Análisis de Algoritmos 2017/2018

Práctica 2

Daniel Santo-Tomás y Lucía Rivas , Grupo 1201

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica,implementaremos dos algoritmos recursivos de ordenación,basados en la técnica de divide y vencerás,para posteriormente haciendo uso de las funciones de medida de tiempos y creadas en la práctica 1,estudiar lo tiempos de ejecución y número de OB de los algoritmos con sus distintas variantes.

2. Objetivos

2.1 Apartado 1

Primeramente implementamos MergeSort. Para ello,es necesario crear la función Merge,la cual combina las subtablas ordenadas que hay dentro de la tabla principal. MergeSort debe devolver el número de OB realizadas,y ERR e caso de error. Con el algoritmo ya programado,ejecutamos el ejercicio4 de la primera práctica, modificandolo para que aplique MergeSort a una tabla cuyo tamaño pasamos como argumento.

2.2 Apartado 2

Modificando el ejercicio5 de la anterior práctica,medimos los tiempos medios de ejecución,numero de OB máximo,medio y mínimo,según distintos tamaños de tablas.

2.3 Apartado 3

En este apartado,implementamos el segundo algoritmo,QuickSort. Para su funcionamiento,es necesario programas dos funciones,partir y medio. Partir es la encargada de ordenar las subtablas de manera que ,a partir de un pivote elegido mediante la función medio,la subtabla quede con el pivote en su posición ,los elementos en posiciones menores sean más pequeños y los elementos en posiciones mayores sean más grandes. En este caso,la función medio devuelve como pivote el elemento en la primera posición.

Posteriormente,probamos el funcionamiento del algoritmo en el ejercicio4,como hicimos en el apartado 1 con MergeSort.

2.4 Apartado 4

Con el ejercicio5 medimos los tiempos y OB de QuickSort con la elección del pivote dada por medio.

2.5 Apartado 5

Implementamos otras dos funciones de elección de pivote. Medio_avg devuelve como pivote el elemento en la posición media entre el primer y el ultimo elemento((ip+iu)/2).Medio_stat devuelve ,de los valores de las posiciones ip(primera),iu(ultima) e (ip+iu)/2 (media),aquel que tiene el valor intermedio. Por ejemplo,si los valores de las posiciones ip,iu y m(medio) son 3,5 y 2,devolvería el 3;

3. Herramientas y metodología

Para la práctica hemos utilizado Atom como editor de texto en Linux,el terminal para compilar y ejecutar los programas haciendo uso del makefile dado y GNUplot para crear las gráficas,además de Valgrind para comprobar que no hay errores de memoria.

3.1 Apartado 1

Siguiendo el pseudocódigo de las transparencias de teoría,hemos implementado el algoritmo de ordenación MergeSort,que recibe la tabla a ordenar ,y los indices entre los que ha de ordenar. Devuelve ERR si hay algún problema y sino el número de OB realizadas. La condición de parada del algoritmo(es decir,en que punto ya no se hacen llamadas recursivas al algoritmo) es cuando la tabla a ordenar es de tamaño 1,es decir ,que ip = iu. En cada llamada a la función,se obtiene la posición media entre ip e iu y se llama de nuevo a MergeSort,esta vez dos veces,la primera pasándole los elementos de la tabla menores al medio(con este inclusive),y la segunda pasándole los elementos mayores al medio,generando así dos subtablas que serán ordenadas por MergeSort. Finalmente,la función Merge combina estas dos subtablas ,de manera que la tabla final está ordenada. La función devuelve el número de Obs de las dos llamadas a MergeSort y de la llamada a Merge.

La función Merge recibe tres indices,ip,iu y M,además de la tabla a ordenar. La característica de esta tabla es que esta compuesta de dos subtablas ordenadas,una a la izquierda de M(incluyéndolo) y otra a la derecha. De esta forma,Merge combina ambas tablas en una sola tabla ordenada. Para ello,crea una tabla auxiliar,compara(operación básica) los primeros elementos de ambas subtablas,y el que sea menor lo coloca en la tabla auxiliar,avanzando al siguiente elemento de la subtabla. Así sucesivamente hasta que se recorra una de las dos subtablas entera. Para terminar ,añade a la tabla auxiliar los elementos restantes de la subtabla no recorrida,para seguidamente copiar la tabla auxiliar en la original y devolver el numero de OBs .

Modificamos el ejercicio4 de la práctica 1 para probar MergeSort con una tabla de tamaño 10.

3.2 Apartado 2

Adaptando el ejercicio5 de la práctica anterior ,obtenemos datos de los tiempos de ejecución y de las OBs máximas,mínimas y medias. Para ello,contemplamos los casos de tablas de tamaño variante entre 100 y 1000,con un incremento de 10 cada vez,y estudiando para cada tamaño 100000 tablas permutadas. Los resultados quedan reflejados en las gráficas de más abajo.

3.3 Apartado 3

En este apartado implementamos el algoritmo de ordenación QuickSort. Este recibe una tabla y los indices entre os que ordenarla(ip e iu). La condición de parada es que ip sea igual a iu. Este algoritmo se basa en la función partir,la cual ordena a partir de un pivote la tabla de manera que al final ,el pivote quedará en su posición,con los elementos menores que el a la izquierda y los mayores a la derecha,no necesariamente ordenados. La función recibe la tabla,ip ,iu y la dirección de un entero donde se

guardará la posición final del pivote(&pos) ;además,devuelve el número de operaciones básicas realizadas,siendo la OB la comparación entre los elementos de la tabla y el pivote. La elección del pivote se realiza mediante la función medio,que recibe la tabla,ip,iu y el puntero al entero donde se guardará la posición del pivote(pos).En este apartado ,establece como pivote ip,devolviendo 0,ya que no se realizan OBs de más.

Tras la llamada a partir,con el correspondiente almacenamiento en un contador del numero de OBs,si la posición anterior al pivote utilizado (pos-1,) es mayor que ip,se llama a QuickSort pasando como argumentos la tabla,ip y pos – 1. Seguidamente,si la posición siguiente a la del pivote(pos + 1) es menor que iu,se llama a QuickSort pasándole la tabla,pos+ e iu. Finalmente,la función devuelve las OBs de las dos llamadas a QuickSort más las de partir.

Probamos el funcionamiento de la función modificando el ejercicio4,y haciendo que ordene una tabla de tamaño 10.

3.4 Apartado 4

Modificamos el ejercicio5 y medimos los tiempos de ejecución y las OBs máximas,mínimas y medias. Contemplamos los casos de tablas de tamaño variante entre 100 y 1000,con un incremento de 10 cada vez,y estudiando para cada tamaño 100000 tablas permutadas. Los resultados quedan reflejados en las gráficas de más abajo.

3.5 Apartado 5

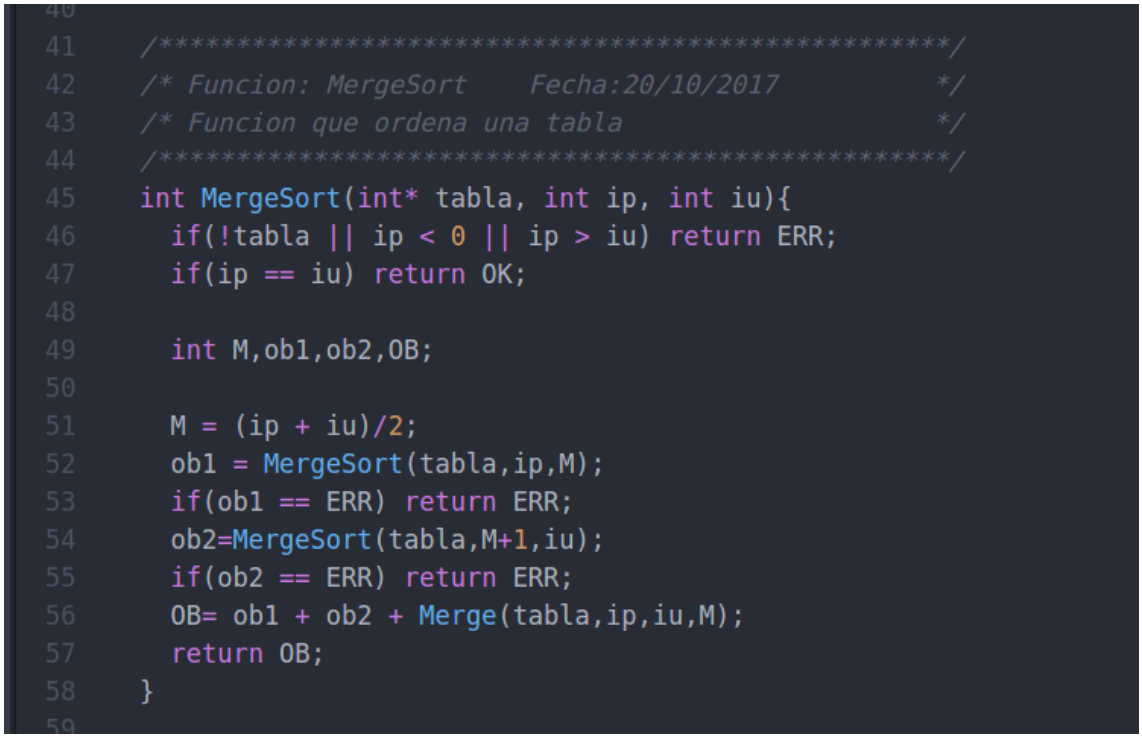
Implementamos otras dos funciones de elección del pivote para la función partir de QuickSort. La primera,medio_avg recibe la tabla,ip,iu y el puntero a las posición,y establece a través de este puntero el pivote,que en este caso es (ip + iu)/2. La función devuelve 0,ya que no realiza operaciones básicas de más. La segunda ,medio_stat,recibe la tabla,ip,iu y el puntero a la posición del pivote,y establece este como el valor intermedio entre los contenidos de ip,iu e (ip+iu)/2.Al realizar comparaciones de más (un mínimo de 2 y un máximo de 3),la función devolverá el contador de las comparaciones,que será almacenado en partir y añadido a las OB de QuickSort.

Probamos su funcionamiento modificando partir con las dos nuevas funciones y ejecutando el ejercicio 4.Posteriormente,obtenemos los datos de tiempos y OBs con el ejercicio5, representando los resultados en las gráficas de más abajo. Utilizamos el mismo numero de permutaciones,tamaño e incremento que en los aparados 2 y 4.

4. Código fuente

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

4.1 Apartado 1



```
/*****************************************************/
/* Funcion: MergeSort    Fecha:20/10/2017            */
/* Funcion que ordena una tabla                      */
/*****************************************************/
int MergeSort(int* tabla, int ip, int iu){
  if(!tabla || ip < 0 || ip > iu) return ERR;
  if(ip == iu) return OK;

  int M,ob1,ob2,OB;

  M = (ip + iu)/2;
  ob1 = MergeSort(tabla,ip,M);
  if(ob1 == ERR) return ERR;
  ob2=MergeSort(tabla,M+1,iu);
  if(ob2 == ERR) return ERR;
  OB= ob1 + ob2 + Merge(tabla,ip,iu,M);
  return OB;
}
```

```c
/*****************************************************/
/* Funcion: Merge   Fecha:20/10/2017                 */
/* Funcion que combina los dos lados de una tabla    */
/*  ambos ordenados                                  */
/*****************************************************/
int Merge(int* tabla, int ip, int iu, int imedio){
  if(!tabla) return ERR;

  int i, j, k,cont = 0;
  int *aux;
  aux = (int*)malloc(sizeof(int)*(iu-ip+1));
  if(!aux) return ERR;

  i = ip;
  j = imedio + 1;
  k = ip;

  while(i <= imedio && j <= iu){
    cont++;
    if (tabla[i] < tabla[j]){
      aux[k - ip] = tabla[i];
      i++;
    }
    else {
      aux[k - ip] = tabla[j];
      j++;
    }
    k++;
  }

  if (i > imedio){
    while(j <= iu) {
      aux[k - ip] = tabla[j];
      j++;
      k++;
    }
  }

  else if ( j > iu){
    while(i <= imedio){
      aux[k - ip] = tabla[i];
      i++;
      k++;
    }
  }

  for(i = ip; i <= iu; i++){
    tabla[i] = aux[i - ip];
  }
  free(aux);
  return cont;
}

```

4.3 Apartado 3

```c
/*************************************************/
/* Funcion: QuickSort   Fecha:20/10/2017         */
/* Funcion que ordena una tabla                  */
/*************************************************/

int QuickSort(int* tabla, int ip, int iu){
  if(!tabla || ip < 0 || ip > iu) return ERR;
  if(ip == iu) return OK;
  int OB = 0,pos = 0,ob1 = 0,ob2 = 0;


  OB += partir(tabla, ip, iu, &pos);
  if(OB == ERR) return ERR;

  if(ip < pos-1) ob1 = QuickSort(tabla, ip, pos-1);
  if(ob1 == ERR) return ERR;
  if(pos+1 < iu) ob2 = QuickSort(tabla, pos+1, iu);
  if(ob2 == ERR) return ERR;

  OB+=(ob1 + ob2);

  return OB;
}
```

```c
/*************************************************/
/* Funcion: Partir      Fecha:20/10/2017         */
/* Funcion que parte una tabla para QuickSort    */
/*************************************************/

int partir(int *tabla, int ip, int iu, int *pos){
  if(!tabla || ip < 0 || ip > iu || !pos) return ERR;
  int k, cont = 0, aux, i;

  cont =  medio_avg(tabla, ip, iu, pos);
  if(cont == ERR) return ERR;

  k = tabla[*pos];
  aux = tabla[ip];
  tabla[ip] = tabla[*pos];
  tabla[*pos] = aux;

  *pos = ip;
  for(i = ip+1; i <= iu; i++){
    cont++;
    if(tabla[i] < k){
      (*pos)++;
      aux = tabla[i];
      tabla[i] = tabla[*pos];
      tabla[*pos] = aux;
    }
  }
  aux = tabla[ip];
  tabla[ip] = tabla[*pos];
  tabla[*pos] = aux;
  return cont;
}
```

```c
/***************************************************/
/* Funcion: Medio       Fecha:20/10/2017           */
/* Funcion que establece un pivote para QuickSort  */
/***************************************************/
int medio(int *tabla, int ip, int iu, int *pos){
  if(!tabla || ip < 0 || ip > iu || !pos) return ERR;
  *pos = ip;
  return 0;
}
```

4.5 Apartado 5

```c
/***************************************************/
/* Funcion: Medio_avg       Fecha:20/10/2017       */
/* Funcion que establece un pivote para QuickSort  */
/***************************************************/
int medio_avg(int *tabla, int ip, int iu, int *pos){
  if(!tabla || ip < 0 || ip > iu || !pos) return ERR;
  *pos = (ip + iu)/2;
  return 0;
}
```

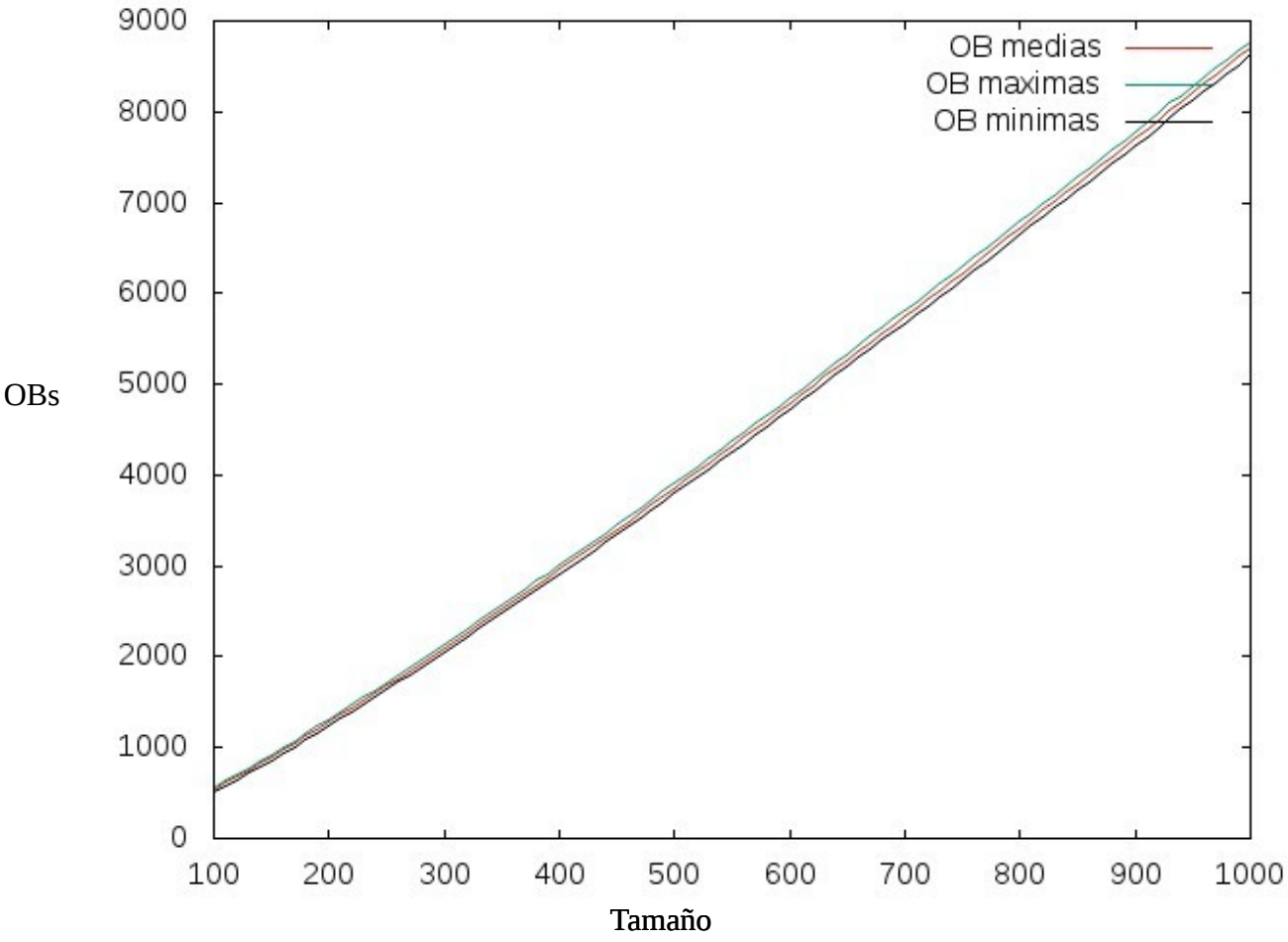
```c
/*****************************************************/
/* Funcion: Medio_stat      Fecha:20/10/2017         */
/* Funcion que establece un pivote para QuickSort   */
/*****************************************************/
int medio_stat(int *tabla, int ip, int iu, int *pos){
  if(!tabla || ip < 0 || ip > iu || !pos) return ERR;
  int m,cont = 0;
  m = (ip + iu)/2;

/*Cuando el primero es más pequeño que el último*/
  cont++;
  if(tabla[ip] < tabla[iu]){
    cont++;
    if(tabla[iu] < tabla[m]){/*ip < iu < m */
      *pos = iu;
      return cont;
    }
    cont++;
    if(tabla[m] < tabla[ip]){ /*m < ip < iu*/
       *pos = ip;
       return cont;
    }
    cont++;
    *pos = m;     /*ip < m < iu*/
    return cont;
  }
  /*Cuando el último es más pequeño que el primero*/
  cont++;
  if(tabla[iu] < tabla[ip]){
    cont++;
    if(tabla[m] < tabla[iu]){     /* m < iu < ip*/
      *pos = iu;
      return cont;
    }
    cont++;
    if(tabla[ip] < tabla[m]){ /* iu < ip < m*/
      *pos = ip;
      return cont;
    }
    cont++;
    *pos = m;    /* iu < m < ip*/
    return cont;
  }
  return ERR;
}
```

5. Resultados, Gráficas

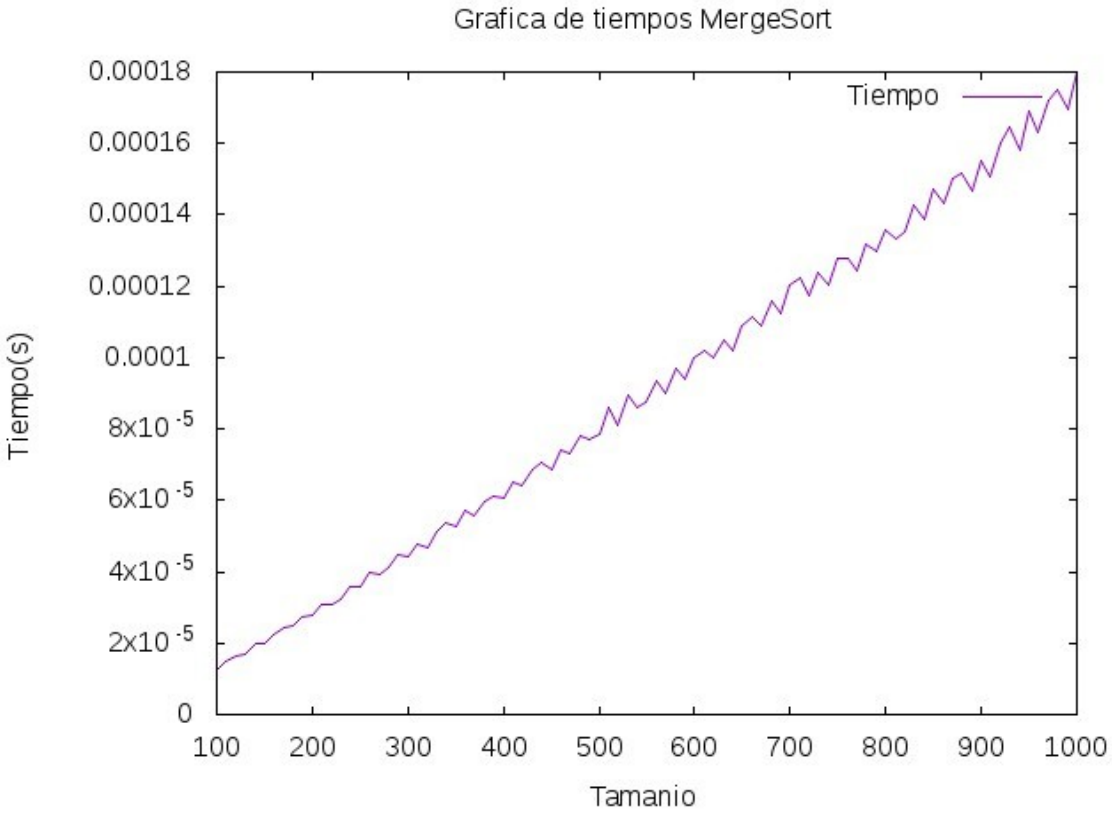
5.1 Apartado 1

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica2$
./ejercicio4 -tamanio 10
Practica numero 2, apartado 1
Realizada por:  Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
0       1       2       3       4       5       6       7       8       9
```

5.2 Apartado 2



Como se puede observar,las tres curvas(OBs máximas,OBs mínimas y OBs medias) tienen una pendiente de crecimiento prácticamente igual,aumentando sus valores a medida que aumenta el tamaño de las tablas y adaptándose a lo visto en teoría.

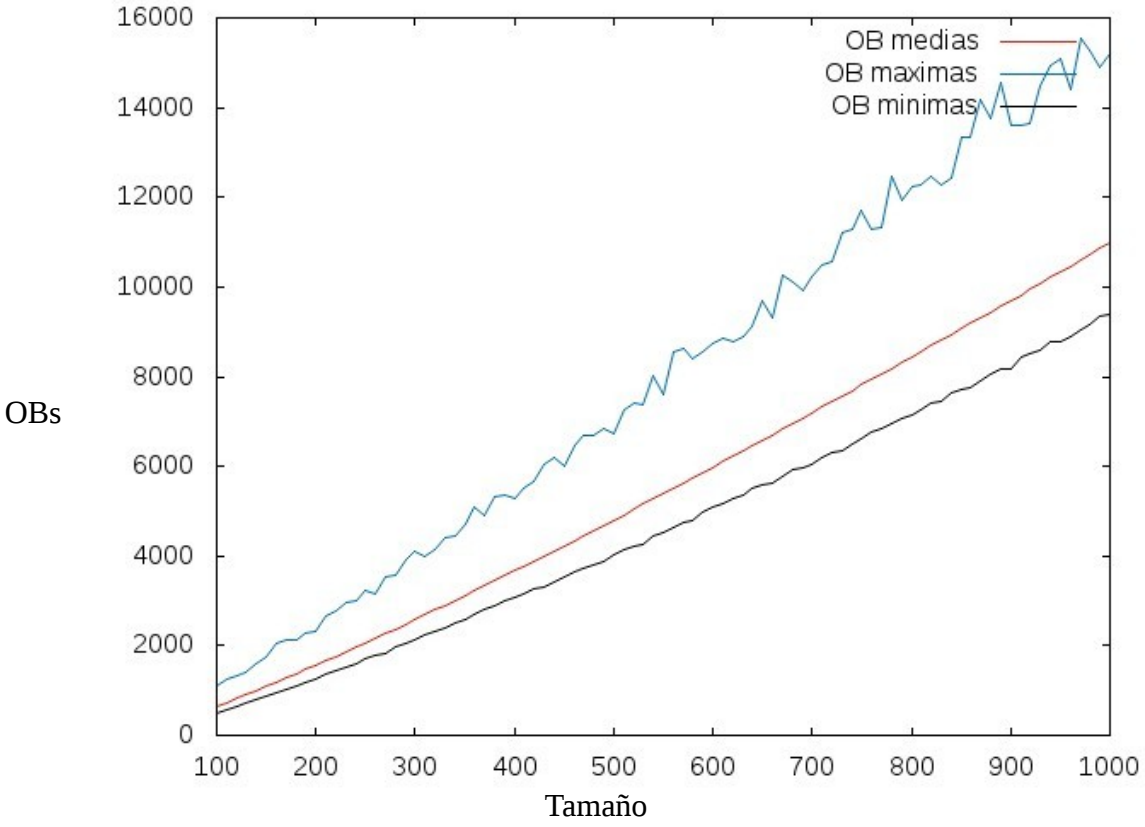


A medida que aumenta el tamaño de las tablas,se hacen más comparaciones para ordenarlas,por lo tanto se emplea más tiempo,tal y como refleja la gráfica.

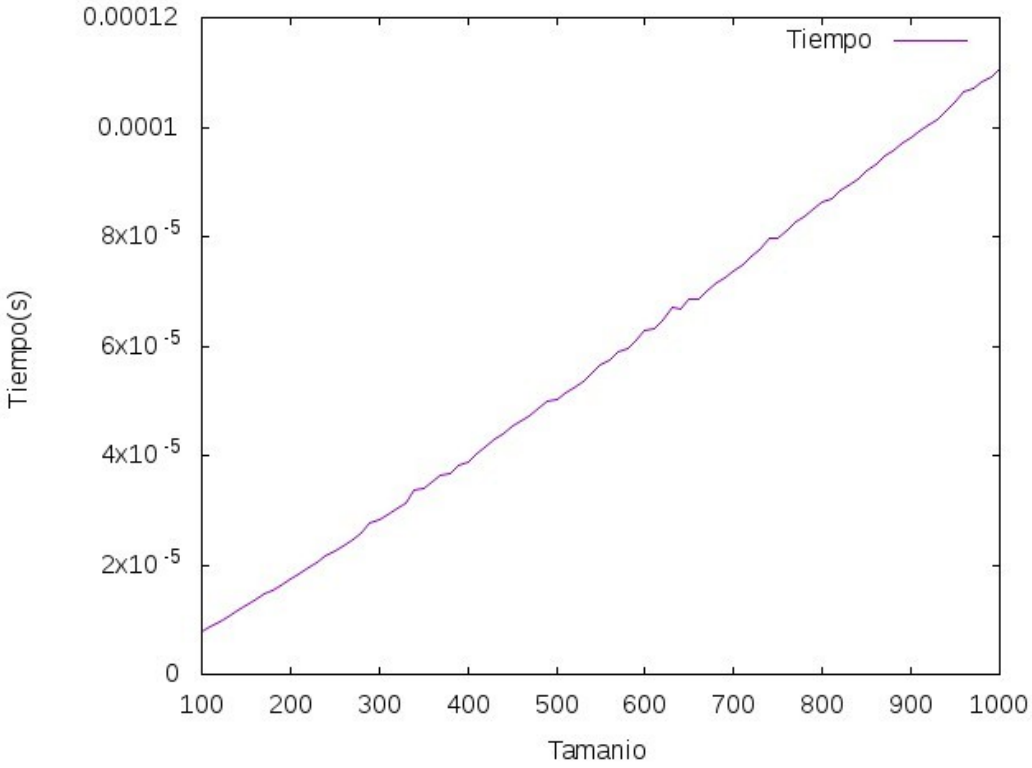
5.3 Apartado 3

```
danist@danist-Lenovo-E51-80:~/Documentos/UAM/PRACTICAS ANALISIS/ANAL/practica2$
./ejercicio4 -tamanio 10
Practica numero 2, apartado 3
Realizada por:  Lucía Rivas y Daniel Santo-Tomás
Grupo: 1201
0       1       2       3       4       5       6       7       8       9
```

5.4 Apartado 4

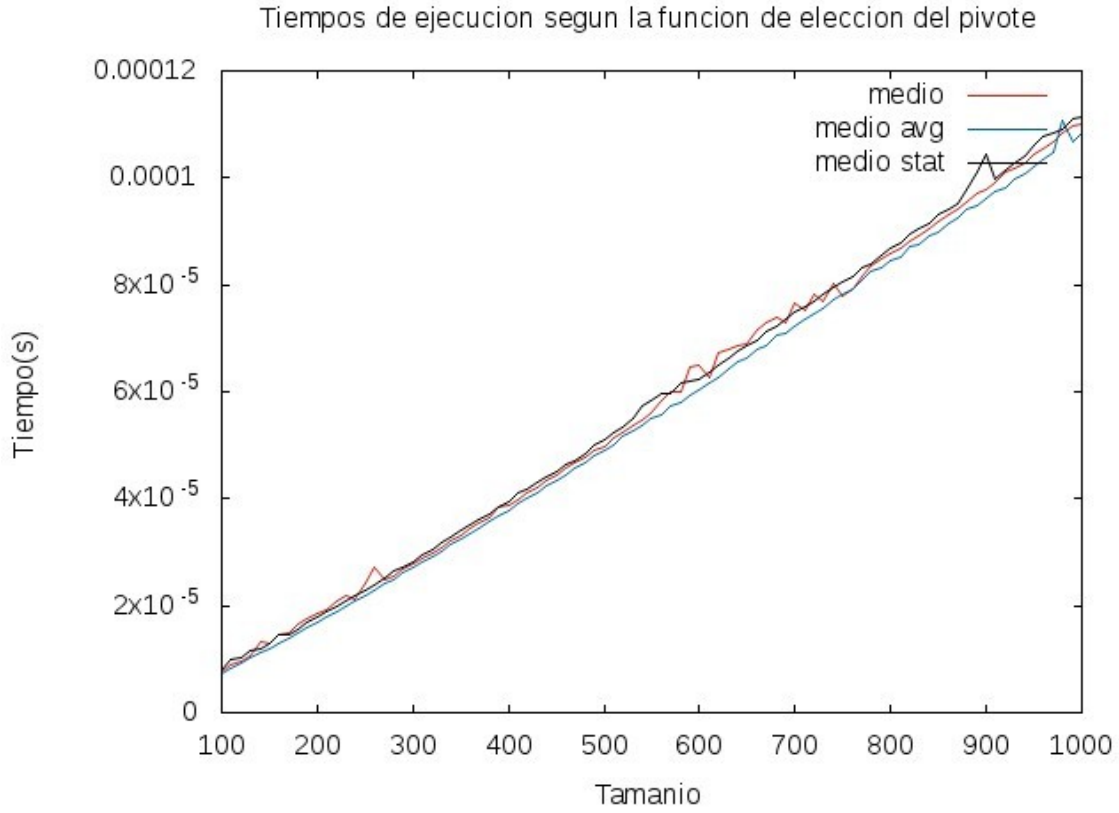


Se puede observar claramente una gran diferencia entre el numero medio y el numero máximo. Esto se debe a que en QuickSort el caso peor es (N^2)/2 – N/2 ,mientras que el medio es 2N * log(N) + O(N) .Entre estos dos valores hay una diferencia importante que se ve reflejada en el gráfico.



El tiempo de ejecución aumenta a medida que aumenta el tamaño de las tablas.

5.5 Apartado 5



Los tiempos son similares para los tres pivotes,pero se observa que el peor pivote es medio stat,mientras que el mejor es medio avg. El pivote determinado por medio en ocasiones es peor que medio_stat,pero en la mayor parte de la gráfica es mejor,pero nunca mejorando medio_avg.

5. Respuesta a las preguntas teóricas.

5.1 Compara el rendimiento empírico 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 es Θ(N log(N)),y en la gráfica de OBs se puede observar que el caso medio tiene una pendiente del tipo N log N.

El caso medio en QuickSort es 2N * log(N) + O(N) ,y en la gráfica se puede ver una pendiente menos pronunciada que la de N log N,esto se debreá a que los valores a los que refiere O(N) haran disminuir la pronunciación de esa gráfica.

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

La elección del elemento intermedio como pivote da resultados mejores que elegir el primero,ya que en general esto produce que los elementos de la tabla se encuentren en su posición antes,realizando menos comparaciones. Medio_stat es el pero ya que es casi como elegir un número aleatorio como pivote,así que depende de cual salga como pivote,dará mejores o peores resultados.

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)?

El caso mejor de MergeSort es mayor o igual que (½) N log(N),y el caso peor es menor o igual que N log(N) + O(N).

El caso mejor de QuickSort es mayor o igual que Θ(N log(N) y el caso peor es menor o igual que $N^2/2 – N/2$.

Habría que introducir las tablas en las que se sabe que se dan el caso pero y el caso mejor,y comprobar las OBs resultantes.Para el caso medio,lo mejor es o que hemos hecho,hacer varios casos y obtener la media.

5.4 ¿Cuál de los dos algoritmos estudiados es más eficiente empíricamente? 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.

MergeSort es menos eficiente que QuickSort,ya que tarda más tiempo en ordenar,debido a que sus operaciones básicas son más costosas que las de QuickSort.

Desde el punto de vista de la memoria ,QuickSort es claramente mejor,ya que MergeSort hace uso de memoria dinámica en su función Merge.

6. Conclusiones finales.

En esta práctica hemos podido estudiar a fondo el funcionamiento de los algoritmos de ordenación recursivos,no solo programando y entendiendo su funcionamiento y sus diversas funciones internas,sino también estudiando y comparando sus resultados a nivel de eficiencia. En el caso de QuickSort,hemos podido apreciar las significativas diferencias entre sus distintos casos y sus distintos pivotes,obteniendo conclusiones de cual es mejor para la ordenación y por lo tanto cual conviene implementar.