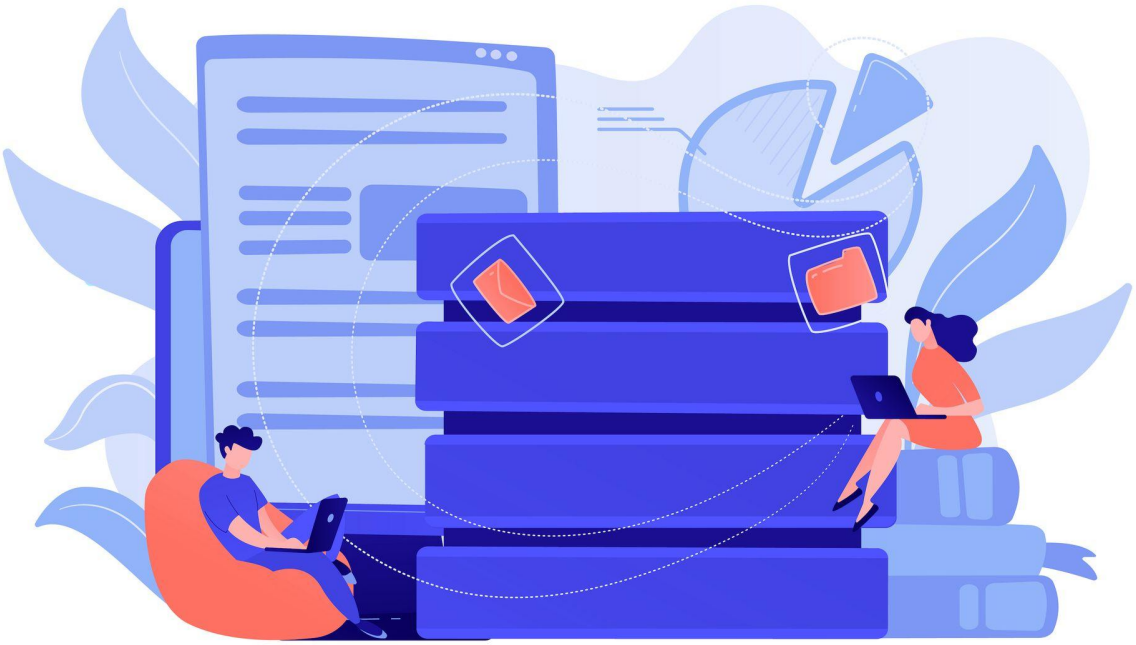


Hybrid Sort

Based on comparison between Insertion, Quick, Heap, Radix sorting algorithms

Nurbek Shukan

20.11.23

Introduction

Sorting algorithms lie at the core of computational efficiency, impacting a myriad of applications. Sorting is needed to speed up the work of programs: when we have a lot of data, we need to somehow organize it so that it is easy to search for and quickly process. On small data sets this is not noticeable, but when you need to work with hundreds of thousands of data units, sorting can either speed up the program or completely crash it. People have invented many sorting algorithms for different data and for different tasks.

In this paper, we present a hybrid sorting algorithm born from the fusion of insertion sort, heap sort, quicksort, and radix sort. Unlike traditional comparison-based methods, our hybrid approach aims to strike a balance between speed and adaptability. The problem is to arrange an array of *n* integers according to some total order which is computed in O(1). We use < as the total order.

1. Insertion Sort

There is a part of the array that is already sorted, and you need to insert the remaining elements of the array into the sorted part, while maintaining the order. To do this, at each step of the algorithm, we select one of the elements of the input data and insert it into the desired position in the already sorted part of the array, until the entire set of input data is sorted.

Since only neighboring elements can change places during the operation of the algorithm, each exchange reduces the number of inversions by one. Therefore, the number of exchanges is equal to the number of inversions in the original array, regardless of the sorting implementation. The maximum number of inversions is contained in an array whose elements are sorted in non-ascending order. The number of inversions in such an array is $n(n-1)/2$

The algorithm runs in $O(n+k)$, where k is the number of exchanges of elements of the input array, equal to the number of inversions. On average and in the worst case - in O(n^2). The minimum estimates occur in the case of an already ordered initial sequence of elements, the worst - when they are arranged in the reverse order.

2. Heap Sort

a sorting algorithm that uses the binary heap data structure. This is a non-robust sorting algorithm with $O(nlogn)$ running time, where n is the number of elements to sort, and uses $O(1)$ additional memory.

It is necessary to sort an array A of size n. Let's build a maximum heap based on this array in $O(n)$. Since the maximum element is at the root, if we swap it with A[n−1], it will fall into place. Next, we call the **siftDown**(0) procedure, having first reduced heapSize by 1. In $O(nlogn)$, it will sift A[0] to the right place and form a new heap (since we have reduced its size, the heap is located from A[0] to A [n−2], and the element A[n−1] is in its place). Let's repeat this procedure for a new heap, only the root will change places not with A[n−1], but with A[n−2]. Doing the same thing until heapSize is 1, we will put the largest remaining number at the end of the unsorted part. Obviously, in this way we will get a sorted array.

3. Quick Sort(Hoare partition)

One of the most famous and widely used sorting algorithms. The average running time is $O(nlogn)$, which is the asymptotically optimal running time for a comparison-based algorithm. Although the worst-case running time of the algorithm for an array of n elements can be $O(n^2)$, in practice this algorithm is one of the fastest.

The quick sort method works on the principle of divide and conquer.

- The array a[l…r] of type T is divided into two (possibly empty) subarrays a[l…q] and a[q+1…r], such that each element a[l…q] is less than or equal to a[q], which in turn does not exceed any element of the subarray a[q+1…r]. The index is calculated during the partitioning procedure.
- The subarrays a[l…q] and a[q+1…r] are sorted using a recursive call to the quicksort procedure.
- Because the subarrays are sorted in place, no action is required to merge them: the entire array a[l...r] is sorted.

The primary step in this sorting algorithm involves a process called partitioning, where we rearrange elements in an array (denoted as a[l...r] of type T) as necessary. Here's how it works: first, we pick an element from the middle of the array (a[(l+r)/2]) as our reference point. Then, we start scanning from both ends of the array. From the left, we look for an element greater than our reference point, and from the right, we look for an element less than the reference point. When we find both, we swap them. We repeat this process until there's no element on the left greater than the reference point and no element on the right less than the reference point.

In simpler terms, we have a variable 'v' holding the value from the middle of the array, and we have two pointers 'i' and 'j' representing the left and right ends. We move 'i' to the right and 'j' to the left until we violate the condition that no element on the left of 'i' is greater than 'v' and no element on the right of 'j' is less than 'v'. Once 'i' and 'j' meet, the partitioning is done.

4. Radix Sort

one of the sorting algorithms that uses the internal structure of the objects being sorted. The algorithm itself consists of sequentially sorting objects using some stable sort for each digit, in order from least significant to highest digit, after which the sequences will be arranged in the required order.

Examples of objects that are useful to partition and sort by are numbers and strings. For numbers there is already a concept of digit, so we will represent numbers as sequences of digits. Of course, in different number systems the digits of the same number are different, so before sorting, let’s present the numbers in a number system convenient for us.

Strings are sequences of characters, so the bits in this case are individual characters, which are usually compared using their corresponding codes from the encoding table. For such a split, the least significant bit is the last character of the line.

For the above objects, counting sort is most often used as stable sorting.

This approach to the algorithm is called LSD sorting (Least Significant Digit radix sort). There is a modification of the digital sorting algorithm that analyzes the digit values, starting from the left, with the most significant digits. This algorithm is known as MSD sorting (Most Significant Digit radix sort).

Methodology

The C++ implementation of the algorithms produced results across four distinct categories:

1) Randomly shuffled arrays of valued 0, 1, …, n
2) Sorted Array(best case insertion sort)
3) Reversed Array(worst case scenarios)
4) Partially Sorted

The algorithms underwent testing on arrays of varying sizes: 10, 20, 100, 200, 1000, 2000, ..., 20000. The same dataset was applied to each algorithm across all five categories. Each array size underwent testing 100 times. The reported result for each size is the average time taken by each algorithm to sort the corresponding input array.

Results

Randomly shuffled array

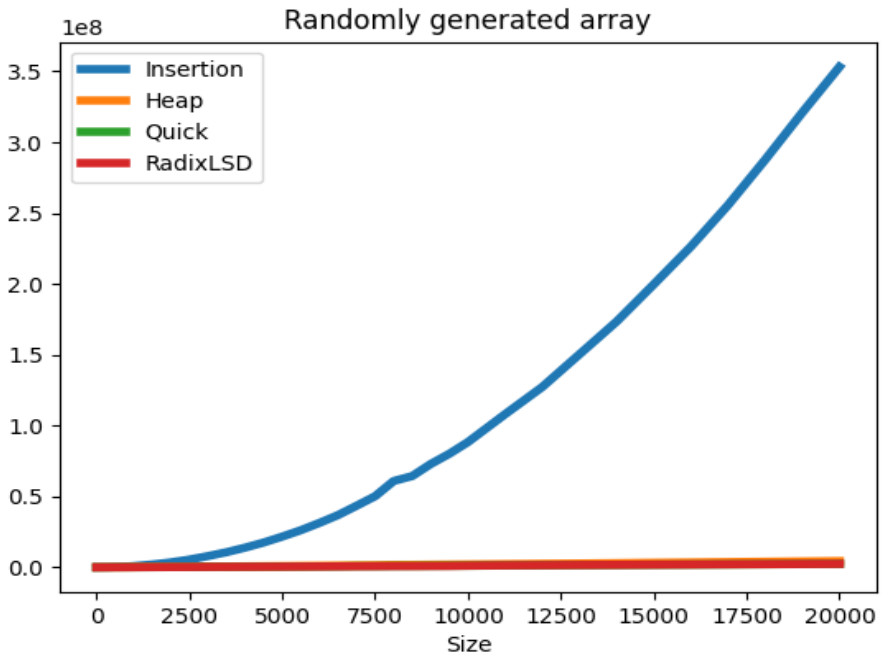


Figure 1: Average time each algorithm took to sort an array of randomly shuffled valued from 1 to n(average -case).

As seen in Figure 1, the graph highlights a significant difference in how long it takes for the Insertion Sort algorithm compared to other sorting methods. Its quadratic time complexity makes it less suitable for larger, randomly generated arrays compared to more advanced sorting algorithms with better average-case performance, such as Quick Sort, Heap sort or Radix sort.

For an array of size 20,000, Insertion Sort is about 80 times slower than Heap Sort and 146 times slower than Quick and Radix sorting algorithms. Similarly, with an array of size 10,000, it remains notably slower—40 times compared to Heap Sort and 78 times compared to the other sorting methods. To better understand how Heap, Quick, and Radix sorting algorithms perform, a more detailed analysis is needed.

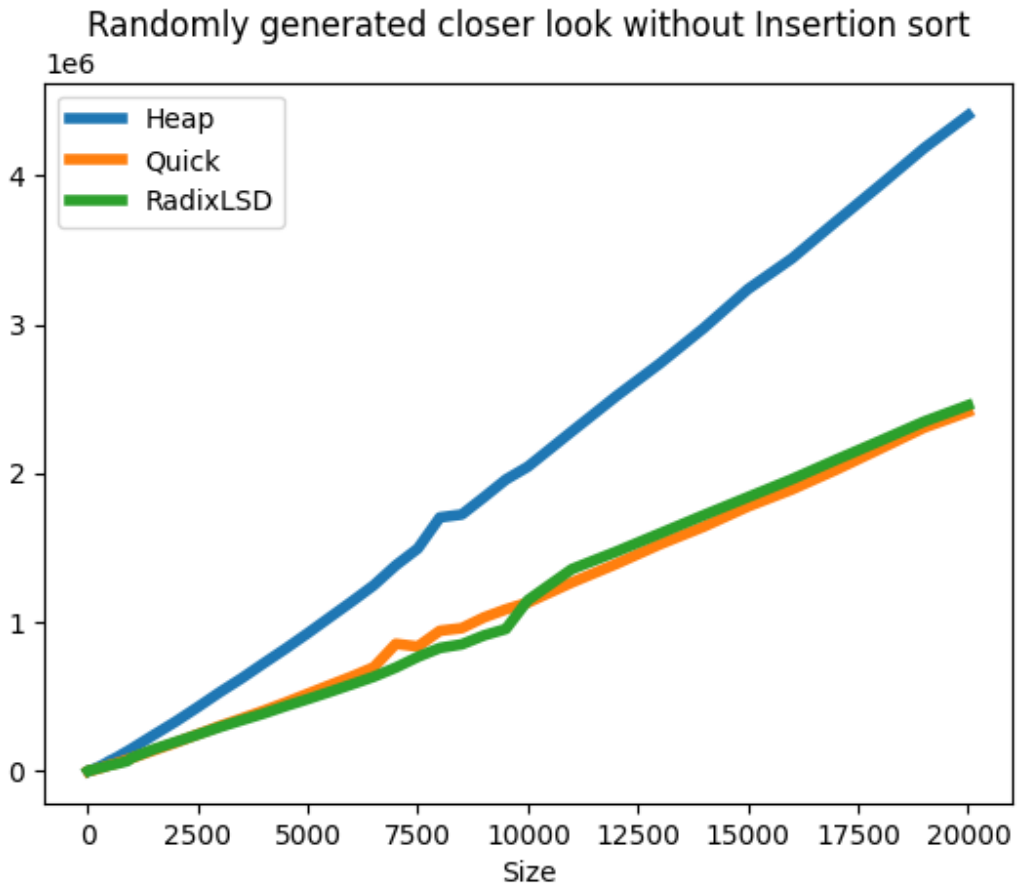


Figure 2: The graph shown in Figure 1 without Insertion Sort

Now, the differences between the algorithms are clearer. Quick and Radix Sorting algorithms show similar performance for array sizes ranging from 0 to 20,000. Interestingly, Radix Sort is a bit faster than Quick Sort in the array size range of 3,000 to 10,000, but as the array size grows, Radix Sort becomes slower, as expected. Quick Sort's efficiency in handling randomly generated arrays is due to its divide-and-conquer strategy, which generally performs well when the data has no specific order. Quick Sort is adept at partitioning the array into smaller segments, efficiently sorting them, and then combining the sorted segments. Heap Sort consistently takes more time, being 1.8 times slower than Quick and Radix sort algorithms. Heap Sort, while still effective, may incur a slightly higher cost in terms of time complexity compared to Quick Sort and Radix Sort when dealing with random data. In summary, all three algorithms have an average time complexity of O(n log n).

Let's have a closer look on small size of randomly generated array, because the results are going to be more interesting.

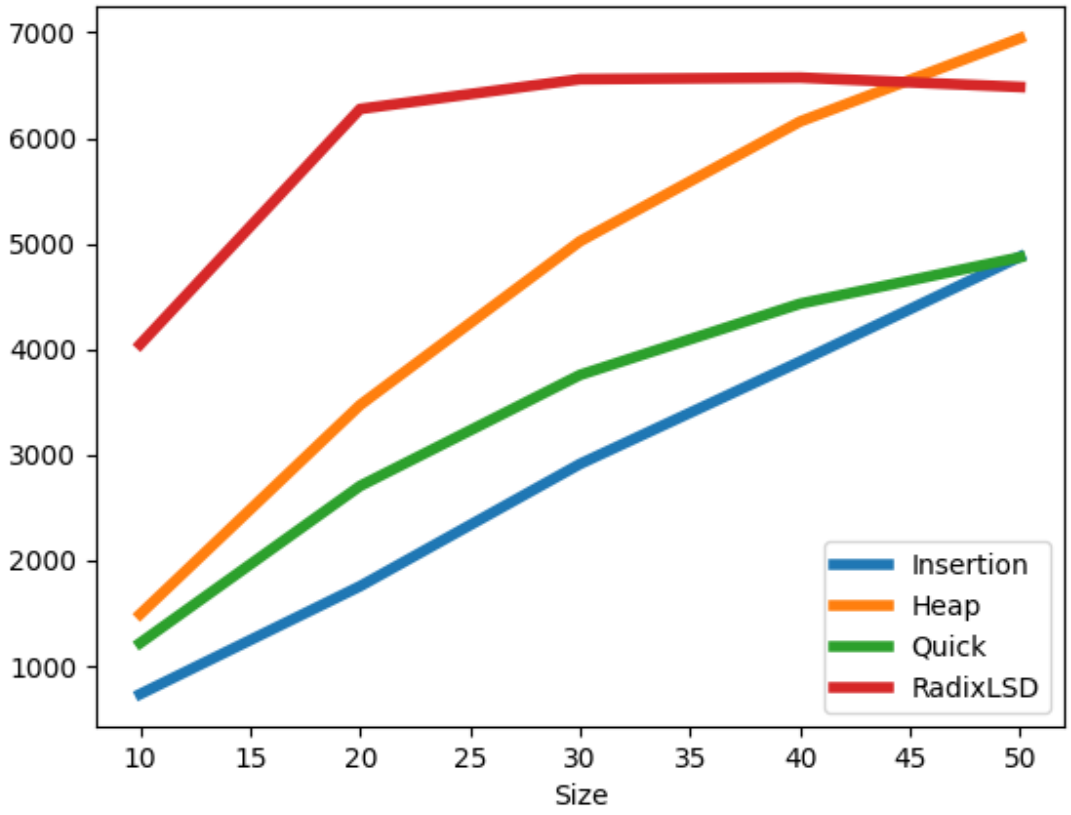


Figure 3: The graph shown in Figure 1 for small size array sizes.

The graph from Figure 3. shows us different results. For small size arrays(it depends on your computer, algorithm implementation, programming language) from 0 to 50, insertion sort is faster than other sorting algorithms. Despite its quadratic time complexity, Insertion Sort can outperform more complex algorithms like Quick Sort and Heap Sort for small data sets, which we going to use build our hybrid sort.

Sorted array

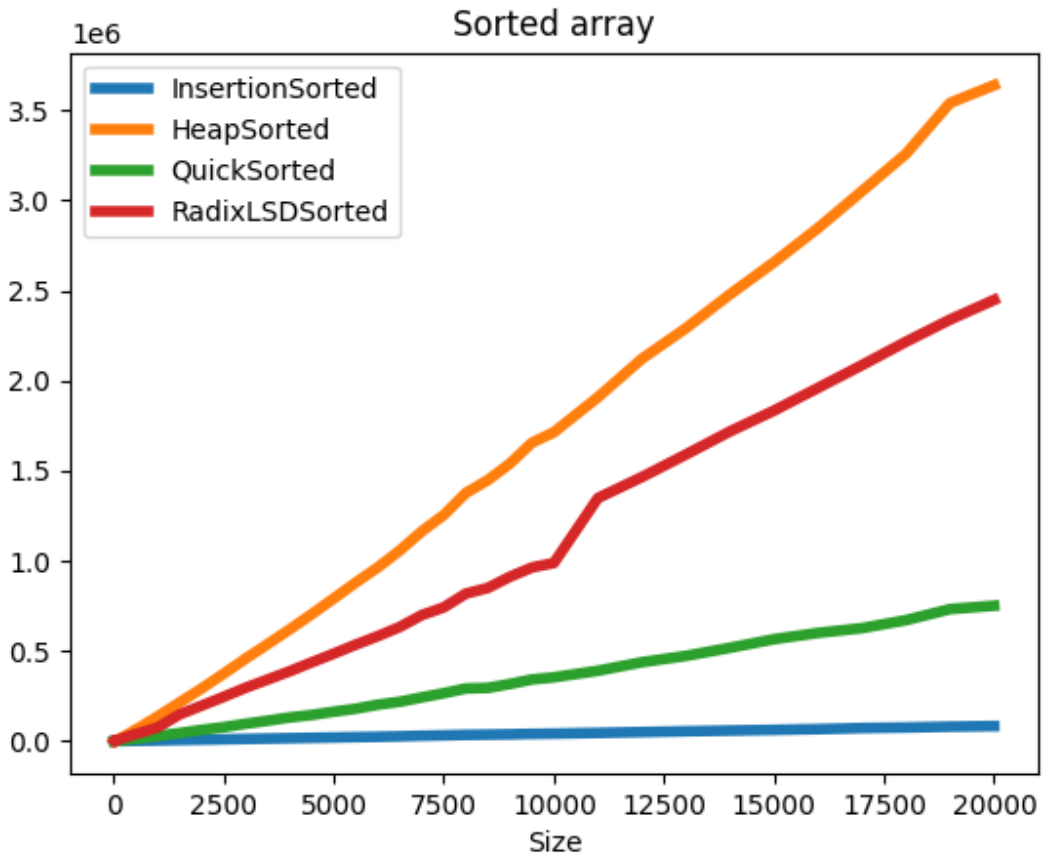


Figure 4: Average time each algorithm took to sort an array of sorted values

Insertion Sort has a favorable characteristic when dealing with nearly sorted or already sorted arrays. Its time complexity is $O(n)$ in the best case scenario, which occurs when the input array is already sorted. This is because in an already sorted array, Insertion Sort has minimal work to do; it only needs to iterate through the sorted portion of the array and insert each element into its correct position. Insertion sort appears to be approximately 8-9 times faster than Quick Sort. Heap sort is the slowest one when the array is sorted, 5 times slower than Quick sort, 40 times slower than Insertion Sort algorithm.

Reversed array

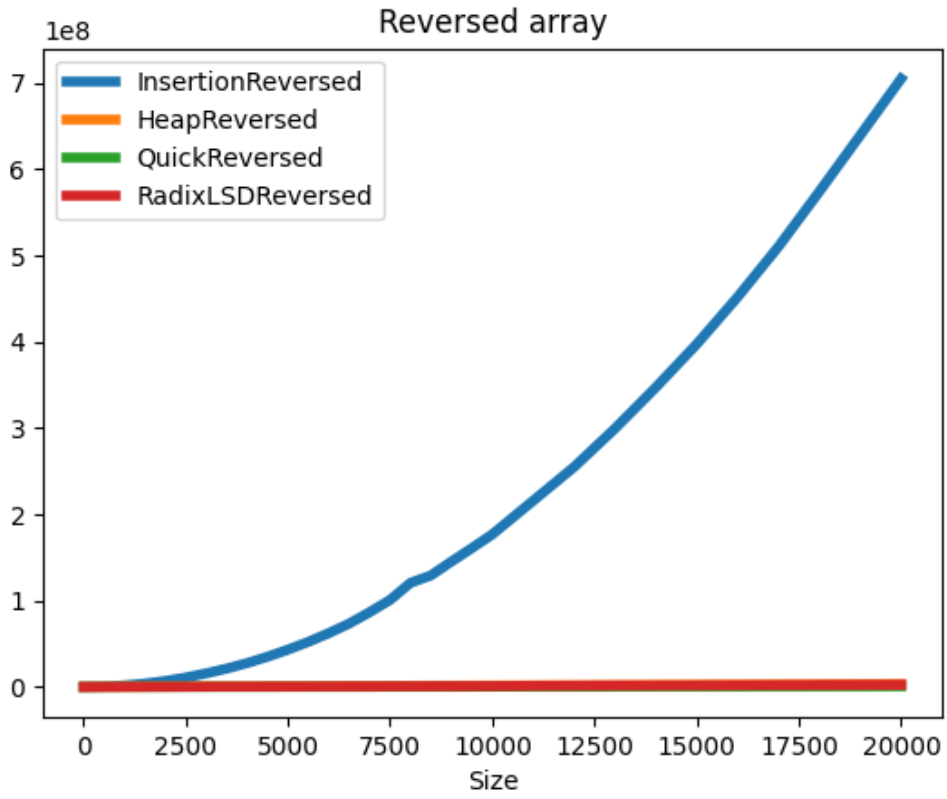


Figure 5: Average time each algorithm took to sort a reversed array(worst-case scenarios)

In Figure 5, the graph shows similiar contrast in the runtime performance of the Insertion Sort algorithm compared to other sorting approaches, particularly Heap, Quick, and Radix sorting algorithms, as in Figure 1. Once again, Insertion Sort exhibits significantly longer execution times, posing challenges in distinguishing its performance from other methods. For an array of size 20,000, Insertion Sort is 921 times slower than Quick Sort. As we have said earlier, in case where array is in non-ascending order, it will take n(n-1)/ 2 operations. Which will lead to worst time complexity $O(n^2)$. To gain deeper insights into the performance of Heap, Quick, and Radix sorting algorithms in reversed array, a more detailed analysis is needed.

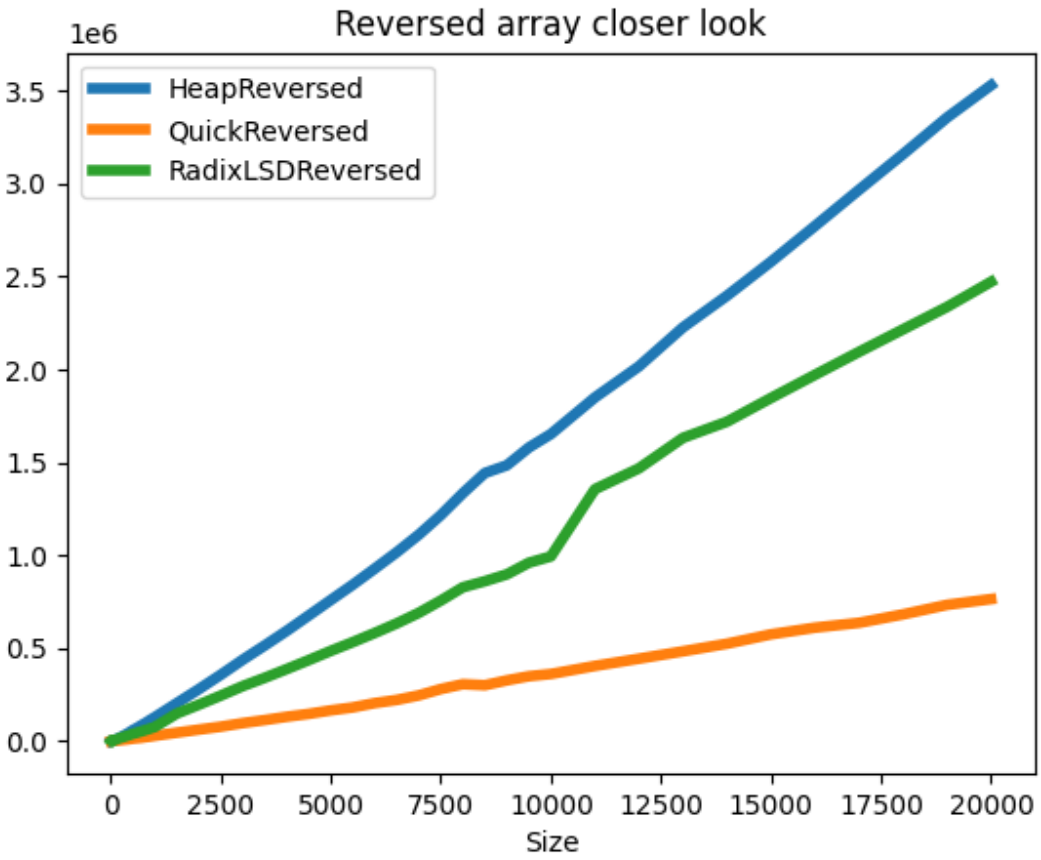


Figure 6: The graph shown in Figure 5, closer look, without InsertionSort

The Figure 6 is similar to Figure 2, but Quick Sort is faster than Radix sort by 3 times and by 4.6 times than Heap sorting algorithms. In Figure 2, Quick and Radix sorting algorithms in range [0, 20,000] were very close to each other. In reversed arrays, Radix Sort remains relatively unaffected by the order of the input data, as its performance is not influenced by the initial arrangement of elements. Quick Sort may encounter poor pivot choices, leading to unbalanced partitions and, consequently, degraded performance. Despite its generally efficient performance on random data, the worst-case behavior can result in increased runtime.

Sorted array

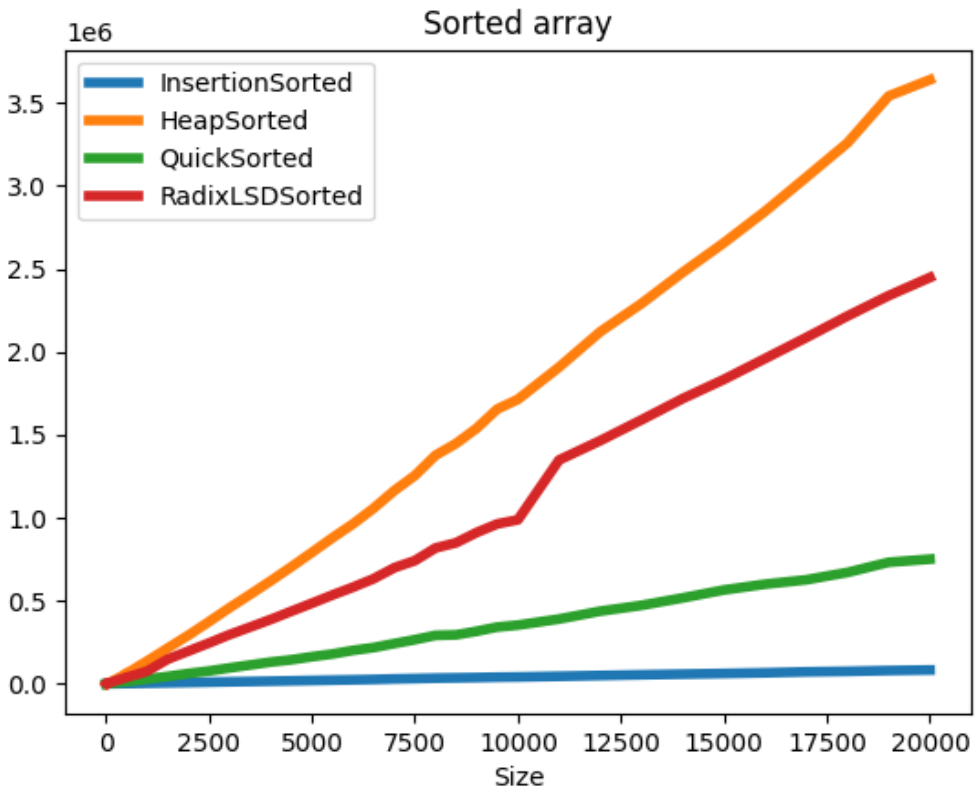


Figure 4: Average time each algorithm took to sort an array of sorted values

Insertion Sort has a favorable characteristic when dealing with nearly sorted or already sorted arrays. Its time complexity is $O(n)$ in the best case scenario, which occurs when the input array is already sorted. This is because in an already sorted array, Insertion Sort has minimal work to do; it only needs to iterate through the sorted portion of the array and insert each element into its correct position. Insertion sort appears to be approximately 8-9 times faster than Quick Sort. Heap sort is the slowest one when the array is sorted, 5 times slower than Quick sort, 40 times slower than Insertion Sort algorithm.

Partially sorted array

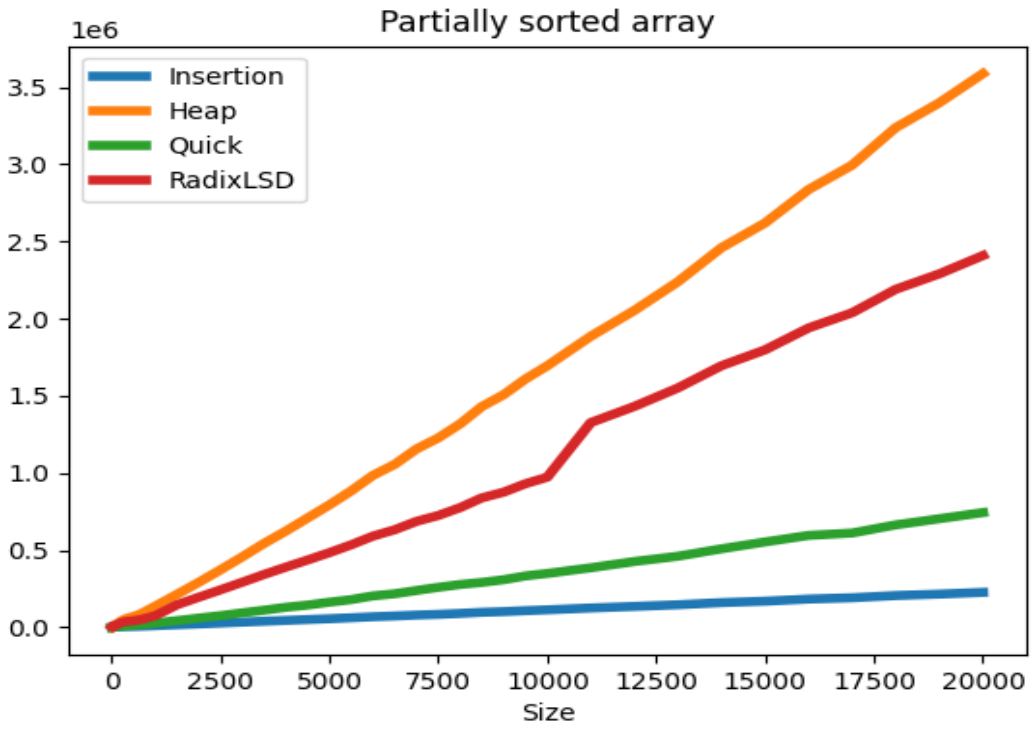


Figure 8: Average time each algorithm took to sort a partially sorted array

Heap Sort has a consistent time complexity of $O(nlogn)$ regardless of the initial order of the elements. While it doesn't take advantage of partial sorting in the same way that Insertion Sort does, it remains a reliable and efficient algorithm for a wide range of scenarios, including partially sorted arrays. **Quick Sort**'s efficiency can be impacted by the degree and pattern of partial sorting. If the array is partially sorted, Quick Sort may still benefit from its divide-and-conquer strategy, resulting in relatively fast sorting. However, if the partial sorting introduces patterns that lead to poor pivot choices, Quick Sort's performance could degrade, approaching its worst-case time complexity of $O(n^2)$. But we can make almost impossible to achieve time complexity of $O(n^2)$ by using other partitioning algorithms, and by choosing right pivot. For example, choosing pivot at random index can make it almost impossible to achieve worst case scenarios. **Radix Sort** generally performs consistently, as its complexity is less affected by the initial ordering of the elements.

Hybrid Sort

Finally let's get into out hybrid sort. I've implemented a hybrid sorting algorithm that combines the strengths of both Insertion Sort and Quick Sort. This approach is a common optimization strategy, taking advantage of the adaptability of Insertion Sort on small data sets and the efficiency of Quick Sort on larger data sets. The modification to switch to Insertion Sort when the size of the subarray is smaller than or equal to 50 is a form of a hybrid sorting algorithm. By using Insertion Sort for small subarrays, you take advantage of its adaptive nature. Insertion Sort's performance improves when dealing with partially sorted or small data sets due to its linear time complexity in these cases. Quick Sort is efficient on larger data sets and is widely used for its average-case time complexity of $O(nlogn)$. By continuing with Quick Sort for subarrays larger than 50, we leverage its strengths for handling larger amounts of data. Ensure that the transition point (50 in your case) is chosen based on empirical testing and analysis of your specific use cases. The optimal threshold may vary depending on factors such as the characteristics of the data, hardware, and compiler optimizations. It's essential to thoroughly test and benchmark your hybrid sorting algorithm on a variety of data sets to validate its performance and confirm that it meets your expectations.

```
void quicksort(a: T[n], int l, int r)
      if (r - l ⩽ 50)
            insertion(a, l, r)
            return
     int i = partition(a, l, r)
     quicksort(a, l, i)
     quicksort(a, i + 1, r)
```

Choosing the median of the first, middle, and last elements as the separating element and cutting off the recursion of smaller subarrays can lead to significant improvements in the efficiency of quicksort. The median function returns the index of the element that is the median of three elements. After that, it and the middle element of the array swap places, with the median becoming the separating element. Small arrays (length M=11 or less) are ignored during the division process, then insertion sort is used to complete the sorting.

```
void quicksort(a: T[n], int l, int r)
      if (r - l ⩽ 50)
            insertion(a, l, r)
            return
     int med = median(a[l], a[(l + r) / 2], a[r])
     swap(a[med], a[(l + r) / 2])
     int i = partition(a, l, r)
     quicksort(a, l, i)
     quicksort(a, i + 1, r)
```

Below the graphs of hybrid sort results where difference seen compared to Insertion, Quick, Heap, and Radix sorting algorithms.

Figure 1.1.Average time each algorithm took to sort an array of randomly shuffled valued from 1 to n(average -case).

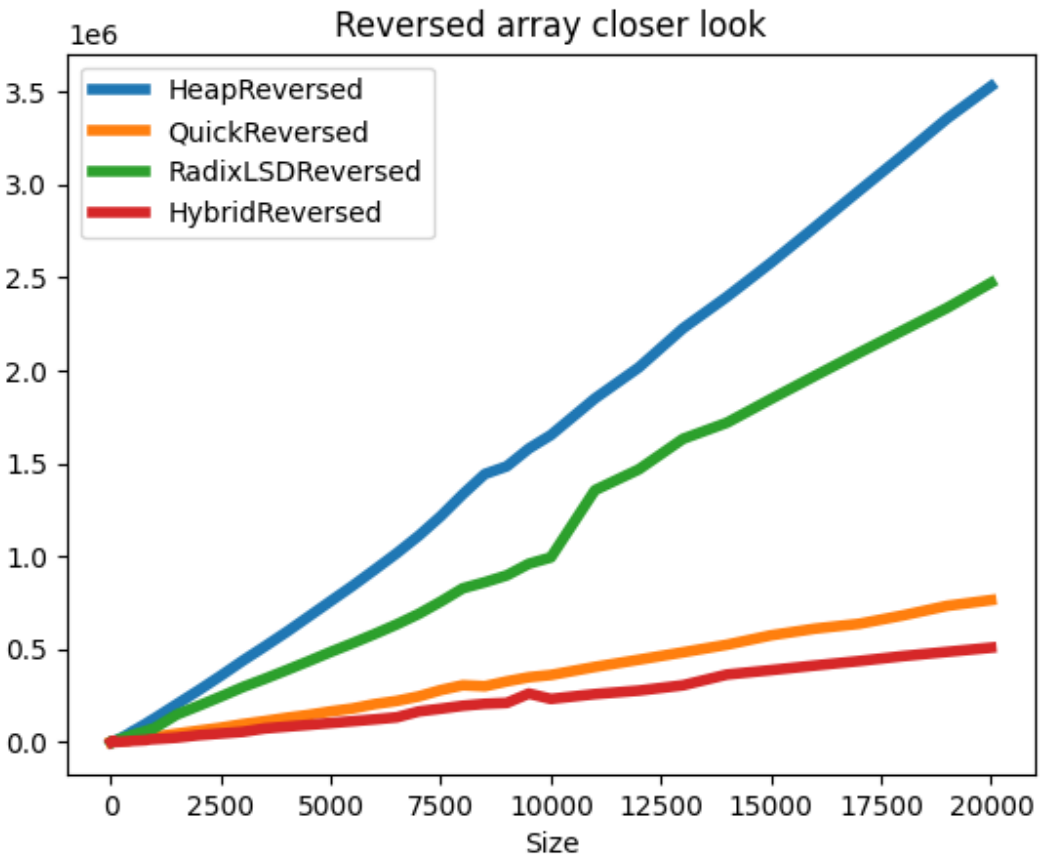
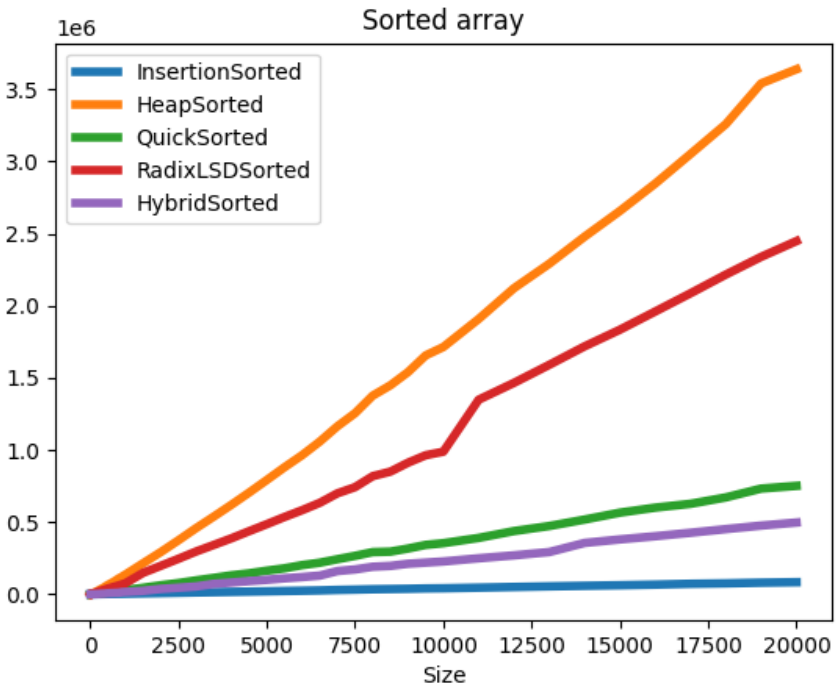


Figure 2.2.Average time each algorithm took to sort a reversed array.



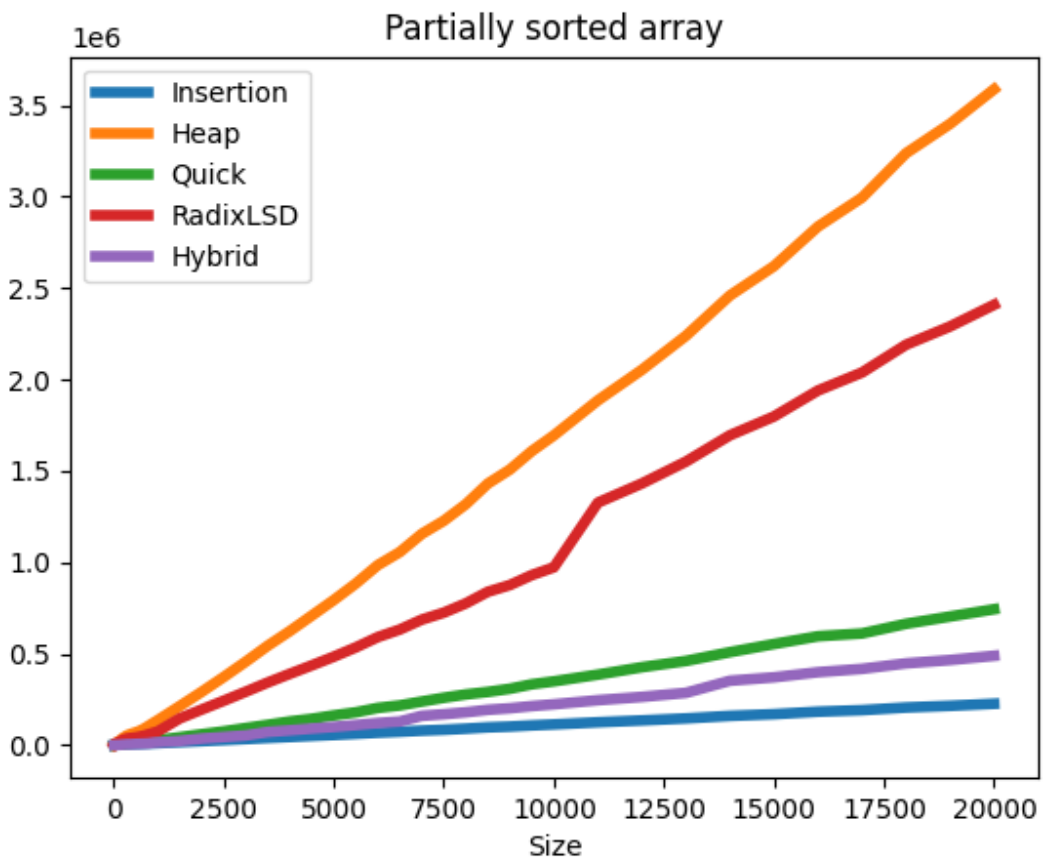


Figure 4.5.Average time each algorithm took to sort a partially sorted array.

Conclusions

In wrapping up our exploration of sorting algorithms, we've uncovered the strengths and weaknesses of classics like Insertion Sort, Quick Sort, Heap Sort, and Radix Sort. Each has its own merits, with Insertion Sort excelling in smaller or partially sorted arrays, Quick Sort dominating on larger datasets, Radix Sort being robust across various data types, and Heap Sort remaining reliable but potentially slower in specific scenarios.

Our research introduces a practical solution—a hybrid sorting algorithm. This hybrid approach combines the adaptability of Insertion Sort and the efficiency of Quick Sort. By smartly switching between them based on the size of the data, our hybrid algorithm aims to find a sweet spot, balancing adaptability and efficiency for different data sizes.

Sorting algorithms are not one-size-fits-all. The best choice depends on the data at hand. Our hybrid solution offers a flexible alternative, emphasizing the importance of considering the specific characteristics of the data and the computational resources available.

In summary, our research provides insights into sorting algorithms, showcasing their diverse performances. The hybrid approach offers a practical way forward, demonstrating the value of combining adaptability and efficiency in the pursuit of effective sorting. As we look ahead, further refinements and adaptations of sorting algorithms will continue to play a crucial role in data processing and computational efficiency.