Algorithm Analysis 2019/2020

Practice 2

Miguel Arnaiz Montes and Pablo Almarza Marques

Group 1292

Couple 03

Code	Plots	Documentation	Total

1. Introduction

In this practice we will use some of the functions created on the previous practice and we will create MergeSort and QuickSort, two new sorting algorithms.

2. Objectives

2.1 Exercise 1

Implement the code for a working merge sort algorithm. This algorithm will be implemented recursively. Also changing the file exercise4 to check the correctness of the algorithm

2.2 Exercise 2

The goal for this exercise is dealing with the problems this new algorithm may cause when implementing it in the exercise5 file the show the average time, the minimum, maximum and average basic operations in relation to the size of the array to order.

2.3 Exercise 3

In this exercise we create the QuickSort algorithm making it work with two additional functions: split, to move the pivot into the correct position, and median, to select the pivot (always the first element of the table in this case).

2.4 Exercise 4

In this exercise we will use the functions created in times.c in the last practice to obtain the maximum, minimum and average number of basic operations, and the average clock time depending on the size of the permutation.

2.5 Exercise 5

In this exercise we will create a new quicksort function deleting the tail recursion.

3. Tools and methodology

We have used Visual Studio on Linux. Using C/C++ and LiveShare extensions for VS and Valgrind when executing the programs to check errors and memory leaks.

3.1 Exercise 1

For exercise 1 we had to implement the merge sort recursively following the pseudocode in moodle. This brought problems like the counting of the basic operations. Regarding the workspace, only the terminal was necessary to obtain the results.

3.2 Exercise 2

In exercise 2 we had to learn how to use GNUplot to make the plots of the time and the basic operations. With the time plots, we only had to use gnuplots commands explained in moodle, but for the basic operations we had to create a file to plot the 3 basic operation cases and fitting this cases to see the theoretical line that the practical cases should follow.

3.3 Exercise 3

For this exercise we got help from the the pseudocode posted on moodle of the quicksort. However we had to modify it in order to implement the position of the pivot as a pointer moving across functions. Also we had to move this pivot to the first position every time we entered split in order for it to work.

3.4 Exercise 4

The only thing we had to do was changing the function used in exercise5.c to quicksort. Then we used GNUplot to create the scatter plot graph as we learned in exercise 2.

3.5 Exercise 5

Here we created the new function quicksort_ntr to eliminate the tail recursion. We used a while loop with first<last condition. So that when first = last the function would not enter the loop and return 1 (as we initialize count to 1) as base case. That way we delete the tail recursion.

4. Code

4.1 Exercise 1

```c
int mergesort(int *table, int ip, int iu)
{
 int m = ip + (iu - ip) / 2, ret = 0, obs = 0;

   if (!table || ip >= iu)
   {
     return ERR;
   }
 ret = mergesort(table, ip, m);
 obs = ret > 0 ? ret : 0;
 ret = mergesort(table, m + 1, iu);
 obs += ret > 0 ? ret : 0;
 return obs + merge(table, ip, iu, m);
}
```

```
int merge(int *table, int ip, int iu, int imiddle)
{
 int i, j, k, count = 0, lowValue = imiddle - ip + 1, upValue = iu
- imiddle, *lowerTab, *upperTab;
  lowerTab = (int*) malloc(lowValue * sizeof(int));
  if(!lowerTab) return ERR;
  upperTab = (int*) malloc(upValue * sizeof(int));
  if(!upperTab)
 {
   free(lowerTab);
   return ERR;
 }
  for (i = 0; i < lowValue; i++)
   lowerTab[i] = table[ip + i];
  for (j = 0; j < upValue; j++)
   upperTab[j] = table[imiddle + 1 + j];

 i = j = 0;
 k = ip;
 /*Perform the key comparison, raise the count by one, and merge
the table */
 while (i < lowValue && j < upValue)
 {
   count++;
   if (lowerTab[i] <= upperTab[j])
   {
     table[k] = lowerTab[i];
     i++;
   }
   else
   {
     table[k] = upperTab[j];
     j++;
   }
   k++;
 }

 /* Copy the remaining elements of lowerTab[], if there are any */
 while (i < lowValue)
 {
   table[k] = lowerTab[i];
   i++;
   k++;
 }
 free(lowerTab);
 /* Copy the remaining elements of upperTab[], if there are any */
 while (j < upValue)
 {
   table[k] = upperTab[j];
   j++;
   k++;
 }
 free(upperTab);
 return count;
```

4.3 Exercise 3

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

 int pos;
 int count = 0;
 int auxCount;


 if(!table || ip > iu) {
   return ERR;
 }

 if (ip == iu) {
   return OK;
 }
 else {
   auxCount = split(table, ip, iu, &pos);
   if(auxCount == ERR) return ERR;
   count = auxCount;

   auxCount = quicksort(table, pos+1, iu);
   count += auxCount > 0 ? auxCount : 0;
   auxCount = quicksort(table, ip, pos-1);
   count += auxCount > 0 ? auxCount : 0;
 }

 return count;
}
```

```c
int split(int* table, int ip, int iu, int* pos) {
  int i, k, count;

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

 count = median(table, ip, iu, pos);
  if((*pos) > iu || (*pos) < ip){
   return count;
 }

 k = table[*pos];
 swap(&table[ip], &table[*pos]);
 *pos = ip;

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

 swap(&table[ip], &table[*pos]);

 return count;
}
```

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

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

 *pos = ip;

 return 0;
}
```

4.5 Exercise 5

```
int quicksort_ntr (int* table, int ip, int iu){
 int pos = 1, count = 1;

 if (table == NULL || ip < 0 || iu < ip) {
   return ERR;
 }

 while (ip < iu) {

   count += split(table, ip, iu, &pos);
   if (count == ERR)return ERR;

   if(pos - ip < iu - pos) {
     quicksort_ntr(table, ip, pos - 1);
     ip = pos + 1;
   } else {
     quicksort_ntr(table, ip, iu);
     iu = pos - 1;
   }
 }

 return count;

}
```

5. Results, plots

5.1 Exercise 1

./exercise4 -size 20

```
0    1    2    3    4    5    6    7    8    9    10    11    12    13    14    15    16    17
18    19
```

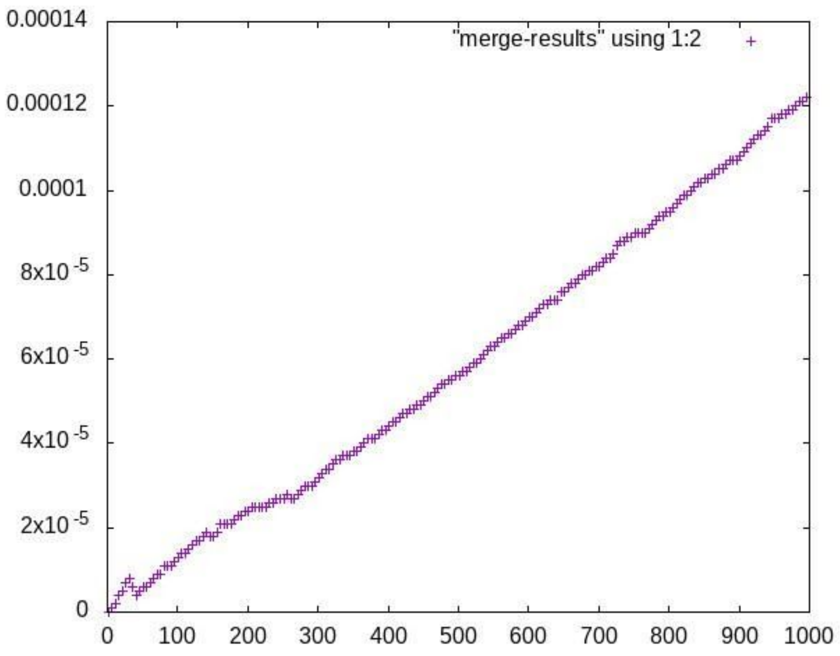
Number of BOs: 64

./exercise4 -size 1000

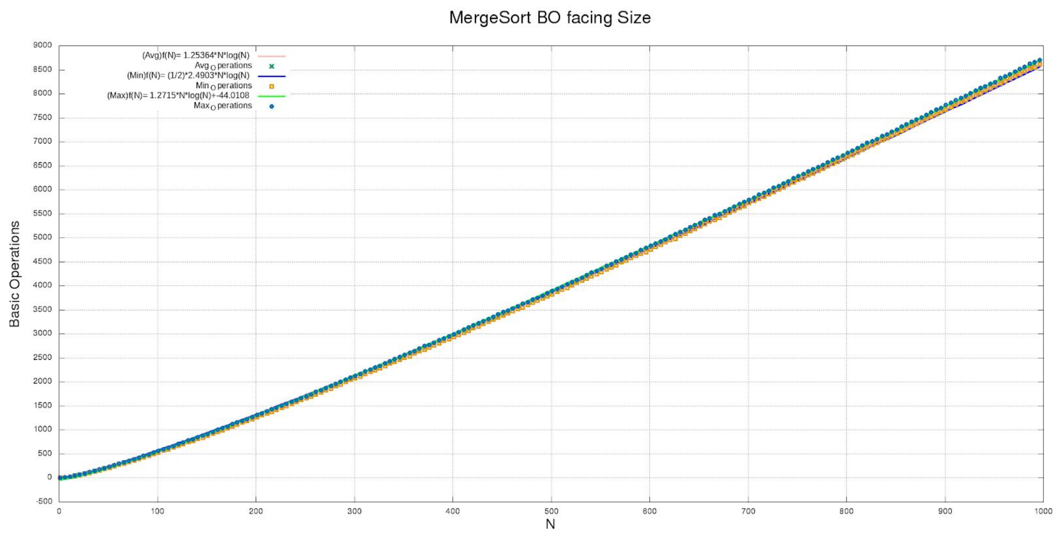
Number of BOs: 8715

The result is the one expected since the array is in order.

5.2 Exercise 2



Here we can see the linear increase of the time (vertical axis) it takes for an, also increasing in size (horizontal axis), array to be arranged in order.



And here we can observe why the merge sort is a very efficient sorting algorithm, because the worst, best and average cases are really close regarding basic operations between them.

5.3 Exercise 3

./exercise4 -size 20

```
0    1    2    3    4    5    6    7    8    9    10    11    12    13    14    15    16    17
18    19
```

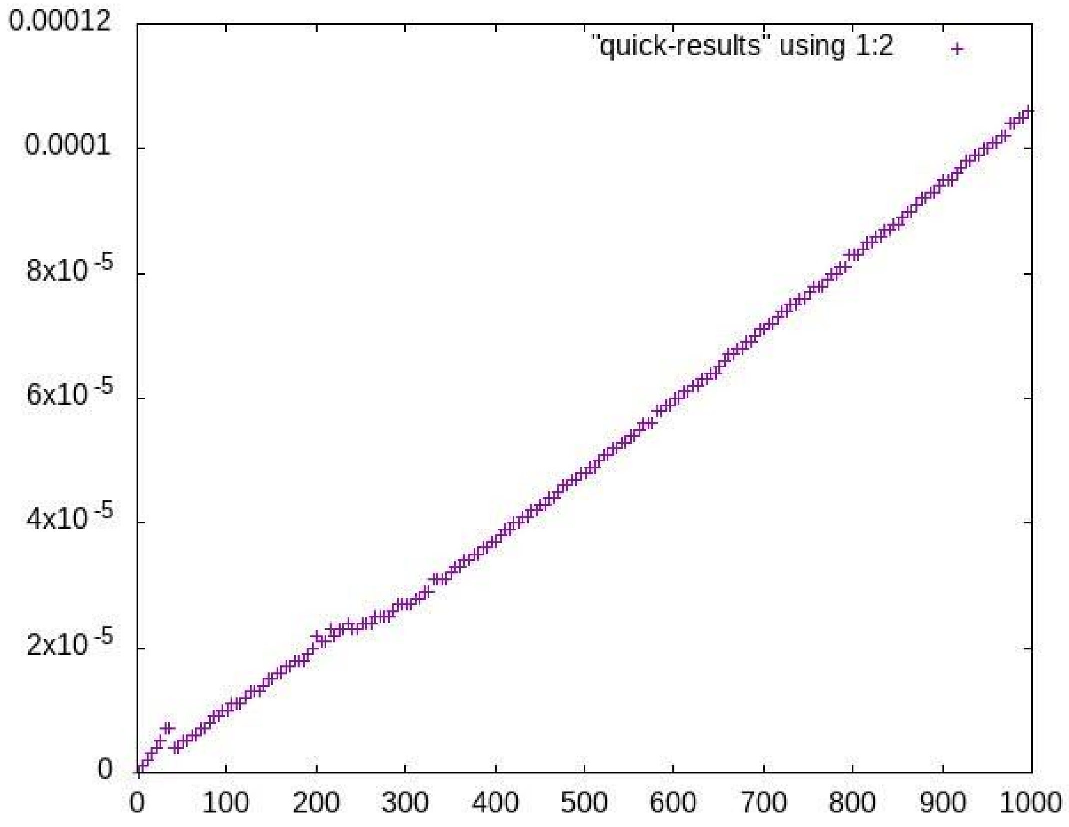
Number of BOs: 85

./exercise4 -size 1000

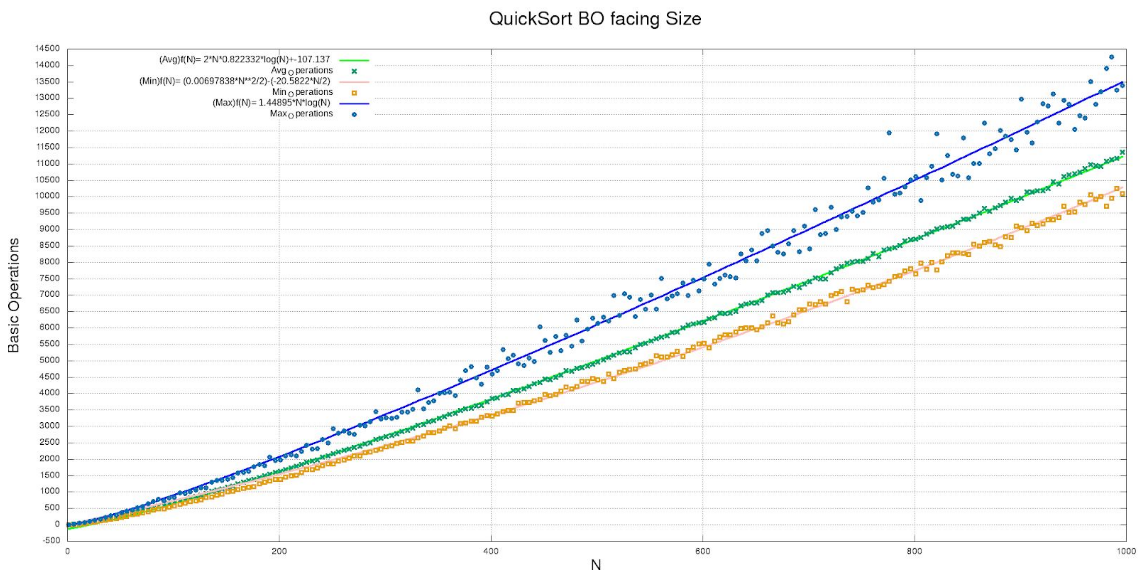
Number of BOs: 11437

Now we can see the correct working of the quick sort. We can also observe a difference between basic operation at big sizes between the merge and quick sorts, being merge better for bigger sizes.

5.4 Exercise 4

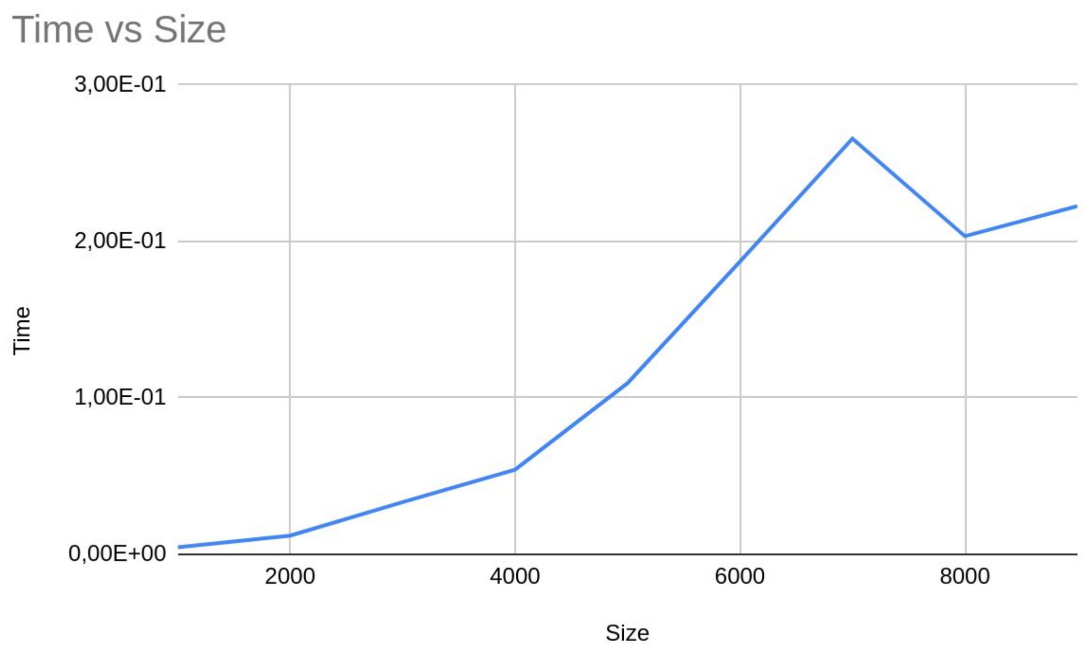


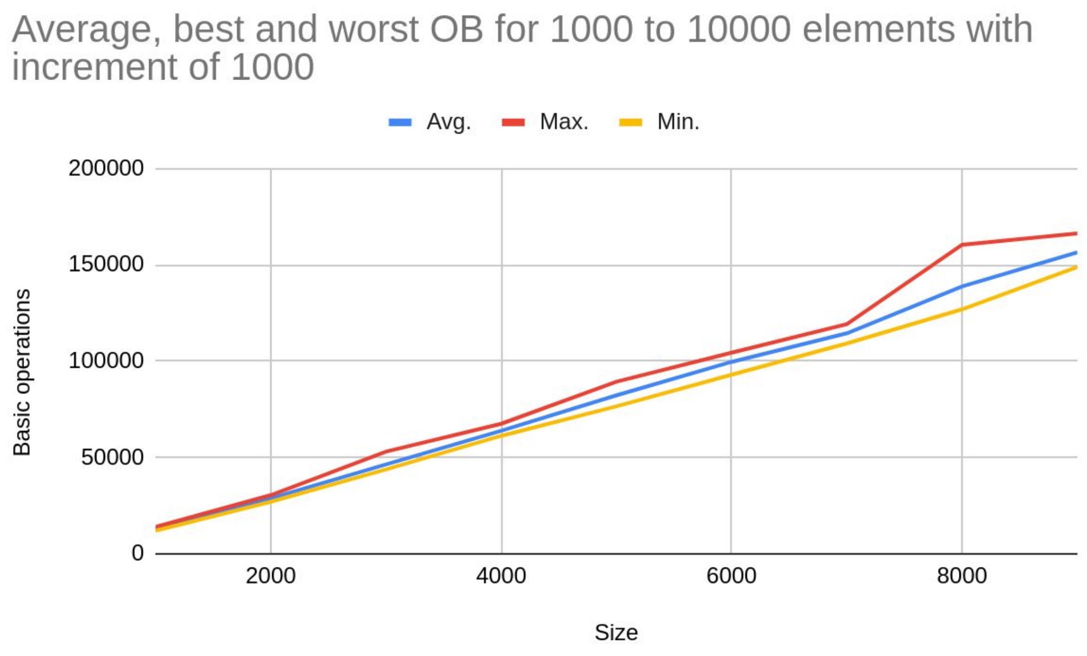
And again we see a linear increase in the time for the quick sort algorithm when increasing the size of the array.



Contrary to the merge sort, the quick sort average, best and worst cases basic operations aren't close between them, This lets us see clearly the fitted lines for each case and that the practical case follows this theoretical lines.

5.5 Exercise 5





By eliminating tail recursion, we limit the space used to O(log n).

6. Response to the theoretical questions.

6.1 Question 1

Both algorithms follow the theoretical lines. But the merge sort seem to follow the theoretical line much sharply. This is because the merge sort is a very efficient sorting algorithm and the random factor won't affect it as much as to other sorting methods.

6.2 Question 2

Just by observation of the plots and results we can confirm that the 3 versions of the quick sort used in this practice are quite similar. We see minor differences but they could be caused by the random factor involved inevitably.

6.3 Question 3

The best case for the merge would be when the smaller numbers are grouped in the left half so the bigger mergings don't take too long to merge. However the merge sort doesn't seem affected as much as other sorting algorithms by the numbers not being ordered. For the QuickSort algorithm the best case scenario is that the pivot is the middle value of the table. To calculate each case we could modify median to search for the intermediate value pivot for the best case, the lower or highest value pivot for the worst case, and the average between the intermediate and lower or highest value pivot for the average case.

6.4 Question 4

Merge sort is a more consistent and more efficient for larger arrays than the quick sort, because of the worst case theoretical formulas. However the quick sort is an internal sorting algorithm, so it requires very little memory management, contrary to the merge which allocates auxiliary arrays.

7. Final conclusions

We have several conclusions in this practice. First of all, the way of sorting, while MergeSort is more intuitive since it divides the array in 2 halves always, in QuickSort there are many possibilities which makes it more difficult to follow.

Another thing is the worst case complexity, while MergeSort have the same complexity for worst case and average case (O(Nlog(N)), QuickSort worst case is equal to O(N^2), that means that if we have the worst case in this second algorithm, it will need a lot more comparisons than QuickSort Furthermore, QuickSort is not good with large arrays but MergeSort is.

A positive thing for QuickSort would be the space requirement because it doesn't require any additional storage, but MergeSort does, in order to store the auxiliary arrays. Not only that, because MergeSort needs even more auxiliary memory for sorting, since it is a external sorting method while QuickSort doesn't require that,

because it is an internal sorting one.
Finally, we can conclude that Quicksort is faster and more efficient than MergeSort if the array is small. If the array is larger, then MergeSort will be the more efficient one.