# Análisis de Algoritmos 2018/2019

Práctica 2

Elena Cano

Rubén García

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

## 1. Introducción.

El objetivo de esta práctica consiste en estudiar los algoritmos que utilizan la metodología divide y vencerás . En este caso estudiaremos Quick Sort y Merge Sort para después hallar sus tiempos de ejecución sobre tablas de distintos tamaños y comparar los resultados con los estudiados teóricamente.

## 2. Objetivos

### 2.1 Apartado 1

Consiste en implementar el algoritmo de ordenación Merge Sort mediante dos funciones, una será la principal a la cual llamaremos MergeSort y tendremos otra la cual será merge que será llamada por la anterior.

### 2.2 Apartado 2

Consiste en generar un documento en el cual tenemos que reflejar el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación, ordenándolas con el algoritmo MergeSort.

### 2.3 Apartado 3

En este caso implementaremos el algoritmo de ordenación Quick Sort mediante tres funciones: una será la principal a la cual llamaremos QuickSort, también utilizaremos medio y partir las cuales serán imprescindibles para la implementación de Quick Sort.

### 2.4 Apartado 4

El objetivo es generar un documento en el cual tenemos que reflejar el tiempo promedio de ejecución, máximo, mínimo y promedio de operaciones básicas en función del tamaño de la permutación, ordenándolas con el algoritmo QuickSort.

### 2.5 Apartado 5

El prospósito de este apartado es implementar dos nuevas funciones que serán medio_avg() y medio_stat() que lo que harán cada una es devolver distintos valores para pivote.

## 3. Herramientas y metodología

Para llevar acabo esta práctica hemos utilizado el entorno de Linux, usando atom para escribir el código y editarlo. Sin embargo debido a los fallos que hemos ido cometiendo hemos necesitado utilizar un depurador, en este caso hemos elegido el de Net Beans. Una vez teníamos el código depurado y funcionando correctamente pasamos Valgrind para detectar así las perdidas de memoria y corregirlas.

### 3.1 Apartado 1

Mergesort la vamos a implementar como una función recursiva que va dividiendo cada tabla por la mitad y contando las OBS. Al salir de la llamada recursiva une las tablas en  que habíamos roto por la mitad anteriormente y las une ya en orden. Para unirlas de forma ordenada llamamos a la función Merge que es la que se va a encargar de comprara los elementos e ir reordenándolos y a su vez contado las OBS que realiza.

### 3.2 Apartado 2

Para implementar este apartado lo único que tenemos que cambiar en nuestro código es una linea del ejercicio5.c. Dentro del programa buscamos donde llama a la función genera_tiempos_ordenación y le pasamos como argumento el algoritmo con el cual queremos que ordene la tabla, en este caso llamará a MergeSort. Además se generará un fichero que se llamará “ejercicio5.log” para así ver reflejados los datos requeridos.

### 3.3 Apartado 3

Quicksort la vamos a implementar como una función recursiva que va dividiendo cada tabla mediante llamadas a la función partir. La función partir va a llamar a su vez a la función medio, la cual igualará la dirección del argumento ‘pos’ a la dirección de la primera posición de la tabla. Una vez implementada la función vamos a ejercicio4.c y  llamamos a QuickSort, así comprobamos que nuestro algoritmo está bien implementado y ordena la tabla correctamente.

3.4 Apartado 4

Para implementar este apartado lo único que tenemos que cambiar en nuestro código es una linea del ejercicio5.c. Dentro del programa buscamos donde llama a la función genera_tiempos_ordenación y le pasamos como argumento el algoritmo con el cual queremos que ordene la tabla, en este caso llamará a QuickSort. Además se generará un fichero que se llamará “ejercicio5.log” para así ver reflejados los datos requeridos.

3.5 Apartado 5

En este caso medio_avg nos devolverá como pivote el elemento que se encuentre en medio de la tabla mientras medio_stat compara los valores de las posiciones, primera, última y la del medio de la tabla y devuelve como pivote la posición que contenga el valor intermedio entre las tres. Modificaremos la rutina partir para que pueda usar estas funciones de elección de pivote y para cada una de ellas ejecutaremos el ejercicio5.c para ver sus tiempos de ejecución que estarán reflejados en el fichero que se crea al ejecutar el programa.

## 4. Código fuente

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

Como aclaración indicamos que en el código hemos escrito primero las funciones que no dependen de otras, por lo que MergeSort y QuickSort son las últimas funciones en aparecer.

4.1 Apartado 1

```
int merge(int* tabla, int ip, int iu, int im) {
  int OBS=0;
  int i, j, k;
  int* tablaAux=NULL;

  if (tabla==NULL || ip>iu)
    return ERR;
  tablaAux=malloc((iu-ip+1)*sizeof(int));
  if (tablaAux==NULL)
    return ERR;
  i=ip;
  j=im+1;
  k=0;
```

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

  if (i>im) {
    while (j<=iu) {
      tablaAux[k]=tabla[j];
      j++;
      k++;
    }
  }
  else if (j>iu) {
    while (i<=im) {
      tablaAux[k]=tabla[i];
      i++;
      k++;
    }
  }
  for(i=ip; i<=iu; i++) {
    tabla[i]=tablaAux[i-ip];
  }
  free(tablaAux);
  return OBS;
}

int MergeSort(int* tabla, int ip, int iu) {
  int im, flag, OBS=0;
  if (tabla==NULL || ip>iu)
    return ERR;
  if (ip==iu)
    return OBS;
  im=(ip+iu)/2;
  flag=MergeSort(tabla, ip, im);
```

```
  if (flag==ERR)
    return ERR;
  OBS+=flag;
  flag=MergeSort(tabla, im+1, iu);
  if (flag==ERR)
    return ERR;
  OBS+=flag;
  flag=merge(tabla, ip, iu, im);
  if (flag==ERR)
    return ERR;
  OBS+=flag;
  return OBS;
}
```

### 4.3 Apartado 3

```
int medio(int* tabla, int ip, int iu, int* pos) {
  int OBS=0;
  if (tabla==NULL || ip<0 || ip>iu || !pos)
    return ERR;
  (*pos)=ip;
  return OBS;
}

int partir(int* tabla, int ip, int iu, int* pos) {
  int OBS=0;
  int i, aux, swap;
  if (tabla==NULL || ip<0 || ip>iu || !pos)
    return ERR;
  OBS=medio(tabla, ip, iu, pos);
  if (OBS==ERR)
    return ERR;
  aux=tabla[*pos];
  tabla[*pos]=tabla[ip];
  tabla[ip]=aux;
  *pos=ip;
  for (i=ip+1; i<=iu; i++) {
    if (tabla[i]<aux) {
      (*pos)++;
      swap=tabla[i];
      tabla[i]=tabla[*pos];
      tabla[*pos]=swap;
    }
```

```
    OBS++;
  }
  swap=tabla[ip];
  tabla[ip]=tabla[*pos];
  tabla[*pos]=swap;
  return OBS;
}

int QuickSort(int* tabla, int ip, int iu) {
  int im, flag, OBS=0;
  if (tabla==NULL || ip<0 || ip>iu)
    return ERR;
  if (ip==iu)
    return OBS;
  else {
    flag=partir(tabla, ip, iu, &im);
    if (flag==ERR)
      return ERR;
    OBS+=flag;
    if (ip<im-1) {
      flag=QuickSort(tabla, ip, im-1);
      if (flag==ERR)
        return ERR;
      OBS+=flag;
    }
    if (im+1<iu) {
      flag=QuickSort(tabla, im+1, iu);
      if (flag==ERR)
        return ERR;
      OBS+=flag;
    }
  }
  return OBS;
}
```

### 4.5 Apartado 5

```
int medio_avg(int* tabla, int ip, int iu, int* pos) {
  int OBS=0;
  if (tabla==NULL || ip<0 || ip>iu || !pos)
    return ERR;
  (*pos)=(ip+iu)/2;
  return OBS;
```

```
}

int medio_stat(int* tabla, int ip, int iu, int* pos) {
  int im, OBS=0;
  if (tabla==NULL || ip<0 || ip>iu || !pos)
    return ERR;
  im=(ip+iu)/2;
  if (tabla[ip]<tabla[iu]) {
    if (tabla[ip]<tabla[im]) {
      if (tabla[im]<tabla[iu]) {
        (*pos)=im;
      }
      else {
        (*pos)=iu;
      }
      OBS++;
    }
    else {
      (*pos)=ip;
    }
    OBS++;
  }
  else {
    if (tabla[iu]<tabla[im]) {
      if (tabla[ip]<tabla[im]) {
        (*pos)=ip;
      }
      else {
        (*pos)=im;
      }
      OBS++;
    }
    else {
      (*pos)=iu;
    }
    OBS++;
  }
  OBS++;
  return OBS;
}
```

## 5. Resultados, Gráficas

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

## 5.1 Apartado 1

En el apartado 1 cambiamos la variable ret (return) en el código de ejercicio4.c, que ahora en vez de ejecutar el algoritmo de ordenación InsertSort, ejecuta el algoritmo recientemente implementado MergeSort. Para ello pasamos una tabla de N elementos que quedan desordenados (a través de los códigos implementados en la práctica anterior) y lo pasamos con MergeSort, de tal manera que queden ordenadas una vez ejecutado el código. El resultado para una tabla de 100 elementos es el siguiente:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

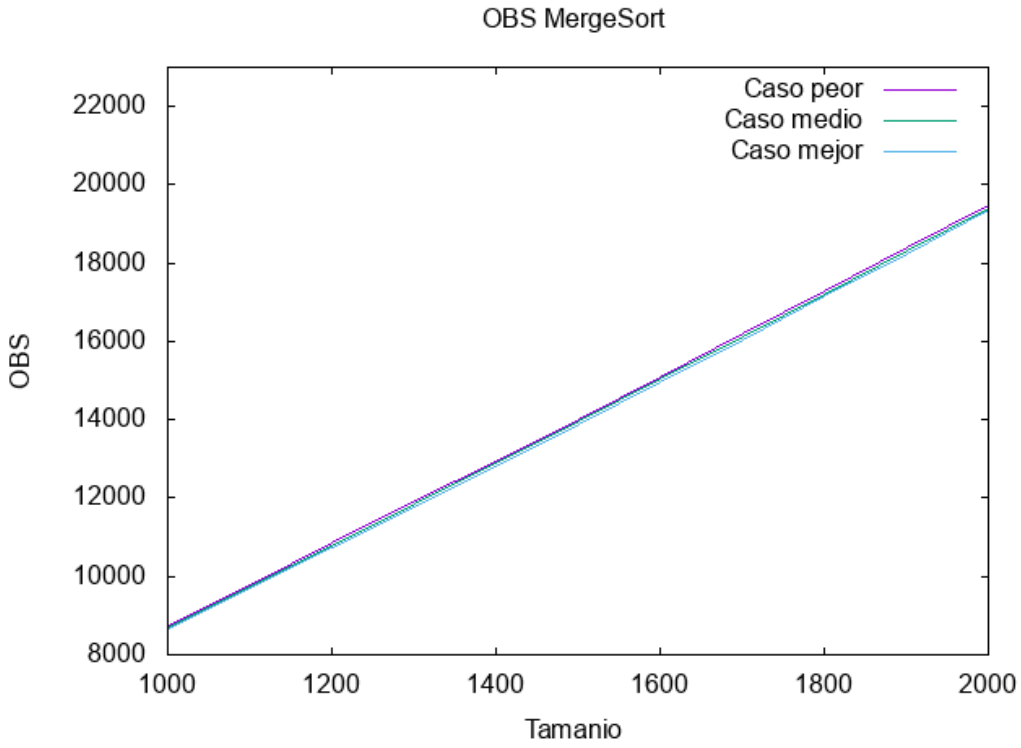
El resultado es una tabla de números ordenados del 1 al 100, lo que era de esperar pues la tabla queda ordenada tras haber ejecutado el código implementado en MergeSort, lo que implica que el código implementado es correcto.

## 5.2 Apartado 2

Para el apartado 2 modificamos el código de ejercicio5.c, siendo ahora el algoritmo MergeSort del cual sacaremos los tiempos y las OBS. Realizamos una tabla que contiene los siguientes columnas de datos respectivamente:

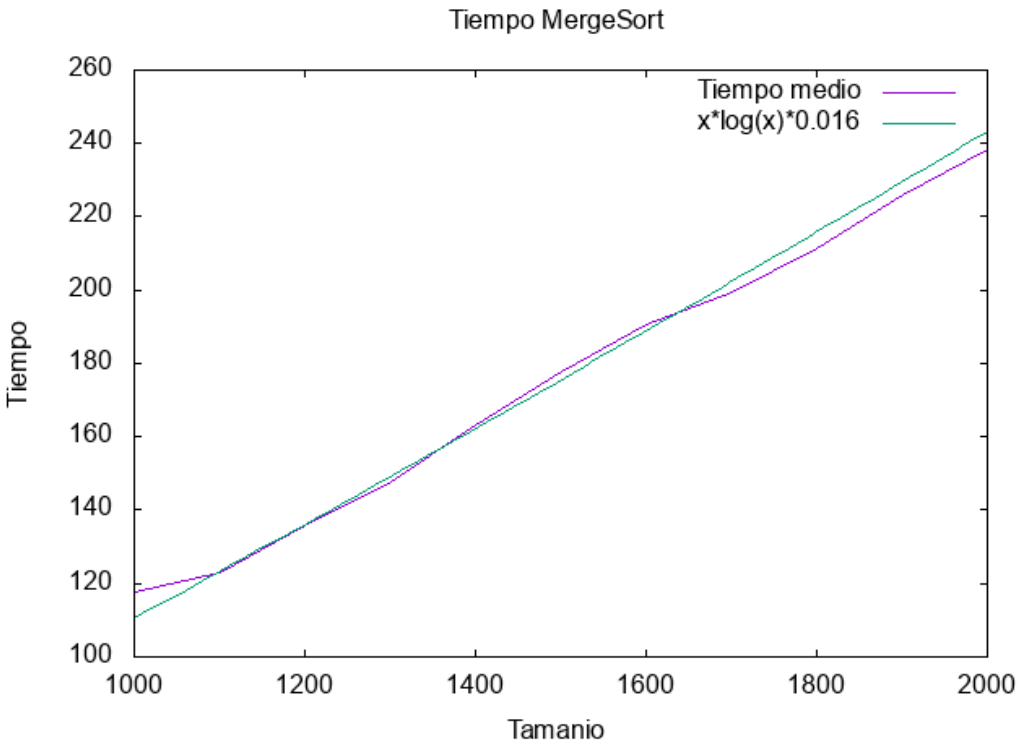
- Columna con tablas que contienen de 1000 a 2000 elementos con incrementos de 100 elementos entre tabla y tabla.
- Columna con los tiempos medios de ejecución de cada una de las tablas.
- Columna con las medias de las operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las mínimas operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las máximas operaciones básicas (OBs) que se ejecutan para cada tabla.

Usando gnuplot, realizamos una gŕafica comparando los tiempos mejor, peor y medio en OBs para MergeSort, quedando de la siguiente manera:




Se puede apreciar como en el algoritmo de ordenación MergeSort las OBS son muy parecidas para todos los casos (peor, medio y mejor). Aunque se puede ver como el caso peor está ligeramente por encima y el mejor ligeramente por debajo, esto concuerda con los resultados teóricos pues para los 3 casos el rendimiento es O(N*logN).

Realizamos una gráfica con el tiempo medio de reloj para MergeSort:




El tiempo medio de MergeSort describe una gráfica de la forma x*logx, por lo que el tiempo medio es O(n*logn). Podemos ver como el tiempo medio de MergeSort se ajusta en concreto a la gráfica x*log(x)*0,016.

5.3 Apartado 3

En el apartado 1 cambiamos la variable ret (return) en el código de ejercicio4.c, que ahora en vez de ejecutar el algoritmo de ordenación MergeSort, ejecuta el algoritmo que acabamos de implementar QuickSort, con la elección del pivote *medio*. Para ello pasamos una tabla de N elementos que quedan desordenados (a través de los códigos implementados en la práctica anterior) y lo pasamos con QuickSort, de tal manera que queden ordenadas una vez ejecutado el código. El resultado para una tabla de 100 elementos es el siguiente:

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| 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 | 80 |
| 81 | 82 | 83 | 84 | 85 | 86 | 87 | 88 | 89 | 90 |
| 91 | 92 | 93 | 94 | 95 | 96 | 97 | 98 | 99 | 100 |

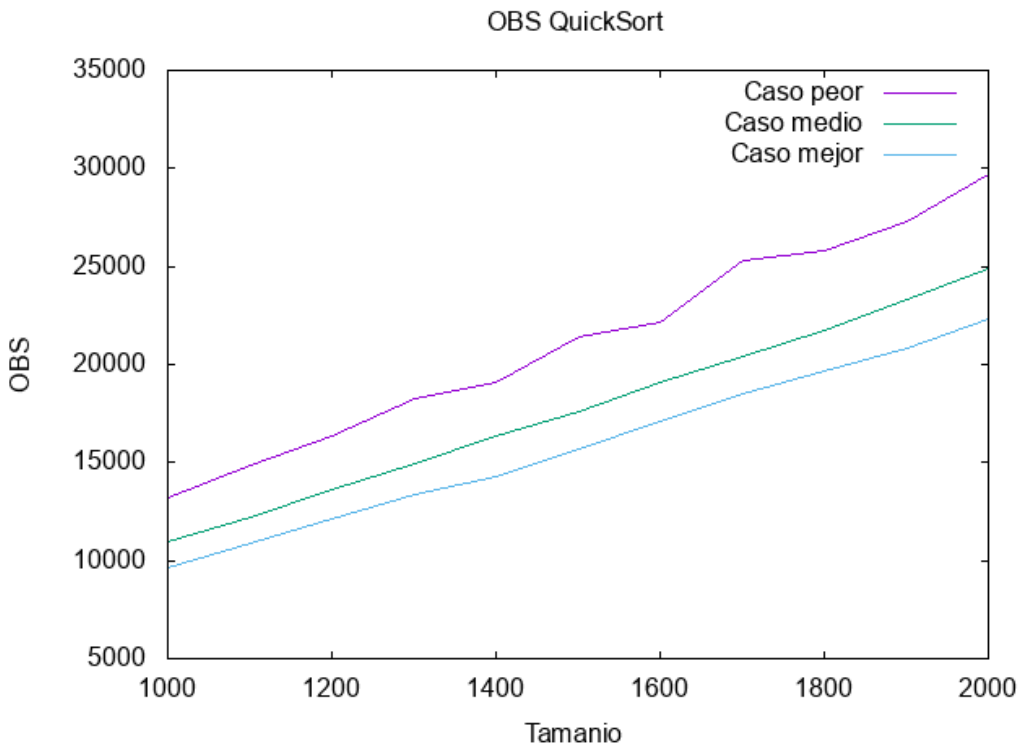
El resultado es una tabla de números ordenados del 1 al 100, resultado esperado pues la tabla queda ordenada tras haber ejecutado el código implementado en QuickSort, lo que implica que el código implementado es correcto.

## 5.4 Apartado 4

Para el apartado 2 modificamos el código de ejercicio5.c, siendo ahora el algoritmo QuickSort con pivote *medio* del cual sacaremos los tiempos y las OBS. Realizamos una tabla que contiene los siguientes columnas de datos respectivamente:

- Columna con tablas que contienen de 1000 a 2000 elementos con incrementos de 100 elementos entre tabla y tabla.
- Columna con los tiempos medios de ejecución de cada una de las tablas.
- Columna con las medias de las operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las mínimas operaciones básicas (OBs) que se ejecutan para cada tabla.
- Columna con las máximas operaciones básicas (OBs) que se ejecutan para cada tabla.

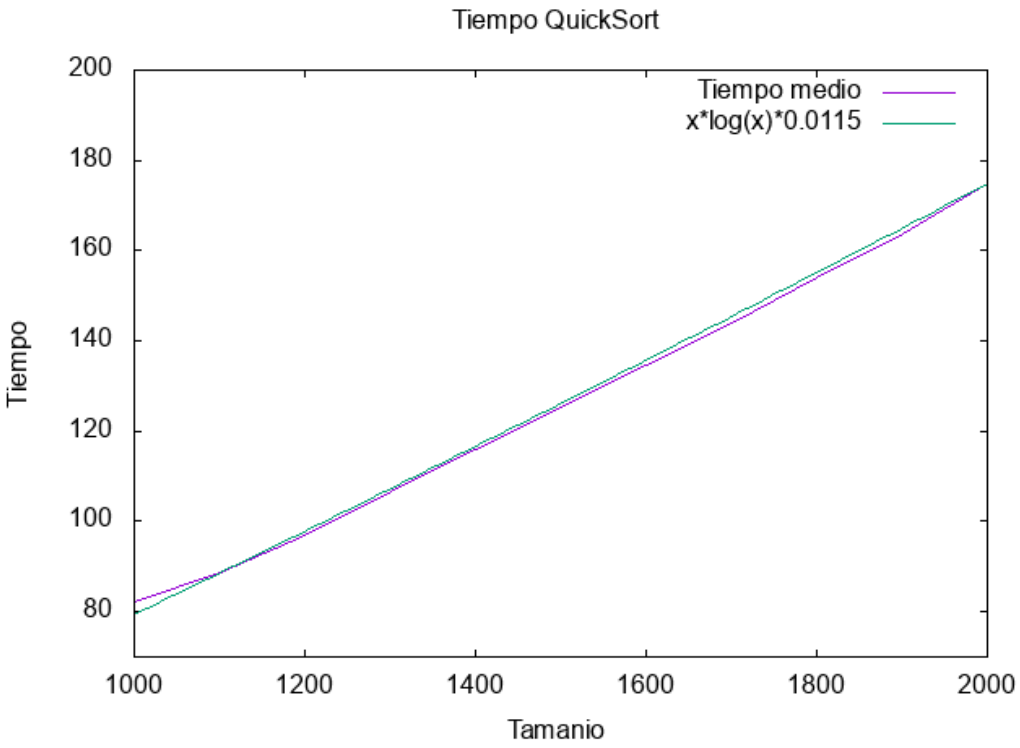
Usando gnuplot, realizamos una gŕafica comparando los tiempos mejor, peor y medio en OBs para QuickSort, quedando de la siguiente manera:




Se puede apreciar como en el algoritmo de ordenación QuickSort las OBS están muy bien diferenciadas en cada uno de los casos, siendo el caso peor el que destaca muy por encima del caso medio y mejor con respecto al número de

operaciones básicas. Esto concuerda con los resultados teóricos pues los rendimientos para QuickSort son $O(N^2)$ para el caso peor, y O(N*logN) para el caso medio y mejor.

Realizamos una gráfica con el tiempo medio de reloj para QuickSort:

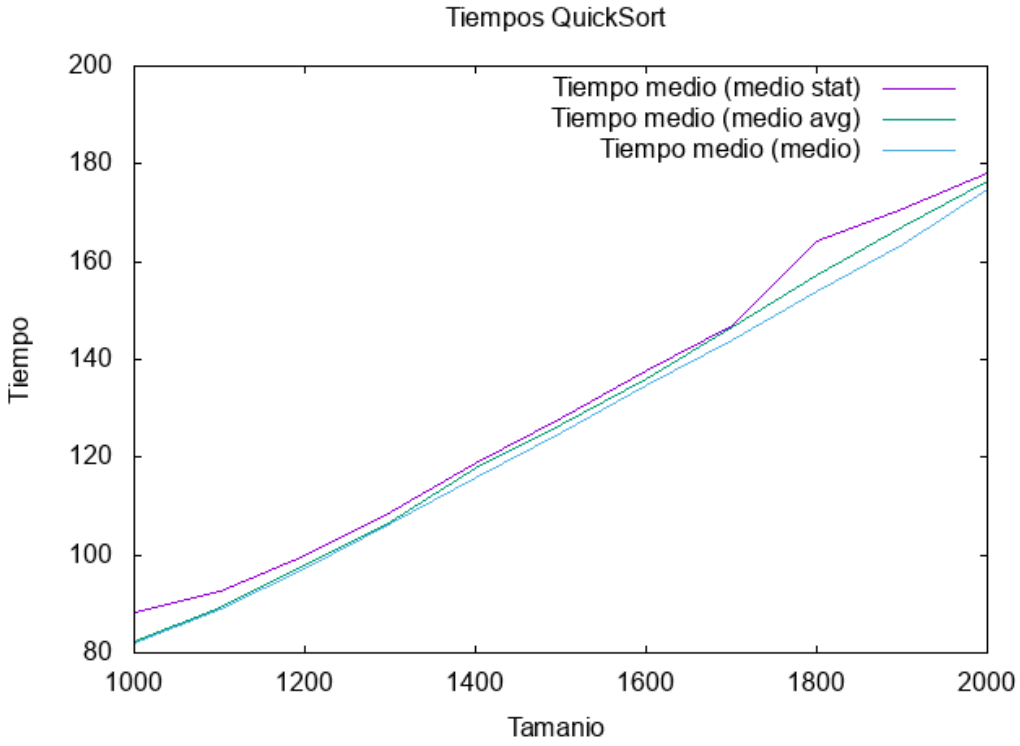



El tiempo medio de QuickSort describe una gráfica de la forma x*logx, por lo que el tiempo medio es O(n*logn). Podemos ver como el tiempo medio de QuickSort se ajusta en concreto a la gráfica x*log(x)*0,0115.

## 5.5 Apartado 5

Implementamos otros dos pivotes a parte del que ya tenemos, medio, que devuelve como pivote el primer elemento de la tabla que se pasa como argumento. Estos dos pivotes desarrollados son medio_avg, que devuelve como pivote el elemento que se encuentra en la posición del medio para cada tabla; y medio_stat, que compara los valores del primer, el último, y elemento del medio de la tabla y devuelve como pivote la posición en la cual se encuentra el valor medio de esos tres valores.

A continuación sustituimos cada uno de los pivotes desarrollados en el algoritmo QuickSort y desarrollamos una gráfica con el tiempo medio de reloj comparando los tres pivotes desarrollados:




Se puede apreciar como el tiempo de ejecución es muy parecido para cada uno de los tres pivotes. Sin embargo, medio_stat tarda algo más que medio_avg y medio en ejecutarse, esto es debido a que el código de selección de medio_stat es más complejo que el de medio_avg y medio respectivamente. Al mismo tiempo, medio_avg tarda más que medio por la misma razón el código es algo más complejo que medio.

## 6. Respuesta a las preguntas teóricas.

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

6.1 Pregunta 1

En MergeSort, tanto el caso peor, el caso medio y el caso mejor son del orden O(N*logN), por tanto el caso medio de MergeSort es O(N*logN), esto concuerda con los datos empíricos obtenidos pues en todos los casos (mejor, medio y peor) las OBS son muy parecidas y están cercanas (como se puede apreciar en la gráfica) precisamente porque el rendimiento en los tres casos es muy parecido.

En QuickSort, el caso medio y el caso mejor son del orden O(N*logN), mientras que el caso peor es del orden de $O(N^2)$. El caso medio de QuickSort es por tanto O(N*logN), se puede apreciar como los datos teóricos y empíricos concuerdan de manera muy aproximada pues se puede comprobar que el caso medio y el caso mejor siguen la misma tendencia gráfica en cuanto a la comparación de claves, mientras que el caso peor aumenta de manera más notable que las otras dos, concordando con el rendimiento esperado.

## 6.2 Pregunta 2

En el apartado anterior hemos generado una gráfica en la que podemos observar los tiempos medio de ejecución de QuickSort con lo diferentes pivotes. Se puede apreciar que el tiempo que tarda en ordenarse la tabla para cada uno de los tres pivotes es muy parecido.

Sin embargo, cuando QuickSort usa la función medio_stat para decidir el pivote tarda algo más en ordenar la tabla que si usara medio_avg y medio. Esta diferencia se debe principalmente a que el código de selección de medio_stat es más complejo que el de medio_avg y medio respectivamente.

De la misma forma podemos observar que el tiempo medio de ejecución al emplear la función de medio_avg es mayor que si usáramos medio. De nuevo está diferencia se debe a que el código usado para implementar medio_avg es más complejo que el de la función medio.

## 6.3 Pregunta 3

En MergeSort, todos los casos son del orden O(N*logN), por tanto el caso peor y mejor de MergeSort es O(N*logN), cosa que se puede apreciar en la gráfica pues todas las OBS son muy parecidas en los tres casos.

En QuickSort, el caso mejor son del orden O(N*logN), mientras que el caso peor es del orden de $O(N^2)$, en concreto $N^2/2-N/2$. Se puede ver como el caso peor aumenta de manera más rápida (gráficamente) que el caso peor y mejor.

Para calcular estrictamente cada uno de los casos habría que añadir más permutaciones, con pocas permutaciones los resultados teóricos obtenidos son poco aproximados con respecto al cálculo teórico. Según se van añadiendo más permutaciones los casos mejor, medio y peor se van aproximando cada vez más. Esto se debe a que, dada una tabla de n elementos, el número de permutaciones posibles es n!, por lo que la probabilidad de que se ejecute la permutación del caso mejor y el caso peor es de 1/n!, y por lo tanto el cálculo del caso medio (que es la media entre las OBS obtenidas en cada una de las permutaciones ejecutadas) es también impreciso.

## 6.4 Pregunta 4

El algoritmo de QuickSort es algo más eficiente que MergeSort, aunque el tiempo de ambos algoritmos sea O(n*logn), se puede apreciar que QuickSort tiene como escalar 0,0115 multiplicando al resultado teórico mientras que MergeSort tiene como escalar 0,016, siendo el primero (QuickSort) más pequeño y, por lo tanto, más eficiente.

El algoritmo QuickSort es más eficiente que MergeSort en cuanto gestión de memoria pues MergeSort necesita arrays auxiliares para cada una de las particiones mientras que QuickSort requiere espacio adicional.

## 7. Conclusiones finales.

A la hora de realizar la práctica no entendíamos muy bien como funcionaban los algoritmos de MergeSort e InsertSort, sin embargo al tener que implementarlos nos ha dado tiempo a razonar y acabar comprendiendo el funcionamiento de estos.

Hemos comprobado empíricamente que el caso medio, mejor y peor de ejecución de MergeSort es el mismo para cada uno de los casos ya que las gráficas obtenidas eran prácticamente iguales, siendo el rendimiento de O(N*logN).

Al implementar QuickSort hemos visto que si usamos distintas funciones para devolver la posición del pivote los tiempos de ejecución varían, pues dependiendo del valor que devolvamos la búsqueda de este puede ser más compleja, lo que provoca una mayor tardanza a la hora de ejecutarse. Además las OBS no permanecen invariantes, es decir, dependiendo la posición del pivote que queramos devolver usando una función u otra el numero de operaciones básicas que se ejecutarán será distinto, ya que a mayor número de comparaciones para obtener la posición del pivote, mayor numero de OBS.

Cabe concluir que a la hora de ordenar tablas de pocos elementos el algoritmo de QuickSort es más eficiente que MergeSort. Sin embargo según va aumentando el número de elementos de las tablas aumenta la efectividad de MergeSort haciendo que esta sea mejor opción para ordenar grandes tablas de elementos.