Tudor Păiușan

# A theoretical and experimental comparison of sorting algorithms

In this paper I will be comparing several sorting algorithms using different techniques and input data.

The source code can be found here: https://github.com/Skatlers/SortingAlgorithmComparison

## Algorithm efficiency

| Sorting Algorithm | Time Complexity | | | Space Complexity |
|---|---|---|---|---|
| | Best Case | Average Case | Worst Case | Worst Case |
| Bubble Sort | Ω(N) | Θ(N²) | O(N²) | O(1) |
| Selection Sort | Ω(N²) | Θ(N²) | O(N²) | O(1) |
| Insertion Sort | Ω(N) | Θ(N²) | O(N²) | O(1) |
| Merge Sort | Ω(N log N) | Θ(N log N) | O(N log N) | O(N) |
| Quick Sort | Ω(N log N) | Θ(N log N) | O(N²) | O(log N) |
| Count Sort | Ω(N + k) | Θ(N + k) | O(N + k) | O(k) |
| Shaker Sort | Ω(N) | Θ(N²) | O(N²) | O(1) |

## The method

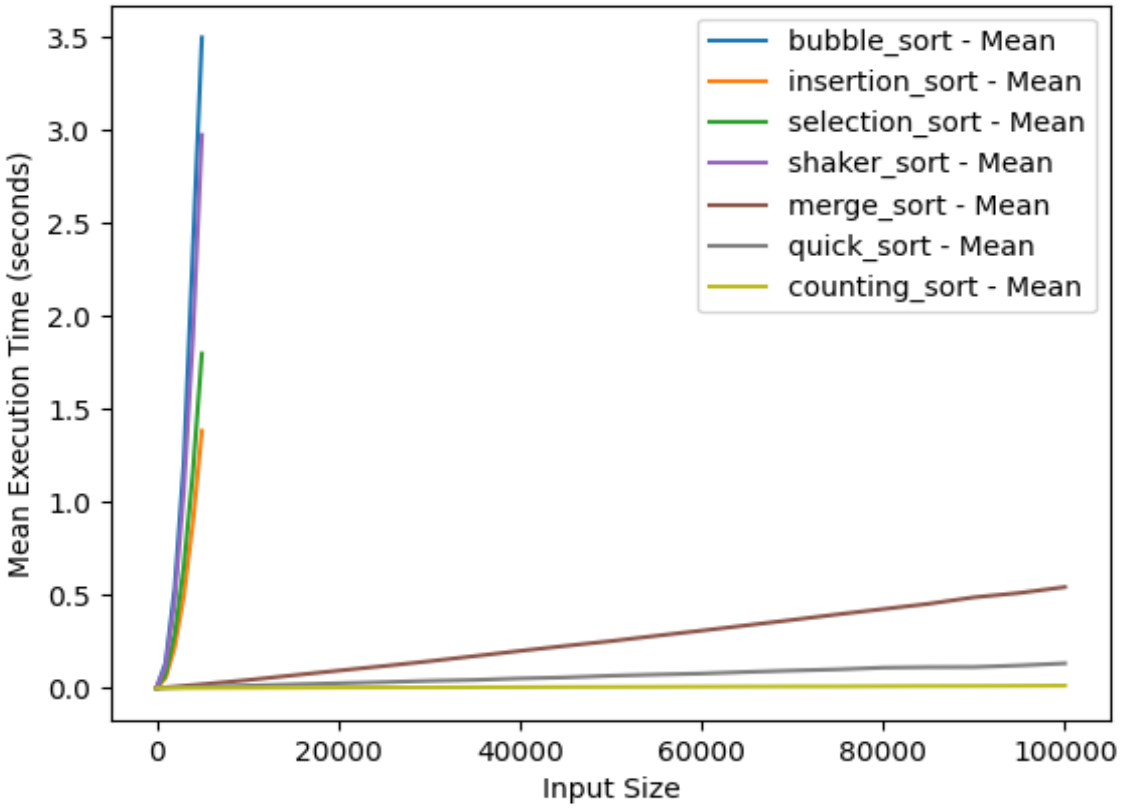
Using the time python library I am measuring each algorithm's execution time. Each algorithm will run n repetitions on the same type of data set. Two figures will show the mean execution times and the individual ones.

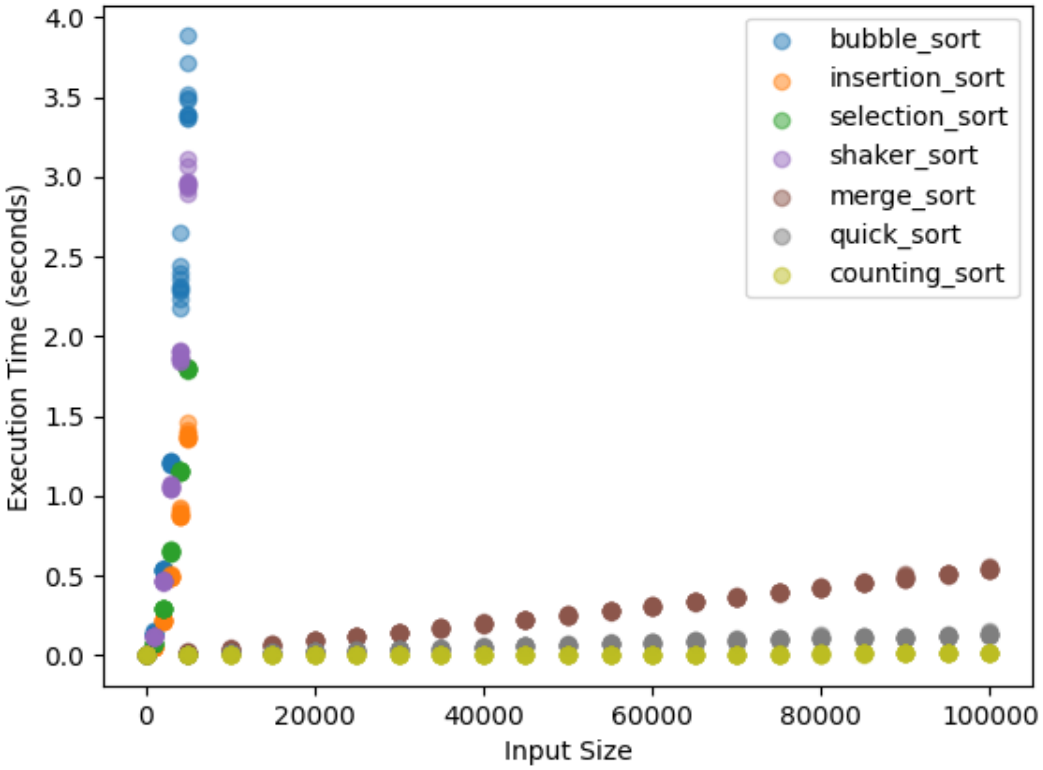
### Random values from 1 to 1000

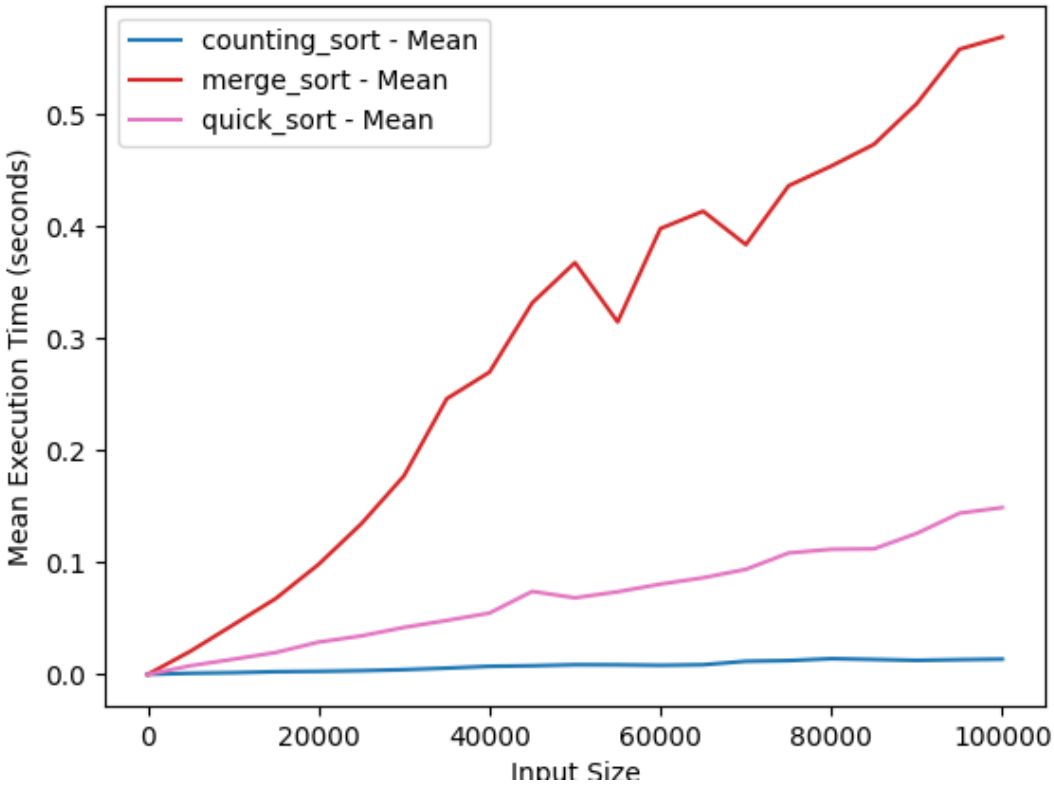
```
def generate_random_data(size):
```

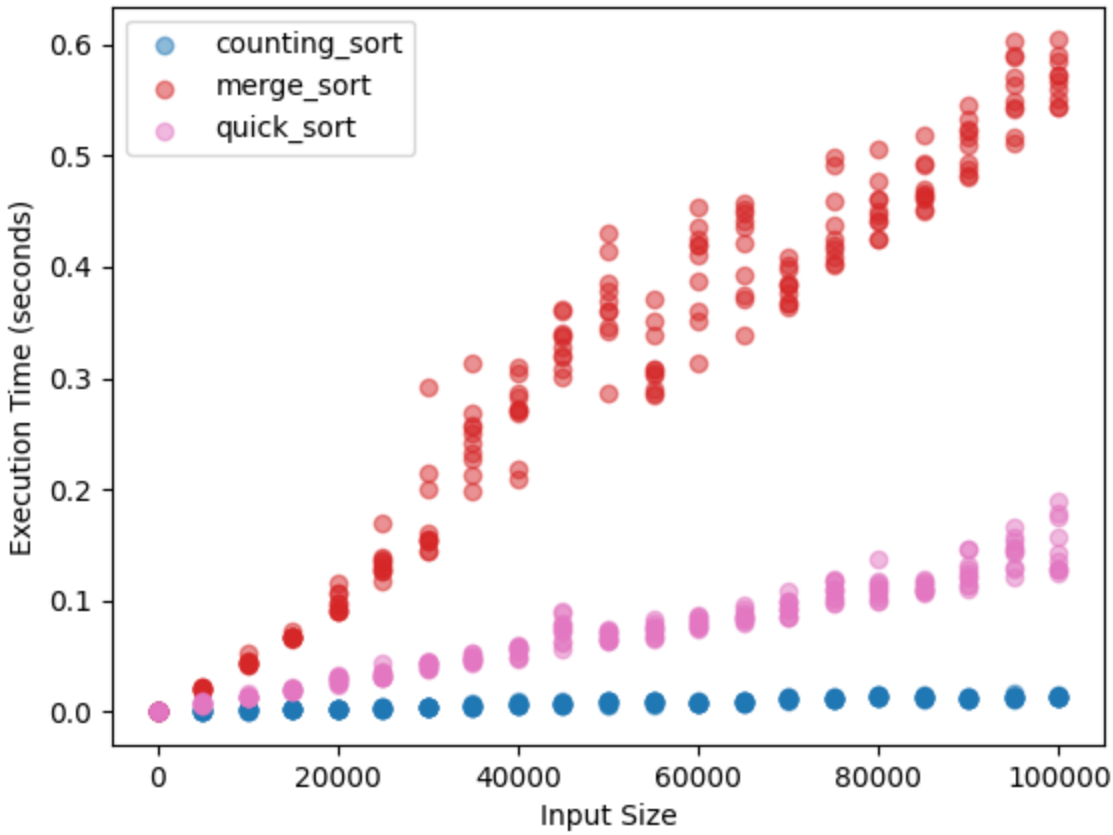
```
    return np.random.randint(1, 1000, size=size)
```

Repetitions: 10




bubble_sort
insertion_sort
selection_sort
shaker_sort
merge_sort
quick_sort
counting_sort
Execution Time (seconds)
Input Size
0.0
0.5
1.0
1.5
2.0
2.5
3.0
3.5
4.0
0
20000
40000
60000
80000
100000


counting_sort - Mean
merge_sort - Mean
quick_sort - Mean
Mean Execution Time (seconds)
Input Size
0.0
0.1
0.2
0.3
0.4
0.5
0
20000
40000
60000
80000
100000

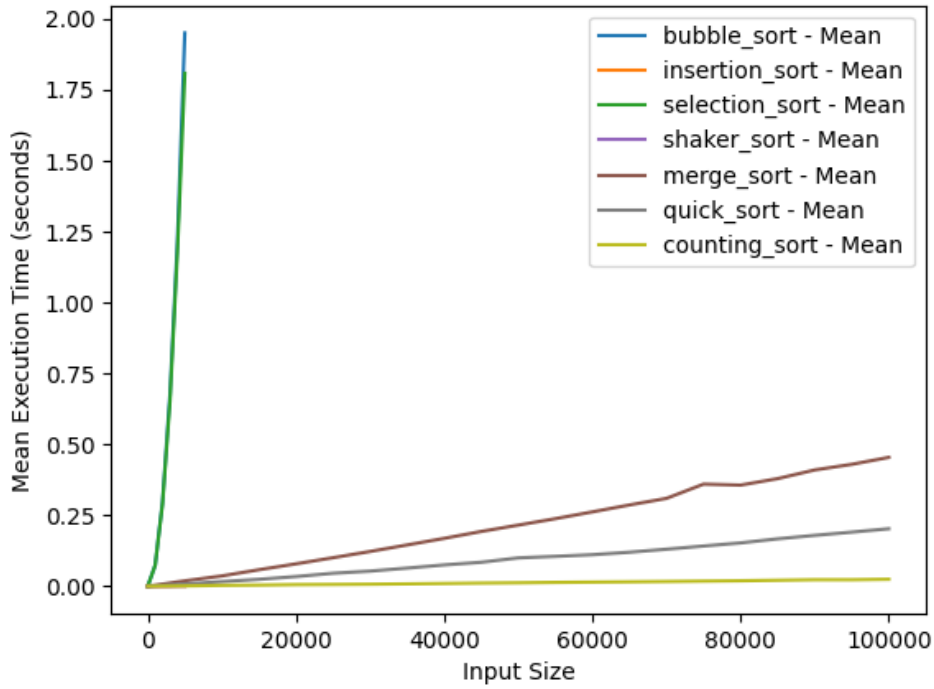


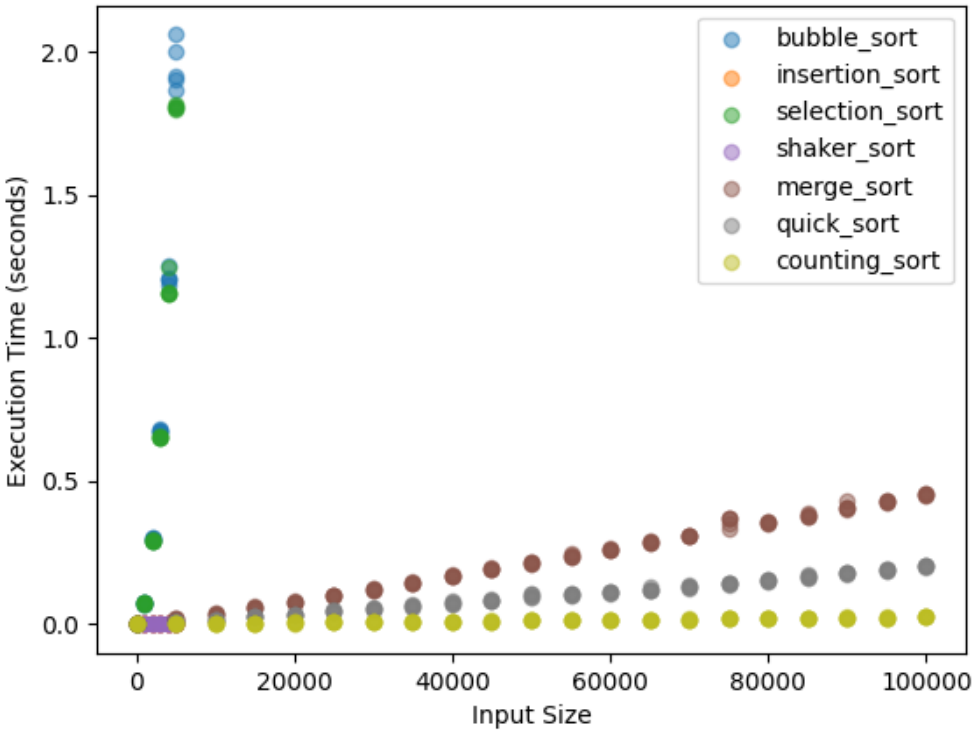

Conclusion: Counting sort is the fastest while Bubble Sort is the slowest. This was expected.

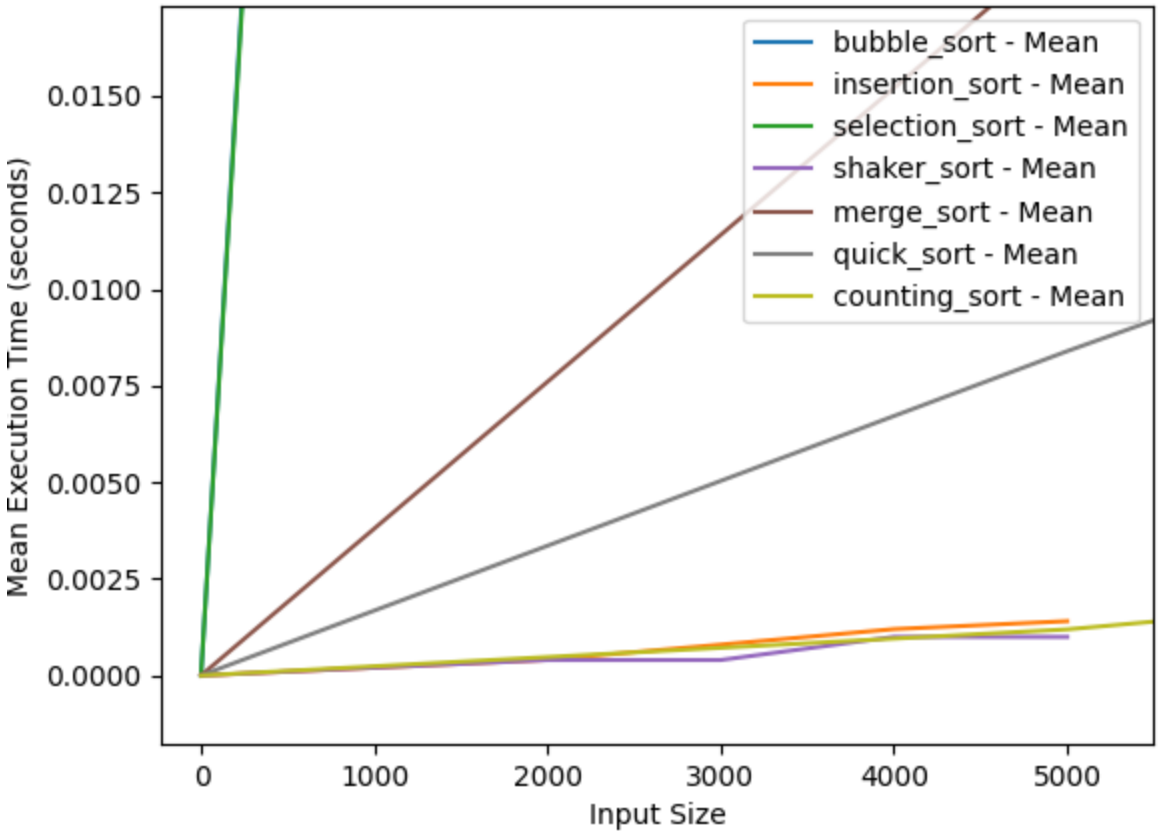
## Already sorted array

```
def generate_random_data(size):
    return np.arange(1, size + 1)
```

Repetitions: 5


bubble_sort - Mean
insertion_sort - Mean
selection_sort - Mean
shaker_sort - Mean
merge_sort - Mean
quick_sort - Mean
counting_sort - Mean
Mean Execution Time (seconds)
Input Size
0.00
0.25
0.50
0.75
1.00
1.25
1.50
1.75
2.00
0
20000
40000
60000
80000
100000


bubble_sort
insertion_sort
selection_sort
shaker_sort
merge_sort
quick_sort
counting_sort
Execution Time (seconds)
Input Size
0.0
0.5
1.0
1.5
2.0
0
20000
40000
60000
80000
100000

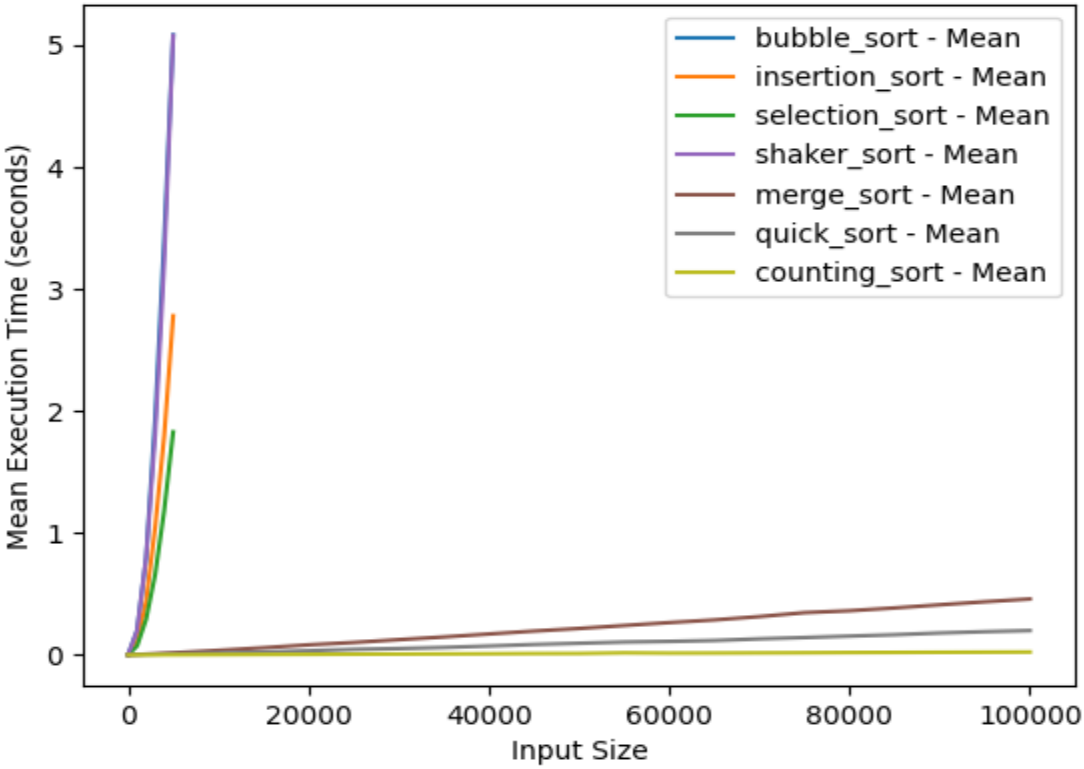


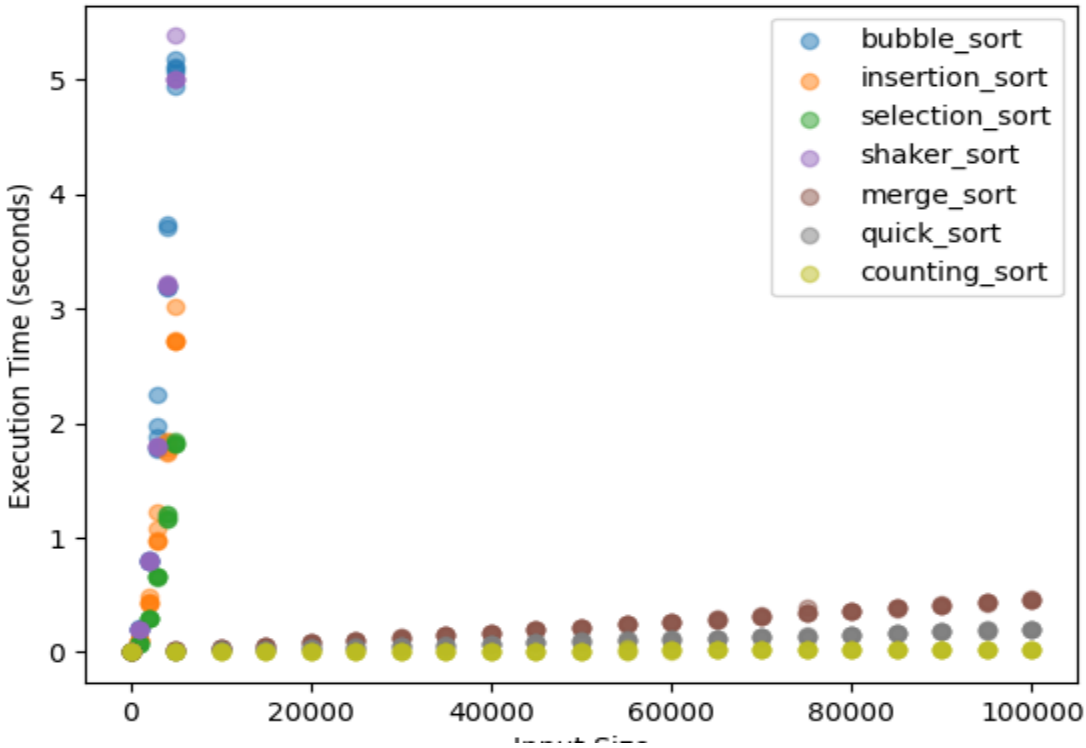

Conclusion: Shaker Sort and insertion sort are much faster due to checking for unsorted elements. Counting sort is still very fast.

## Reverse sorted array

```
def generate_random_data(size):
    return np.arange(size, 0, -1)
```

Repetitons: 5



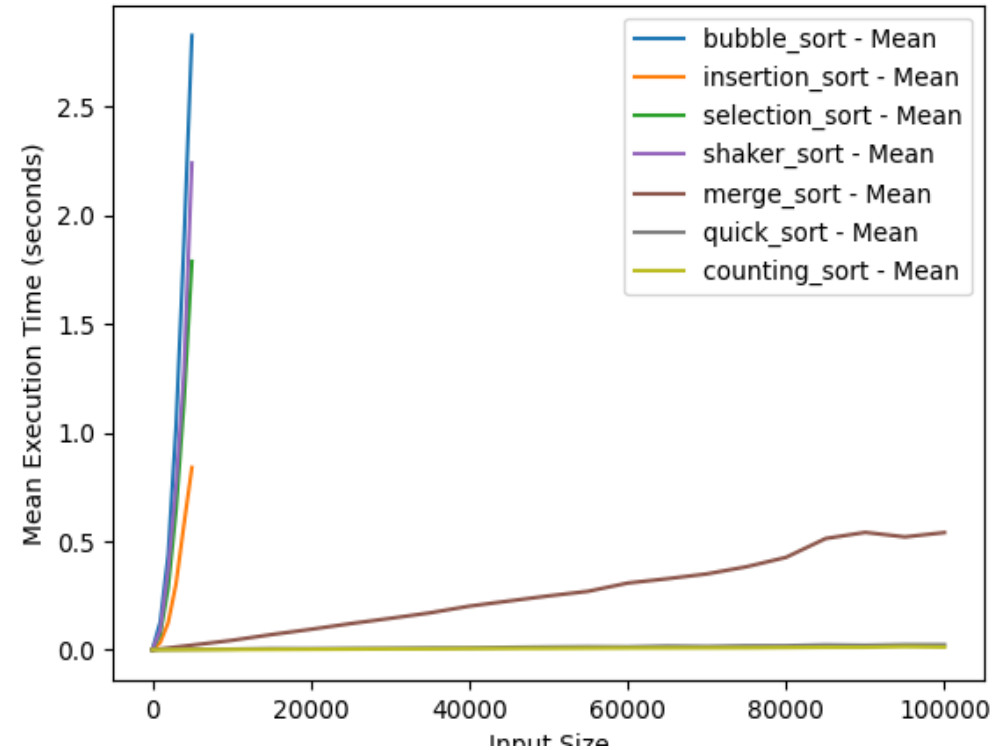



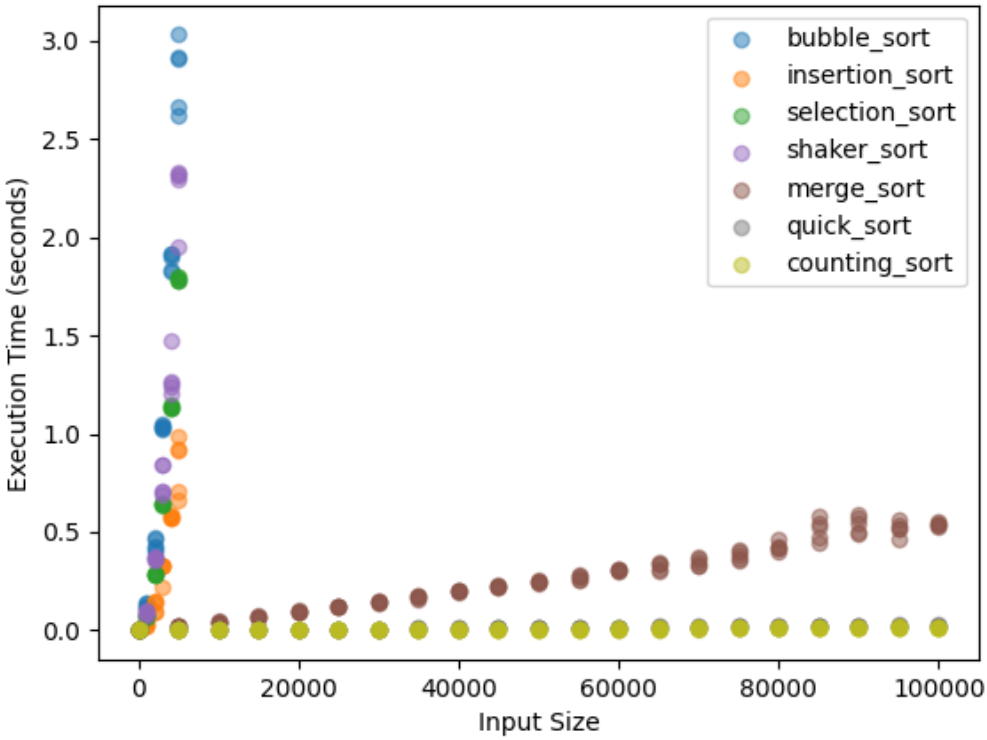

Conclusion: Performance similar to random arrays except for Shaker Sort which is slower

# Array with few unique elements

```python
def generate_random_data(size):
    unique_elements = np.random.choice(10, size=3)
    return np.random.choice(unique_elements, size=size)
```

Repetitions: 5



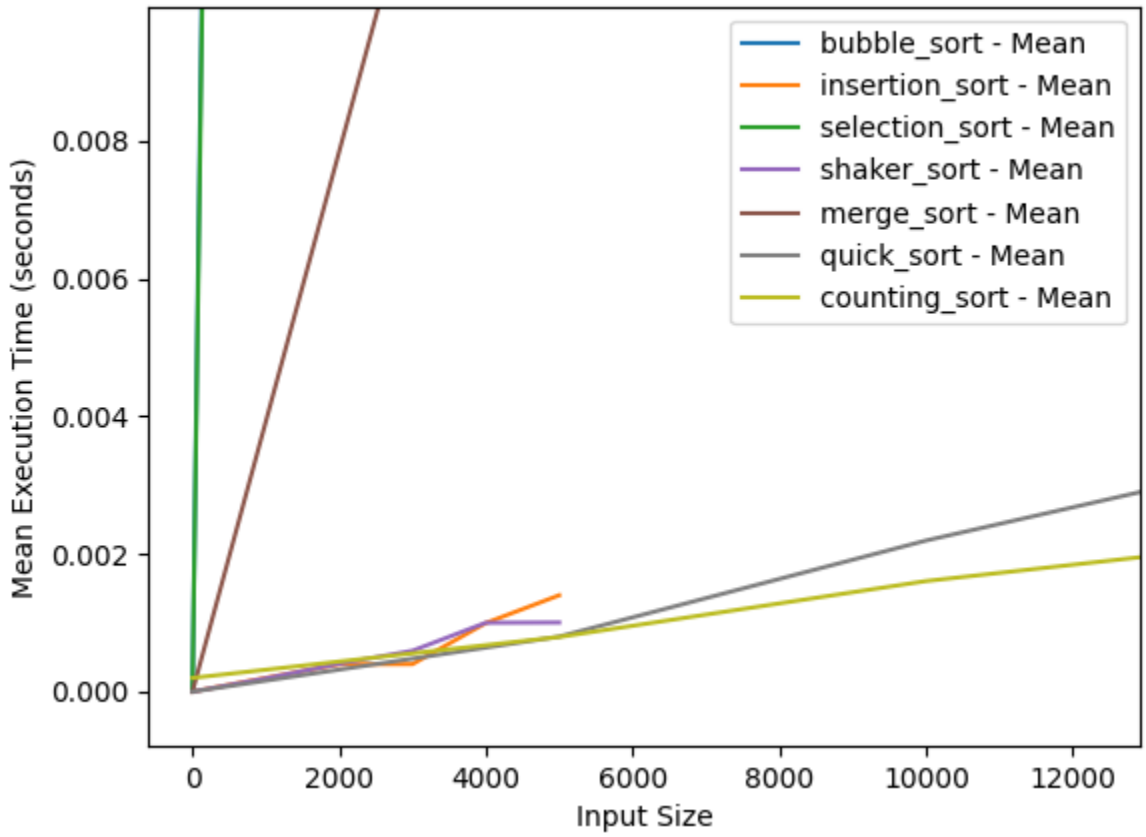




Conclusion: Quick Sort is almost as fast as Couting Sort

## Array filled with a single element

```
def generate_random_data(size):
    return np.full(size, fill_value=1337)
```

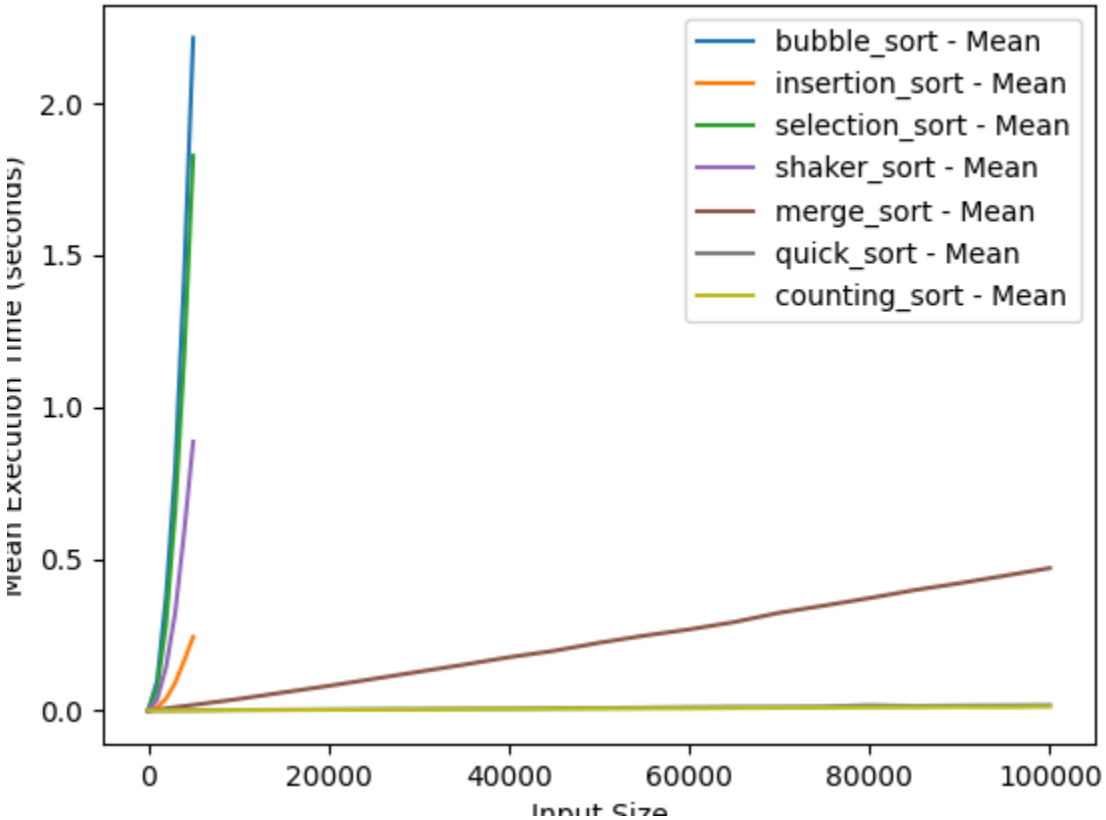
Repetitions: 5




Conclusion: Shaker Sort and Insertion Sort are much faster. Quicksort is almost as fast as Counting Sort.

# Sparse array, most elements are 0

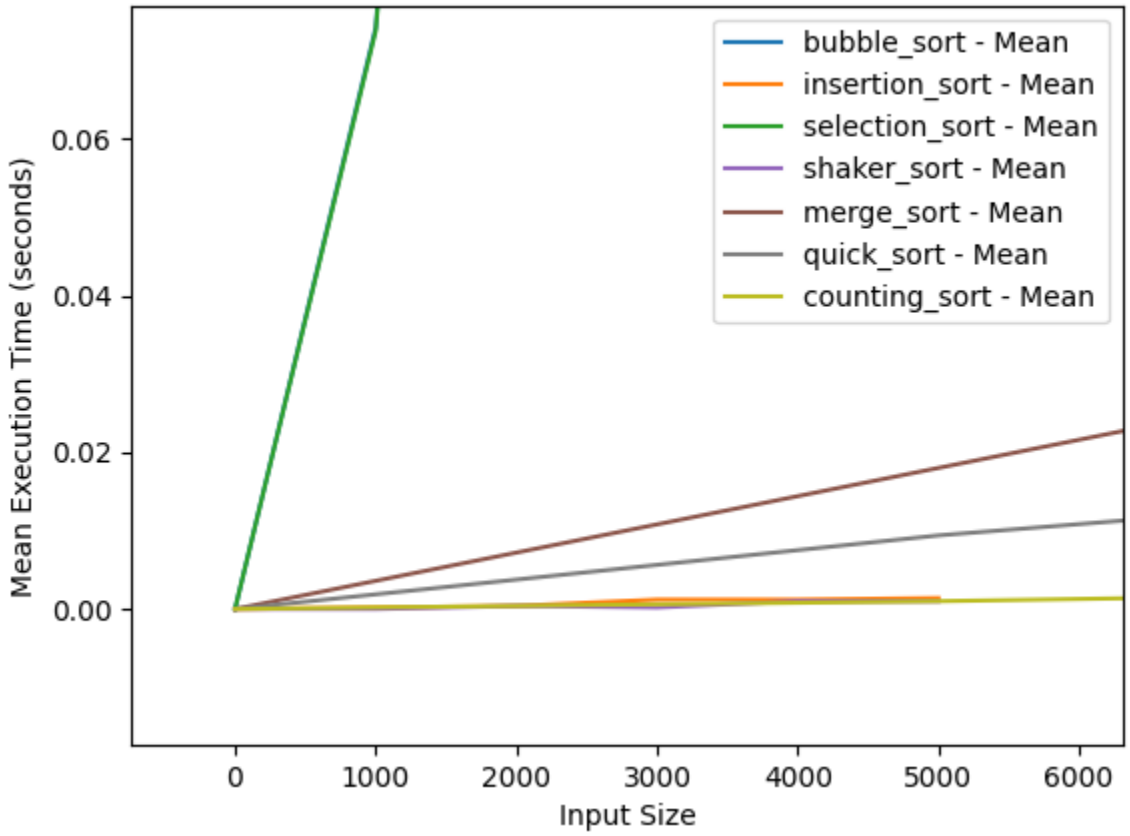
```
def generate_random_data(size):
    sparse_ratio = 0.9
    return np.random.choice([0, 1], size=size, p=[sparse_ratio, 1 -
sparse_ratio])
```




Conclusion: Quick Sort and Insertion Sort are faster

# Array with Pre-sorted Blocks

```
def generate_random_data(size):
    block_size = 10
    num_blocks = size // block_size
    return np.concatenate([np.arange(i, i + block_size) for i in range(0, num_blocks *
block_size, block_size)])
```




Conclusion: Insertion sort is almost as fast as Counting sort