EasyChair Preprint

№ 2669

Analysis of sorting techniques a naive approach

Varunkumar Pande

February 15, 2020

Analysis of sorting techniques a naïve approach

Pande Varunkumar
Computer science engineering (C.S.E)
University of Texas at Arlington
(U.T.A)
Arlington, U.S.A.
centrix94@gmail.com

***Abstract*—Study of various sorting techniques and understanding the relation between the input and execution time. The main point of this study is to help determine a sorting algorithm suitable for various nature and size of inputs, in order to efficiently sort a given input.**

I. INTRODUCTION

Sorting is a fundamental operation in many of the complex algorithms, thus a good understanding of different sorting techniques is an important aspect for many of computer science engineering related fields like data science, machine learning, database technologies, etc. In this paper we start with a small introduction on some commonly known sorting techniques, and then dive deep to understand the relation between the input and time taken by these algorithms to sort the provided input. "One size does not fit all!" as this famous quote states, its pretty much similar in the world of sorting algorithms.

II. SORTING TECHNIQUES

A. *Bubblesort:*

It is the easiest and straight forward algorithm to sort a given list. It works by continuously swapping the adjacent elements that are out of order. The out of order pair is usually termed as inversions. Following is a pseudocode for bubblesort:

[1]

```
BUBBLESORT(A)
1  for i = 1 to A.length − 1
2      for j = A.length downto i + 1
3          if A[j] < A[j − 1]
4              exchange A[j] with A[j − 1]
```

For optimization purposes in our implementation we add another condition to check if any swapping has occurred and if not then we stop the execution.

B. Insertionsort:

Insertionsort works by dividing the given array into two parts the sorted to the left and unsorted to the right. Assuming the first element to be sorted it begins sorting from the second element until the last element, and sorts one element at a time. Following is a pseudocode for insertionsort:

[1]

```
INSERTION-SORT(A)
1  for j = 2 to A.length
2      key = A[j]
3      // Insert A[j] into the sorted sequence A[1 .. j − 1
4      i = j − 1
5      while i > 0 and A[i] > key
6          A[i + 1] = A[i]
7          i = i − 1
8      A[i + 1] = key
```

C. *Heapsort:*

Heapsort also works by dividing the given input into sorted and unsorted region. Heapsort uses a special data structure named "Heap". There are mainly two types of heaps that are used in heapsort, the min-heap and the max-heap. For ascending order, we use max-heap and for descending order we use min-heap. Following is the pseudocode for heapsort:

[1]

```
MAX-HEAPIFY(A, i)
1  l = LEFT(i)
2  r = RIGHT(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     MAX-HEAPIFY(A, largest)

BUILD-MAX-HEAP(A)
1  A.heap-size = A.length
2  for i = ⌊A.length/2⌋ downto 1
3      MAX-HEAPIFY(A, i)

HEAPSORT(A)
1  BUILD-MAX-HEAP(A)
2  for i = A.length downto 2
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size − 1
5      MAX-HEAPIFY(A, 1)
```

D. Mergesort:

Mergesort is a well-known example of divide and conquer approach. This algorithm recursively divides the given input into set of two groups, until there is only individual number remaining and then starts merging them recursively in the required sorting order. Also, an important feature of merge sort is that, it is a stable sort i.e. the order of equal elements is the same in the input and output. Following is the pseudocode for mergesort:

[2]

```
MergeSort(arr[], l, r)
If r > l
1.Find the middle point to
divide the array into two
halves:
     middle m = (l + r)/2

2.Call mergeSort for first
half:
  Call mergeSort(arr, l, m)

3.Call mergeSort for second
half:
 Call mergeSort(arr, m+1, r)

4.Merge the two halves sorted
in step 2 and 3:
  Call merge(arr, l, m, r)
```

E. Quicksort:

Quicksort is also based on divide and conquer approach. This algorithm recursively divides the given input into two sets, but the two sets are divided based on the middle element, and later sorted based on this middle number. We use the "Median-of-three" approach to avoid the worst-case selection of middle element which helps us to reduce the number of swaps required. Following is the pseudocode for quicksort:

[1]

```
QUICKSORT(A, p, r)
1  if p < r
2       q = PARTITION(A, p, r)
3       QUICKSORT(A, p, q − 1)
4       QUICKSORT(A, q + 1, r)

PARTITION(A, p, r)
1  x = A[r]
2  i = p − 1
3  for j = p to r − 1
4       if A[j] ≤ x
5            i = i + 1
6            exchange A[i] with A[j]
7  exchange A[i + 1] with A[r]
8  return i + 1
```

The partition method does not contain, logic for median-of-three pivot selection.

III. TOOLS AND SOURCE CODE

Source code for implementing sorting algorithm is available at https://github.com/varunpande/DAA-Project. The project is implemented in "node.js" to provide an interactive interface. Please refer to the "project report.pdf", for detailed instruction on how to install the application locally.

IV. INSIGHTS ON THE RUNTIME AND INPUT

Analysis for a small list of numbers:

1. A sorted list input:

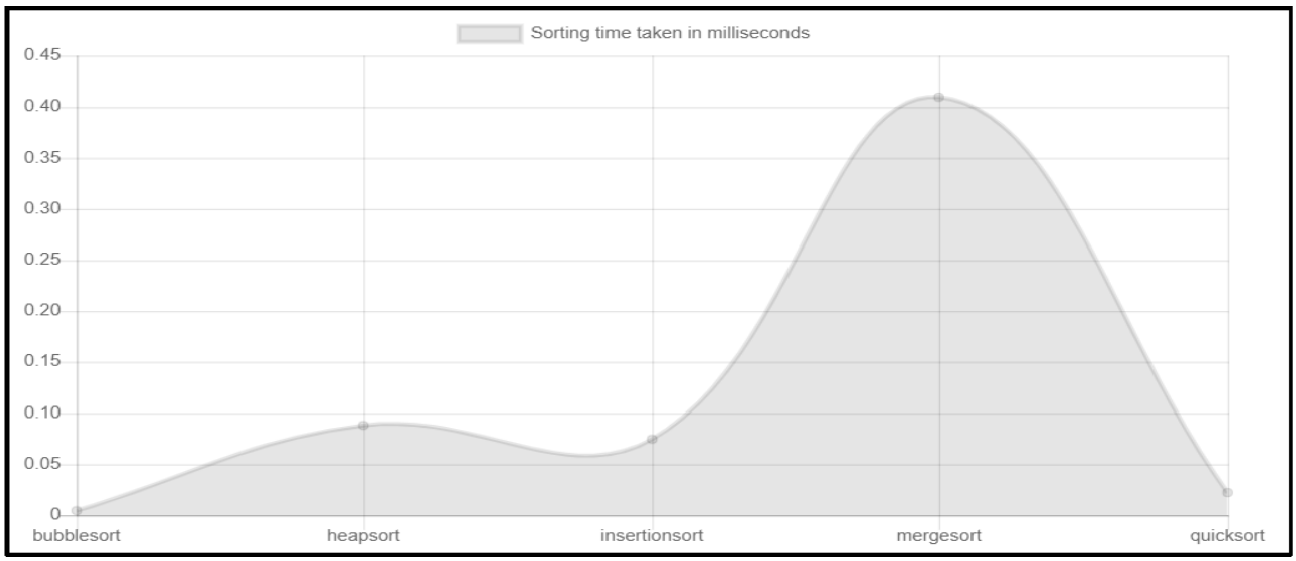

Fig. 1. Graph plot of execution time for sorted small input.

- For a sorted small input (200- integers) bubblesort takes the least time (approx. 0.04 milliseconds) and mergesort takes the worst time (approx. 0.4 milliseconds).
- Quicksort performs better than heapsort, insertionsort and mergesort.
- Mergesort and insertionsort take approximately equal time to sort the list.

2. A reverse sorted list input:

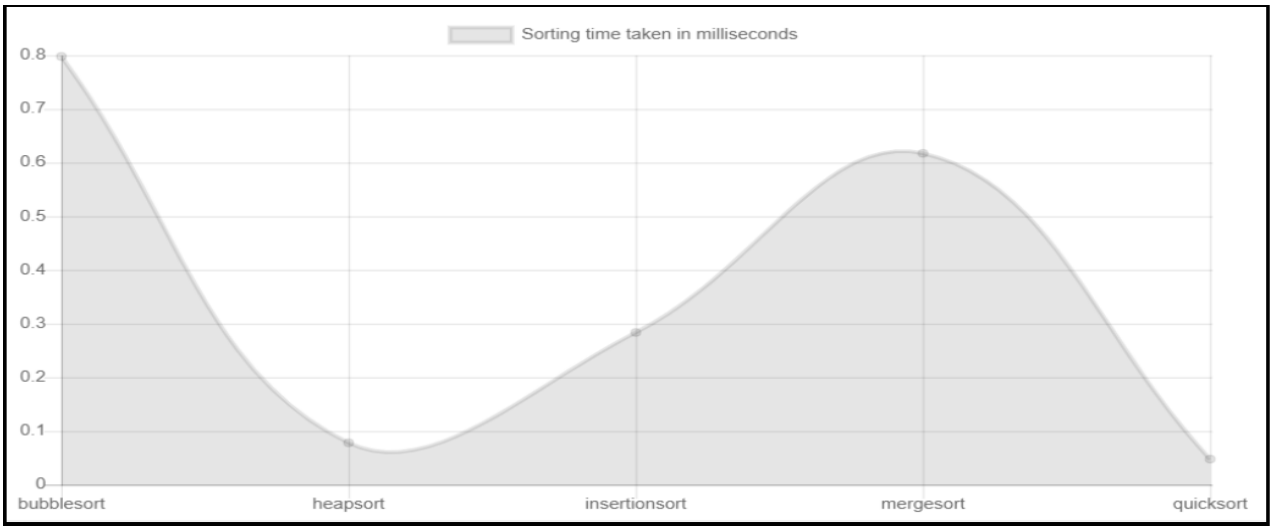

Fig. 2. Graph plot of execution time for reverse sorted small input. (reverse - meaning for sorting in ascending order, descending order input, and vice versa)

- For a reverse sorted small input (200- integers) quicksort takes the least time (approx. 0.039 milliseconds) and bubblesort takes the most time (approx. 0.84 milliseconds).

- heap sort performs second best, mergesort takes the most time after bubblesort and insertionsort takes less time as compared to mergesort to sort the list.

3. A list with repeating numbers:

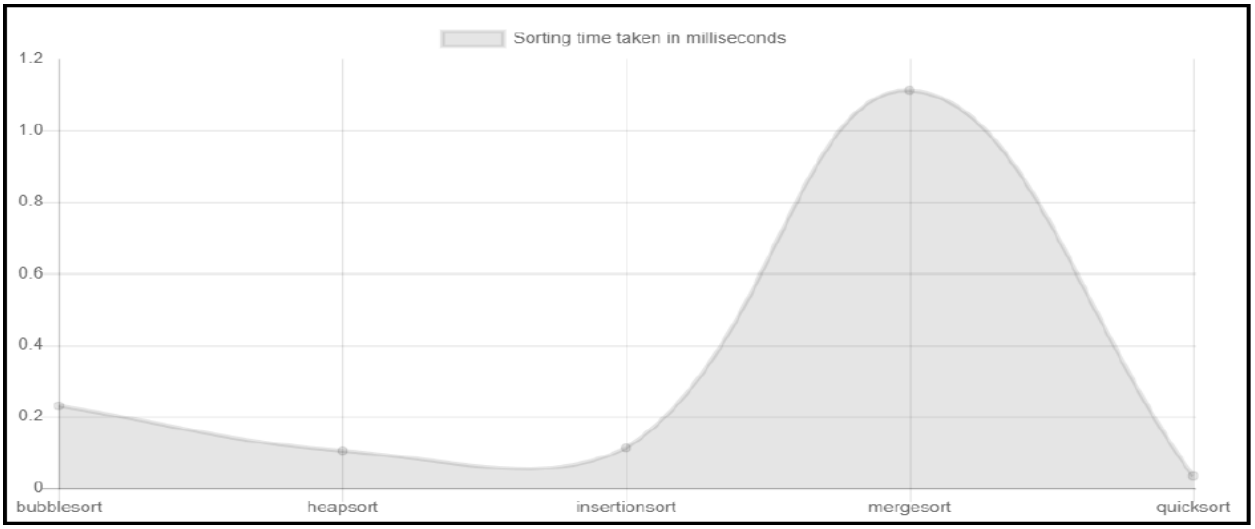

Fig. 3. Graph plot of execution time for small input with repeated numbers.

- For a small input (200- integers) list with repeating numbers quicksort performs best (approx. time taken 0.0321ms) and mergesort has a poor performance (approx. time taken 0.8468 ms).
- Insertionsort and heapsort have a similar runtime.
- Bubblesort performs better than mergesort while numbers are repeated.

4. Unsorted list:

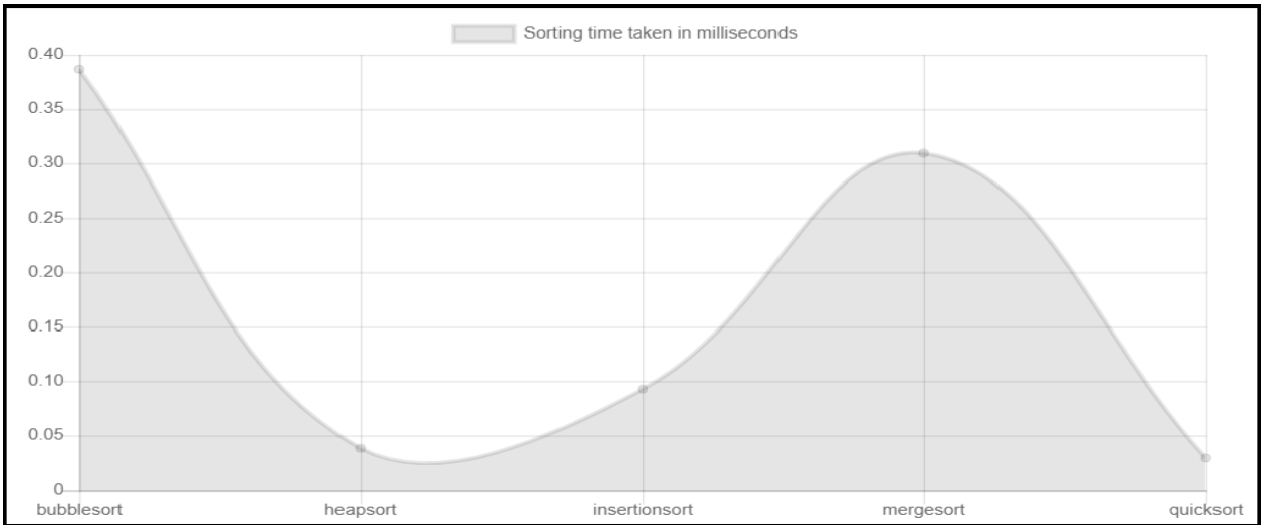

Fig. 4. Graph plot of execution time for unsorted small input.

- For a small input (200- integers) of unsorted numbers quicksort and heapsort perform best (approx. time taken 0.0321ms).
- Bubblesort performs poorly as compared to other algorithms (approx. time taken 0.3867 ms).

  Mergesort takes the most time after bubblesort, and insertionsort performs better than mergesort.

Analysis for a large list of numbers:

1. A sorted list input:

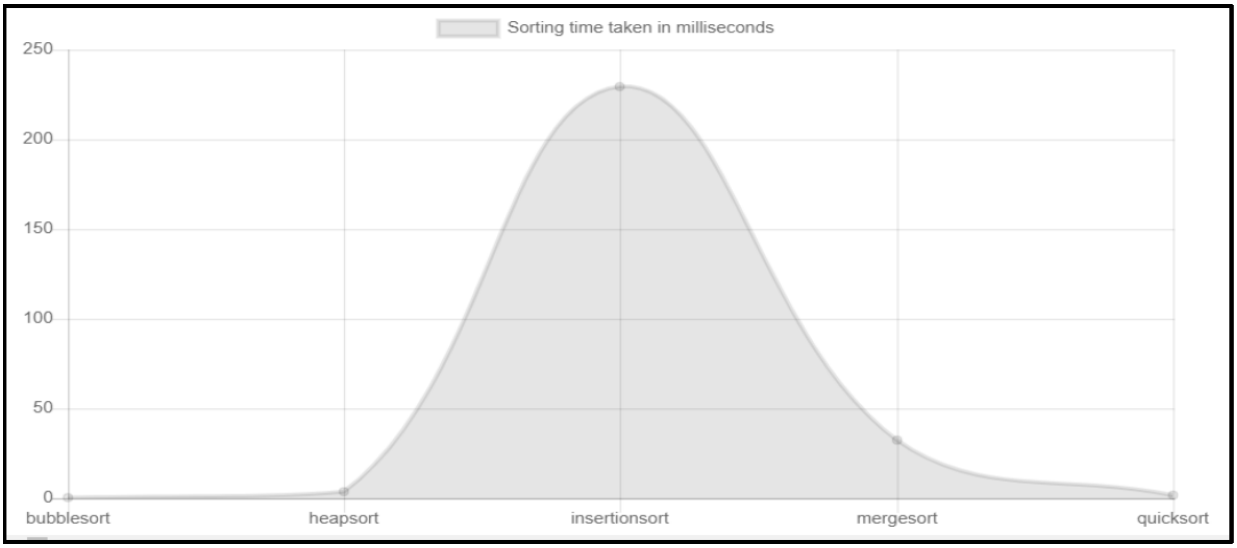

Fig. 5. Graph plot of execution time for sorted large input.

- For a large sorted input (12999- integers) bubblesort takes the least time (approx. 0.3075 milliseconds) and insertionsort takes the worst time (approx. 229.40 milliseconds).
- For other algorithms the order of time taken is as follows:
  Quicksort < Heapsort < Mergesort.

2. A reverse sorted list input:

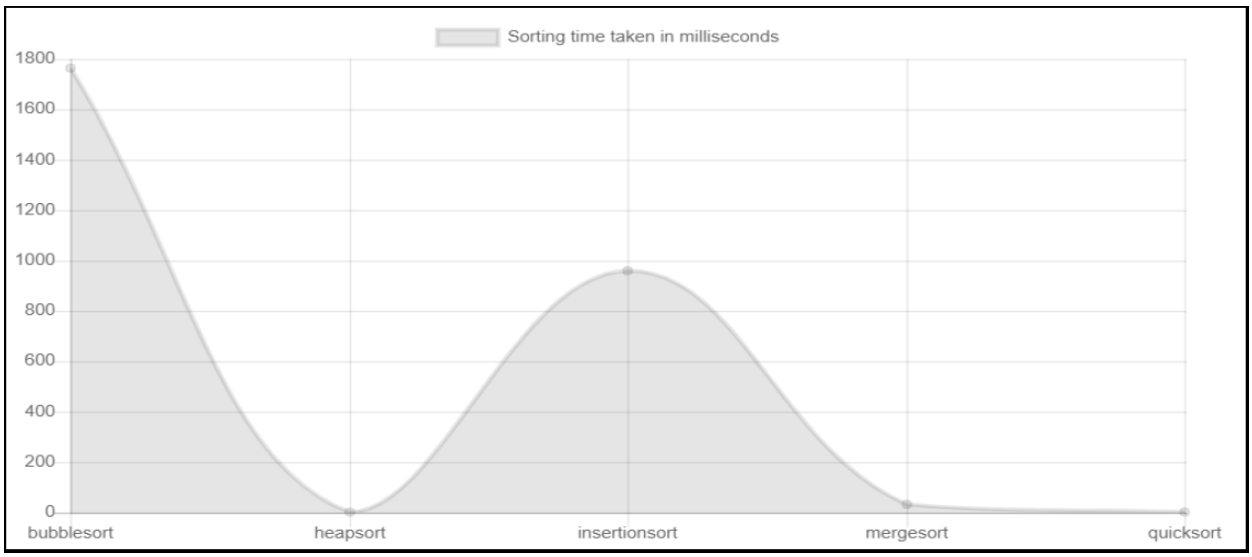

Fig. 6. Graph plot of execution time for reverse sorted large input. (reverse - meaning for sorting in ascending order, descending order input, and vice versa)

- For a sorted list in reverse order quicksort and heapsort perform the best and have almost similar runtime (approx. 1.2 milliseconds), mergesort performs second best.
- Bubblesort takes the most time (approx. 764.15 milliseconds), followed by insertionsort taking

the most time as compared to other algorithms.

3. A list with repeating numbers:

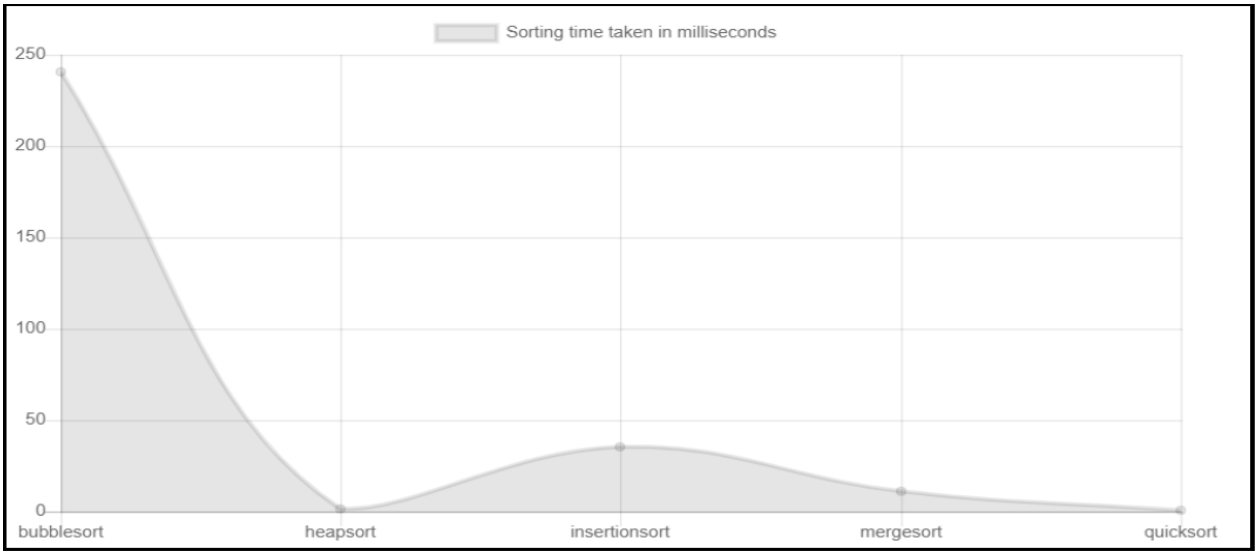

Fig. 7. Graph plot of execution time for large input with repeated numbers.

- For an unsorted list with repeating numbers quicksort performs best (approx. runtime 0.6763 milliseconds) followed by heapsort (approx. runtime 1.39 milliseconds) on an average almost equal to quicksort, bubblesort performs the worst amongst the other algorithms (approx. runtime 239.13 milliseconds).
- Mergesort performs better than insertionsort.

4. Unsorted list:

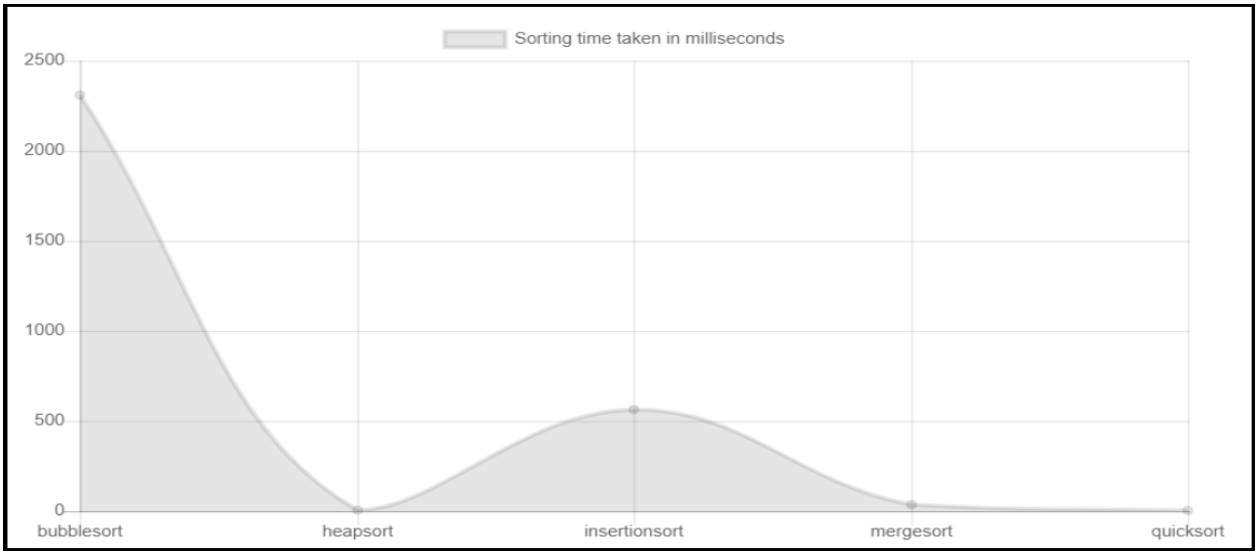

Fig. 8. Graph plot of execution time for unsorted large input.

- From the above graph we can clearly state that the fastest algorithm to sort large amount of numbers (12999- integers) is quicksort (taking approx. 3.36 milliseconds), on an average heapsort performs almost as good as quicksort, but the worst

performing algorithm is bubblesort taking (taking approx. 2328.34 milliseconds).

- Mergesort performs better than insertionsort.

V. CONCLUSION

Performance matrix:

TABLE I. SUMMARY OF SMALL INPUT

Nature of input	Order of time taken to sort data
Sorted input list	Bubblesort < Quicksort < Insertionsort < Heapsort < Mergesort
Reverse sorted list input	Quicksort < Heapsort < Insertionsort < Mergesort < Bubblesort
List with repeating numbers	Quicksort < Heapsort, Insertionsort < Bubblesort < Mergesort
Unsorted list	Heapsort, Quicksort < Insertionsort < Mergesort < Bubblesort

a. The above ordering is based on average of multiple runs.

TABLE II. SUMMARY OF LARGE INPUT

Nature of input	Order of time taken to sort data
Sorted input list	Bubblesort < Quicksort < Heapsort < Mergesort <Insertionsort
Reverse sorted list input	Heapsort, Quicksort < Mergesort < Insertionsort < Bubblesort
List with repeating numbers	Quicksort < Heapsort < Mergesort < Insertionsort < Bubblesort
Unsorted list	Quicksort < Heapsort < Mergesort < Insertionsort < Bubblesort

a. The above ordering is based on average of multiple runs.

Individual algorithm performance for same input (unsorted):

TABLE III. PERFORMANCE OF INDIVIDUAL SORTING ALGORITHM

Sorting Algorithm	Small input		Large input	
	Ascending	***Descending***	***Ascending***	***Descending***
Bubblesort	0.96 ms	0.78 ms	1 s 0.5 ms	9 ms
Heapsort	0.15 ms	0.5 ms	2.4 ms	2.3 ms
Insertionsort	0.18 ms	0.09 ms	315 ms	161 ms
Mergesort	0.2 ms	0.15 ms	20 ms	16 ms
Quicksort	0.13 ms	0.45 ms	2 ms	1.8 ms

a. The above values are an average of multiple runs.

The above table highlights the finding that insertionsort is overall good for small input of unsorted list and quicksort using the median-of-three approach is good for large input of unsorted list. Bubblesort generally performs bad, but is good if there are very few inversion pairs. Heapsort performs best for most of the input if the heapify function is implemented with high efficiency.

ACKNOWLEDGMENT

I would like to thank Senior Lecturer Negin Fraidouni for teaching the subject data structure and algorithm analysis, which helped me understand the concepts of sorting algorithms. Also, Dr. Rajesh Bansode for providing me guidance on publishing this paper.

REFERENCES

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein "Introduction to Algorithms"–3rd ed. pg.18,40,154-160,171.

[2] Mergesort pseudocode provided by - Chitranayal, Mayank Khanna 2 at https://www.geeksforgeeks.org/merge-sort/