Analysis of Algorithms 2020/2021

Practice 2

Kevin de la Coba Malam and Marcos Bernuy, 1292.

Code	Plots	Memory	Total

1. Introduction.

In this document we are going to explain how we solved the assignment 2. The main goal of the assignment is to build two sorting functions: *MergeSort* and *QuickSort*. In order to build these 2 sorting functions, we used C. Apart from these functions we also have to build the ones used by the algorithms, such as *merge*, *split*, *median*… All the code related to these algorithms is in the sorting.c file which was the one we had to modify because the other files had to remain as they were in the assignment 1. After building the functions we just have to generate data regarding the performance of the algorithms given different sizes.

2. Objectives

2.1 MergeSort Algorithm

The main goal of the section is to implement the algorithm MergeSort in C, in the sorting.c file.

2.1.1 Implement the MergeSort in C.

We were given the prototype of the function *MergeSort*:

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

And the prototype of the function *merge* (which is used by MergeSort):

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

Given these two functions we had to implement the algorithm in the sorting.c file.

2.1.2 Run exercise5.c with the function MergeSort.

Once the function is built we have to check the performance of it by running the exercise5.c, but this time the generate_sorting_times must use the MergeSort function.

2.2 Quicksort Algorithm

The main goal of this section is to implement the algorithm Quicksort in C, in the sorting.c file. Additionally, we have to build another version of the algorithm which doesn't have tail recursion.

2.2.1 Implement Quicksort in C.

We were given the prototype of the function *Quicksort*:

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

The *split* prototype used by Quicksort was given as well:

```
int split(int* table, int ip, int iu,int *pos)
```

And the last prototype given was *median*:

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

With these three prototypes we had to build the functions in such a way that it sorts permutations of different sizes.

2.2.2 Run exercise5.c with the function Quicksort.

When the functions are built and ready to use, we had to run the exercise5.c with the function Quicksort, so we can generate the data related with the performance of the function. Again, we had to change the function generate_sorting_times in the exercise5.c

2.2.3 Create the function Quicksort_ntr (no tail recursion) and test it.

In this section we had to do two things. First build a Quicksort function which does not have tail recursion.

After building this second function we had to generate the performance of the function using exercise5.c.

3 Tools and Methodology

We implemented and tested the code in Linux, Ubuntu. We used visual studio code for implementing the code in C, for testing it and debugging it too.

3.1 MergeSort implementation and testing.

3.1.1 Implementation

In this part we used the pseudocode in the theory slides. The implementation was not exactly as it was in theory slides. The functions mergesort and merge return the basic operations performed by each one.

We were doubting about incrementing the basic operations when the algorithm copies the auxiliar table values in the original table, we decided to not count those operations as basic operations. A basic operation is the main operation performed by the algorithm, the one which defines the algorithm, in sorting algorithms this operation is called key comparison because we compare two values from a table in order to do or not a swap or whatever is done in the algorithm, but as the name says it is a comparison. When we copy the values from the auxiliar table we are not doing any comparisons, and we are not taking into account the comparisons done in loops. This would be an example of the

comparisons I refer to “int I = 0; I < j;”, these comparisons are not between values from the table. This is the logic behind our solution.

3.1.2 Testing

For this part we just simply took the exercise5.c file from the previous practice and we used the MergeSort function in order to check the behavior of it. The exercise5.c generated a file with all the data related with the performance of the algorithm.

3.2 Quicksort implementation and testing.

3.2.1 Implementation

For implementing the algorithm, we just took the pseudocode and adapted it to C. As before, the function is not the same, in our case the pivot is sent through the split function by reference, this means that the function split had to return the basic operations performed by it. Apart from this, the code is very similar to the pseudocode.

3.2.2 Testing

As we did in the section 3.1.2, we changed the exercise5.c and we used the quicksort function, and we generated all the data needed to check the performance of the algorithm.

3.2.3 Building quicksort without tail recursion.

In this part we followed the schema that our lab teacher did when she explained the way we should implement the function. We removed the second call to quicksort in the quicksort function and we created a loop which stopped when the first index was the same as the second one. This implementation was giving us errors, we debugged the code and we saw that we were missing a condition for the loop, that the first index is lower than the second one. When these two conditions are not met, the while stops.

3.2.4 Testing quicksort without tail recursion.

The same procedure as before, change exercise5.c with our new function and run it.

4. Source code

4.1 MergeSort

```
/**
 * @brief Function which sorts an array
 *        with the mergesort algorithm
 *
 * @param table Array to be sorted
 * @param ip Index of the first element to be sorted
 * @param iu Index of the last element to be sorted
 * @return int Basic operations that the algorithm performs
 */
int mergesort(int *table, int ip, int iu) {

    int m = 0, bo = 0, ret = 0;

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip)
        return ERR;

    /* In case there's only one integer in the array */
    if (ip == iu)
        return OK;

    /* Getting the middle value of the array */
    m = (ip + iu) / 2;

    ret = mergesort(table, ip, m);
    if (ret == ERR) return ERR;
    bo += ret;

    ret = mergesort(table, m + 1, iu);
    if (ret == ERR) return ERR;
    bo += ret;

    ret = merge(table, ip, iu, m);
    if (ret == ERR) return ERR;
    bo += ret;

    return bo;
}
```

4.1.1 Merge

```
/**
 * @brief Function used by mergesort which
 *        merges (sorts) one table divided
 *               by a middle index.
 *
 * @param table Array to be sorted
 * @param ip Index of the first element to be sorted
 * @param iu Index of the last element to be sorted
 * @param imiddle Middle index that separates both tables
 * @return int Basic operations that the algorithm performs
 */
int merge(int *table, int ip, int iu, int imiddle) {

    int i = 0, j = 0, k = 0, size = 0, counter = 0, *table2 = NULL;

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip || imiddle < 0 ||
ip > imiddle || iu < imiddle)
        return ERR;

    size = iu - ip + 1;
    table2 = (int *)malloc(size * sizeof(table2[0]));
    if (table2 == NULL)
        return ERR;

    i = ip;
    j = imiddle + 1;
    k = 0;
    while (i <= imiddle && j <= iu) {
        if (table[i] < table[j]) {
            table2[k] = table[i];
            i++;
        }
        else {
            table2[k] = table[j];
            j++;
        }
        k++;
        counter++;
    }

    if (i > imiddle) {
        while (j <= iu) {
            table2[k] = table[j];
            j++;
            k++;
        }
    }
    else if (j > iu){
        while (i <= imiddle) {
            table2[k] = table[i];
            i++;
            k++;
```

```
        }
    }

    /* Copying table2 on table */
    for (i = ip, j = 0; i <= iu; i++, j++){
        table[i] = table2[j];
    }

    free(table2);
    table2 = NULL;
    return counter;
}
```

4.2 Quicksort

```
/**
 * @brief Function which sorts an array
 *        with the quicksort algorithm
 *
 * @param table Array to be sorted
 * @param ip Index of the first element to be sorted
 * @param iu Index of the last element to be sorted
 * @return int Basic operations that the algorithm performs
 */
int quicksort(int *table, int ip, int iu) {

    int pivot = 0, bo = 0, ret = 0;

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip)
        return ERR;

    /* In case there's only one integer in the array */
    if (ip == iu)
        return OK;

    ret = split(table, ip, iu, &pivot);
    if (ret == ERR) return ERR;
    bo += ret;

    if (ip < pivot - 1) {
        ret = quicksort(table, ip, pivot - 1);
        if (ret == ERR) return ERR;
        bo += ret;
    }
    if (pivot + 1 < iu) {
        ret = quicksort(table, pivot + 1, iu);
        if (ret == ERR) return ERR;
        bo += ret;
    }

    return bo;
}
```

4.2.1 Split

```
/**
 * @brief Function used by quicksort which
 *        splits the table in two parts
 *
 * @param table Table to be splited
 * @param ip First index of the table
 * @param iu Last index of the table
 * @param pos Variable where we are going to store the value of the
pivot
 * @return int Basic operations that the function performs
 */
int split(int *table, int ip, int iu, int *pos) {

    int ret = 0, counter = 0, k = 0, i = 0;

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip || pos == NULL)
        return ERR;

    ret = median(table, ip, iu, pos);
    if (ret == ERR) return ERR;
    counter += ret;

    k = table[*pos];

    if (swap(&table[ip], &table[*pos]) == ERR) return ERR;

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

    if (swap(&table[ip], &table[*pos]) == ERR)
        return ERR;

    return counter;
}
```

4.2.2 Median

```
/**
 * @brief Function used by quicksort which
 *        calculates the pivot
 *
 * @param table Table where the pivot is calculated
 * @param ip First index of the array
 * @param iu Last index of the array
 * @param pos Variable where we are going to store the value of the
pivot
 * @return int Basic operations that the function performs
 */
int median(int *table, int ip, int iu, int *pos) {

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip || pos == NULL)
        return ERR;

    *pos = ip;

    return 0;
}
```

4.2.3 Quicksort_ntr

```
/**
 * @brief Function which sorts an array
 *        with the quicksort algorithm,
 *        this version doesn't have recursion
 *
 * @param table Array to be sorted
 * @param ip Index of the first element to be sorted
 * @param iu Index of the last element to be sorted
 * @return int Basic operations that the algorithm performs
 */
int quicksort_ntr(int *table, int ip, int iu) {

    int pivot = 0, bo = 0, ret = 0;

    /* Checking arguments */
    if (table == NULL || ip < 0 || iu < 0 || iu < ip)
        return ERR;

    /* In case there's only one integer in the array */
    if (ip == iu)
        return OK;

    while(ip != iu && ip <= iu){

        ret = split(table, ip, iu, &pivot);
        if (ret == ERR) return ERR;
        bo += ret;

        if (ip < pivot - 1) {
            ret = quicksort_ntr(table, ip, pivot - 1);
            if (ret == ERR) return ERR;
            bo += ret;
        }
        ip = pivot+1;
    }

    return bo;
}
```

4.3 Swap function (auxiliar)

```
/**
 * @brief Function which swaps two elements
 *
 * @param first element to be swapped
 * @param second element to be swapped
 * @return int State of the function
 *             ERR if fail, OK if success
 */
int swap(int *el1, int *el2) {

    int aux = 0;

    /* Checking arguments */
    if (el1 == NULL || el2 == NULL)
        return ERR;

    /* The swap */
    aux = *el1;
    *el1 = *el2;
    *el2 = aux;

    return OK;
}
```

5. Results, Plots

5.1 Mergesort results

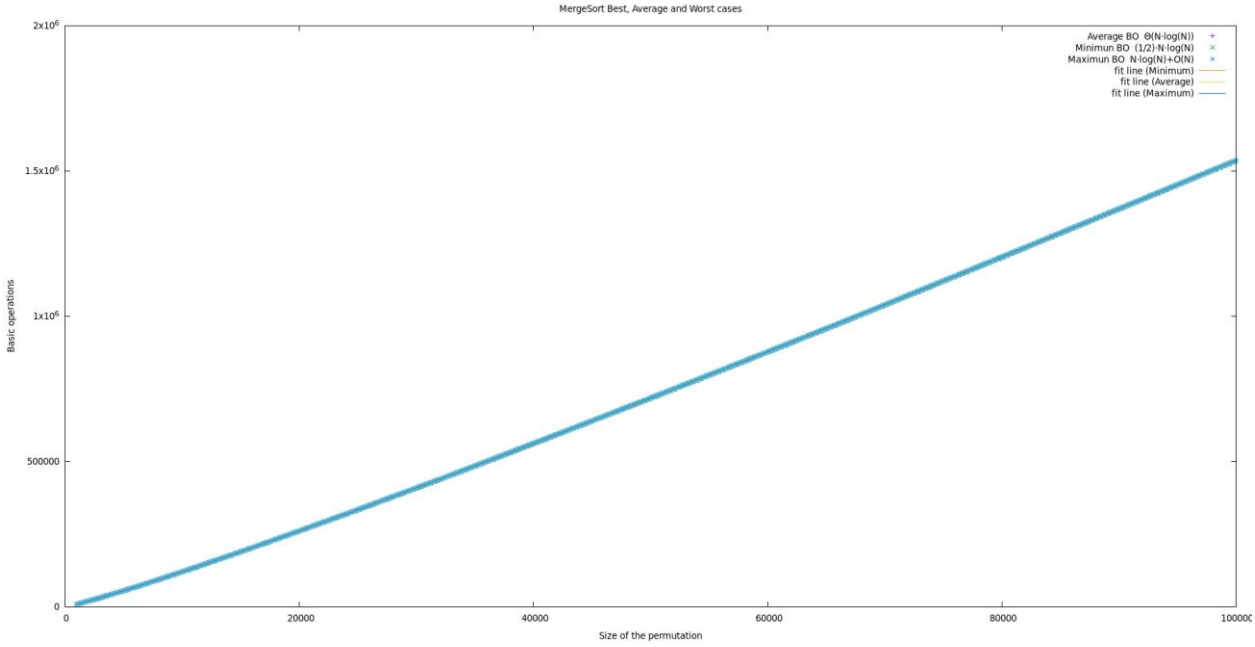
exercise4.c was modified in order to check the correctness of the function.

./exercise4 -size 10

1 2 3 4 5 6 7 8 9 10

Given a random permutation the mergesort function sorts it in the expected way.

5.2 Mergesort basic operation plot

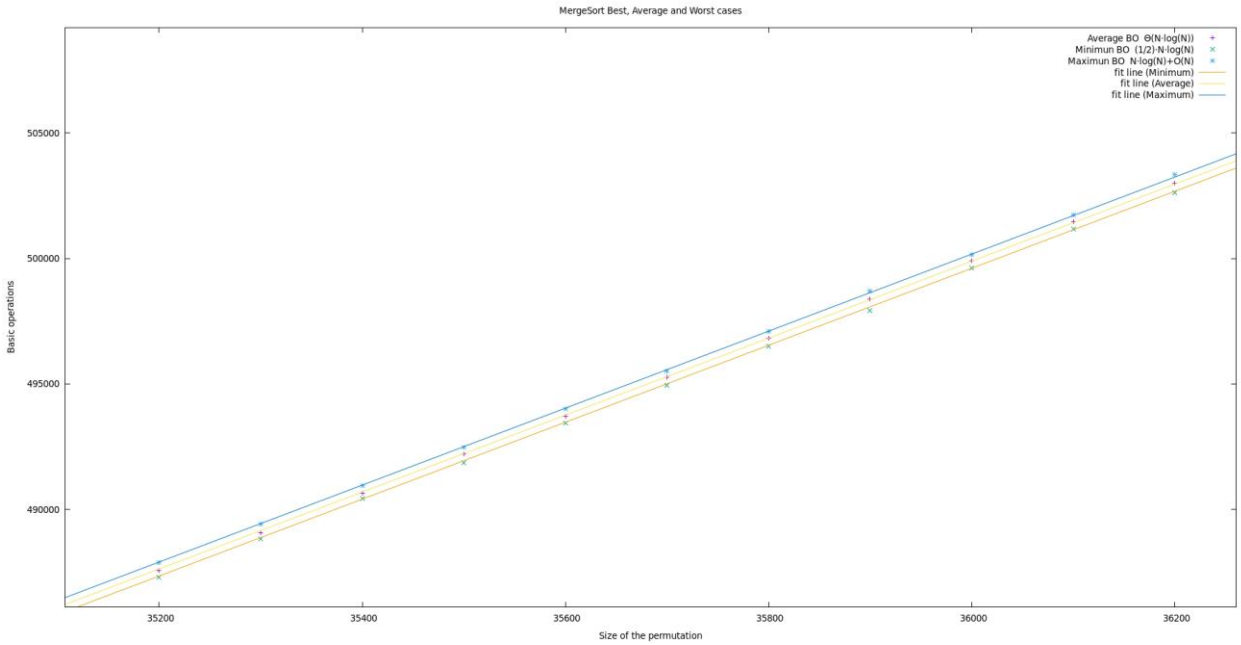


100	1000	2.374100e-04	8708.540000	8662	8761
100	100000	9.009070e-03	1536365.370000	1535817	1536832

In this plot we can see the data related with the performance of the Mergesort algorithm after sorting 100 tables with sizes of 1000 elements to 100000 incrementing in each iteration the size by 100.

We can see in the plot that the increase of the slop for either the average, the worst or the best case is pretty steady, keeping the lines representing each case really close between them. The reason would be that the three cases grow in a logarithm way, all of them have log(N) as part of the function and therefore as a general view we barely see any difference.

Another reason why we can't see a whole lot of difference is because we are using small values to represent the plot. Nevertheless, if we used bigger number of permutations, using mergesort we still wouldn't see much difference between the average case, the worst case and the best case.



Here we have a representation of the Mergesort plot but zoomed in so we can check and visualize the growth of the function better. As it has been previously mentioned we can see that the growth stays steady and monotonous along the function keeping the same distance between each case. When comparing the minimum and the average case we could say that the distance between the fit lines will stay the same. This is because each case is N·log(N) multiplied by a constant which can take values 1 (average case) or ½ (minimum case). And finally when it comes to the worst case, it will keep a close distance but it won't be as constant as the other two, because the difference is due to an addition.

5.3 Quicksort results

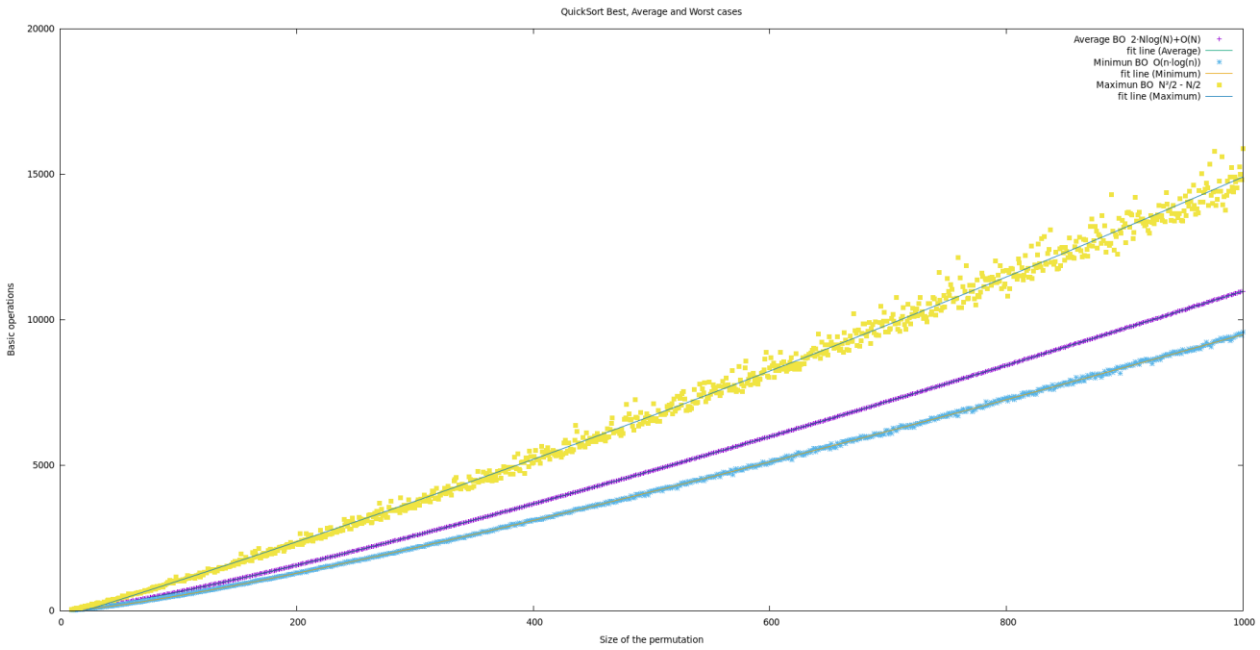
exercise4.c was modified in order to check the correctness of the function.

./exercise4 -size 10

1 2 3 4 5 6 7 8 9 10

Given a random permutation the Quicksort function sorts it in the expected way.

5.4 Quicksort basic operations plot.



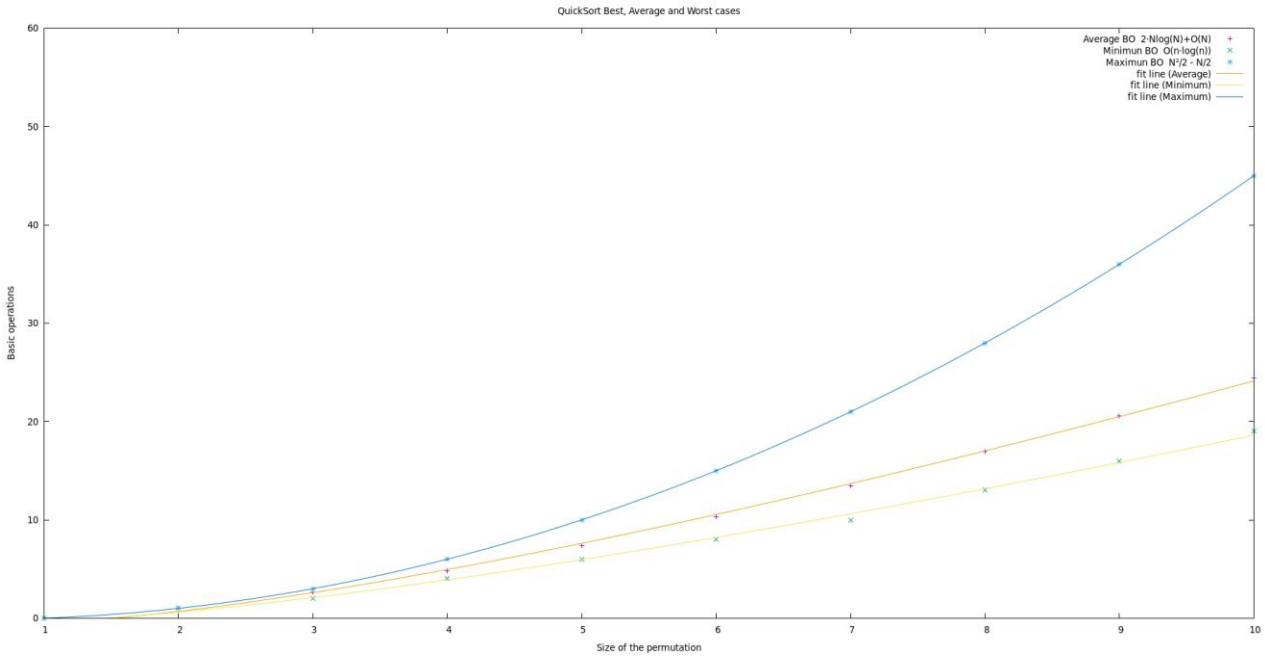
100	1000	6.736000e-05	10965.190000	9709	12801
100	100000	7.260190e-03	2024725.760000	1898994	2179866

In this plot we can see the data related with the performance of the Quicksort algorithm after sorting 100 tables with sizes of 1000 elements to 100000 incrementing in each iteration the size by 100.

We can see the maximum, minimum and average basic operations done after sorting. We can see how the growth of the worst case is every time stepping aside from the other two, the reason for this is that the worst case of quicksort is growing in a quadratic way, but the other 2 are growing in a similar way as a logarithm function (average case – $2 \cdot N \cdot \log N$, best case $O(N \cdot \log N)$).

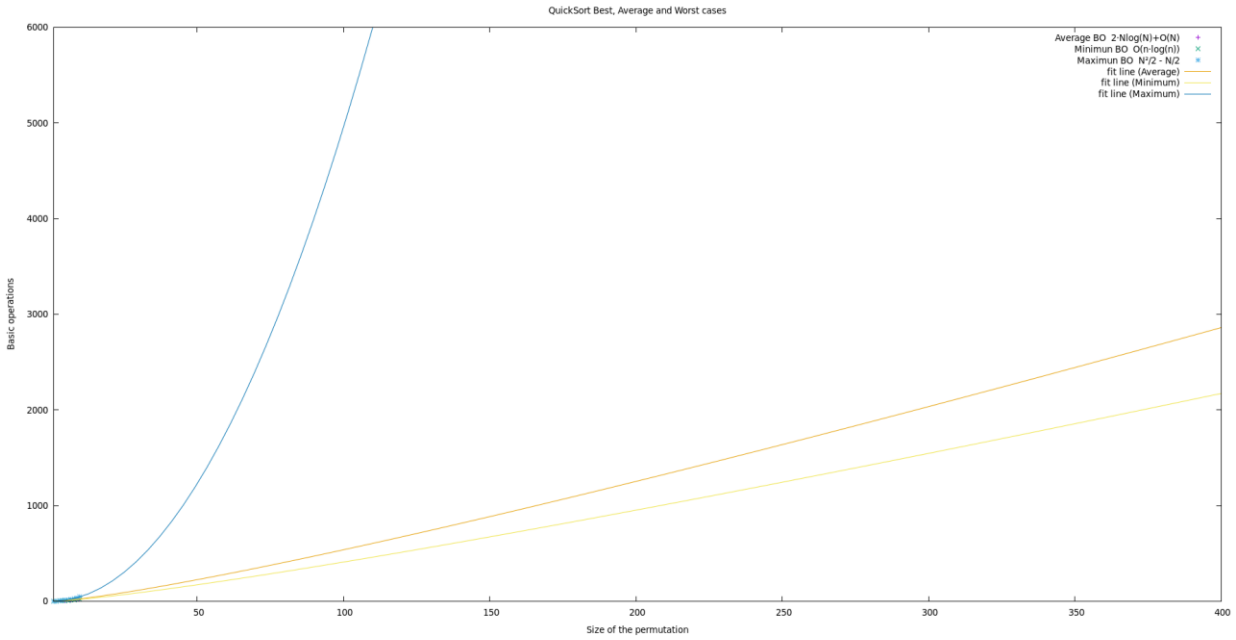
As we saw in theory, that each sample (worst, best and average) is fitting correctly to their corresponding growth seen in theory.

But, if we check the worst case we can see that something strange happens. The sample should have bigger values, it looks like the growth of it is not $N^2/2-N/2$. This has a very simple explanation and is that we are generating 100 permutations per each iteration, the first size is 1000, so there are 1000! possible tables and we are just generating 100. This means that the probably of having the worst possible table is 100/1000!, which is pretty is small. What is going to happen is that most of the tables made are going to be disordered, but they are not going to be the real worst case.



10000000	1	3.522500e-09	0.000000	0	0
10000000	10	1.943755e-07	24.437160	19	45

For this plot we used a very small size, from 1 to 10, but we set 10000000 permutations to solve. What we get with these parameters are the real worst cases. Now, we are able to see in a better way the quadratic growth of the worst case.



For a better view of the growth, we are showing now how the results would look if we were able to generate in each iteration the worst case. As we can see the worst case grows way faster than the others.

For creating this plot, we used the same arguments as before, the main difference is that in this case the data is not related with the basic operations, it is related with the time spent for sorting. We can see that as it was before the growth of the sample is a logarithm growth. This time we don't have too many outliers in the sample, which let us see how the sample fits to the logarithm growth, this confirms us what we saw in theory about the growth of quicksort.

5.5 Quicksort_ntr results

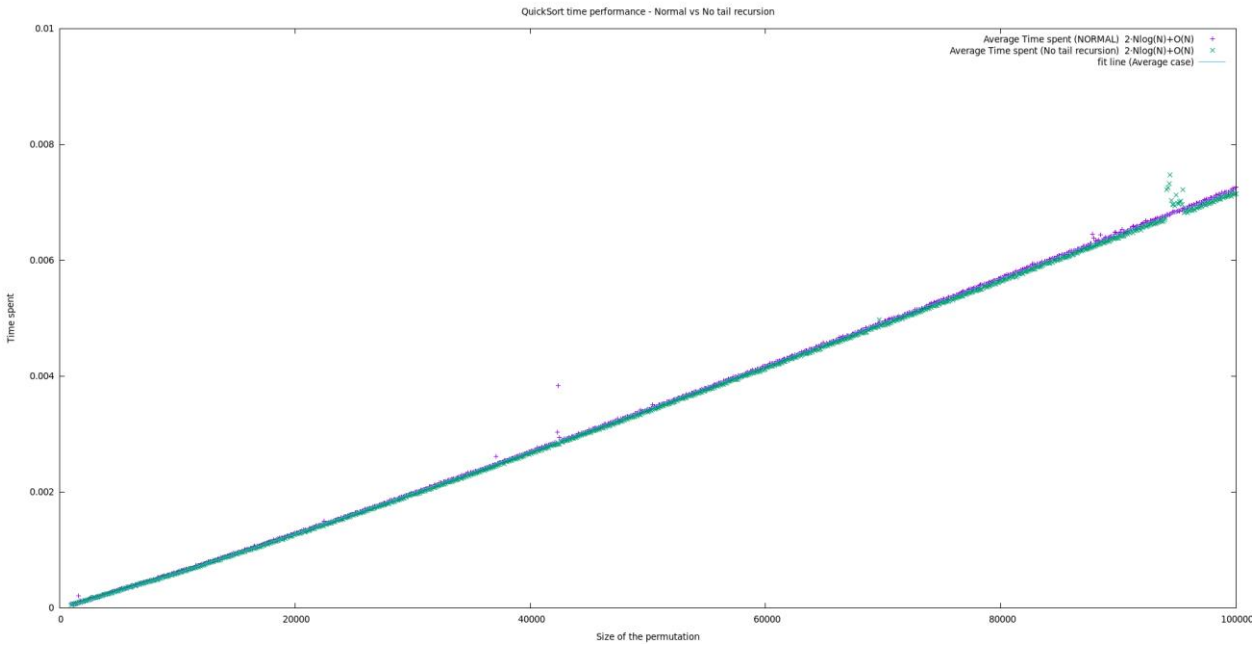
Quicksort_ntr

exercise4.c was modified in order to check the correctness of the function.

./exercise4 -size 10

1 2 3 4 5 6 7 8 9 10

Given a random permutation the Quicksort_ntr function sorts it in the expected way.



As we can see in the plot the behavior of both functions regarding time is pretty similar. Removing tail recursion doesn't improve the performance of the algorithm regarding time, but if we could see a diagram representing the use of the stack we would see that there's a big difference because we removed from the algorithm the second call.

5. Answers to theoretical Questions.

5.1 *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?*

Mergesort

Average case - Θ(N·log(N))

We generated a file where we sorted 10000000 permutations from size = 1 to 10.

lim 22,66/10·log(10) = 2,62 != 0

$$if\ f(N) = \Theta(N \cdot \log(N))\ then\ \lim_{N\to\infty} \frac{f(N)}{N \cdot \log(N)} = L\ \neq 0$$

For us, f(N) is the result obtained while running the algorithm. f(N) = 22,66.

$$\lim_{N\to 10} \frac{22{,}66}{10 \cdot \log(10)} = 2{,}62\ \neq 0$$

We can conclude that the average is correct.

Quicksort

$$\text{Average case} - \ 2 \cdot \text{Nlog(N)} + \text{O(N)}\ ) \rightarrow\ 2 \cdot 10 \cdot \log 10\ +\ \text{O}(10)\ =\ 20\ +\ \text{O}(10)$$

$$\text{O}(10) =\ \text{C} \cdot 10\ \text{ where C is bigger or equal to 0.}$$

We have that the average case with size 10 after sorting 10000000 permutations is 24,43 so 20+O(10) ≈24,43

5.2 *Analyze the results obtained when comparing quicksort con quicksort src both in the cases you observe differences or not.*

If we compare the time performance we can see that there are not differences between both algorithms, this also happens comparing the basic operations. The only improvement is regarding the stack. Removing the tail recursion makes the algorithm use less stack, so this is the unique improvement.

5.3 *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

Best case - $(1/2)\cdot N\cdot\log(N)$

Worst case - $N\cdot\log(N)+O(N)$

Average case - $\Theta(N\cdot\log(N))$

Quicksort

Best case - $N^2/2 - N/2$

Worst case - $O(n\cdot\log(n))$

Average case - $2\cdot N\log(N)+O(N)$

In order to calculate strictly each case of the algorithm we should at least make N! permutations to sort, but even we sorted N! in each iteration, it's possible that the worst case or the best case is not generated.

To sum up, we have to generate more than N!.

5.4 *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.*

The most efficient algorithm is **mergesort** (although it uses more memory). If we, for example, take the average cases of both algorithms, we can see in our samples that with a size of 100000 the basic operations done by mergesort are 1536365.37, and by quicksort are 2017454.41, this shows that mergesort is way more efficient algorithm regarding basic operations.

Let's take now the theoretical prediction with the same size:

- For quicksort we have that the average case is $2\cdot N\cdot\log N + O(N)$, $2*100000*\log100000+O(100000) = 3321928$, which is not close to 2017454, but we have to take into account what it was said in the 5.4 part, we are not generating enough permutations to have the real values, in order to have that we would have to use the small sample.
    - Worst case: $N^2/2 – N/2$ || $10^2/2\text{-}10/2 = 45$. It is the same
    - Average case: $2\cdot N\cdot\log N + O(N)$|| $2\cdot10\cdot\log10 = 20$ so $24.47 \leq 2\cdot10\cdot\log10+C\cdot10$ being $C \geq 0$.
    - Best case: $O(N\cdot\log N)$ || $10\cdot\log(10) = 10$, so $19 \leq C\cdot(10\cdot\log(10))$ being $C \geq 0$.
- For mergesort we have that the average case is $\Theta(N\cdot\log N)$, $100000*\log100000=1660964$, which is close to our result 1536365.37.

5.5 (Extra) *What other thing instead of removing the tail recursion we could have changed in order to really impact the time complexity of quicksort?*

We could change the position of the pivot. In our environment in which we have to solve a lot of tables we could have a pivot in a random position in each iteration, this will improve the performance. We could also put the pivot in one position depending on the first, last and middle values (that's what we did last year and improved the performance).

6. Final Conclusions.

To conclude this report, we can confirm that we have fully understood the mergesort and quicksort algorithm, and we have analyzed their growth regarding the size of the permutations as well as their time performance. We then could verify that these algorithms have a better performance than the ones used in the previous practice. This is due to the way these algorithms perform, and grow, in general as a logarithmic way.