

En primer lugar, las características del ordenador donde se ha realizado la práctica 1 son:

- Procesador Intel Core i5-4200H con una velocidad base de 2.8Ghz y con velocidad turbo máxima de 3.4Ghz, con 2 cores físicos y 4 lógicos.
- Memoria RAM de 8Gb
- Sistema Operativo Ubuntu 18.04.3 LTS montado en una arquitectura de 64 bits.

```

reko98@R98: ~
Archivo Editar Ver Buscar Terminal Ayuda
reko98@R98:~$ lscpu
Arquitectura:                x86_64
Modo(s) de operación de las CPUs: 32-bit, 64-bit
Orden de los bytes:          Little Endian
CPU(s):                      4
Lista de la(s) CPU(s) en línea: 0-3
Hilo(s) de procesamiento por núcleo: 2
Núcleo(s) por «socket»:      2
«Socket(s)»:                  1
Modo(s) NUMA:                 1
ID de fabricante:             GenuineIntel
Familia de CPU:               6
Modelo:                       60
Nombre del modelo:            Intel(R) Core(TM) i5-4200H CPU @ 2.80GHz
Revisión:                     3
CPU MHz:                      2044.242
CPU MHz máx.:                 3400.0000
CPU MHz mín.:                 800.0000
BogoMIPS:                     5586.92
Virtualización:               VT-x
Cache L1d:                    32K
Cache L1i:                    32K
Cache L2:                     256K
Cache L3:                     3072K
CPU(s) del nodo NUMA 0:       0-3
Indicadores:                  fpu vme de pse tsc msr pae mce cx8 apic sep mtrr pge mca cmov pat pse36 clflush dt
  
```

(Captura de pantalla de características del ordenador)

Ejercicio 1

```

1 void ordenar(int *v, int n){
2     for (int i=0; i<n-1; i++)
3         for(int j=0; j<n-i-1; j++)
4             if(v[j]>v[j+1]){
5                 //swap(v[j], v[j+1]);
6                 int aux = v[j];
7                 v[j] = v[j+1];
8                 v[j+1] = aux;
9             }
10 }
  
```

(Captura de pantalla algoritmo de ordenación por burbuja)

Para calcular la eficiencia teórica del algoritmo de ordenación por burbuja vamos a calcular primero el numero de operaciones elementales que realiza:

- En la línea 8: Declaración, asignación, decremento y comparación → 4 OE
- En la línea 9: Declaración, asignación, 2 decrementos y comparación → 5 OE
- En la línea 10: 2 accesos, asignación e incremento → 4 OE
- En la línea 12: declaración, asignación y acceso → 3 OE
- En la línea 13: 2 accesos, asignación e incremento → 4 OE
- En la línea 14: acceso, asignación e incremento → 3 OE

Ahora podemos calcular la eficiencia teórica:

$$\begin{aligned}
 &4 + \left(\sum_{i=0}^{n-2} 5 + \left(\sum_{j=0}^{n-i-2} 3 + 4 + 3 + 4 + 2 \right) + 2 \right) = 4 + \left(\sum_{i=0}^{n-2} 7 + \sum_{j=0}^{n-i-2} 16 \right) = 4 + \sum_{i=0}^{n-2} 7 + \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 16 \\
 &\rightarrow 4 + \sum_{i=0}^{n-2} 7 = 4 + 7 \sum_{i=0}^{n-2} 1 = 4 + 7(n-1) \\
 &\rightarrow \sum_{i=0}^{n-2} \sum_{j=0}^{n-i-2} 16 = \sum_{i=0}^{n-2} \left(16 \sum_{j=0}^{n-i-2} 1 \right) = \sum_{i=0}^{n-2} 16(n-i-1) = 16 \sum_{i=0}^{n-2} (n-i-1) = 16 \sum_{i=0}^{n-2} n - \sum_{i=0}^{n-2} i - \sum_{i=0}^{n-2} 1 \rightarrow \\
 &\rightarrow 16n(n-1) - 8(n-1)(n-2) - 16(n-1)
 \end{aligned}$$

por tanto:

$$\begin{aligned}
 &4 + 7(n-1) + 16n(n-1) - 8(n-1)(n-2) - 16(n-1) = 4 + (n-1)(7 + 16n - 8(n-2) - 16) \rightarrow \\
 &\rightarrow 4 + (n-1)(8n+7) = 8n^2 + n + 3
 \end{aligned}$$

La eficiencia teórica de este algoritmo tiene un orden $O(n^2)$.

Los resultados de la ejecución del script *ejecuciones_ordenacion.csh* son:

Tamaño	Tiempo
100	2.5e-5
600	0.000713
1100	0.002375
1600	0.00471
2100	0.008728
2600	0.014164
3100	0.018187
3600	0.025179
4100	0.037181
4600	0.043447
5100	0.055946
5600	0.066595
6100	0.085112
6600	0.097576
7100	0.117823
7600	0.139591
8100	0.156908
8600	0.17763
9100	0.202924
9600	0.227778

Tamaño	Tiempo
10100	0.251989
10600	0.281214
11100	0.310764
11600	0.339029
12100	0.373241
12600	0.40485
13100	0.441979
13600	0.479325
14100	0.5125
14600	0.558019
15100	0.599239
15600	0.640674
16100	0.681108
16600	0.736553
17100	0.77499
17600	0.81807
18100	0.872487
18600	0.915192
19100	0.973112
19600	1.0258

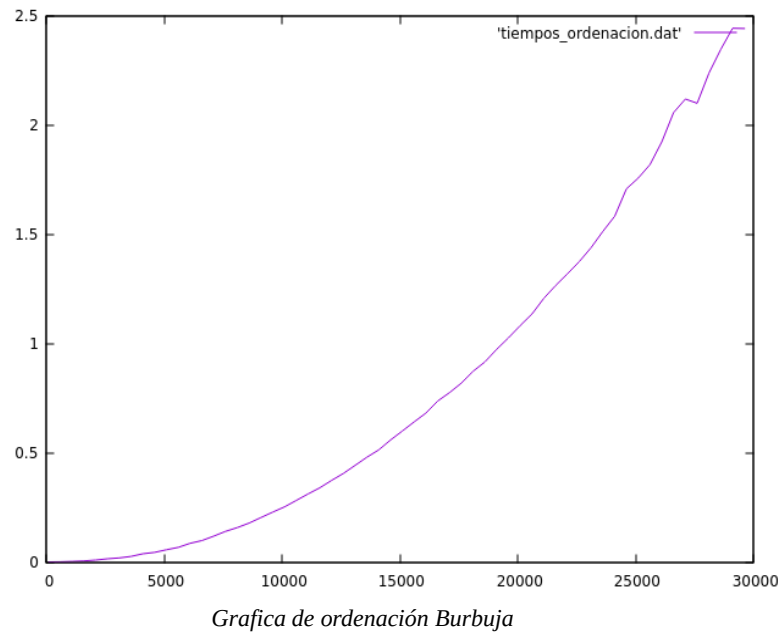
Tamaño	Tiempo
20100	1.08214
20600	1.13506
21100	1.20632
21600	1.26522
22100	1.31872
22600	1.37444
23100	1.43763
23600	1.51246
24100	1.58196
24600	1.70753
25100	1.75682
25600	1.81809
26100	1.92302
26600	2.0561
27100	2.11855
27600	2.09898
28100	2.23757
28600	2.34615
29100	2.44207
29600	2.44072

Como podemos observar en las tablas de tamaño y tiempo:

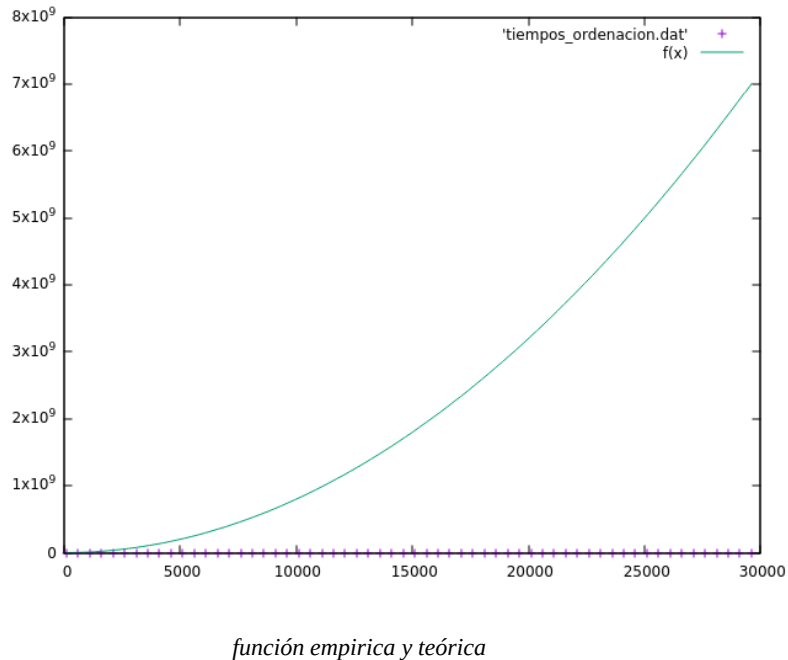
-A partir del tamaño 19600 el tiempo comienza a valer más de 1 segundo.

-Este no llega a 2.5 segundos en ninguna de las ejecuciones que realiza, quedándose en 2.44072 segundos en la última iteración de tamaño 29600.

La gráfica que representa los resultados anteriores es:

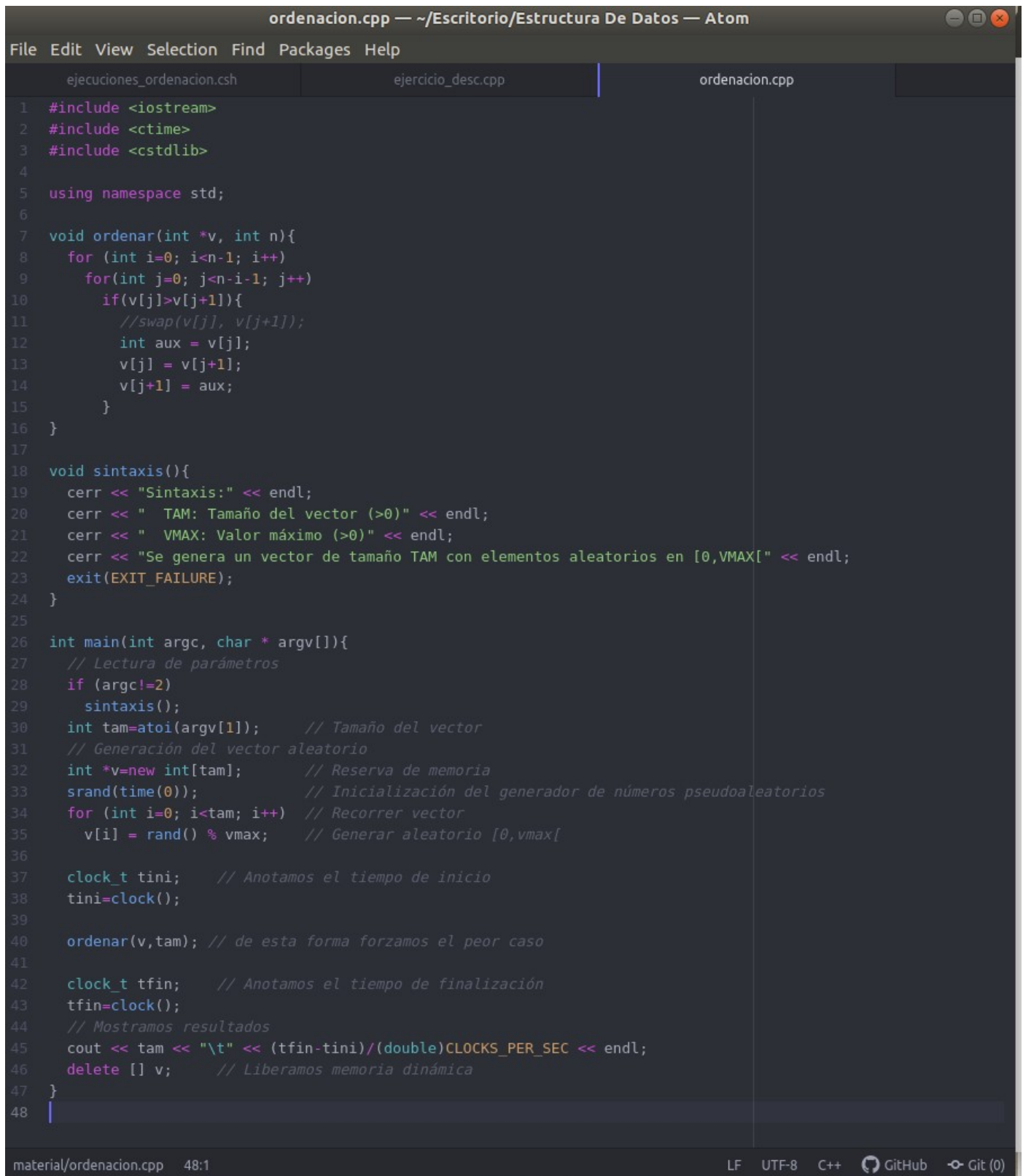


Si contrastamos los resultados de la grafica anterior con los de la función teórica obtenemos:



Se observa claramente que no son similares, pues la función teórica crece de forma más rápida que la función empírica, que representada en este gráfico parece una función lineal.

A continuación se muestra una captura de pantalla del código empleado para realizar el ejercicio 1 de Ordenacion.cpp

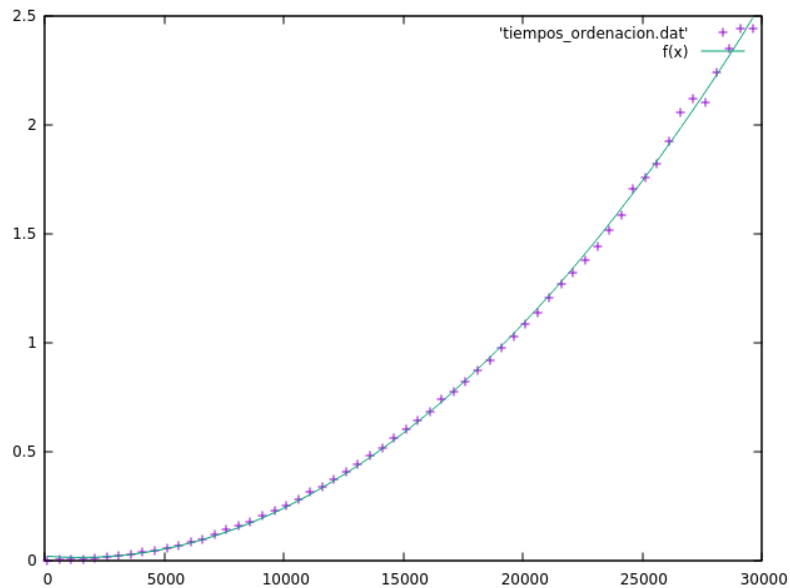


```
ordenacion.cpp — ~/Escritorio/Estructura De Datos — Atom
File Edit View Selection Find Packages Help
ejecuciones_ordenacion.csh ejercicio_desc.cpp ordenacion.cpp
1 #include <iostream>
2 #include <ctime>
3 #include <cstdlib>
4
5 using namespace std;
6
7 void ordenar(int *v, int n){
8     for (int i=0; i<n-1; i++)
9         for(int j=0; j<n-i-1; j++)
10             if(v[j]>v[j+1]){
11                 //swap(v[j], v[j+1]);
12                 int aux = v[j];
13                 v[j] = v[j+1];
14                 v[j+1] = aux;
15             }
16 }
17
18 void sintaxis(){
19     cerr << "Sintaxis:" << endl;
20     cerr << "  TAM: Tamaño del vector (>0)" << endl;
21     cerr << "  VMAX: Valor máximo (>0)" << endl;
22     cerr << "Se genera un vector de tamaño TAM con elementos aleatorios en [0,VMAX[" << endl;
23     exit(EXIT_FAILURE);
24 }
25
26 int main(int argc, char * argv[]){
27     // Lectura de parámetros
28     if (argc!=2)
29         sintaxis();
30     int tam=atoi(argv[1]); // Tamaño del vector
31     // Generación del vector aleatorio
32     int *v=new int[tam]; // Reserva de memoria
33     srand(time(0)); // Inicialización del generador de números pseudoaleatorios
34     for (int i=0; i<tam; i++) // Recorrer vector
35         v[i] = rand() % vmax; // Generar aleatorio [0,vmax[
36
37     clock_t tini; // Anotamos el tiempo de inicio
38     tini=clock();
39
40     ordenar(v,tam); // de esta forma forzamos el peor caso
41
42     clock_t tfin; // Anotamos el tiempo de finalización
43     tfin=clock();
44     // Mostramos resultados
45     cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
46     delete [] v; // Liberamos memoria dinámica
47 }
48
```

material/ordenacion.cpp 48:1 LF UTF-8 C++ GitHub Git (0)

Ejercicio 2

En este ejercicio, la función con la que se van a ajustar los resultados del ejercicio 1 tiene la forma $ax^2 + bx + c$. La gráfica obtenida es la siguiente:



Podemos observar que en este caso, la gráfica de ordenación se ajusta bastante bien a la función declarada.

Las constantes ocultas obtenidas de ajustar $f(x)$ a los resultados obtenidos son:

```

reko98@R98: ~/Escritorio/Estructura De Datos/material
Archivo Editar Ver Buscar Terminal Ayuda

After 12 iterations the fit converged.
final sum of squares of residuals : 0.0242312
rel. change during last iteration : -1.63723e-11

degrees of freedom      (FIT_NDF)                : 57
rms of residuals        (FIT_STDFIT) = sqrt(WSSR/ndf) : 0.0206182
variance of residuals   (reduced chisquare) = WSSR/ndf : 0.000425109

Final set of parameters      Asymptotic Standard Error
=====
a      = 3.12882e-09          +/- 3.971e-11   (1.269%)
b      = -9.283e-06           +/- 1.219e-06   (13.13%)
c      = 0.0188184           +/- 0.007831   (41.61%)

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

```

Ejercicio 3

El código del ejercicio 3 que se va a analizar es el siguiente:

```

7  int operacion(int *v, int n, int x, int inf, int sup) {
8      int med;
9      bool enc=false;
10     while ((inf<sup) && (!enc)) {
11         med = (inf+sup)/2;
12         if (v[med]==x)
13             enc = true;
14         else if (v[med] < x)
15             inf = med+1;
16         else
17             sup = med-1;
18     }
19     if (enc)
20         return med;
21     else
22         return -1;
23 }

```

En él podemos observar que la operación que realiza es el método de búsqueda binaria, que busca un elemento x pasado como argumento dentro de un vector (ya ordenado).

Para calcular la eficiencia teórica del algoritmo vamos a anotar el número de operaciones elementales que realiza:

- En la línea 8: declaración → 1 OE
- En la línea 9: declaración y asignación → 2 OE
- En la línea 10: comparación, negación y operador lógico → 3 OE
- En la línea 11: asignación, suma y división → 3 OE
- En la línea 12: comparación y acceso → 2 OE
- En la línea 13: asignación → 1 OE
- En la línea 14: acceso y comparación → 2 OE
- En la línea 15: incremento y asignación → 2 OE
- En la línea 17: decremento y asignación → 2 OE
- En la línea 19: comprobación → 1 OE
- En la línea 20: return → 1 OE
- En la línea 22: return → 1 OE

Por tanto, su eficiencia teórica:

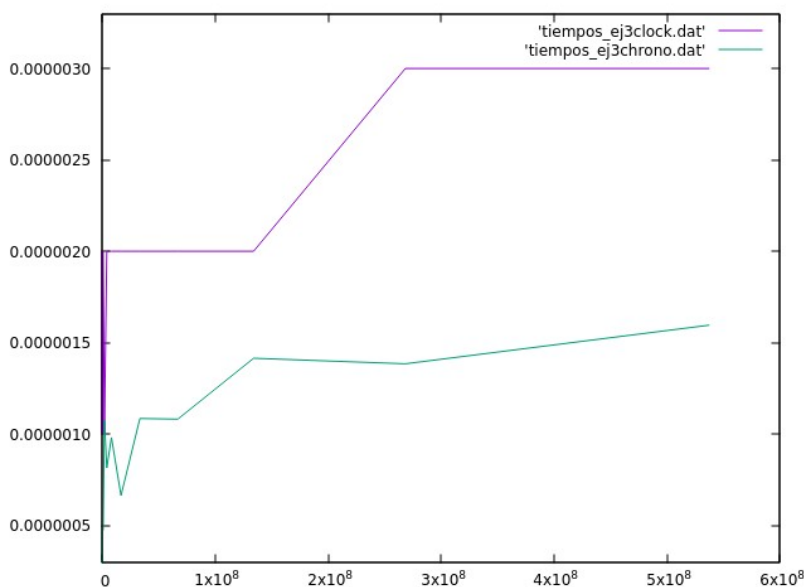
$$1+2+3+\sum_{i=0}^{\log(n)} (3+4)+1+1=8+\sum_{i=0}^{\log(n)} 7=8+7\sum_{i=0}^{\log(n)} 1=8+7(\log(n)+1)=7\log(n)+15$$

De este resultado podemos concluir que tiene un orden $O(\log(n))$.

Para estudiar el comportamiento del algoritmo he realizado varias pruebas con ambos ficheros, tanto aquel que usa la librería *chrono* como aquel que no, el script realiza saltos en potencias de dos, y los resultados que se obtienen son:

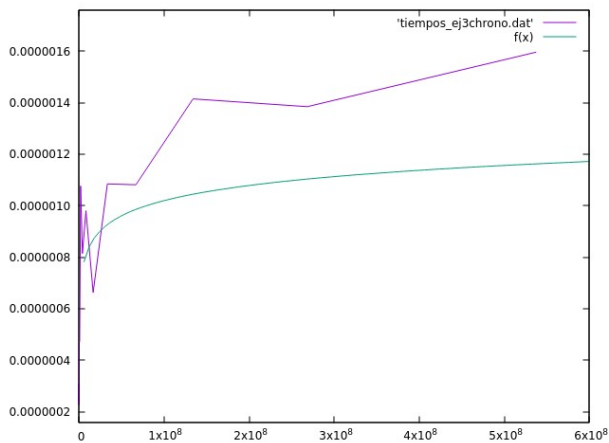
Tamaño	Tiempo (<i>clock()</i>)	Tiempo (<i>chrono</i>)
1024	2e-06	2.53e-07
4096	2e-06	2.26e-07
16384	1e-06	3.08e-07
65536	2e-06	2.74e-07
131072	1e-06	2.4e-07
262144	1e-06	2.34e-07
524288	2e-06	4.7e-07
1048576	2e-06	4.74e-07
2097152	1e-06	1.076e-06
4194304	2e-06	8.14e-07
8388608	2e-06	9.8e-07
16777216	2e-06	6.63e-07
33554432	2e-06	1.084e-06
67108864	2e-06	1.081e-06
134217728	2e-06	1.414e-06
268435456	3e-06	1.384e-06
536870912	3e-06	1.595e-06

Si representamos los resultados en una gráfica:

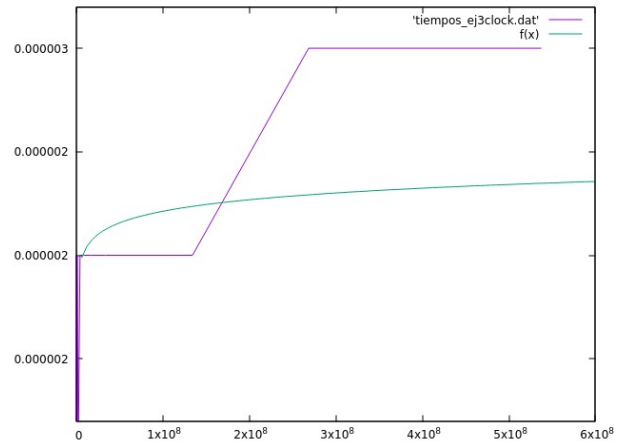


Podemos observar que, aunque con ciertas anormalidades, ambos resultados se asimilan a la gráfica de una función logarítmica, pero la diferencia de tiempo es bastante notable entre el algoritmo que usa la función *clock* y la que usa la librería *chrono*, siendo esta última más rápida que la primera.

Realizando el ajuste de las gráficas a la función logarítmica obtenemos:

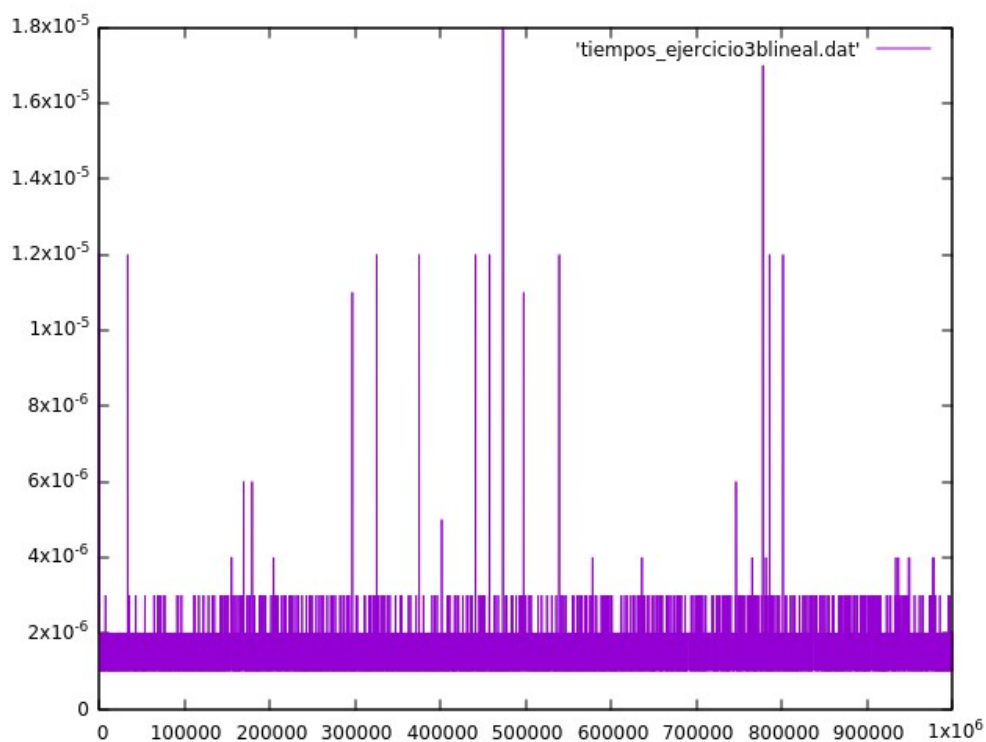


ajuste gráfica chrono



ajuste gráfica clock

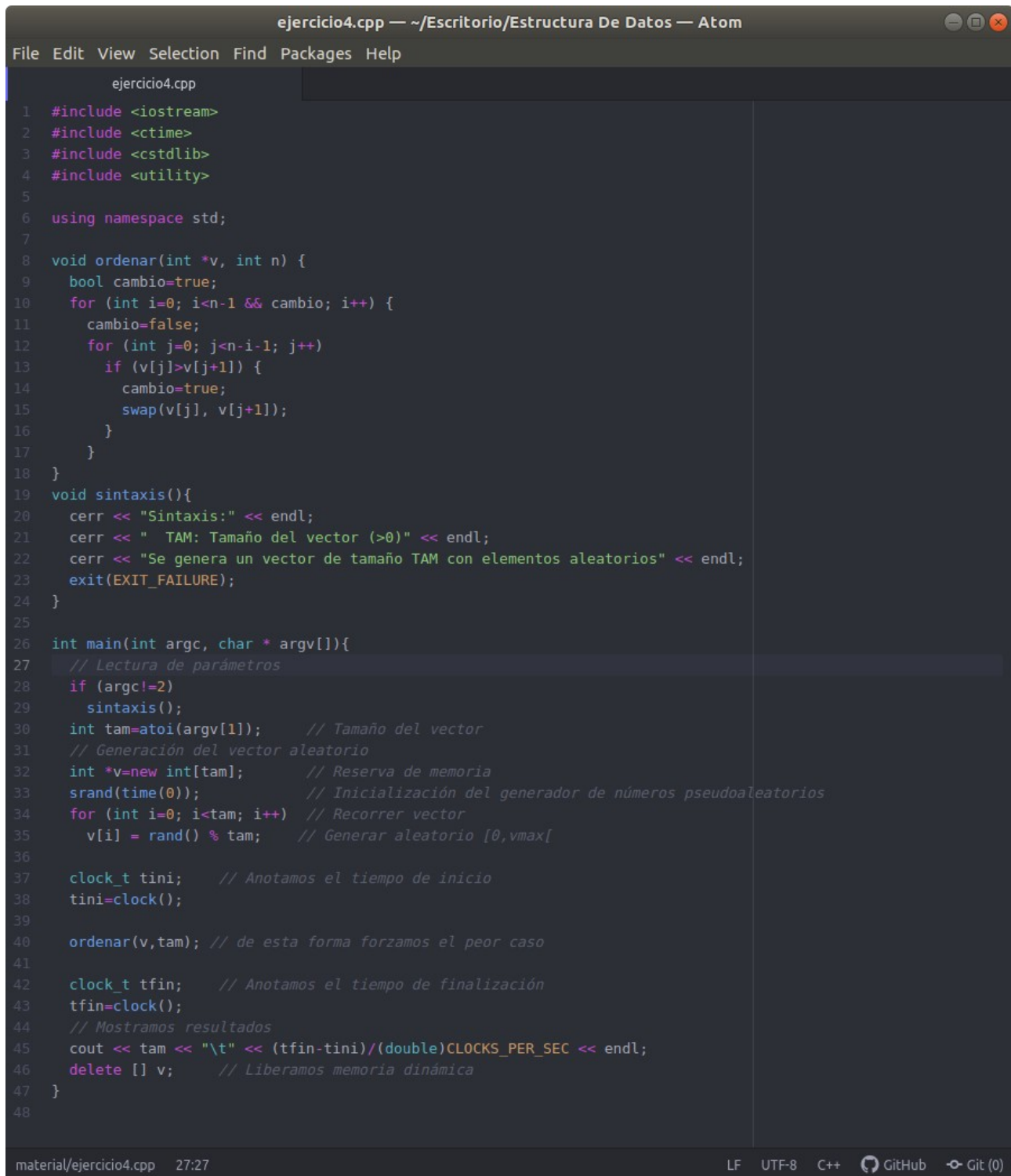
Si empleamos el script de *ejecuciones_blineal.csh* para obtener el comportamiento del algoritmo si se hubiesen realizado incrementos lineales, obtenemos este resultado:



Si observamos esta gráfica podemos ver claramente que existen picos demasiado sobresalientes mientras que la mayor parte del algoritmo los tiempos parecen ser estables, como si se tratase de una función lineal

Ejercicio 4

Esta es una captura del código que se va a emplear en el ejercicio 4:



```
ejercicio4.cpp — ~/Escritorio/Estructura De Datos — Atom
File Edit View Selection Find Packages Help

ejercicio4.cpp
1  #include <iostream>
2  #include <ctime>
3  #include <cstdlib>
4  #include <utility>
5
6  using namespace std;
7
8  void ordenar(int *v, int n) {
9      bool cambio=true;
10     for (int i=0; i<n-1 && cambio; i++) {
11         cambio=false;
12         for (int j=0; j<n-i-1; j++)
13             if (v[j]>v[j+1]) {
14                 cambio=true;
15                 swap(v[j], v[j+1]);
16             }
17     }
18 }
19 void sintaxis(){
20     cerr << "Sintaxis:" << endl;
21     cerr << "  TAM: Tamaño del vector (>0)" << endl;
22     cerr << "Se genera un vector de tamaño TAM con elementos aleatorios" << endl;
23     exit(EXIT_FAILURE);
24 }
25
26 int main(int argc, char * argv[]){
27     // Lectura de parámetros
28     if (argc!=2)
29         sintaxis();
30     int tam=atoi(argv[1]);    // Tamaño del vector
31     // Generación del vector aleatorio
32     int *v=new int[tam];       // Reserva de memoria
33     srand(time(0));            // Inicialización del generador de números pseudoaleatorios
34     for (int i=0; i<tam; i++)  // Recorrer vector
35         v[i] = rand() % tam;   // Generar aleatorio [0,vmax[
36
37     clock_t tini;              // Anotamos el tiempo de inicio
38     tini=clock();
39
40     ordenar(v,tam); // de esta forma forzamos el peor caso
41
42     clock_t tfin;              // Anotamos el tiempo de finalización
43     tfin=clock();
44     // Mostramos resultados
45     cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
46     delete [] v;              // Liberamos memoria dinámica
47 }
48
```

material/ejercicio4.cpp 27:27 LF UTF-8 C++ GitHub Git (0)

La parte del código que vamos a analizar es:

```

8 void ordenar(int *v, int n) {
9     bool cambio=true;
10    for (int i=0; i<n-1 && cambio; i++) {
11        cambio=false;
12        for (int j=0; j<n-i-1; j++)
13            if (v[j]>v[j+1]) {
14                cambio=true;
15                swap(v[j], v[j+1]);
16            }
17    }
18 }

```

Para obtener la eficiencia empírica vamos a analizar primero cuantas operaciones elementales realiza:

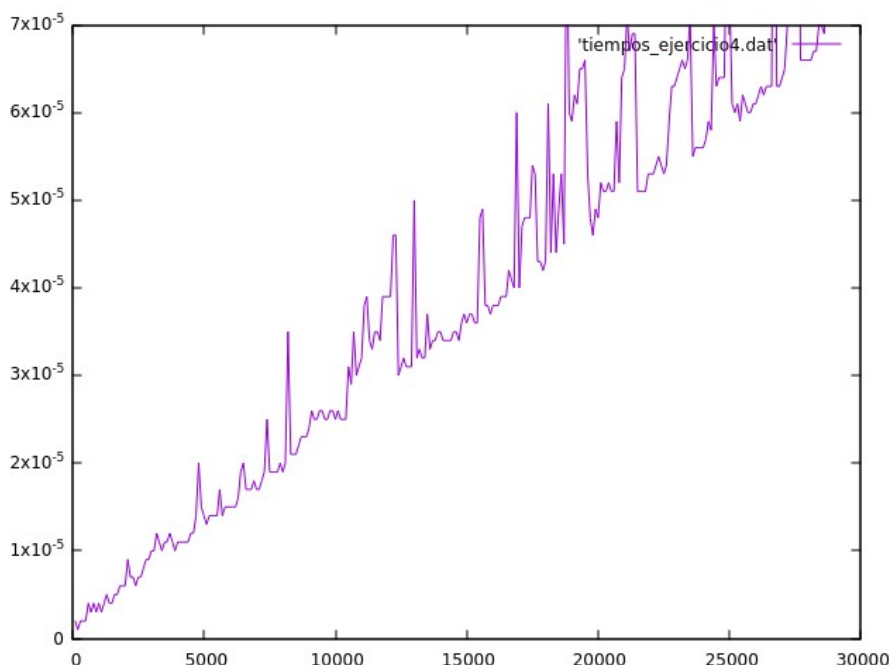
- En la línea 9: declaración y asignación → 2 OE
- En la línea 10: declaración, asignación, decremento y 2 comparaciones → 4 OE
- En la línea 11: asignación → 1 OE
- En la línea 12: declaración, asignación, 2 decrementos y comparación → 5 OE
- En la línea 13: 2 accesos, comparación, incremento → 4 OE
- En la línea 14: asignación → 1 OE
- En la línea 15: función swap → 6 OE

En el mejor de los casos, el primer for sólo se haría para $i=0$, el segundo para $j<n-1$ y en cada iteración se hace la comparación del If. Por ende, la ecuación resultante sería:

$$2+4+1+5+\left(\sum_{j=0}^{n-2} 4+2\right)+3=15+6(n-1)=15+6n-6=6n+9$$

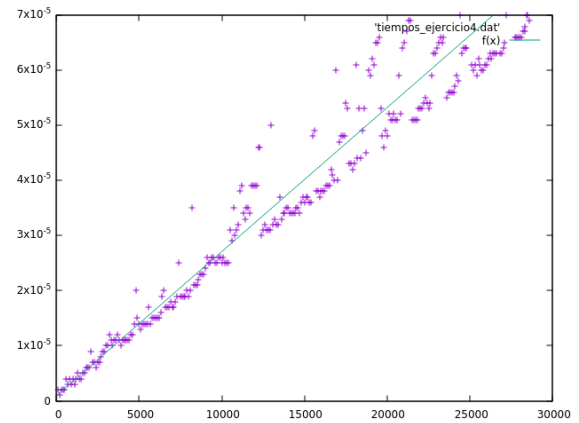
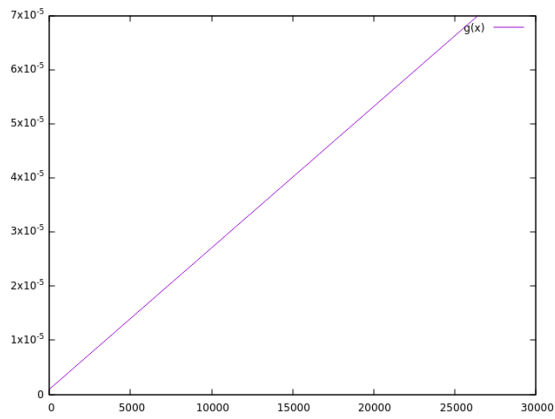
La eficiencia es de orden $O(n)$.

El gráfico obtenido de hacer la prueba del algoritmo con el script de *ejecuciones_blineal.csh* con un comienzo de 100 hasta 30000 con un incremento de 100 durante cada iteración es:



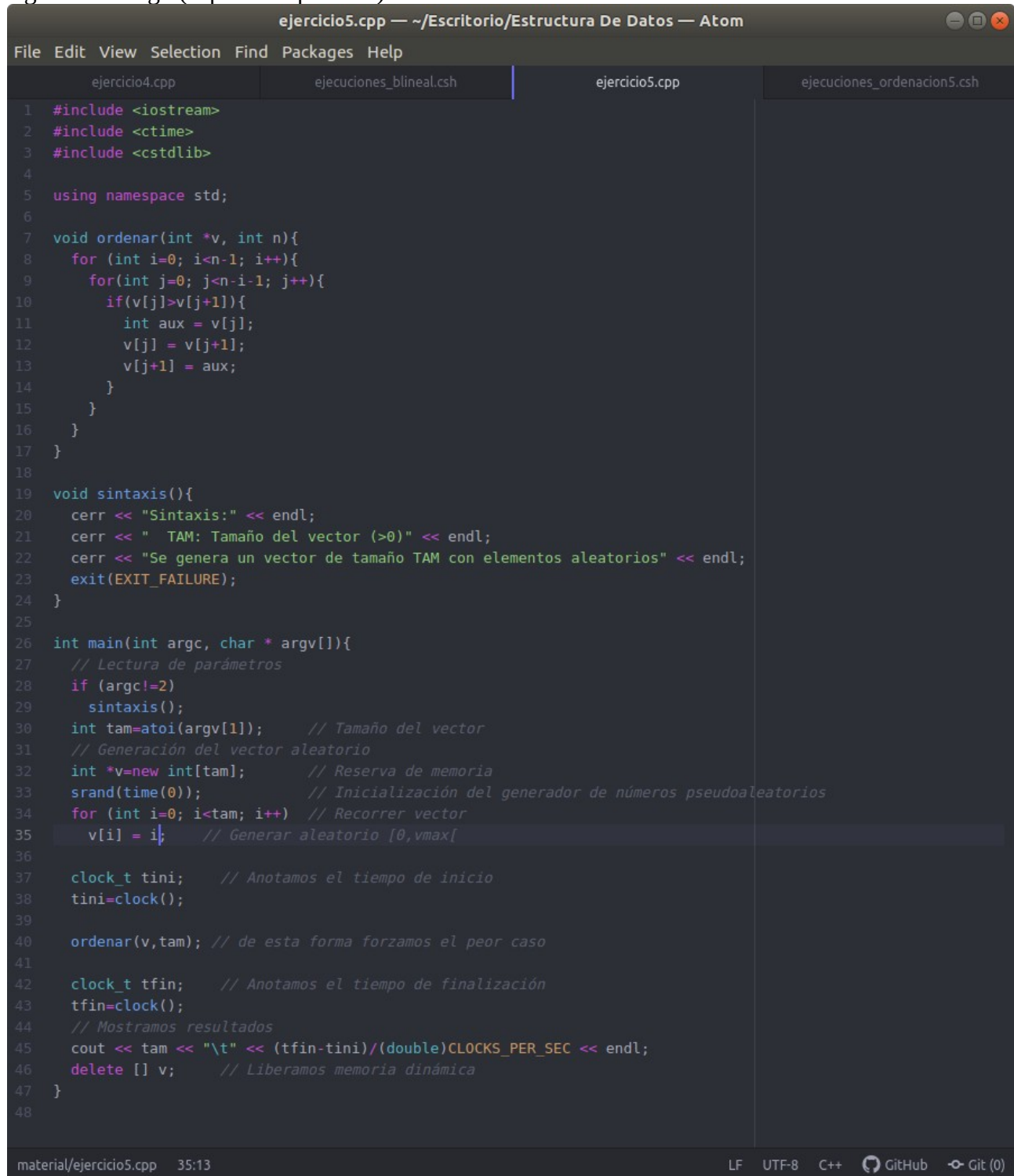
Como podemos observar, esta función tiene, aunque presente bastantes picos, una tendencia similar a la de una función lineal.

En la primera grafica se encuentra la eficiencia empírica (izquierda) y en la segunda se encuentra la misma pero ajusta junto con la gráfica de ejecuciones.



Ejercicio 5

Para analizar el mejor de los casos en el algoritmo de ordenación por burbuja se ha empleado el siguiente código (captura de pantalla):



```
ejercicio5.cpp — ~/Escritorio/Estructura De Datos — Atom
File Edit View Selection Find Packages Help
ejercicio4.cpp ejecuciones_blineal.csh ejercicio5.cpp ejecuciones_ordenacion5.csh
1 #include <iostream>
2 #include <ctime>
3 #include <cstdlib>
4
5 using namespace std;
6
7 void ordenar(int *v, int n){
8     for (int i=0; i<n-1; i++){
9         for(int j=0; j<n-i-1; j++){
10             if(v[j]>v[j+1]){
11                 int aux = v[j];
12                 v[j] = v[j+1];
13                 v[j+1] = aux;
14             }
15         }
16     }
17 }
18
19 void sintaxis(){
20     cerr << "Sintaxis:" << endl;
21     cerr << " TAM: Tamaño del vector (>0)" << endl;
22     cerr << "Se genera un vector de tamaño TAM con elementos aleatorios" << endl;
23     exit(EXIT_FAILURE);
24 }
25
26 int main(int argc, char * argv[]){
27     // Lectura de parámetros
28     if (argc!=2)
29         sintaxis();
30     int tam=atoi(argv[1]); // Tamaño del vector
31     // Generación del vector aleatorio
32     int *v=new int[tam]; // Reserva de memoria
33     srand(time(0)); // Inicialización del generador de números pseudoaleatorios
34     for (int i=0; i<tam; i++) // Recorrer vector
35         v[i] = i; // Generar aleatorio [0,vmax[
36
37     clock_t tini; // Anotamos el tiempo de inicio
38     tini=clock();
39
40     ordenar(v,tam); // de esta forma forzamos el peor caso
41
42     clock_t tfin; // Anotamos el tiempo de finalización
43     tfin=clock();
44     // Mostramos resultados
45     cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
46     delete [] v; // Liberamos memoria dinámica
47 }
48
material/ejercicio5.cpp 35:13 LF UTF-8 C++ GitHub Git (0)
```

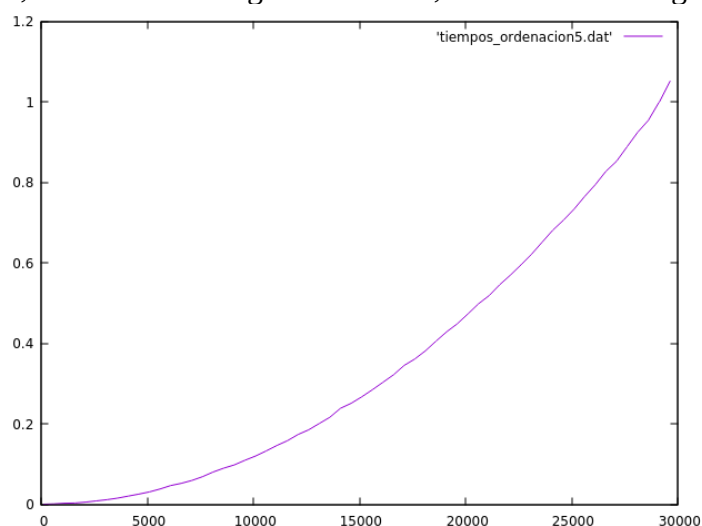
Como se trata de una especie de continuación el ejercicio 1, he utilizado también el mismo script con los mismos valores de inicio y fin e incluyendo un salto de valor 500 en cada iteración. He obtenido los siguientes resultados:

Tamaño	Tiempo
100	2.2e-05
600	0.000822
1100	0.002538
1600	0.003099
2100	0.005311
2600	0.008244
3100	0.01141
3600	0.015215
4100	0.020157
4600	0.025178
5100	0.030625
5600	0.037829
6100	0.046474
6600	0.051761
7100	0.059
7600	0.068323
8100	0.080044
8600	0.089796
9100	0.097595
9600	0.109135

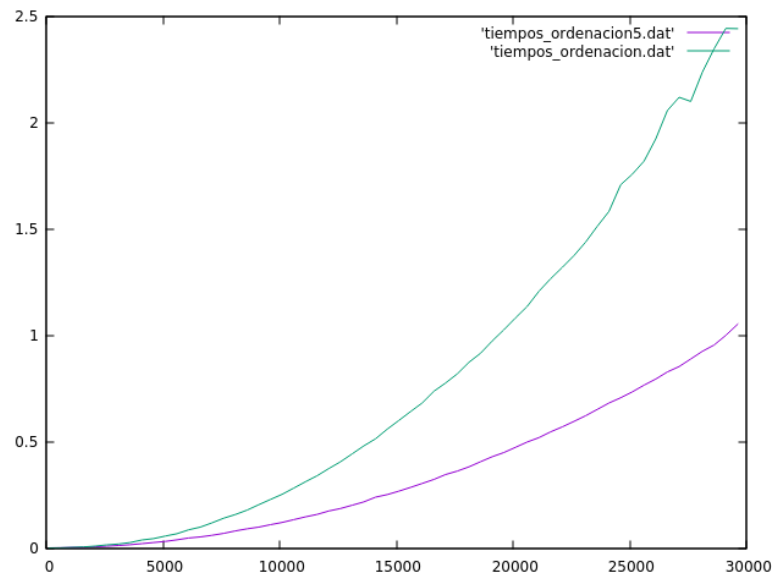
Tamaño	Tiempo
10100	0.119804
10600	0.132571
11100	0.145884
11600	0.157774
12100	0.173396
12600	0.184824
13100	0.200622
13600	0.216163
14100	0.238473
14600	0.250882
15100	0.266695
15600	0.284393
16100	0.303008
16600	0.321686
17100	0.344885
17600	0.360993
18100	0.381284
18600	0.405355
19100	0.428569
19600	0.44837

Tamaño	Tiempo
20100	0.472863
20600	0.497927
21100	0.518502
21600	0.54556
22100	0.569341
22600	0.594858
23100	0.621394
23600	0.651472
24100	0.680931
24600	0.705721
25100	0.733226
25600	0.764994
26100	0.793992
26600	0.827196
27100	0.852327
27600	0.888354
28100	0.924489
28600	0.953833
29100	0.998873
29600	1.05061

El resultado es bastante bueno pues, como podemos observar, en su máximo tamaño, con un vector de 29600 elementos, tarda 1.05061 segundos. Ahora, si observamos la grafica que genera:



Para comparar la diferencia entre la gráfica generada por el ejercicio 1 simplemente he representado las dos gráficas en una sola tabla, así:



En esta representación, podemos ver que si bien ambas gráficas tienen una tendencia similar a la de una función cuadrática, existe una diferencia de crecimiento de tiempo bastante notable entre el mejor caso del ejercicio 5 y el caso estándar del ejercicio 1 siendo mejores los tiempos del mejor caso (ejercicio5).

Ahora, para analizar el peor de los casos, únicamente he modificado de forma que los valores que contiene el vector se encuentren ordenados de mayor a menor, quedando de la siguiente forma:

```

26 int main(int argc, char * argv[]){
27     // Lectura de parámetros
28     if (argc!=2)
29         sintaxis();
30     int tam=atoi(argv[1]);    // Tamaño del vector
31     // Generación del vector aleatorio
32     int *v=new int[tam];       // Reserva de memoria
33     srand(time(0));            // Inicialización del generador de números pseudoaleatorios
34     for (int i=0; i<tam; i++)  // Recorrer vector
35         v[i] = tam - i;       // Generar aleatorio [0,vmax]
36
37     clock_t tini;              // Anotamos el tiempo de inicio
38     tini=clock();
39
40     ordenar(v,tam); // de esta forma forzamos el peor caso
41
42     clock_t tfin;              // Anotamos el tiempo de finalización
43     tfin=clock();
44     // Mostramos resultados
45     cout << tam << "\t" << (tfin-tini)/(double)CLOCKS_PER_SEC << endl;
46     delete [] v;              // Liberamos memoria dinámica
47 }
48

```

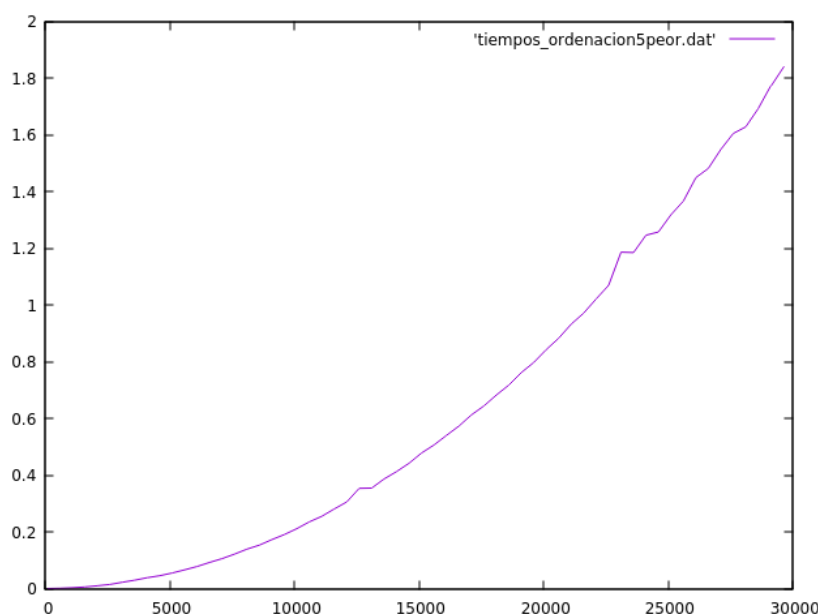
Utilizando el mismo script, con los mismos parámetros, los resultados son:

Tamaño	Tiempo
100	2.5e-05
600	0.000831
1100	0.002815
1600	0.005358
2100	0.009435
2600	0.014075
3100	0.021895
3600	0.029506
4100	0.038358
4600	0.045248
5100	0.054844
5600	0.065978
6100	0.077806
6600	0.092056
7100	0.105307
7600	0.121478
8100	0.138948
8600	0.153313
9100	0.172676
9600	0.190537

Tamaño	Tiempo
10100	0.210967
10600	0.235158
11100	0.254916
11600	0.280359
12100	0.305131
12600	0.353237
13100	0.354241
13600	0.386171
14100	0.412317
14600	0.442143
15100	0.477823
15600	0.506292
16100	0.539666
16600	0.572973
17100	0.612232
17600	0.643732
18100	0.682658
18600	0.717614
19100	0.761346
19600	0.796831

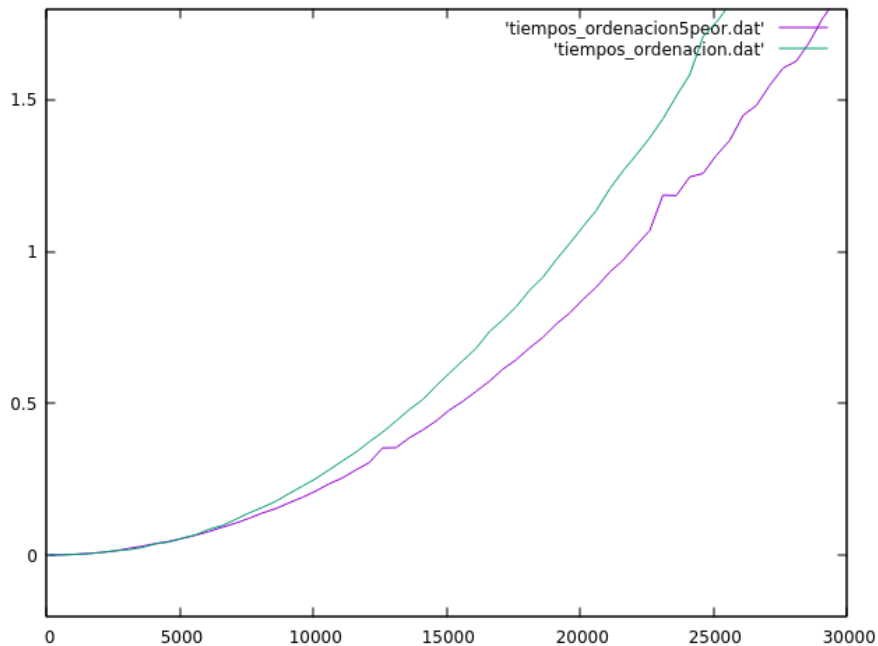
Tamaño	Tiempo
20100	0.84229
20600	0.882347
21100	0.932207
21600	0.932207
22100	1.02102
22600	1.06916
23100	1.18605
23600	1.18482
24100	1.24599
24600	1.25732
25100	1.31736
25600	1.36636
26100	1.44898
26600	1.48111
27100	1.54838
27600	1.60447
28100	1.62787
28600	1.6919
29100	1.77203
29600	1.83842

La gráfica que se obtiene tiene la siguiente forma:



Podemos observar que la tendencia es la misma que en el mejor de los casos, no obstante el valor de los tiempos si que ha cambiado notablemente, pues en su última iteración, con 29600 elementos, llega a consumir un tiempo de 1.83842, llegando a aproximarse a los 2 segundos.

Si analizamos las gráficas del peor caso junto con la del ejercicio 1 obtenemos la siguiente tabla de gráficas:



Como podemos observar la diferencia en este caso no es tan notable como ocurría en la comparación con el mejor caso y el ejercicio 1.

Sin embargo, podemos observar una curiosidad, y es que aquel algoritmo que ha obtenido unos mejores resultados de tiempo es el algoritmo del peor caso. Se puede observar que este tiene tiempos menores en comparación con el algoritmo 'estándar', algo que, en realidad, no tiene mucho sentido.

Ejercicio 6

Los datos generados por el código generado con la orden de compilación -O3 y utilizando el mismo script son:

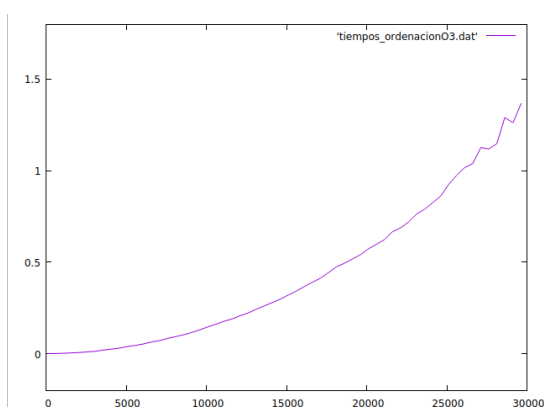
Tamaño	Tiempo
100	1.8e-05
600	0.000536
1100	0.001898
1600	0.004382
2100	0.006085
2600	0.009544
3100	0.013088
3600	0.019707
4100	0.024729
4600	0.030524
5100	0.039197
5600	0.045303
6100	0.053507
6600	0.064145
7100	0.071271
7600	0.083862
8100	0.093431
8600	0.10356
9100	0.115905
9600	0.130113

Tamaño	Tiempo
10100	0.146457
10600	0.161011
11100	0.176566
11600	0.189367
12100	0.207201
12600	0.222391
13100	0.242545
13600	0.260203
14100	0.278728
14600	0.29781
15100	0.319939
15600	0.341573
16100	0.366252
16600	0.389408
17100	0.411593
17600	0.442206
18100	0.473956
18600	0.494067
19100	0.516912
19600	0.540301

Tamaño	Tiempo
20100	0.572483
20600	0.597102
21100	0.623151
21600	0.666059
22100	0.686895
22600	0.719127
23100	0.762695
23600	0.78853
24100	0.824384
24600	0.860371
25100	0.9244
25600	0.974537
26100	1.0168
26600	1.03811
27100	1.12582
27600	1.11798
28100	1.14687
28600	1.29003
29100	1.26214
29600	1.36599

Representando ambas gráficas, observamos que la función compilada con O3 es más rápida que la que no tiene ninguna optimización de compilación.

gráfica con optimización O3



ambas gráficas comparadas

