Análisis de Algoritmos 2018/2019

Práctica 2

Guillermo López Rodríguez

Javier Fernández de Alarcón Gervás

1261

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

**1. Introducción.**

En esta práctica hay dos objetivos principales, implementar los algoritmos de ordenación MergeSort y QuickSort. Se compararán también los diferentes mecanismos de elección de pivote.

**2. Objetivos**

2.1 Apartado 1

Se busca implementar el algoritmo MergeSort usando como rutina principal la función llamada mergesort y para función de combinación merge. Se modifica el ejercicio4.c para probar que funcione correctamente.

2.2 Apartado 2

Se modifica el el ejercicio5.c cambiando de el primer argumento de la función genera\_tiempos\_ordenación por mergesort.

2.3 Apartado 3

Se implementa el algoritmo QuickSort con la rutina principal quicksort en ordenación.c. Se utilizan dos funciones de apoyo, partir y medio, donde partir es la función que realiza swaps con el pivote definido y medio la función que establece que el pivote sea el primer índice. También habría que modificar el ejercicio4.c para que funcione mediante el algoritmo QuickSort.

2.4 Apartado 4

Al igual que en el apartado 2, se modifica el ejercicio5.c para que se pueda usar con el algoritmo QuickSort.

2.5 Apartado 5

El objetivo del apartado 5 es crear diferentes funciones medio para elegir distinto pivote que en el apartado 3. Mediante la función medio\_avg se elige como pivote el valor medio usando la siguiente expresión: (ip + iu)/2. El otro método de elección de pivote se realiza con la función medio\_stat, donde se elige el valor intermedio de una tabla. Por consiguiente, al usar distintos métodos para elegir pivote, se utilizan distintas funciones para devolver los pivotes, partir\_avg y partir\_medio respectivamente.

**3. Herramientas y metodología**

En esta Práctica hemos utilizado tanto Linux como MacOS. En cuanto a herramientas,

un miembro del grupo usó Visual Studio y el otro Sublime Text, evidentemente para comprobar las fugas de memoria se ha utilizado Valgrind. En el caso de las gráficas/histograma se ha hecho uso de Gnuplot y Excel.

3.1 Apartado 1

En este apartado implementamos el algoritmo MergeSort donde se usa la recursión como herramienta principal.

3.2 Apartado 2

Usamos los datos obtenidos del ejercicio5.c, para obtener una tabla de Excel y representar así las gráficas correspondientes.

3.3 Apartado 3

Al igual que en el primer apartado, se implementa QuickSort usando recursión.

3.4 Apartado 4

También de forma idéntica que en el apartado 2, se obtienen los datos del ejercicio5.c a través de la terminal, y se generan tablas para comparar gráficamente los valores.

3.5 Apartado 5

En este apartado se obtienen los diferentes pivotes mediante las distintas expresiones: medio = ip, medio\_avg = (ip + iu)/2 y medio\_stat = intermedio(ip, iu, medio\_avg). En este último se escoge el valor intermedio entre el primer valor, al último y el medio.

**4. Código fuente**

Aquí ponéis el código fuente **exclusivamente de las rutinas que habéis desarrollado vosotros** en cada apartado.

4.1 Apartado 1

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

int m,ob=0;

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

if(ip==iu) return OK;

else{

m=(ip+iu)/2;

ob+=mergesort(tabla,ip,m);

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

ob+=merge(tabla,ip,iu,m);

}

return ob;

}

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

int \*tablaAux=NULL;

int i,j,k,ob;

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

tablaAux=(int\*)calloc(iu+1,sizeof(int));

if(tablaAux==NULL)return ERR;

for(i=ip,j=imedio+1,k=ip,ob=0;(i<=imedio) && (j<=iu);k++){

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

tablaAux[k]=tabla[i];

i++;

}

else{

tablaAux[k]=tabla[j];

j++;

}

ob++;

}

if(i>imedio){

while(j<=iu){

tablaAux[k]=tabla[j];

j++;

k++;

}

}

else if(j>iu){

while(i<=imedio){

tablaAux[k]=tabla[i];

i++;

k++;

}

}

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

tabla[i]=tablaAux[i];

}

free(tablaAux);

return ob;

}

4.2 Apartado 2

No se implementa ninguna función

4.3 Apartado 3

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

int \*m=NULL;

int OB=0;

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

if(ip==iu) return OK;

m=(int\*)malloc(sizeof(int));

OB+=partir(tabla,ip,iu,m);

if(ip<\*m-1){

OB+=quicksort(tabla,ip,\*m-1);

}

if(\*m+1<iu){

OB+=quicksort(tabla,\*m+1,iu);

}

free(m);

return OB;

}

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

int OB=0,k,i;

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

OB+=medio(tabla,ip,iu,pos);

k=tabla[\*pos];

swap(&tabla[ip],&tabla[\*pos]);

\*pos=ip;

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

OB++;

if(tabla[i]<k){

(\*pos)++;

swap(&tabla[i],&tabla[\*pos]);

}

}

swap(&tabla[ip],&tabla[\*pos]);

return OB;

}

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

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

if(ip>iu)return ERR;

\*pos=ip;

return 0;

}

4.4 Apartado 4

No se implementa ninguna funcion

4.5 Apartado 5

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

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

if(ip>iu)return ERR;

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

return 0;

}

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

int x,y,z;

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

if(ip>iu)return ERR;

medio\_avg(tabla, ip,iu,pos);

x=tabla[ip];

y=tabla[iu];

z=tabla[\*pos];

if(x>y){

if(y>z){

\*pos=iu;

}

if(x<z){

\*pos=ip;

}

}

return 3;

}

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

int OB=0,k,i;

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

OB+=medio\_avg(tabla,ip,iu,pos);

k=tabla[\*pos];

swap(&tabla[ip],&tabla[\*pos]);

\*pos=ip;

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

OB++;

if(tabla[i]<k){

(\*pos)++;

swap(&tabla[i],&tabla[\*pos]);

}

}

swap(&tabla[ip],&tabla[\*pos]);

return OB;

}

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

int OB=0,k,i;

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

OB+=medio\_stat(tabla,ip,iu,pos);

k=tabla[\*pos];

swap(&tabla[ip],&tabla[\*pos]);

\*pos=ip;

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

OB++;

if(tabla[i]<k){

(\*pos)++;

swap(&tabla[i],&tabla[\*pos]);

}

}

swap(&tabla[ip],&tabla[\*pos]);

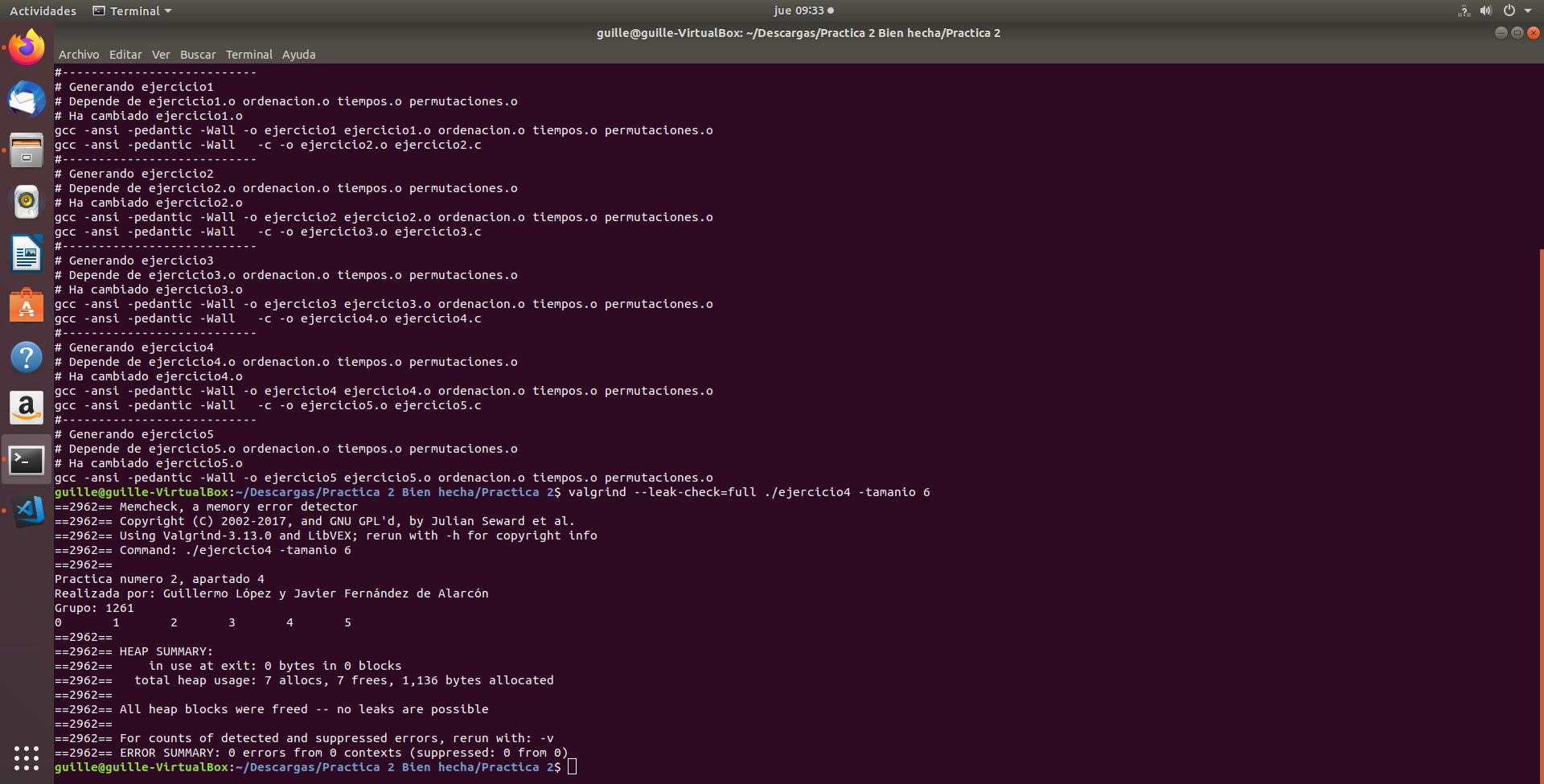
return OB;

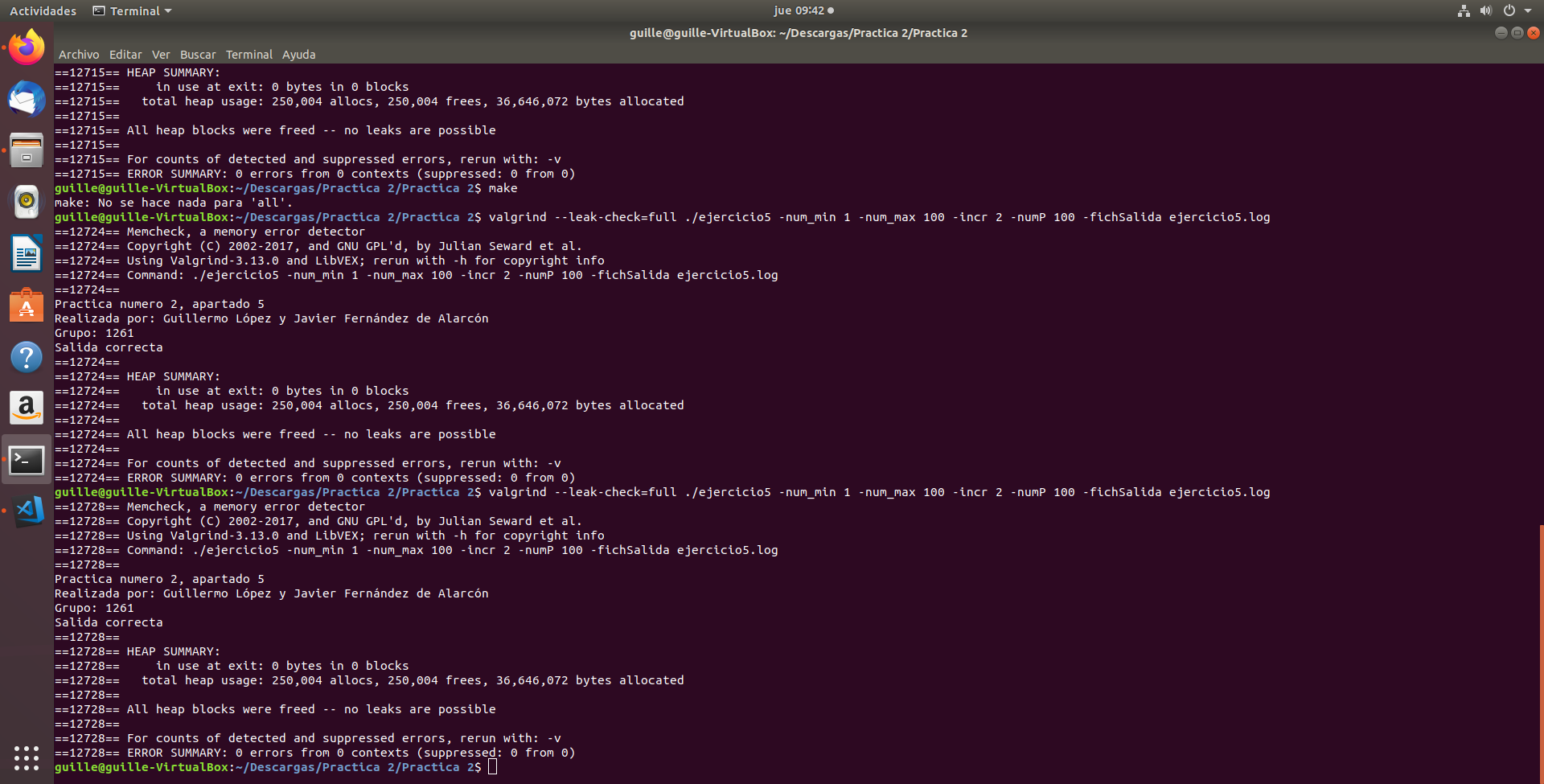
}

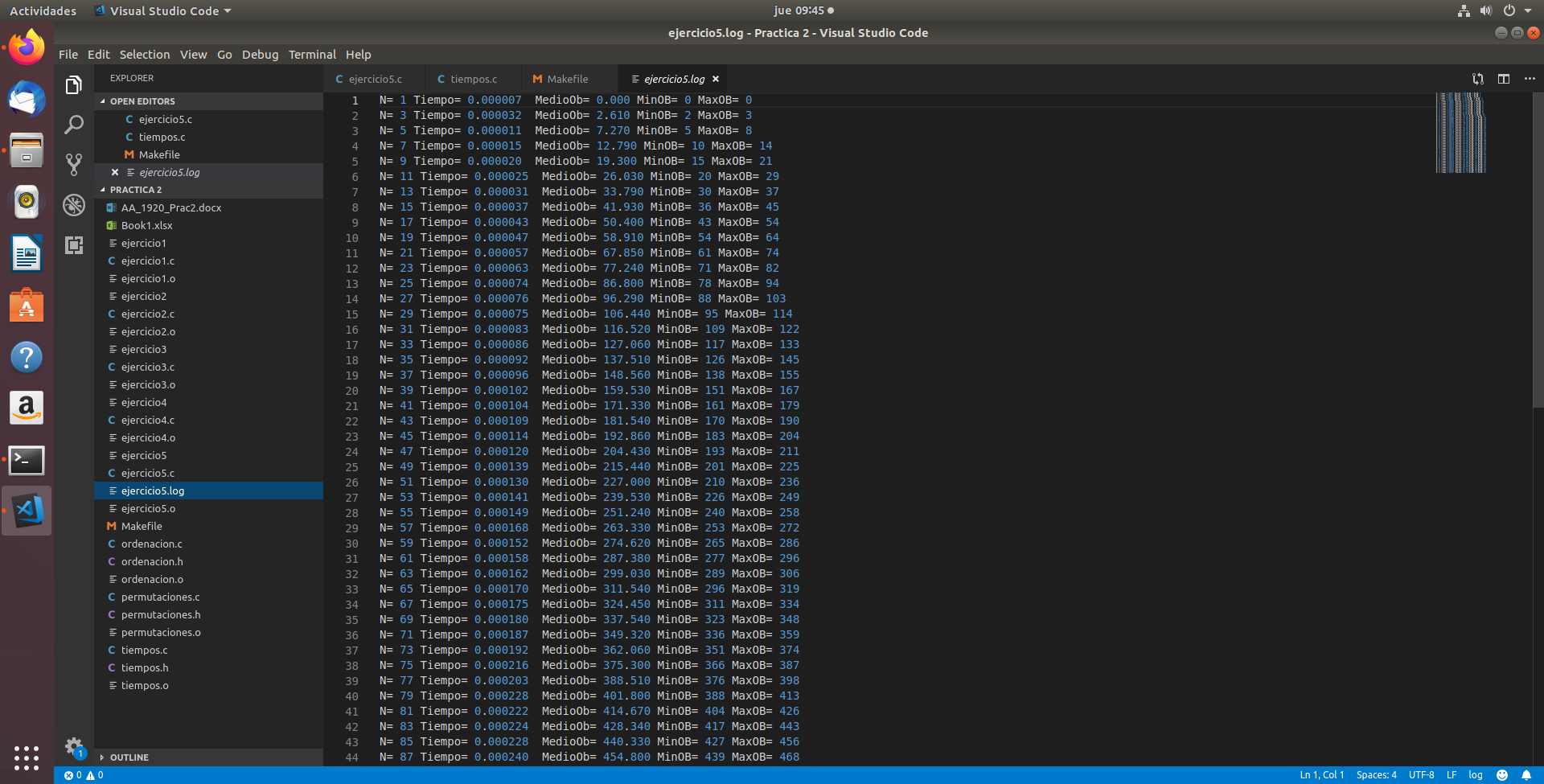
**5. Resultados, Gráficas**

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

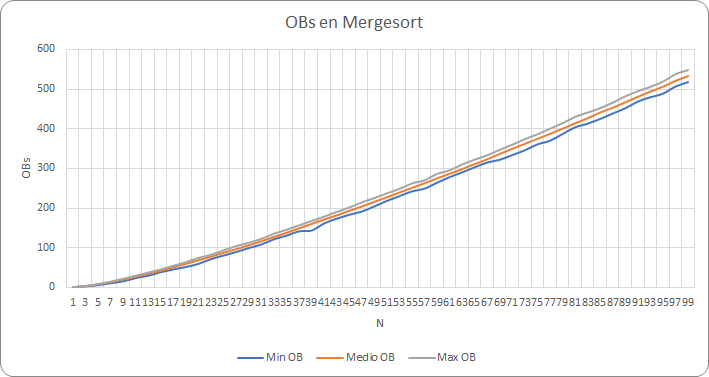
5.1 Apartado 1



5.2 Apartado 2 

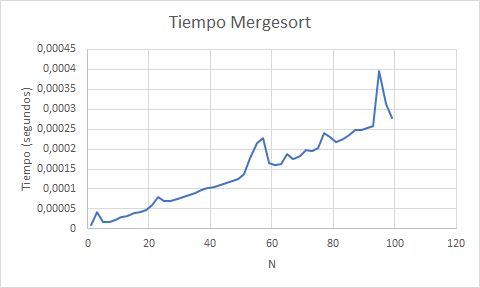
Resultados del apartado 2.

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

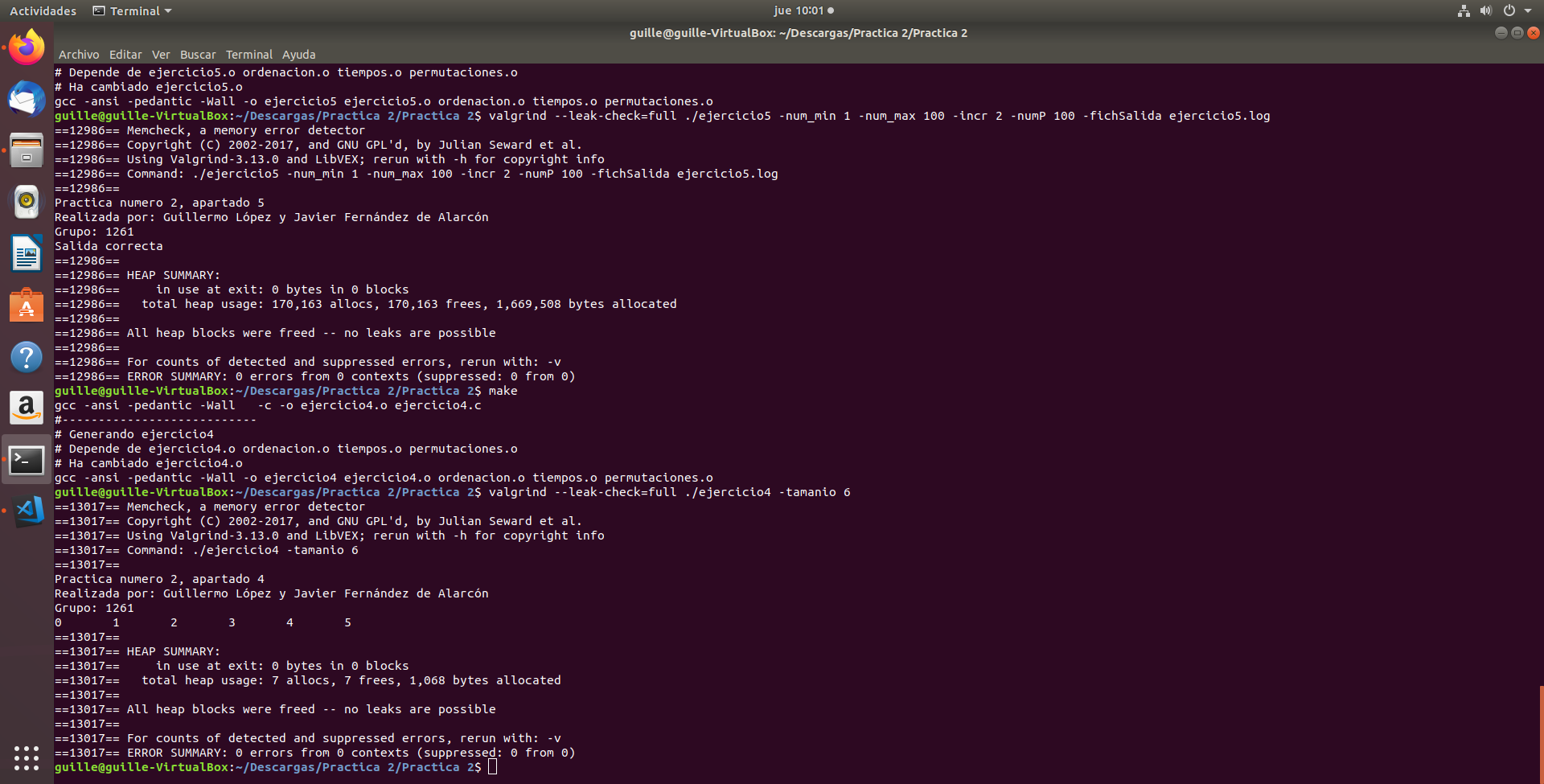


Podemos ver en la gráfica que los tres casos, mínimo, medio y mejor se mantienen bastante parejos y la diferencia es ínfima.

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

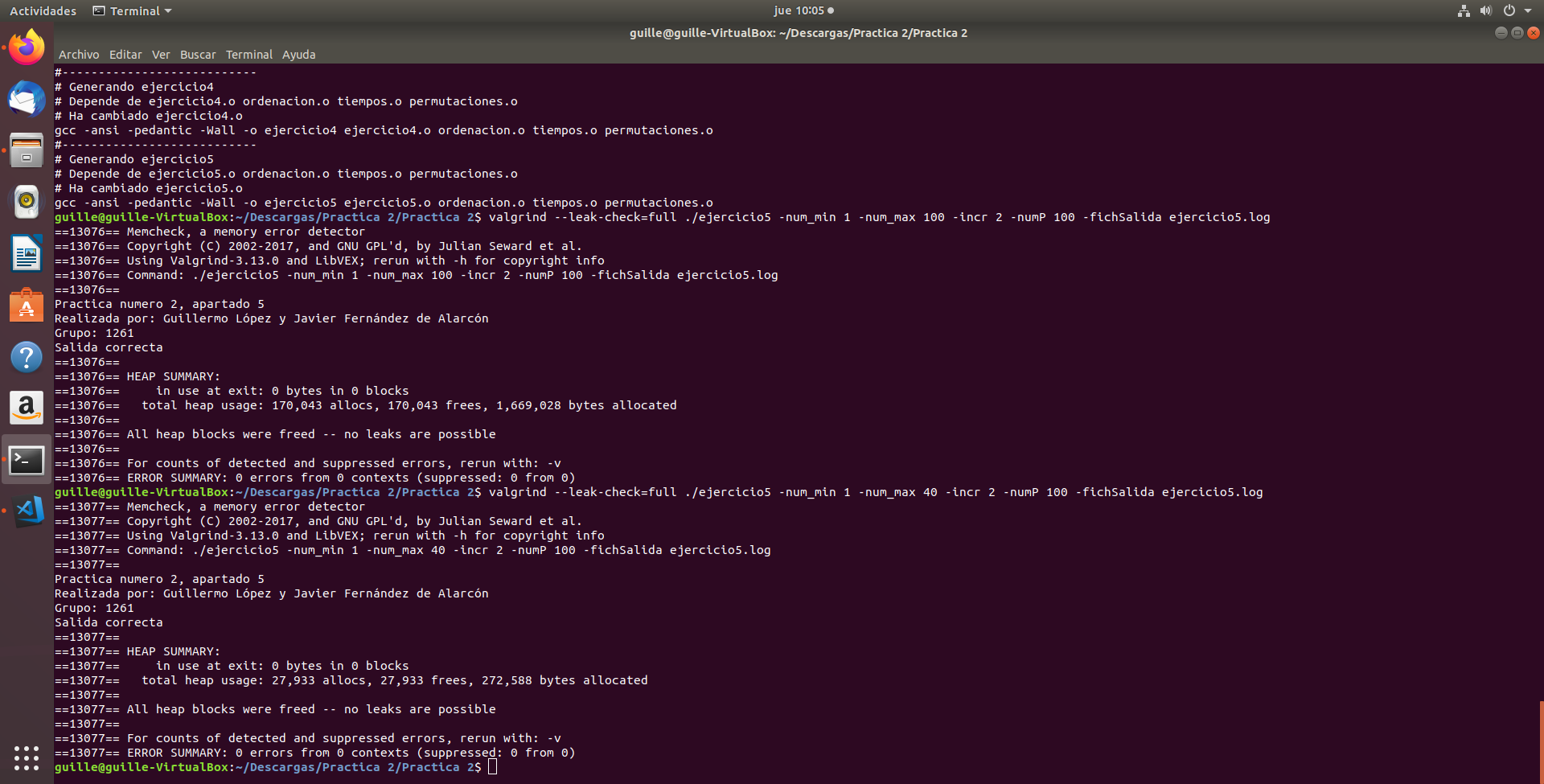


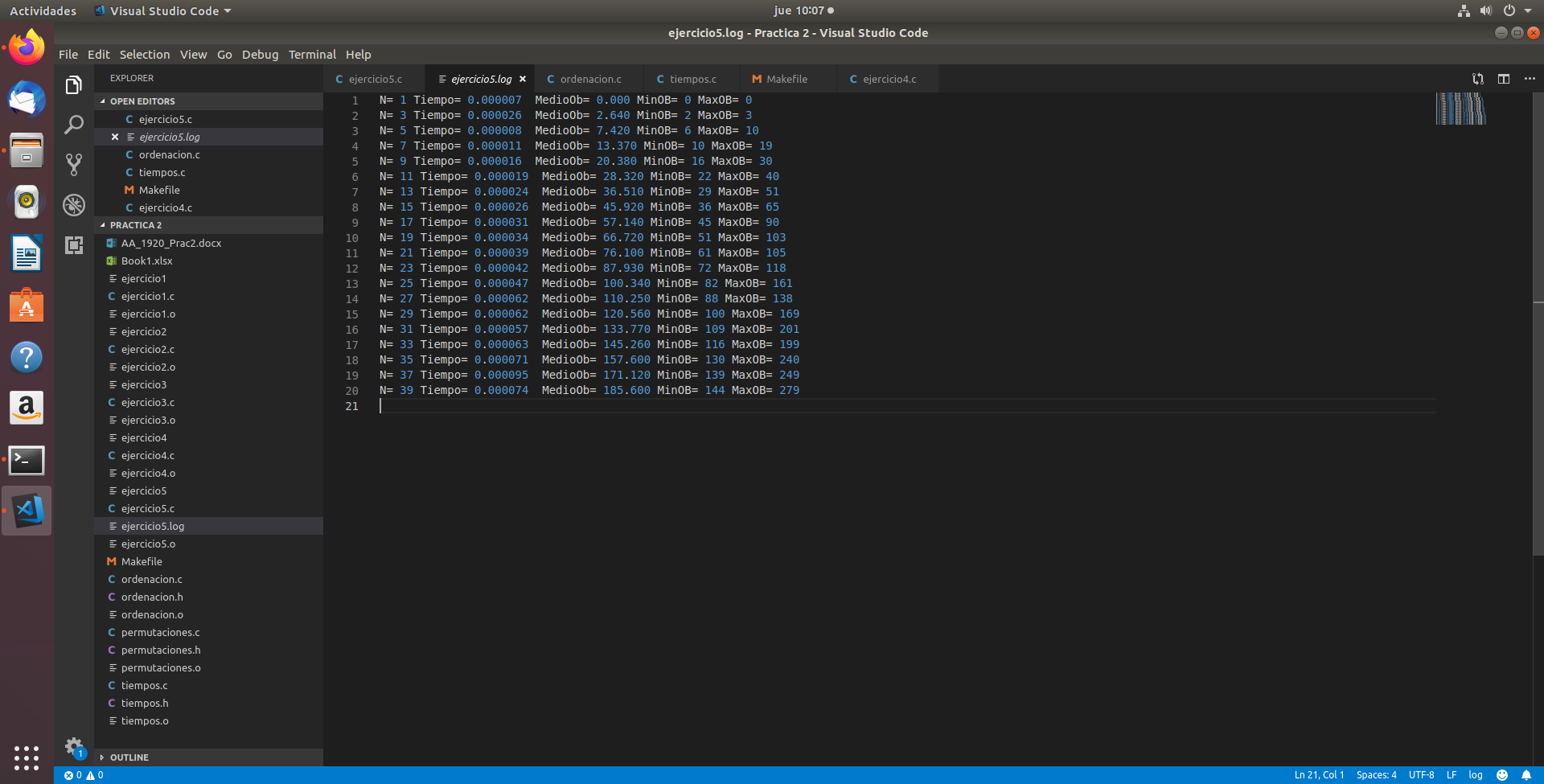
5.3 Apartado 3

Resultados del apartado 3.

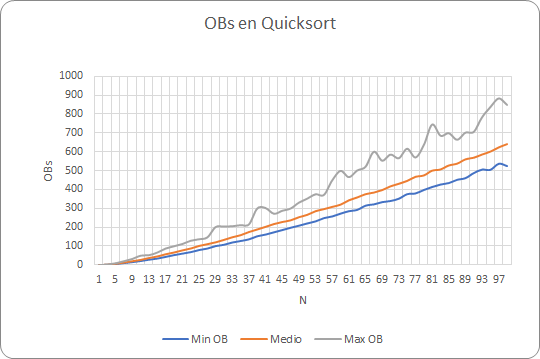
5.4 Apartado 4

Resultados del apartado 4.

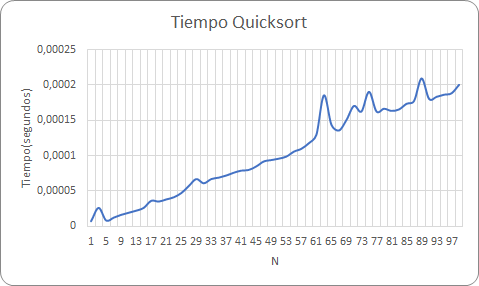




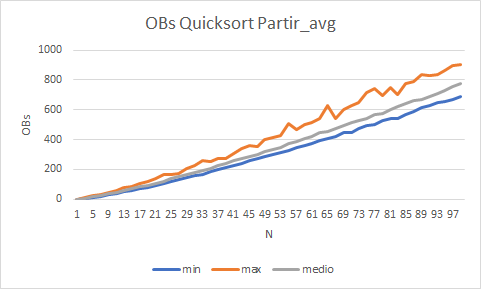
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.

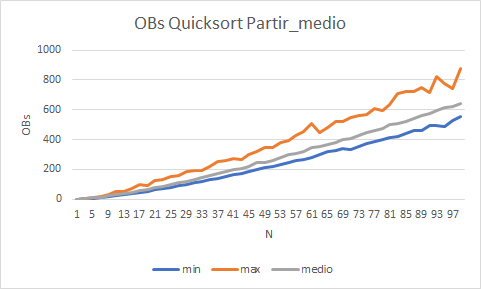


Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.

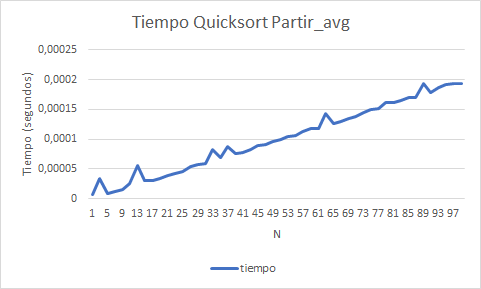


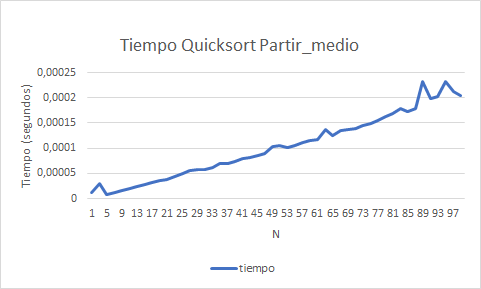
5.5 Apartado 5





Comparando los resultados, llegamos a la conclusión que el partir\_medio es más eficaz puesto que el número de OBs tanto en mínimo, máximo y medio son menores.





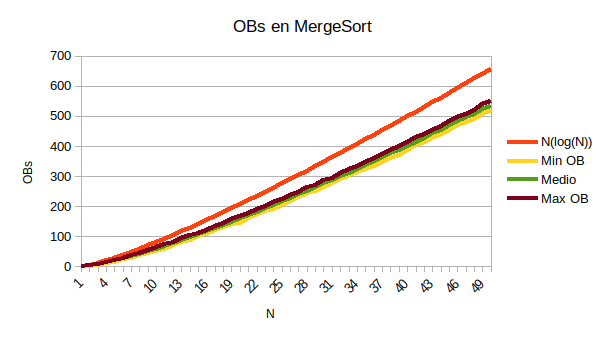
En cuanto al tiempo apreciamos un ligero aumento de tiempo al usar partir\_medio pero es ínfimo por lo que ambas opciones de pivote son muy parejas en este sentido sin tener en cuenta los altibajos que se producen en determinadas zonas.

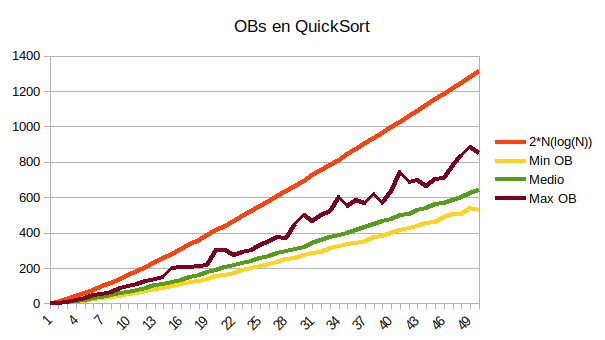
**5. Respuesta a las preguntas teóricas.**

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

5.1 Pregunta 1

El caso medio teórico del algoritmo Mergesort es Θ (Nlog(N)) y del algoritmo Quicksort es 2N log(N).





Vemos como en el algoritmo Mergesort está parejo con los resultados, sin embargo Quicksort sale ligeramente superior el caso medio teorico.

Las gráficas son muy picudas porque trabajamos con permutaciones aleatorias por lo que habrá casos en los que para el algoritmo será más favorable la ordenación y otros que se tendrán que emplear más OBs para ello.

5.2 Pregunta 2

Al comparar los tres pivotes nos damos cuenta que tanto el pivote en la primera posición como el pivote medio dan rendimientos muy parecidos,casi identicos. Sin embargo el algoritmo quicksort con el pivote tomando el valor medio entre el primero, medio y último (medio\_stat) realiza más OBs aunque el tiempo es muy parecido a los otros.

5.3 Pregunta 3

En el algoritmo Mergesort , el caso peor es Nlog(N) + O(N), el caso medio Θ (Nlog(N)) y el caso mejor es Nlog(N).

En el algoritmo Quicksort, el caso peor es /2 -N/2, el caso medio es 2N log(N) + O(N) y el caso mejor es O Nlog(N).

Para calcular estrictamente cada caso tendríamos que insertar como permutación una en la que, en el caso peor tuviera que realizar el máximo número de OBs. En el caso mejor tendríamos que insertar una permutación en la que tendría que realizar el número mínimo de comparaciones y en el caso medio

5.4 Pregunta 4

Empíricamente como vemos en los resultados de ambos algoritmos, Mergesort realiza menos comparaciones de clave que Insertsort por lo que es más eficiente, al igual que vimos en teoría.

En cuanto a la gestión de memoria, para el mismo número de permutaciones, en concreto desde 1 hasta 40 con incremento de 2 y número de permutaciones 100. El algoritmo Mergesort se realizan 40.004 allocs mientras que el algoritmo Quicksort realiza 28.021 allocs. Por lo que el algoritmo Quicksort gestiona mejor el uso de la memoria.

**6. Conclusiones finales.**

En esta práctica hemos implementado dos algoritmos estudiados en clase como son el Mergesort y Quicksort. Una vez implementados correctamente y almacenar la información que obtenemos de de los algoritmos llegamos a la conclusión de que Mergesort nos da resultados esperados con la teoría. Sin embargo nuestro algoritmo de ordenación Quicksort nos da resultados más diferentes que en el teórico, pero no podemos esperar resultados perfectos comparándolo con la teoría ya que utilizamos permutaciones aleatorias.