

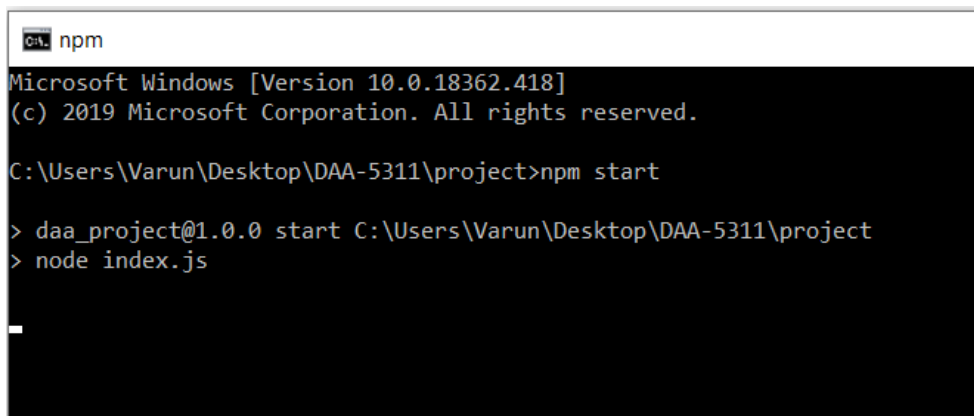
## Project Execution and Setup

### Pre-requisites:

- Node must be installed.
- NPM must be installed.

### Steps to execute the project:

1. Unzip the project.
2. Go to the unzipped project folder.
3. Open a command prompt at the current project folder level.
4. Execute command “npm start”.



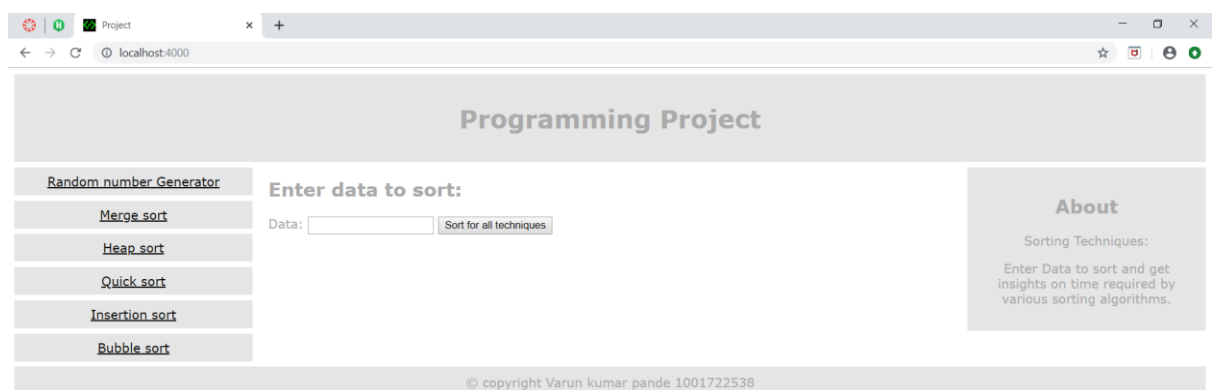
```
C:\> npm

Microsoft Windows [Version 10.0.18362.418]
(c) 2019 Microsoft Corporation. All rights reserved.

C:\Users\Varun\Desktop\DAA-5311\project>npm start

> daa_project@1.0.0 start C:\Users\Varun\Desktop\DAA-5311\project
> node index.js
```

5. A server will start in the command prompt, now open a browser and go to “http://localhost:4000”.
6. You will see the following screen:



7. Now navigate as per your requirement.

## **Main data structures and components**

### **Data structures:**

Arrays: arrays are extensively used in this project as an input to the sorting algorithm and storing the sorted list of numbers.

Number type variables: For storing temporary data for swapping. JavaScript is not a type safe scripting language and hence does not require explicit declaration of integer or floating variable.

### **Main components:**

- Each sorting algorithms is implemented as a function which are separately placed under the “src” folder of the project.
- The index.js file is the entry point of the program.
- For ease of use the project is created as a website which works on express.js server module, but the sorting algorithms can be individually executed irrespective of the express.js module. Instructions for the same are provided under the Important design limitations section.

## Navigation and important Tabs

### Random number generator:

The fields for sorting the numbers takes input of comma separated values, you can use this page to get a random list of numbers. Select all, copy and paste this list into the required sorting technique input field and get sorted data.

The screenshot shows a web browser window with the URL `localhost:4000/public/mg.html`. The page is titled "Random number generator". On the left, there is a navigation panel with links: Home, Random number Generator, Merge sort, Heap sort, Quick sort, Insertion sort, and Bubble sort. The main content area is titled "Please enter data to sort:" and contains input fields for "Upper Limit" (set to 19000), "Lower Limit" (set to 0), and "Number Of Random numbers" (set to 12999). There is also a "Select Type of random number:" section with radio buttons for "Integer" (selected) and "Float". A "Get Random Numbers" button is at the bottom. On the right, there is an "About" section with the text: "Use the random number generator to generate arrays of random number." The footer shows the copyright notice: "© copyright Varun kumar pande 1001722538".

The second screenshot shows the same web browser window, but the "Number Of Random numbers" field is now populated with a long list of random integers: 6452,6023,16880,11254,7493,79,3497,5683,3645,16405,13035,13709,11853,141,3457,18011,7220,2795,14357,3299,1409,16310,9562,5589,3761,692,11037,8566,18456,6897,18632,15924,5567,9657,3979,3548,3142,16165,1972,7896,1.

### Home Tab:

To get the time comparison between all the sorting techniques you can use the home page. Just enter the data into the sorting field and after sometime you will get the details of all the sorting algorithm.

The screenshot shows a web browser window with the URL `localhost:4000`. The page is titled "Programming Project". On the left, there is a navigation panel with links: Random number Generator, Merge sort, Heap sort, Quick sort, Insertion sort, and Bubble sort. The main content area is titled "Enter data to sort:" and contains a "Data:" input field and a "Sort for all techniques" button. On the right, there is an "About" section with the text: "Sorting Techniques: Enter Data to sort and get insights on time required by various sorting algorithms." The footer shows the copyright notice: "© copyright Varun kumar pande 1001722538".

### Navigation Panel:

Use the navigation panel at the left side to navigate to various sorting techniques.

The screenshot shows a vertical navigation panel with a list of sorting techniques: Merge sort, Heap sort, Quick sort, Insertion sort, and Bubble sort. The "Merge sort" option is highlighted.

## Important design limitations

### Input size limitations:

Since the above implementation is done in a server and client infrastructure, the input size for the sorting techniques are limited to the maximum size possible by the request header. Hence the following error might reflect in the terminal:

```
> daa_project@1.0.0 start C:\Users\Varun\Desktop\DAA-5311\project
> node index.js

PayloadTooLargeError: request entity too large
    at readStream (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\raw-body\index.js:155:17)
    at getRawBody (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\raw-body\index.js:108:12)
    at read (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\body-parser\lib\read.js:77:3)
    at urlencodedParser (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\body-parser\lib\types\urlencoded.js:116:5)
    at Layer.handle [as handle_request] (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\layer.js:95:5)
    at next (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\route.js:137:13)
    at Route.dispatch (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\route.js:112:3)
    at Layer.handle [as handle_request] (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\layer.js:95:5)
    at C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\index.js:281:22
    at param (C:\Users\Varun\Desktop\DAA-5311\project\node_modules\express\lib\router\index.js:354:14)
```

In such a case, kindly reduce the list size and continue.

### Overcoming the issue of size limitation:

In order to still be able to sort the given input we can edit the source file as follows and run the sorting algorithm individually.

1. Open the src folder under the project folder.
2. Now open the source file of the sorting algorithm you want to execute.
3. Enter the following command:

“console.log(exports.<sorting technique function name>(list\_to\_sort,order of sorting));”

Eg: For bubblesorting.

```
// record end time of algorithm
let hrend = process.hrtime(hrstart);
let bubblesort_data = {};
bubblesort_data.executionTime = 'Execution time: ' + hrend[0] + ' s ' + hrend[1] / 1000000 + ' ms';
bubblesort_data.rawTime = hrend[0]*1000 + hrend[1] / 1000000;
bubblesort_data.sortedData = inputArray;
bubblesort_data.swaps = number_of_swaps_per_stage;
return bubblesort_data;

console.log(exports.bubblesorting([1,2,3,4,5],0));

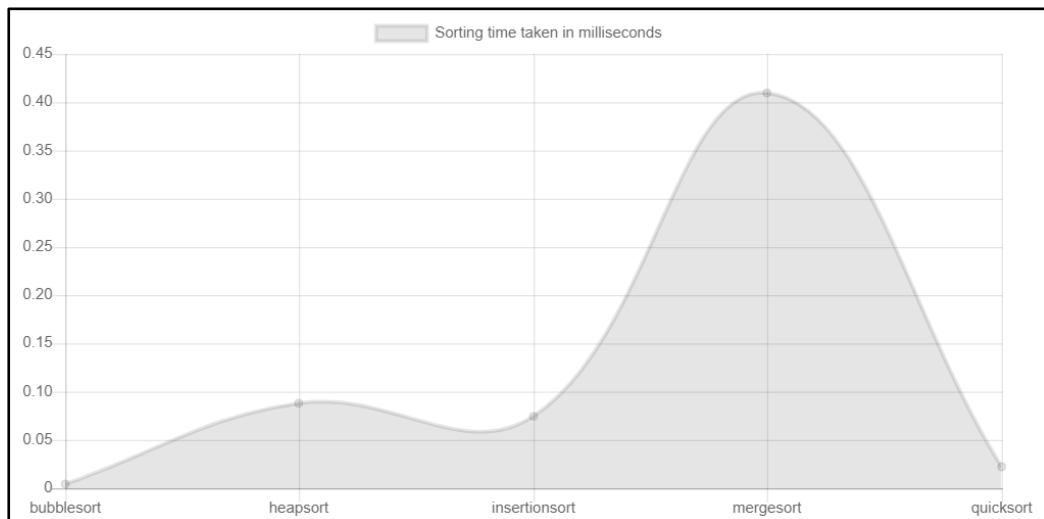
module.exports
```

4. The output will be visible on the terminal.

### Some important insights on sorting techniques

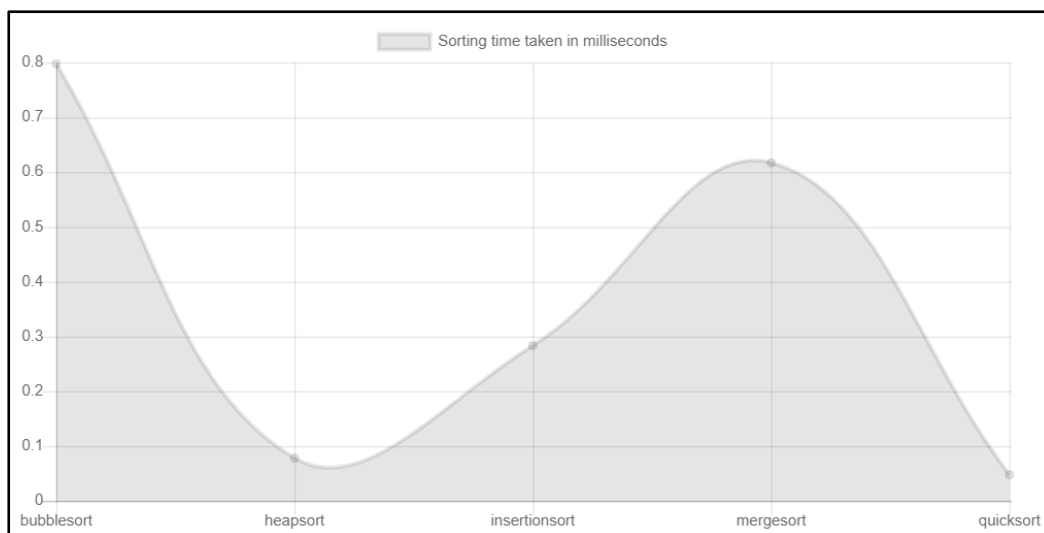
#### A small list of numbers:

- A sorted list input:



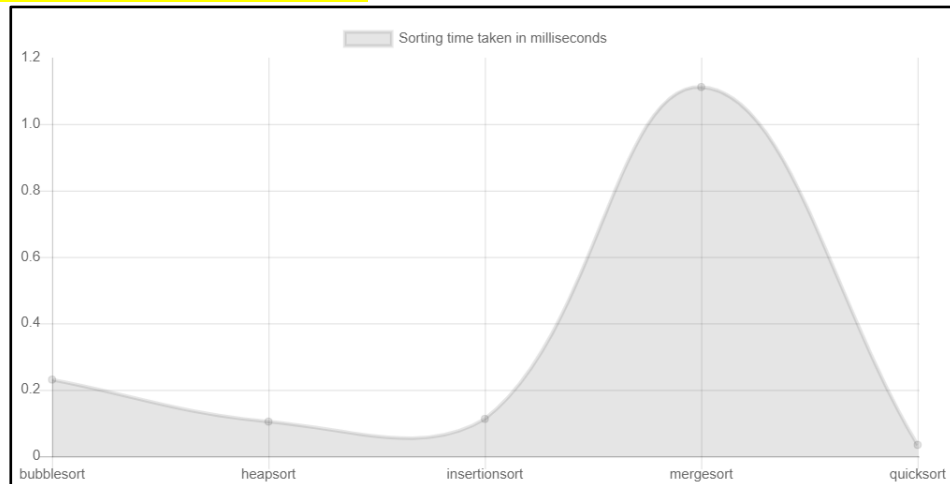
1. For a sorted small input (200- integers) bubblesort takes the least time (approx 0.04 milliseconds) and mergesort takes the worst time (approx 0.4 milliseconds).
2. Quicksort performs better than heapsort, insertionsort and mergesort.
3. Mergesort and insertionsort take approximately equal time to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

- A reverse sorted list input:



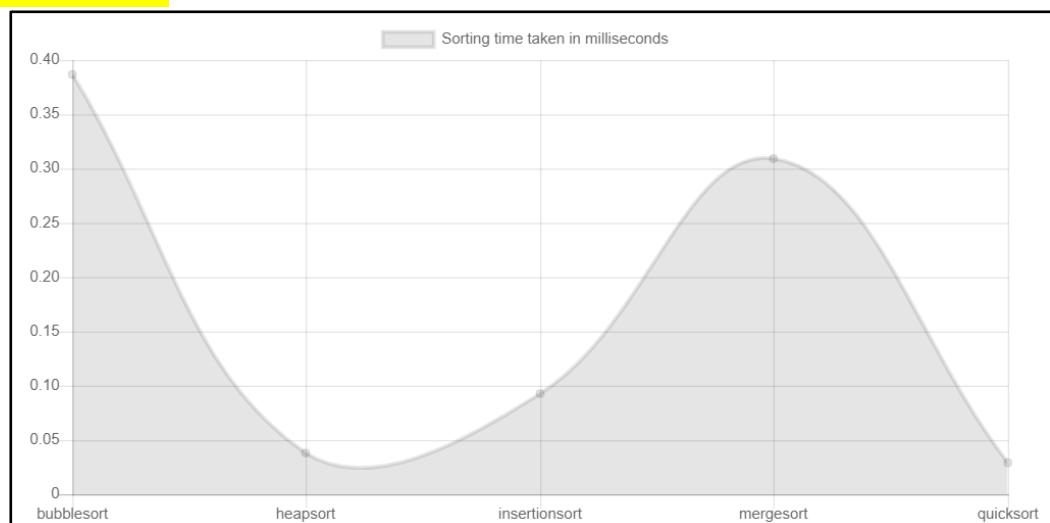
1. For a reverse sorted small input (200- integers) quicksort takes the least time (approx 0.039 milliseconds) and bubblesort takes the most time (approx 0.84 milliseconds).
2. heap sort performs second best, mergesort takes the most time after bubblesort and insertionsort takes less time as compared to mergesort to sort the list.

- A list with repeating numbers:



1. 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).
2. Insertionsort and heapsort have a similar runtime.
3. Bubblesort performs better than mergesort while numbers are repeated.

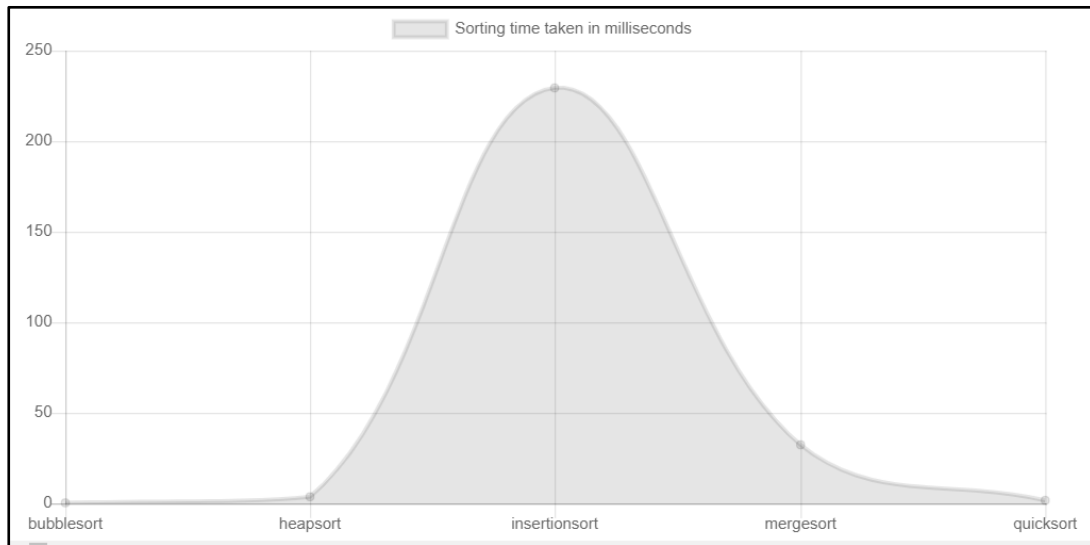
- Unsorted list:



1. For a small input (200- integers) of unsorted numbers quicksort and heapsort perform best (approx time taken 0.0321ms).
2. Bubblesort performs poorly as compared to other algorithms (approx time taken 0.3867 ms).
3. Mergesort takes the most time after bubblesort and insertion sort performs better than mergesort.

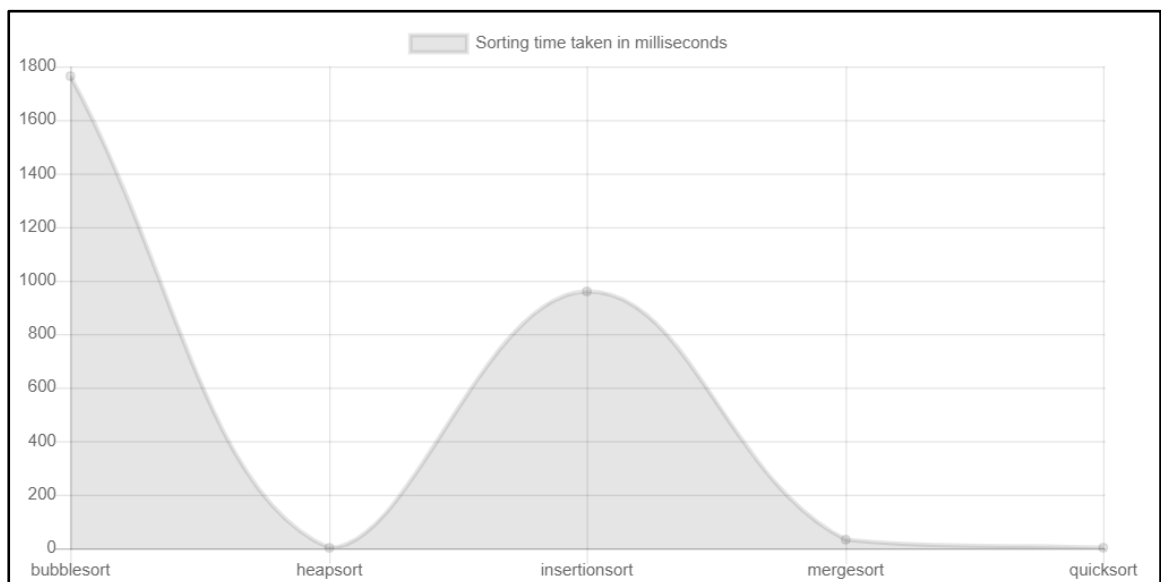
**A large list of numbers:**

- A sorted list input:



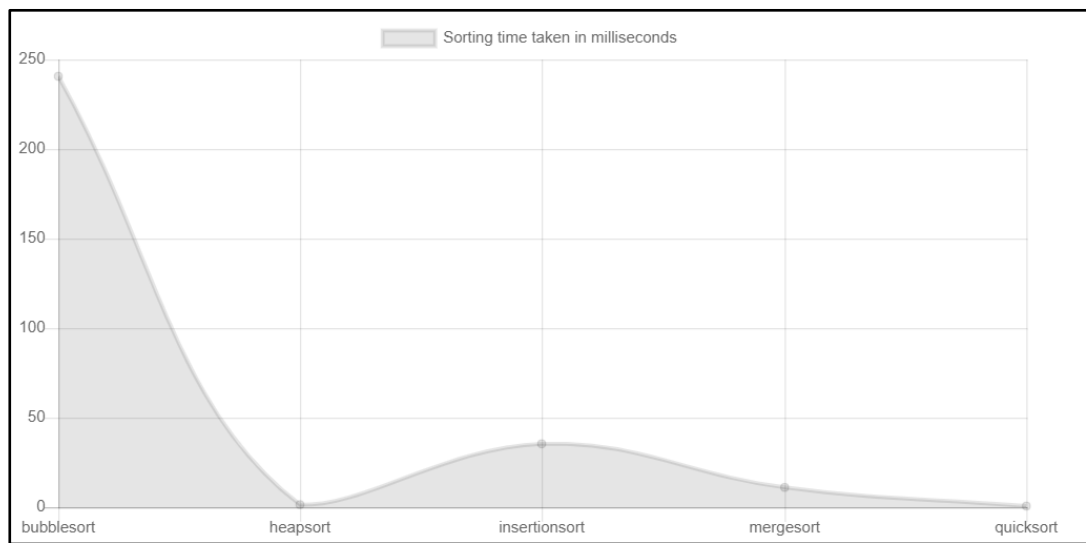
1. For a large sorted input (12999- integers) bubblesort takes the least time (approx 0.3075 milliseconds) and insertionsort takes the worst time (approx 229.40 milliseconds).
2. For other algorithms the order of time taken is as follows:  
Quicksort < Heapsort < Mergesort.

- A reverse sorted list input:



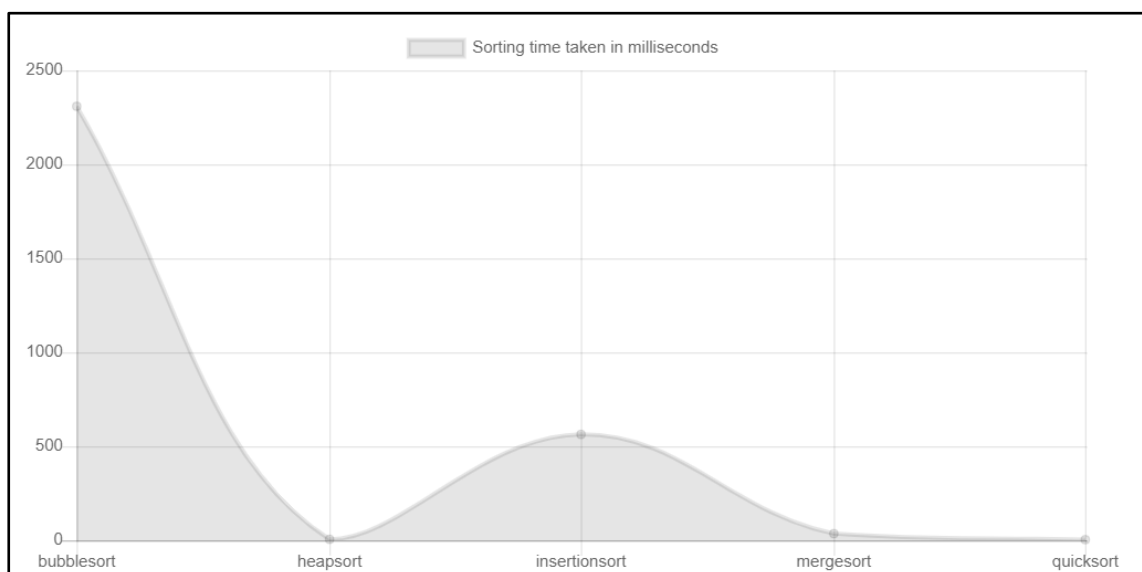
1. For a sorted list in reverse order quicksort and heapsort perform the best and have almost similar runtime(approx 1.2 milliseconds), mergesort performs second best.
2. Bubblesort takes the most time (approx 764.15 milliseconds), followed by insertionsort taking the most time as compared to other algorithms.

- A list with repeating numbers:



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

- Unsorted list:



1. 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).
2. Mergesort performs better than insertion sort.



## Conclusion

Performance matrix:

Small input	
Case	Order of performance
<i>sorted list input.</i>	Bubblesort < Quicksort < Insertionsort < Heapsort < Mergesort
<i>reverse sorted list input.</i>	Quicksort < Heapsort < Insertionsort < Mergesort < Bubblesort
<i>list with repeating numbers.</i>	Quicksort < Heapsort, Insertionsort < Bubblesort < Mergesort
<i>Unsorted list.</i>	Heapsort, Quicksort < Insertionsort < Mergesort < Bubblesort
Large input	
Case	Order of performance
<i>sorted list input.</i>	Bubblesort < Quicksort < Heapsort < Mergesort < Insertionsort
<i>reverse sorted list input.</i>	Heapsort, Quicksort < Mergesort < Insertionsort < Bubblesort
<i>list with repeating numbers.</i>	Quicksort < Heapsort < Mergesort < Insertionsort < Bubblesort
<i>Unsorted list.</i>	Quicksort < Heapsort < Mergesort < Insertionsort < Bubblesort

Individual algorithm performance for same input (unsorted) (mentioned in input list txt file):

\*The below values are an average of multiple runs.

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

Insertion sort is overall good for small input and quicksort is good for large input