MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova Faculty of Computers, Informatics and Microelectronics Department of Software and Automation Engineering

Postoronca Dumitru FAF-233

Report

Laboratory Work №2

of AA

Checked by:
Fistic Cristofor, *university assistant*
DISA, FCIM, UTM

Chișinău – 2024

Conditions of the Task

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, one of your choice (Counting Sort)

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Input data

To analyze and compare the sorting algorithms, we define the following input data conditions:

1. Size of the Input (n):
    - Small datasets ($n = 100$)
    - Large datasets ($n = 1{,}000{,}000$)
2. Order of the input Elements:
    - Random ordered elements
    - Elements are already sorted in **ascending order**
    - Elements are already sorted in **descending order**
3. Range of Values:
    - Small Range [1, 100]
    - Large Range [1, 1 000 000]
4. Edge Cases:
    - All elements identical

Metrics in Algorithm analysis

Disabling Turbofan

In order to perform the algorithm analysis I've written Javascript programs that would track the execution time of each algorithm and run them in Node js. Node js uses V8 engine in order to execute the JavaScript programs. This engine consists of 2 components:

- Interpreting machine Ignition[1]
- Optimizing compiler **Turbofan**, that can optimize and translate some of the JavaScript code into the machine code.

To reduce the influence of Turbofan on the validity of data, I have run the algorithms with *–no-opt* flag to disable Turbofan. This decision is motivated by the unpredictability of the optimizing compiler which can optimize the code for an algorithm with a long execution time and not optimize the efficient algorithm. Thus I put all algorithms to equal execution conditions.

Algorithms Implementation

All the algorithms can be found in the GitHub[2] repository of this laboratory work. Each of them corresponds to the most popular implementations in JavaScript that are designed to reduce the influence on call stack and memory

Results

After running of all the tests these are the results obtained:

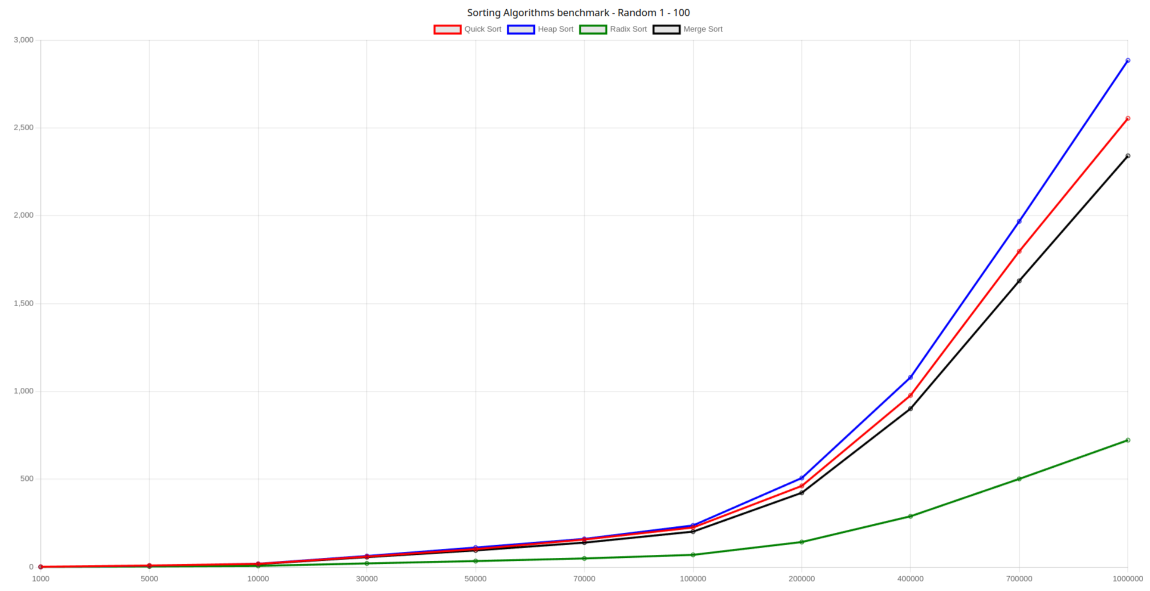


Figure 1: Random numbers from range 1 - 100

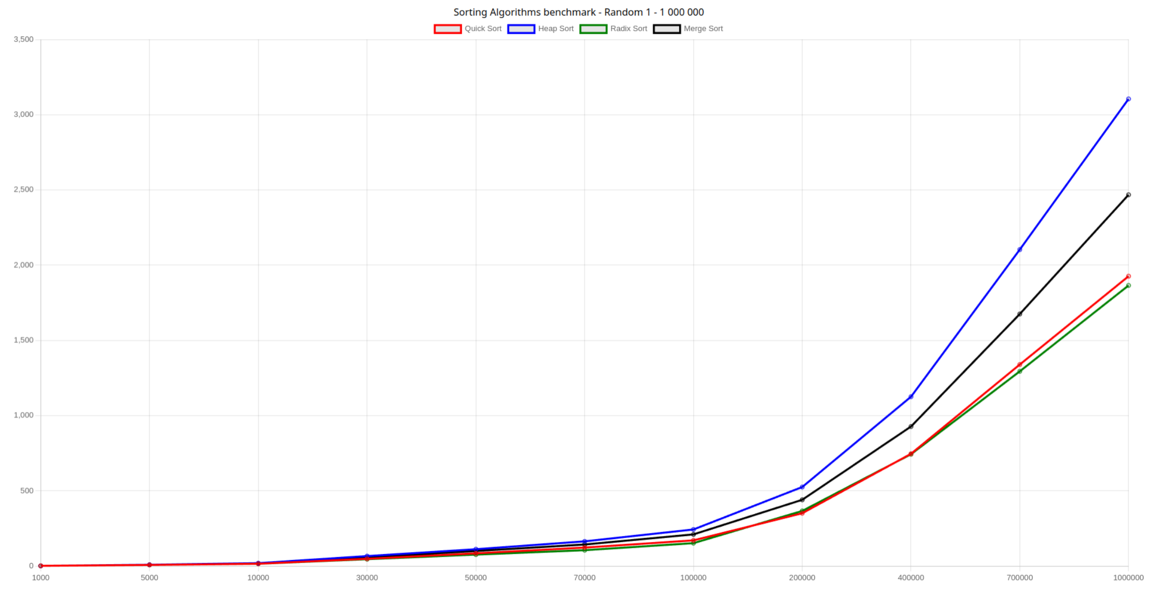


Figure 2: Random numbers from range 1 - 1 000 000

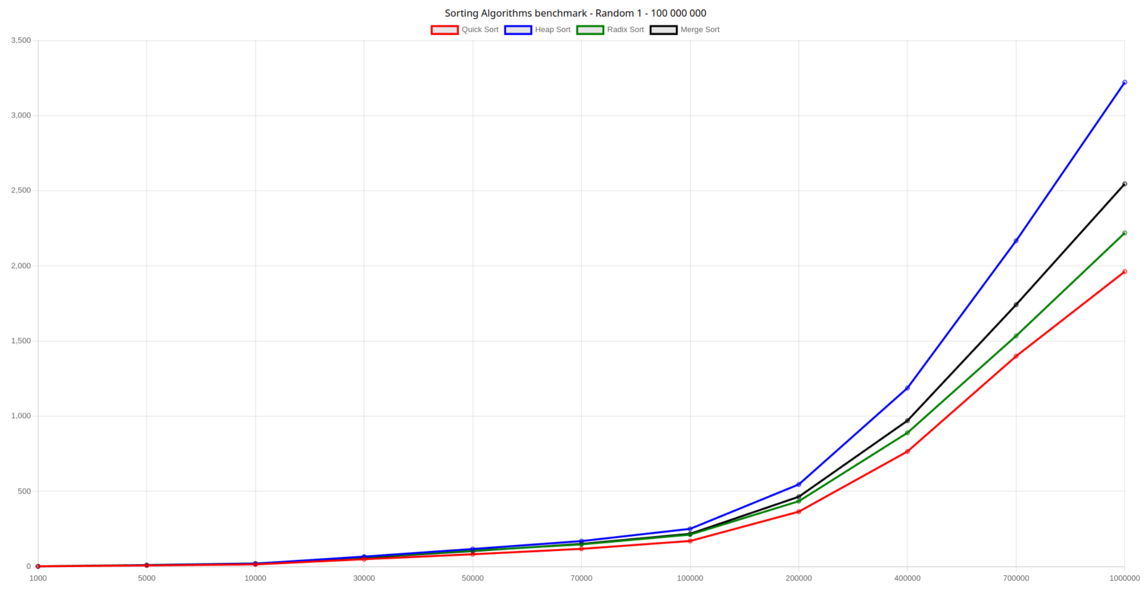


Figure 3: Random numbers from range 1 - 100 000 000

We can easily observe that even if sorting algorithms are performing nearly the same as expected, ***Radix sort*** is vastly influenced by the number of digits implied in the input data. You can see the differences in figure 1 and figure 3. Even if the length of arrays is still the same, Radix sort has to perform editional placements of elements for each aditional digit that the largest number has. This is why it is important to mention that even if performance of Radix sort can be low, it's time execution is highly influenced by the number of digits of the largest number and complexity of the Radix sort is always mentioned as ***O(n*d)***, not O(n)

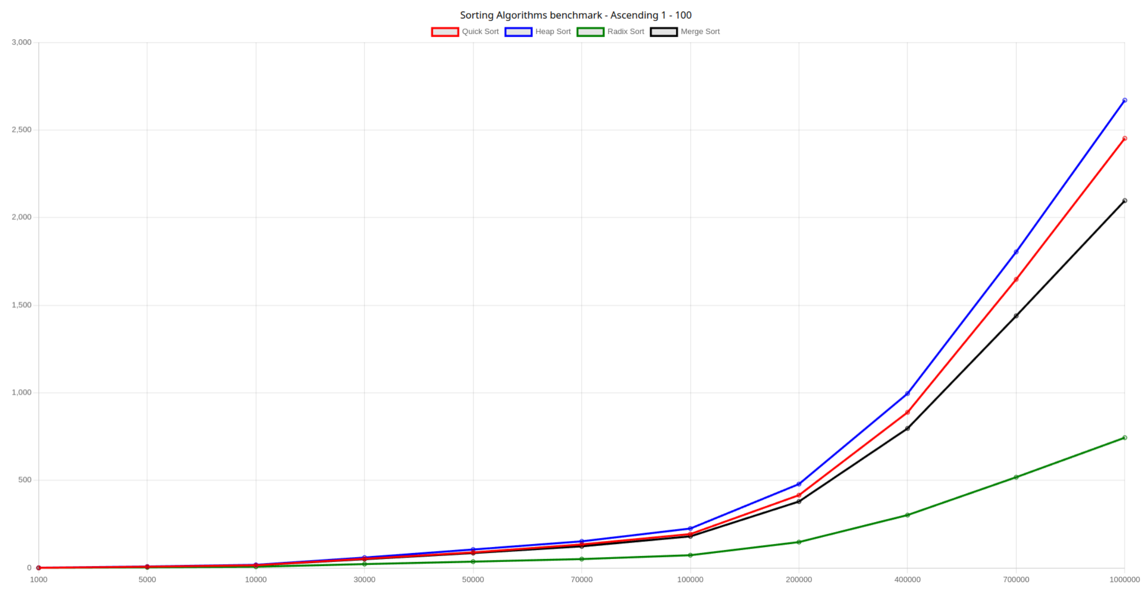


Figure 4: Numbers in ascending order in range 1 - 100

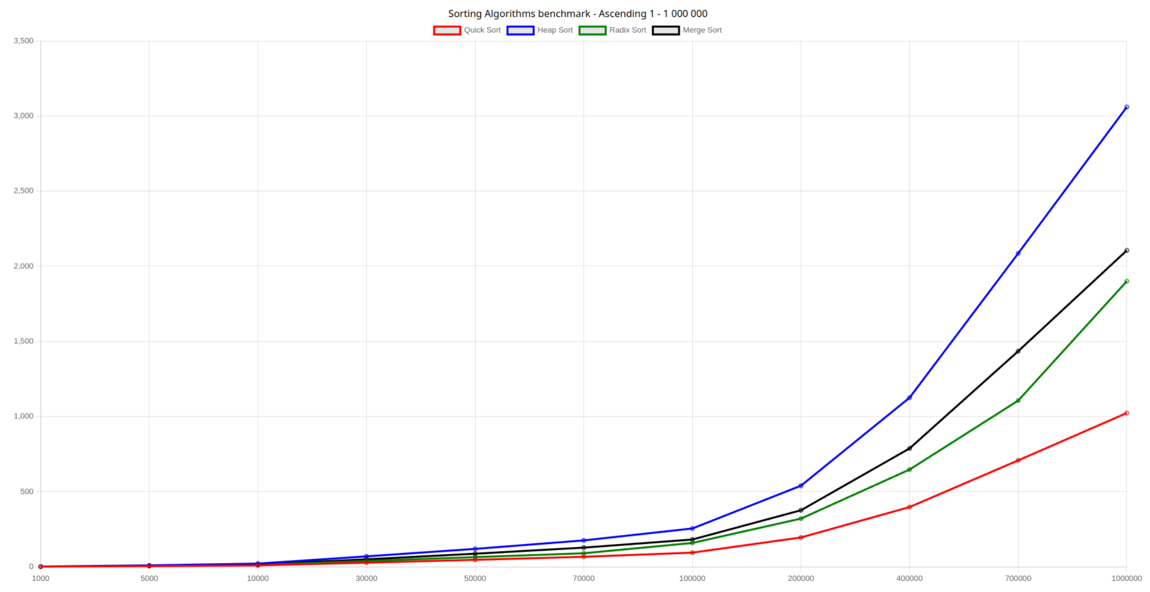


Figure 5: Numbers in ascending order in range 1 - 100 000 000

Besides the slow down of Radix sort already described above we can see that **Quick Sort** is already loosing his positions to Merge Sort (figure 5 and only there). This happens because of logic behind the Quick sort. To perform the sorting it uses a *pivot*. It is a flag that is used to arrange the elements smaller than pivot to the left side of the partition and the greater numbers to the right side of the partition. In best case, the Time Complexity of Quick sort is ***O(n*logN)***. This scenario can occur if everytime the chosen pivot is the element that should be placed in the middle of the partition. BUT. **If on each iteration the chosen pivot leads to biggest possible number of comparisons, then the Time Complexity of Quick Sort is** $O(n^2)$. In this case smaller range meant a big number of redundant comparisons between same elements. This issue is not so evident in figure 5

It will perform the comparison on all the elements of the array with no proper partitioning, because elements are repeating themself. Same effect we can see on next diagrams.

Also we can see a speed improovement for over sorting algorithms.

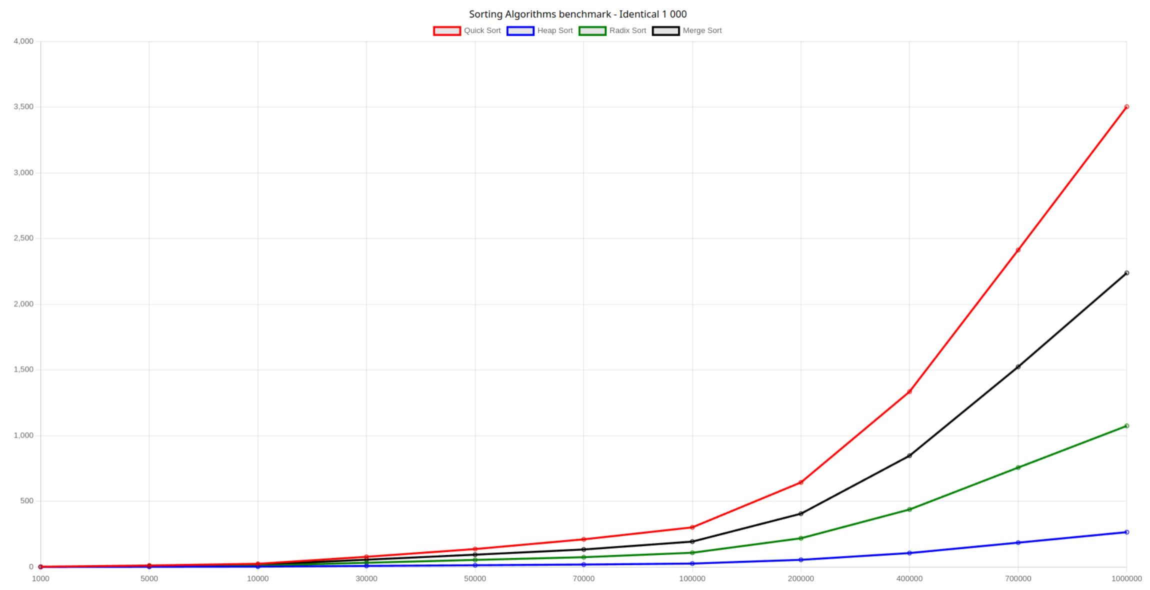


Figure 6: Array with the same number 1000

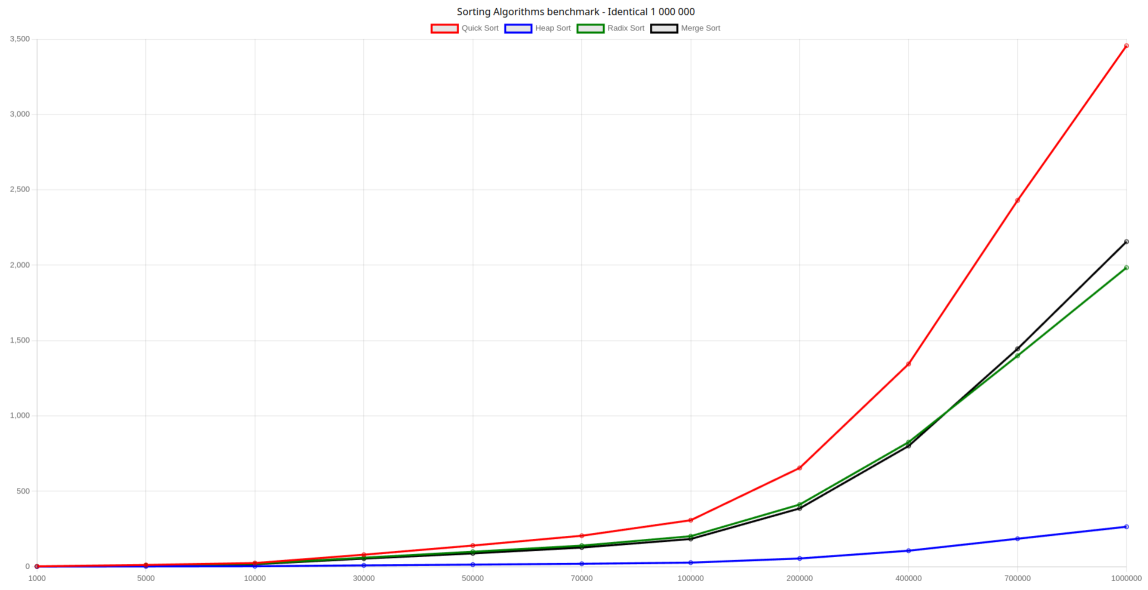


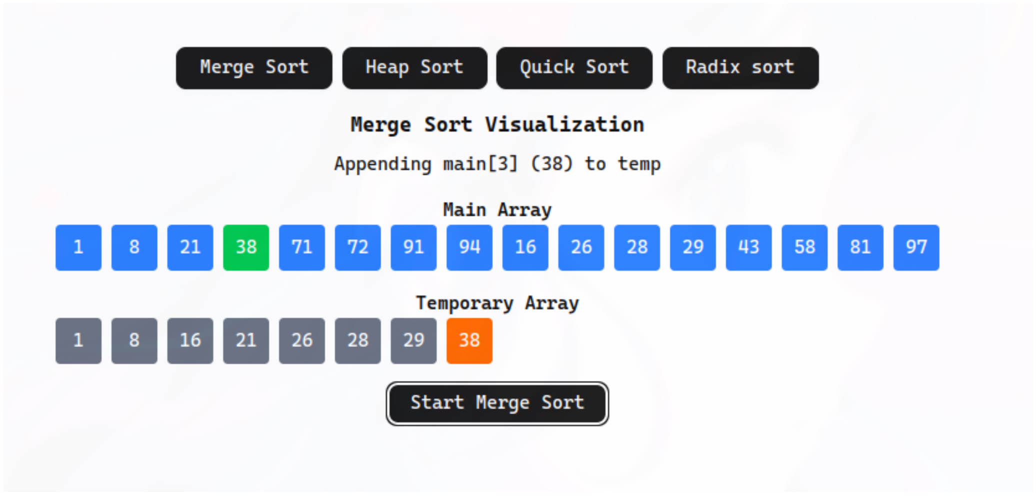
Figure 7: Array with the same number 1 000 000

Identical numbers are drastically disadvantaging Quick sort, because it has to do the comparisons literarly for all of the elements in current partition and reach its highest time complexity.

On the other hand Heap sort is the fastest in this case. This happens because the hepify function does not have to swap any of the elements in the heap and can directly return the same tree as the result. Only thing that it needs to do is to pass each node and just check it.

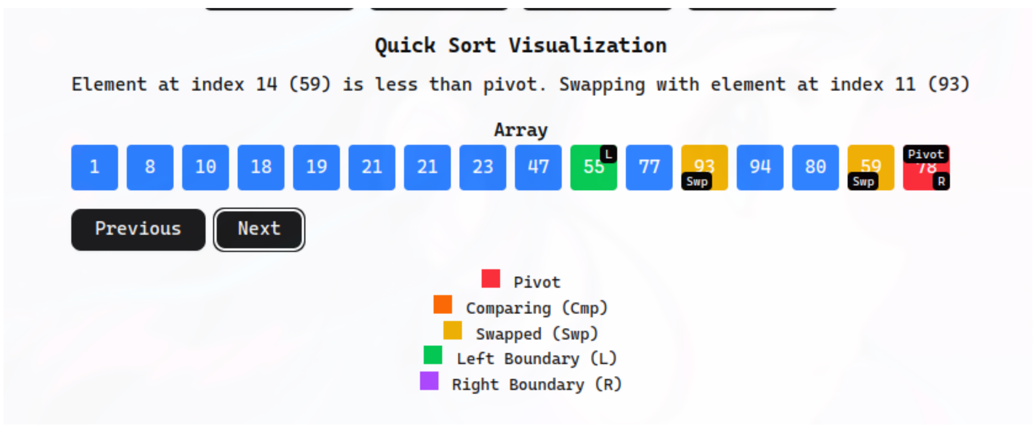
Graphical Representation

Graphical representation was done in React, deployed on Vercel and can be accessed online by link [3]. I decided to provide a more practical representation of the sorting algorithms rather than moving bars on the screen. In my subjective opinion it is not a good way to understand sorting algorithms.



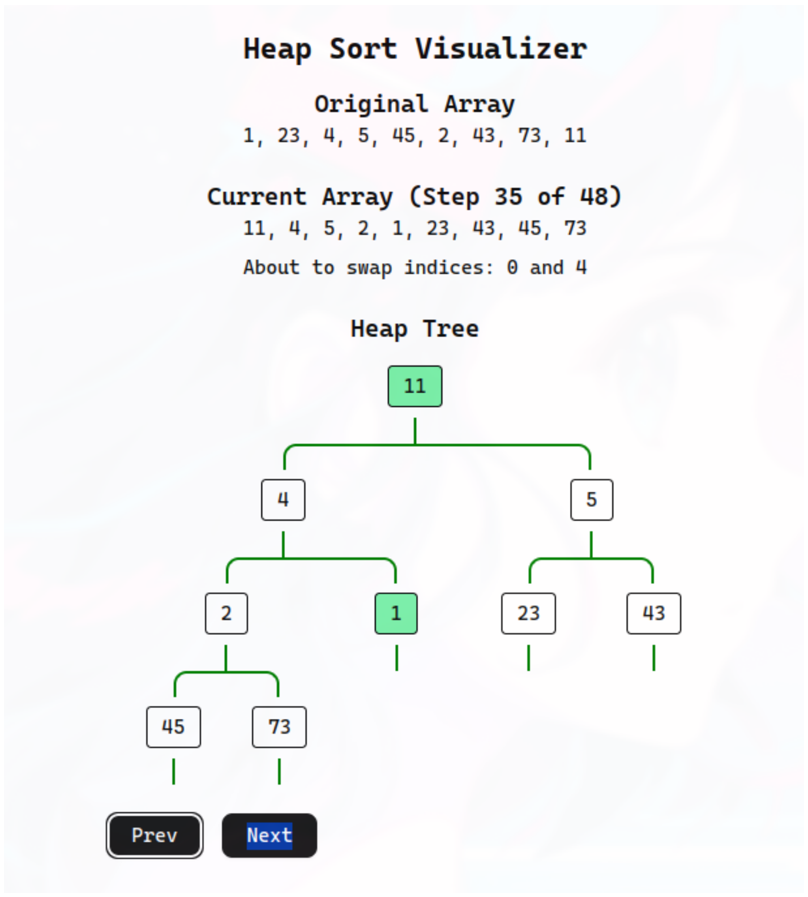
Appending main[3] (38) to temp

Figure 8: Representation of Merge Sort



Element at index 14 (59) is less than pivot. Swapping with element at index 11 (93)

Figure 9: Representation of Quick Sort



Heap Sort Visualizer

Original Array

1, 23, 4, 5, 45, 2, 43, 73, 11

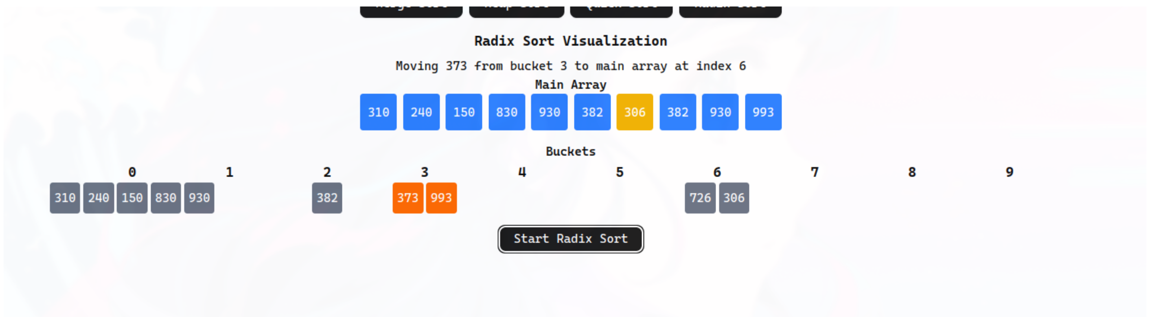
Current Array (Step 35 of 48)

11, 4, 5, 2, 1, 23, 43, 45, 73

About to swap indices: 0 and 4

Heap Tree

Figure 10: Representation of Heap Sort



Radix Sort Visualization

Moving 373 from bucket 3 to main array at index 6

Main Array

Figure 11: Representation of Radix Sort

Conclusions

In conclusion I can say that the most reliable and predictable sorting algorithm is Merging Sort. In each case it performed relatively well, which is a very good result considering that Quick Sort(considered the fastest sorting algorithm) is running with complexity near to n^2. If you look at each metric, you can see that the execution time of Merge Sort is constant. This is why it is mostly used as default sorting algorithms.

References

[1] Franceska Hinkelman (2017) - Understanding bytecode *Medium*

[2] GitHub repository of current laboratory work

[3] Graphical representation of the sorting algorithms