CS 584 - Algorithm Design and Analysis Project Report

Analysis and Comparison of QuickSort, TimSort, and RadixSort

PSU ID: 909715404

Name: Devyani Shrivastava

Computer Science

June 2019

Contents

1. Introduction:

The sorting algorithms are broadly classified into comparison based and non - comparison based algorithms based upon how they work. The fundamental difference between them is the comparison decision.

Quicksort and **Timsort** are comparison based algorithm, a type of sorting algorithm that only reads the list elements through a single abstract comparison operation that determines which of two elements should occur first in the final sorted list. A comparator is required to compare numbers or items. Basically, this comparator defines the ordering e.g. numerical order, lexicographical order, also known as dictionary order, to arrange elements. QuickSort (sometimes called partition-exchange sort) is one of the most efficient sorting algorithms and is based on the splitting of an array into smaller ones. It is capable of sorting a list of data elements significantly faster than any of the common sorting algorithms. TimSort is a sorting algorithm based on Insertion Sort and Merge Sort and is a stable algorithm. It has been Python's standard sorting algorithm since version 2.3.

There are some sorting algorithms that perform sorting without comparing the elements rather by making certain assumption about the data they are going to sort. **Radix sort** is one of the non-comparison based sorting algorithm, which examines individual bits of keys, and hence it doesn't need to go through comparison decision tree.

The report includes the analysis of performance of Quicksort, Tim sort, and Radix Sort on various type of inputs. **Section 2** will discuss about Quicksort, having subsection as: **subsection 2.1** briefly describing Lomuto Partitioning Technique, **subsection 2.2** comprises of brief discussion of implementation of quicksort with naïve pivot selection along with its best and the worst case complexity. **Subsection 2.3 and 2.4** will be discussing about quicksort with median of medians and random pivot selection methods respectively. **Section 3** gives a short introduction to Timsort, briefing about Minrun and complexities of TimSort. **Section 4** presents us with Radix Sort, giving a short description about its working, and time complexities. **Section 5** consists of the description of experiment and all the variations that has been done on the input in order to analyse the algorithms in detail on different input types and sizes. **Section 6** contains results and execution performance of the three algorithms and comparison between these techniques. **Section 7** will be the conclusion.

2. QuickSort

Quicksort is a comparison sort, meaning that it can sort items of any type for which a "less-than" relation (formally, a total order) is defined. It is a divide and conquer algorithm. It first divides a large array into two smaller sub-arrays: the low elements and the high elements. Quicksort can then recursively sort the sub-arrays. The steps are:

1. Pick an element, called a pivot, from the array.
2. Partitioning: reorder the array so that all elements with values less than the pivot come before the pivot, while all elements with values greater than the pivot come after it (equal values can go either way). After this partitioning, the pivot is in its final position. This is called the partition operation.
3. Recursively apply the above steps to the sub-array of elements with smaller values and separately to the sub-array of elements with greater values.

The following procedure implements quicksort for sorting a typical subarray A[p…r] :

Algorithm: QuickSort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = PARTITION(A, p, r)
4:         QUICKSORT(A, p, q - 1)
5:         QUICKSORT(A, q + 1, r)
```

The pivot selection and partitioning steps can be done in several different ways; the choice of specific implementation schemes greatly affects the algorithm's performance. There are different algorithms available for pivot selection with strategies such as median of-median using group 5, median-of-median using group 7 and randomized pivot selection. The algorithm has a worst-case running time of n^2 on an input array of n numbers. Despite this slow worst-case running time, quicksort is often the best practical choice for sorting because it is remarkably efficient on the average: its expected running time is O(n log n),and the constant factors hidden in the O(n log n) notation are quite small.

2.1 Lomuto Partitioning

This partitioning method selects the pivot as a last element of an array, around which to partition the subarray. After selection of pivot, algorithm uses index 'i' and 'j' to scan the array such that elements from index 'p' to 'i' i.e. A[p..i] are less than or equal to pivot 'x' and elements from index 'i+1' to 'j-1' i.e. A[i+1...j-1] are greater than pivot 'x'. Here 'p' is the leftmost index of an array 'A'. This partitioning method is easy to understand so it is frequently used than Hoare partition even though Hoare partition is more efficient.

The algorithm to partition is given as:

Algorithm: Lomuto Partitioning

```
1. procedure PARTITION(A, p, r)
2.     x = A[r]
3.     i = p -1
4.     for j = p to r-1 do
5.         if A[j] ≤ x then
6.             i = i+1
7.             exchange A[i] with A[j]
8.     exchange A[i+1] with A[r]
9.     return i+1
```

The procedure PARTITION(A, p, r) takes 3 arguments as an input: input array A, leftmost array index 'p' and rightmost array index 'r'. This method selects A[r] i.e. last element of an array A as a pivot element in line 2 to partition the subarray A[p..r]. Lines 4-8 will execute till we get the pivot to its correct position in the array. To get the pivot's new index, it swaps the element A[j] which is less than or equal to pivot x with A[i] and final two lines finish up by swapping the pivot with leftmost element greater than x, thereby moving the pivot into its correct place in the partitioning array, and then returning the pivot's new index.

The running time of partitioning the array A[p..r] is $\Theta(n)$ since lines 4-7 executes n times where $n = r - p + 1$.

2.2 QuickSort with Naïve implementation

Quicksort uses divide and conquer paradigm to sort the subarray. In naïve implementation of the quicksort, algorithm chooses last element of an array as a pivot around which to partition the subarray. To partition the array into subarrays, the Lomuto Partition technique has been used.

The following algorithm shows the implementation of quicksort using last element of an array as a pivot:

Algorithm: Quicksort with Naïve implementation

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:     q = PARTITION(A, p, r)
4:     QUICKSORT(A, p, q - 1)
5:     QUICKSORT(A, q + 1, r)
```

Time Complexity: Algorithm QuickSort takes constant time at line2. Lines 4-5 recursively call the QUICKSORT, so this will take Θ(log n) time as the recurrence terminates at depth (log n) and partitioning takes Θ(n) time. So overall it will take Θ(n log n) time. Like insertion sort, quicksort has tight code and so the hidden constant factor in its running time is small.

The performance of this algorithm depends on whether the partitioning is balanced or unbalanced. The unbalanced partition is like one subproblem with (n-1) elements and one subproblem with 0 elements. The recurrence relation for subproblem with 0 elements is

$T(0) = \Theta(1)$ and recurrence for running time is,

$$\mathbf{T(n) = T(n-1) + \Theta(n) \quad for\ n > 0.}$$

This recurrence relation can be proved by using substitution method to get the solution

$\mathbf{T(n) = \Theta(n^2)}$ in the worst case.

For balanced partition where partition produces two subproblems of equal size, the recurrence for running time is then,

$$\mathbf{T(n) = 2\,T(n/2) + \Theta(n)}$$

This recurrence relation can be solved as

$\mathbf{T(n) = \Theta(n \log n)}$ in the best case.

Therefore using this method quicksort will take Θ(n log n) best case time and $\Theta(n^2)$ worst case time.

2.3 QuickSort with median of medians

The idea used here is to perform quicksort using median of median as a pivot element around which to partition the subarray. Using this strategy, we can get the O(n log n) running time of quicksort. The project involves the pivot selection using median-of-median by group 5 and group 7. The Implementation of quicksort using median-of-median by group 5 is shown below.

Algorithm: Median – of – Median element using group 5

```
1: procedure MEDIAN_QUICK(q, n)
2:     sublist = [ ]
3:     median = [ ]
4:     pivot = None
5:     if q.length ≥1 then
6:         for i in range(0, n, 5) do
7:             sublist.append (q [i: i+5])
8:         for j in sublist do
9:             s = sorted(j)
10:            if s.length > 0 then
11:                median.append (s[(s.length // 2)])
12:        if median.length ≤ 5then
13:            sorted(median)
14:            pivot = median [median.length // 2]
15:        else
16:            sorted(median)
17:            pivot = median_quick (median, median.length // 2)
18:    return pivot
```

Algorithm MEDIAN_QUICK works as follows**:** It takes two parameters as input i.e. array ’q’ and ’n’ is the length of array. It divides the n elements into [n/5] groups of 5 elements each and store these groups or subarrays. The medians of all these subarrays are calculated and stored into an array. Finally, the median is calculated from the sorted list of medians which serve as the pivot.

Algorithm QUICKSORT takes 3 arguments as input: an input array A, leftmost index 'p' of an array A and rightmost index 'r' of an array A. The only difference in this algorithm compared to naïve quicksort algorithm is lines3-4. In line 3, MEDIAN_QUICK method is called to select the median of medians element of an array A. This method will return the pivot element. Line 4 call PARTITION methods with 4 parameters as an input i.e. array A, leftmost index p, rightmost index r and pivot q. Lines 5 - 6 call QUICKSORT recursively to sort the array A.

Algorithm: QuickSort

```
1: procedure QUICKSORT(A, p, r)
2:     if p < r then
3:         q = MEDIAN_QUICK (A[p: r + 1], A.length)
4:         par = PARTITION(A, p , r, q)
5:         QUICKSORT(A, p, par-1)
6:         QUICKSORT(A, par +1, r)
```

Time Complexity: Lines 6-7 of MEDIAN_QUICK takes O(n) time to divide the array into subarrays. lines 8-11 consist of O(n) call of insertion sort on the set of size O(1). Line 17 takes T(n/ 5) times to recursively find the median-of-medians. This algorithm guarantees that (3/ 10) of input elements are less than the output pivot and other (3/ 10) elements are greater than output pivot. This yields that one partition operation reduces the length of A from n to (7n/ 10) in the worst case. Therefore, the algorithm MEDIAN_QUICK takes O(n) time to find the median-of-median element. PARTITION algorithm also takes O(n) time and QUICKSORT algorithm takes O(n log n) time to sort the array recursively as recurrence terminate at depth log n.

Therefore, total time complexity of sorting with this method is **O(n log n).**

The overall recurrence relation is,

T(n)=2T(n/2) + O(n) + O(n)

By solving this recurrence by master theorem, we get the solution

T(n) = O(n log n).

2.4 Quicksort with random pivot selection

This strategy involves selecting the randomly chosen element from subarray A[p..r] as a pivot. It is done by first exchanging last element of array A i.e. A[r] with element chosen at random from A[p..r]. By randomly sampling the range p...r, it can be ensured that pivot element x = A[r] is equally likely to be any of the r - p +1 elements in the subarray. Because the pivot is randomly chosen, the split of the input array is expected to be reasonably well balanced on average. The change to PARTITION and QUICKSORT are small. In this new partition procedure i.e., RANDOMIZED_PARTITION, we simply implement the swap before actually partitioning.

Algorithm: Selecting Random element as a pivot

```
1: procedure RANDOMIZED_PARTITION(A, p, r)
2:     i = randint (p, r)
3:     exchange A[i] with A[r]
4:     return PARTITION(A, p, r)
```

Time Complexity: The worst case time for RANDOMIZED_QUICKSORT is Θ (n^2). The recurrence relation is,

T(n) = max (T(q) + T (n – q − 1) + Θ(n))

where q ranges from 0 to n - 1 because the procedure PARTITION produces two subproblems with total size n - 1.

By using substitution, we can prove this recurrence to get the solution as

T(n) = O (n^2).

The expected running time of RANDOMIZED_QUICKSORT is O(n log n) because if in each level of recursion the split introduced by RANDOMIZED_PARTITION puts any constant fraction of elements on one side of the partition, then the recursion tree has the depth Θ(log n) and O(n) work performed at each level.

Therefore, expected running time is **O (n log n)** when element values are distinct.

3. TimSort

Timsort is a hybrid sorting algorithm, derived from merge sort and insertion sort, designed to perform well on many kinds of real-world data. It iterates over the data collecting the elements into runs and simultaneously merging those runs together into one. It is an adaptive sorting algorithm which needs O (n log n) comparisons to sort an array of n elements. The algorithm finds subsequence of the data that are already ordered and uses that knowledge to sort the remainder more efficiently. This is done by merging an identified subsequence, called a run, with existing runs until certain criteria are fulfilled. It iterates over the data collecting elements into runs, and simultaneously merging those runs together. When there are runs, doing this decreases the total number of comparisons needed to fully sort the list.

3.1 Minrun

The minrun is a size which is determined based on the size of the array. The algorithm selects it so that most runs in a random array are, or become minrun, in length. Merging 2 arrays is more efficient when the number of runs is equal to, or slightly less than, a power of two.

The algorithm chooses minrun from the range 32 to 64 inclusive. It chooses minrun such that the length of the original array, when divided by minrun, is equal to or slightly less than a power of two.

If the length of the run is less than minrun, we calculate the length of that run away from minrun. Using this new number, we grab that many items ahead of the run and perform an insertion sort to create a new run.

So if minrun is 63 and the length of the run is 33, you do 63–33 = 30. You then grab 30 elements from in front of the end of the run, so this is 30 items from run [33] and then perform an insertion sort to create a new run.
After this part is completed, we now have a bunch of sorted runs in a list.
Steps of the algorithm can be written as below:

1. Divide the array into the number of blocks known as run.
2. Consider size of run either 32 or 64(in the below implementation, size of run is 32.)
3. Sort the individual elements of every run one by one using insertion sort.

4. Merge the sorted runs one by one using merge function of merge sort.
5. Double the size of merged sub-arrays after every iteration.

Time complexity: In the worst case, Timsort takes **O(n log n)** comparisons to sort an array of *n* elements. In the best case, which occurs when the input is already sorted, it runs in linear time, implying that it is an adaptive sorting algorithm.

4. Radix Sort

Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping keys by the individual digits which share the same significant position and value. Radix sorts can be implemented to start at either the most significant digit (MSD) or least significant digit (LSD). For example, when sorting the number 1234 into a list, one could start with the 1 or the 4. Radix sort uses counting sort as a subroutine to sort.

The code for radix sort is straightforward. It assumes that each element in the n-element array has d digits, where digit 1 is the lowest-order digit and digit d is the highest-order digit. The algorithm for radix sort can be given as below.

Algorithm: Radix Sort

```
1. Procedure Radix-Sort (list, n)
2.     shift = 1
3.     for loop = 1 to keysize do
4.         for entry = 1 to n do
5.             bucketnumber = (list[entry].key / shift) mod 10
6.             append (bucket[bucketnumber], list[entry])
7.     list = combinebuckets()
8.     shift = shift * 10
```

Time Complexity: Given n d-digit numbers in which each digit can take on up to k possible values, RADIX-SORT sorts these numbers in Θ(d (n+k)) time if the stable sort it uses takes Θ(n+k) time. The analysis of the running time depends on the stable sort used as the intermediate sorting algorithm. When each digit is in the range 0 to k - 1(so that it can take on

k possible values), and k is not too large, counting sort is the obvious choice. Each pass over n d-digit numbers then takes time **Θ(n+k).**

There are d passes, and so the total time for radix sort is **Θ(d (n+k))** .

When d is constant and k = O(n), we can make radix sort run in linear time.

5. Description of the experiment

The experiment was carried in order to analyse the efficiency of QuickSort, TimSort and RadixSort. Quicksort has been implemented using different pivot selection methods namely using last element as pivot, random pivot selection and median of medians. To scrutinize the algorithms thoroughly, different input lists are generated as below:

1) Unique random integers list
2) Few unique random integers list
3) Nearly sorted (first half sorted, second half random)
4) Nearly sorted (second half sorted, first half random)
5) Reverse integers list (sorted in decreasing order)

Additionally, the sizes for each input list are varied as 1000, 10000, 100000, 1000000. The input range for all lists has been kept from 1 to 2000000. These inputs are generated and saved in a .txt file. These generated files served as input for the main sorting code. The output or the sorted array after the execution of algorithm is written in the output.txt file. To verify the correctness of the sorted elements generated by each algorithm, python's inbuilt sort is used on the original input list and the output is compared with the output of the sorting algorithms.

To represent the performance analysis of algorithms, five different graphs are plotted taking into account five different input lists mentioned above. All 6 algorithms i.e. Quicksort (naïve, randomized, MoM5 and MoM7) , TimSort and RadixSort have been run on all input types of different sizes to draw the results. To eliminate random influences within each execution the average value of 10 execution results to evaluate the performance of each algorithm is taken.

6. Results and Analysis

Below figures shows the comparison between these 6 algorithms based on the input types.

1) Random integers (few unique)

Random - Few Unique	quicksort naïve	randomized_quicksort	quicksort using mom group5	quicksort using mom group7	timsort	radix sort
1000	0.004001379	0.007994413	0.124992371	0.226868629	0.003997564	0.652623653
10000	0.071982622	0.067980528	9.558454037	16.72139549	0.062514305	6.75064826
100000	0.707835197	0.519678593	1712.999288	2919.487107	0.7318151	68.69365931
1000000	5.702554703	6.672992706	No results	No results	9.689579964	844.8019216

Table 1: Execution Time for random few unique input

Figure 1: Comparison of algorithms for random few unique input

2) Random integers (unique)

Random - Unique	quicksort naïve	randomized_quicksort	quicksort using mom group5	quicksort using mom group7	timsort	radix sort
1000	0.003999472	0.015626907	0.117949963	0.142914772	0.003999233	0.67063427
10000	0.035992384	0.06249547	13.71775818	17.64686537	0.067981482	7.314804316
100000	0.471880198	0.579852581	2481.352149	2910.263069	0.767790794	73.87229705
1000000	5.954472542	6.546931744	No results	No results	9.645593882	869.8118033

Table 2: Execution Time for random unique input

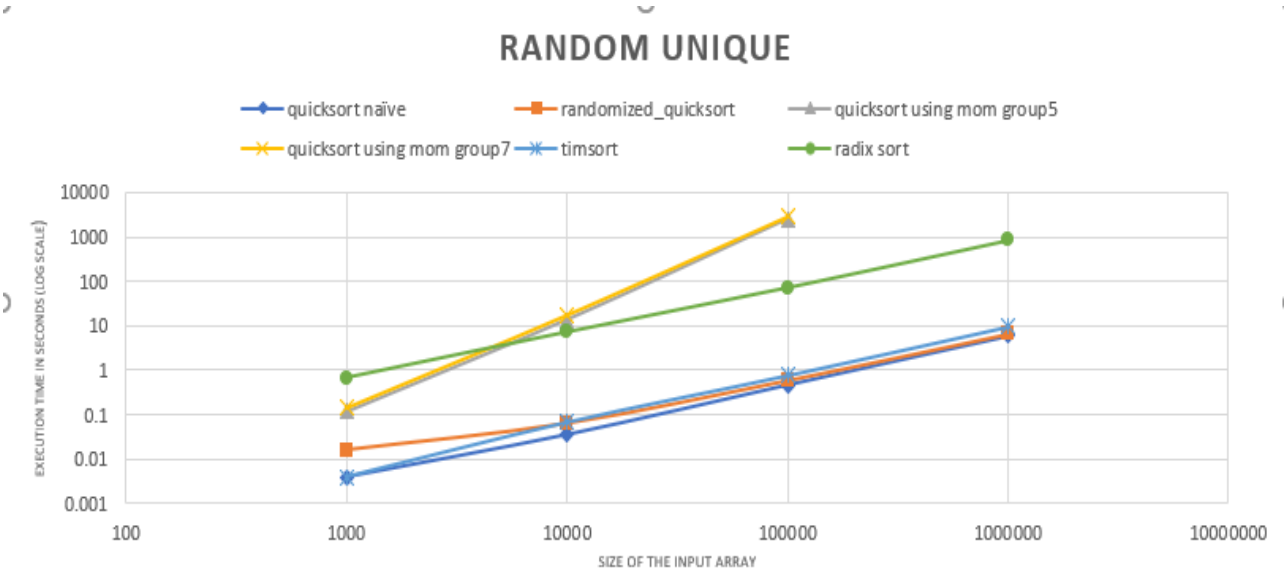

Figure 2: Comparison of algorithms for random unique input

3) Nearly sorted integers (first half sorted)

first half sorted	quicksort naïve	randomized_quicksort	quicksort using mom group5	quicksort using mom group7	timsort	radix sort
1000	0.00399971	0.003999472	0.148034573	0.09198761	0.003998518	0.669633627
10000	0.143963337	0.052139282	17.78869009	6.906236172	0.047980309	7.074960947
100000	0.475896597	0.495893002	1961.117919	2829.541473	0.607847691	69.78997302
1000000	6.038489342	6.345487356	No results	No results	7.858047962	789.2753203

Table 3: Execution Time for first half sorted input

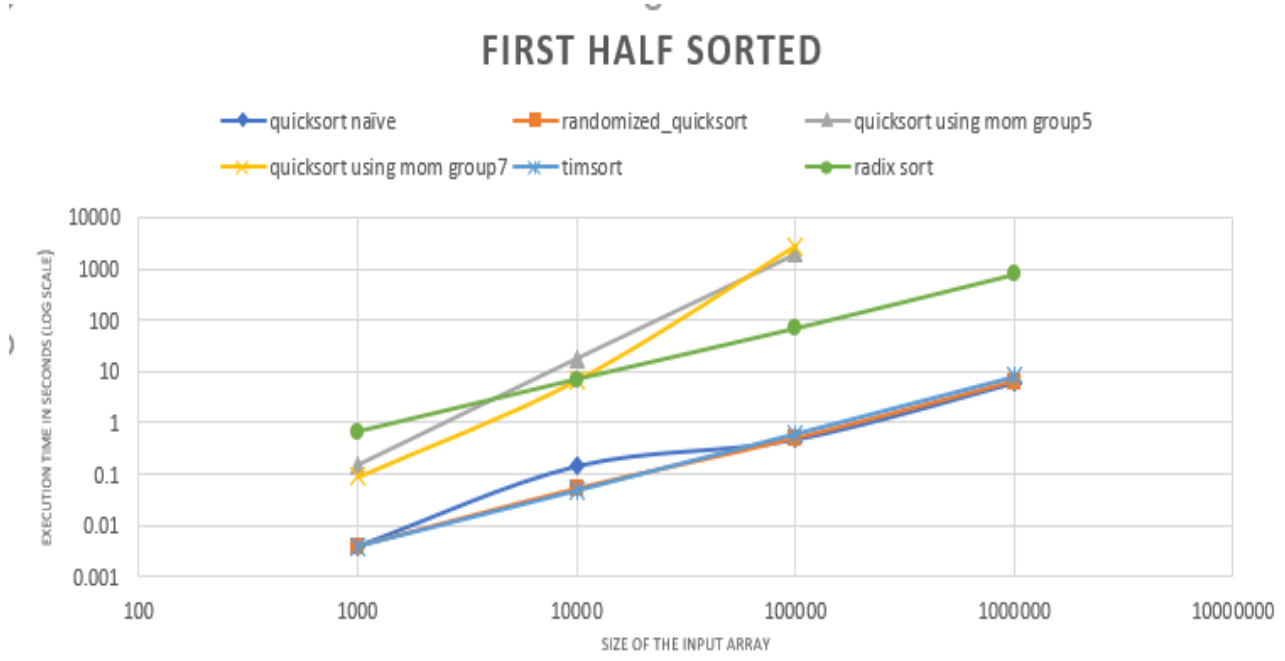

Figure 3: Comparison of algorithms for first half sorted input

4) Nearly sorted integers (second half sorted)

second half sorted	quicksort naïve	randomized_quicksort	quicksort using mom group5	quicksort using mom group7	timsort	radix sort
1000	0.055985689	0.015623093	0.091945648	0.059964657	0.003998041	0.659845352
10000	No results	0.055980444	17.49681234	6.446355343	0.076956511	7.16220808
100000	No results	0.603847027	2128.27177	2817.100644	0.977441788	81.34346104
1000000	No results	6.115347862	No results	No results	7.947479725	822.9443438

Table 4: Execution Time for second half sorted input

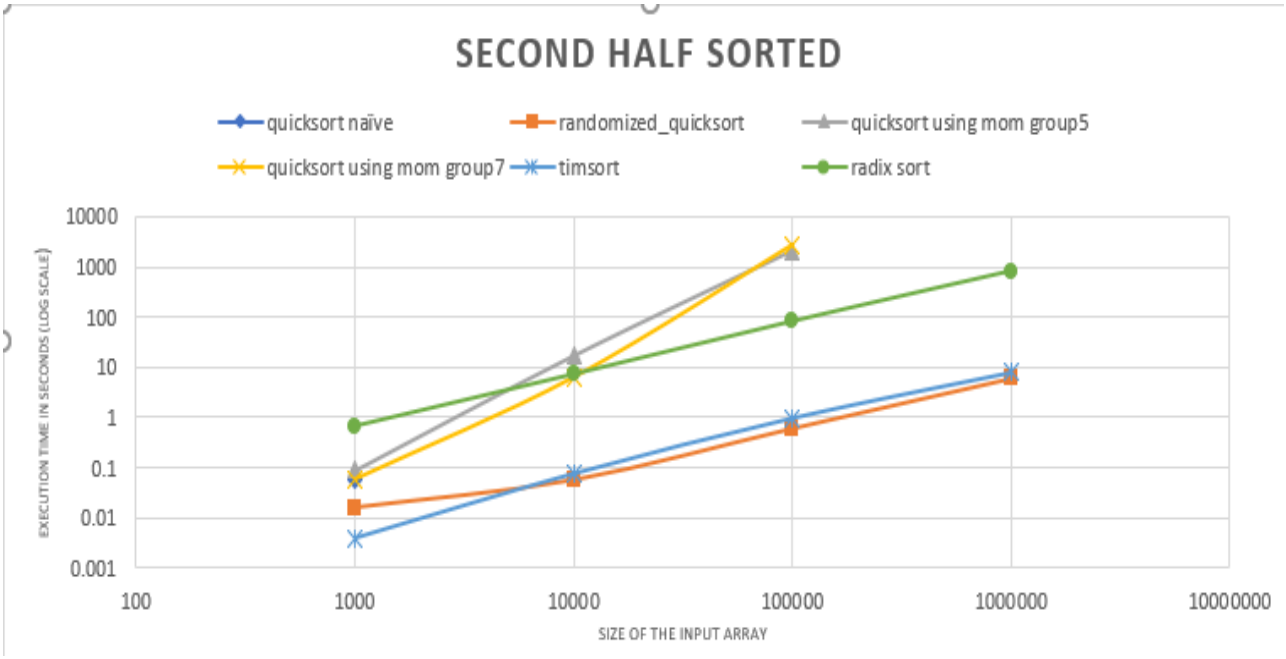

Figure 4: Comparison of algorithms for second half sorted input

5) Reversed integers (arranged in descending order)

Reversed	quicksort naïve	randomized_quicksort	quicksort using mom group5	quicksort using mom group7	timsort	radix sort
1000	0.109351873	0.003999949	0.114932775	0.067981005	0.005998373	0.939760447
10000	No results	0.059984446	17.29307079	6.538363934	0.076973915	8.033969402
100000	No results	0.735860825	2287.360619	2600.847338	0.846536636	70.68873262
1000000	No results	5.544569492	No results	No results	9.617511749	800.774735

Table 5: Execution Time for reversed input

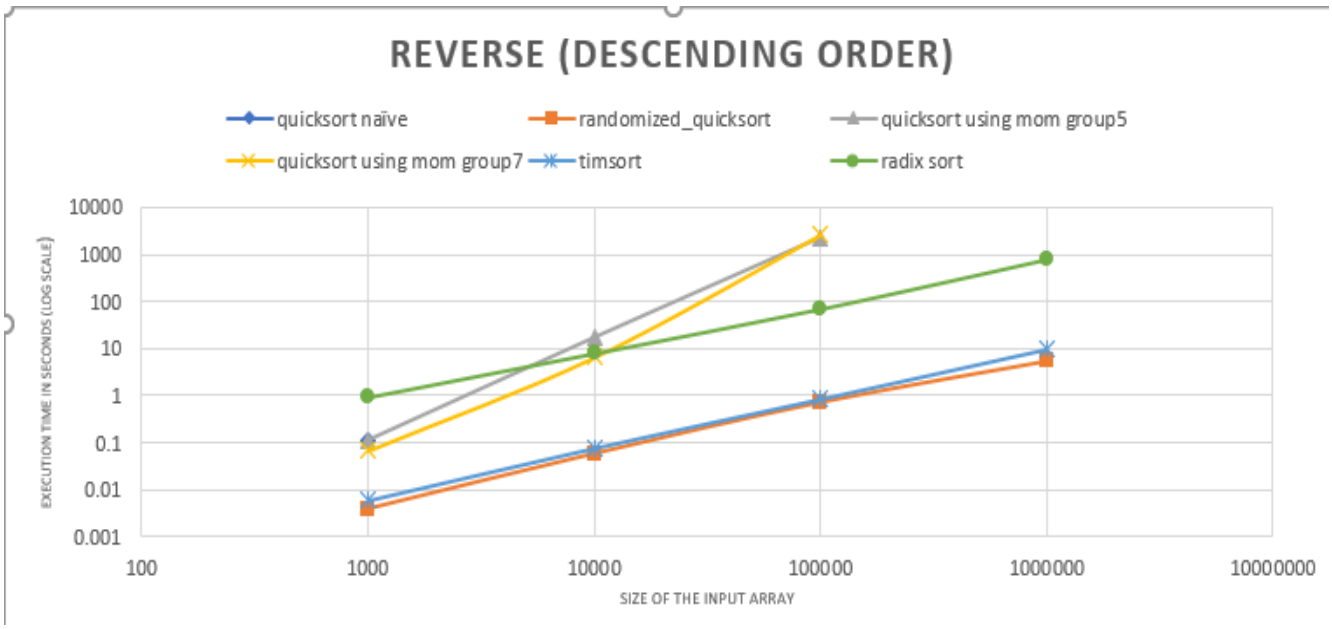

Figure 5: Comparison of algorithms for reversed input

Performance Analysis: As can be observed from above, in case of random integers for a small set of 1000 datasize timsort and quicksort with naïve implementation takes almost the same time, while it can be said that randomized quicksort seems to be bit slower than both. However, as we increase the datasize, the time taken by quicksort, with naïve and randomized implementation, is almost equal and the timsort seems to be a bit slower than both.

For the list of half sorted integers (first half sorted and other half random), again the three algorithms, timsort and quicksort with naïve and randomized selection seems to be working at the same pace, giving nearly the same performance.

However, for the other list of sorted integers i.e. second half sorted list Naïve quicksort doesn't give any results for a dataset 10000 and larger. The reason being the pivot selection method. As it always selects the last element as the pivot, in a series of already sorted elements it reaches the recursion limit. Even after increasing the recursion limit of the program to 10000, there was no output. The same performance is observed in the case of input with elements arranged in reverse order suggesting us that quicksort with naïve implementation is perhaps not a good idea when the input array is already in some kind of sorting order.

For almost all types of inputs, quicksort with randomized selection and timsort has performed consistently, giving us their average time complexity, however, for large datasets timsort has proved to be a bit expensive.

As far as other versions of quicksort is concerned i.e. median of medians with group 5 and 7, they are rather costly than the other two versions because of the overhead of pivot selection. For the input size of 100000, the algorithm takes long time on all input types. Additionally, when the size of 1000000 is reached, no result is obtained, suggesting us that quicksort with median of medians is certainly not a good option for larger datasets.

Radix sort is a non-comparison based algorithms and thus its output does not only depends on the size of the input array but also on the highest number of digits among the input key set (d) and the constant k (10 in our case since we are dealing with decimal numbers) and thus its results has always been greater than the quicksort (naïve and randomized) and timsort. However, for large datasets it is still better than the other two versions of quicksort i.e. quicksort with median of medians group 5 and group 7.

7.Conclusion

The project is about comparing and analysing the performance of quicksort, timsort and radix sort on different input types and sizes. Quicksort is implemented with four different pivot selection method. As can be observed from the results, quicksort with randomized pivot selection and timsort is undoubtfully a good choice on almost all input types. However, timsort might prove to be more advantageous in case of already sorted (nearly sorted) input since we have seen a surge in its performance when the input was half sorted. Quicksort with median pivot selection and radix sort might not be a good option as they seem to take a higher simulation time, especially with the large datasets.

References

1) https://www.interviewbit.com/tutorial/quicksort-algorithm/
2) https://en.wikipedia.org/wiki/Median_of_medians
3) https://en.wikipedia.org/wiki/Timsort
4) https://en.wikipedia.org/wiki/Radix_sort
5) Introduction to Algorithms Third Edition by CLRS