Análisis de Algoritmos 2022/2023

Práctica 2

Nicolas Victorino Ruiz y Antonio Van-Oers Luis, 1272.

Código	Gráficas	Memoria	Total

1. Introducción.

Continuando con la resolución de varios ejercicios relacionados con algoritmos de ordenación, en esta segunda práctica trabajaremos con Quicksort y MergeSort. En cada ejercicio implementaremos los algoritmos, así como las funciones de apoyo necesarias para su correcto funcionamiento. Una de las diferencias con la primera práctica es que ambos algoritmos son recursivos según lo visto en teoría por lo que habrá que pensar en una forma un poco distinta para contar el número de operaciones básicas realizadas.

Para nosotros, el objetivo de la práctica es familiarizarnos con los algoritmos de ordenación recursivos, así como el tiempo de ejecución de estos con el fin de buscar la mayor optimización y rendimiento posible. Apoyándonos y modificando los excercise_test de la práctica anterior podremos dar una tabla desordenada a cada algoritmo para que la ordene y comprobar el coste y operaciones básicas que realiza.

2. Objetivos.

2.1 Apartado 1

Implementar de forma eficiente el algoritmo conocido como MergeSort cuya función es: int *mergesort(int* tabla, int ip, int iu)*. A su vez, necesitaremos implementar una función para combinar las tablas que genera el algoritmo: *int merge(int* tabla, int ip, int iu, int imedio)*

2.2 Apartado 2

Debemos modificar el excercise5.c de forma que nos sirva para probar el correcto funcionamiento del algoritmo MergeSort así como para obtener el número de operaciones básicas que realiza.

2.3 Apartado 3

Implementar de forma eficiente el algoritmo conocido como QuickSort cuya función es: int *quicksort(int* tabla, int ip, int iu)*. Esta devolverá Error en caso de fallo y el numero de operaciones báscias en caso de funcionar correctamente.

Como funciones de apoyo necesitaremos implementar *int partition(int* tabla, int ip, int iu,int *pos)* y *int median(int *tabla, int ip, int iu,int *pos).* Usando el primer elemento de cada tabla como pivote.

Para su comprobación modificaremos el excercise4.

2.4 Apartado 4

Con el fin de obtener el tiempo de ejecución medio de Quicksort, así como el número de operaciones básicas que realiza en sus diferentes casos, modificaremos el excercise5.c y representaremos las gráficas de sus resultados comparándolos con los teóricos

2.5 Apartado 5

Implementando la función: *median_avg(int *tabla, int ip, int iu, int *pos)* ,junto con: *int median stat(int 1 *tabla, int ip, int iu, int *pos)* variamos el pivote utilizado en el algoritmo QuickSort para comprobar cuál de todos los métodos es más eficiente en cuanto a tiempo de ejecución y número de operaciones básicas.

3. Herramientas y metodología

3.1 Apartado 1

Usando Visual Studio en Linux, hemos implementado la función mergesort y merge utilizando como apoyo las diapositivas del tema 2 de teoría. Siguiendo el pseudocódigo en estas pero adaptándolo a los parámetros de nuestra práctica, implementamos el algoritmo recursivamente, con control de errores y en la función merge tras reservar memoria para la tabla auxiliar y combinar ambas tablas utilizamos un free para evitar fugas de memoria. Con esta implementación conseguimos que el algoritmo, tal y como se pide, vaya dividiendo la tabla original en tablas más pequeñas y ordenando estas antes de juntarlas todas, obteniendo el resultado pedido

3.2 Apartado 2

Usando Visual Studio en Linux, y gracias a nuestra anterior implementación del excercise5.c, simplemente hemos tenido que modificar el nombre de la función de la anterior práctica por mergesort para obtener el log con el rendimiento de los casos mejor, peor y medio, así como los tiempos de ejecución de este algoritmo

3.3 Apartado 3

Usando Visual Studio en Linux, hemos implementado las tres funciones necesarias para el algoritmo QuickSort. De nuevo, siguiendo el pseudocódigo proporcionado en las transparencias del tema 2, implementamos en C Quicksort, Partition y median para el algoritmo recursivo, la función para partir la tabla en torno al pivote y la función que obtiene el propio pivote. En esta primera implementación de median simplemente se trata de usar el primer elemento e ir comparando los valores del resto de la tabla en torno a él.

3.4 Apartado 4

Usando Visual Studio en Linux, y gracias a nuestra anterior implementación del excercise5.c, simplemente hemos tenido que modificar el nombre de la función de la anterior práctica por quicksort para obtener el log con el rendimiento de los casos mejor, peor y medio, así como los tiempos de ejecución de este algoritmo

3.5 Apartado 5

Usando Visual Studio en Linux, implementamos dos variantes de median: median_avg, que obtiene como pivote el elemento que está en mitad de la tabla (ip+iu)/2 y median_stat que compara los elementos primero, medio y último y selecciona el valor medio entre ellos haciendo que después, en las gráficas, se pueda apreciar una mejora en el rendimiento del algoritmo.

4. Código fuente

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

4.1 Apartado 1

```
int mergesort(int* tabla, int ip, int iu)

{
  int m, ret = 0, counts = 0;

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

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

```
    counts += ret;

    ret = merge(tabla, ip, iu, m);
    if (ret == ERR)
    {
      return ERR;
    }
    counts += ret;

    return counts;
  }
}

int merge(int* tabla, int ip, int iu, int imedio)
{
  int *t_aux = NULL;
  int i = ip, j = imedio+1, k = 0, ob = 0;

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

  /*Coloca elementos*/
  while (i <= imedio && j <= iu)
  {
    ob++;
    if (tabla[i] < tabla[j])
    {
      t_aux[k] = tabla[i];
      i++;
    } else
    {
      t_aux[k] = tabla[j];
      j++;
    }

    k++;
  }

  /*Rellena elementos no colocados del array*/
  if (i > imedio)
  {
    while (j <= iu)
    {
      t_aux[k] = tabla[j];
      j++;
      k++;
    }
```

```
  } else if (j > iu)
  {
    while (i <= imedio)
    {

      t_aux[k] = tabla[i];
      i++;
      k++;
    }
  }
  for (i = ip, j = 0; i <= iu && j <= iu-ip; i++, j++)
  {
    tabla[i] = t_aux[j];
  }
  free(t_aux);
  return ob;
}
```

4.2 Apartado 2

```
      fprintf(stderr, "Wrong paramenter %s\n", argv[i]);
    }
  }

  /* compute times */
  ret = generate_sorting_times(mergesort, nombre,num_min, num_max,incr,
n_perms);
  if (ret == ERR) { /* ERR_TIME should be a negative number */
    printf("Error in function generate_sorting_times\n");
    exit(-1);
  }
  printf("Correct output \n");

  return 0;
}
```

4.3 Apartado 3

```
int quicksort(int* tabla, int ip, int iu)
{
  int m= 0, ret = 0;

  /*Control Error*/
  if((ip > iu) || !tabla)
  {
    return ERR;
  }
```

```
  if(ip==iu)
  {
    return count;
  }

    else
    {
      /*La m es el elemento medio despues de median, pero en partition
cambia a la posicion inicial*/
      ret=partition(tabla, ip, iu, &m);
      if(ret!= (-1))
      {
        count+=ret;
      }
      else{
        return ERR;
      }

      if(ip < m-1)
      {
        count+=quicksort(tabla,ip, m-1);
      }
      if(m+1 < iu)
      {
        count+=quicksort(tabla, m+1, iu);
      }
    }
  return count;

}

int partition(int* tabla, int ip, int iu,int *pos)
{
  int k=0, aux=0, i, aux2=0, aux3=0, count = 0;

  /*Error control*/
  if(median(tabla, ip, iu, pos)!=0 || ip > iu || !tabla){
    return ERR;
  }

  /*K = primer elemento del trozo de tabla dado*/
  k=tabla[(*pos)];

  /*Cambia el valor del primer elemento de la tabla por el del ultimo*/
  aux = tabla[ip];
  tabla[ip]=tabla[(*pos)];
  tabla[(*pos)]= aux;
```

```
  (*pos)=ip;

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

    if(tabla[i]<k) /*Si la posicion comprobada es menor al valor de la
posicion media*/
    {
      count++;
      (*pos)++;
      /*Hace un switch de la posicion comprobada pos+ 1*/
      aux2 = tabla[i];
      tabla[i]=tabla[(*pos)];
      tabla[(*pos)]= aux2;
    }
    /*Hace un switch del primer elemento por el elemento correspondiente
a pos.*/
  }
    aux3 = tabla[ip];
    tabla[ip]=tabla[(*pos)];
    tabla[(*pos)]= aux3;

  return count;
}

int median(int *tabla, int ip, int iu,int *pos)
{
  /*Error control*/
  if(!tabla || ip > iu){
    return ERR;
  }

  (*pos)=ip;

  if(!pos)
  {
    return ERR;
  }

  return 0;

}
```

4.4 Apartado 4

```
  /* compute times */
  ret = generate_sorting_times(quicksort, nombre,num_min, num_max,incr,
n_perms);
```

```
  if (ret == ERR) { /* ERR_TIME should be a negative number */
    printf("Error in function generate_sorting_times\n");
    exit(-1);
  }
  printf("Correct output \n");

  return 0;
}
```

4.5 Apartado 5

```
int median_avg(int *tabla, int ip, int iu,int *pos)
{
     /*Error control*/
  if(!tabla || ip > iu){
    return ERR;
  }

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

  if(!pos)
  {
    return ERR;
  }
  return 0;
}

int median_stat(int *tabla, int ip, int iu,int *pos)
{
int aux;
/*Error control*/
  if(!tabla || ip > iu){
    return ERR;
  }

  aux=(ip+iu)/2;

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

```
        (*pos) = iu;
    }
  if(!pos)
  {
    return ERR;
  }
  return 0;
}
```

5. Resultados, Gráficas

5.1 Apartado 1

Resultados del apartado 1.

Tras ejecutar el ejercicio 4 para merge sort con estos valores:

```
exercise4_test:
    @echo Running exercise4
    @./exercise4 -size 100
```

Obtenemos el array ordenado por la terminal:

```
nicolas@nicolas-Nitro-AN515-52:~/Universidad/Segundo/Aalg/Practicas/Practica 2/git 17-11$ make exercise4_test
Running exercise4
Practice number 1, section 4
Done by: Nicolas Victorino && Antonio Van-Oers
Grupo: 1271
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
```

5.2 Apartado 2

Resultados del apartado 2.

Tras ejecutar el ejercicio 5 para mergesort con estos valores en el makefile:

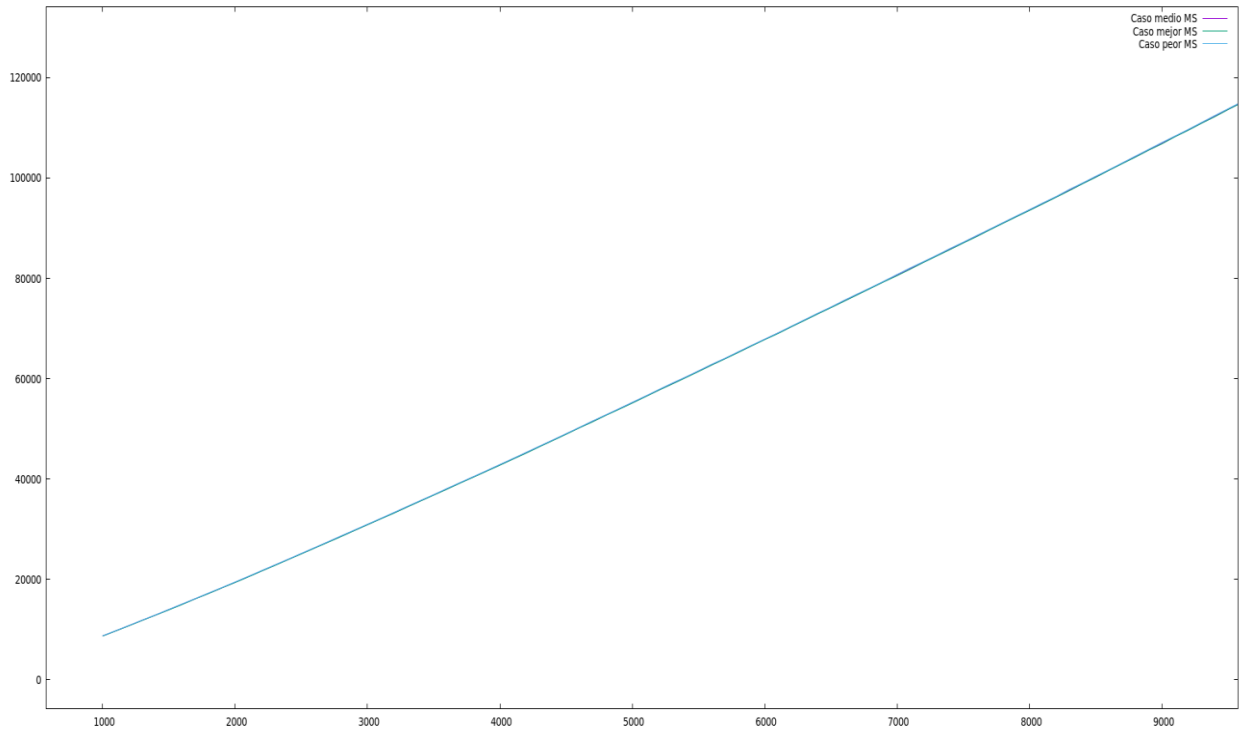
```
exercise5_test:
    @echo Running exercise5
    @./exercise5 -num_min 5000 -num_max 10000 -incr 100 -numP 20 -outputFile exercise5.log
```

Obtenemos un fichero de salida que tiene este aspecto:

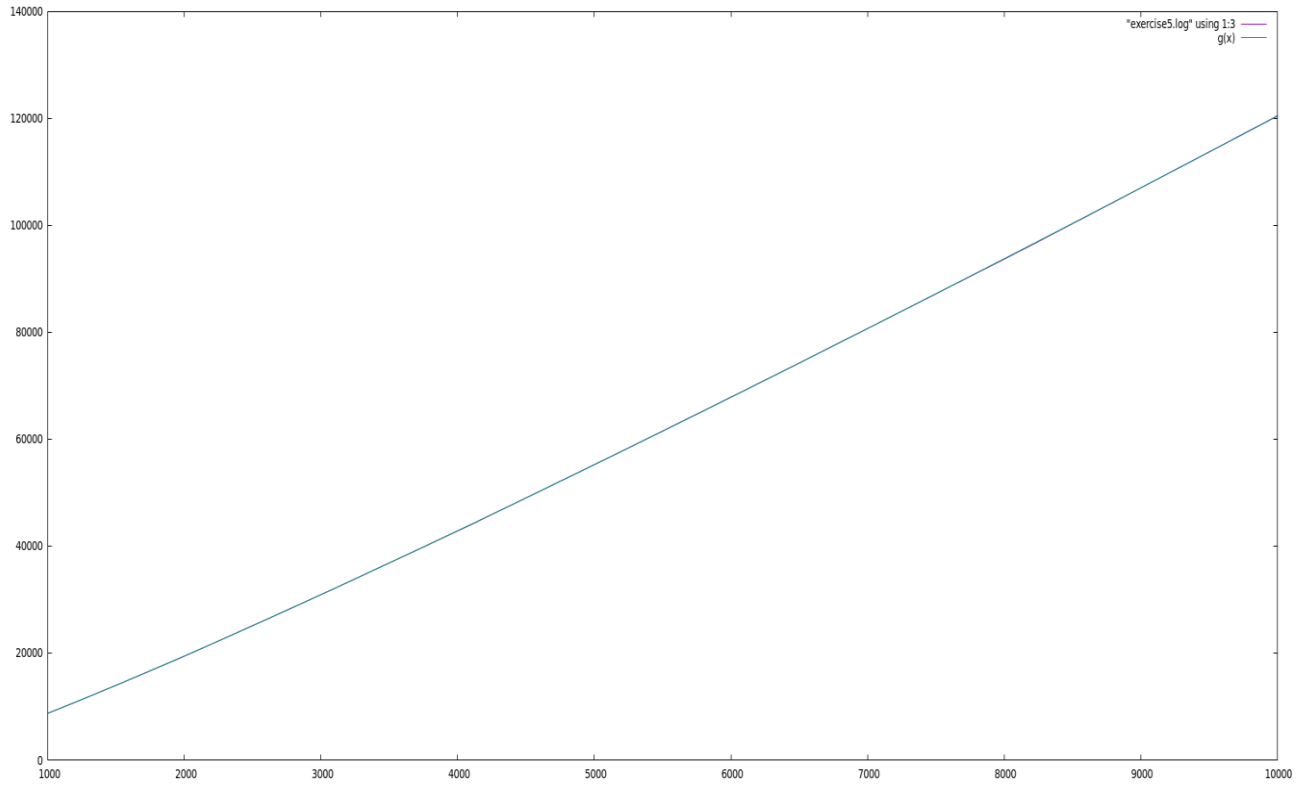
Tamaño	Tiempo de ejecución	PromedioOB	Min_OB	Max_OB
5000	4.117850E-03	55232.950	55142	55329
5100	3.782750E-03	56468.950	56383	56563
5200	2.906600E-03	57731.450	57635	57837
5300	1.471050E-03	59007.600	58943	59094
5400	1.360350E-03	60258.100	60187	60324
5500	1.389050E-03	61508.900	61432	61597
5600	1.411900E-03	62745.250	62646	62834
5700	1.439100E-03	64035.000	63958	64126
5800	1.469750E-03	65301.650	65218	65370
5900	1.498900E-03	66556.700	66454	66650
6000	1.521450E-03	67827.400	67694	67940
6100	1.544200E-03	69111.850	69033	69226
6200	1.575800E-03	70382.000	70287	70433
6300	1.611750E-03	71665.650	71569	71781
6400	1.636050E-03	72945.750	72851	73046
6500	1.664050E-03	74241.200	74132	74370
6600	1.692550E-03	75518.150	75452	75594
6700	1.721650E-03	76818.000	76712	76879
6800	1.749200E-03	78095.650	77928	78169
6900	1.768350E-03	79390.100	79240	79464
7000	1.792550E-03	80693.650	80617	80786
7100	1.813900E-03	81965.250	81854	82056
7200	1.838000E-03	83252.550	83152	83393
7300	1.869400E-03	84561.950	84463	84625
7400	1.896250E-03	85844.800	85777	85950
7500	1.924350E-03	87155.150	87050	87262
7600	1.948350E-03	88454.300	88379	88569
7700	1.971950E-03	89753.300	89651	89857
7800	1.994700E-03	91040.950	90961	91143
7900	2.018600E-03	92317.800	92247	92415
8000	2.043500E-03	93642.050	93563	93727
8100	2.067100E-03	94946.350	94849	95079
8200	2.092300E-03	96231.050	96128	96349
8300	2.130400E-03	97589.000	97512	97674
8400	2.160500E-03	98946.600	98867	99071
8500	2.192950E-03	100262.350	100217	100331
8600	2.223600E-03	101607.950	101446	101758
8700	2.251400E-03	102926.450	102839	103028
8800	2.289950E-03	104277.150	104209	104376
8900	2.320050E-03	105616.950	105541	105730
9000	2.364200E-03	106973.900	106870	107049
9100	2.408850E-03	108335.100	108203	108455
9200	2.429100E-03	109671.000	109578	109862
9300	2.459600E-03	111004.950	110891	111144
9400	2.484000E-03	112358.200	112254	112495
9500	2.521700E-03	113720.600	113635	113804
9600	2.545850E-03	115061.400	114976	115228

Como se puede observar, los valores de las operaciones básicas corresponden con los valores teóricos para cada tamaño del array.

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



- Gráfica con el tiempo medio de reloj para MergeSort, comentarios a la gráfica.
  g(x) es el caso medio teórico del algoritmo MS, y representa NlogN



5.3 Apartado 3

Resultados del apartado 3.

Tras ejecutar el ejercicio 4 para quicksort con estos valores:

```
exercise4_test:
	@echo Running exercise4
	@./exercise4 -size 100
```

```
nicolas@nicolas-Nitro-AN515-52:~/Universidad/Segundo/Aalg/Practicas/Practica 2/git 17-11$ make exercise4_test
Running exercise4
Practice number 1, section 4
Done by: Nicolas Victorino && Antonio Van-Oers
Grupo: 1271
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
```

Obtenemos el array ordenado por la terminal:

5.4 Apartado 4

Resultados del apartado 4.

Tras ejecutar el ejercicio 5 para quicksort con estos valores en el makefile:

```
exercise5_test:
	@echo Running exercise5
	@./exercise5 -num_min 5000 -num_max 10000 -incr 100 -numP 20 -outputFile exercise5.log
```

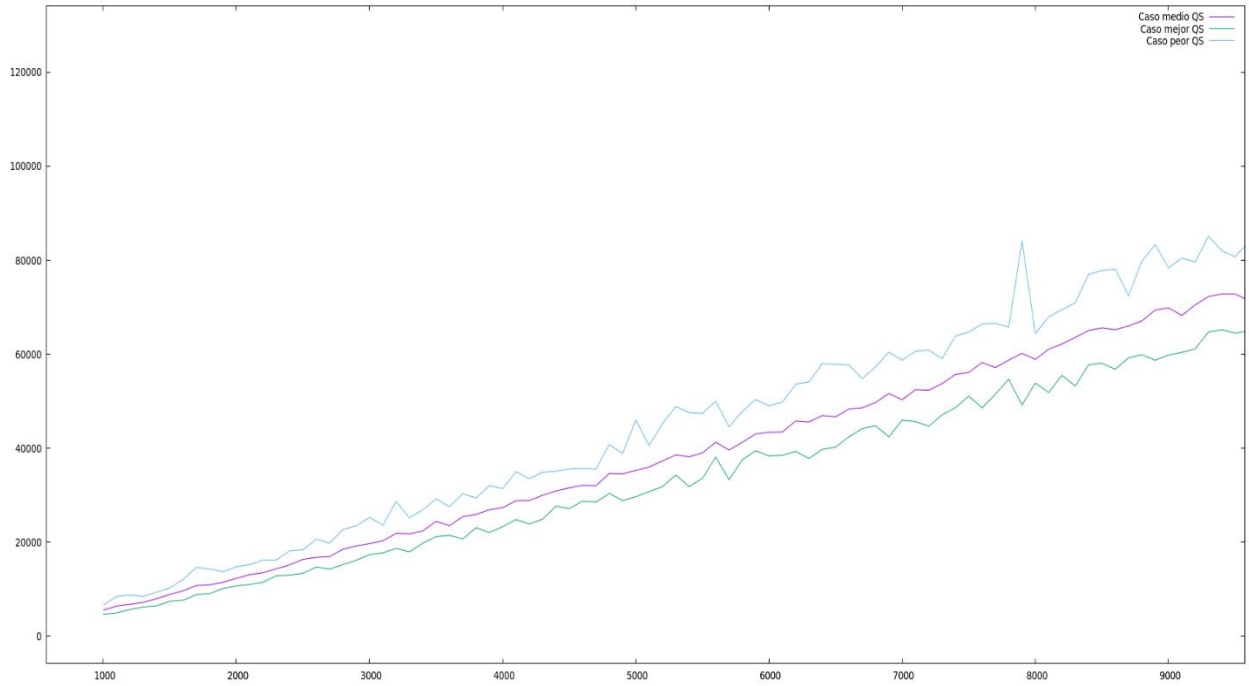
Obtenemos un fichero de salida que tiene este aspecto:

Tamaño	Tiempo de ejecución	PromedioOB	Min_OB	Max_OB
5000	2.491300E-03	70640.250	65552	78760
5100	2.655650E-03	73168.850	68314	82575
5200	2.595550E-03	73789.600	69107	81549
5300	1.004450E-03	78485.150	71323	91201
5400	7.498000E-04	77397.450	73732	86446
5500	7.960500E-04	78633.300	75099	87476
5600	5.345500E-04	81218.550	75472	90449
5700	5.357000E-04	82991.450	75472	89519
5800	5.411000E-04	84358.850	78549	95570
5900	5.512500E-04	85456.600	79167	98057
6000	5.652000E-04	85454.150	80296	91346
6100	5.816500E-04	87834.850	82931	99165
6200	5.997500E-04	91550.850	85656	104596
6300	6.001000E-04	92560.150	87542	104439
6400	6.162500E-04	94206.300	87500	104081
6500	6.275000E-04	95657.150	87907	106537
6600	6.351500E-04	96064.300	89960	103361
6700	6.197000E-04	99268.050	91774	110157
6800	6.285000E-04	101932.600	94542	113870
6900	6.391500E-04	103385.950	96745	119222
7000	6.436500E-04	102407.950	97797	108712
7100	6.511000E-04	104318.400	98237	111310
7200	6.640500E-04	107132.350	100954	118848
7300	6.890000E-04	111081.050	102803	121906
7400	6.904500E-04	110738.700	104806	121934
7500	7.077500E-04	112452.500	104100	119537
7600	7.176500E-04	112553.000	107395	119241
7700	7.209000E-04	116053.300	109485	129298
7800	7.245000E-04	117147.250	111004	125150
7900	7.383000E-04	117881.350	109787	132289
8000	7.481500E-04	120665.200	115569	129861
8100	7.719000E-04	122680.300	117480	131463
8200	7.715000E-04	124945.300	119168	130728
8300	7.893500E-04	125390.550	115862	141435
8400	7.913500E-04	127988.350	121086	136956
8500	8.159500E-04	130960.150	124573	138316
8600	8.079000E-04	129738.750	123335	138725
8700	8.232500E-04	132033.950	125950	149842
8800	8.449000E-04	135350.800	124863	147417
8900	8.419500E-04	136943.750	127586	145387
9000	8.617500E-04	138366.950	131129	146664
9100	8.600000E-04	138088.900	127972	148508
9200	8.772000E-04	141111.850	134187	150630
9300	8.916000E-04	144428.300	136695	156732
9400	9.130500E-04	144549.900	133735	156438
9500	9.099000E-04	147375.800	140131	163252
9600	9.181500E-04	148800.450	140160	163304

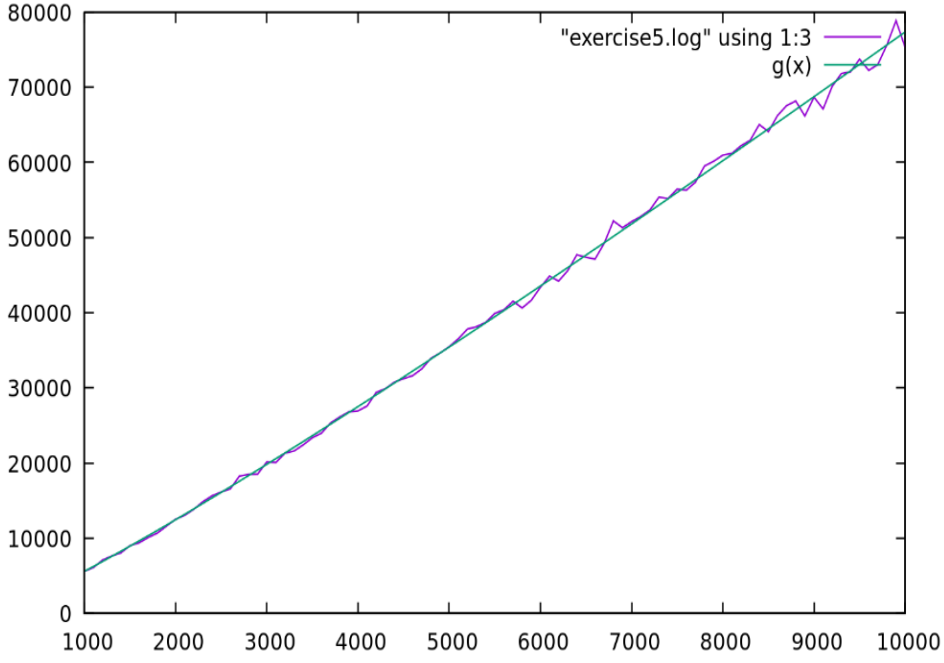
Como se puede observar, los valores de las operaciones básicas corresponden con los valores teóricos para cada tamaño del array.

**Empleado median en partition como elemento pivote.*

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



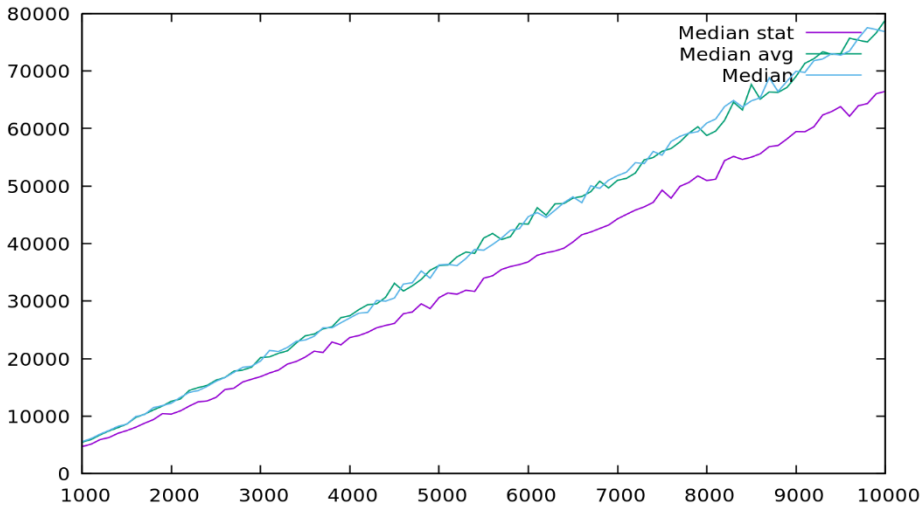
- Gráfica con el tiempo medio de reloj para QuickSort
  g(x) es el caso medio teórico del algoritmo MS, y representa 2NlogN + O(N)



5.5 Apartado 5

Resultados del apartado 5.

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



Las salidas en el archivo tras ejecutar el ejercicio 5 para cada variante median(sin el median mostrado en el ejercicio anterior son las siguientes:
-Median_avg:

Tamaño	Tiempo de ejecución	PromedioOB	Min_OB	Max_OB
5000	2.707700E-03	70928.600	65623	80286
5100	2.428350E-03	71435.550	68822	76190
5200	2.792900E-03	74487.800	68437	84206
5300	2.269600E-03	76544.550	72140	88021
5400	1.026600E-03	77902.350	74542	83979
5500	9.298000E-04	79029.650	74389	89660
5600	9.354000E-04	79403.400	75726	87128
5700	9.717500E-04	82578.150	76206	93262
5800	9.805000E-04	83299.500	77241	95402
5900	1.010950E-03	86093.250	79528	100566
6000	1.019850E-03	85880.550	80746	93279
6100	1.040100E-03	88160.450	84326	96118
6200	1.064950E-03	91824.650	85367	101150
6300	1.071900E-03	92479.500	87155	102078
6400	1.087500E-03	93309.450	88909	99219
6500	1.110150E-03	94525.200	90679	101349
6600	1.140000E-03	98720.500	90714	107555
6700	1.150700E-03	99073.200	94434	106548
6800	1.174950E-03	100650.150	92858	108589
6900	1.199350E-03	103126.250	95853	117606
7000	1.210200E-03	102479.850	98617	111132
7100	1.231250E-03	104857.400	98804	110473
7200	1.274300E-03	107915.450	101311	118655
7300	1.264550E-03	107060.450	100344	111542
7400	1.297900E-03	112242.750	106186	128847
7500	1.324700E-03	113004.950	106689	119652
7600	1.333500E-03	114337.100	107646	127555
7700	1.365050E-03	116381.000	109867	128570
7800	1.362800E-03	114961.600	109981	120432
7900	1.385400E-03	118641.250	111688	124541
8000	1.408300E-03	121418.450	115721	127418
8100	1.427250E-03	123666.800	115484	132816
8200	1.451450E-03	125589.900	112791	135980
8300	1.491700E-03	126770.250	119735	138456
8400	1.499200E-03	128492.700	118624	142258
8500	1.514500E-03	129742.200	120325	136724
8600	1.547950E-03	130651.200	124340	142003
8700	1.587600E-03	133323.550	124812	149917
8800	1.583850E-03	135623.150	127037	165552
8900	1.594550E-03	137666.350	129747	153133
9000	1.610500E-03	137476.650	127774	147593
9100	1.628850E-03	140425.900	131555	163377
9200	1.672000E-03	142574.500	133046	158705
9300	1.683950E-03	142599.500	133448	151380
9400	1.716200E-03	146324.250	136352	160822
9500	1.718950E-03	146153.500	140097	165701
9600	1.725050E-03	147860.600	137809	162648

-Median_stat:

Tamaño	Tiempo de ejecución	PromedioOB	Min_OB	Max_OB
5000	2.843450E-03	60466.600	58127	65524
5100	2.667200E-03	61481.900	58879	64768
5200	2.489700E-03	62759.900	60710	64980
5300	2.395800E-03	64162.450	61645	68071
5400	1.209000E-03	65109.200	63377	67910
5500	1.107600E-03	67147.350	63734	69959
5600	1.096950E-03	69344.900	65543	73083
5700	9.661500E-04	70009.350	68178	74485
5800	9.852500E-04	72087.550	69512	75032
5900	1.002350E-03	73701.700	71617	77482
6000	1.017900E-03	74430.800	70850	80847
6100	1.042600E-03	75642.700	72883	79658
6200	1.057750E-03	77473.100	74504	85208
6300	1.088600E-03	78352.550	75644	81850
6400	1.100000E-03	79568.900	75075	85475
6500	1.120350E-03	81703.950	78544	87245
6600	1.139350E-03	82359.650	79537	86332
6700	1.160950E-03	84193.450	81261	89377
6800	1.178900E-03	86131.600	83005	92498
6900	1.205950E-03	85685.400	84074	88501
7000	1.220100E-03	89402.350	86080	94986
7100	1.232350E-03	89902.800	88067	92013
7200	1.253050E-03	91048.200	88028	96461
7300	1.266550E-03	93347.400	89739	99763
7400	1.287100E-03	94026.550	91153	104009
7500	1.309350E-03	96467.150	91896	103646
7600	1.326350E-03	96629.450	93774	98714
7700	1.350400E-03	98972.800	95532	105146
7800	1.367500E-03	100467.000	96391	107810
7900	1.386650E-03	101568.650	97604	115182
8000	1.412300E-03	103148.650	98118	119819
8100	1.423100E-03	104015.300	100141	108838
8200	1.445350E-03	106737.600	101701	117027
8300	1.459650E-03	108234.250	103656	114894
8400	1.479950E-03	109262.850	105458	113697
8500	1.498150E-03	109789.550	105322	115598
8600	1.516550E-03	110742.700	107199	116803
8700	1.538850E-03	114526.050	108114	123561
8800	1.563250E-03	115883.100	110892	123867
8900	1.583600E-03	116487.300	111367	125031
9000	1.598200E-03	119083.450	114364	130798
9100	1.610750E-03	119035.950	115406	123533
9200	1.643850E-03	120379.650	116959	127114
9300	2.015250E-03	121575.650	117118	127785
9400	1.683450E-03	123228.550	118093	130002
9500	1.703850E-03	125294.200	121280	132862
9600	1.718150E-03	126603.150	122432	135561

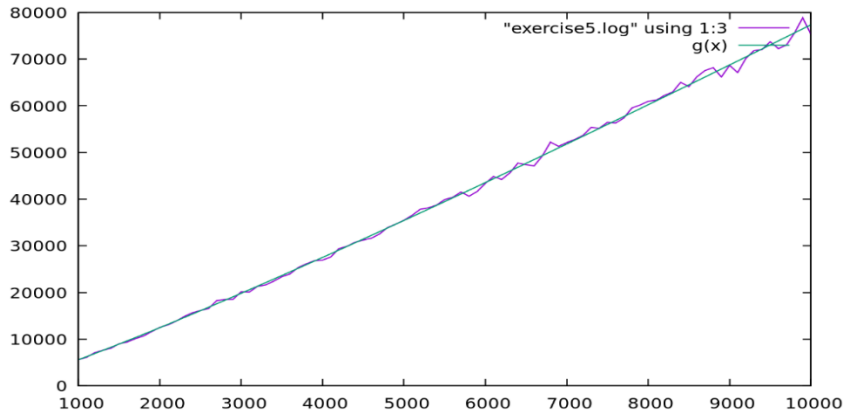
**todas realizadas con los mismos valores en makefile que los resultados anteriores.*

6. Respuesta a las preguntas teóricas.

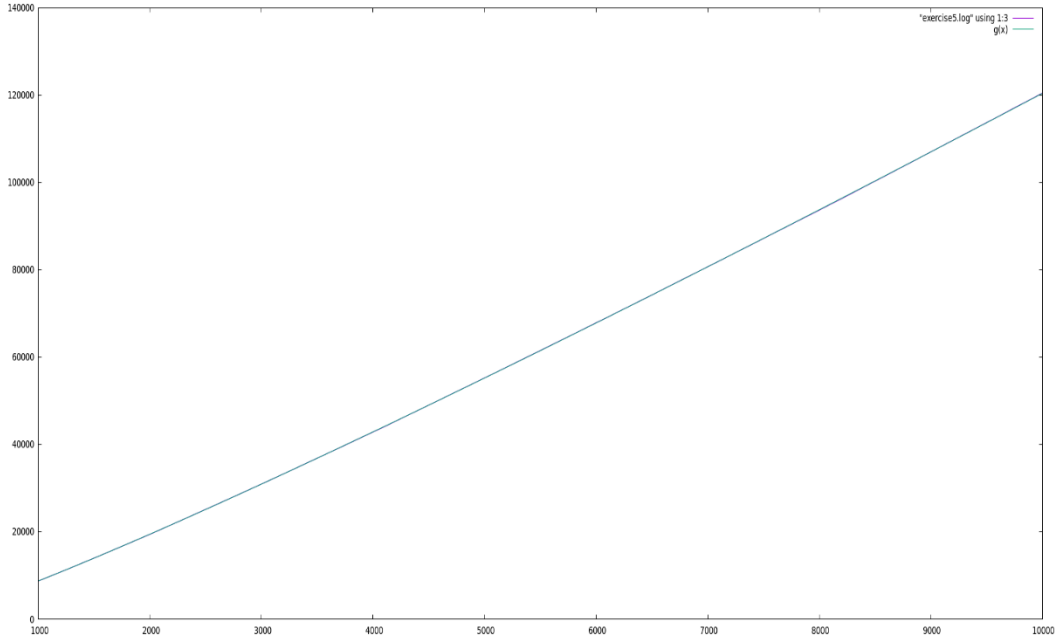
6.1 Pregunta 1

Grafica de la formula teoría caso medio MS y QS comparada con la empírica. Razonar por qué son picudas.

QS:

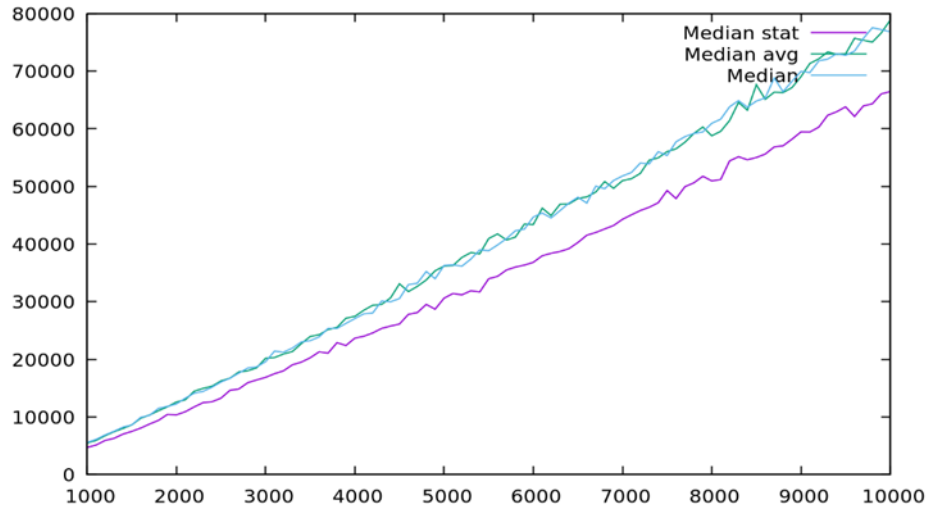


MS:



Justificación: La gráfica del QuickSort es picuda por la naturaleza del algoritmo y las funciones que utiliza. Dependiendo del tamaño de la lista a ordenar y debido al pivote(median) que utiliza, en este caso, el primer elemento, la tabla puede requerir más o menos OBs haciendo que la gráfica no sea constante y que el O(N) del coste, si bien no afecta a una aproximación asintótica a 2NlogN, se vea reflejado en la gráfica en forma de picos.

6.2 Pregunta 2
Graficas resultados QS los 3 distintos pivotes



Como se ha explicado en la pregunta anterior, el algoritmo QS no es constante ya que utiliza la función median. En esta gráfica se puede apreciar como el coste de utilizar el primer elemento o el medio como pivote tienen un coste muy similar. Sin embargo, la función median_stat es más óptima ya que se tienen en cuenta el primer, medio y último elemento seleccionando el más óptimo para el algoritmo, obteniendo un menor coste como resultado.

6.3 Pregunta 3
Casos mejor y peor MS y QS. ¿Modificaciones al code para calcularlo?

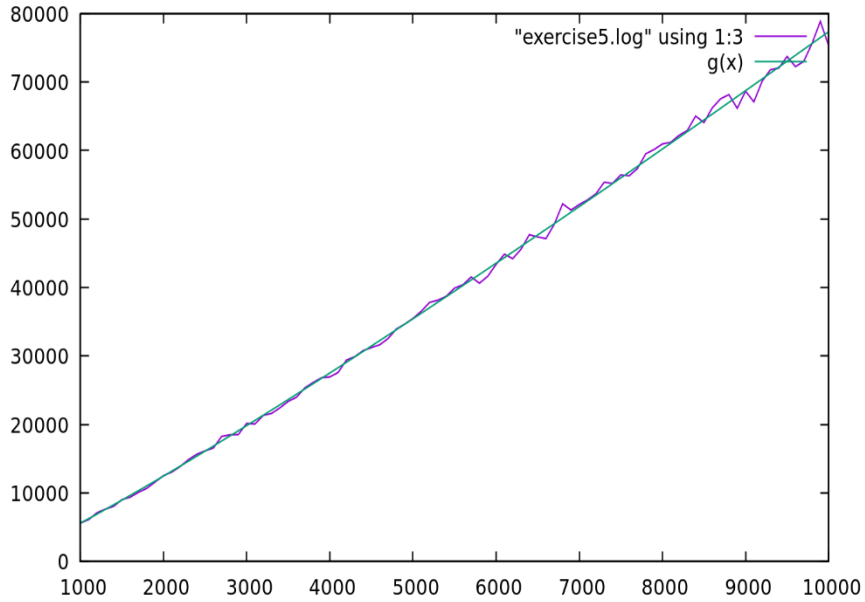
- **Caso mejor:**
  MS: $\frac{1}{2}NlogN$
  QS: $NlogN$

- **Caso peor:**
  MS: $NlogN$
  QS: $\frac{N^2}{2} - \frac{N}{2}$

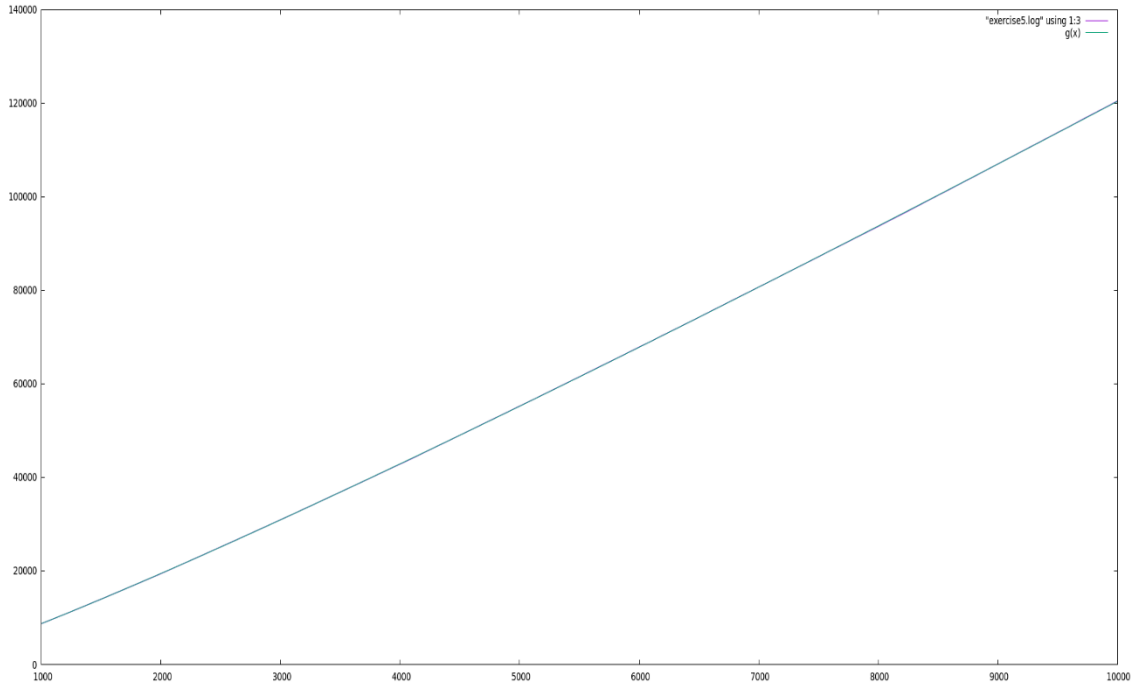
Nuestro código actual (excercise5.c) ya calcula el resultado de cada uno de los casos.

6.4 Pregunta 4
Comparar las empíricas con las teóricas.

QuickSort



MergeSort



En las gráficas como en la teoría se puede apreciar cómo QS es mucho más costoso (llegando incluso a 80000OBs para N=10000) que MS (roza las 12000OBs para N=10000). Sin embargo, QS es más eficiente en cuanto a memoria ya que este no utiliza tablas auxiliares como sí lo hace MS, teniendo que reservar en la función merge memoria dinámica para funcionar.

7. Conclusiones finales.

Continuando con la resolución de varios ejercicios relacionados con algoritmos de ordenación, en esta segunda práctica trabajaremos con Quicksort y MergeSort. En cada ejercicio implementaremos los algoritmos, así como las funciones de apoyo necesarias para su correcto funcionamiento. Una de las diferencias con la primera práctica es que ambos algoritmos son recursivos según lo visto en teoría por lo que habrá que pensar en una forma un poco distinta para contar el número de operaciones básicas realizadas.

Para nosotros, el objetivo de la práctica es familiarizarnos con los algoritmos de ordenación recursivos, así como el tiempo de ejecución de estos con el fin de buscar la mayor optimización y rendimiento posible. Apoyándonos y modificando los excercise_test de la práctica anterior podremos dar una tabla desordenada a cada algoritmo para que la ordene y comprobar el coste y operaciones básicas que realizan.