Sorting Algorithms Showdown: A Comparative Analysis of Quicksort, Insertion Sort, Heap Sort, and Radix Sort

Mykola Horbenko

18 November 2023

Introduction

In this small research work we are going to compare three sorting algorithms: Insertion sort, Quick sort, Heap sort and Radix sort. The problem to solve is to sort an array of integers of size *n*, size of integers inside is [0;*n*).

Insertion Sort

The Insertion sort, as follows from the name, inserts an element into a sorted sequence. Starting from the second element, the first loop iterates through the array. Every iteration it takes an element, we would call it *e*, and enters the second loop. In the second loop it compares *e* with preceding elements and swaps them if *e* is smaller. The second loop continues until it finds an element which smaller than *e*. Then the first loop takes the next element.

The average-case complexity for Insertion sort is said to be $O(n^2)$.

Heap Sort

The key element of heap sort is building a heap from the given array. Heap itself is a tree data structure, but with one distinct trait. In the heap each parent element is greater or equal to all its children.

To build a heap the algorithm iterates through the given array and for each element heap sort compares it with its parents. If the element is larger, they are swapped. The procedure is done till it gets to the beginning of the array. Then it takes the next element.

When a heap is built, the first element, which has the most significant value, is swapped with the last and said to be out of the array. After that the heap structure may become violated. To fix that, the first element is recursively swapped with the greatest of its children, until it is the biggest one or it has no more children. Then the values at the beginning and at the end are swapped again, and the heap is fixed again, until there are no more members in the heap.

The average-case complexity for Heap Sort is *O (n log(n))*.

Quick Sort

The Quick sorting algorithm works by recursively dividing the array based on a pivot. All values larger are placed to the right and everything smaller – to the left.

Any element can be chosen as a pivot. In our tests, let us choose the first one.

After the pivot is set, two indices are initialized. One at the end of the array, another one on the next value after the pivot. They are moving towards each other and if under the right-one is an element with a value less than pivot and under the left one- greater, elements are swapped. When two indices intersect the pivot is swapped with the last value smaller than it. Now the pivot takes the exact same place as it would take in sorted array.

The described process is repeated recursively for both left-side and right-side arrays.

The average-case complexity for Quick Sort is *O (n log(n)).*

Radix Sort

Radix Sort is a linear-time, non-comparative sorting algorithm that works by distributing elements into buckets according to their individual digits. In our tests it processes the digits from the least significant to the most significant.

Firstly, it determines the maximum number of digits among all the elements in the array, by finding the largest element. This determines the number of passes the algorithm needs to make through the array. Then it creates 10 buckets (0 through 9), one for each digit value. Secondly, it passes through the array for each digit place, starting with the rightmost (least significant) digit and moving towards the leftmost (most significant) digit. In each pass, distribute the elements into the buckets based on the digit at the current place. Finally, after each pass, it collects the elements from the buckets in order (from 0 to 9) and concatenates them back into the array. Described is repeated for each digit place until all digits have been considered.

The average-time complexity for Radix Sort is *O (k * n)*, where k is the number of digits in the greatest element.

Methodology

The size of tested arrays starts from 10 and will be growing by formula *k * n*, where *k* is from range [1;10]. Every time *k* reaches 10, we will multiply *n* by 10. So firstly, we will go 10,20,30…100. Then 100,200,300…1000 and so on until we reach the size of 100000 integers. The sorting procedure is the following: for each size four similar, randomly generated, arrays, of the following size, are sorted by each algorithm. The procedure is repeated 100 times for each size.

Results

On the graph where all algorithms are depicted (Fig. 1) we can clearly see that Insertion sort takes so much time to sort, that other sorts are painted almost in the same line. And it confirms the complexity of Insertion Sort. If we take away the Insertion Sort (Fig. 2), we will see that Heap Sort turned out to perform slower than Radix and Quick Sort. And on the last graph (Fig. 3), we can see that the complexities of Quick Sort and Radix Sort are almost identical.

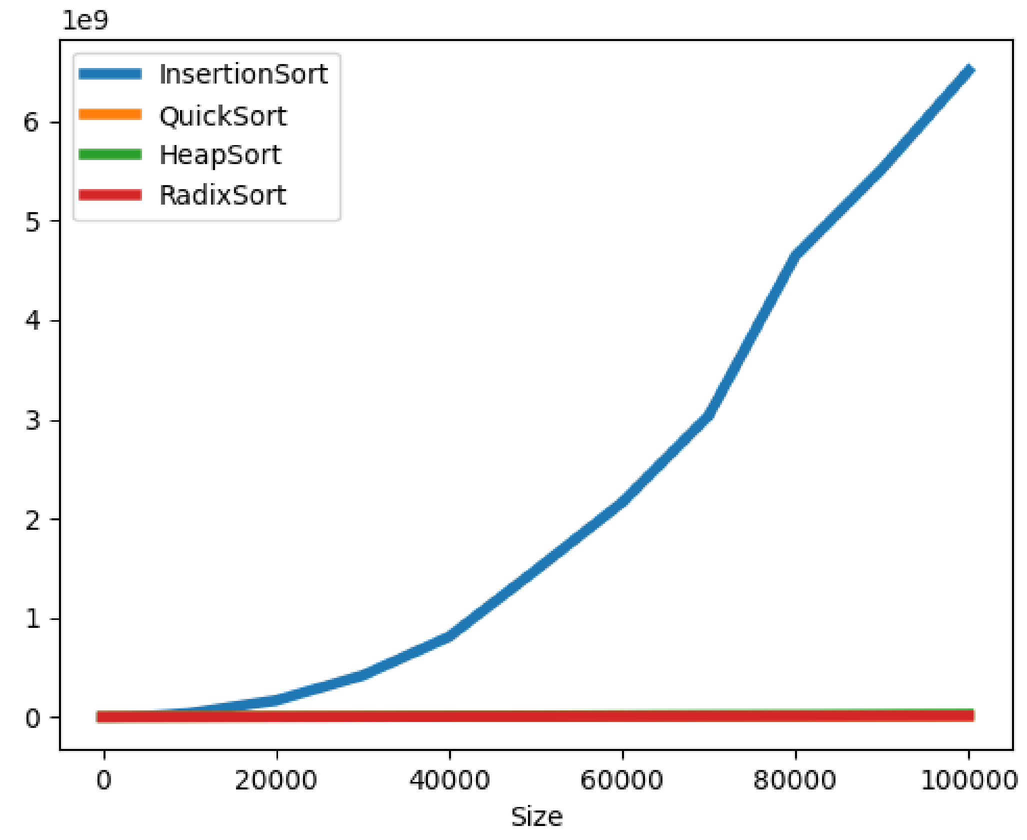

Figure 1, all four sorting algorithms.

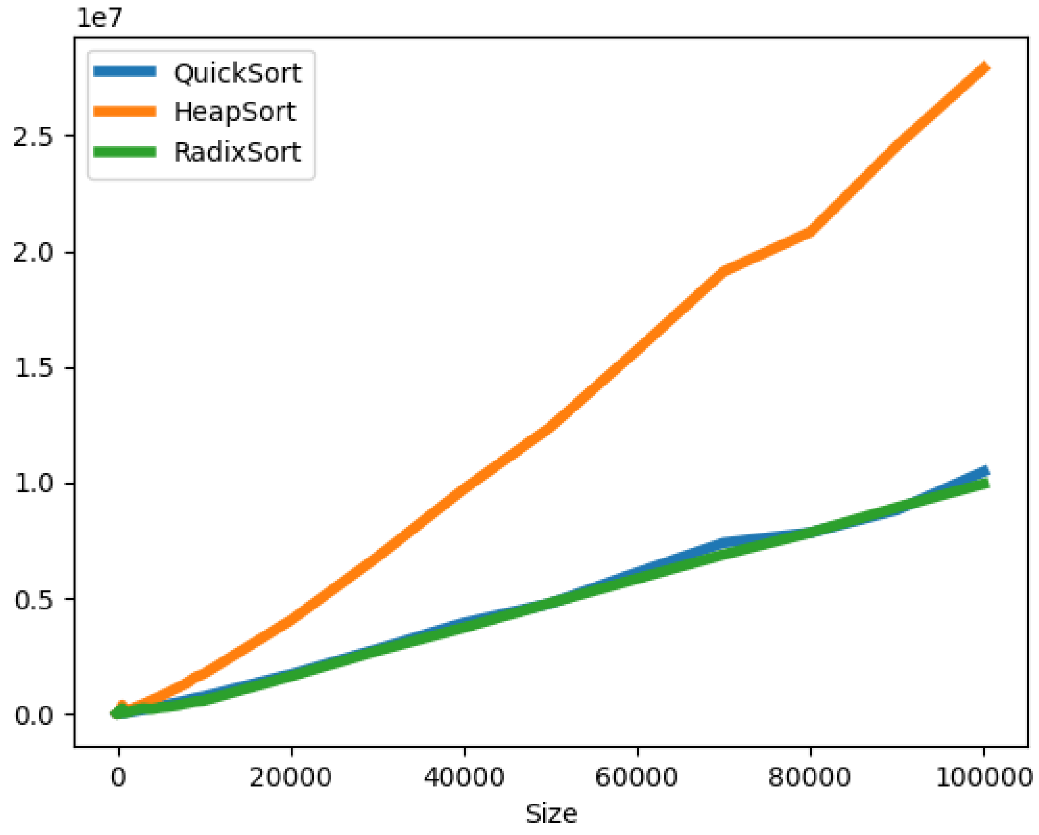

Figure 2, Insertion Sort is removed.

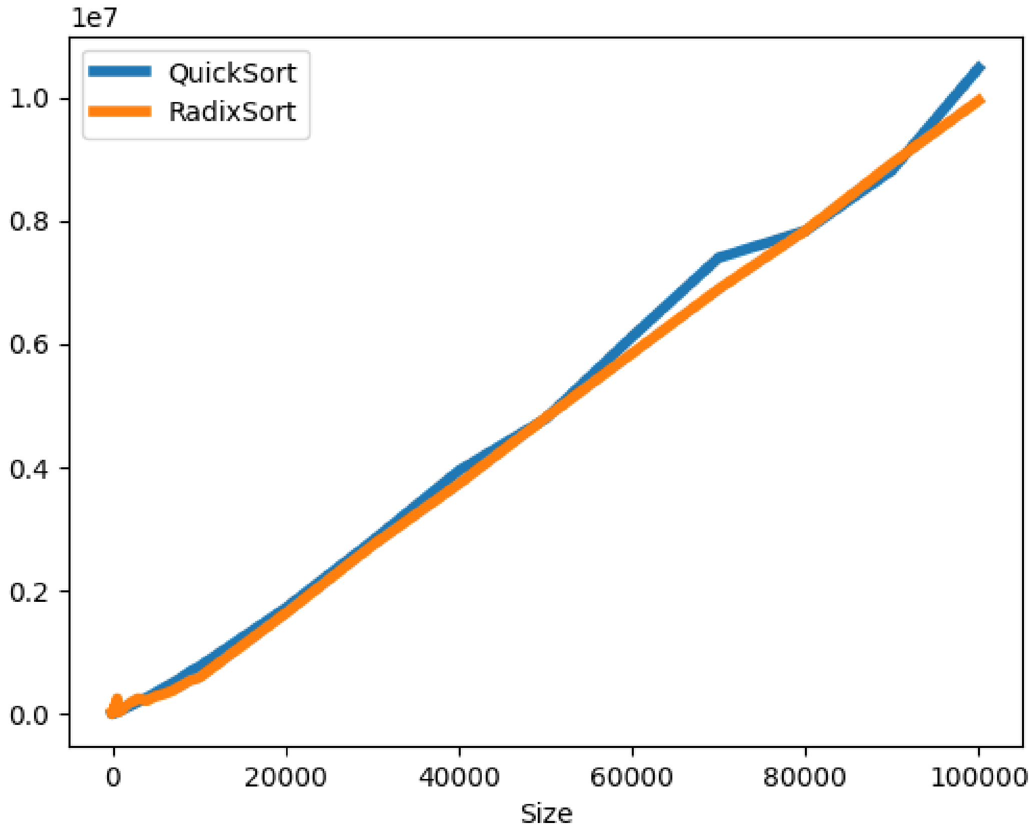

Figure 3, only Radix and Quick Sort.

Conclusion

To conclude, we can say that Quick Sort and Radix Sort have almost identical complexities in case of an random-shuffled array of size *n* with integer elements of size [0;*n*).