Análisis de Algoritmos 2017/2018

Práctica 2

Victoria Pelayo e Ignacio Rabuñal,  Grupo 1273.

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

**1. Introducción.**

El objetivo que se nos ha propuesto en esta práctica es el analizar dos conocidos algoritmos que siguen la metodología de divide y vencerás: MergeSort y QuickSort.

A lo largo de la práctica deberemos implementar ambos algoritmos y analizar su rendimiento para poder comparar los resultados con los supuestos teóricos.

**2. Objetivos**

2.1 Apartado 1

En este apartado debemos crear las funciones “merge” y “mergesort” que serán necesarias para seguir el algoritmo de ordenación MergeSort. Una vez creadas debemos probar con el programa ejercicio4.c que el algoritmo ordena correctamente.

2.2 Apartado 2

El objetivo de este apartado es modificar el archivo ejercicio5.c para obtener las tablas de tiempo y operaciones básicas de MergeSort. Una vez obtenidas las tablas, serán representadas para comparar los resultados con los supuestos teóricos.

2.3 Apartado 3

En este apartado debemos implementar el algoritmo QuickSort. Para ello, crearemos primero las funciónes “medio” y “partir” que se utilizaran en la implementación del algoritmo.

2.4 Apartado 4

El objetivo de este apartado es modificar el archivo ejercicio5.c para obtener las tablas de tiempo y operaciones básicas de QuickSort. Una vez obtenidas las tablas, serán representadas para comparar los resultados con los supuestos teóricos.

2.5 Apartado 5

En este último apartado debemos implementar dos funciones “medio\_avg” y “medio\_stat” que eligen el pivote para QuickSort de manera diferente. Consecuentemente, deberemos modificar la función “partir” para que sea posible usar cualquier función de elección de pivote. Para acabar, compararemos los tiempos de ejecucion y promedio de Obs de QuickSort con las diferentes rutinas pivote.

También hemos creado tres funciones del tipo pfunc\_ordena (definido en ordenación.h) ya que al ejercicio5 se le pasan funciones de ese tipo. Estas funciones nuevas se encargaran de llamar a QuickSort pasándole como argumento el método medio, medio\_avg o medio\_stmt.

**3. Herramientas y metodología**

En todos los apartados hemos utilizado el entorno Linux, el editor Atom para crear las funciones necesarias o modificar archivos anteriores. A la hora de comprobar nuestos ficheros, hemos utilizado gcc para compilar los archivos, Valgrind para asegurar una correcta gestión de memoria y el programa Gnuplot para representar los resultados obtenidos.

3.1 Apartado 1

Para empezar implementamos la función “merge” siguiendo el pseudocódigo propuesto en las transparencias de teoría y a continuación creamos “mergesort” siguiendo la misma metodología. Por último comprobamos la corrección de nuestro código con el programa ejercicio4.c.

3.2 Apartado 2

En el apartado anterior creamos el algoritmo MergeSort con un contador de OBs por lo que después de la ligera modificación de ejercicio5.c para adaptarlo a MergeSort y medimos los tiempos y OBs sin problemas.

3.3 Apartado 3

En primer lugar, implementamos la función “medio” trivialmente. A continuación nos servimos de los pseudocódigos de las transparencias de teoría de la asignatura para construír las funciones “partir” y “QuickSort”. Por último, modificamos el archivo ejercicio4.c para que sea aplicable al algoritmo creado previamente y comprobamos su corrección.

3.4 Apartado 4

Realizamos una ligera modificación en ejercicio5.c para adaptarlo a “QuickSort” y medimos los timepos y OBs sin problemas.

3.5 Apartado 5

La función “medio\_avg” fue adaptada trivialmente a partir de la función medio y para crear la función “medio\_stat” nos ayudamos de un árbol de decisión para representar la elección del pivote. Por último, adaptamos la función “partir” y “QuickSort” añadiéndole un puntero a función para escoger el método de elección de pivote.

También creamos tres funciones en ordenación.c que solo reciben tres parámetros y estas serán las que pasaremos en ejercicio5.c para guardar en un fichero los tiempos y el número de ejecuciones de OBs de “QuickSort” con sus distintintas funciones para elegir el pivote.

**4. Código fuente**

4.1 Apartado 1

int mergesort(int\* tabla, int ip, int iu){

int M = (iu + ip)/2;

int ob = 0;

if (ip == iu) return ob;

ob += mergesort(tabla, ip, M);

ob += mergesort(tabla, M + 1, iu);

ob += merge (tabla, ip, iu, M);

return ob;

}

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

int \*aux = NULL;

int i = ip;

int k = 0;

int j = imedio +1;

int ob = 0;

aux = (int\*)malloc(sizeof(aux[0]) \* (iu - ip + 1));

if(aux == NULL) return ERR;

while (i <= imedio && j<= iu){

ob++;

if (tabla[i] < tabla[j]){

aux[k] = tabla[i];

i++;

}

else{

aux[k] = tabla[j];

j++;

}

k++;

}

if(i > imedio){

while(j <= iu){

aux[k] = tabla[j];

k++;

j++;

}

}

else if(j > iu){

while( i <= imedio){

aux[k] = tabla[i];

k++;

i++;

}

}

tabla+=ip;

for(i = 0; i < k; i++){

tabla[i] = aux[i];

}

tabla-=ip;

free(aux);

return ob;

}

4.3 Apartado 3

int medio(int \*tabla, int ip, int iu,int \*pos){

\*pos = ip;

return 0;

}

int partir(int\* tabla, int ip, int iu, int \*pos, pfunc\_medio metodo){

int k;

int i;

int ob = 0;

ob = metodo(tabla,ip,iu,pos);

\*pos = ip;

k = tabla[\*pos];

swap(&tabla[ip], &tabla[\*pos]);

\*pos = ip;

for(i = ip + 1; i <= iu; i++ ){

ob++;

if(tabla[i] < k){

(\*pos)++;

swap(&tabla[i], &tabla[\*pos]);

}

}

swap(&tabla[ip], &tabla[\*pos]);

return ob;

}

int quicksort(int\* tabla, int ip, int iu, pfunc\_medio metodo){

int pos = 0;

int ob = 0;

if(ip > iu) return ERR;

else if(ip == iu) return ob;

else{

ob += partir(tabla, ip, iu, &pos, metodo);

if( ip < pos - 1)

ob += quicksort(tabla, ip, pos - 1, metodo);

if( pos + 1 < iu)

ob += quicksort(tabla, pos + 1, iu, metodo);

}

return ob;

}

4.5 Apartado 5

int medio\_avg(int \*tabla, int ip, int iu, int \*pos){

\*pos = (ip + iu) / 2;

return 0;

}

int medio\_stat (int\* tabla, int ip, int iu, int\* pos){

int im, ob;

ob++;

if (tabla[ip] <= tabla[im]){

ob++;

if(tabla[iu] <= tabla[im]){

ob++;

if(tabla[ip] <= tabla[iu]){

\*pos = iu;

}

else \*pos = ip;

}

else \*pos = im;

}

else {

ob++;

if(tabla[im] <= tabla[iu]){

ob++;

if(tabla[iu] <= tabla[ip]){

\*pos = iu;

}

else \*pos = ip;

}

else \*pos = im;

}

return ob;

}

int quicksort1(int\* tabla, int ip, int iu){

return quicksort(tabla,ip,iu,medio);

}

int quicksort2(int\* tabla, int ip, int iu){

return quicksort(tabla,ip,iu,medio\_avg);

}

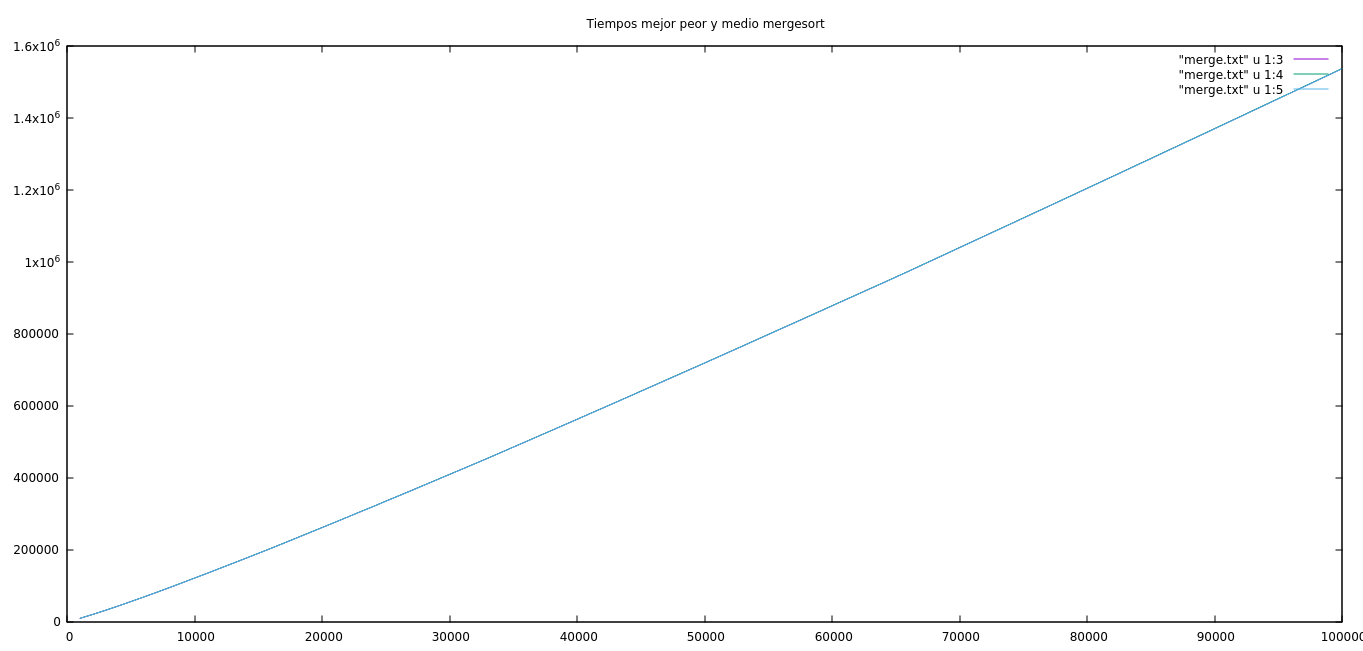
int quicksort3(int\* tabla, int ip, int iu){

return quicksort(tabla,ip,iu,medio\_stat);

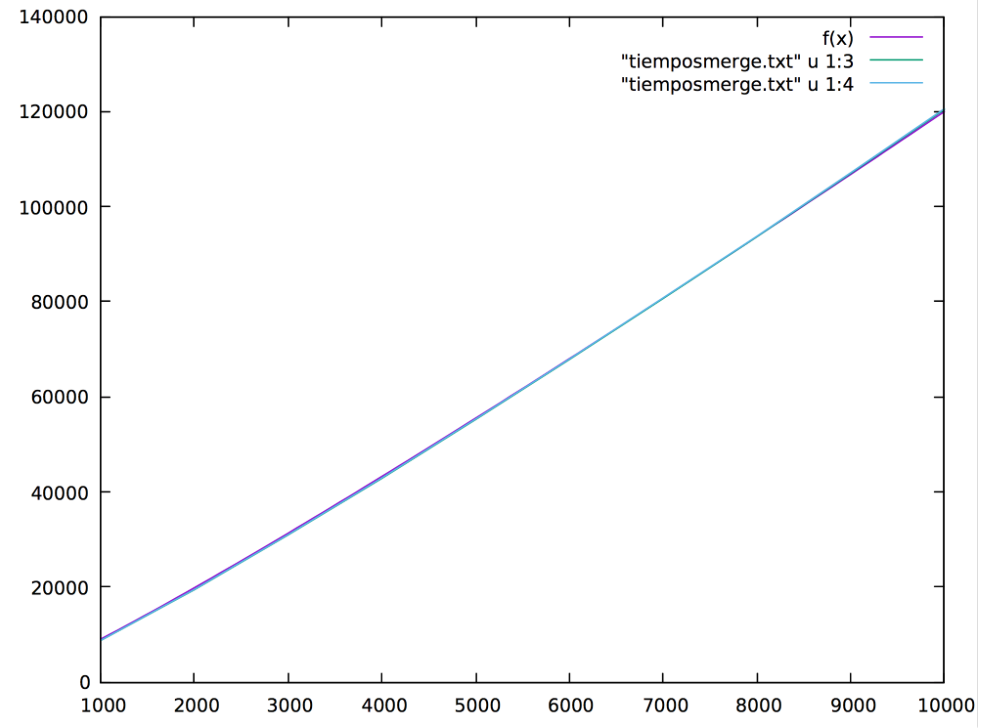
}

**5. Resultados, Gráficas**

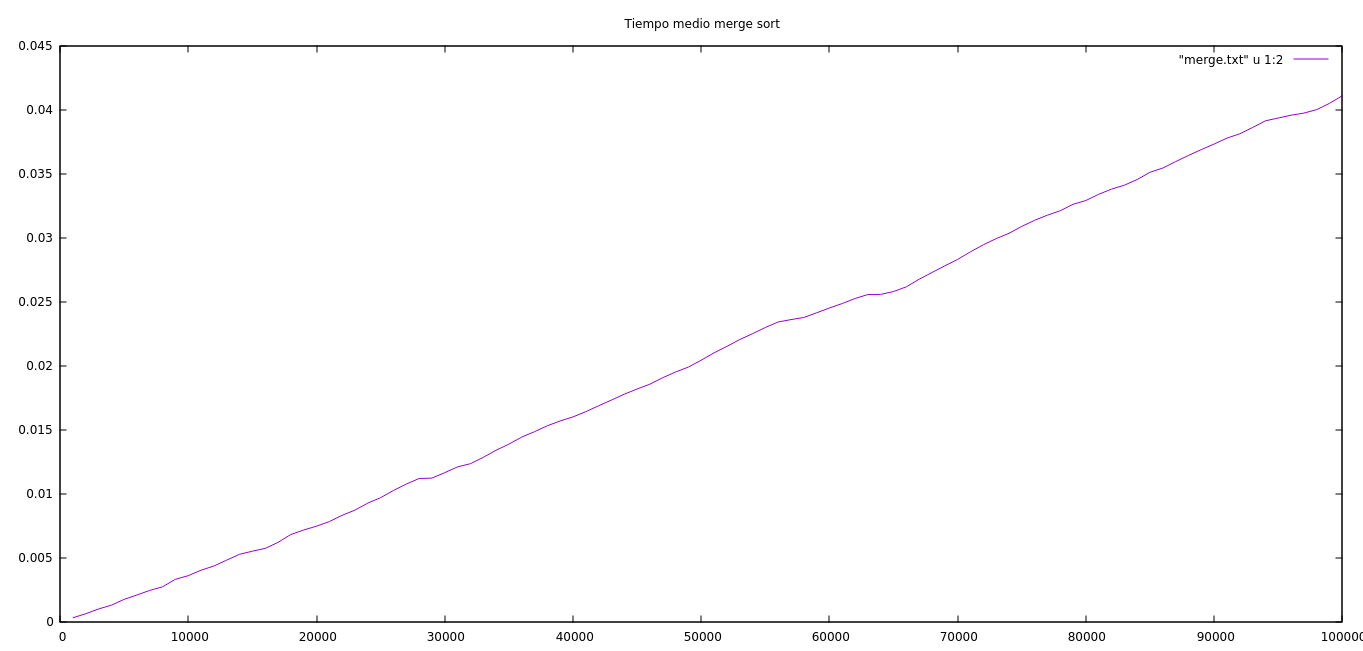
Gráfica comparando los tiempos mejor,peor y medio en OBs para MergeSort y la función que más se acerca al medio.



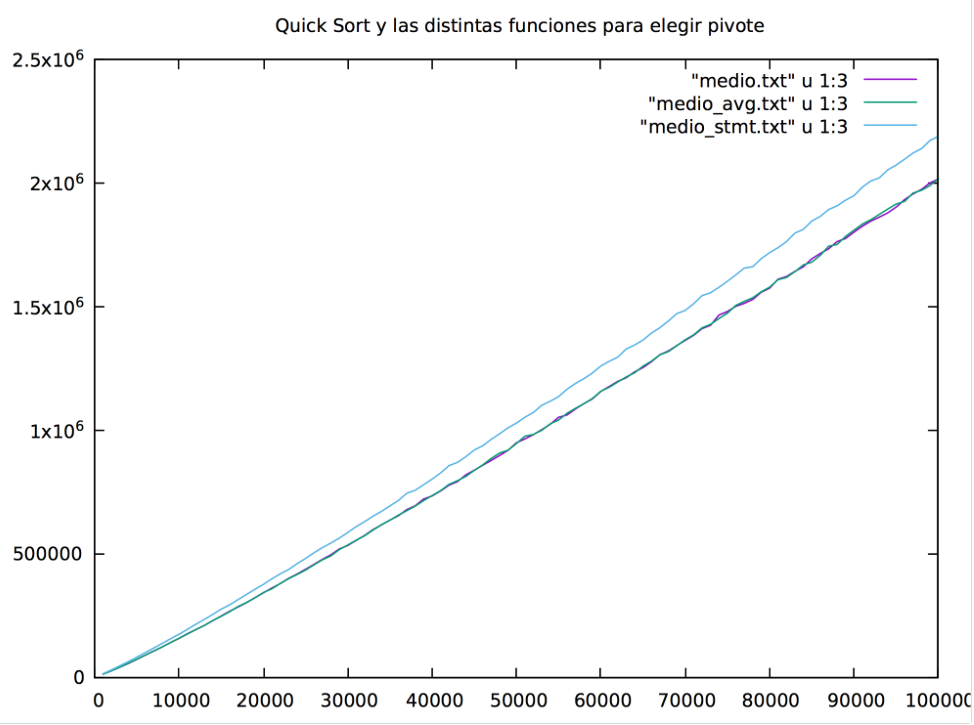
Grafica con los tiempos mejor, medio de MergeSort y la función que más se aproxima al medio que como vemos se aproxima mucho también a los casos medios. Esto vemos que coincide con los valores teóricos, además esta función es aproximadamente



Gráfica con el tiempo medio de reloj para MergeSort.



Gráfica comparando los tiempos medios de QuickSort con las tres funciones distintas para elegir el pivote.

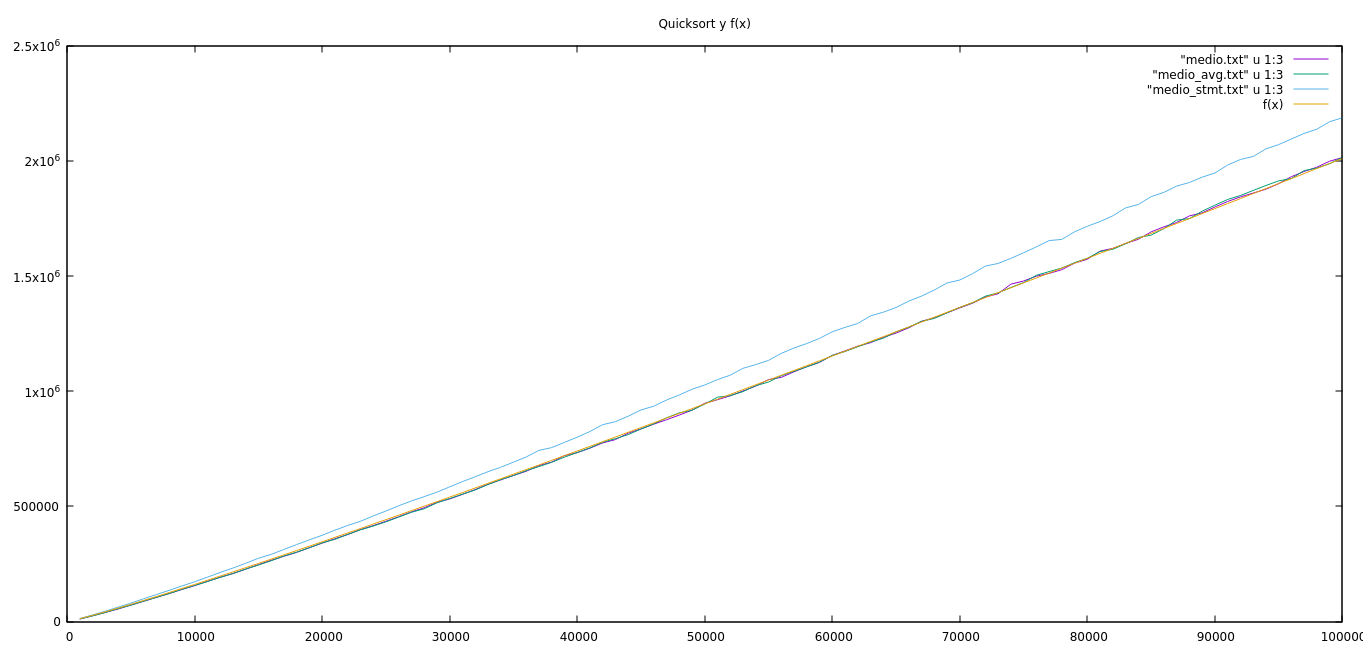


Como vemos en las gráficas de medio y medio\_avg no hay prácticamente diferencia ya que ambas eligen siempre el elemento de una posición de una tabla independiente del valor de este elemento, luego que la elección de este elemento sea más o menos acertada es de manera aleatoria.

Al implementar QuickSort como medio\_stmt vemos como el número medio de OBs es mayor ya que para hallar la mediana esta función dentro realiza comparaciones de clave, a diferencia de medio\_avg y medio, por ello el número de OBs es mayor.

Hemos hecho primero una aproximación de el tiempo medio con f(x) = a\*x\*log(x) y obtenemos que a = 1.7, si f(x) la aproximamos a medio\_stmt obtenemos a = 1.9 (más aproximado a lo esperado).

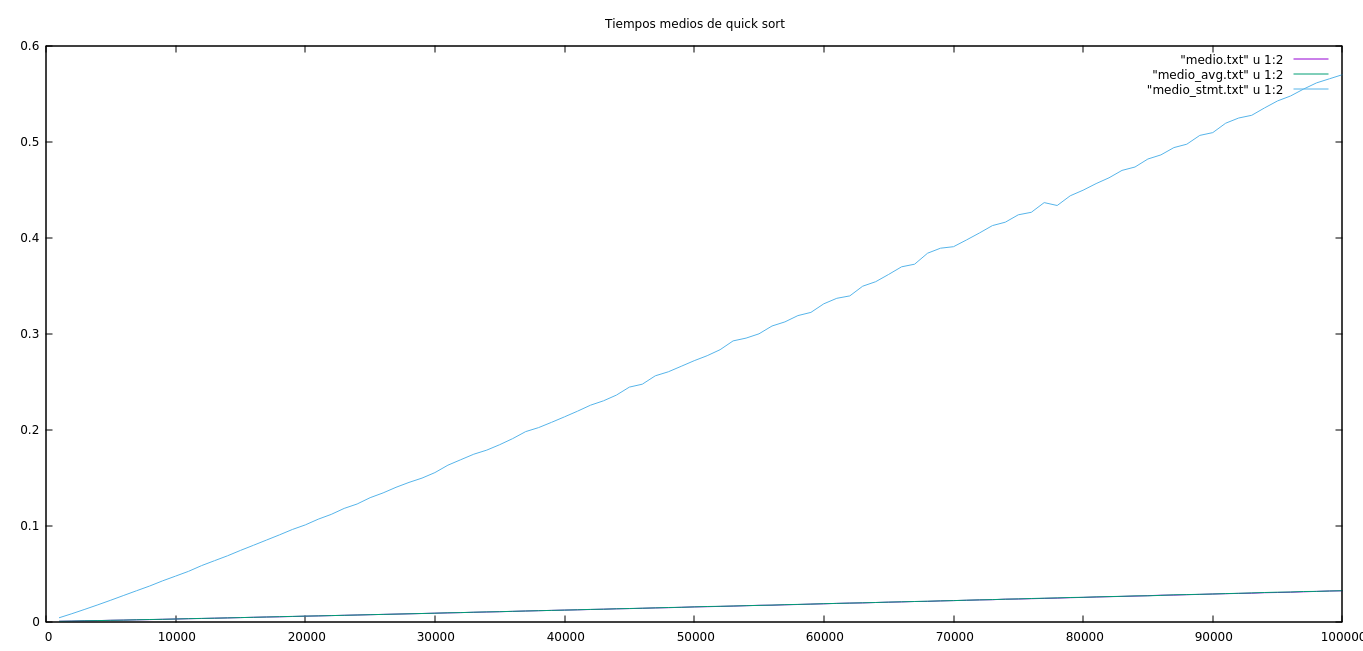
Aproximando f a tiempo medio de QuickSort, usando para elegir el pivote medio o medio\_avg.



Ahora aproximando f a la gráfica de QuickSort cuando utiliza medio\_stmt:

../../../Volumes/S%20I/fxmedi

Gráfica con el tiempo medio de reloj para QuickSort.



Como vemos cuando usa la función medio\_stmt tarda mucho más. Esto es debido a que las otras dos funciones solo tienen una línea de código que e suna asignación, sin embargo, medio\_stmt realiza múltiples comparaciones y asignaciones hasta ya devovler finalmente el pivote.

**5. Respuesta a las preguntas teóricas.**

5.1

El tiempo medio de mergesort se comporta tal y como habíamos previsto.

En el caso de QuickSort hemos podido comprobar como el tiempo de ejecución y de realización de la ob de una función recursiva puede llegar a variar muchísimo según la función recursiva que utilicemos.

Como hemos comprobado en la tabla de tiempos de QuickSort, dependiendo del método que se usase para elegir el pivote, la función podía tardar mucho más o no.

Con gnuplot comprobamos como el número medio de comparaciones de clave de quicksort era bastante aproximado a 2\*n\*log(n), lo que se espera de manera teórica. En nuestra práctica ha sido ligeramente menor, pero esto puede ser debido al “grado de desorden” de las permutaciones que se hayan creado aleatoriamente.

En cuanto a MergeSort hemos visto que los resultados de las gráficas se aproximan también bastante a lo esperado ya que al aproximar f(x) a los números máximos de OBs se aproximaba a n\*log(n), y esto vemos que también está muy cercano a los casos medios, como era de esperar ya que Ams(N) = O(N\*log(N)).

5.2

En la gráfica en la que comparamos el número de comparaciones de las OOs obtenemos que medio y medio\_avg realizan prácticamente siempre el mismo número de comparaciones. Esto coincide con lo esperado ya que ambas funciones no realizan dentro ninguna comparación, asignan el pivote a una posición determinada sin tener en cuenta el valor del elemento de esa posición, luego el número de comparaciones que se realizará dependerá de la “suerte” que se tenga al elegir el pivote y por tanto el número de veces que haya que ejecutar QuickSort.

Sin embargo, al usar la rutina medio\_avg vemos que el número de comparaciones es mucho mayor, esto es debido a que dentro de la propia función medio\_stmt se realizan comparaciones para elegir un pivote más acertado, vemos que cuánto menos es el tamaño de la permutación más se acerca a cuando ejecutamos QuickSort con las otras rutinas, esto es debido a que hay menos elementos y por tanto al realizar la mediana se ejecutan menos comparaciones. Sin embargo cuando la permutación es muy grande aunque el pivote se espere que sea bueno, se han realizado muchas comparaciones dentro de esta función.

Y algo similar ocurre con los timepos medios de ejecución pero de una manera mucho mas pronunciada, ya que los tiempos de medio y medio\_avg son prácticamente iguales y medio\_stmt son muy superiores debido a que son bastantes líneas de código a diferencia de las otras rutinas que son solo una línea que consiste en una asignación.

5.3

MergeSort:

-Caso peor: N\*log(N) + O(N)

-Caso mejor: ½ \* N \* log(N)

-Caso medio: = O(N\*log(N))

QuickSort:

-Caso peor: N^2/2 – N/2

-Caso mejor:

-Caso medio: 2\*n\*log(N) + O(N)

Si quisiéramos calcular el caso peor de manera empírica tendríamos que pasarle distintas tablas (totalmente desordenadas, es decir, de la forma {N, N – 1, N- 2, …, 1}) y medir el número de OBs, después sirviéndonos de Gnuplot al aproximarlo a f(x) de la forma f(x) = a\*x^2 + b^x +c veríamos como a = ½ y b = - ½, en QuickSort y en MergeSort haríamos lo mismo y comprobaríamos como al aproximarlo a f(x) sería f(x) = x\*log(x) + O(x) .

Para calcular el caso mejor tendríamos que pasarles tablas totalmente ordenadas y al realizar el mismo procedimiento que anteriormente veríamos como f(x) coincide con las expresiones esperadas.

Para calcular los casos medios nos sirve el procedimiento que estamos llevando a cabo ahora ya que les pasamos 1000 permutaciones por cada tamaño y debido a la función que implementamos para generar permutaciones éstas pueden estar muy desordenadas o no. Medimos el número de OBs al ejecutar el programa y luego seguimos el procedimiento anterior, en este caso f(x) puede que no coincida al 100% pero si vemos como se aproxima notablemente.

5.4

De manera empírica hemos obtenido que tanto por el número de comparaciones de clave y por el tiempo que tarda es más eficiente mergesort.

Esto coincide con lo esperado de manera teórica ya que Ams =O( N\*log(N)) y Aqs = 2\*N\*log(N).

Desde el punto de vista de gestión de memoria es más eficiente QuickSort, ya que no reserva memoria en ningún momento. MergeSort si utilizamos tablas pequeñas y disponemos de memoria suficiente que necesite reservar memoria no tiene por qué ser un inconveniente, sin embargo si las tablas son muy grandes será necesario reservar mucha memoria por lo que desde este punto de vista es más eficiente QuickSort.

**6. Conclusiones finales.**

En esta práctica hemos implementado los algoritmos de ordenación recursivos MergeSort y QuickSort (con 3 funciones distintas para elegir el pivote).

Hemos medido sus tiempos y el número de comparaciones de clave y lo hemos comparado con lo estudiado teóricamente.

Hemos comprobado como dependiendo de la función usada para elegir un pivote un mismo algoritmo puede variar mucho su tiempo de ejecución o no.

También hemos analizado cuál de los dos algoritmos es más eficiente y por qué ya que hemos concluido que uno en cuanto al tiempo de ejecución es mejor y el otro en cuánto a la gestión de memoria.