# Practical Work 2
# Sorting Methods

Name: Matheus Alves Kühl;
Federal University of Minas Gerais
Belo Horizonte-MG-Brazil

makalvesk@ufmg.br

## Introduction

This project was implemented thinking about testing different sorting methods for an array with arbitrarily chosen keys from a seed generated with the srand method. The implementation was all done in c++. The metrics used were: total execution time of the method to complete the sorting, number of copies of the memory register and number of comparisons between keys. The comparison methods tested were usual quicksort with recursion, median quicksort where the chosen pivot is defined as the median between x different elements randomly chosen from the interval to be sorted, quicksort selection where the recursive quicksort is done until subintervals of size are reached smaller than n, then the sorting method by selection, stack quicksort that uses a stack to store the sorting steps that were previously called by recursion, and smart stack quicksort that uses the stack and does sorting from the smallest interval. In the second part of the work I compared the best quicksort method with the mergesort and heapsort methods for the same generated seeds.

## Implementation

The project tree can be seen in figure 1, The main project is within the include and src folders. In the include folder we can observe all the methods and classes used, I chose to make a quicksort class with the metrics and the other quicksort with its different methods inheriting from it but with the same metrics, many functions of the main quicksort method (recursion) are the same for the other methods so I found it useful to make a parent class called quicksort for the other methods. The abstract objects that will be ordered are of the Item Loaded type, which is an Item type with several structures inside it, in this case 15 strings with 200 characters each and 10 double type. In the median quicksort method whose file is quicksort_median.hpp I used the Insertion method that was defined in methods.hpp to order the randomly chosen elements that I will use to take the median. In the quicksort selection method defined in the quicksort_selection.hpp file, I used the selection method as mentioned in the introduction, which was also defined in methods.hpp. In the stack quicksort and smart stack quicksort method, although the average number of recursions does not exceed log(n) in the best case, it can happen that there are up to n-1 recursions in the worst case so I determined the stack size as n-1.

The main /src/main file that calls the other methods has been structured in a way which receives the user parameters in the following format: “metodo” -v “version” -s “seed” -k “k” -m “m” -i “entrada.txt” -o “saida.txt”

Where “method” can be passed as “quicksort”, “mergesort” or “heapsort” (without the quotes), -v is a valid option only for quicksort and defines which of the quicksort methods will be used, 1: for recursive, 2: for median, 3: for selection, 4: for stack, 5: for smart stack. -s sets the seed number, -k is a valid option only for the median and sets the number of elements randomly chosen for the median calculation, -m is a valid option only for the median method and defines the minimum size of the subarray used. -i defines the input file where the sizes of the object vectors to be sorted will be read. -o defines the output file where the metrics of the chosen method will be printed.

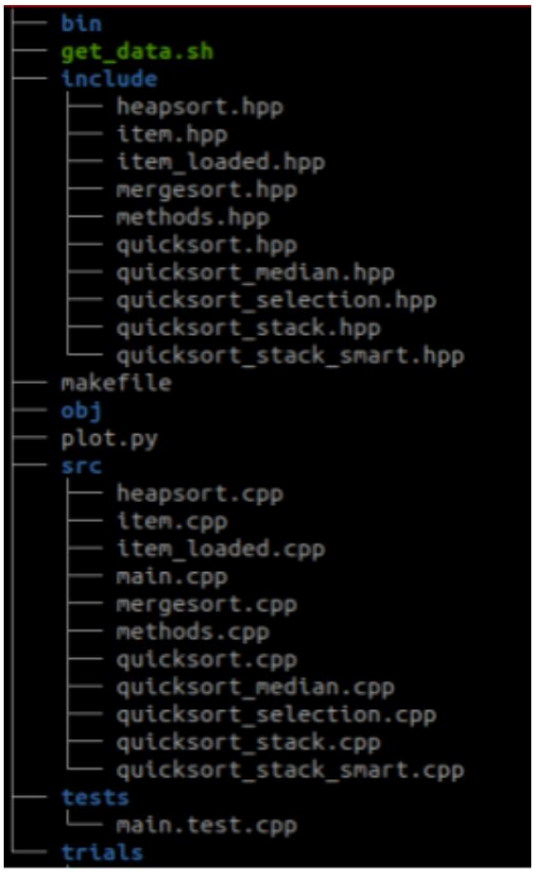


*Figura 1: Árvore do Projeto*

The tests folder was created for carrying out tests while I was making the project, initially there were more tests but I ended up with just one main.test.cpp test where I tested the methods to check if they were ordering the vectors.

The trials folder holds some tests I did for the seeds I used to analyze the methods. The makefile was made in such a way that the main and test binary files are in the /bin folder, and the object type files in the /obj folder.

The get_data.sh file is a bash script used to generate metrics from an arbitrary amount of methods defined in the script. The plot.py file was a python script used to generate the averages and graphs that will be presented in this documentation. From the tested seeds.

## output format

The output format is of the form:

“version,k,m,size,comps,swaps,stime,utime,totaltime” where “version” corresponds to the method, being 1 for recursive quicksort, 2 for median quicksort, 3 for selection quicksort, 4 for stack quicksort, 5 for smart stack quicksort, 6 for mergesort and 7 for heapsort. “k” and “m” is initially defined as 0 and if the chosen method has “k” and “m” they are changed with their respective values. “comps” is the number of key comparisons, “swaps” the number of copies made in memory, “stime” the time spent by the system while executing the method in microseconds,

“utime” the time used by the user in processing the method in microseconds and “totaltime” the sum of these time intervals.

.

# Complexity Analysis

Quicksort is a fast sorting algorithm that works by dividing a large array of data into smaller arrays. This implies that each iteration works by splitting the input into two components, sorting them, and recombining them. For large datasets, the technique is highly efficient, as its average and best-case temporal complexity is O(n*log(n)), however the worst case can happen when the chosen pivot is the largest element or smallest element. range, the worst case is of order $O(n^2)$. Its space complexity also makes the method a great choice being of order O(log(n)). A disadvantage can also be the fact that it uses multiple recursions which can be a problem if recursion is not allowed. To remedy these disadvantages, variations of the quicksort method with the same time and space complexity were invented. A disadvantage of all quicksort methods is that they are not stable. The median quicksort method for example can avoid the worst case quicksort that happens in choosing the pivot. The quicksort selection takes advantage of the fact that the selection algorithm has a better performance for arrays of small size.

The quicksort stack method uses stacks to overcome the problem of recursions.

Mergesort and heapsort can be a viable alternative to quicksort that overcomes the worst cases, as the time complexity order of these cases is always on the order of O(n*log(n)) but the space complexity is always on the order of O (n). Both methods are not stable.

# Robustness Strategies

A test was implemented to verify if the algorithms really sorted the vectors, for this I used a simple function in main.test.cpp where it checks starting from the last element if the previous element is bigger, if it is bigger the loop stops and an error is printed stating that the array was not correctly sorted.

In the main program main.cpp, I initialized the variable that stores the arguments with an allocation which initially sets the arguments all to 0, which prevents the program from crashing if a required option is not passed. This way, if no option is passed, the expected value for seed will be 0, as well as for the input and output files. If the input file does not exist, the program will return a error.

# Experimental Analysis

The analyzes were all performed with seeds 3, 14, 15, 92 and 65. The results can be checked in the /trials folder. The averages were made using the plot.py script, as well as the graphs (the explanation of the column names can be found in the part

Output Format. The result of the averages can be seen in the images below, the time was counted in seconds with microsecond precision. The graphs are similarly trended in all of O(n*log(n)) and can be found in Appendix B.
One fact about the memory analysis is that as each string is 200 bytes if accounting for each character to be 1 byte, and each double to be 8 bytes the cases with array size one million over 3 gigabytes of data were manipulated.

Quicksort:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 0 | 1.0 | 0.0 | 0.0 | 1000.0 | 12476.8 | 8117.4 | 0.000390 | 0.040419 | 0.040809 |
| 1 | 1.0 | 0.0 | 0.0 | 5000.0 | 80704.8 | 48211.2 | 0.000567 | 0.198050 | 0.198617 |
| 2 | 1.0 | 0.0 | 0.0 | 10000.0 | 173408.0 | 103896.6 | 0.000040 | 0.391149 | 0.391189 |
| 3 | 1.0 | 0.0 | 0.0 | 50000.0 | 1050706.2 | 597637.8 | 0.000816 | 2.250958 | 2.251772 |
| 4 | 1.0 | 0.0 | 0.0 | 100000.0 | 2288447.4 | 1262196.6 | 0.000082 | 4.534986 | 4.535066 |
| 5 | 1.0 | 0.0 | 0.0 | 500000.0 | 12561216.6 | 7147505.4 | 0.000876 | 25.516360 | 25.517240 |
| 6 | 1.0 | 0.0 | 0.0 | 1000000.0 | 26671432.0 | 14990055.6 | 0.011327 | 53.413440 | 53.424780 |

Quicksort Mediana:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 7 | 2.0 | 3.0 | 0.0 | 1000.0 | 13239.0 | 8139.0 | 0.000820 | 0.039048 | 0.039868 |
| 8 | 2.0 | 3.0 | 0.0 | 5000.0 | 83099.2 | 48595.2 | 0.000132 | 0.184089 | 0.184221 |
| 9 | 2.0 | 3.0 | 0.0 | 10000.0 | 177238.6 | 104208.6 | 0.000044 | 0.381114 | 0.381157 |
| 10 | 2.0 | 3.0 | 0.0 | 50000.0 | 1070917.6 | 600326.4 | 0.000092 | 2.210324 | 2.210418 |
| 11 | 2.0 | 3.0 | 0.0 | 100000.0 | 2237399.8 | 1271386.8 | 0.000017 | 4.602932 | 4.602950 |
| 12 | 2.0 | 3.0 | 0.0 | 500000.0 | 12774537.8 | 7185460.2 | 0.000029 | 25.787440 | 25.787480 |
| 13 | 2.0 | 3.0 | 0.0 | 1000000.0 | 27244754.8 | 14999562.6 | 0.000822 | 53.611280 | 53.612100 |
| | version | k | m | size | comps | swaps | stime | utime | totaltime |
| 14 | 2.0 | 5.0 | 0.0 | 1000.0 | 13361.6 | 8092.8 | 0.000274 | 0.039857 | 0.040131 |
| 15 | 2.0 | 5.0 | 0.0 | 5000.0 | 81551.8 | 48646.8 | 0.000203 | 0.182547 | 0.182750 |
| 16 | 2.0 | 5.0 | 0.0 | 10000.0 | 178589.4 | 103724.4 | 0.000049 | 0.384178 | 0.384227 |
| 17 | 2.0 | 5.0 | 0.0 | 50000.0 | 1095109.2 | 597482.4 | 0.000104 | 2.207626 | 2.207730 |
| 18 | 2.0 | 5.0 | 0.0 | 100000.0 | 2270378.4 | 1271407.8 | 0.000054 | 4.665268 | 4.665320 |
| 19 | 2.0 | 5.0 | 0.0 | 500000.0 | 13020853.6 | 7134169.2 | 0.000100 | 25.846900 | 25.847000 |
| 20 | 2.0 | 5.0 | 0.0 | 1000000.0 | 27026069.2 | 15020543.4 | 0.000740 | 53.878740 | 53.879480 |
| | version | k | m | size | comps | swaps | stime | utime | totaltime |
| 21 | 2.0 | 7.0 | 0.0 | 1000.0 | 13573.8 | 8111.4 | 0.001001 | 0.042541 | 0.043542 |
| 22 | 2.0 | 7.0 | 0.0 | 5000.0 | 82985.8 | 48494.4 | 0.000157 | 0.181136 | 0.181293 |
| 23 | 2.0 | 7.0 | 0.0 | 10000.0 | 176779.2 | 104106.6 | 0.000108 | 0.383287 | 0.383395 |
| 24 | 2.0 | 7.0 | 0.0 | 50000.0 | 1065012.2 | 601287.6 | 0.000088 | 2.215208 | 2.215298 |
| 25 | 2.0 | 7.0 | 0.0 | 100000.0 | 2235103.4 | 1272348.6 | 0.000087 | 4.622608 | 4.622694 |
| 26 | 2.0 | 7.0 | 0.0 | 500000.0 | 12879252.0 | 7165393.8 | 0.000116 | 25.887720 | 25.887840 |
| 27 | 2.0 | 7.0 | 0.0 | 1000000.0 | 26732818.0 | 15039141.0 | 0.000113 | 53.909900 | 53.910040 |

Quicksort Seleção:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 28 | 3.0 | 0.0 | 10.0 | 1000.0 | 9991.2 | 7654.2 | 0.000970 | 0.030967 | 0.031937 |
| 29 | 3.0 | 0.0 | 10.0 | 5000.0 | 68350.6 | 46024.2 | 0.000132 | 0.144271 | 0.144404 |
| 30 | 3.0 | 0.0 | 10.0 | 10000.0 | 148680.0 | 99525.0 | 0.000088 | 0.311558 | 0.311646 |
| 31 | 3.0 | 0.0 | 10.0 | 50000.0 | 926548.4 | 575894.4 | 0.000136 | 1.898176 | 1.898314 |
| 32 | 3.0 | 0.0 | 10.0 | 100000.0 | 2040162.6 | 1218820.8 | 0.000128 | 3.871784 | 3.871908 |
| 33 | 3.0 | 0.0 | 10.0 | 500000.0 | 11320406.0 | 6929926.2 | 0.000988 | 22.177700 | 22.178720 |
| 34 | 3.0 | 0.0 | 10.0 | 1000000.0 | 24189011.0 | 14554344.6 | 0.000088 | 46.814780 | 46.814860 |
| | version | k | m | size | comps | swaps | stime | utime | totaltime |
| 35 | 3.0 | 0.0 | 100.0 | 1000.0 | 7635.0 | 5761.8 | 0.000479 | 0.025585 | 0.026064 |
| 36 | 3.0 | 0.0 | 100.0 | 5000.0 | 56041.2 | 36760.8 | 0.000248 | 0.111472 | 0.111720 |
| 37 | 3.0 | 0.0 | 100.0 | 10000.0 | 124181.6 | 80798.4 | 0.000081 | 0.244582 | 0.244662 |
| 38 | 3.0 | 0.0 | 100.0 | 50000.0 | 804530.0 | 482598.0 | 0.000056 | 1.517604 | 1.517660 |
| 39 | 3.0 | 0.0 | 100.0 | 100000.0 | 1795600.6 | 1031021.4 | 0.000074 | 3.223418 | 3.223494 |
| 40 | 3.0 | 0.0 | 100.0 | 500000.0 | 10097269.4 | 5993227.2 | 0.000828 | 18.754040 | 18.754860 |
| 41 | 3.0 | 0.0 | 100.0 | 1000000.0 | 21745227.4 | 12682397.4 | 0.000083 | 40.300500 | 40.300580 |

Quicksort Pilha:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 42 | 4.0 | 0.0 | 0.0 | 1000.0 | 12476.8 | 8117.4 | 0.000000 | 0.038021 | 0.038021 |
| 43 | 4.0 | 0.0 | 0.0 | 5000.0 | 80704.8 | 48211.2 | 0.000070 | 0.176590 | 0.176661 |
| 44 | 4.0 | 0.0 | 0.0 | 10000.0 | 173408.0 | 103896.6 | 0.000072 | 0.378835 | 0.378907 |
| 45 | 4.0 | 0.0 | 0.0 | 50000.0 | 1050706.2 | 597637.8 | 0.000036 | 2.185516 | 2.185552 |
| 46 | 4.0 | 0.0 | 0.0 | 100000.0 | 2288447.4 | 1262196.6 | 0.000020 | 4.533796 | 4.533818 |
| 47 | 4.0 | 0.0 | 0.0 | 500000.0 | 12561216.6 | 7147505.4 | 0.000002 | 25.493060 | 25.493060 |
| 48 | 4.0 | 0.0 | 0.0 | 1000000.0 | 26671432.0 | 14990055.6 | 0.000853 | 53.896640 | 53.897500 |

Quicksort Pilha Inteligente:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 49 | 5.0 | 0.0 | 0.0 | 1000.0 | 12476.8 | 8117.4 | 0.000898 | 0.043181 | 0.044080 |
| 50 | 5.0 | 0.0 | 0.0 | 5000.0 | 80704.8 | 48211.2 | 0.000121 | 0.202653 | 0.202774 |
| 51 | 5.0 | 0.0 | 0.0 | 10000.0 | 173408.0 | 103896.6 | 0.000047 | 0.378583 | 0.378630 |
| 52 | 5.0 | 0.0 | 0.0 | 50000.0 | 1050706.2 | 597637.8 | 0.000111 | 2.192114 | 2.192228 |
| 53 | 5.0 | 0.0 | 0.0 | 100000.0 | 2288447.4 | 1262196.6 | 0.000019 | 4.539186 | 4.539206 |
| 54 | 5.0 | 0.0 | 0.0 | 500000.0 | 12561216.6 | 7147505.4 | 0.000804 | 25.547880 | 25.548700 |
| 55 | 5.0 | 0.0 | 0.0 | 1000000.0 | 26671432.0 | 14990055.6 | 0.000038 | 53.736780 | 53.736820 |

Mergesort:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 63 | 6.0 | 0.0 | 0.0 | 1000.0 | 8695.2 | 19952.0 | 0.009099 | 0.103186 | 0.112285 |
| 64 | 6.0 | 0.0 | 0.0 | 5000.0 | 55238.8 | 123616.0 | 0.013182 | 0.438615 | 0.451796 |
| 65 | 6.0 | 0.0 | 0.0 | 10000.0 | 120409.6 | 267232.0 | 0.019328 | 0.984998 | 1.004327 |
| 66 | 6.0 | 0.0 | 0.0 | 50000.0 | 718181.0 | 1568928.0 | 0.143281 | 5.655902 | 5.799184 |
| 67 | 6.0 | 0.0 | 0.0 | 100000.0 | 1536377.6 | 3337856.0 | 0.404727 | 11.811500 | 12.216220 |
| 68 | 6.0 | 0.0 | 0.0 | 500000.0 | 8837344.4 | 18951424.0 | 3.481826 | 68.209920 | 71.691740 |
| 69 | 6.0 | 0.0 | 0.0 | 1000000.0 | 18674322.2 | 39902848.0 | 10.791500 | 141.494200 | 152.285800 |

Heapsort:

| | version | k | m | size | comps | swaps | stime | utime | totaltime |
|---|---|---|---|---|---|---|---|---|---|
| 56 | 7.0 | 0.0 | 0.0 | 1000.0 | 5434.2 | 15073.4 | 0.000488 | 0.080270 | 0.080758 |
| 57 | 7.0 | 0.0 | 0.0 | 5000.0 | 32705.2 | 87157.0 | 0.000367 | 0.482482 | 0.482849 |
| 58 | 7.0 | 0.0 | 0.0 | 10000.0 | 70531.6 | 184193.0 | 0.000775 | 1.032454 | 1.033227 |
| 59 | 7.0 | 0.0 | 0.0 | 50000.0 | 410724.2 | 1037378.2 | 0.002101 | 3.902672 | 3.904776 |
| 60 | 7.0 | 0.0 | 0.0 | 100000.0 | 870936.4 | 2174867.4 | 0.000350 | 7.380128 | 7.380474 |
| 61 | 7.0 | 0.0 | 0.0 | 500000.0 | 4950826.4 | 12024033.6 | 0.003027 | 41.132040 | 41.135040 |
| 62 | 7.0 | 0.0 | 0.0 | 1000000.0 | 10400792.0 | 25048457.4 | 0.001888 | 86.627020 | 86.628900 |

According to the averages taken from the chosen seeds, we can observe that the quicksort selection method with m = 100 has the best performance of all in all aspects, including in comparison with the heapsort and mergesort method.

# Conclusion

From the experimental analysis carried out, I can attest that the quicksort selection method with m = 100 is the one that had the best performance, although in order to carry out a good enough analysis it would be necessary to calculate the deviation from the mean of each of the seeds tested to also verify the consistency of the results when varying the seeds. That said the method seems to best perform sorting for an arbitrary seed value. We can also verify by analyzing the methods that the greater the number of

comparisons and copies in memory the time complexity grows with these values.

## Bibliography

CORMEN, Thomas. Introduction to Algorithms. 4th Edition, MIT Press, ISBN-10:9780262033848, 223-339, 2022.

# Appendix A

## How to Run this program?

The main program can be run with the make or make main command from the main directory where the makefile is installed. It is necessary to have the standard c++ 11 compiler installed (g++), the program was only tested in a linux Ubuntu 21 environment.

## How to run the tests?

It can be run with the make test command from the main directory where the makefile is installed.

## How to test multiple seeds at the same time and multiple methods?

Modify get_data.sh to your liking. Instructions are given inside the file.

# Appendix B

Charts of tested methods: