Análisis de Algoritmos 2018/2019

Práctica 2

Andrés Mena Godino y Juan Moreno Díez, Grupo 1272.

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica, implementaremos una serie de funciones de ordenación de números las cuales nos ayudarán a comprobar el rendimiento de los algoritmos estudiados en la parte teórica de la asignatura (**MergeSort** y **QuickSort)**. Con la implementación de estos algoritmos, buscamos también ver la diferencia con el algoritmo que habíamos implementado en la primera práctica (SelectSort), el cual era un algoritmo que tenía un rendimiento cuadrático para sus casos mejor, peor y medio.

A lo largo de la práctica, utilizaremos una serie de herramientas que posteriormente serán mencionadas las cuales nos facilitan la implementación del código de los algoritmos (de los cuales poseemos los pseudocódigos, que deberemos modificar para una correcta implementación de estos). Los ejecutable creados comprueban los algoritmos implementados, para confirmar que estos están funcionando de forma esperada. Por otra parte, crearemos una serie de ejecutables para recopilar más información acerca de los algoritmos, como el tiempo que tarda en ordenarse una permutación y el número mínimo, medio y máximo de operaciones básicas que realiza el algoritmo, todo ello en función del tamaño de distintas permutaciones. Estos datos los analizaremos mediante el uso de gráficas, con los que podremos obtener una mejor visión de cómo evolucionan los algoritmos, y compararlos una vez más con lo estudiado matemáticamente.

2. Objetivos

2.1 Apartado 1

El objetivo principal de este apartado es comprobar que el algoritmo que queremos implementar ordena correctamente y obtener el número de operaciones básicas que ha realizado el algoritmo. Para ello, tenemos que desarrollar las funciones: **int mergesort(int* tabla, int ip, int iu)** y la función de combinación **int merge(int* tabla, int ip, int iu, int imedio).** En este algoritmo, el peso de la ordenación recae sobre la función de combinar (merge).

2.2 Apartado 2

Con las implementaciones del apartado anterior, queremos obtener los siguientes datos con respecto al **MergeSort:** una tabla que nos muestre el tiempo medio que han tardado en ordenarse todas las tablas y el máximo, mínimo y promedio de operaciones básicas realizadas en función del tamaño de la permutación. Importante comparar con el resultado teórico del algoritmo, para lo cual utilizaremos gráficas.

2.3 Apartado 3

En este apartado habrá que implementar la rutina **QuickSort** definida como: **int quicksort(int* tabla, int ip, int iu)**. Se realizará con la ayuda de las siguientes funciones: **int partir (int* tabla, int ip, int iu, int *pos)** y **int medio(int *tabla, int ip, int iu, int *pos).** En este algoritmo, el peso de la ordenación está en la función partir, y nuestra función medio es la encargada de escoger el pivote (del cual, queremos comprobar, si la elección del pivote modificará el rendimiento del algoritmo), el cual se guardará en la posición pos. Las rutinas devolverán ERROR en caso de que se produzca un fallo y el número de operaciones básicas si no se ha producido ningún error. Tendremos que comprobar que la tabla se ha ordenado correctamente con el nuevo algoritmo implementado.

2.4 Apartado 4

Con las implementaciones del apartado anterior, queremos obtener los siguientes datos con respecto al **QuickSort:** una tabla que nos muestre el tiempo medio que han tardado en ordenarse todas las tablas y el máximo, mínimo y promedio de operaciones básicas realizadas en función del tamaño de la permutación. Importante comparar con el resultado teórico del algoritmo, para lo cual utilizaremos gráficas.

2.5 Apartado 5

El objetivo principal del apartado es implementar la rutina **quicksort_src**, la cual quita la recursión de cola de **QuickSort**. Realizaremos los mismos ejecutables que hemos hecho con la rutina **quicksort**, es decir ordenar la tabla y mostrar los tiempos de ejecución y número mínimo, medio y máximo de operaciones básicas dependiendo de la entrada. Una vez tengamos estos datos, podremos compararlos con la versión de Quicksort original.

3. Herramientas y metodología

3.1 Apartado 1

Como entorno de desarrollo se ha utilizado **Linux** (más en concreto Ubuntu), ya que ofrece herramientas muy útiles a la hora de programar y compilar proyectos. Como editor de texto, hemos utilizado **ATOM,** ya que estamos muy familiarizados con este programa. Uno de los aspectos más a favor de este editor de texto es su estructuración y la movilidad que existe para trabajar entre archivos. Para la compilación se ha utilizado **gcc**. Por último, para analizar las pérdidas de memoria de nuestro programa, hemos utilizado **Valgrind**.

3.2 Apartado 2

Como entorno de desarrollo se ha utilizado **Linux** (más en concreto Ubuntu), ya que ofrece herramientas muy útiles a la hora de programar y compilar proyectos. Como editor de texto, hemos utilizado **ATOM**, ya que estamos muy familiarizados con este programa. Uno de los aspectos más a favor de este editor de texto es su estructuración y la movilidad que existe para trabajar entre archivos. Para la compilación se ha utilizado **gcc**. Para realizar las gráficas que comparan los distintos algoritmos hemos utilizado **GNUPLOT**. Por último, para analizar las pérdidas de memoria de nuestro programa, hemos utilizado **Valgrind**.

3.3 Apartado 3

Como entorno de desarrollo se ha utilizado **Linux** (más en concreto Ubuntu), ya que ofrece herramientas muy útiles a la hora de programar y compilar proyectos. Como editor de texto, hemos utilizado **ATOM,** ya que estamos muy familiarizados con este programa. Uno de los aspectos más a favor de este editor de texto es su estructuración y la movilidad que existe para trabajar entre archivos. Para la compilación se ha utilizado **gcc**. Por último, para analizar las pérdidas de memoria de nuestro programa, hemos utilizado **Valgrind**.

3.4 Apartado 4

Como entorno de desarrollo se ha utilizado **Linux** (más en concreto Ubuntu), ya que ofrece herramientas muy útiles a la hora de programar y compilar proyectos. Como editor de texto,

hemos utilizado **ATOM**, ya que estamos muy familiarizados con este programa. Uno de los aspectos más a favor de este editor de texto es su estructuración y la movilidad que existe para trabajar entre archivos. Para la compilación se ha utilizado **gcc**. Para realizar las gráficas que comparan los distintos algoritmos hemos utilizado **GNUPLOT**. Por último, para analizar las pérdidas de memoria de nuestro programa, hemos utilizado **Valgrind**.

3.5 Apartado 5

Como entorno de desarrollo se ha utilizado **Linux** (más en concreto Ubuntu), ya que ofrece herramientas muy útiles a la hora de programar y compilar proyectos. Como editor de texto, hemos utilizado **ATOM**, ya que estamos muy familiarizados con este programa. Uno de los aspectos más a favor de este editor de texto es su estructuración y la movilidad que existe para trabajar entre archivos. Para la compilación se ha utilizado **gcc**. Para realizar las gráficas que comparan los distintos algoritmos hemos utilizado **GNUPLOT**. Por último, para analizar las pérdidas de memoria de nuestro programa, hemos utilizado **Valgrind**.

4. Código fuente

4.1 Apartado 1

```
int merge(int* tabla, int ip, int iu, int imedio){
  int i = 0, k = 0 , j = 0, obs = 0;
  int *aux;

  if(tabla == NULL || iu < ip || imedio < ip || iu < imedio){
    return ERR;
  }

  aux = (int*)malloc((iu-ip+1)*sizeof(int));
  if(aux == NULL)
    return ERR;

  i = ip;
  j = imedio + 1;

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

    }

      k++;

    }

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

  for(i=0 ; i <= iu-ip; i++)
    tabla[ip+i] = aux[i];
```

```
  free(aux);

  return obs;
}

int mergesort(int* tabla, int ip, int iu){
  int m, obs=0, ret;

  if (ip>iu || tabla == NULL)
    return ERR;

  else if (ip==iu)
    return 0;

  else{
    m=(ip+iu)/2;
    ret = mergesort(tabla, ip, m);
    if(ret == ERR)
      return ERR;

    obs+=ret;

    ret = mergesort(tabla, m+1, iu);
    if(ret == ERR)
      return ERR;

    obs+=ret;

    ret = merge(tabla, ip, iu, m);
    if(ret == ERR)
      return ERR;

    obs+=ret;
  }

  return obs;
}
```

4.3 Apartado 3

```
int quicksort(int* tabla, int ip, int iu){
  int pivote=0, obs_aux1=0, obs_aux2=0, obs_aux3=0;

  if (ip>iu)
    return ERR;

  else if (ip==iu)
    return 0;

  else{
    if ((obs_aux1=partir(tabla, ip, iu, &pivote))==ERR)
      return ERR;

    if (ip < pivote-1)
      if ((obs_aux2=quicksort(tabla, ip, pivote-1))==ERR)
        return ERR;


    if (pivote+1 < iu)
      if ((obs_aux3=quicksort(tabla, pivote+1, iu))==ERR)
        return ERR;
  }

  return obs_aux1+obs_aux2+obs_aux3;
}


int partir(int* tabla, int ip, int iu, int *pos){
  int i, k, obs=0;

  if(ip > iu){
    return ERR;
  }

  medio(tabla, ip, iu, pos);

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

  *pos = ip;

  for(i=ip+1; i <= iu; i++){
    if(tabla[i] < k){
      (*pos)++;
      swap(&tabla[i], &tabla[*pos]);
    }
    obs++;
  }
```

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

  return obs;
}


int medio(int *tabla, int ip, int iu, int *pos){
  if (ip > iu)
    return ERR;

  *pos=(ip+iu) / 2;

  return 0;
}
```

4.5 Apartado 5

```
int quicksort_src(int* tabla, int ip, int iu){
  int pivote=0, obs_aux1=0, obs=0;

  if (ip==iu)
    return 0;

  while (ip<iu){
    if ((obs_aux1=partir(tabla, ip, iu, &pivote))==ERR)
      return ERR;

    obs+=obs_aux1;

    if ((obs_aux1=quicksort_src(tabla, ip, pivote-1))==ERR)
        return ERR;

    obs+=obs_aux1;

    ip=pivote+1;
  }

  return obs;
}
```

5. Resultados, Gráficas

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

5.1 Apartado 1

```
e378187@12-28-66-197:~/Andres_Mena_Juan_Moreno$ ./ejercicio4_merge -tamanio 10
Practica numero 2, apartado 1
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
1       2       3       4       5       6       7       8       9       10
```

Como podemos comprobar, el algoritmo mergesort se ha implementado correctamente ya que ordena de forma ascendente una tabla de tamaño 10.

5.2 Apartado 2

Ejecución para generar un archivo del cual nos interesan las operaciones básicas que realiza el algoritmo mergesort (medias, mínimas y máximas):

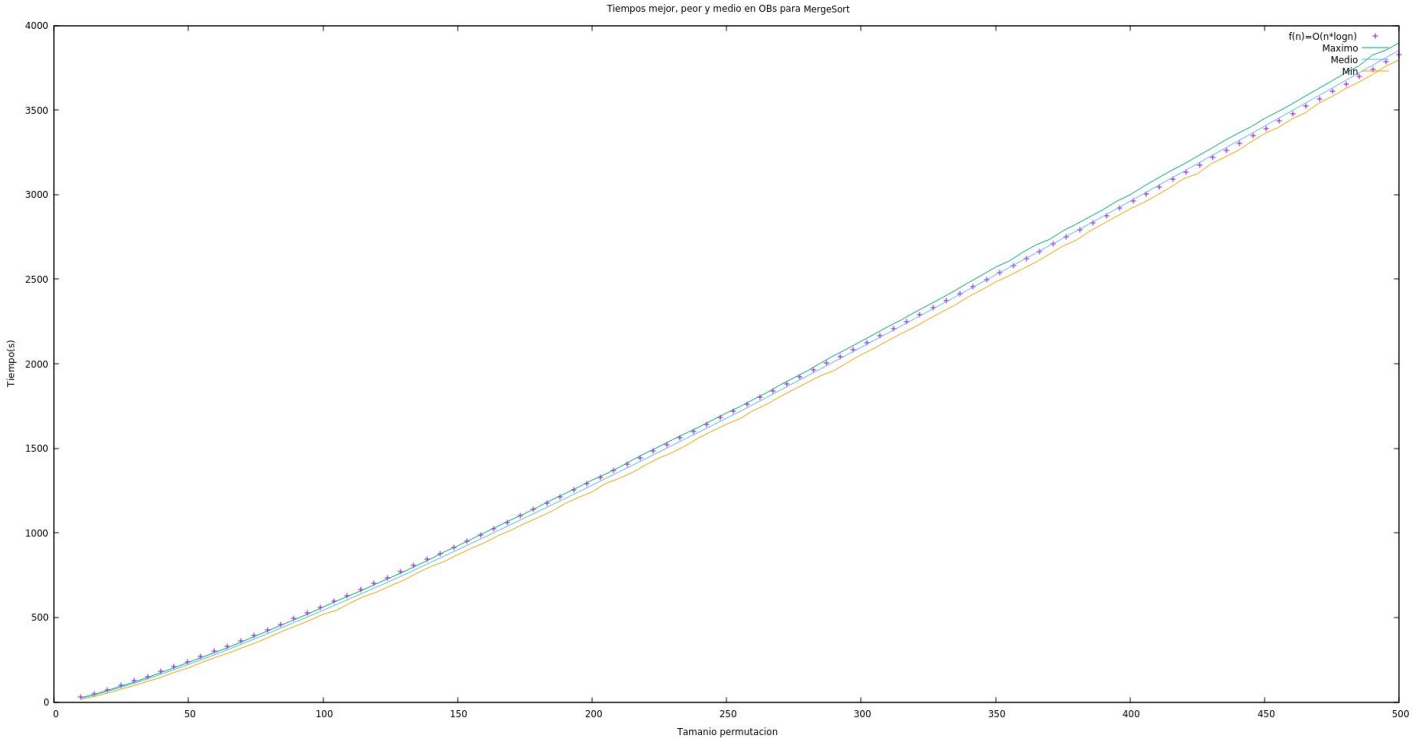
```
e378187@12-31-66-200:~/UnidadH/aalg/P2/Andres_Mena_Juan_Moreno(1)$ ./ejercicio5_merge -num_min 10 -num_max 500 -incr 5 -numP 10000 -fichSalida obs-merge.txt
Practica numero 2, apartado 2
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
Salida correcta
```

Ejecución para generar un archivo del cual nos interesan el tiempo medio de ordenación de las permutaciones. La diferencia frente a la anterior ejecución es que en este caso comenzaremos con un tamaño mínimo de permutación de 10000 (frente a los 10 de antes), y un tamaño máximo de 100000 (frente a los 500 de antes). El incremento también aumentara proporcionalmente, de los 500 de la primera ejecución a 2000 en la segunda (con lo que esperamos obtener 45 observaciones de tiempo). Finalmente, reducimos el número de permutaciones de 10000 a 1000 ya que sino la ejecución del programa llevaría demasiado tiempo debido al valor tan elevado escogido para el resto de los parámetros. Como el objetivo de esta segunda ejecución es medir tiempos, es lógico que esta ejecución tarde más tiempo en darnos el resultado, ya que si usamos los datos de tiempo obtenidos en la anterior ejecución (muy rápida), serán muy pequeños y poco precisos:

```
e378187@12-31-66-200:~/UnidadH/aalg/P2/Andres_Mena_Juan_Moreno(1)$ ./ejercicio5_merge -num_min 10000 -num_max 100000 -incr 2000 -numP 1000 -fichSalida tiempos_merge.txt
Practica numero 2, apartado 2
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
Salida correcta
```

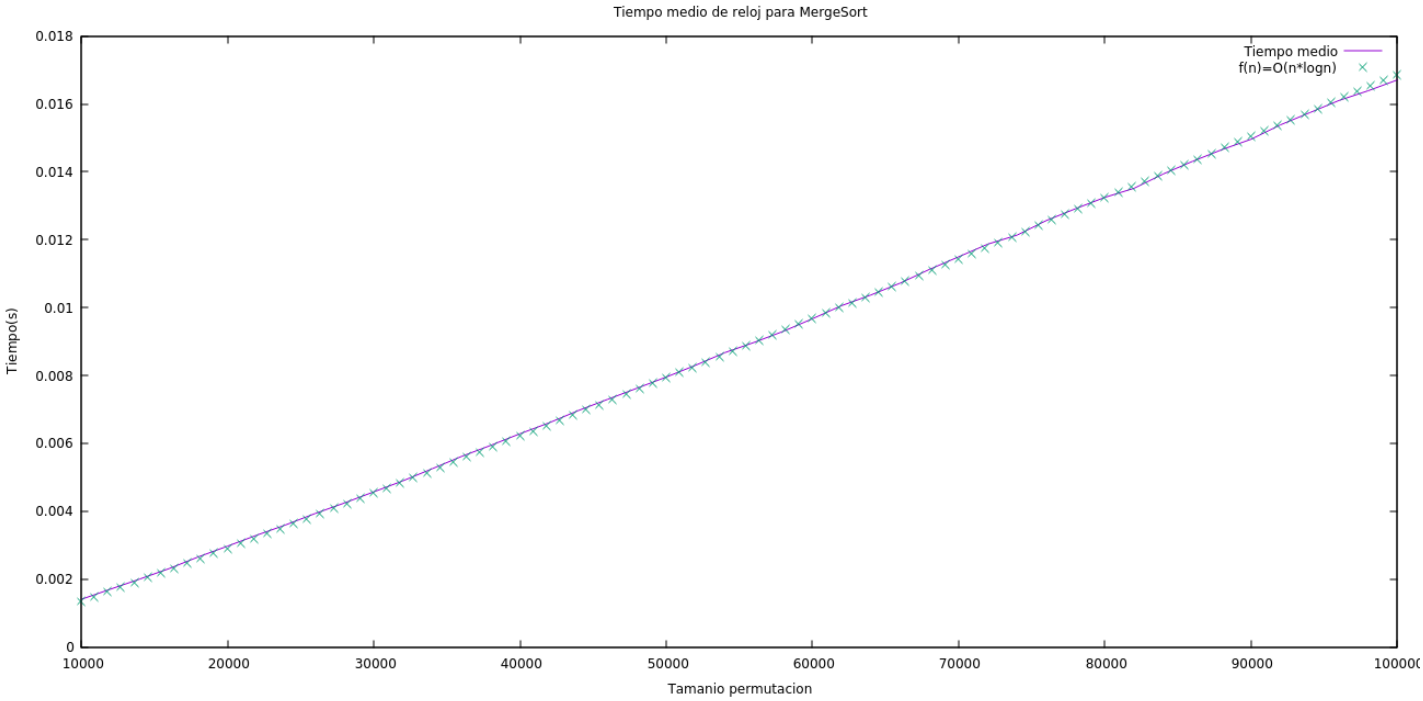
A partir de los ficheros que hemos generado con las anteriores ejecuciones, realizamos las siguientes gráficas, dejando en ambos casos en el eje de las x el tamaño de la permutación, para representarlo, en una de las gráficas, frente al número de operaciones básicas en caso mejor, peor y medio realizadas por el algoritmo y, en la otra gráfica, frente al tiempo medio que ha llevado ordenar las tablas.

Gráfica comparando los tiempos mejor, peor y medio en OBs para MergeSort:



Como podemos observar a partir de esta gráfica, el caso peor, medio y mejor nos dan un rendimiento muy similar usando el algoritmo de MergeSort. Además, hemos conseguido ajustar una función loglineal (representada con aspas) a los datos que habíamos obtenido de forma experimental ejecutando el algoritmo con el ordenador, por lo que concluimos que el rendimiento en operaciones básicas es de orden loglineal, que es lo que habíamos predicho que íbamos a obtener.

Gráfica con el tiempo medio de reloj para MergeSort:



Como podemos observar en esta gráfica, en la que se muestra el tiempo medio que tarda en ordenarse una permutación con el tamaño indicado el eje x, el crecimiento de la recta es de orden loglineal, ya que hemos conseguido ajustar una función loglineal (representada con aspas), que es casi idéntica a la recta obtenida mediante valores experimentales (es decir, con la ejecución del programa). Comprobamos que como habíamos analizado matemáticamente en la asignatura de teoría, el crecimiento del tiempo y de la entrada para el algoritmo MergeSort es loglineal.

5.3 Apartado 3

```
e378187@12-28-66-197:~/Andres_Mena_Juan_Moreno$ ./ejercicio4_quick -tamanio 10
Practica numero 2, apartado 3
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
1       2       3       4       5       6       7       8       9       10
```

Como podemos comprobar, el algoritmo quicksort se ha implementado correctamente ya que ordena de forma ascendente una tabla de tamaño 10.

5.4 Apartado 4

Ejecución para generar un archivo del cual nos interesan las operaciones básicas que realiza el algoritmo quicksort (medias, mínimas y máximas):

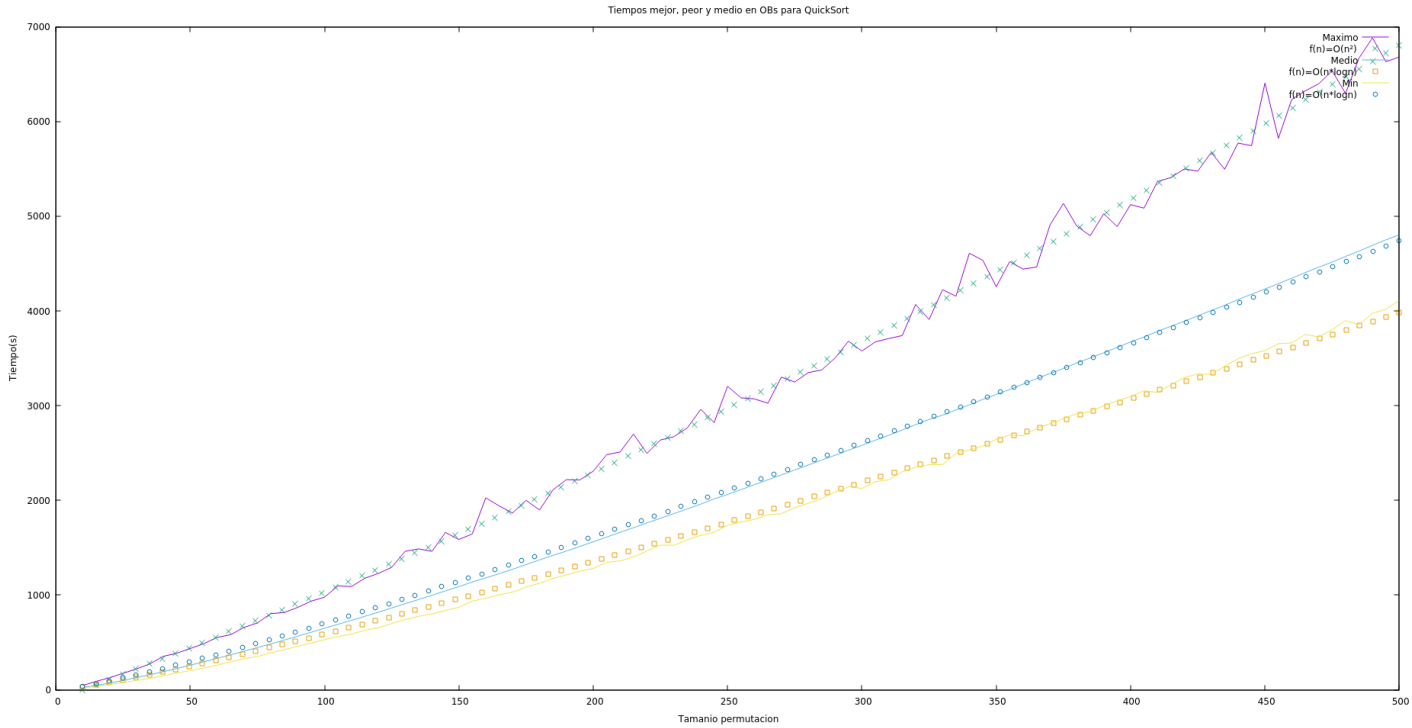
```
e378187@12-31-66-200:~/UnidadH/aalg/P2/Andres_Mena_Juan_Moreno(1)$ ./ejercicio5_quick -num_min 10 -num_max 500 -incr 5 -numP 10000 -fichSalida obs-quick.txt
Practica numero 2, apartado 4
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
Salida correcta
```

Ejecución para generar un archivo del cual nos interesan el tiempo medio de ordenación de las permutaciones. La diferencia frente a la anterior ejecución es que en este caso comenzaremos con un tamaño mínimo de permutación de 10000 (frente a los 10 de antes), y un tamaño máximo de 100000 (frente a los 500 de antes). El incremento también aumentara proporcionalmente, de los 500 de la primera ejecución a 2000 en la segunda (con lo que esperamos obtener 45 observaciones de tiempo). Finalmente, reducimos el número de permutaciones de 10000 a 1000 ya que sino la ejecución del programa llevaría demasiado tiempo debido al valor tan elevado escogido para el resto de los parámetros. Como el objetivo de esta segunda ejecución es medir tiempos, es lógico que esta ejecución tarde más tiempo en darnos el resultado, ya que si usamos los datos de tiempo obtenidos en la anterior ejecución (muy rápida), serán muy pequeños y poco precisos:

```
e378187@12-28-66-197:~/Andres_Mena_Juan_Moreno$ ./ejercicio5_quick -num_min 10000 -num_max 100000 -incr 2000 -numP 1000 -fichSalida tiempos_quick.txt
Practica numero 2, apartado 4
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
--------------------Salida correcta
```

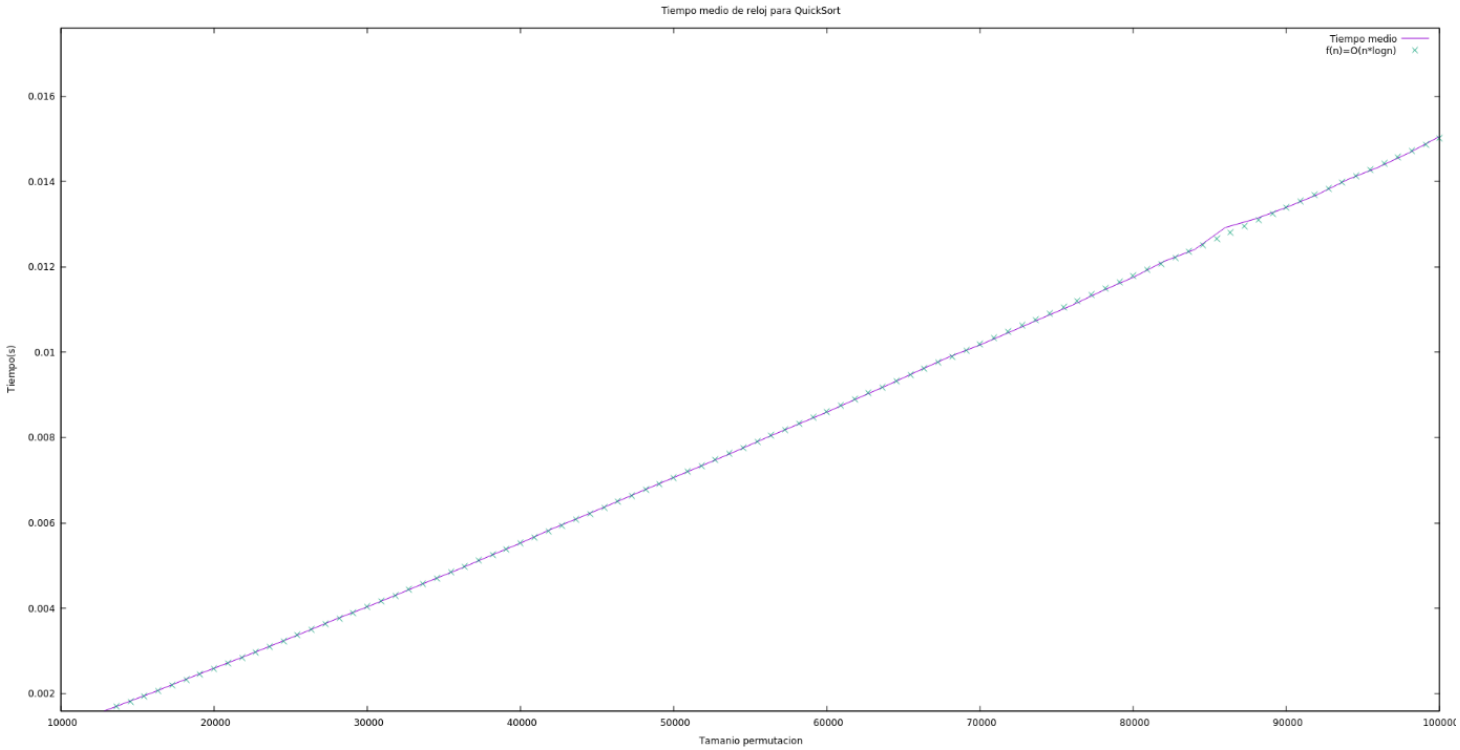
A partir de los ficheros que hemos generado con las anteriores ejecuciones, realizamos las siguientes gráficas, dejando en ambos casos en el eje de las x el tamaño de la permutación, para representarlo, en una de las gráficas, frente al número de operaciones básicas en caso mejor, peor y medio realizadas por el algoritmo y, en la otra gráfica, frente al tiempo medio que ha llevado ordenar las tablas.

Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort:



Como podemos observar a partir de esta gráfica, a diferencia del algoritmo de ordenación Mergesort, en el caso del Quicksort sí que existe una diferencia considerable en el número de operaciones básicas que se realizan en el caso peor, medio y mejor respecto al tamaño de la permutación (a mayor tamaño, mayor es esta diferencia como podemos observar en la gráfica). Para analizar mejor esta diferencia, y poder cuantificarla de forma matemática, hemos intentado ajustar estas curvas a ecuaciones matemáticas, obteniendo como resultado las curvas dibujadas en la gráfica con círculos, cuadrados y aspas. En el caso peor del algoritmo, el rendimiento es cuadrático, lo cual supone uno de los mayores defectos de este algoritmo. Sin embargo, el caso medio y el caso mejor tienen un rendimiento de orden loglineal, que coincide con el rendimiento que tenía el mergesort (pero en su caso, para todos los casos). Algo llamativo de esta gráfica son los picos que hay en el número máximo de operaciones básicas, que será explicado más adelante. Los rendimientos cuadráticos (para el caso peor) y loglineal (para los caso mejor y medio) eran los resultados que esperábamos obtener cuando habíamos estudiado el algoritmo de forma teórica, por lo que confirmamos que la gráfica es correcta.

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



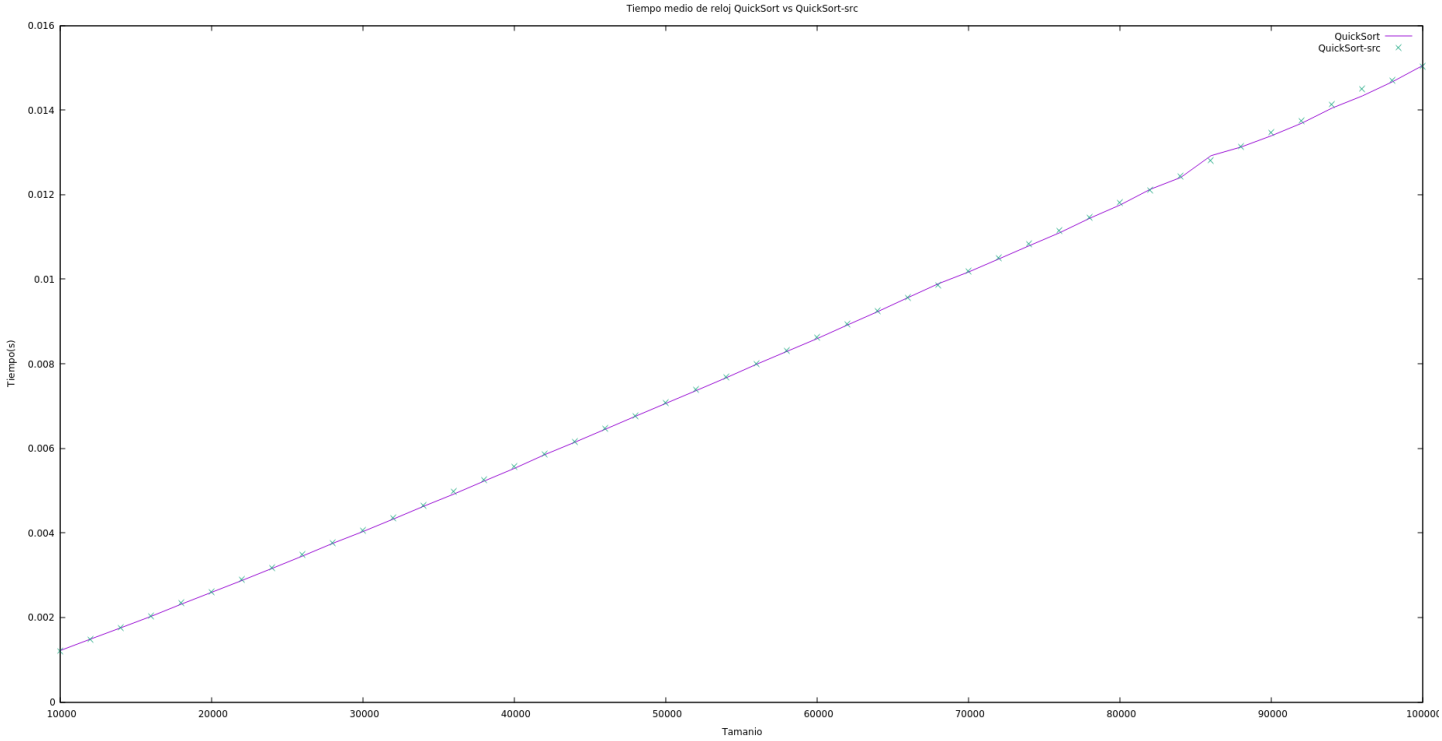
Como podemos observar en esta gráfica, en la que se muestra el tiempo medio que tarda en ordenarse una permutación con el tamaño indicado el eje x, el crecimiento de la recta es de orden loglineal, ya que hemos conseguido ajustar una función loglineal (representada con aspas), que es casi idéntica a la recta obtenida mediante valores experimentales (es decir, con la ejecución del programa). Comprobamos que como habíamos analizado matemáticamente en la asignatura de teoría, el crecimiento del tiempo y de la entrada para el algoritmo QuickSort es loglineal.

5.5 Apartado 5

```
e378187@12-28-66-197:~/Andres_Mena_Juan_Moreno$ ./ejercicio4_quick_src -tamanio 10
Practica numero 2, apartado 5.1
Realizada por: Andrés Mena y Juan Moreno
Grupo: 9(1272)
1       2       3       4       5       6       7       8       9       10
```

Comprobación de que **quicksort_src** ordena correctamente: Como podemos comprobar, el algoritmo Quicksort (eliminando la recursión de cola) se ha implementado correctamente ya que ordena de forma ascendente una tabla de tamaño 10.

Gráfica con el tiempo medio de reloj comparando la rutina quicksort y quicksort_src:



En esta gráfica vemos que no hay ninguna diferencia notable entre el Quicksort usando recursión de cola y el Quicksort-src sin recursión de cola en cuanto a tiempo medios respecto del tamaño de las tablas de entrada respecta. Sin embargo, al implementar el Quicksort_src estamos ahorrando posibles problemas de llenar la pila del sistema, por lo que concluimos que el código del Quicksort_src es una mejor forma de implementar el algoritmo QuickSort que la primera versión.

5. Respuesta a las preguntas teóricas.

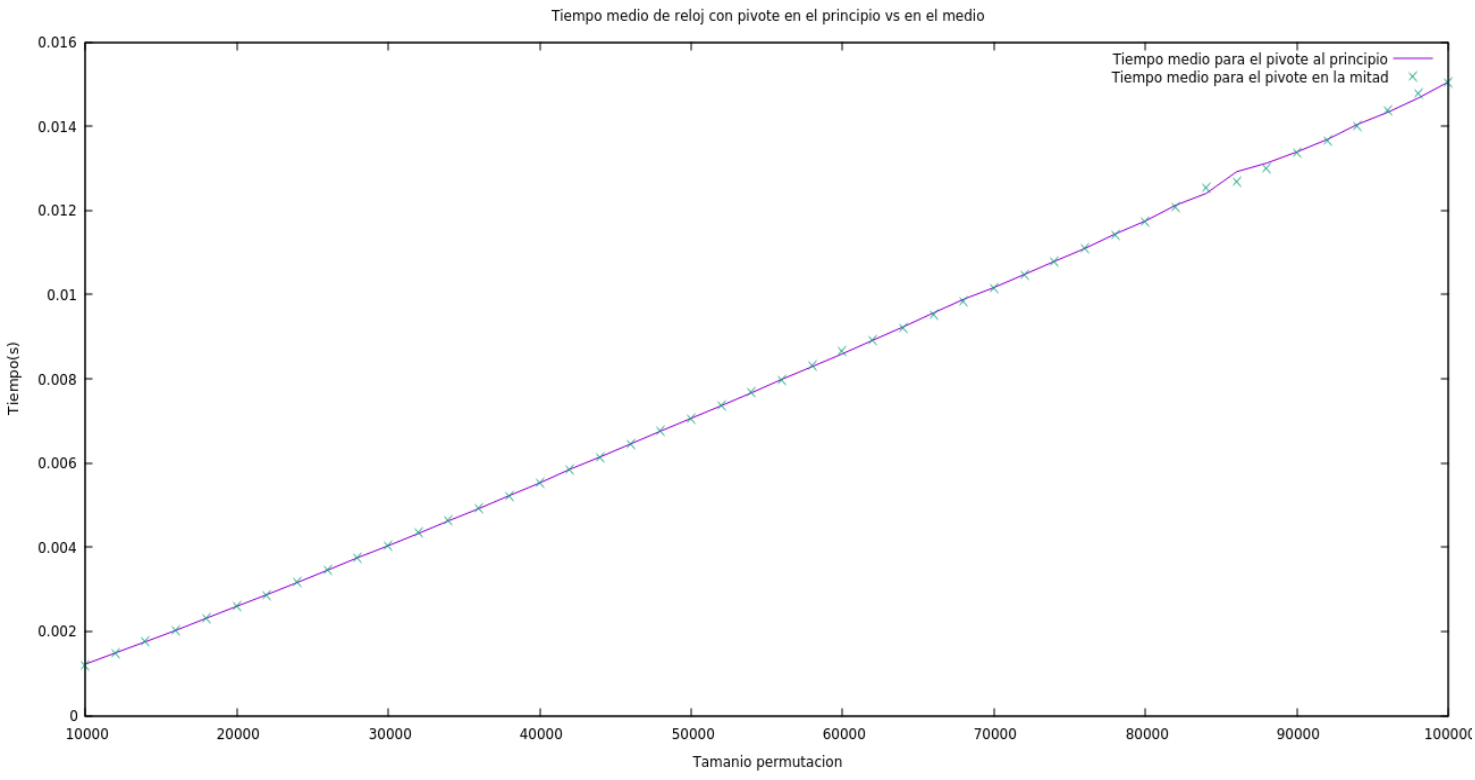
5.1 Pregunta 1

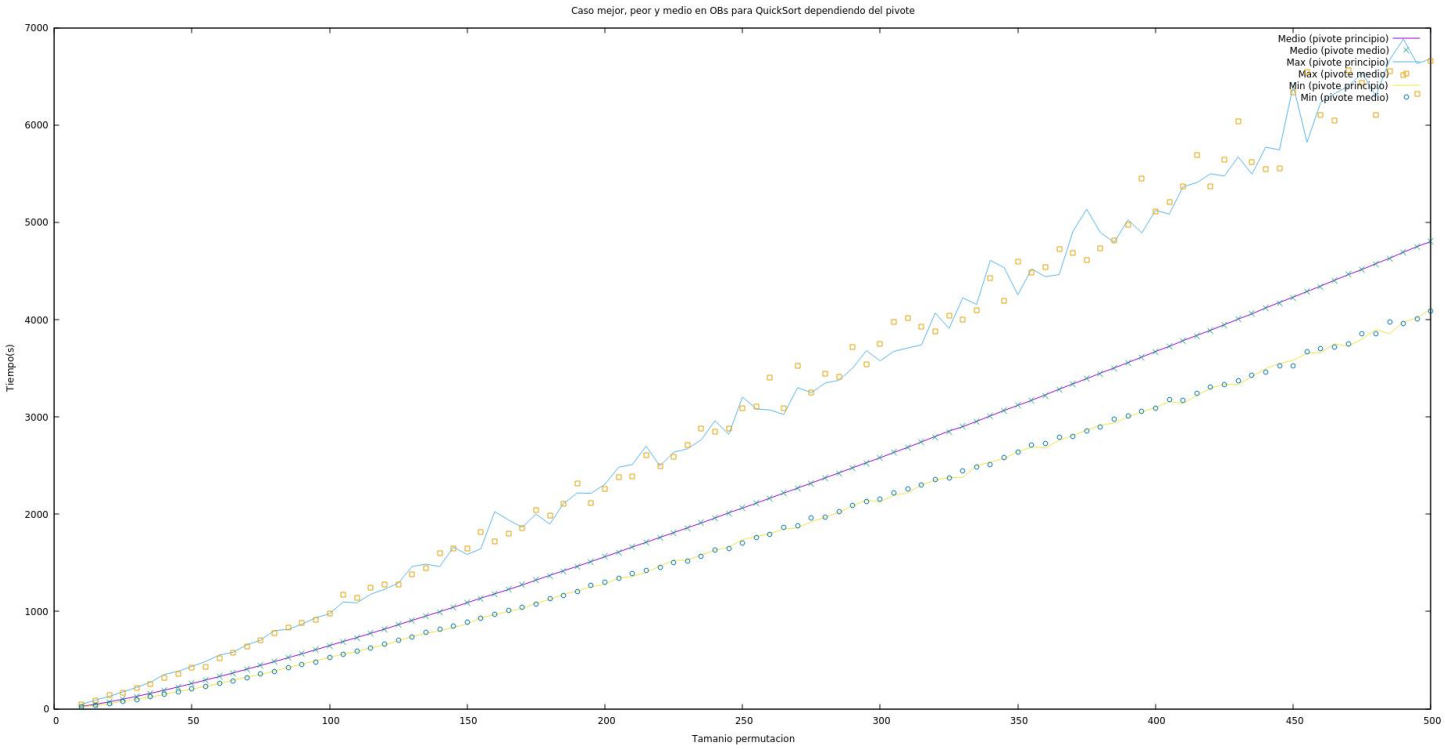
En el apartado de gráficas 5.2, podemos observar dos gráficas que comparan el rendimiento (en función del tiempo de ordenación medio de las tablas, y en función de las operaciones básicas en otra gráfica) empírico con el teórico para el algoritmo **MergeSort**. Tanto en la gráfica en función del tiempo, como en la gráfica de las operaciones básicas, se comprueba (gracias al ajuste de una función loglineal usando ciertos parámetros calculados por el programa GNUPLOT matemáticamente) que existe una función loglineal casi idéntica tanto en la representación respecto al tiempo, como en la representación respecto a las operaciones básicas, lo que quiere decir que lo hemos implementado de forma correcta. Para comparar los distintos casos del algoritmo QuickSort hemos creado una gráfica (apartado 5.4) la cual muestra sus rendimientos empíricos y teóricos. Comprobamos como el caso peor de QuickSort es de orden cuadrático al realizarse la recta que se muestra, y los casos medio y mejor son de orden loglineal como hemos estudiado en teoría.

En los rendimientos en los que apreciamos picos pronunciados o excesivos, es debido a que el ordenador está ejecutando operaciones en segundo plano y por lo tanto puede producir irregularidades.

5.2 Pregunta 2

Para comprobar si el pivote es un elemento que influye en el rendimiento del Quicksort, hemos modificado la elección del pivote original (que era el primer elemento de la tabla), a la elección de un pivote en el centro de la tabla. Una vez cambiado esto, hemos ejecutado de nuevo la rutina Quicksort con los mismos parámetros de entrada, y a continuación se muestran las gráficas comparando el rendimiento del mismo algoritmo, pero usando un pivote distinto, representando tanto las operaciones básicas como el tiempo medio que lleva la ordenación de la tabla.





Como podemos observar en las dos gráficas, no se obtienen diferencias apreciables, y esto es porque la elección del pivote para permutaciones completamente aleatorias no modifica el rendimiento del algoritmo. Esto lo podemos deducir tanto a partir de la primera gráfica (las aspas que representan el pivote en el medio coinciden en su mayor parte con la línea que representa el pivote en el principio) como en la segunda gráfica (vemos que los símbolos representados coinciden en su mayor parte con líneas).

5.3 Pregunta 3

Respecto al MergeSort, sus rendimientos son los siguientes: Wms(N)≤ N*logN + O(N) || Bms(N) $\geq \frac{N}{2}$ * log(N) por lo que $\frac{N}{2}$ * log(N) ≤ Bms(N) ≤ Ams(N) ≤ Wms(N) ≤ N*logN + O(N). A partir de estos resultados se puede extrapolar que el rendimiento del MergeSort para cualquier caso va a ser O(n*logn), una ecuación loglineal.

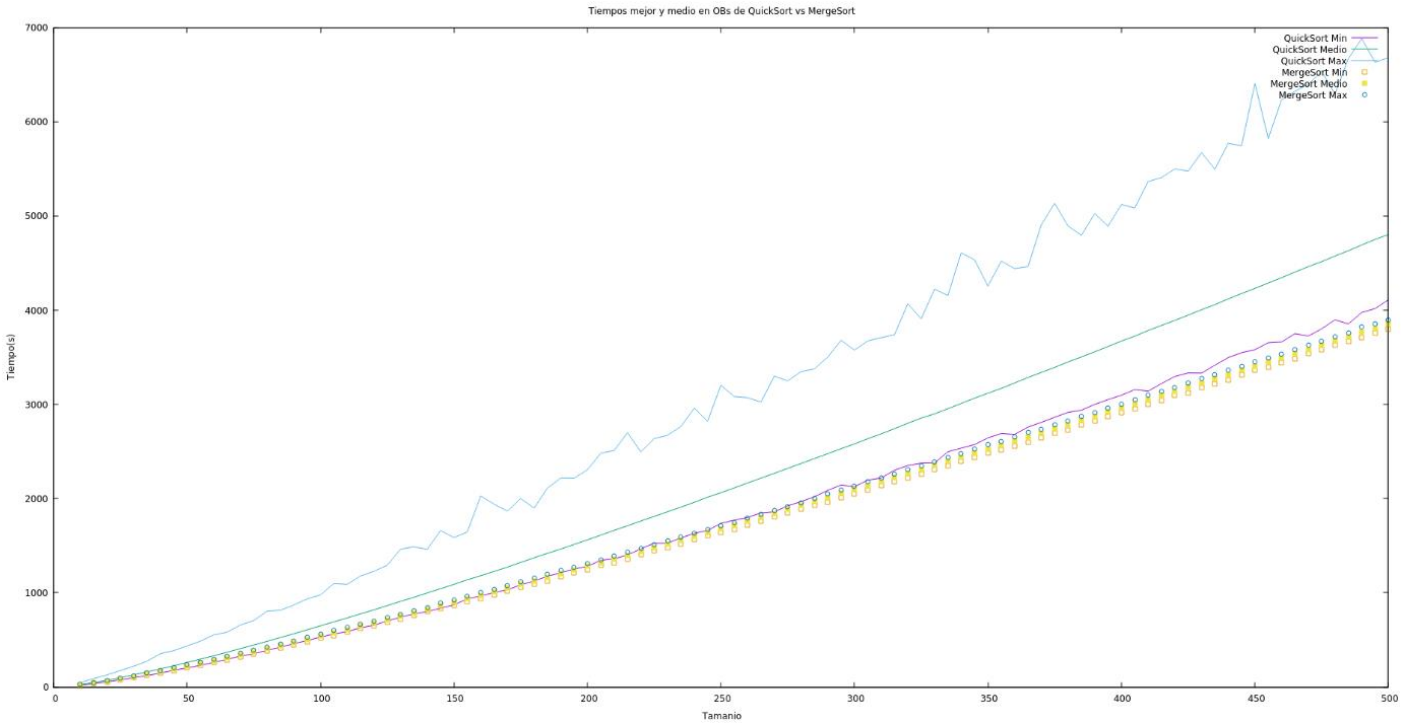
El QuickSort sin embargo tiene los siguientes rendimientos: Wqs(N) ≥ $\frac{N\text{^}2}{2}$ - $\frac{N}{2}$ || Aqs(N)=2*logN+O(N) y por lo tanto, el caso peor del QuickSort es de orden O(n^2) y el caso medio y mejor de O(n*logn).

Para calcular estrictamente cada uno de los casos, lo que habría que hacer es introducirle situaciones “extremas”, es decir, situaciones poco reales pero que nos van a dar eso hipotético rendimiento mínimo, máximo y medio. Para el caso del Quicksort, el peor rendimiento se obtendría con una tabla ordenada, y con una tabla completamente desordenada o justo ordenada a medias (es decir, el elemento mas grande opuesto con el más pequeño), se obtendría el resto de los rendimientos extremos (mejor y medio). Lo mismo pasa con el MergeSort, habría que mandar las posibilidades extremas (tabla completamente ordenada, completamente desordenada, u ordenada a medias como previamente he mencionado) y comprobar la respuesta que nos da el algoritmo de ordenación.

Para probar esto en la práctica, habría que modificar el generador de permutaciones para que devolviese una permutación con las características previamente mencionadas.

5.4 Pregunta 4

Para comparar que algoritmo es mas eficiente de forma experimental, hemos creado una gráfica en la que representamos las operaciones básicas realizadas en caso mejor, peor y medio a la hora de ordenar una tabla, tanto para el MergeSort como para el QuickSort:



Como podemos observar, el algoritmo Mergesort realiza menos comparaciones en todos los casos, incluso el peor caso del MergeSort es mejor que el mejor caso del QuickSort, por lo que concluimos que el MergeSort es un mejor sistema de ordenación.

En cuanto al tema de la gestión de memoria, hay que tener en cuenta varios factores. Por una parte, el algoritmo MergeSort tiene que duplicar la memoria para almacenar todo en una tabla auxiliar, lo que puede suponer un problema y es una gran desventaja, sobre todo cuando estamos tratando con grandes cantidades de datos que nos podemos quedar sin memoria.
Sin embargo, el QuickSort tiene un problema igual de grave, y es que, al tratarse de un algoritmo de recursión, puede sobrecargar la pila del sistema.
Por estos motivos, ninguno de los dos algoritmos sería óptimo, sin embargo, la recursión de cola del QuickSort se puede eliminar, (es lo que hemos hecho en el QuickSort_src) lo que hace que nuestra rutina QuickSort_src se convierta en la más eficiente en términos de memoria, ya que en vez de ocupar nuevas posiciones de pila cada vez que llamamos a la función, lo que se hace es pisar todo el rato la misma posición de memoria, dejando únicamente la información que nos interesa de verdad.

5.5 Pregunta 5

Como podemos observar en la gráfica del apartado 5.5, no hay ninguna diferencia destacable entre el rendimiento del quicksort y el quicksort sin recursión de cola. Pero si tendríamos que elegir uno, la mejor opción sería quitando la recursión de cola ya que al eliminar las llamadas recursivas, no se sobrecarga la pila y solo hay que ocupar una posición de la pila la cual se va "pisando" con información nueva que es la que queremos.

6. Conclusiones finales.

En esta practica hemos examinado el rendimiento de dos nuevos algoritmos; El MergeSort y el QuickSort. En la practica 1, habíamos estudiado cual era el rendimiento del SelectSort, y habíamos llegado a la conclusión de que siempre era de orden cuadrático. Sin embargo, con está practica hemos demostrado que estos algoritmos son mejores, ya que el MergeSort tiene un rendimiento loglineal para su caso mejor, peor y medio, y el QuickSort un rendimiento cuadrático para su caso peor (la tabla ordenada), y un rendimiento loglineal para el resto de los casos.

A la hora de implementar nuevos algoritmos, no solo hay que tener en cuenta como de rápido ordenan o cual es el número de operaciones básicas que realizan, sino que también hay que tener en cuenta otros factores como por ejemplo qué uso hacen de la memoria, porque podría ser que estos algoritmos no fuesen óptimos para la cantidad de datos que queremos ordenar, o que incluso no pudiesen funcionar en nuestros dispositivos.

Gracias a esta práctica, hemos podido verificar que los valores matemáticos que obteníamos de los rendimientos de los algoritmos efectivamente se correspondían con los valores experimentales, y esto lo hemos podido realizar mediante el uso de gráficas.