

Análisis de Algoritmos 2018/2019

Práctica 2

Martín Sánchez Signorini, César Ramírez Martínez.

Grupo 1201

Código	Gráficas	Memoria	Total

1. Introducción.

Esta práctica consta de dos partes, cada una relacionada con un algoritmo, recursivo, de ordenación por comparación de clave, MergeSort y QuickSort. A lo largo de la práctica se hará uso de los resultados teóricos vistos en clase, se implementarán los métodos de ordenación y se obtendrán resultados que compararemos con las expectativas teóricas.

Primero se implementará MergeSort y se comprobará su correcto funcionamiento, a continuación se hará lo mismo con QuickSort. El objetivo será crear programas a los que se les pasará una serie de tamaños de tablas con el objetivo de que estos los ordenen mediante el correspondiente algoritmo. Finalmente se hará un estudio de los datos obtenidos, con lo cual se pretende profundizar en el estudio de algoritmos y ver su viabilidad en la vida real frente a la teoría.

2. Objetivos

2.1 Apartado 1

El objetivo de este apartado era el de implementar el famoso algoritmo de ordenación recursivo MergeSort en el fichero ordenacion.c ya utilizado en la práctica anterior.

Para ello creamos dos funciones. Por un lado MergeSort, que es la función principal y la que realiza las llamadas recursivas. Y por el otro está Merge, que es la función auxiliar encargada de combinar a la vez que ordenar las subtablas que MergeSort le va pasando. Ambas funciones devuelven ERR en caso de error o el número de operaciones básicas en caso de éxito. Este hecho nos permitirá medir el rendimiento, eficacia y coste de dicho algoritmo.

Para comprobar el correcto funcionamiento del algoritmo, lo probamos en el programa ejercicio4.c, reciclado de la práctica 1.

2.2 Apartado 2

En el segundo apartado hicimos uso del programa ejercicio5.c, pero con la modificación de que el algoritmo a probar ahora es MergeSort.

Gracias a este programa fuimos capaces de crear un fichero.log en el que se nos mostraba en 5 columnas diferentes:

- 1). El tamaño de las tablas que el algoritmo iba a ordenar
- 2). El tiempo medio que tardaba en ordenarlas las tablas de dicho tamaño.
- 3). El número medio de OBS que el algoritmo hacía para tablas de dicho tamaño
- 4). El número máximo de OBS realizadas por el algoritmo para tablas de dicho tamaño.
- 5). El número mínimo de OBS realizadas por el algoritmo para tablas de dicho tamaño.

Gracias a este fichero vamos a ser capaces de determinar el rendimiento del algoritmo en cuestión además de realizar las gráficas.

2.3 Apartado 3

El objetivo de este apartado es el de implementar el algoritmo de ordenación recursivo QuickSort. Para ello crearemos tres funciones en total.

Primeramente creamos QuickSort, que será la función encargada de realizar las llamadas recursivas y de llamar a otra importante función que conforma este algoritmo, Partir.

Partir será la función auxiliar encargada de ordenar los elementos de las tablas que la función QuickSort le va pasando. Para ello se apoyará en la función Medio, que, a su vez, se encarga de elegir el pivote que Partir empleará para ordenar relativamente los elementos de la tabla. Medio, en este caso, elegirá como pivote el primer elemento de la tabla.

Luego con estas tres funciones (QuickSort, Partir y Medio) seremos capaces de implementar el algoritmo QuickSort y probaremos su correcto funcionamiento de nuevo en el programa ejercicio4.c.

2.4 Apartado 4

En este apartado realizaremos algo muy parecido a lo hecho en el apartado 2. Es decir, probando el algoritmo QuickSort con el programa ejercicio5.c, seremos capaces de crear un fichero.log de la misma manera que antes. De esta forma podremos comprobar el rendimiento de este algoritmo. De nuevo este fichero será sobre los que luego nos apoyaremos para realizar las gráficas.

2.5 Apartado 5

En este apartado se nos pide que implementemos dos funciones más para elegir el pivote sobre el que la función Partir trabajará de maneras distintas. Es decir, se nos pide implementar dos variantes de la función Medio.

Por un lado tenemos la función Medio_Avg, que simplemente elige como pivote el índice del elemento que está en la posición intermedia de la tabla.

Y por otro tenemos Medio_Stat, una función algo más interesante. Esta función escoge la posición del valor intermedio entre el primer elemento, el último y el que ocupa la posición intermedia de la tabla.

De las tres formas de escoger el pivote que hemos visto, esta es la más eficiente. Esta es la forma en la que se tienen más probabilidades de escoger como pivote la posición de un elemento cuyo valor sea intermedio con respecto al resto de valores de los elementos de la tabla. Algo que mejora considerablemente el rendimiento del algoritmo.

3. Herramientas y metodología

A lo largo de la práctica hemos empleado como entorno de desarrollo el sistema operativo Ubuntu instalado en los ordenadores de la EPS. También hemos empleado, aunque en menor medida, Windows y MacOS para modificaciones esporádicas y pruebas del código realizadas fuera del horario lectivo.

Con respecto a las herramientas de desarrollo, hemos usado el editor de texto Atom para escribir todo el código. Además, para sincronizarnos también hemos optado por crear un repositorio en Github donde fuimos guardando los resultados y el código. Para compilar y generar las gráficas hemos optado por usar exclusivamente Valgrind y Gnuplot, disponibles en los laboratorios de la escuela.

3.1 Apartado 1

En el primer apartado tuvimos que implementar el método de ordenación MergeSort. Para ello escribimos las rutinas necesarias *MergeSort* y *merge* utilizando los algoritmos vistos en clase. Además, tuvimos que modificar el ejercicio 4 de la anterior práctica, esta vez empleando MergeSort, programa que escribimos en el archivo *MergeSort_1*.

3.2 Apartado 2

Para la segunda parte de MergeSort únicamente modificamos solamente el ejercicio 5 de la anterior práctica, pudiendo obtener así ficheros log con la recopilación de los resultados de MergeSort. A este archivo lo llamamos *MergeSort_2*.

3.3 Apartado 3

En este apartado implementamos el método de ordenación QuickSort. El apartado consistió en escribir las rutinas *QuickSort*, *partir* y *medio*. Para los dos primeros usamos los algoritmos vistos en teoría, mientras que *medio* consistió en una primera implementación de selección de pivote, escogiendo simplemente el primero de la tabla. Además, tuvimos que modificar el ejercicio 4 nuevamente, esta vez empleando QuickSort. Lo escribimos en el archivo *QuickSort_1*.

3.4 Apartado 4

Para la segunda parte de QuickSort únicamente fue modificado el ejercicio 5 usando esta vez QuickSort. Así pudimos obtener ficheros con la recopilación de los datos de QuickSort. A este archivo lo llamamos *QuickSort_2*.

3.5 Apartado 5

Finalmente el apartado 5 consistía en experimentar con distintos métodos de selección de pivote. Primero implementamos *medio_avg* y *medio_stat* que seleccionaban el pivote de en medio de la tabla y el que estaba tenía el valor medio entre el primero, el último y el de en medio de la tabla. Para usar estas funciones optamos por cambiar manualmente el código de partir cada vez que usamos una. Además, *medio_stat* fue implementada usando entre 2 o 3 OBs con 'ifs' encadenados.

4. Código fuente

Código fuente **exclusivamente de las rutinas que hemos desarrollado nosotros** en cada apartado.

4.1 MergeSort

```
int MergeSort(int* tabla, int ip, int iu) {
    if(tabla == NULL || ip < 0 || iu < ip) return ERR;
    if(ip == iu) return 0;
    int imedio = (iu+ip)/2;
    int obs = 0;
    int total = 0;
    /* Comprobaciones de error necesarias pues merge reserva memoria */
    obs = MergeSort(tabla, ip, imedio);
    if(obs == ERR) return ERR;
    total += obs;
    obs = MergeSort(tabla, imedio+1, iu);
    if(obs == ERR) return ERR;
    total += obs;
    obs = merge(tabla, ip, iu, imedio);
    if(obs == ERR) return ERR;
    total += obs;
    return total;
}

int merge(int* tabla, int ip, int iu, int imedio)
{
    if(tabla == NULL || ip < 0 || iu < ip || imedio < ip || imedio > iu) return ERR;
    if(ip == iu) return 0;
    int* Taux = NULL;
    int i = 0;
    int j = 0;
    int k = 0;
    int obs = 0;
    Taux = (int*)malloc((iu-ip+1)*sizeof(int));
    if(Taux == NULL) {
        return ERR;
    }
    /* Combinacion de las subtablas */
    i = ip; /* i va de ip a imedio*/
    j = imedio+1; /* j va de imedio+1 a iu*/
    while(i <= imedio && j <= iu) {
        obs++;
        if(tabla[i] < tabla[j]) {
            Taux[k++] = tabla[i++];
        }
        else {
            Taux[k++] = tabla[j++];
        }
    }
    /* Si el indice desbordado es i, volcamos j*/
    for(; j <= iu; j++) {
        Taux[k++] = tabla[j];
    }
    /* Si el indice desbordado es j, volcamos i*/
    for(; i <= imedio; i++) {
        Taux[k++] = tabla[i];
    }
}
```

```

/* Copiar la Taux en tabla, k = tam de Taux */
for(i = 0; i < k; i++) {
    tabla[i + ip] = Taux[i];
}
free(Taux);
return obs;
}

```

4.3 QuickSort

```

int QuickSort(int* tabla, int ip, int iu) {
    if(tabla == NULL || ip < 0 || iu < ip) return ERR;
    if(ip == iu) return 0;
    int ipiv = 0; /* Index del pivote tras partir */
    int obs = 0;
    int total = 0;
    obs = partir(tabla, ip, iu, &ipiv);
    if(obs == ERR) return ERR;
    total += obs;
    /* Si ip < ipiv - 1 hace falta seguir ordenando la parte izquierda */
    if(ip < ipiv-1) {
        obs = QuickSort(tabla, ip, ipiv-1);
        if(obs == ERR) return ERR;
        total += obs;
    }
    /* Si iu > ipiv + 1 hace falta seguir ordenando la parte derecha */
    if(iu > ipiv+1) {
        obs = QuickSort(tabla, ipiv+1, iu);
        if(obs == ERR) return ERR;
        total += obs;
    }
    return total;
}

```

```

int partir(int* tabla, int ip, int iu, int *pos) {
    if(tabla == NULL || ip < 0 || iu < ip || pos==NULL) return ERR;
    int obs = 0;
    int k = 0;
    int i = 0;
    /* pos es la posicion del pivote, usamos algún 'medio' */
    obs = medio_stat(tabla, ip, iu, pos);
    if(obs == ERR) return ERR;
    /* ordenar relativamente al pivote pos */
    k = tabla[*pos];
    SWAP(tabla[*pos], tabla[ip]);
    *pos = ip;
    for(i = ip+1; i <= iu; i++) {
        /* pos va representando la posicion final del pivote */
        /* aumenta de 1 en 1 por cada elemento menor (izq) */
        obs++;
        if(tabla[i] < k) {
            *pos += 1;
            SWAP(tabla[i], tabla[*pos]);
        }
    }
    SWAP(tabla[ip], tabla[*pos]);
    return obs;
}

```

4.5 Selección de Pivote

```
int medio(int *tabla, int ip, int iu, int *pos) {
    if(tabla == NULL || ip < 0 || iu < ip || pos == NULL) return ERR;
    *pos = ip;
    return 0;
}

int medio_avg(int *tabla, int ip, int iu, int *pos) {
    if(tabla == NULL || ip < 0 || iu < ip || pos == NULL) return ERR;
    *pos = (ip + iu)/2;
    return 0;
}

int medio_stat(int *tabla, int ip, int iu, int *pos) {
    if(tabla == NULL || ip < 0 || iu < ip || pos == NULL) return ERR;
    int im = (ip + iu)/2;
    if(tabla[ip] < tabla[iu]) {
        if(tabla[ip] > tabla[im]) {
            *pos = ip;
            return 2;
        }
        if(tabla[im] > tabla[iu]) {
            *pos = iu;
            return 3;
        }
        *pos = im;
        return 3;
    }
    if(tabla[iu] > tabla[im]) {
        *pos = iu;
        return 2;
    }
    if(tabla[im] > tabla[ip]) {
        *pos = ip;
        return 3;
    }
    *pos = im;
    return 3;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

En este apartado fuimos capaces de comprobar el correcto funcionamiento del algoritmo MergeSort implementado gracias al programa ejercicio4.c. Como se puede apreciar en la siguiente imagen, la tabla de 20 elementos ha sido correctamente ordenada, y no se ha dado ningún tipo de pérdida de memoria.

Aquí se aprecia su correcto funcionamiento:

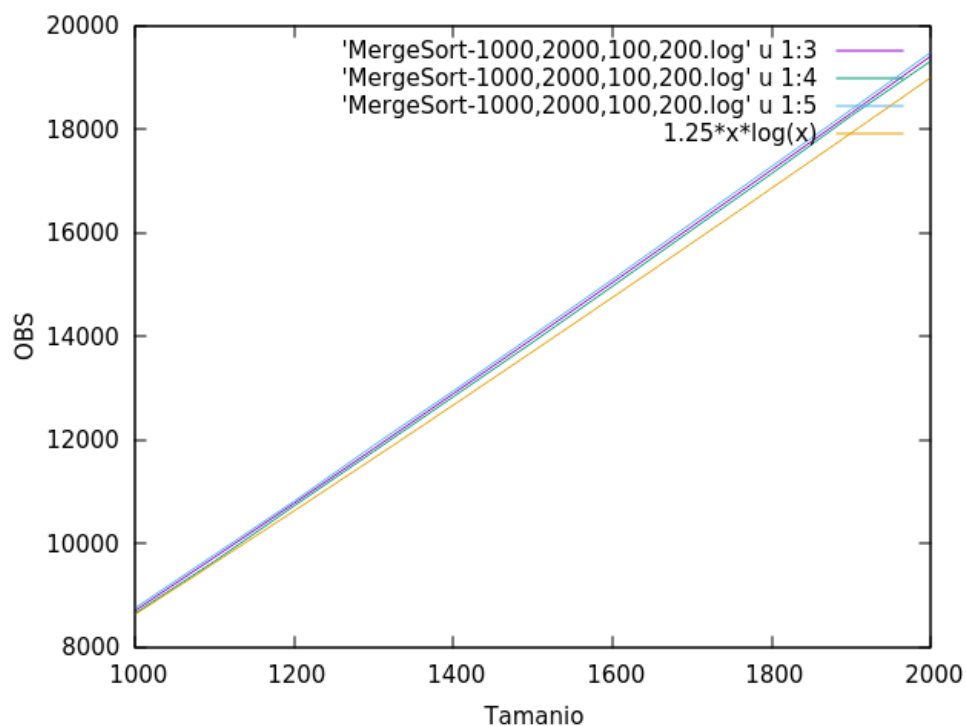
```

root@kali: ~/Documentos/Análisis de algoritmos/practica 2
Archivo Editar Ver Buscar Terminal Ayuda
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make
make: No se hace nada para 'all'.
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make MergeSort-test
make: *** No hay ninguna regla para construir el objetivo 'MergeSort-test'. Alto.
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make MergeSort-test
make: *** No hay ninguna regla para construir el objetivo 'MergeSort-test'. Alto.
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make MergeSort_test
Ejecutando MergeSort
==3002== Memcheck, a memory error detector
==3002== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3002== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3002== Command: ./MergeSort_1 -tamano 20
==3002==
Practica numero 2, MergeSort - 1
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20
==3002==
==3002== HEAP SUMMARY:
==3002== in use at exit: 0 bytes in 0 blocks
==3002== total heap usage: 21 allocs, 21 frees, 1,456 bytes allocated
==3002==
==3002== All heap blocks were freed -- no leaks are possible
==3002==
==3002== For lists of detected and suppressed errors, rerun with: -s
==3002== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@kali:~/Documentos/Análisis de algoritmos/practica 2#

```

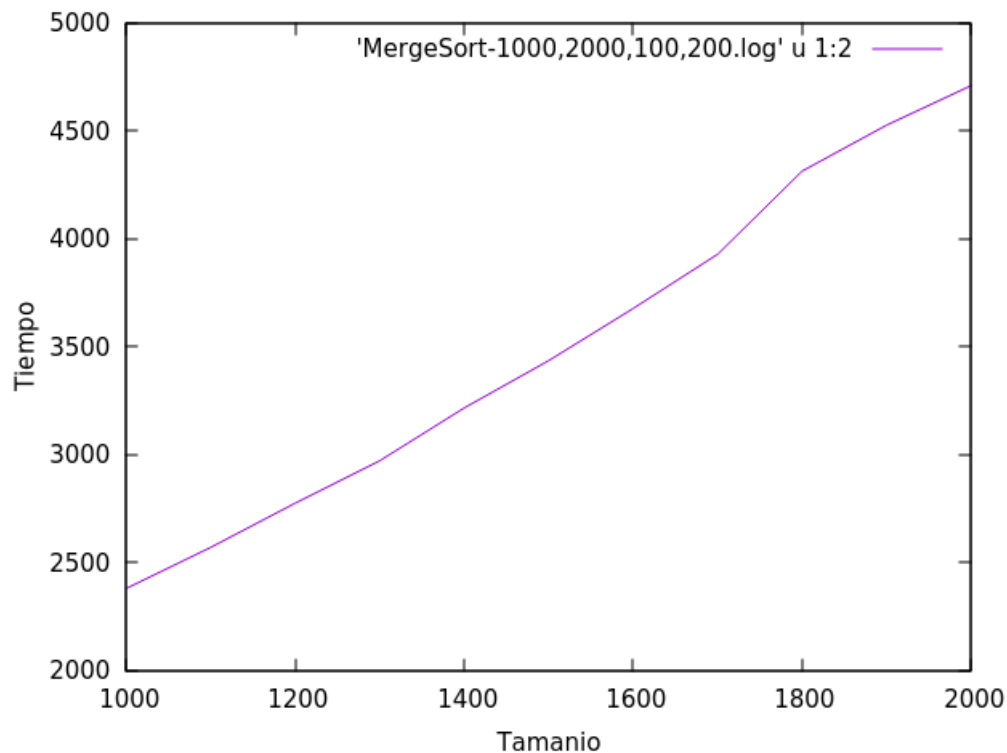
5.2 Apartado 2

En este apartado utilizamos el programa ejercicio5.c para que se nos generase un fichero.log gracias al que pudimos observar con eficacia el rendimiento de MergeSort. Lo que le pedimos a MergeSort fue que nos ordenase 200 tablas de 1000 a 2000 elementos con un incremento de 100 en 100 elementos.



En esta gráfica podemos apreciar el número máximo (en azul), número medio (en rosa) y el número mínimo (en verde) de OBS que realiza el algoritmo para tablas de tamaño creciente hasta 2000 elementos por tabla. También hemos añadido la gráfica de la función $1,25 \cdot x \cdot \log(x)$, que se ajusta bastante bien a las anteriores.

Este último hecho nos hace darnos cuenta del correcto funcionamiento del algoritmo. Ya que como sabemos el caso mejor, medio y peor de este es de $O(N \log(N))$.



En esta gráfica se aprecia el tiempo medio que tarda MergeSort en ordenar tablas de 1000 a 2000 elementos. Vemos que es coherente al ir creciendo a medida que también lo hace el tamaño de las tablas.

5.3 Apartado 3

En este apartado fuimos capaces de comprobar el correcto funcionamiento del algoritmo QuickSort implementado gracias de nuevo al programa ejercicio4.c. Como se puede apreciar en la siguiente imagen, la tabla de 20 elementos ha sido correctamente ordenada, y no se ha dado ningún tipo de pérdida de memoria.

Aquí se aprecia su correcto funcionamiento:

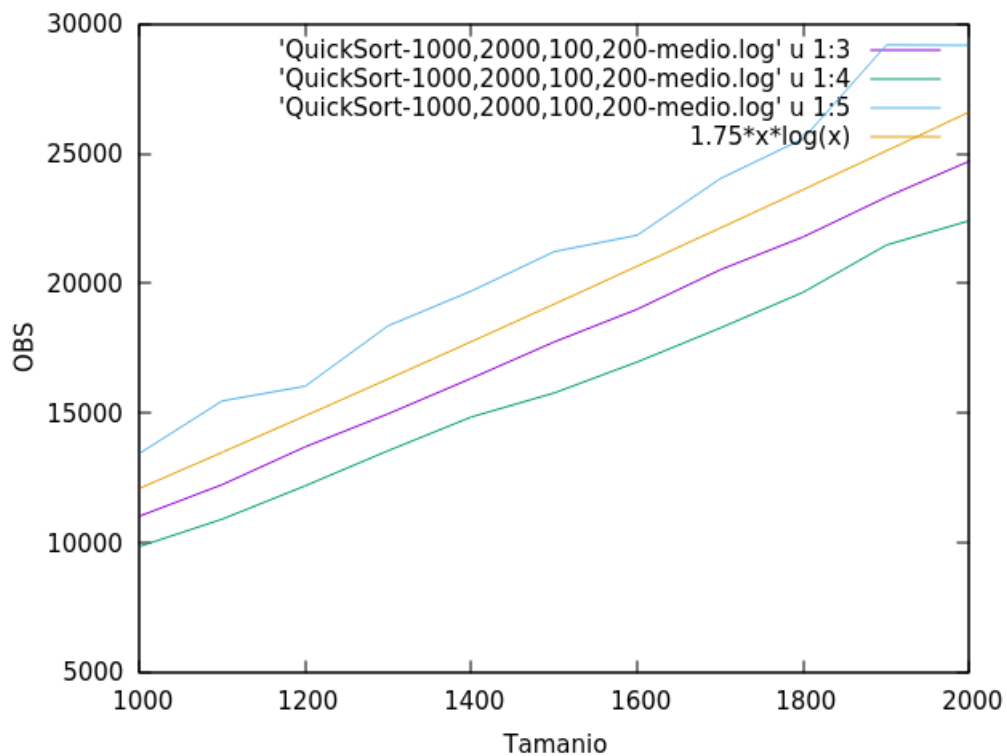
```

root@kali:~/Documentos/Análisis de algoritmos/practica 2# make QuickSort_test
Ejecutando QuickSort
==3013== Memcheck, a memory error detector
==3013== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3013== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3013== Command: ./QuickSort_1 -tamaño 20
==3013==
Practica numero 2, QuickSort - 1
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
16 17 18 19 20
==3013==
==3013== HEAP SUMMARY:
==3013== in use at exit: 0 bytes in 0 blocks
==3013== total heap usage: 2 allocs, 2 frees, 1,104 bytes allocated
==3013==
==3013== All heap blocks were freed -- no leaks are possible
==3013== For lists of detected and suppressed errors, rerun with: -s
==3013== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
root@kali:~/Documentos/Análisis de algoritmos/practica 2#

```

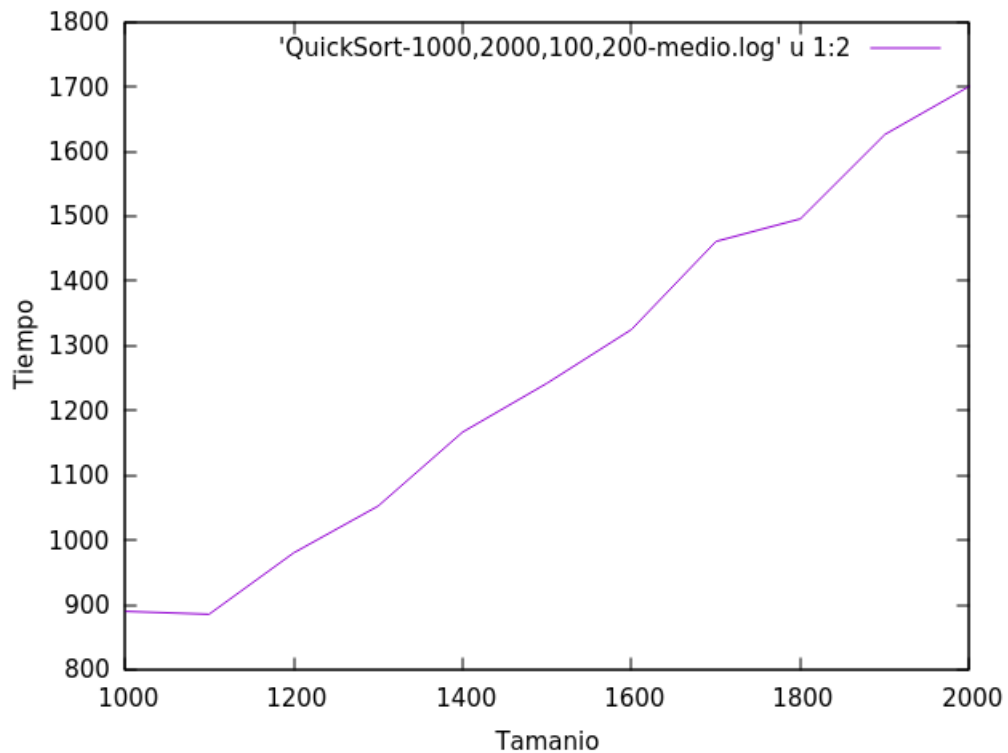
5.4 Apartado 4

En este apartado utilizamos el programa ejercicio5.c para que se nos generase un fichero.log gracias al que pudimos observar con eficacia el rendimiento de QuickSort. Lo que le pedimos a QuickSort fue que nos ordenase 200 tablas de 1000 a 2000 elementos con un incremento de 100 en 100 elementos.



En esta gráfica podemos apreciar el número máximo (en azul), número medio (en rosa) y el número mínimo (en verde) de OBS que realiza el algoritmo para tablas de tamaño creciente hasta 2000 elementos por tabla. También hemos añadido la gráfica de la función $1,75 \cdot x \cdot \log(x)$, que se ajusta bastante bien a las gráficas azul (máximo de OBS) y rosa (media OBS).

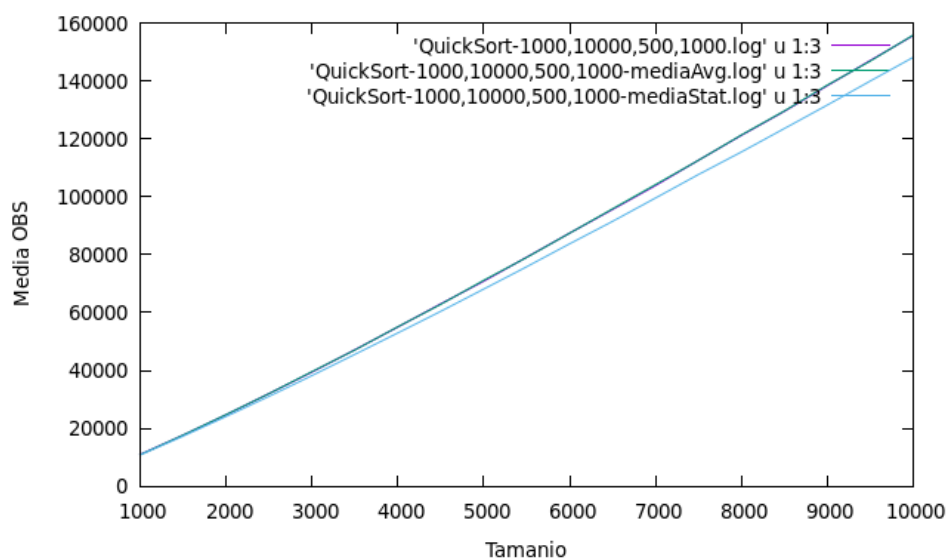
Este último hecho nos hace darnos cuenta del correcto funcionamiento del algoritmo. Ya que como sabemos el caso mejor y medio de este es de $O(N \log(N))$.



En esta gráfica se aprecia el tiempo medio que tarda QuickSort en ordenar tablas de 1000 a 2000 elementos. Vemos que es coherente pues para tablas pequeñas el tiempo se desvía de la media y en QuickSort la diferencia entre el tiempo medio y el peor es mucho mayor. A continuación, para mayores tablas se verá una gráfica más precisa.

5.5 Apartado 5

En este apartado compararemos mediante una gráfica las diferencias existentes entre el funcionamiento y rendimiento de QuickSort con las diferentes variantes de la función Medio que implementamos antes. En este caso le pedimos a QuickSort que nos ordenase 1000 tablas de 1000 a 10000 elementos con un incremento de 500 en 500 elementos.



La gráfica rosa se corresponde con el número medio de OBS que realiza QuickSort para ordenar tablas (de tamaño 1000 a 10000) con la función Medio inicial, la que simplemente elegía como pivote el primer elemento de la tabla.

La gráfica verde se corresponde con el número medio de OBS que realiza QuickSort para ordenar tablas (de tamaño 1000 a 10000) con la función Medio_Avg. Esta elegía como pivote el elemento que se encontraba justo en la posición intermedia de la tabla.

Y por último tenemos la gráfica azul, que se corresponde con el número medio de OBS que realiza QuickSort para ordenar tablas (de tamaño 1000 a 10000) con la función Medio_Stat. Esta comparaba los valores de los elementos inicial, intermedio y último de la tabla, y elegía como pivote el elemento cuyo valor fuese el intermedio de los tres.

Como es de esperar es esta última la función que mejor rendimiento da al algoritmo. Esto se debe a que la manera más rápida de ordenar una tabla es eligiendo como pivote el elemento cuyo valor es el intermedio de todos los valores de los elementos de la tabla. Y esta función es la que más se acerca a realizar exitosamente esta tarea.

Las otras dos variantes de Medio, son exactamente igual de eficientes.

5. Respuesta a las preguntas teóricas.

5.1 Pregunta 1

Los casos medios teóricos son:

$$\text{MergeSort: } A_{MS}(N) = \Theta(N \cdot \log(N))$$

$$\text{QuickSort: } A_{QS}(N) = 2N \cdot \log(N) + O(N)$$

A partir de las gráficas en las que se representa la media de operaciones básicas y la función $x \cdot \log(x)$ multiplicada por una constante, observamos que en efecto, las gráficas se asemejan.

Una observación relevante es que, en el caso de MergeSort, los resultados experimentales concuerdan más directamente con la teoría, mientras que en QuickSort, observamos picos y un considerablemente peor caso medio. Esto se debe probablemente a que el número de tablas empleadas no es muy significativo y que en QuickSort el caso peor es de la forma $(N^2 - N)/2$, y al haber un número no muy elevado de tablas que ordenar, se obtienen permutaciones desviadas del caso medio.

Como MergeSort no tiene un caso peor tan malo, sino uno de la forma $N \cdot \log(N) + O(N)$, el resultado es más uniforme.

5.2 Pregunta 2

En la última gráfica se puede apreciar que al usar QuickSort con un gran número de tablas muy grandes, la diferencia entre las distintas selecciones de pivotes se va haciendo más notable.

Por un lado, resulta que escoger el primer elemento o el de en medio de la tabla no tiene diferencia, lo cual es lógico pues en ambos casos la selección es arbitraria y existen las mismas posibilidades de que un elemento esté al principio o en medio de la tabla.

Por otro lado, si escogemos el mejor pivote entre el primer elemento, el último y el de en medio de la tabla, quedándonos con el que tenga el valor intermedio; aunque realicemos más comparaciones para elegir el pivote, resulta intuitivo que estadísticamente esta selección de pivote acabe mejorando la velocidad de QuickSort, pues escogemos el mejor entre tres posibilidades. Sin embargo, la mejora, aunque apreciable, no es muy significativa, y el algoritmo sigue siendo del orden $2N \cdot \log(N) + O(N)$ en el caso medio.

5.3 Pregunta 3

Los casos peores teóricos son:

$$\text{MergeSort: } W_{MS}(N) = N \cdot \log(N) + O(N)$$

$$\text{QuickSort: } W_{QS}(N) = (N^2 - N)/2$$

Para obtener estrictamente estos casos peores en la práctica habría que utilizar únicamente aquellas permutaciones que provoquen un caso peor en los algoritmos.

Por ejemplo, para QuickSort habría que usar solamente la tabla ya ordenada, en caso de coger como pivote el primer elemento. Obtendríamos así su caso peor estrictamente para cada tamaño.

En el caso de MergeSort, el caso peor sería más complicado de hallar, pero se podría obtener empezando con la tabla ordenada y viendo que subtablas serían necesarias para que cuando se desborde la primera variable al combinarlas, la otra ya esté en el último elemento, y recursivamente podríamos hallar la peor tabla posible para MergeSort, la cual sería la que usásemos para hallar su peor caso estrictamente.

5.4 Pregunta 4

Observando las gráficas que representan el tiempo necesario en función del tamaño de la tabla se puede ver que MergeSort llega a tardar 4000 unidades de tiempo, mientras que, para el mismo tamaño, QuickSort lo hace en apenas 1700 unidades de tiempo. Luego empíricamente QuickSort es superior a MergeSort pues su tiempo de ejecución es considerablemente inferior.

Sin embargo, la predicción teórica nos decía que MergeSort sería mejor. Esto se debe a que en la teoría solamente contamos el número de comparaciones de clave que los algoritmos realizan, cuando en la vida real también influyen factores como: cualquier otro tipo de operación costosa (SWAPs, reservas de memoria, acceso a memoria etc.) las cuales no se tenían en cuenta en el análisis teórico y que, sin embargo, acaban resultando decisivas en la práctica.

En cuestión de memoria, MergeSort requiere reservas de memorias adicionales para ir guardando la combinación de las subtablas, mientras que QuickSort es un algoritmo *in-place*, es decir, no necesita memoria adicional. Por lo tanto, QuickSort es indudablemente superior a MergeSort en cuanto a gasto de memoria. Esto se puede ver además de manera experimental al usar Valgrind, obteniendo tres veces más bytes reservados en MergeSort con respecto a QuickSort.

6. Conclusiones finales.

Esta práctica nos ha hecho darnos cuenta de una manera práctica de todos los contenidos explicados en el tema 2 en clases teóricas.

Hemos sido capaces de observar y entender con mucho mayor detalle cómo funcionan algunos algoritmos de ordenación recursivos y mucho más eficientes que los anteriores vistos como son MergeSort e QuickSort y, sobre todo, cómo calcular su rendimiento y eficacia.

Gracias a las gráficas hemos podido comparar los números de OBs que hacían nuestros algoritmos para permutaciones de diferentes tamaños y entender cómo estos funcionaban en realidad. Otra cosa que llama la atención es como, de manera práctica, se puede observar que, en cuanto al rendimiento de ambos algoritmos (QuickSort e MergeSort) para ordenar tablas, no existe demasiada diferencia en cuanto OBs se refiere. Pero ni nos fijamos en el tiempo que tarda cada algoritmo en ordenar y no en las OBs, te das cuenta de que QuickSort es bastante más rápido. Además de que este requiere usar mucha menos memoria de la que lo hace MergeSort. Al utilizar herramientas como Valgrind te das cuenta rápidamente de este hecho.

Hemos concluido en esta práctica que claramente el uso de algoritmos recursivos frente a los no recursivos mejora muy considerablemente el rendimiento y el tiempo que estos tardan en ordenar las permutaciones.

```
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make MergeSort_analysi
s MergeSort_ MergeSort_ ordenacion.c ordenacion.h
Ejecutando MergeSort recopilando datos
==3348== Memcheck, a memory error detector
==3348== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3348== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3348== Command: ./MergeSort_2 -num_min 1 -num_max 10 -incr 1 -numP 1000000 -fi
chSalida MergeSort-1,10,1,1000000.log
==3348==
Practica numero 2, MergeSort - 2 permutaciones.h permutaciones.o
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
Salida correcta
==3348==
==3348== HEAP SUMMARY:
==3348==   in use at exit: 0 bytes in 0 blocks
==3348==   total heap usage: 55,000,014 allocs, 55,000,014 frees, 900,005,992 by
tes allocated 0 || iu < ip || pos == NULL) return ERR;
==3348==
==3348== All heap blocks were freed -- no leaks are possible
==3348==
page 0/22 2019
```

```

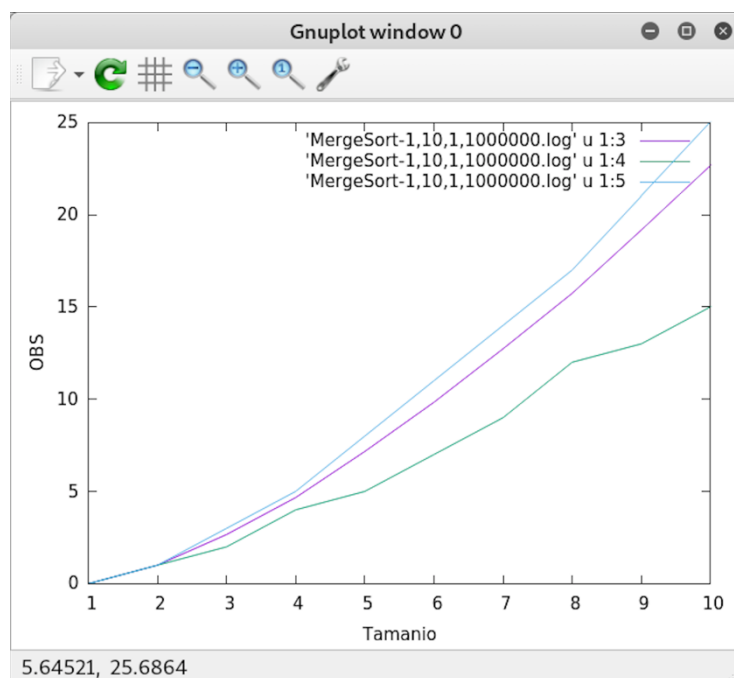
root@kali:~/Documentos/Análisis de algoritmos/practica 2# make QuickSort_analysi
s
Ejecutando QuickSort recopilando datos
==3317== Memcheck, a memory error detector
==3317== Copyright (C) 2002-2017, and GNU GPL'd, by Julian Seward et al.
==3317== Using Valgrind-3.15.0 and LibVEX; rerun with -h for copyright info
==3317== Command: ./QuickSort_2 -num_min 1 -num_max 10 -incr 1 -numP 1000000 -fi
chSalida QuickSort-1,10,1,1000000.log
==3317==
Practica numero 2, QuickSort - 2
Realizada por: César Ramírez & Martín Sánchez
Grupo: 120
Salida correcta
==3317==
==3317== HEAP SUMMARY:
==3317==   in use at exit: 0 bytes in 0 blocks
==3317==   total heap usage: 10,000,014 allocs, 10,000,014 frees, 300,005,992 by
tes allocated
==3317== All heap blocks were freed -- no leaks are possible
==3317==
==3317== For lists of detected and suppressed errors, rerun with: -s
==3317== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)

```

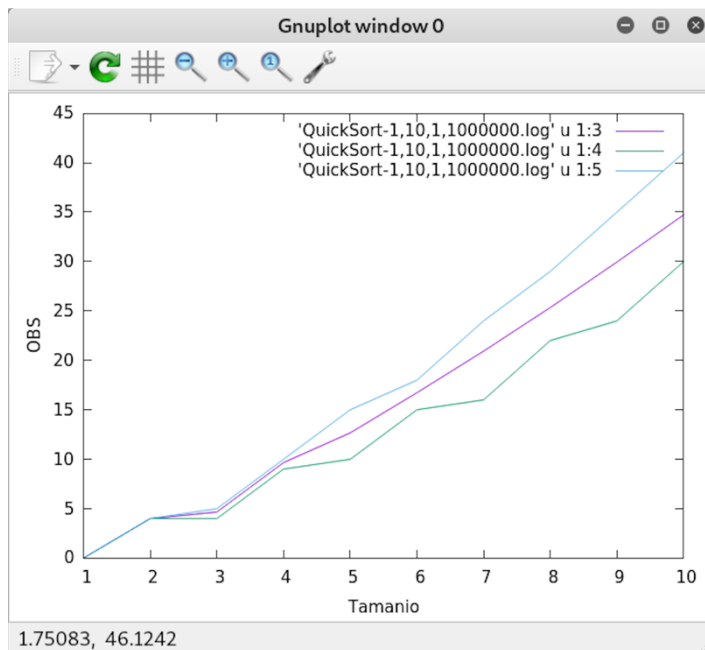
Para comprobar si los resultados experimentales de los casos peor, mejor y medio se asemejaban a los teóricos, nos generamos 1000000 de tablas de 1 a 10 elementos, para intentar obtener todas las posibles permutaciones. A continuación las ordenamos con ambos algoritmos. Y finalmente vemos si lo que obtuvimos concuerda con lo explicado en clase. Aquí se aprecia la diferencia en cuanto al uso de memoria se refiere.

Estas fueron las gráficas obtenidas:

-Para MergeSort:



-Para QuickSort:



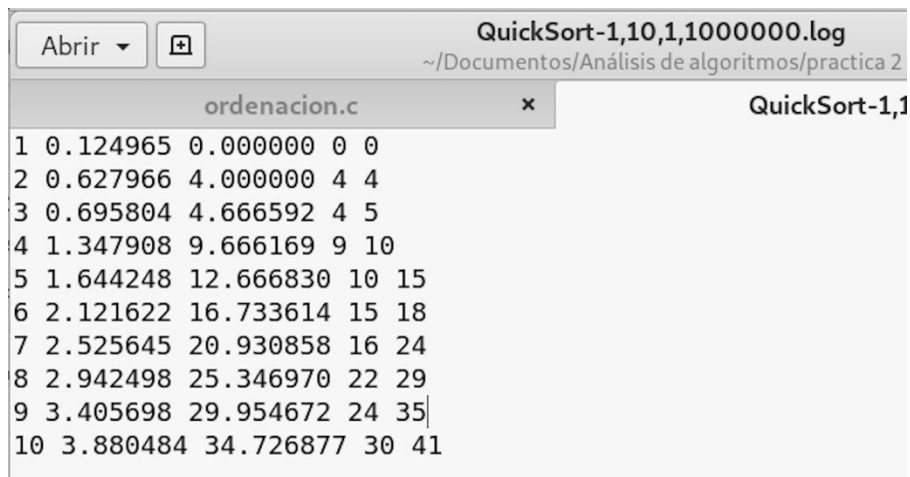
Y estos fueron los ficheros.log generados:

-Para MergeSort:

MergeSort-1,10,1,1000000.log					
~/Documentos/Análisis de algoritmos/practica 2					
1	0.127391	0.000000	0	0	
2	1.972898	1.000000	1	1	
3	3.794449	2.666227	2	3	
4	5.070841	4.665943	4	5	
5	6.906820	7.168704	5	8	
6	8.715594	9.833831	7	11	
7	10.642351	12.731996	9	14	
8	12.452279	15.734114	12	17	
9	14.495543	19.167369	13	21	
10	16.366545	22.665513	15	25	

Un ejemplo claro de que estos resultados se asemejan con los teóricos, es el caso de, por ejemplo, las tablas de $N = 9$ elementos, en las que se aprecia un caso mejor de 13 OBs ($0,46 \cdot N \cdot \log_2(N)$), uno peor de 21 OBs ($0,74 \cdot N \cdot \log_2(N)$) y uno medio de 19 ($0,67 \cdot N \cdot \log_2(N)$). Luego todos los casos son del orden de $O(N \cdot \log_2(N))$, tal y cómo esperábamos.

-Para QuickSort:



	ordenacion.c	QuickSort-1,1
1	0.124965 0.000000 0 0	
2	0.627966 4.000000 4 4	
3	0.695804 4.666592 4 5	
4	1.347908 9.666169 9 10	
5	1.644248 12.666830 10 15	
6	2.121622 16.733614 15 18	
7	2.525645 20.930858 16 24	
8	2.942498 25.346970 22 29	
9	3.405698 29.954672 24 35	
10	3.880484 34.726877 30 41	

Un ejemplo en este caso de que estos resultados se asemejan con los teóricos, es el caso de, por ejemplo, las tablas de $N = 9$ elementos, en las que se aprecia un caso mejor de 24 OBs ($0,84 \cdot N \cdot \log_2(N)$), uno peor de 35 OBs ($\frac{N^2}{2} - \frac{N^1}{2}$) y uno medio de 29 ($N \cdot \log_2(N)$). Luego los casos medio y mejor son del orden de $O(N \cdot \log_2(N))$, y el peor del $O(N^2)$, tal y cómo esperábamos.

Gracias a los ficheros.log, se puede observar fácilmente el hecho de que MergeSort realiza menos OBs para ordenar las mismas tablas que QuickSort, pero, en cambio, este último es bastante más rápido (en cuanto a tiempo se refiere), que es lo que realmente interesa al final.