Análisis de Algoritmos 2018/2019

Práctica 2

Claudia González Arteaga y Laura Iniesta Rodríguez, Grupo 1261.

|  |  |  |  |
| --- | --- | --- | --- |
| Código | Gráficas | Memoria | Total |
|  |  |  |  |

**1. Introducción.**

En esta práctica de Análisis de algoritmos vamos a implementar las funciones necesarias para la ordenación de menor a mayor de permutaciones mediante los algoritmos MergeSort y QuickSort que usan recursividad lo cuál hace más complejas estas funciones que las de la práctica anterior.

Hacemos uso de las funciones implementadas en la práctica anterior para generar permutaciones y calcular los tiempos de ejecución para medir la efectividad de las funciones.

**2. Objetivos**

Aquí indicáis el trabajo que vais a realizar en cada apartado.

**2.1 Apartado 1**

Implementar las funciones int mergesort(int\* tabla, int ip, int iu) que es recursiva y llama a la otra función que hay que crear, int merge(int\* tabla, int ip, int iu, int imedio) que ordena una permutación de menor a mayor y calcula las OB, número de veces que se realiza la comparación de clave.

**2.2 Apartado 2**

Ejecutar el ejercicio5.c, cambiando la función que pasábamos como argumento en la práctica 1 por mergesort. Obtenemos la tabla del número máximo, mínimo y promedio de Obs y el tiempo de ejecución para diferentes tamaños de permutación

**2.3 Apartado 3**

Implementamos 3 funciones, int quicksort(int\* tabla, int ip, int iu) que de forma recursiva calcula las OB llamando a int partir(int\* tabla, int ip, int iu ,int \*pos) que ordena la tabla desde el pivote que recibe como argumento \*pos, devuelve el número de Obs y llama a int medio(int \*tabla, int ip, int iu ,int \*pos) que se encarga de calcular el pivote desde el que se ordena que en este caso es la primera posición de la tabla y devuelve el número inicial de Obs que es 0.

**2.4 Apartado 4**

Ejecutar el ejercicio5.c, cambiando la función que pasábamos como argumento en el apartado 1 por quicksort. Obtenemos la tabla del número máximo, mínimo y promedio de Obs y el tiempo de ejecución para diferentes tamaños de permutación

**2.5 Apartado 5**

Implementamos int medio\_avg(int \*tabla, int ip, int iu, int \*pos) que funciona igual que la función del apartado 3, medio pero devuelve como pivote la posición media de la tabla, también implementamos las funciones quicksort\_avg y partir\_avg que son iguales a las del apartado 3 pero llaman a medio\_avg en vez de medio.

También implementamos la función int medio\_stat(int \*tabla, int ip, int iu, int \*pos) con su respectivo partir\_stat y quicksort\_stat que son iguales al apartado 3 pero en la función medio\_stat se comprueba cuál es el valor intermedio de los que se encuentran en la tabla en la primera posición, la última posición y la posición media haciendo que el pivote sea la posición que corresponde a ese valor por lo que se realizan 3 comparaciones de clave y la función devolverá 3 Obs.

Ejecutamos el ejercicio5.c con los dos quicksort y obtenemos los valores de tiempo, ob máxima, ob mínima y ob media.

**3. Herramientas y metodología**

El entorno que hemos usado para realizar esta práctica es Linux (Ubuntu), hemos programado en Atom y para la ejecución de los ejercicios hemos usado valgrind en la terminal de comandos para comprobar que no hay fallos de memoria.

**3.1 Apartado 1**

Para dar solución a este apartado de realización del algoritmo merge sort hemos estudiado cómo funciona dicho algoritmo (hemos visto el pseudocódigo en Moodle) y lo hemos implementado en c de la forma más eficaz posible y hemos calculado las obs en las comparaciones de claves. Hemos usado valgrind para ejecutar el ejercicio 4 con 0 fallos.

**3.2 Apartado 2**

Para poder hacer la gráfica que estudia la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación de nuestra función y la tabla cambiamos los programas ejercicio4.c y ejercicio5.c para que contengan el algoritmo escogido(merge sort en este caso).

Metemos como argumentos un ip y un iu negativos para ver si las comprobaciones funcionan correctamente y un ip más alto que un iu para comprobar que devuelve ERR.

**3.3 Apartado 3**

Para dar solución a este apartado hemos elaborado en c el algoritmo de Quicksort de la forma más eficaz que hemos podido, para ello hemos elaborado dos funciones más: partir y medio, que nos ayudan a realizar el algoritmo. Al ejecutar el ejercicio 4 y 5 con valgrind hemos obtenido 0 fallos de memoria. Hemos programado en Atom.

Metemos como argumentos un ip, un iu y un pos negativos para ver si las comprobaciones funcionan correctamente y un ip más alto que un iu para comprobar que devuelve ERR y así comprobar el correcto funcionamiento de la función.

**3.3 Apartado 4**

Para poder hacer la gráfica que estudia el promedio de ejecución, máximo, mínimo y promedio de operaciones básicas del algoritmo QuickSort en función del tamaño de la permutación y elaborar una tabla con estos datos cambiamos los programas ejercicio4.c y ejercicio5.c para que contengan el algoritmo escogido(quick sort en este caso).

**3.4 Apartado 5**

En este apartado elaboramos otras funciones medio\_avg( que nos devuelva la posición media de la tabla) y medio\_stat (que compara los valores de las posiciones ip, iu y la posición media de la tabla y devuelve la posición que contenga el valor intermedio entre las tres) para elaborar un quicksort\_stat y un quicksort\_avg. Hemos elaborado 3 quicksort para que no haya que estar cambiando continuamente el programa por medio\_stat, medio\_avg o partir\_avg.

Para poder hacer la gráfica que estudia los tiempos promedio de ejecución, máximo, mínimo y promedio de operaciones básicas con cada una de las tres rutinas pivote implementadas y su tabla correspondiente con dichos datos cambiamos los programas ejercicio4.c y ejercicio5.c para que contengan el algoritmo escogido(quick sort\_stat o quicksort\_avg en este caso).

**4. Código fuente**

**4.1 Apartado 1**

int mergesort(int\* tabla, int ip, int iu){

int aux = 0;

if(!tabla || ip< 0 || iu<0 || ip>iu) return ERR;

if (tabla[ip]==tabla[iu]){

return 1;

} else {

aux = (ip+iu)/2;

mergesort(tabla,ip,aux);

mergesort(tabla,aux+1,iu); }

return merge(tabla,ip,iu,aux);

}

int merge(int\* tabla, int ip, int iu, int imedio){

int\* taux = NULL;

int i=0 ,j=0, k=0, ob=0;

int tamanio=iu+1;

if(!tabla || ip< 0 || iu<0 || ip>iu) return ERR;

taux = (int\*)malloc(tamanio\*sizeof(int));

if(!taux)return ERR;

i=ip;

j=imedio+1;

k=ip;

while (i<=imedio && j<= iu){

if(tabla[i]<tabla[j]){

taux[k]=tabla[i];

i++;

}else{

taux[k]=tabla[j];

j++;

}

k++;

ob++;

}

if(i>imedio){

while(j<=iu){

taux[k]=tabla[j];

j++;

k++;

}

}else if(j>iu){

while(i<=imedio){

taux[k]=tabla[i];

i++;

k++;

}

}

for(k=ip;k<tamanio;k++){

tabla[k] = taux[k];

}

free(taux);

return ob;

}

**4.3Apartado 3**

int quicksort(int\* tabla, int ip, int iu){

int ob=0, m=0, pos=0;

if(ip > iu || !tabla || ip < 0 || iu < 0){

return ERR;

}

if(ip==iu){

return ob;

} else{

ob+=partir(tabla, ip, iu, &pos);

m=pos;

if (ip < m-1){

ob+= quicksort(tabla, ip, m-1);

}

if (m+1 < iu){

ob+= quicksort(tabla, m+1, iu);

}

}

return ob;

}

int partir(int\* tabla, int ip, int iu,int \*pos){

int aux=0, ob=0, i, m=0;

if(ip > iu || !tabla || ip < 0 || iu < 0 || !pos){

return ERR;

}

ob=medio(tabla, ip, iu, pos);

if(ob == ERR){

return ERR;

}

aux=tabla[ip];

tabla[ip]=tabla[\*pos];

tabla[\*pos]=aux;

(\*pos)=ip;

m=\*pos;

for(i=ip+1; i<=iu; i++){

ob++;

if (tabla[i]<tabla[m]){

(\*pos)++;

aux=tabla[i];

tabla[i]=tabla[\*pos];

tabla[\*pos]=aux;

}

}

aux=tabla[ip];

tabla[ip]=tabla[\*pos];

tabla[\*pos]=aux;

return ob;

}

int medio(int \*tabla, int ip, int iu,int \*pos){

if(ip > iu || !tabla || ip < 0 || iu < 0 || !pos){

return ERR;

}

\*pos=ip;

return 0;

}

4.5 Apartado 5

int medio\_avg(int \*tabla, int ip, int iu, int \*pos){

if(ip > iu || !tabla || ip < 0 || iu < 0 || !pos){

return ERR;

}

\*pos=(ip+ iu)/2;

return 0;

}

int medio\_stat(int \*tabla, int ip, int iu,int \*pos){

int m=0, ob=0;

m=(ip+iu)/2;

if(ip > iu || !tabla || ip < 0 || iu < 0 || !pos){

return ERR; }

ob=3;

if(tabla[ip] < tabla[m]){

if(tabla[m] < tabla[iu]){

\*pos=m;

} else if(tabla[ip] < tabla[iu]){

\*pos=iu;

} else{

pos=&ip;

}

} else{

if(tabla[ip] < tabla[iu]){

\*pos=ip;

} else if(tabla[m] < tabla[iu]){

\*pos=m;

} else{

\*pos=iu;

}

}

return ob;

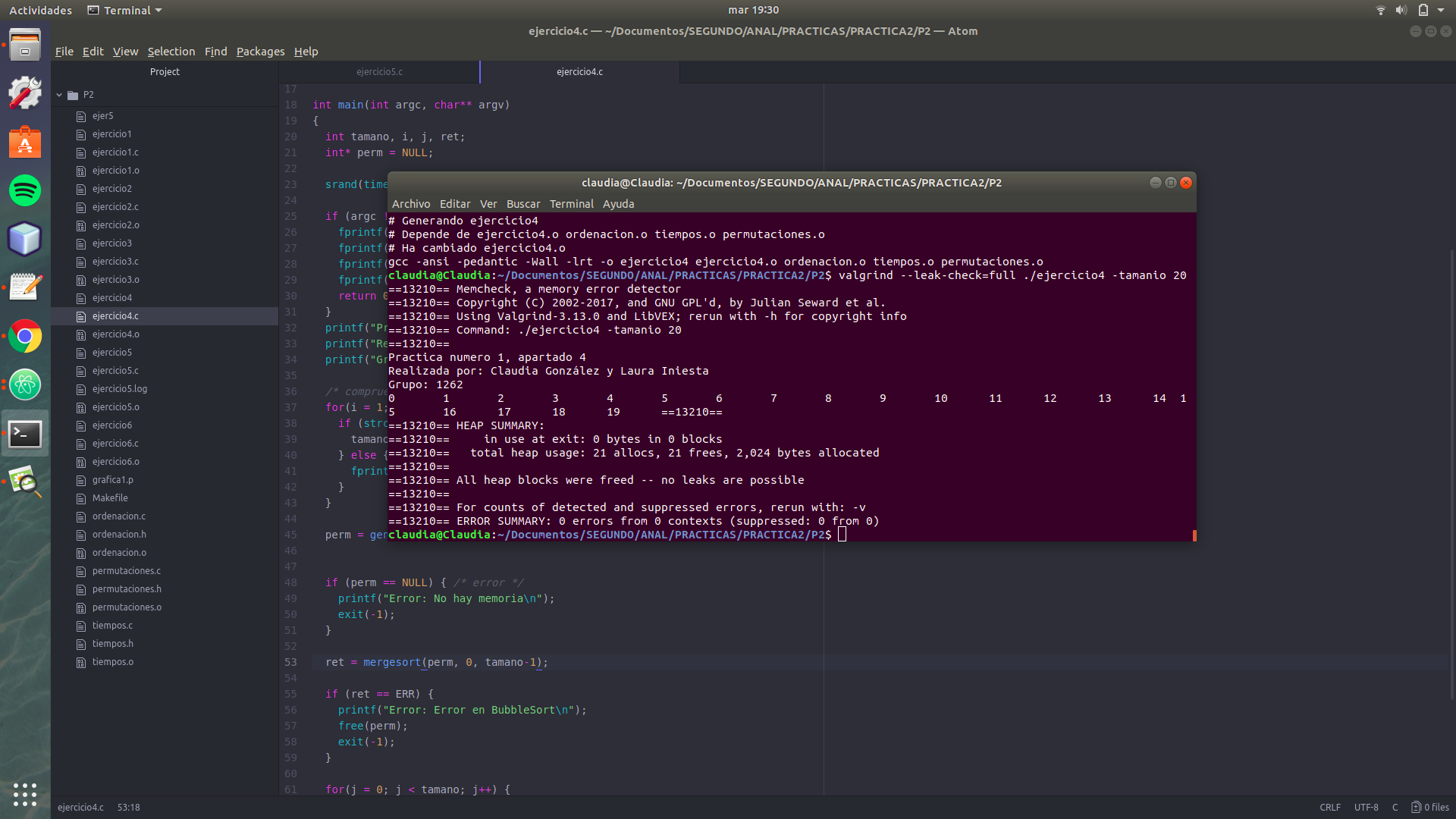
}

**5. Resultados, Gráficas**

Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

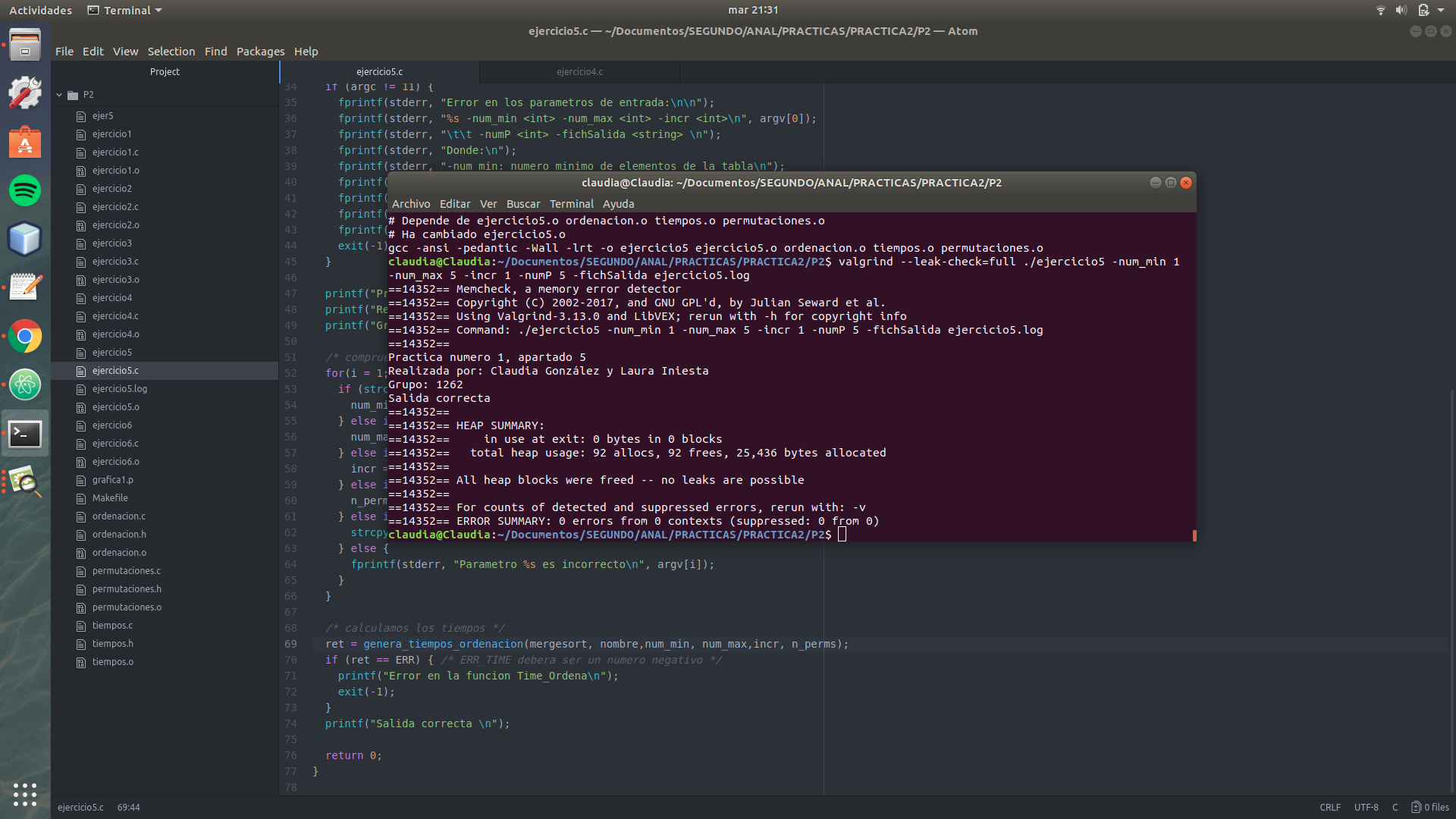
5.1 Apartado 1

Hacemos make en la terminal y posteriormente valgrind --leak-check=full ./ejercicio4 -tamanio 20 y obtenemos la salida esperada:

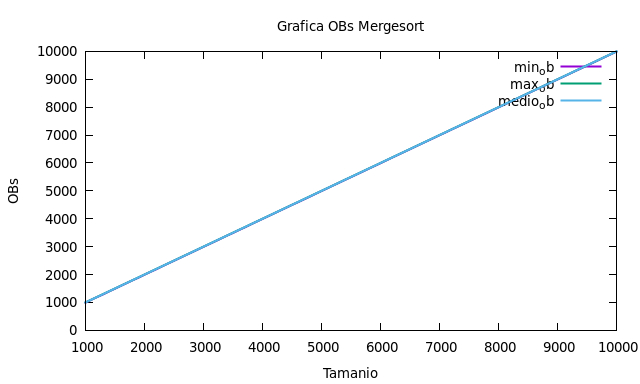


5.2 Apartado 2

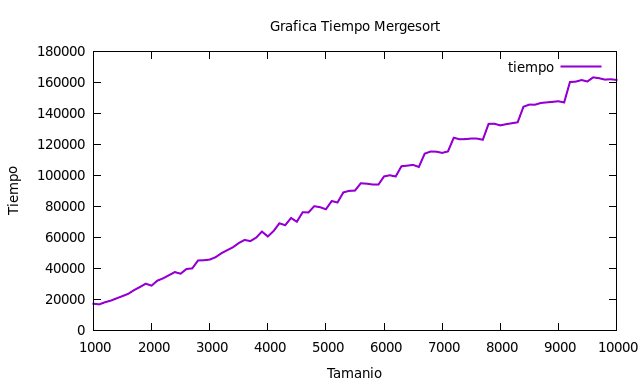
Hacemos make y a continuación valgrind --leak-check=full ./ejercicio5 -num\_min 1 -num\_max 5 -incr 1 -numP 5 -fichSalida ejercicio5.log y obtenemos los resultados esperados:



Gráfica comparando los tiempos mejor peor y medio en OBs para MergeSort, comentarios a la gráfica:



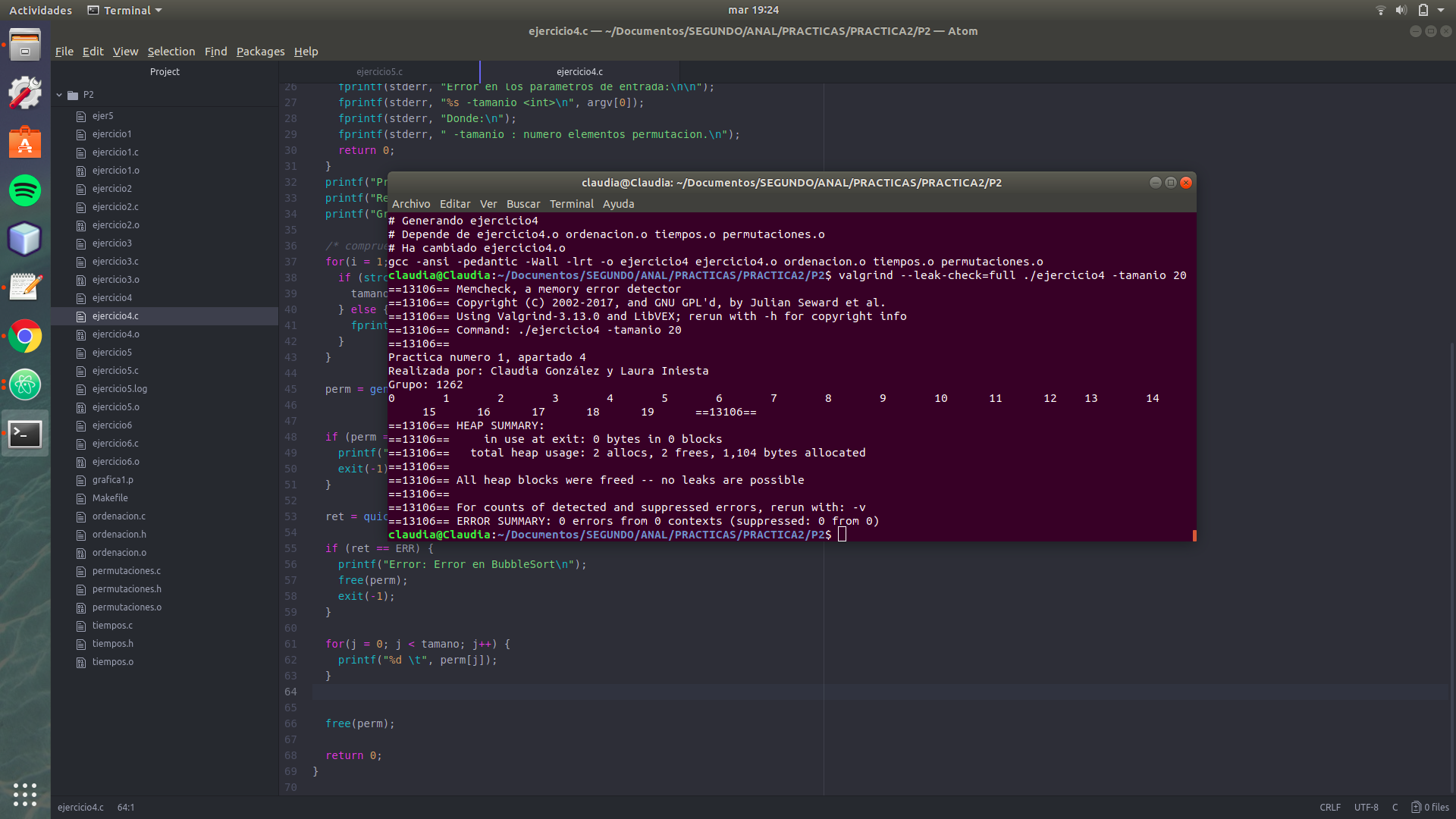
Como podemos observar min max y medio ob coinciden.

Gráfica con el tiempo medio de reloj para MergeSort:

Como podemos observar, a medida que aumenta el tamaño, aumentan los tiempos.

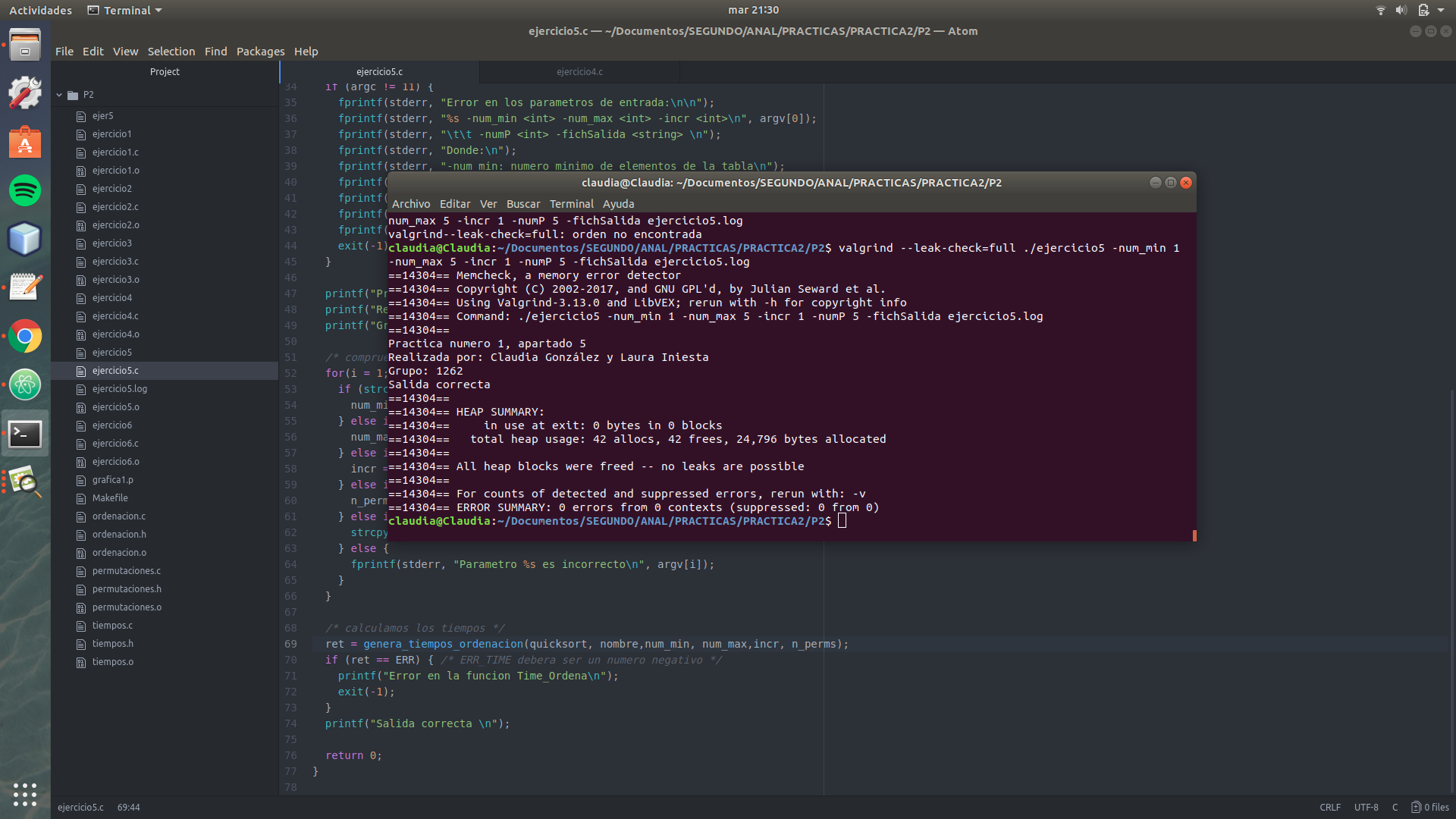
5.3 Apartado 3

Hacemos make en la terminal y posteriormente valgrind --leak-check=full ./ejercicio4 -tamanio 20 y obtenemos la salida esperada:

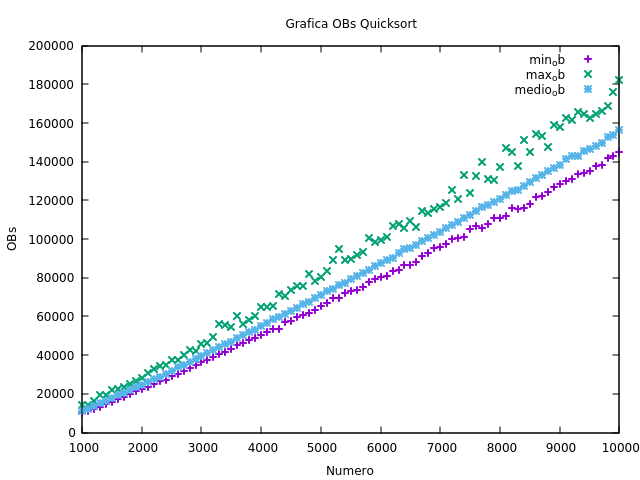


5.4 Apartado 4

Hacemos make y a continuación make ejercicio5\_test (se ha modificado el fichero ejercicio5.c nombre del algoritmo quicksort) y obtenemos la salida esperada:



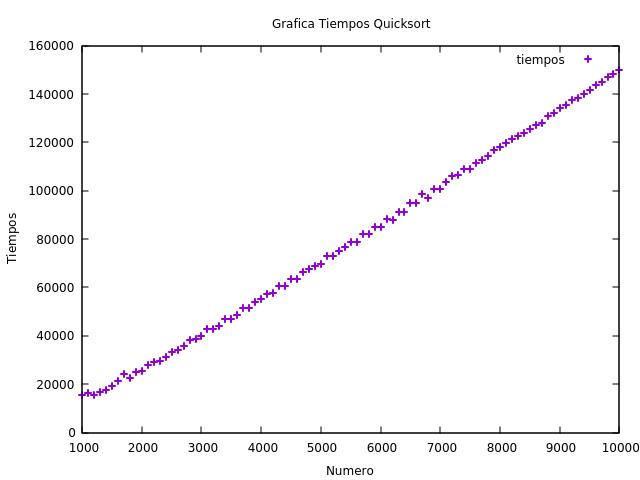
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.



Como podemos observar, las obs van aumentado a medida que el tamaño aumenta, y además, el número máximo de Obs va por encima de la del mínimo, por lo que nuestra gráfica es correcta.

También podemos observar que las ob\_max varian más mientras que el crecimiento de las ob\_medio es muy igual y el de ob\_min tiene variaciones muy pequeñas.

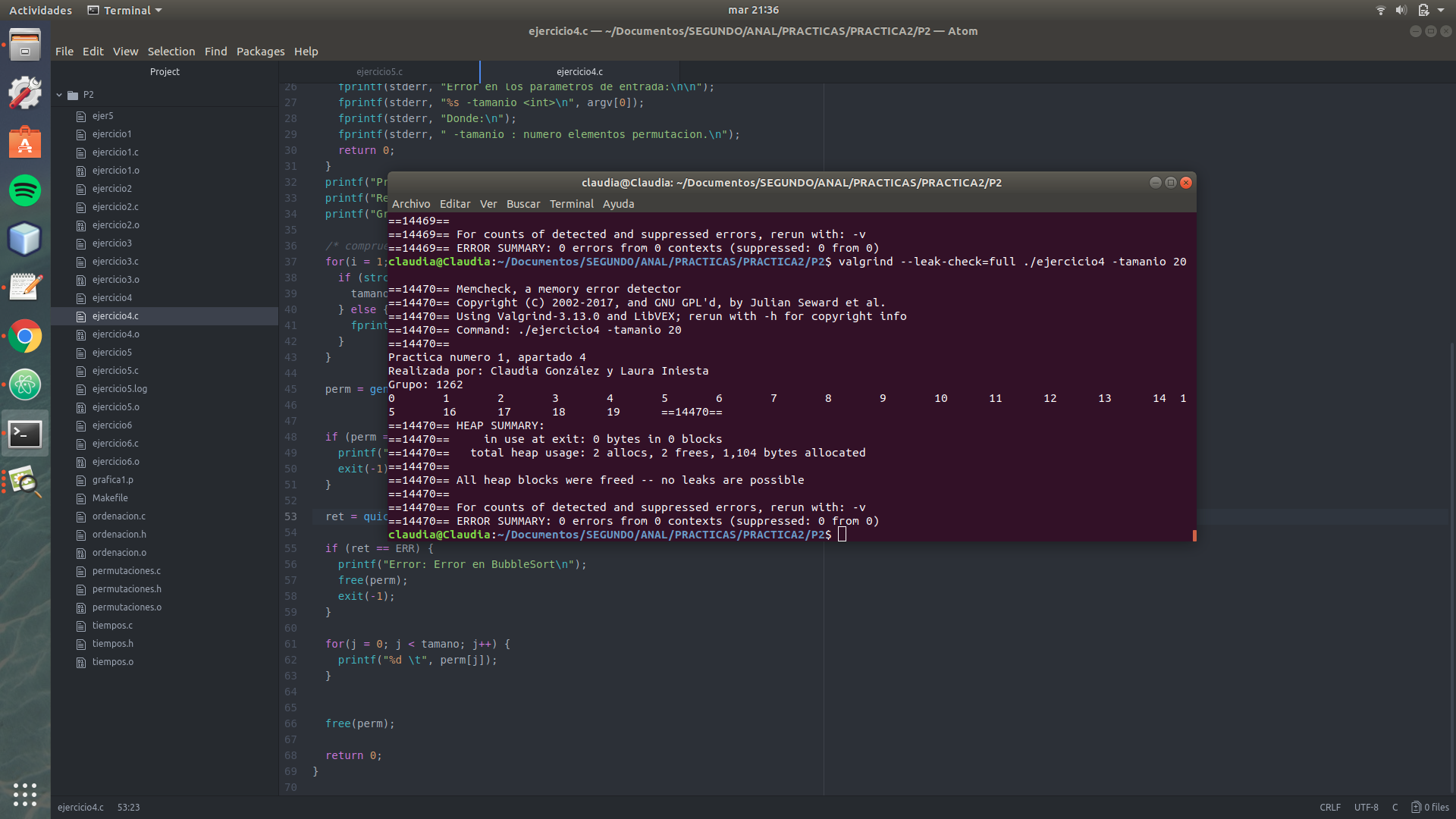
Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.

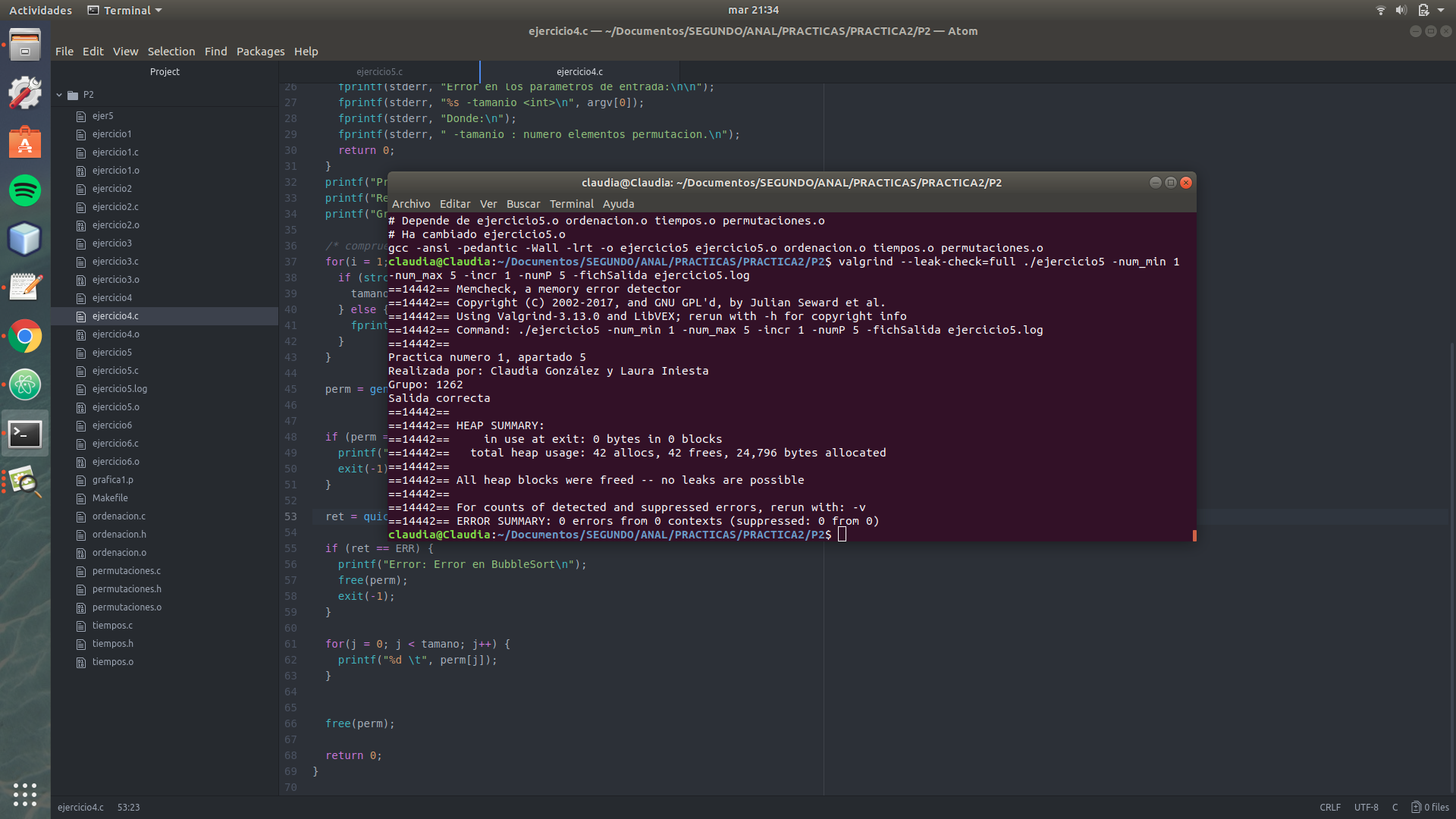


Como podemos observar el tiempo medio aumenta a medida que aumenta el tamaño, haciendo muchos clocks por segundo.

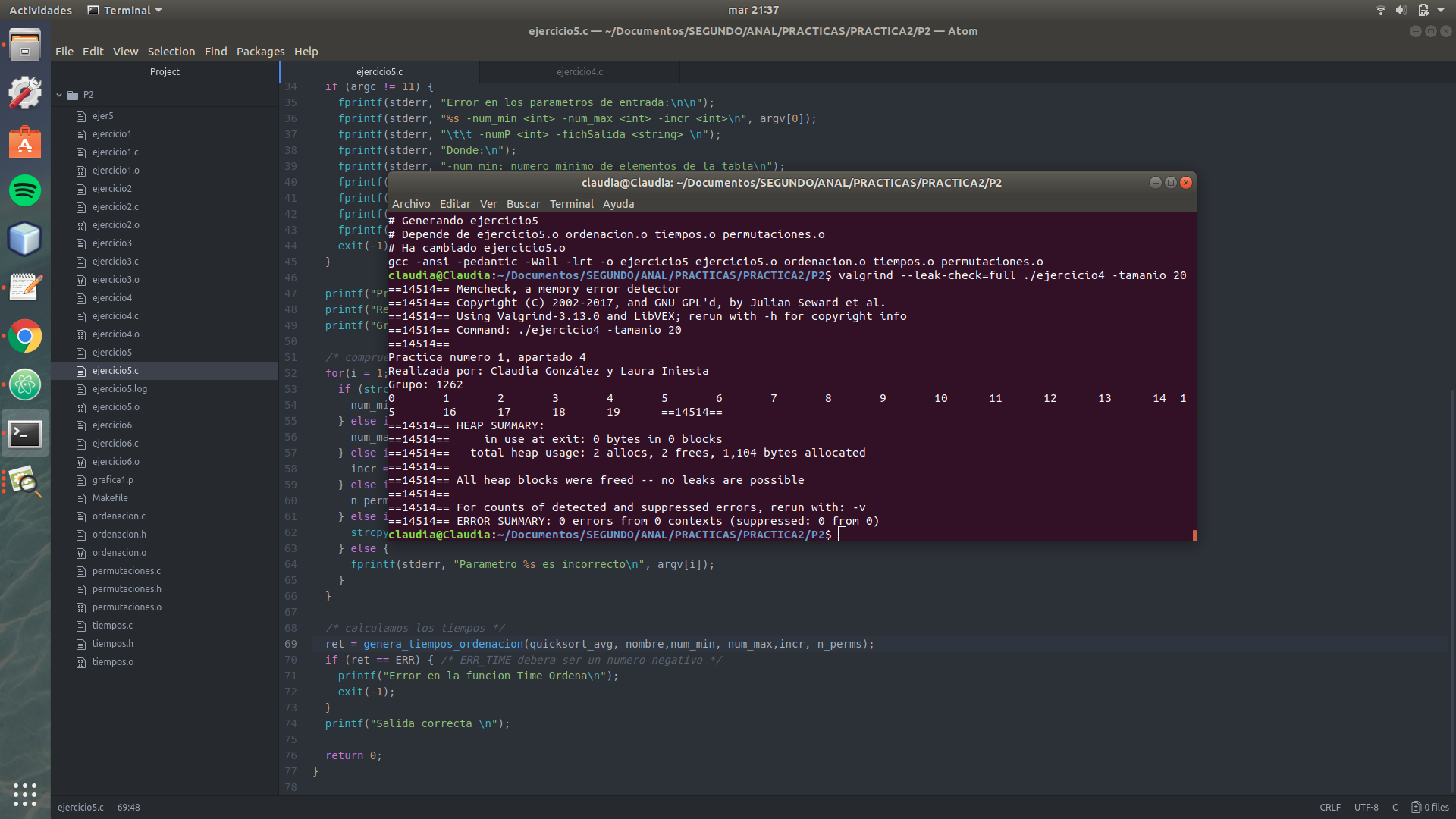
5.5 Apartado 5

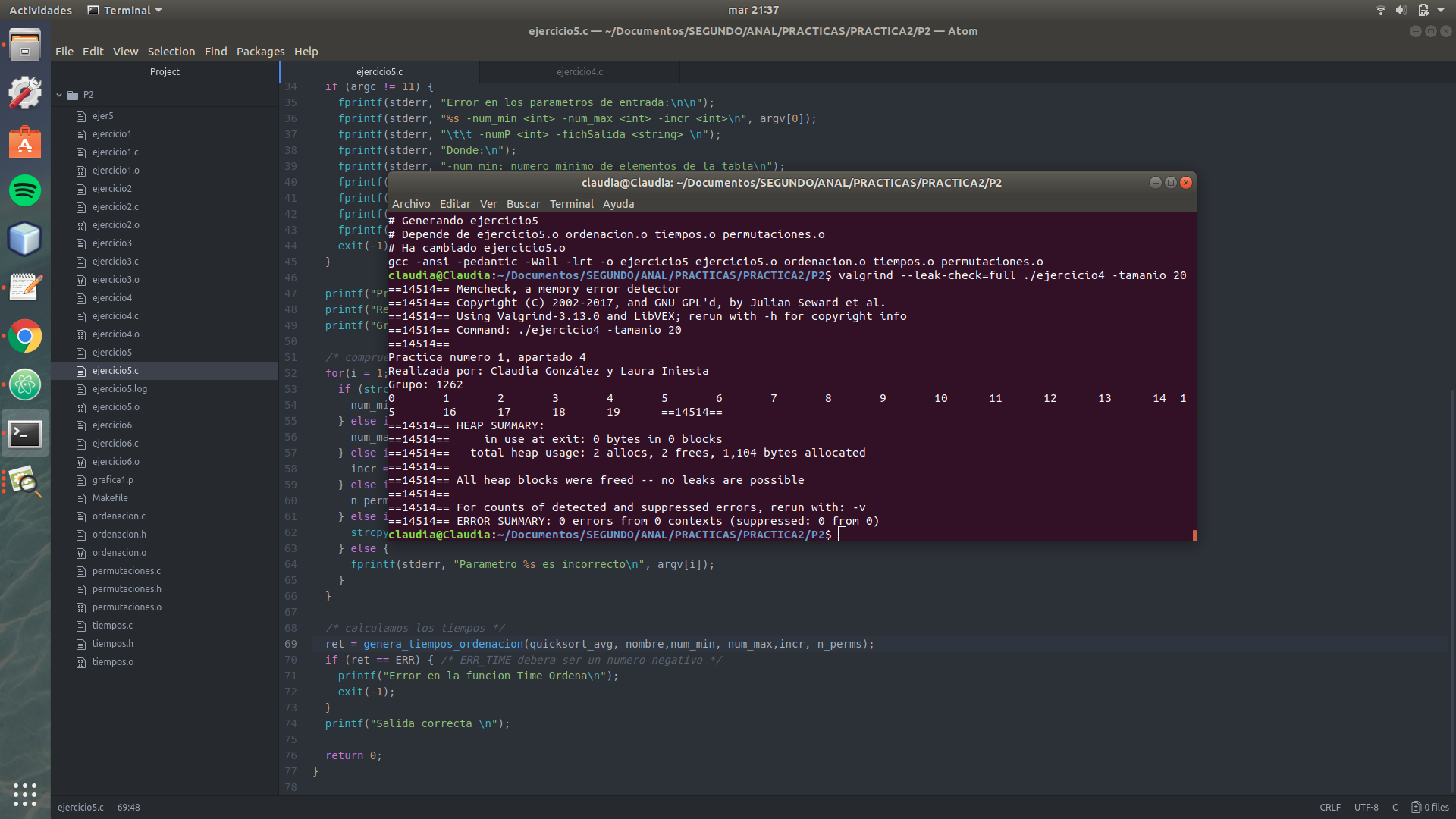
Quicksort\_stat

Ejercicio4:

Ejercicio5: 

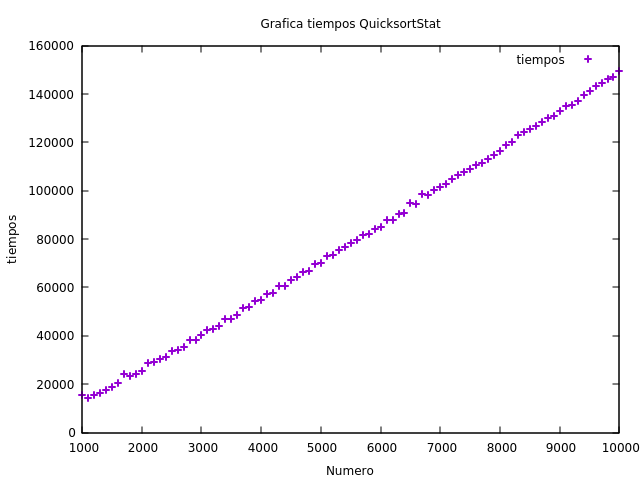
Quicksort\_avg:

Ejercicio4: 

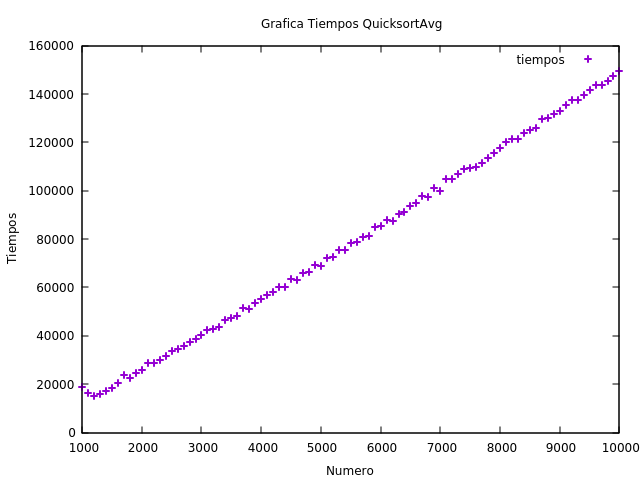
Ejercicio5:

Gráficas con el tiempo medio de reloj comparando los tres pivotes desarrollados.

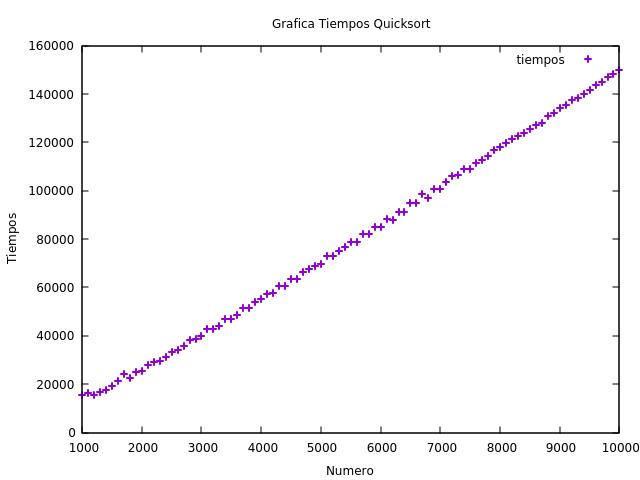
Quicksort stat:



Quicksort avg:



Quicksort:



Como podemos observar en las 3 gráficas, el tiempo medio empleado para hacer los distintos algoritmos es prácticamente igual, pero en el caso del quicksort stat el tiempo al inicio es un poco inferior y si observamos detenidamente la de avg podemos ver que tiene altos y bajos muy pequeños pero en cambio los otros quicksort no los tienen.

**5. Respuesta a las preguntas teóricas.**

**5.1 Compara el rendimiento empı́rico de los algoritmos con el caso medio teórico en cada caso. Si las trazas de las gráficas del rendimiento son muy picudas razonad porqué ocurre esto.**

El caso medio de MergeSort teórico cumple la fórmula AMS(N)=Ω(Nlg(N)) por lo que vamos a calcular el número de obs teórico para el caso medio de permutaciones con distintos tamaños.

AMS(1000)= 996,5 apróximadamente.

AMS(5000)= 6143,8 apróximadamente

AMS(10000)= 13287,7 apróximadamente

Como podemos observar los valores obtenidos se asemejan a nuestra gráfica.

El caso medio de QuickSort teórico cumple la fórmula AQS (N) = 2N log( N)+O(N) por lo que vamos a calcular el número de obs teórico para el caso medio de permutaciones con distintos tamaños.

AQS(1000)= 19.931,52 + O(N) apróximadamente

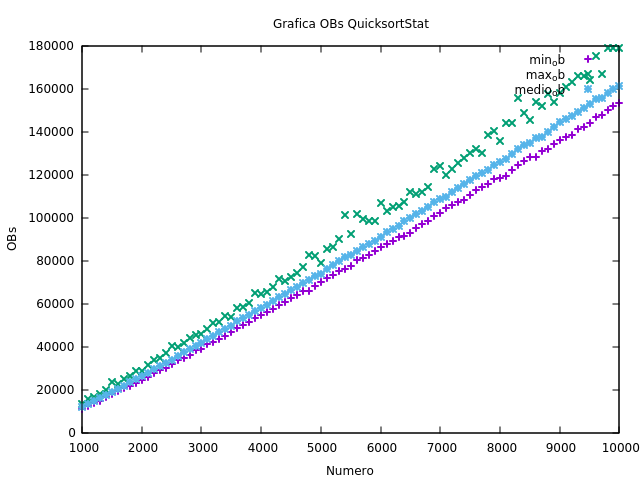
AQS(5000)= 122.877,1 + O(N) apróximadamente

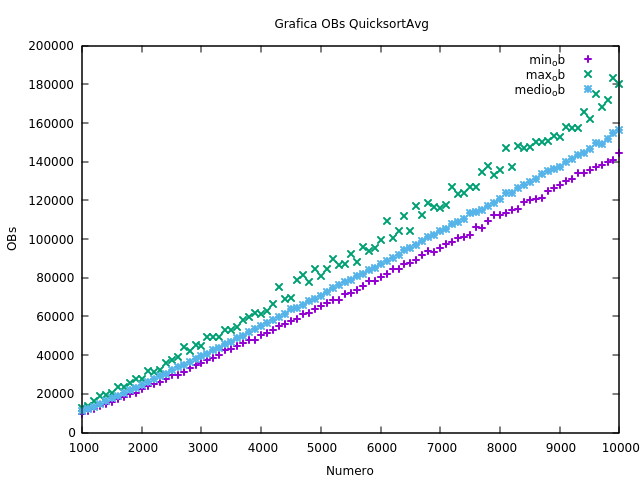
AQS(10000)= 165.753,6 + O(N) apróximadamente

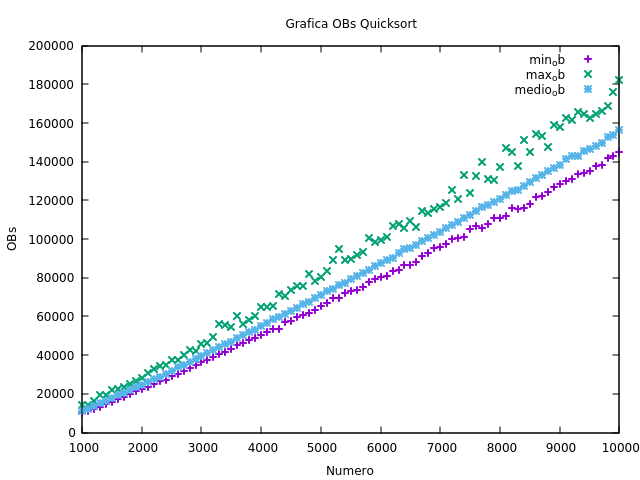
Como podemos observar los valores obtenidos se asemejan a los de nuestra gráfica aunque los calculados teóricamente son superiores que los obtenidos.

En nuestra gráfica, las trazas no son muy picudas, salvo para el caso máximo ya que depende de lo desordenada que esté la tabla y en cada permutación generada el desorden es distinto.

**5.2 Razonad el resultado obtenido al comparar las versiones de quicksort con los diferentes pivotes tanto si se obtienen diferencias apreciables como si no.**







Las tres nos dan gráficas con diferencias casi imperceptibles en cuanto a tiempos de ordenación pero las obs si que se diferencian debido a que dependiendo del pivote que se elija la ordenación será más o menos óptima.

Si observamos las gráficas podemos ver que Avg realiza más comparaciones de clave que stat por lo que stat es más óptimo que avg.

En el caso de quicksort normal podemos ver que las diferencias con avg son mínimas pero que realiza más obs que stat.

En conclusión quicksort\_stat es mucho mejor que quicksort y quicksort\_avg, eso es debido a que quicksort\_stat usa medio\_stat, una función que elige el pivote que más le conviene a la tabla en cada ocasión teniendo en cuenta el valor de los elementos primero, último y medio de la tabla.

Hemos colocado aquí las gráficas para que sea más sencillo de ver al estar todas juntas.

**5.3 ¿Cuáles son los casos mejor y peor para cada uno de los algoritmos? ¿Qué habrá que modificar en la práctica para calcular estrictamente cada uno de los casos (también el caso medio)?**

Mergesort:

Wms(N) ≤ Nlg(N)+O(N) (Caso peor).

Bms(N) ≥ (1/2)Nlg(N) (Caso mejor).

Quicksort:

Wqs(N) ≤ N² /2 - N/2 (Caso peor).

Bqs (N) ≥ Nlg(N) (Caso mejor).

Para poder probar el caso mejor y el caso peor deberíamos introducir permutaciones ordenadas 12345 o completamente desordenadas, inversas 54321 para demostrar que en el mejor de los casos, cuando la tabla ya está ordenada, el algoritmo realizaría Bms/qs(N) comparaciones de clave como mínimo y como máximo al estar completamente desordenada serían Wms/qs(N) cdcs.

Para el caso medio podríamos introducir una tabla medio ordenada 123654 para demostrar que la fórmula se cumple.

5.4 **¿Cuál de los dos algoritmos estudiados es más eficiente empı́ricamente? Compara este resultado con la predicción teórica. ¿Cual(es) de los algoritmos es/son más eficientes desde el punto de vista de la gestión de memoria? Razona este resultado.**

Como podemos observar en las gráficas la cantidad de OBs realizados por quicksort es mucho más alta que las realizadas por mergesort.

Si miramos la pregunta 5.3 podemos ver que las fórmula para el caso mejor de MS va a dar resultados más pequeños que la de QS ya que es la misma pero en MS dividida entre 2, lo mismo sucede con el caso peor, el crecimiento de la función para el caso peor de MS es más lento que el de QS.

Debido a esto podemos concluir que nos coincide la teoría con la práctica.

En cuanto a memoria quicksort es mucho más eficiente ya que hace menos uso de memoria que mergesort, por ejemplo para ejecutar el mismo enunciado del ejercicio 5 Quicksort hace 45 allocs mientras que Mergesort hace 120.

**6. Conclusiones finales.**

En esta práctica hemos aprendido a implementar los algoritmos de ordenación quicksort y mergesort asentando nuestros conocimientos teóricos sobre ellos y viendo su correcto funcionamiento.

Al realizar las gráficas hemos podido comprobar que en efecto los algoritmos con mecánica “Divide y vencerás” son algoritmos más eficientes que el algoritmo local insertsort usado en la práctica 1.