Análisis de Algoritmos 2023/2024

Práctica 2

Iker González Sánchez, Grupo 1261

Código	Gráficas	Memoria	Total

1. Introducción.

En esta práctica implementaremos los algoritmos MergeSort y QuickSort, además de crear gráficas con los datos que generen y comparándolos con los que obtenemos teóricamente.

2. Objetivos

2.1 Apartado 1

Realizar la correcta implementación del algoritmo MergeSort y sus correspondientes pruebas.

2.2 Apartado 2

Obtener y representar los datos generados con el algoritmo MergeSort y compararlos con los que obtenemos teóricamente.

2.3 Apartado 3

Realizar la correcta implementación del algoritmo QuickSort y sus correspondientes pruebas.

2.4 Apartado 4

Obtener y representar los datos generados con el algoritmo QuickSort y compararlos con los que obtenemos teóricamente.

2.5 Apartado 5

Modificar la función de partición hecha para QuickSort en el apartado 3 para que sea más flexible y podamos utilizar cualquier función de pivote. Además también hemos creado 2 funciones de pivote (median_avg y median_stat). Por último hemos comparado los datos de estas 2 nuevas funciones con median.

3. Herramientas y metodología

El entorno de desarrollo seleccionado ha sido linux-ubuntu, haciendo uso de VSCode y el terminal.

3.1 Apartado 1

Merge Sort divide recursivamente el arreglo en mitades, ordena cada mitad por separado y luego fusiona las dos mitades ordenadas para obtener un arreglo globalmente ordenado. De esta forma garantiza que el arreglo se ordene eficientemente. Para ello nos hemos basado en el pseudocódigo dado en teoría.

3.2 Apartado 2

Se han guardado los datos utilizando el programa times.c realizado en la clase anterior, para su representación se ha utilizado gnuplot. Después siguiendo la fórmula de MergeSort, O(N*logN), se ha comparado con los resultados obtenidos.

3.3 Apartado 3

QuickSort toma un arreglo tabla, los índices de inicio y fin del subarreglo que se va a ordenar. Si el subarreglo tiene más de un elemento, elige un pivote, organiza el subarreglo alrededor del pivote utilizando la función partition, y luego aplica recursivamente QuickSort a los subarreglos izquierdo y derecho del pivote. Para ello nos hemos basado en el pseudocódigo dado en teoría.

3.4 Apartado 4

Se han guardado los datos utilizando el programa times.c realizado en la clase anterior, para su representación se ha utilizado gnuplot. Después siguiendo la fórmula de QuickSort, O(N*logN), se ha comparado con los resultados obtenidos.

3.5 Apartado 5

Para la realización de este apartado, se han seguido las instrucciones dadas en el enunciado: para median_avg se selecciona la posición media como pivote, proporcionando un enfoque simple pero más predecible que elegir el primer elemento; para median_stat se comparan los valores en las posiciones de los extremos y en la posición media, seleccionando la posición que contiene el valor intermedio entre los tres; por último, para partition(llamado partition2) se toma como argumento una función de pivote, permitiendo la flexibilidad en la elección del pivote.

4. Código fuente

4.1 Apartado 1

```c
int mergesort(int* tabla, int ip, int iu)
{
  if(!tabla || ip < 0 || iu < 0 || ip > iu) return ERR;

  if(ip == iu) return 0;

  int imedio = (ip + iu) / 2;
  int ob1 = mergesort(tabla, ip, imedio);
  if (ob1 == ERR) return ERR;
  int ob2 = mergesort(tabla, imedio + 1, iu);
  if (ob2 == ERR) return ERR;
  int ob3 = merge(tabla, ip, iu, imedio);
  if (ob3 == ERR) return ERR;


  return ob1 + ob2 + ob3;
}


int merge(int* tabla, int ip, int iu, int imedio)
{
  if(!tabla || ip < 0 || iu < 0 || ip > iu) return ERR;

  int* aux = NULL;
  int i = ip, j = imedio + 1, k = 0, ob = 0;

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

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

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

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

  free(aux);

  return ob;
}
```

4.3 Apartado 3

```c
int partition(int* tabla, int ip, int iu, int* pos) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu){
      printf("Error en los parametros de entrada partition\n");
      return ERR;
    }

    int pivot = tabla[ip];
    int i = ip + 1;
    int j;

    for (j = ip + 1; j <= iu; j++) {
        if (tabla[j] < pivot) {
            /* Intercambiar elementos*/
            int temp = tabla[j];
            tabla[j] = tabla[i];
            tabla[i] = temp;
            i++;
        }
    }

    /*Colocar el pivote en su posición final*/
    tabla[ip] = tabla[i - 1];
    tabla[i - 1] = pivot;

    *pos = i - 1;  /*Posición final del pivote*/
    return iu - ip;  /*Número de operaciones básicas*/
}

int median(int* tabla, int ip, int iu, int* pos) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu){
      printf("Error en los parametros de entrada median\n");
      return ERR;
    }

    *pos = ip;
    return 0;
}

/*Función de pivote que devuelve la posición media de la tabla*/
int median_avg(int* tabla, int ip, int iu, int* pos) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu) return ERR;

    *pos = (ip + iu) / 2;
    return 0;
}
```

```c
int quicksort(int* tabla, int ip, int iu) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu){
      printf("Error en los parametros de entrada quick\n");
      return ERR;
    }

    if (ip < iu) {
        int pos;
        int ob = median(tabla, ip, iu, &pos);
        if (ob == ERR) {
          printf("Error en ob\n");
          return ERR;
        }

        /*Intercambiar el pivote con el primer elemento*/
        int temp = tabla[ip];
        tabla[ip] = tabla[pos];
        tabla[pos] = temp;

        int ob_partition = partition2(tabla, ip, iu, &pos, median_stat);
        if (ob_partition == ERR) return ERR;

        int ob1=0;
        if(pos-1>ip){
          ob1 = quicksort(tabla, ip, pos - 1);
          if (ob1 == ERR) {
            printf("Error en ob1\n");
            return ERR;
          }
        }
        int ob2=0;
        if(pos+1<iu){
          ob2 = quicksort(tabla, pos + 1, iu);
          if (ob2 == ERR) {
            printf("Error en ob2\n");
            return ERR;
          }
        }

        if (ob1 == ERR || ob2 == ERR) {
            printf("Error en ob1 o ob2\n");
            return ERR;
        }

        return ob + ob_partition + ob1 + ob2;
    }

    return 0;  /*Caso base: ya está ordenado*/
}
```

4.5 Apartado 5

```c
/*Función de pivote que compara los valores de las posiciones ip, iu y (ip+iu)/2*/
/*Devuelve la posición que contiene el valor intermedio entre los tres*/
int median_stat(int* tabla, int ip, int iu, int* pos) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu) return ERR;

    int medio = (ip + iu) / 2;

    if ((tabla[ip] >= tabla[iu] && tabla[ip] <= tabla[medio]) ||
        (tabla[ip] <= tabla[iu] && tabla[ip] >= tabla[medio])) {
        *pos = ip;
    } else if ((tabla[iu] >= tabla[ip] && tabla[iu] <= tabla[medio]) ||
               (tabla[iu] <= tabla[ip] && tabla[iu] >= tabla[medio])) {
        *pos = iu;
    } else {
        *pos = medio;
    }

    return 3;  /*3 operaciones básicas adicionales realizadas por median_stat*/
}


/*Función de partición modificada que utiliza la función de pivote especificada*/
int partition2(int* tabla, int ip, int iu, int* pos, int (*pivote)(int*, int, int, int*)) {
    if (!tabla || ip < 0 || iu < 0 || ip > iu) return ERR;

    /* Llamada a la función de pivote*/
    int ob_pivote = pivote(tabla, ip, iu, pos);

    /*Intercambiar el pivote con el primer elemento*/
    int temp = tabla[ip];
    tabla[ip] = tabla[*pos];
    tabla[*pos] = temp;

    int pivot = tabla[ip];
    int i = ip + 1;
    int j;

    for (j = ip + 1; j <= iu; j++) {
        if (tabla[j] < pivot) {
            /*Intercambiar elementos*/
            temp = tabla[j];
            tabla[j] = tabla[i];
            tabla[i] = temp;
            i++;
        }
    }

    /*Colocar el pivote en su posición final*/
    tabla[ip] = tabla[i - 1];
    tabla[i - 1] = pivot;

    *pos = i - 1;  /*Posición final del pivote*/
    return ob_pivote + iu - ip;  /*Operaciones básicas adicionales + operaciones básicas realizadas por partition*/
}
```

5. Resultados, Gráficas

5.1 Apartado 1

```
e462135@LAPTOP-995AQRBR:~/ANAL/P2$ make exercise4_test
Running exercise4
Practice number 2, section 1
Done by: Iker González Sánchez
Group: 1261
0       1       2       3       4       5       6       7       8       9
```

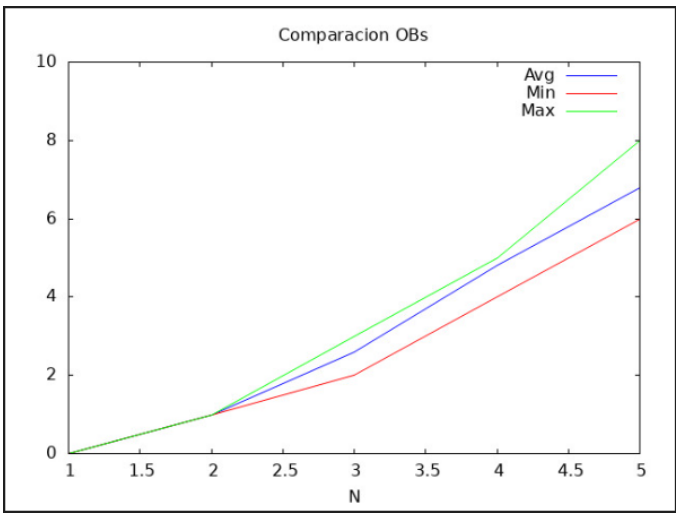
5.2 Apartado 2

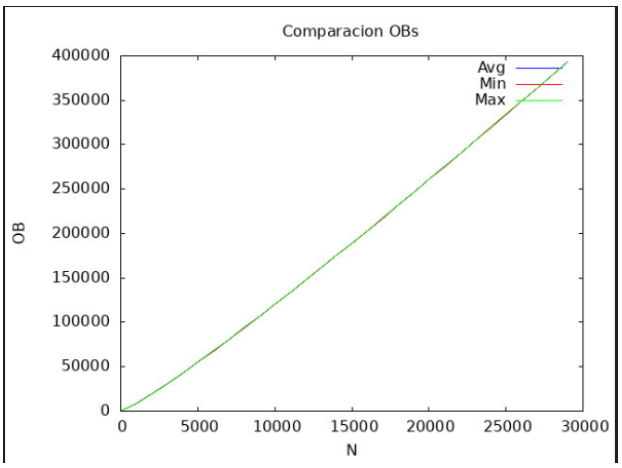
Resultados del apartado 4.

```
1,1.000000,0.000000,0,0
2,0.400000,1.000000,1,1
3,0.400000,2.600000,2,3
4,0.600000,4.800000,4,5
5,0.600000,6.800000,6,8
```

```
1,0.000000,0.000000,0,0
1001,95.200000,8720.000000,8706,8732
2001,201.400000,19439.600000,19417,19464
3001,310.200000,30924.600000,30905,30950
4001,801.800000,42842.600000,42771,42896
5001,528.600000,55237.400000,55192,55279
6001,1033.800000,67862.000000,67792,67951
7001,715.400000,80663.200000,80611,80724
8001,1257.600000,93644.400000,93558,93704
9001,1653.000000,107042.600000,106996,107091
10001,2163.200000,120451.600000,120413,120489
11001,2400.600000,134085.000000,133983,134173
12001,2971.000000,147634.800000,147556,147712
13001,2707.200000,161455.200000,161422,161496
14001,3298.200000,175291.200000,175257,175323
15001,3605.200000,189314.200000,189229,189438
16001,3699.600000,203350.200000,203325,203427
17001,3906.000000,217545.000000,217498,217673
18001,4111.200000,231971.000000,231898,232057
19001,4754.800000,246424.400000,246366,246489
20001,4559.800000,260923.800000,260785,261038
21001,5163.800000,275471.600000,275348,275638
22001,5382.600000,290096.800000,289937,290201
23001,5524.400000,304777.600000,304745,304853
24001,6057.000000,319330.400000,319197,319443
25001,5936.000000,334150.200000,334028,334226
26001,6605.200000,348989.600000,348839,349100
27001,4490.400000,363832.800000,363718,363893
28001,4496.600000,378767.600000,378674,378909
29001,6741.800000,393584.400000,393525,393637
```

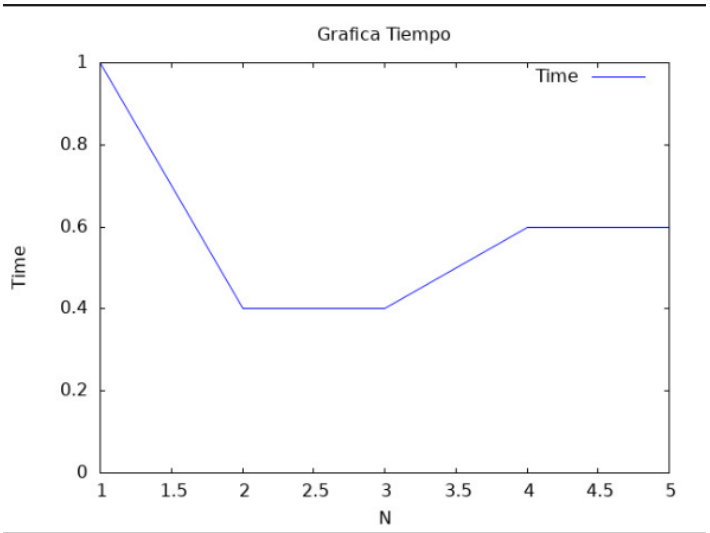
Gráfica comparando los tiempos mejor peor y medio en OBs para MergeSort, comentarios a la gráfica.

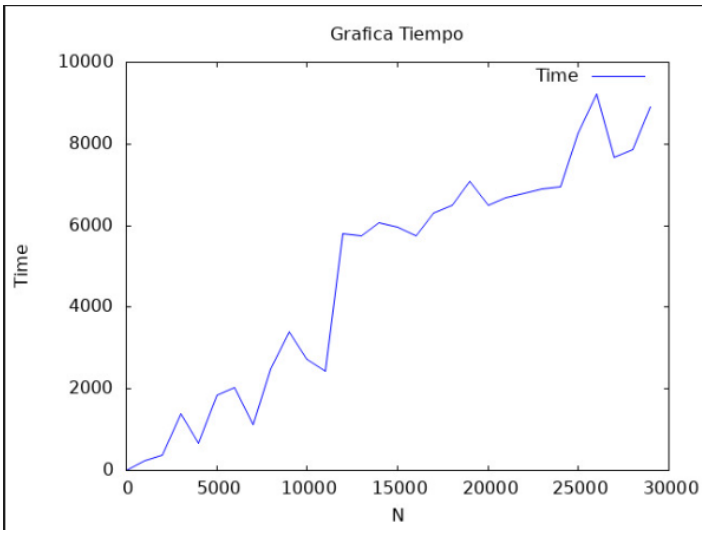




Con tamaños pequeños observamos diferencia ya que max es el mayor, avg el intermedio y min el menor; sin embargo cuando utilizamos valores más altos vemos que tienden a lo mismo, en específico a O(N*logN).

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





El algoritmo es eficiente, pese a que se observan picos puede deberse a la ejecución de procesos no relacionados que consumen recursos en momentos específicos.

Calculando los resultados teóricos, observamos valores mayores a los obtenidos, aunque similares.

Ejemplo: 7001 → 80678.800000,80644,80704

Teórico: 7001*log7001=89426,19

5.3 Apartado 3

```
e462135@LAPTOP-995AQRBR:~/ANAL/P2$ make exercise4_test
Running exercise4
Practice number 2, section 3
Done by: Iker González Sánchez
Group: 1261
0       1       2       3       4       5       6       7       8       9
```

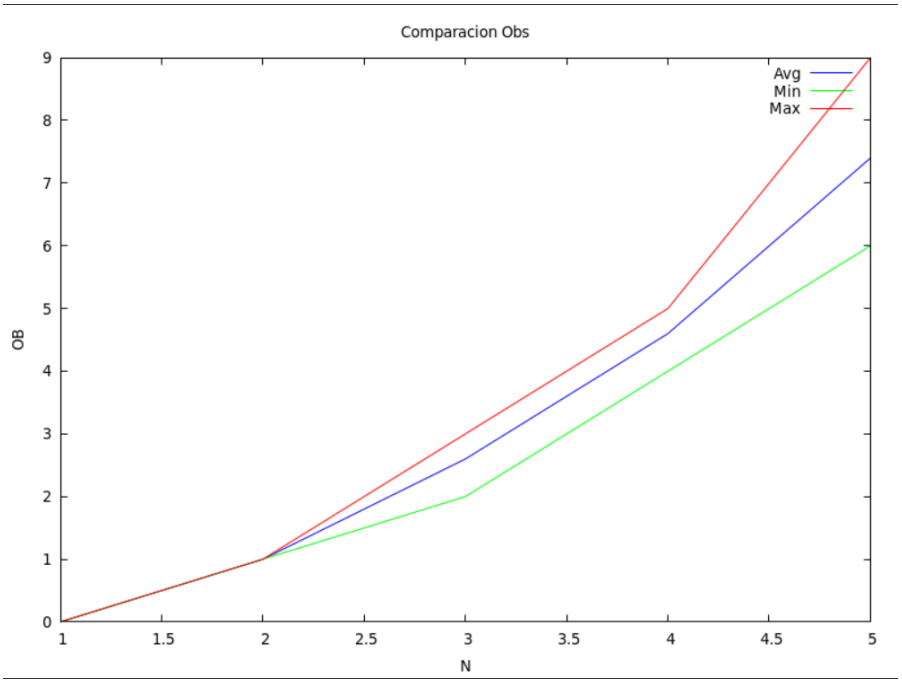
5.4 Apartado 4

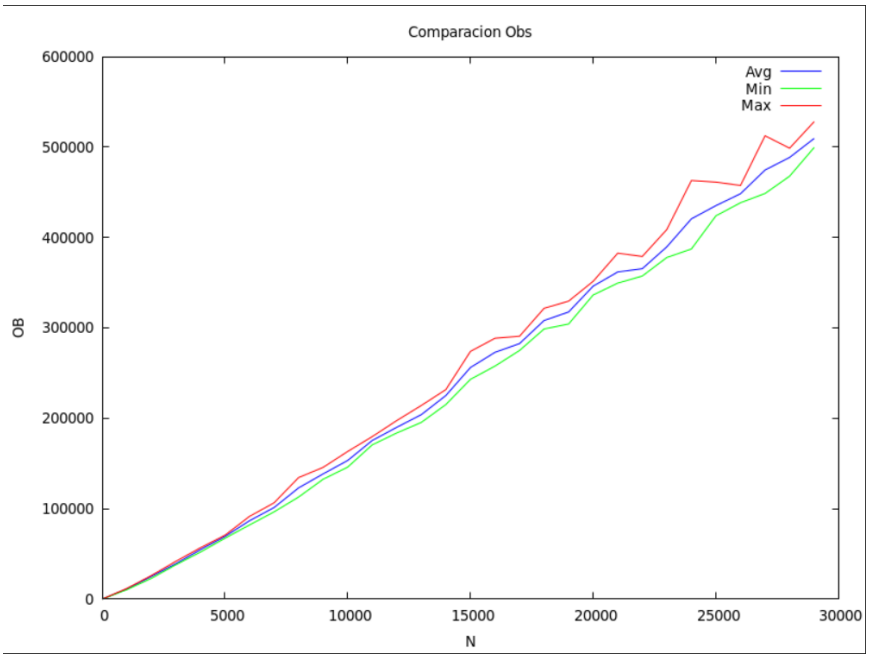
Resultados del apartado 4.

```
1,0.200000,0.000000,0,0
2,0.400000,1.000000,1,1
3,0.200000,2.600000,2,3
4,0.200000,4.600000,4,5
5,0.400000,7.400000,6,9
```

```
1,0.600000,0.000000,0,0
1001,105.000000,11087.000000,10375,11781
2001,207.000000,25071.400000,22934,26008
3001,312.000000,39123.800000,37927,41751
4001,349.000000,54645.400000,51965,56724
5001,380.000000,69264.600000,67326,70643
6001,458.400000,86606.800000,81987,91485
7001,534.000000,101221.800000,96498,106411
8001,777.800000,123024.800000,112851,134443
9001,737.400000,138469.200000,132571,145718
10001,954.800000,153302.400000,145976,163293
11001,900.400000,175399.000000,170561,179693
12001,1691.000000,190007.400000,183740,197456
13001,1625.000000,204056.400000,195461,214209
14001,2135.000000,225153.400000,215222,231722
15001,1908.200000,256115.200000,243061,273939
16001,2025.400000,272940.400000,257759,288486
17001,2095.400000,282600.400000,274994,290706
18001,2312.200000,308153.600000,298801,321630
19001,2875.600000,317547.200000,304236,329583
20001,2575.600000,346147.800000,336389,351543
21001,2880.000000,361819.000000,349490,382575
22001,2915.600000,365429.200000,357312,378972
23001,2899.400000,389635.800000,377806,408793
24001,3314.800000,420597.400000,387274,462952
25001,3406.000000,435206.800000,423917,461159
26001,3465.600000,448367.800000,438396,457445
27001,3416.400000,474591.400000,448643,512439
28001,3612.800000,488477.400000,467752,498719
29001,3861.800000,509467.400000,499467,528026
```

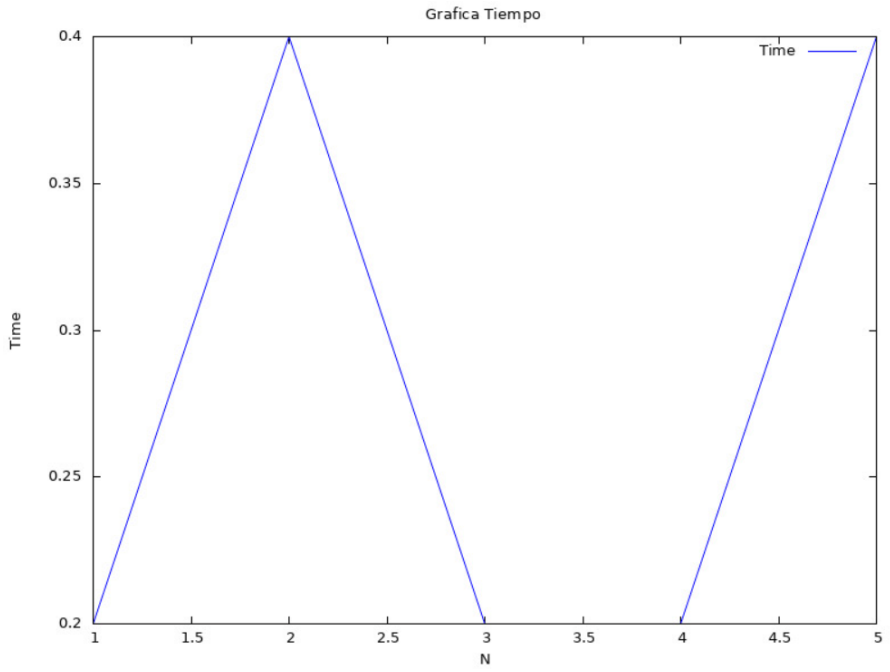
Gráfica comparando los tiempos mejor peor y medio en OBs para QuickSort, comentarios a la gráfica.

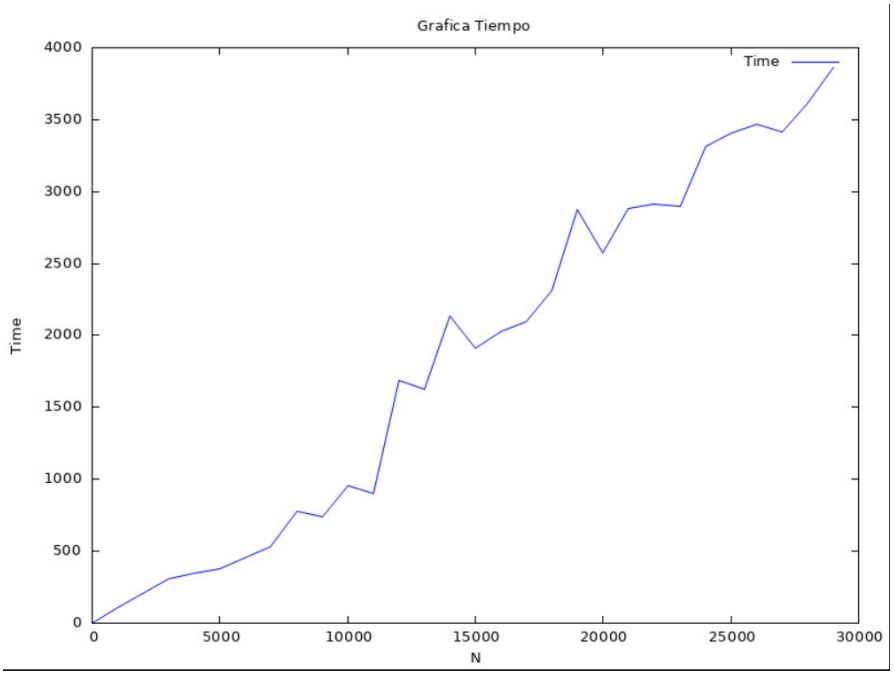




Se aprecia una mayor diferencia entre Max, Avg y Min pero se sigue apreciando que sigue la fórmula N*logN.

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





En la primera gráfica no sé puede apreciar debido a los pocos datos, en la segunda en cambio se aprecia mejor y se siguen viendo los mismos picos que en el algoritmo anterior, además de tener tiempos similares.

Calculando los resultados teóricos, observamos valores dentro de lo esperado.

Ejemplo: 7001 → 800000,96498,106411

Teórico: 7001*log7001=89426,19

5.5 Apartado 5

Resultados del apartado 5.

- Median:

```
1,0.400000,0.000000,0,0
2,0.000000,1.000000,1,1
3,0.400000,2.600000,2,3
4,0.400000,4.600000,4,6
5,0.400000,7.400000,6,10
```

```
1,0.400000,0.000000,0,0
1001,96.600000,10641.600000,10333,11426
2001,209.200000,25107.800000,23012,26659
3001,305.200000,39631.400000,37146,42330
4001,821.200000,55801.400000,53801,59204
5001,395.800000,75789.200000,68051,84272
6001,486.000000,85212.800000,81971,89300
7001,1072.000000,102459.600000,98597,105664
8001,653.600000,124836.600000,120754,128240
9001,1270.600000,139975.800000,128140,146304
10001,1335.800000,151391.000000,145734,159466
11001,956.000000,174296.200000,162444,184663
12001,1501.400000,186845.200000,183219,192573
13001,1796.000000,205915.400000,203009,213095
14001,1870.000000,228959.000000,220265,235624
15001,1944.600000,244059.800000,235296,258363
16001,2142.000000,269441.000000,254962,295295
17001,2581.400000,284369.400000,272147,303179
18001,3223.400000,300994.600000,289388,322828
19001,3369.200000,326486.000000,317839,336187
20001,3514.400000,351723.800000,336514,367944
21001,3544.000000,354672.200000,344683,369497
22001,3854.200000,384594.000000,375167,400867
23001,3757.200000,402014.200000,377444,428432
24001,2689.400000,414111.600000,392467,434496
25001,2786.400000,428859.200000,423078,433587
26001,4090.000000,457179.000000,449184,477198
27001,4466.200000,492083.600000,479229,511420
28001,4975.400000,483811.200000,476370,491346
29001,4763.000000,521132.000000,509042,534471
```

- Median_avg:

```
1,0.200000,0.000000,0,0
2,0.400000,1.000000,1,1
3,0.200000,2.800000,2,3
4,0.200000,4.400000,4,6
5,0.400000,7.600000,6,10
```

```
1,0.000000,0.000000,0,0
1001,82.000000,10996.000000,10180,11652
2001,178.400000,24467.600000,23117,27101
3001,223.200000,40440.600000,37948,42292
4001,823.000000,56822.600000,52629,60325
5001,385.400000,69866.600000,66148,74378
6001,489.400000,87728.200000,83115,91190
7001,1295.400000,102293.200000,100608,104374
8001,1429.600000,120505.000000,117137,128050
9001,1145.600000,136798.800000,132898,141541
10001,1144.600000,153202.200000,143035,157653
11001,2950.200000,170697.800000,163231,179601
12001,1932.200000,189124.800000,179969,202255
13001,2063.000000,208534.600000,204479,215318
14001,2258.000000,232832.400000,221220,245540
15001,2355.000000,247945.400000,235505,268549
16001,2577.200000,259979.800000,253563,272242
17001,2678.800000,286463.800000,275696,299600
18001,3247.200000,309056.200000,291418,322048
19001,3495.200000,316924.600000,302549,327283
20001,3529.600000,334545.800000,322171,355940
21001,3634.800000,358175.600000,345593,385147
22001,3730.600000,366906.400000,359411,377382
23001,3916.200000,387661.000000,376982,395869
24001,4005.800000,405428.400000,392596,418412
25001,4154.000000,431099.400000,412765,448248
26001,4587.000000,474906.800000,452330,501480
27001,4991.800000,493753.800000,465497,536291
28001,3637.200000,493006.600000,476950,510567
29001,4113.600000,527797.600000,504570,557255
```

- Median_stat:

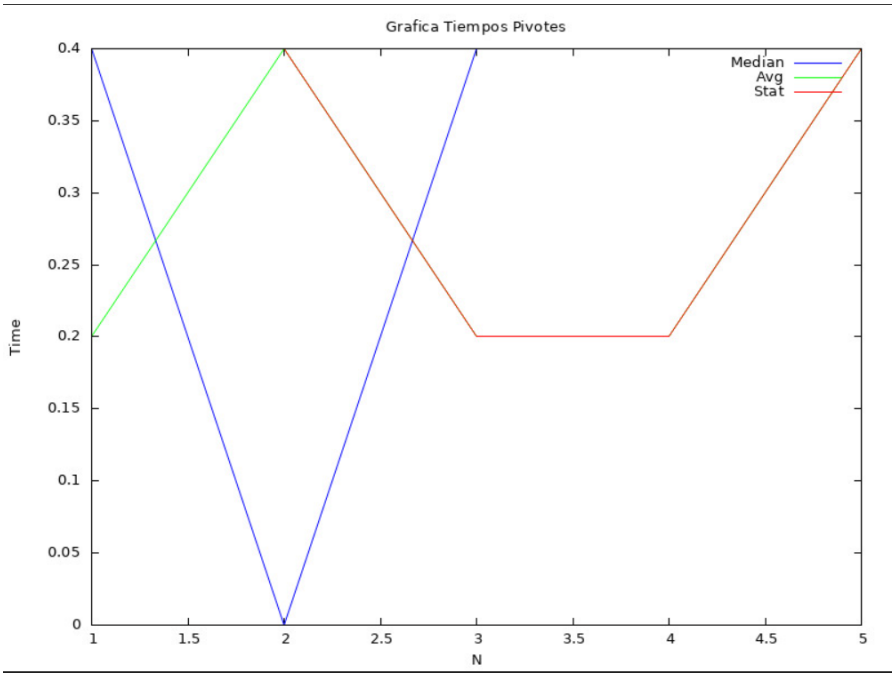
```
1,0.400000,0.000000,0,0
2,0.400000,4.000000,4,4
3,0.200000,5.000000,5,5
4,0.200000,10.000000,10,10
5,0.400000,12.000000,12,12
```

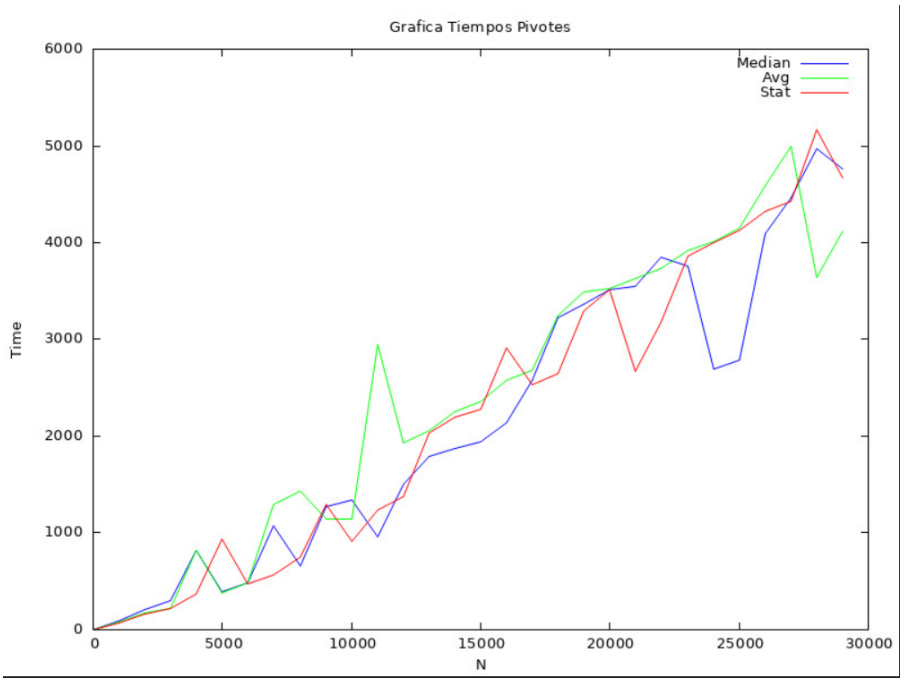
```
1,0.000000,0.000000,0,0
1001,74.400000,11263.800000,10705,11706
2001,156.200000,24516.200000,24107,25326
3001,224.000000,38686.400000,37737,39568
4001,373.400000,53486.800000,52509,55338
5001,939.200000,67915.000000,66708,69227
6001,473.200000,85115.200000,84116,87422
7001,563.200000,101518.400000,98339,105887
8001,756.400000,116283.600000,113355,119589
9001,1297.200000,132730.000000,131114,133454
10001,914.400000,149709.800000,147095,152113
11001,1239.400000,166100.200000,159997,173198
12001,1381.200000,183984.400000,180737,188229
13001,2034.000000,205813.600000,199955,212227
14001,2193.800000,220131.400000,218027,225191
15001,2278.800000,234714.200000,227170,238887
16001,2910.000000,249179.400000,244031,253410
17001,2536.400000,268665.000000,259737,278808
18001,2645.000000,289820.000000,286425,292149
19001,3292.200000,302194.400000,298547,304186
20001,3510.600000,324580.200000,317370,338141
21001,2670.200000,339083.200000,335249,345904
22001,3183.400000,359772.600000,353655,364497
23001,3859.600000,378556.200000,373564,382808
24001,3995.600000,397300.000000,385772,412118
25001,4131.800000,407986.600000,403847,412575
26001,4320.000000,436232.400000,429212,442813
27001,4427.200000,449502.200000,442871,457101
28001,5172.600000,464482.200000,451786,491226
29001,4670.000000,482425.400000,479595,486108
```

Observamos que median suele tener una media más baja que avg ya que tiene un max menor. Además vemos que al inicio (y con pocos datos) stat parece mucho peor dado que necesita muchas mas OBs, sin embargo cuanto más avanza, vemos que obtiene muchas menos.

Esto se debe a la estrategia por la que opta cada versión.

> Gráfica con el tiempo medio de reloj comparando las versiones de Quicksort con las funciones pivote **median, median_avg** y **median_stat.**





Como hemos observado otras veces, la primera gráfica no aporta mucho. La segunda, en la que se ven mucho mejor los tiempos tiene picos y varía mucho, por lo que no se puede decir con certeza que sea todo debido a las diferencias entre los median, ya que podría deberse a la ejecución de procesos no relacionados que consumen recursos en momentos específicos. Pese a todo siguen una línea similar exceptuando algunos picos más marcados.

5. Respuesta a las preguntas teóricas.

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

5.1 Pregunta 1

Esta pregunta ya ha sido abordada en los anteriores apartados. Como conclusión podemos decir que los picos pueden deberse a otros procesos puntuales que realice el ordenador.

5.2 Pregunta 2

Esta pregunta ha sido abordada en el último apartado.

5.3 Pregunta 3

- MergeSort:

Caso mejor, peor y medio:

MergeSort tiene un rendimiento constante en cualquier situación, y su rendimiento no está influenciado por el estado inicial de los datos. En todos los casos, tiene una complejidad de tiempo de O(n log n).

- QuickSort:

Caso Mejor:

El caso mejor ocurre cuando el pivote elegido divide la lista en dos mitades aproximadamente iguales. En este caso, el rendimiento es O(n log n).

Caso Peor:

El caso peor para QuickSort ocurre cuando la lista está ordenada de manera ascendente o descendente y se elige siempre el elemento más pequeño o más grande como pivote. Esto puede llevar a una complejidad de tiempo cuadrática, O(n^2).

Caso Medio:

En promedio, QuickSort tiene un rendimiento muy bueno, con una complejidad de tiempo esperada de O(n log n). Sin embargo, la elección de pivotes puede influir en el rendimiento en casos específicos.

Calcular:

Caso Mejor:

En general, el caso mejor para MergeSort y QuickSort es cuando la entrada ya está parcial o totalmente ordenada. Se podría calcular estrictamente este caso proporcionando una entrada preordenada y midiendo el tiempo de ejecución.

Caso Peor:

Para el caso peor, se podría proporcionar una entrada que maximice el número de operaciones realizadas. Por ejemplo, para QuickSort, una lista invertida sería un caso peor.

Caso Medio:

El caso medio se refiere a un rendimiento promedio sobre todas las posibles entradas. Podríamos aproximarnos al caso medio generando entradas aleatorias y promediando los resultados de varias ejecuciones.

MergeSort es en general indiferente

5.4 Pregunta 4

La eficiencia empírica de los algoritmos MergeSort y QuickSort puede variar según diversos factores, como el tamaño de los datos y la implementación específica. La eficiencia teórica, nos proporciona una guía general, pero la implementación y las características del sistema también juegan un papel importante.

Eficiencia Empírica:

MergeSort: En términos de eficiencia empírica, MergeSort suele ser constante y eficiente en una variedad de situaciones. Su rendimiento es predecible y se mantiene cerca de su complejidad teórica O(N*logN).

QuickSort: En términos generales, QuickSort también es muy eficiente en la práctica y a menudo supera a otros algoritmos de ordenamiento. Sin embargo, su rendimiento puede depender de la elección del pivote y la implementación específica.

Predicción Teórica:

La predicción teórica para ambos algoritmos es O(N*logN) lo que sugiere que ambos algoritmos deberían tener un rendimiento similar en grandes conjuntos de datos.

En la práctica, QuickSort a menudo supera a MergeSort en términos de eficiencia debido a una menor sobrecarga en la constante y una menor necesidad de memoria adicional.

Gestión de Memoria:

MergeSort: Utiliza memoria adicional para almacenar las sublistas durante el proceso de combinación. Esto puede requerir más espacio, especialmente para grandes conjuntos de datos. Sin embargo, su uso de memoria es predecible y no depende del estado de los datos de entrada.

QuickSort: La implementación de QuickSort no necesita memoria adicional para crear sub-listas temporales. Sin embargo, la recursión implica el uso de la pila de llamadas, que

podría consumir memoria en el sistema. En términos de memoria adicional, QuickSort tiene una ventaja al no requerir espacio adicional para sublistas.

Conclusión:

En términos generales, ambos algoritmos son eficientes y tienen una complejidad asintótica similar, pero QuickSort a menudo tiene un rendimiento superior en situaciones promedio debido a su menor sobrecarga constante. Sin embargo, la elección entre los dos puede depender de los requisitos específicos del problema y de las características del sistema. En situaciones donde la gestión de memoria es crítica, QuickSort puede ser preferido por su menor necesidad de espacio adicional.

6. Conclusiones finales.

Como conclusión, podemos decir que ambos algoritmos son eficientes, aunque cada uno con sus respectivas desventajas, MergeSort utilizando más memoria y QuickSort siendo menos regular; aunque en general podemos decir que QuickSort es superior. En la práctica hemos visto esto sobre todo con el tiempo, aunque en operaciones básicas suele ser mejor MergeSort.

Por lo tanto podemos concluir diciendo que dependiendo el problema y requisitos será recomendable usar uno sobre el otro o viceversa.