EJERCICIO 4

Sistema operativo: Ubuntu 18.04.1 LTS
Compilador usado: ii g++ 4:7.3.0-3ubuntu2 amd64 GNU C++ compiler

a) El mejor caso posible. Para este algoritmo, si la entrada es un vector que ya está ordenado el tiempo de cómputo es menor ya que no tiene que intercambiar ningún elemento.

He ordenado antes el vector y luego he calculado la eficiencia del vector ya ordenado. El código cpp es el siguiente:

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

using namespace std;

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;
      }
    }
}

void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "  VMAX: Valor máximo (>0)" << endl;
  cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[])
{
  // Lectura de parámetros
  if (argc!=3)
    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 del generador de números pseudoaleatorios
  for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = rand() % vmax;    // Generar aleatorio [0,vmax[
```

```
  ordenar(v,tam); // de esta forma forzamos el peor caso

  //Vamos a calcular el tiempo cuando ya está ordenado (mejor caso)
  clock_t tini;    // Anotamos el tiempo de inicio
  tini=clock();

  ordenar(v,tam); // de esta forma forzamos el peor caso

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

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

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

Luego he creado el fichero “ejecuciones_ordenacion_ej4_mejor.csh”:

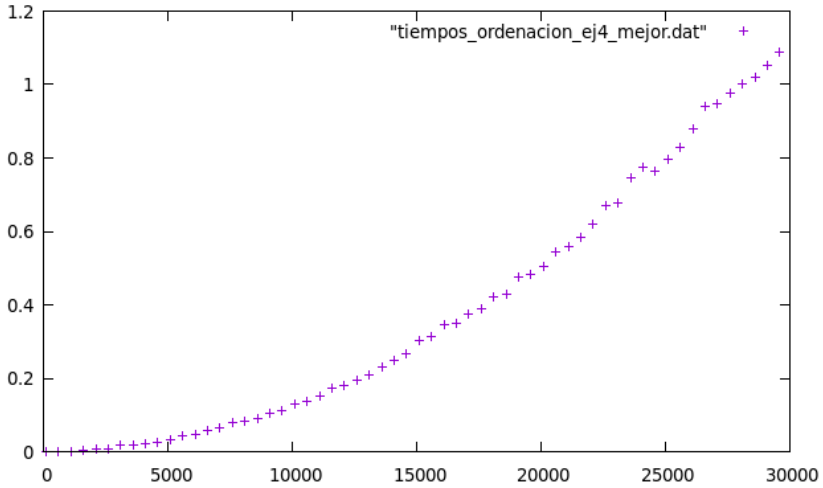
```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500
set ejecutable = ordenacion_ej4_mejor
set salida = tiempos_ordenacion_ej4_mejor.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i 10000` >> $salida
  @ i += $incremento
end
```

Y he obtenido los datos:

```
100 1.9e-05
600 0.000538
1100 0.001885
1600 0.003432
2100 0.00572
2600 0.008381
3100 0.016846
3600 0.016746
4100 0.020873
4600 0.026921
5100 0.032013
5600 0.043431
6100 0.048431
```

6600 0.056151
7100 0.064861
7600 0.080199
8100 0.081738
8600 0.090708
9100 0.104213
9600 0.113388
10100 0.130567
10600 0.137953
11100 0.151473
11600 0.173025
12100 0.18132
12600 0.196335
13100 0.211121
13600 0.230735
14100 0.247673
14600 0.266359
15100 0.303645
15600 0.313982
16100 0.345223
16600 0.350945
17100 0.37339
17600 0.388951
18100 0.423201
18600 0.428456
19100 0.474992
19600 0.481705
20100 0.503167
20600 0.545177
21100 0.560211
21600 0.584537
22100 0.620377
22600 0.668849
23100 0.678285
23600 0.747315
24100 0.774825
24600 0.764508
25100 0.795666
25600 0.828243
26100 0.879474
26600 0.939668
27100 0.948861
27600 0.975969
28100 1.00101
28600 1.02118
29100 1.05126
29600 1.08927

Y la gráfica:



b) El peor caso posible. Si la entrada es un vector ordenado en orden inverso estaremos en la peor situación posible ya que en cada iteración del bucle interno hay que hacer un intercambio.

Para este caso, he creado otro fichero en el que he ordenado el vector primero, y luego le he introducido a otro vector los datos desde le final hacia el principio, para que esté ordenado en el orden inverso. Finalmente he calculado el tiempo que tarda en ordenarlo. El código es:

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

using namespace std;

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;
      }
    }
}

void sintaxis()
{
  cerr << "Sintaxis:" << endl;
  cerr << "  TAM: Tamaño del vector (>0)" << endl;
  cerr << "  VMAX: Valor máximo (>0)" << endl;
  cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
  exit(EXIT_FAILURE);
}

int main(int argc, char * argv[])
{
```

```
  // Lectura de parámetros
  if (argc!=3)
    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 del generador de números pseudoaleatorios
  for (int i=0; i<tam; i++)  // Recorrer vector
    v[i] = rand() % vmax;    // Generar aleatorio [0,vmax[

  ordenar(v,tam); // de esta forma forzamos el peor caso

  int *w = new int[tam];
  for (int i=0; i<tam; i++)
    w[i] = v[tam-1-i];

  //Vamos a calcular el tiempo cuando ya está ordenado (peor caso)
  clock_t tini;    // Anotamos el tiempo de inicio
  tini=clock();

  ordenar(w,tam); // de esta forma forzamos el peor caso

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

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

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

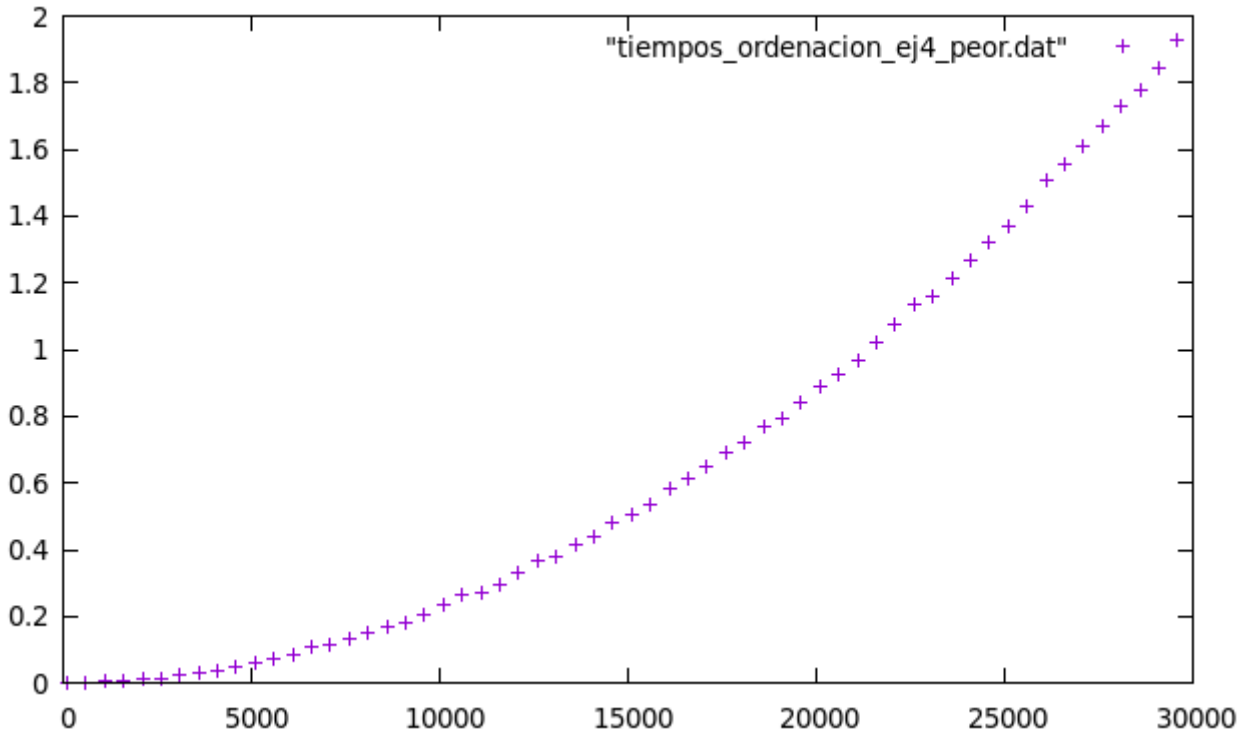
Luego he creado el fichero “ejecuciones_ordenacion_ej4_peor.csh”:

```
#!/bin/csh
@ inicio = 100
@ fin = 30000
@ incremento = 500
set ejecutable = ordenacion_ej4_peor
set salida = tiempos_ordenacion_ej4_peor.dat

@ i = $inicio
echo > $salida
while ( $i <= $fin )
  echo Ejecución tam = $i
  echo `./{$ejecutable} $i 10000` >> $salida
  @ i += $incremento
end
```

Y he obtenido los datos:
100 2.8e-05
600 0.000993
1100 0.004425
1600 0.007616
2100 0.009898
2600 0.014975
3100 0.023196
3600 0.029289
4100 0.038789
4600 0.049604
5100 0.059507
5600 0.069931
6100 0.082521
6600 0.109399
7100 0.114905
7600 0.129679
8100 0.151649
8600 0.167717
9100 0.180819
9600 0.207847
10100 0.235487
10600 0.263234
11100 0.272016
11600 0.29496
12100 0.331428
12600 0.364959
13100 0.381194
13600 0.414212
14100 0.440407
14600 0.482492
15100 0.506679
15600 0.537459
16100 0.584982
16600 0.612498
17100 0.649382
17600 0.689556
18100 0.722832
18600 0.768444
19100 0.795237
19600 0.839319
20100 0.886637
20600 0.927432
21100 0.969414
21600 1.02147
22100 1.0732
22600 1.13468
23100 1.1589
23600 1.213
24100 1.26874
24600 1.32002

25100 1.3677
25600 1.43239
26100 1.50562
26600 1.55686
27100 1.60968
27600 1.66841
28100 1.73104
28600 1.77863
29100 1.84552
29600 1.92861

Y la gráfica:



En la que observamos que en este segundo caso tarda el doble que en el apartado a).
Aunque no es tan mal algoritmo como el uno, que tarda más que este (el eje Y del ejercicio 1 va desde el 0 al 3, y este desde el 0 al 2).