Análisis de Algoritmos 2015/2016

Práctica 2

Carlos Li Hu y Javier Gómez Martínez, Grupo 1201.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica vamos a experimentar con los algoritmos MergeSort y QuickSort implementándolos y, con las funciones de la práctica anterior, comprobaremos los tiempos de ejecución y las operaciones básicas que ejecutan cada una de ellas para aprender sobre su funcionamiento.

2. Objetivos

Algoritmo MergeSort

2.1 Apartado 1

Implementad en ordenar.c el algoritmo de ordenación correspondiente al método MergeSort.

2.2 Apartado 2

A continuación, modificando el programa ejercicio5.c de la práctica anterior para utilizar MergeSort, obtener la tabla correspondiente a la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Representar dichos valores y compararlos con el resultado teórico del algoritmo.

Algoritmo QuickSort

2.3 Apartado 3

Implementad la rutina int quicksort(int* tabla, int ip, int iu) para el método de ordenación QuickSort.

2.4 Apartado 4

A continuación, modificando el programa ejercicio5.c de la práctica anterior para utilizar QuickSort, obtener la tabla correspondiente a la variación del tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación. Representar dichos valores y compararlos con el resultado teórico del algoritmo.

2.5 Apartado 5

Modificar la rutina quicksort de tal forma que elimine la recursión de cola.

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.

Entorno de desarrollo: Linux

Herramientas: Terminal Linux, Netbeans,Valgrind y Gnuplot

3.1 Apartado 1

Para realizar este apartado hemos tomado como base el pseudocódigo de las transparencias y hemos corregido pequeñas erratas en él.

3.2 Apartado 2

En este apartado simplemente comprobamos en el ejercicio5.c que los valores de OBs que da mergesort tienen sentido, y poniendo muchas permutaciones comprobamos que son del orden de NlogN (N=tamaño de la permutación) como esperábamos.

3.3 Apartado 3

Para realizar este apartado hemos tomado como base el pseudocódigo de las transparencias y hemos corregido pequeñas erratas en él. Lo único resaltable que hemos tenido que hacer es averiguar que para obtener el número de OBs de partir simplemente tenemos que restar iu-ip, porque la operación básica se encuentra dentro de un for que se ejecuta siempre y por tanto no era necesario modificar el prototipo de la función.

3.4 Apartado 4

En este apartado simplemente comprobamos en el ejercicio5.c que los valores de OBs que da quicksort tienen sentido, y poniendo muchas permutaciones, comprobamos que son del orden de 2NlogN (N=tamaño de la permutación) como esperábamos. Aun así podemos ver que para valores más grandes no se cumple siempre, esto es debido en parte a que son valores enormes y para conseguir los casos peores, medios y mejores necesitaríamos N! permutaciones como mínimo teóricamente para asegurar que calculamos bien los casos con un N grande, y siendo un número tan grande, sería muy costoso para el programa ejecutarlo.

3.5 Apartado 5

Para eliminar la recursión de cola metemos un while y la operación ip=imedio+1 al final. Con ello eliminamos el número de llamadas a recursión y cuando veamos las gráficas de tiempo de ejecución, veremos que tarda menos que la versión con recursión de cola.

4. Código fuente

4.1 Apartado 1

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

    int* tabla2 = NULL;
    int i, j, k, OB;
    i = (iu - ip + 1);
    tabla2 = (int*) malloc(i * sizeof (int));
    if (!tabla2)
        return ERR;
```

```
    for (OB = 0, i = ip, j = imedio + 1, k = 0; i <= imedio && j <=
iu; k++, OB++) {
        if (tabla[i] < tabla[j]) {
            tabla2[k] = tabla[i];
            i++;
        } else {
            tabla2[k] = tabla[j];
            j++;
        }
    }
    if (i > imedio)
        for (; j <= iu; j++, k++)
            tabla2[k] = tabla[j];
    else if (j > iu)
        for (; i <= imedio; i++, k++)
            tabla2[k] = tabla[i];
    for (i = 0, j = ip; i < (iu - ip + 1) && j <= iu; j++, i++)
        tabla[j] = tabla2[i];
    free(tabla2);
    return OB;
}

int mergesort(int* tabla, int ip, int iu) {
    int imedio, OB, OB2;
    OB = 0;
    OB2 = 0;
    if (ip > iu)
        return ERR;
    if (ip == iu)
        return 0;
    else {
        imedio = (ip + iu) / 2;
        OB = mergesort(tabla, ip, imedio);
        if (OB == ERR)return ERR;
        OB2 = mergesort(tabla, imedio + 1, iu);
        if (OB2 == ERR)return ERR;
        OB += OB2;
        OB2 = merge(tabla, ip, iu, imedio);
        if (OB2 == ERR)return ERR;
        return (OB + OB2);
    }
```

4.3 Apartado 3

```
void swap(int* a, int* b) {
    int aux;
    aux = *a;
    *a = *b;
    *b = aux;
}

int medio(int *tabla, int ip, int iu) {
    return ip;
}

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

```
    int imedio, k, i;
    imedio = medio(tabla, ip, iu);
    k = tabla[imedio];
    swap(&tabla[ip], &tabla[imedio]);

    imedio = ip;
    for (i = ip + 1; i <= iu; i++) {
        if (tabla[i] < k) {
            imedio++;

            swap(&tabla[i], &tabla[imedio]);
        }
    }
    swap(&tabla[ip], &tabla[imedio]);
    return imedio;


}

int quicksort(int* tabla, int ip, int iu) {
    int imedio, OB1, OB2, OB3;
    if (ip > iu) return ERR;
    if (ip == iu) return 0;
    else {
        imedio = partir(tabla, ip, iu);
        OB1 = iu - ip;
        if (ip < (imedio - 1)) {
            OB2 = quicksort(tabla, ip, imedio - 1);
            if (OB2 == ERR)return ERR;
            OB1 += OB2;
        }
        if ((imedio + 1) < iu) {
            OB3 = quicksort(tabla, imedio + 1, iu);
            if (OB3 == ERR)return ERR;
            OB1 += OB3;
        }
    }
    return (OB1);
}
```

4.5 Apartado 5

```
int quicksort_src(int* tabla, int ip, int iu) {
    int imedio, OB1, OB2;

    OB1 = 0;
    if (ip > iu) return ERR;
    if (ip == iu) return 0;
    while (iu > ip) {
        imedio = partir(tabla, ip, iu);
        OB1 += iu - ip;
        if (ip < (imedio - 1)) {
            OB2 = quicksort_src(tabla, ip, imedio - 1);
            if (OB2 == ERR)return ERR;
            OB1+=OB2;
        }
            ip = imedio + 1;
```

```
    }
    return (OB1);
}
```

5. Resultados, Gráficas

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

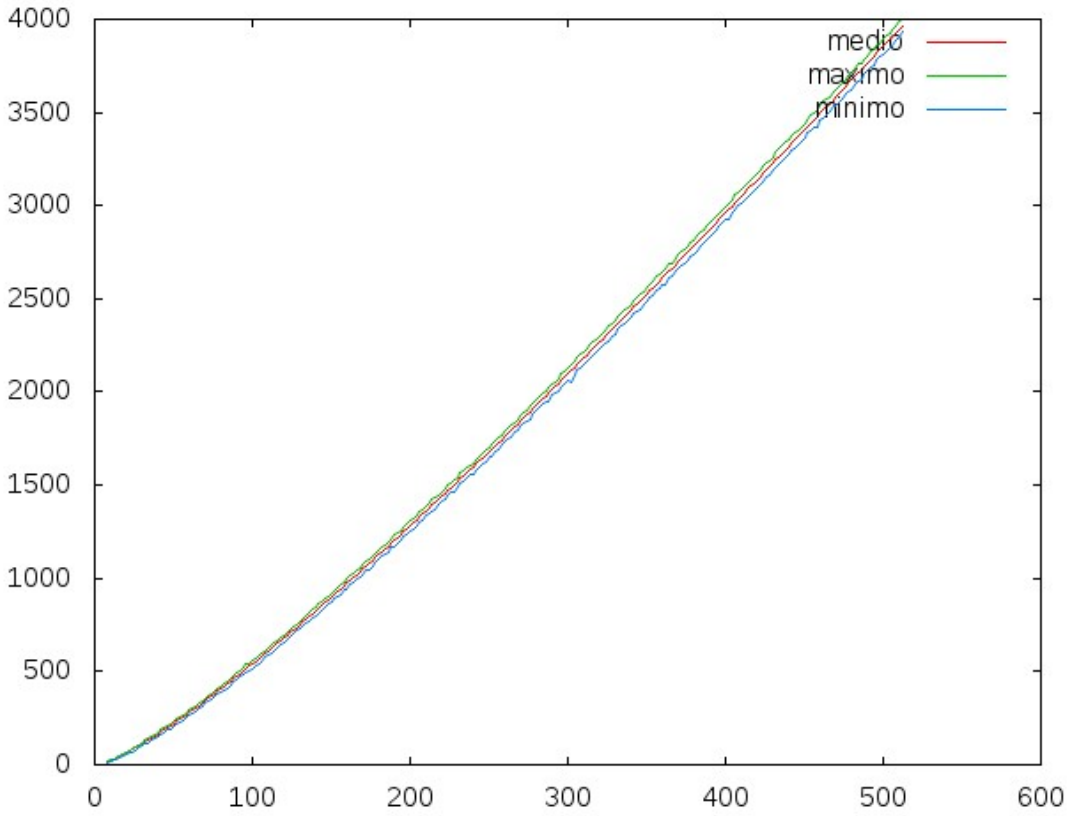
5.1 Apartado 1

Aquí comprobamos que el algoritmo mergesort funciona y nos ordena las permutaciones en el ejercicio4.c

5.2 Apartado 2

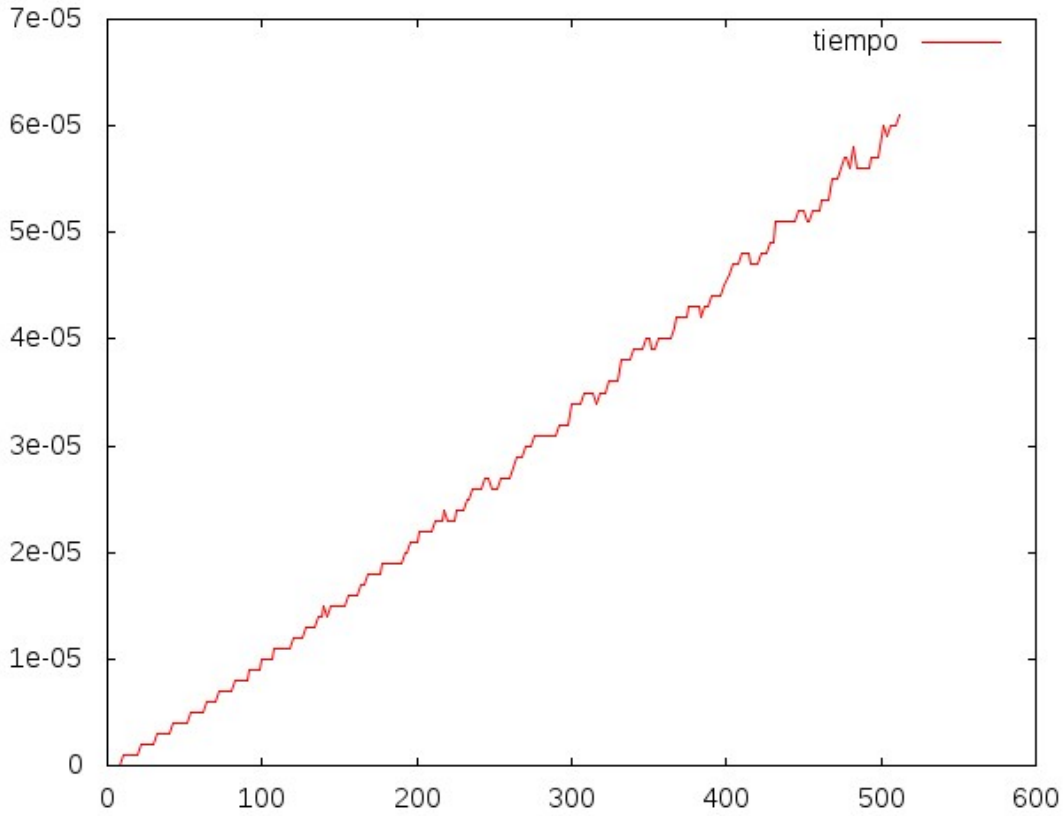
Resultados del apartado 2.

Gráfica comparando los tiempos mejor peor y medio en OBs para MergeSort, comentarios a la gráfica.



Como vemos la gráfica es del orden de NlogN con los valores medio, máximo y mínimo tomando valores muy cercanos al principio y se van dispersando según aumenta el tamaño de la permutación, pero aun así podemos ver como los valores siguen esa tendencia.

Gráfica con el tiempo medio de reloj para MergeSort, comentarios a la gráfica.



Podemos notar como en tiempo medio de ejecución la gráfica empieza regular pero empieza a fluctuar de forma muy irregular sin salirse de la tendencia que sigue, esto se debe a que con tamaños más grandes necesitamos más permutaciones para asegurar que el tiempo medio es más exacto.

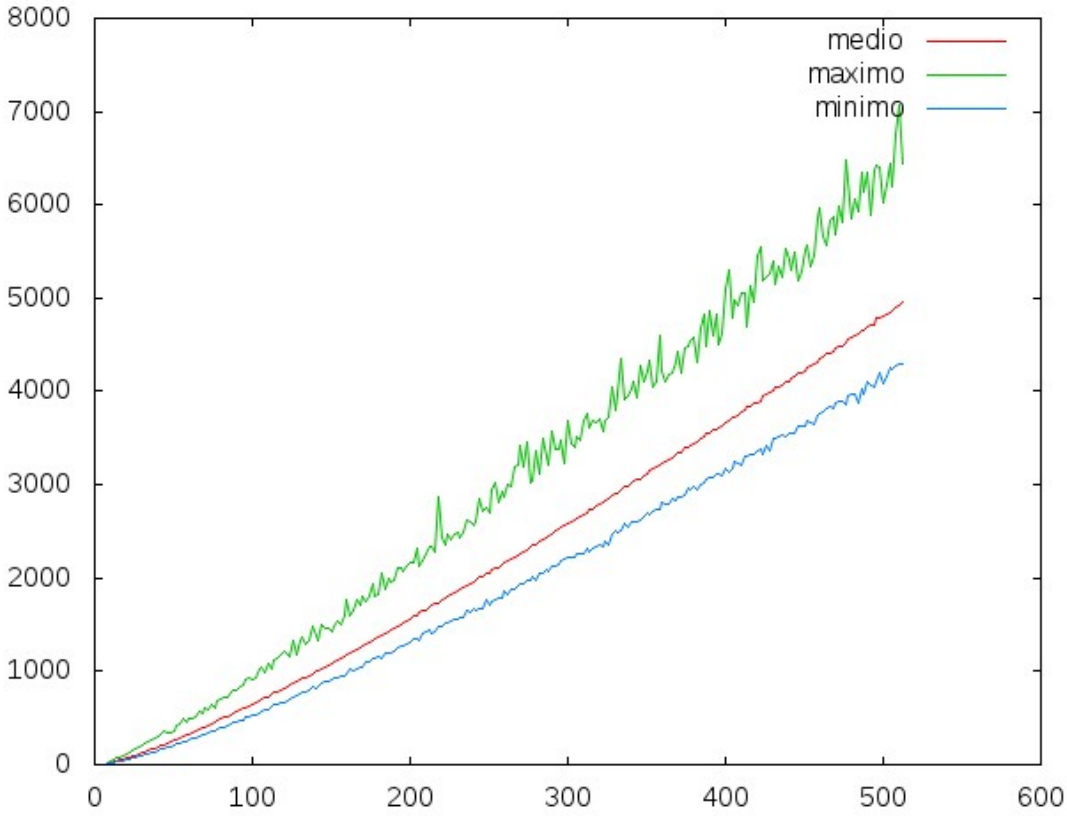
5.3 Apartado 3

Aquí probamos quicksort en el ejercicio4.c y comprobamos que ordena correctamente las permutaciones.

5.4 Apartado 4

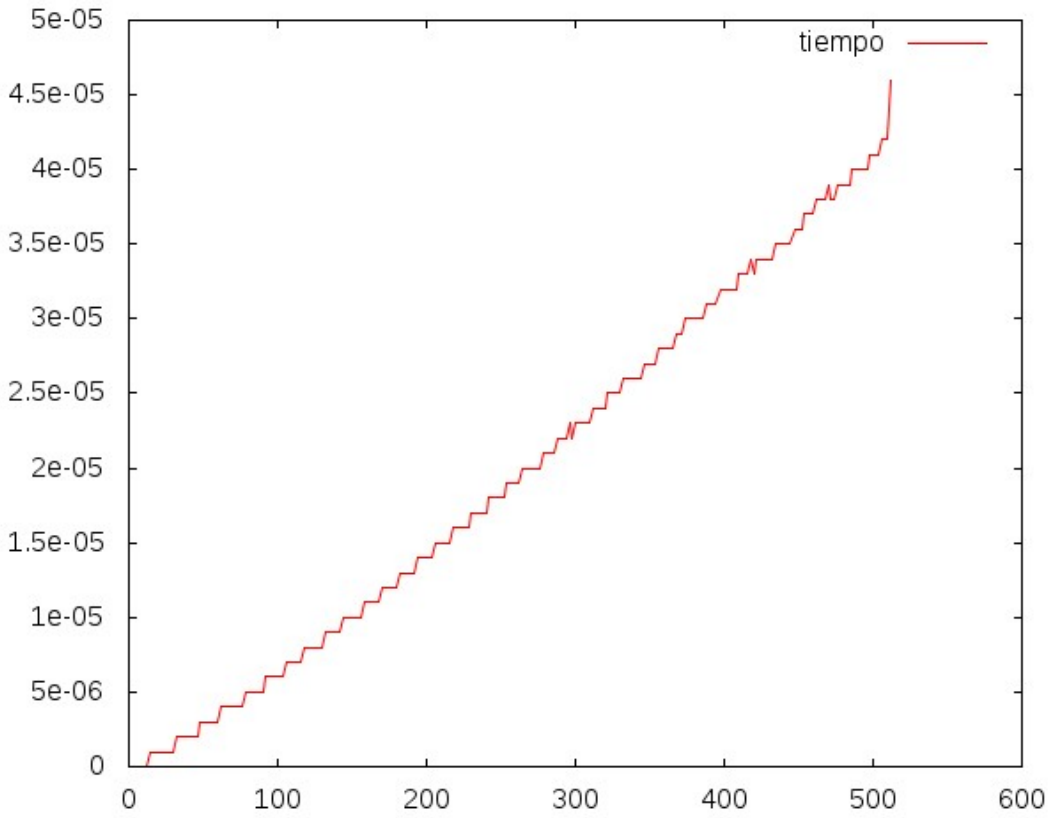
Resultados del apartado 4.

Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.



Como podemos notar en quicksort los OBs toman valores más dispares debido a que el caso mejor es mejor que el de mergesort, pero el caso peor también es peor que mergesort, sólo siendo parecido en el caso medio

Gráfica con el tiempo medio de reloj para QuickSort, comentarios a la gráfica.

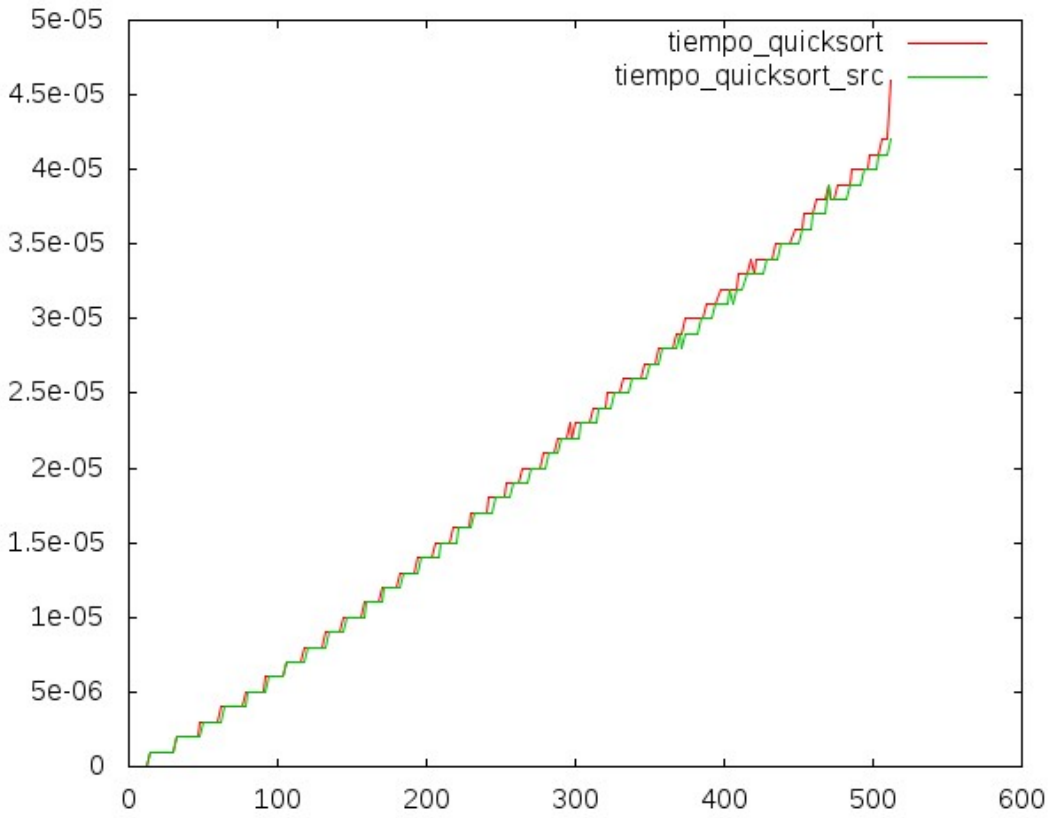


Podemos resaltar que la gráfica es parecida a la de mergesort, pero los tiempos que toma son en general menores, debido a que su caso medio es ligeramente mejor que el de mergesort.

5.5 Apartado 5

Resultados del apartado 5.

Gráfica con el tiempo medio de reloj para QuickSort y QuickSort_src, comentarios a la gráfica.



Podemos notar que quicksort sin recursividad de cola tiene un tiempo de ejecución medio menor que el de quicksort normal, esto debe de ser porque la recursión en sí cuesta un poco más de tiempo que si lo hiciéramos con recursión. Además, cada llamada recursiva hace un control de errores, por tanto, al eliminar la recursión de cola, todos esos controles de errores no se realizan en cada iteración del bucle. Al ser simplemente unos if's, tienen un impacto mínimo en el tiempo medio a menos que la permutación sea bastante grande, tal y como podemos corroborar mirando la gráfica.

6. Respuesta a las preguntas teóricas.

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

6.1 Pregunta 1

$Ams(N)=\Theta(NlogN)$

$Aqs(N)=2NlogN+O(N)$

Podemos comprobar, si tomamos valores de la gráfica de mergesort o de quicksort con OBs, que toman valores en el orden de NlogN, pero no son exactamente ese valor pues son valores asintóticos y difieren un poco por un factor O(N).

Las trazas de algunas gráficas son picudas sobre todo en quicksort debido a la aleatoridad de las permutaciones, y lo que dependen de ellas. Otro factor que influye es el número de permutaciones realizadas pues con más permutaciones sería menos picuda,

pero para abarcar la equiprobabilidad de la generación de permutaciones habría que crear como mínimo N! permutaciones, pero al ser un número tan grande al programa le costaría mucho ejecutarlo.

6.2 Pregunta 2

Claramente, quicksort_src es mejor para valores más grandes de N, pues la gráfica de tiempo es menor que la de quicksort, esto debe de ser debido a que usar recursión en sí debe de costar más tiempo que si procuramos evitarla, además del tiempo que la función se ahorra en los controles de errores que hemos comentado antes.

6.3 Pregunta 3

$Wms(N)=NlogN -N+1$

Esto se da cuando los elementos están desordenados de una manera que si dividimos la tabla entre 2 para ordenar la tabla haya que ir copiando los elementos de forma que intentemos que quede siempre un elemento en cada lado el máximo tiempo posible en cada iteración de la recursión, así hacemos que el número de OBs se maximice.

$Bms(N)=(NlogN)/2$

Este caso se da cuando la tabla está ya ordenada.

$Wqs(N)=N(N-1)/2$

si los elementos están ordenados [1,…,N] y el pivote es el primer elemento

$Bqs(N)=2logN+O(1)$

Este caso se da si al dividir las tablas con partir el pivote siempre queda en el medio, es decir, que en la primera posición siempre esté el valor medio de la subtabla correspondiente.

6.4 Pregunta 4

Por una parte, quicksort tiene varias desventajas, como que sus casos peor y mejor están más dispersos del Θ (NlogN) del mergesort y que depende de la aleatoridad de la permutación.

Pero mergesort tiene la desventajas de que para que su ejecución sea mejor, la tabla debe está parcialmente ordenada y que necesita usar memoria auxiliar, por tanto es peor en uso de memoria, pues tenemos que crear tablas de N elementos y si N es un valor muy grande es posible que nos consuma toda la memoria.

Por eso quicksort es mejor para nuestros casos por su mejor tiempo de ejecución y su mejor uso en permutaciones aleatorias equiprobables, ya que hay más probabilidades de que salga una permutación sin ordenar que ordenada.

7. Conclusiones finales.

Mientras programábamos los algoritmos hemos aprendido mucho sobre su funcionamiento general. También nos ha ayudado a tener una idea más clara del trabajo que realiza cada uno de ellos.