Algorithm Analysis 2020/2021

Practice 2

Pablo Soto Martín and Sergio Leal Andrés, Group 1251

Code	Plots	Memory	Total

1. Introduction

In this practice of algorithm analysis, using all the functions that we implemented for the file times.c in practice 1, in order to take clock times and count ob times for the algorithms IS and IS inv, but this time we will use them for another kind of algorithms, that are not local. These ones are MergeSort, QuickSort and QuickSort without tail recursion, that are recursive algorithms that at first sight seem to be better in terms of basic operations(KC) and clock times. So our global objective in this practice is to compare these three algorithms between them and see how efficient they are.

2. Objectives

2.1 Section 1

Given a natural number, N, and creating an array with the first N integer numbers starting by 0, sorted in random way by the function generate_perm, the objective is to create a function called MergeSort, that sorts the array in a recursive way obtaining also the number of times the OB (in this case will be the comparison operation, that gets counted in an auxiliary function named merge, that according to its name it will combine the arrays that we got from dividing the given one multiple times) was executed.

2.2 Section 2

The objective of section 2 is to, using the functions of times.c of practice 1, be able to know if our algorithm, MergeSort in this case, works, representing the best, worst, average, and the clock time in milliseconds.

2.3 Section 3

Given a natural number, N, and creating an array with the first N integer numbers starting by 0, sorted in random way by the function generate_perm, the objective is to create a function called QuickSort, that sorts the array in a recursive way obtaining also the number of times the OB (in this case will be the comparison operation, that gets counted in an auxiliary function named Split, that using a pivot that in this practice will be the first element will sort a part of the array) was executed.

2.4 Section 4

The objective of section 4 is to, using the functions of times.c of practice 1, be able to know if our algorithm, QuickSort, works, representing the best, worst, average, and the clock time in milliseconds.

2.5 Section 5

In section 5, our objective is to design a funcion, quicksortntr, that is very similar to quicksort and the only difference between them is that quicksortntr will not have tail recursion, which can be effective in order to avoid overhead produced by high number of function calls.

3 Tools and Methodology

The environment that we are going to use is Linux(Ubuntu 20.04 LTS).

All of the sections are going to be developed in Visual Studio Code, and shared between the two members of the group using an online repository (GITHUB). Inside VS code we are going to use the C compiler to compile and the internal console that vs code has to run the exercises. Besides VS code, in order to check the memory use we are going to use the valgrind tool, which is going to tell us how many bytes of memory we are allocating and how many we are releasing.

Our methodology will consist in compiling, executing, and using the valgrind tool manually for each exercise before going to the next one in order to check if it is correct before continuing with the next one. At the end, using the makefile, in the VS console we will make sure all of our programs compile and work correctly, using also the appropriate memory check tool(Valgrind). We have also used Sagemath as a tool for obtaining the values of the approximated regressions of the data.

3.1 Section 1

We only used the common tools described at the beginning.

3.2 Section 2

Apart from the common tools mentioned at the beginning we used Jupyter Notebook to, by using Python 3's library Matplotlib and Seaborn, plot the graphs that compare the size of the array with the max OB, min OB, average OB and execution time of the MergeSort function. In order to plot them, we created a csv with the exercise data and then read it as a pandas dataframe. Then, we could plot it thanks to the function matplotlib.pyplot.plot().

3.3 Section 3

We only used the common tools described at the beginning.

3.4 Section 4

Apart from the common tools mentioned at the beginning we used Jupyter Notebook to, by using Python 3's library Matplotlib and Seaborn, plot the graphs that compare the size of the array with the max OB, min OB, average OB and execution time of the QuickSort function. In order to plot them, we created a csv with the exercise data and then read it as a pandas dataframe. Then, we could plot it thanks to the function matplotlib.pyplot.plot().

3.5 Section 5

Apart from the common tools mentioned at the beginning we used Jupyter Notebook to, by using Python 3's library Matplotlib and Seaborn, plot the graphs that compare the size of the array with the max OB, min OB, average OB and execution time of the QuickSortntr function to the ones we got in section 4. In order to plot them, we created a csv with the exercise data and then read it as a pandas dataframe. Then, we could plot it thanks to the function matplotlib.pyplot.plot().

4. Source code

4.1 Section 1

```
int MergeSort(int* table, int ip, int iu){

 int M=0,count=0,count2=0;

 if (!table || ip<0 || iu<0 || ip>iu) return ERR;

 if (ip == iu) return OK;



 M= ((ip+iu) - ((ip+iu) %2))/2;



 count= MergeSort(table,ip,M);

 if (count == ERR) return ERR;

 count2 = MergeSort(table,M+1,iu);

 if (count2 == ERR) return ERR;

 count += count2;

 count2 = merge(table,ip,iu,M);

 if(count2 == ERR) return ERR;

 return (count + count2);

}

int merge(int *table, int ip, int iu, int imiddle){

 int *t=NULL;

 int i=0,j=0,k=0,count=0;

 t = (int *)malloc((iu-ip+1)*sizeof(int));

 if (!t) return ERR;
```

```
i = ip;

j = imiddle+1;

k = 0;



while (i<=imiddle && j<=iu){

  if (table[i]<table[j]){

    t[k] = table[i];

    i++;

  }

  else{

    t[k] = table[j];

    j++;

  }

  k++;

  count++;

}

if (i>imiddle){

  while (j<=iu){

    t[k] = table[j];

    j++;

    k++;

  }

}

else if (j>iu){

  while (i<=imiddle){

    t[k] = table[i];

    i++;

    k++;

  }
```

```
 }

 for(i=ip,j=0;i<=iu;i++,j++){

   table[i] = t[j];

 }

 free(t);

 return count;

}
```

4.3 Section 3

```
int quicksort(int* table, int ip, int iu){



   int pos=0, cont=0, c=0;



   if(!table||ip<0||iu<0){



     return ERR;

   }



   if(iu<ip){



     return ERR;

   }



   if(ip==iu){



     return 0;

   }



   cont+=split(table, ip, iu, &pos);
```

```
   if(ip<((pos)-1)){


       c=quicksort(table, ip, ((pos)-1));
   if(c==ERR)return ERR;


   cont+=c;


   }
   if(iu>pos+1)
       c=quicksort(table, ((pos)+1), iu);
   if(c==ERR)return ERR;
   cont+=c;
   return cont;
}


int median(int *table, int ip, int iu,int *pos){
   (*pos)=ip;
   return 0;


}


int split(int* table, int ip, int iu,int *pos){
   int i=0, k=0, aux=0, cont=0;
   cont+=median(table, ip, iu, pos);
   k=table[(*pos)];
   for(i=ip+1;i<=iu;i++){
       cont++;
       if(table[i]<k){
```

```
            (*pos)++;

            aux=table[i];

            table[i]=table[(*pos)];

            table[(*pos)]=aux;



        }

    }

    aux=table[ip];

    table[ip]=table[(*pos)];

    table[(*pos)]=aux;

    return cont;

}
```

4.5 Section 5

```
int quicksortntr(int* table, int ip, int iu){

    int pos=0, cont=0, c=0;

    if(iu<ip||!table||ip<0||iu<0){

      return ERR;

    }

 while(ip<iu){

        cont+=split(table, ip, iu, &pos);

        if(ip<((pos)-1)){

             c=quicksortntr(table, ip, pos-1);

            if(c==ERR)return ERR;

            cont+=c;

        }

        ip=(pos)+1;

    }

    return cont;
```

```
}
```

5. Results, Plots

In this section we will write the results obtained in each section with its corresponding plots.

5.1 Section 1

After implementing the MergeSort algorithm, we have proved its functionality with a test running exercise4.c. In the previous image we can see the results we've got. We can see the array of size 20 completely sorted. We have repeated this command and we have seen that the implementation is in fact correct.

5.2 Section 2

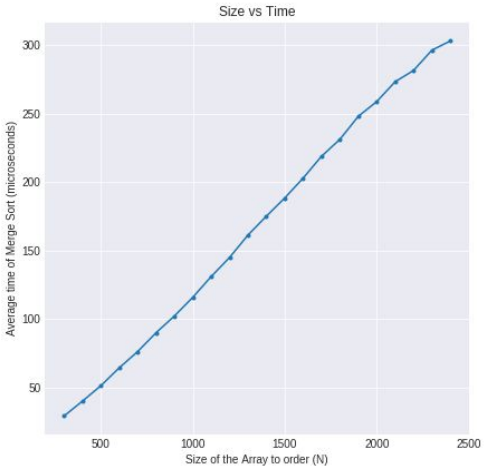
	Size	Time	average_ob	max_ob	min_ob
0	300	29.45178	2097.46964	2137	2050
1	400	40.28768	2961.51842	3007	2908
2	500	51.52784	3854.61406	3904	3791
3	600	64.59334	4792.82542	4852	4714
4	700	76.38524	5749.19300	5816	5686
5	800	90.13826	6721.00808	6787	6648
6	900	102.36384	7709.60752	7780	7636
7	1000	116.03660	8707.20068	8777	8624
8	1100	131.12182	9738.41026	9815	9655
9	1200	145.06296	10783.84648	10867	10697
10	1300	161.37472	11835.23364	11925	11731
11	1400	175.15290	12896.25772	12999	12802
12	1500	188.44630	13961.42314	14059	13867
13	1600	202.93894	15040.08562	15138	14931
14	1700	218.87856	16126.94040	16225	16028
15	1800	231.22752	17217.15882	17324	17121
16	1900	247.99654	18313.81830	18410	18211
17	2000	258.73320	19412.56884	19504	19311
18	2100	273.20394	20533.71100	20634	20404
19	2200	281.40316	21674.67150	21780	21567
20	2300	296.21408	22818.19286	22925	22686
21	2400	302.94330	23965.30062	24089	23834

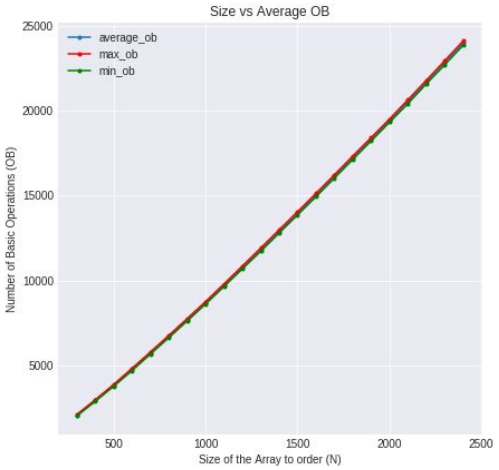
In this section, the result obtained by exercise5.c is a table that in each rank contains the number of elements in the array to sort, the avg time that for each different permutations it took to sort it, the average times that the OB was executed, the worst case in terms of OB's and the best case in terms of OB's. We have checked that the output is correct by generating 100000 perms of each number, from 300 to 2400 incrementing each time 100, and we looked that the min OB should be approximately Nlog(n) and the max OB times should be approximately Nlog(n) + O(n) , and the average OB times should be around Nlog(n). In the table below the average times are expressed in microseconds so that the times don't look like 0,0000…

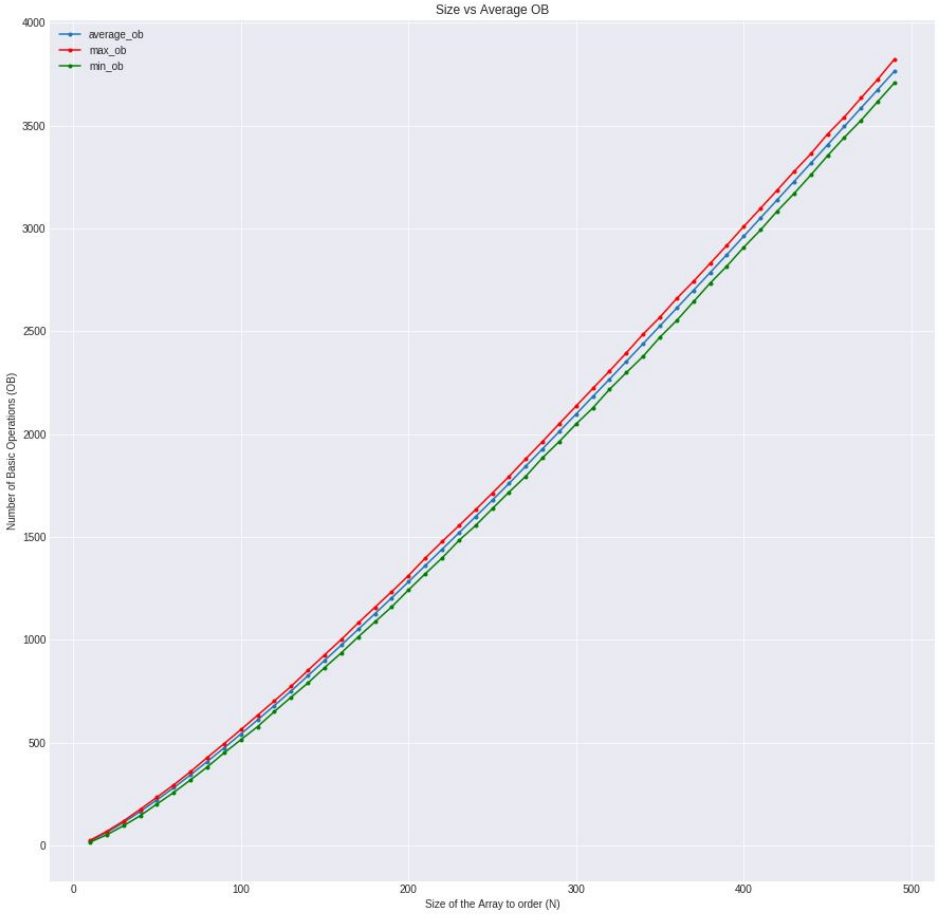
We then proceeded to plot the results of the previous table. In the plots we can see that the size vs time graph shows how the average time grows like n*log(n) with the size of the array.

When we take a look at the graph that compares how the average, max and min OB times grow with the size of the array, in the first graph we see that if we use big sizes of the arrays, but not generating too many permutations, the difference between best, worst and avg is not as big as if

we, like in the second graph, use smaller sizes but we do more permutations, where we can see that there is a more notorious difference between the three functions, and that is because having smaller array's sizes and more permutations lets us having a higher chance to reach the best and worst case or at least get closer. So taking a look at the average OB we can see that it seems correct because what should appear is that the average number of times the OB is executed nlog(n) +O(n).







5.3 Section 3

After implementing the QuickSort algorithm, we have proved its functionality with a test running exercise4.c. In the previous image we can see the results we've got. We can see the array of size 20 completely sorted. We have repeated this command and we have seen that the implementation is in fact correct.

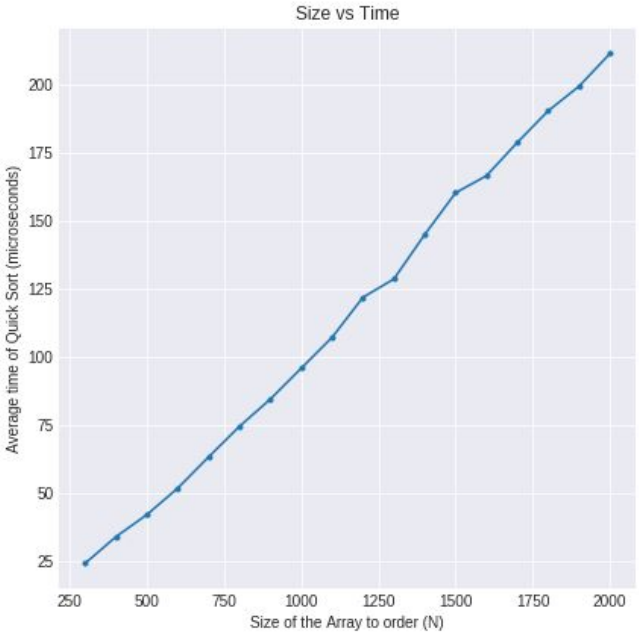
5.4 Section 4

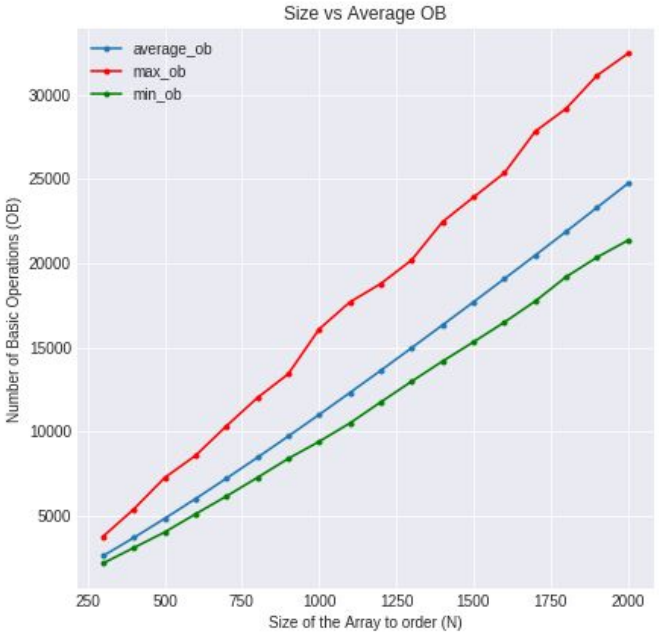
	Size	Time	average_ob	max_ob	min_ob
0	300	24.44254	2582.31448	3735	2142
1	400	34.19128	3669.61166	5366	3069
2	500	42.28034	4807.62940	7239	3998
3	600	51.98074	5986.38584	8557	5078
4	700	63.42426	7196.68242	10312	6136
5	800	74.63508	8434.02308	11989	7240
6	900	84.62994	9700.79448	13405	8372
7	1000	95.95462	10985.99214	16072	9386
8	1100	107.27650	12295.85850	17688	10485
9	1200	121.93798	13618.30028	18780	11739
10	1300	128.64422	14963.25574	20192	12969
11	1400	144.95270	16321.67800	22459	14166
12	1500	160.24562	17687.50932	23921	15310
13	1600	166.48882	19070.42360	25363	16481
14	1700	178.71606	20465.21882	27836	17737
15	1800	190.28622	21877.39508	29196	19184
16	1900	199.30200	23306.92716	31161	20348
17	2000	211.27086	24737.61834	32468	21351
18	2100	223.76060	26173.01990	38627	22607
19	2200	240.97250	27625.55286	35972	24146
20	2300	256.79002	29070.06436	39447	25282
21	2400	264.19614	30539.70852	40818	26610

In this section, the result obtained by exercise5.c is a table that in each rank contains the number of elements in the array to sort, the avg time that for each different permutations it took to sort it, the average times that the OB was executed, the worst case in terms of OB's and the best case in terms of OB's. We have checked that the output is correct by generating 100000 perms of each number, from 300 to 2400 incrementing each time 100, and we looked that the min OB should be approximately Nlog(n) and the max OB times should be approximately N^2 , and the average OB times should be around 2Nlog(n). In the table below the average times are expressed in microseconds so that the times don't look like 0,0000…

We then proceeded to plot the results of the previous table. In the plots we can see that the size vs time graph shows how the average time grows like 2n*log(n) with the size of the array.

When we take a look at the graph that compares how the average, max and min OB times grow with the size of the array, we can see that in this case we can clearly see the differences between the best, worst and average case of the quicksort algorithm. With this we can observe that quicksort has a bigger typical deviation than the MergeSort algorithm. Moreover, we can observe





	Size	Time	average_ob	max_ob	min_ob
0	300	20.9607	2581.0686	3534	2162
1	400	27.6158	3667.3514	5317	3071
2	500	34.7356	4804.5422	6528	4100
3	600	42.9578	5978.4629	8671	5092
4	700	51.0820	7188.2202	9859	6184
5	800	59.7447	8428.9584	11526	7258
6	900	67.5928	9699.2175	12929	8332
7	1000	76.4476	10991.9117	14664	9496
8	1100	85.2893	12303.8321	16204	10654
9	1200	95.3555	13612.0150	18239	11628
10	1300	104.3954	14966.8226	20099	12891
11	1400	113.4416	16316.4859	21780	14157
12	1500	123.7438	17688.8709	25280	15456
13	1600	132.9607	19074.5942	24491	16723
14	1700	147.2299	20465.6204	26849	17935
15	1800	159.1725	21865.2111	28357	19289
16	1900	168.5392	23309.4423	30615	20301
17	2000	179.2011	24732.8762	33354	21754
18	2100	190.6770	26173.6186	34107	23140
19	2200	201.2386	27618.1656	35601	24170
20	2300	213.0899	29066.2761	37029	25739
21	2400	227.0097	30541.1764	39744	26754

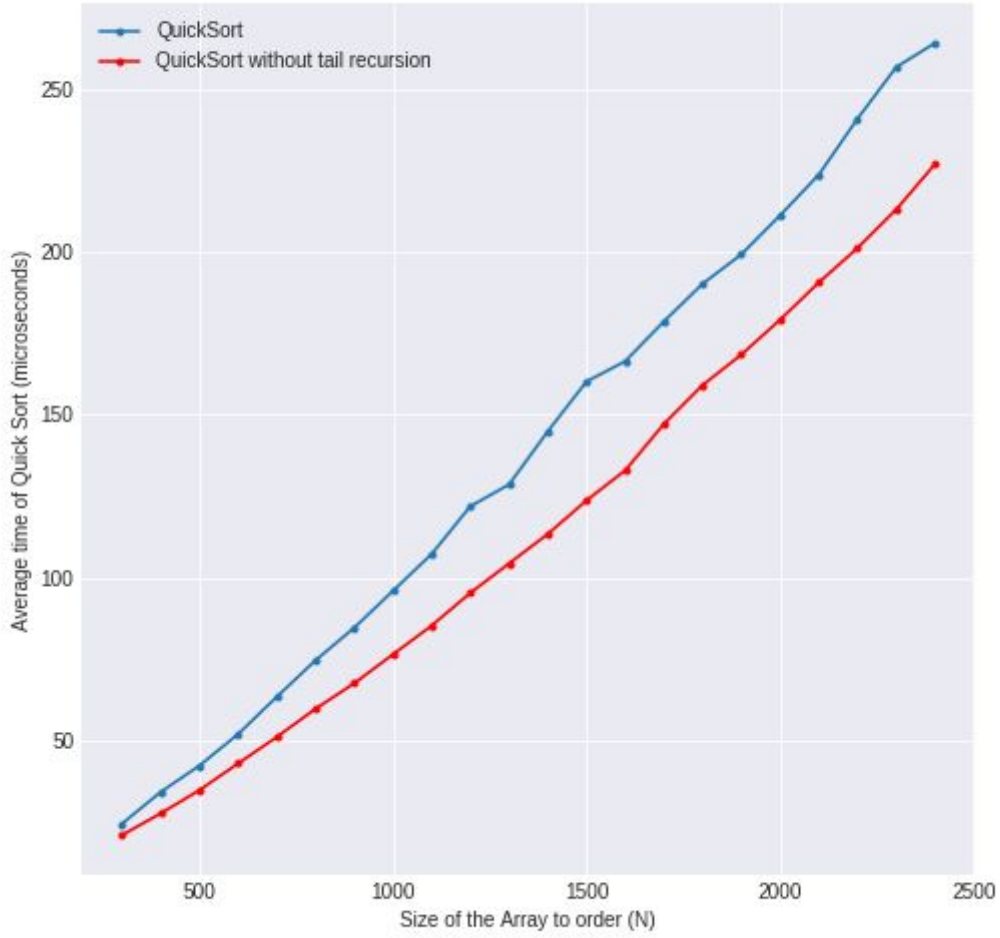
that the worst case is further apart from the average than the best case. That is what we expected theoretically, as the worst case in the QuickSort algorithm grows as N^2 whereas the best case grows as Nlog(N) (We can see in fact that when N gets big the red line is more and more away from the blue line).

5.5 Section 5

After implementing this new version of the quicksort algorithm, we proceeded to calculate the same results as with MergeSort and QuickSort implementations. In this section, the result obtained by exercise5.c is a table that in each rank contains the number of elements in the array to sort, the avg time that for each different permutations it took to sort it, the average times that the OB was executed, the worst case in terms of OB's and the best case in terms of OB's. We have checked that the output is correct by generating 100000 perms of each number, from 300 to 2400 incrementing each time 100, and we looked that the min OB should be approximately Nlog(n) and the max OB times should be approximately N^2 , and the average OB times

should be around 2Nlog(n). In the table below the average times are expressed in microseconds so that the times don't look like 0,0000…

After having the results, we then plot them with the Quicksort implementation that has tail recursion. We didn't plot the basic operation part as they are the same algorithm, they take the same number of basic operations to run. In the size vs time graph, we can observe that in all points the implementation without tail recursion has a better performance than the normal implementation. We will discuss at the questions section why we believe this happens.



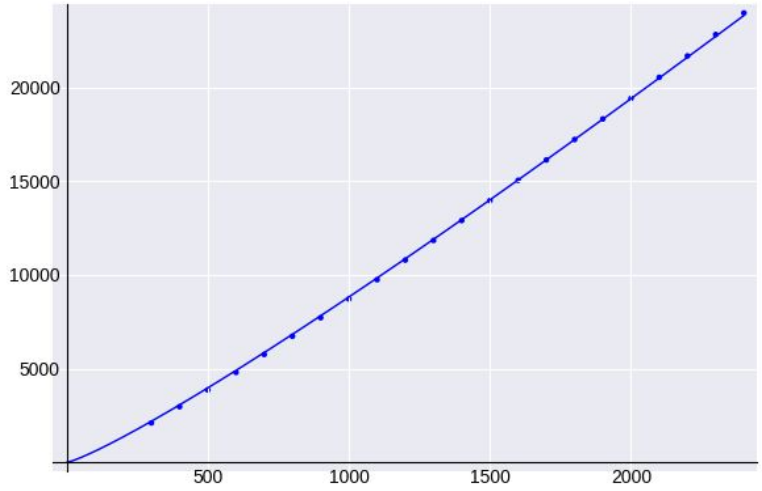
6. Answers to theoretical Questions

Here you answer the theoretical questions in the practice.

5.1 Question 1

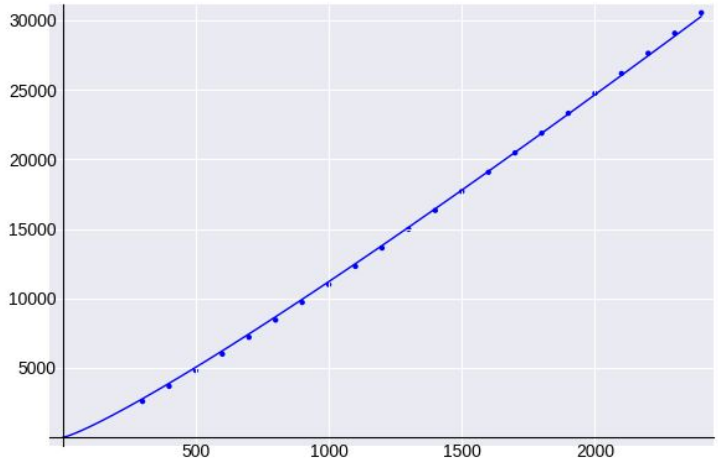
We will analyze both cases separately doing a regression where, comparing each function to their theoretical average case, that we approximate that it is a*n*log(n), and calculating the value of the constant a in order to know how accurate where our measures are. Taking a look at MergeSort, the graph below, the points are our measures of the number of basic operations in the average case and we plotted them with the regression of the form a*n*logn calculating

the value of a, and we got that a= 0.8844017550149819, which is an accurate approximation to n*log(n).



In the Y-axis it is shown the number of basic operations and in the X-axis it is shown the size of the array.

We followed the same process for QuickSort obtaining another similar graph which is inserted below and obtaining a value of a=1.1237812656734798.



In the Y-axis it is shown the number of basic operations and in the X-axis it is shown the size of the array.

In our results, the traces of the performance graphs are not sharp, but they could be sharp and this would be because the number of perms that we generate can't reach all the possible permutations of the arrays of that size, so the average moves away from the value that should

have. If this happened, we could increment the number of perms in order to make the average get closer.

5.2 Question 2

quicksort and quicksort_ntr are different implementations of the same algorithm. As we can see in image 6, the time needed to perform the quicksort algorithm is higher than the time needed to perform the algorithm without tail recursion. Here we can see how recursion is something we should avoid as the computer has a lower performance in recursion algorithms. Regarding the number of basic operations in both implementations we can see that they are the same. We have confirmed that given the same input both implementations use the same number of basic operations (which is logical as they are the same algorithm). We don't include a graph comparing the number of basic operations as the difference is so tiny that we can't visualize anything. So we can conclude that quicksort_ntr is a better implementation of the quicksort algorithm than the quicksort function as with the same number of basic operations it takes less time to finish.

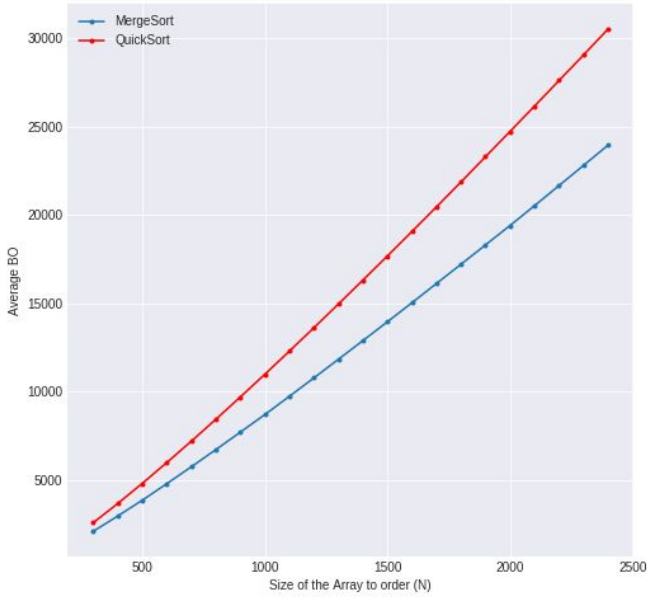
5.3 Question 3

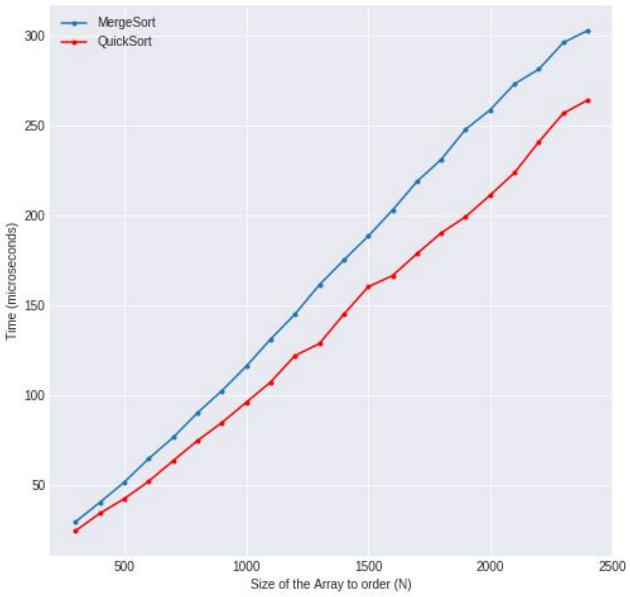
As it's impossible to calculate the empirical average case for every N, we are going to look at some of the results that we have in order to compare the avg number of basic operations with the theoretical ones. Also, in the second part of the question it is answered to why is it difficult to show the strict best and worst cases.

For example, with N=300, the best case of MergeSort is 2050, and worst case is 2137, that is approximately the same as the theoretical best and worst cases of this algorithm, theta(nlogn). With N=1000, the best case is 8624 and the worst case is 8777, which again is close to the theoretical results, theta(nlogn). On the other side, taking a look at QuickSort, with N=300, the best case is 2142 and the worst case is 3735, which are similar to the theoretical ones(theta(nlogn) and n^2/2 + O(n) respectively). The same happens with N=1000, where the best case is 9386, and the worst case 10072. In the next paragraph we will analyze how to make these results look closer.

In order to calculate exactly the best and worst cases for each algorithm, for a given size of the array, we should calculate all the possible permutations of that array, so we could not use the function that generates random perms, and we should only count ob's and take times once for each perm, and that would exactly count how many basic operations does the algorithm do for each perm(this is if we also wanted to get the average case strictly). Then we would take the OB times that correspond to the sorted and unsorted arrays for both algorithms, and we would be able to know it strictly.

5.4 Question 4





Here we can see two graphs that compare the average time and average number of basic operations both MergeSort and QuickSort take. In red we can see the performance of QuickSort and in blue the performance of MergeSort. We can observe that the QuickSort algorithm has a higher average number of basic operations than the MergeSort. Theoretically, this is the expected result, as in a general case (for large arrays) MergeSort had a lower average case than QuickSort. However, we can clearly see that QuickSort takes less time to run than MergeSort. Our main hypothesis of this behavior is that MergeSort uses less average operations, but its usage of dynamic memory increases notably the running time. Regarding the aspect of memory management, QuickSort is better than MergeSort as MergeSort has to duplicate the array in order to run. Furthermore, MergeSort uses dynamic memory, which lowers the time performance of the implementation.

7. Final Conclusions.

In this practice we had studied two recursive sorting algorithms: MergeSort and QuickSort. These algorithms used a Divide and combine method: they divide the array needed to sort, and they combine it in a sorted way. To implement these algorithms we needed auxiliary functions: the functions that divided the arrays and the functions that combined them. Thanks to the tools developed in the previous practice, we were able to measure the performance of these algorithms.

After analysing the algorithms we can conclude that they are both much better than the local sorting algorithms as they grow approximately as Nlog(n) whereas the local sorting algorithms grew approximately as N^2.

Regarding the Merge Sort algorithm, we can conclude that it has a general good performance, with few typical deviations between its best and worst cases. So, Merge Sort is a robust and fiable algorithm, with the counterpart that it has to use a higher amount of memory (it has to double the memory of the array in order to complete). Also, with our implementation

MergeSort has made a worse performance than QuickSort mainly because the usage of dynamic memory makes the implementation run slowly. Regarding the future, it will be interesting to try to create an implementation that uses static memory, and we predict that it will be quicker than the quicksort algorithm.

Regarding the Quick Sort algorithm, we can conclude that it has a general good performance. However, it proved to have a much higher typical deviation than MergeSort. In general, quick sort is a good algorithm as it doesn't need more memory to run and is fast ($2N\log(N) + O(N)$) in the average case, but we can have cases in which the running time goes up to N^2. So, we can say that quicksort is a good algorithm, but not for all cases. From this algorithm we have analysed two main implementations: the classic one, and another one in which we eliminated the tail recursion. After comparing both of them, we can conclude that the implementation without tail recursion is faster and therefore better than the classic one. This is another proof of why recursion enlarges the running time of algorithms.