Análisis de Algoritmos 2020/2021

Práctica 2

Eduardo Terrés Caballero, Víctor Pérez Cano.

Grupo 6.

Código	Gráficas	Memoria	Total

1. Introducción.

La dinámica principal de la práctica consiste en implementar y analizar dos algoritmos de ordenación, que corresponden al tipo "divide y vencerás", así como una modificación del último. En ambos casos, el método pedido será desarrollado en C y serán de carácter recursivo, esto es, que los propios algoritmos llevarán a cabo llamadas a sí mismos hasta obtener el resultado esperado. En este caso, en el control de errores tendrán una presencia significativa las comprobaciones de casos que es imposible que sucedan, pero se sigue tratando de condiciones necesarios para el correcto funcionamiento del algoritmo. Es así que hemos optado por emplear la función "assert(*condicion*)", de forma que sus apariciones puedan ser ignoradas si así lo deseáramos. Para facilitar la introducción del código en una misma página, hemos creado un Anexo al final del documento que incluye la cabecera de cada función.

2. Objetivos

A continuación, se explicará de forma resumida la finalidad que existe detrás de las implementaciones realizadas. Como es evidente, la información será complementada con el resto de puntos de la memoria. Así pues, el propósito de esta sección es reflejar el problema tal y como se nos planteó.

2.1 Apartado 1

En el primer apartado, se pide implementar el método de ordenación *MergeSort*. La función principal y que será visible para el usuario es la siguiente:

```
int mergesort(int* tabla, int ip, int iu);
```

Los parámetros de entrada son la tabla que contiene los elementos a ordenar - *int* tabla* -, el índice a partir del cual se va a llevar a cabo el proceso – *int ip* – y el índice superior hasta el cual se ordena – *int iu* -. El retorno de la función es un número entero, que almacena las veces que se ha ejecutado la operación básica; ERR[1] en caso de error.

2.2 Apartado 2

Una vez llevada a cabo la implementación del apartado anterior, la obtención y el análisis de resultados resumen la finalidad de este. De esta forma, se engloban todas las pruebas realizadas con programas de comprobación de la gestión de memoria, así como que los resultados obtenidos son coherentes desde un punto de vista conceptual (costes).

2.3 Apartado 3

De forma análoga a la implementación anterior, se pide desarrollar el código en C del método de ordenación QuickSort.

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

1. ERR es una constante definida en los ficheros .h; tiene valor (-1).

Los parámetros de entrada son los mismos que en *MergeSort*, esto es: la tabla a ordenar - *int* tabla* -, y los índices superior – *int iu* – e ínfimo – *int ip* –. El retorno de la función es el número de OBs llevadas a cabo; ERR en caso de error.

2.4 Apartado 4

De nuevo, en la sección posterior a la implementación de la rutina, la toma de resultados y su posterior análisis son el motivo del apartado. Incluye todos los *test* realizados con programas de análisis para la gestión de memoria, así como que la información en sí misma es correcta.

2.5 Apartado 5

En el último apartado y una vez implementado el método *QuickSort*, se plantea uno de los problemas que existen a la hora de trabajar con métodos recursivos: la sobrecarga. Así pues, se pide crear la función *QuickSort_src*, que consiste en una modificación de la mencionada previamente, eliminando la recursión de cola. Su prototipo es el siguiente:

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

Los parámetros de entrada y salida son análogos a los de la función con recursión de cola.

3. Herramientas y metodología

La práctica ha sido realizada en un entorno de ejecución de *Linux*, utilizando *gcc* como compilador del lenguaje *C*, *valgrind* como revisor de la gestión de la memoria, y *Visual Studio Code* y *Netbeans* como IDEs. El control de versiones se ha realizado mediante *git* y a través de un repositorio privado en *Github*. Las gráficas se han realizado mediante *scripts* del programa *Gnuplot*.

3.1 Apartado 1

Una vez conocido el funcionamiento del algoritmo, y haciendo uso de un pseudocódigo, hemos implementado la rutina en C. Para la implementación de *MergeSort*, se pide y es necesario desarrollar una función auxiliar, de carácter privado, donde se desarrolla la lógica del método, y se ejecuta la operación básica, que hemos decidido sea la c.d.c siguiente:

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

El prototipo de esta función auxiliar es:

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

La función recibe cuatro parámetros de entrada: la tabla que contiene los elementos con los que se trabajará – *int* tabla* -, los índices entre los que quedan comprendidos los elementos de la tabla empleados – *int ip, int iu* -, y la posición del índice que marca la mitad de la tabla – *int imedio* -. La funcionalidad del parámetro *imedio* es la de poder trabajar con dos tablas, de manera conceptual, separadas por imedio, aunque en la práctica se trate de un solo *array* de eneteros. Es necesario el uso de memoria dinámica para almacenar los elementos a medida que estos se comparan.

Cabe señalar que, se ha añadido control de errores en la función, siguiendo la metodología, previamente explicada, del uso de *asserts*. No sería necesario efectuar estas comprobaciones, ya que, si la ejecución del programa ha avanzado hasta ese punto, las condiciones ya se han complido, pero la posibilidad de eliminarlas permite añadir una capa más de garantía de ejecución correcta, que puede ser eliminada.

Finalmente, la función principal tiene como único trabajo realizar llamadas recursivas a la primera y segunda mitad de la tabla inicial, así como a *merge*, para finalmente devolver el resultado.

3.2 Apartado 2

Para las comprobaciones, hemos hecho uso del *Ejercicio 5* de la Práctica 1, creando una nueva versión que utiliza este algoritmo y añadiendo una entrada en el *Makefile* que permite ejecutar este *test*. Así pues, hemos obtenido la información relativa al número mínimo, medio y máximo de OBs, así como el tiempo medio de ejecución de *MergeSort*, todos ellos para unos determinados valores, y hemos salvado estos datos en el fichero *mergesort.dat* para su posterior representación gráfica. Como forma de comprobar el correcto manejo de la memoria, y teniendo en cuenta que el método emplea memoria auxiliar, hemos hecho uso de *valgrind*.

3.3 Apartado 3

Conociendo el funcionamiento de *QuickSort*, o a través de un pseudocódigo, hemos implementado la rutina, haciendo uso de dos rutinas auxiliares y privadas. En primer lugar,

```
int medio(int *tabla, int ip, int iu,int *pos)
```

Asigna la posición del pivote, que puede ser cualquier posición de la tabla – aunque *medio* hace uso de primer elemento de la tabla en específico - con el que trabajará la siguiente función:

```
int partir(int* tabla, int ip, int iu,int *pos)
```

La finalidad de la función *partir* es llevar a cabo la lógica del algoritmo de ordenación *QuickSort*. De esta forma, se encarga de llamar a *medio*, y su funcionalidad se basa en, a partir de una tabla como parámetro de entrada y de los índices superior e ínfimo, colocar el elemento que ocupa la posición de pivote en el lugar que ocuparía si la tabla estuviese ordenada. Asimismo, cumple la condición siguiente: la subtabla izquierda al pivote está formada por elementos de menor valor que el pivote; y la derecha por elementos de mayor valor. Además, contiene la operación básica, que consiste en la comparación de clave siguiente:

```
tabla[i] < p_aux
```

Finalmente, el prototipo de la función principal, y a la que emitirá llamadas el usuario que la emplee es:

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

Esta función tiene como finalidad gestionar las llamadas recursivas a sí misma, las llamadas a partir y sumar las operaciones básicas que se realizan.

Los parámetros de entrada de las dos funciones auxiliares son: la tabla que contiene los elementos con los que se trabajará – *int* tabla* -, los índices entre los que quedan comprendidos los elementos de la tabla empleados – *int ip, int iu* – y un puntero a un entero, donde se almacenará el índice del pivote – *int *pos* -. La función principal recibe los primeros tres parámetros mencionados.

3.4 Apartado 4

De forma análoga al apartado 2, hemos modificado el *Ejercicio 5* de la práctica anterior para que haga uso de *QuickSort*, y hemos añadido su correspondiente entrada en el *Makefile*. Además de reafirmar el correcto funcionamiento del algoritmo y su uso óptimo de la memoria, nos proporciona los datos necesarios para poder comparar el rendimiento empírico de este algoritmo con otros como *MergeSort*. Concretamente se obtiene el número de OBs medias, y de los casos peor y mejor, así como el tiempo medio de ejecución. Dichos datos han sido escritos en el fichero *quicksort.dat* lo que nos permitirá llevar a cabo su posterior representación gráfica.

3.5 Apartado 5

Como se pide implementar una función que es, en esencia, una modificación de la anterior, las funciones auxiliares se mantienen. El prototipo de esta nueva función principal es el siguiente:

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

Donde los parámetros de entrada y salida coinciden con los de la función del apartado 3. El problema planteado para eliminar la recursión de cola ha sido emplear un bucle – de tipo *while* es este caso -, de tal forma que la subtabla derecha de ordena consecutivamente, y la subtabla izquierda se ordena mediante llamadas recursivas.

Con respecto a las pruebas efectuadas, hemos conseguido probar el buen funcionamiento de la función modificando el *Ejercicio 4* y *Ejercicio 5* de la Práctica 3.

4. Código fuente

A continuación, se muestra el código fuente de las rutinas implementadas en la práctica:

4.1 Apartado 1 (*Mergesort)*

```c
/*************************************************/
/* Funcion: Merge         Fecha: 23/10/2020      */
/* Autores: Eduardo Terrés y Víctor Pérez        */
/*                                               */
/* Funcion que combina dos tablas ordenadas en   */
/* otra tabla ordenada también. Se trata de una  */
/* funcion privada, auxiliar de MergeSort        */
/*                                               */
/* Entrada:                                      */
/* int* tabla: Array en el que se hace la        */
/* combinación                                   */
/* int ip: indice inferior de la primera tabla   */
/* int iu: indice superior de la segunda tabla   */
/* int imedio: indice superior de la primera tabla */
/* e inferior de la segunda                      */
/* Salida:                                       */
/* Numero de veces que se ejecuta la comparacion */
/* de indices                                    */
/* ERR en caso de error                          */
/*************************************************/
int merge(int *tabla, int ip, int iu, int imedio)
{
    int tam, k, i, j, cont;
    int *tabla_aux = NULL;

    /* Control de Errores */
    assert(ip <= iu);
    assert(ip >= 0);
    assert(tabla != NULL);

    /* Inicializacion de parametros */
    tam = iu - ip + 1;
    i = ip;
    j = imedio + 1;
    k = 0;

    tabla_aux = (int *)malloc(tam * sizeof(tabla_aux[0]));
    if (tabla_aux == NULL) return ERR;

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

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

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

    /* Copia tabla_aux en tabla en el intervalo [ip,iu] */
    for (i = 0; i < tam; i++) tabla[ip + i] = tabla_aux[i];

    free(tabla_aux);

    return cont;
}
```

```c
/*************************************************/
/* Funcion: MergeSort     Fecha: 23/10/2020      */
/* Autores: Eduardo Terrés y Víctor Pérez        */
/*                                               */
/* Funcion que ordena los elementos de menor a   */
/* mayor de una tabla dada siguiendo la idea de  */
/* divide y vencerás.                            */
/*                                               */
/* Entrada:                                      */
/* int* tabla: Array de numeros a ordenar        */
/* int ip: indice inferior a partir del cual se  */
/* quiere ordenar                                */
/* int iu: indice superior de la tabla hasta el  */
/* que se quiere ordenar                         */
/* Salida:                                       */
/* int cont: numero de veces que se ejecuta la   */
/* operacion básica.                             */
/* ERR en caso de error                          */
/*************************************************/
int mergesort(int *tabla, int ip, int iu)
{
    int imedio, cont1, cont2, cont3;

    /* Control de errores */
    if (ip > iu)
        return ERR;
    if (ip < 0)
        return ERR;
    if (tabla == NULL)
        return ERR;

    /* Caso base de recursion */
    if (iu == ip)
        return 1;

    /* Division de las tablas */
    imedio = (int)(iu + ip) / 2;

    cont1 = mergesort(tabla, ip, imedio);
    if (cont1 == ERR)
        return ERR;

    cont2 = mergesort(tabla, imedio + 1, iu);
    if (cont2 == ERR)
        return ERR;

    cont3 = merge(tabla, ip, iu, imedio);
    if (cont3 == ERR)
        return ERR;

    return cont1 + cont2 + cont3;
}
```

4.3 Apartado 3 (*medio, partir, y QuickSort*)

```c
/**************************************************/
/* Funcion: Medio, auxiliar de QuickSort          */
/* Fecha: 30/10/2020                              */
/* Autores: Eduardo Terrés y Víctor Pérez         */
/*                                                */
/* Función que asigna el valor de ip al puntero pos*/
/*                                                */
/* Entrada:                                       */
/* int* tabla: Array de numeros a ordenar         */
/* int ip: indice inferior a partir del cual se   */
/* quiere ordenar                                 */
/* int iu: indice superior de la tabla hasta el   */
/* que se quiere ordenar                          */
/* int pos: indice superior del pivote            */
/* Salida:                                        */
/* int *pos: indice del pivote                    */
/* ERR en caso de error                           */
/**************************************************/
int medio(int *tabla, int ip, int iu, int *pos){
    /* Control de Errores */
    assert(ip>=0);
    assert(ip<=iu);
    assert(tabla != NULL);


    if(pos == NULL) return ERR;

    *pos = ip;
    return 0;
}
```

```c
/**************************************************/
/* Funcion: Partir, auxiliar de QuickSort         */
/* Fecha: 30/10/2020                              */
/* Autores: Eduardo Terrés y Víctor Pérez         */
/*                                                */
/* Funcion desarrolla la logica del algoritmo     */
/* Quicksort                                      */
/*                                                */
/* Entrada:                                       */
/* int* tabla: Array de numeros a ordenar         */
/* int ip: indice inferior a partir del cual se   */
/* quiere ordenar                                 */
/* int iu: indice superior de la tabla hasta el   */
/* que se quiere ordenar                          */
/* int pos: indice superior del pivote            */
/* Salida:                                        */
/* int *pos: indice del pivote                    */
/* ERR en caso de error                           */
/**************************************************/
int partir(int* tabla, int ip, int iu, int *pos){
    int t_aux, p_aux,i;
    int cont = 0;

    /* Control de Errores */
    assert(ip>=0);
    assert(ip<=iu);
    if(pos == NULL || tabla == NULL)return ERR;

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

    /* Almacena el valor del pivote */
    p_aux = tabla[*pos];

    *pos = ip;

    /* swap T[pos] <-> T[ip]*/
    t_aux = tabla[*pos];
    tabla[*pos] = tabla[ip];
    tabla[ip] = t_aux;

    for(i = ip + 1; i <= iu; i++){
        if(tabla[i] < p_aux){
            (*pos) ++;
            /* swap T[pos] <-> T[i] */
            t_aux = tabla[*pos];
            tabla[*pos] = tabla[i];
            tabla[i] = t_aux;
        }
        cont ++;
    }

    /* swap T[pos] <-> T[ip] */
    t_aux = tabla[*pos];
    tabla[*pos] = tabla[ip];
    tabla[ip] = t_aux;

    return cont;
}
```

(Quicksort)

```c
/**************************************************/
/* Funcion: QuickSort Fecha: 30/10/2020           */
/* Autores: Eduardo Terrés y Víctor Pérez         */
/*                                                */
/* Funcion que ordena los elementos de mayor a    */
/* menor de una tabla dada.                       */
/*                                                */
/* Entrada:                                       */
/* int* tabla: Array de numeros a ordenar         */
/* int ip: indice inferior a partir del cual se   */
/* quiere ordenar                                 */
/* int iu: indice superior de la tabla hasta el   */
/* que se quiere ordenar                          */
/* Salida:                                        */
/* int cont: numero de veces que se ejecuta la    */
/* operacion principal                            */
/* ERR en caso de error                           */
/**************************************************/
int quicksort(int *tabla, int ip, int iu)
{
    int cont = 0, cont1 = 0, cont2 = 0, ret;
    int pos;

    /* Control de Errores */
    if (ip > iu)
        return ERR;
    if (ip < 0)
        return ERR;

    /* Caso base de recursion */
    if (ip == iu)
        return 0;

    if (tabla == NULL)
        return ERR;

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

    /* Subtabla izquierda */
    if (pos - 1 >= ip)
    {
        ret = quicksort(tabla, ip, pos - 1);
        if (ret == ERR)
            return ERR;
        cont1 += ret;
    }

    /* Subtabla derecha */
    if (pos + 1 <= iu)
    {
        ret = quicksort(tabla, pos + 1, iu);
        if (ret == ERR)
            return ERR;
        cont2 += ret;
    }

    return cont + cont1 + cont2;
}
```

4.5 Apartado 5 (*QuickSort sin recursión de cola*)

```c
/**************************************************/
/* Funcion: QuickSort_SRC Fecha: 6/11/2020        */
/* Autores: Eduardo Terrés y Víctor Pérez         */
/*                                                */
/* Funcion que ordena los elementos de mayor a    */
/* menor de una tabla dada. Es analoga a quicksort */
/* pero eliminando la recursion de cola           */
/*                                                */
/* Entrada:                                       */
/* int* tabla: Array de numeros a ordenar         */
/* int ip: indice inferior a partir del cual se   */
/* quiere ordenar                                 */
/* int iu: indice superior de la tabla hasta el   */
/* que se quiere ordenar                          */
/* Salida:                                        */
/* numero de veces que se ejecuta la operacion    */
/* principal                                      */
/* ERR en caso de error                           */
/**************************************************/
int quicksort_src(int *tabla, int ip, int iu)
{
    int cont = 0, cont1 = 0, ret;
    int pos;

    /* Control de Errores */
    if (ip > iu)
        return ERR;
    if (ip < 0)
        return ERR;
    if (tabla == NULL)
        return ERR;

    while (ip <= iu)
    {
        ret = partir(tabla, ip, iu, &pos);
        if (ret == ERR)
            return ERR;
        cont += ret;

        /* subtabla izquierda */
        if (ip <= pos - 1)
        {
            ret = quicksort_src(tabla, ip, pos - 1);
            if (ret == ERR)
                return ERR;
            cont1 += ret;
        }

        ip = pos + 1;
    }

    return cont + cont1;
}
```

5. Resultados, Gráficas

A continuación, se muestran los resultados gráficos en cuanto a número de operaciones básicas y tiempo de ejecución de las tres rutinas implementadas, obtenidos mediante el programa *Gnuplot*. La dinámica seguida para la obtención de los resultados gráficos ha sido crear ficheros con extensión *.gp*, que contenían las órdenes a ejecutar en el *Gnuplot*. Se incluyen en la entrega.

5.1 Apartado 1

El análisis de resultados de la implementación de *MergeSort* es meramente cualitativo, ya que no se puede valorar de forma nominal, sino el correcto funcionamiento de la rutina. Cabe mencionar que la ejecución de *ejercicio4_mergesort.c* es la herramienta que hemos usado para comprobar el correcto funcionamiento de la función. Haciendo uso del comando:

make ejercicio4_mergesort_test

Se obtiene el siguiente resultado:

Resultado ejecución ejercicio4_mergesort.c

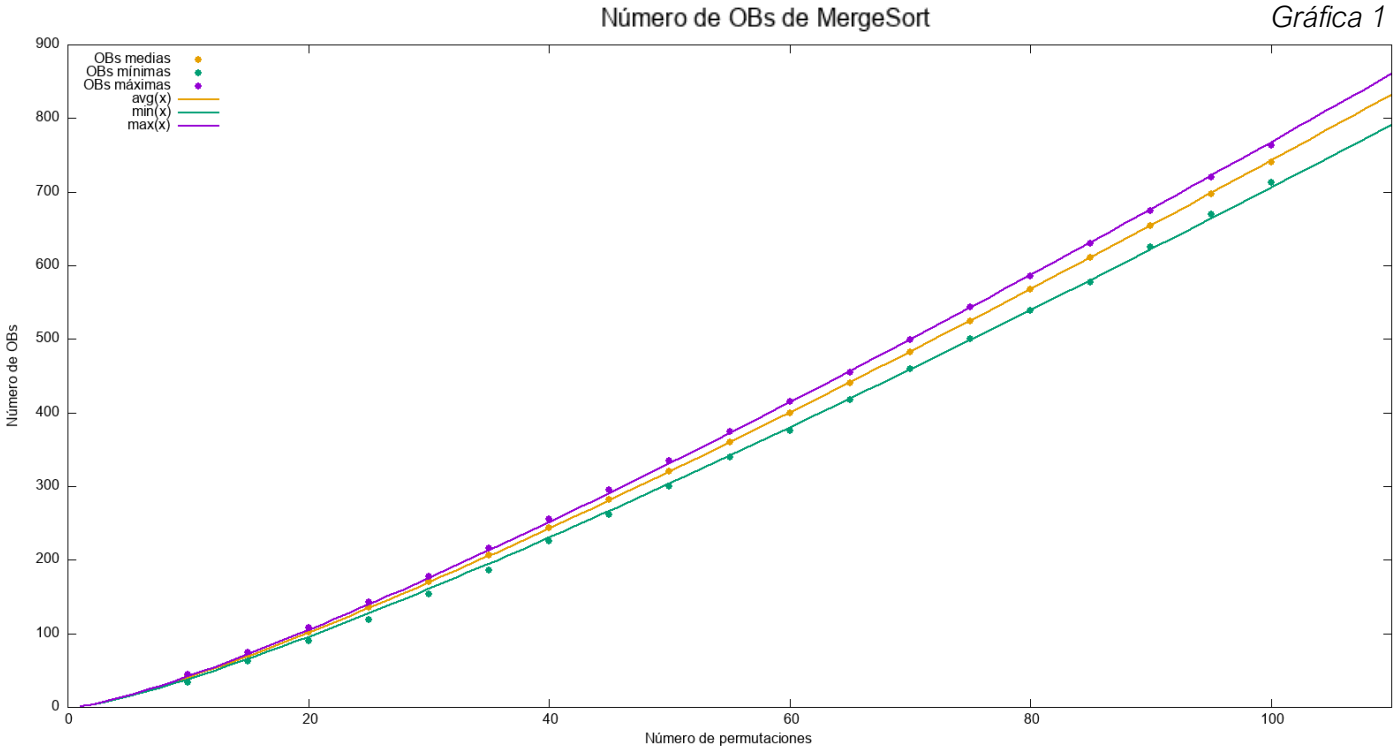
```
#-------------------------------------------------------------#
 Ejecutando ejercicio4_mergesort, haciendo uso de MergeSort
Practica numero 2, MergeSort
Realizada por: Eduardo Terrés y Víctor Pérez
Grupo: 06
 0       1       2       3       4       5       6       7       8       9
10       11      12      13      14      15      16      17      18      19
20       21      22      23      24      25      26      27      28      29
30       31      32      33      34      35      36      37      38      39
40       41      42      43      44      45      46      47      48      49
50       51      52      53      54      55      56      57      58      59
60       61      62      63      64      65      66      67      68      69
70       71      72      73      74      75      76      77      78      79
80       81      82      83      84      85      86      87      88      89
90       91      92      93      94      95      96      97      98      99
```

Se comprueba entonces que la rutina es funcional.

5.2 Apartado 2

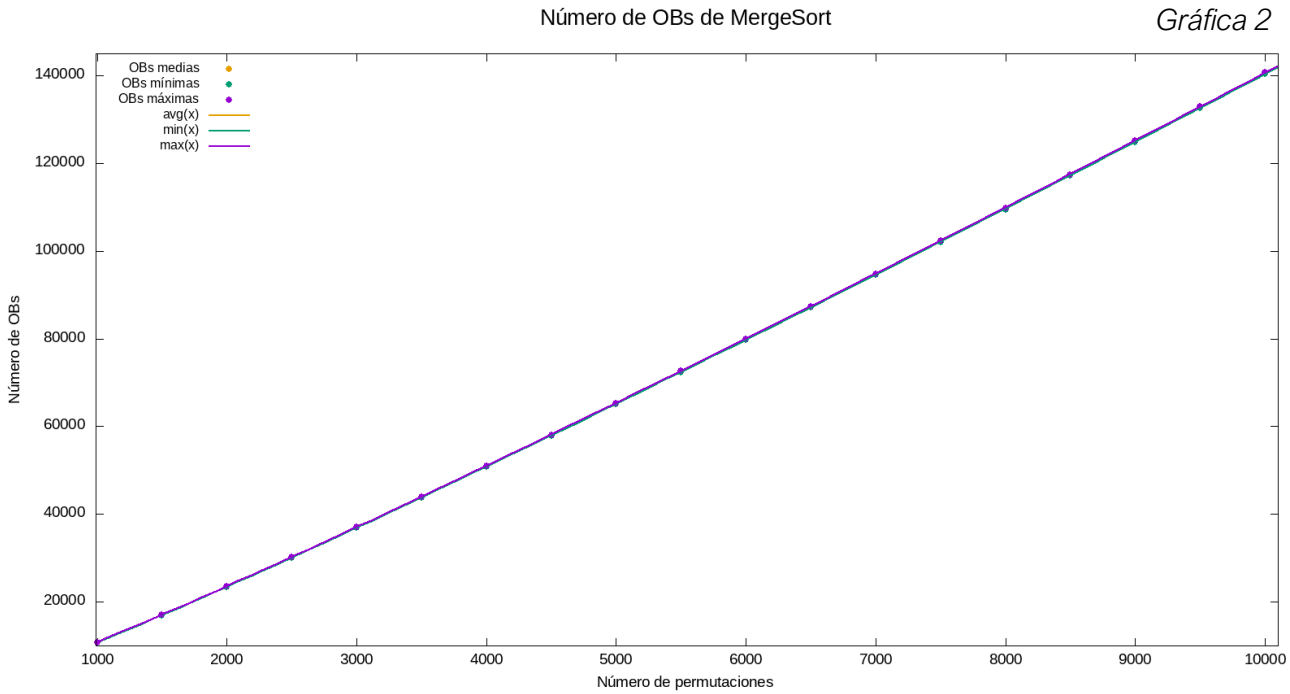
En los resultados gráficos proporcionados por el algoritmo *MergeSort* aparece un fenómeno interesante: para un número bajo de permutaciones (intervalo [5,100], con saltos de 5 en 5 y 100.000 iteraciones por cada tamaño) los costes máximos, mínimos y medios en cuanto a operaciones básicas están ligeramente separados, como se puede observar en la *Gráfica 1*.

Gráfica 1



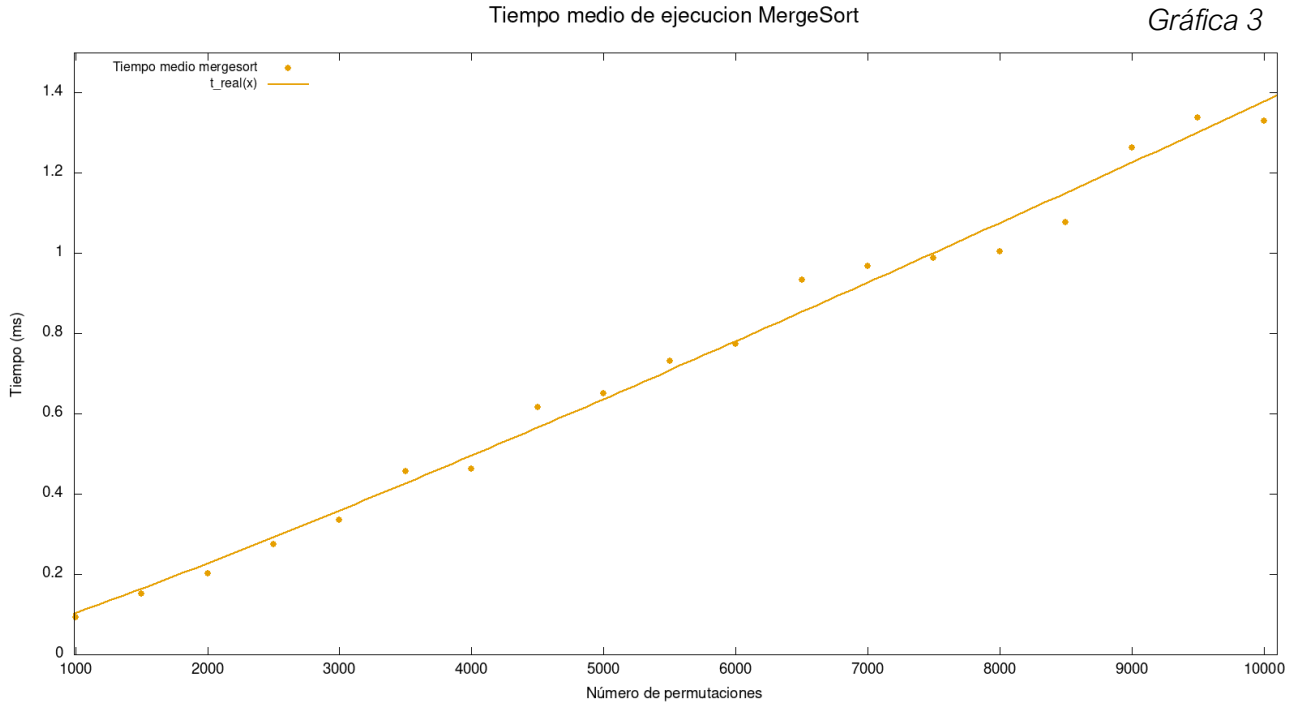
Por otra parte, para un número mucho mayor las tres curvas son prácticamente iguales, como se puede observar en la *Gráfica 2*. Tomando tamaños de permutaciones en el intervalo [1000,10000], con saltos de 500 en 500 y 500 iteraciones por cada tamaño:

Gráfica 2



Entre ambas gráficas, resulta evidente que la segunda representa de forma más fidedigna el comportamiento del algoritmo cuando el tamaño de la permutación se encuentra en valores límite. De esta forma, se confirma que $W_{MS} = O(n*\log(n))$, $B_{WS} = O(n*\log(n))$ y $A_{MS} = \Theta(n*\log(n))$. Entonces, se confirma que el tipo de crecimiento – en este caso del orden de n*log(n) - del número de OBs de los casos mejor, peor y medio del algoritmo de ordenación *MergeSort* coincide. En la práctica, este hecho es respaldado por la superposición de las gráficas en el *Gráfico 2.*

A continuación, se muestra la gráfica con el tiempo medio de ejecución de *MergeSort*, con las mismas condiciones de ejecución que la *Gráfica 2* – tamaños de permutaciones en el intervalo [1000,10000], con saltos de 500 en 500 y 500 iteraciones por tamaño -:



Gráfica 3

De forma análoga a lo previamente mencionado, y teniendo en cuenta la complejidad temporal del caso medio de *MergeSort* ($A_{MS} = \Theta(n*\log(n))$), se puede comprobar que los crecimientos teórico y experimental se corresponden.

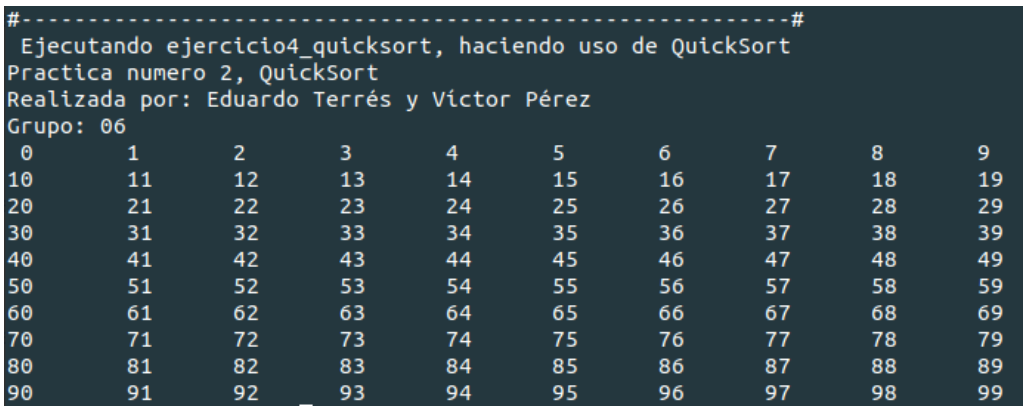
5.3 Apartado 3

De nuevo, el análisis de resultados del apartado 3 - *QuickSort* - es meramente cualitativo, ya que no se puede valorar de forma nominal la implementación realizada, sino el funcionamiento de la rutina. Al ejecutar el archivo *ejercicio4_quicksort.c* se comprueba el correcto funcionamiento de la función programada. Haciendo uso del comando:

make ejercicio4_quicksort_test

Se obtiene el siguiente resultado:

Resultado ejecución ejercicio4_quicksort.c

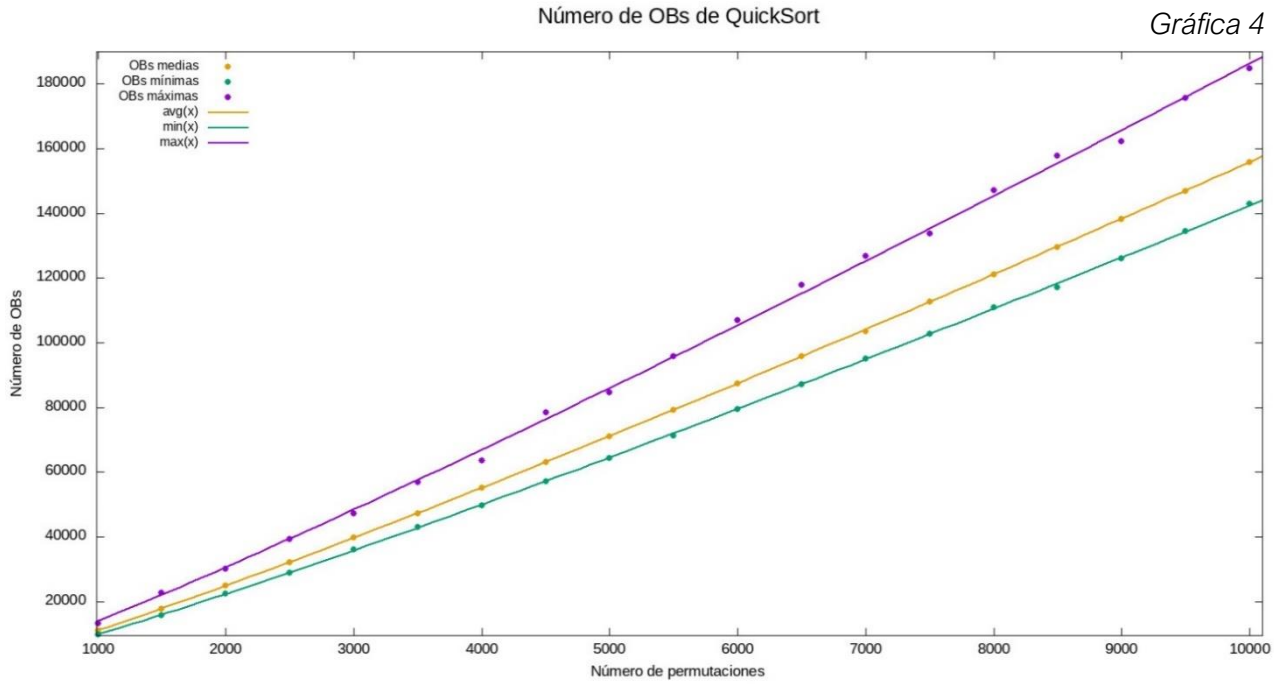


Consecuentemente, se comprueba el correcto funcionamiento de la rutina.

5.4 Apartado 4

En la siguiente gráfica se representa el máximo, mínimo y medio número de operaciones básicas ejecutadas por el algoritmo *QuickSort*. Se puede observar que el mientras que el número medio y mínimo de OBs se mantiene bajo, la curva del máximo pronto se escinde de manera paulatina, pero notoria a la larga.

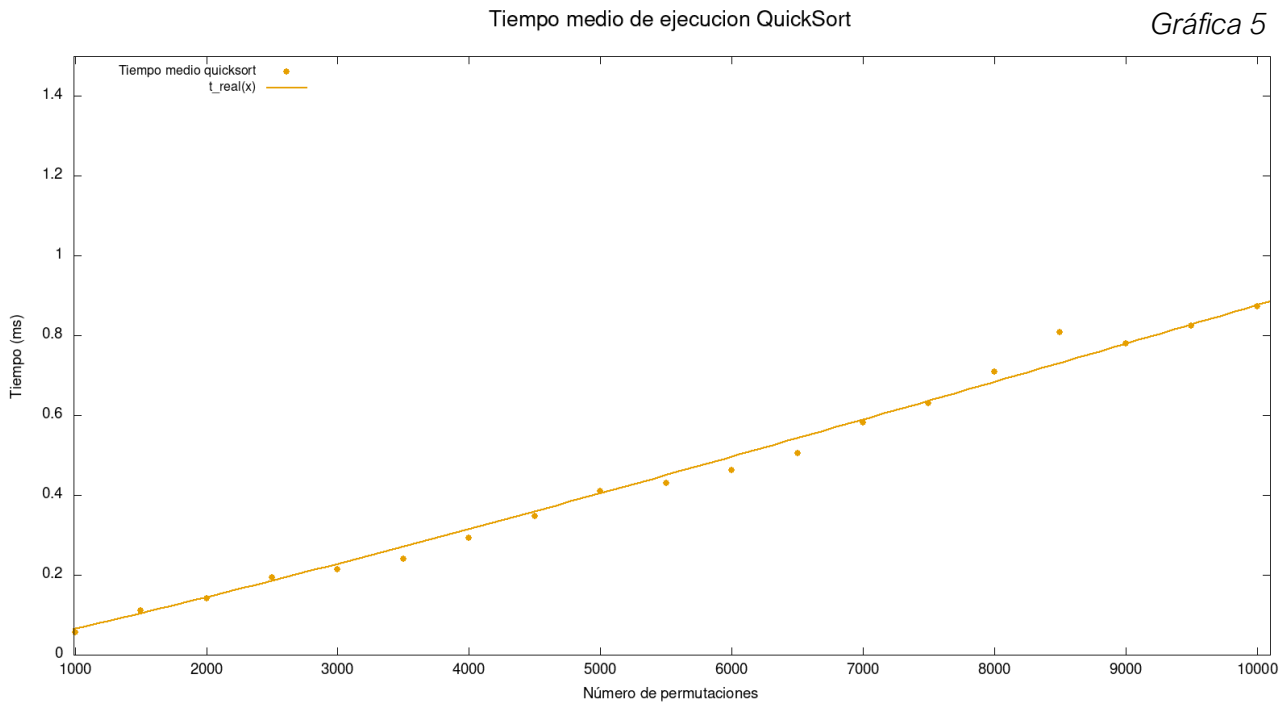
A continuación, tomando puntos de 500 en 500 en el intervalo [1000,10000], y efectuando 500 iteraciones por cada tamaño, se muestra la gráfica con el tiempo medio de ejecución de *QuickSort*:



Gráfica 4

Este resultado concuerda con la deducción teórica del coste de ejecución de *QuickSort* para su caso medio, $A_{QS} = 2*nlog(n) + O(n) = O(n*log(n))$, y peor, $W_{QS} = (N^2 – N)/2$, respectivamente. En el caso mejor, $B_{QS} = (n*log(n))$. La posición relativa de los casos mejor y medio difiere en el factor de escala que acompaña a n*log(n), en este caso de 2. No obstante, se puede identificar un crecimiento parecido. Por otro lado, las funciones mejor y media se alejarán respecto del caso peor a medida que el tamaño de permutación aumenta, dado el carácter exponencial de este último.

A continuación, tomando puntos de 500 en 500 en el intervalo [1000,10000], y bajo las mismas condiciones de ejecución que la *Gráfica 4*, se muestra la gráfica con el tiempo medio de ejecución de *QuickSort*:

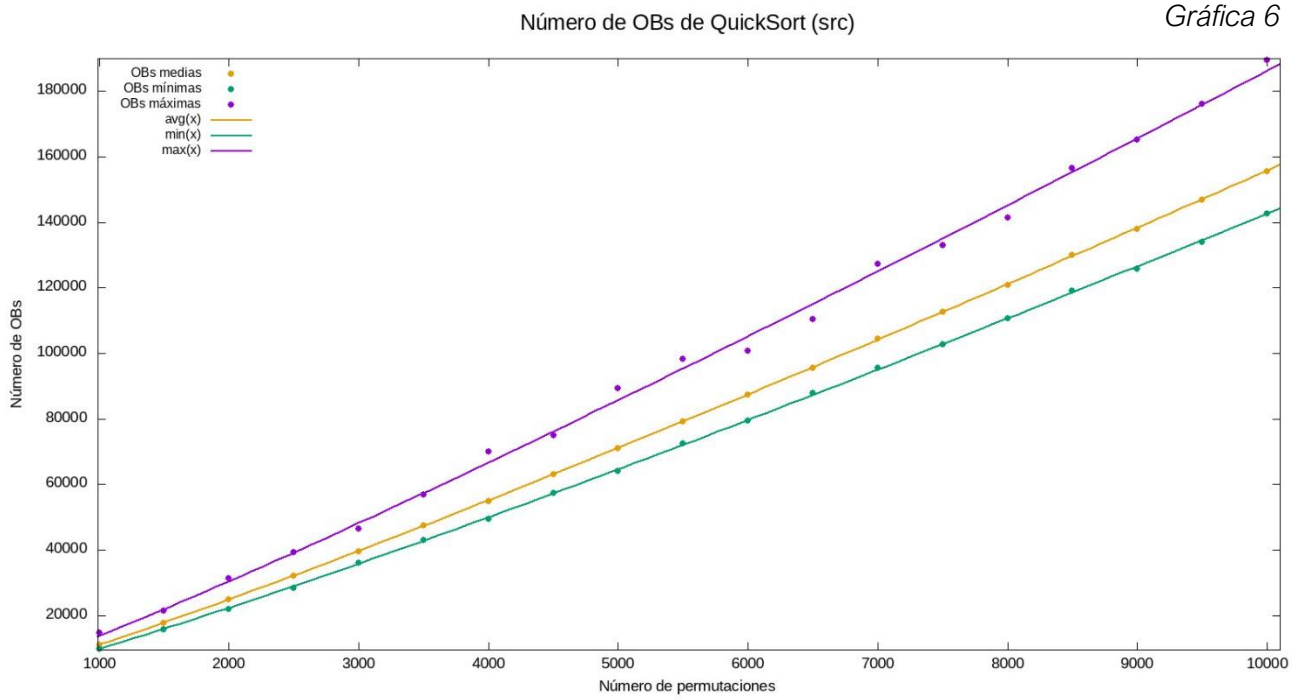


Gráfica 5

Entonces, se confirma que se trata de un crecimiento que corresponde a complejidad temporal n*log(n), como indicaban las deducciones teóricas.

5.5 Apartado 5

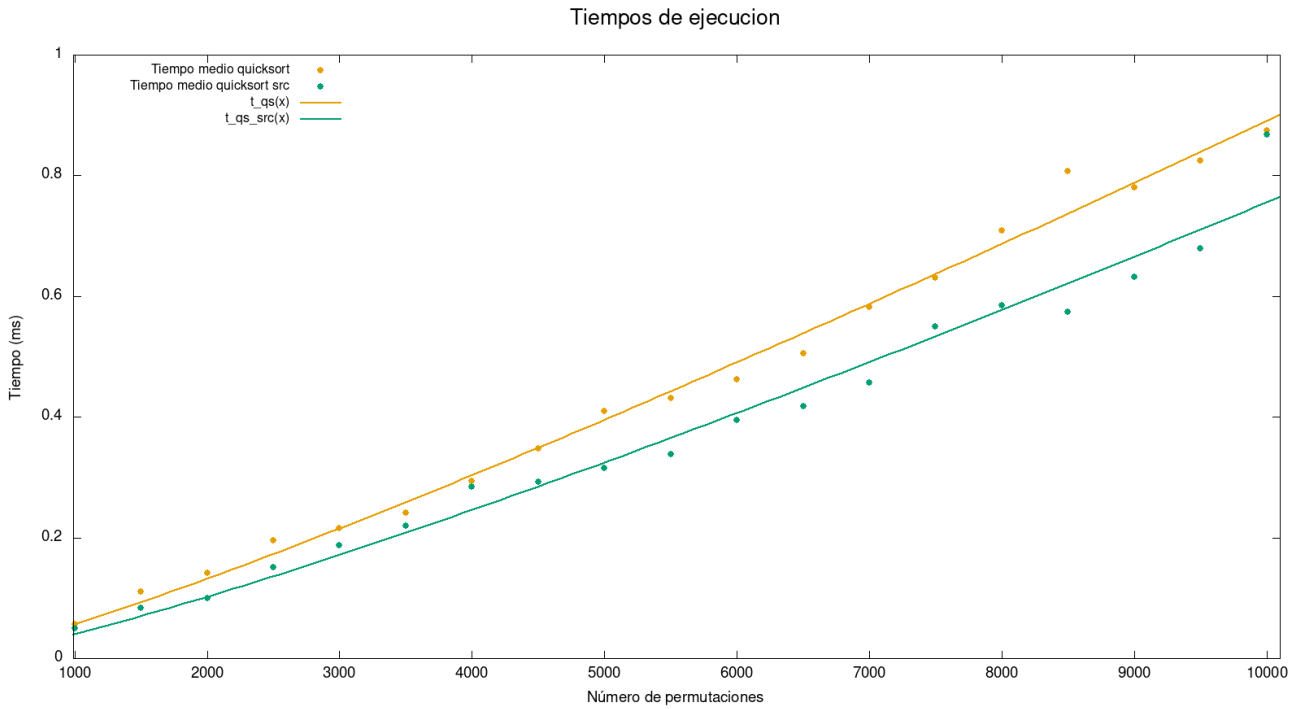
Tras reimplementar el algoritmo *QuickSort* en una función que no hace uso de la recursión de cola, vemos a continuación su resultado gráfico en cuanto a número de operaciones básicas, tomando valores de 500 en 500 en el intervalo [1000,10000], llevando a cabo 500 iteraciones por cada tamaño:



Gráfica 6

Como cabe esperar, y teniendo en cuenta que la función encargada del conteo de OBs es *partir*, los órdenes temporales del número medio, peor y mejor de operaciones básicas de *QuickSort_src* es invariante respecto de *QuickSort*. Los pequeños desajustes se justifican teniendo en cuenta la naturaleza de los *test* realizados, esto es, que existe cierto grado de aleatoriedad causado por el uso de *aleat_num* en la generación de permutaciones. No obstante, para una muestra suficientemente grande, estas disonancias se reducen, aunque no desaparecen.

A continuación, se muestra una gráfica comparando los tiempos medios de ejecución de las dos versiones de *QuickSort*: la recursiva, y la que no tiene recursión de cola:



En la misma línea que el planteamiento del enunciado, los resultados muestran un tiempo de ejecución menor en el caso de la implementación sin recursión de cola. La justificación de este fenómeno apunta directamente a la forma en que el compilador trata la recursión, y el uso de la pila en este proceso. Cuando se llama a una función, en la pila debe ser guardada la información local de esta función, de forma que no se pierda. Es por esto que, cuando se ejecuta una rutina que tiene varias llamadas a sí misma, cuando la ejecución avanza hasta la última – o recursión de cola –, los valores que previamente se habían almacenado en la pila tendrán que permanecer inoperativos en esta, hasta que se termine le ejecución de la última llamada. Si entre medias hay 100.000 interacciones con la función, tendrán que ser almacenados durante la mayoría de la ejecución del programa. Como es evidente, esta situación ocupa espacio de la pila y tiempo de ejecución, ya que se resume en muchas interacciones con esta. Uno de los problemas que se deriva es el desbordamiento de la pila, que puede no soportar almacenar más valores.

Una forma de mejorar significativamente el funcionamiento de *QuickSort* consiste en eliminar por completo la recursión. A fin de cuentas, por el razonamiento anterior, aunque se anule la cantidad de variables que almacena la pila, pasando estas, por ejemplo, en la llamada a *QuickSort*, la pila aún debe guardar la dirección de retorno, por lo que el desbordamiento todavía es posible. En consecuencia, la mejor opción sería implementar el algoritmo de forma no recursiva, para seguir las normas de una programación más dinámica y que minimice los costes temporales.

6. Respuesta a las preguntas teóricas

6.1 Pregunta 1

Como se puede observar en los *scripts* de *Gnuplot* realizados para la representación gráfica del rendimiento medio de *Mergesort* y *Quicksort*, hemos fijado una función $f(n) = an \cdot \log(an)$ en ambos casos para realizar el *fit* o regresión de los valores del número medio de OBs de ambos algoritmos:

```
avg(x) = a*x*log(a*x)
```

Si observamos la salida de estas regresiones podemos ver qué valor ha asignado *Gnuplot* a a. Cuánto más próximo sea a 1 este valor, más pequeño será el error entre los valores del número medio de OBs y la propia función $n \log n$.

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 1.46433       +/- 0.0002137    (0.01459%)
```

```
Final set of parameters            Asymptotic Standard Error
=======================            ==========================
a                  = 1.59706       +/- 0.00287      (0.1797%)
```

Vemos que los valores son próximos a 1 y que el error asintótico es cercano al 0% lo que significa que tanto *Mergesort* como *Quicksort* tienen el rendimiento medio esperado por la deducción teórica, que es en ambos casos de $\Theta(N \cdot \log \mathrm{N})$.

Cabe destacar que en las gráficas del número medio de OBs de ambos algoritmos no se observan "picos", pero si en la curva del máximo de OBs de *Quicksort.* Esto se debe a que, en su caso peor, el rendimiento de *Quicksort* pierde su carácter logarítmico para pasar a ser cuadrático: $\mathrm{W_{QS}} = \frac{\mathrm{N}^2}{2} - \frac{\mathrm{N}}{2}$. Esto es lo que provoca que su gráfica se vea más pronunciada.

6.2 Pregunta 2

Comparando las gráficas de ambas versiones del algoritmo *Quicksort* podemos notar, en el caso de las operaciones básicas, que se obtienen unos resultados prácticamente iguales. Se puede demostrar matemáticamente por inducción que, en efecto, el número de operaciones básicas que ejecutan ambas funciones es exactamente el mismo para una misma tabla. Esto también se puede comprobar si hacemos un programa que utilice los dos algoritmos para ordenar la misma tabla. El resultado será el mismo, pues en esencia se está ejecutando el mismo algoritmo.

No obstante, existe una pequeña pero notable diferencia entre las curvas del tiempo medio de ejecución de *quicksort* con respecto a *quicksort_src*. Puesto que esta diferencia no se debe al número de operaciones básicas por lo que se ha explicado anteriormente, podemos concluir que *Quicksort* es más rápida sin recursión de cola que con ella porque al eliminar llamadas a sí misma, se reduce el número de datos que quedan "suspendidos" en la pila de ejecución y su acceso a ellos. En la versión con recursión de cola, esos datos se acumularían en la pila hasta que se llega al caso base de la recursión y se recuperan al volver a las llamadas previas de la función, lo que reduce el tiempo de ejecución.

6.3 Pregunta 3

Aunque hemos representado curvas que indican el máximo y el mínimo número de OBs que cada algoritmo ejecuta en función del tamaño de las permutaciones, estas no sirven para representar con exactitud la tendencia que tendría el rendimiento de cada algoritmo en su caso mejor y peor, pues no dejan de ser permutaciones aleatorias.

Sería necesario que cada algoritmo ordenase tablas perfectamente ordenadas para obtener su rendimiento en el caso mejor. De la misma manera, cada algoritmo tiene una tabla para la cuál ejecutaría su mayor número de OBs, que no tiene por que ser la tabla ordenada a la inversa, y que permitiría representar el rendimiento del algoritmo en su caso peor. Así, conforme aumenta el tamaño de cada una de estas tablas, podríamos observar la evolución en el número de operaciones básicas de cada algoritmo en sus casos mejor y peor, y representar así sus respectivos rendimientos. Los resultados que se obtendrían serían los siguientes:

Caso peor *Mergesort*: $\mathrm{W_{MS}} \leq \mathrm{N} \cdot \log \mathrm{N} + \mathrm{O(N)}$.
Caso mejor *Mergesort*: $\mathrm{B_{MS}} \geq \frac{\mathrm{N}}{2} \log \mathrm{N} + \mathrm{O(N)}$.
Obsérvese que ambos cumplen $\Theta(N \cdot \log N)$.

Caso peor *Quicksort*: $\mathrm{W_{QS}} = \frac{\mathrm{N}^2}{2} - \frac{\mathrm{N}}{2}$.
Caso mejor *Quicksort*: $\mathrm{B_{QS}} = \Theta(N \cdot \log \mathrm{N})$.

6.4 Pregunta 4

Empíricamente, a través de la deducción teórica, tenemos los siguientes resultados:

$$\left\{ \begin{array}{l} W_{MS} = O(n*\log(n)) \\ B_{MS} = O(n*\log(n)) \\ A_{MS} = \Theta(n*\log(n)) \end{array} \right\} \qquad \left\{ \begin{array}{l} W_{QS} = O(n^2) + O(n) \\ B_{QS} = O(n*\log(n)) \\ A_{QS} = O(n*\log(n)) \end{array} \right\}$$

El algoritmo de ordenación *MergeSort* es mejor en lo que a complejidad temporal y de OBs se refiere, que es precisamente lo que hemos entendido por "mejor empíricamente". Es así que sus casos mejor y medio se encuentran en la misma línea, y corresponden a un mismo tipo de crecimiento temporal. Entonces, es el caso peor el que marca la diferencia ya que en *QuickSort* se trata de crecimiento exponencial, frente a crecimiento del orden n*log(n) en *MergeSort*. Además, es fácil predecir el comportamiento de los tiempos asociados a este último – información que puede resultar útil -, ya que, dada cualquier tabla, tardará aproximadamente lo mismo en ordenarla.

Aun así, hemos podido observar en la comparación de tiempos que el primer algoritmo - *MS* - tarda más en ordenar tablas que el segundo - *QS* -, dadas las mismas condiciones de partida. La principal razón reside en el uso de memoria auxiliar por parte de *MergeSort*. Para ordenar una permutación de 100 elementos, la primera función necesita reservar 1424 bytes de memoria, mientras que la segunda necesita 4112. En el caso de permutaciones en el intervalo [5,100] con saltos de 5 en 5 y 1000 iteraciones por tamaño, *QuickSort* emplea 4.366.242 de

bytes, frente a 29.670.232 que emplea *MergeSort*. Se pueden extrapolar estas conclusiones, para deducir que en un caso con tamaños de permutación significativamente grandes, la diferencia en gestión de memoria y los tiempos asociados a su manejo marca la diferencia. Por estas razones, *QuickSort* se convierte en una mejor opción.

7. Conclusiones finales

Tras haber realizado la implementación de *Mergesort*, *Quicksort*, y la versión de este último sin recursión de cola, haber estudiado sus respectivos rendimientos de forma empírica y haberlos comparado con la teoría aprendida en clase, podemos sacar una serie de conclusiones finales acerca la esta práctica de Análisis de Algoritmos.

En primer lugar, hemos aprendido, estudiado e implementado dos algoritmos del grupo conocido como "divide y vencerás", que buscan resolver el problema de la ordenación de una tabla mediante su reiterada división. Para ello hemos visto que hacen uso de la recursión, una estrategia muy útil en programación, pero que también tiene una serie de desventajas derivadas de la manera en que el compilador trata con ella.

Al igual que hicimos con los algoritmos de la práctica anterior, nos hemos centrado en su análisis empírico, principalmente a través de las funciones desarrolladas en *tiempos.c* que nos permiten exportar datos acerca de las operaciones básicas y el tiempo medio de ejecución de cada algoritmo. Dichos datos han sido más tarde representados por la herramienta *Gnuplot*, sobre la cual hemos aprendido bastantes utilidades como el uso de *scripts* para realizar las gráficas o la verdadera utilidad detrás de la función *fit* de este programa.

Por último, hemos podido comparar entre sí *Mergesort* y *Quicksort*, lo que nos ha permitido llegar a la conclusión firme de que no basta con conocer el rendimiento teórico de un algoritmo, sino que también es necesario experimentar con él para ver si en la práctica tiene lugar dicho rendimiento. Es el caso de *Mergesort*, que desde el punto de vista teórico parece más eficiente que *Quicksort,* pero debido al uso de memoria dinámica en la práctica no es tan efectivo.