# ITCS 6114/8114: Algorithm & Data Structures
# Summer 2018

**Project Report 1:**
Comparison-based Sorting Algorithms

**Programming Language used**
Python


**<u>Team Members:</u>**

Shashikant Jaiswal (801053461)
Shaily Barjatya (801054460)

**PART 1**

**A) <u>Input Array → Randomly Generated numbers</u>**

**Results Table**

(1) Table showing time taken by each sort to sort numbers for n = 500, 1000, 2000, 4000, 5000, 10000, 20000, 30000, 40000 and 50,000 for different randomized input ranges.

Runtimes for Randomized Array

| #Input | #Round | Insertion Sort | Merge Sort | In-Place Quick Sort | Modified Quick Sort |
|---|---|---|---|---|---|
| N=500 | Round 1 | 9.14 | 2.501 | 1.571 | 0.692 |
| | Round 2 | 8.512 | 2.318 | 0.923 | 0.779 |
| | Round 3 | 8.162 | 2.198 | 0.852 | 0.738 |
| Average | | 8.604666667 | 2.339 | 1.115333333 | 0.736333333 |
| | | | | | |
| N=1000 | Round 1 | 31.988 | 5.257 | 2.307 | 1.729 |
| | Round 2 | 32.88 | 5.93 | 2.961 | 1.541 |
| | Round 3 | 32.06 | 4.663 | 2.596 | 2.376 |
| Average | | 32.30933333 | 5.283333333 | 2.621333333 | 1.882 |
| | | | | | |
| N=2000 | Round 1 | 135.52 | 10.105 | 4.836 | 3.73 |
| | Round 2 | 129.849 | 11.112 | 5.573 | 5.519 |
| | Round 3 | 133.902 | 9.584 | 3.982 | 3.67 |
| Average | | 133.0903333 | 10.267 | 4.797 | 4.306333333 |
| | | | | | |
| N=4000 | Round 1 | 532.672 | 27.414 | 10.684 | 8.159 |
| | Round 2 | 513.074 | 21.553 | 9.816 | 8.285 |
| | Round 3 | 527.215 | 21.303 | 11.908 | 7.492 |
| Average | | 524.3203333 | 23.42333333 | 10.80266667 | 7.978666667 |
| | | | | | |
| N=5000 | Round 1 | 829.566 | 27.886 | 12.771 | 10.707 |
| | Round 2 | 842.541 | 27.48 | 12.491 | 9.676 |
| | Round 3 | 810.067 | 27.161 | 15.338 | 10.311 |
| Average | | 827.3913333 | 27.509 | 13.53333333 | 10.23133333 |
| | | | | | |
| N=10000 | Round 1 | 3350.735 | 56.941 | 25.434 | 22.334 |
| | Round 2 | 3299.884 | 56.621 | 24.647 | 24.578 |
| | Round 3 | 3339.337 | 58.677 | 24.796 | 23.283 |
| Average | | 3329.985333 | 57.413 | 24.959 | 23.39833333 |

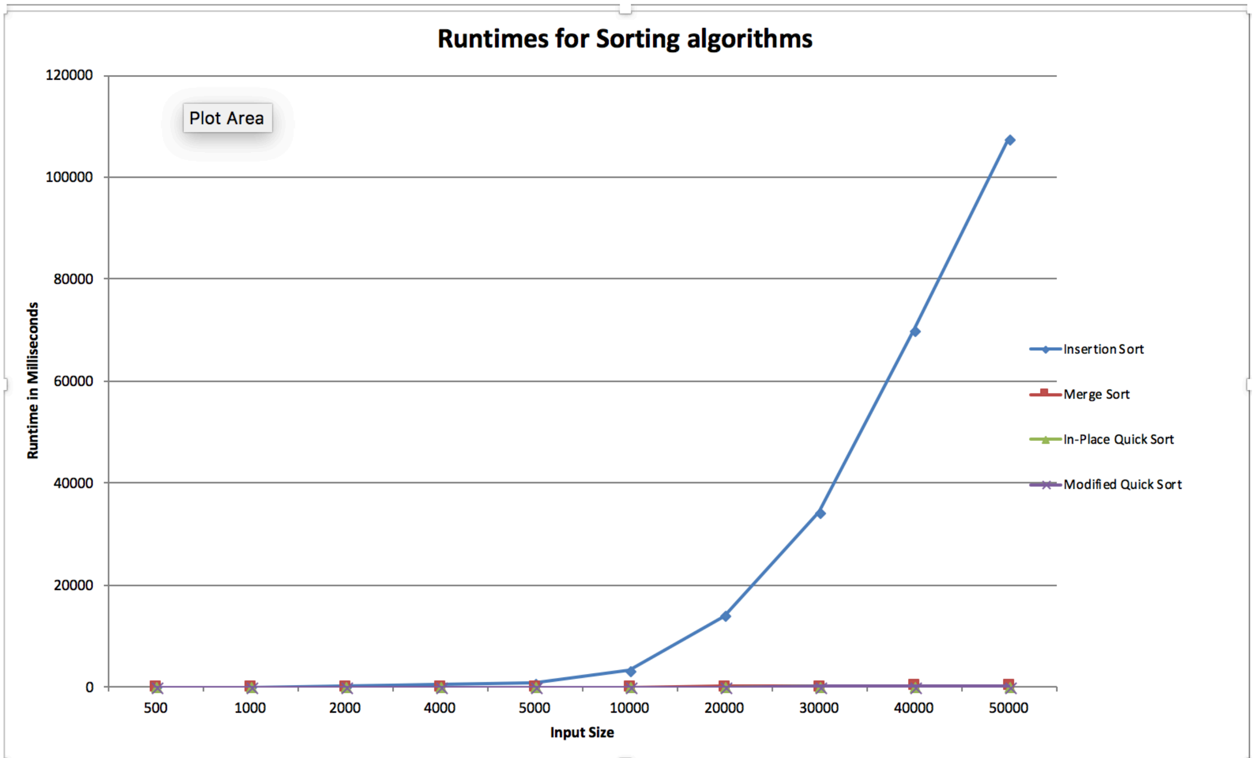
| | | | | | |
|---|---|---|---|---|---|
| | | | | | |
| N=20000 | Round 1 | 14166.676 | 126.681 | 59.887 | 46.049 |
| | Round 2 | 13777.151 | 121.967 | 57.168 | 53.473 |
| | Round 3 | 14066.201 | 116.817 | 53.975 | 45.75 |
| Average | | 14003.34267 | 121.8216667 | 57.01 | 48.424 |
| | | | | | |
| N=30000 | Round 1 | 33676.353 | 192.611 | 84.545 | 69.486 |
| | Round 2 | 33433.072 | 190.217 | 123.839 | 154.174 |
| | Round 3 | 35813.97 | 188.363 | 92.753 | 73.632 |
| Average | | 34307.79833 | 190.397 | 100.379 | 99.09733333 |
| | | | | | |
| N=40000 | Round 1 | 66148.726 | 266.168 | 138.297 | 98.327 |
| | Round 2 | 67102.526 | 343.562 | 135.754 | 115.512 |
| | Round 3 | 76607.43 | 307.769 | 143.217 | 114.185 |
| Average | | 69952.894 | 305.833 | 139.0893333 | 109.3413333 |
| | | | | | |
| N=50000 | Round 1 | 110863.227 | 323.574 | 147.196 | 131.553 |
| | Round 2 | 115258.345 | 391.118 | 181.808 | 144.927 |
| | Round 3 | 96266.963 | 442.076 | 158.095 | 141.718 |
| Average | | 107462.845 | 385.5893333 | 162.3663333 | 139.3993333 |

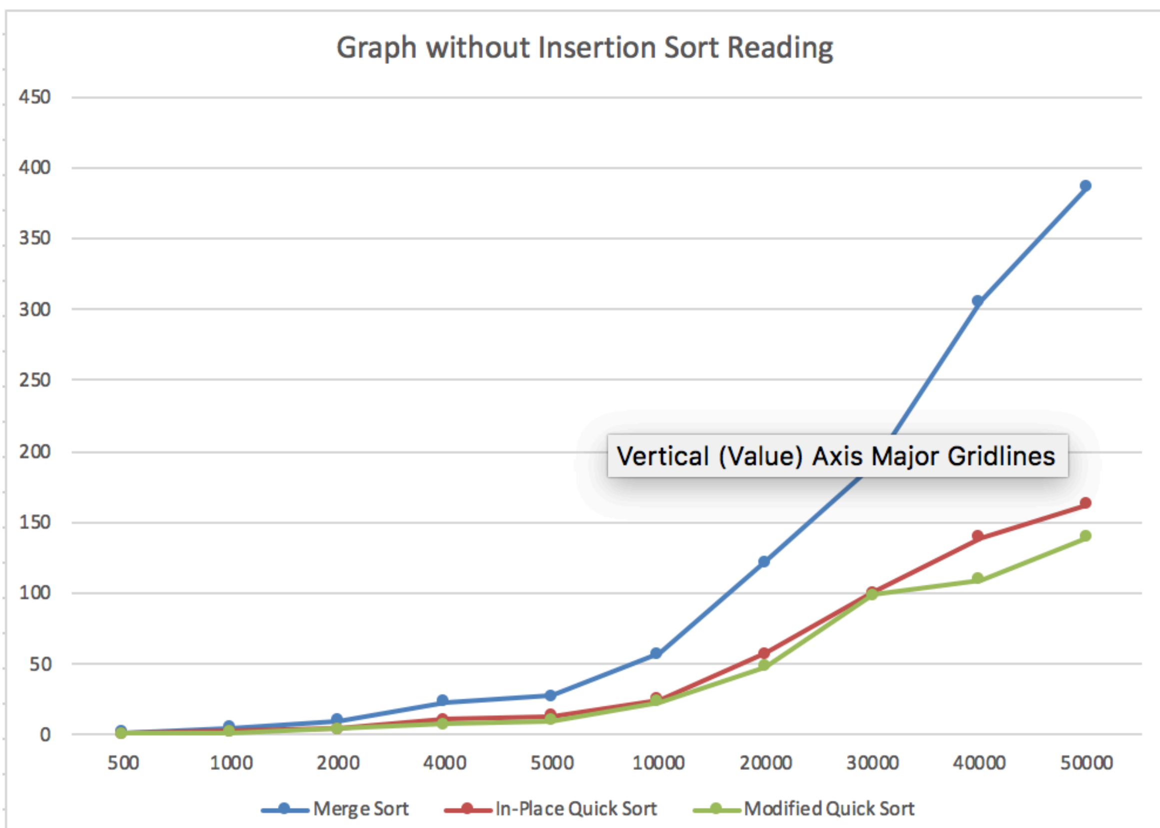
(2) Average time taken by each sort for randomized input ranges.

| | Insertion Sort | Merge Sort | In-Place Quick Sort | Modified Quick Sort |
|---|---|---|---|---|
| 500 | 8.604666667 | 2.339 | 1.115333333 | 0.736333333 |
| 1000 | 32.30933333 | 5.283333333 | 2.621333333 | 1.882 |
| 2000 | 133.0903333 | 10.267 | 4.797 | 4.306333333 |
| 4000 | 524.3203333 | 23.42333333 | 10.80266667 | 7.978666667 |
| 5000 | 827.3913333 | 27.509 | 13.53333333 | 10.23133333 |
| 10000 | 3329.985333 | 57.413 | 24.959 | 23.39833333 |
| 20000 | 14003.34267 | 121.8216667 | 57.01 | 48.424 |
| 30000 | 34307.79833 | 190.397 | 100.379 | 99.09733333 |
| 40000 | 69952.894 | 305.833 | 139.0893333 | 109.3413333 |
| 50000 | 107462.845 | 385.5893333 | 162.3663333 | 139.3993333 |

## Graph

Graph plotted for showing time performance of sorting algorithms



## Observation

It can be observed from the above graph that of all the sorting techniques, Insertion sort is the worst choice for sorting randomized array for input size > 10,000 as its runtime is increasing exponentially for input size > 10,000. Whereas, other sorting techniques exhibit almost the same performance for all input sizes. However, according to the average runtime recorded in the above table we can conclude that of all the sorting techniques, Modified Quicksort is the fastest of all, followed by In-place Quicksort, Merge sort and then Insertion sort for randomized arrays.

## Output Result Screenshots

Output result of all the sorting algorithms for different input randomized array for n=50000

```
*********INSERTION SORT**********

Time elapsed in milliseconds after Insertion sort is :
110863.227



*********MERGE SORT**********


Time elapsed in milliseconds after Merge-sort is :
323.574



*********IN-PLACE QUICKSORT**********


Time elapsed in milliseconds after Quicksort is :
147.196



*********MODIFIED QUICKSORT**********


Time elapsed in milliseconds after Modified Quicksort is :
131.583
```

```
*********INSERTION SORT**********

Time elapsed in milliseconds after Insertion sort is :
115258.345


*********MERGE SORT**********


Time elapsed in milliseconds after Merge-sort is :
391.118


*********IN-PLACE QUICKSORT**********


Time elapsed in milliseconds after Quicksort is :
181.808


*********MODIFIED QUICKSORT**********


Time elapsed in milliseconds after Modified Quicksort is :
144.927
```

```
*********INSERTION SORT**********

Time elapsed in milliseconds after Insertion sort is :
96266.963


*********MERGE SORT**********


Time elapsed in milliseconds after Merge-sort is :
442.076


*********IN-PLACE QUICKSORT**********


Time elapsed in milliseconds after Quicksort is :
158.095


*********MODIFIED QUICKSORT**********


Time elapsed in milliseconds after Modified Quicksort is :
141.718
```

## PART 2

**B) Input Array → Sorted Array**

Note: Excluding In-place Quicksort as it is going in infinite loop for input size >=995. So, runtime values could not be captured. (Refer Appendix A)

**Results Table**

(1) Table showing time taken by each sort to sort numbers for n = 500, 1000, 2000, 4000, 5000, 10000, 20000, 30000, 40000 and 50,000 for different input ranges of sorted array.

**Runtimes for Sorted Array**

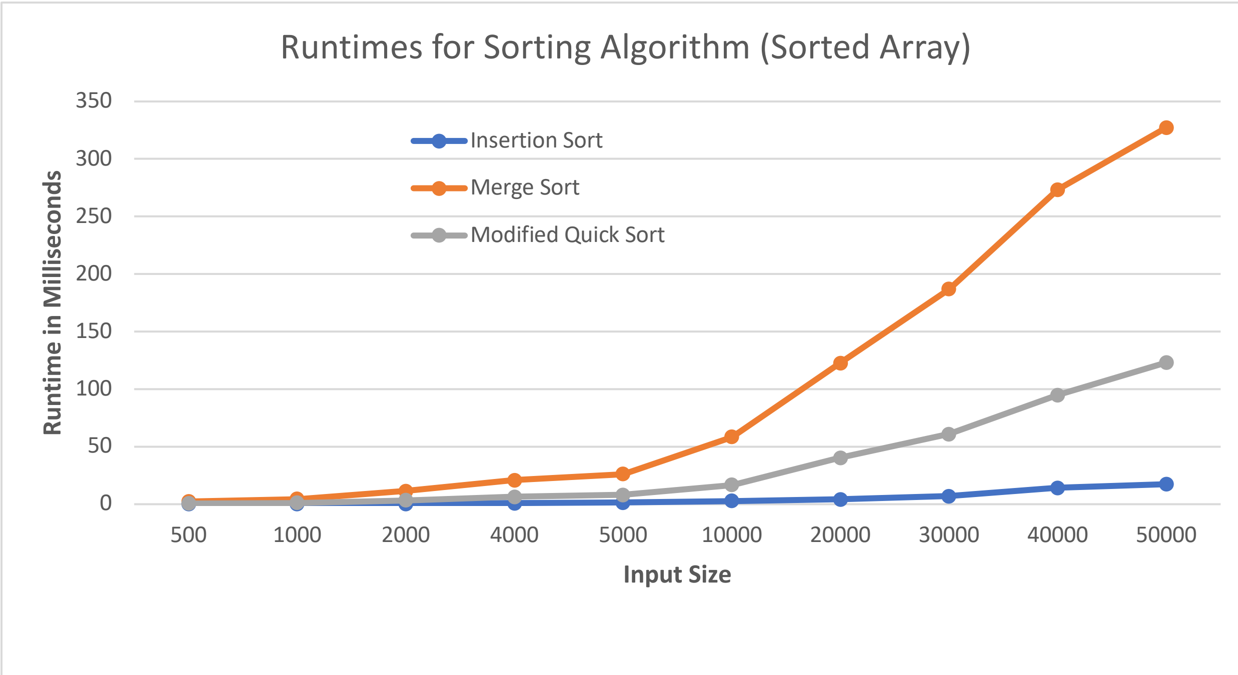
| #Input | #Round | Insertion Sort | Merge Sort | In-Place Quick Sort | Modified Quick Sort |
|---|---|---|---|---|---|
| N=500 | Round 1 | 0.111 | 2.534 | 21.222 | 0.585 |
| | Round 2 | 0.199 | 2.247 | 21.439 | 0.61 |
| | Round 3 | 0.103 | 2.245 | 21.783 | 0.552 |
| **Average** | | 0.137666667 | 2.342 | 21.48133333 | 0.582333333 |
| | | | | | |
| N=1000 | Round 1 | 0.18 | 4.408 | | 1.289 |
| | Round 2 | 0.28 | 4.347 | | 1.108 |
| | Round 3 | 0.25 | 4.879 | | 1.261 |
| **Average** | | 0.236666667 | 4.544666667 | 0 | 1.219333333 |
| | | | | | |
| N=2000 | Round 1 | 0.377 | 12.741 | | 2.825 |
| | Round 2 | 0.418 | 11.452 | | 2.767 |
| | Round 3 | 0.504 | 10.435 | | 4.24 |
| **Average** | | 0.433 | 11.54266667 | 0 | 3.277333333 |
| | | | | | |
| N=4000 | Round 1 | 0.937 | 21.627 | | 6.27 |
| | Round 2 | 0.858 | 21.624 | | 7.125 |
| | Round 3 | 0.725 | 19.714 | | 5.678 |
| **Average** | | 0.84 | 20.98833333 | 0 | 6.357666667 |
| | | | | | |
| N=5000 | Round 1 | 0.99 | 26.79 | | 8.887 |
| | Round 2 | 1.009 | 26.602 | | 8.404 |
| | Round 3 | 2.036 | 25.106 | | 6.576 |
| **Average** | | 1.345 | 26.166 | 0 | 7.955666667 |
| | | | | | |
| N=10000 | Round 1 | 1.979 | 56.055 | | 17.324 |
| | Round 2 | 1.882 | 67.344 | | 14.887 |

| | | | | | |
|---|---|---|---|---|---|
| | Round 3 | 4.296 | 51.842 | | 17.844 |
| **Average** | | 2.719 | 58.41366667 | 0 | 16.685 |
| | | | | | |
| N=20000 | Round 1 | 3.984 | 129.359 | | 33.352 |
| | Round 2 | 4.874 | 120.693 | | 46.059 |
| | Round 3 | 3.997 | 118.114 | | 41.849 |
| **Average** | | 4.285 | 122.722 | 0 | 40.42 |
| | | | | | |
| N=30000 | Round 1 | 6.052 | 192.633 | | 57.905 |
| | Round 2 | 7.649 | 180.151 | | 63.597 |
| | Round 3 | 7.091 | 187.74 | | 60.968 |
| **Average** | | 6.930666667 | 186.8413333 | 0 | 60.82333333 |
| | | | | | |
| N=40000 | Round 1 | 14.292 | 272.819 | | 108.046 |
| | Round 2 | 13.787 | 292.684 | | 84.309 |
| | Round 3 | 14.286 | 255.006 | | 92.014 |
| **Average** | | 14.12166667 | 273.503 | 0 | 94.78966667 |
| | | | | | |
| N=50000 | Round 1 | 20.334 | 308.117 | | 109.923 |
| | Round 2 | 11.959 | 329.205 | | 136.004 |
| | Round 3 | 19.898 | 344.49 | | 123.416 |
| **Average** | | 17.397 | 327.2706667 | 0 | 123.1143333 |

(2) Average time taken by each sort for input ranges of sorted array.

| | **Insertion Sort** | **Merge Sort** | **In-Place Quick Sort** | **Modified Quick Sort** |
|---|---|---|---|---|
| 500 | 0.137666667 | 2.342 | 21.48133333 | 0.582333333 |
| 1000 | 0.236666667 | 4.544666667 | | 1.219333333 |
| 2000 | 0.433 | 11.54266667 | | 3.277333333 |
| 4000 | 0.84 | 20.98833333 | | 6.357666667 |
| 5000 | 1.345 | 26.166 | | 7.955666667 |
| 10000 | 2.719 | 58.41366667 | | 16.685 |
| 20000 | 4.285 | 122.722 | | 40.42 |
| 30000 | 6.930666667 | 186.8413333 | | 60.82333333 |
| 40000 | 14.12166667 | 273.503 | | 94.78966667 |
| 50000 | 17.397 | 327.2706667 | | 123.1143333 |

## Graph




## Observation

It is evident from above graph that of all the sorting techniques, Insertion sort is the fastest of all, followed by Modified Quicksort, and then Mergesort for sorted arrays. The runtimes for insertion sort are more or less constant for all the input sizes and the runtimes for merge sort increase drastically as compared to insertion sort and modified quicksort for input size > 10,000. (In-place quicksort is not being considered as runtimes could not be captured due to issue referred in Appendix A)

### C) Input Array → Reversely Sorted Array

Note: Excluding In-place Quicksort as it is going in infinite loop for input size >=995. So, runtime values could not be captured. (Refer Appendix A)

## Results Table

(1) Table showing time taken by each sort to sort numbers for n = 500, 1000, 2000, 4000, 5000, 10000, 20000, 30000, 40000 and 50,000 for different input ranges of reverse sorted array.

**Runtimes for Reversed Sorted Array**

| #Input | #Round | Insertion Sort | Merge Sort | In-Place Quick Sort | Modified Quick Sort |
|---|---|---|---|---|---|
| N=500 | Round 1 | 16.851 | 2.408 | 14.567 | 1.161 |

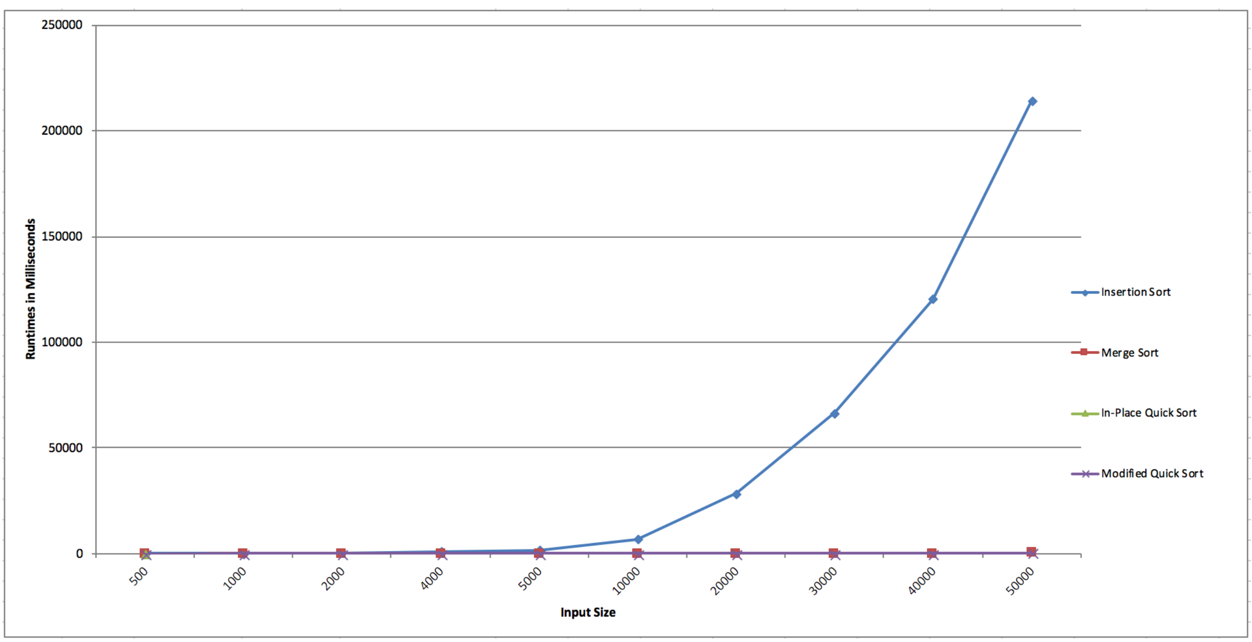
| | | | | | |
|---|---|---|---|---|---|
| | Round 2 | 16.389 | 2.258 | 13.755 | 1.242 |
| | Round 3 | 15.081 | 2.129 | 13.594 | 1.082 |
| **Average** | | 16.107 | 2.265 | 13.972 | 1.161666667 |
| | | | | | |
| N=1000 | Round 1 | 65.99 | 4.552 | | 2.727 |
| | Round 2 | 66.351 | 5.206 | | 2.437 |
| | Round 3 | 62.059 | 4.366 | | 2.553 |
| **Average** | | 64.8 | 4.708 | 0 | 2.572333333 |
| | | | | | |
| N=2000 | Round 1 | 257.851 | 9.442 | | 5.818 |
| | Round 2 | 253.345 | 9.899 | | 5.654 |
| | Round 3 | 262.49 | 9.642 | | 5.333 |
| **Average** | | 257.8953333 | 9.661 | 0 | 5.601666667 |
| | | | | | |
| N=4000 | Round 1 | 1025.84 | 26.425 | | 12.362 |
| | Round 2 | 1024.904 | 19.931 | | 12.569 |
| | Round 3 | 1033.517 | 19.547 | | 13.915 |
| **Average** | | 1028.087 | 21.96766667 | 0 | 12.94866667 |
| | | | | | |
| N=5000 | Round 1 | 1615.46 | 24.972 | | 15.841 |
| | Round 2 | 1658.826 | 26.106 | | 14.093 |
| | Round 3 | 1646.245 | 25.2 | | 16.321 |
| **Average** | | 1640.177 | 25.426 | 0 | 15.41833333 |
| | | | | | |
| N=10000 | Round 1 | 6575.451 | 53.818 | | 33.89 |
| | Round 2 | 6918.287 | 51.94 | | 36.817 |
| | Round 3 | 7377.16 | 53.164 | | 34.607 |
| **Average** | | 6956.966 | 52.974 | 0 | 35.10466667 |
| | | | | | |
| N=20000 | Round 1 | 28441.174 | 125.373 | | 78.961 |
| | Round 2 | 28644.696 | 112.031 | | 76.24 |
| | Round 3 | 28680.259 | 118.386 | | 75.213 |
| **Average** | | 28588.70967 | 118.5966667 | 0 | 76.80466667 |
| | | | | | |
| N=30000 | Round 1 | 69720.443 | 202.267 | | 121.258 |
| | Round 2 | 64577.633 | 172.102 | | 126.988 |
| | Round 3 | 65607.947 | 168.911 | | 148.821 |
| **Average** | | 66635.341 | 181.0933333 | 0 | 132.3556667 |
| | | | | | |
| N=40000 | Round 1 | 118285.649 | 250.17 | | 171.129 |
| | Round 2 | 125743.62 | 241.17 | | 171.478 |
| | Round 3 | 117388.4 | 262.151 | | 190.866 |
| **Average** | | 120472.5563 | 251.1636667 | 0 | 177.8243333 |
| | | | | | |

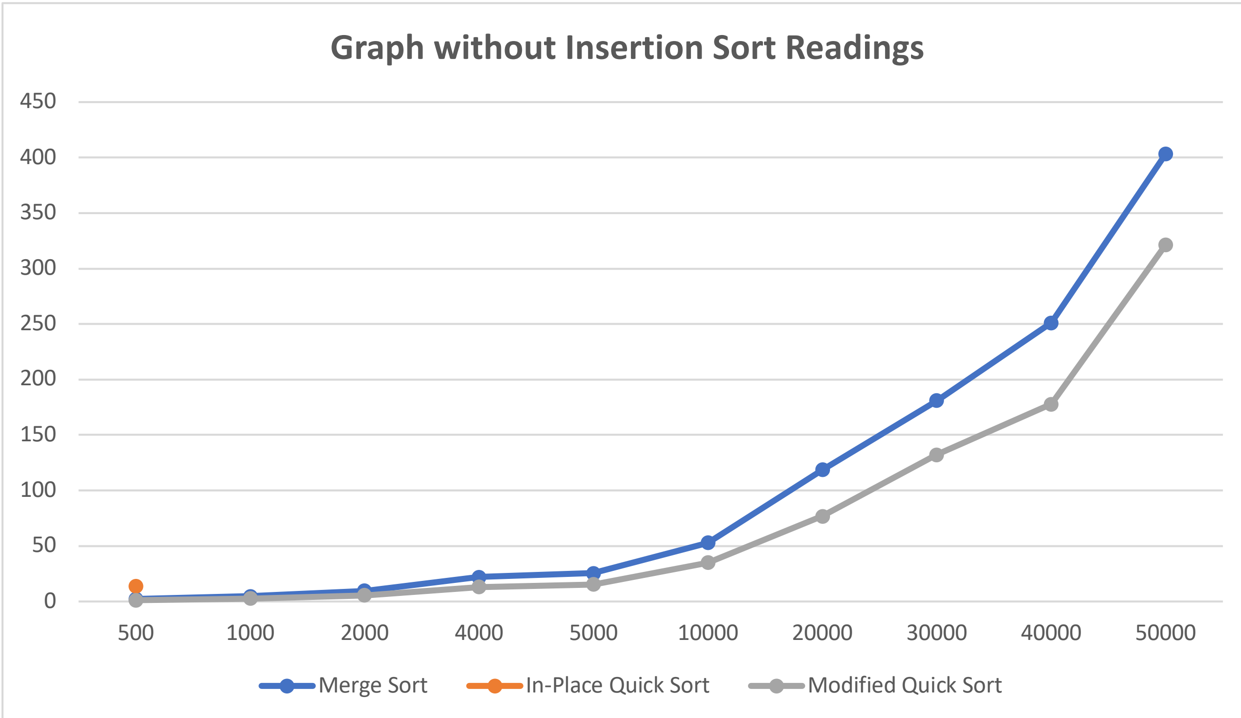
| | | | | | |
|---|---|---|---|---|---|
| N=50000 | Round 1 | 208411.559 | 305.857 | | 233.917 |
| | Round 2 | 210996.431 | 394.536 | | 271.374 |
| | Round 3 | 222613.828 | 509.411 | | 459.068 |
| **Average** | | 214007.2727 | 403.268 | 0 | 321.453 |

(2) Average time taken by each sort for input ranges of reverse sorted array.

| | Insertion Sort | Merge Sort | In-Place Quick Sort | Modified Quick Sort |
|---|---|---|---|---|
| 500 | 16.107 | 2.265 | 13.972 | 1.161666667 |
| 1000 | 64.8 | 4.708 | | 2.572333333 |
| 2000 | 257.8953333 | 9.661 | | 5.601666667 |
| 4000 | 1028.087 | 21.96766667 | | 12.94866667 |
| 5000 | 1640.177 | 25.426 | | 15.41833333 |
| 10000 | 6956.966 | 52.974 | | 35.10466667 |
| 20000 | 28588.70967 | 118.5966667 | | 76.80466667 |
| 30000 | 66635.341 | 181.0933333 | | 132.3556667 |
| 40000 | 120472.5563 | 251.1636667 | | 177.8243333 |
| 50000 | 214007.2727 | 403.268 | | 321.453 |

## Graph





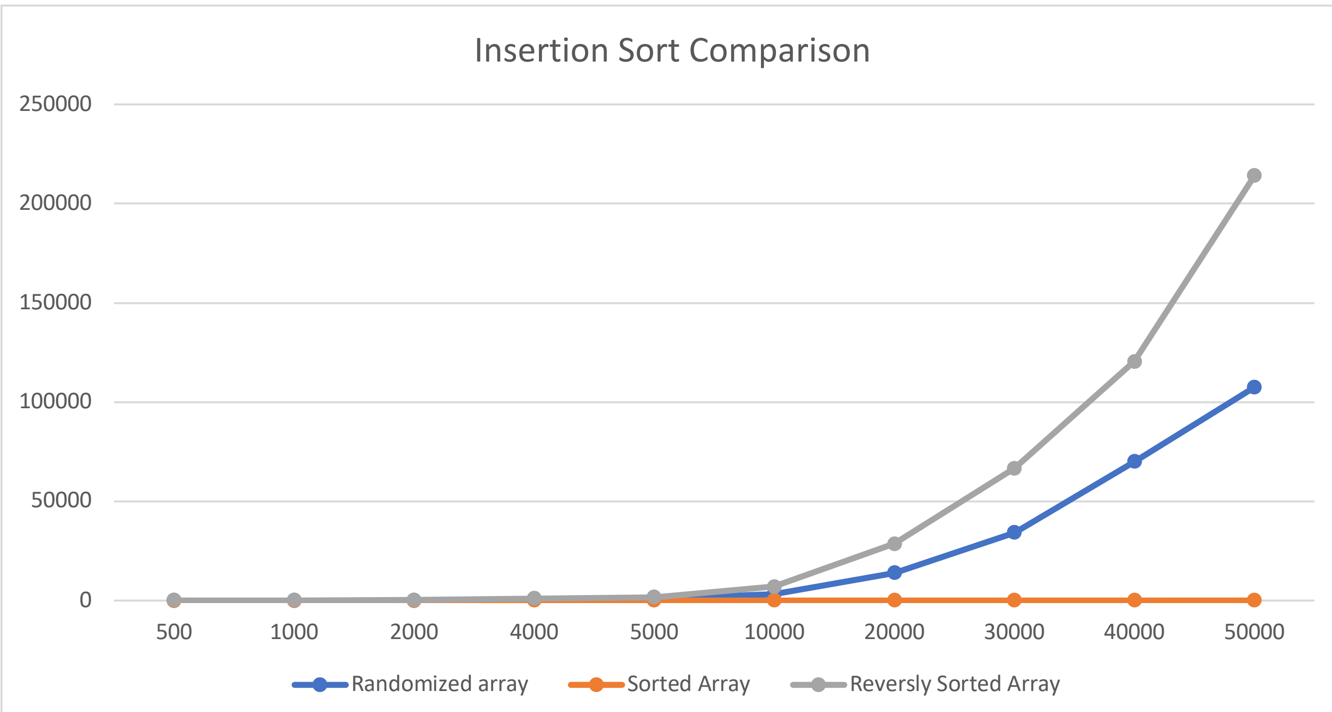

**Observation**

It can be observed from the above graph that of all the sorting techniques, Insertion sort is the worst choice for sorting reversely sorted array. The runtime for Insertion sort is increasing exponentially for input size > 10,000, whereas other sorting techniques exhibit almost the same performance for all input sizes. However, according to the average runtime recorded in the above table we can conclude that Modified Quicksort is the slightly better than Mergesort for reversely sorted array.
(In-place quicksort is not being considered as runtimes could not be captured due to issue referred in Appendix A)

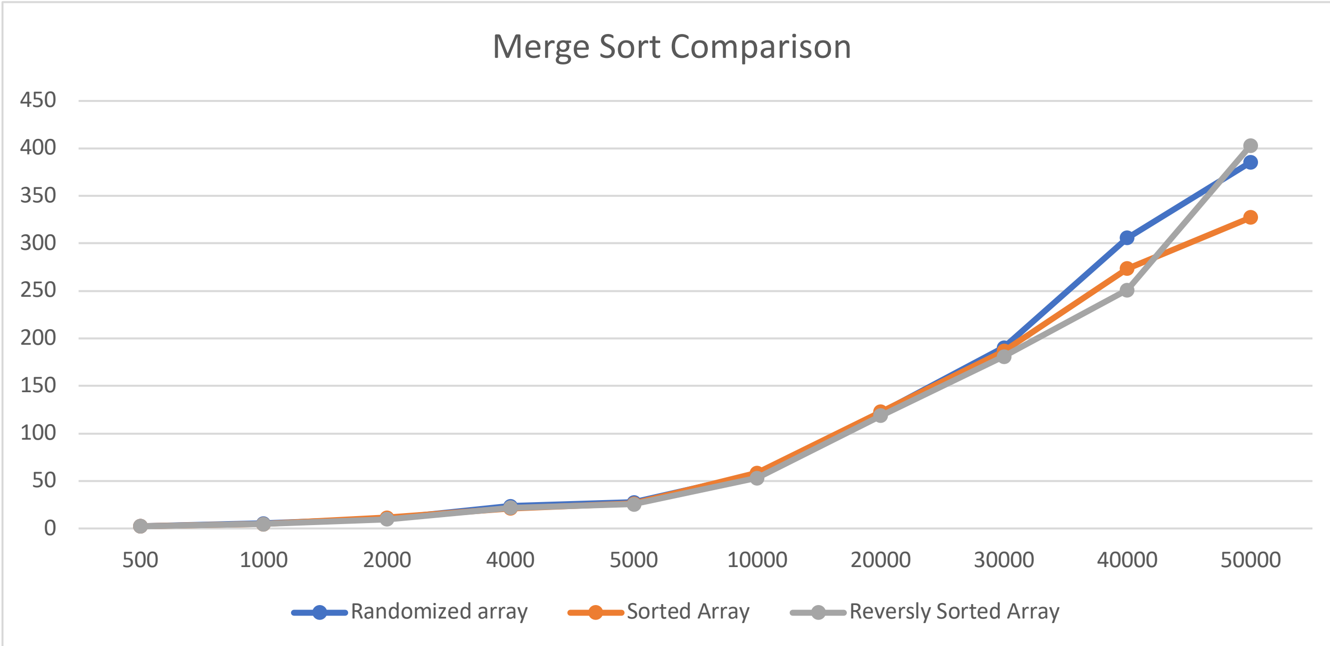
# PART 3

**Runtimes Comparison of an Algorithm for different type of inputs (randomized array, sorted array and reversely sorted array)**

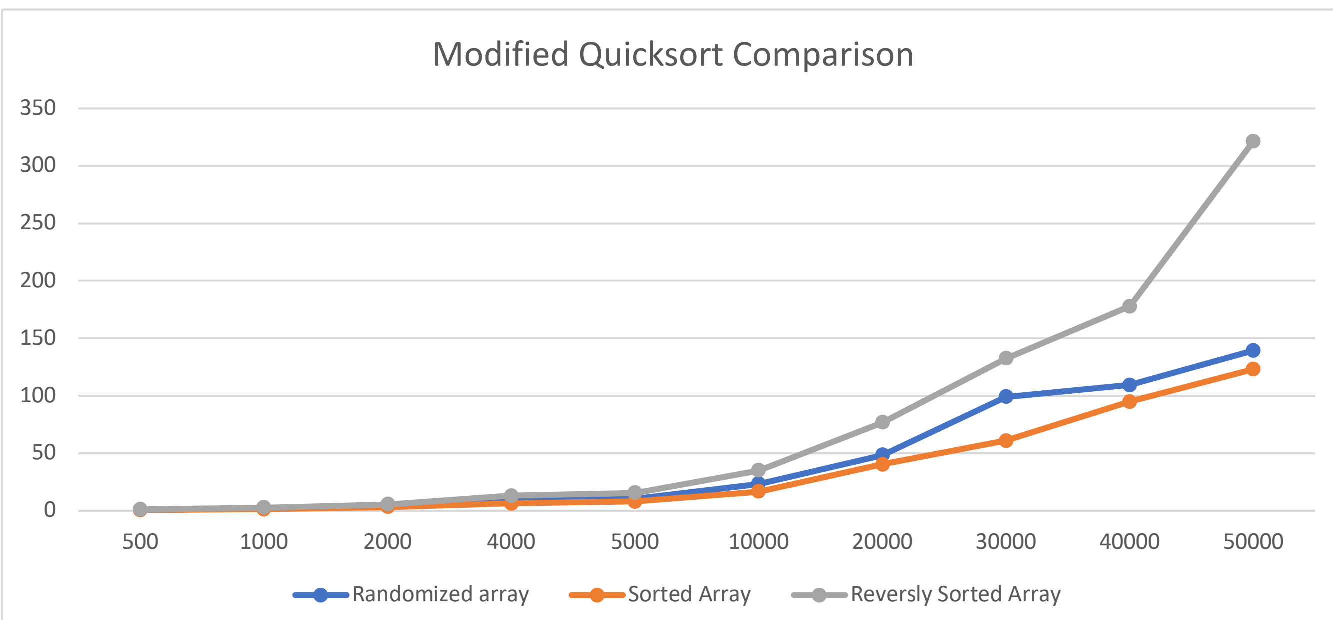
1. **Insertion Sort**
   We can observe from the below graph that Insertion sort works the fastest for sorted arrays, the runtime is almost constant for input size ranging from 500 to 50,000. We can also observe that the runtimes for different input arrays do not differ by much for input size <=10,000. However, for input size >10,000, sorting of reversely sorted array takes considerably more time as compared to randomized array.




2. **Merge Sort**
   We can observe from the below graph that Merge sort takes more or less the same time for sorting different types of input arrays. Just that, the runtimes increase noticeably for input size > 10,000




3. **Modified Quicksort**
We can observe from the below graph that Modified quicksort takes the maximum time to sort reversely sorted array and the runtimes for the same are considerably higher as compared to randomized array and sorted array specially for input size>10,000. While, sorting of randomized array and sorted array takes more or less the same time for all input sizes.



# CODE FOR EACH SORTING TECHNIQUE

## (1) Insertion sort

```
def insertionSort(array):
        for j in range (1, len(array)):
            key = array[j]
            i = j-1
            while i >= 0 and array[i]>key:
                array[i+1]=array[i]
                i=i-1
            array[i+1]=key
        return array
```

## (2) Merge sort

```
def merge(array_to_be_sorted, left, mid, right):
    n1 = mid - left + 1
    n2 = right - mid

    # creation of temporary arrays for merging
    L = [0] * (n1)
    R = [0] * (n2)

    # Copy data to temp arrays L[] and R[]
    for x in range(0 , n1):
        L[x] = array_to_be_sorted[left + x]

    for y in range(0 , n2):
        R[y] = array_to_be_sorted[mid + 1 + y]

    # Merge the temp arrays back into array_to_be_sorted[left..right]
    i = 0     # Initial index of first subarray
    j = 0     # Initial index of second subarray
    k = left  # Initial index of merged subarray

    while i < n1 and j < n2 :
        if L[i] <= R[j]:
            array_to_be_sorted[k] = L[i]
            i += 1
        else:
            array_to_be_sorted[k] = R[j]
            j += 1
        k += 1

    # Copy the remaining elements of L[], if there are any items remaining
    while i < n1:
        array_to_be_sorted[k] = L[i]
        i += 1
        k += 1

    # Copy the remaining elements of R[], if there are any items remaining
    while j < n2:
        array_to_be_sorted[k] = R[j]
```

```
        j += 1
        k += 1



def mergeSort(array_to_be_sorted,left,right):
    if left < right:

        #find the middle index of the array
        mid = (left+right)/2

        #Recursively calling mergesort to sort first and second half of
the array
        mergeSort(array_to_be_sorted, left, mid)
        mergeSort(array_to_be_sorted, mid+1, right)
        merge(array_to_be_sorted, left, mid, right)
```

**(3) <u>In place quick sort</u>**

```
# array_to_be_sorted[] = Array which is to be sorted,
# low  = Starting index of the array,
# high = Ending index of the array

def array_partition(array_to_be_sorted,low,high):
    pivot = array_to_be_sorted[high]     # last element of the array is
being taken as pivot
    x = ( low-1 )

    for y in range(low , high):
        # check whether current element is smaller than or equal to pivot
        if   array_to_be_sorted[y] <= pivot:
            # increment index of smaller element
            x = x+1
            array_to_be_sorted[x],array_to_be_sorted[y] =
array_to_be_sorted[y],array_to_be_sorted[x]
    array_to_be_sorted[x+1],array_to_be_sorted[high] =
array_to_be_sorted[high],array_to_be_sorted[x+1]
    return ( x+1 )


# Quick sort function
def quickSort(array_to_be_sorted,low,high):
    if low < high:
        # pi is partitioning index
        pi = array_partition(array_to_be_sorted,low,high)

        # Sorting of elements separately before and after partition
        quickSort(array_to_be_sorted, low, pi-1)
        quickSort(array_to_be_sorted, pi+1, high)
```

**(4) Modified Quick sort**

```
import datetime
import quick_insertion_sort
```

```
import pdb
import random
from numpy import median

    #length = high - low
    #length = len(arr)
    #middle = length/2
    #pivot = int(median([arr[low],arr[high],arr[middle]]))  # pivot as
median of three
    #arr[high],arr[pivot

def MedianOfThree(arr, low, high):
    mid = (low + high)/2
    if arr[high] < arr[low]:
        Swap(arr, low, high)
    if arr[mid] < arr[low]:
        Swap(arr, mid, low)
    if arr[high] < arr[mid]:
        Swap(arr, high, mid)
    Swap(arr,mid,high)
    return arr[high]


# Generic Swap for manipulating list data.
def Swap(arr, left, right):
    temp = arr[left]
    arr[left] = arr[right]
    arr[right] = temp


# array_to_be_sorted[] --> Array which is to be sorted,
# low  --> Starting index,
# high  --> Ending index

# Partition Function
def array_partition(array_to_be_sorted,low,high):
    x = ( low-1 )         # index of smaller element
    pivot= MedianOfThree (array_to_be_sorted,low,high)
    for y in range(low , high):

        # If current element is smaller than or equal to pivot
        if   array_to_be_sorted[y] <= pivot:

            # increment index of smaller element
            x = x+1
            array_to_be_sorted[x],array_to_be_sorted[y] =
array_to_be_sorted[y],array_to_be_sorted[x]

    array_to_be_sorted[x+1],array_to_be_sorted[high] =
array_to_be_sorted[high],array_to_be_sorted[x+1]
    return ( x+1 )

# Quick sort function
def quickSort(array_to_be_sorted,low,high):
    #pdb.set_trace()
    if (high-10) > low:
        if low < high:
```

```
            # pi is partitioning index
            pi = array_partition(array_to_be_sorted,low,high)

            # Separately sort elements before partition and after
partition
            quickSort(array_to_be_sorted, low, pi-1)
            quickSort(array_to_be_sorted, pi+1, high)
    else:
        quick_insertion_sort.insertionSort(array_to_be_sorted,low,high)
```

**quick_insertion_sort file :**

```
def insertionSort(array,low,high):
        for j in range (low+1, high+1):
            key = array[j]
            i = j-1
            while i >= 0 and array[i]>key:
                array[i+1]=array[i]
                i=i-1
            array[i+1]=key
        return array
```

**(5) Main function calling test file**

```
import insertion_sort
import merge_sort
import modified_quicksort
import inplace_quicksort
import datetime
import random

# Code to generate random numbers and append them to a list
# start = starting range,
# end = ending range
# num = number of elements needs to be appended
def Rand(start, end, num):
    res = []

    for j in range(num):
        res.append(random.randint(start, end))

    return res

def sorting_algorithms(arrayToBeSorted):
    # Calling Insertion sort and calculating time elapsed
    print("\n\n*********INSERTION SORT**********")
    #print("\nArray Before Sort")
    arrayToBeSorted1=list(arrayToBeSorted)
    #print(arrayToBeSorted1)
    startTime1 = datetime.datetime.now()
    sorted_array= insertion_sort.insertionSort(arrayToBeSorted1)
    endTime1 = datetime.datetime.now()
    diff = endTime1 - startTime1
    timeElapsed1=diff.total_seconds() * 1000
```

```
    #print ("\n\nSorted array after Insertion sort is:")
    #print(sorted_array)
    print("\nTime elapsed in milliseconds after Insertion sort is : ")
    print(timeElapsed1)


    # Calling merge sort and calculating time elapsed
    print("\n\n\n*********MERGE SORT**********")
    #print("\nArray Before Sort")
    arrayToBeSorted2=list(arrayToBeSorted)
    #print(arrayToBeSorted2)
    n = len(arrayToBeSorted2)
    startTime2 = datetime.datetime.now()
    merge_sort.mergeSort(arrayToBeSorted2,0,n-1)
    endTime2 = datetime.datetime.now()
    diff = endTime2 - startTime2
    timeElapsed2=diff.total_seconds() * 1000
    #print ("\n\nSorted array after merge sort is")
    #print (arrayToBeSorted2)
    print("\n\nTime elapsed in milliseconds after Merge-sort is : ")
    print(timeElapsed2)

    # Calling In-place quicksort sort and calculating time elapsed
    print("\n\n\n*********IN-PLACE QUICKSORT**********")
    #print("\nArray Before Sort")
    arrayToBeSorted3=list(arrayToBeSorted)
    #print(arrayToBeSorted3)
    n = len(arrayToBeSorted3)
    startTime3 = datetime.datetime.now()
    inplace_quicksort.quickSort(arrayToBeSorted3,0,n-1)
    endTime3 = datetime.datetime.now()
    diff = endTime3 - startTime3
    timeElapsed3=diff.total_seconds() * 1000
    #print ("\n\nSorted array after In-place quicksort is:")
    #print (arrayToBeSorted3)
    #for i in range(n):
    #print ("%d" %arrayToBeSorted3[i]),
    print("\n\nTime elapsed in milliseconds after Quicksort is : ")
    print(timeElapsed3)

    # Calling Modified Quicksort and calculating time elapsed
    print("\n\n\n*********MODIFIED QUICKSORT**********")
    #print("\nArray Before Sort")
    arrayToBeSorted4=list(arrayToBeSorted)
    #print(arrayToBeSorted4)
    n = len(arrayToBeSorted4)
    startTime4 = datetime.datetime.now()
    modified_quicksort.quickSort(arrayToBeSorted4,0,n-1)
    endTime4 = datetime.datetime.now()
    diff = endTime4 - startTime4
    timeElapsed4=diff.total_seconds() * 1000
    #print ("\n\nSorted array after Modified quicksort is:")
    #print (arrayToBeSorted4)
    print("\n\nTime elapsed in milliseconds after Modified Quicksort is :
")
    print(timeElapsed4)
```

```
# Driver code to generate array list of random numbers
input_size_array =[500]
#input_size_array =[500,1000,2000,4000,5000,10000,20000,30000,40000,50000]
start = 1
end = 100000
for x in range(len(input_size_array)):
    for y in range(3):
        num = input_size_array[x]
        arrayToBeSorted= Rand(start, end, num)

        #Case 1: Randomized array
        sorting_algorithms(arrayToBeSorted)

        #Case 2:
 #       insertion_sort.insertionSort(arrayToBeSorted)
 #       temp_array = list(arrayToBeSorted_temp)
 #       n = len(temp_array)
#        modified_quicksort.quickSort(temp_array,0,n-1)
 #      sorting_algorithms(arrayToBeSorted)

        #Case 3:
  #      temp_array = list(arrayToBeSorted)
#        n = len(temp_array)
#        sorted_array = modified_quicksort.quickSort(temp_array,0,n-1)
#        reversed_sorted_array = list(reversed(sorted_array)
#        sorting_algorithms(reversed_sorted_array)
```

## Appendix A

Excluding In-place Quicksort as it is going in infinite loop for input size >=995. So, runtime values could not be captured.
We did some research on as why the algorithm is going in infinite loop for sorted and reverse sorted array. We came across the below article:

(1) WAYS TO FAIL HORRIBLY WHILE IMPLEMENTING QUICKSORT
*(Ref: http://www.mkrevuelta.com/en/2016/02/28/7-ways-to-fail-horribly-while-implementing-quicksort/)*

To use always the first element as pivot (or always the last): - In our case we are using last element as pivot

This would be less important if all possible permutations happened with the same probability. But reality is different. The special cases of "already sorted data" (either in straight order or reverse) or "almost sorted data" are quite frequent in comparison with other permutations. And in these cases, the first and last elements are the worst candidates we can choose as pivot.
If we use always the first element (or always the last) as pivot, quicksort degenerates into its worst behaviour with already sorted data, no matter whether they are in ascending or descending order.
This problem is usually attacked in two different ways:

- To use as pivot the median of three elements: the fist, the last and the middle one
- To use as pivot the element of a position chosen at random – We tried this case by randomly choosing any number as pivot, but even for this case it is going in infinite loop.

Both methods reduce the probability of the worst case, but they can't eliminate it completely.