Análisis de Algoritmos 2022/2023

Práctica 2

Miguel Lozano Alonso y Eduardo Junoy Ortega, Grupo 1263.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta memoria se presenta el trabajo realizado durante las prácticas. En ella están incluidos los objetivos, las herramientas, la metodología, el código, los resultados y las gráficas de cada apartado. Además de incluir las respuestas a las cuestiones y la conclusión final de la práctica.

2. Objetivos

A continuación, se presentan los objetivos a conseguir en cada apartado.

2.1 Apartado 1

Conseguir implementar el código del método de ordenación MergeSort y el de la función que combina los dos arrays. Para ello se tiene que sustituir el algoritmo de ordenación de exercise4.c realizado en la primera práctica, este caso SelectSort.

2.2 Apartado 2

Representar y comparar las tablas correspondientes a la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación con los resultados teóricos.

2.3 Apartado 3

Implementar el código del método de ordenación QuickSort, el de la función que parte el array y el de hallar el elemento medio.

2.4 Apartado 4

Representar las tablas de tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas del algoritmo QuickSort en función del tamaño de la permutación y compararlas con los resultados teóricos.

2.5 Apartado 5

Adquirir los conocimientos para escribir el código de las funciones de la media y la mediana.

3. Herramientas y metodología

Hemos trabajado en Manjaro (Linux) y Windows, utilizando las aplicaciones Visual Studio Code, Github y Whatsapp y las herramientas que hemos utilizado son gcc, valgrind, gnuplot, ...

3.1 Apartado 1

En primer lugar, la función MergeSort, separa la tabla en dos subtablas. Después, ordena estas dos subtablas de manera recursiva, para finalmente combinar los elementos de estas dos tablas con la función merge. Esta función crea una tabla auxiliar y realiza la OB de MergeSort y de merge. Ahí copia los elementos de las dos subtablas cuando sea necesario y, cuando se ha completado este proceso, copia la tabla auxiliar en la tabla original y elimina la tabla auxiliar, devolviendo la tabla resultante de la unión de las dos subtablas.

3.2 Apartado 2

Tomamos el algoritmo MergeSort del apartado anterior y obtenemos la gráfica con la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación, representándolos y comparándolos con los resultados teóricos.

3.3 Apartado 3

Lo primero que realiza la función QuickSort es obtener el número de operaciones básicas resultantes de la función partition, es decir, el número de particiones que se realizan correctamente. En esta función se realizan los intercambios necesarios para ordenar las tablas resultantes a partir de la posición del elemento pivote.

3.4 Apartado 4

Tomamos el algoritmo QuickSort del apartado anterior y obtenemos la tabla de tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas del algoritmo en función del tamaño de la permutación, representándolos y comparándolos con los resultados teóricos.

3.5 Apartado 5

En la función median_avg simplemente se halla el valor medio del primer y el último elemento. Sin embargo, en la función median_stat se recibe la posición de la tabla del primer elemento, el último y la media de estos, los compara y devuelve como pivote la posición que contiene el valor intermedio o mediano de las tres.

4. Código fuente

A continuación, se muestra el código elaborado para cada apartado, incluyendo los comentarios para valgrind.

4.1 Apartado 1

```
/**

* @brief Ordena de menor a mayor una tabla

*

* @param tabla la tabla a ordenar

* @param ip primer elemento de la tabla

* @param iu último elemento de la tabla

*

* @return el número de operaciones básicas (comparaciones de clave)
que ha realizado el algortimo

*/

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

 int medio, n_obs, st;



 if(ip > iu) return ERR;



 /* Caso Base */

 if(ip == iu) {

   return OK;

 }
```

```
 n_obs = 0;

 medio = (ip+iu)/2;

 st = mergesort(tabla, ip, medio);

 if (st == ERR) return ERR;

 n_obs += st;

 st = mergesort(tabla, medio+1, iu);

 if(st==ERR) return ERR;

 n_obs += st;

  n_obs += merge(tabla, ip, iu, medio);

 return n_obs;

}



/**

* @brief Combina las dos subtablas en una tabla auxiliar y esta la
copia en la tabla original

*

* @param tabla la tabla a ordenar

* @param ip primer elemento de la tabla

* @param iu último elemento de la tabla

* @param imedio valor medio del número de elementos

*

* @return la tabla combinada

*/

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

```
int i, j, k, n_obs;

int *taux;



taux = (int*)malloc(sizeof(int) * (iu+1-ip));

if(taux == NULL) return ERR;

n_obs = 0;



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

  n_obs++;

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

    taux[k] = tabla[i];

    i++;

  }

  else {

    taux[k] = tabla[j];

    j++;

  }

}



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++;

    }

  }



  /* Copiar */

  for(i=ip, j=0; i<=iu; i++, j++) {

    tabla[i] = taux[j];

  }



  free(taux);

  return n_obs;

}
```

4.3 Apartado 3

```
/**
* @brief Elige un pivote, que puede ser la media, mediana o el
* elemento central y ordena la tabla en función de ese pivote
*
* @param tabla la tabla a ordenar
* @param ip primer elemento de la tabla
* @param iu último elemento de la tabla
*
* @return la tabla ordenada
*/
int quicksort(int* tabla, int ip, int iu) {
 int pos;
 int n_obs, st;


 if(ip>iu) return ERR;
 if (ip==iu) return 0;


 n_obs = partition(tabla, ip, iu, &pos);
 if(n_obs == ERR) return ERR;


 if (ip < pos-1)
 {
```

```
    st = quicksort(tabla, ip, pos-1);
    if(st == ERR) return ERR;
    n_obs += st;
  }
  if ((pos+1) < iu)
  {
    st = quicksort(tabla, pos+1, iu);
    if(st == ERR) return ERR;
    n_obs += st;
  }

  return n_obs;
}

/**
* @brief realiza una partición en dos respecto al total de la tabla
*
* @param tabla la tabla a ordenar
* @param ip primer elemento de la tabla
* @param iu último elemento de la tabla
* @param pos pivote divisiorio sobre el cual se va a ordenar la
tabla
*
* @return el número de operaciones básicas realizadas
```

```
*/

int partition(int* tabla, int ip, int iu, int *pos) {

 int i, k, n_obs;

 n_obs = 0;



 if(median(tabla, ip, iu, pos) == ERR) {

   return ERR;

 }



 k = tabla[*pos];



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

 *pos = ip;



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

 {

   n_obs++;

   if (tabla[i] < k)

   {

     (*pos)++;

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

   }

 }
```

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


 return n_obs;
}
/**
* @brief sitúa como pivote el primer elemento de la tabla
*
* @param tabla la tabla a ordenar
* @param ip primer elemento de la tabla
* @param iu último elemento de la tabla
* @param pos pivote divisiorio sobre el cual se va a ordenar la
tabla
*
* @return OK si el procedimiento se ha realizado correctamente.
* ERROR si existe algún error en el procedimiento
*/
int median(int *tabla, int ip, int iu, int *pos) {
 if(tabla == NULL || ip > iu) return ERR;


 (*pos) = iu;
 return OK;
}
```

4.5 Apartado 5

```
/**

* @brief sitúa como pivote el valor medio del primer elemento y el
último

*

* @param tabla la tabla a ordenar

* @param ip primer elemento de la tabla

* @param iu último elemento de la tabla

* @param pos pivote divisiorio sobre el cual se va a ordenar la
tabla

*

* @return OK si el procedimiento se ha realizado correctamente.

* ERROR si existe algún error en el procedimiento

*/

int median_avg(int *tabla, int ip, int iu, int *pos){

 if(tabla==NULL || ip>iu) return ERR;



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



 return OK;

}



/**

* @brief sitúa como pivote el valor mediano, que es el elemento
situado
```

```
* en la posición de la mitad de una tabla ordenada

*

* @param tabla la tabla a ordenar

* @param ip primer elemento de la tabla

* @param iu último elemento de la tabla

* @param pos pivote divisiorio sobre el cual se va a ordenar la
tabla

*

* @return OK si el procedimiento se ha realizado correctamente.

* ERROR si existe algún error en el procedimiento

*/

int median_stat(int *tabla, int ip, int iu, int *pos) {

 if(tabla == NULL || ip>iu) return ERR;



 (*pos) = (tabla[ip] + tabla[iu] + tabla[(iu+ip)/2]) / 3;



 return OK;

}
```

La función indicada arriba la hemos considerado más eficiente para ordenar permutaciones ya que no se requieren OBs adicionales.

Una función alternativa a esta y que funciona tanto para permutaciones como para otros tipos de array sería:

```
int median_stat(int *tabla, int ip, int iu, int *pos) {

  int m;
```

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

m = (iu+ip)/2;

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

{

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

  {

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

    {

      *pos=m;

      return 3;

    }

    else

    {

      *pos=iu;

      return 3;

    }

  }

  else

  {

    *pos=ip;

    return 2;

  }

}
```

```
  else
  {
    if(tabla[iu]<tabla[m])
    {
      if (tabla[m]<tabla[ip])
      {
        *pos=m;
        return 3;
      }
      else
      {
        *pos=ip;
        return 3;
      }
    }
    else
    {
      *pos=iu;
      return 2;
    }
  }
  return 0;
}
```

5. Resultados, Gráficas

En este punto se presentan los resultados y gráficas obtenidas en cada apartado.

5.1 Apartado 1

Aplicamos el comando de ejecución:

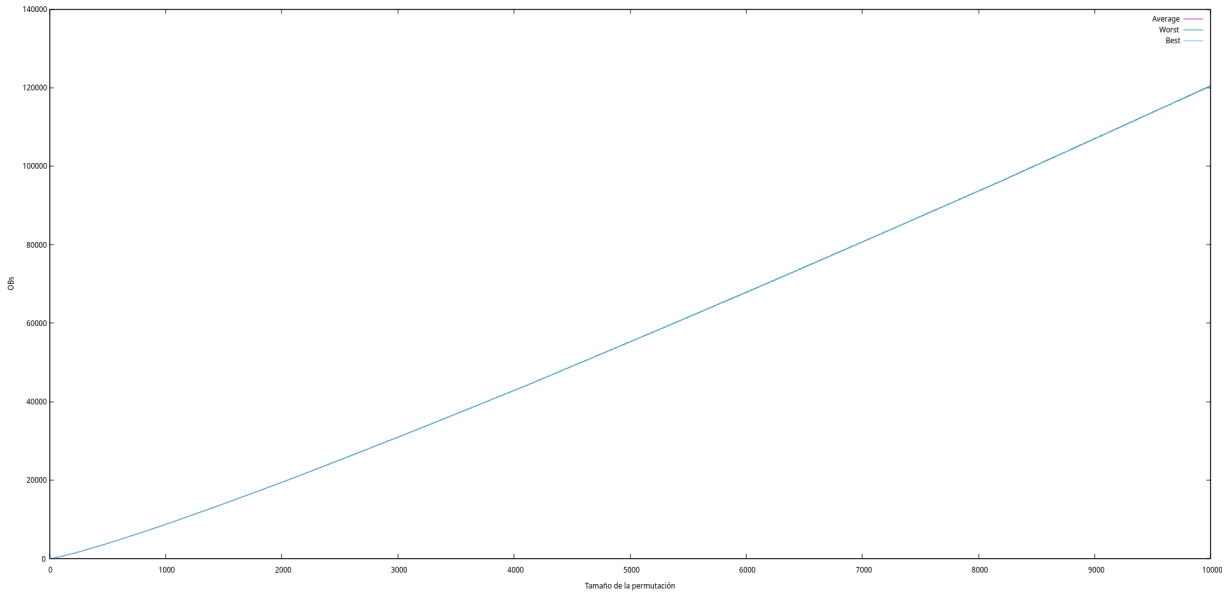
./exercise1 -size 10

Nos da el siguiente resultado:

```
Running exercise1
Practice number 2, section 1
Done by: Miguel Lozano and Eduardo Junoy
Group: 1263
1       2       3       4       5       6       7       8       9       10
```

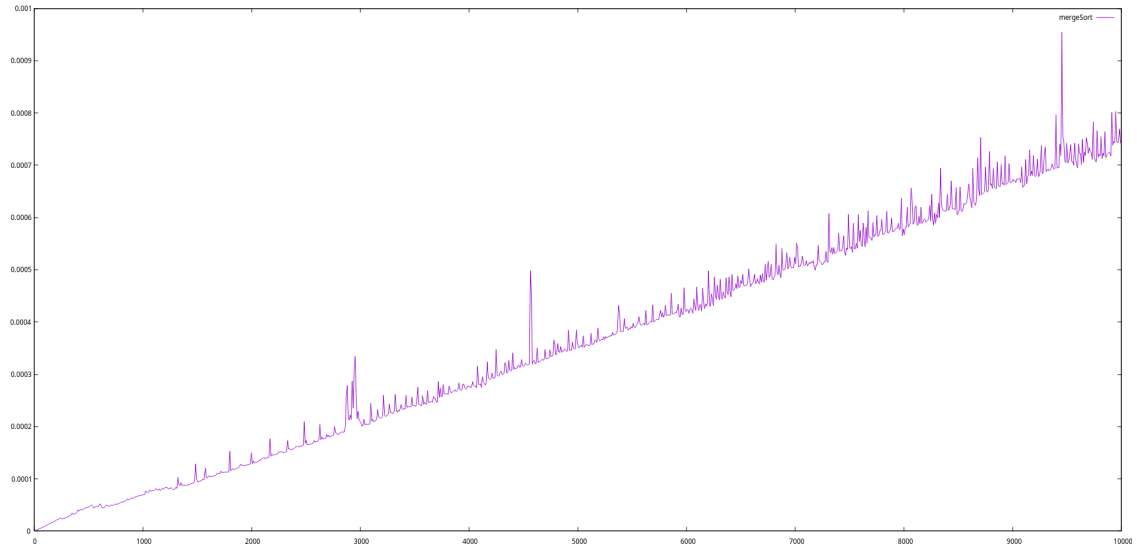
5.2 Apartado 2

Gráfica comparando los tiempos mejor, peor y medio en OBs para MergeSort:



Como podemos ver no hay gran diferencia entre el caso peor, el caso medio y el caso más optimista, lo que se ajusta a nuestras mediciones teóricas de MergeSort.

Gráfica con el tiempo medio de reloj para MergeSort:



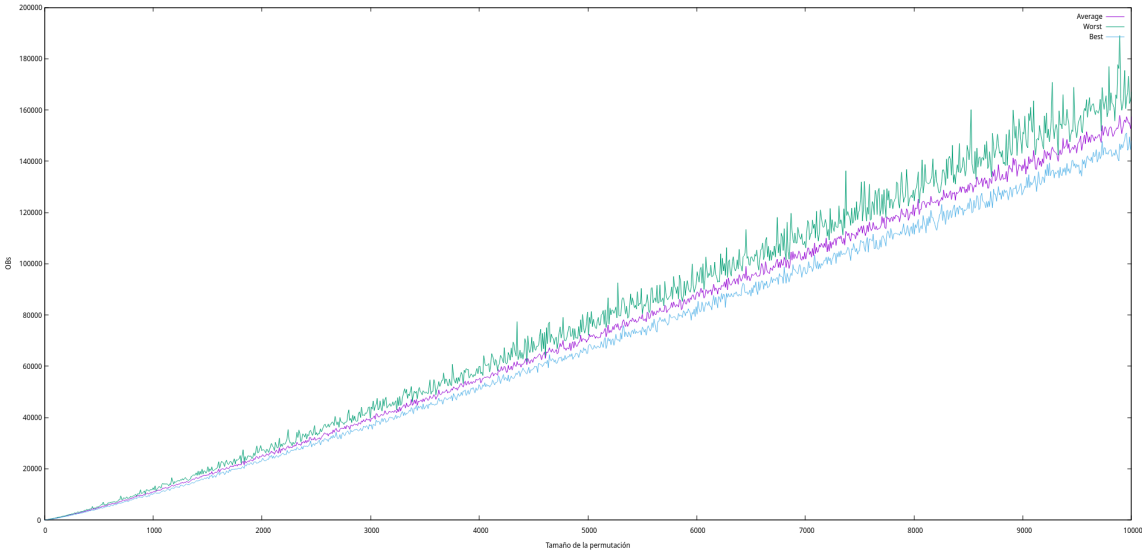
Como se puede observar en la gráfica, MergeSort tiene un rendimiento aproximado de $O(Nlog_2(N))$.

5.3 Apartado 3

```
 miguel@msi  ~/Documentos/Universidad/AALG-practicas/p2  master ±  ./exercise2y3 -size 10
Practice number 2
Done by: Miguel Lozano and Eduardo Junoy
Group: 1263
1       2       3       4       5       6       7       8       9       10
 miguel@msi  ~/Documentos/Universidad/AALG-practicas/p2  master ±
```

5.4 Apartado 4

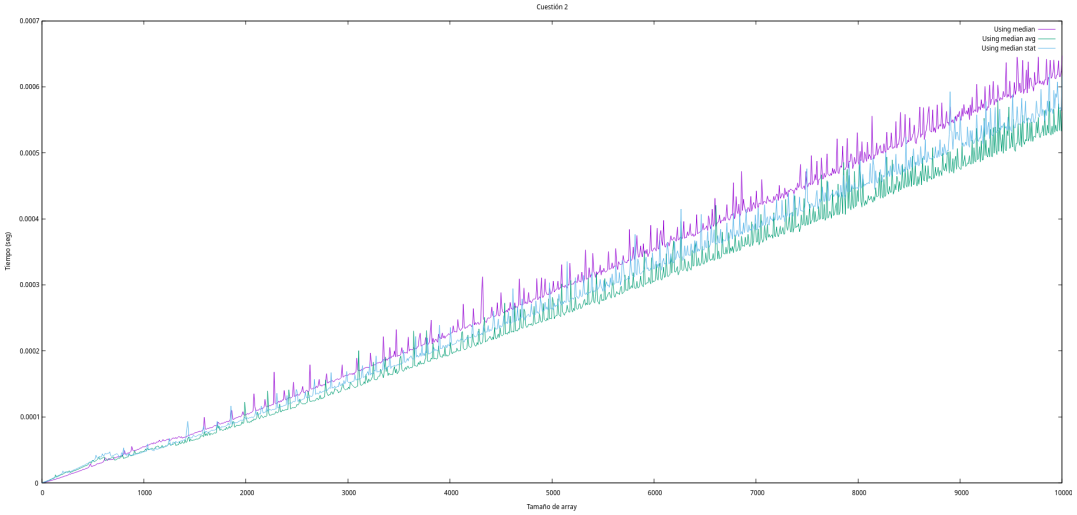
Gráfica de caso medio, peor y mejor de QuickSort:



En este caso, al contrario que en el caso de MergeSort, sí se observa diferencia entre el caso mejor, peor y medio, lo cual se corresponde correctamente con los resultados teóricos.

5.5 Apartado 5

Gráfica con el tiempo medio de reloj comparando las versiones de QuickSort con las funciónes pivote **median, median_avg** y **median_stat:**

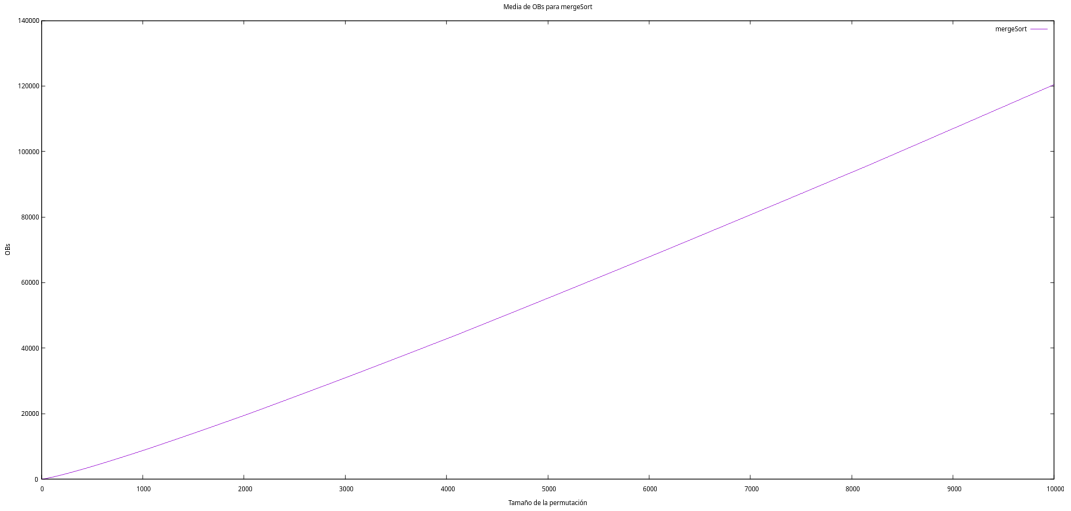


En la gráfica podemos observar como hay menor coste de ejecución usando la función median_avg, después con median_stat y por último, la que se ejecuta con mayor coste es median. Esto tiene sentido, ya que con gran cantidad de números, al introducir el valor del primer elemento se tardará más, después con el valor mediano y el más rápido será el valor medio.

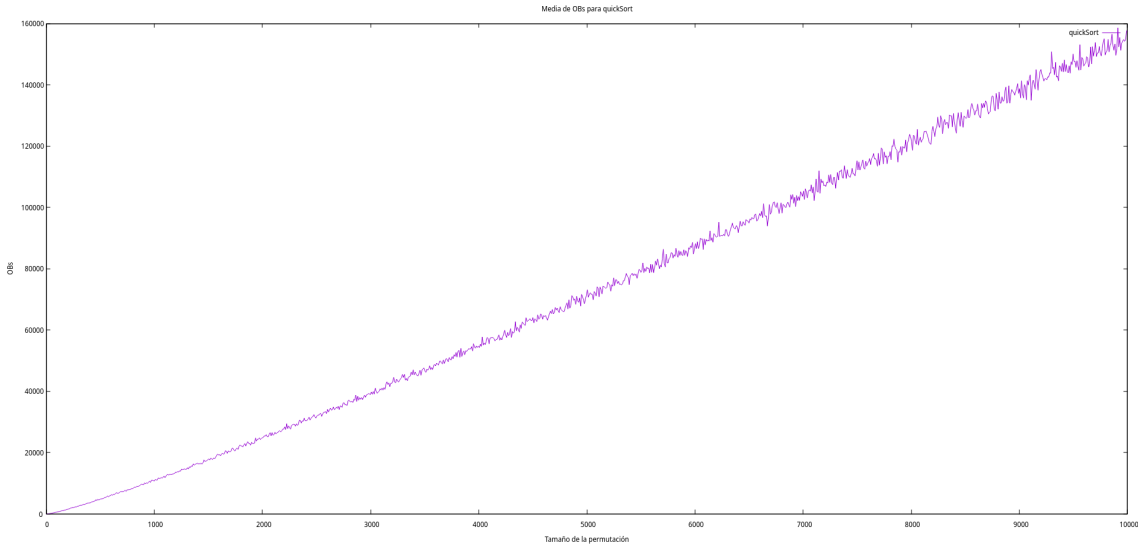
5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

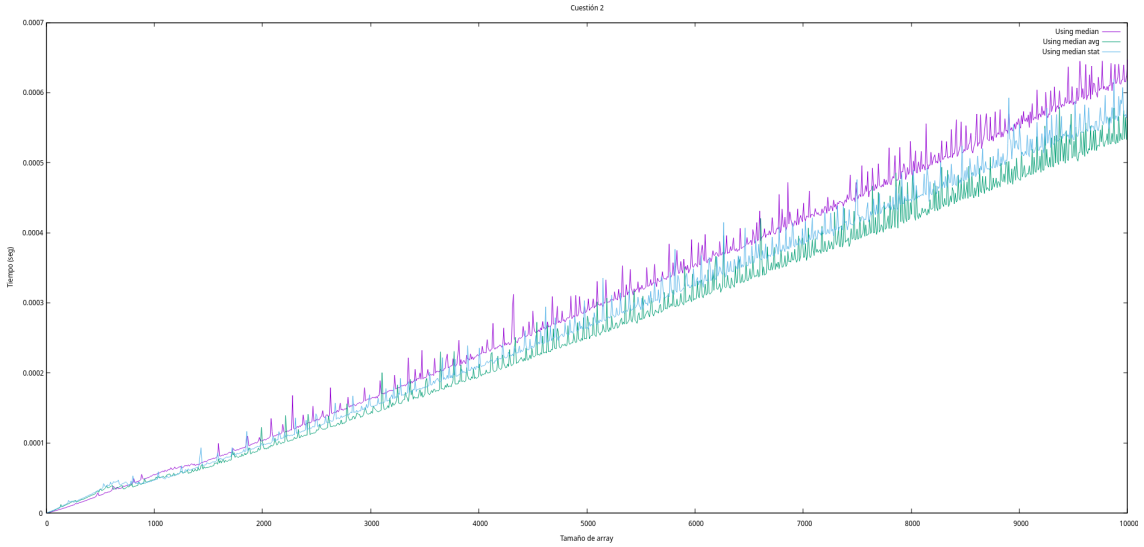
Por un lado, el caso medio de MergeSort es de $Nlog_2(N)$.



Por el otro, el caso medio de QuickSort es $2Nlog_2(N) + O(N)$.



5.2 Pregunta 2



En la gráfica podemos observar como hay menor coste de ejecución usando la función median_avg, después con median_stat y por último, la que se ejecuta con mayor coste es median. Esto tiene sentido, ya que con gran cantidad de números, al introducir el valor del primer elemento se tardará más, después con el valor mediano y el más rápido será el valor medio.

5.3 Pregunta 3

Los casos mejor y peor de MergeSort son el mismo que el medio: $Nlog_2(N)$

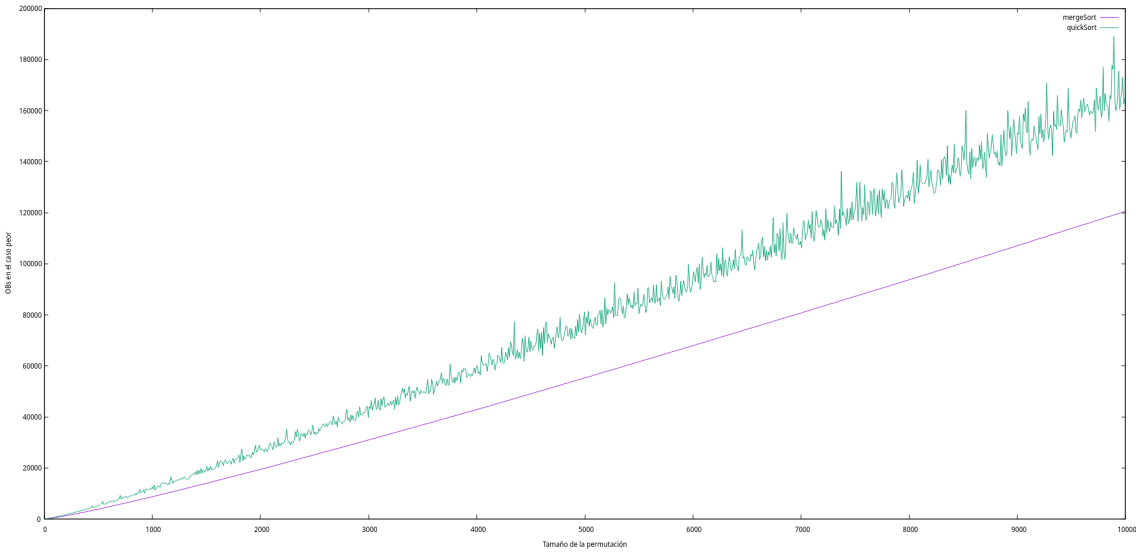
El caso mejor de QuickSort es: $Nlog_2(N)$ y el peor: $\frac{N(N-1)}{2}$

Para calcular el mejor caso en QuickSort hay que aportar una tabla completamente desordenada y para el peor una tabla ordenada, ya que el algoritmo está obligado a hacer todas las comparaciones de clave $\frac{N(N-1)}{2}$

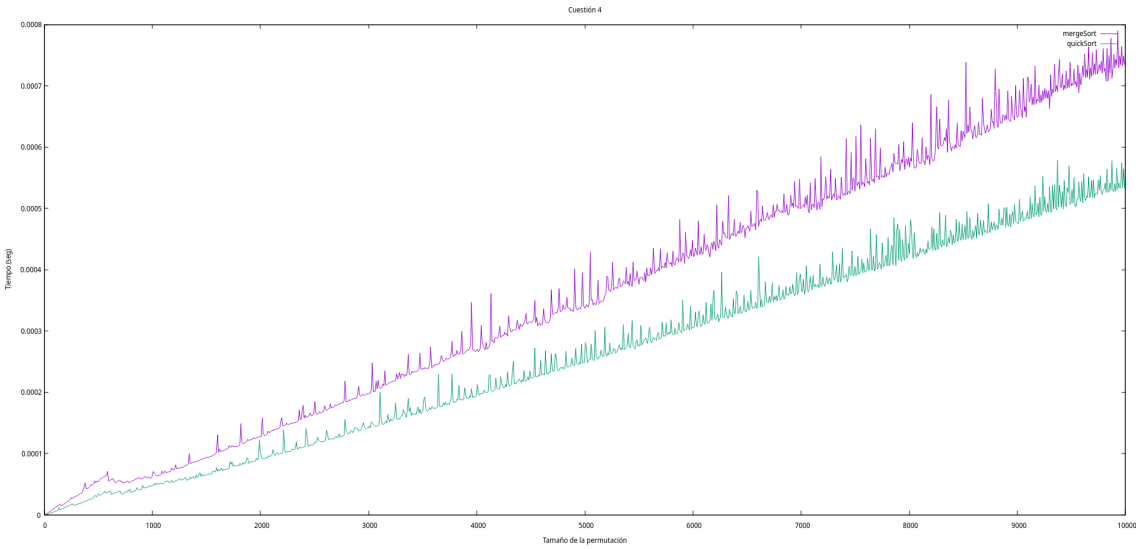
5.4 Pregunta 4

El algoritmo más eficiente teóricamente es MergeSort, ya que siempre tiene un rendimiento de $Nlog_2(N)$. Sin embargo el algoritmo QuickSort tiene ese rendimiento como caso mejor pero otros rendimientos inferiores como caso peor.

Empíricamente nos encontramos con que QuickSort ha efectuado más OBs que MergeSort en el caso peor.



Sin embargo, QuickSort ha sido más rápido en tiempo medio de reloj que MergeSort, seguramente debido al tiempo que tarda en reservar memoria o en ejecutar otras líneas de código.



6. Conclusiones finales.

La práctica nos ha ayudado a aplicar los conocimientos que hemos obtenido en las clases teóricas sobre los algoritmos de MergeSort y QuickSort. Así como a aprender a cómo cambiar el valor del pivote en los algoritmos situándolo como primer elemento, la media y mediana. Y también a entender el rendimiento de estos algoritmos y poder compararlos empíricamente.