Análisis de Algoritmos 2015/2016

Práctica 2

Litzy Tatiana Rochas Ávila y Lucía Colmenarejo Pérez, 1201.

Código	Gráficas	Memoria	Total

1. Introducción y objetivos:

En esta segunda práctica vamos a ver el funcionamiento de los algortimos propios de la metodología de *divide y vencerás*: MergeSort y QuickSort. Nuestro objetivo va a ser determinar experimentalmente los tiempos de ejecución de estos dos algoritmos sobre tablas de diferentes tamaños.

En el primer apartado hemos implementado el algoritmo de ordenación de MergeSort . Este método de ordenación consiste en dividir la lista desordenada en dos subtablas (aproximadamente la mitad de tamaño) y ordenar a su vez cada subtabla de manera recursiva mediante una función que llamaremos *mergesort_re* que será llamada desde la función principal *mergesort*(a al cual introducimos al principio la tabla y el primer y último elemento de ella).A esta función le hemos pasado la tabla, el primer, el último y el punto medio de la tabla. *Mergesort_re* también a su vez llama a una función llamada *merge* y que lo que nos devuelve es el número de comparaciones de claves realizadas durante la ordenación (además de conseguir ordenar la tabla). Finalmente unimos las subtablas ya ordenadas para conseguir la tabla entera ordenada. Para ver que funciona bien el nuevo método de ordenación implementado lo que hemos hecho ha sido cambiar el ejercicio4.c implementado en la práctica anterior para que nos devuelva la tabla ordenada. Para ello lo que le pasamos como retorno es el resultado de realizar este método sobre una tabla generada mediante la función ya implementada en la práctica1 de *genera_perm*, como primer elemento establecemos el 0 y como último introducimos el tamaño de la tabla menos 1.

En el apartado segundo lo que hemos hecho ha sido modificar el programa ejercicio5.c de la práctica para obtener la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Posteriormente lo que hemos hecho ha sido representar estos valores y compararlos con su resultado teórico (hecho más adelante).

En el tercer apartado lo que hemos hecho ha sido implementar el método de ordenación QuickSort. Este método de ordenación comienza eligiendo un elemento de la lista o tabla al cual llamaremos pivote. Este elemento suele ser por lo general el primero, el último o el elemento medio de la tabla, aunque en este apartado el pivote que elegimos es el primero. Para ello hemos implementado la función *medio* a la cual le pasamos la tabla, el primer elemento de la lista, el último elemento de la lista y el puntero *pos al cual atribuiremos el valor de nuestro pivote. El paso siguiente es reordenar los elementos de la lista a cada lado del pivote. Así la lista inicial queda dividida en dos sublistas mediante la función *partir*(la del lado derecho del pivote y la

del lado izquierdo); y este proceso lo repetiremos de manera recursiva con cada una de las sublistas mientras que tengan más de un elemento. Para ello hemos implementado una función recursiva la cual es llamada desde *quicksort*. Esta función se llama *quicksort_re* y le pasamos como argumentos: la tabla a ordenar, el primer elemento de la tabla, el último elemento de la tabla y un puntero a entero en el cual guardaremos el contador de operaciones básicas .Una vez terminado este proceso ya tendremos todos los elementos ordenados. Para ver que funciona de un manera correcta lo que hemos hecho ha sido cambiar el ejercicio4.c implementado en la práctica anterior para que nos devuelva la tabla ordenada. Para ello lo que le pasamos como retorno es el resultado de realizar este método sobre una tabla generada mediante la función ya implementada en la práctica1 de *genera_perm*, como primer elemento establecemos el 0 y como último introducimos el tamaño de la tabla menos 1.

En el cuarto apartado lo que hemos hecho ha sido modificar el ejercicio5.c de igual forma que lo hicimos en el segundo apartado de la práctica para obtener la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Posteriormente lo que hemos hecho ha sido representar estos valores y compararlos con su resultado teórico (hecho más adelante).

En el quinto y último apartado lo que hemos hecho ha sido implementar otra función que lo que devuelva sea la posición media de la tabla como pivote. Esta función se llama *medio_avg*, y lo que le pasamos es la tabla, el primer y último elemento de la tabla y un puntero a entero donde guardaremos el pivote. Además hemos implementado otra función llamada *medio_stat*, a la cual le pasamos los mismos elementos que a *medio_avg* y lo que hace es devolver la posición que contenga el valor intermedio entre el primer, el último y el elemento medio de la tabla. Una vez implementadas estas funciones lo que hemos tenido que hacer ha sido cambiar la función partir para que utilice estas funciones pivote y acumule también las OBs básicas adicionales que realiza *medio_stat*. Finalmente, en este apartado lo que hemos hecho ha sido comparar los tiempos promedio de ejecución, máximo, mínimo y promedio de operaciones básicas obtenidos con cada una de las tres rutinas pivote implementadas (hecho más adelante).

2. Herramientas y metodología:

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

2.1 Apartado 1:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos). Además hemos empleado la herramienta de Gnuplot para la realización del histograma.

La solución que hemos propuesto para este apartado es la implementación de la función *mergesort_re* (se encarga de dividir de forma recursiva la tabla inicial)que es llamada desde la función principal *MergeSort* y que esta función a su vez llama a otra función, *merge*(nos devolverá el número de OBs realizadas, así como la tabla ordenada). Adicionalmente hemos tenido que cambiar el ejercicio4.c para que utilice como método de ordenación el *MergeSort.*

2.2 Apartado 2:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos). Además hemos utilizado el programa Gnuplot para la realización de la gráfica de los tiempos mínimo, máximo y promedio de operaciones básicas y poder comparar esos valores con los valores teóricos.

La solución adoptada para este apartado es una solución que nos servirá posteriormente para los siguientes apartados. En este apartado hemos introducido una nueva variable llamada algoritmo, y dependiendo del valor que le demos a esta variable se medirá el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación utilizando un algoritmo distinto. En este caso, tenemos que medirlo utilizando el algoritmo *MergeSort*, por lo que el valor de algoritmo será 1.

2.3 Apartado 3:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución por la que hemos optado en este problema es la realización de una función recursiva *quicksort_re* que es llamada desde la función principal *QuickSort*. Además hemos implementado otra función llamada *partir* que se encarga de dividir la tabla teniendo en cuenta el pivote que hemos escogido. Otra función que hemos tenido que implementar es la llamada *medio*, que devuelve como pivote el primer elemento de la lista. Adicionalmente hemos tenido que cambiar el ejercicio4.c para que utilice como método de ordenación el *QuickSort.*

2.4 Apartado 4:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos).

La solución adoptada para este apartado es cambiar el ejercicio5.c. En este apartado hemos utilizado método explicado en el apartado 2 de la variable algoritmo para medir el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. En este caso, tenemos que medirlo utilizando el algoritmo *QuickSort*, por lo que el valor de algoritmo será 2.

2.5 Apartado 5:

Este apartado lo hemos realizado en el entorno de desarrollo de Linux y para programarlo hemos utilizado la herramienta de Cloud 9 (para poder trabajar ambos miembros del grupo sobre el mismo programa desde lugares distintos). Además hemos utilizado el programa Gnuplot para la realización de la gráfica de los tiempos mínimo, máximo y promedio de operaciones básicas y poder comparar esos valores con los valores teóricos.
La solución que hemos implementado en este apartado es la realización de dos funciones para obtener como pivotes el valor último y el medio de la tabla (*medio_stat* y *medio_avg*). De esta forma hemos tenido que modificar la rutina de partir para que utilice etas nuevas funciones pivote implementadas y acumule las OBs básicas adicionales de estas funciones. Además hemos comparado los tiempos promedio de ejecución, máximo, mínimo y promedio de operaciones básicas obtenidos con cada una de las tres rutinas pivote implementadas.

Cabe destacar que en todos los apartados hemos utilizado las herramientas de valgrind y gcc para la detección de errores y el compilado de los programas. Además en ciertos apartados hemos tenido que emplear el debbuger para encontrar el punto en el cual se encuentra el error y así poder arreglarlo, y, además nos ha ayudado bastante a la comprensión del funcionamiento del algoritmo.

3.Código fuente:

Aquí ponemos el código fuente exclusivamente de las rutinas que hemos desarrollado nosotros en cada apartado.

Apartado1:

```
/************************************************/
/* Funcion:mergesort      Fecha:21/10/2016                */
/* Autores:Lucia Colmenarejo Pérez y                      */
/* Litzy Tatiana Rocha Avila                              */
/* Función que dada una tabla desordenada la              */
/* ordena de menor a mayor                                */
/* Entrada:                                               */
/* int* tabla:Tabla desordenada a ordenar                 */
/* int ip:primer elemento de la tabla                     */
/* int iu:ultimo elemento de la tabla                     */
/* Salida:                                                */
/* Contador:Devuelve el número de comparación de          */
/* claves realizadas durante la ordenación                */
```

```
/* -1: En caso de error                                          */
/*************************************************/
int Mergesort(int* tabla, int ip, int iu){
   int  contador=0;
   if(!tabla||ip<0||iu<0) return ERR;
   mergesort_re(tabla, ip, iu, &contador);
   return contador;
}
/*************************************************/
/* Funcion:mergesort_re   Fecha:21/10/2016                       */
/* Autores:Lucia Colmenarejo Pérez y                             */
/* Litzy Tatiana Rocha Avila                                     */
/* Función que dada una tabla desordenada la                     */
/* ordena de menor a mayor                                       */
/* Entrada:                                                      */
/* int* tabla:Tabla desordenada a ordenar                        */
/* int ip:primer elemento de la tabla                            */
/* int iu:ultimo elemento de la tabla                            */
/* Salida:                                                       */
/* Contador:Devuelve el número de comparación de                 */
/* claves realizadas durante la ordenación                       */
/* -1: En caso de error                                          */
/*************************************************/
int mergesort_re(int * tabla, int ip, int iu, int* contador){
   int imedio=0;
    if(ip>iu){
      return ERR;
   }
   else if(ip==iu){
      return 0;
   }
   imedio=(iu+ip)/2;
   mergesort_re(tabla,ip,imedio, contador);
   mergesort_re(tabla,imedio+1,iu, contador);
   (*contador)+=merge(tabla,ip,iu,imedio);
   return *contador;
}

/*************************************************/
```

```
/* Funcion:merge      Fecha:21/10/2016                          */
/* Autores:Lucia Colmenarejo Pérez y                            */
/* Litzy Tatiana  Rocha Avila                                   */
/* Función que dada una tabla desordenada la                    */
/* ordena de mayor a menor                                      */
/* Entrada:                                                     */
/* int* tabla:Tabla desordenada a ordenar                       */
/* int ip:primer elemento de la tabla                           */
/* int iu:ultimo elemento de la tabla                           */
/* Salida:                                                      */
/* Contador:Devuelve el número de comparación de                */
/* claves realizadas durante la ordenación                      */
/* -1: En caso de error                                         */
/************************************************/
int merge(int* tabla, int ip, int iu, int imedio){
   int* tablaAux=NULL;
   int i=0,j=0,k=0, contador=0;

 if(!tabla || ip<0 || iu<0 || imedio<0){
      return ERR;
   }
   tablaAux=(int *)malloc(sizeof(int)*(iu-ip+1));
   if(tablaAux==NULL)return ERR;

   for(i=ip, j=imedio+1, k=0; i<=imedio && j<=iu; k++, contador++){
       if(tabla[i]<tabla[j]){
          tablaAux[k]=tabla[i];
          i++;
       }
       else{
          tablaAux[k]=tabla[j];
           j++;
       }
   }
   if(i>imedio){
```

```
        for( ; j<=iu;j++, k++){
            tablaAux[k]=tabla[j];
        }
    }
    else if(j>iu){
        for( ;i<=imedio;i++,k++){
            tablaAux[k]=tabla[i];
        }
    }
    for(i=ip;i<=iu;i++){
        tabla[i]=tablaAux[i-ip];
    }
    free(tablaAux);
  return contador;
}
```

Apartado2:

```
/*************************************************************/
/* Programa: ejercicio5                Fecha:13/10/2016      */
/* Autores:Lucia Colmenarejo Pérez y                         */
/* Litzy Tatiana Rocha Avila                                 */
/* Programa que escribe en un fichero                        */
/* los tiempos medios del algoritmo de                       */
/* ordenacion  por  Selección                                */
/* Entrada: Linea de comandos                                */
/* -num_min: numero minimo de elementos de la tabla          */
/* -num_max: numero minimo de elementos de la tabla          */
/* -incr: incremento\n                                       */
/* -numP: Introduce el numero de permutaciones a promediar   */
/* -fichSalida: Nombre del fichero de salida                 */
/* Salida: 0 si hubo error                                   */
/* -1 en caso contrario                                      */
/*************************************************************/
int main(int argc, char** argv)
{
```

```
  int i, num_min, num_max, incr, n_perms, algoritmo;
  char nombre[256];
  short ret;

  srand(time(NULL));

  if (argc != 13) {
    fprintf(stderr, "Error en los parametros de entrada:\n\n");
    fprintf(stderr, "%s -num_min <int> -num_max <int> -incr <int>\n", argv[0]);
    fprintf(stderr, "\t\t -numP <int> -fichSalida <string> \n");
    fprintf(stderr, "Donde:\n");
    fprintf(stderr, "-num_min: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-num_max: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-incr: incremento\n");
    fprintf(stderr, "-numP: Introduce el numero de permutaciones a promediar\n");
    fprintf(stderr, "-fichSalida: Nombre del fichero de salida\n");
    fprintf(stderr, "-pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre
las tres\n");
    exit(-1);
  }

  printf("Practica numero 1, apartado 5\n");
  printf("Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila\n");
  printf("Grupo: 1201\n");

  /* comprueba la linea de comandos */
  for(i = 1; i < argc ; i++) {
    if (strcmp(argv[i], "-num_min") == 0) {
      num_min = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-num_max") == 0) {
      num_max = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-incr") == 0) {
      incr = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-numP") == 0) {
      n_perms = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-fichSalida") == 0) {
      strcpy(nombre, argv[++i]);
    } else if (strcmp(argv[i], "-pivote") == 0) {
      algoritmo=atoi(argv[++i]);
    } else {
      fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
```

```
    }
  }

  /* calculamos los tiempos */
  if(algoritmo==0){
    ret = genera_tiempos_ordenacion(InsertSort, nombre,num_min, num_max,incr, n_perms, algoritmo);
  }
  else if(algoritmo==1){
    ret = genera_tiempos_ordenacion(MergeSort, nombre ,num_min , num_max , incr, n_perms ,algoritmo);
  }
  else if(algoritmo==2){
  ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min , num_max , incr , n_perms, algoritmo);
  }
 else if(algoritmo==3){
    ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min, num_max, incr, n_perms, algoritmo);
 }
 else if(algoritmo==4){
    ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min, num_max, incr, n_perms, algoritmo);
 }
else{
  printf("Número del algoritmo incorrecto");
  exit(-1);
}
  if (ret == ERR) { /* ERR_TIME debera ser un numero negativo */
    printf("Error en la funcion Time_Ordena\n");
    exit(-1);
  }
  printf("Salida correcta \n");


  return 0;
}
```

Apartado3:

```
/*************************************************/
/* Funcion:medio      Fecha:08/11/2016           */
/* Autores:Lucia Colmenarejo Pérez y             */
/* Litzy Tatiana Rocha Avila                     */
/* Función que devuelve como pivote el primer    */
/* elemento de la tabla                          */
/* Entrada:                                      */
/* int* tabla:Tabla desordenada a ordenar        */
```

```
/* int ip:primer elemento de la tabla                    */
/* int iu:ultimo elemento de la tabla                    */
/* int* pos: posición del pivote                         */
/* Salida:                                               */
/* Contador:Devuelve el número de comparación de         */
/* claves realizadas durante la ordenación               */
/* -1: En caso de error                                  */
/*******************************************************/
int medio(int *tabla, int ip, int iu, int* pos){
   if(!tabla || ip<0 || iu<0 || !pos)return ERR;
   *pos=ip;
   return 0;
}
/*******************************************************/
/* Funcion:partir        Fecha:08/11/2016                */
/* Autores:Lucia Colmenarejo Pérez y                     */
/* Litzy Tatiana Rocha Avila                             */
/* Función que divide la tabla en funcion del            */
/* pivote escogido                                       */
/* Entrada:                                              */
/* int* tabla:Tabla desordenada a ordenar                */
/* int ip:primer elemento de la tabla                    */
/* int iu:ultimo elemento de la tabla                    */
/* int* pos: posición del pivote                         */
/* int algoritmo: elegimos qué tipo de pivote            */
/* queremos llamando a las funciones                     */
/* implementadas anteriormente                           */
/* Salida:                                               */
/* Contador:Devuelve el número de comparación de         */
/* claves realizadas durante la ordenación               */
/* -1: En caso de error                                  */
/*******************************************************/
int partir(int* tabla, int ip, int iu, int* position, int algoritmo){
   int contador=0;
   int k=0;
   int i=0;
  if(algoritmo==2){
   if(medio(tabla,ip,iu,position)==ERR)return ERR;
  }
  else if(algoritmo==3){
```

```
    if(medio_avg(tabla,ip,iu,position)==ERR)return ERR;
  }
  else{
      medio_stat(tabla, ip, iu, position);
  }
   k=tabla[*position];
   intercambiar(&tabla[ip],&tabla[*position]);
   *position=ip;
   for(i=ip+1; i<=iu ;i++){
      if(tabla[i]<k){
        (*position)++;
         intercambiar(&tabla[i],&tabla[*position]);
      }
      contador++;
   }
    intercambiar(&tabla[ip], &tabla[*position]);
    return contador;
}



/**************************************************/
/* Funcion:quicksort_re   Fecha:21/10/2016        */
/* Autores:Lucia Colmenarejo Pérez y              */
/* Litzy Tatiana Rocha Avila                      */
/* Función que dada una tabla desordenada la      */
/* ordena de menor a mayor                        */
/* Entrada:                                       */
/* int* tabla:Tabla desordenada a ordenar         */
/* int ip:primer elemento de la tabla             */
/* int iu:ultimo elemento de la tabla             */
/* int algoritmo: elegimos qué tipo de pivote     */
/* queremos llamando a las funciones              */
/* implementadas anteriormente                    */
/* Salida:                                        */
/* Contador: Devuelve el número de comparación de */
/* claves realizadas durante la ordenación        */
/* -1: En caso de error                           */
/**************************************************/

int quicksort_re(int *tabla, int ip, int iu, int *contador, int algoritmo){
```

```
    int m = 0;
    int n=0;
    if(ip>iu){
        return ERR;
    }
    if(ip==iu){
        return 0;
    }
    else{
        m=partir(tabla, ip, iu, &n, algoritmo);
        (*contador)+=m;

        if(m==ERR){
            return ERR;
        }
        if(ip<n-1){
            quicksort_re(tabla, ip, n-1, contador,algoritmo);
        }
        if(n+1<iu){
            quicksort_re(tabla, n+1, iu, contador, algoritmo);
        }
    }
        return *contador;
}

/***************************************************/
/* Funcion: QuickSort      Fecha:21/10/2016        */
/* Autores:Lucia  Colmenarejo  Pérez  y             */
/* Litzy Tatiana Rocha Avila                       */
/* Función que dada una tabla desordenada la       */
/* ordena de menor a mayor                         */
/* Entrada:                                        */
/* int* tabla:Tabla desordenada a ordenar          */
/* int ip:primer elemento de la tabla              */
/* Salida:                                         */
/* Contador:Devuelve el número de comparación de   */
/* claves realizadas durante la ordenación         */
/* -1: En caso de error                            */
/***************************************************/
```

```
int QuickSort(int* tabla, int ip, int iu, int algoritmo){
   int contador=0;
   quicksort_re(tabla, ip, iu, &contador, algoritmo);
   return contador;
}
```

Apartado4:

```
/*****************************************************/
/* Programa: ejercicio5              Fecha:13/10/2016  */
/* Autores:Lucia Colmenarejo Pérez y                   */
/* Litzy Tatiana Rocha Avila                           */
/* Programa que escribe en un fichero                  */
/* los tiempos medios del algoritmo de                 */
/* ordenacion  por  Selección                          */
/* Entrada: Linea de comandos                          */
/* -num_min: numero minimo de elementos de la tabla    */
/* -num_max: numero minimo de elementos de la tabla    */
/* -incr: incremento\n                                 */
/* -numP: Introduce el numero de permutaciones a promediar */
/* -fichSalida: Nombre del fichero de salida           */
/* Salida: 0 si hubo error                             */
/* -1 en caso contrario                                */
/*****************************************************/
int main(int argc, char** argv)
{
  int i, num_min, num_max, incr, n_perms, algoritmo;
  char nombre[256];
  short ret;

  srand(time(NULL));

  if (argc != 13) {
    fprintf(stderr, "Error en los parametros de entrada:\n\n");
    fprintf(stderr, "%s -num_min <int> -num_max <int> -incr <int>\n", argv[0]);
    fprintf(stderr, "\t\t -numP <int> -fichSalida <string> \n");
    fprintf(stderr, "Donde:\n");
```

```
    fprintf(stderr, "-num_min: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-num_max: numero minimo de elementos de la tabla\n");
    fprintf(stderr, "-incr: incremento\n");
    fprintf(stderr, "-numP: Introduce el numero de permutaciones a promediar\n");
    fprintf(stderr, "-fichSalida: Nombre del fichero de salida\n");
    fprintf(stderr, "-pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre las tres\n");
    exit(-1);
  }

  printf("Practica numero 1, apartado 5\n");
  printf("Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila\n");
  printf("Grupo: 1201\n");

  /* comprueba la linea de comandos */
  for(i = 1; i < argc ; i++) {
    if (strcmp(argv[i], "-num_min") == 0) {
      num_min = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-num_max") == 0) {
      num_max = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-incr") == 0) {
      incr = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-numP") == 0) {
      n_perms = atoi(argv[++i]);
    } else if (strcmp(argv[i], "-fichSalida") == 0) {
      strcpy(nombre, argv[++i]);
    } else if (strcmp(argv[i], "-pivote") == 0) {
      algoritmo=atoi(argv[++i]);
    } else {
      fprintf(stderr, "Parametro %s es incorrecto\n", argv[i]);
    }
  }

  /* calculamos los tiempos */
  if(algoritmo==0){
    ret = genera_tiempos_ordenacion(InsertSort, nombre,num_min, num_max,incr, n_perms, algoritmo);
  }
  else if(algoritmo==1){
    ret = genera_tiempos_ordenacion(MergeSort, nombre ,num_min , num_max , incr, n_perms ,algoritmo);
  }
  else if(algoritmo==2){
```

```
 ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min , num_max , incr , n_perms, algoritmo);
 }
 else if(algoritmo==3){
   ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min, num_max, incr, n_perms, algoritmo);
 }
 else if(algoritmo==4){
  ret = genera_tiempos_ordenacion(QuickSort, nombre , num_min, num_max, incr, n_perms, algoritmo);
 }
else{
 printf("Número del algoritmo incorrecto");
 exit(-1);
}
 if (ret == ERR) { /* ERR_TIME debera ser un numero negativo */
   printf("Error en la funcion Time_Ordena\n");
   exit(-1);
 }
 printf("Salida correcta \n");
 return 0;
}
```

Apartado5:

```
/**************************************************/
/* Funcion: medio_avg                  Fecha:08/11/2016          */
/* Autores:Lucia  Colmenarejo  Pérez  y                          */
/* Litzy Tatiana Rocha Avila                                     */
/* Función que devuelve como pivote el elemento                  */
/* de la posición media de la tabla                              */
/* Entrada:                                                      */
/* int* tabla:Tabla desordenada a ordenar                        */
/* int ip:primer elemento de la tabla                            */
/* int iu:ultimo elemento de la tabla                            */
/* int* pos: posición del pivote                                 */
/* Salida:                                                       */
/* Contador:Devuelve el número de comparación de                 */
/* claves realizadas durante la ordenación                       */
/* -1: En caso de error                                          */
/**************************************************/
int medio_avg(int *tabla, int ip, int iu, int *pos){
   if(!tabla || ip<0 || iu<0 || !pos) return ERR;
```

```
    *pos =(ip+iu)/2;
    return 0;
}
/**************************************************/
/* Funcion: medio_stat          Fecha:08/11/2016  */
/* Autores:Lucia  Colmenarejo  Pérez  y           */
/* Litzy Tatiana Rocha Avila                      */
/* Función que compara los valores de las         */
/* posiciones ip, iu e (ip+ iu)/2 y devuelva      */
/* la posición que contenga el valor intermedio   */
/* entre las tres.                                */
/* Entrada:                                       */
/* int* tabla:Tabla desordenada a ordenar         */
/* int ip:primer elemento de la tabla             */
/* int iu:ultimo elemento de la tabla             */
/* int* pos: posición del pivote                  */
/* Salida:                                        */
/* Contador:Devuelve el número de comparación de  */
/* claves realizadas durante la ordenación        */
/* -1: En caso de error                           */
/**************************************************/
int medio_stat(int *tabla, int ip, int iu, int *pos){
    int v_ip,v_m,v_iu, m, contador=0;
    if(!tabla || ip<0 || iu<0 || !pos) return ERR;
     m=(ip+iu)/2;
    v_ip=tabla[ip];
    v_m=tabla[m];
    v_iu=tabla[iu];
    if(v_ip<=v_iu){
        contador++;
        if(v_ip<=v_m){
            contador++;
            if(v_m<=v_iu){
                contador++;
                *pos=m;
            } else{
                contador++;
                *pos=iu;
            }
        }
```

```
        else {
            contador++;
            *pos=ip;
        }
    } else{
        if(v_iu<=v_m){
            contador++;
            if(v_ip<=v_m){
                contador++;
                *pos=ip;
            }else{
                contador++;
                *pos=m;
            }
        }else{
            contador++;
            *pos=iu;
        }
    }
    return contador;
}
```

4.Resultados:

Resultados del apartado 1:

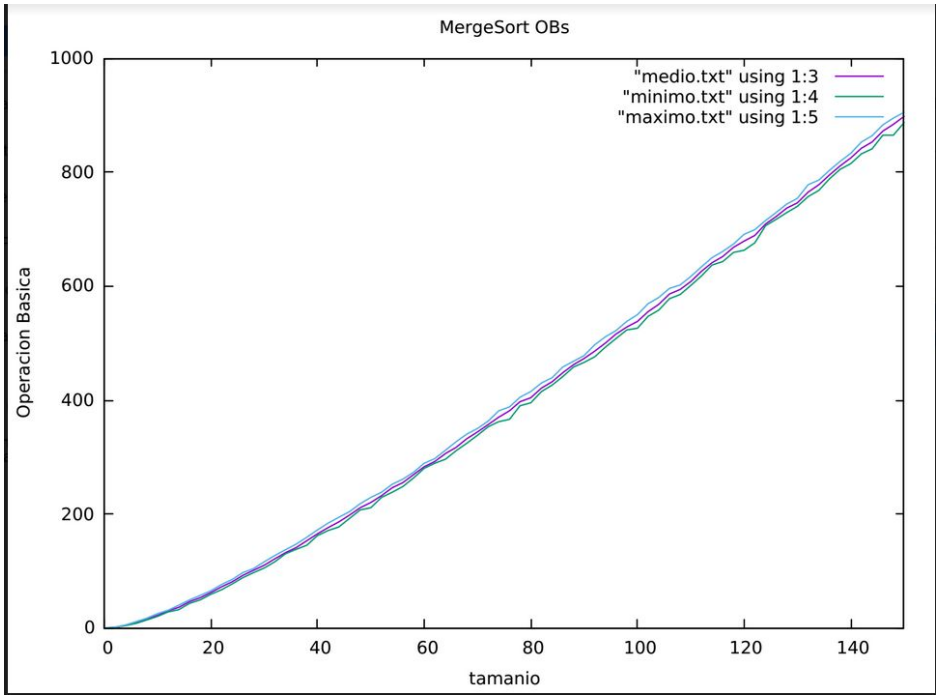
```
ibbliu:~/workspace $ ./ejercicio4
Error en los parametros de entrada:

./ejercicio4 -tamanio <int> -pivote <int>
Donde:
 -tamanio : numero elementos permutacion.
 -pivote : -pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre las tres
ibbliu:~/workspace $ ./ejercicio4 -tamanio 10 -pivote 1
Practica numero 1, apartado 4
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
1       2       3       4       5       6       7       8       9       10
```

Resultados del apartado 2:

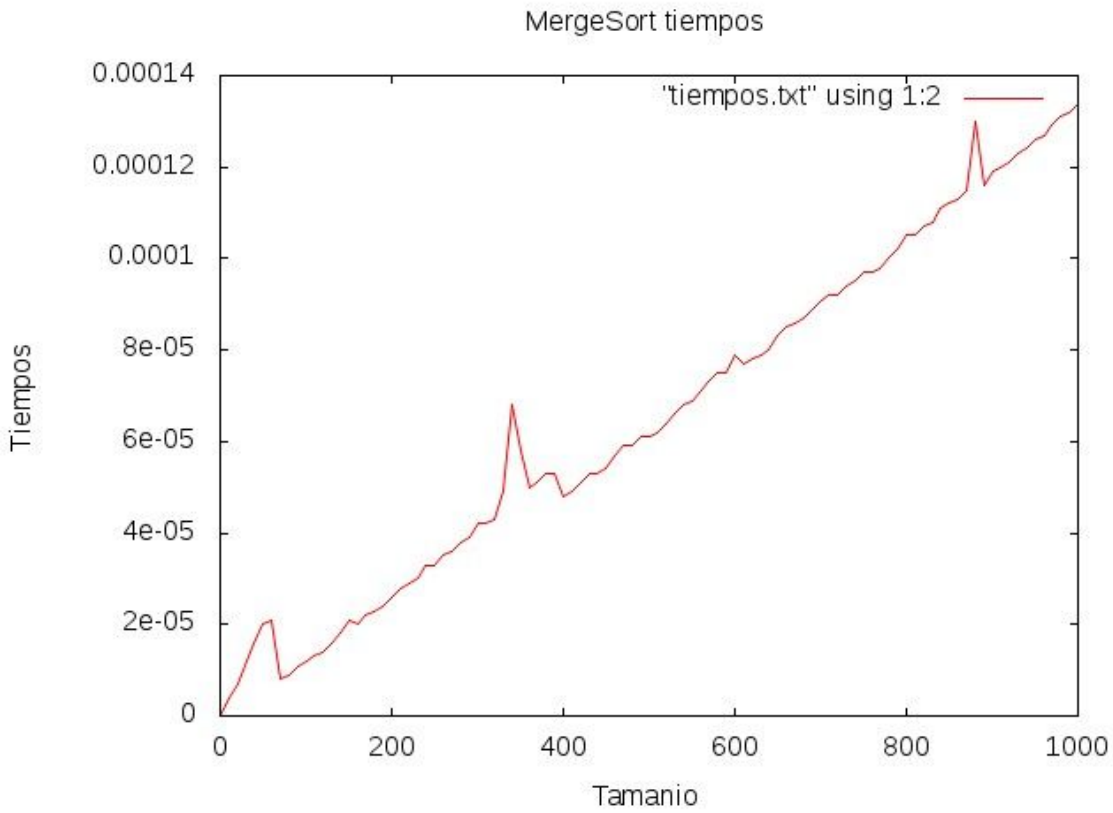
```
ibbliu:~/workspace $ ./ejercicio5
Error en los parametros de entrada:

./ejercicio5 -num_min <int> -num_max <int> -incr <int>
                -numP <int> -fichSalida <string>
Donde:
-num_min: numero minimo de elementos de la tabla
-num_max: numero minimo de elementos de la tabla
-incr: incremento
-numP: Introduce el numero de permutaciones a promediar
-fichSalida: Nombre del fichero de salida
-pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre las tres
ibbliu:~/workspace $ ./ejercicio5  -num_min 0 -num_max 30 -incr 2 -numP 10 -fichSalida "apartado2.txt" -pivote 1
Practica numero 1, apartado 5
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
Salida correcta
```

```
0  0.000000  0.000000  0  0
2  0.000000  1.000000  1  1
4  0.000000  4.000000  4  5
6  0.000000  9.000000  8  11
8  0.000001  15.000000  14  17
10  0.000001  23.000000  22  25
12  0.000001  30.000000  28  32
14  0.000002  37.000000  36  40
16  0.000001  45.000000  43  48
18  0.000002  55.000000  53  57
20  0.000002  62.000000  60  67
22  0.000002  72.000000  70  74
24  0.000002  82.000000  79  85
26  0.000002  92.000000  86  97
28  0.000003  103.000000  100  106
30  0.000003  112.000000  107  117
```

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



Gráfica con el tiempo medio de reloj para MergeSort



Como podemos observar hay algunos picos, suponemos que estos son debido a que hay algunas permutaciones para la cuales el algoritmo no es del todo óptimo y tarda más en ejecutarlas. Para esta gráfica utilizamos una salida distinta ya que utilizar las permutaciones de 10 en 10 y solo hasta 100 nos impide apreciar correctamente los tiempos.

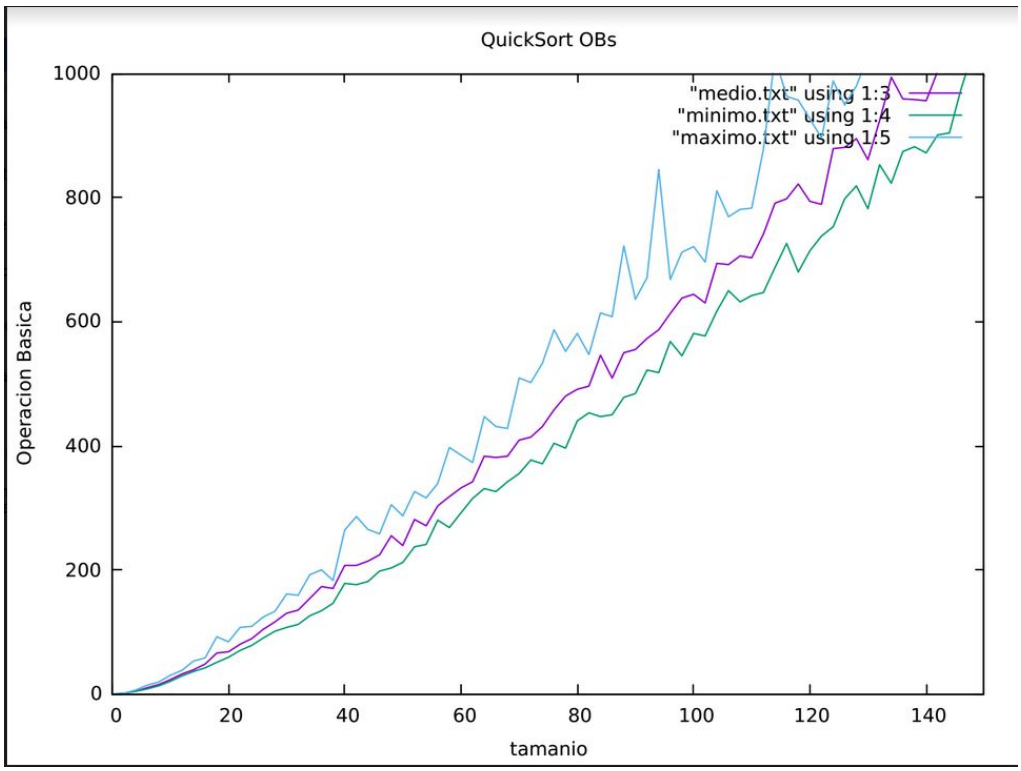
Resultados del apartado 3:

```
ibbliu:~/workspace $ ./ejercicio4
Error en los parametros de entrada:

./ejercicio4 -tamanio <int> -pivote <int>
Donde:
 -tamanio : numero elementos permutacion.
 -pivote : -pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre las tres
ibbliu:~/workspace $ ./ejercicio4  -tamanio 10 -pivote 2
Practica numero 1, apartado 4
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
1       2       3       4       5       6       7       8       9       10
```

Resultados del apartado 4:

```
0  0.000000  0.000000  0  0
2  0.000000  1.000000  1  1
4  0.000000  4.000000  4  6
6  0.000000  9.000000  8  11
8  0.000000  17.000000  15  21
10  0.000001  23.000000  19  28
12  0.000001  31.000000  26  36
14  0.000001  38.000000  33  46
16  0.000001  48.000000  43  56
18  0.000001  56.000000  51  64
20  0.000001  70.000000  65  82
22  0.000001  78.000000  68  94
24  0.000002  87.000000  74  114
26  0.000002  99.000000  88  125
28  0.000002  117.000000  101  148
30  0.000002  124.000000  108  156
```

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

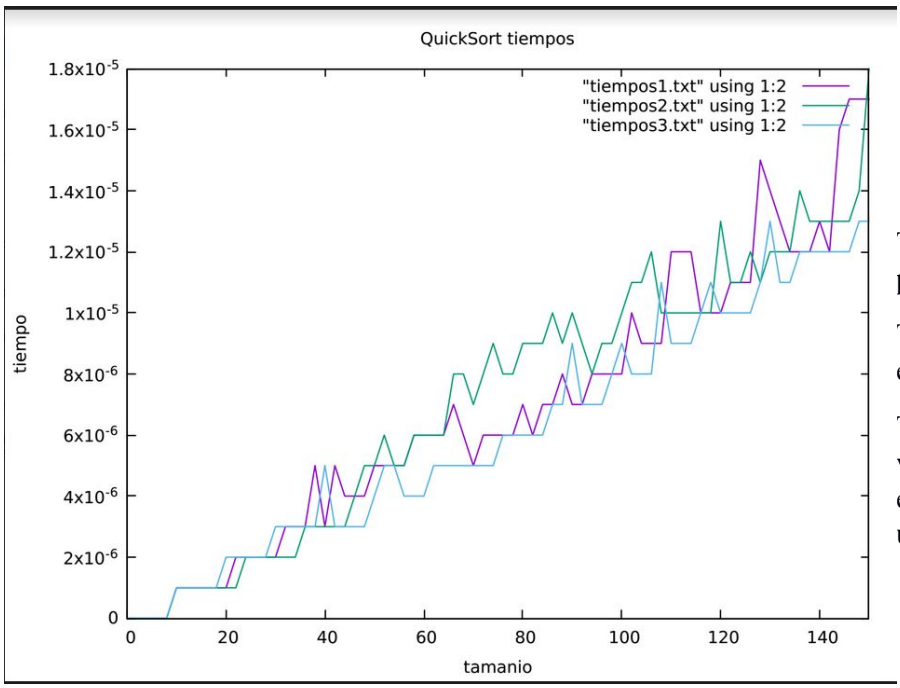


Resultados del apartado 5:

```
ibbliu:~/workspace $ ./ejercicio5
Error en los parametros de entrada:

./ejercicio5 -num_min <int> -num_max <int> -incr <int>
                -numP <int> -fichSalida <string>
Donde:
-num_min: numero minimo de elementos de la tabla
-num_max: numero minimo de elementos de la tabla
-incr: incremento
-numP: Introduce el numero de permutaciones a promediar
-fichSalida: Nombre del fichero de salida
-pivote:0 IS, 1 MS, 2 QS pivote primer elemento, 3 QS pivote elemento medio, 4 QS pivote valor intermedio entre las tres
ibbliu:~/workspace $ ./ejercicio5 -num_min 0 -num_max 30 -incr 2 -numP 10 -fichSalida "apartado5.txt" -pivote 3
Practica numero 1, apartado 5
Realizada por: Lucia Colmenarejo Pérez y Litzy Tatiana Rocha Avila
Grupo: 1201
Salida correcta
```

```
0  0.000000  0.000000  0  0
2  0.000000  1.000000  1  1
4  0.000000  5.000000  4  6
6  0.000000  10.000000  8  14
8  0.000000  15.000000  15  18
10  0.000001  24.000000  21  29
12  0.000001  36.000000  27  49
14  0.000001  42.000000  34  59
16  0.000001  49.000000  42  63
18  0.000001  56.000000  53  63
20  0.000001  72.000000  60  101
22  0.000001  82.000000  73  98
24  0.000002  94.000000  78  117
26  0.000002  105.000000  89  137
28  0.000002  112.000000  96  137
30  0.000002  131.000000  115  143
```

Gráfica con el tiempo medio de reloj para QuickSort de cada uno de los pivotes posibles.



Tiempos1: pivote primer elemento

Tiempos2: pivote elemento medio

Tiempos3: pivote valor intermedio entre el primero, el último y el medio.

5.Cuestiones:

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.

Las trazas de las gráficas son picudas debido a cuánto esté ordenada la permutación y esto es aleatorio, de ahí que los picos salgan de forma aleatoria. La forma de reducir estos picos es aumentar el parámetro -numP. Al generar una mayor cantidad de resultados sobre una tabla de mismo tamaño utilizando el mismo algoritmo y coger su media obtendremos el valor más próximo al teórico y así no habría picos en la gráfica.

$A_{MS}(N) = O(N \lg(N))$
$A_{QS}(N) = 2N \log(N) + O(N)$

2. Razonad el resultado obtenido al comparar las versiones de quicksort con los diferentes pivotes tanto si se obtienen diferencias apreciables como si no.

No se aprecian grandes diferencias entre los distintos tipos de pivotes a utilizar, pero podemos distinguir como el caso en el que cogemos el valor intermedio entre le primero, el último y el medio es el caso que menos tiempo tarda por muy poco, caso en el que utilizamos la función *medio_stat.*

3. ¿Cuáles son los casos mejor y peor para cada uno de los algoritmos? ¿Qué habría que modificar en la práctica para calcular estrictamente cada uno de los casos (también el caso medio)?

El caso peor del algoritmo Mergesort esWms(N)= Nlog(N)+O(N) y el caso mejor es Bms(N)=½(Nlog(N)) . Por lo que el caso medio es Ams(N) = θ(Nlog(N)).

El caso peor del algoritmo Quicksort es Wqs(N)= $N^2/2 - N/2$ y el caso mejor es Bqs(N) = O(Nlog(N)). Por lo que el caso medio es Aqs(N) = 2Nlog(N) + O(N).

Una de las primeras ideas sería introducir en -numP un número muy grande para poder obtener todas las posibles permutaciones y por lo tanto obtener la mejor y la peor entrada para cada algoritmo, sin embargo esto no nos garantiza que obtengamos el resultado deseado. Se puede aproximar mucho pero en más de una ocasión no nos daría el resultado correcto. Por lo tanto optamos por la siguiente solución:
para conseguir el mejor tendríamos que buscar la permutación más óptima para la entrada de cada algoritmo. De forma similar haríamos para el peor caso. En cambio, para el caso medio habría que generar todas las posibles permutaciones de cada tamaño y coger la media aritmética.

4. ¿Cuál de los dos algoritmos estudiados es más eficiente empíricamente? Compara este resultado con la predicción teórica. ¿Cuál(es) de los algoritmos es/son más eficientes desde el punto de vista de la gestión de memoria? Razona este resultado.

Al observar las gráficas podemos observar que el algoritmo de ordenación de MergeSort efectúa un número menor de operaciones básicas con respecto a QuickSort . Sin embargo, en cuanto al tiempo de ejecución QuickSort tarda menos que MergeSort. Podemos ver que coinciden con los resultados de la predicción teórica expuestos en la pregunta anterior, ya que el caso peor del MergeSort es de un orden menor que el del QuickSort, por lo que tiene sentido que el MergeSort haga menos operaciones básicas que el QuickSort (pese a que coincidan en el número de OBs del caso medio).

Desde el punto de vista de la gestión de memoria es mejor el algoritmo de QuickSort que el de MergeSort. Esto se debe a que en QuickSort no reserva ni libera memoria dinámica, mientras que MergeSort sí. Por ello MergeSort es menos eficiente en cuanto a gestión de memoria se refiere, porque hay que tener en cuenta que utiliza memoria dinámica y esto tiene un alto coste.

Podemos llegar a la conclusión de que QuickSort es el algoritmo más óptimo estudiado ya que haciendo balance entre operaciones básicas tiempo de ejecución y gestión de memoria es más eficiente.

5.Conclusiones finales:

Esta práctica nos ha ayudado a reforzar nuestros conocimientos en cuanto a algoritmos de ordenación se refiere. En concreto hemos estudiado la implementación y análisis del algoritmo MergeSort y QuickSort.

Algo a destacar sería el uso de punteros ya que todos nuestros contadores son punteros por lo que le hemos dado bastante importancia al tema de punteros.

Por otra parte también hemos reforzado la parte de recursión ya que hemos realizado nuestras propias funciones recursivas debido a que nos parecía más claro el implementarlas así.