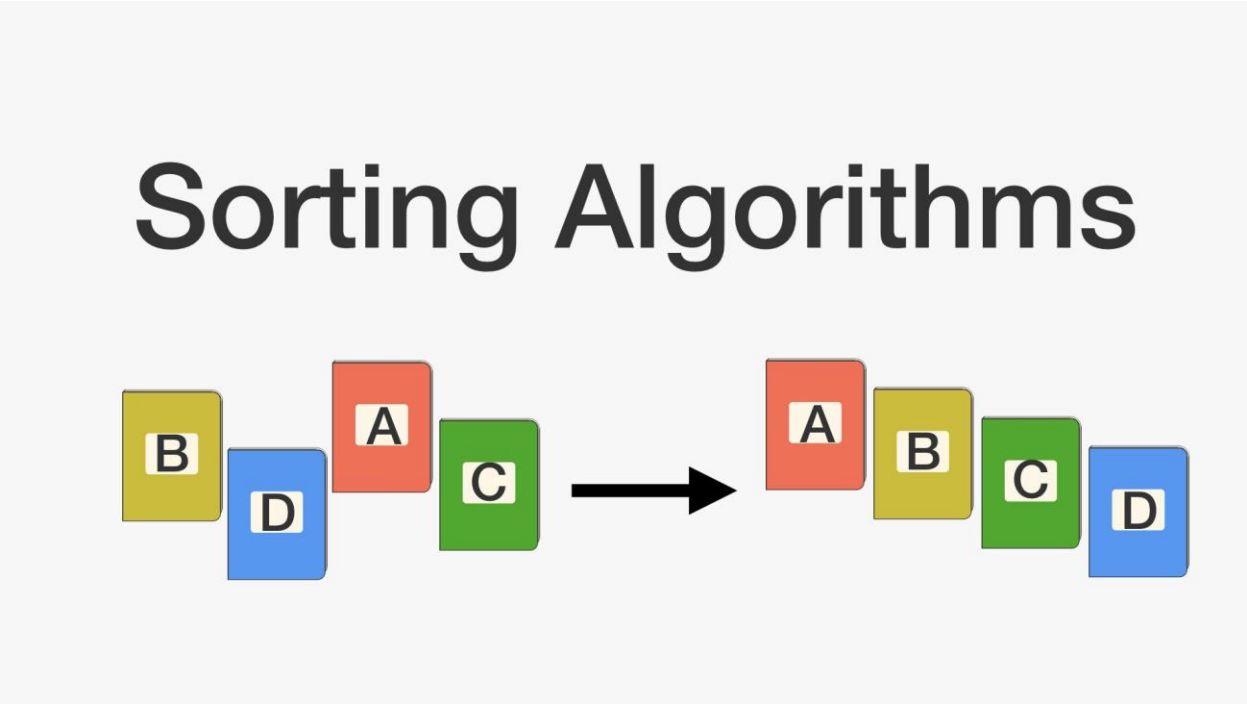
Data Structures II

Assignment 1

Sorting Techniques



By:

Name: Ahmed Wael Mohamed

ID: 6071

Name: Adel Ashraf Mohammed

ID: 6297

Name: Mazen Medhat Farid

ID: 6160

Report Contents

- Description of the program
- Pseudo code for each algorithm
- **Time** vs **Array Size** Graph
- Sample Runs

Program Description

This program is used to calculate the execution time of each sorting algorithm. It also discusses the variation in time between each sorting algorithm and the other when it comes to sorting the same array. Some sorting algorithms take much more time than the others depending on the sorting technique used. The user is asked to enter the number of elements of the array and the program generates a random array of the same number of elements entered by the user and displays the execution time of each sorting algorithm used in the program for the same array. Sorting algorithms used in the program are: Insertion, Heap, Quick, Merge, Bubble and Selection.

Algorithms' Pseudocodes

Bubble Sort (array, n)

```
for (pass = 1 to n-1, pass < n and flag = 0)
{
    flag = 1
    for (j = 0 to n-pass-1)
    {
        If (array[j] > array[j+1])
        {
            swap(array[j], array[j+1])
            flag = 0
        }
    }
}
```

Selection Sort (array, n)

```
for (i = 0 to n-2) {
        min = i
        for (j = i + 1 to n-1) {
                if (a[j] < a[min]) {
                min = j
                }
        }
        Swap (a[i], a[min])
  }
```

Merge Sort (array, left, right)

```
If (left < right)
{
        mid = (left + right) / 2
        Merge Sort (array, left, mid)
        Merge Sort (array, mid+1, right)
        Merge (array, left, mid, right)
 }
```

Merge (array, left, mid, right)

```
a [] = Left array
b [] = Right array
n1 = length (a)
n2 = length (b)
i = 0
j = 0
k = left
```

```
while (i < n1 and j < n2) {
        if (a[i] <= b[j]) {
                c[k] = a[i]
                i++
                k++
        }
        else {
                c[k] = b[j]
                j++
                k++
        }
}
while (i < n1) {
        c[k] = a[i]
        i++
        k++
}
while (j < n2) {
        c[k] = b[j]
        j++
        k++
}
```

Insertion (array)

```
for (I = 1 to length(array))
{
    key = array[i]
    j = i-1
    while (j >= 0 and array[j] > key)
    {
        array[j+1] = array[j]
        j = j-1
    }
    array[j+1] = key
}
```

Quick Sort (array, low index, high index)

```
if (low index < high index)
{
    pivot = partition (array, low index, high index)
    Quick Sort (array, low index, pivot - 1)
    Quick Sort (array, pivot + 1, high index)
}
```

Partition (array, low index, high index)

```
pivot = array [high index]
 i = (low index - 1)
for (j = low index to high index - 1)
{
        if (array [j] < pivot)
        {
                i++
                swap (array [i], array [j])
        }
}
swap (array[i+1], array [high index])
return (i+1)
```

Heap Sort (array, n)

```
Heap Bottom Up (array, n)
for (i = (n/2)-1) to 0)
{
        Max Heapify (array, n, i)
}
for (i = n-1 to 1)
{
        Delete Max (array, i)
}
```

Delete Max (array, n)

```
swap (array [0], array [n])
Max Heapify (array, n, 0)
```

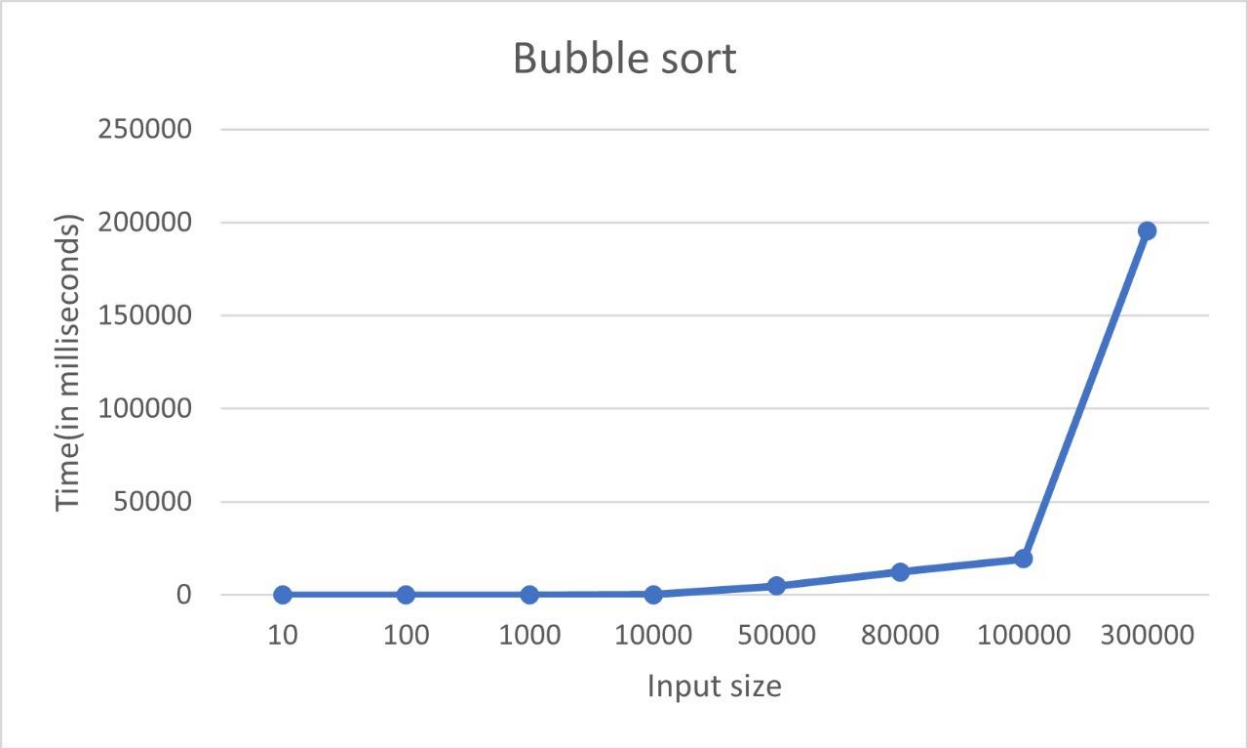
Max Heapify (array, n, i)

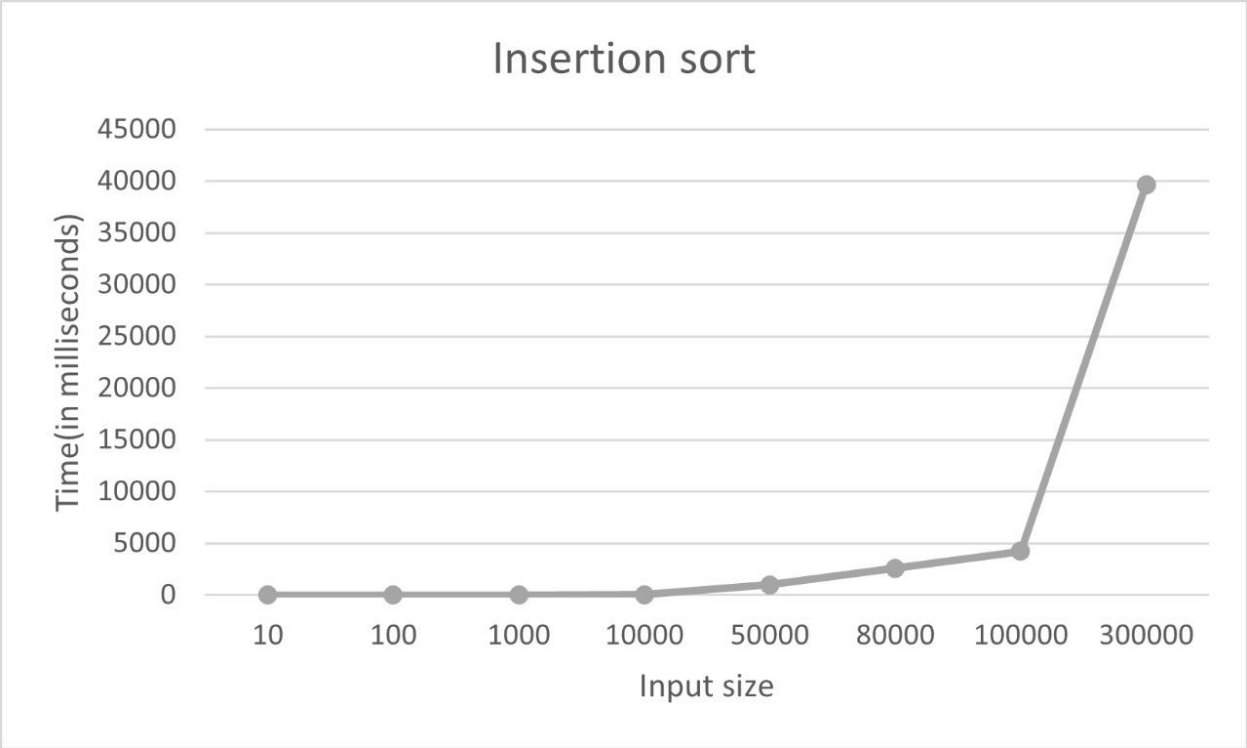
```
largest = i
l = 2*i+1
r = 2*i+2
if (l < n and array[l] > array[largest])
{
        largest = l
}
if (r < n and array[r] > array[largest])
{
        Largest = r
}
if (largest ≠ i)
{
        swap (array[i], array[largest])
        Max Heapify (array, n, largest)
}
```

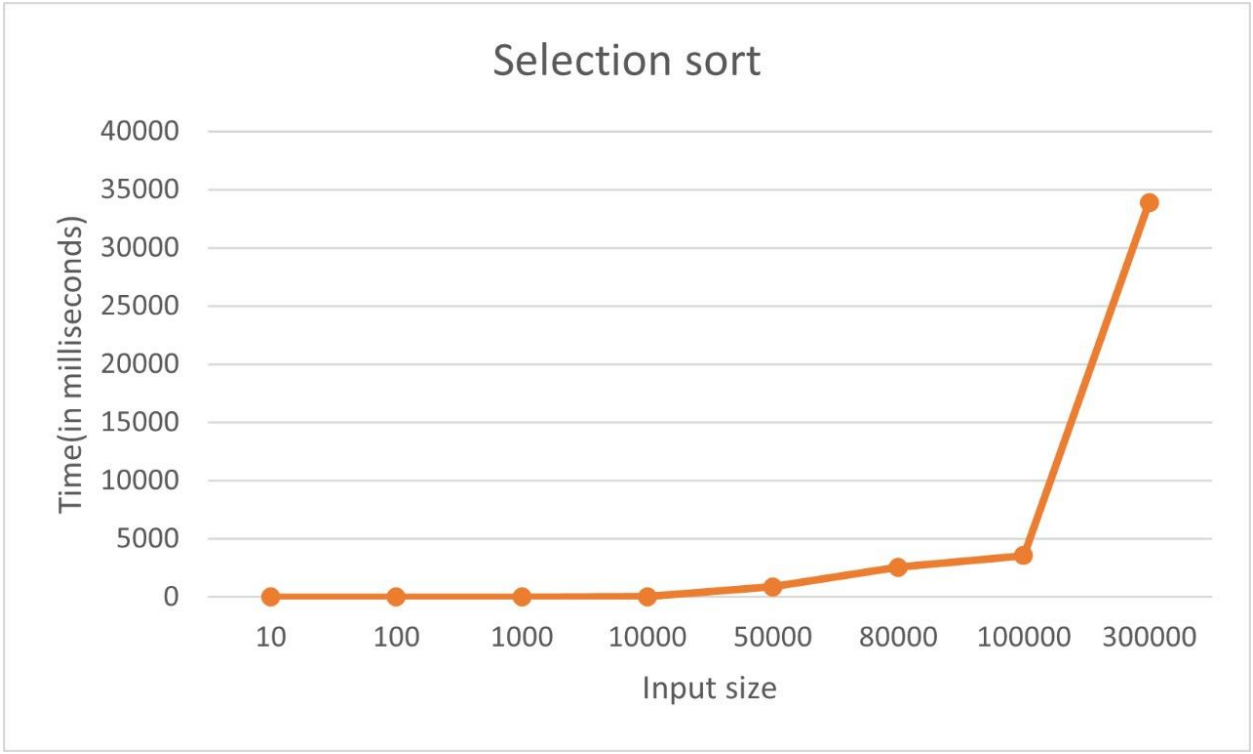
Heap Bottom Up (array, n)

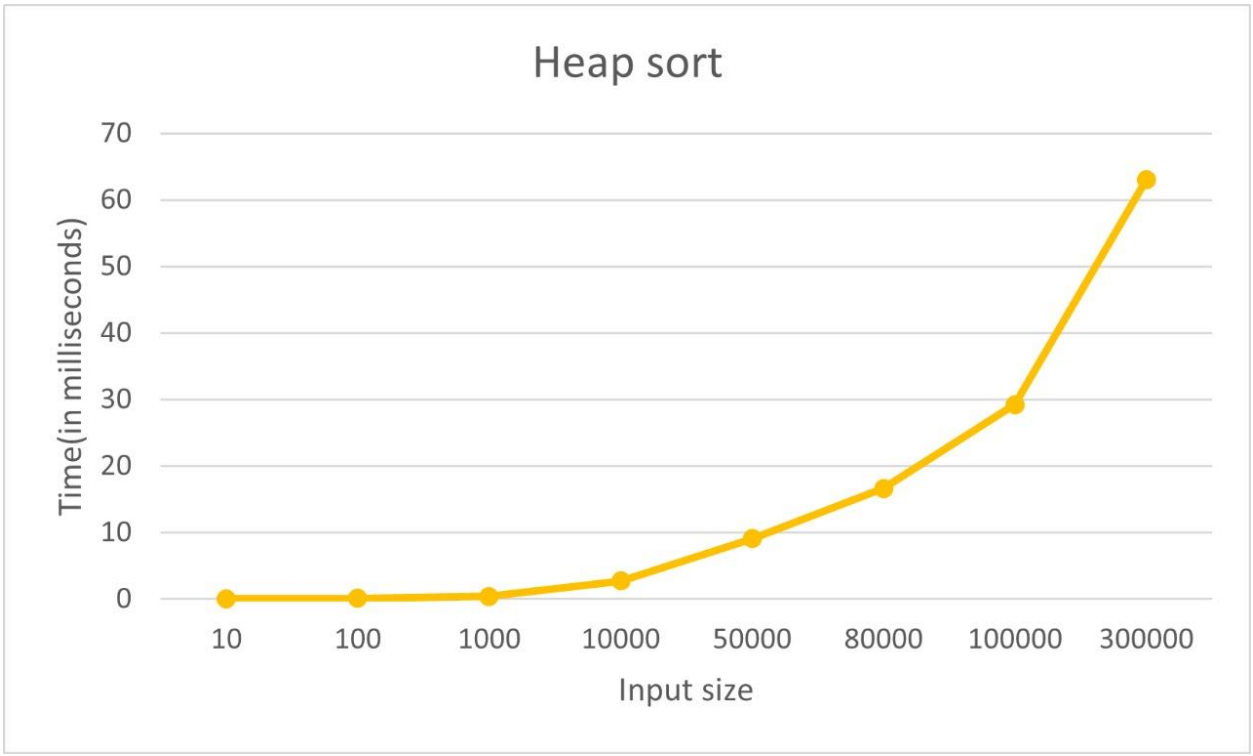
```
for (i = (n/2)-1 to 0)
{
        Max Heapify (array, n, i)
}
```

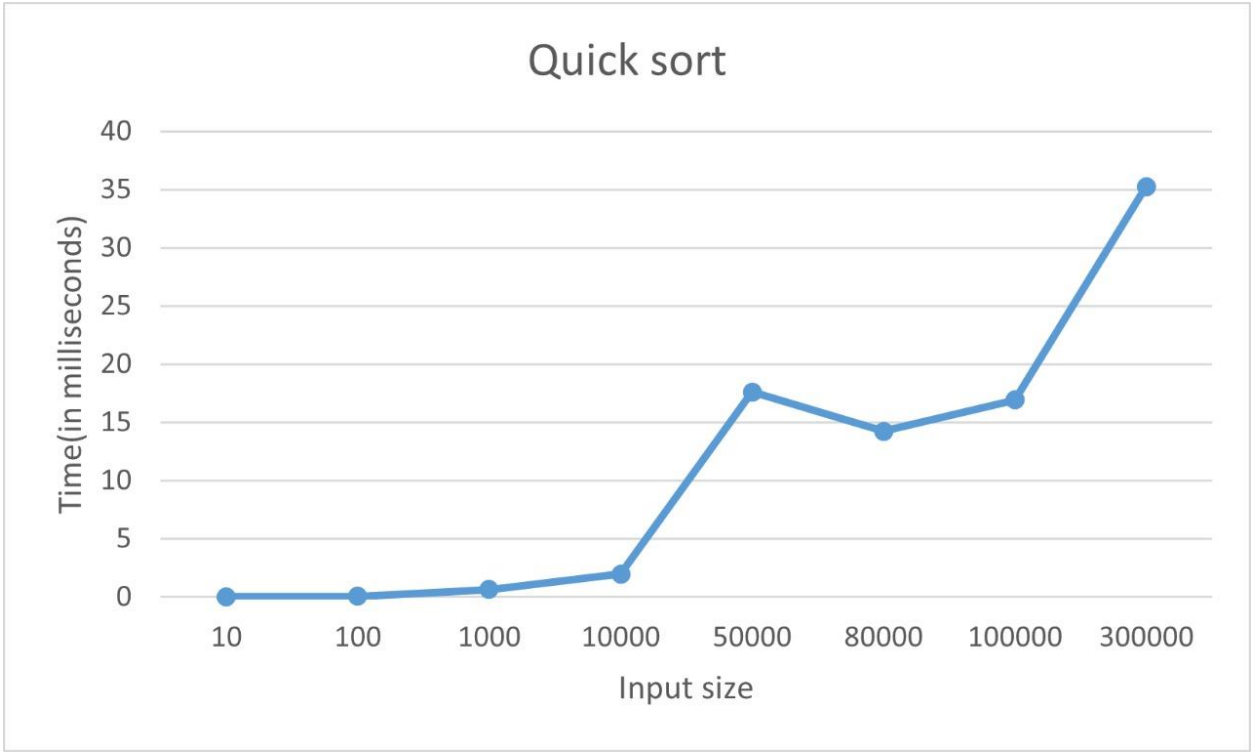
Time vs Array size graphs

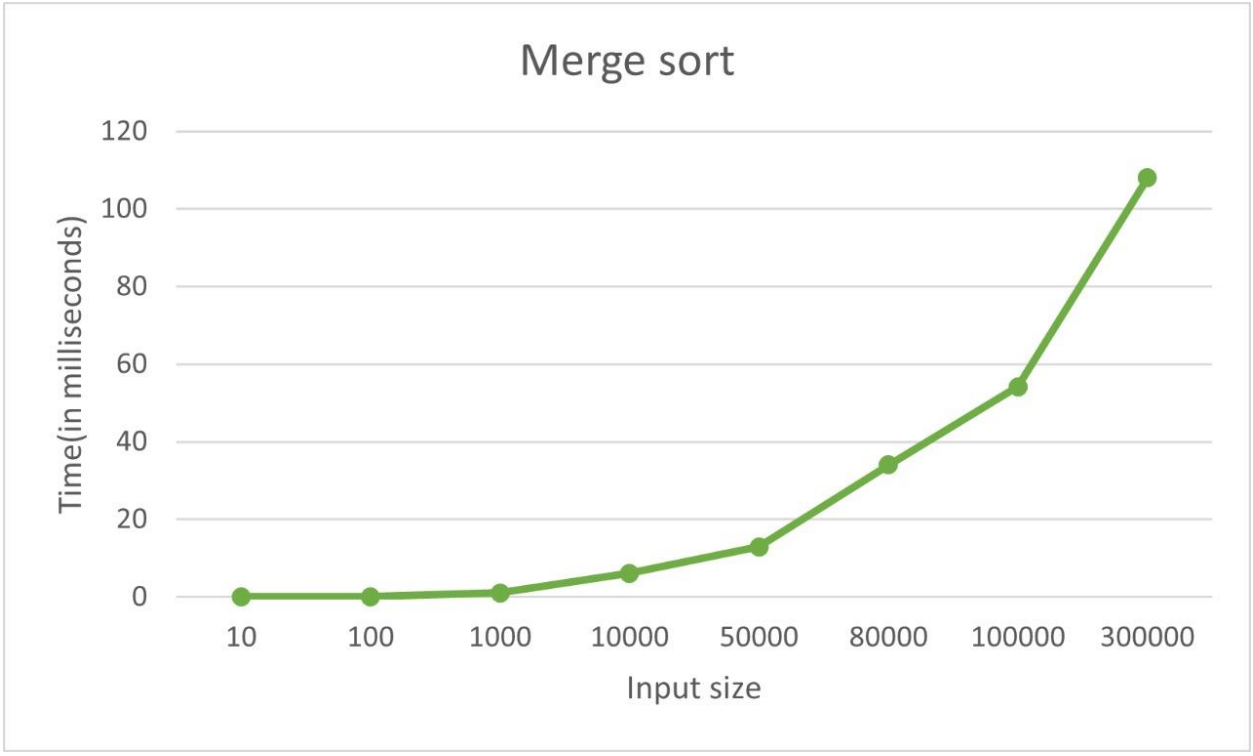


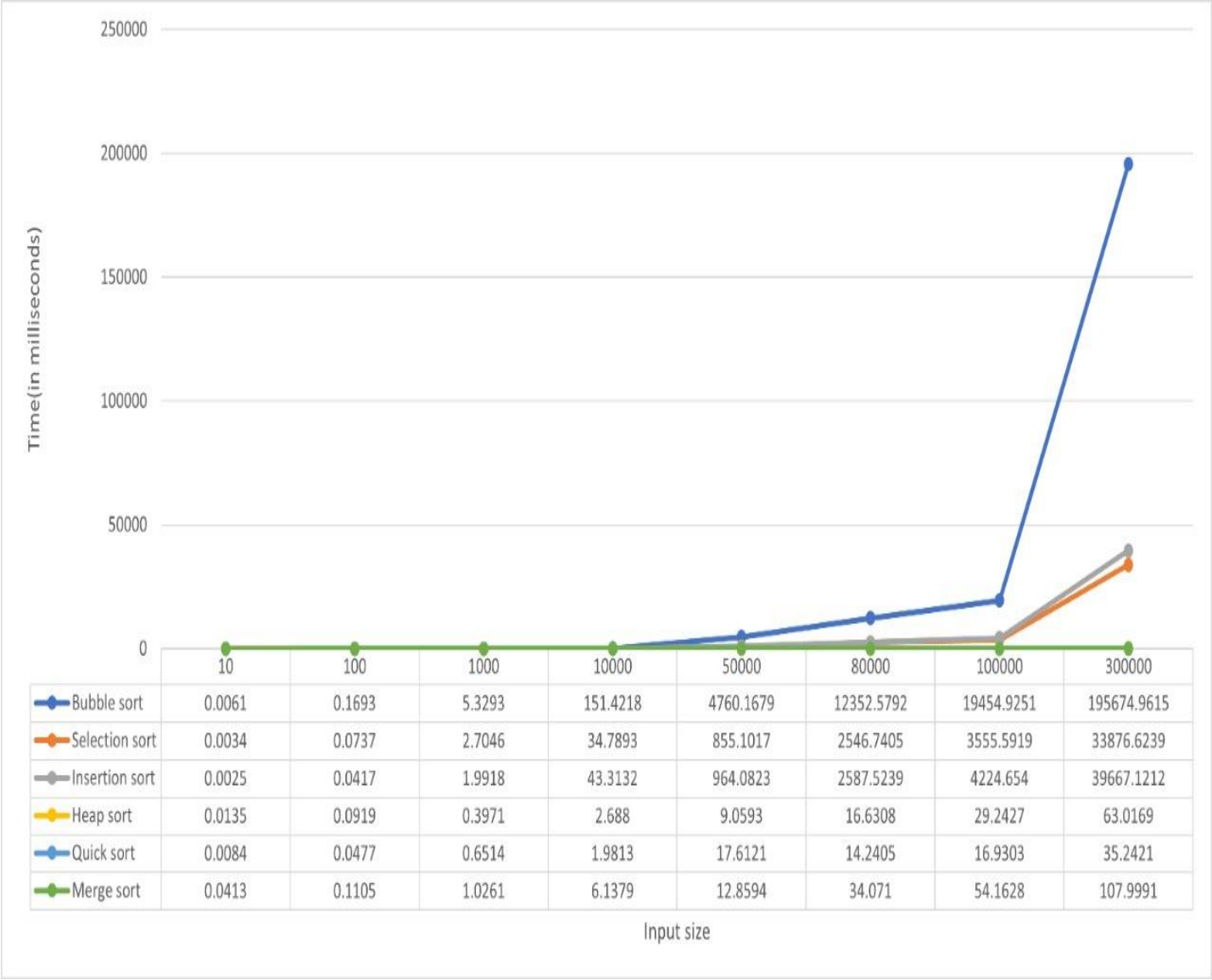



Selection sort
40000
35000
30000
25000
20000
15000
10000
5000
0
Time(in milliseconds)
10
100
1000
10000
50000
80000
100000
300000
Input size


Heap sort
70
60
50
40
30
20
10
0
Time(in milliseconds)
10
100
1000
10000
50000
80000
100000
300000
Input size


Quick sort
Time(in milliseconds)
40
35
30
25
20
15
10
5
0
10
100
1000
10000
50000
80000
100000
300000
Input size


Merge sort
Time(in milliseconds)
120
100
80
60
40
20
0
10
100
1000
10000
50000
80000
100000
300000
Input size



	10	100	1000	10000	50000	80000	100000	300000
Bubble sort	0.0061	0.1693	5.3293	151.4218	4760.1679	12352.5792	19454.9251	195674.9615
Selection sort	0.0034	0.0737	2.7046	34.7893	855.1017	2546.7405	3555.5919	33876.6239
Insertion sort	0.0025	0.0417	1.9918	43.3132	964.0823	2587.5239	4224.654	39667.1212
Heap sort	0.0135	0.0919	0.3971	2.688	9.0593	16.6308	29.2427	63.0169
Quick sort	0.0084	0.0477	0.6514	1.9813	17.6121	14.2405	16.9303	35.2421
Merge sort	0.0413	0.1105	1.0261	6.1379	12.8594	34.071	54.1628	107.9991

Sample runs

Input = 10

```
*** Sorting Techniques Complexities ***
* Enter the number of array elements: 10
* Random array generated.
* Bubble sorting.....
* Bubble sort execution time in milliseconds: 0.0102ms
* Selection sorting.....
* Selection sort execution time in milliseconds: 0.0051ms
* Insertion sorting.....
* Insertion sort execution time in milliseconds: 0.0042ms
* Heap sorting.....
* Heap sort execution time in milliseconds: 0.0191ms
* Quick sorting.....
* Quick sort execution time in milliseconds: 0.0126ms
* Merge sorting.....
* Merge sort execution time in milliseconds: 0.0578ms
*** Thanks! ***
```

Input = 100

```
*** Sorting Techniques Complexities ***
* Enter the number of array elements: 100
* Random array generated.
* Bubble sorting.....
* Bubble sort execution time in milliseconds: 0.1429ms
* Selection sorting.....
* Selection sort execution time in milliseconds: 0.1152ms
* Insertion sorting.....
* Insertion sort execution time in milliseconds: 0.09ms
* Heap sorting.....
* Heap sort execution time in milliseconds: 0.1129ms
* Quick sorting.....
* Quick sort execution time in milliseconds: 0.0286ms
* Merge sorting.....
* Merge sort execution time in milliseconds: 0.1151ms
*** Thanks! ***
```

Input = 1000

```
*** Sorting Techniques Complexities ***
* Enter the number of array elements: 1000
* Random array generated.
* Bubble sorting.....
* Bubble sort execution time in milliseconds: 3.8029ms
* Selection sorting.....
* Selection sort execution time in milliseconds: 2.3104ms
* Insertion sorting.....
* Insertion sort execution time in milliseconds: 2.0037ms
* Heap sorting.....
* Heap sort execution time in milliseconds: 0.4874ms
* Quick sorting.....
* Quick sort execution time in milliseconds: 0.5041ms
* Merge sorting.....
* Merge sort execution time in milliseconds: 1.1957ms
*** Thanks! ***
```

Input = 10000

```
*** Sorting Techniques Complexities ***
* Enter the number of array elements: 10000
* Random array generated.
* Bubble sorting.....
* Bubble sort execution time in milliseconds: 138.3036ms
* Selection sorting.....
* Selection sort execution time in milliseconds: 57.2789ms
* Insertion sorting.....
* Insertion sort execution time in milliseconds: 17.9472ms
* Heap sorting.....
* Heap sort execution time in milliseconds: 2.612ms
* Quick sorting.....
* Quick sort execution time in milliseconds: 1.6508ms
* Merge sorting.....
* Merge sort execution time in milliseconds: 3.9365ms
*** Thanks! ***
```

Input = 100000

```
run:
*** Sorting Techniques Complexities ***
* Enter the number of array elements: 100000
* Random array generated.
* Bubble sorting.....
* Bubble sort execution time in milliseconds: 21175.0389ms
* Selection sorting.....
* Selection sort execution time in milliseconds: 6936.7ms
* Insertion sorting.....
* Insertion sort execution time in milliseconds: 1274.0608ms
* Heap sorting.....
* Heap sort execution time in milliseconds: 19.2741ms
* Quick sorting.....
* Quick sort execution time in milliseconds: 14.7161ms
* Merge sorting.....
* Merge sort execution time in milliseconds: 22.6589ms
*** Thanks! ***
```