Análisis de Algoritmos 2017/2018

Práctica 2

David Cabornero Pascual, Alejandro Santorum Varela

Grupo 2101

Código	Gráficas	Memoria	Total

1. Introducción.

Esta práctica se basa en tres partes: elaborar el aloritmo MergeSort y calcular los tiempos de ejecución con tiempos.c, elaborar el algoritmo QuickSort y calcular los tiempos de ejecución como con MergeSort, y finalmente se modificará QuickSort con diversas funciones que decidirán el criterio para escoger el pivote y, de nuevo, se calcularán los tiempos de ejecución con el mismo método.

2. Objetivos

2.1 Apartado 1

Inicialmente, necesitamos crear el algoritmo de ordenación recursivo MergeSort, al igual la función Merge. En MergeSort, debemos dividir recursivamente la lista de números por la mitad hasta llegar a unidades indivisibles (listas de un elemento). A partir de ahí, iremos utilizando la función merge para ir creando listas ordenadas combinando dos de las listas divididas, recuperando así nuestras listas divididas, pero ahora ordenadas.

2.2 Apartado 2

Este apartado consta únicamente de obtener la tabla de tiempos obtenida en el apartado 5 de la primera práctica, solamente que el algoritmo de ordenación que analizará ahora tiempos.c será MergeSort.

2.3 Apartado 3

Ahora, realizaremos el algoritmo de ordenación recursivo QuickSort, en él realizaremos la función partir, que partirá la tabla por cierto punto y dejará todos los números menores que él detrás suyo, mientras que todos los mayores quedarán delante suyo, aunque esos dos conjuntos seguirán estando desordenados. Por ello, deberemos ordenar recursivamente las dos tablas desordenadas, llamando a QuickSort otra vez para cada una de las subtablas.

Para el cálculo del pivote utilizaremos la función medio, que en este caso será el primer elemento de la tabla.

2.4 Apartado 4

En este apartado deberemos hacer lo análogo al 2 para QuickSort, es decir, modificar el programa del ejercicio 5 de la primera práctica para que tiempos.c analice el nuevo algoritmo de ordenación y calcule los tiempos de ejecución.

2.5 Apartado 5

En el apartado 3 hemos utilizado el primer elemento de la tabla como pivote. Ahora, se nos pide implementar dos nuevas funciones: medio_avg nos dará como pivote el punto medio de la tabla que le pasemos, sin embargo sigue siendo igual de ineficiente que la primera función.

La segunda función será más eficiente: ahora, tendremos el primer elemento de la tabla, el último y el punto medio. Veremos el valor de cada uno de ellos, y nos

quedaremos con el valor que no sea ni el mayor ni el menor. De esta forma, devolveremos la posición del valor mencionado.

La función partir se verá cambiada por la función pivote que queramos utilizar manualmente. Además, se nos pide calcular de nuevo los tiempos de ejecución.

3. Herramientas y metodología

Hemos utilizado, al igual que en la práctica anterior, la página web https://c9.io para programar, utilizando la terminal de Linux para ejecutar los programas realizados y realizamos las comprobaciones de reserva y liberación de memoria con Valgrind.

Por otro lado, para la representación de gráficas hemos utilizado el programa recomendado por la asignatura, GNUPLOT, comentar que es una pena que la documentación se tenga que entregar impresa, ya que gnuplot te permite poner cada línea de la gráfica de un color para hacerlo todo más visible. Aún así se ha intentado guiar al lector con indicaciones sobre que corresponde cada línea en cada gráfica, haciendo todo lo más trivial posible.

3.1 Apartado 1

Para realizar MergeSort, necesitamos la tabla de elementos desordenados y los índices del primer y último elemento de la tabla. Deberemos comprobar por lo tanto que la tabla no está vacía, que el índice del primer elemento de la tabla es menor que el índice del último y que ambos sean positivos.

Como hemos explicado antes, MergeSort es recursivo, por lo que necesitamos un caso base. En este caso será cuando las tablas partidas dadas sean de longitud 1. Es decir, pararemos la recursión cuando el primer y último índice de la tabla sean los mismos.

Ahora, calcularemos el punto medio, resultado de aplicar la función suelo sobre la media del mayor y menor índice. Ahora, haremos la recursión, llamando a la función MergeSort, dando como primer argumento la tabla, pero ahora haremos dos funciones MergeSort en las que se trabajarán las dos mitades de la tabla por separado. Así, se dividirán las tablas hasta ser indivisibles. Cuando termine la recursión, se irá aplicando la función merge sobre tablas ordenadas, combinando así dos subtablas ordenadas para formar una sola subtabla ordenada.

Ahora describiremos la función auxiliar de mergesort, merge. Esta función la podemos describir como una función capaz de ordenar una tabla cuyas dos mitades están ordenadas entre ellas. Le pasaremos a merge por tanto cuatro argumentos: la tabla dada, el primer índice de la tabla, el último y índice final de la primera tabla ordenada. Las comprobaciones son obvias: la tabla no puede ser NULL y el primer, medio y último índice deben estar ordenados de menor a mayor.

Creamos una tabla auxiliar que va a tener capacidad para albergar los elementos que se encuentren entre el primer y último índice. En nuestro primer bucle, compararemos los inicios de las dos mitades de las tablas, donde introduciremos en nuestra tabla auxiliar el menor elemento de los dos iniciales. Compararemos el siguiente

elemento de la tabla con el elemento introducido con el elemento no introducido. Continuaremos comparando con el siguiente elemento al introducido y el no introducido hasta terminar una de las dos tablas, ahí terminará nuestro primer bucle. Como el argumento a devolver es el número de comparaciones de clave, necesitamos contarlas, y contaremos una cada vez que terminemos una iteración de este bucle.

Una vez terminada una de las dos tablas, el resto de elementos no introducidos de la otra tabla están ordenados, así que nuestro if/else realiza exactamente esta acción, terminar de introducir los elementos ordenados en la tabla auxiliar.

Por último, en nuestro bucle for introducimos los datos ordenados en su lugar correspondiente de la tabla, es decir el elemento 0 de la tabla auxiliar corresponderá con el elemento de índice iu.

3.2 Apartado 2

Esta parte no tiene más misterio que escribir una línea en ejercicio5.c, ya que la tabla correpondiente a tiempos de ejecución ya nos la da tiempos.c, programa que ya realizamos en la práctica anterior. Para ello, en el ejercicio5.c, llamaremos a genera_tiempos_ordenacion.

3.3 Apartado 3

A la función de ordenación quicksort empezaremos pasándole una tabla de elemento, donde especificaremos cuál es su primer índice y el último (es necesario especificar el primero, ya que esta función es recursiva). Las comprobaciones son simples: la tabla no puede ser NULL, y el índice primero tiene que ser menor que el último, además de que ambos deben ser positivos. Acto seguido, al ser una función recursiva, deberemos poner nuestro caso base: la tabla de longitud 1. Después le pasaremos nuestros argumentos a la función partir, además de la dirección de un entero, que nos devolverá un pivote. La función partir dejará todos los elementos menores al elemento del pivote a su izquierda, y todos los elementos mayores a su derecha. Después de esto, aplicaremos recursivamente quicksort sobre las dos subtablas desordenadas separadas por el pivote. Finalmente, devolveremos las comparaciones de clave generadas por partir y por los dos quicksort recursivos.

Ahora explicaremos la función partir, que tiene como objetivo dejar a la derecha del pivote todos los elementos mayores que el elemento del pivote y a la izquierda todos los elementos menores que el elemento del pivote. Recibimos como argumentos la tabla, el primer y último índice y un puntero a un entero, que será nuestro pivote. Lo primero son las comprobaciones: la tabla no puede ser NULL, y el índice primero tiene que ser menor que el último, además de que ambos deben ser positivos. Ahora, calcularemos el pivote, quedándonos con el número de cdc realizadas, la función medio utlilizada para calcular el pivote será explicada más adelante.

Primero, nos guardaremos el elemento pivote y realizaremos nuestro primero swap, entre el primer elemento y el elemento pivote. Después comenzaremos un bucle, en el que comenzaremos con un entero *mid* igualado al primero elemento y un entero *i*, encargado de aumentar según se lleven acabo las iteraciones del bucle. Así pues, se comparará siempre (comparación de clave) el elemento de índice *i* con el pivote, y

cuando así sea se aumentará la posición de *mid* en 1 y se realizará un swap entre el elemento de índice *mid* y el de índice *i*, llevando así acabo el algoritmo. En cuanto acabe el bucle, se realizará un último swap entre el primer elemento (el pivote, pues no se ha movido desde el comienzo del algoritmo) y la posición original del pivote, que coincide con *mid*. Devolveremos el número de comparaciones de clave realizadas.

La función medio es simple, ya que el pivote que nos piden es el primer elemento. Aunque puede ser redundante porque las comprobaciones que realizamos son las mismas que realizamos en merge (y la función medio siempre es llamada desde merge) hemos preferido asegurarnos y volver a hacer las comprobaciones, ya que así no tenemos que preocuparnos de cambiar esa función en caso de que modifiquemos este código en una práctica posterior o volvamos a usar esta función en algún momento.

El implementar en ejercicio4.c este ejercicio es simple, ya que simplemente tenemos que sustituir en la línea 53 el algoritmo de ordenación que esté por el que estemos utlizando.

3.4 Apartado 4

El código a cambiar en el ejercicio5.c es completamente análogo a lo realizado en el apartado 2. Sustituiremos simplemente el primer argumento de genera_tiempos_ordenacion por el algoritmo de ordenación quicksort.

3.5 Apartado 5

En primer lugar: como no se nos ha dicho que tengamos que incluir más argumentos a las funciones, no incluiremos un puntero a función que permita incluir como argumento la variante de la función medio que queramos utilizar. Esto quiere decir que para cambiar el tipo de función medio deberemos cambiar directamente el código de la función partir.

También repetimos que volveremos a hacer las comprobaciones redundantes que hicimos en el primer medio por la misma razón que especificamos anteriormente. En medio_avg tenemos que devolver la el elemento medio de la tabla dada. Es bastante simple, la única especificación de decisión de código que merece mención es que nos hemos decantado por utilizar la función suelo cuando la media salga decimal.

En medio_stat es en el único algoritmo medio donde realizamos comparaciones de clave (en los anteriores devolvíamos 0 cdc realizadas). Calculamos de nuevo la media con la función suelo, y ahora realizamos las comparaciones de clave necesarias para encontrar el término medio. Dependiendo de la situación, el número de comparaciones de clave necesarias para encontrar el término medio puede ser 2 o 3, por lo que devolveremos un número u otro en función del número de cdc necesarios para cada caso.

4. Código fuente

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

4.1 Apartado 1

```
/*----------------------------MERGESORT----------------------*/
/*
Creadores: Alejandro Santorum y David Cabornero
Fecha: 25/09/2017
Pareja. 10
Función: mergesort
    Recive:
        -tabla: array de enteros
        -ip: índice del primer elemento de la tabla
        -iu: índice del último elemento de la tabla
    Devuelve:
        -int: número de comparaciones de clave o ERR en
            caso de error.
*/
int mergesort(int* tabla, int ip, int iu){
    int M, a=0, b=0, c=0;
    double aux;

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida.\n");
        return ERR;
    }

    if(ip > iu){
        printf("Error. Último menor que primero.\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*---------------------------------*/

    /*Comprobación de parada de la recursión*/
    if(ip == iu){
        return 0;
    }
    /*---------------------------------*/

    aux = (iu + ip)/2;
    M = floor(aux);

    a = mergesort(tabla, ip, M);
    b = mergesort(tabla, M+1, iu);

    c = merge(tabla, ip, iu, M);

    return (a+b+c);
}
```

```c
/*Función auxiliar de mergesort*/
int merge(int *tabla, int ip, int iu, int M){
    int *tablaAux=NULL;
    int tam, i, j, k, ncdc=0;

    /*Comprobaciones*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida II (merge).\n");
        return ERR;
    }
    if(iu<ip || ip > M || iu < M){
        printf("Error en los índices en la función de combinar.\n");
        return ERR;
    }
    /*-------------------------------------*/

    tam = (iu - ip)+1;

    tablaAux = (int *) malloc(tam * sizeof(int));
    if(tablaAux == NULL){
        printf("Error. Problemas de reserva de memoria tabla auxiliar.\n");
        return ERR;
    }

    i=ip;
    j=M+1;
    k=0;

    while(i<= M && j<=iu){
        if(tabla[i]<tabla[j]){
            tablaAux[k] = tabla[i];
            i++;
        }
        else{
            tablaAux[k]=tabla[j];
            j++;
        }
        k++;
        ncdc++;
    }

    if(i>M){
        while(j<=iu){
            tablaAux[k] = tabla[j];
            j++;
            k++;
        }
    }
    else if(j>iu){
        while(i<=M){
            tablaAux[k] = tabla[i];
            i++;
            k++;
        }
    }

    for(i=0, j=ip; j<=iu ;i++, j++){
        tabla[j] = tablaAux[i];
    }

    free(tablaAux);

    return ncdc;
}
```

4.3 Apartado 3

```c
/*----------------------------QUICKSORT-----------------------------*/
/*
Creadores: Alejandro Santorum y David Cabornero
Fecha: 25/09/2017
Pareja. 10
Función: quicksort
    Recive:
        -tabla: array de enteros
        -ip: índice del primer elemento de la tabla
        -iu: índice del último elemento de la tabla
    Devuelve:
        -int: número de comparaciones de clave o ERR en
            caso de error.
*/
int quicksort(int *tabla, int ip, int iu){
    int a=0, b=0, c=0;
    int pos;

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida (qs).\n");
        return ERR;
    }
    if(ip > iu){
        printf("Error. Primero mayor que último (qs).\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*--------------------------*/

    if(ip == iu){
        return 0;
    }

    c = partir(tabla, ip, iu, &pos);

    if(ip < pos-1){
        a = quicksort(tabla, ip, pos-1);
        if(a == ERR){
            printf("Error en QS parte recursiva (a).\n");
            return ERR;
        }
    }
    if(pos+1 < iu){
        b = quicksort(tabla, pos+1, iu);
        if(b == ERR){
            printf("Error en QS parte recursiva (b).\n");
            return ERR;
        }
    }


    return (a+b+c);
}
```

```c
/*Función auxiliar de quicksort*/
int partir(int *tabla, int ip, int iu,int *pos){
    int k, i, ncdc=0, aux, mid;

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida (partir) o de pos.\n");
        return ERR;
    }
    if(ip > iu){
        printf("Error. Primero mayor que último (partir).\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*--------------------------*/

    /* IMPORTANTE: Se recomienda sustituir aqui debajo la función
    que calcula el pivote para poder comparar la eficacia del algoritmo
    dependiendo de la correcta elección del pivote.*/
    ncdc = medio(tabla, ip, iu, pos);
    if(ncdc == ERR){
      printf("Error en el retorno de la función medio.\n");
      return ERR;
    }
    mid = *pos;

    k = tabla[mid];

    aux = tabla[mid];
    tabla[mid] = tabla[ip];
    tabla[ip] = aux;

    mid = ip;

    for(i=ip+1; i<=iu; i++){
        ncdc++;
        if(tabla[i] < k){
            mid++;
            aux = tabla[i];
            tabla[i] = tabla[mid];
            tabla[mid] = aux;
        }
    }

    aux = tabla[ip];
    tabla[ip] = tabla[mid];
    tabla[mid] = aux;

    *pos = mid;

    return ncdc;
}

/* --------------Funciónes axuliares de partir--------------------------*/
int medio(int *tabla, int ip, int iu, int *pos){
    /*A pesar de que las mismas comprobaciones se han realizado en la función anterior,
    nos cercionamos de que el paso de argumentos entre funciones es correcto y no
    hay ninguna anomalía*/

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida (partir) o de pos.\n");
        return ERR;
    }
    if(ip > iu){
        printf("Error. Primero mayor que último (partir).\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*--------------------------*/

    *pos = ip;

    return 0;
}
```

4.5 Apartado 5

```c
int medio_avg(int *tabla, int ip, int iu,int *pos){
    int avg;

    /*A pesar de que las mismas comprobaciones se han realizado en la función anterior,
    nos cercionamos de que el paso de argumentos entre funciones es correcto y no
    hay ninguna anomalía*/

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida (partir) o de pos.\n");
        return ERR;
    }
    if(ip > iu){
        printf("Error. Primero mayor que último (partir).\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*--------------------------*/

    avg = floor((ip+iu)/2);
    *pos = avg;

    return 0;
}
```

```c
int medio_stat(int *tabla, int ip, int iu,int *pos){
    int avg;

    /*A pesar de que las mismas comprobaciones se han realizado en la función anterior,
    nos cercionamos de que el paso de argumentos entre funciones es correcto y no
    hay ninguna anomalía*/

    /*Comprobaciones de error*/
    if(tabla == NULL){
        printf("Error. Memoria de la tabla perdida (partir) o de pos.\n");
        return ERR;
    }
    if(ip > iu){
        printf("Error. Primero mayor que último (partir).\n");
        return ERR;
    }
    if(ip<0 || iu<0){
        printf("Error. ip o iu menor que cero.\n");
        return ERR;
    }
    /*--------------------------*/

    avg = floor((ip+iu)/2);

    if(tabla[ip] < tabla[iu]){
        if(tabla[avg] < tabla[ip]){
            *pos = ip;
            return 2;
        }
        else if(tabla[avg] < tabla[iu]){
            *pos = avg;
            return 3;
        }
        else{
            *pos = iu;
            return 3;
        }
    }


    else{
        if(tabla[avg] < tabla[iu]){
            *pos = iu;
            return 2;
        }
        else if(tabla[avg] < tabla[ip]){
            *pos = avg;
            return 3;
        }
        else{
            *pos = ip;
            return 3;
        }
    }
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

Resultados del apartado 1.

```
santorum:~/workspace (master) $ valgrind ./ejercicio4 -tamanio 10
==3487== Memcheck, a memory error detector
==3487== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3487== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3487== Command: ./ejercicio4 -tamanio 10
==3487==
Practica numero 2, apartado 1
Realizada por: Alejandro Santorum & David Cabornero
Grupo: 1201
0       1       2       3       4       5       6       7       8       9
==3487==
==3487== HEAP SUMMARY:
==3487==     in use at exit: 0 bytes in 0 blocks
==3487==   total heap usage: 10 allocs, 10 frees, 176 bytes allocated
==3487==
==3487== All heap blocks were freed -- no leaks are possible
==3487==
==3487== For counts of detected and suppressed errors, rerun with: -v
==3487== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

```
santorum:~/workspace (master) $ valgrind ./ejercicio4 -tamanio 80
==3476== Memcheck, a memory error detector
==3476== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3476== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3476== Command: ./ejercicio4 -tamanio 80
==3476==
Practica numero 2, apartado 1
Realizada por: Alejandro Santorum & David Cabornero
Grupo: 1201
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
==3476==
==3476== HEAP SUMMARY:
==3476==     in use at exit: 0 bytes in 0 blocks
==3476==   total heap usage: 80 allocs, 80 frees, 2,368 bytes allocated
==3476==
==3476== All heap blocks were freed -- no leaks are possible
==3476==
==3476== For counts of detected and suppressed errors, rerun with: -v
==3476== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.2 Apartado 2

Resultados del apartado 2.

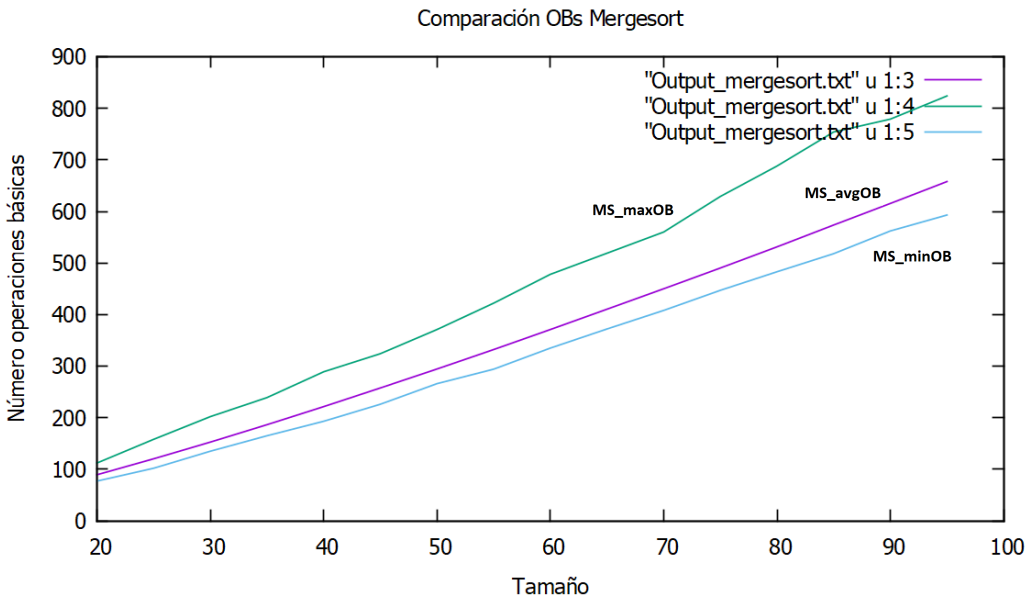
```
santorum:~/workspace (master) $ valgrind --leak-check=full ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000
==4486== Memcheck, a memory error detector
==4486== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==4486== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==4486== Command: ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000 -fichSalida Output_mergesort.txt
==4486==
Practica numero 2, apartado 2
Realizada por: Alejandro Santorum & David Cabornero
Pareja 10.
Grupo: 1201
array = 16
Salida correcta
==4486==
==4486== HEAP SUMMARY:
==4486==     in use at exit: 0 bytes in 0 blocks
==4486==   total heap usage: 1,840,018 allocs, 1,840,018 frees, 52,009,080 bytes allocated
==4486==
==4486== All heap blocks were freed -- no leaks are possible
==4486==
==4486== For counts of detected and suppressed errors, rerun with: -v
==4486== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
santorum:~/workspace (master) $
```

Output_mergesort.txt

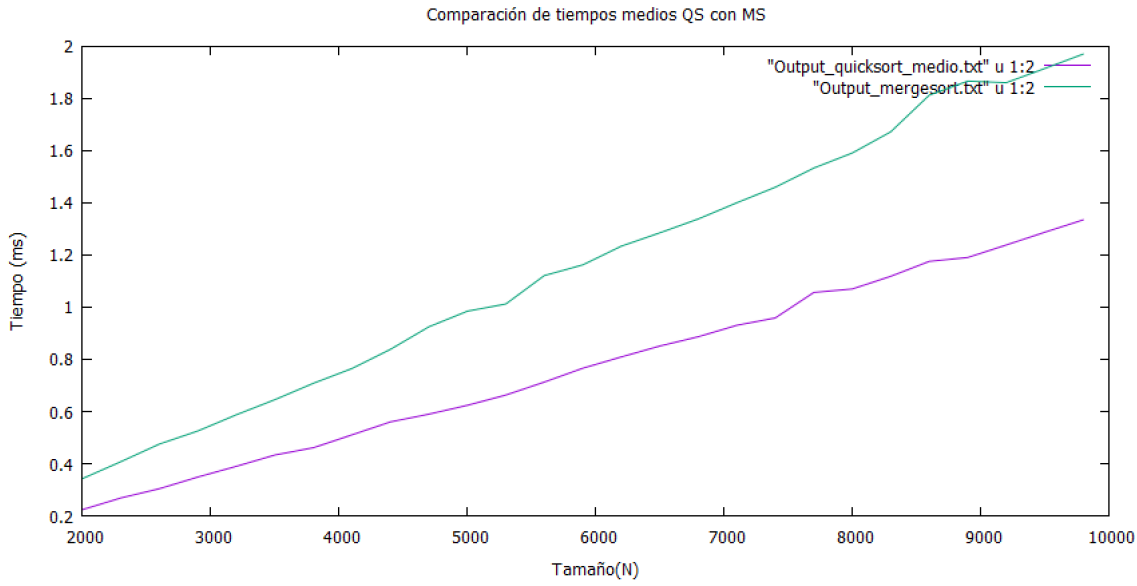
size	time(ms)	avg_ob	max_ob	min_ob.
20	0.001768	89.36	112	77
25	0.002235	120.31	158	102
30	0.002657	152.62	202	135
35	0.003110	186.40	239	165
40	0.003658	221.42	289	193
45	0.004166	257.63	324	226
50	0.004581	294.32	371	266
55	0.005000	332.14	422	294
60	0.005495	370.86	478	335
65	0.005910	410.33	519	372
70	0.006446	450.00	560	408
75	0.007198	490.29	629	447
80	0.007931	531.22	688	483
85	0.008754	573.63	754	518
90	0.009122	615.48	779	562
95	0.009385	657.85	824	593

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

A continuación mostramos la gráfica que compara el número de OB´s realizadas por Mergesort en su caso mejor, peor y medio. No hay mucho que resultar, la gráfica sale según lo esperado (el caso peor es el que realiza más OB´s seguido del medio y por último el mejor). La línea de representación es bastante lineal, lo que demuestra la buena implementación tanto de la función de ordenación Mergesort como de la encargada de calcular y guardar los tiempos en el fichero de salida.



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



La gráfica mostrada también incluye el tiempo(ms) de Quicksort pero será más adelante cuando se hable de el y se compare con Mergesort.

Centrándonos únicamente en la línea de Mergesort (la de arriba si no la puede ver en color), se puede apreciar que no es una gráfica exponencial ni cuadrática, sino una parecida a xlog(x) + O(x) (más adelante se aporta una prueba interesante sobre esto) que era lo que cabía esperar.

5.3 Apartado 3

Resultados del apartado 3.

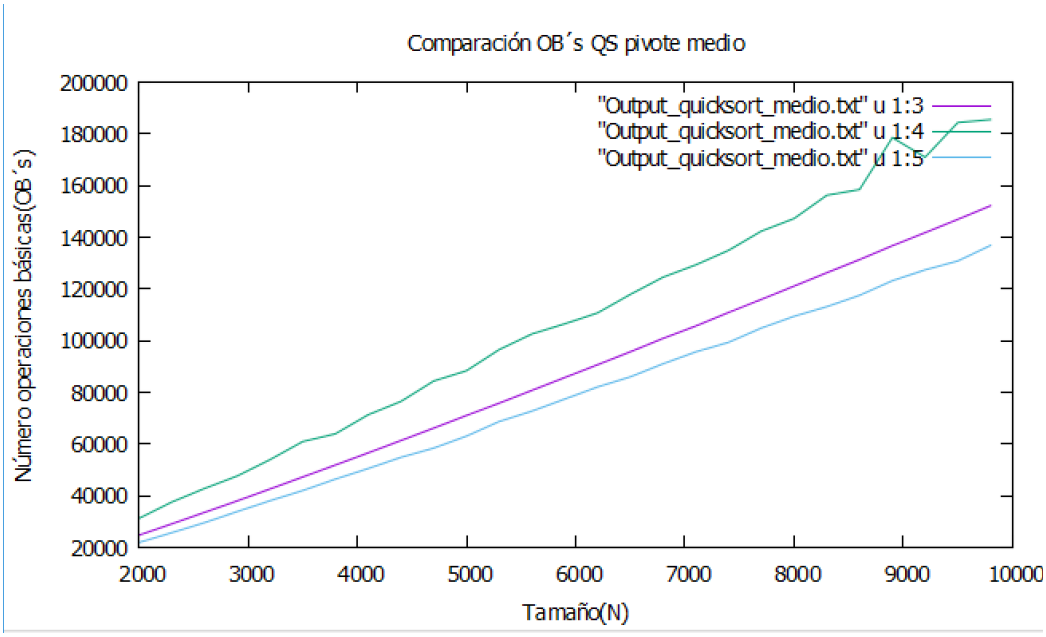
```
santorum:~/workspace (master) $ valgrind ./ejercicio4 -tamanio 50
==3510== Memcheck, a memory error detector
==3510== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==3510== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==3510== Command: ./ejercicio4 -tamanio 50
==3510==
Practica numero 2, apartado 1
Realizada por: Alejandro Santorum & David Cabornero
Grupo: 1201
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
==3510==
==3510== HEAP SUMMARY:
==3510==     in use at exit: 0 bytes in 0 blocks
==3510==   total heap usage: 35 allocs, 35 frees, 336 bytes allocated
==3510==
==3510== All heap blocks were freed -- no leaks are possible
==3510==
==3510== For counts of detected and suppressed errors, rerun with: -v
==3510== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

5.4 Apartado 4

Resultados del apartado 4.

```
santorum:~/workspace (master) $ valgrind --leak-check=full ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000
==7705== Memcheck, a memory error detector
==7705== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7705== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==7705== Command: ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000 -fichSalida Output_mergesort.txt
==7705==
Practica numero 2, apartado 2
Realizada por: Alejandro Santorum & David Cabornero
Pareja 10.
Grupo: 1201
array = 16
Salida correcta
==7705==
==7705== HEAP SUMMARY:
==7705==     in use at exit: 0 bytes in 0 blocks
==7705==   total heap usage: 32,018 allocs, 32,018 frees, 7,617,080 bytes allocated
==7705==
==7705== All heap blocks were freed -- no leaks are possible
==7705==
==7705== For counts of detected and suppressed errors, rerun with: -v
==7705== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

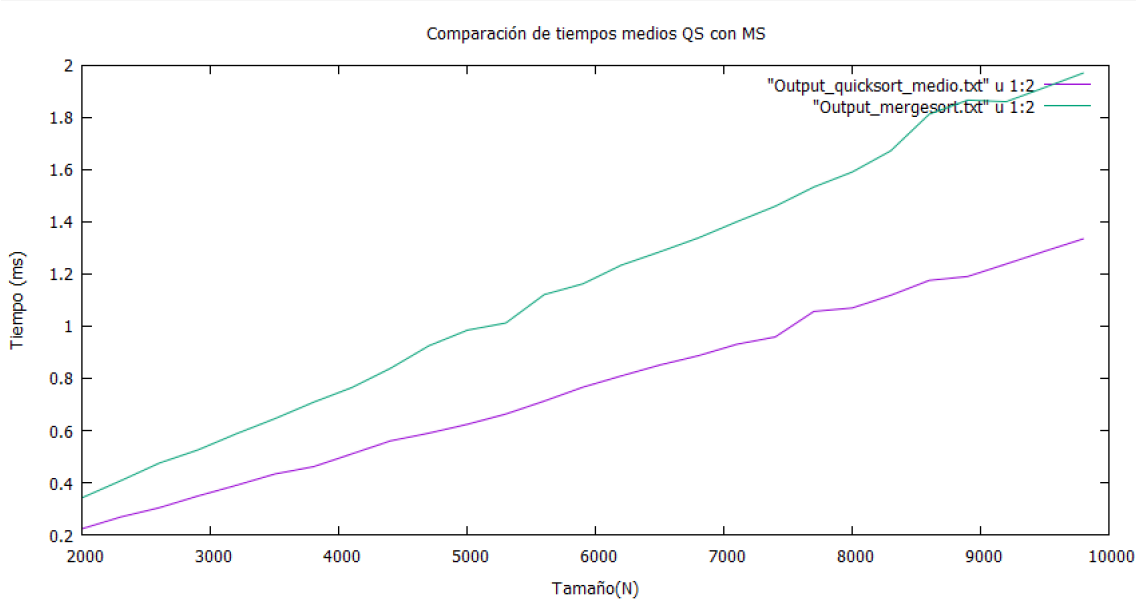
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.



Si dispone de la imagen únicamente en blanco y negro, le indico que la línea de la gráfica superior se corresponde con OBmáx, la línea inferior con OBmín y la de el medio con OBavg de la salida del algoritmo Quicksort utilizando la función de pivote “Medio”.

Se ve que para el caso medio y mejor la gráfica es suave y del tipo de xlog(x) + O(x), todo dentro de lo esperado en cuanto a rendimiento. De la misma forma se percibe que la línea superior se aleja cada vez más de las dos de abajo, esto se debe a que el caso peor de Quicksort es de N^2/2 – N/2, por lo que es una función cuadrática y que tiende a diverger de una forma mucho más potente que las otras dos.

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



Ya hemos visto esta gráfica anteriormente, pero ahora nos centraremos en la línea que representa el tiempo(ms) de Quicksort, que es la línea de abajo. De la misma forma que comentamos para Mergesort, el caso medio de Quicksort se hacerca bastante a la gráfica de la función 2xlog(x) + O(x) (se entrega una prueba interesante en la resolución de las cuestiones).

Aprovechamos para comentar que, tal como su nombre indica, Quicksort es más rápido que Mergesort tal y como podemos ver empíricamente.

5.5 Apartado 5

Resultados del apartado 5.

Salida con función medio_avg

```
santorum:~/workspace (master) $ valgrind --leak-check=full ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000
==7745== Memcheck, a memory error detector
==7745== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7745== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==7745== Command: ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000 -fichSalida Output_mergesort.txt
==7745==
Practica numero 2, apartado 2
Realizada por: Alejandro Santorum & David Cabornero
Pareja 10.
Grupo: 1201
array = 16
Salida correcta
==7745==
==7745== HEAP SUMMARY:
==7745==     in use at exit: 0 bytes in 0 blocks
==7745==   total heap usage: 32,018 allocs, 32,018 frees, 7,617,080 bytes allocated
==7745==
==7745== All heap blocks were freed -- no leaks are possible
==7745==
==7745== For counts of detected and suppressed errors, rerun with: -v
==7745== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

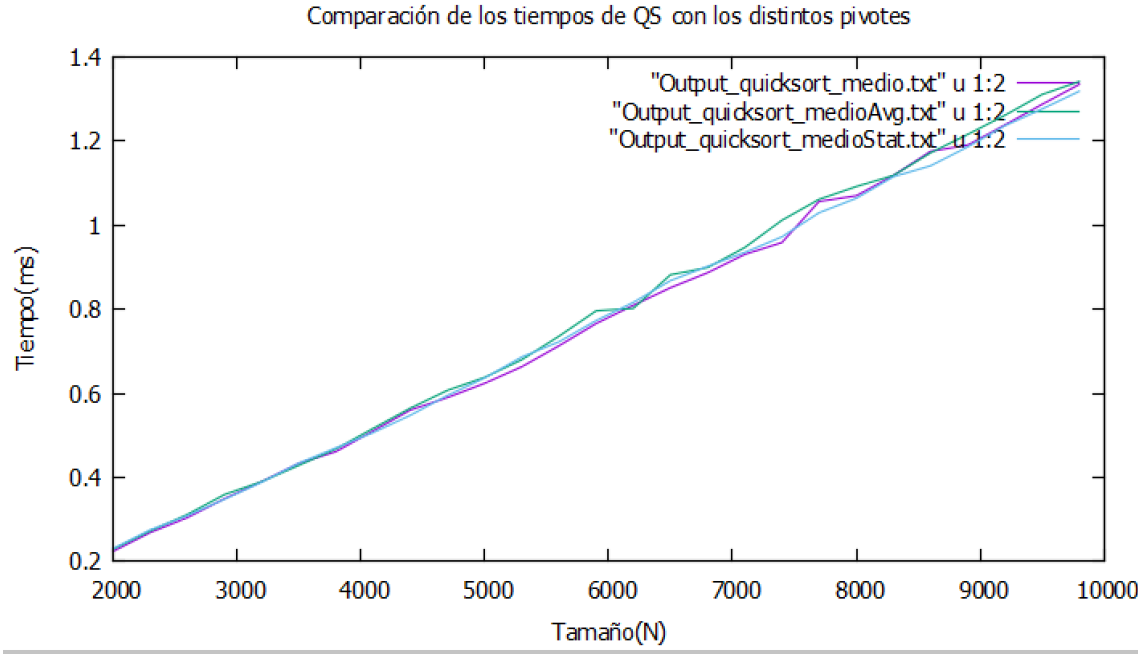
size	time(ms)	avg_ob	max_ob	min_ob.
20	0.001583	70.90	127	54
25	0.001984	98.52	163	75
30	0.002413	127.60	226	99
35	0.002861	158.90	263	124
40	0.003301	190.85	313	149
45	0.003768	224.19	355	175
50	0.004327	259.09	405	206
55	0.004776	294.27	445	236
60	0.005195	331.45	543	257
65	0.005732	367.91	569	291
70	0.006277	405.65	694	325
75	0.006694	444.60	666	356
80	0.007206	484.92	746	387
85	0.007717	523.82	810	423
90	0.008188	564.52	833	458
95	0.008801	606.54	901	486

Salida con función medio_stat

```
santorum:~/workspace (master) $ valgrind --leak-check=full ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000
==7783== Memcheck, a memory error detector
==7783== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==7783== Using Valgrind-3.10.1 and LibVEX; rerun with -h for copyright info
==7783== Command: ./ejercicio5 -num_min 20 -num_max 100 -incr 5 -numP 2000 -fichSalida Output_mergesort.txt
==7783==
Practica numero 2, apartado 2
Realizada por: Alejandro Santorum & David Cabornero
Pareja 10.
Grupo: 1201
array = 16
Salida correcta
==7783==
==7783== HEAP SUMMARY:
==7783==     in use at exit: 0 bytes in 0 blocks
==7783==   total heap usage: 32,018 allocs, 32,018 frees, 7,617,080 bytes allocated
==7783==
==7783== All heap blocks were freed -- no leaks are possible
==7783==
==7783== For counts of detected and suppressed errors, rerun with: -v
==7783== ERROR SUMMARY: 0 errors from 0 contexts (suppressed: 0 from 0)
```

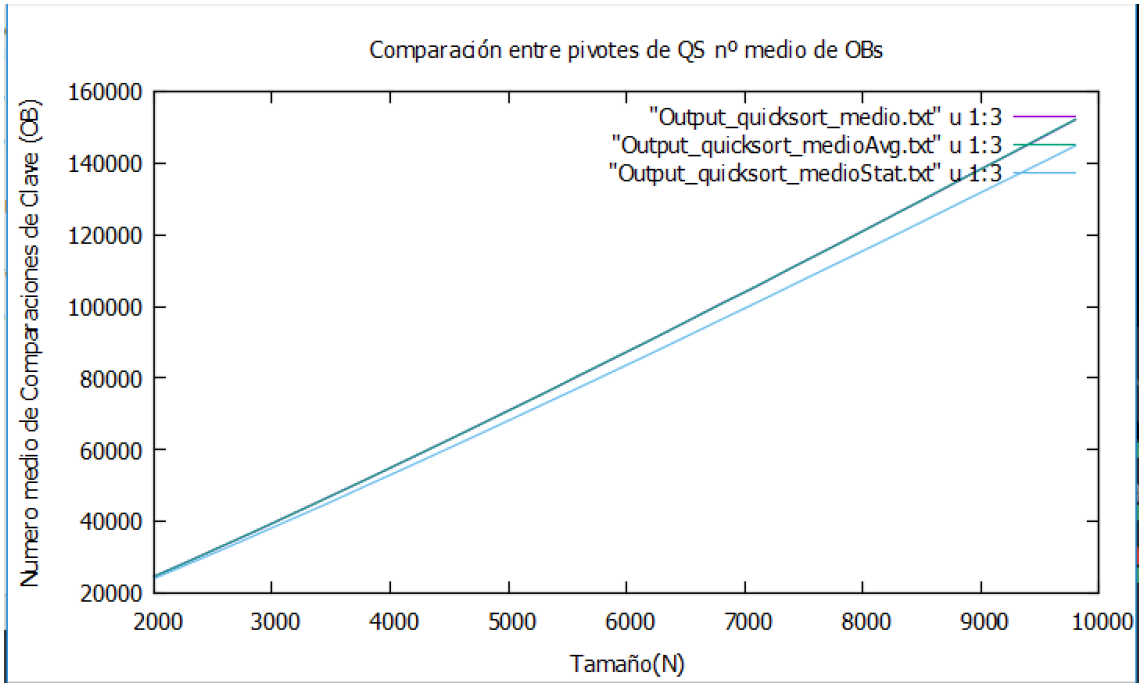
size	time(ms)	avg_ob	max_ob	min_ob.
20	0.002180	89.41	118	77
25	0.001891	120.04	157	105
30	0.002318	152.76	194	132
35	0.002808	186.38	258	165
40	0.003187	221.75	272	196
45	0.003666	257.67	327	230
50	0.004146	294.53	387	263
55	0.004589	331.94	410	299
60	0.005113	370.99	479	335
65	0.005626	410.27	536	368
70	0.006028	450.74	581	411
75	0.006523	491.21	634	444
80	0.006983	531.97	673	483
85	0.007635	573.78	718	521
90	0.007994	615.28	796	559
95	0.008440	657.87	811	594

Gráfica con el tiempo medio de reloj comparando los tres pivotes empleados, comentarios a la gráfica.



Se recomienda ver esta imagen en el archivo pasado por Moodle ya que las tres líneas se solapan a lo largo de la gráfica. Podemos clarificar que la línea que tiende a un mejor redimiento (menor tiempo) es la que se corresponde con la función “Medio_stat” como cabía de esperar. Para tablas desordenadas de tamaño 10000 la diferencia no se puede decir que es una barbaridad, pero cuanto más crece N, la diferencia es cada vez más notoria.

Aprovechamos para introducir la gráfica comparando las tres funciones de pivotes pero ahora midiendo las OB´s (caso medio):



Si solo ve dos líneas es porque la línea de las OB´s realizadas con medio y medio_avg se solapan debido a que tienen ambas la misma eficiencia. La línea que muestra el mejor rendimiento vuelve a ser como se esperaba la línea del quicksort implementado con medio_stat.

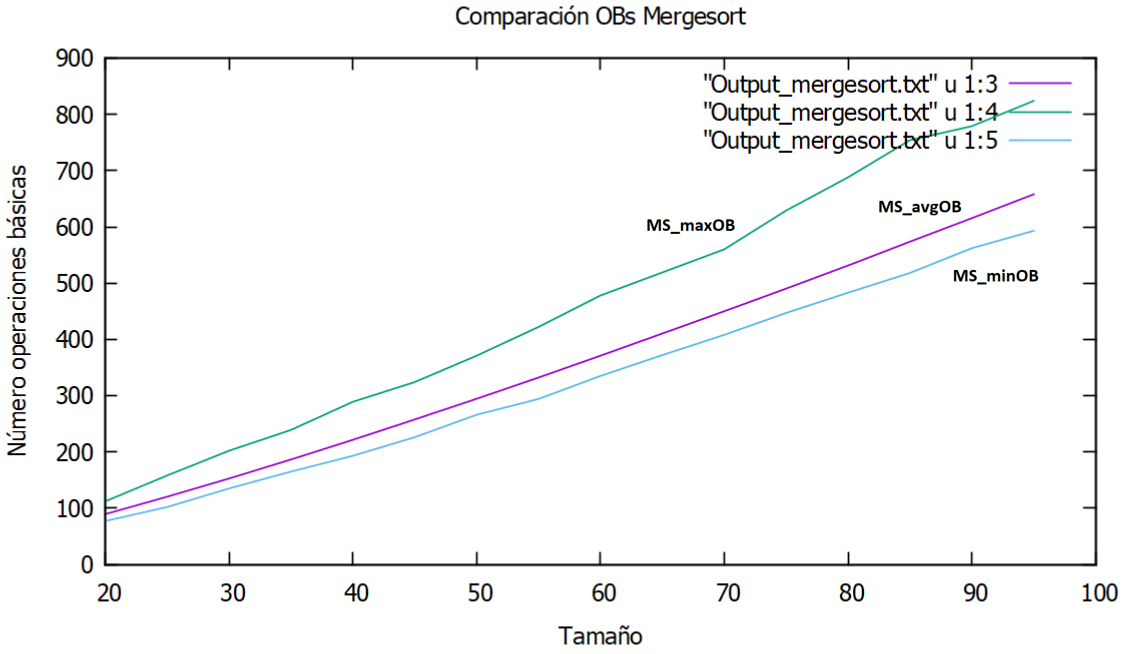
Podemos concluír entonces, que medio_stat es la función que más favorece a la eficiencia de Quicksort.

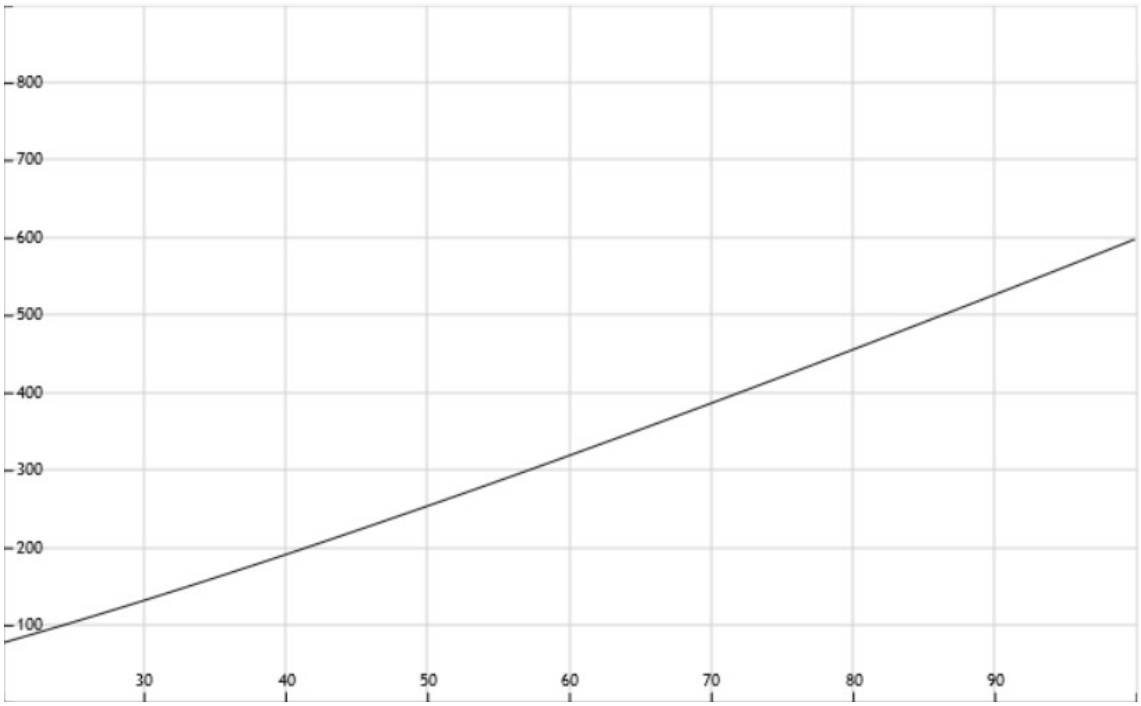
5. Respuesta a las preguntas teóricas.

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

5.1 Pregunta 1

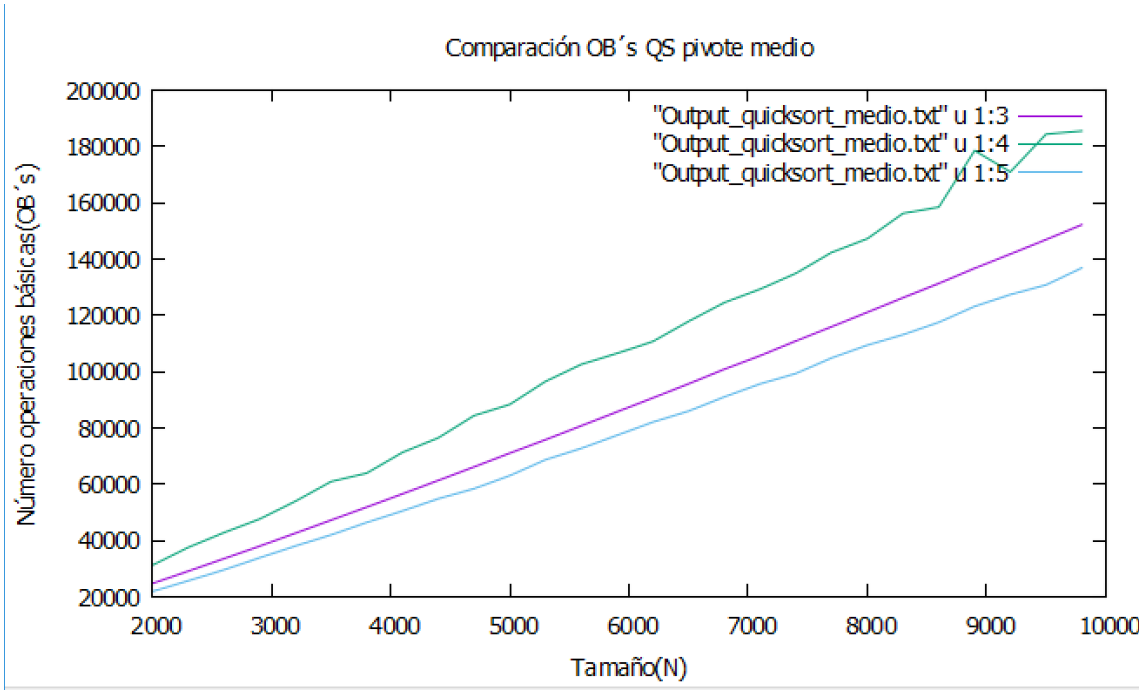
El caso medio teórico de Mergesort es θ(NlogN). A continuación se muestra de nuevo la gráfica de comparación de OB´s de Mergesort; debemos fijarnos en la línea del caso medio, ya que la compararemos con una gráfica de la función Kxlog(x) (con K = 3 en este caso). Este es legal, ya que KNlogN = θ(NlogN) para K diferente de 1 y 0.

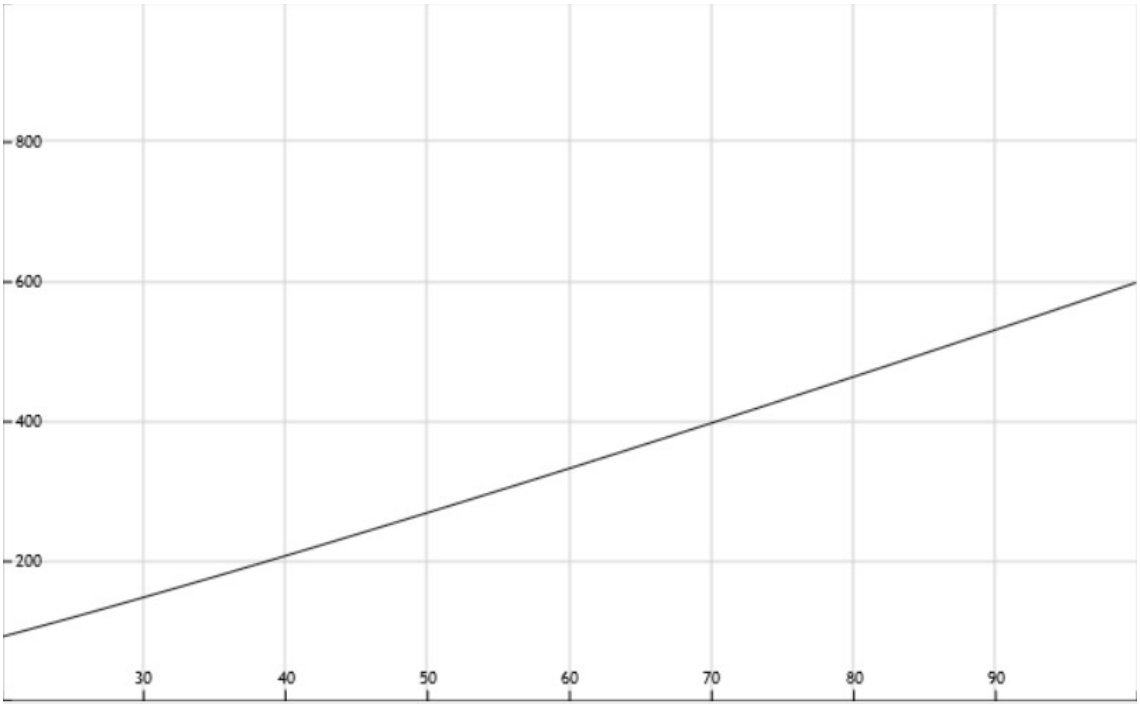




Se puede apreciar una clara correspondencia entre la gráfica del caso empírico (la primera) y la gráfica del caso teórico (la segunda, que no aporta ni título ni leyenda porque ha sido extraída de internet).

Por otro lado, el caso medio de Quicksort es de 2NlogN * O(N). De la misma forma que hemos procedido con Mergesort lo haremos ahora con Quicksort, mostrando una gráfica obtenida experimentalmente y otra que simboliza la expresión teórica (la segunda que no lleva leyenda porque ha sido extraída de internet).





De nuevo la primera gráfica es la resultante de los datos empírcos y, fijándonos en la línea del caso medio, podemos ver un claro parentesco con la gráfica de la expresión teórica: 2xlog(x) + 3x.

5.2 Pregunta 2

Poner como pivote un elemento fijo es ineficiente, ya que cabe la posibilidad de que ese pivote genere dos subtablas desordenadas muy descompensadas o directamente solo genere una subtabla desordenada con un elemento menos que antes.

La mejor elección para el pivote en Quicksort es obtener el valor medio entre, por ejemplo, el elemento en primer lugar de la tabla, en el lugar medio y el elemento de la última posición.

De todas formas, con tablas desordenadas de tamaño entre N = 2000 y N = 10000 (obsérvese una de las gráficas anterior), la diferencia no es muy notoria, pero a medida que N crece, aumenta la eficiencia de medio_stat.

5.3 Pregunta 3

El caso peor teórico de Mergesort es NlogN + O(N) y el caso mejor es N/2*logN + O(N). Además, el caso medio es θ(NlogN).

Sabemos que la operación básica de MS se encuentra en el bucle más interno de la rutina merge(combinar), por lo que es ahí donde se invierte más trabajo. Se realizan más comparaciones de clave en merge, y por lo tanto resulta de ahí el caso peor de Mergesort, cuando al dividir la tabla en dos subtablas los elementos de estas tablas se encuentran alternados, por ejemplo para una tabla de 6 elementos el caso peor sería: [1, 3, 5, 2, 4, 6], que después de la función partir resultarían dos subtablas [1, 3, 5] y [2, 4, 6] cuyos elementos están alternados.

Por otro lado, el caso mejor de Mergesort se alcanzaría cuando después de la función partir, todos los elementos de la primera subtabla son menores que todos los elementos de la segunda subtabla, ya que combinar haría el menor número de comparaciones de clave; es decir, el caso mejor para una tabla de 6 elementos sería: [1, 2, 3, 4, 5, 6] (tabla ya ordenada).

En cuanto al caso medio, es más difícil nombrar un candidato seguro para el cual se alcanza el número medio de OB´s de Mergesort, pero lógicamente se alcanzará en alguna de las variantes intermedias de los casos peor y mejor mostradas anteriormente. Más abajo se comenta una idea de un posible programa para estimar dicha tabla.

El caso peor teórico de Quicksort es N^2/2 – N/2 y el caso mejor se desconoce. Además, el caso medio es 2NlogN + O(N).

Como se ha visto en teoría, el caso peor de Quicksort se alcanza curiosamente en una tabla ya totalmente ordenada, por ejemplo para una tabla de 6 elementos el caso peor de QS sería en la tabla [1, 2, 3, 4, 5, 6].

Se comentó anteriormente que el caso mejor de QS se desconoce, por lo que no es de extrañar que la permutación en concreto en que alcanza el mejor rendimiento sea desconocida tamién. De todas formas, más abajo se aporta una buena de idea de un programa para tantear la mejor permutación para el caso de una tabla de N elementos.

En cuanto al caso medio, sostenemos lo mismo dicho para Mergesort.

Para calcular estrictamente el caso peor y mejor sería programar una función que te devolviese todas las tablas posibles de N elementos (N! Tablas). A continuación ordenarlas con el algoritmo deseado, guardando previamente el estado inicial de la tabla. En dos arrays auxiliares se irán guardando y actualizando los estados iniciales de las tablas con que se vayan alcanzando el menor y mayor número de OB´s (casos mejor y peor respectivamente) y una vez acabadado de ordenar todas las tablas sabríamos cual ha sido la permutación en concreto que ha necesitado más comparaciones de clave y la que menos.

Para el cálculo del caso medio, lo obtendríamos sumando todas las OB´s realizadas para cada una de las tablas y dividirlo entre N!.

5.4 Pregunta 4

Como los casos medios teóricos solo pueden ser referidos a las operaciones básicas debemos referirnos a su análogo en el estudio empírico.

Anteriormente se ha mostrado la gráfica en la que se puede apreciar los diferentes rendimientos de los dos algoritmos. No es muy grande la diferencia, pero se puede observar fácilmente que el rendimiento de Mergesort es peor que el de Quicksort centrándonos en el caso medio. En el estudio teórico, el caso medio de Mergesort es $\theta(NlogN)$ y el caso medio de Quicksort es $2NlogN + O(N)$. Esto concuerda con lo obtenido experimentalmente, ya que $\theta(NlogN)$ es menos preciso que $2NlogN + O(N)$, debido a que $\theta(NlogN)$ significa que la función F, que describe perfectamente el rendimiento medio de Mergesort, dividida entre NlogN, tiene a una constante distinta de cero y uno cuando N tiende a infinito. Esta constante puede ser perfectamente mayor que dos, por lo que a partir de ese momento el rendimiento de Mergesort para el caso medio es peor que el de Quicksort.

En definitiva, los resultados empíricos concuerdan con los teóricos.

Por otro lado, podemos sostener que Mergesort es un algoritmo peor que Quicksort en cuanto a manejo de memoria, ya que Quicksort es un algoritmo *in-place*, por lo que noi precisa de memoria auxiliar porque la ordenación se realiza en la propia tabla a ordenar. Sin embargo, Mergesort sí que necesita de esta memoria auxiliar en la función merge(combinar) para crear una “tabla auxiliar” de ayuda a la ordenación.

6. Conclusiones finales.

Tanto el código pedido para esta práctica como las cuestiones preguntadas se han ajustado al temario de la forma más ceñida posible, pues se basaba en implementar QuickSort y MergeSort y medir sus tiempos, para después medir tiempos con distintos pivotes de QuickSort.

No hemos tenido ningún problema especial: nuestras gráficas son más o menos uniformes (exceptuando el caso peor de QuickSort, por ser N^2) sin especiales picos, creemos que con nuestras comprobaciones hemos abarcado todos los errores posibles (incluso en algunos casos especificamos que son redundantes, pero que no le hacen ningún mal al código) y todos los cálculos nos dan según la teoría predice.

Comentar que los algoritmos implementados en esta práctica son mundialmente conocidos en el ámbito informático, por lo que no son una simple herramienta educativa para nuestra formación, sino que que tienen un uso significativo en el mundo fuera del aula. Dependiendo de nuestra prioridad (memoria, velocidad, recursos, etc.) existirán algoritmos de ordenación peores o mejores que estos dos, pero centrándonos únicamente en algoritmos de ordenación basados en comparaciones de clave se podría decir que estamos en frente de lo mejor.

Si nos preguntasen con qué algoritmo, Quicksort o Mergesort, nos quedaríamos, contestaríamos que depende. Depende de la cantidad de datos que queramos ordenar y de la memoria. Si el número es realmente grande decidiríamos utilizar Mergesort ya que el caso peor de Quicksort es horrendo. Aun así, podríamos incluso intentarlo con Quicksort siempre y cuando nos ayudásemos con la función de pivote “medio_stat”, y debido a que tiene una ventaja de gran peso, la memoria auxiliar de Quicksort es prácticamente nula ya que la ordenación se hace en la propia tabla (algoritmo *in place*) y que por el contrario Mergesort necesitaría, para una tabla enorme, una imporante cantidad de memoria axiliar.

Por otro lado, si el tamaño de la tabla a ordenar es moderada (no decimos pequeña) escogeríamos Quicksort sin ninguna duda, más rápida y menos abuso de la memoria auxiliar.

Sentimos si la longitud de esta memoria es demasiado amplia, hemos intentado contestar a todo lo que se nos pedía de la forma más completa y a la vez resumida que hemos sido capaces. Cada imagen de código y cada una de las gráficas ocupan un gran porcetaje de la longitud de esta memoria.