ALGORITHMS AND DATA STRUCTURES - PROJECT 1

COMPARISON-BASED SORTING ALGORITHMS

Developed by:
Balasundaram Avudai Nayagam
Nivedita Veeramanigandan

INTRODUCTION:

In this project, five comparison-based sorting algorithms have been implemented and their running times are noted for various cases. These cases involve the numbers being in a random order, sorted order or a reversely sorted order. The sorting algorithms considered here are Insertion Sort, Merge Sort, Heap Sort, Quick Sort(In place), Quick Sort with Median 3 (considering pivot as the median of three elements). We have used Python language for implementing the sorting algorithms.

IMPLEMENTATION

INSERTION SORT:

Insertion sort is a very simple comparative sorting algorithm which is not recursive. This is an in-place algorithm as it does not require any extra space while sorting and its space complexity is O(1). But since the algorithm takes $O(n^2)$ time in the worst case, it is not as efficient as other algorithms for larger datasets.

MERGE SORT:

Merge sort is a comparative divide and conquer algorithm. The algorithm recursively divides up the list until the length is one and then recombines using a merge function that maintains the sorted order as it recombines the lists. Merge sort is often described to be an O(nlogn) algorithm. However, the implementation takes extra memory and hence when large data sets are given as input it takes a large memory to execute.

HEAP SORT:

Heapsort is a sorting algorithm based on the structure of a heap. Initially, a tree is created with the values to be sorted, starting from the left, the root node, with the first value. We create child nodes as well as heapify the tree. The idea is that the parent nodes always have bigger values than the child nodes. Now we start to compare parent and child nodes values looking for the biggest value between them, and when we find it, we change places reordering the values. We do this recursively until all the nodes are greater than their children nodes. The time complexity taken for this algorithm is O(nlogn).

QUICK SORT (In-place):

Quicksort, like merge sort, is another divide and conquer algorithm that is recursive in nature. The idea is that a pivot element is chosen either at the beginning or middle or end of the list and then the list is partitioned into three sub-lists namely the numbers less than the pivot, the pivot, and the numbers greater than the pivot. Then these sub-lists are recursively partitioned based on pivot until they cannot be further partitioned. The time complexity taken for this algorithm is $O(n^2)$.

QUICK SORT WITH MEDIAN OF 3:

This is a variation of the above mentioned Quicksort. The major difference is in the selection of the pivot element. Here the pivot is chosen based on the median of the left-most, rightmost and the middle element. Based on that pivot, the process of the quick sort again continues recursively. However for large datasets, the implementation is observed to take more time for execution than the merge and quick sort, yet significantly lesser than the insertion sort implementation. The time complexity taken for this algorithm is $O(n^2)$.

COMPARISON AMONG THE SORTING ALGORITHMS:

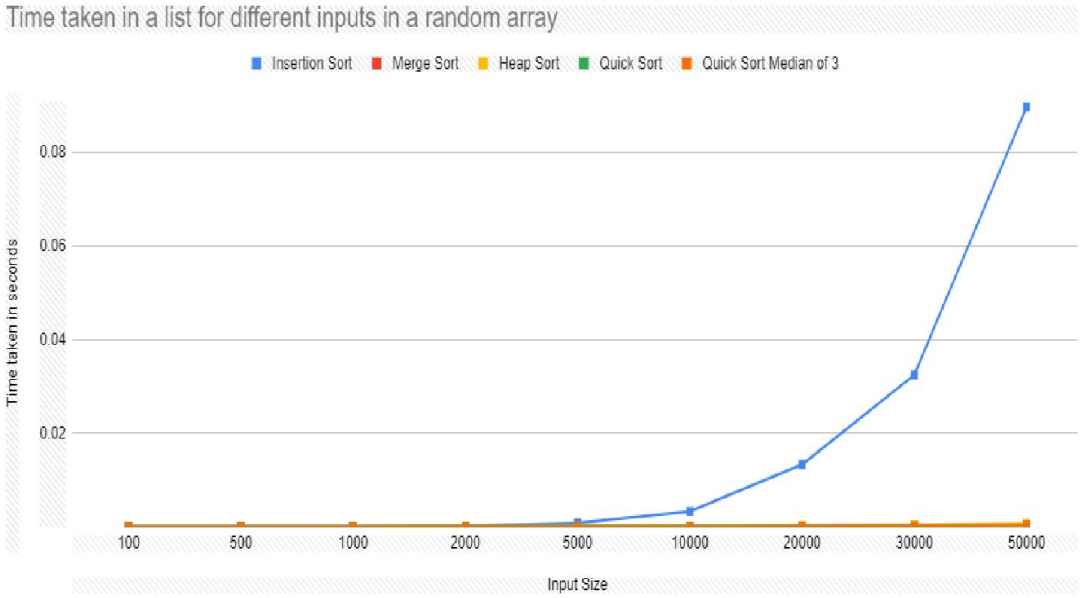
Case 1: When random set of numbers are given as Input

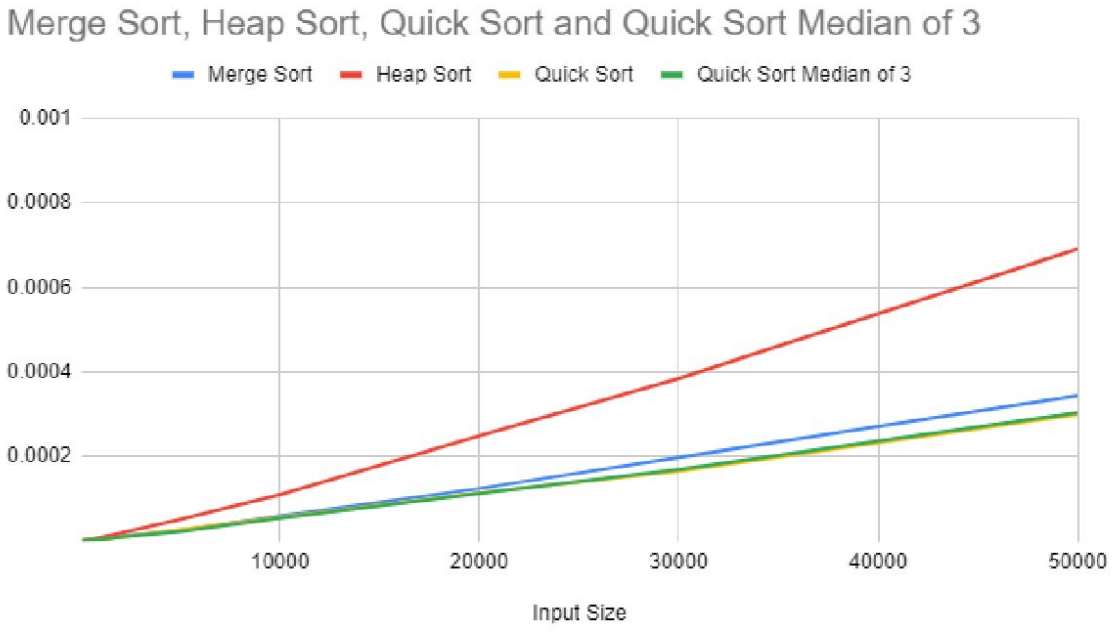
Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort Median of 3
100	0.0007	0.0006	0.001	0.0004	0.0004
500	0.009	0.002	0.0032	0.0018	0.0022
1000	0.0325	0.0047	0.0085	0.0039	0.0038
2000	0.13	0.0111	0.0185	0.0095	0.0098
5000	0.8015	0.0245	0.0512	0.0264	0.023
10000	3.2751	0.0589	0.1097	0.0559	0.0546
20000	13.2971	0.1246	0.2488	0.1145	0.1125
30000	32.4556	0.1979	0.3848	0.166	0.1701
50000	89.83	0.3446	0.6909	0.2992	0.3034

Time in Seconds

From the observation, we have recorded the execution time for each of the sorting algorithm and plotted a graph for the same to analyze the performance. First and second graph below shows the performance of sorting algorithms for randomly generated elements given as input. It can be inferred that execution time for insertion sort is suitable only for small input size and the very high as the input size increases. Merge sort, quicksort and modified quicksort takes less execution time than insertion sort.

The merge sort and quicksort take less execution time among the others, and among those two, quick sort is faster and efficient taking less execution time.



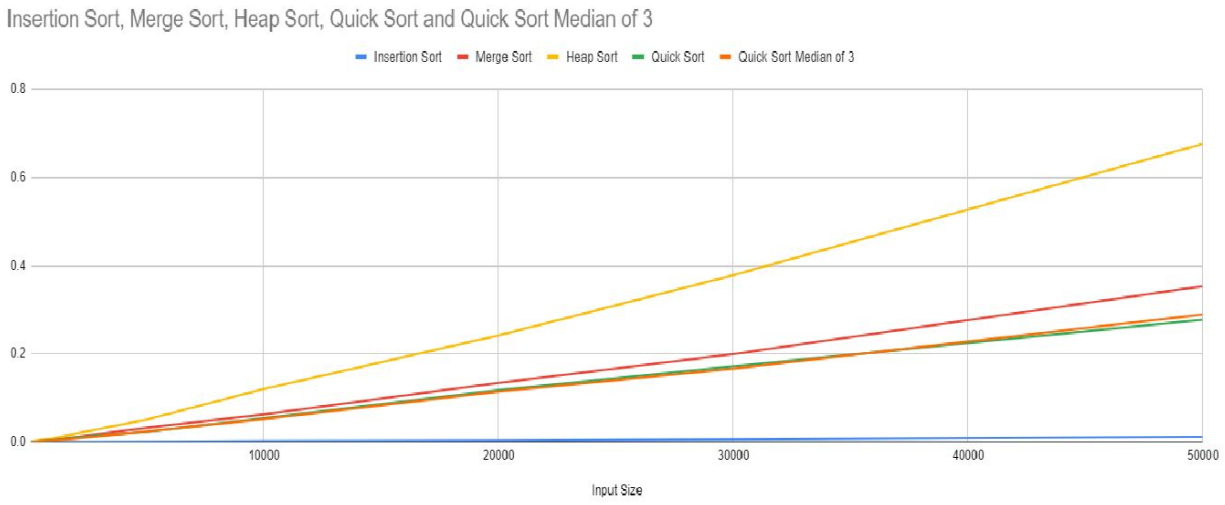


Note: Since in the first graph, the plotting for merge, heap, and quick sort are not clear because of high execution time values for insertion sort, we have plotted a separate graph to show the time values of Merge ,Heap and Quick sort.

Case 2: When input array is already sorted

Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort Median of 3
100	4.60E-05	0.0005	0.0007	0.0004	0.0006
500	0.0001	0.0029	0.0048	0.0023	0.0022
1000	0.0002	0.0057	0.0082	0.0045	0.0038
2000	0.0004	0.0114	0.0197	0.0112	0.0081
5000	0.0011	0.0325	0.0511	0.0237	0.0242
10000	0.0021	0.063	0.1207	0.0542	0.0525
20000	0.0036	0.1343	0.2417	0.1176	0.1144
30000	0.0065	0.1995	0.3786	0.1719	0.1668
50000	0.0121	0.3535	0.676	0.2777	0.2895

Time in Seconds

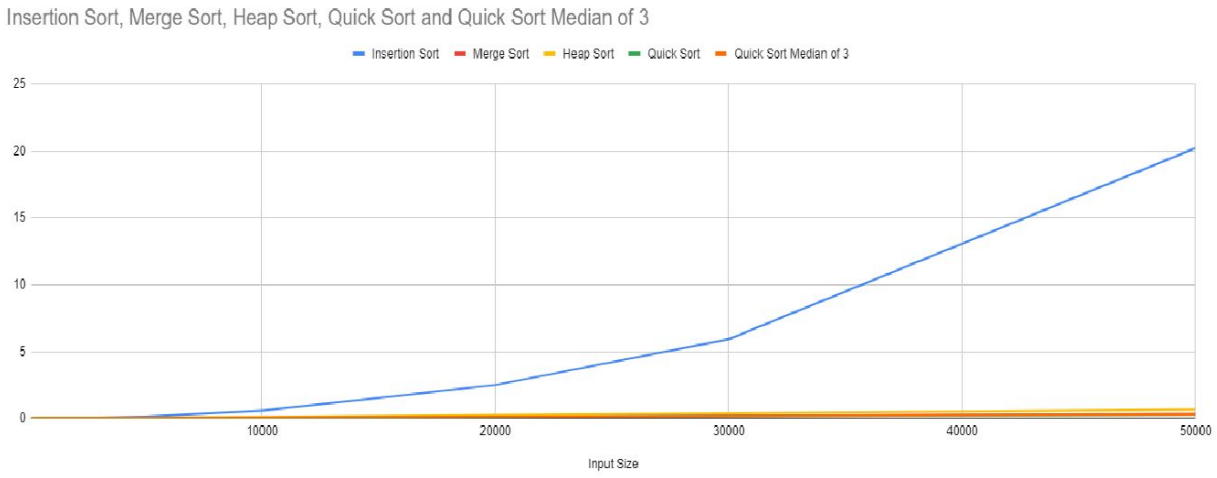


When the input is a sorted array, Insertion sort runs faster than all other sorting algorithms as it does not involve any swapping operation. Followed by insertion sort, Quick sort is faster than the other two algorithms i.e, merge and heap sort.

Case 3: When the input array is reversely sorted

Input Size	Insertion Sort	Merge Sort	Heap Sort	Quick Sort	Quick Sort Median of 3
100	0.0002	0.0004	0.0007	0.0004	0.0006
500	0.0022	0.0025	0.004	0.0029	0.0023
1000	0.0059	0.0059	0.009	0.0053	0.0048
2000	0.02	0.0094	0.0192	0.0089	0.0099
5000	0.1332	0.0251	0.0526	0.0217	0.0239
10000	0.5856	0.0622	0.1183	0.0545	0.0541
20000	2.4962	0.1281	0.2554	0.1139	0.1113
30000	5.9207	0.2052	0.3823	0.1714	0.1826
50000	20.212	0.3464	0.6634	0.2835	0.292

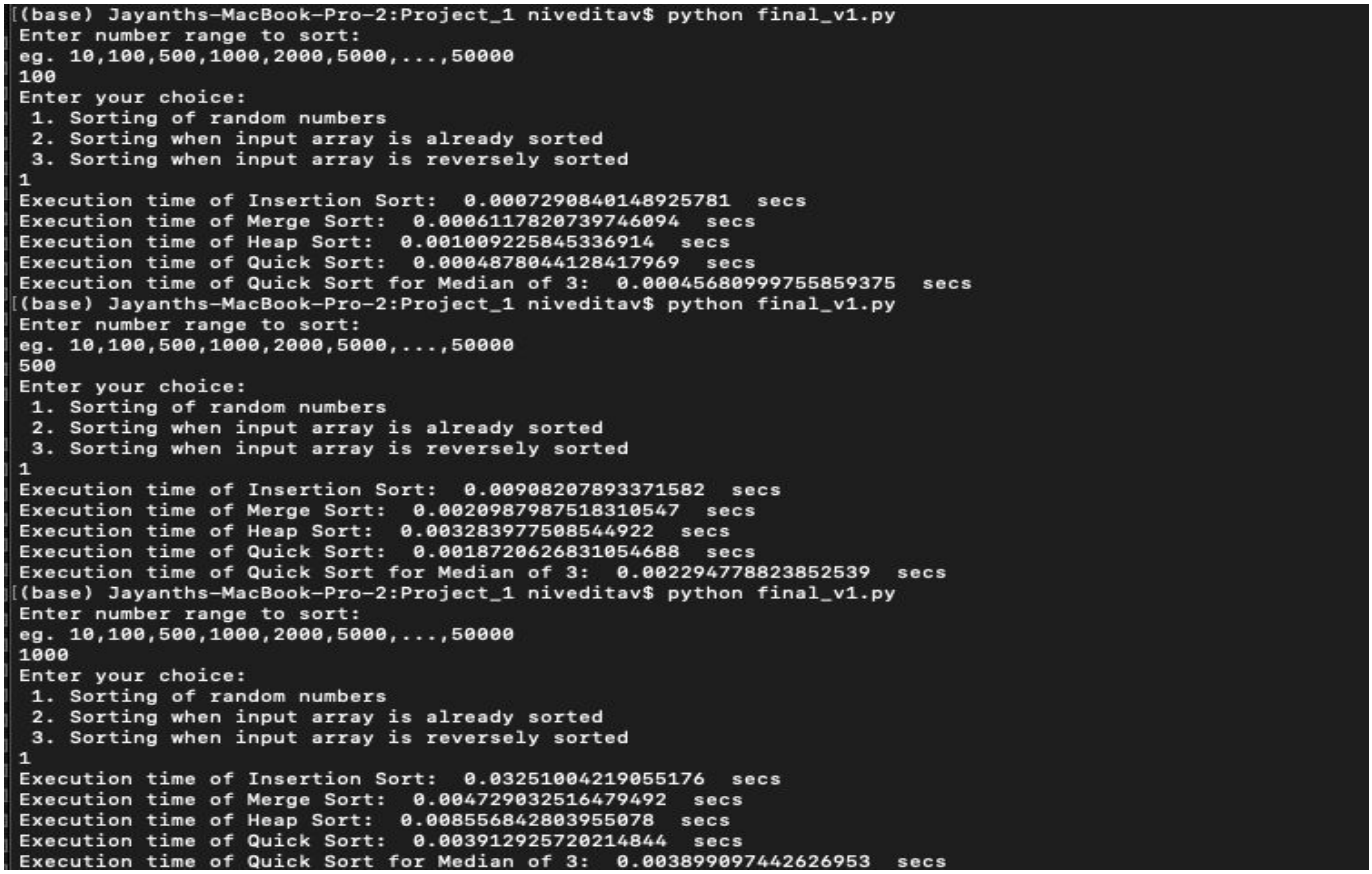
Time in Seconds



When the input array is reversely sorted, Quick sort performs faster than other sorting algorithms. On the other side, the insertion sort takes a large time for execution since each element needs to be compared and swapped at each and every iteration which is the reason for high execution time. Other algorithms execution is not much impacted by the input even when it is reversely sorted.

SAMPLE EXECUTION RESULTS

The below images displays the execution time taken for each sort during each special case and various sets of inputs that have been previously mentioned.

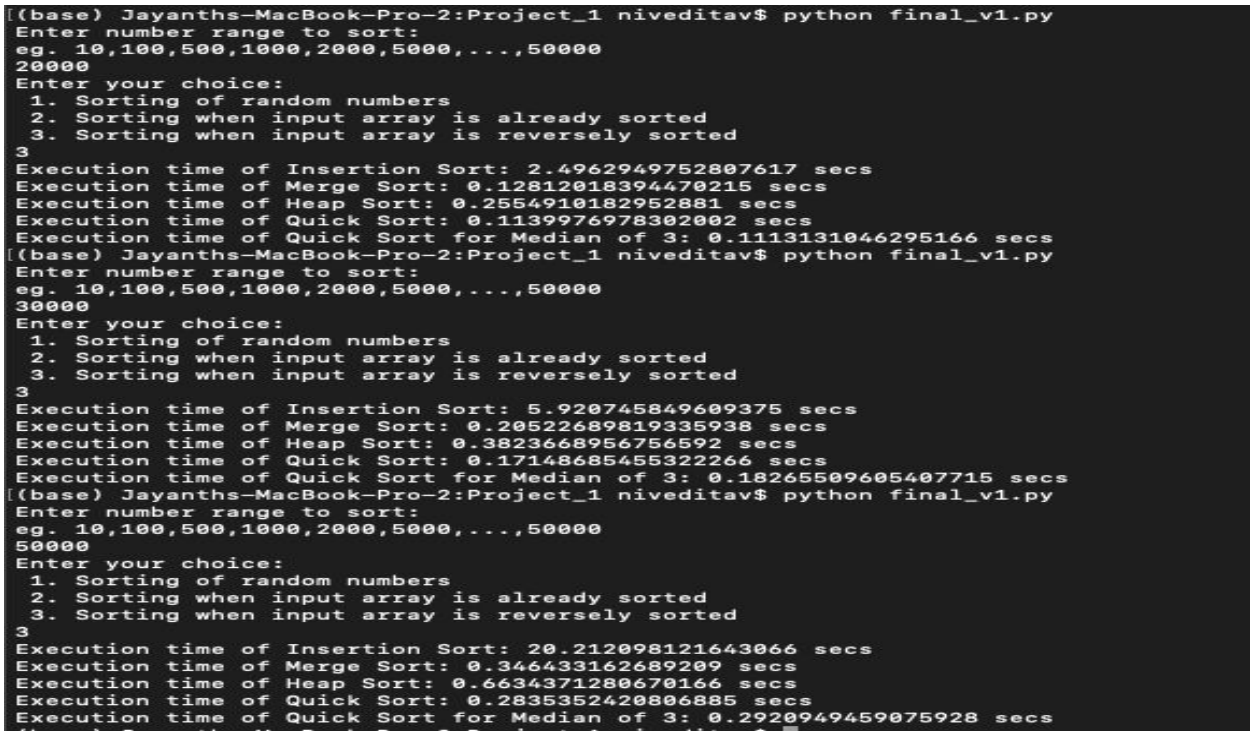


```
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
2000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  0.1300060749053955  secs
Execution time of Merge Sort:  0.011174201965332031  secs
Execution time of Heap Sort:  0.018557071685791016  secs
Execution time of Quick Sort:  0.009556055068969727  secs
Execution time of Quick Sort for Median of 3:  0.009824752807617188  secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
5000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  0.8015742301940918  secs
Execution time of Merge Sort:  0.024507284164428 71  secs
Execution time of Heap Sort:  0.051205873489379 88  secs
Execution time of Quick Sort:  0.026462316513061523  secs
Execution time of Quick Sort for Median of 3:  0.023049116134643555  secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
10000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  3.2751410007476807  secs
Execution time of Merge Sort:  0.05894899368286133  secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
20000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  13.297152042388916  secs
Execution time of Merge Sort:  0.1246180534362793  secs
Execution time of Heap Sort:  0.24883198738098145  secs
Execution time of Quick Sort:  0.11454606056213379  secs
Execution time of Quick Sort for Median of 3:  0.11253666877746582  secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
30000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  32.45567727088928  secs
Execution time of Merge Sort:  0.1979351043701172  secs
Execution time of Heap Sort:  0.3848111629486084  secs
Execution time of Quick Sort:  0.16601109504699707  secs
Execution time of Quick Sort for Median of 3:  0.17016863822937012  secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
50000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
1
Execution time of Insertion Sort:  89.83002305030823  secs
Execution time of Merge Sort:  0.344602108001709  secs
Execution time of Heap Sort:  0.6909098625183105  secs
Execution time of Quick Sort:  0.29926204681396484  secs
Execution time of Quick Sort for Median of 3:  0.3034639358520508  secs
```

```
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
2000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.0004680156707763672 secs
Execution time of Merge Sort: 0.011433124542236328 secs
Execution time of Heap Sort: 0.01978325843811035 secs
Execution time of Quick Sort: 0.01120901107788086 secs
Execution time of Quick Sort for Median of 3: 0.008110761642456055 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
5000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.001194000244140625 secs
Execution time of Merge Sort: 0.03259706497192383 secs
Execution time of Heap Sort: 0.05114603042602539 secs
Execution time of Quick Sort: 0.023777008056640625 secs
Execution time of Quick Sort for Median of 3: 0.024231910705566406 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
10000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.002112865447998047 secs
Execution time of Merge Sort: 0.06301426887512207 secs
Execution time of Heap Sort: 0.12076997756958008 secs
Execution time of Quick Sort: 0.05424809455871582 secs
Execution time of Quick Sort for Median of 3: 0.05259203910827637 secs
```

```
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
100
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
3
Execution time of Insertion Sort: 0.00026488304138183594 secs
Execution time of Merge Sort: 0.0004889965057373047 secs
Execution time of Heap Sort: 0.0007841587066650391 secs
Execution time of Quick Sort: 0.0004589557647705078 secs
Execution time of Quick Sort for Median of 3: 0.0006818771362304688 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
500
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
3
Execution time of Insertion Sort: 0.0022869110107421875 secs
Execution time of Merge Sort: 0.0025646686553955078 secs
Execution time of Heap Sort: 0.004057168960571289 secs
Execution time of Quick Sort: 0.0029709339141845703 secs
Execution time of Quick Sort for Median of 3: 0.0023219585418701172 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
1000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
3
Execution time of Insertion Sort: 0.005943775177001953 secs
Execution time of Merge Sort: 0.005964756011962891 secs
Execution time of Heap Sort: 0.009030818939208984 secs
Execution time of Quick Sort: 0.005326032638549805 secs
Execution time of Quick Sort for Median of 3: 0.004854917526245117 secs
```

```
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
20000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.0036940574645996094 secs
Execution time of Merge Sort: 0.1343092918395996 secs
Execution time of Heap Sort: 0.2417306900024414 secs
Execution time of Quick Sort: 0.11765909194946289 secs
Execution time of Quick Sort for Median of 3: 0.11444091796875 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
30000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.006562948226928711 secs
Execution time of Merge Sort: 0.19954824447631836 secs
Execution time of Heap Sort: 0.37869691848754883 secs
Execution time of Quick Sort: 0.1719050407409668 secs
Execution time of Quick Sort for Median of 3: 0.16686487197875977 secs
(base) Jayanths-MacBook-Pro-2:Project_1 niveditav$ python final_v1.py
Enter number range to sort:
eg. 10,100,500,1000,2000,5000,...,50000
50000
Enter your choice:
 1. Sorting of random numbers
 2. Sorting when input array is already sorted
 3. Sorting when input array is reversely sorted
2
Execution time of Insertion Sort: 0.012183189392089844 secs
Execution time of Merge Sort: 0.35357213020324707 secs
Execution time of Heap Sort: 0.6760210990905762 secs
Execution time of Quick Sort: 0.27778697013854980 secs
Execution time of Quick Sort for Median of 3: 0.28955698013305664 secs
```



CODE:

```
import random
import time
size = int(input("Enter number range to sort:\neg. 100,500,1000,2000,5000,...,50000\n"))
def Random_input_gen(start, size):
    input_list = []

    for n in range(size):
        input_list.append(random.randint(start, size))
    return input_list

arr = Random_input_gen(1,size)
choice = int(input("Enter your choice:\n 1. Sorting of random numbers\n 2. Sorting when input
array is already sorted\n 3. Sorting when input array is reversely sorted\n"))
if(choice == 1):
###----------------------------------------INSERTION SORT----------------------------------------###
        start = time.time()
        insertionSort(arr)
        end = time.time()
        print("Execution time of Insertion Sort:",end - start,"secs")
        print(arr)
```

```
###----------------------------------------MERGE SORT----------------------------------------###
        start = time.time()
        mergeSort(arr)
        end = time.time()
        print("Execution time of Merge Sort:",end - start,"secs")
        print(arr)
###----------------------------------------HEAP SORT----------------------------------------###
        start = time.time()
        heapSort(arr)
        end = time.time()
        print("Execution time of Heap Sort:",end - start,"secs")
        print(arr)
###----------------------------------------QUICK SORT----------------------------------------###
        start = time.time()
        quickSort(arr)
        end = time.time()
        print("Execution time of Quick Sort:",end - start,"secs")
        print(arr)
###---------------------------------QUICK SORT FOR MEDIAN OF 3---------------------------------###
        if(size <= 10):
                print("Calling Insertion sort since array size is less than or equal to 10\n")
                start = time.time()
                insertionSort(arr)
                end = time.time()
                print("Execution time of Insertion Sort :",end - start,"secs")
        else:
                start = time.time()
                mquickSort(arr)
                end = time.time()
                print("Execution time of Quick Sort for Median of 3:",end - start,"secs")
                print(arr)

elif (choice == 2):
        arr_sort = sorted(arr)
###----------------------------------------INSERTION SORT----------------------------------------###
        start = time.time()
        insertionSort(arr_sort)
        end = time.time()
        print("Execution time of Insertion Sort:",end - start,"secs")
        print(arr_sort)
###----------------------------------------MERGE SORT----------------------------------------###
        start = time.time()
        mergeSort(arr_sort)
```

```
        end = time.time()
        print("Execution time of Merge Sort:",end - start,"secs")
        print(arr_sort)
###-----------------------------------------HEAP SORT-----------------------------------------###
        start = time.time()
        heapSort(arr_sort)
        end = time.time()
        print("Execution time of Heap Sort:",end - start,"secs")
        print(arr_sort)
###-----------------------------------------QUICK SORT-----------------------------------------###
        start = time.time()
        quickSort(arr_sort)
        end = time.time()
        print("Execution time of Quick Sort:",end - start,"secs")
        print(arr_sort)
###---------------------------------QUICK SORT FOR MEDIAN OF 3---------------------------------###
        if(size <= 10):
                print("Calling Insertion sort since array size is less than or equal to 10\n")
                start = time.time()
                insertionSort(arr_sort)
                end = time.time()
                print("Execution time of Insertion Sort :",end - start,"secs")
        else:
                start = time.time()
                mquickSort(arr_sort)
                end = time.time()
                print("Execution time of Quick Sort for Median of 3:",end - start,"secs")
                print(arr_sort)

else:
        arr_rev = sorted(arr,reverse = True)
###-----------------------------------------INSERTION SORT-----------------------------------------###
        start = time.time()
        insertionSort(arr_rev)
        end = time.time()
        print("Execution time of Insertion Sort:",end - start,"secs")
        #print(arr_rev)
###-----------------------------------------MERGE SORT-----------------------------------------###
        start = time.time()
        mergeSort(arr_rev)
        end = time.time()
        print("Execution time of Merge Sort:",end - start,"secs")
        print(arr_rev)
```

```
###----------------------------------------HEAP SORT----------------------------------------###
        start = time.time()
        heapSort(arr_rev)
        end = time.time()
        print("Execution time of Heap Sort:",end - start,"secs")
        print(arr_rev)
###----------------------------------------QUICK SORT----------------------------------------###
        start = time.time()
        quickSort(arr_rev)
        end = time.time()
        print("Execution time of Quick Sort:",end - start,"secs")
        print(arr_rev)
###--------------------------------QUICK SORT FOR MEDIAN OF 3--------------------------------###
        if(size <= 10):
                print("Calling Insertion sort since array size is less than or equal to 10\n")
                start = time.time()
                insertionSort(arr_rev)
                end = time.time()
                print("Execution time of Insertion Sort :",end - start,"secs")
        else:
                start = time.time()
                mquickSort(arr_rev)
                end = time.time()
                print("Execution time of Quick Sort for Median of 3:",end - start,"secs")
                print(arr_rev)
```

Insertion Sort.py

```
def insertionSort(arr):
  for i in range(1,len(arr)):
     if arr[i] >= arr[i-1]:
        continue
     for j in range(i):
        if arr[i] < arr[j]:
           arr[j],arr[j+1:i+1] = arr[i],arr[j:i]
           break
```

Merge Sort.py

```
def mergeSort(arr):
        if len(arr) >1:
                middle = len(arr)//2
```

```
        left = arr[:middle]
        right = arr[middle:]

        mergeSort(left)
        mergeSort(right)

        i = j = k = 0
        while i < len(left) and j < len(right):
                if left[i] < right[j]:
                        arr[k] = left[i]
                        i+=1
                else:
                        arr[k] = right[j]
                        j+=1
                k+=1

        while i < len(left):
                arr[k] = left[i]
                i+=1
                k+=1

        while j < len(right):
                arr[k] = right[j]
                j+=1
                k+=1
```

Heap Sort.py

```
def build_min_heap(array):
    for i in reversed(range(len(array)//2)):
        min_heapify(array, i)

def min_heapify(array, i):
    left = 2 * i + 1
    right = 2 * i + 2
    length = len(array) - 1
    smallest = i
    if left <= length and array[i] > array[left]:
        smallest = left
    if right <= length and array[smallest] > array[right]:
        smallest = right
    if smallest != i:
        array[i], array[smallest] = array[smallest], array[i]
```

```
        min_heapify(array, smallest)

def heapSort(arr):
    array = arr.copy()
    build_min_heap(array)
    sorted_array = []
    for i in range(len(array)):
        array[0], array[-1] = array[-1], array[0]
        sorted_array.append(array.pop())
        min_heapify(array, 0)
    return sorted_array
```

Quick Sort.py

```
def swap(arr,a,b):
  arr[a],arr[b] = arr[b],arr[a]
def partition(arr,start,end):
  median = (end - 1 - start) // 2
  median = median + start
  left = start + 1
  if (arr[median] - arr[end-1])*(arr[start]-arr[median]) >= 0:
     swap(arr,start,median)
  elif (arr[end - 1] - arr[median]) * (arr[start] - arr[end - 1]) >=0:
      swap(arr,start,end - 1)
  pivot = arr[start]
  for right in range(start,end):
     if pivot > arr[right]:
        swap(arr,left,right)
        left = left + 1
  swap(arr,start,left-1)
  return left-1
def quick(arr,left,right):
  if left < right:
     split = partition(arr,left,right)
     quick(arr,left,split)
     quick(arr,split+1,right)
def quickSort(arr):
  quick(arr,0,len(arr))
```

Quick Sort with Median of 3

```
def swap(arr,a,b):
  arr[a],arr[b] = arr[b],arr[a]
```

```
def partition(arr,start,end):
   med = ((end - 1) - start) // 2
   med = med + start
   left = start + 1
   if (arr[med] - arr[end-1])*(arr[start]-arr[med]) >= 0:
      swap(arr,start,med)
   elif (arr[end - 1] - arr[med]) * (arr[start] - arr[end - 1]) >=0:
       swap(arr,start,end - 1)
   pivot = arr[start]
   for right in range(start,end):
      if pivot > arr[right]:
         swap(arr,left,right)
         left = left + 1
   swap(arr,start,left-1)
   return left-1
def quick(arr,left,right):
   if left < right:
      split = partition(arr,left,right)
      quick(arr,left,split)
      quick(arr,split+1,right)
def mquickSort(arr):
   quick(arr,0,len(arr))
```