Análisis de Algoritmos 2015/2016

Práctica 2

Simon Valcarcel, Lucia Asencio,  1202.

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

**1. Introducción.**

En esta práctica estudiaremos el comportamiento empírico de funciones de ordenación mediante recursión. A pesar de no tener los mejores tiempos, poseen una elegancia que las hace atractivas de implementar. Se basan en la idea fundamental de *“divide and conquer”*, divide y vencerás, que es tan aplicable a la teoría bélica como a la programación: Dividiendo un problema complejo sucesivas veces hasta que logramos obtener un problema que sí sabemos resolver. En esta práctica nos centraremos en MergeSort y QuickSort.

**2. Objetivos**

2.1 Apartado 1

En este apartado implementaremos el algoritmo de ordenación MergeSort. La idea detrás del algoritmo es la siguiente: hacer llamadas recursivas a si misma hasta que cada llamada haga referencia a un solo elemento de la tabla. Después, estos elementos se van ordenando sucesivamente mediante la función merge(), que nos crea una subtabla ordenada hasta ese punto. Posteriores llamadas a merge() cogen dos subtablas ordenadas, y aprovechando este hecho, devuelven otra subtabla ordenada con los elementos de las anteriores. Este procedimiento se repite hasta tener toda la tabla ordenada.

2.2 Apartado 2

Una vez implementado el programa MergeSort, estudiaremos su comportamiento empírico, representándolo con GNUplot.

2.3 Apartado 3

Seguimos la práctica implementando el algoritmo quickSort. Este algoritmo comienza eligiendo un elemtno de la tabla, el “pivote”. La elección de este elemento se realiza mediante la función Medio(). Posteriormente, la función partir() desplaza todos los elementos menores que el pivote a la izquierda de este y los mayores a la derecha. Luego, la función quickSort se llama recursivamente sobre ambas subtablas (la menor y la mayor que el pivote), eligiendo para cada subtabla un nuevo pivote. El algoritmo habrá ordenado la tabla cuando cada elemento de esta haya sido pivote, y por lo tanto, está en la posición adecuada.

2.4 Apartado 4

Ahora que ya tenemos implementado QuickSort, procedemos a obtener empíricamente su rendimiento y representarlo gráficamente.

2.5 Apartado 5

Finalmente, modificaremos la rutina medio() para que nos de diferentes pivotes según el argumento que se introduzca.

**3. Herramientas y metodología**

Aquí ponéis qué entorno de desarrollo (Windows, Linux, MacOS) y herramientas habéis utilizado (Netbeans, Eclipse, gcc, Valgrind, Gnuplot, Sort, uniq, etc) y qué metodologías de desarrollo y soluciones al problema planteado habéis empleado en cada apartado. Así como las pruebas que habéis realizado a los programas desarrollados.

3.1 Apartado 1

Obtuvimos el pseudocódigo de las diapositivas y desarrollamos el código en Linux, usando Valgrind para asegurarnos de que no hay pérdidas de memoria. La realización del ejercicio no supuso mayor problema. Continuamos en Linux y usando ejercicio4.c de la práctica anterior, nos cercioramos de que el código ordena adecuadamente.

3.2 Apartado 2

Utilizando GNUplot, obtuvimos las gráficas de rendimiento mediante los datos que produce ejercico5.c

3.3 Apartado 3

Desarrollamos el código en Linux, y lo finalizamos en netbeans para debuggearlo. Posteriormente, usamos nuevamente ejercicio4.c para asegurarnos que esta función también ordena como debería.

3.4 Apartado 4

Al igual que en el apartado 2, obtuvimos datos mediante ejercicio5.c y luego los representamos con GNUplot.

3.5 Apartado 5

Fuimos ampliando progresivamente la rutina medio() para ir devolviendo diferentes pivotes. Usamos el entorno Netbeans para comprobar que la función devolvía lo esperado. Añadimos un parámetro de entrada adicional a la función quicksort() para indicar qué pivote coger

**4. Código fuente**

4.1 Apartado 1

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: Mergesort Fecha: 10/16 \*/

/\* Entrada: \*/

/\* tabla de enteros a ordenar (descendente) entre \*/

/\* los indices ip (inferior) e iu (superior) \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

int mitad;

int ob=0;

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

return ERR;

mitad = (ip+iu)/2;

if(ip == iu)

return OK;

ob +=mergesort(tabla, ip, mitad, pivote\_inutil);

ob+=mergesort(tabla, mitad + 1, iu, pivote\_inutil);

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

return ob;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: merge Fecha: 10/16 \*/

/\* Entrada: \*/

/\* tabla de enteros a ordenar (descendente) entre \*/

/\* los indices ip (inferior) e iu (superior) \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

int i,j,k, ob=0;

int tam = iu-ip+1;

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

if(!taux)

return ERR;

i=ip;

j=mitad +1;

k=0;

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

ob++;

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

taux[k]=tabla[i];

i++;

}

else{

taux[k]=tabla[j];

j++;

}

k++;

}

if(i > mitad){

while(j<=iu){

taux[k]=tabla[j];

j++;

k++;

}

}else if(j > iu){

while(i<=mitad){

taux[k]=tabla[i];

i++;

k++;

}

}

for(i=ip, k=0;i<=iu; tabla[i]=taux[k], i++,k++);

free(taux);

return ob;

}

4.3 Apartado 3

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: Quicksort Fecha: 10/16 \*/

/\* Entrada: \*/

/\* tabla de enteros a ordenar (descendente) entre \*/

/\* los indices ip (inferior) e iu (superior) \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

int ob=0, pos;

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

return ERR;}

if(ip==iu)

return OK;

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

if(ip < pos-1)

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

if(iu > pos+1)

ob+=quicksort(tabla, pos+1, iu, pivote);

return ob;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: partir Fecha: 10/16 \*/

/\* Entrada: \*/

/\* Divide la tabla en dos mitades ordenadas una \*/

/\* con respecto a la otra. Pivote seguarda en pos \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

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

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

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

m = \*pos;

k = tabla[m];

/\* Swap tabla[ip] y tabla[m] \*/

aux = tabla[ip];

tabla[ip] = tabla[m];

tabla[m] = aux;

m = ip;

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

ob++;

if (tabla[i] < k){

m++;

/\* Swap tabla[i] y tabla[m] \*/

aux = tabla[i];

tabla[i] = tabla[m];

tabla[m] = aux;

}

}

/\* Swap tabla[ip] y tabla[m] \*/

aux = tabla[ip];

tabla[ip] = tabla[m];

tabla[m] = aux;

\*pos=m;

return ob;

}

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: medio Fecha: 10/16 \*/

/\* Entrada: \*/

/\* tabla de enteros a ordenar (descendente) entre \*/

/\* los indices ip (inferior) e iu (superior) \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

if(!tabla || ip>iu || ip<0 || pivote<0 || pivote >2)

return ERR;

int ob = 0, med = (ip+iu)/2;

switch (pivote) {

case 0:

/\*Pivote es el primer elemento\*/

\*pos = ip;

break;

case 1:

/\*El pivote es el elemento medio\*/

\*pos = med;

break;

case 2:

/\*El pivote es "medio\_stat" del enunciado\*/

ob++;

if(ip < iu){

if(med < ip){

ob++;

\*pos = ip;

}

else if(med > iu){

/\*2 comparaciones: if ip<iu y if med>iu\*/

ob += 2;

\*pos = iu;

}else{

ob += 2;

\*pos = med;

}

}else{

if(med > ip){

ob++;

\*pos = ip;

}

else if(med < iu){

/\*2 comparaciones: if ip<iu y if med>iu\*/

ob += 2;

\*pos = iu;

}else{

ob += 2;

\*pos = med;

}

}

break;

default:

ob = 0;

}

return ob;

}

4.5 Apartado 5

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

/\* Funcion: medio Fecha: 10/16 \*/

/\* Entrada: \*/

/\* tabla de enteros a ordenar (descendente) entre \*/

/\* los indices ip (inferior) e iu (superior) \*/

/\* Salida: \*/

/\* Numero de veces que la OB ha sido ejecutada/ERR \*/

/\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*/

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

if(!tabla || ip>iu || ip<0 || pivote<0 || pivote >2)

return ERR;

int ob = 0, med = (ip+iu)/2;

switch (pivote) {

case 0:

/\*Pivote es el primer elemento\*/

\*pos = ip;

break;

case 1:

/\*El pivote es el elemento medio\*/

\*pos = med;

break;

case 2:

/\*El pivote es "medio\_stat" del enunciado\*/

ob++;

if(ip < iu){

if(med < ip){

ob++;

\*pos = ip;

}

else if(med > iu){

/\*2 comparaciones: if ip<iu y if med>iu\*/

ob += 2;

\*pos = iu;

}else{

ob += 2;

\*pos = med;

}

}else{

if(med > ip){

ob++;

\*pos = ip;

}

else if(med < iu){

/\*2 comparaciones: if ip<iu y if med>iu\*/

ob += 2;

\*pos = iu;

}else{

ob += 2;

\*pos = med;

}

}

break;

default:

ob = 0;

}

return ob;

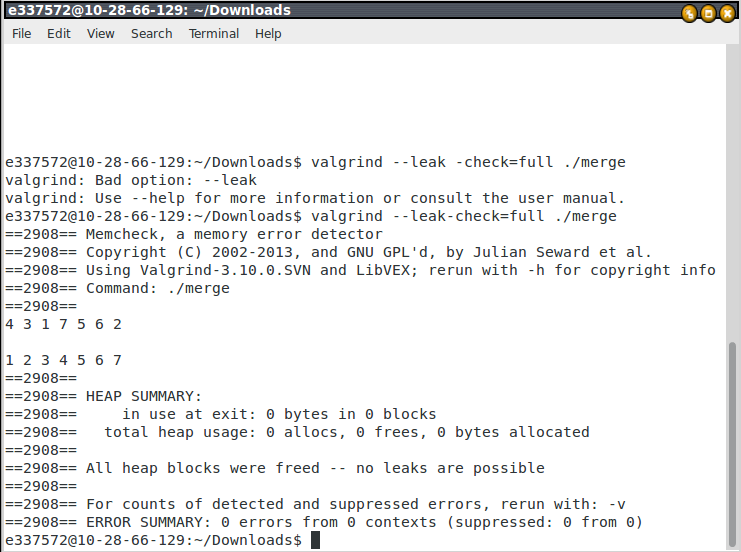
}

**5. Resultados, Gráficas**

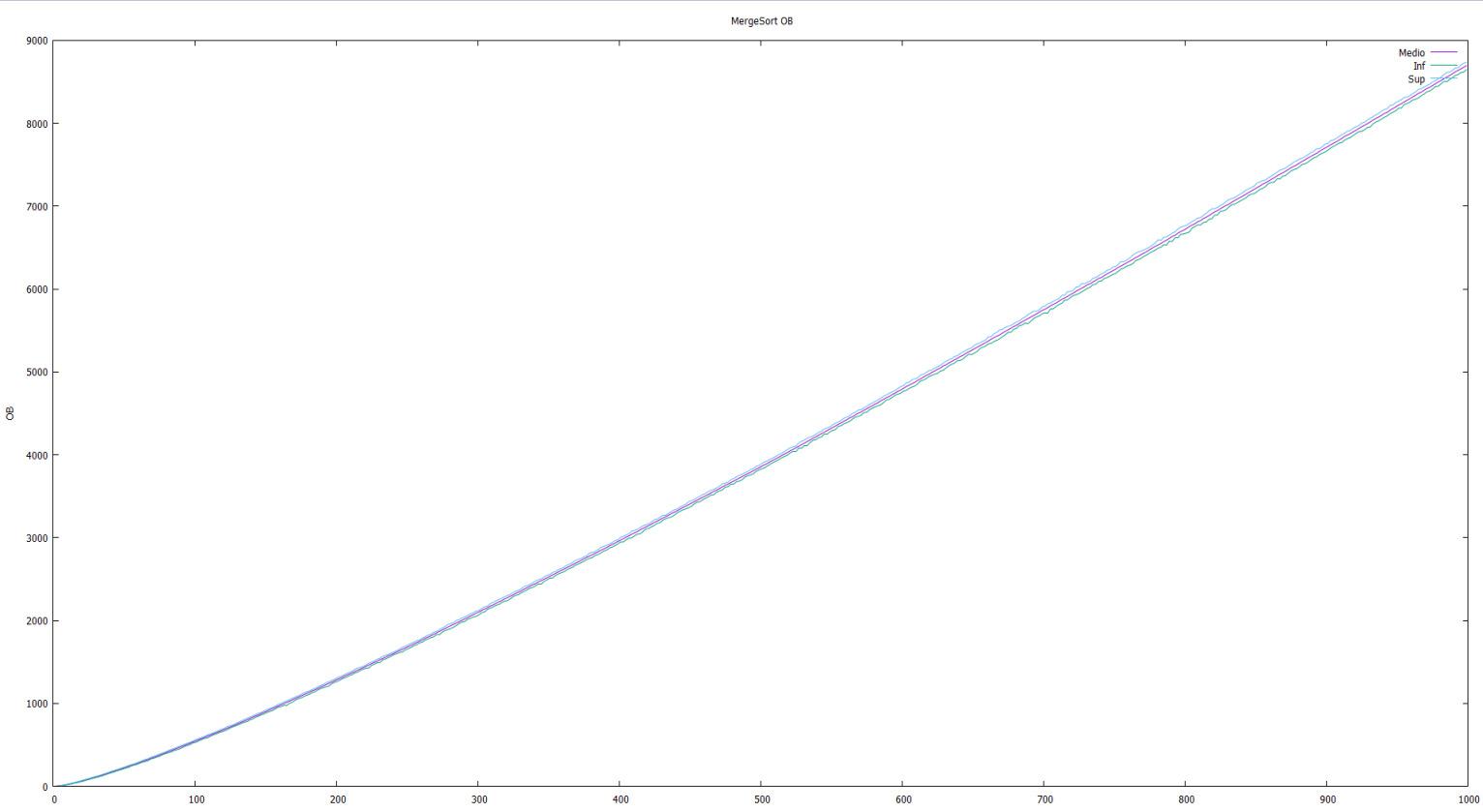
Aquí ponis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

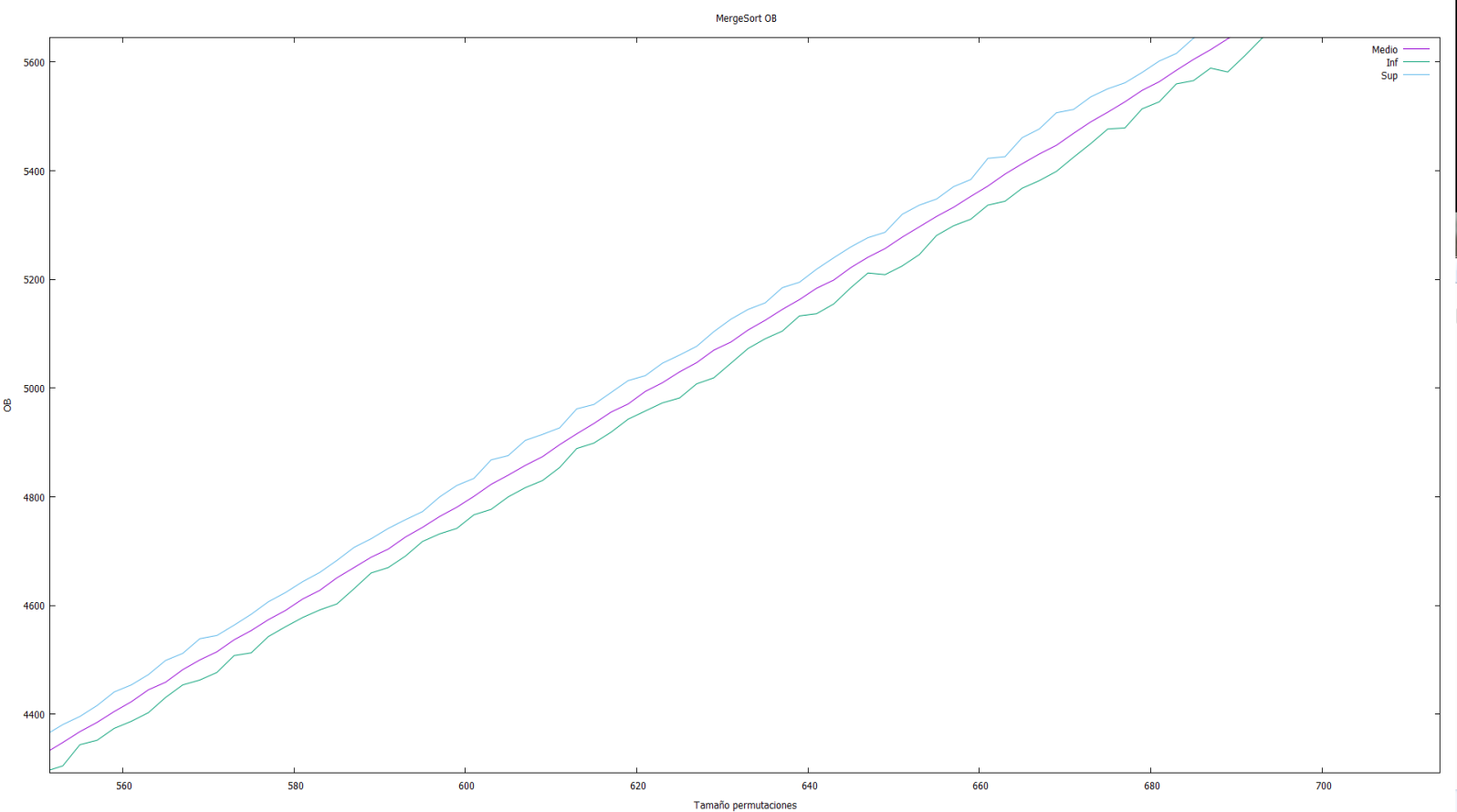
5.1 Apartado 1

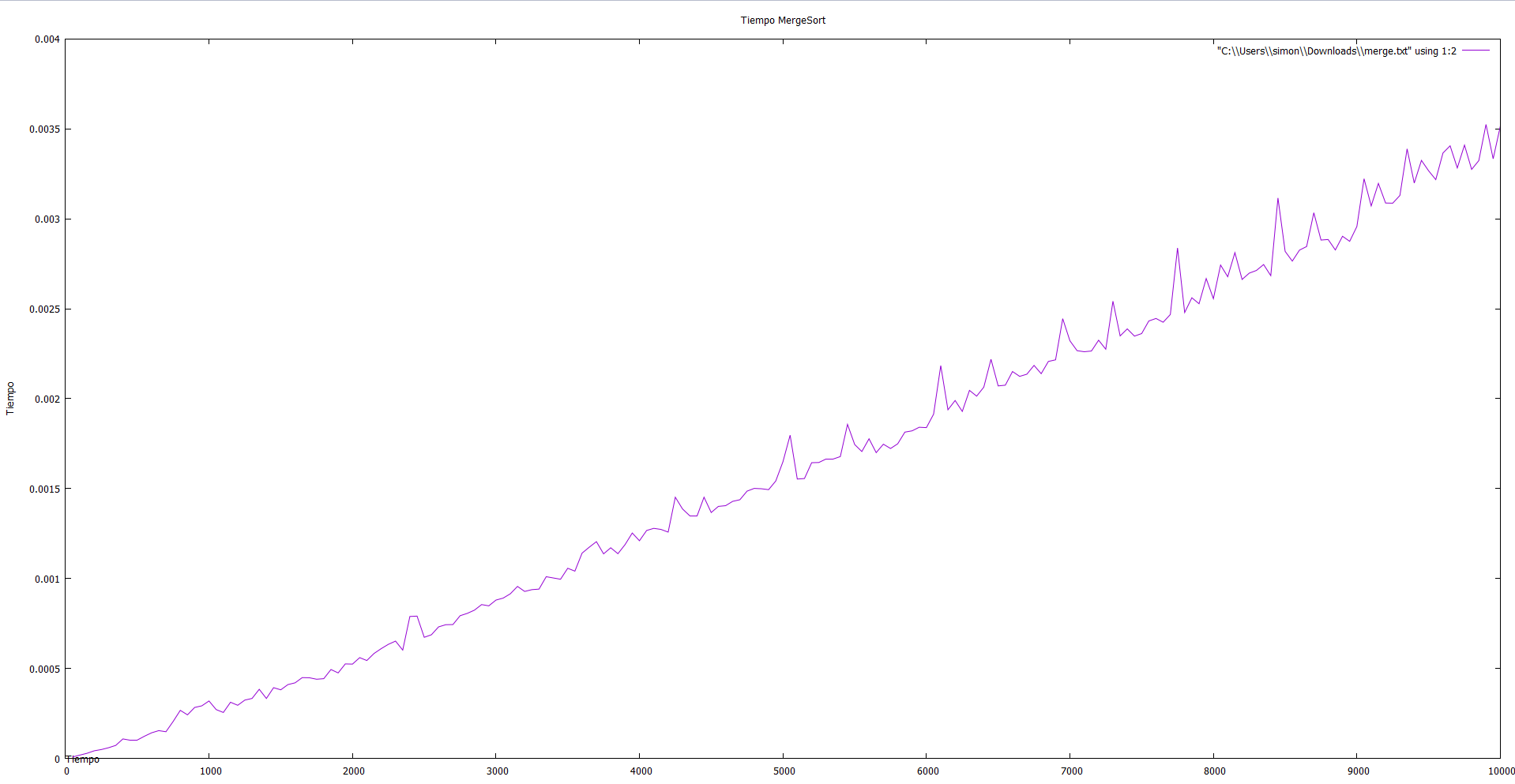
MergeSort ordena correctamente. Adjuntamos su comportamiento.



5.2 Apartado 2



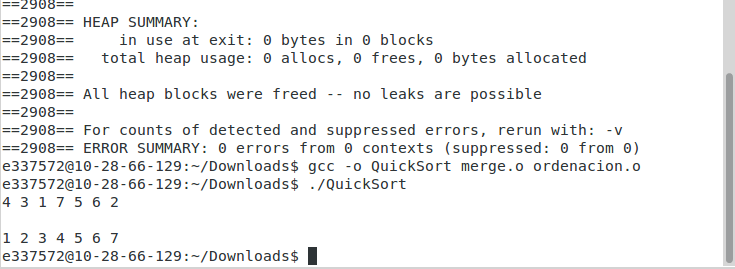
Como podemos ver, el caso medio queda acotado entre el superior y el inferior. Debido a lo cerca que están, a continuación adjuntamos un “zoom”:



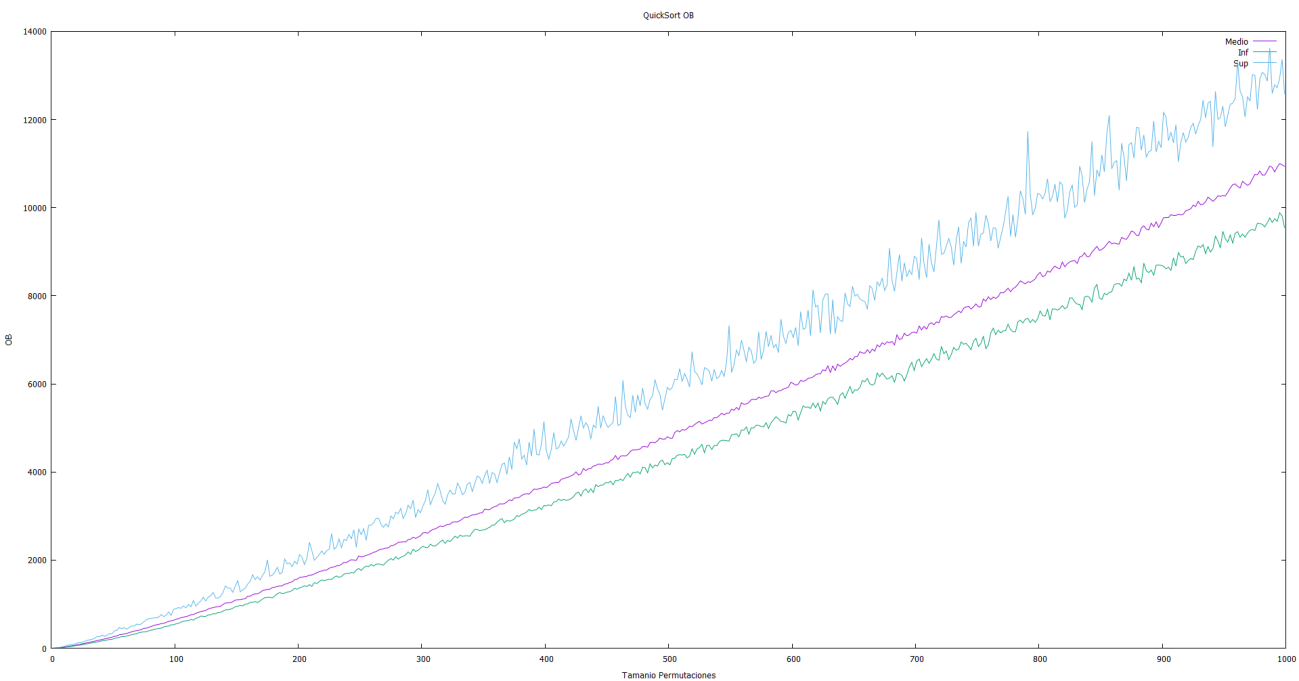
Como podemos ver, la gráfica se corresponde hasta cierto punto con los datos vistos en teoría. El comportamiento de los datos es en exceso lineal, pero debido a las limitaciones de nuestra máquina, no podemos realizar más iteraciones. En cambio, los OB aportan una descripción más apropiada del comportamiento de Mergesort.

5.3 Apartado 3

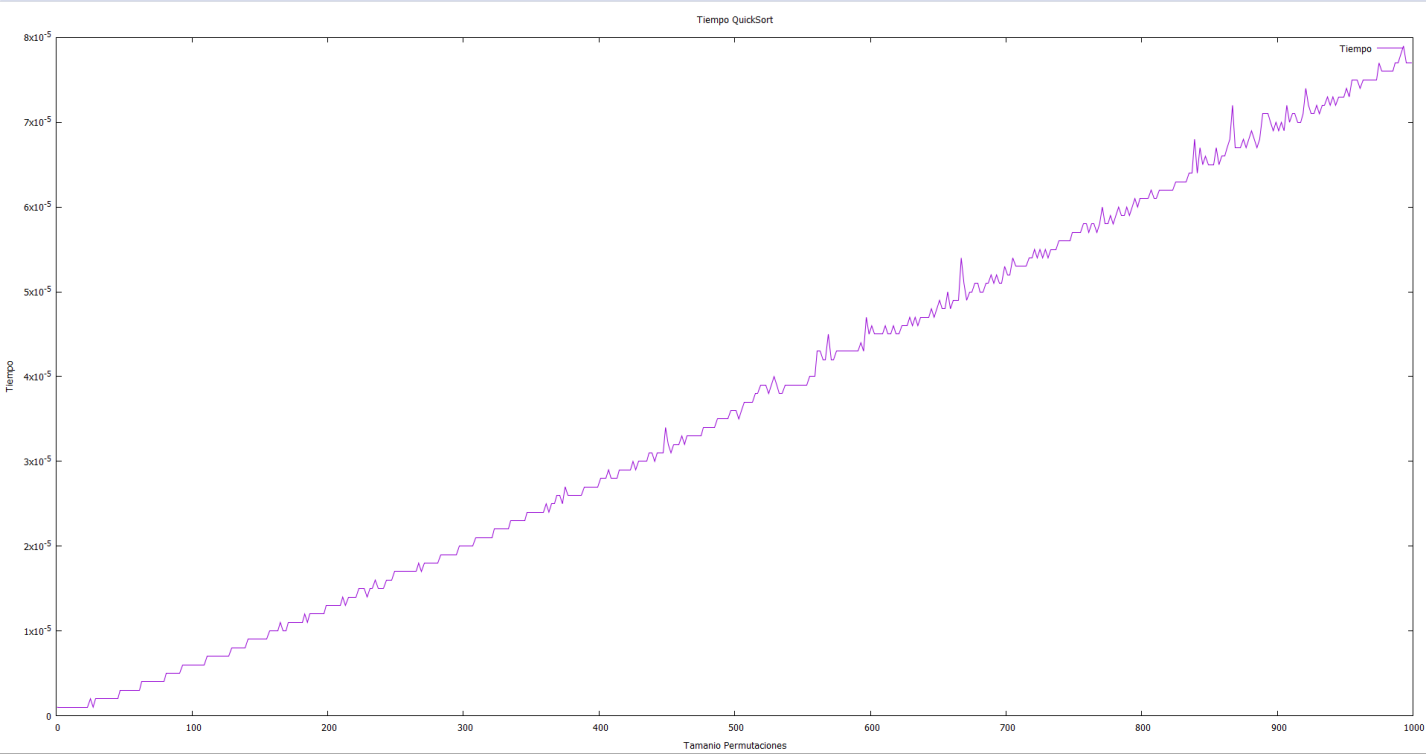
QuickSort ordena correctamente. Adjuntamos una captura de dicha ordenación.



5.4 Apartado 4



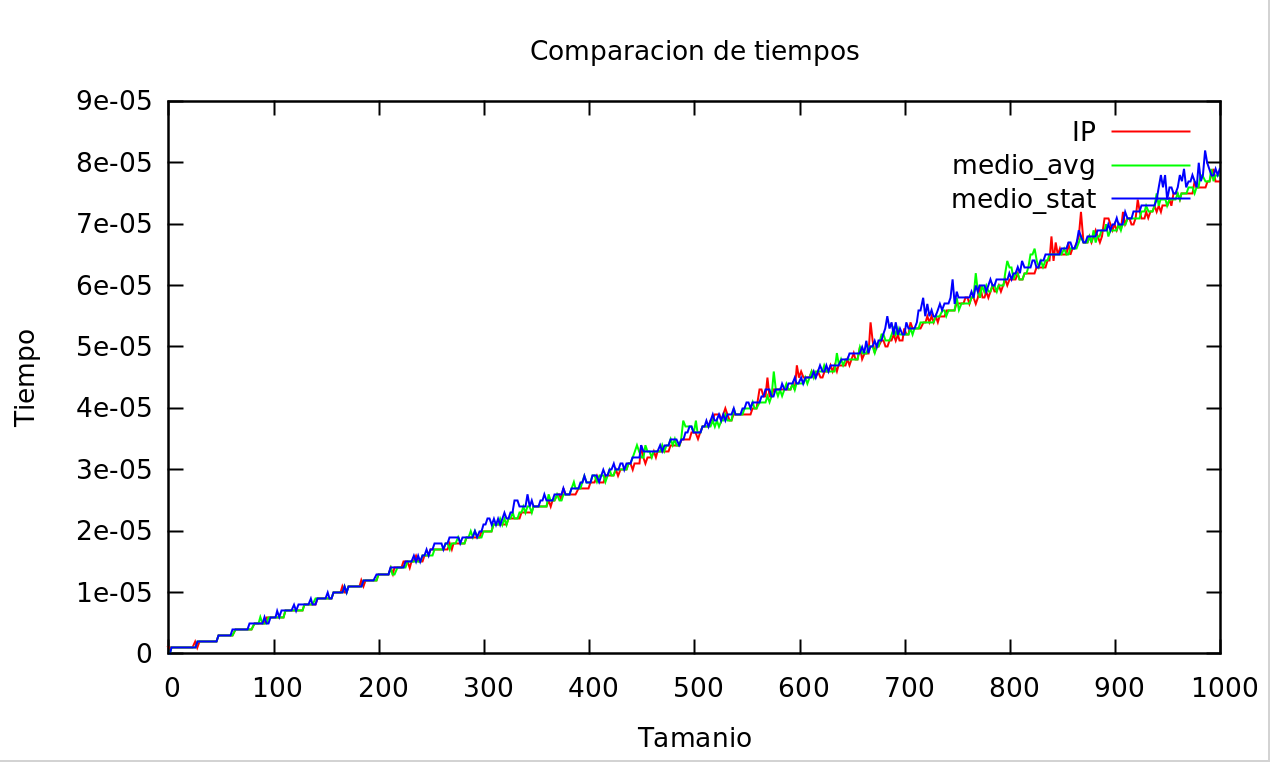
Como podemos observar, los OB medios se sitúan adecuadamente entre la cota inferior y superior. Cabe destacar la alta fluctuación que se observa al final de la superior. Puede deberse al comportamiento no-lineal de QuickSort, pero la causa más probable es tráfico en la CPU que ralentizara los cálculos para ciertas permutaciones.



El tiempo corresponde adecuadamente a las predicciones realizadas en previas clases de teoría.

5.5 Apartado 5

Podemos elegir el pivote con éxito.

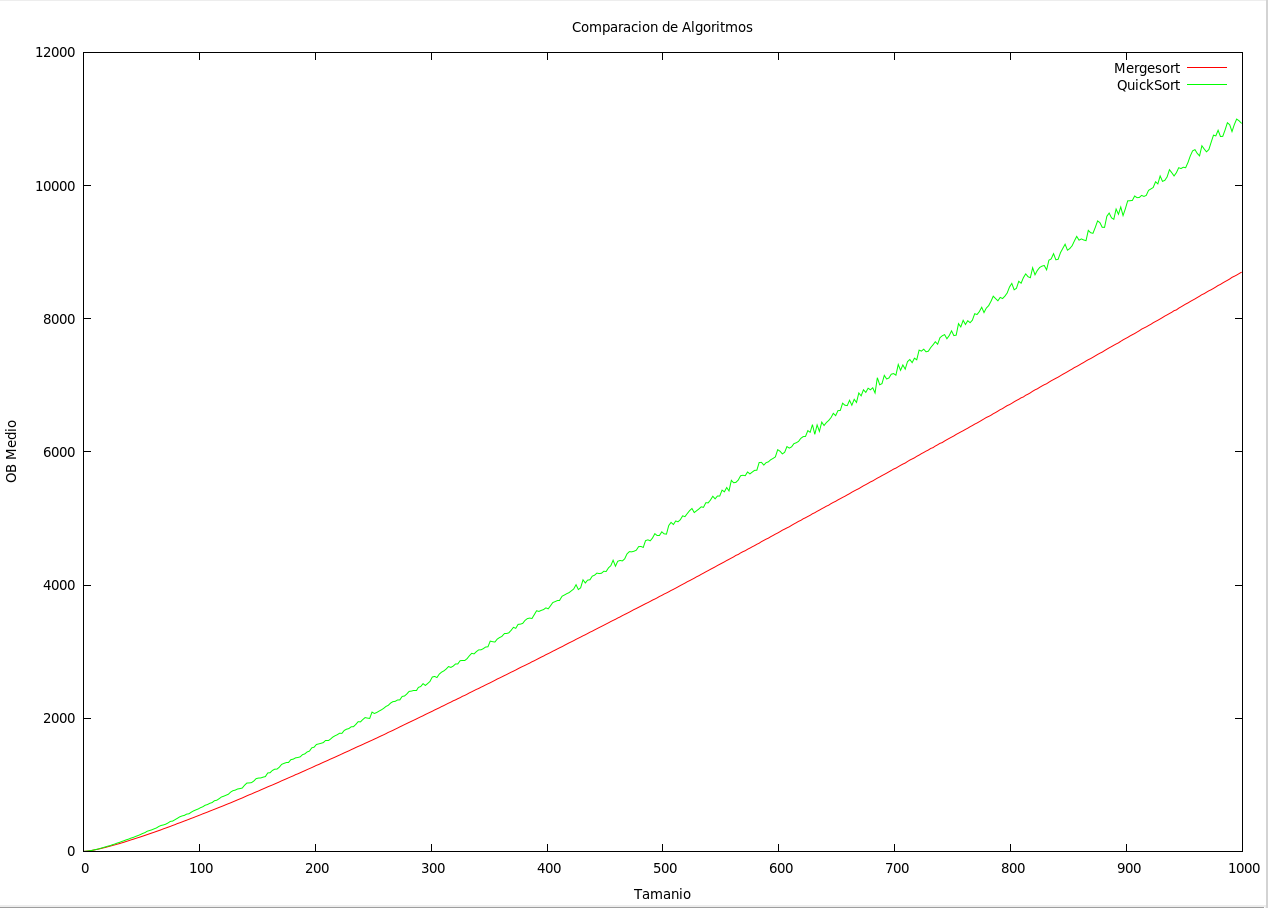


Como podemos ver, prácticamente no hay diferencia entre un pivote y otro, lo que se ajusta a lo esperado.

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

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

5.1 Pregunta 1



Tal y como expone la teoría, QuickSort realiza más Obs que Mergesort.

5.2 Pregunta 2

No, de hecho, las diferencias son tan poco apreciables que al intentar comparar el tiempo de ejecución de diferentes pivotes en un gráfico, estos se solapaban de manera perfecta, lo cual tiene sentido, teniendo en cuenta el alto número de permutaciones que ordenamos por tamaño, lo que nos asegura que cualquier diferencia local por elección de pivote desaparezca en la media estadística.

5.3 Pregunta 3

**Quicksort:**

Worst: n\*(n-1)/2

Average: 2\*(n\*log(n))+ O(n)

Best: O(n\*log(n))

**MergeSort:**

Worst: O(n\*log(n))

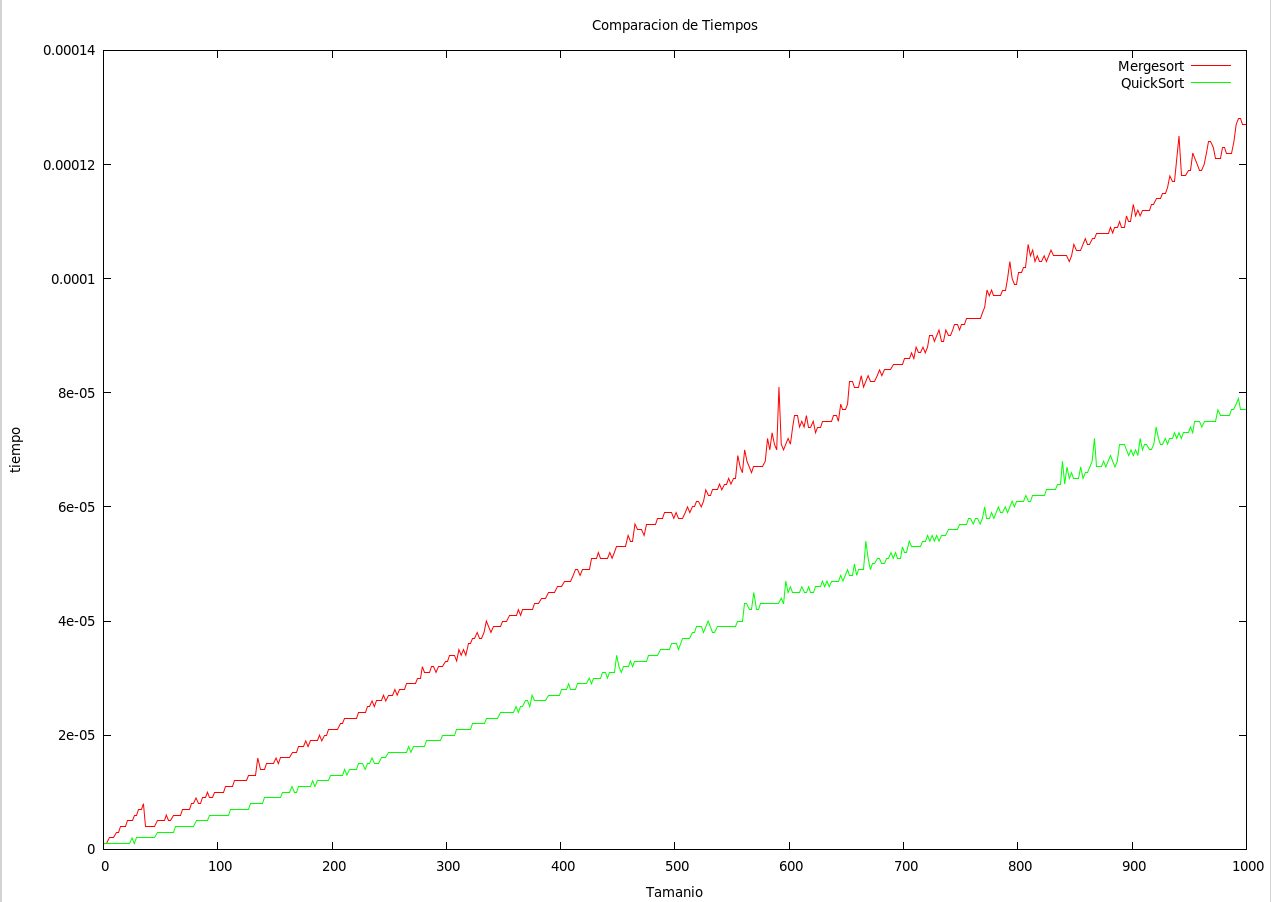
Average:n\*log(n)/2

Best: O(n\*log(n))

Habría que modificar el ejercicio5.c de manera que sólo incluyera los peores y mejores casos para imprimir al archivo.

5.4 Pregunta 4

En términos de memoria, QuickSort es muy superior, ya que es de los tipos “In Place”, lo que implica que no gasta memoria adicional. En cuanto al rendimiento práctico, son prácticamente idénticos, con MergeSort tomando más tiempo para la reserva de memoria. A continuación exponemos una comparativa de tiempos de ejecución, pudiendo confrimar que QuickSort es superior:



**Conclusion: Mergesort realiza menos OBs, pero QuickSort es más eficiente por su gestión de memoria.**

**6. Conclusiones finales.**

Esta práctica nos ha servido para reconocer la elegancia y complejidad de los algoritmos de ordenación recursivos. Además, pudimos confirmar empíricamente los resultados obtenidos de manera teórica en clase. Es verdad que hay ciertas discrepancias entre los teóricos y los obtenidos en esta práctica, pero ambos se ajustan de manera considerablemente buena a una tendencia general.