Algorithm Design and Analysis

***Abstract*—In this project, I have implemented different sorting algorithms using Python. The code tries to plot the graph showing the time taken for sorting an array of length n, for each sorting algorithm based on different data scenarios. The graphs from different sorting algorithms are then used to compare and analyze the algorithms.**

***Index Terms*—Data, Link, Quick Sort, Heap Sort, Merge Sort, Radix Sort, TimSort, Bucket Sort.**

I. Introduction

The sorting algorithm in computer science has evolved through ages and is used in a wide variety of places. The project's purpose is to perform an in-depth analysis of different sorting algorithms. So the project tries to show the strengths and weaknesses of the algorithms through a comparison of each algorithm in different arrays of different sizes and element compositions. The algorithms we have considered are quick sort, heap sort, merge sort, radix sort, timsort, and bucket sort.

II. Data

I have considered data from each of the scenarios of data mentioned by the professor. For this, I have used the Python numpy library to generate the data in a given range. In running algorithms, I have made sure that the array taken for comparison is consistent for each algorithm. This is ensured by randomly generating an array a of size i and then calling all algorithms one by one on the same array a to get an unbiased comparison. **The quick sort is executed separately with other algorithms with array size as 10k.** While for other sorting algorithms I ran it for array length 1000 to 50,000 with data belonging to following scenario:

- scenario 1: n randomly chosen integers in the range [0, n]
- scenario 2: n randomly chosen integers in the range [0, K], $K < 1000$
- scenario 3: n randomly chosen integers in the range [0, $n \times n \times n$]
- scenario 4: n randomly chosen integers in the range [0, $\log(n)$]
- scenario 5: n randomly chosen integers that are multiples of 1000 in the range [0, n]
- scenario 6: the in order [0, n] integers where $\log(n)/2$ randomly chosen values have been swapped with another value

III. Code Link

For more information, visit the GitHub repository:

- https://github.com/iit2018062/sortingalgos

IV. Quick Sort

Quick Sort is a sorting algorithm based on the divide and conquer approach. It involves selecting a pivot element and partitioning the array around this chosen pivot, aiming to find the precise position of the pivot in the array.we choose an element from the array, this element is called a pivot. This is used to divide the array into two sublists one sublist consisting of elements greater than the pivot and another less than the pivot After rearranging the element we get the final position of the pivot we recursively apply the algorithm in the subsequent two sublists.

A. Time Complexity

- **Best Case:** $\Omega(N \log N)$ - This occurs when the pivot choice consistently divides the array into roughly two equal halves.
- **Average Case:** $\theta(N \log N)$ - On average, Quick Sort performs in $\theta(N \log N)$.
- **Worst Case:** $\mathcal{O}(n^2)$ - The worst-case scenario happens when the pivot chosen at each step consistently leads to highly unbalanced partitions. This is common when the array is already sorted, and the pivot is the lowest or highest element.

B. Space Complexity

- **Auxiliary Space:** $O(1)$ - This represents the additional space required by the algorithm. When not considering the recursive stack space, the auxiliary space is constant. However, if we consider the recursive stack space, it could go up to $O(n)$ in the worst case.

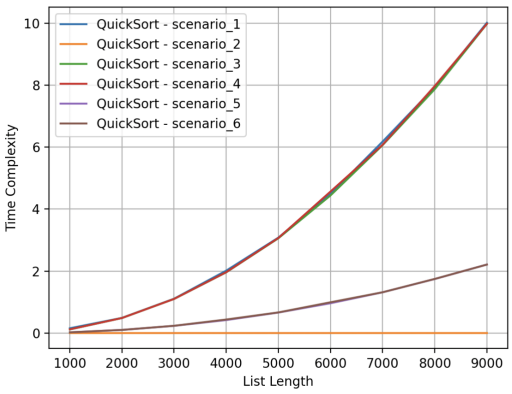

Fig. 1. Quick sort time vs length graph. The time is in seconds.

For quick sort we show an unexpected time-complexity. The code ran for array length starting with 1000 to 10,000. The time taken reached up to 10 seconds for array size 10,000 in certain Scenario. In scenario 6 the data in array is almost

ordered with only logn/2 elements swapped so quick sort will perform badly here.

V. Heap Sort

This sorting technique is based on binary heap. It is somewhat similar to the selection sort, where we find the minimum element, place the element at the start, and repeat the process until we process every element. The algorithm is based on min heap or max heap technique. we try to build a max heap of elements present in the array. once our heap is ready, the largest element will be on the root node i.e, on A[0] Now we swap the element at A[0] with the last element of the array and call the heapify with the array element excluding the last element. Repeat the process till all elements are sorted

A. Time Complexity

- **Overall time Complexity:** $\theta(N \log N)$.

B. Space Complexity

- **Auxiliary Space:** $logN)$ - T this is due to the recursive call stack, if we do it using an iterative method this will be O(1) The best thing about this algorithm is it is an in-place sorting algorithm, this algorithm is non-stable but can be made stable.

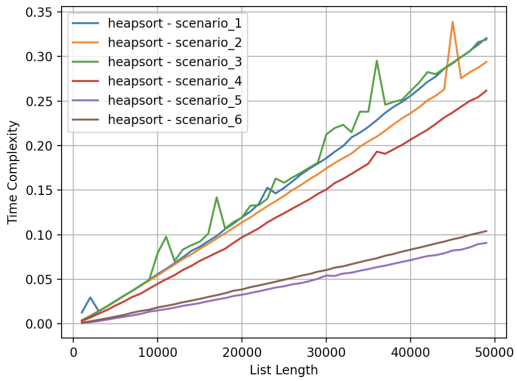

Fig. 2. Heap sort time vs length graph. The time is in seconds.

The performance of heap sort was quite good. With highest time taken reaching to 0.35 seconds. The array length varies from 1000 to 50k. Interesting thing to notice is the algorithm preformed really well in scenario 5 and 6 were data consist of more orders.

VI. Merge Sort

This sorting algorithm is based on dividing the array into subarrays of smaller size, and then sorting these subarrays and then merging them back.we divide the array around a midpoint which is calculated as (start+end)/2.we recursively apply merge sort in each half obtained from step 1. We stop when the array size is 1 as an array of size 1 is always sorted. we merge the sorted subarrays into one large array.Continuously process until elements are merged into one single sorted array. Image taken from source :

A. Time Complexity

- **Best Case:** $\theta(N \log N)$ - On average, Quick Sort performs in $\theta(N \log N)$.

B. Space Complexity

- **Auxiliary Space:** $O(n)$ - all elements are copied into an auxiliary array.

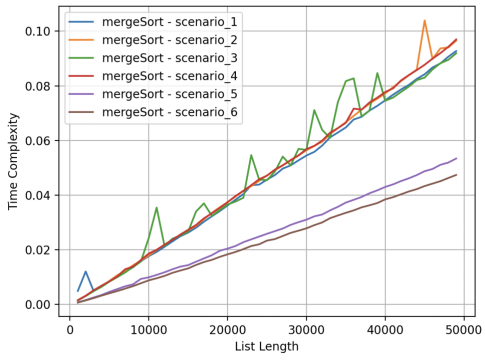

Fig. 3. Merge sort time vs length graph. The time is in seconds.

The performance of merge sort was good. The maximum time reached came out to be 0.1 which is less than heap sort and way too less than quick sort.

VII. Radix Sort

This is a linear sorting algorithm which sorts the elements in an array by processing them digit by digit. This algorithm performs efficient sorting for integers or strings of fixed size.Find out the maximum value in an array, then count the number of digits in the maximum value. Let's suppose this number comes around k. we will iterate k times, for each significant digit place. sort the array based on the values of digits present in the kth significant place. After processing k times we get a sorted array

A. Time Complexity

- **Best Case:** - The time complexity for the radix sort comes around to be 0(d*(n+b)) where d is the number of digits, n is the number of elements in tha array and b is the base of the number system we are using. As we can see this is a very efficient algorithm however the time complexity increases linearly with a number of digits so it is not that great for small data sets.

B. Space Complexity

- **Auxiliary Space:** $O(n + b)$ - Radix sort takes a space complexity of O(n+b)

As expected radix sort shows a higher time taken for scenario 3 where element in array range from [0, n*n*n]. This was expected because radix sort works digit by digit. For other scenarios radix sort performed well.

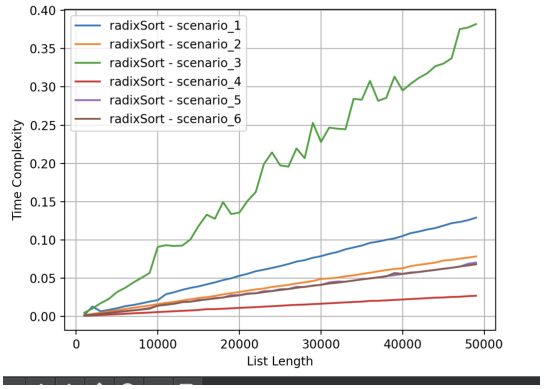

Fig. 4. Radix sort time vs length graph. The time is in seconds.

VIII. Tim Sort

The idea behind tim sort is taking advantage of existing order in the elements of array to minimize the number of swaps. The algorithm iterates over the elements and collect them into runs, these runs are then put into stack. While adding a run in a stack if the existing top run of stack matches the criteria of merge we merge the two runs. This goes on till all the runs are merged into one. We have seen how merge sort take the advantage of merging ordered list. This makes comparison less. In the similar way tim sort utilizes the advantage of merging ordered lists. The size of runs is pre-decided while writing the code.

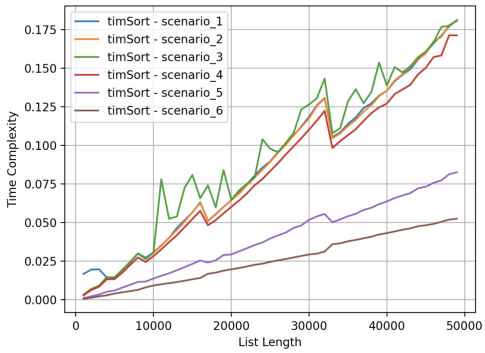

Fig. 5. Tim sort time vs length graph. The time is in seconds.

A. Time Complexity

- **Best Case:** $\Omega(N)$
- **Average Case:** $\theta(N \log N)$ - On average, Tim Sort performs in $\theta(N \log N)$.
- **Worst Case:** $\theta(N \log N)$

B. Space Complexity

- **Auxiliary Space:** $O(N)$

Tim sort shows a time complexity similarly to radix sort. In all cases in-fact better than radix sort.

IX. Bucket Sort

This algorithm relies on dividing the elements in the array into different groups a commonly called buckets. The distribution of elements is uniform. These buckets are then sorted using any sorting algorithms and finally, these sorted elements are merged to form a sorted array. This sorting algorithm is mainly used when the elements in an array are well distributed over a given range. It is commonly used with floating point numbers because for this it is easier to determine the range of each bucket. we create n different bucket.The element in the i th position of the array is stored at bucket[n*arr[i]] These buckets are then sorted using insertion sort. we merge all the buckets

A. Time Complexity

- **Worst case:** $\mathcal{O}(n^2)$
- **Average case:**$O(n)$

B. Space Complexity

- **Auxiliary Space:** $O(n+k)$ - where k is the average size of each bucket

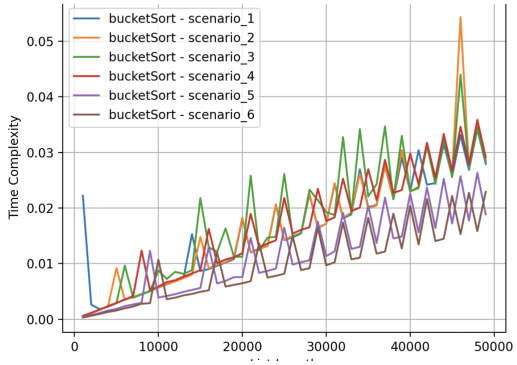

Fig. 6. Bucket sort time vs length graph. The time is in seconds.

Bucket short performed well in all cases. It was able to utilize the distribution of data over a range.

X. Different algorithm versus data scenario:

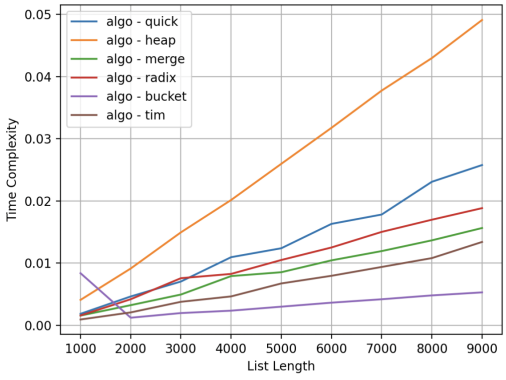

Fig. 7. Scenario-1 versus algorithms. The time is in seconds. Almost every algorithm performed the same with heap taking top place and bucket taking bottom.

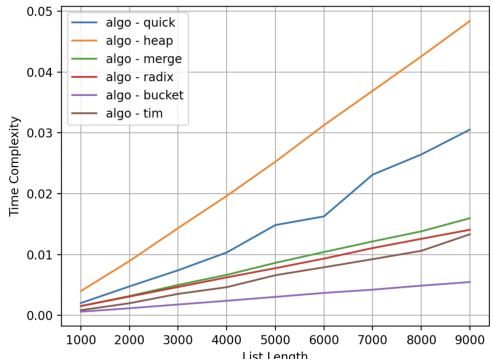

Fig. 8. Scenario-2 versus algorithms. The time is in seconds. In this heap sort performed the top while bucket the bottom of the performance.

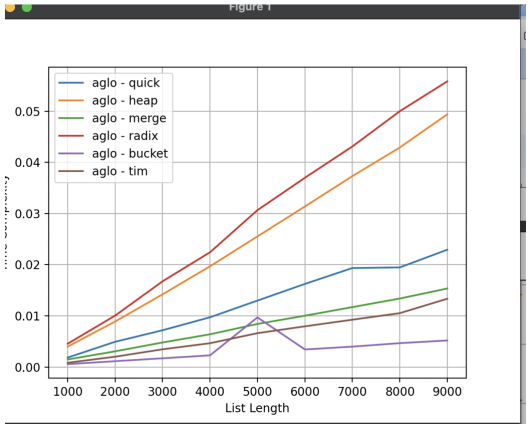

Fig. 9. Scenario-3 versus algorithms. The time is in seconds. As seen bucket sort is performing bad here due to size of the digits in number.

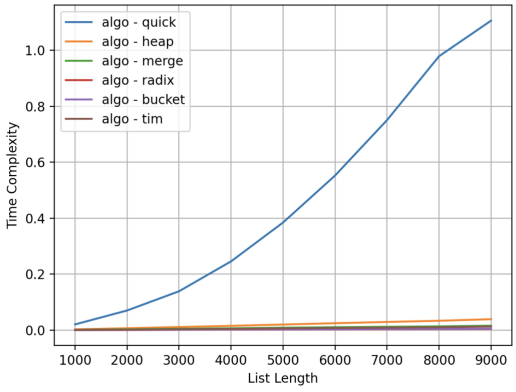

Fig. 10. Scenario-4 versus algorithms. The time is in seconds. The quick sort performs the worst in this case.

XI. Thought on Quick Sort

The quick sort complexity comes around O(nlogn) for average case. However what we saw in this peroject seems little contradicting as it performed worst out of all the algorithms in terms of time complexity. Upon analysis, there can be following reasons for such result:

- The pre-existing sorted pattern leading to worst case complexity.
- Recursion depth, array size of 50k exceeds the maximum recusive depth normally provided by python. Upon inserting the sys.setrecursionlimit(10 ** 6), the code was able to exceed the limit. With excessive recursion comes the recursive time overhead. This is true for merge sort as well, but in case of merge sort it is a stable sorting algorithm depending on cache efficiency while quick sort involves more random access pattern in term of caching.
- Quick sort is not cache friendly which create to more cache misses for a huge data structure.
- with this said the selection of pivot is very important for quick sort, as we are working with fixed pattern for choosing pivot, we can definitely adopt some better pivot selection algorithm.

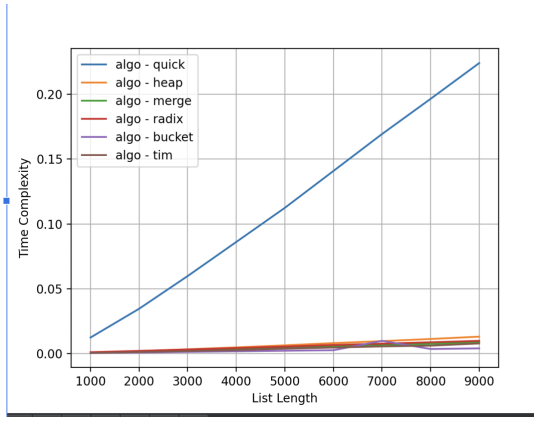

Fig. 11. Scenario-5 versus algorithms. The time is in seconds. Here quick sort is performs the worst.

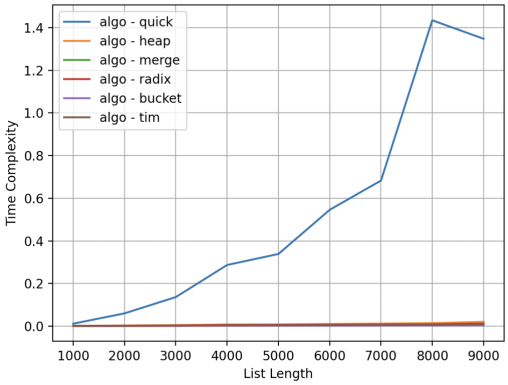

Fig. 12. Scenario-6 versus algorithms. The time is in seconds. Here quick sort performs the worst because the data has orders in it.

XII. Conclusion:

All the algorithms was run over array size starting from 1000 to 50,000. Some interesting observation can be made looking at these graphs:

- Quick sort performed the worst out of all in terms of time-complexity for most cases.

- Out of all algorithm bucket sort performed the best followed by merge sort, and tim sort.
- For scenario 1 almost every sorting algorithm performed in same time.
- scenario 2, we did not see much change in this, it was similar to first scenario 1.
- scenario 3, Radix sort took a long time for this as it sort elements digit by digit. The numbers in this case range from 0 to n*n*n which resulted in a large digits number.
- scenario 4, this case has data in logarithmic fashion. Radix sort works best for data of this sort range. This is also confirmed by the graph we got for this case from radix sort.
- scenario 5: for this scenario bucket sort works best as the data is discrete and range is limited.
- scenario 6, this scenario has a certain pattern in data which means quick sort will not perfrom well. In a ordered array we have swapped, (log(n)/2) elements which still leaves the array mostly ordered for larger scenario. The quick sort seems to perform bad for this.

XIII. Reference

- https://www.geeksforgeeks.org/python-code-for-time-complexity-plot-of-heap-sort/
- https://www.geeksforgeeks.org/quick-sort/
- https://www.geeksforgeeks.org/merge-sort/
- https://www.geeksforgeeks.org/sorting-algorithms/