Analysis of Algorithms 2019/2020

Practice 2

José Manuel Freire and Rodrigo Juez, 1292.

Code	Plots	Memory	Total

1. Introduction.

In this practice we will implement divide and conquer algorithms, MergeSort and QuickSort, for this we will create two main sorting.c functions and their corresponding supporting functions (merge in MS and split and medium in QS). Our objective is to see the advantages of this kinds of algorithms and try to implement the QS algorithm in several ways (with different pivots). The output expected is an ordered array by the both algorithms in exercise4.c and the running times and BOs of both algorithms (with different implementations of the QS medium).

2. Objectives

2.1: MergeSort Algorithm: The objective is to implement it into the previous exercises and calculate its running time.

2.2: QuickSort Algorithm: The objective is to implement it with several pivots into the previous exercises, calculate its different running times and compare them.

3. Tools and methodology

3.1: MergeSort Algorithm: Understanding the algorithm, writing the code and using the debugging flag in conjunction with VSCode to see how it evolved for solving bugs. After finishing it we run extensive tests with big cases to test its limit.

3.2: QuickSort Algorithm: Understanding the algorithm, writing the code and using the debugging flag in conjunction with VSCode to see how it evolved for solving bugs. We made several split functions and run them to see the difference they made in time and how far was the limit of this algorithm.

4. Source code

4.0 Swap

```
void swap(int *a, int *b){
  int aux;
  aux=*b;
  *b=*a;
  *a=aux;
}
```

4.1 MergeSort

```
int MergeSort(int* table, int ip, int iu){
  int imiddle, OBs=0;
  if(ip>iu||!table) return ERR;
  if(ip==iu) return OK;
  else{
    imiddle=(ip+iu)/2; /*we don't use the floor function because as we're
                  dividing ints it automatically rounds down the int*/
    OBs+=MergeSort(table, ip, imiddle);
    OBs+=MergeSort(table,imiddle+1,iu);
    return merge(table, ip ,iu ,imiddle) + OBs;
  }
}

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

4.2 QuickSort

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

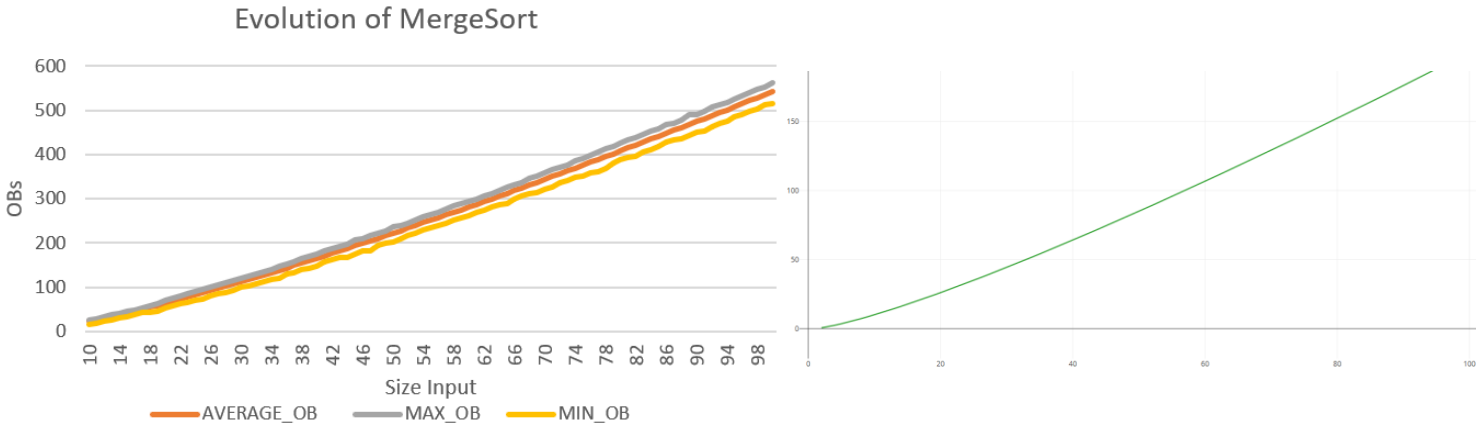
```
  int pos, BOs=0;
  if(!table || ip>iu){return ERR;}
  BOs += split_stat(table, ip, iu, &pos);
  BOs += quicksort(table, ip, pos-1);
  BOs += quicksort(table, pos+1, iu);
  return BOs;
}

int split(int* table, int ip, int iu, int *pos){
  int i, k=0, BOs=0;
  if(!table || ip<0 || pos==NULL){return ERR;}
  BOs+=median(table, ip, iu, pos);
  k=table[*pos];
  swap(&table[ip], &table[(*pos)]);
  (*pos)=ip;
  for(i=ip; i<=iu ; i++){
    if(BOs++, table[i]<k){
      (*pos)++;
      swap(&table[(*pos)], &table[i]);
    }
  }
  swap(&table[ip], &table[*pos]);
  return BOs;
}

int split_avg(int* table, int ip, int iu, int *pos){
  int i, k=0, BOs=0;
  if(!table || ip<0 || pos==NULL){return ERR;}
  BOs+=median_avg(table, ip, iu, pos);
  k=table[*pos];
  swap(&table[ip], &table[(*pos)]);
  (*pos)=ip;
  for(i=ip; i<=iu ; i++){
    if(BOs++, table[i]<k){
      (*pos)++;
      swap(&table[(*pos)], &table[i]);
    }
  }
  swap(&table[ip], &table[*pos]);
  return BOs;
}

int split_stat(int* table, int ip, int iu, int *pos){
  int i, k=0, BOs=0;
  if(!table || ip<0 || pos==NULL){return ERR;}
  BOs+=median_stat(table, ip, iu, pos);
  k=table[*pos];
  swap(&table[ip], &table[(*pos)]);
```

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

int median(int *table, int ip, int iu,int *pos){
  if(ip>iu || !table || !pos) return ERR;
  *pos = ip;
  return 0;
}

int median_avg(int *table, int ip, int iu,int *pos){
  if(ip>iu || !table || !pos) return ERR;
  *pos = (ip+iu)/2;
  return 0;
}

int median_stat(int *table, int ip, int iu,int *pos){
  int med, BOs=0; /*CAMBIAR EL median_stat y tiene que devolver 3 BOs*/
  int max=0, min=0;
  if(ip>iu || !table || !pos) return ERR;
  med = (ip+iu)/2;

  min_max(table[ip], table[iu], table[med], &max, &min);
  if(BOs++, table[ip] != max && table[ip] != min) *pos = ip; /*one BO*/
  else if(BOs++, table[iu]!=max && table[iu]!=min) *pos = ip; /*two BO*/
  else {BOs++; (*pos) = med;} /*although we don't do a comparison here
                               the equivalent for min_max should be 3
                               so i add a BO*/
  return BOs;
}
void min_max (int ip, int iu, int med, int* max, int* min){
  if (!max || !min) return;

  if(ip > iu && ip > med){ *max = ip;}
  else if (iu > ip && iu > med) {*max = iu;}
  else {*max = med;}

  if(ip < iu && ip < med){ *min = ip;}
  else if (iu < ip && iu < med) {*min = iu;}
  else {*min = med;}
}
```

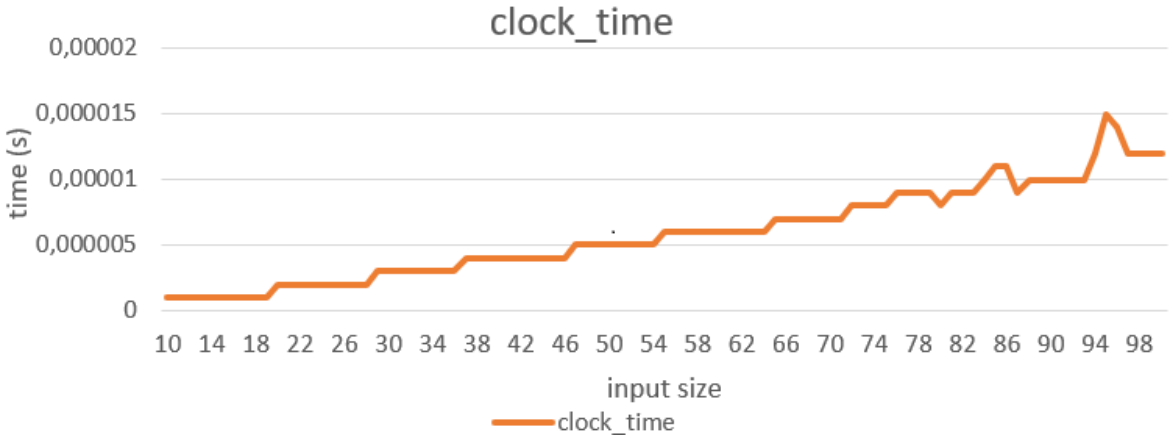
5. Results

The number of permutations for each running test was 10K and the size range went from 10 to 100:

5.1 MergeSort



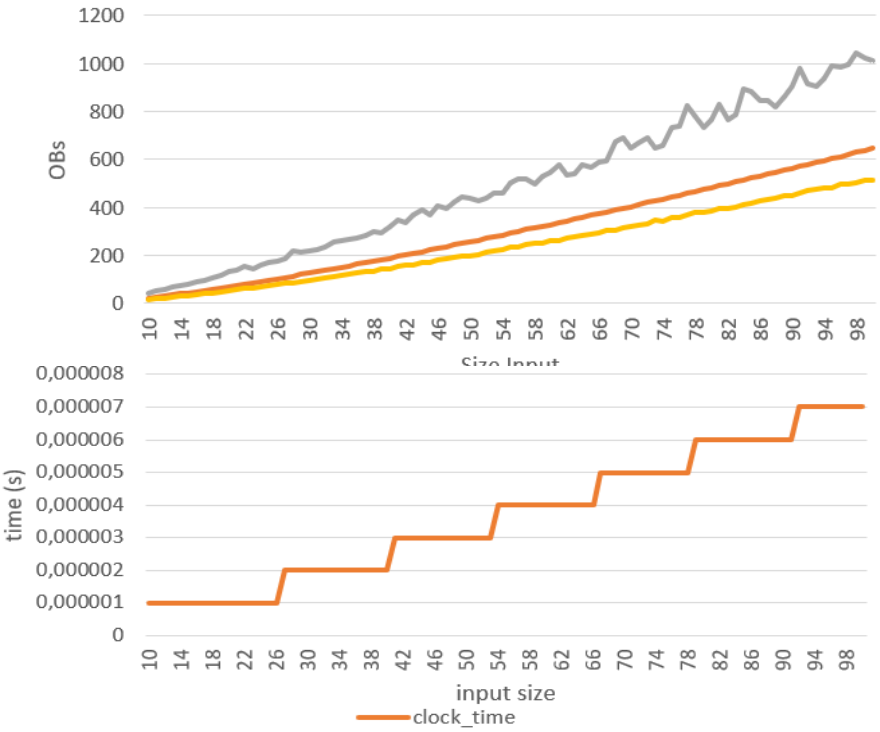
If we compare the results (left graph) with the theoretical average case of MergeSort (NlogN right graph) we can clearly see how it adjusts with its theoretical value.



When we plot the clock_time we have a trend that resembles NlogN but as the tests were run in an uncontrolled environment the average running time varies from run to run depending on the current CPU load.

5.2 QuickSort

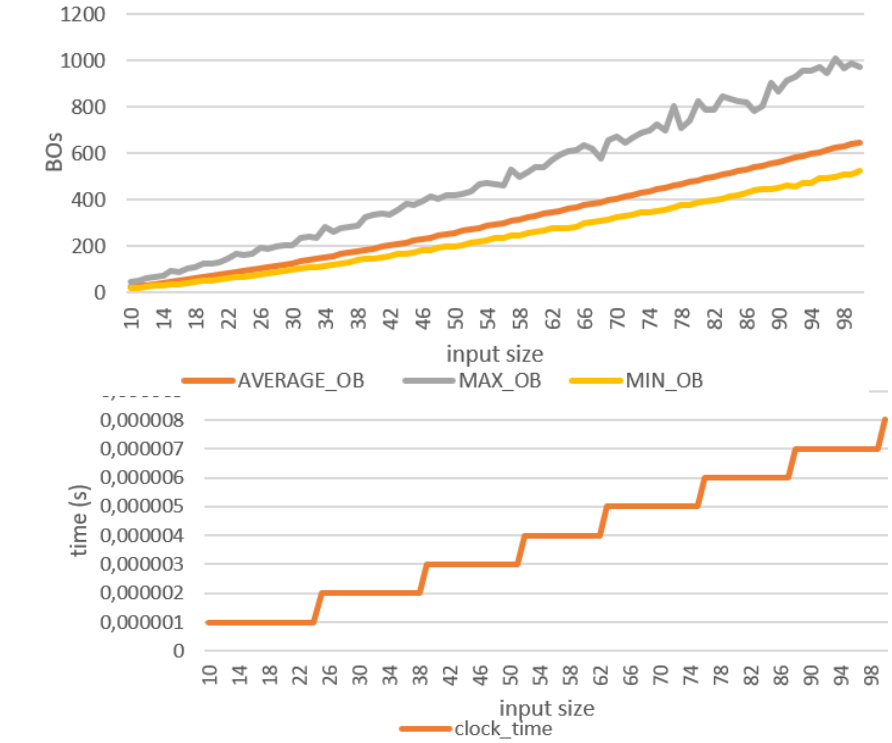
5.2.1 QuickSort (median):



The theoretical value of QS is 2N*log(N)+O(N) and as we can see in the graph the average case evolves as expected. Nevertheless, if we observe the max_obs we can see several spikes across when sorting bigger arrays.

The clock also evolves as expected but as always it's not a reliable data as the CPU run time wasn't measured under a controlled environment.

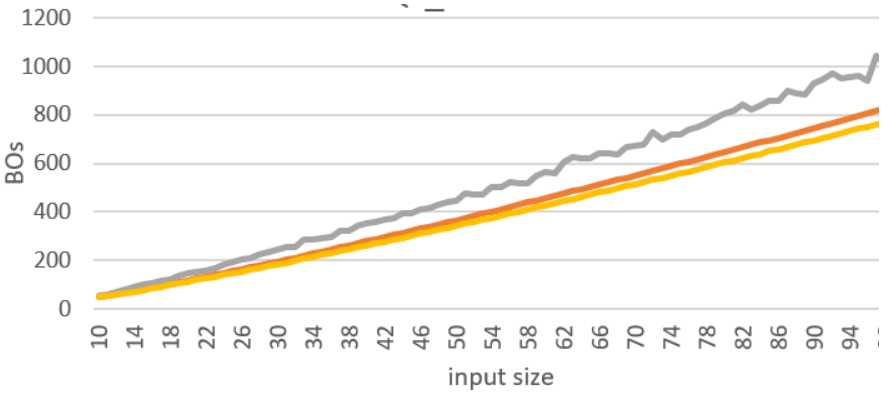
5.2.1 QuickSort (median_avg):



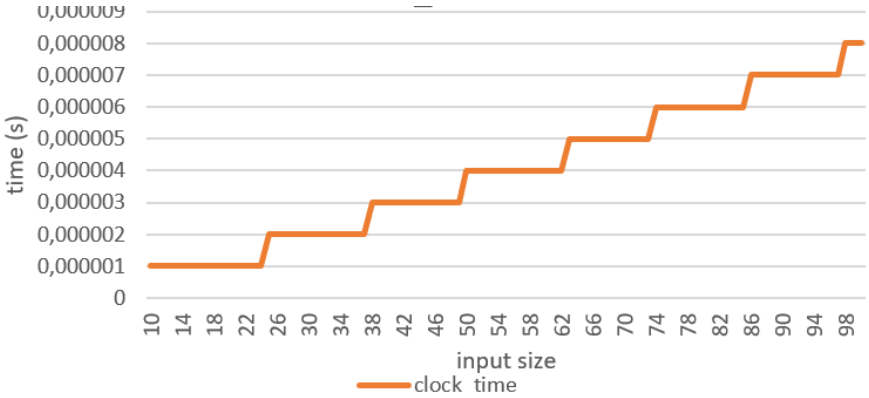
The behaviour is the same as with the normal median although there are less spikes in the max_ob its still quite irregular.

The clock times behave the same as with the normal OB.

5.2.1 QuickSort (median_stat):



It's in the median_stat where we see the difference, while the previous medians did not pass 650 OBs in 100 (because they had 0 Bos), the median_stat returns up to 3 BOs which means that it increments the total QS BOs to 800 more or less.



The advantage of median_stat is that max_obs is more predictable as it has less spikes.

6. Questions.

6.1 Compare the empirical performance of the algorithms with the theoretical average case for each case. If the traces of the performance graphs are very sharp, why do you think this happens?

The empirical performance of **MergeSort** adjusts to a great degree with the theoretical average case (NlogN), we can see how the max_obs also correspond with the theoretical values as the worst case would also be NlogN + O(N) (that's why is over the average) and the min_obs are close to the theoretical value (1/2NLogN). The best, worst and average cases in this algorithm are very similar that's why the three evolve very close to each other in the graph.

In all three cases of **QuickSort** the average empirical performance adjusts to its theoretical value of 2NlogN + O(N). As a matter of fact, if we compare the graphs of QS and MS we can see how in 100 of input size QS has more average cases than MergeSort (as it is NlogN). Also, in QS we can clearly see in the median and median_avg spikes in the max_obs, this is due to the way we choose the pivot as in median_stat we will not have this kind of behaviour. In the next exercise we compare and analyse why the different medians behave in a different way.

6.2 Analyse the result obtained with the different versions of quicksort (versions with different pivot election).

The empirical performance of **QuickSort** varies greatly depending on the pivot implemented. As the routine split orders the table by separating the elements that are less or more than the pivot if we choose a pivot without thinking, like in median and median_avg, we will have the problem that in most cases split we will have to do more ordering.
When we implement median_stat we are choosing a pivot that is more or less well positioned so that the sorting in split won't last so long. Although we will have more average BOs (because it needs to do some comparisons for obtaining the pivot) the advantage is that we have a much more regular and spike-free max_obs as the number of BOs in split will be more consistent from run to run.

6.3 What are the best and worst cases for each algorithm? What should be modified in practice to strictly calculate each case (include average case too)?

MergeSort: $(1/2)N*Log(N) \leq B_{MS}(N) \leq A_{MS}(N) \leq W_{MS}(N) \leq NLog(N)+O(N)$
QuickSort:
Worst Case: $N^2/2 - N/2$
Average Case: $2N*Log(N)$
Best Case: $N*Log(N)$

If we want to obtain MergeSort worst case we need an input already ordered so that when it's merged we do the maximum number of comparisons. The best would be obtained by previously sorting it in two halves. So that when doing the comparisons, it inserts one half and then as we run out of thing to compare it to we insert the other half without doing BOs.

To obtain the best case of QS: for each element, pivot position happens to be in the middle, resulting in a balanced binary tree of height log(n) and hence the total complexity becomes O(NlogN), while the worst case would be obtained by having the pivot in the last position of the array hence having to do every comparison.

6.4 Which of the two algorithms studied is empirically more efficient? Compare this result with the theoretical prediction. Which algorithm(s) is/are more efficient from the point of view of memory management? Justify your answer.

While MergeSort does less BOs on average and has a better best and worst case (as we can see in the graph), QuickSort manages a better memory management. This is because MergeSort is not "onsite" which means that performs the algorithm in the same array. MergeSort needs much more memory to work. Still, MergeSort has its advantages, for example, it's widely used in systems were only sequential read of the memory is allowed.