Análisis de Algoritmos 2016/2017

Práctica 2

David García Fernández, Antonio Martín Masuda,  Grupo 2101, Pareja 1.

|  |  |  |  |
| --- | --- | --- | --- |
| Código | Gráficas | Memoria | Total |
|  |  |  |  |

**1. Introducción.**

Aquí ponéis una introducción y discusión previa a la práctica.

**2. Objetivos**

Aquí indicáis el trabajo que vais a realizar en cada apartado.

2.1 Apartado 1

Objetivos del apartado 1.

2.2 Apartado 2

Objetivos del apartado 2.

2.3 Apartado 3

Objetivos del apartado 3.

2.4 Apartado 4

Objetivos del apartado 4.

2.5 Apartado 5

Objetivos del apartado 5.

**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.

3.1 Apartado 1

Metodología y solución adoptada del apartado 1

3.2 Apartado 2

Metodología y solución adoptada del apartado 2

3.3 Apartado 3

Metodología y solución adoptada del apartado 3

3.4 Apartado 4

Metodología y solución adoptada del apartado 4

3.5 Apartado 5

Metodología y solución adoptada del apartado 5

**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;

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;

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;

if(medio(tabla, ip, iu, pos, type) == 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: 217

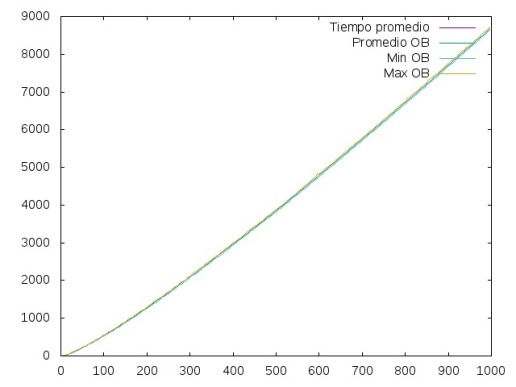
Como podemos comprobar la tabla queda perfectamente ordenada.

5.2 Apartado 2

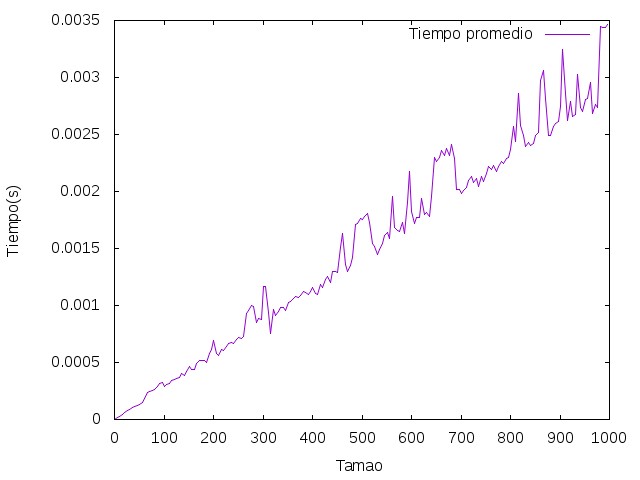
Introduciendo como parámetros de entrada:

./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:

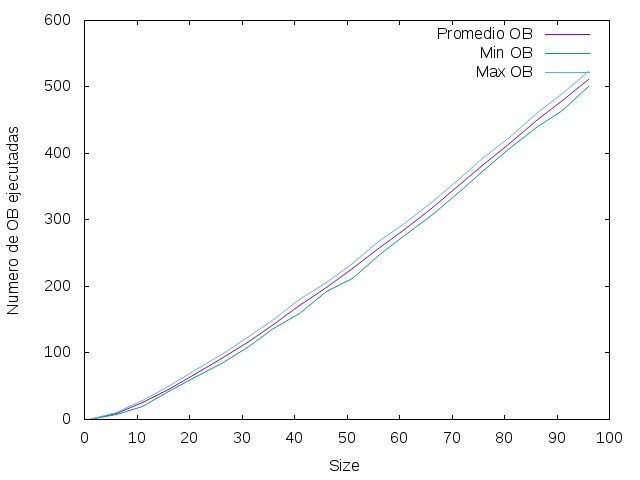


*Gráfica 1: Número OB frente Tamaño*



*Gráfica 2: Tiempo promedio frente Tamaño*

En la primera gráfica es inapreciable la 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

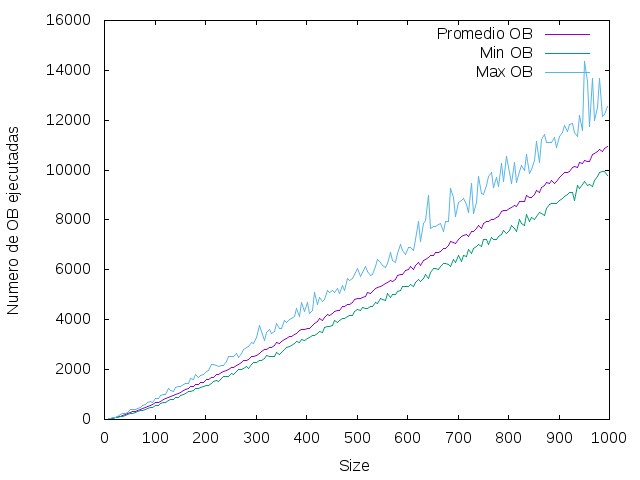
Como podemos comprobar la tabla queda perfectamente ordenada.

5.4 Apartado 4

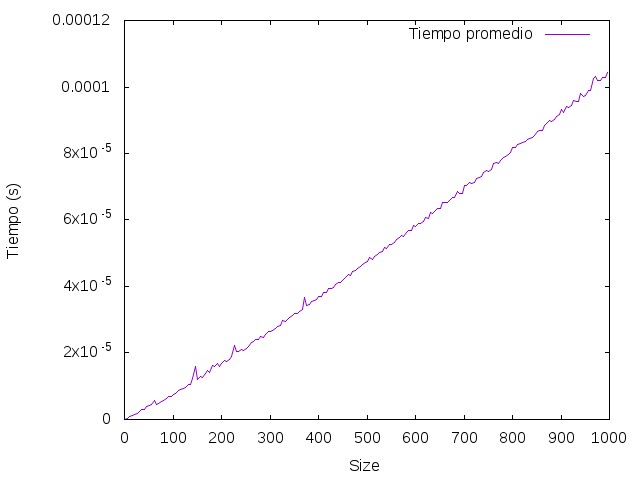
./ejercicio5 -num\_min 1 -num\_max 1000 -incr 5 -numP 50 -fichSalida datos

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*

**

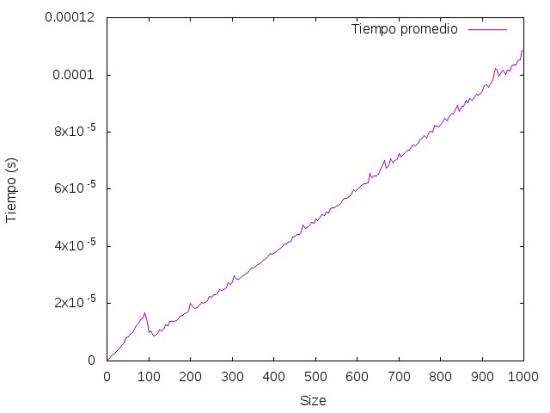
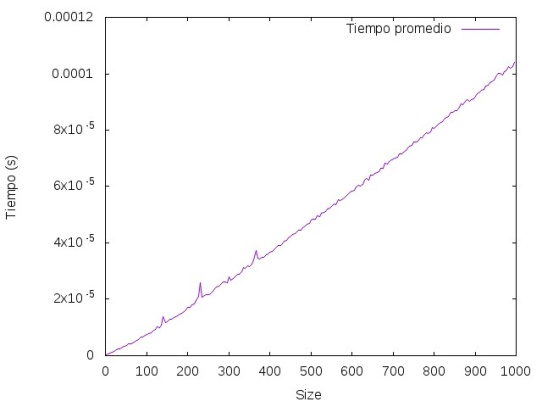
*Gráfica 5: Tiempo promedio con pivote 1 frente Tamaño*

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.

**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:

Caso mejor:

Caso medio:

Caso peor:

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:

Caso mejor: *BMS = O(N log(N))*

Caso medio:

Caso peor:

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

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

**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.