Analysis of Algorithms 2019/2020

Practice 2

Álvaro Castillo & Pablo Ernesto Soëtard, Group 1291

Code	Plots	Memory	Total

1. Introduction.

This second practice tries to experimentally determine the execution times of algorithms which use divide and conquer approach. The algorithms to be studied are MergeSort and QuickSort. Each algorithm will be analyzed on tables and the results obtained will be compared with the theoretical study of algorithm.

2. Objectives

The first two exercises are related to the Merge Sort algorithm while the next three are about Quick Sort.

2.1 Exercise 1

We are going to create a function *int mergesort(int* table, int ip, int iu)* that sorts a table with the Merge Sort algorithm and another one i*nt merge(int* table, int ip, int iu, int imiddle).* This second function will be a private one, just used by mergesort, to merge the different sub-arrays.

2.2 Exercise 2

We will have to change the method from the file exercise 5 and execute it in order to get data about the Merge Sort method implemented. We will have to plot that data and compare it with the theoretical one.

2.3 Exercise 3

We will program the algorithm Quick Sort by adding this functions *int quicksort(int* table, int ip, int iu)* which will hold the recursive calls, *int split(int* table, int ip, int iu,int *pos)* which will put the numbers that are less than the pivot at the left of the table and the ones that are greater at the right and *int median(int *table, int ip, int iu,int *pos)* which will calculate the pivot of the table. We will check the correct functioning by using the exercise 4 file.

2.4 Exercise 4

We will have to change the method from the file exercise 5 and execute it in order to get data about the Quick Sort method implemented. We will have to plot that data and compare it with the theoretical one.

2.5 Exercise 5

We will have to implement two new functions to calculate the pivot:

i*nt median avg(int *table, int ip, int iu, int *pos);*

*int median stat(int *table,int ip, int iu, int *pos);*

The first one will choose the average position in the array, while the second one will compare the elements that are in the first last and middle position and will return the position of the middle element. Then we will have to execute the exercise 5 file to get the number of basic operations (max, min, average) and average time of both routines.

3 Tools and Methodology

Our environment to program is Ubuntu. The tools that we have used are Atom and GitKraken to code, as well as github to share our code and Valgrind to check the memory leaks. We have also used gnuplot to plot the functions and gcc to compile all the files.

In general, implementing the functions was easy because, as we have learn in the theory lessons how the algorithms work, we have the pseudocode of both sorting methods on Moodle.

3.1 Exercise 1

The only difficulty with this exercise was counting the number of basic operations as the pseudocode on Moodle does not take it in account. We had to check also that the recursive calls to Merge Sort do not return an error and in that case return ERR.

3.2 Exercise 2

Obtaining the data was a simple task, while plotting was a bit messy as we had to introduce a lot of commands to plot everything in gnuplot.

3.3 Exercise 3

As with section 1, the difficulty was based on the count of basic operations. Furthermore, we had some problems with the split function as we did not change the pivot in the right way, so when executing the array was not sorted at all. We had to debug and correct the functioning of split.

3.4 Exercise 4

As in the exercise 2, obtaining the data was a easy, but plotting was a bit messy.

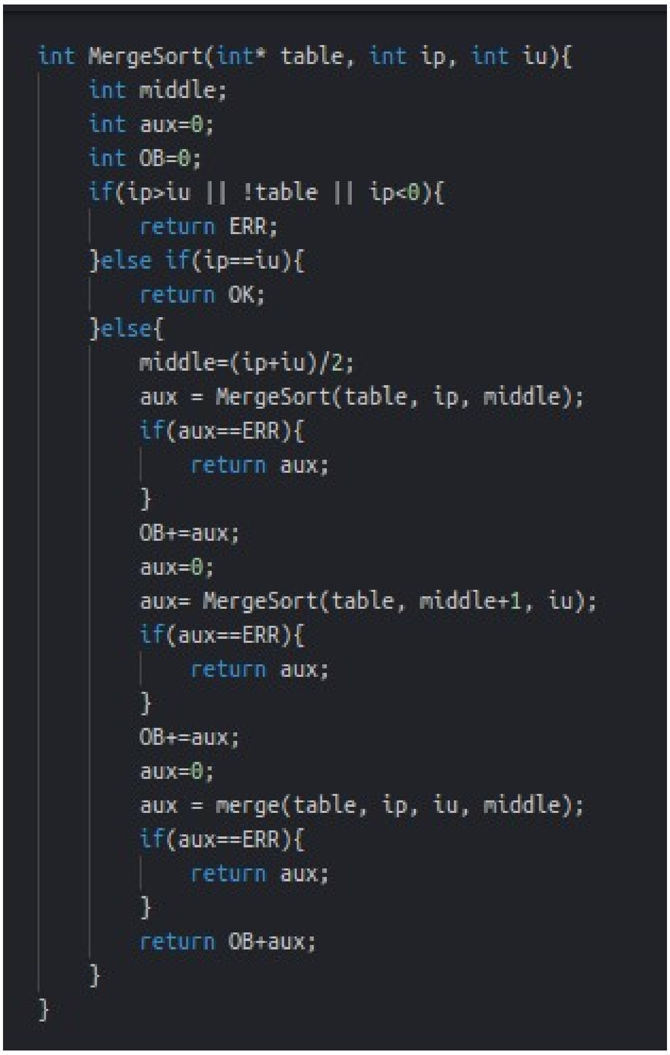
3.5 Exercise 5

Creating median_avg was easy while median_stat confusing, as we have to compare the elements of the table, not calculate a position with indexes. In addition we had to create a larger function as it compares elements and return the number of comparisons.

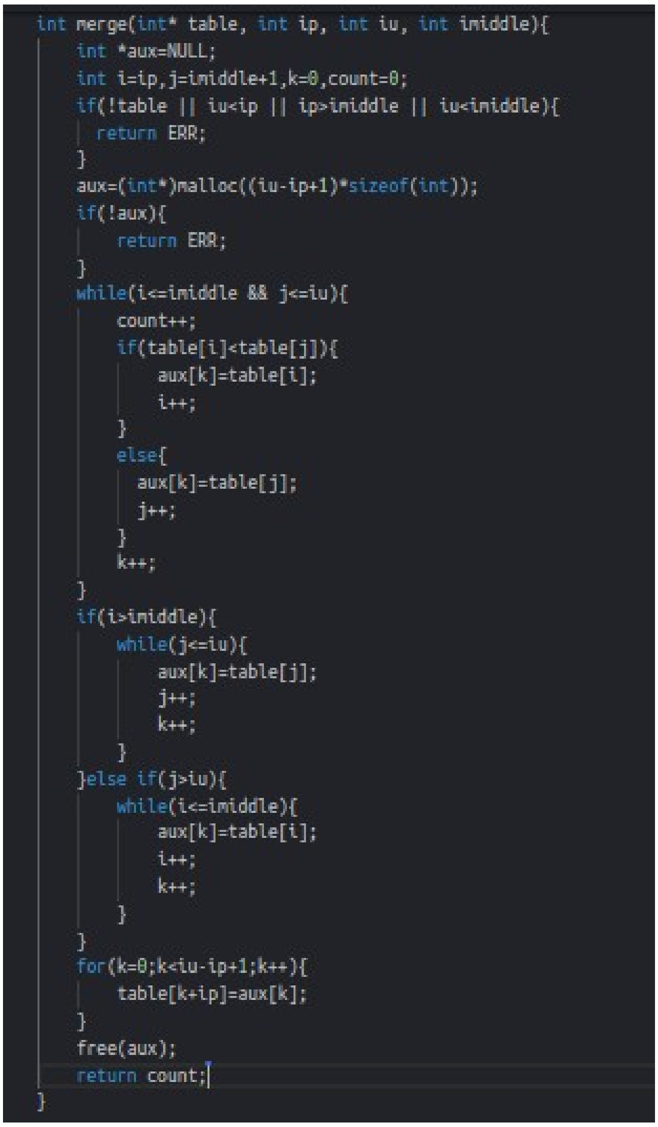
Retrieving the data for the plots and plotting was also a simple task.

4. Source code

4.1 Exercise 1

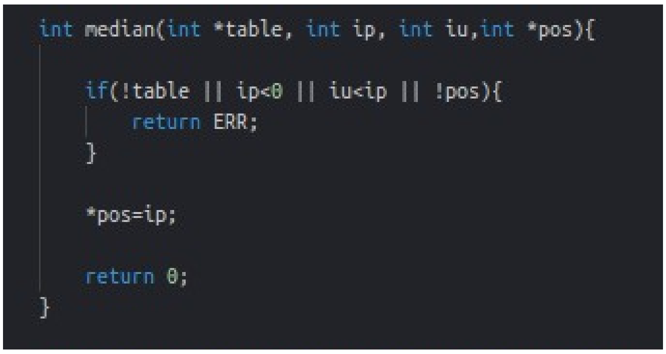


```
int MergeSort(int* table, int ip, int iu){
    int middle;
    int aux=0;
    int OB=0;
    if(ip>iu || !table || ip<0){
        return ERR;
    }else if(ip==iu){
        return OK;
    }else{
        middle=(ip+iu)/2;
        aux = MergeSort(table, ip, middle);
        if(aux==ERR){
            return aux;
        }
        OB+=aux;
        aux=0;
        aux= MergeSort(table, middle+1, iu);
        if(aux==ERR){
            return aux;
        }
        OB+=aux;
        aux=0;
        aux = merge(table, ip, iu, middle);
        if(aux==ERR){
            return aux;
        }
        return OB+aux;
    }
}
```

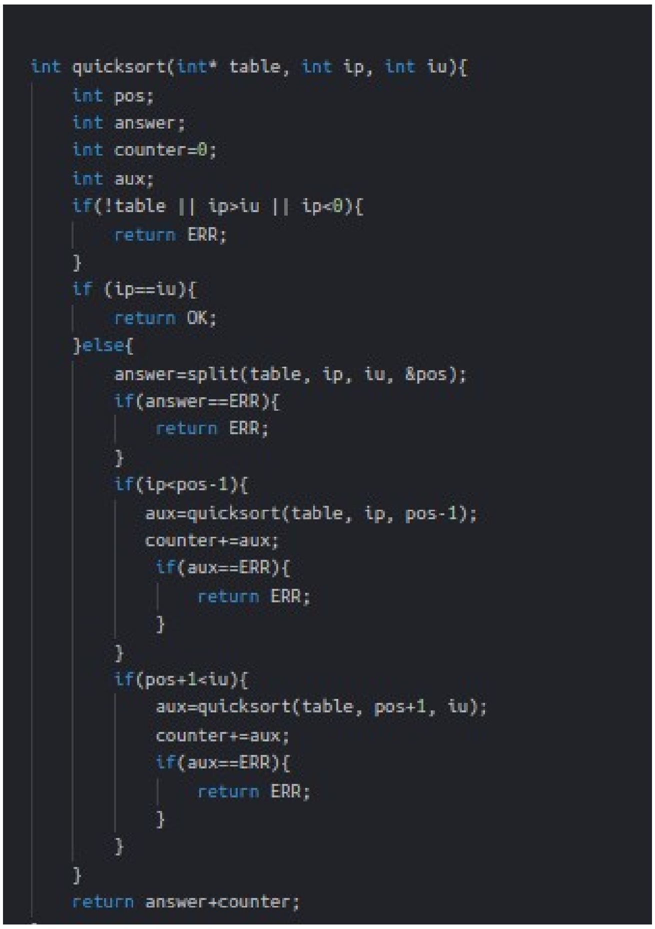


```c
int merge(int* table, int ip, int iu, int imiddle){
    int *aux=NULL;
    int i=ip,j=imiddle+1,k=0,count=0;
    if(!table || iu<ip || ip>imiddle || iu<imiddle){
      return ERR;
    }
    aux=(int*)malloc((iu-ip+1)*sizeof(int));
    if(!aux){
        return ERR;
    }
    while(i<=imiddle && j<=iu){
        count++;
        if(table[i]<table[j]){
            aux[k]=table[i];
            i++;
        }
        else{
          aux[k]=table[j];
          j++;
        }
        k++;
    }
    if(i>imiddle){
        while(j<=iu){
            aux[k]=table[j];
            j++;
            k++;
        }
    }else if(j>iu){
        while(i<=imiddle){
            aux[k]=table[i];
            i++;
            k++;
        }
    }
    for(k=0;k<iu-ip+1;k++){
        table[k+ip]=aux[k];
    }
    free(aux);
    return count;
}
```

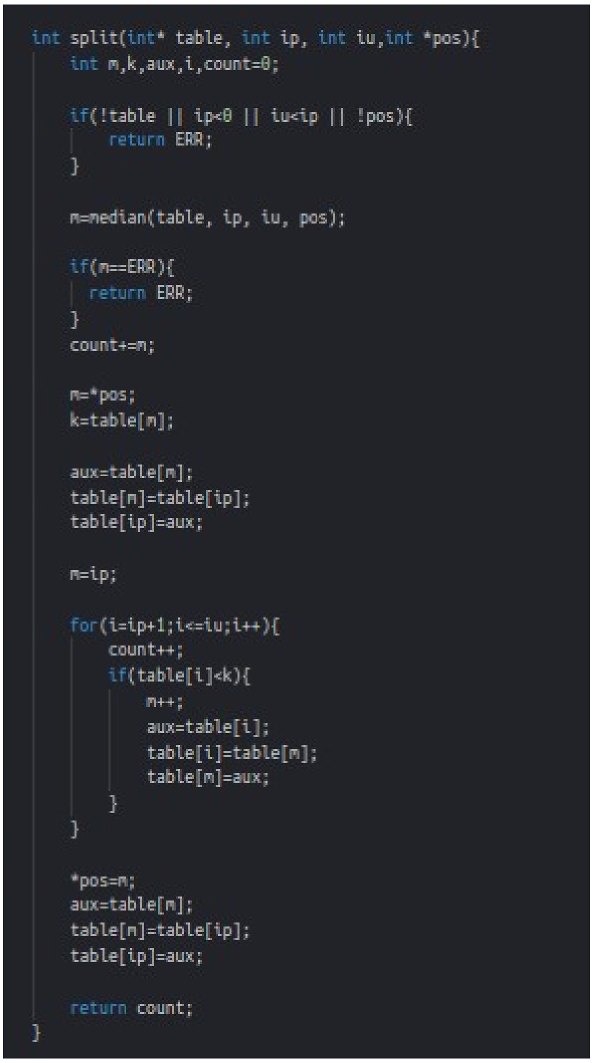
4.3 Exercise 3



```c
int median(int *table, int ip, int iu,int *pos){

    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }

    *pos=ip;

    return 0;
}
```

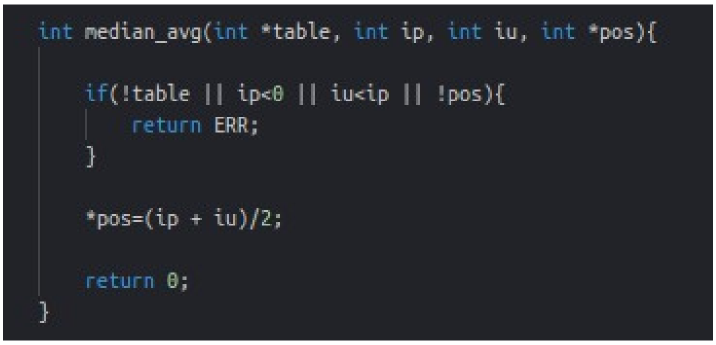


```
int quicksort(int* table, int ip, int iu){
    int pos;
    int answer;
    int counter=0;
    int aux;
    if(!table || ip>iu || ip<0){
        return ERR;
    }
    if (ip==iu){
        return OK;
    }else{
        answer=split(table, ip, iu, &pos);
        if(answer==ERR){
            return ERR;
        }
        if(ip<pos-1){
           aux=quicksort(table, ip, pos-1);
           counter+=aux;
            if(aux==ERR){
                return ERR;
            }
        }
        if(pos+1<iu){
            aux=quicksort(table, pos+1, iu);
            counter+=aux;
            if(aux==ERR){
                return ERR;
            }
        }
    }
    return answer+counter;
```

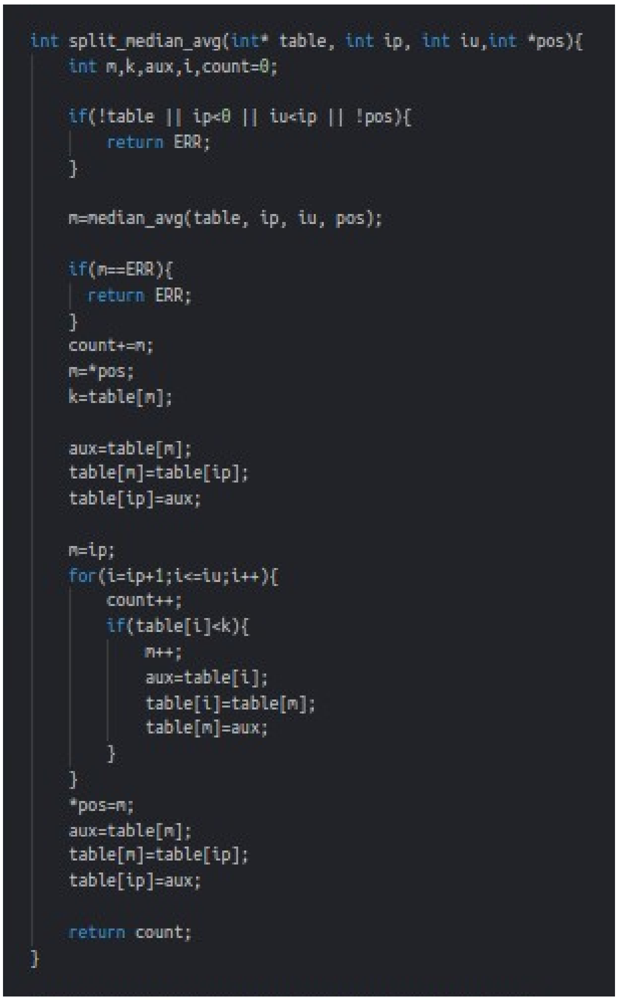


```
int split(int* table, int ip, int iu,int *pos){
    int m,k,aux,i,count=0;

    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }

    m=median(table, ip, iu, pos);

    if(m==ERR){
      return ERR;
    }
    count+=m;

    m=*pos;
    k=table[m];

    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    m=ip;

    for(i=ip+1;i<=iu;i++){
        count++;
        if(table[i]<k){
            m++;
            aux=table[i];
            table[i]=table[m];
            table[m]=aux;
        }
    }

    *pos=m;
    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    return count;
}
```

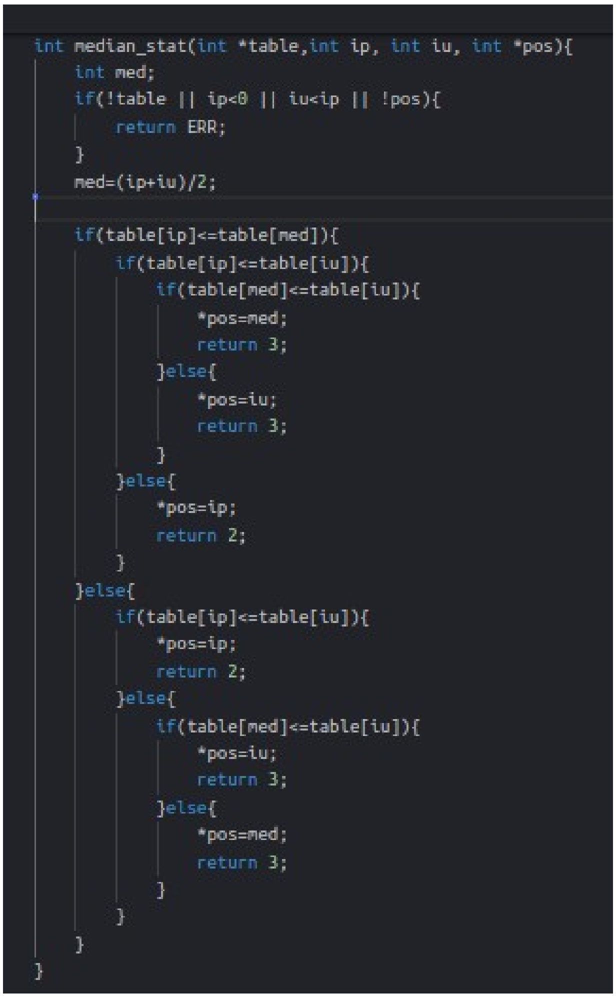
4.5 Exercise 5



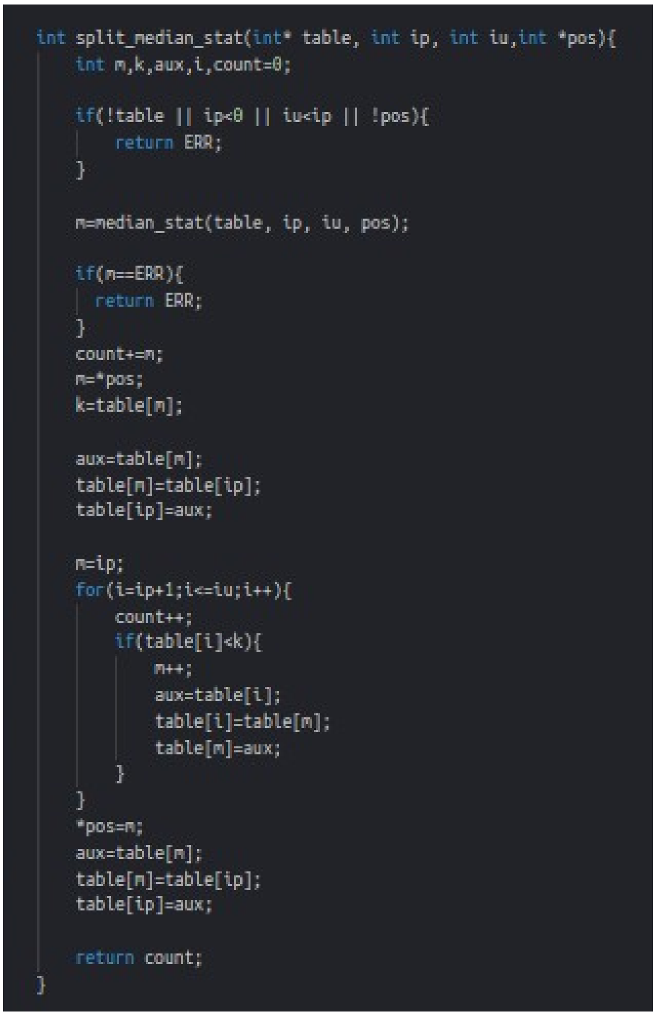
```c
int median_avg(int *table, int ip, int iu, int *pos){

    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }

    *pos=(ip + iu)/2;

    return 0;
}
```



```c
int split_median_avg(int* table, int ip, int iu,int *pos){
    int m,k,aux,i,count=0;

    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }

    m=median_avg(table, ip, iu, pos);

    if(m==ERR){
      return ERR;
    }
    count+=m;
    m=*pos;
    k=table[m];

    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    m=ip;
    for(i=ip+1;i<=iu;i++){
        count++;
        if(table[i]<k){
            m++;
            aux=table[i];
            table[i]=table[m];
            table[m]=aux;
        }
    }
    *pos=m;
    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    return count;
}
```



```c
int median_stat(int *table,int ip, int iu, int *pos){
    int med;
    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }
    med=(ip+iu)/2;

    if(table[ip]<=table[med]){
        if(table[ip]<=table[iu]){
            if(table[med]<=table[iu]){
                *pos=med;
                return 3;
            }else{
                *pos=iu;
                return 3;
            }
        }else{
            *pos=ip;
            return 2;
        }
    }else{
        if(table[ip]<=table[iu]){
            *pos=ip;
            return 2;
        }else{
            if(table[med]<=table[iu]){
                *pos=iu;
                return 3;
            }else{
                *pos=med;
                return 3;
            }
        }
    }
}
```

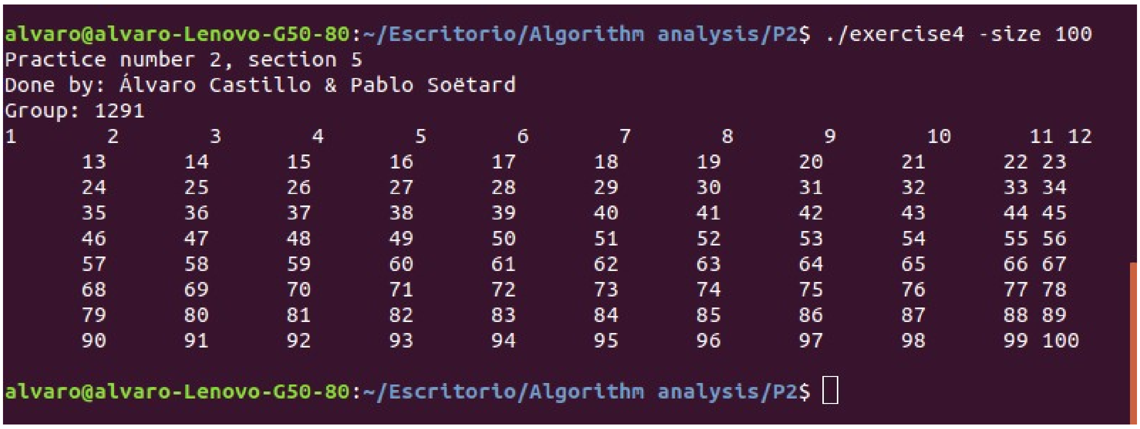


```c
int split_median_stat(int* table, int ip, int iu,int *pos){
    int m,k,aux,i,count=0;

    if(!table || ip<0 || iu<ip || !pos){
        return ERR;
    }

    m=median_stat(table, ip, iu, pos);

    if(m==ERR){
      return ERR;
    }
    count+=m;
    m=*pos;
    k=table[m];

    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    m=ip;
    for(i=ip+1;i<=iu;i++){
        count++;
        if(table[i]<k){
            m++;
            aux=table[i];
            table[i]=table[m];
            table[m]=aux;
        }
    }
    *pos=m;
    aux=table[m];
    table[m]=table[ip];
    table[ip]=aux;

    return count;
}
```

5. Results, Plots

5.1 Exercise 1

We have executed the file exercise4.c to check that the algorithm works correctly. Here we have an example of the output:

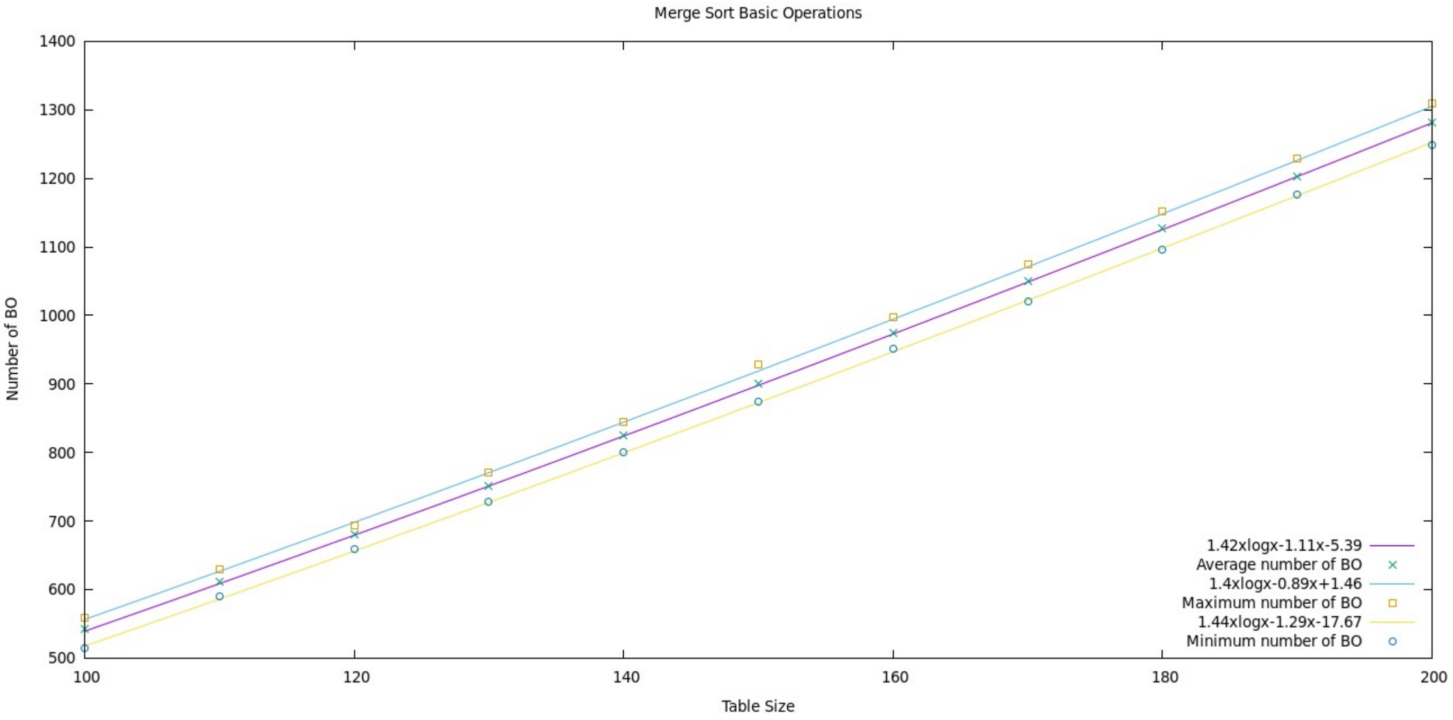


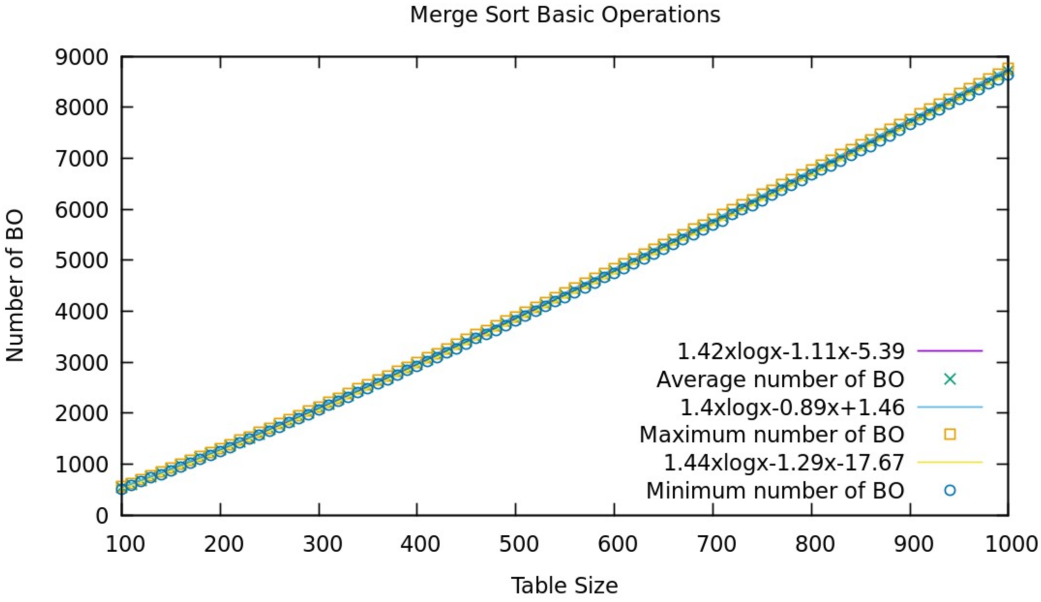
```
alvaro@alvaro-Lenovo-G50-80:~/Escritorio/Algorithm analysis/P2$ ./exercise4 -size 100
Practice number 2, section 5
Done by: Álvaro Castillo & Pablo Soëtard
Group: 1291
1       2       3       4       5       6       7       8       9       10      11 12
      13      14      15      16      17      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

alvaro@alvaro-Lenovo-G50-80:~/Escritorio/Algorithm analysis/P2$ 
```

We can see that it orders the array successfully.

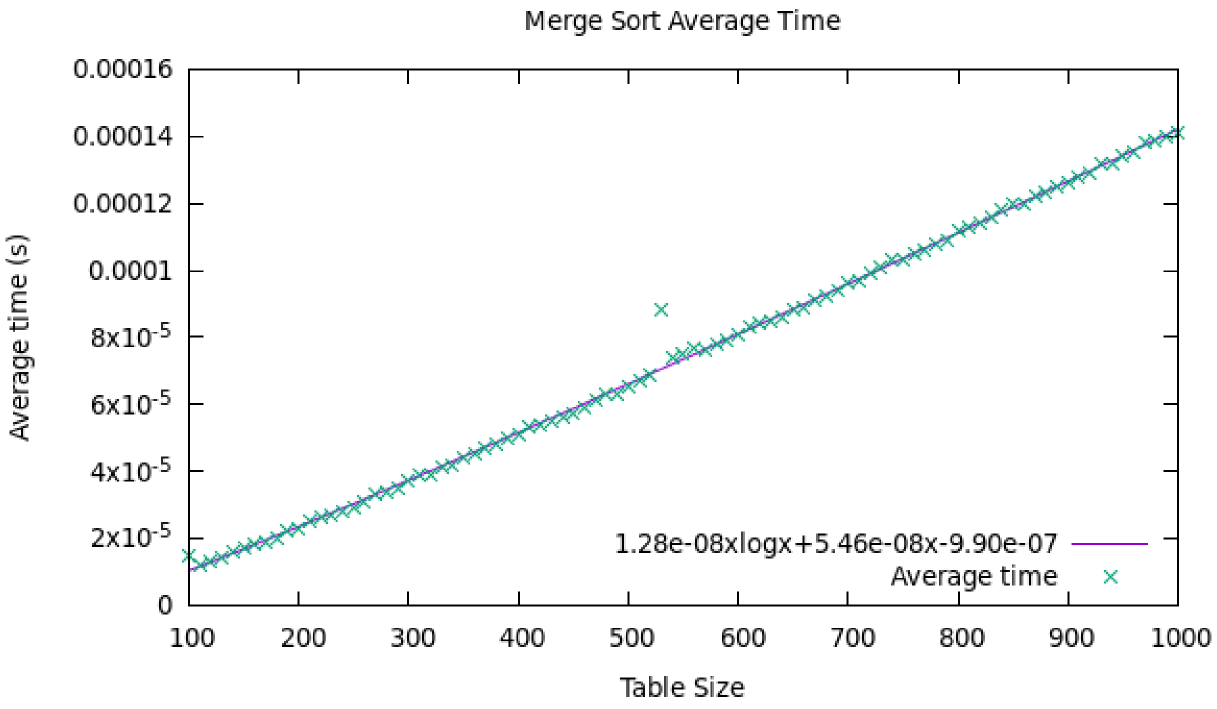
5.2 Exercise 2

We have executed the file exercise5.c to get the data and it gave an output without any error. We have generated data for 10000 permutations, taking sizes from 100 to 1000 in increments of 10.





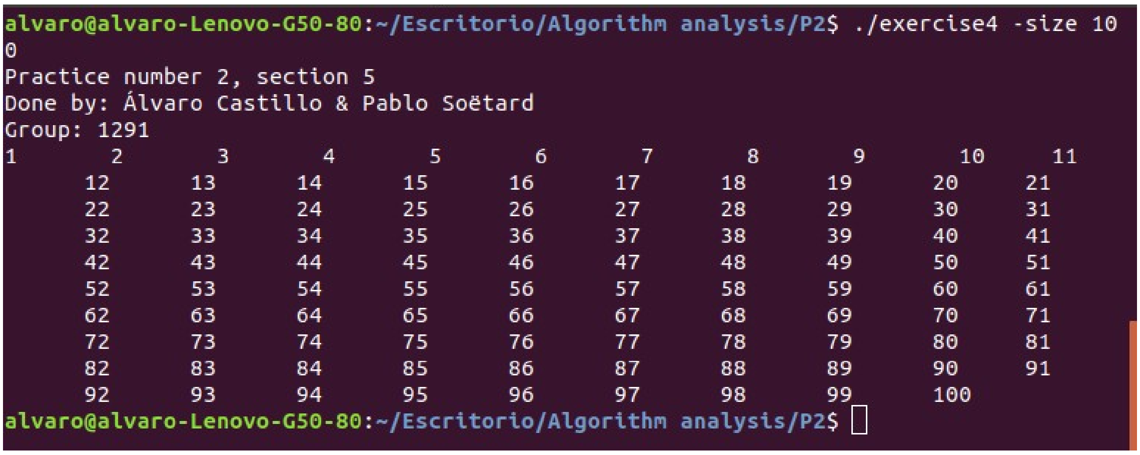
In the x axis the size of the array is represented, and in the y axis the number of BOs is represented. In the first plot we can see closer the difference between the maximum, minimum and average number of basic operations for Merge Sort, especially we can notice that the minimum number of BO is less than the average and this one is less than then maximum. While on the second plot we can see that each case fits in the same function (axlogx + bx+c) with different parameters a,b,c and they tend to be the same. The purple line fits the average number of BO, the blue one fits the maximum and the yellow one, the minimum. We can see that all of them seem to be coherent with the theoretical predictions.



In the x axis the size of the array is represented, and in the y axis the time in seconds. We can see that it also them seems to be coherent with the theoretical predictions, although there are some points that differ from the regression line like 530 or 100.

5.3 Exercise 3

We have executed the file exercise4.c to check that the algorithm works correctly. Here we have an example of the output:

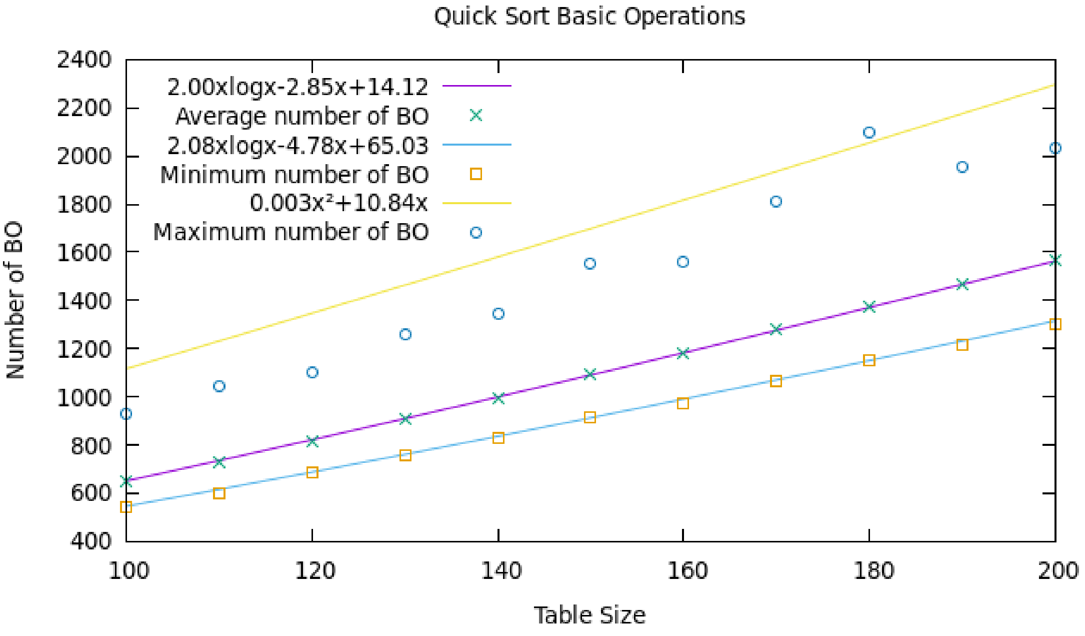


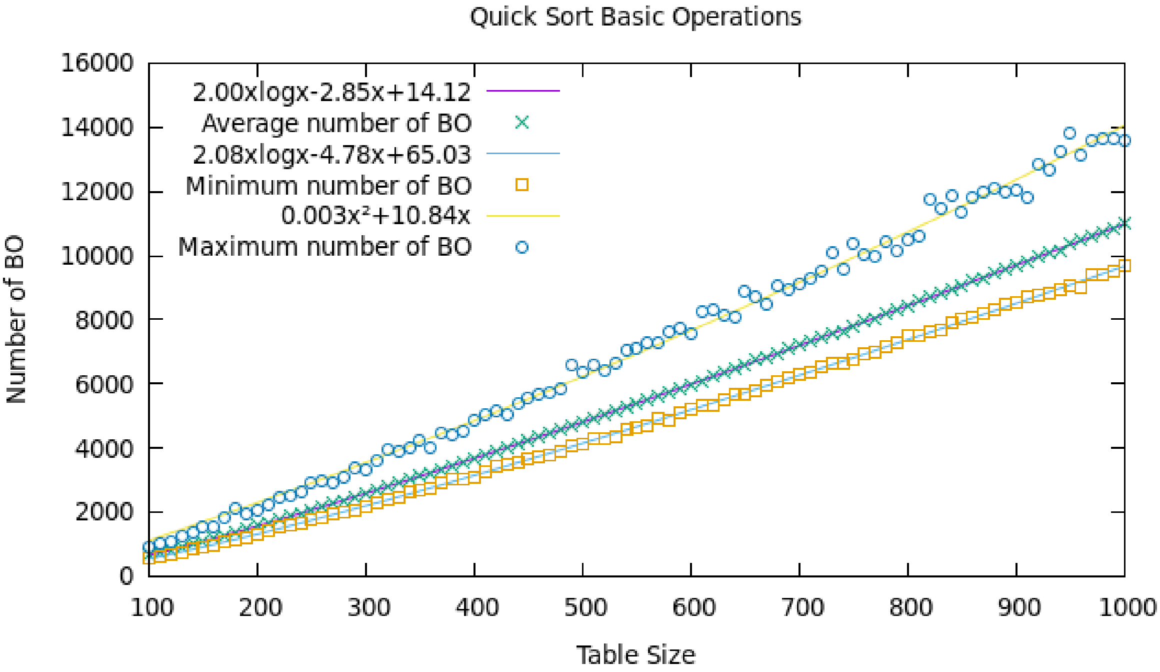
```
alvaro@alvaro-Lenovo-G50-80:~/Escritorio/Algorithm analysis/P2$ ./exercise4 -size 10
0
Practice number 2, section 5
Done by: Álvaro Castillo & Pablo Soëtard
Group: 1291
1       2       3       4       5       6       7       8       9       10      11
      12      13      14      15      16      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
alvaro@alvaro-Lenovo-G50-80:~/Escritorio/Algorithm analysis/P2$ 
```

We can see that it orders the array successfully.

5.4 Exercise 4

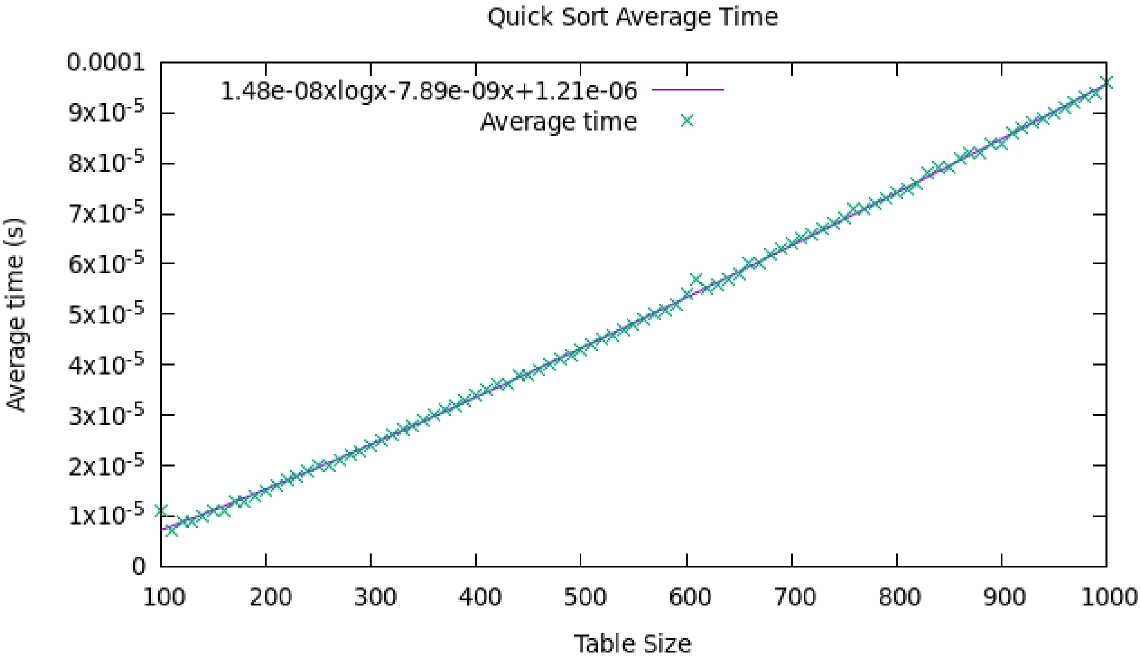
We have executed the file exercise5.c to get the data and it gave an output without any error. We have generated data for 10000 permutations, taking sizes from 100 to 1000 in increments of 10.





In the x axis the size of the array is represented, and in the y axis the number of BOs . In the first plot we can see closer the difference between the maximum, minimum and average number of basic operations for Quick Sort, especially we can notice that the minimum number of BO is less than the average and this one is less than then maximum. We can also see that our maximum number of operations data does not fit really well in a quadratic function. This is because there is a very little probability of having the worst case, wich will be to have the table already sorted.

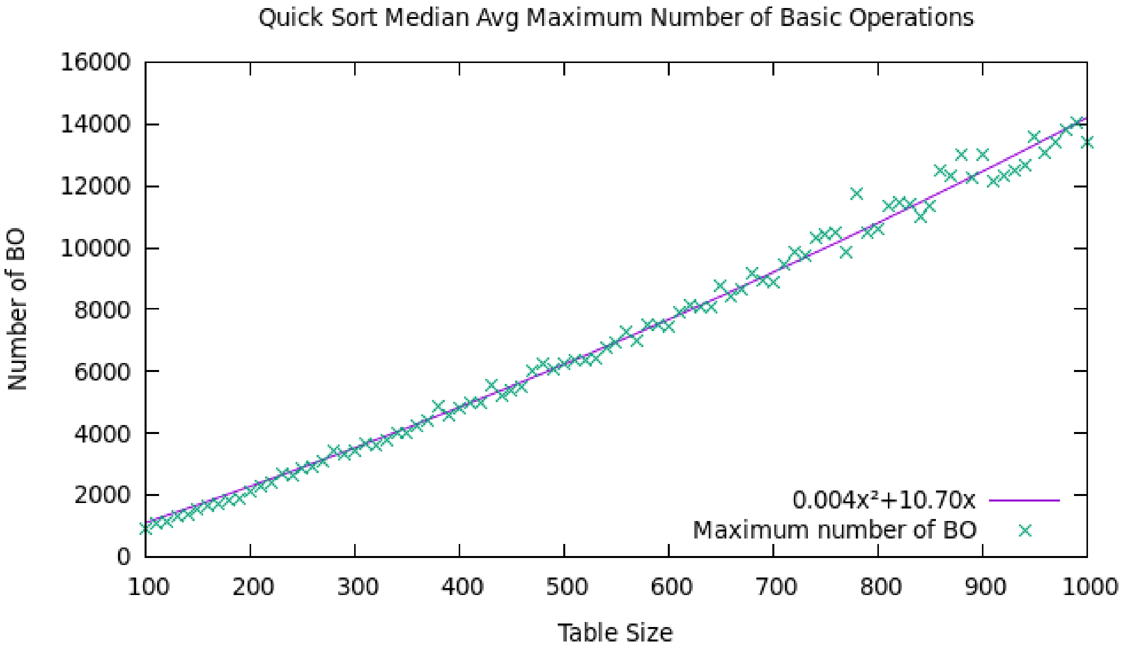
While on the second plot we can see that best case and average case fits in the same function (axlogx + bx+c) with different parameters a,b,c and worst case fits in a quadratic one (ax²+bx). The purple line fits the average number of BO, the blue one fits the minimum and the yellow one, the maximum. We can see that all of them seem to be coherent with the theoretical predictions.

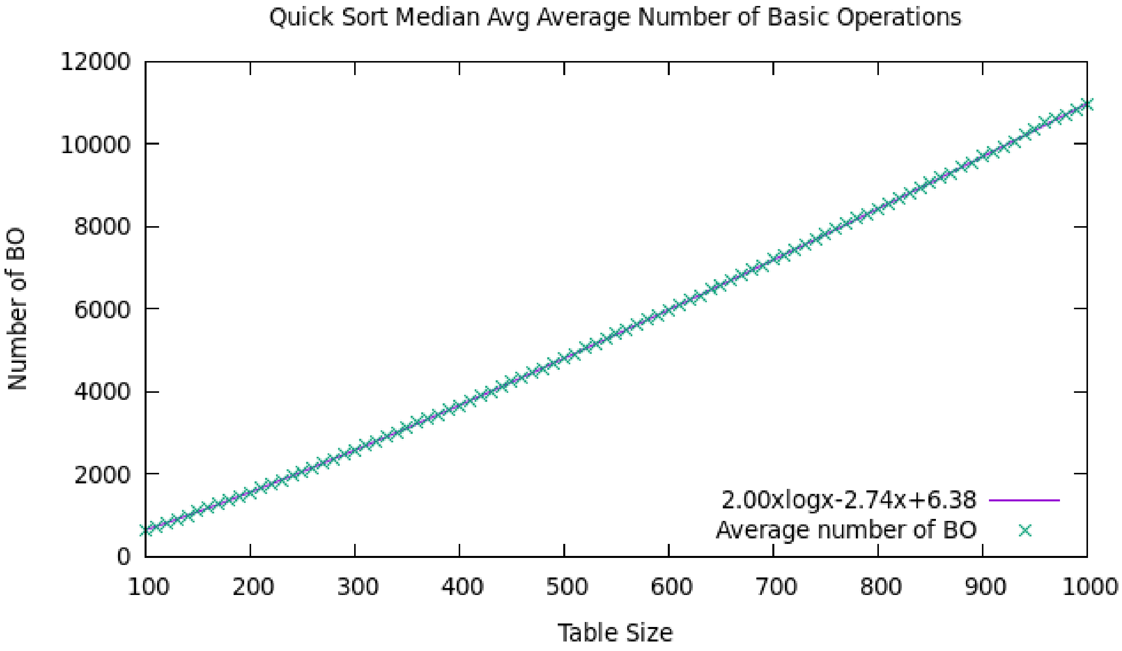


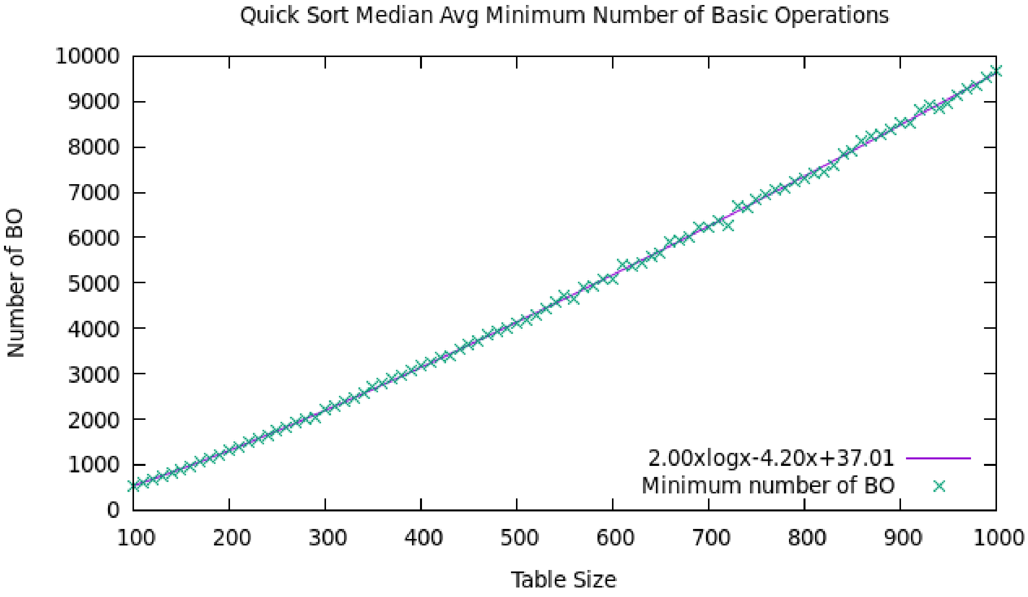
In the x axis the size of the array is represented, and in the y axis the time in seconds in the Average Time. We can see that all of them seems to be coherent with the theoretical predictions, although there are some points that differ from the regression line, for example 100, 610.

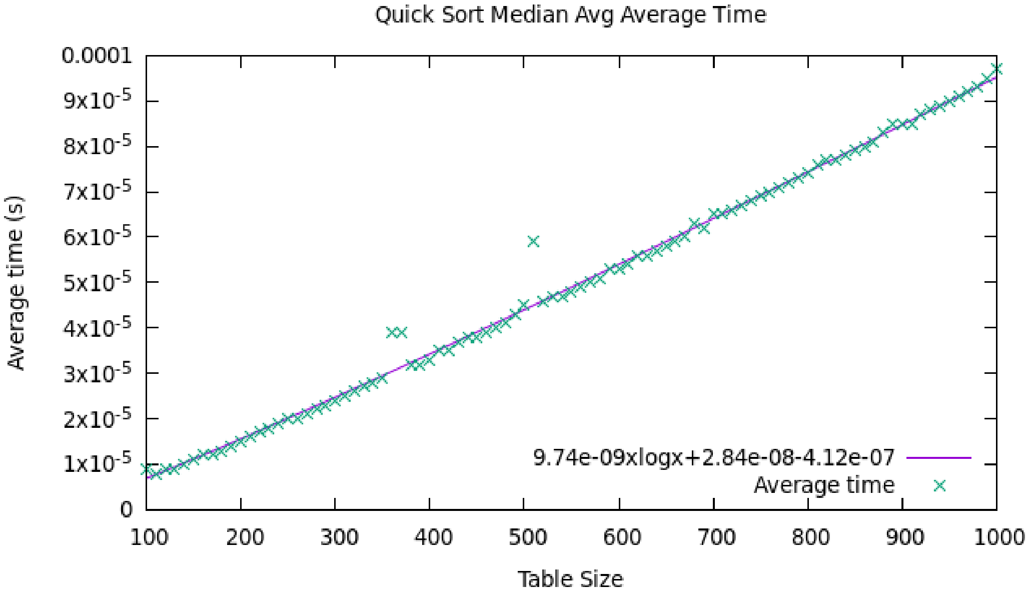
5.5 Exercise 5

We have executed the file exercise5.c to get the data and it gave an output without any error. We have generated data for 10000 permutations, taking sizes from 100 to 1000 in increments of 10.

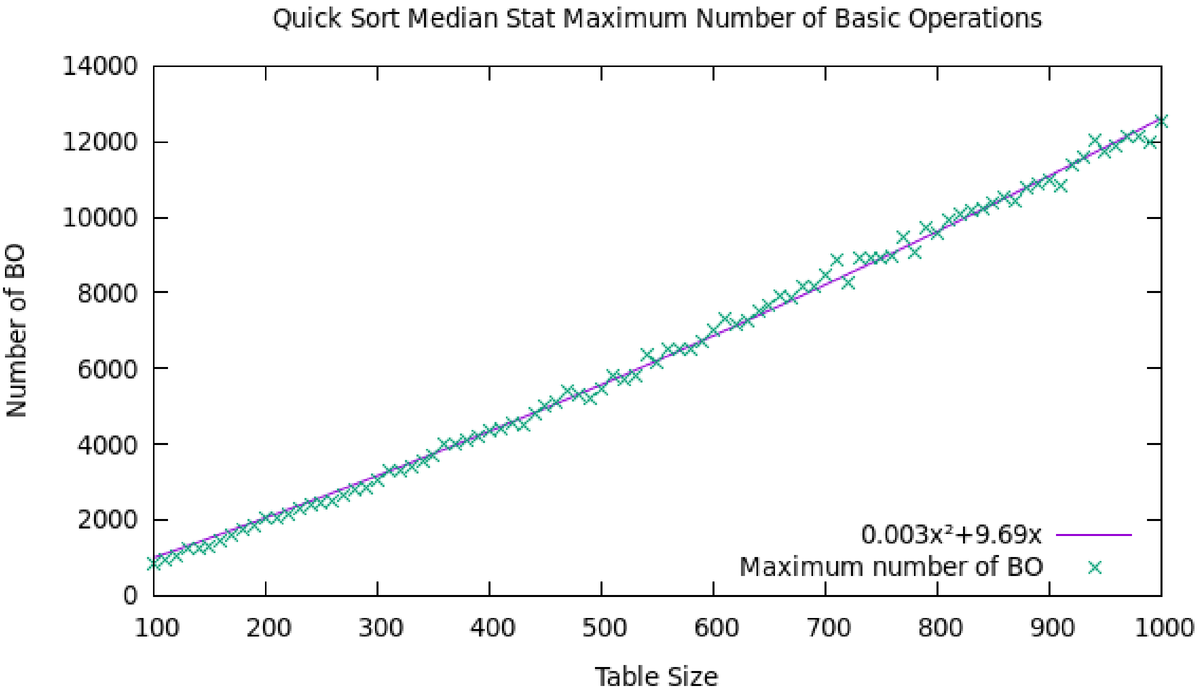


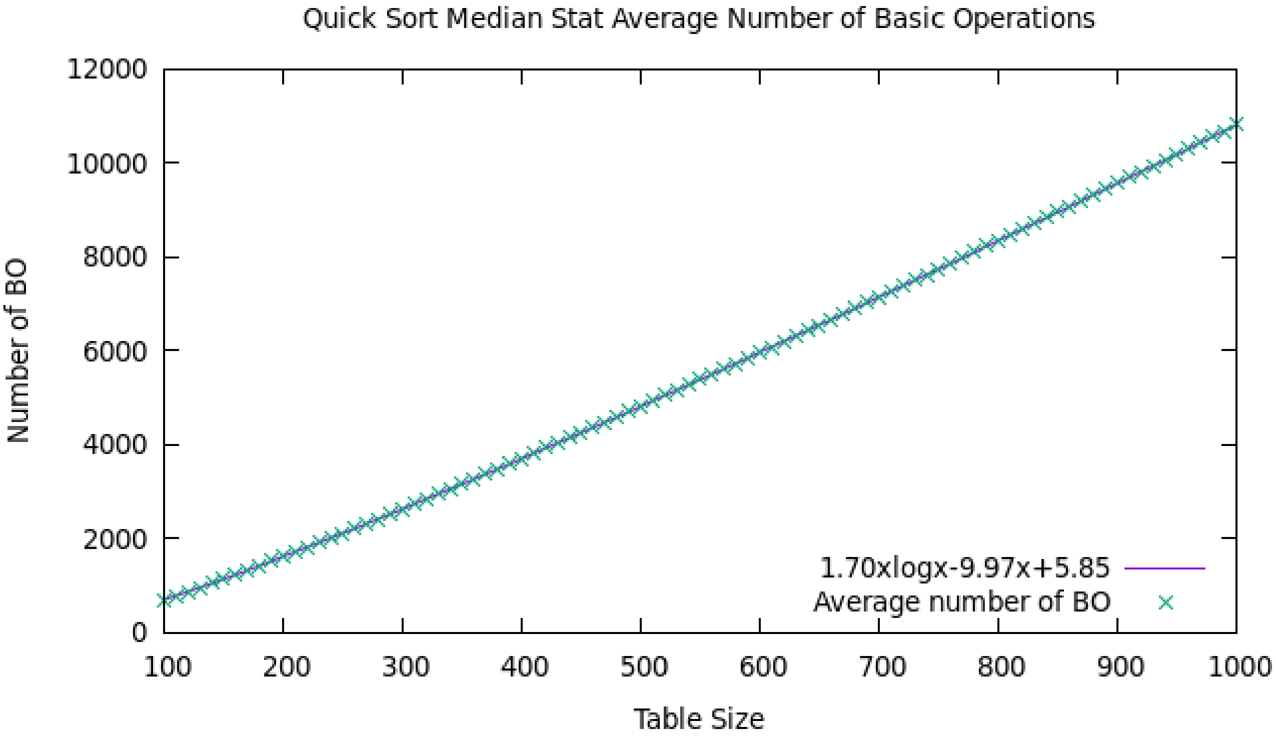


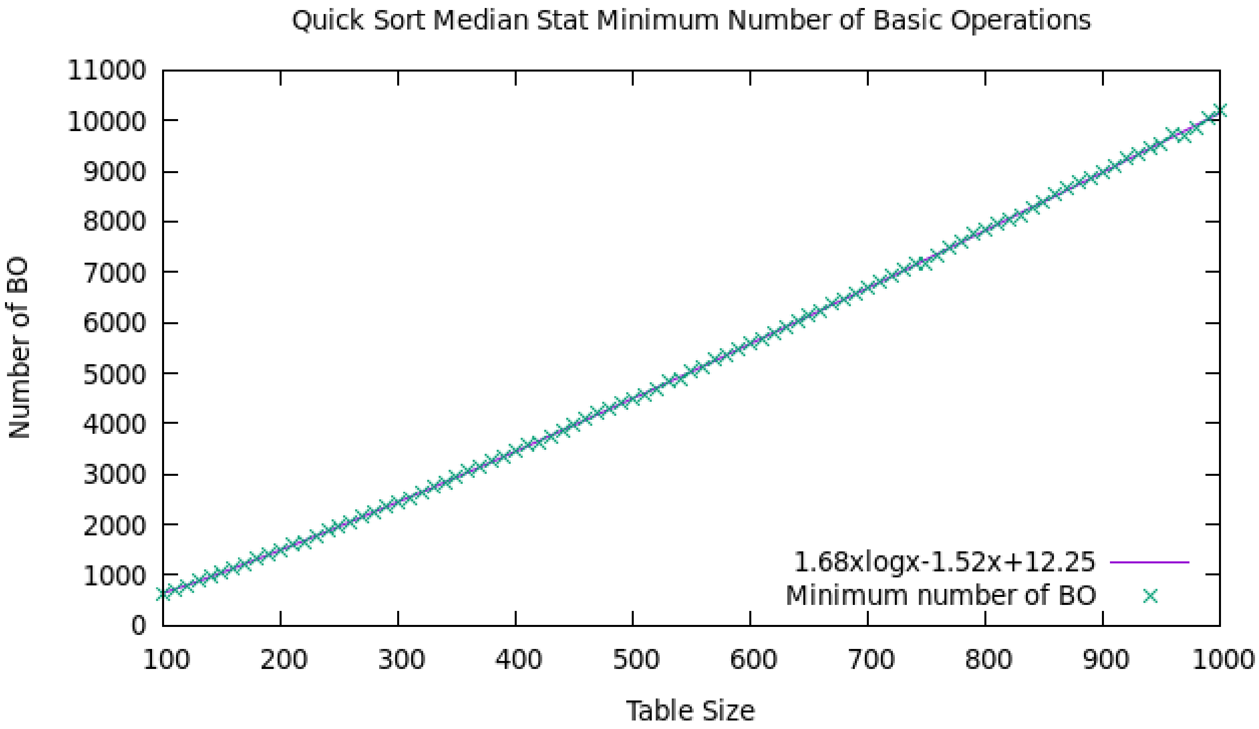


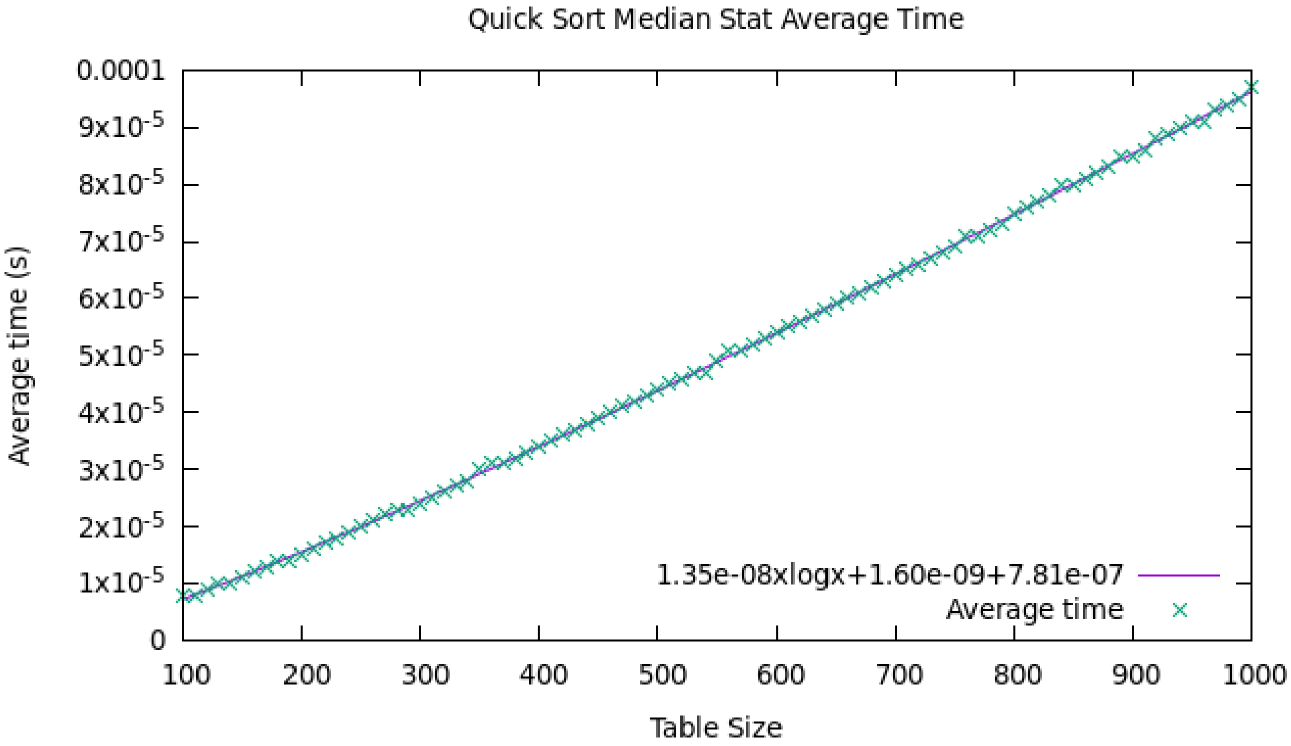


All these plots are from the Quick Sort algorithm, implemented with the median avg method. In the x axis the size of the array is represented, and in the y axis the time in seconds in the Average Time plot and in the rest the number of BOs are represented. We can see that all of them seems to be coherent with the theoretical predictions and the average and minimum fits perfectly with a xlogx +O(x) function, while the maximum fits to a quadratic function (ax^2+bx). On the other hand, there are some points on the average time plot and maximum number of BO plot, that differ from the regression line. We can also see that our maximum number of operations data does not fit really well in a quadratic function. This is because there is a very little probability of having the worst case, wich will be to have the table already sorted.


Quick Sort Median Stat Maximum Number of Basic Operations
14000
12000
10000
8000
6000
4000
2000
0
100
200
300
400
500
600
700
800
900
1000
Number of BO
Table Size
0.003x²+9.69x
Maximum number of BO


Quick Sort Median Stat Average Number of Basic Operations
12000
10000
8000
6000
4000
2000
0
100
200
300
400
500
600
700
800
900
1000
Number of BO
Table Size
1.70xlogx-9.97x+5.85
Average number of BO





All these plots are from the Quick Sort algorithm, implemented with the median stat method. In the x axis the size of the array is represented, and in the y axis the time in seconds in the Average Time plot and in the rest the number of BOs are represented. We can see that all of them seems to be coherent with the theoretical predictions and the average and minimum fits perfectly with a xlogx +O(x) function, while the maximum fits to a quadratic function (ax^2+bx). On the other hand, there are some points on the average time plot and maximum number of BO plot, that differ from the regression line.

6. Answers to theoretical Questions.

6.1 The average number of basic operations for the algorithms are: MergeSort (1.42xlogx – 1.11x -5.39), QuickSort (2.00xlogx -2.85x +14.12), QuickSort_Avg (2.00xlogx – 2.74x +6.38), QuickSort_Stat (1.70xlogx -9.97x +5.85). For MergeSort the theoretical average case is xlogx + O(x) and for QuickSort 2xlogx + O(x), so we can see that the experimental and theoretical values match pretty much. If the traces of the graphs are sharp, it could be because of the input permutation, as there could be some of them that were already sorted or really unsorted.

6.2 As stated above, the average basic operations for the different versions of QuickSort are: QuickSort_Median (2.00xlogx+O(x)), QuickSort_Avg (2.00xlogx+O(x)), QuickSort_Stat (1.70xlogx+O(x)). We can see that there is not a huge difference between the pivot selection method and performance ratio. This is because the arrays passed to the algorithms are randomly generated, thus different methods of selecting a pivot should result in almost the same performance in the overall case. Although we could state that the stat method performs better than the other two median selection functions, because it has more chances to select a middle point in respect to the values of the array.

6.3 For MergeSort we have a best case of 1.44xlogx – 1.29x – 17.67 and a worst case of 1.40xlogx – 0.89x + 1.46. And for QuickSort a best case of 2.08xlogx – 4.78x + 65.03 and a worst case of $0.003x^2+10.84x$. For QuickSort with median average we have a best case of 2xlogx – 2.74x + 6.38 and a worst case of $0.004x^2+10.70x$. Finally, for QuickSort median stat we get a best case of 1.68xlogx – 1.52x + 12.25 and for the worst case $0.003x^2+9.69x$.

Theoretically Merge Sort has as best case and worst case NlogN+O(N), while Quick Sort $N^2/2-N/2$ as worst case and 2NlogN + O(N).

To strictly calculate each case we would have to sort arrays that have a big amount of elements, in order to get more accurate results.

6.4 As shown with the experiments, MergeSort is more efficient than QuickSort in the worst case. That is coherent with the theoretical predictions, as MergeSort has a worst case of xlogx + O(x) and QuickSort $x^2/2 – x/2$. In the average case and minimum, we get that Merge Sort is more efficient than Quick Sort as we can see in the fitted functions. Theoretically we have studied that Quick Sort is more efficient than Merge Sort on the best and average case, but they have the same complexity 2xlogx + O(x) and xlogx + O(x) respectively.

Regarding the space efficiency, MargeSort is less efficient than QuickSort because it allocates an auxiliary array in order to execute the combine function.

7. Final Conclusions.

Working on this practice, we have understand more deeply how local sorting algorithms work, in particular Merge Sort and Quick Sort with its different’s versions.

We have had the opportunity to put in practice what we have learnt in class, and therefore reinforcing our learning process.

Also we had to overcome some difficulties as coding errors or plots, that had lead us to a better understanding of the tools that we used to accomplish this practice.

After analyzing the results of our algorithms, we had to figure out why some of them were not showing as expected, and finally we came across a logical conclusion.

To sum up, we have demonstrated the theoretical calculus that we made in class by working on this practice, and learn how to use new tools such as Gnuplot, as well as overcoming some problems that appeared during the process.