Analysis of Algorithms 2020/2021

Practice 2

Pablo Cuesta Sierra and Álvaro Zamanillo Sáez. Group 1251.

Code	Plots	Memory	Total

1. Introduction.

In this practice, we implement two Divide and Conquer algorithms: MergeSort and QuickSort, in order to collect data from their performances. As well as an alternative version of QuickSort, in which we remove the tail recursion to empirically compare the effectiveness of making far less function calls.

2. Objectives

Here you indicate the work you are going to do in each section.

2.1 Section 1

In this section, we implement in C the MergeSort algorithm.

2.2 Section 2

Here we measure the efficiency for the MergeSort algorithm with the functions implemented in the previous practice.

2.3 Section 3

In this section, we write the code for the QuickSort algorithm.

2.4 Section 4

Here we will use the generate_sorting_times function to collect data from the efficiency of QuickSort, and compare this data to the theoretical results.

2.5 Section 5

Finally, we will remove the tail recursion of QuickSort and measure whether there is any difference in the times of execution, which is expected to decrease.

3 Tools and Methodology

For this practice we have used again GitHub to share the code and keep track of the versions; for checking errors in the code regarding memory, valgrind; for compiling, gcc; and finally, for plotting the results obtained, GNUPlot.

3.1 Section 1

We followed the pseudocode provided in class for coding the algorithm merge sort, taking into account the dynamic memory use and recursive calls of functions.

3.2 Section 2

Just like in practice I, we ran the algorithm with a ”large” number of permutation and value of N in order to get data for the plots. Because merge sort is considerably faster than local algorithms, we had to increase the number of permutations to order.

3.3 Section 3

For the QuickSort algorithm, the pseudocode from the theory is followed, taking care of memory losses and using the requested function prototypes.

3.4 Section 4

For measuring the efficiency of this algorithm, we have used permutations from 10^6 to 10^7, with an increment of 5.10^5. We have observed that these algorithms are much faster than InsertSort, and so we could not measure any time increments with the sizes used for InsertSort. Thus, we had to multiply the sizes by 100 to be able to have a good sample of the times.

3.5 Section 5

Here we have removed the tail recursion from QuickSort (the second recursive call to the function).

4. Source code

Here you write the source code **only the routines you have developed** in each section.

4.1 Section 1

```
/************************************************/
/* Function: MergeSort    Date:   2020/10/26    */
/* Implementation of the algorithm Mergesort to  */
/* ip: first index of the array                  */
/* iu: last index of the array                   */
/************************************************/

int mergesort(int *table, int ip, int iu){

  int im, count=0;

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

  if(ip==iu)
    return OK;

  im=(ip+iu)/2;

  count+=mergesort(table,ip,im);
  count+=mergesort(table,im+1,iu);
  count+=merge(table,ip,iu,im);

  return count;
}
/************************************************/
/* Function: Merge        Date:   2020/10/26    */
/* Merges two ordered tables into one.          */
/*                                              */
/* ip: first index of the array                 */
/* iu: last index of the array                  */
/* imiddle: index of the                        */
/************************************************/

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

  int* taux=NULL;
  int i,j,k,count=0;

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

  if(!(taux=(int*)calloc(iu-ip+1,sizeof(int))))
    return ERR;
```

```
  for(i=ip,j=imiddle+1,k=0;i<=imiddle && j<=iu;k++,count++){
    if(table[i]<table[j]){
      taux[k]=table[i];
      i++;
    }else{
      taux[k]=table[j];
      j++;
    }
  }

  if(i>imiddle){
    while(j<=iu){
      taux[k]=table[j];
      j++;
      k++;
    }
  }else{
    while (i<= imiddle){
      taux[k] = table[i];
      i++;
      k++;
    }
  }

  for(i=ip,k=0;i<=iu;i++,k++){
    table[i]=taux[k];
  }

  free(taux);
  return count;
}
```

4.3 Section 3

```
/**
 * Function: QuickSort        Date: 2020-10-21
 *
 * This routine returns ERR in case of error
 * or the number of basic operations in case
 * the table is ordered rightly, where table
 * is the table to sort, ip is the first element
 * of the table and iu is the last element of the table.
 *
 * functions: partition and median
 * are used to implement this function
**************************************************/
int quicksort(int* table, int ip, int iu){
    int im, count=0;
    if(ip > iu || table==NULL){
        return ERR;
    }else if(ip==iu){
        return OK;
    }else{
        count=partition(table, ip, iu, &im);
        if(ip < im-1){
            count+=quicksort(table, ip, im-1);
        }
        if(im+1 < iu){
            count+=quicksort(table, im+1, iu);
        }
    }
    return count;
}
int partition(int* table, int ip, int iu, int *pos){
    int im, key, i, count=0;

    median(table, ip, iu, &im);

    key=table[im];

    swap(table+ip, table+im);

    im=ip;
```

```
/**
```

```
 * The following function swaps values of the info of two pointers:

 * -used so that the code of quicksort is easier to read

*/

void swap(int *a, int *b){

    int aux = *(a);

    (*(a))=(*(b));

    (*(b))=aux;

}
```

```
    for(i=ip+1;i<=iu;i++){
        count++;
        if(table[i]<key){
            im++;
            if(i!=im){
                swap(table+i, table+im);
            }
        }
    }
    swap(table+ip, table+im);

    (*(pos))=im;

    return count;
}

/*
 * @brief median
 * This function selects the first index and
 * assigns it to the value pointed by pos
 * */
int median(int *table, int ip, int iu,int *pos){
    (*(pos))=ip;
    return 0;
}
```

4.5 Section 5

```c
/**
 * QuickSort implementation with the
 * removal of the tail recursion
*/
int quicksort_ntr(int* table, int ip, int iu){
    int im, count=0;

    if(ip > iu || table==NULL){
        return ERR;
    }
    if(ip == iu){
        return OK;
    }
    while(ip<iu){
        count+=partition(table, ip, iu, &im);
        if(ip < im-1){
            count+=quicksort_ntr(table, ip, im-1);
        }
        ip=im+1;
    }
    return  count;
}
```

5. Results, Plots

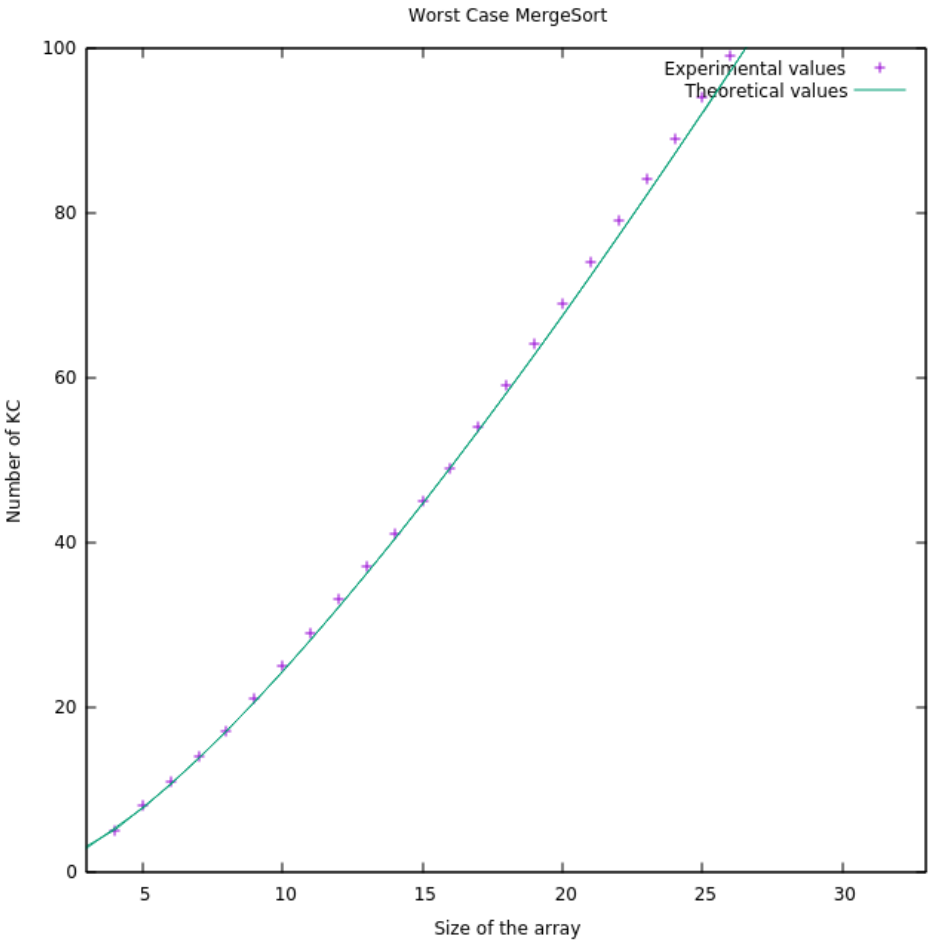
Note: For these results we have tested our algorithms using sizes of 2 to 30, and with one million permutations per size, so that the minimum and maximum KC counts matched as closely as possible the results from the theory. With these sizes, we can still see that as we get to 10-30, the numbers begin to differ from the theory, as it is not possible to make 10! repetitions, or more, in a reasonable time. When measuring the times, we had to use bigger sizes, so that we could measure something, with the precision that our measuring tools has. Therefore, we have used sizes of 1 to 10 million, and 100 permutations for each size.

All the plots follow the same format: the x axis shows the size of the array (N) where the y axis the number of key comparisons or the execution time measured.

5.2 Section 2 **(MergeSort)**

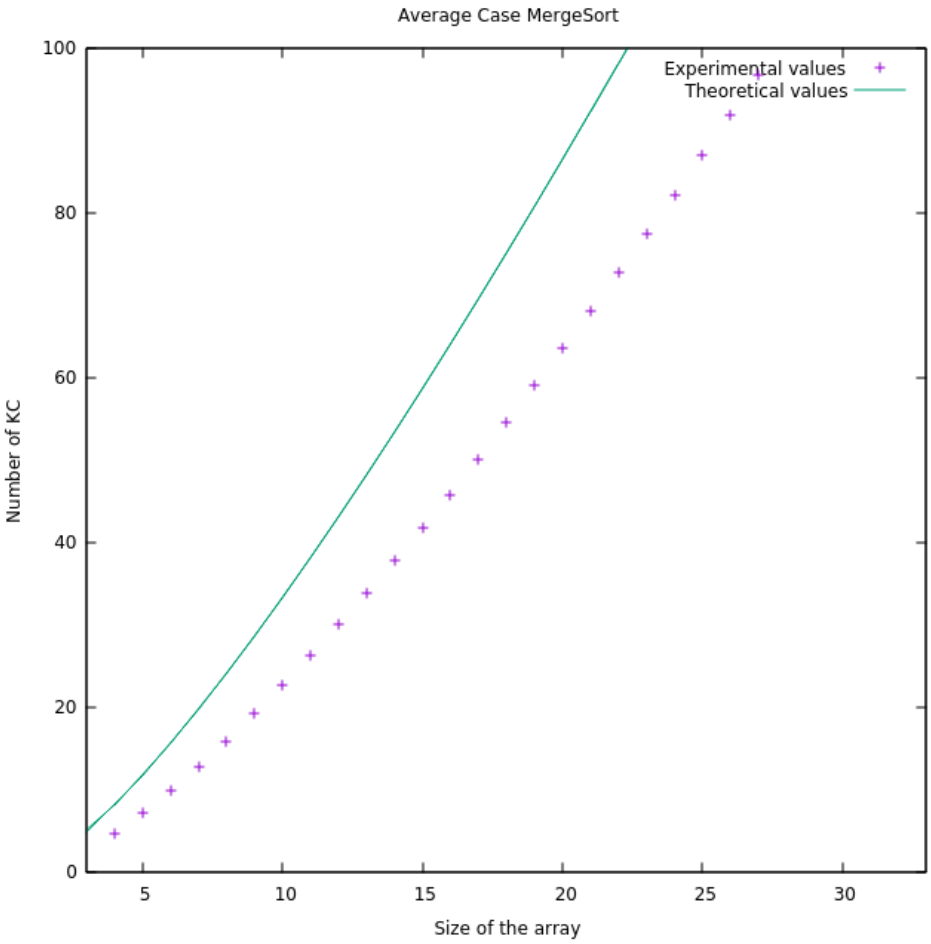
Worst case:

As seen in class, the number of KC in the worst case is a function equivalent to NlogN+O(N), where the logarithm is in base 2. However, it can be more accurately expressed as NlogN-N+1. This last estimation is the one we have used to compare the experimental values. As we see in the plot, the obtained results are very close to what we expected.

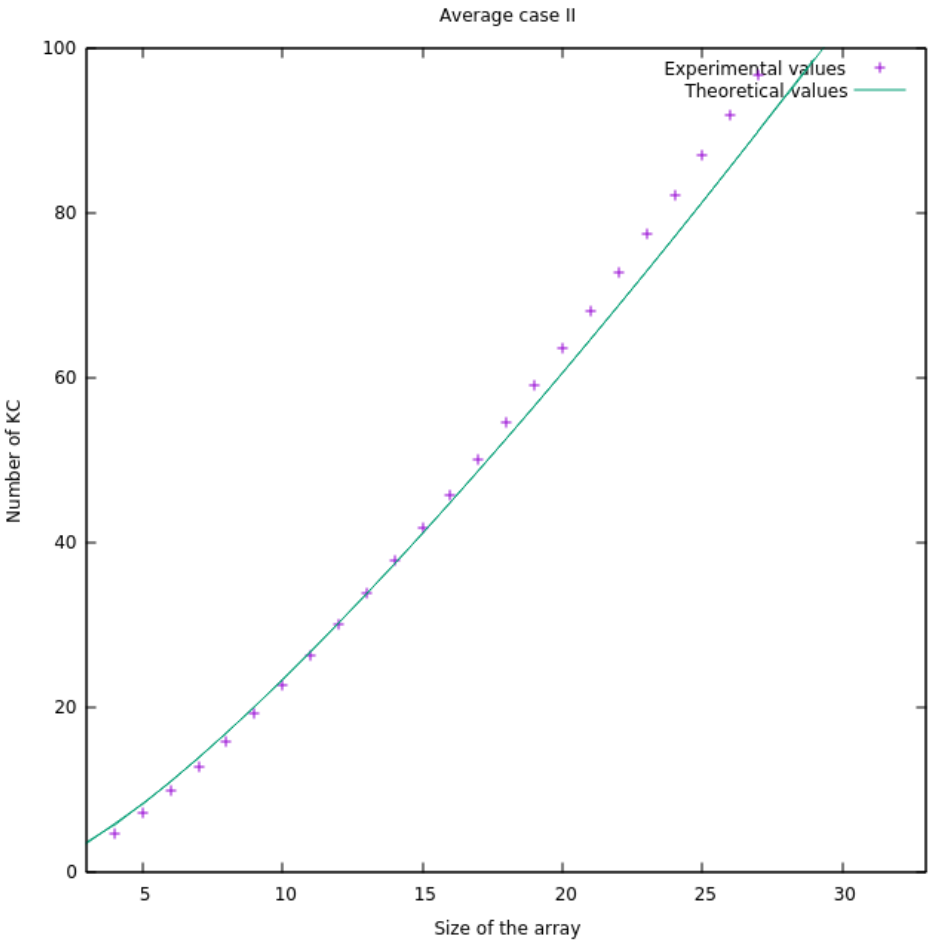


Average case:

This time, the theoretical estimation is not as specific as it was in the worst case. Indeed, it is Θ (Nlog(N)). This estimation does not consider any possible constant in the term Nlog(N) or smaller terms such as something in the order of O(N). In fact, Nlog(N) is slightly higher than Nlog(N)-N+1, which is the worst case. As a result, there is a noticeable difference with the experimental data if we use Nlog(N) as the estimation.



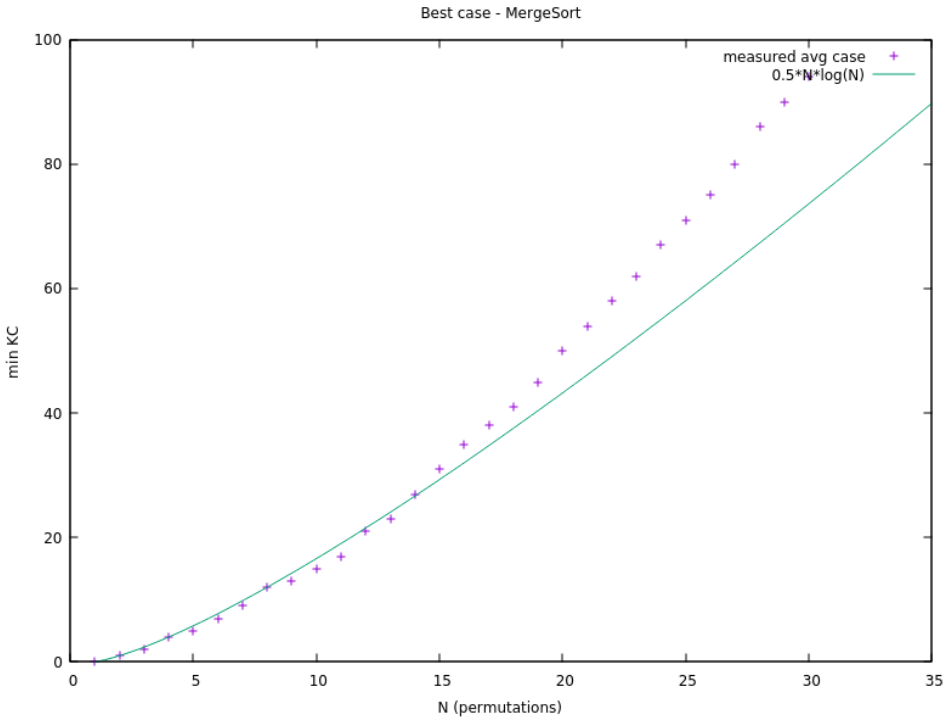
Just to exemplify how does the theoretical curve vary depending on a possible constant in the term NlogN, we have replotted the data using 0.7Nlog(N) as the theorical estimation (0.7Nlog(N))= Θ(Nlog(N)), and it is between Worst Case and Best Case: as in the worst case, the constant is 1, and for the best case, it is 0.5).



As we see in this last plot, the values are closer to the curve than before. Anyway, in both occasions the function associated to the obtained data fulfilled the condition to be Θ(Nlog(N)).

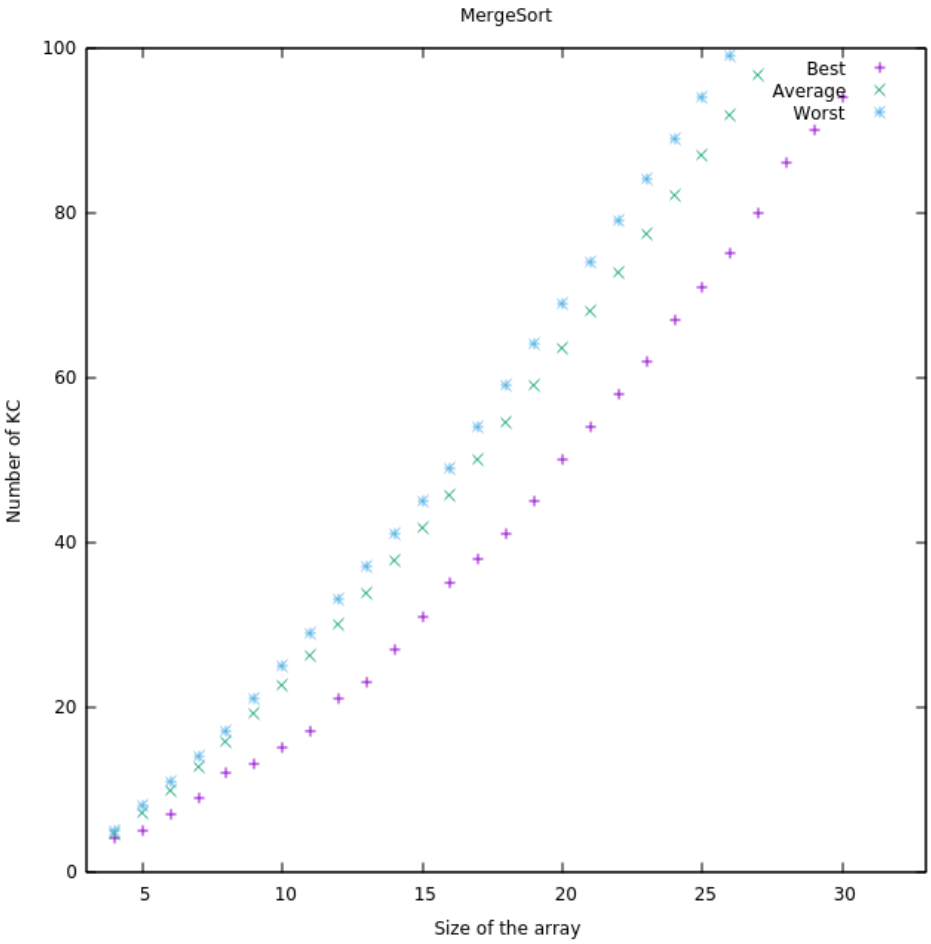
Best Case:

Lastly, the number of KC for the best case is going to be >= (½)Nlog(N) +O(N). It is clear that the empirical data differs considerably with the ideal curve. This is due to a matter of probability: as discussed in the previous note, the number of permutations for each N is N!. We have ordered just a couple of millions of permutations whereas for N=20 we have around 2*10^18 permutations. There is no doubt that there is almost no chance of getting the best case for MergeSort which is when the largest number of a sub-array is smaller to the first element of the other sub-array for each step).



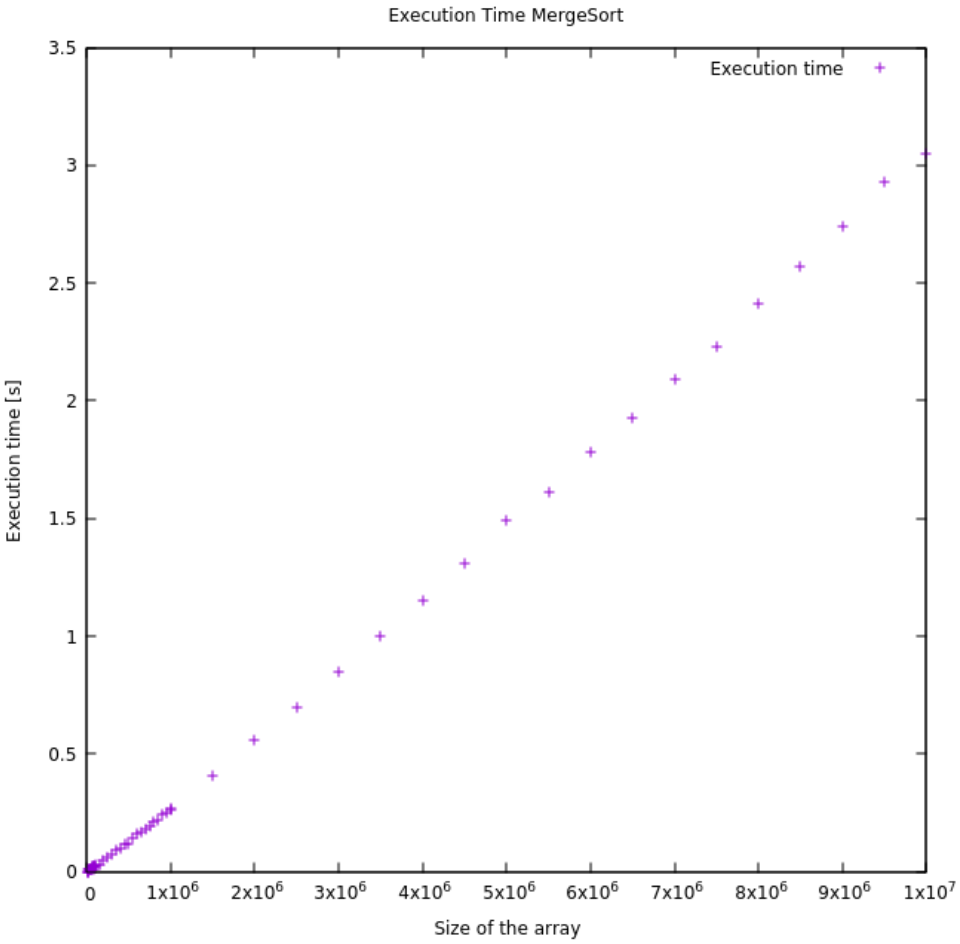
It is worth mentioning the points around N=10 as there are some bellow the theoretical curve. At a glance this may seem wrong as the best case is the lower bound, and therefore, the number of operations must be greater or equal than it. However, we are plotting N/2*log(N), so it is not the perfect lower bound if we analyze the absolute value of KC rather than its asymptotical growth.

Finally, here is a comparation between best, worst and average case for this algorithm



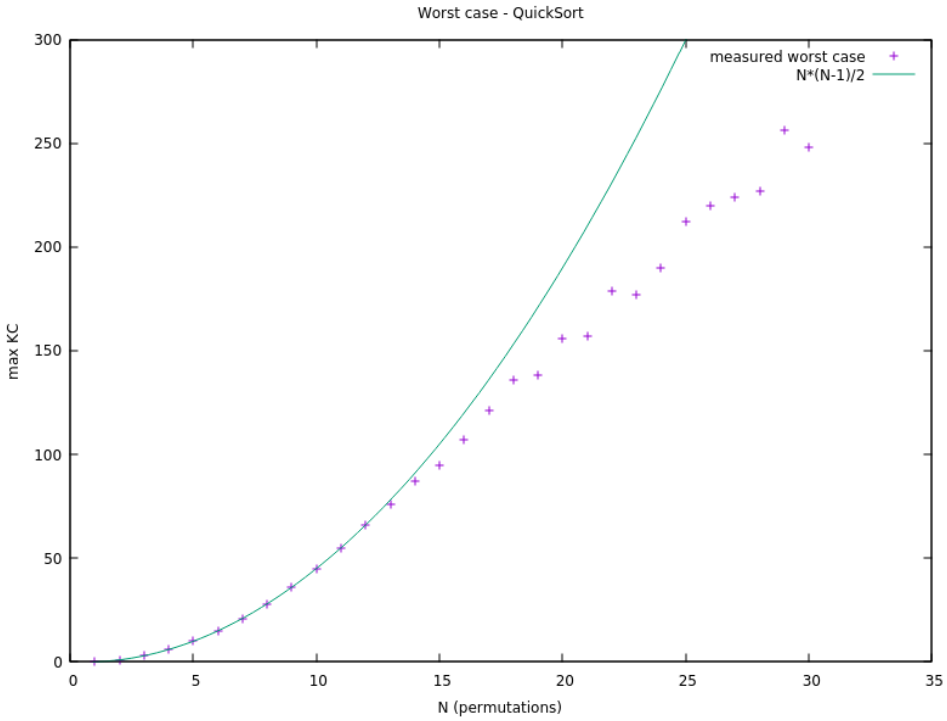
Plot with the average clock time for Mergesort, comments to the plot.

We have chosen large values of N for this part in order to be able to measure the execution due to the limited precision we can get (of milliseconds). As shown in the graph the execution time is no longer a quadratic function as it used to be in the local algorithms. Once more because of the precision, it may seem that the times grows in a linear way, which is not the case.



5.4 Section 4 (QuickSort)

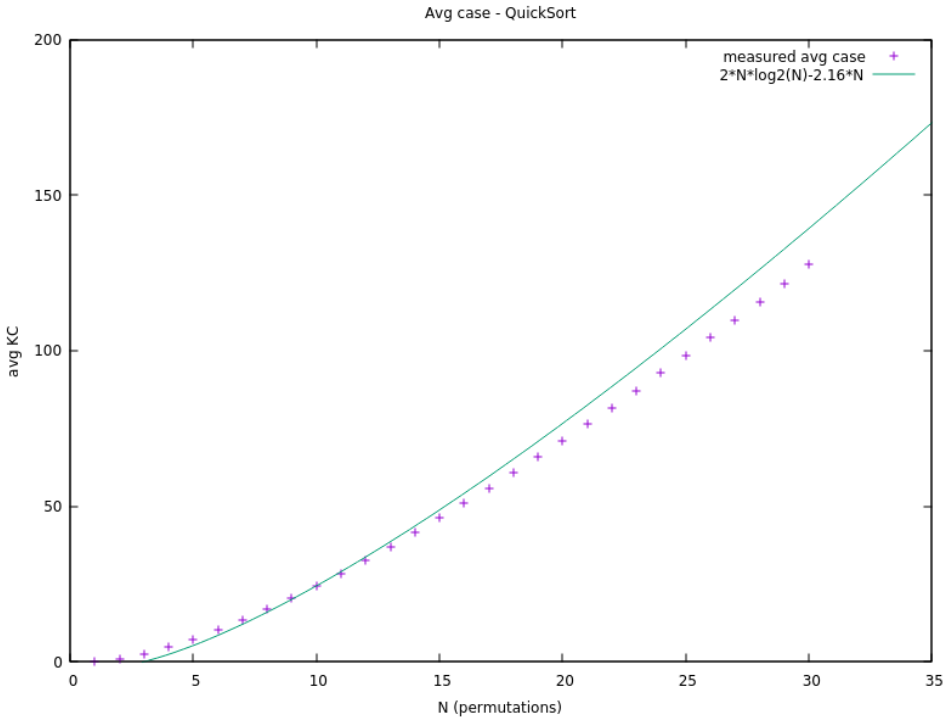
For the worst case of quicksort, we know that it should always be less than (and of the order of) $(N^2-N)/2$. This is exactly the case for the results that we have obtained:



Once again, we see the deviation for N bigger than 15 because of the number of permutations we ordered (not even close to 20!).

For the average case, in theory we proved that $A_{QS}(N)=2Nlog(N)+O(N)$, so we have made the calculation for N=10 of a constant t where $A_{QS}(10)=2*10*log(10)+t*10$, an so $t=(A_{QS}(N)- 2*10*log(10))/10=-2.16$, $A_{QS}(10)$ being the measured avg case for N=10.

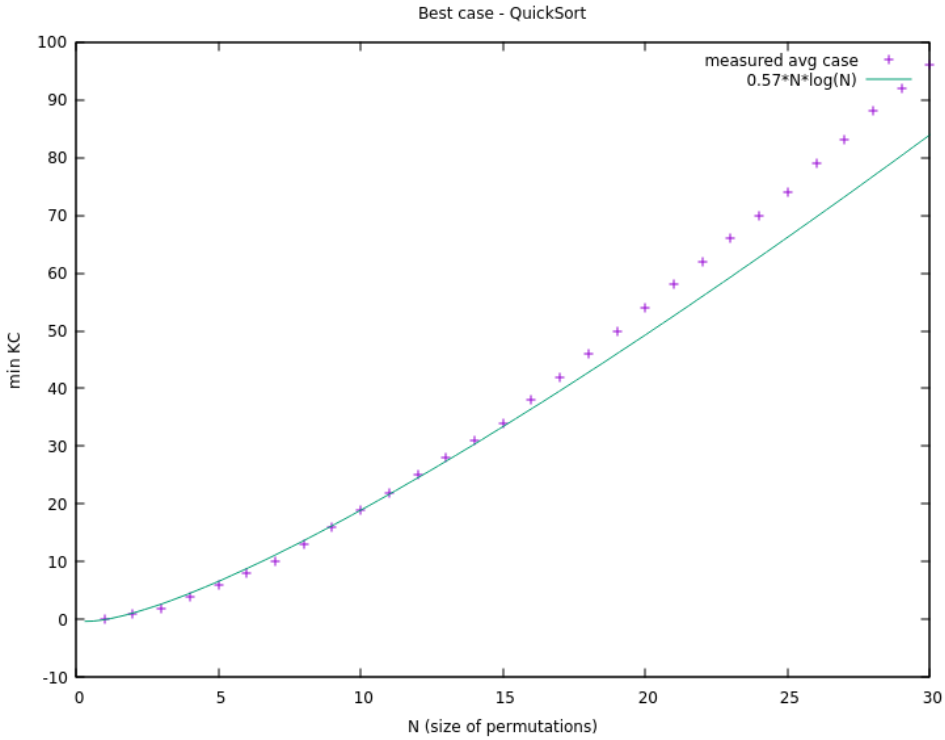
Plotting the measured avg case along with 2Nlog(N)-2.16N:



We can see that the results adjust very well to the expected value.

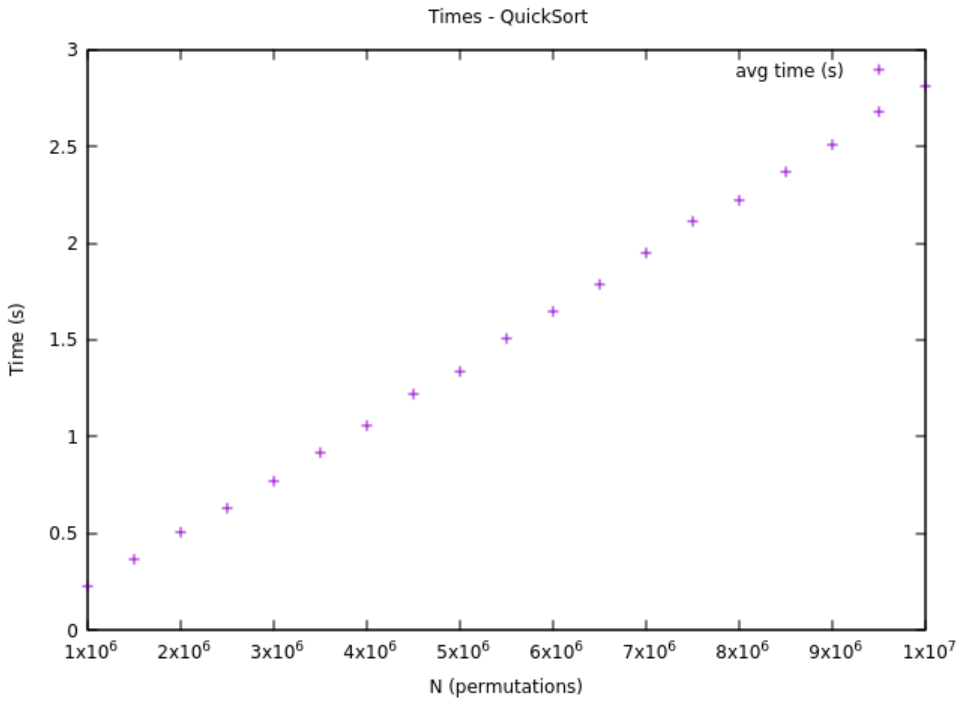
For the best case, we have done the same when calculating the expected function. We know that it should be O(Nlog(N)), so we have calculated a t such that for a measured value (we have again chosen 10), t*10*log(10)=19, which is the measured value for 10=N, so t=0.57.

Here, we plot the function 0.57*N*log(N) alongside our measured data for the minimum KC using QuickSort:



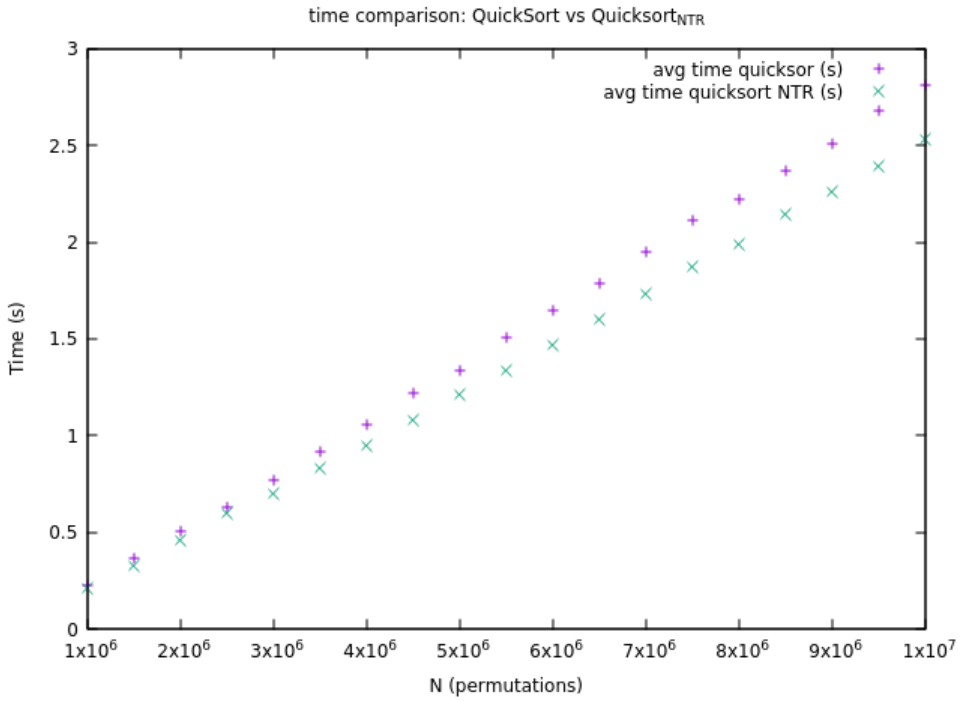
Again, the results are very similar to the expected approximation.

Now, for the times of QuickSort, we can see similar results as in MergeSort, the function seems like a straight line although we know that it has a growth of O(NlogN). And these times were faster than those of MergeSort (this is explained in next section, when answering the questions).



5.5 Section 5 (QuickSort vs QuickSort_NTR)

Using this new function that does not make a many recursive calls as Quicksort, the number of key comparisons is the same, as the two algorithms perform the same steps. The difference is that one does not call so many functions and so makes a better use of the memory. We can see that this makes a notable difference in the times, the function where the tail recursion is removed performs better (needs less time to sort) that the other one. Also, this difference gets bigger the higher the sizes of the permutations are.



6. Answers to theoretical Questions.

Here you answer to the theoretical questions in the practice.

6.1 Question 1

As we have commented on the previous section, the results of the data collected match very well the expected values. For both algorithms, the Average Case is of the order of Θ(NlogN), but QuickSort is 2NlogN+O(N) and, as we have seen (calculated in the previous section), our MergeSort results are like 0,7*NlogN+O(N). Which was not seen in the theory, we only knew that the constant multiplying NlogN was between 0,5 and 1.

6.2 Question 2

The number of operations (key comparisons) needed by quicksort and quicksort_ntr is the same as both algorithms use the same strategy to sort the arrays. However, the use of memory differs. The version without tail recursion uses less memory because it stores less recursive calls in the stack, and therefore its times are lower than those of the version with more recursive calls.

In fact, the implementation with recursion could cause stack overflow for a large enough N.

6.3 Question 3

The best case for MergeSort is when the largest number of the (right) sub-array is smaller than the first number of the opposite (left) sub-array for every step. This is because we want the smallest table to be emptied the first, which means the minimum number of key comparisons. And the right table will be of size less or equal than the left one (when the size of any table found in the recursion is odd, the right will be smaller). As a result, although there is not a unique best array for N-size, the table ordered in reverse order will always be the best case: [N, N-1, …, 3, 2, 1]. For example, we can find other best cases: considering N=4, [1,2,3,4] is one of the best cases and so is also the array [2,1,4,3].

Consequently, the worst case for MergeSort is when the subarrays contain alternate number for each recursive step. If this is the case, each element has to be compared once, hence getting the worst case. For example, for N=8 the worst case would be [1,5,3,7,2,4,6,8].

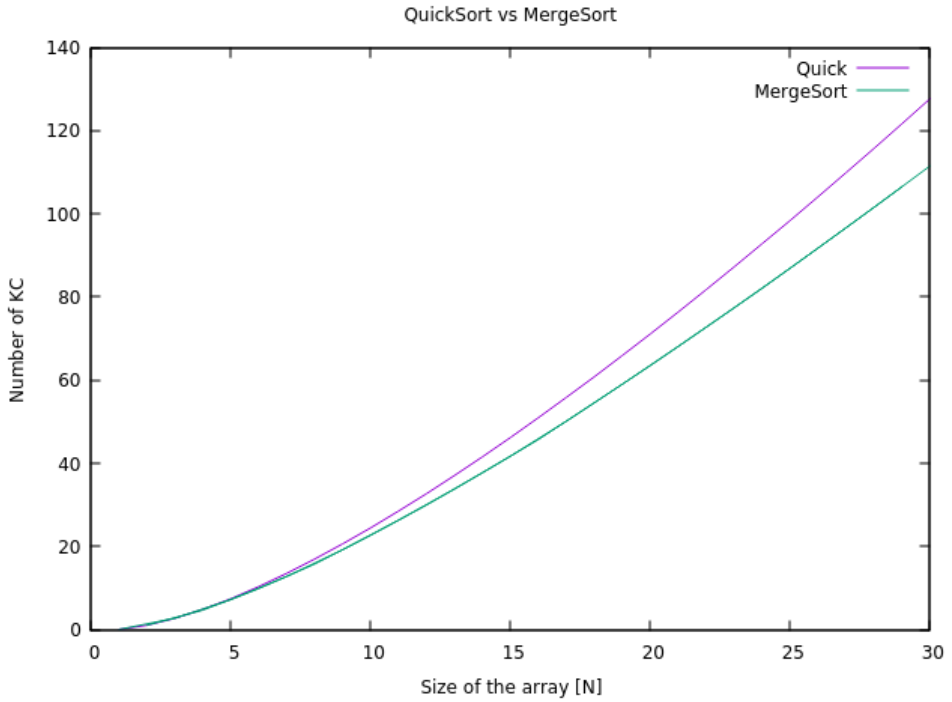
For analyzing the best case we will assume N=2^k for the sake of clarity. The best case would be to select N/2 as the first pivot, the N/4 … If we take into account that the pivot is the first element, our desired table would be [N/2, N/4, N/8 …,1,…]

Finally, the worst case for QuickSort occurs with the ordered array (or the inverse ordered array) because of how we are choosing the pivot, which is always the first element. In the step i the algorithm will do N-i comparisons which result in a cost of (N^2-N)/2, just as inefficient as the local algorithms.

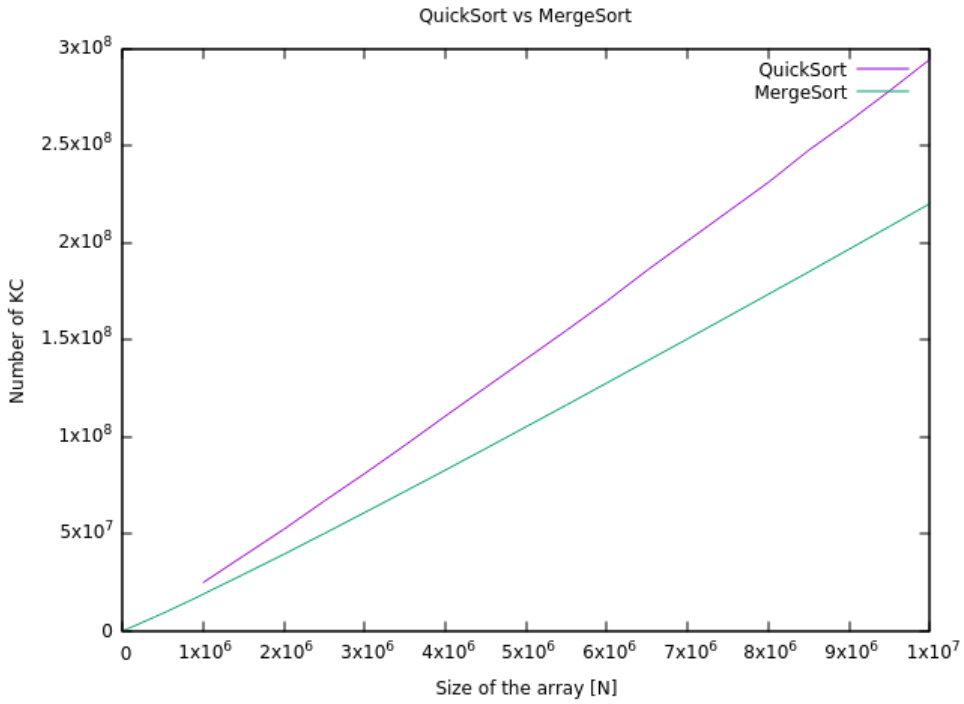
If we wanted to calculate the worst and best case for a given number N, we would have to make a new function that generates all possible permutations for size N and performs the algorithm on each of them, storing always the table for which the algorithm performs the least (or the most) operations.

6.4 Question 4

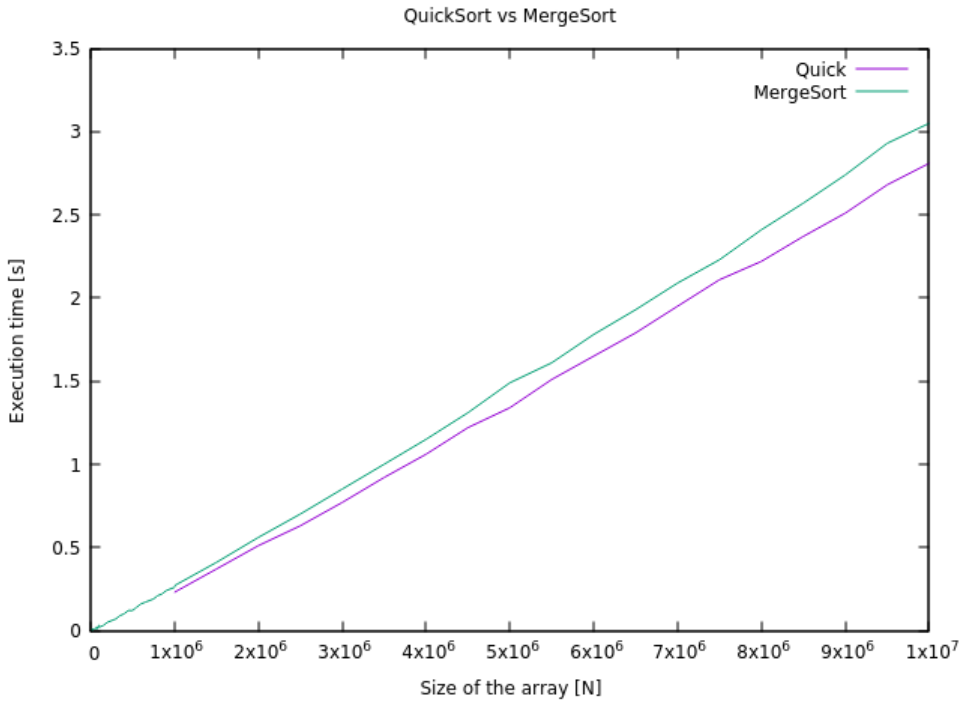
For small values of N we can see that there is no much difference between the number of key comparisons each algorithm performs in average, MergeSort does a few less though.



However, when we look to large values of N the difference is quite considerable; we can conclude that for big arrays MergeSort is more efficient in terms of key comparisons. Notice that once again the curves of KC for large values of N seem straight lines, especially in the case of MergeSort. This is because of how we are plotting and the range of the axes.



Regarding to memory use, MergeSort has to double the memory during the merge operation whereas QuickSort only has to deal with one recursive call (in the non-tail recursion version). Therefore, QuickSort is much more efficient in terms of memory use. Indeed, every time Merge is called (combining two sub-arrays) a calloc is made. This extra operation affects the execution time of Merge, which is slower than Quick even though it is performing less KC.



7. Final Conclusions.

In this practice we have been able to notice the great difference between local algorithms and “divide and conquer” algorithms. Furthermore, as explained in the Question section, the tradeoff between memory use and execution time (ART) is one of the key aspects when comparing these two algorithms. Although one is significantly faster, it needs the double of memory which could be a problem depending on the task and limitations of the system.