Análisis de Algoritmos 2022/2023

Práctica 2

Daniel Cruz Navarro y Daniel Birsan, 1272.

Código	Gráficas	Memoria	Total

1. Introducción.

En este proyecto vamos a usar la recursión para la ordenación de arrays, los famosos algoritmos QuickSort y MergeSort serán puestos en práctica, veremos algunas de sus variaciones(sobre todo en el quicksort) y que nos conviene más para ordenar según nuestras condiciones. Veremos como además, como la teoría nos dice, son algoritmos más efectivos que los ya estudiados SelectSort o InsertSort.

2. Objetivos

2.1 Apartado 1

Implementar la función **int mergesort(int* tabla, int ip, int iu)**, utilizará el conocido algoritmo de ordenación MergeSort, valiéndose de la función auxiliar **int merge(int* tabla, int ip, int iu, int imedio)**, esta función realizará las combinaciones entre las tablas para ordenar la tabla inicial. Para comrpobar su correcto funcionamiento realizaremos los cambios necesarios en el ficher exercise4.c.

2.2 Apartado 2

Modificar el fichero exercise5.c para utilizar el algoritmo MergeSort y así obtener el fichero .log con todos los datos correspondientes a cada tamaño de la tabla además de poder generar el fichero datamerge.txt que será representado gráficamente utilizando Gnuplot que será comparado con la función del coste teórico de MergeSort.

2.3 Apartado 3

Implementar la función a **int quicksort(int* tabla, int ip, int iu)**, la cual se llama a si misma de manera recursiva y la cual necesita a **int partition (int* tabla, int ip, int iu,int *pos),** la cual usará **int median(int *tabla, int ip, int iu,int *pos)** para elegir el pivote.

2.4 Apartado 4

Modificar exercise5.c para utilizar el algoritmo quicksort y medir sus tiempos, OB media, OB peor y OB mejor. Los resultados los compararemos con ayuda de gráficas.

2.5 Apartado 5

Implementamos dos nuevas maneras de elegir el pivote, i**nt median avg(int *tabla, int ip, int iu, int *pos)** y **int median stat(int 1 *tabla, int ip, int iu, int *pos),** que escoge un pivote u otro según las características de la tabla.

3. Herramientas y metodología

Para la realización de esta práctica hemos utilizado *Linux Ubuntu 20.04* donde hemos instalado *Visual Studio Code* para escribir y modificar el código. Con el código escrito nos hemos valido de comandos de la terminal de Ubuntu para la compilación, enlazado y ejecución de nuestras funciones, entre estos comandos destacan *make, gcc, gnuplot, valgrind, gdb,* además de los comandos básicos de manejo de terminal. Para compartir el código y mantener versiones antiguas de este hemos utilizado Git y *Github.*

3.1 Apartado 1

Modificamos el pseudocódigo estudiado en las clases de teoría para realizar correctamente las divisiones de la tabla y poder calcular las operaciones básicas. Utilizaremos recursividad en la función principal para las divisiones y la auxiliar para combinar ordenadamente los elementos. Utilizamos el makefile y el fichero exercise4.c para comprobar el funcionamiento del algoritmo implementado.

3.2 Apartado 2

Usando el fichero exercise5.c de la práctica anterior cambiaremos llamadas a funciones y argumentos de estas para utilizar el MergeSort y registrar los datos en exercise5.log del algoritmo pertinente. Implementaremos código nuevo en times.c para no reemplazar el fichero data.txt sino crear uno nuevo que será representado gráficamente con Gnuplot, con replot crearemos una nueva gráfica con el valor teórico del coste de MergeSort.

3.3 Apartado 3

Modificamos el pseudocódigo existente en las clases de teoría para adaptarlo a nuestras funciones pedidas, empezamos implementando partition y median que vamos a utilizar en quicksort.

3.4 Apartado 4

Cambiamos el Makefile, exercise5.c y times.c para comprobar los resultados de nuestro algoritmo, con diferentes ficheros, usamos el gnuplot parar comparar todos los resultados, incluyendo una función ajustada para comparar con los resultados teóricos.

3.5 Apartado 5

Hacemos cambios en el Makefile y times.c para comprobar los diferentes resultados según nuestro pivote, con gnuplot comparamos los resultados entre sí y con su valor teórico.

4. Código fuente

4.1 Apartado 1

```
int MergeSort(int* tabla, int ip, int iu){
  int m, bo1=0, bo2=0, bo3=0;

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

  if(ip==iu) return OK;

  m=(ip+iu)/2;

  bo1 += MergeSort(tabla,ip,m);
  if(bo1==ERR) return ERR;

  bo2 += MergeSort(tabla,m+1,iu);
  if(bo2==ERR) return ERR;

  bo3 += merge(tabla,ip,iu,m);
  if(bo3==ERR) return ERR;

  return bo1+bo2+bo3;
}


int merge(int* tabla, int ip, int iu, int imedio){
  int* tabla_aux=NULL;
  int i,j,k=0,c=0;

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

  tabla_aux=(int*)malloc((iu-ip+1)*sizeof(int));
  if(tabla_aux==NULL) return ERR;

  i=ip;
  j=imedio+1;

  while(i<=imedio && j<=iu){
    if(tabla[i]<tabla[j]){
      tabla_aux[k]=tabla[i];
      i++;
      c++;
    }
    else{
      tabla_aux[k]=tabla[j];
      j++;
      c++;
    }
    k++;
```

```
  }

  while(j<=iu){
    tabla_aux[k]=tabla[j];
    j++;
    k++;
  }

  while(i<=imedio){
    tabla_aux[k]=tabla[i];
    i++;
    k++;
  }

  for(i=ip;i<=iu;i++){
    tabla[i]=tabla_aux[i-ip];
  }

  free(tabla_aux);
  return c;
}
```

4.3 Apartado 3

```
int quicksort(int* tabla, int ip, int iu)

{

 int n;

 int m;



 n = 0;

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

   return ERR;



 if (ip == iu)

   return n;

 else
```

```
 {
   n += partition(tabla, ip, iu, &m);
   if (n == ERR)
     return ERR;
   if (ip < m - 1) {
     n += quicksort(tabla, ip, m - 1);
     if (n == ERR)
       return ERR;
   }
   if(m + 1 < iu) {
     n += quicksort(tabla, m + 1, iu);
     if (n == ERR)
       return ERR;
   }
 }
 return n;
}


int partition(int* tabla, int ip, int iu, int *pos)
{
 int i, n, k, aux;
  if (!tabla || !pos || ip < 0 || ip > iu)
   return ERR;
```

```
n = 0;
n = median_stat(tabla, ip, iu, pos);
if (n == ERR)
  return ERR;
k = tabla[*pos];
tabla[*pos] = tabla[ip];
tabla[ip] = k;
*pos = ip;
for (i = ip+1; i <= iu; i++)
{
  if (tabla[i] < k)
  {
    (*pos)++;
    aux = tabla[i];
    tabla[i] = tabla[*pos];
    tabla[*pos] = aux;
  }
  aux = tabla[*pos];
  n++;
}
tabla[*pos] = tabla[ip];
tabla[ip] = aux;
```

```
 return n;

}



int median(int *tabla, int ip, int iu,int *pos)

{

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

   return ERR;

 *pos = ip;

 return (0);

}
```

4.5 Apartado 5

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

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

   return ERR;

 *pos = (ip + iu)/2;

 return (0);

}



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

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

   return ERR;
```

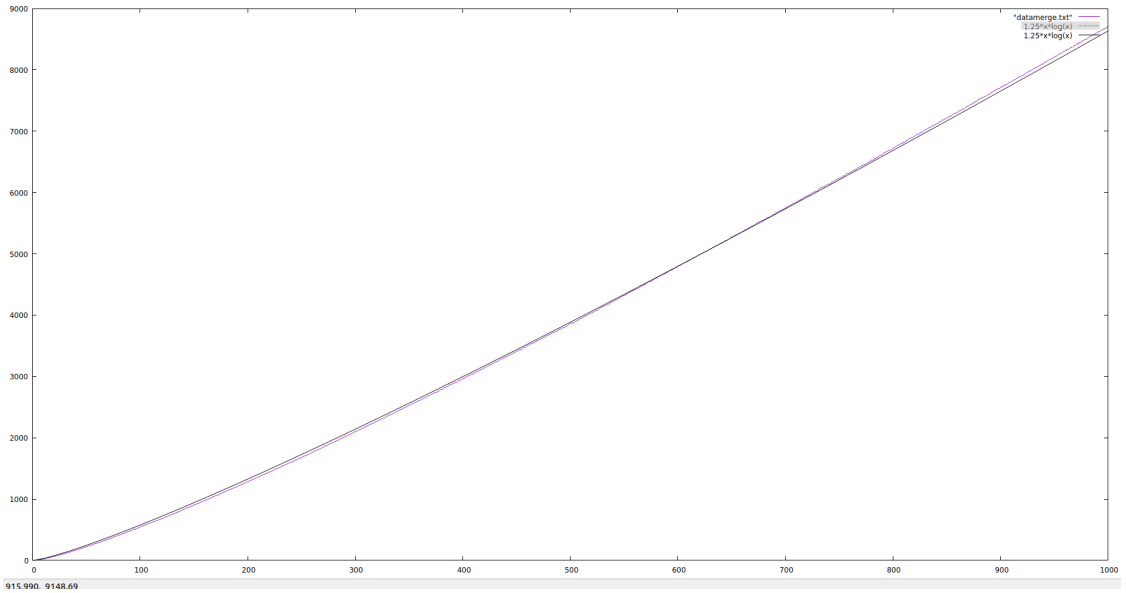
```
 *pos = (ip + iu)/2;

 if ((tabla[iu] > tabla[*pos] && tabla[ip] > tabla[iu]) ||
(tabla[iu] > tabla[ip] && tabla[*pos] > tabla[iu])) {

   *pos = iu;

   return (4);

 }

 if ((tabla[ip] > tabla[*pos] && tabla[iu] > tabla[ip]) ||
(tabla[ip] > tabla[iu] && tabla[*pos] > tabla[ip]))

   *pos = ip;

 return (8);

}
```

5. Resultados, Gráficas

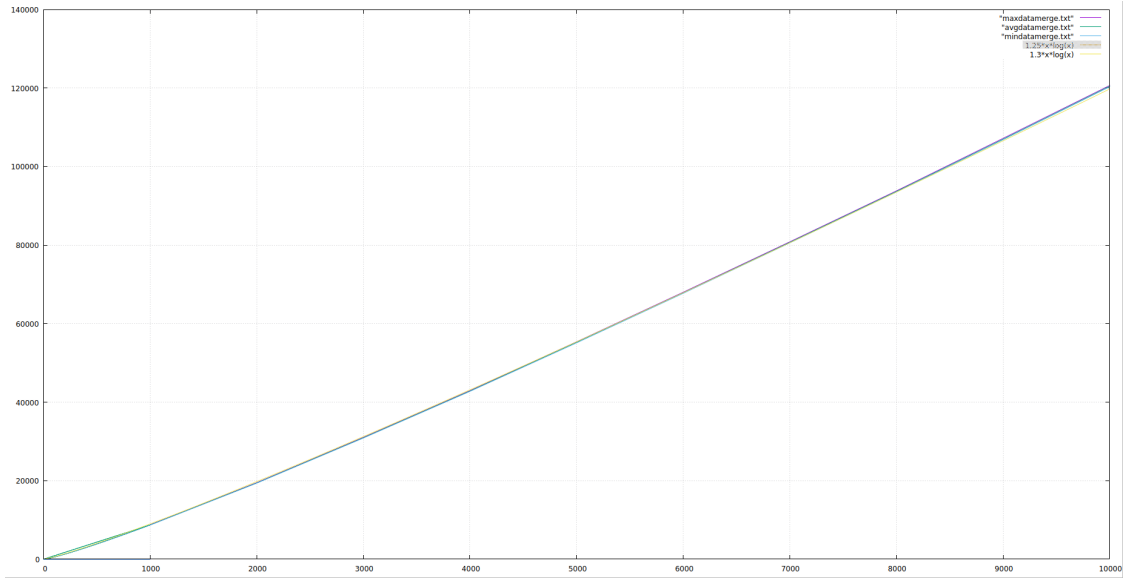
5.1 Apartado 1

Tras ejecutar el exercise4.c el resultado será una tabla ordenada de menor a mayor habiendo utilizado MergeSort como algoritmo de ordenación. La tabla tendrá el tamaño seleccionado en el Makefile, en nuestro caso para la gráfica del punto siguiente utilizaremos una tabla desordenada con permutaciones aleatorias de 1000 elementos.

5.2 Apartado 2



Tiempo medio de MergeSort (morado) y valor teórico (negro)



Caso medio, mejor y peor en obs comparado con valor teórico.

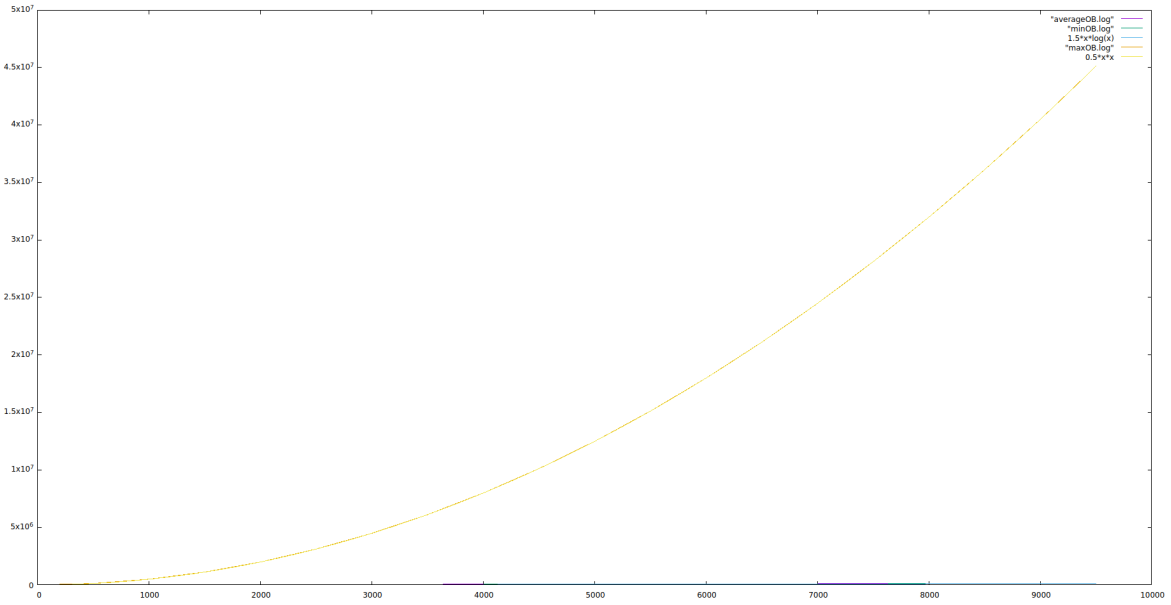
5.3 Apartado 3

Resultados del apartado 3.

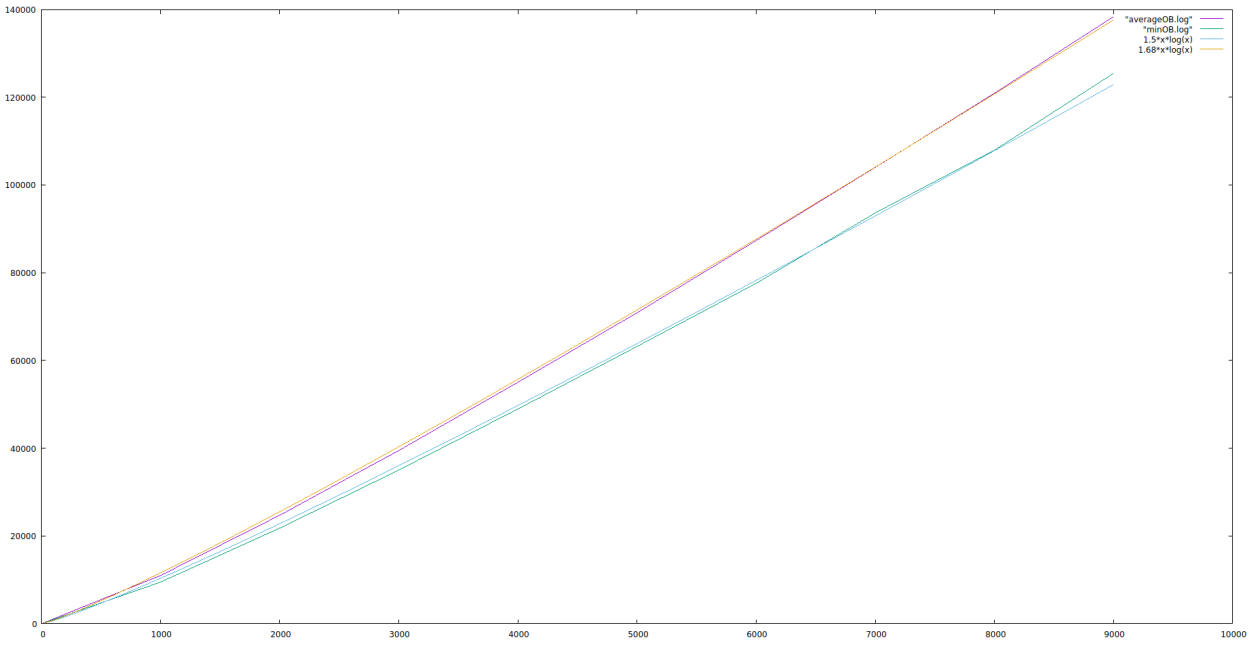
5.4 Apartado 4

Resultados del apartado 4.

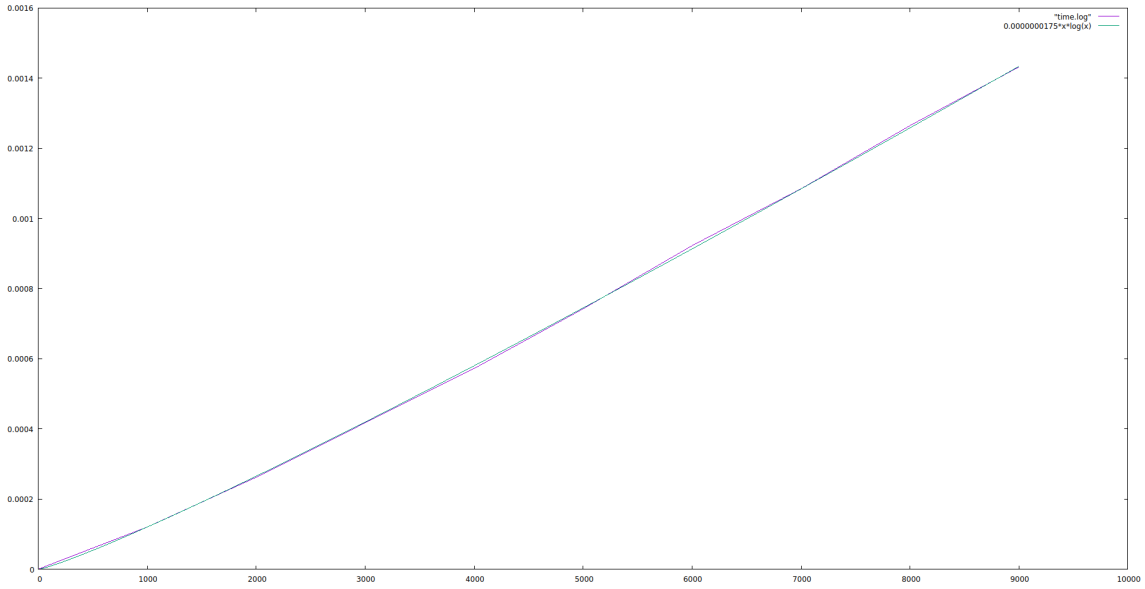
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.



Al ser el caso peor O(n²) y el caso medio O(n*logn), la diferencia en la gráfica entre el caso peor y los demás es demasiado grande como se puede apreciar. Hemos hecho otra gráfica comparando solo los valores del caso medio y el caso mejor, ajustando con nlogn



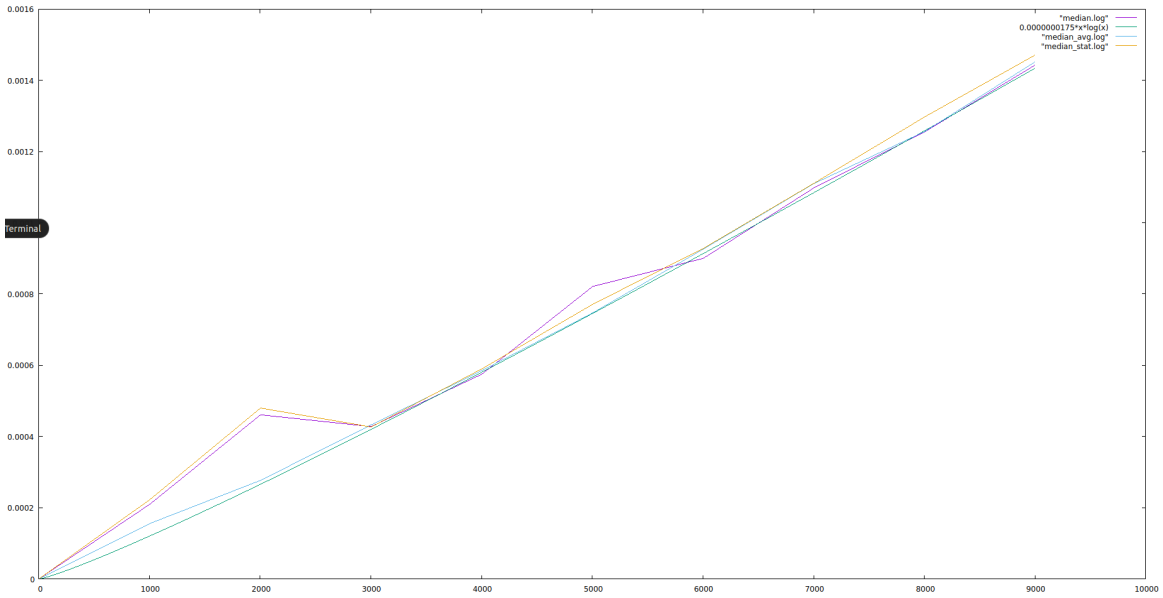
Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.



5.5 Apartado 5

Resultados del apartado 5.

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



5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

En la gráfica de MergeSort se aprecia que el tiempo medio está aproximado al valor teórico con un desvío de un 1.25, es decir, el valor real es aproximadamente 1.25*x*log(x) lo que supone que nuestra implementación es un poco más lenta que la teórica.

Como vemos en las gráficas del Quicksort, coinciden los casos medios teóricos $O(n^2)$ en caso peor, O(n*logn) en casos medios y mejor. Solamente hemos hecho ajustes lineales multiplicando por constantes para cuadrarlos.

5.2 Pregunta 2

Los resultados son muy parejos, sin embargo, observamos más picos hacia arriba usando la función median. Median_stat parece ser un poco más lenta pero tiene una recta más uniforme.

5.3 Pregunta 3

Para MergeSort el caso mejor y peor es igual al medio, O(n log n), esto es debido a que las tablas siempre serán divididas y posteriormente ordenadas da igual el tamaño, por ello el coste no varía.

El peor caso en el quicksort es cuando la lista está completamente ordenada o completamente desordenada(ordenada al revés), es mucho más lento que el caso medio ya que es $O(n^2)$, contra O(nlogn) del caso medio. El caso mejor es cuando el pivote se mantiene en el medio, es O(nlogn), es más rápido que el caso medio pero no hay tanta diferencia como ocurre con el caso peor.

Para obtener los casos medios, mejor y peor gráficamente tendremos que modificar la función del fichero times.c donde se crean los .txt a representar, para poder crear un .txt para cada algoritmo y caso.

5.4 Pregunta 4

El caso medio en ambos es O(nlogn). Sin embargo, el quicksort es algo más lento que el mergesort y utiliza más operaciones básicas de media, además de que su caso peor es mucho peor que el del mergesort. La ventaja del quicksort es el ahorro de espacio al ser in-place, en el mergesort hay que usar tablas auxiliares con memoria dinámica que en el quicksort no.

6. Conclusiones finales.

Podemos concluir, como suponíamos en la introducción, que MergeSort y QuickSort son algoritmos más rápidos que los ya estudiados InsertSort o SelectSort. Además, el MergeSort es efectivamente mejor que el QuickSort en la rapidez, por los peores casos del QuickSort que son $O(n^2)$, además de que nos sale ligeramente más rápido el MergeSort de media. El QuickSort es mejor si tenemos limitación de espacio al ejecutar el programa, ya que no crea memoria dinámica al ser in place.