2208-CSE-5311-002-DSGN & ANLY ALGORITHMS : Project 1- sorting algorithms
Done by-
Name : Sai Spandana Gali
ID: 1001831591

Bubble Sort :

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order.

The main Data structure of the bubble sort algorithm is the function call – bubbleSort(array)

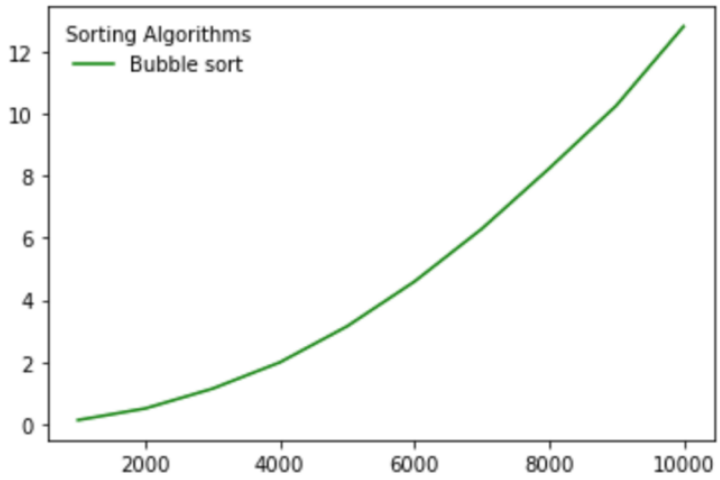
- What the function call does –
    1. The function takes an input array from the user.
    2. With a while loop each element is swapped by keeping track of the iteration.
    3. If the current element is greater than its next element, the two elements are swapped and increment the iteration. Till the no.of iterations id same as the length of the array.
    4. Return the sorted array

Experimental results :

Runtime :
[0.13344883918762207, 0.5068740844726562, 1.1436669826507568, 1.9907817840576172, 3.1508400440216064, 4.589542865753174, 6.285131931304932, 8.225250005722046, 10.253808975219727, 12.801832914352417]

Input Size:
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

Graph :



Insertion sort :

This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.

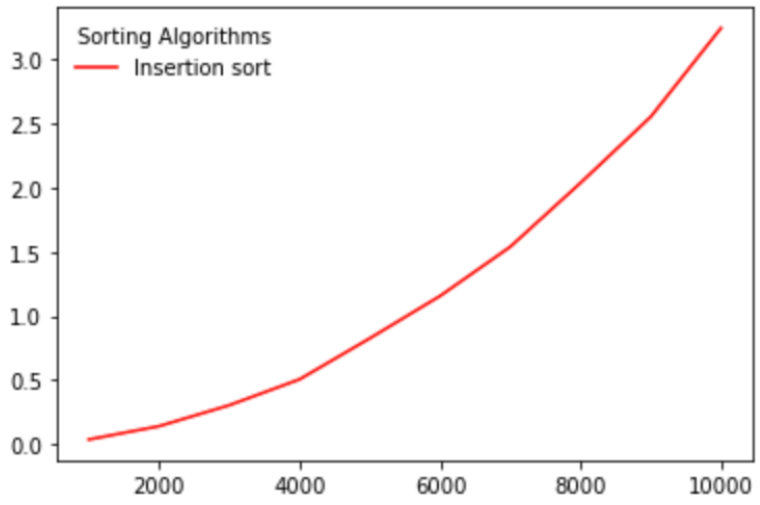
The main Data structure of the Insertion sort algorithm is the function call – insertionSort(array)

- What the function call does :
    1. The function takes input of an array.
    2. We assume the first element is sorted and assign the next element to key value.
    3. Compare the key with the element left to it(other value).
    4. Then we check if the other value is greater than or equal to 0 and if key Value is lesser than the other value.
    5. If it is, then key is placed in front of the other element.
    6. Repeat this until every element is in its correct position
    7. Return the array

Experimental results :

Runtime. :
[0.03701210021972656, 0.14052104949951172, 0.3040599822998047, 0.5057599544525146, 0.8256180286407471, 1.1556499004364014, 1.539273977279663, 2.0374178886413574, 2.5529017448425293, 3.244885206222534]

Input size :
1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000



Selection sort :

The selection sort algorithm sorts an array by repeatedly finding the minimum element from unsorted part and putting it at the beginning and maintain a sorted array.

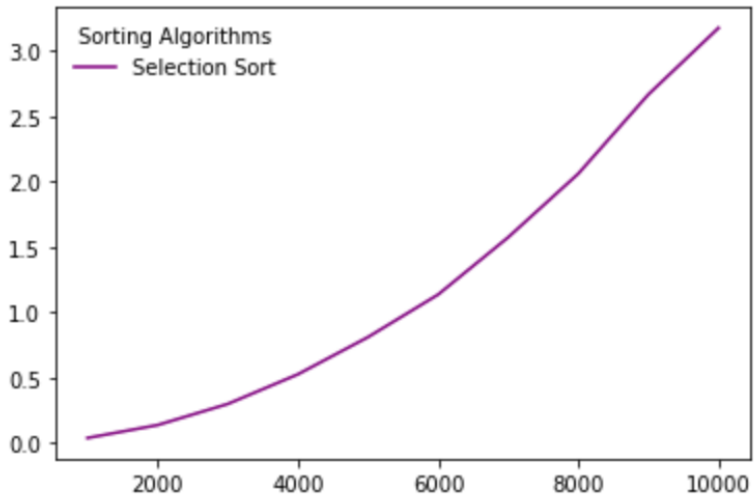
The main Data structure of the Selection sort algorithm is the function call – SelectionSort(array)

- What the function call does :
    1. It takes input of an array.
    2. There are 2 for loops to keep track of the sorted array and to compare each element with the rest of the array and pick the smallest value and swap it.
    3. In the second for loop, elements are checked to find the smallest element throughout the for loop. Once the smallest is found the minimum is updated.
    4. Then within the first for loop and outside the second the two elements – the current value and the minimum are swapped.
    5. This runs till the end of the array and till the unsorted list is empty.
    6. Return the array

Experimental results :

Runtime :
[0.03958702087402344, 0.13849115371704102, 0.29978394508361816, 0.5266411304473877, 0.8112969398498535, 1.1370172500610352, 1.5754711627960205, 2.062452793121338, 2.6692259311676025, 3.178209066390991]

Input Size :
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]



Merge Sort :

It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.

The main Data structure of the Merge sort algorithm is the function call –

- merge_sort(array)
- What the function call does :
    1. It takes input of an array, return the array if the number of elements in the array is 1.
    2. Splits the array into two array a1 and a2 with respect to the middle point.
    3. Calls the function merge_sort on both the arrays to futher split.
    4. Return the values of calling merge function on the a1 and a2.

- merge(arrayA,arrayB)
- What the function call does :
    1. Takes 2 array as input(the input array split into 2) , and define another array to store the sorted list.
    2. When array A and array B both have elements
        - Check which among them has the smaller element and pop that element and add it to the array C.

    3.When array A is not empty but array B is
        - pop the first element and add it to C, until A is empty.

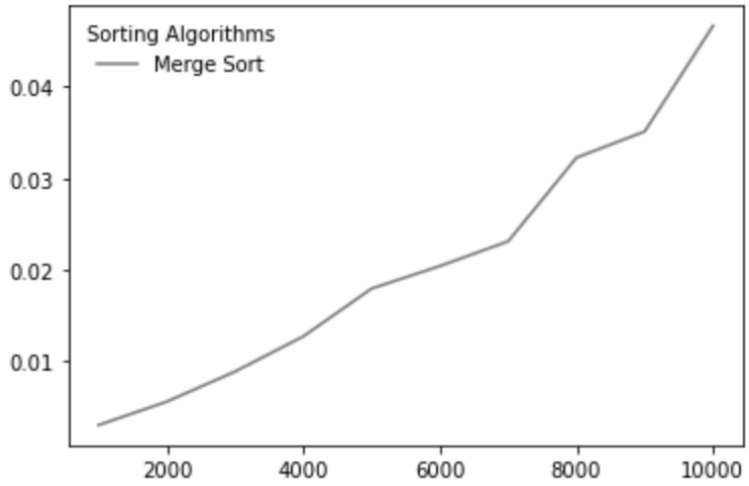
    4. When array B is not empty but array a is
        - pop the first element and add it to C, until B is empty.

    5. Return array C

Experimental results :

Runtimes :
[0.0030138492584228516, 0.005576372146606445, 0.008867025375366211, 0.01271510124206543, 0.017940044403076172, 0.020405054092407227, 0.02310490608215332, 0.03225207328796387, 0.0351161619567871094, 0.0466611385345459]

Input Size :
[1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]



QUICK SORT :

It picks an element as pivot and partitions the given array around the picked pivot.
Here pivot is the last element.

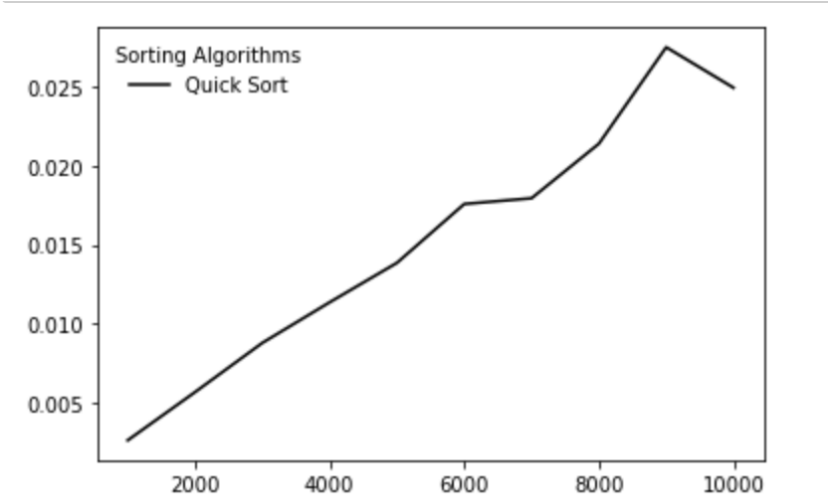
The main Data structure of the Quick sort algorithm is the function call –

- quick_sort(array,low,high)
    1. It takes 3 inputs the array, left most element(low), right most element(high).
    2. It checks if the left most element is less than the right most element
       - If it is, then partition function is called with passing the array, low, high, take the value of the array[index smaller value]
    3. Then quick_sort function is called over the left most partition and right most partition with respect to the partition.
    4. return the array
- Partition(array,low,high)
    1. In this algorithm the right most element is taken as pivot.
    2. A for loop is to iterate over the array, and compare each element with the pivot, if it is smaller than the pivot then the two elements current and smaller element are swapped and increment the index of the smaller value otherwise ignore the current element.
    3. Swap array[index smaller value] with array[high]
    4. Return the array[index smaller value]

Experimental results :

Runtime :
[0.0026602745056152344, 0.0057032108306884766, 0.008810043334960938, 0.01137399673461914, 0.013864994049072266, 0.017582178115844727, 0.0179440975189209, 0.021387100219726562, 0.027478694915771484, 0.024920940399169922]

Input Size: [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]



QUICK SORT MEDIAN 3 :

It picks an element as pivot and partitions the given array around the picked pivot.
Here pivot is the median of first, middle and last element.

The main Data structure of the Quick sort algorithm with 3 medians is the function call –

- choose_median(array,low,middle,high):
    1. This function picks the median among the 3 number – the leftmost , middle, rightmost element in an array.
    2. It compares all the 3 elements in the different if loops in the function and returns the median of the 3 for any type of array.

- swapping(array,c,low,high): c- pivot index
    1. This function does the task of swapping the pivot that is selected as the median of the 3 numbers and putting at it in the end of the array as the rightmost element.
    2. Return the array

- quickSort_3Median (array,low,high)
    1.It takes 3 inputs the array, left most element(low), right most element(high).

2.It checks if the left most element is less than the right most element
- If it is, then partition function is called with passing the array, low, high, take the value of the array[index smaller value]

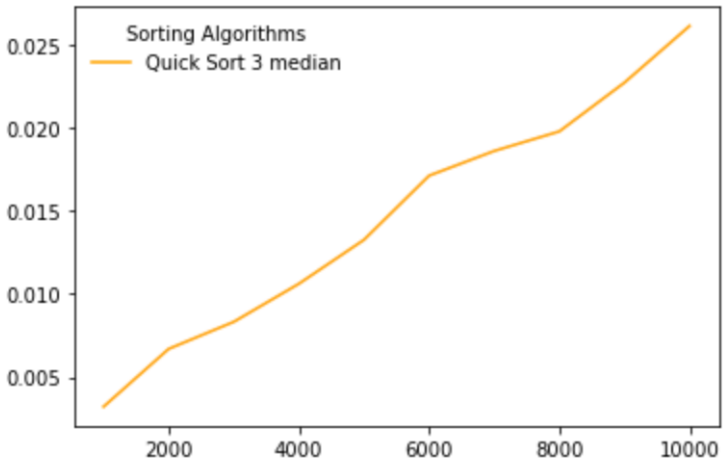
3. return the array

- partition_3Median (array,low,high)
  1.In this algorithm the right most element is taken as pivot.
  2.A for loop is to iterate over the array, and compare each element with the pivot, if it is smaller than the pivot then the two elements current and smaller element are swapped and increment the index of the smaller value otherwise ignore the current element.
  3.Swap array[index smaller value] with array[high]
  5. Return the array[index smaller value]

Experimental results :

Runtime :
[0.003222942352294922, 0.0066950321197509766, 0.00832223892211914, 0.010609865188598633, 0.013252735137939453, 0.017107009887695312, 0.01858997344970703, 0.019765377044677734, 0.022669887924194336, 0.02610301971435547]

Input Size : [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]



Heap Sort :

It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for the remaining elements

The main Data structure of the Heap sort is the function call –
In heap sort we traverse through the elements from the rightmost to the leftmost, whereas most of the other algorithms we do the opposite direction.

- heapify(array,n,i)
  - What the function call does :

  - This function is used to find the largest element, the left child and the right child
  By comparing the elements to the left and the right.
  - Initially the root is taken at the largest element to maintain the max heap.
- Then you swap the values if the root is not the largest when compared with the left and the right child.
- Call the heapify function again

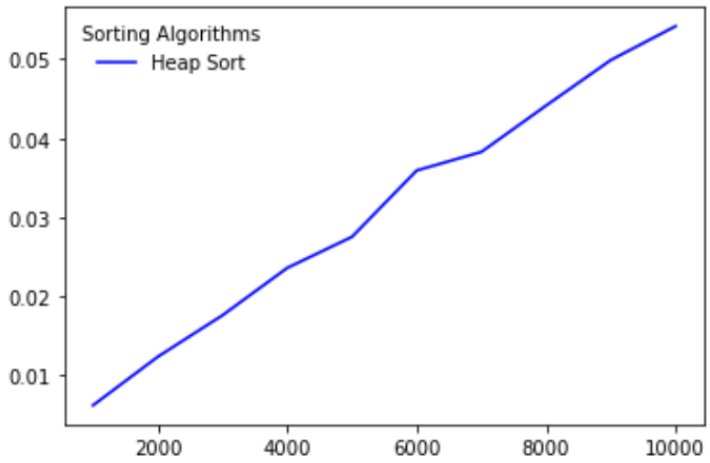
- build_heap(array,n)
  - What the function call does :
    - Build heap is used to heapify each sub tree, and the max heap function is applied to all the elements.

- heap_sort(array,n)
  - What the function call does :
    - you call the function build heap to build the heap.
    - for a from the last element of the array : you swap root element and the last element and call the heapify function.
    - return the array after passing through all the elements in the list.

Experimental results :

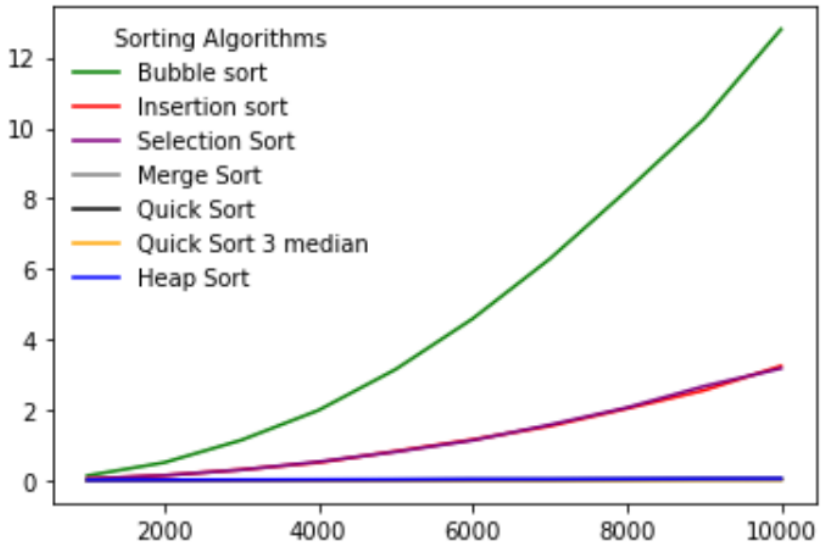
Runtime :
[0.006188869476318359, 0.012333154678344727, 0.017586231231689453, 0.023582935333251953, 0.027513980865478516, 0.03591561317443848, 0.038269996643066406, 0.04414486885070801, 0.049902915954589844, 0.05418801307678223]

Input size : [1000, 2000, 3000, 4000, 5000, 6000, 7000, 8000, 9000, 10000]

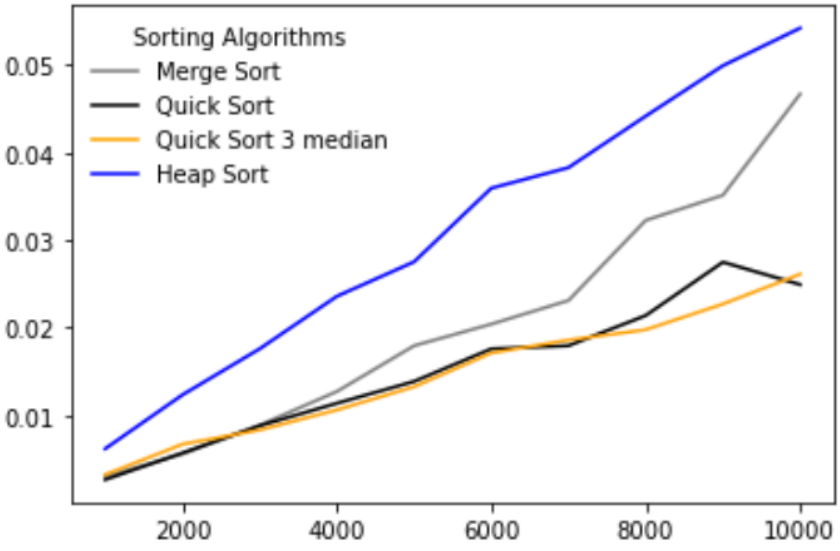


GRAPH COMPARISIONS :

(A) All algorithms time complexity graph :

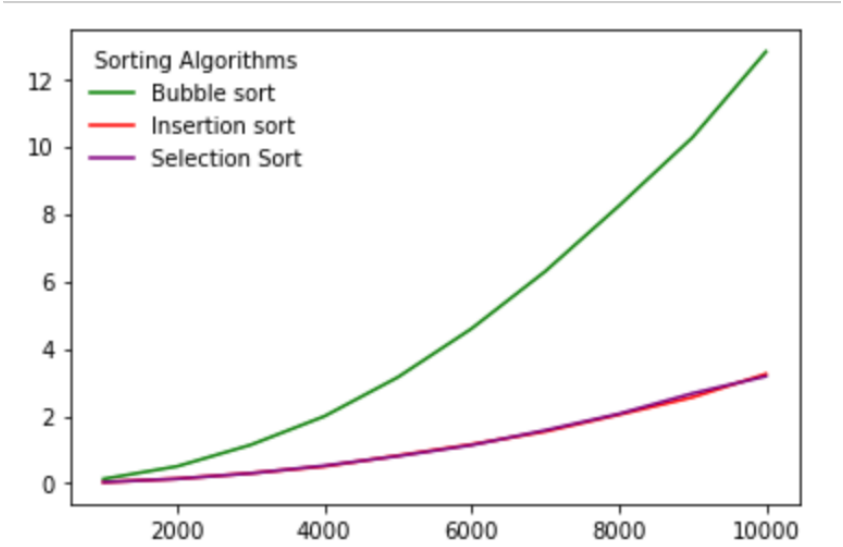


(B) Merge vs Quick Vs Quick Sort 3 median Vs Heap sort



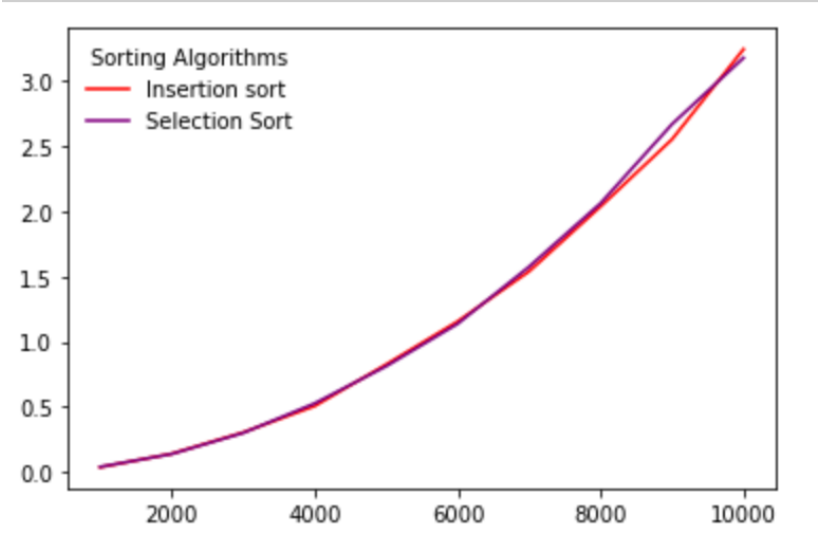
When comparing Merge, Quick, Quick Sort 3 median and Heap sort, it can be seen that Quick Sort 3 median performs the best with large datasets.

(C) Bubble vs selection sort Vs insertion Sort:



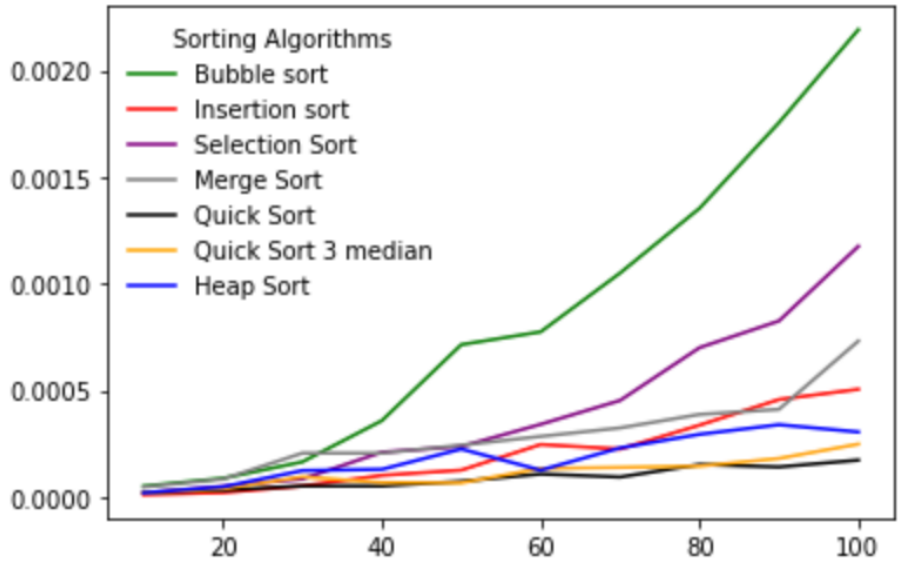
When comparing Bubble, Insertion and selection sort – It is clear that Bubble sort performs the worst when it comes data sets that are large.

(D)selection vs insertion sort :



For a large array set, it can be seen that Insertion sort and Selection Sort almost perform the same, but Insertion is a little better than Selection sort.

For a smaller set the graph looks something like this :



In the case of arrays of sizes between 10 to 100, It can be seen that as the array size keeps increasing there is a clear difference in how each algorithm works.

Hence, we can conclude from the graphs that –

- Bubble sort algorithm is best suitable for small array sizes but not the best.
    - Runtime of bubble sort – $O(n^2)$

- Selection sort algorithm is only better than bubble sort, as it does make unnecessary swaps.
    - Runtime of Selection sort – $O(n^2)$

- Insertion sort is better than bubble sort and Selection sort, but for larger data sets than 100 it can be have a greater increase in runtime, especially if the array is in a reverse order.
    - Runtime of Insertion sort – $O(n^2)$

- Merge Sort is good algorithm to choose, but it is not the best for smaller algorithms. Merge sort performs better with larger data sets than smaller.
    - Runtime of Merge Sort- O(n logn)

- Heap sort algorithm performs better for smaller data sizes when compared to its performance with larger data sets.
    - Runtime of Heap Sort- O(n logn)

- Quick sort with pivot as last element or pivot as median are the best algorithms to choose for any type of data set. As the runtime in both cases is low and optimal.
    - Runtime of Quick Sort - O(n logn)

Sample output :

- For a randomly generated sample of inputs :

```
Do you want to enter the input? Type Y for yes and N for no : N
Enter the size of the input : 10
Enter the start of the range : 0
Enter the end of the range : 10
Your array :  [4, 6, 3, 7, 9, 5, 3, 2, 6, 6]

After Bubble Sort Algorithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for bubble sort is :  0.0000538826

After Insertion Sort Algorithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for Insertion sort is :  0.0000109673

After Selection Sort Algorithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for Selection sort is :  0.0000269413

After Merge Sort Algorithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for merge sort is :  0.0000810623

After Quick Sort Algorithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for quick sort is :  0.0000488758

After Quick Sort Algorithm 3 Median :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for quick sort 3 median is :  0.0000371933

After Heap sort Alogrithm :  [2, 3, 3, 4, 5, 6, 6, 6, 7, 9]
The runtime for Heap Sort  :  0.0000519753
```

For Sorted array :

```
Your array :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]

After Bubble Sort Algorithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for bubble sort is :  0.0002012253

After Insertion Sort Algorithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for Insertion sort is :  0.0001187325

After Selection Sort Algorithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for Selection sort is :  0.0000827312

After Merge Sort Algorithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for merge sort is :  0.0000960827

After Quick Sort Algorithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for quick sort is :  0.0000970364

After Quick Sort Algorithm 3 Median :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for quick sort 3 median is :  0.0002260208

After Heap sort Alogrithm :  [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]
The runtime for Heap Sort  :  0.0000979900
```

Observations from the sample outputs :

In these sample outputs, where the size is the same (i.e. 10) but the input values are different , can affect the overall runtimes of each algorithms and which algorithms performs better in different inputs.

For a sorted array – It can be seen that the order of the runtime complexity for an array of size 10 is Selection sort, Merge sort, Quick sort, Heap sort, Insertion Sort, Insertion sort, Bubble Sort, Quick Sort 3 median in ascending order based on runtimes.

For an Unsorted array - It can be seen that the order of the runtime complexity for an array of size 10 is Insertion Sort, Selection sort, Quick sort 3 median, Quick Sort, Heap Sort, Bubble Sort, Merge sort in ascending order based on runtimes.

For array of a large input size, the inputs we have taken for plotting the graph it can be seen that the order of runtimes are Quick sort 3 median, Quick sort, merge sort, Heap sort, Insertion sort, selection sort, Bubble sort in ascending order based on runtimes.

Hence, it can be seen that for different input sizes, arrangement of the numbers in the input array and the range of the input values affect in choosing the best sorting algorithm to perform.

We can conclude that for a large set of arrays – Picking Quick sort 3 median, Quick sort, merge sort, Heap sort is a good choice.

For a small set of arrays – Picking Bubble Sort ,Insertion Sort, Selection sort is a good choice but not the best.

For a sorted array – Picking Selection sort, Merge sort, Quick sort is a good choice.

Also from the 2 graphs large data sizes Vs small data sizes, it clear that as the sizes increases the runtime increases by a lot for bubble sort, insertion sort and selection when compared with the runtimes of heap sort, merge sort and quick sort.

It is important to note that although the runtime changes for different inputs and different sizes, it is not by a lot, the most efficient algorithm is quick sort and the least efficient one is bubble sort.

A brief structure of the program (project1.py) –

The program's function calls :

* To create the input array -
def create_an_array()

* Bubble sort -
def bubbleSort(array)

* Insertion Sort -
def insertionSort(array)

*Selection Sort -
def SelectionSort(array)

*Merge Sort - (has two function calls)
def merge_sort(array)
def merge(arrayA,arrayB)

* Quick Sort - (has 2 functions)
def quick_sort(array,low,high)
def partition(array,low,high)

* Quick Sort 3 median - (has 4 functions)
def swapping(array,c,low,high)
def choose_median(array,low,middle,high)
def quickSort_3Median(array,low,high)
def partition_3Median(array,low,high)

*Heap Sort - (3 functions)
def heapify(array,n,i)
def build_heap(array,n)
def heap_sort(array,n)

import time – is used to measure the runtime.
Import random – is used to generate random values for the user.