de parámetros
  if (argc != 2)
    sintaxis();
  int tam = atoi(argv[1]); // Tamaño del vector
  if (tam <= 0)
    sintaxis();

  // Generación del vector aleatorio
  int *v = new int[tam];       // Reserva de memoria
  srand(time(0));              // Inicialización del generador de números pseudoaleatorios
  for (int i = 0; i < tam; i++) // Recorrer vector
    v[i] = i;

  clock_t tini; // Anotamos el tiempo de inicio
  tini = clock();

  // Algoritmo a evaluar
  for (int i = 0; i < 10000000 ; i++)
    operacion(v, tam, tam + 1, 0, tam - 1);

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

  // Mostramos resultados
```

```
  cout << tam << "\t" << (tfin - tini) / 10000000 / (double)CLOCKS_PER_SEC << endl;


  delete[] v; // Liberamos memoria dinámica
}
```

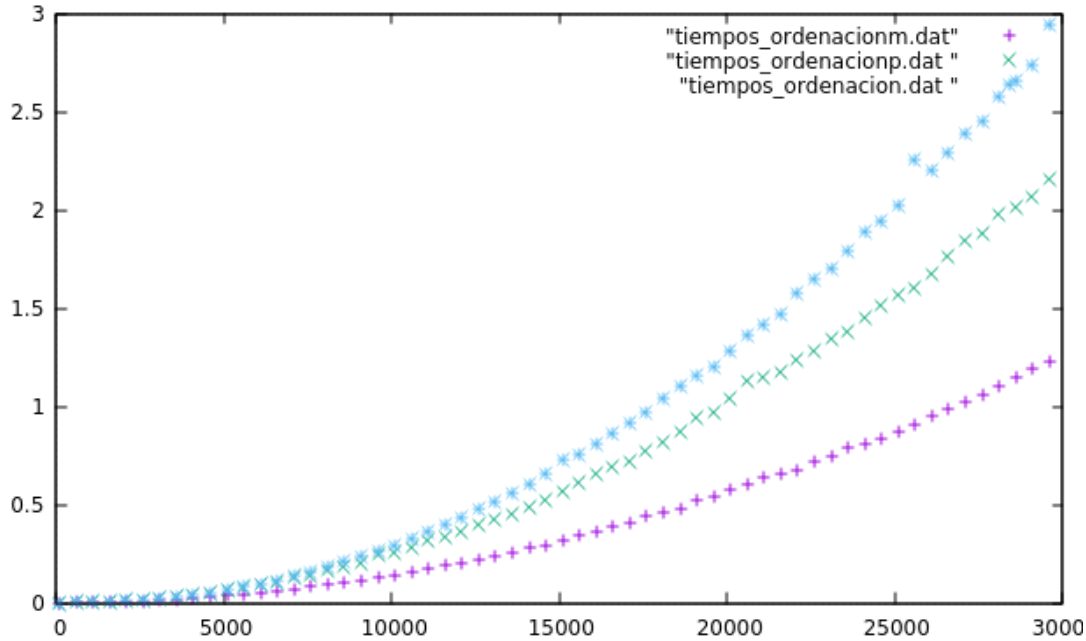
//

Este algoritmo se trata de la búsqueda binaria, cuyo funcionamiento es el siguiente: Compara el valor con el elemento en el medio del array, si no son iguales, la mitad en la cual el valor no puede estar es eliminada y la búsqueda continúa en la mitad restante hasta que el valor se encuentre. Para poder hacer uso de este algoritmo es necesario que el array esté ordenado.

La eficiencia de este algoritmo es de orden logarítmico O(logn), para calcular la eficiencia empírica es necesario modificar el código para que realice el algoritmo en repetidas ocasiones para que el cronómetro que incorpora ctime sea capaz de captar el tiempo, ya que no es capaz de registrar tiempos excesivamente bajos.

EJERCICIO 4:

<u>Comparación en 3 casos de la ordenación burbuja:</u>



Como podemos apreciar cuando el vector está ordenado del revés tarda más tiempo en ejecutarse que cuando se encuentra con el vector ordenado, ya que en este caso no tiene

que realizar ningún intercambio.

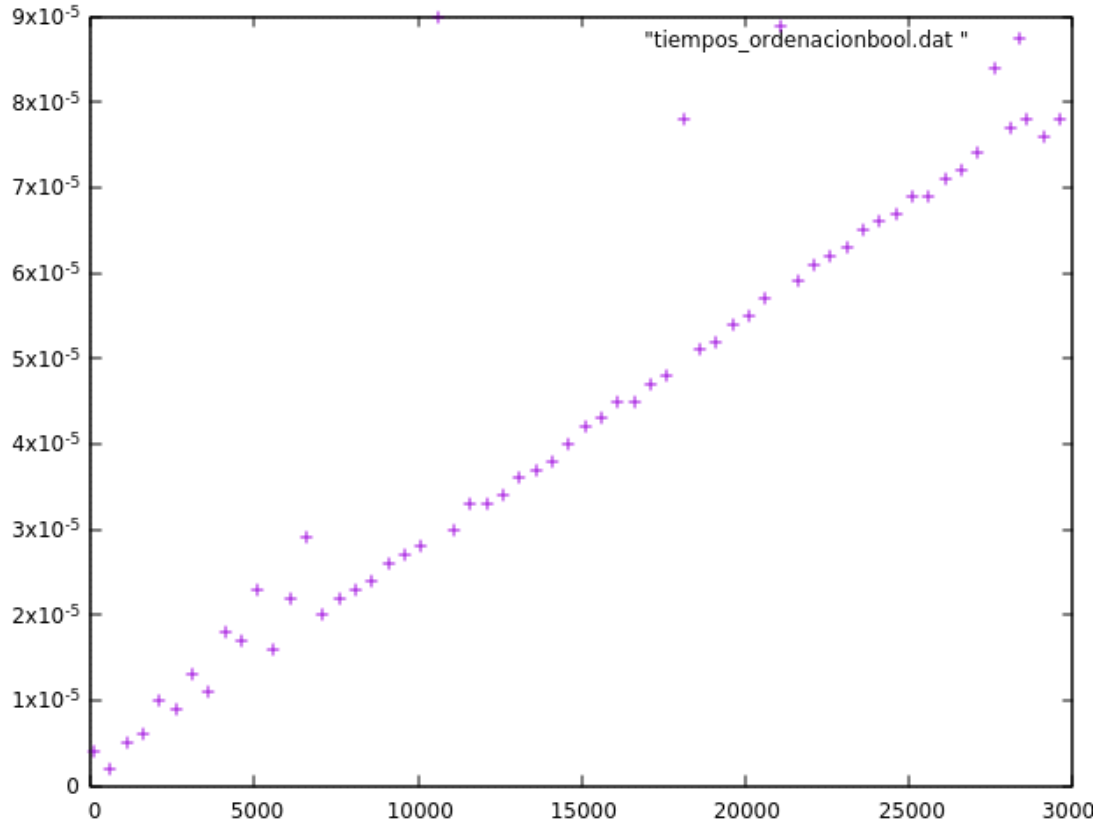
EJERCICIO 5

Algoritmo:

```
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;
            }
        }
 }
```

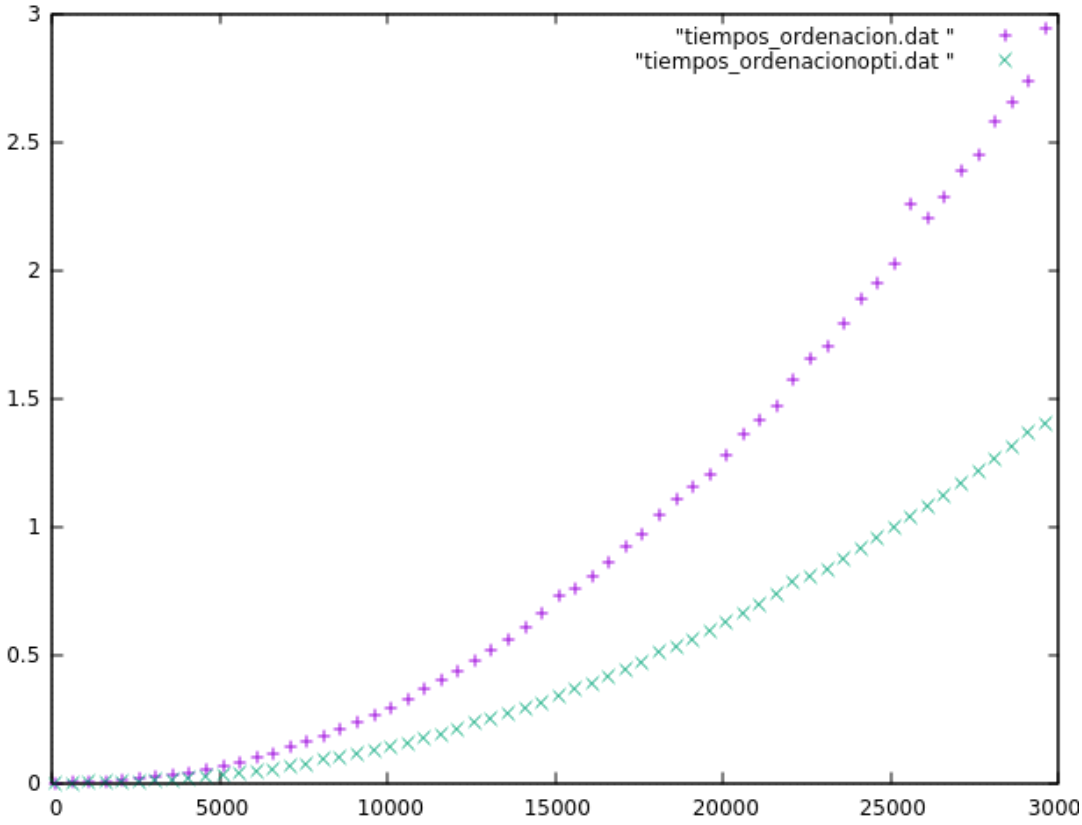
En el caso planteado en el enunciado en el que el vector que reciba el algoritmo ya esté ordenado el tiempo de ejecución sería de O(n)

Gráfica:



EJERCICIO 6:

Como podemos apreciar en la gráfica al aplicar el factor de optimización -O3 la velocidad de ejecución aumenta en varios puntos.



ESPECIFICACIONES HARDWARE

Intel Core i7-5500U @ 2.40GHz 2.40GHz

16 GB de memoria RAM

SISTEMA OPERATIVO

Ubuntu 16.04 LTS

COMPILADOR

G++ de gnu con las opciones predeterminadas a excepción del ejercicio 6 que se ha usado el modificador de optimización -O3