Análisis de Algoritmos 2016/2017

Práctica 2

David García Fernández, Antonio Martín Masuda, Grupo 1201, Pareja 1.

Código	Gráficas	Memoria	Total

1. Introducción.

Al principio de la práctica no tuvimos que tomar muchas decisiones. Dado que teníamos el pseudocódigo de las funciones a implementar en los apuntes de teoría, no tuvimos que pensar demasiado en cuanto a la programación de ambos métodos de ordenación. Para el último apartado, sí que nos planteamos algunas cuestiones sobre cómo implementar las tres funciones medio y cómo modificar ejericio4.c y ejercicio5.c para comprobar los resultados. Al final, decidimos hacer una sola función medio que encapsule la elección de los tres posibles pivotes y decidimos hacer un solo ejercicio4.c y ejercicio5.c que llamen a MergeSort y a tres QuickSort distintos, cada uno con un picote.

2. Objetivos

2.1 Apartado 1

El objetivo en este apartado es implementar el método de ordenación MergeSort como función recursiva y comprobar que ordena correctamente ejecutando el ejercicio4.c

2.2 Apartado 2

En este apartado, se modificará el ejercicio5.c para llamar a MergeSort y se ejecutará utilizando valgrind para comprobar que no hay pérdidas de memoria y con los ficheros obtenidos se utilizará gnuplot para comparar los resultados obtenidos con los teóricos.

2.3 Apartado 3

En este apartado, el objetivo es implementar el método de ordenación QuickSort también como función recursiva y comprobar que ordena correctamente ejecutando el ejercicio4.c

2.4 Apartado 4

En este apartado, se modificará el ejercicio5.c para llamar a QuickSort y se ejecutará utilizando valgrind para comprobar que no hay errores relacionados con la memoria (aunque QuickSort no reserva memoria dinámicamente) y se utilizará gnuplot con los ficheros obtenidos para comparar los datos obtenidos con los teóricos.

2.5 Apartado 5

Finalmente, se modificará QuickSort, ejercicio4.c y ejercicio5.c para que al llamar a este método de ordenación se pueda comprobar que funciona sin importar el pivote escogido y obtener los tiempos promedio y el número de operaciones básicas de QuickSort con cada pivote y comparar los resultados obtenidos entre ellos y con los teóricos.

3. Herramientas y metodología

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

Todos los apartados se han realizado en Linux y se ha pasado valgrind para comprobar que no ha habido pérdidas de memoria. Para comprobar los tiempos de ejecución y el número de operaciones básicas ejecutadas por cada método, se ha utilizado el gnuplot.

3.1 Apartado 1

En este apartado hemos implementado el método MergeSort. La función mergesort es una función recursiva que va dividiendo las tablas en dos y cuando llega al caso base (la tabla solo tiene un elemento) llama a merge que es la que combina las tablas. Esta última devuelve el número de operaciones básicas (la comparación de claves que se realiza en el primer bucle while) que se ejecutan. Después modificamos el fichero ejercicio4.c para llamar a MergeSort y comprobar que ordenaba correctamente la tabla.

3.2 Apartado 2

En este apartado modificamos el fichero ejercicio5.c para llamar a MergeSort y ver los tiempos de ejecución. Para ver las gráficas correspondientes, se usó gnuplot. Posteriormente, en el apartado cinco de resultados y gráficas, se pueden ver las pruebas realizadas y los datos obtenidos.

3.3 Apartado 3

En este apartado se ha implementado el método QuickSort. La función quicksort es una función recursiva que llama a partir. Esta a su vez llama a medio, la cual elige como pivote la primera posición. Posteriormente, partir coloca los elementos menores que el pivote a la izquierda y los mayores a la derecha. Finalmente, QuickSort se llama recursivamente con la tabla izquierda y derecha hasta que la tabla solo tiene un elemento. Después, modificamos el fichero ejercicio4.c para llamar a QuickSort y comprobar que ordena correctamente. Una vez hecho esto, decidimos hacer un único jercicio4.c que llame tanto a MergeSort como a QuickSort. Para ello, añadimos un argumento de entrada que elija el método con el que ordenar y devuelva un mensaje de error si lo que se introduce no es “mergesort” o “quicksort”.

3.4 Apartado 4

En este apartado modificamos el fichero ejercicio5.c para llamar a QuickSort y ver los tiempos de ejecución. Las gráficas correspondientes las hemos incluido en el apartado de resultados y gráficas. Estas se hicieron mediante gnuplot.

3.5 Apartado 5

Para realizar este apartado, decidimos hacer una única función medio que reciba un argumento de entrada extra de tipo entero que decidirá el pivote mediante un switch. Si el argumento vale uno devuelve 0 cdc y cambia el pivote a la primera posición, si vale dos devuelve 0 también ya que no se han relizado comparaciones de clave y asigna el pivote a la posición intermedia; si vale tres devuelve 3 cdc (según el enunciado de la práctica) y asigna el pivote a la posición intermedia de la posición primera, intermedia y última. En cualquier otro caso se devolverá error. Tuvimos que modificar la función QuickSort que realizamos en el apartado tres añadiendo un argumento de entrada de tipo entero que se pasará a partir y este se lo pasará a medio para elegir el pivote. Sin embargo, la función genera_tiempos_ordenacion de tiempos.c recibe un puntero a función y este solo vale para funciones que tienen como argumentos un puntero a entero y dos enteros. Por tanto, tuvimos que hacer tres funciones QuickSort más, los cuales llaman al QuickSort anterior con una elección de pivote distinto. Al haber cambiado QuickSort, teníamos que añadir un nuevo argumento de entrada en ejercicio4.c que eligiese el pivote de QuickSort (1 para el primero, 2 para el medio y 3 para el valor intermedio entre el primero, el último y el medio). En caso de querer llamar a MergeSort, hay que introducir un número cualquiera. Finalmente, faltaba modificar el ejercicio5.c. En este caso, no añadimos ningún argumento más. Simplemente, al fichero pasado como argumento, concatenamos cuatro cadenas distintas, los cuales serán los distintos ficheros creados para almacenar los tiempos promedio y las operaciones básicas ejecutas de MergeSort y los tres QuickSort.

4. Código fuente

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

4.1 Apartado 1

Rutinas MergeSort y Merge:

```
int mergesort(int* tabla, int ip, int iu){
	int a, b, imedio;

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

	imedio = (ip + iu)/2;
	a = mergesort(tabla, ip, imedio);
	if(a == ERR) return ERR;
	b = mergesort(tabla, imedio + 1, iu);
	if(b == ERR) return ERR;

	return a + b + merge(tabla, ip, iu, imedio);
}

int merge(int* tabla, int ip, int iu, int imedio){
	int *taux, i, j, k, counter = 0;
	taux = (int*)calloc((iu - ip + 1), sizeof(int));
	if(!taux) return ERR;
```

```
	i = ip;
	j = imedio + 1;
	k = 0;

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



	if(i > imedio){
		while(j <= iu){
			taux[k] = tabla[j];
			j++;
			k++;
		}
	}
	else if(j > iu){
		while(i <= imedio){
			taux[k] = tabla[i];
			i++;
			k++;
		}
	}
	for(i = ip; i < iu + 1; i++){
		tabla[i] = taux[i - ip];
	}
	free(taux);
	return counter;
}
```

4.3 Apartado 3 y 4.5 Apartado 5

Hemos decidido pasar un argumento nuevo en quicksort para pasar el tipo de pivote, por lo que ambos apartados se completan en uno.

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

	int medio, counter = 0;

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

	medio = (ip + iu)/2;

	switch(type){
		case 1:
			*pos = ip;
			break;
```

```
            case 2:
                  *pos =(ip + iu)/2;
                  break;
            case 3:
                  if((tabla[iu] <= tabla[ip] && tabla[iu] >=
                        tabla[medio]) ||(tabla[iu] >= tabla[ip] &&
                        tabla[iu] <= tabla[medio]))
                        *pos = iu;
                  else if((tabla[ip] <= tabla[iu] && tabla[ip] >=
                        tabla[medio]) || (tabla[ip] >= tabla[iu] &&
                        tabla[ip] <= tabla[medio]))
                        *pos = ip;
                  else
                        *pos = medio;
                  counter = 3;
                  break;
            default:
                  return ERR;
      }
      return OK;

}
int partir(int* tabla, int ip, int iu, int* pos, int type){

      int k, i, buff, counter = 0;

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

      counter += medio(tabla, ip, iu, pos, type);

      if(counter == ERR)
            return ERR;

      k = tabla[*pos];

      buff = tabla[ip];
      tabla[ip] = tabla[*pos];
      tabla[*pos] = buff;

      *pos = ip;

      for(i = ip + 1 ; i <= iu ; i++){
            if(tabla[i] < k){
                  (*pos)++;
                  buff = tabla[i];
                  tabla[i] = tabla[*pos];
                  tabla[*pos] = buff;
            }
            counter++;
      }

      buff = tabla[ip];
      tabla[ip] = tabla[*pos];
      tabla[*pos] = buff;

      return counter;
}
```

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

	int pos, counter = 0;

	if(!tabla || ip < 0 || ip > iu || type < 1 || type > 3) return
ERR;

	if(ip == iu) return 0;

	counter = partir(tabla, ip, iu, &pos, type);
	if(counter == ERR)
		return ERR;

	if(ip < pos)
		counter += quicksort(tabla, ip, pos-1, type);

	if(pos < iu)
		counter += quicksort(tabla, pos+1, iu, type);

	return counter;
}

/*
	Las funciones siguientes encapsulan la función quicksort,
	las utilizaremos para llamar al ejercicio5.c y poder pasar como
	puntero la función quicksort con cada uno de los pivotes.
*/

int quicksortFIRST(int* tabla, int ip, int iu){
	return quicksort(tabla, ip, iu, 1);
}

int quicksortAVG(int* tabla, int ip, int iu){
	return quicksort(tabla, ip, iu, 2);
}

int quicksortSTAT(int* tabla, int ip, int iu){
	return quicksort(tabla, ip, iu, 3);
}
```

5. Resultados, Gráficas

Aquí ponéis los resultados obtenidos en cada apartado, incluyendo las posibles gráficas.

5.1 Apartado 1

Introduciendo como parámetros de entrada :

```
./ejercicio4 -tamanio 10 -metodo mergesort -tipo 0
```

Recibimos la salida:

```
1	2	3	4	5	6	7	8	9	10

Número de veces que se ejecuta la OB: 27
```

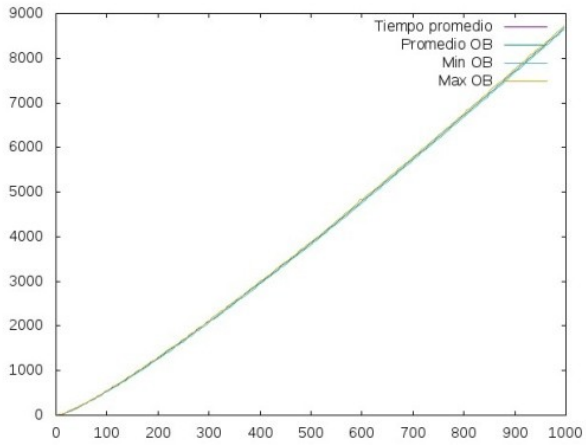
Como podemos comprobar la tabla queda perfectamente ordenada.

5.2 Apartado 2

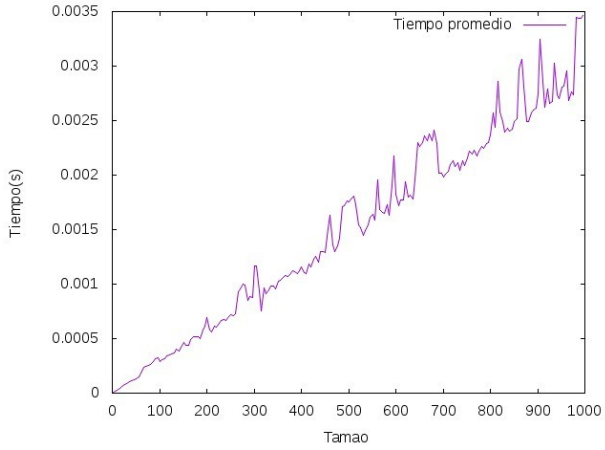
Introduciendo como parámetros de entrada:

```
./ejercicio5 -num_min 1 -num_max 1000 -incr 5 -numP 50
-fichSalida datos
```

El programa generó un extenso fichero, cuya representación gráfica podemos observar aquí:

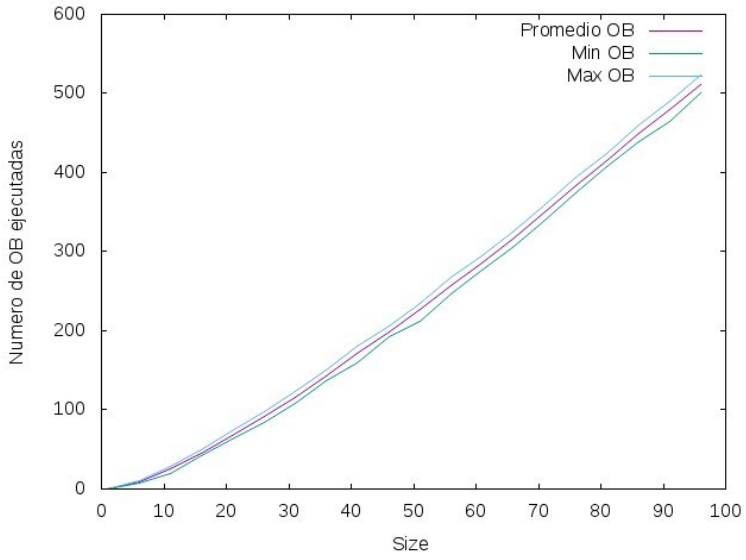


Gráfica 1: Número OB frente Tamaño



Gráfica 2: Tiempo promedio frente Tamaño

En la primera gráfica es inapreciable la diferencia entre las OB mínimas, máximas y medias ejecutadas, por lo que representaremos otra en la que num_max sea 100:



Gráfica 3: Número de OB frente Tamaño (num_max = 100)

Comprobamos que el tiempo medio de ejecución crece de una manera prácticamente lineal, si bien es cierto que en ocasiones la gráfica presenta picos que pueden ser debidos a que cierta permutación estaba "especialmente" desordenada respecto del resto o incluso a procesos internos del equipo en el que estamos ejecutando el programa que hayan podido ralentizar la ejecución.

Por otro lado tenemos la gráfica de el número de OB realizadas para cada tamaño. Comprobamos que en todo momento el número de OB medias se sitúa entre las máximas y las mínimas sin acercarse especialmente a ninguna de ellas. Haciéndose más notable la separación entre ellas cuanto mayor es la tabla. Esto se debe a que los casos peor, mejor y medio de MergeSort son muy parecidos y su crecimiento es prácticamente el mismo.

5.3 Apartado 3

Introduciendo como parámetros de entrada :

```
./ejercicio4 –tamanio 10 –metodo quicksort –tipo 1
```

Recibimos la salida:

```
1     2     3     4     5     6     7     8     9     10

Número de veces que se ejecuta la OB: 25
```

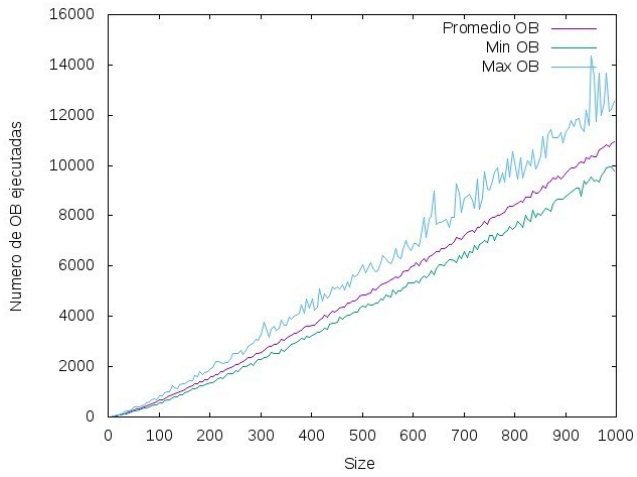
Como podemos comprobar la tabla queda perfectamente ordenada.

5.4 Apartado 4

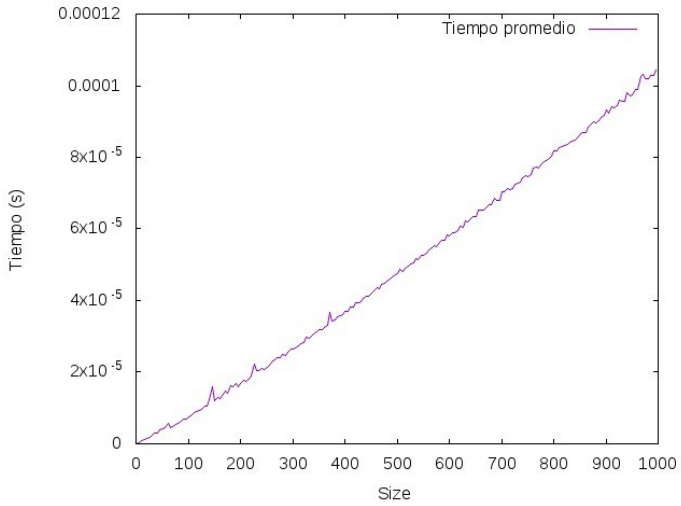
```
./ejercicio5 -num_min 1 -num_max 1000 -incr 5 -numP 50
-fichSalida datos
```

Representamos solamente los resultados obtenidos para el pivote 1.

Obtenenemos un extenso fichero, que representamos con las siguientes gráficas:



Gráfica 4: Número de OB frente Tamaño



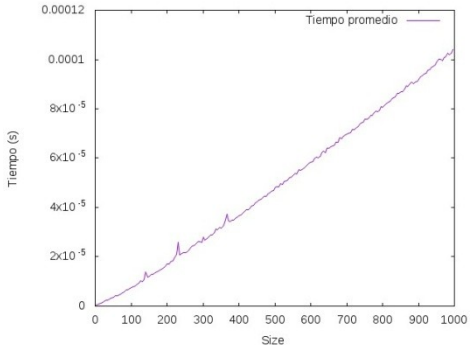
Gráfica 5: Tiempo promedio con pivote 1 frente Tamaño

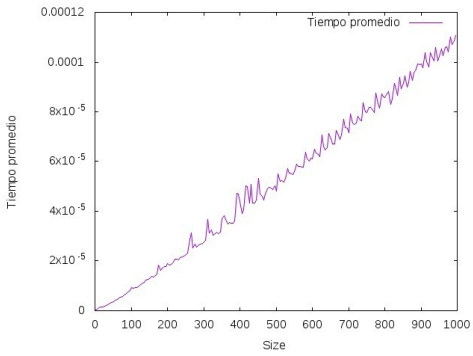
Vemos que el tiempo promedio en esta ocasión se hace más obvio que el tiempo medio crece de una forma lineal, sin apenas picos. Cabe destacar que es mucho menor que el obtenido para MergeSort, debido posiblemente al hecho de no tener que reservar memoria.

Por otro lado tenemos la gráfica del número de OB, en este caso el número de OB promedio sigue entre el máximo y el mínimo como cabía esperar, pero se acerca mucho más a las mínimas. Esto se debe a que el caso peor de QuickSort se aleja bastante del medio y el mejor. El número de OB es mucho más pequeño que las de MergeSort.

5.5 Apartado 5

Comparemos los tiempos medios obtenidos para los tres pivotes:



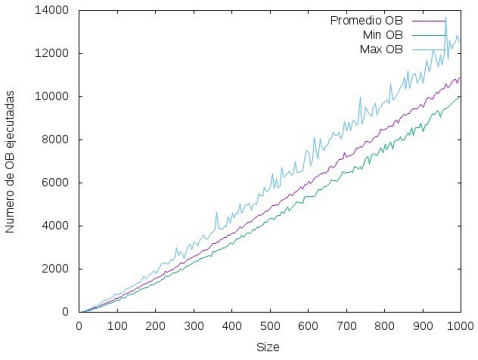


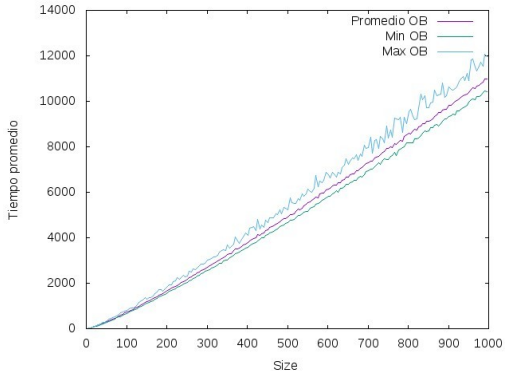
Gráfica 6: Tiempo promedio con pivote 2 frente Tamaño

Gráfica 7: Tiempo promedio con pivote 3 frente Tamaño

Observando las gráficas 5, 6 y 7 y analizando los resultados podemos concluir que el tiempo medio de ejecución es idéntico en cada uno de los pivotes, ya que al realizar tantas permutaciones para las que uno de los pivotes realice más tiempo los demás realizarán menos y viceversa. Quizás sea un poco mayor en algunos puntos de la gráfica 3 (podemos apreciar esos picos) por las CDC que realiza la función medio al elegir el pivote, pero como podemos observar no cambia significativamente el tiempo de ejecución.

Comparemos ahora las OB promediadas:





Gráfica 8: OB promedio con pivote 2 frente Tamaño

Gráfica 9: OB promedio con pivote 3 frente Tamaño

Comparando las gráficas 4, 8 y 9 vemos que resultan bastante parecidas, la diferencia significativa está en el caso peor del tercer pivote, es más pequeño ya que, aunque por cada llamada a la rutina medio tenemos 2 operaciones básicas más la elección del pivote es la mejor de las tres y por tanto el algoritmo en general se optimiza.

5. Respuesta a las preguntas teóricas.

Aquí respondéis a las preguntas teóricas que se os han planteado en la práctica.

5.1 Pregunta 1

Para MergeSort:

$$\text{Caso mejor:} \quad B_{MS} \geq \frac{N}{2} \log N + O(N)$$

$$\text{Caso medio:} \quad A_{MS} = \theta(N \log N)$$

$$\text{Caso peor:} \quad W_{MS} \leq N \log N + O(N)$$

Teóricamente el caso medio, mejor y peor de MergeSort son altamente parecidos, por lo que en la práctica no debería haber mucha diferencia entre las tres gráficas (Gráfica 3). En nuestro caso las gráficas no son nada picudas y se ajustan perfectamente a los resultados teóricos, ambos crecimientos son muy parecidos entre ellas, casi pegadas unas a otras y crecen con la misma tendencia que *N log(N)*.

Para QuickSort:

$$\text{Caso mejor: } B_{QS} = O(N \log(N))$$

$$\text{Caso medio:} \quad A_{QS} = 2N \log N + O(N)$$

$$\text{Caso peor:} \quad W_{QS} = \frac{N^2}{2} - \frac{N}{2}$$

El caso peor de QuickSort es muy malo y alejado del caso medio. Como podemos observar en la Gráfica 4. Se puede comprobar que el caso medio sigue esa tendencia parecida a la de *N log(N)* pero el caso peor presenta muchos picos y tiene un crecimiento mucho mayor. Los picos se deben a que el caso peor esté tan alejado del caso medio, en esas permutaciones cercanas al caso peor el número de OB ejecutadas se ha disparado mientras que en la que el número de OB máximas no sea tan cercano a este el crecimiento de la función es más parecido a la del caso medio.

5.2 Pregunta 2

Las gráficas 5, 6 y 7 muestran los tiempos promedios de QuickSort eligiendo como pivote la primera posición, la posición media y la posición que contiene el valor intermedio entre el primero, el último y el del medio, respectivamente. Si comparamos las tres gráficas, apenas se pueden ver diferencias apreciables en los tiempos de ejecución. Esto es debido a que las permutaciones a ordenar son aleatorias. Por lo tanto, en algunas ocasiones puede que sea más rápido ordenar la tabla cogiendo como pivote el primero, pero en otras, puede que lo sea el último o el medio. En promedio, los tiempos son prácticamente iguales. Un caso singular es el del pivote 3 “stat”. Vemos que los valores no son tan lineales y fluctúan arriba y abajo, esto puede ser debido a las cdc realizadas para escoger pivote (le hacen ser más lento en ocasiones) y el hecho de que es la elección de pivote más eficiente (le hacen ser más rápido en ocasiones).

5.3 Pregunta 3

Los casos peor, mejor y medio de QuickSort y MergeSort ya han sido especificados en la Pregunta 1.

En nuestra práctica estamos calculando el caso medio de una manera bastante eficaz, ya que ordenamos muchas permutaciones aleatorias y medimos las OB y el tiempo de ejecución promedio. Para medir el caso mejor (análogamente podríamos medir el peor) tendríamos varias opciones. Para cada tamaño de permutación podríamos coger solamente el mejor tiempo conseguido en cada caso y con eso conseguir el mejor tiempo de ordenación, al igual que podemos quedarnos solamente con el número mínimo de OB para cada tamaño. Otra opción posible sería cambiar la forma en que medimos, y en vez de pasar siempre una permutación aleatoria pasar la permutación "más ordenada", esta es la que teóricamente tiene el menor número de comparaciones de clave, para así medir empíricamente el tiempo de ejecución y comprobar si las OB utilizadas concuerdan con los resultados teóricos. Análogamente podemos hacer el caso peor.

5.4 Pregunta 4

A pesar de que en la teoría hemos visto que los casos medios de MergeSort y QuickSort son muy parecidos ($A_{MS} = \Theta(A_{QS})$), en la práctica MergeSort es del orden de 30 veces más lento MergeSort. Dado que en QuickSort, la elección del pivote no afecta prácticamente nada a los tiempos de ejecución, si comparamos la gráfica 5 (tiempo promedio de QuickSort con pivote primero) y la gráfica 2 (tiempo promedio de MergeSort) se puede ver que la escala de tiempos de MergeSort es alrededor de 30 veces mayor. Esto es debido a que en la teoría no hemos tenido en cuenta varios factores que ralentizan el método de MergeSort. El más significativo es que este método no es "in place", es decir, no trabaja sobre la propia tabla, sino que reserva memoria dinámicamente en cada llamada recursiva y al final de esta, copia la tabla en la original y la libera. Tanto la reserva de memoria como la copia de una tabla en otra son tiempos que no se han tenido en cuenta, pero que experimentalmente se ha comprobado que afectan mucho al rendimiento de MergeSort. En cambio, el método QuickSort sí es "in place" y por tanto, no necesita reservar memoria. Por lo que se concluye que, al ser QuickSort más eficiente en cuanto a gestión de memoria, su rendimiento es mucho mejor.

6. Conclusiones finales.

En esta práctica hemos utilizado las herramientas de medida de eficiencia de algoritmos que implementamos en la anterior, para poder estudiar en profundidad de una manera más empírica los algoritmos QuickSort y MergeSort. Estos algoritmos son recursivos por lo que hemos aprendido a utilizar la herramienta de la recursividad de manera práctica en esta tarea. Hemos ganado experiencia tomando decisiones de diseño que nos ha facilitado la implementación de la práctica, sobre todo en el caso del ejercicio 5 y los distintos pivotes de QuickSort. Además hemos podido ilustrar con resultados experimentales los rendimientos teóricos de los algoritmos estudiados en clase.