Sorting Techniques

Ali Mohamed – 6268

Mohamed Fahmi – 6157

Youssef Darwish – 6250

- **Programming language:**

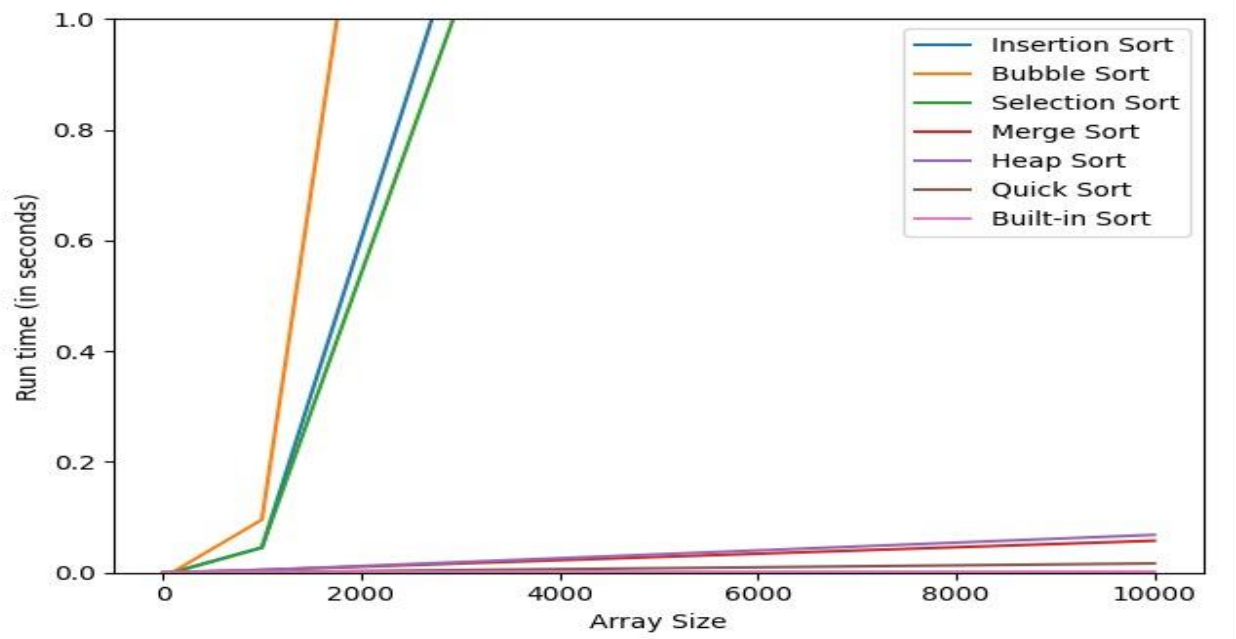
  We used python to implement and design our program, because when it comes to data structures python is one of the best and most commonly used programming languages.

- **Program Description:**

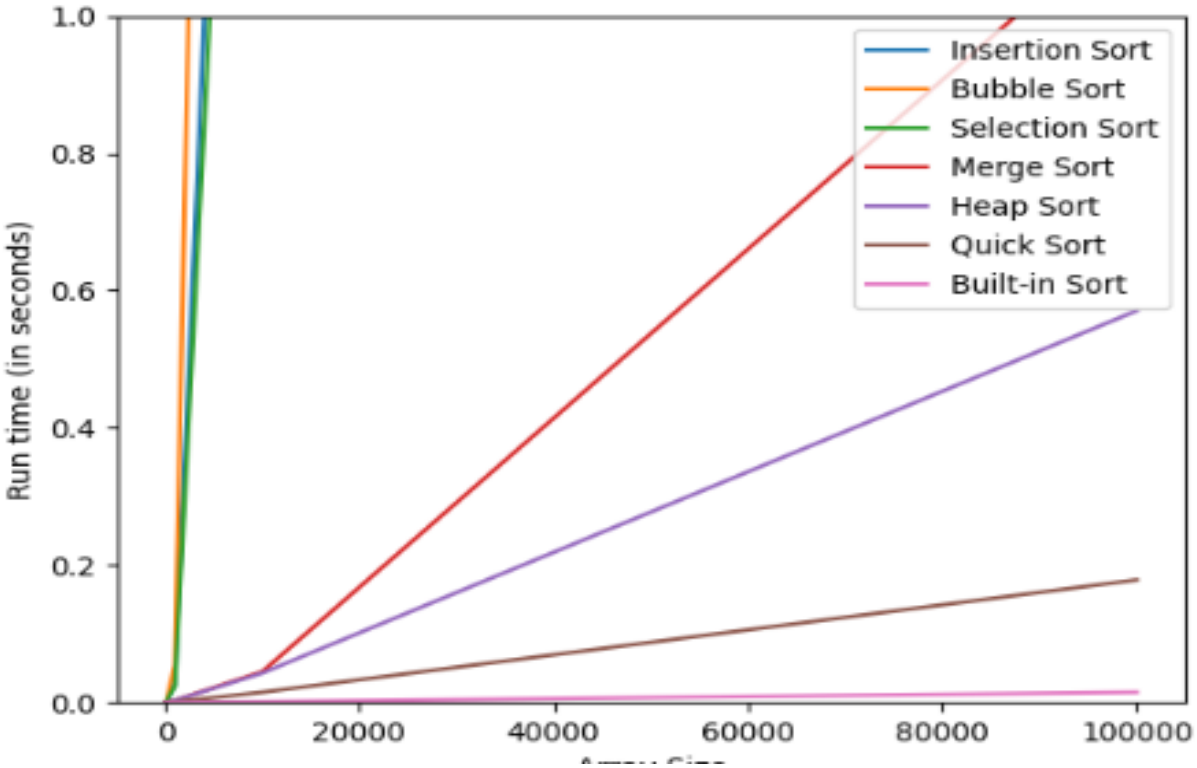
  Our program was developed in python, it creates random arrays of different sizes and sorts them using three O(n^2) algorithms (Bubble sort ,Insertion Sort, Quick Sort), and three O(nlog(n)) algorithms (Quick sort ,Merge Sort ,Heap Sort), and calculates the run time for each of them, we also compared them to the built in python sort which is called Tim Sort, which is a combination of Merge Sort and Insertion Sort, the program then plots the runtimes of each algorithm against the array sizes in a single plot using the matplotlib library so we can see the difference between the run times for each algorithm, hence we could deduce that O(nlog(n)) algorithms are much faster than the O(n^2) ones, also the built in python Tim Sort seemed to be by far the fastest algorithm.

- **graph between Time (seconds) vs Array Size**

array size from 10 to 10k

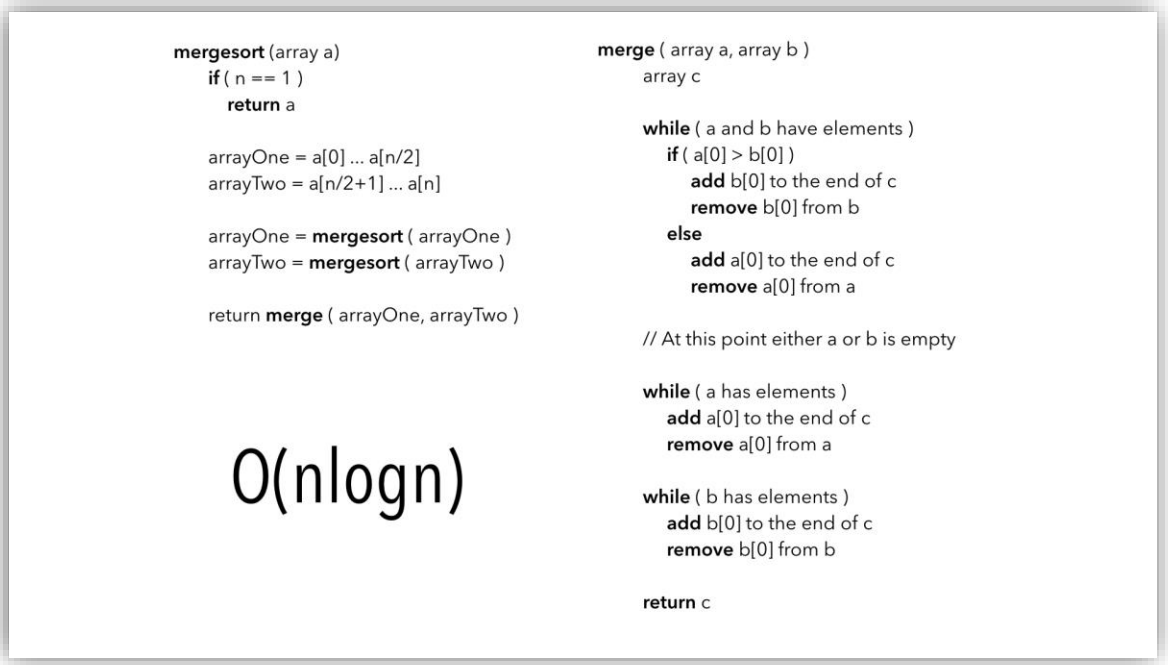


Array size from 100 to 100k



- pseudocode

- merge sort



```
mergesort (array a)
    if ( n == 1 )
      return a

    arrayOne = a[0] ... a[n/2]
    arrayTwo = a[n/2+1] ... a[n]

    arrayOne = mergesort ( arrayOne )
    arrayTwo = mergesort ( arrayTwo )

    return merge ( arrayOne, arrayTwo )
```

O(nlogn)

```
merge ( array a, array b )
    array c

    while ( a and b have elements )
      if ( a[0] > b[0] )
        add b[0] to the end of c
        remove b[0] from b
      else
        add a[0] to the end of c
        remove a[0] from a

    // At this point either a or b is empty

    while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a

    while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b

    return c
```

- quick sort

```
quicksort (array){
  if (array.length > 1){
    choose a pivot;
    while (there are items left in array){
      if (item < pivot)
        put item into subarray1;
      else
        put item into subarray2;
    }
    quicksort(subarray1);
    quicksort(subarray2);
  }
}
```

- heap sort

O(nlogn)

build-max-heap : O(n)
heapify : O(logn), called n–1 times

```
Heapsort (A as array)
    BuildMaxHeap(A)
    for i = n to 1
        swap (A[1], A[i])
        n = n - 1
        Heapify (A, 1)

BuildMaxHeap (A as array)
    n = elements_in(A)
    for i = floor (n/2) to 1
        Heapify (A,i)

Heapify (A as array, i as int)
    left = 2i
    right = 2i+1

    if (left <= n) and (A[left] > A[i])
        max = left
    else
        max = i

    if (right<=n) and (A[right] > A[max])
        max = right

    if (max != i)
        swap (A[i], A[max])
        Heapify (A, max)
```

- selection sort

```
for (j = 0; j < n-1; j++)
    int iMin = j;
    for (i = j+1; i < n; i++)
        if (a[i] < a[iMin])
            iMin = i;
    if (iMin != j)
        swap(a[j], a[iMin]);
```

$O(n^2)$

- insertion sort

```
insertion_sort(array)
        {
        for i=1 to n-1:
                {
                 j=i-1
                 entered=false
                 temp=array[i]
                 while (temp<array[j] && j>=0)
                        {
                         entered=true
                         array[j+1]=array[j]
                         j-=1
                        }
                 if entered:
                   array[j+1]=temp
                }
   }
```

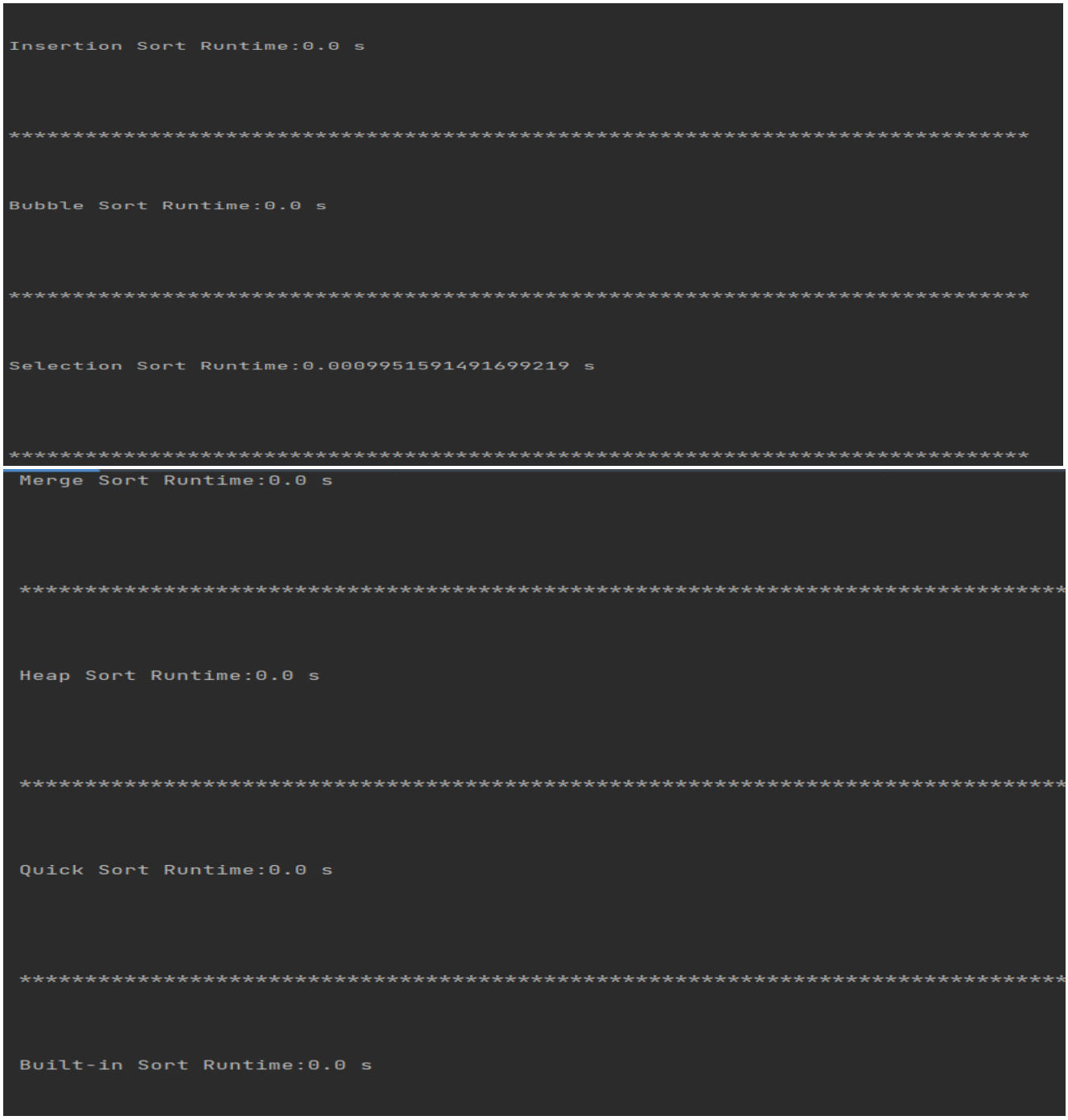
- bubble sort

```
bubble_sort(array)
    {
        for i=0 to n-2:
            {entered=false
                for j=0 to n-i-2:
                    {
                        if array[j+1]<array[j]:
                                {
                                entered=true
                                swap(array[j+1],array[j])
                                }
            }
                if not entered:
                        break
                }
        }
```

Sample run

We will start our samples from 100 as any array less than a 100 element results 0 second run time in all sorting techniques

- Array size – 100 element



- Array size - 1000 element

```
Insertion Sort Runtime:0.026926040649414062 s

**********************************************************************************

Bubble Sort Runtime:0.05385541915893555 s

**********************************************************************************

Selection Sort Runtime:0.02393651008605957 s

**********************************************************************************
```

```
Merge Sort Runtime:0.0029916763305664062 s

**********************************************************************************

Heap Sort Runtime:0.0029914379119873047 s

**********************************************************************************

Quick Sort Runtime:0.0009963512420654297 s

**********************************************************************************

Built-in Sort Runtime:0.0 s
```

- Array size - 10000 element

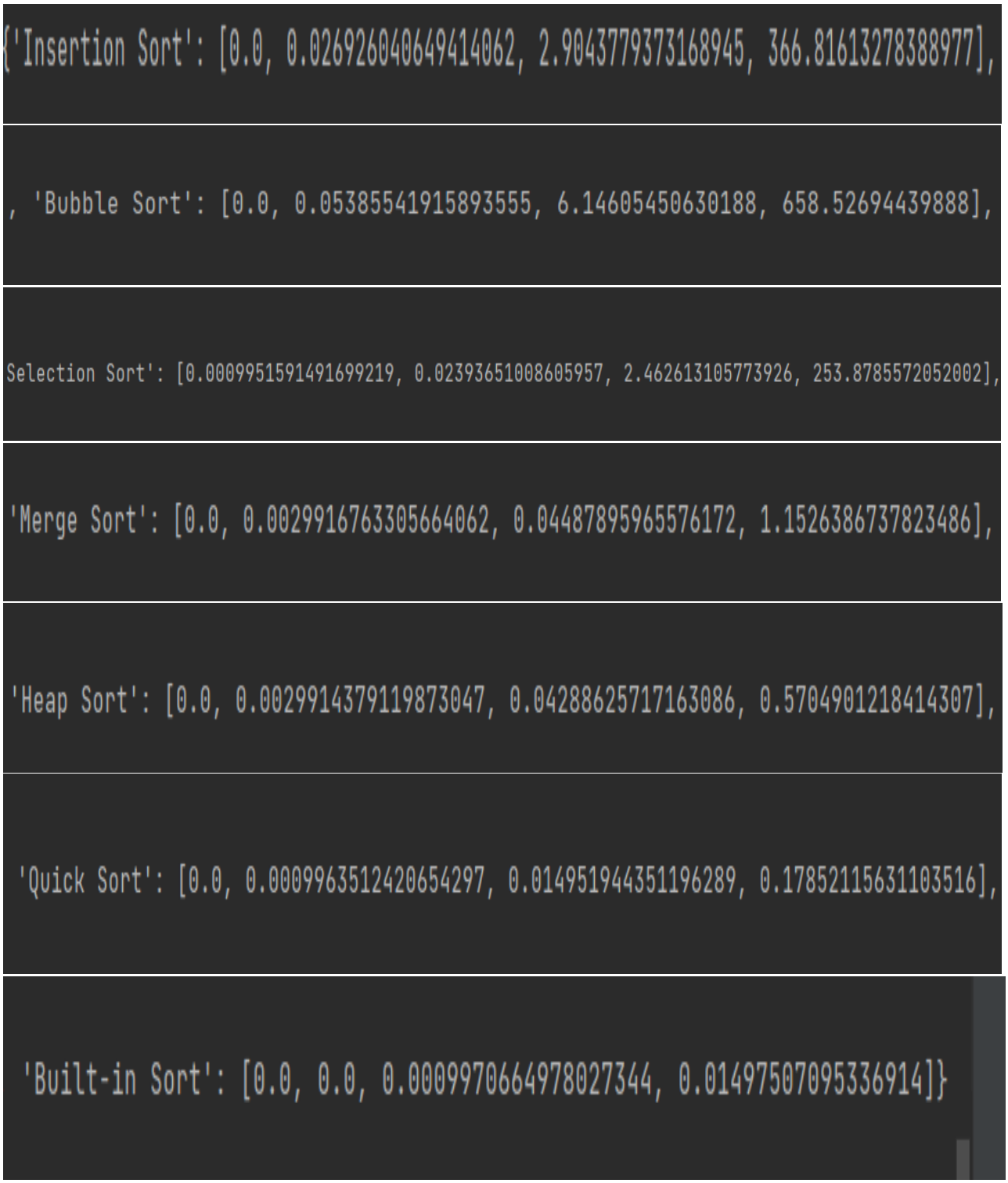
```
Insertion Sort Runtime:2.9043779373168945 s

**********************************************************************************

Bubble Sort Runtime:6.14605450630188 s

**********************************************************************************

Selection Sort Runtime:2.462613105773926 s

**********************************************************************************

Merge Sort Runtime:0.04487895965576172 s

**********************************************************************************

Heap Sort Runtime:0.04288625717163086 s

**********************************************************************************

Quick Sort Runtime:0.014951944351196289 s

**********************************************************************************

Built-in Sort Runtime:0.0009970664978027344 s
```

- Array size of - 100000 element

```
Insertion Sort Runtime:366.81613278388977 s


********************************************************************************


Bubble Sort Runtime:658.52694439888 s


********************************************************************************


Selection Sort Runtime:253.8785572052002 s


********************************************************************************
```

```
Merge Sort Runtime:1.1526386737823486 s


********************************************************************************


Heap Sort Runtime:0.5704901218414307 s


********************************************************************************


Quick Sort Runtime:0.17852115631103516 s


********************************************************************************


Built-in Sort Runtime:0.01497507095336914 s
```

- Array size of [100-1000-10000-100000] compared to each other



Another sample run when number of elements =10

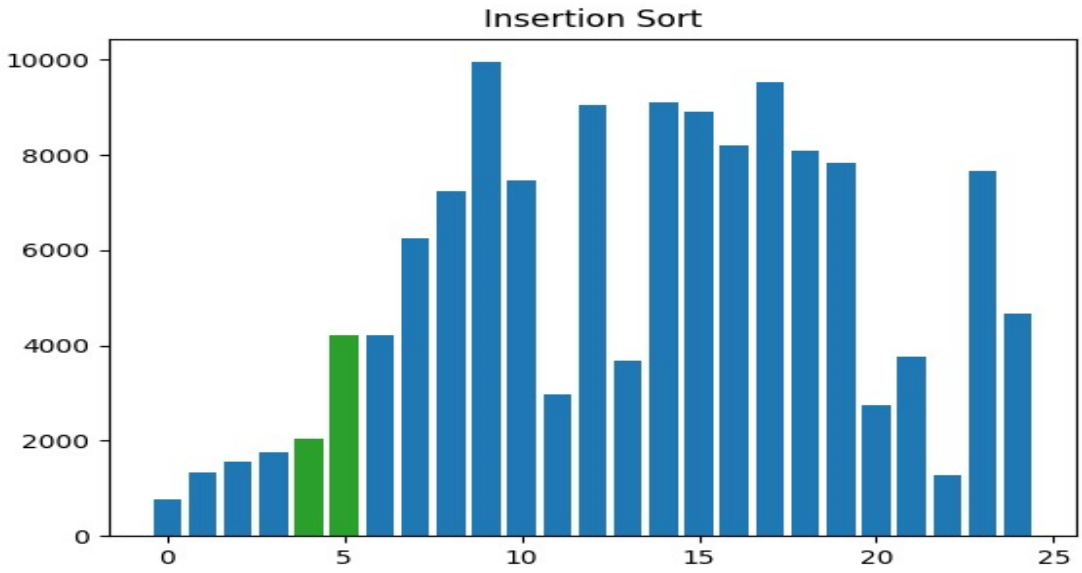
```
--------------------------------------------------> ARRAY SIZE:10 <--------------------------------------------------


Array Before Insertion Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

Array After Insertion Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

Insertion Sort Runtime:0.0 s



********************************************************************************


Array Before Bubble Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

Array After Bubble Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

Bubble Sort Runtime:0.0 s



********************************************************************************
```

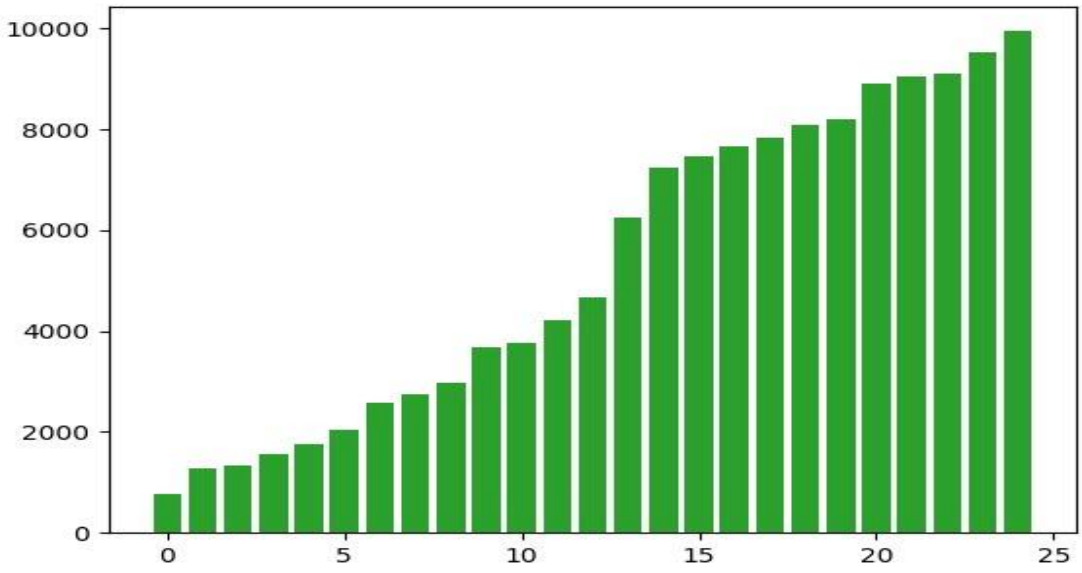
```
 ********************************************************************************


 Array Before Selection Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

 Array After Selection Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

 Selection Sort Runtime:0.0 s



 ********************************************************************************


 Array Before Merge Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

 Array After Merge Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

 Merge Sort Runtime:0.0 s



 ********************************************************************************
```

```
**************************************************************************


Array Before Heap Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

Array After Heap Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

Heap Sort Runtime:0.0 s



**************************************************************************


Array Before Quick Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

Array After Quick Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

Quick Sort Runtime:0.0 s



**************************************************************************
```

```
**************************************************************************


Array Before Built-in Sort:[-2352, -6991, -2235, 3186, 9776, -8422, 3533, 7267, -5234, 7184]

Array After Built-in Sort:[-8422, -6991, -5234, -2352, -2235, 3186, 3533, 7184, 7267, 9776]

Built-in Sort Runtime:0.0 s



**************************************************************************
```

VISUALIZER





An extra visualizer was made but not added to the main code if you would like to see it visit our github .

https://github.com/Alimohamad21/Sorting-Techniques-Visualizer/blob/main/main.py