An experiment on sorting algorithms

Rareș-Cătălin Trif
Faculty of Mathematics and Informatics,
West University,
Timișoara, Romania
Email: rares.trif03@e-uvt.ro

May 14, 2022

Abstract

This paper is aimed at comparing the time to six studied sorting algorithms: selection sort, merge sort, quick sort, heap sort, radix sort and bucket sort.

These algorithms were written in C++.

The paper contains charts for a better visualisation of the time resulted after running each algorithm. The measure unit in every chart from this paper is milliseconds.

Contents

1 Introduction

Sorting is an ancient problem, where people came with different approaches. Since the apparition of computers, this problem has gained popularity. The algorithms studied in this paper can be categorized as comparison based and non-comparison based algorithms.

The majority of these algorithms compare the values and the rest compare the indices and sort them accordingly.

2 Theoretical Section

This paragraph has the role of giving a quick explanation and a theoretical overview for the studied algorithms.

2.1 Selection Sort

This algorithm sorts in place and it is based on comparisons. First we find the smallest element, switch it with the first element of the list and then we divide the list into two: the sorted part and the unsorted part. We will use a counter for the delimitation between the two lists. Then, we take the next element and compare it with the next one. If it is lower, we switch values, otherwise continue with the comparison. This process is repeated until the list is sorted.

Memory wise, it is good because it does not need an additional list to store values, therefore it will have O(1) space complexity. This means that, when memory is not an impediment, this algorithm is not the best option, but when you are limited, then it will do a good job.

2.2 Merge Sort

This is a divide and conquer algorithm, invented by John von Neumann. The idea behind this algorithm is to divide the unsorted lists into multiple lists, eventually having a list for every element (n lists). Then, we merge each list and interchange the values in the correct order. The process repeats until we remain with the sorted list.

Memory wise, the algorithm uses a lot of resources because, besides the initial array, we need another one for the merging procedure. So, the space complexity will be O(n), which is not bad or good either.

2.3 Quick Sort

Quick sort is a comparison-based algorithm, invented by Antony Hoare. Conceptually, this algorithm works in the following way: we select a 'pivot' element and divide the list into two parts. In the left half we have the elements smaller than the pivot and in the right one the elements greater than the pivot. Then we call the function for the left part and the right part until the array is sorted.

Memory wise, it has O(log(n)) or O(n) because it is a recursive algorithm and every time it is calling itself, a new stack of memory has to be allocated.

2.4 Heap Sort

This is a comparison-based sorting algorithm. Heap sort can be thought of as an improved selection sort: like selection sort, this algorithm divides its input into a sorted and an unsorted region, and it iteratively shrinks the unsorted region by extracting the largest element from it and inserting it into the sorted region. It also maintains the unsorted region in a heap to more quickly find the largest element at each step.

Memory wise, the algorithm is doing good, having O(1) space complexity. In comparison with Merge Sort or Quick Sort, which are known to be within the most efficient sorting algorithms, Heap Sort is better.

2.5 Radix Sort

The idea behind this algorithm is to sort the elements by their radix, from the least significant digit to the most significant digit. For the sorting part, counting sort is used. (or any sorting algorithm)

Memory wise, the algorithm has O(n+k) space complexity because it is bound to what sorting algorithm is using. For example, when using counting sort, we create multiple arrays, one of size N and another one of size K.

2.6 Bucket Sort

The idea of this sorting algorithm is to create 'buckets' to put elements in them. If we have n elements, we create n buckets. Elements are placed in buckets with this formula: n*list[x]. Then we apply insertion sort to sort the elements in each bucket. In the end, we join the buckets in the sorted array.

Memory wise, the algorithm has O(n+k), where n is the number of elements and k the number of buckets. But it gets worse with the increase in the size of the input.

3 Experimental Section

This section will be split into three parts: the first part with the comparison of all algorithms, the second part is a comparison between unoptimized and optimized versions of merge sort and quick sort, and the third part where we will compare the difference between two computer specs.

3.1 Comparison of Sorting Algorithms

The goal of this subparagraph is to see the performance and confirm/infirm the myths regarding certain algorithms.

The comparison is done be using lists of lenght 10.000, 100.000 and 1.000.000 with numbers. There are also three types of list: sorted, reversed sorted and random lists.

The results obtained are:

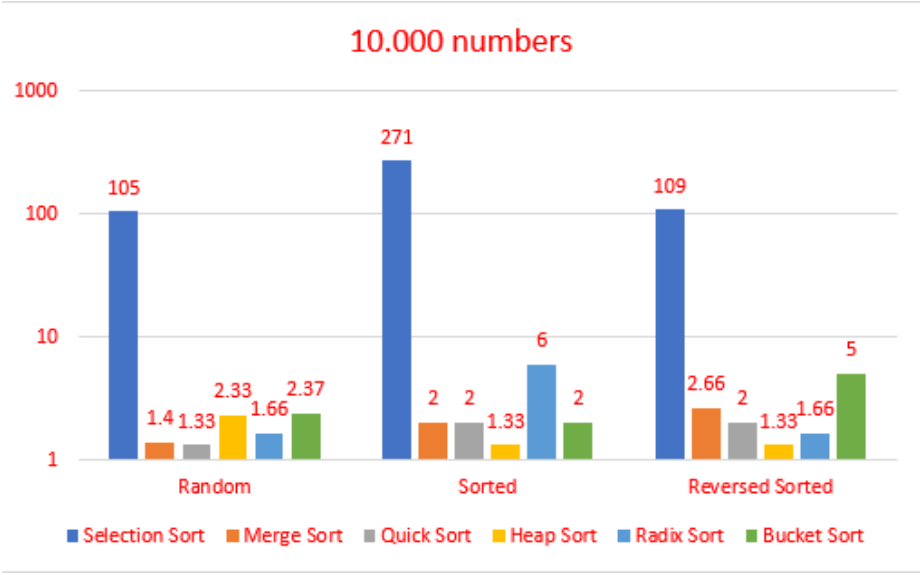

Fig.1

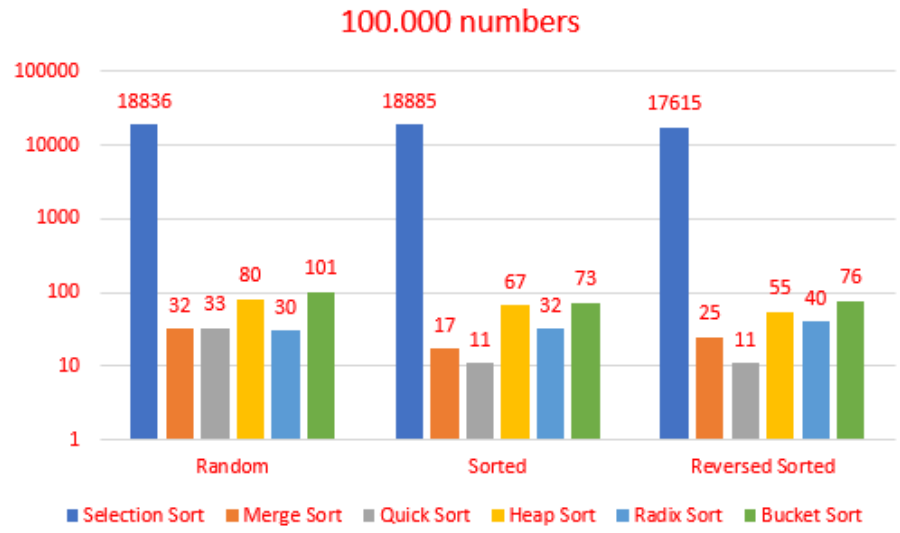

Fig.2

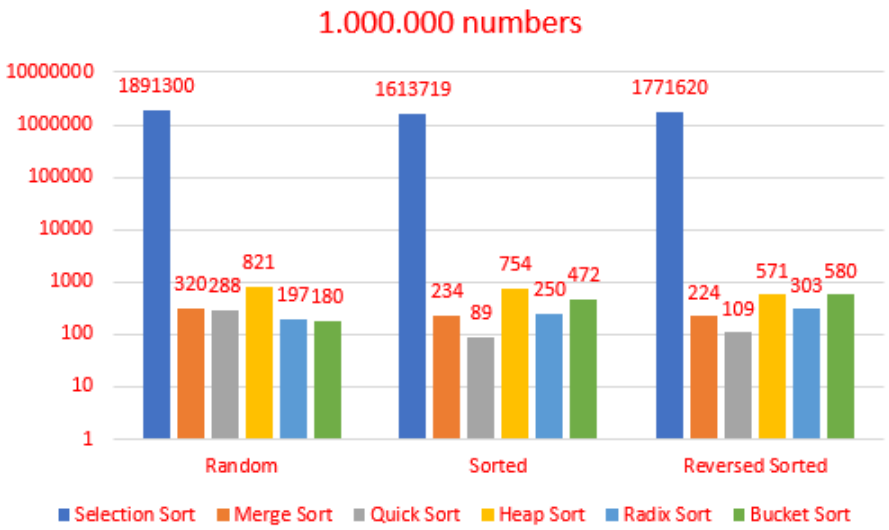

Fig.3

In these tests, quick sort resulted to be the best one. Taking into account that the measure unit is milliseconds, we can say that there is no visible difference between the best algorithms, just that some can be worse with a different implementation, such as merge sort and quick sort, or can be used just for numbers like radix sort.

3.2 Optimized versus Unoptimized

During the first tests, there where used unoptimized versions for merge sort and quick sort. The result were the following: merge sort had poor performance, while quick sort, besides the fact that it was slower than radix sort, it didn't work for list with more than 10.000 numbers. The times presented in the previous paragraph were obtained with the optimized version and below is a comparison between both versions.

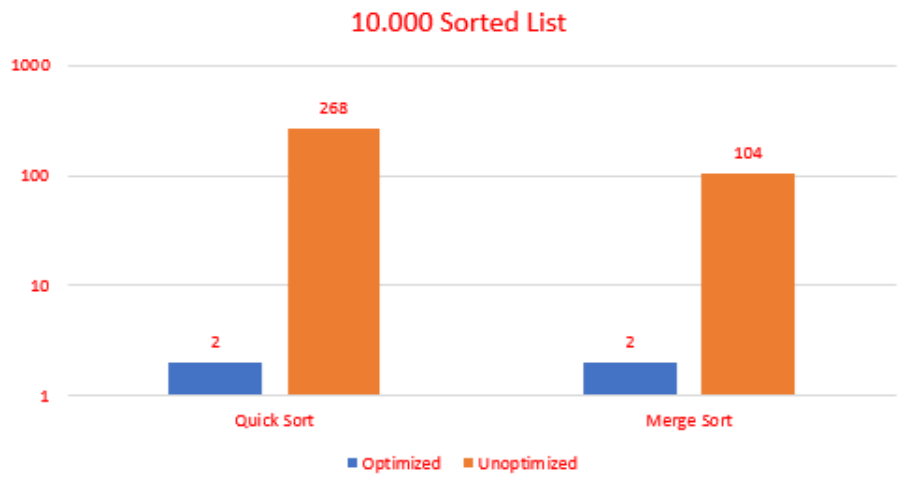

Fig.4

3.3 Comparison of performance between two different computers

This comparison was conducted in collaboration with Claudiu Stefan. In this subparagraph we thought that it would be interesting to test the difference in time between two different computer specs. The time for small numbers it's not that relevant and the difference is hardly seen, so we thought to compare just the 1 million lists.

The first computer has the following configuration (CPU and amount of RAM): Intel(R) Core(TM) i7-8750H CPU @ 2.20GHz, 16 GB RAM.

The second computer has the following configuration (CPU and amount of RAM): Intel(R) Core(TM) i3-3110M CPU @ 2.40GHz, 8 GB RAM.

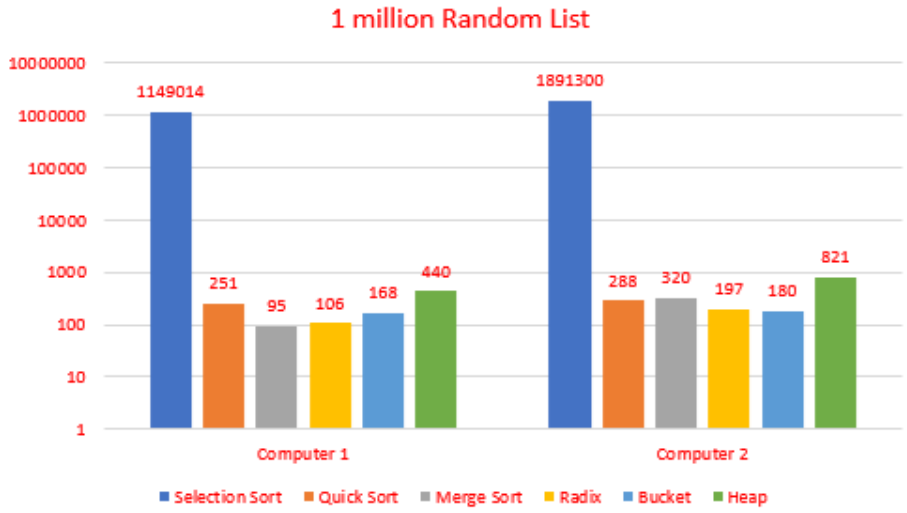
1 million Random List
10000000
1000000
100000
10000
1000
100
10
1
1149014
251
95
106
168
440
1891300
288
320
197
180
821
Computer 1
Computer 2
Selection Sort
Quick Sort
Merge Sort
Radix
Bucket
Heap

Fig.5

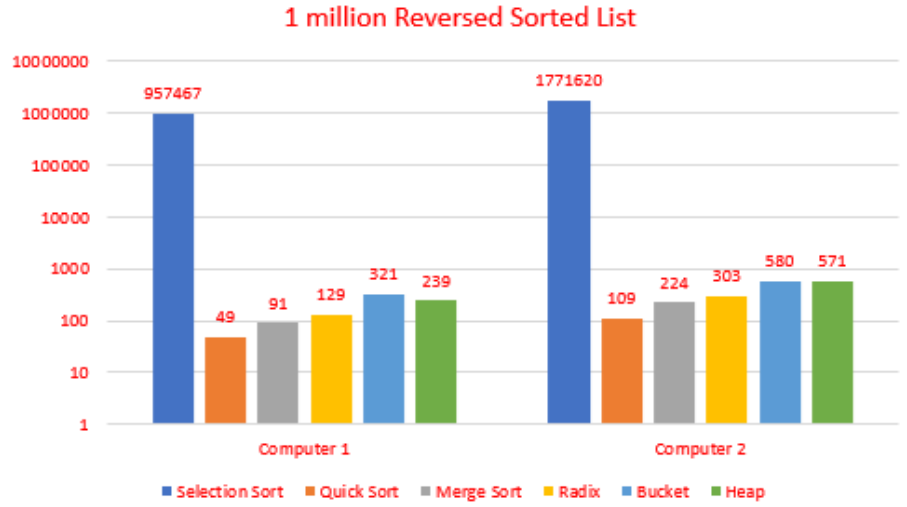
1 million Reversed Sorted List
10000000
1000000
100000
10000
1000
100
10
1
957467
49
91
129
321
239
1771620
109
224
303
580
571
Computer 1
Computer 2
Selection Sort
Quick Sort
Merge Sort
Radix
Bucket
Heap

Fig.6

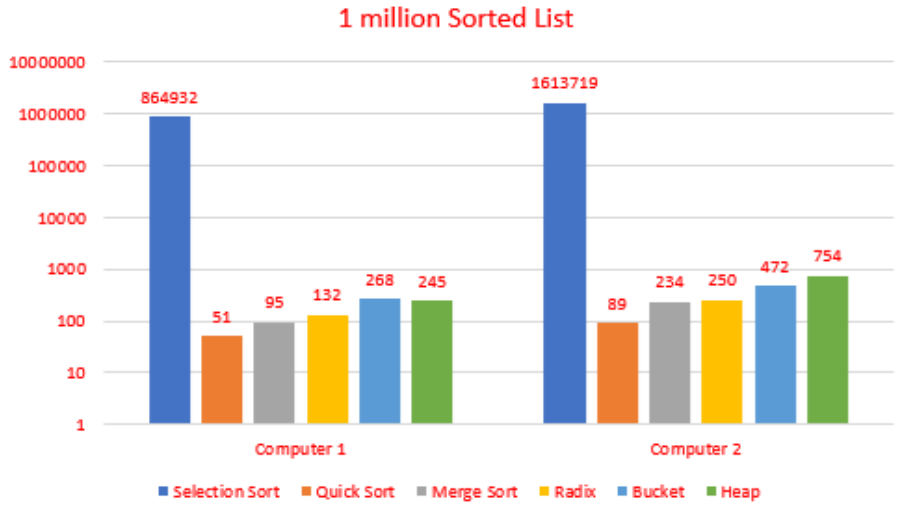

Fig.7

These results were obvious, where the first computer will have better performance than the second one. The difference can easily be observed for selection sort, where the time is almost doubled from one to another.

4 Conclusion

In conclusion, the results were predictable. Even if in the beginning radix sort seemed to be the fastest, after switching to the optimized versions of quick sort and merge sort the expectations came to reality. Still, quick sort is not a stable algorithm and with a bad implementation it will not work on certain lists of numbers. Merge sort on the other hand, will deliver a poor performance. The other algorithms did well as well and since the measure unit was milliseconds, there are no big differences between them. It is up to the user to choose what algorithm he wants to use and for what purpose.

5 Bibliography

Github Project
https://github.com/7fritz/MPI-project/

Thomas H. Cormen, Charles E. Leiseron, Ronald L. Rivest, Clifford Stein. *Introduction to Algorithms, 3rd Edition*

Geegforgeeks Website
`https://www.geeksforgeeks.org/sorting-algorithms/`

Wikipedia
`https://en.wikipedia.org/wiki/Heapsort`

Programiz website
https://www.programiz.com/dsa/sorting-algorithm