Analysis of Various Sorting Algorithms
19CSE212

Deebakkarthi C R
CB.EN.U4CSE20613

Devaraja G
CB.EN.U4CSE20614

Krisha Vardhni M
CB.EN.U4CSE20633

April 10, 2022

Contents

Abstract

Sort (*verb*)

To put a number of things in order or separate them into groups

We come across sorting a lot in our day to day lives. For example a teacher asking the students to stand in the order of their height, a supermarket assistant replenishing stocks in the shelves, a deck of cards getting sorted, and the sort by price, popularity, etc. feature in ecommerce websites. All these involve arranging a set of things based on a set of rules. In relevance to computer science, sorting plays a key role in searching as it's very efficient to search for an element in a sorted list rather than an unsorted one. The rules(algorithms) to sort efficiently have been a hot topic in computer science for decades and due to its relevance, numerous people have worked on building various sorting algorithms, as a result we now have a large number of those, out of which only a few are dominant in practical implications. We'll be analyzing five of the most common algorithms in this report.

Before we get into that we need to understand the notion of time and space complexity. Computers are fast but even they can only do so much. They are limited by the number of instructions they can run per unit time and the amount of storage they can use. In building these algorithms it's crucial that we consider these as they play a major role in the speed and space taken. In general the running time takes precedence over the space required as modern PCs have ample storage space available for most of the everyday tasks.

1 Sorting Algorithms

1.1 Bubble Sort

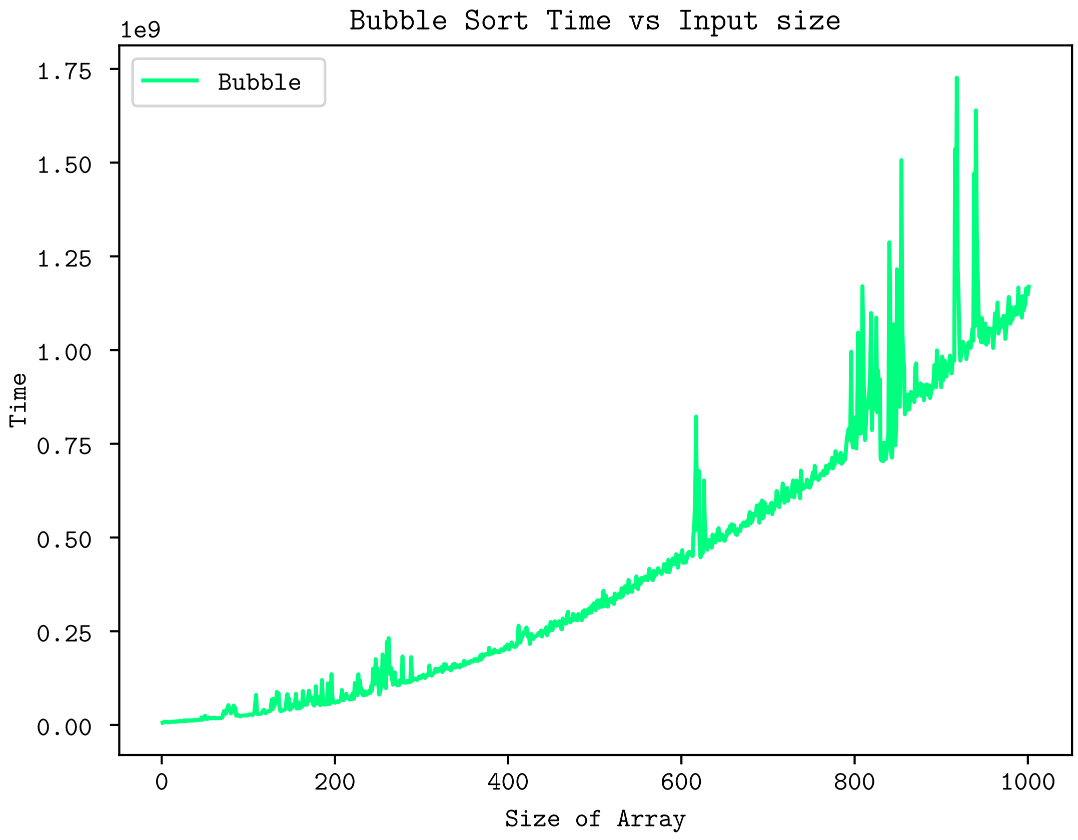
Principle

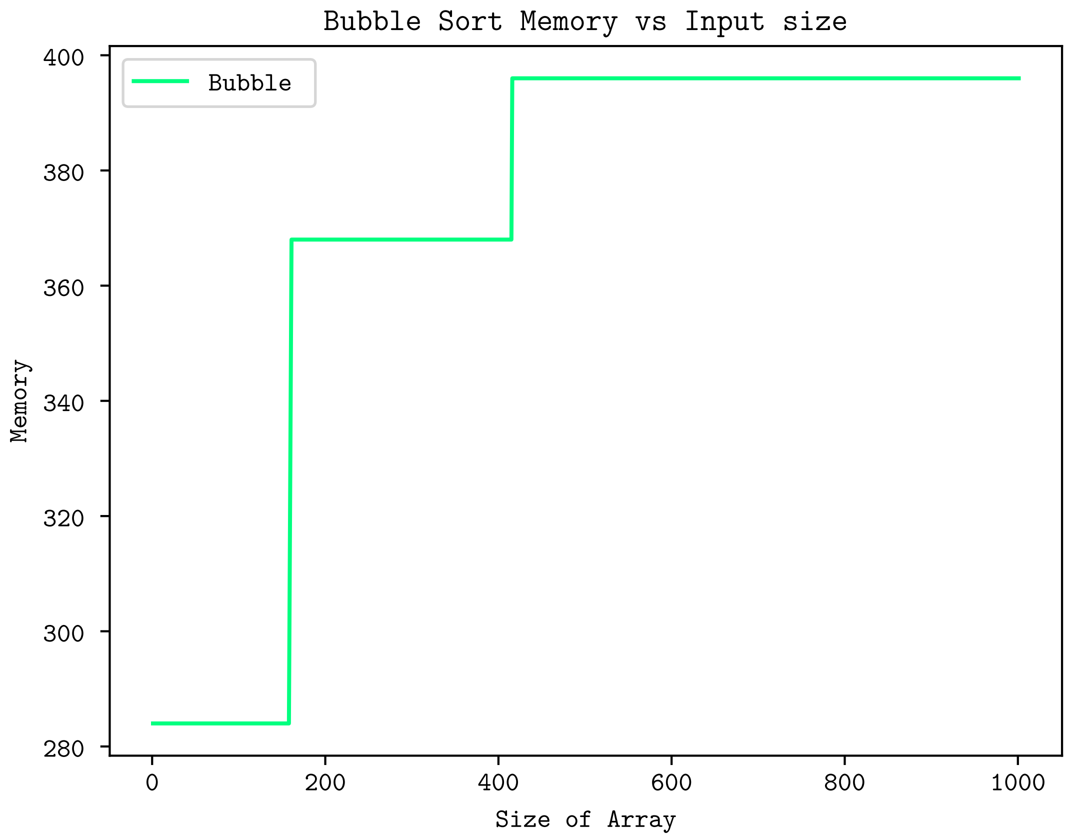
In this sorting technique, the array is sequentially scanned several times and during each iteration, the pairs of consecutive elements are compared and interchanged(if required), to bring them into ascending order. At the end of the first iteration the largest element in the array is pushed to the end. It is an easy but time consuming method when a large number of swaps are required to take place.

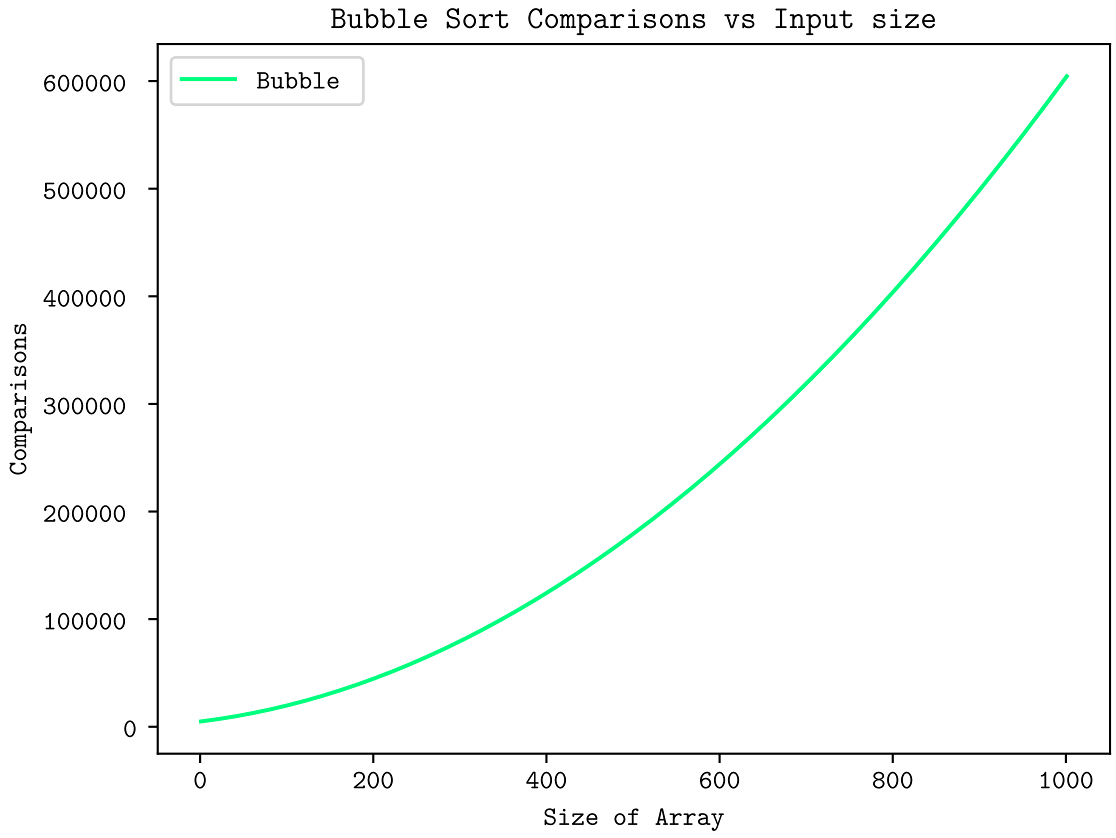
Code

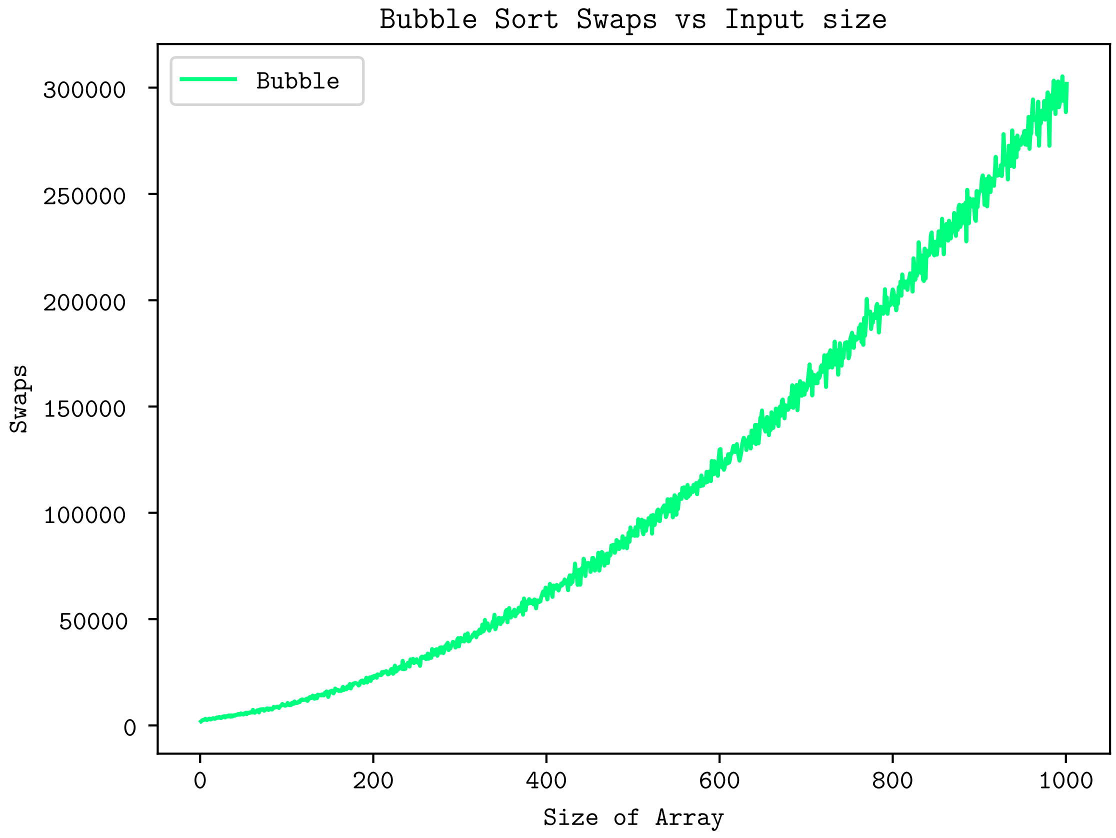
```
def bubbleSort(unsortedList):
        swap = 0
        itr = 0
        comp = 0
        tracemalloc.start()
        t_s = perf_counter_ns()
        for i in range(len(unsortedList)):
                for j in range(len(unsortedList) - 1 - i):
                itr += 1
                comp += 1
                if unsortedList[j] > unsortedList[j + 1]:
                        t = unsortedList[j]
                        unsortedList[j] = unsortedList[j + 1]
                        unsortedList[j + 1] = t
                        swap += 1
        t_e = perf_counter_ns()
        mem = tracemalloc.get_traced_memory()[1]
        tracemalloc.stop()
        return {"Time":t_e-t_s,
        "Memory":mem,
        "Comparisons":comp,
        "Swaps":swap,
        "Iterations":itr}
```

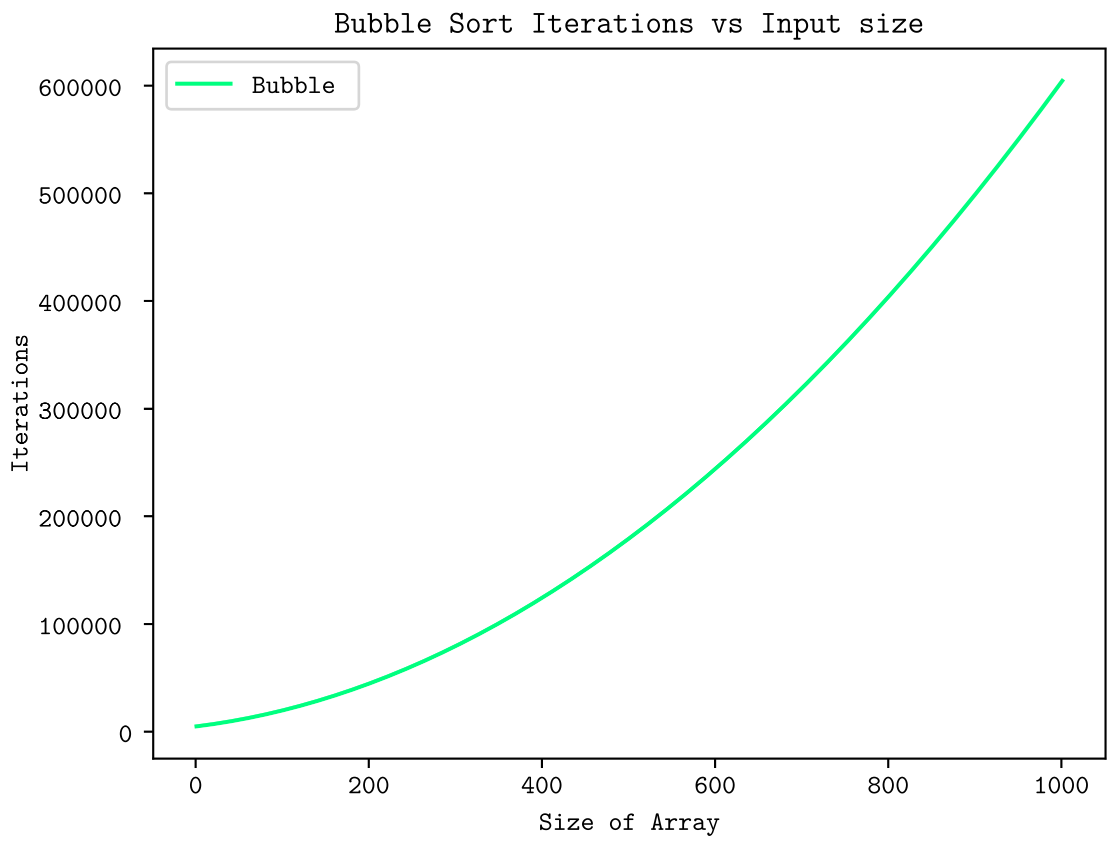
Graphs


Bubble Sort Time vs Input size
1e9
1.75
1.50
1.25
1.00
0.75
0.50
0.25
0.00
Time
Bubble
0
200
400
600
800
1000
Size of Array


Bubble Sort Memory vs Input size
Bubble
Memory
Size of Array
400
380
360
340
320
300
280
0
200
400
600
800
1000


Bubble Sort Comparisons vs Input size
Bubble
Comparisons
600000
500000
400000
300000
200000
100000
0
0
200
400
600
800
1000
Size of Array


Bubble Sort Swaps vs Input size
Bubble
Swaps
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
Size of Array



Insights

It has a runtime complexity of $O(N^2)$. Its space complexity is $O(1)$.

1.2 Selection Sort

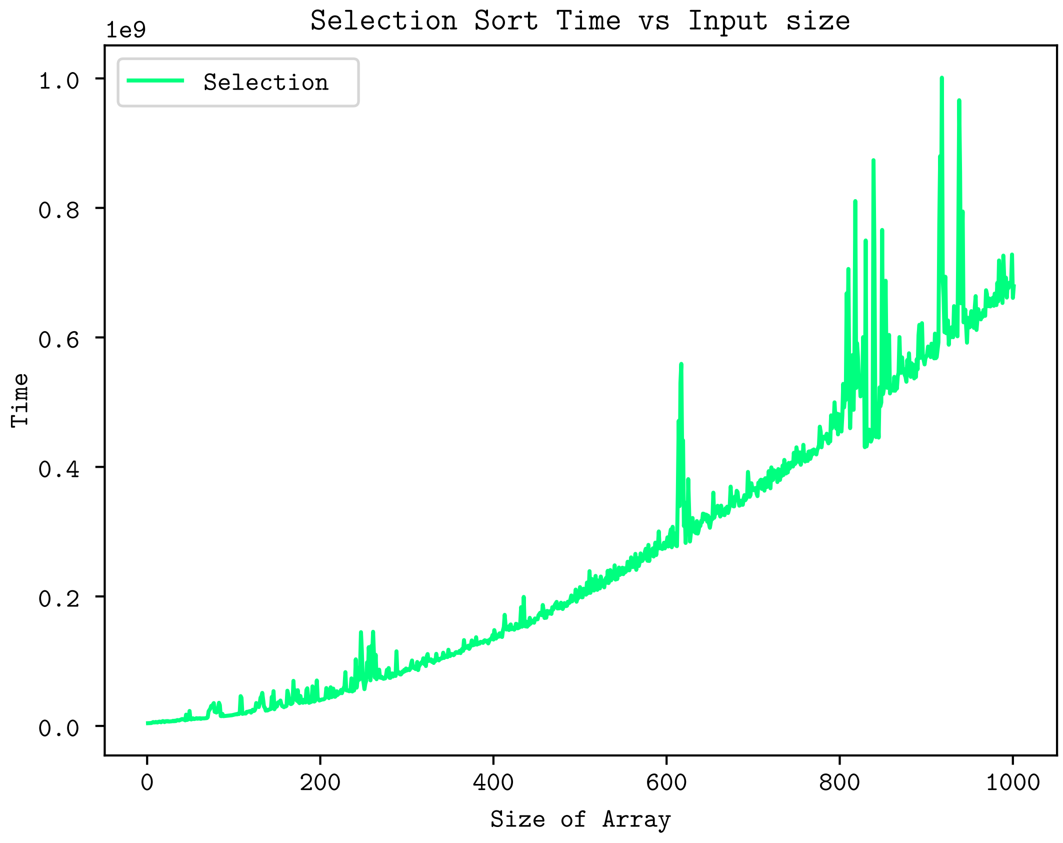
1.2.1 Principle

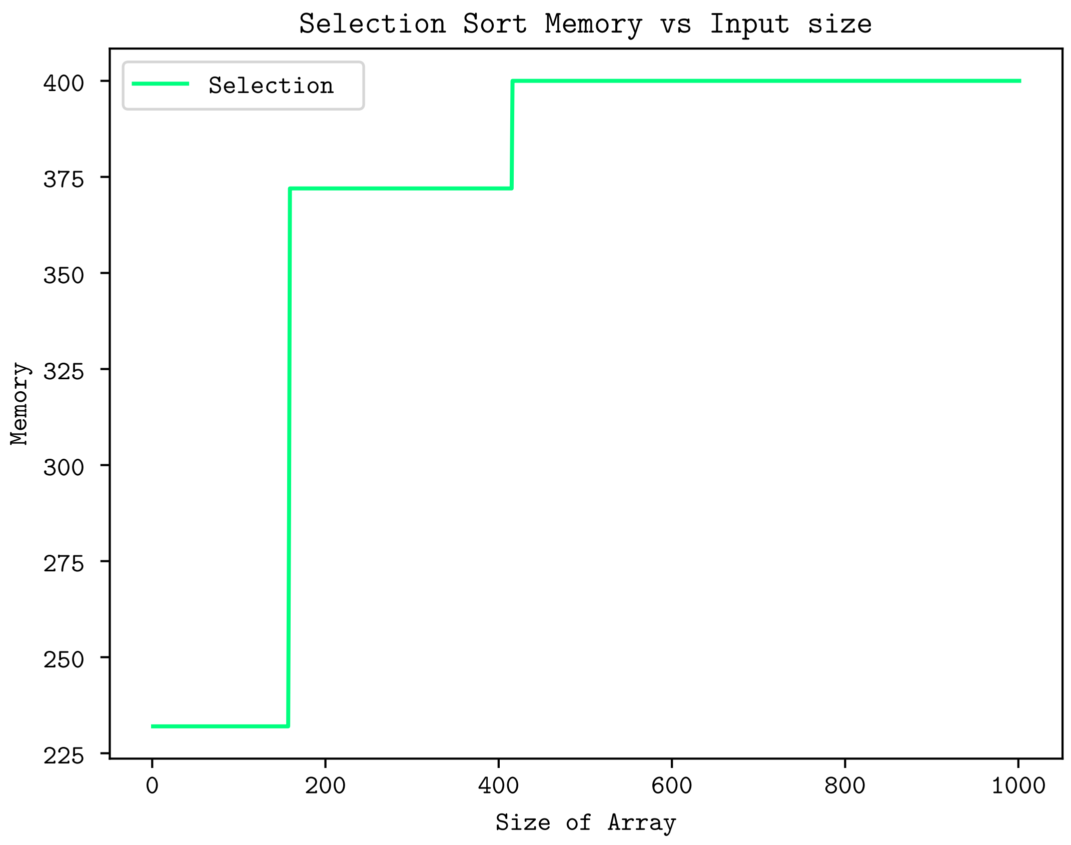
To sort an array in ascending order using selection sort, in each iteration we repeatedly select the smallest element from the unsorted part of the array and append it to the end of the sorted part. In the first iteration, take the 0th element as minimum and compare with the next element. If it is larger than the next element, assign the 1st element as minimum and repeat till the end of the array. At the end of the first iteration we get the smallest element of the array. At this stage place the minimum element in the beginning of the array and proceed to the next iteration to find the next smallest value.

Code

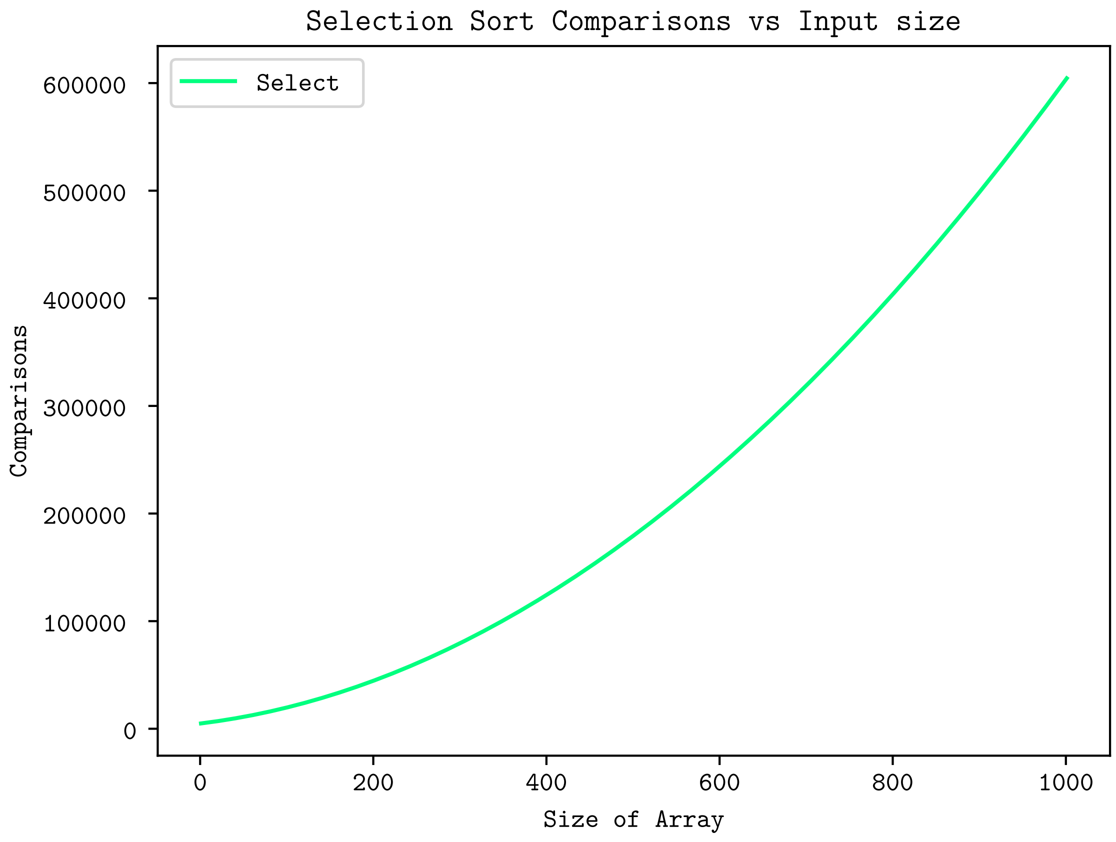
```python
def selectionSort(unsortedList):
    swap = 0
    itr = 0
    comp = 0
    tracemalloc.start()
    t_s = perf_counter_ns()
    for i in range(len(unsortedList)):
        minIndex = i
        for j in range(i + 1, len(unsortedList)):
            itr += 1
            comp += 1
            if unsortedList[minIndex] > unsortedList[j]:
                minIndex = j
        (unsortedList[i], unsortedList[minIndex]) = \
            (unsortedList[minIndex], unsortedList[i])
        swap += 1
    t_e = perf_counter_ns()
    mem = tracemalloc.get_traced_memory()[1]
    tracemalloc.stop()
    return {"Time":t_e-t_s,
            "Memory":mem,
            "Comparisons":comp,
            "Swaps":swap,
            "Iterations":itr}
```

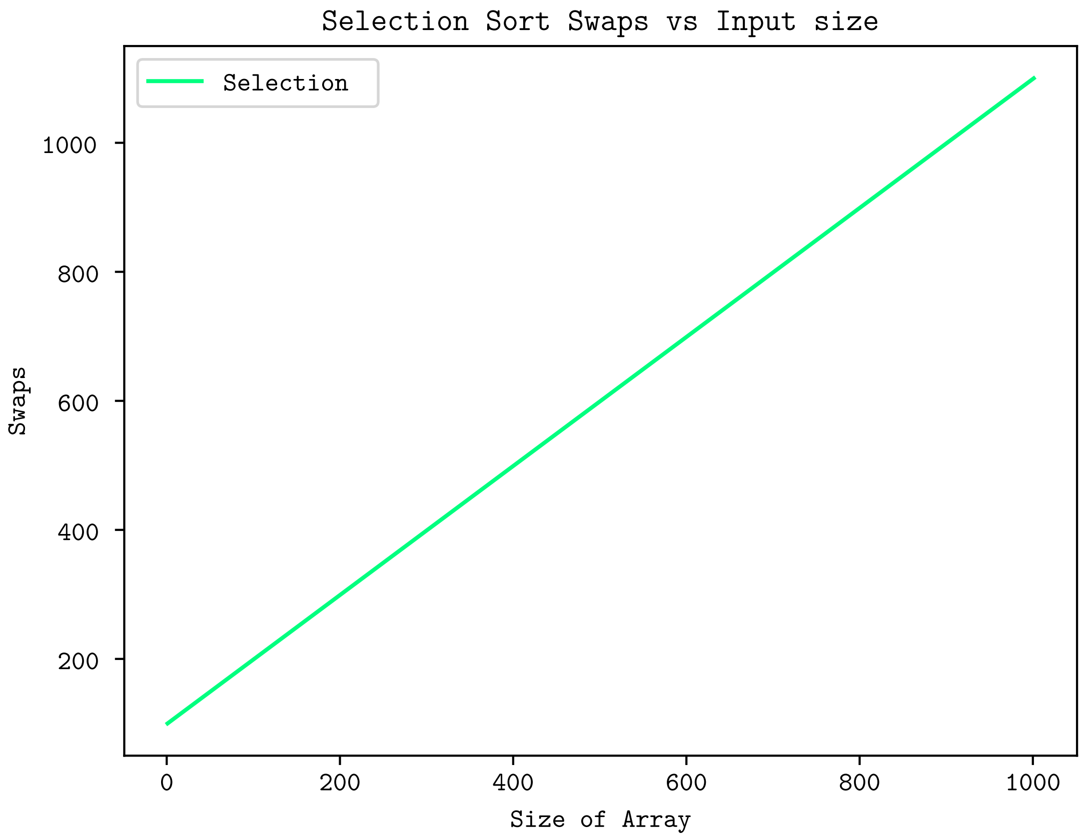
Graphs

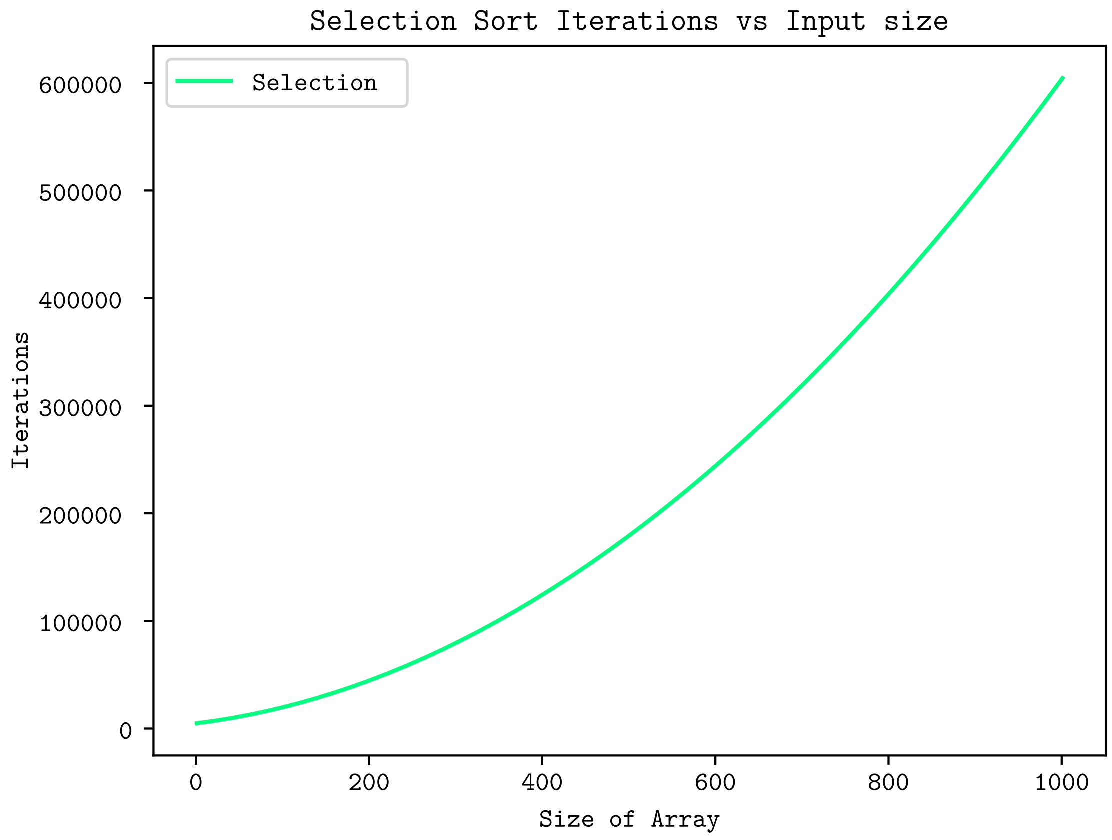

Selection Sort Time vs Input size
1e9
Selection
1.0
0.8
0.6
0.4
0.2
0.0
Time
0
200
400
600
800
1000
Size of Array



Selection Sort Memory vs Input size
Selection
Memory
Size of Array
400
375
350
325
300
275
250
225
0
200
400
600
800
1000


Selection Sort Comparisons vs Input size
Select
Comparisons
0
100000
200000
300000
400000
500000
600000
0
200
400
600
800
1000
Size of Array


Selection Sort Swaps vs Input size
Selection
1000
800
600
400
200
Swaps
0
200
400
600
800
1000
Size of Array



Insights

Selection sort is a faster way to arrange smaller arrays but can take more time for larger ones, it has a runtime complexity of $O(N^2)$. However it is faster compared to bubble sort. Its space complexity is $O(1)$.

1.3 Insertion Sort

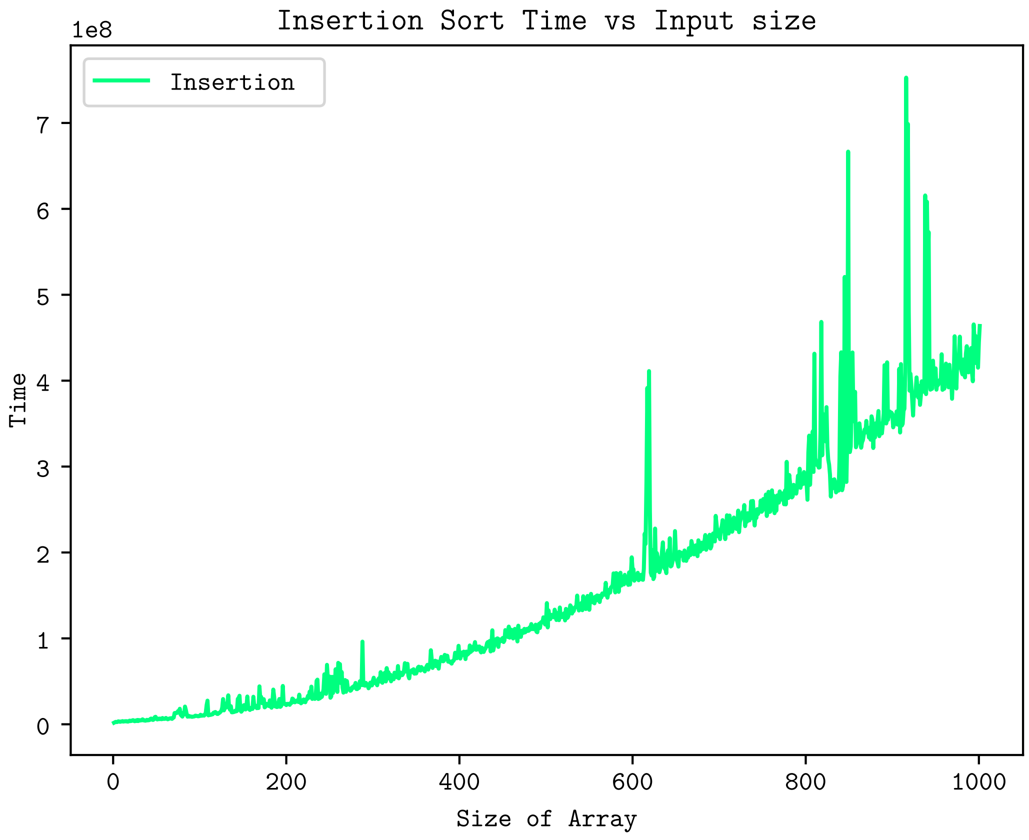
Principle

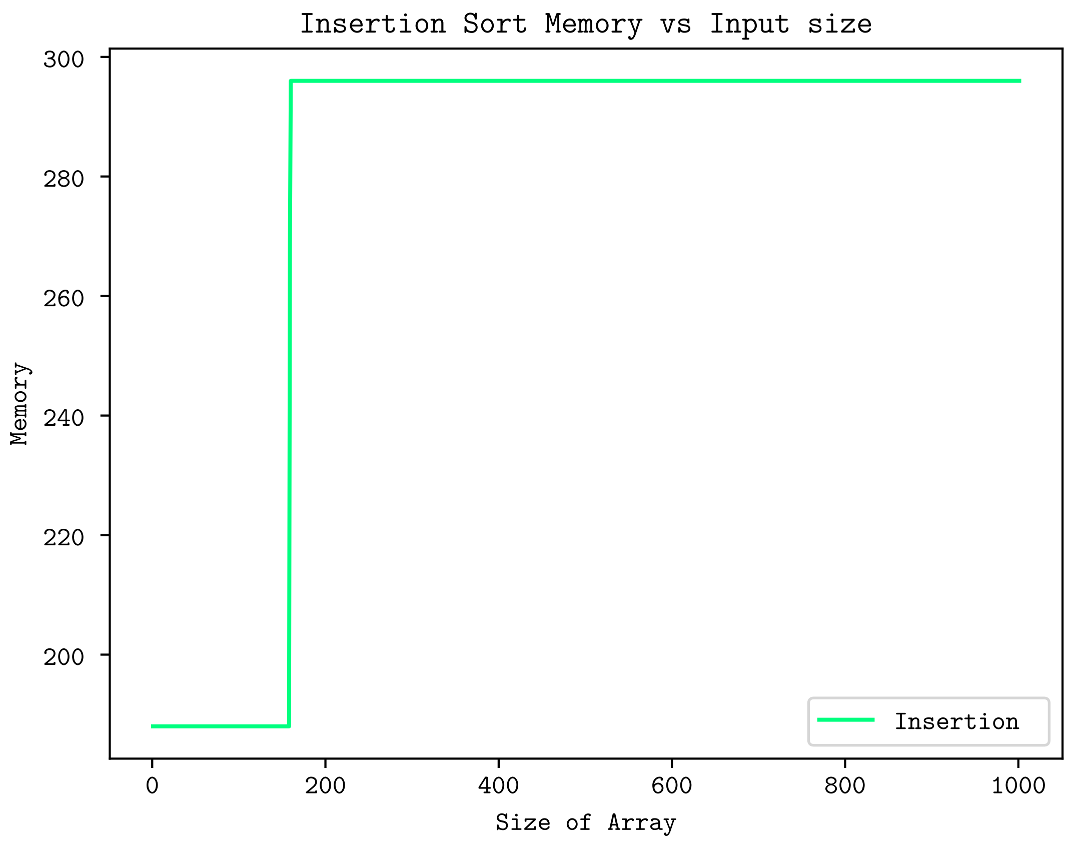
In insertion sort the array is split up virtually into a sorted and unsorted part. Initially the first element is assumed to be already sorted. We pick elements from the remaining unsorted part of the array and compare it with each element of the sorted part. If the selected unsorted element is greater than the sorted element, it will be placed after the sorted element otherwise it is placed in front of the sorted element. All the remaining elements are arranged into the sorted part in a similar way.

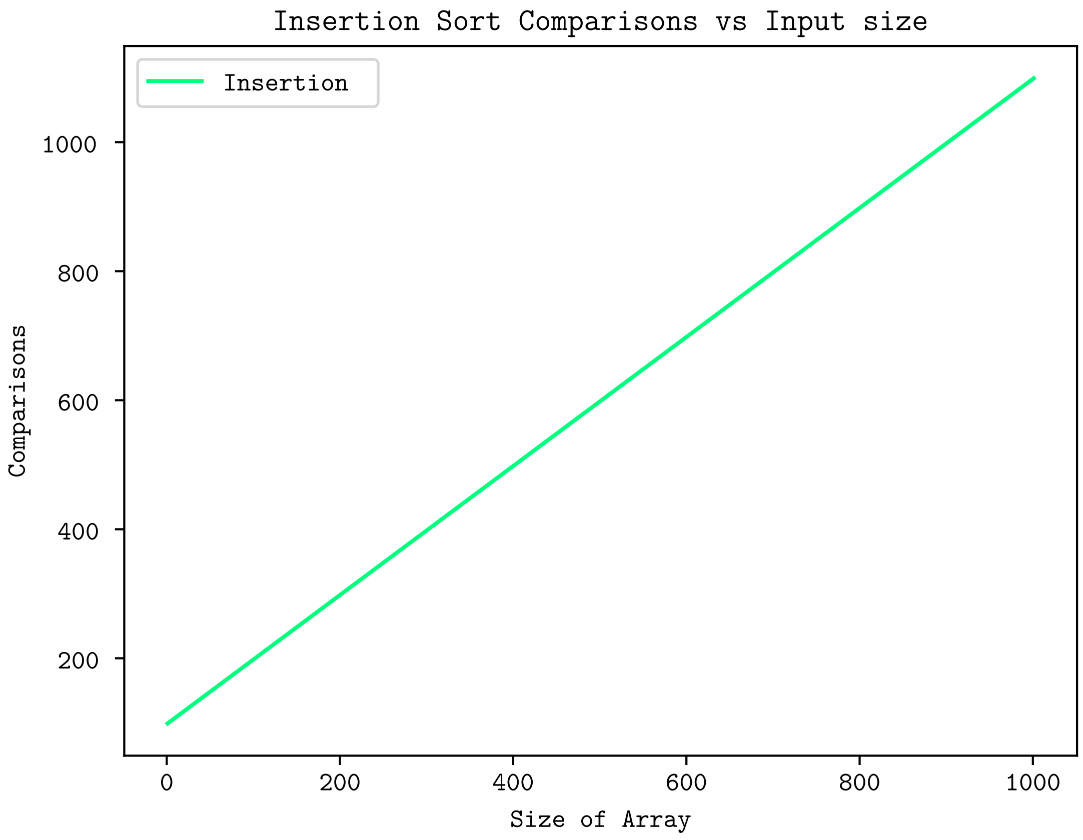
Code

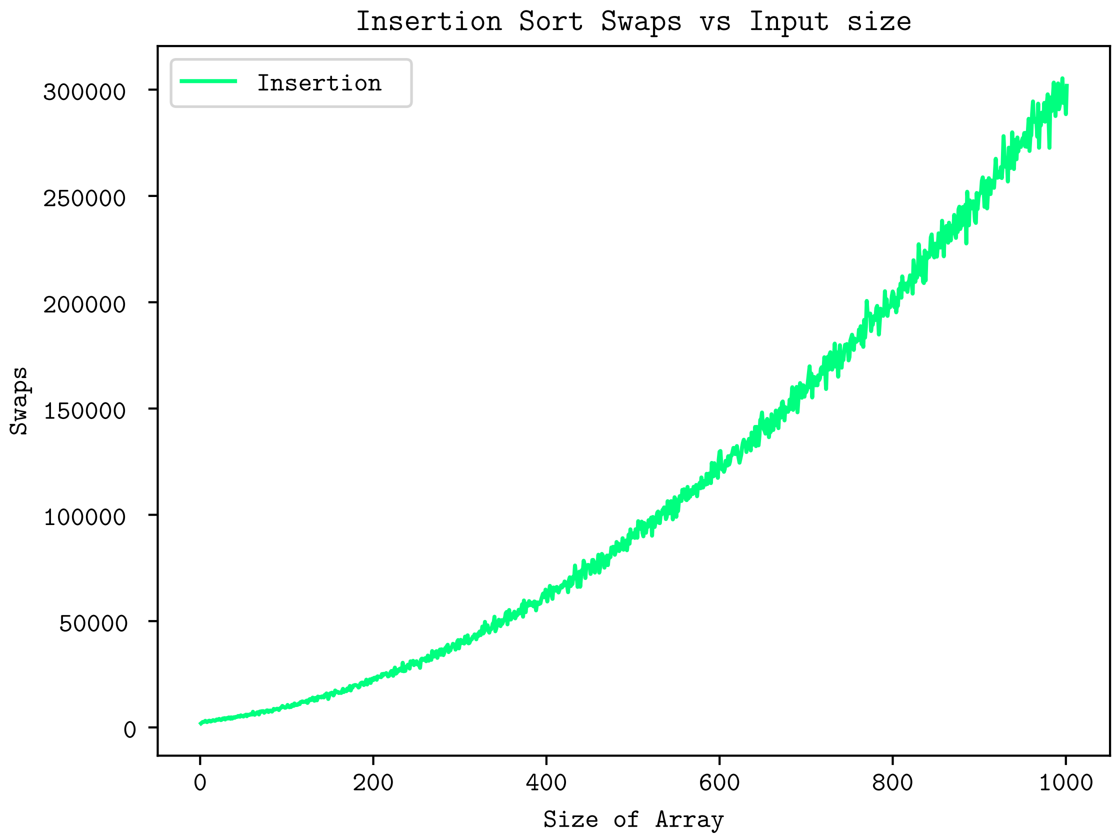
```python
def insertionSort(unsortedList):
    swap = 0
    itr = 0
    comp = 0
    tracemalloc.start()
    t_s = perf_counter_ns()
    for i in range(1, len(unsortedList)):
        itr += 1
        key = unsortedList[i]
        ptr = i - 1
        comp += 1

        while ptr >= 0 and unsortedList[ptr] > key:
            unsortedList[ptr + 1] = unsortedList[ptr]
            ptr -= 1
            swap += 1
            itr += 1

        unsortedList[ptr + 1] = key
    t_e = perf_counter_ns()
    mem = tracemalloc.get_traced_memory()[1]
    tracemalloc.stop()
    return {"Time":t_e-t_s,
            "Memory":mem,
            "Comparisons":comp,
            "Swaps":swap,
            "Iterations":itr}
```

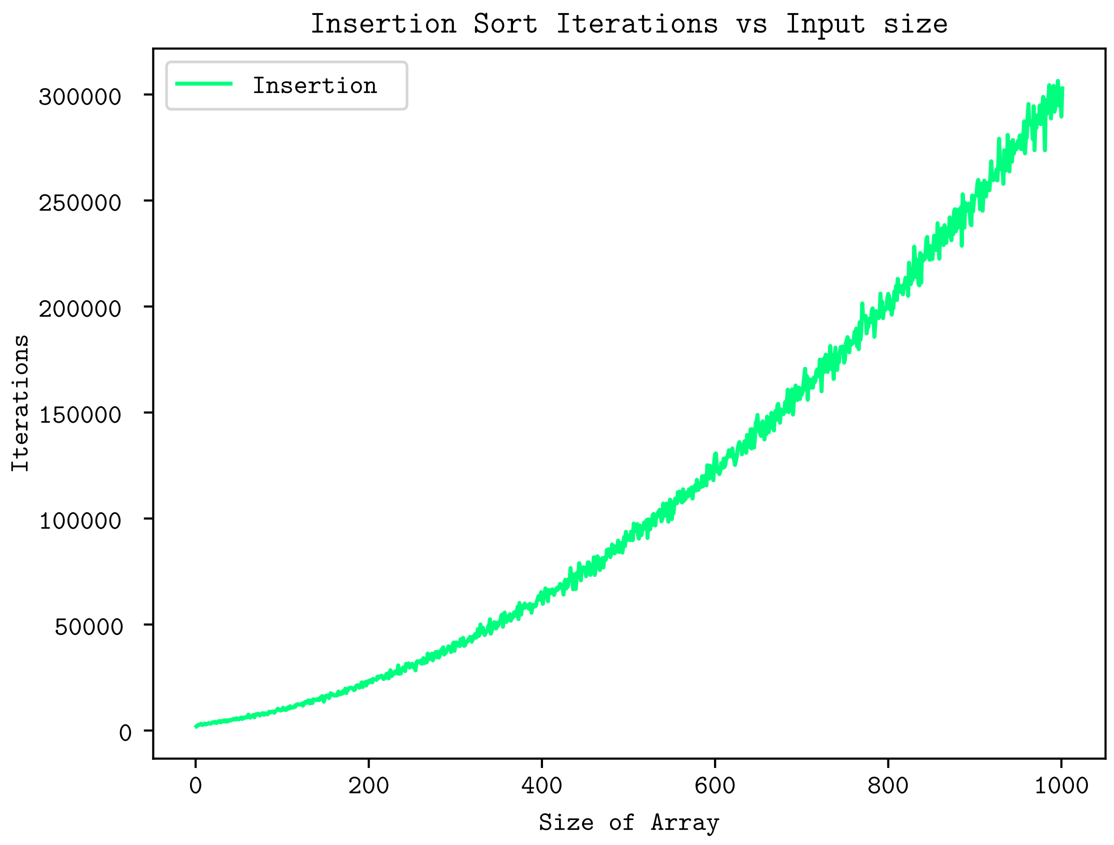
Graphs


Insertion Sort Time vs Input size
1e8
Insertion
Time
Size of Array
0
1
2
3
4
5
6
7
0
200
400
600
800
1000


Insertion Sort Memory vs Input size
Memory
300
280
260
240
220
200
0
200
400
600
800
1000
Size of Array
Insertion


Insertion Sort Comparisons vs Input size
Insertion
1000
800
600
400
200
Comparisons
0
200
400
600
800
1000
Size of Array


Insertion Sort Swaps vs Input size
Insertion
Swaps
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000
Size of Array



1.3.1 Insights

We can see that insertion sort is a faster way to arrange smaller arrays but can take more time for larger ones, it has a runtime complexity of $O(N^2)$. Its space complexity is $O(1)$. The number of comparisons,swaps and iterations increases with the increase in input size.

1.4 Merge Sort

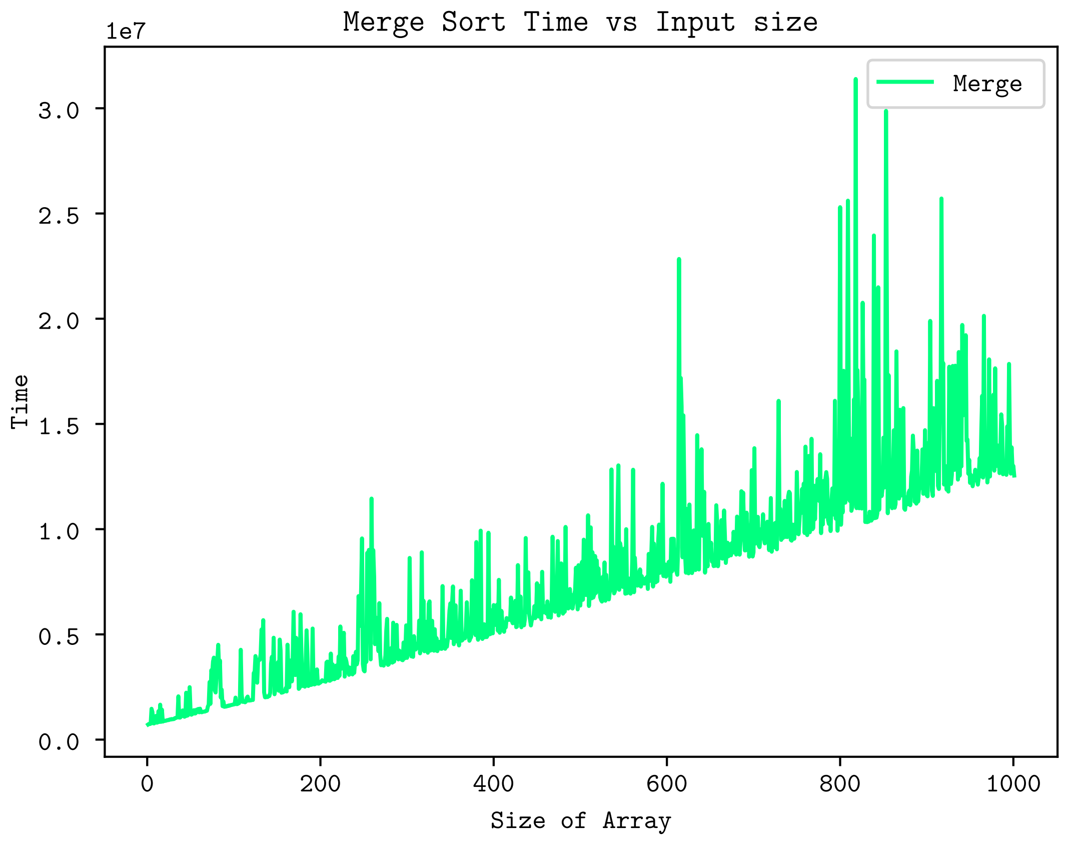
Principle

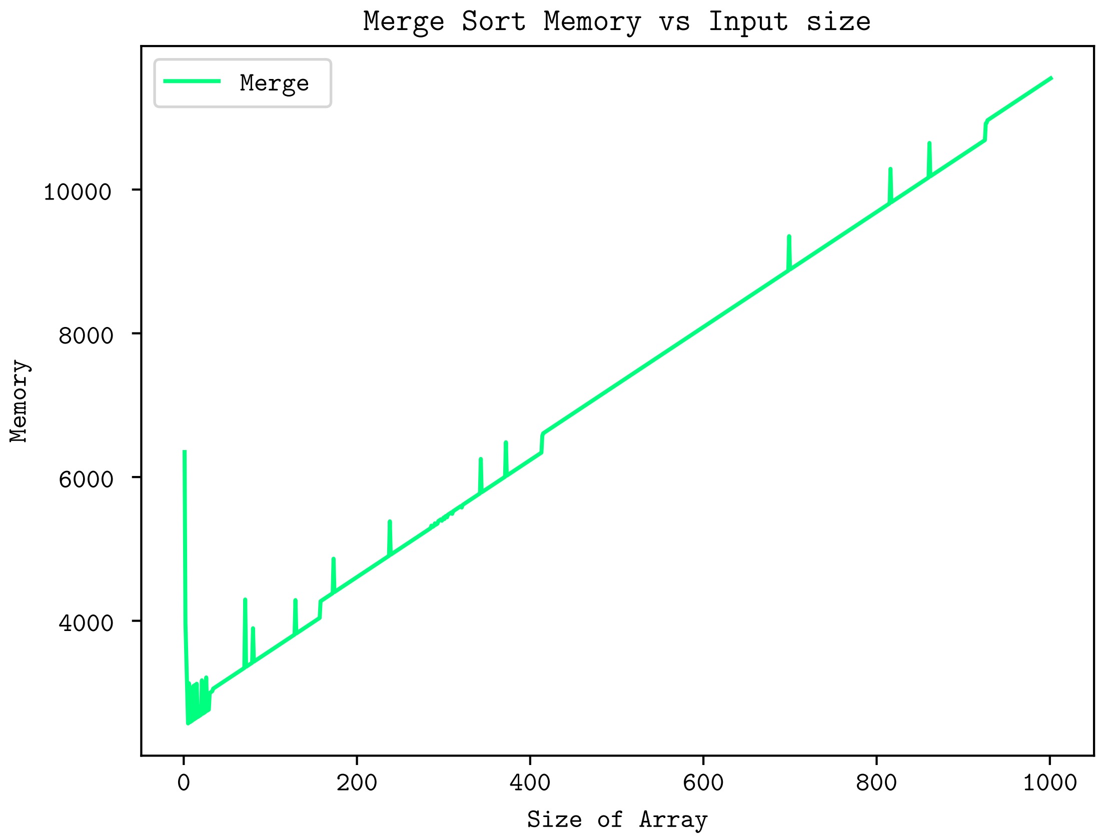
Merge sort is based on the divide and conquer principle. It divides the array into two equal halves and calls itself into each half recursively and then merges the two sorted halves. The subarrays are divided repeatedly until they become arrays of size 1. At this point they start merging in the desired order(ascending or descending) the one element arrays become 2 element arrays which merge again and become 4 elements and so on until the complete sorted list is reached.

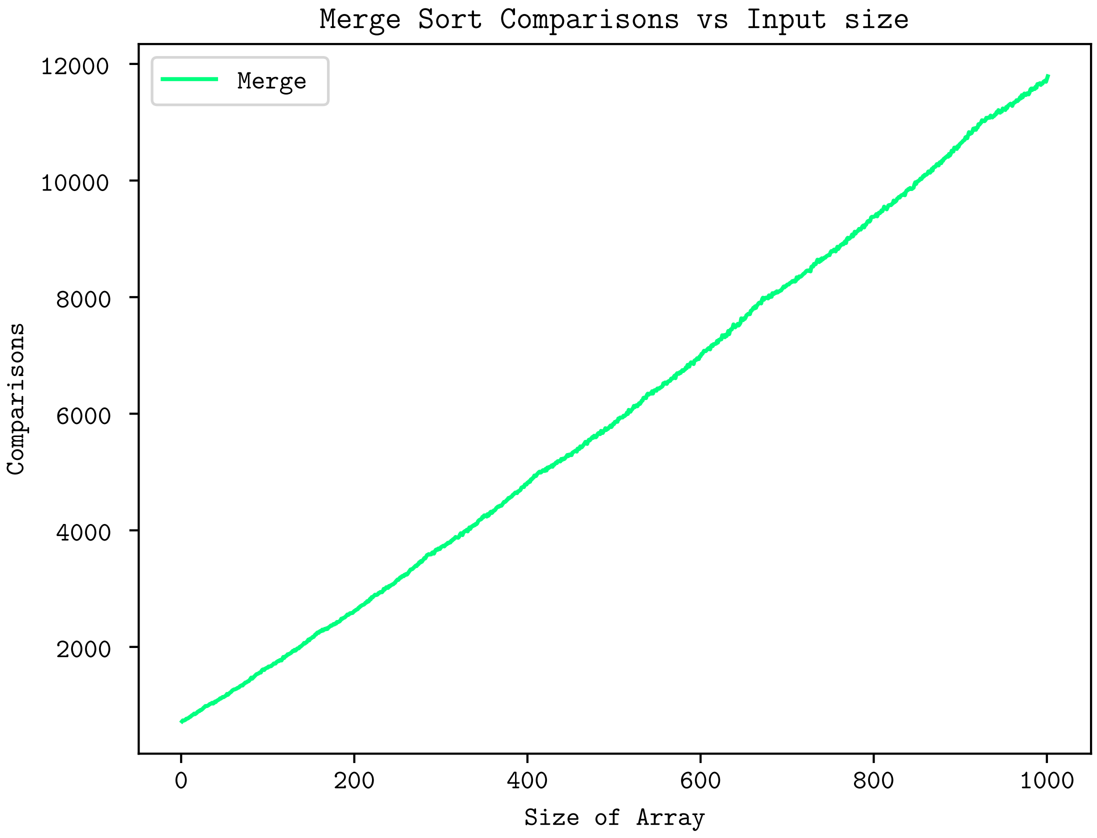
Code

```
def merge_sort(a):
    global t_comp
    global t_swap
    global t_itr
    if len(a) > 1:
        m = len(a) // 2
        l = a[:m]
        r = a[m:]
        merge_sort(l)
        t_itr+=1
        merge_sort(r)
        t_itr+=1
        size_l = len(l)
        size_r = len(r)
        i = j = k = 0
        while True:
            t_comp += 1
            if l[i] <= r[j]:
                a[k] = l[i]
                i += 1
                k += 1
                t_comp += 1
                if i == size_l:
                    while j < size_r:
                        a[k] = r[j]
                        j += 1
                        k += 1
                    return
            else:
                t_swap += 1
                a[k] = r[j]
                j += 1
                k += 1
                t_comp += 1
                if j == size_r:
                    while i < size_l:
                        a[k] = l[i]
                        i += 1
                        k += 1
                    return
```

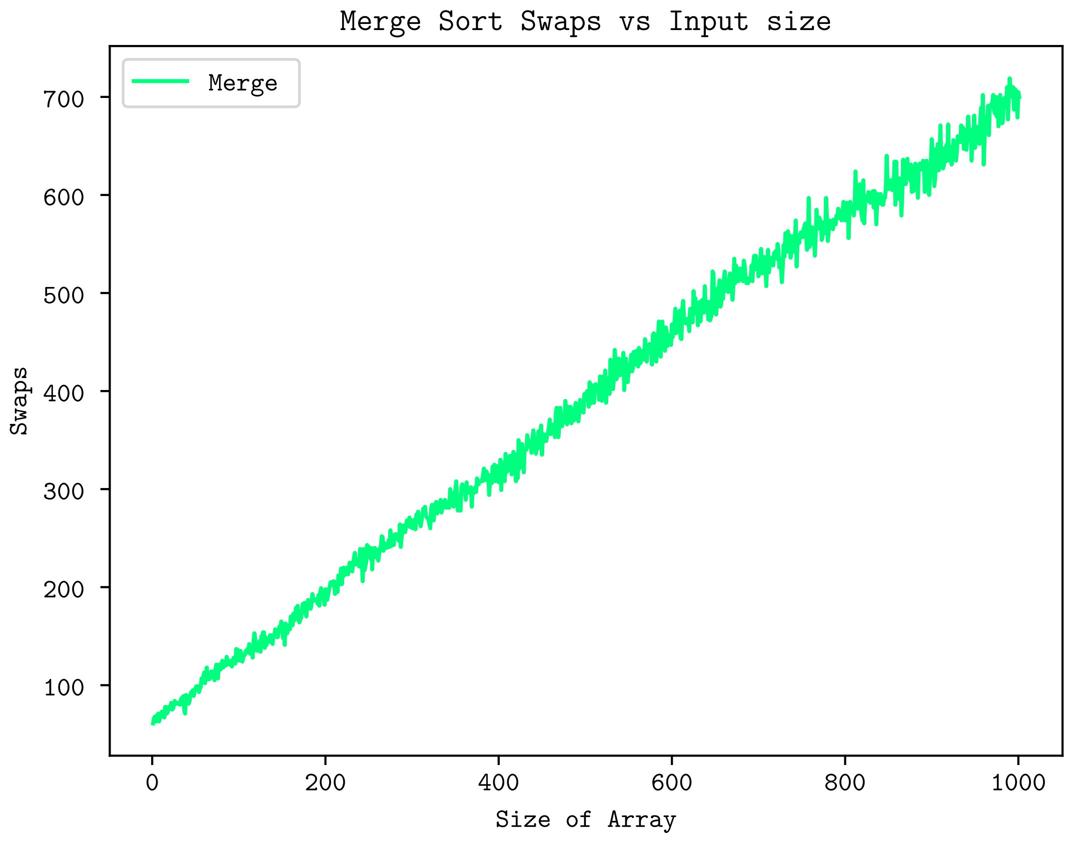
Graphs

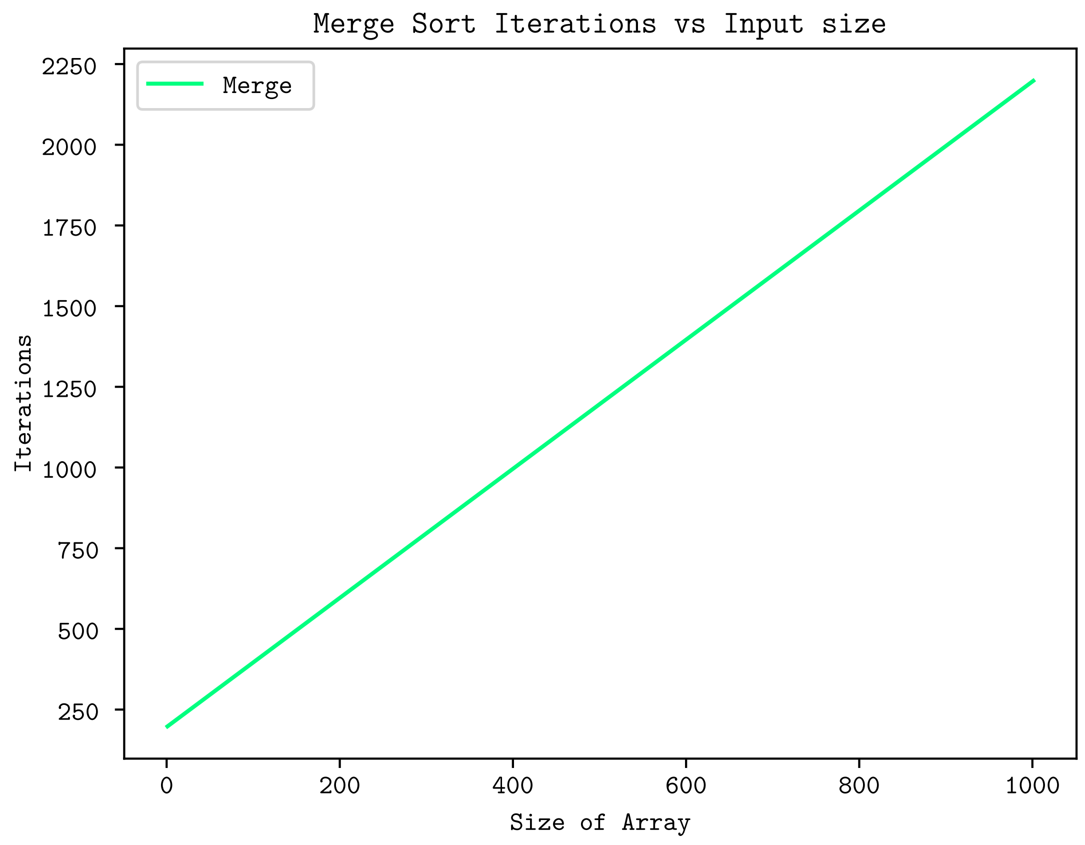



Merge Sort Memory vs Input size
Merge
Memory
Size of Array
4000
6000
8000
10000
0
200
400
600
800
1000



Merge Sort Comparisons vs Input size
Merge
Comparisons
Size of Array
12000
10000
8000
6000
4000
2000
0
200
400
600
800
1000


Merge Sort Swaps vs Input size
Merge
Swaps
100
200
300
400
500
600
700
0
200
400
600
800
1000
Size of Array



Insights

The time complexity is $O(n \log n)$ and its space complexity is $O(n)$.

1.5 Quick Sort

Principle

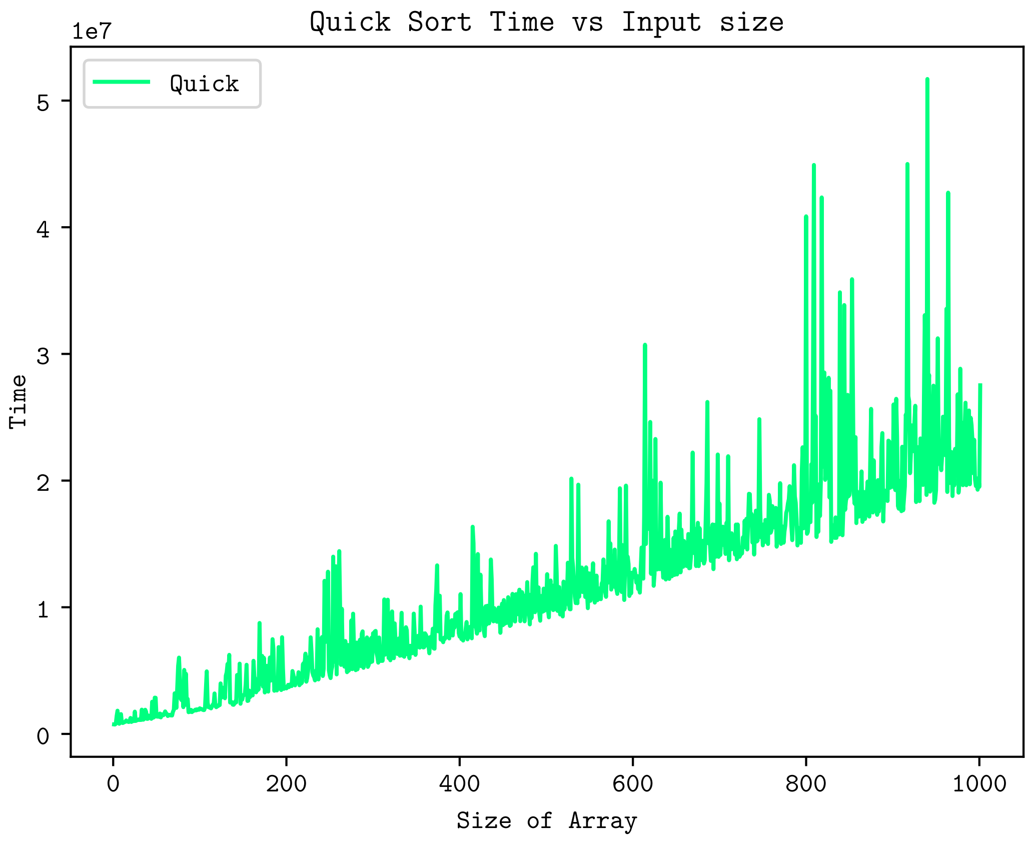
In quick sort the array is divided into two arrays, one array holds values that are smaller than the pivot and the other holds values that are larger. The pivot can be either selected randomly or it can be the rightmost or leftmost element of the array. The resulting two arrays are again divided in the similar manner. This continues until each subarray has only one element.

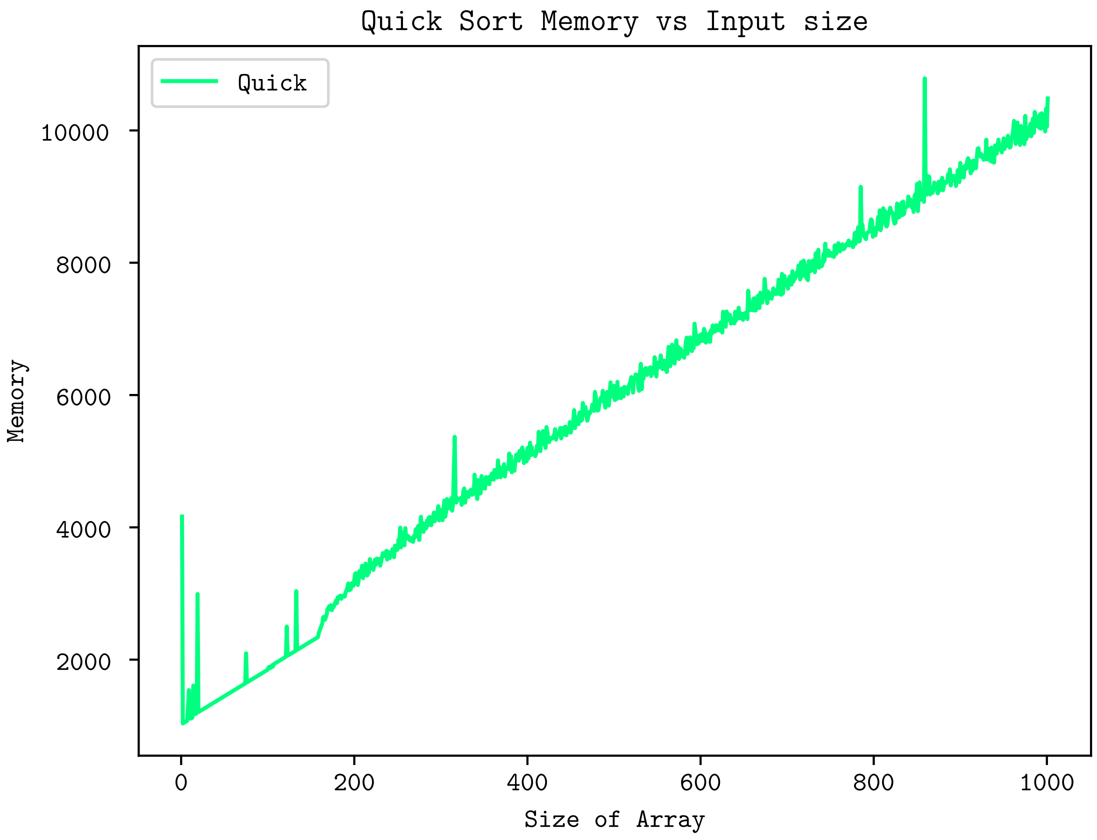
At this point the elements are already sorted, so we can just combine them to get a sorted array. This method of sorting is fast and very efficient but the performance largely depends on the selection of the pivot

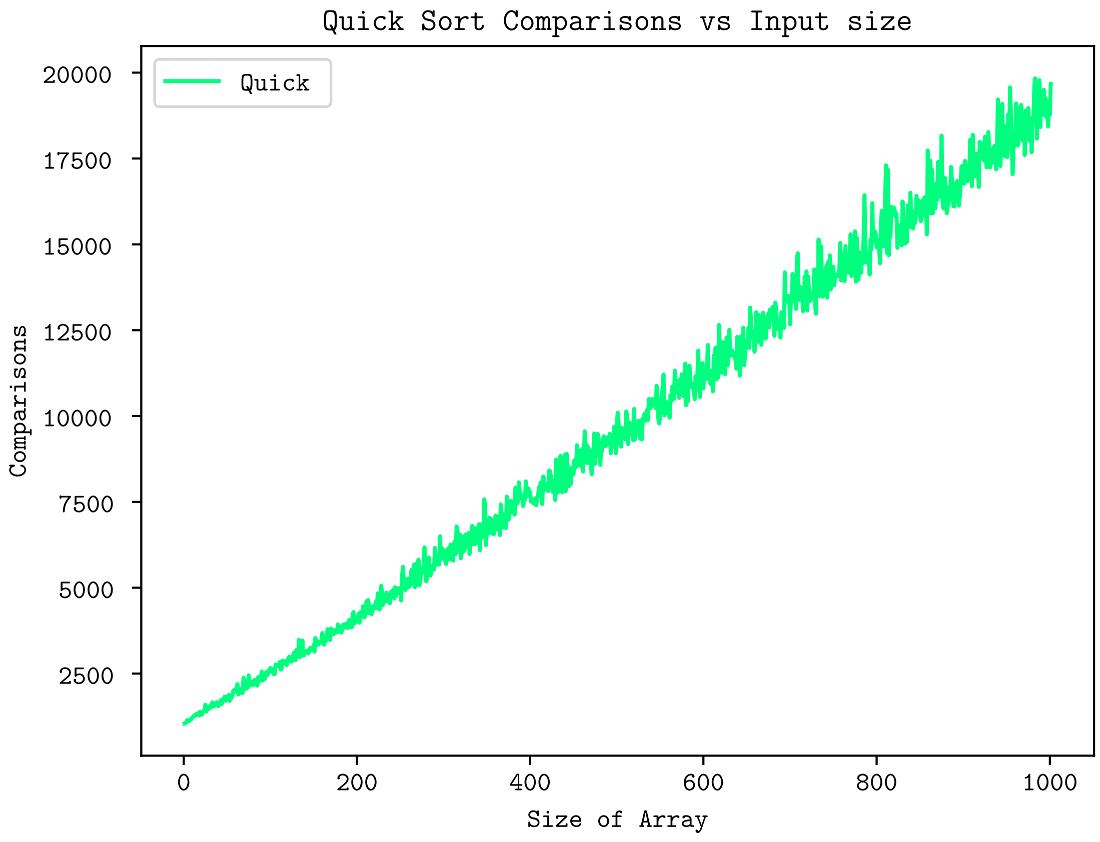
Code

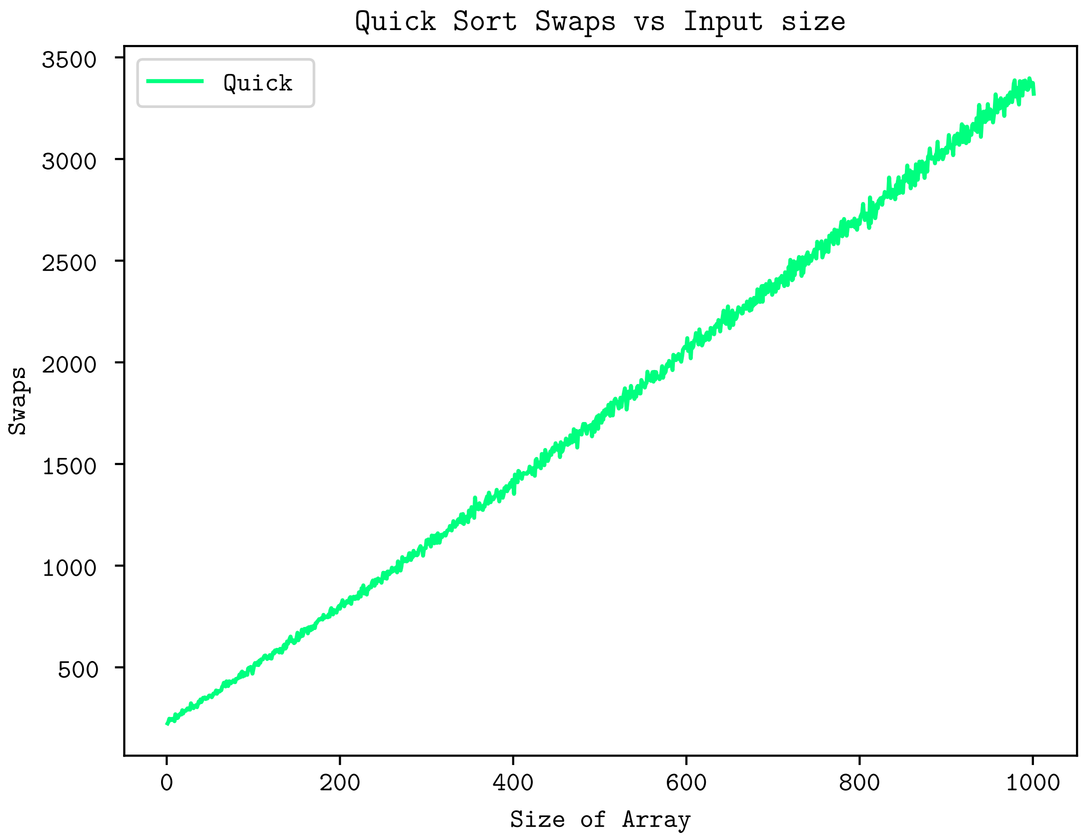
```
def qsort(a, l, r):
    global t_comp
    global t_swap
    global t_itr
    t_comp += 1
    if l < r:
        p = part(a, l, r)
        qsort(a, l, p - 1)
        t_itr += 1
        qsort(a, p + 1, r)
        t_itr += 1


def part(a, l, r):
    global t_comp
    global t_swap
    global t_itr
    t_swap += 1
    # swap middle element with the left most
    (a[l], a[(l + r) // 2]) = (a[(l + r) // 2], a[l])
    # pivot_index
    p_i = l
    # pivot value
    p = a[p_i]
    while l < r:
        t_comp += 1
        while l < len(a) and a[l] <= p:
            t_comp += 1
            l = l + 1
        while a[r] > p:
            t_comp += 1
            r = r - 1
        # Check for overlap
        if l < r:
            t_comp += 1
            t_swap += 1
            # swap misplaced elements
            (a[l], a[r]) = (a[r], a[l])
    t_swap += 1
    # put pivot in the correct place
    (a[p_i], a[r]) = (a[r], a[p_i])
    return r
```

Graphs

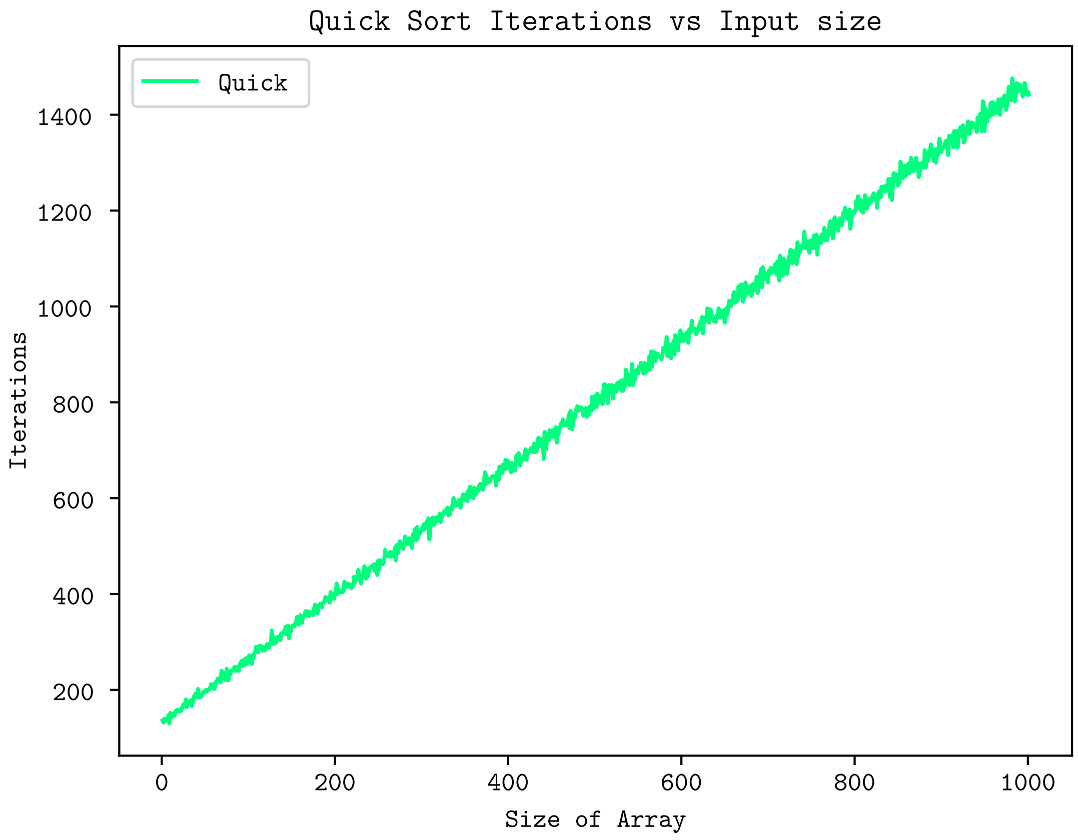



Quick Sort Memory vs Input size
Quick
10000
8000
6000
4000
2000
Memory
0
200
400
600
800
1000
Size of Array


Quick Sort Comparisons vs Input size
Quick
Comparisons
20000
17500
15000
12500
10000
7500
5000
2500
0
200
400
600
800
1000
Size of Array



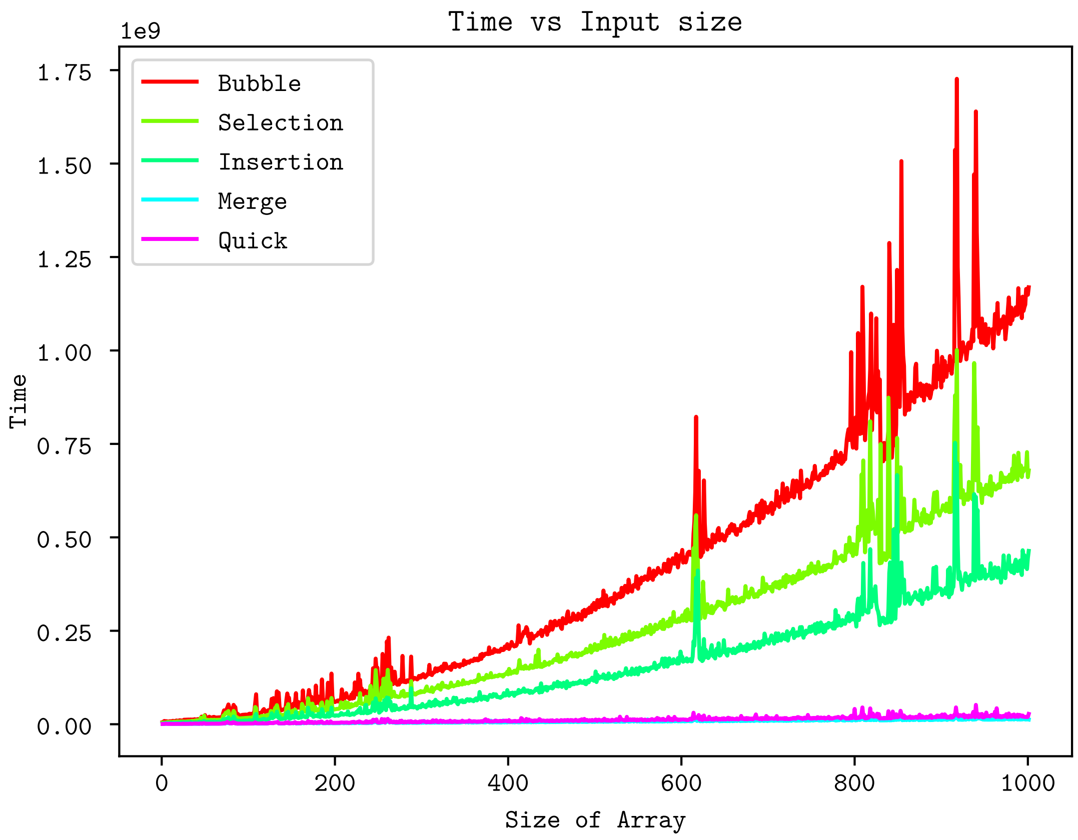
Quick Sort Swaps vs Input size
Quick
Swaps
Size of Array
0
200
400
600
800
1000
500
1000
1500
2000
2500
3000
3500

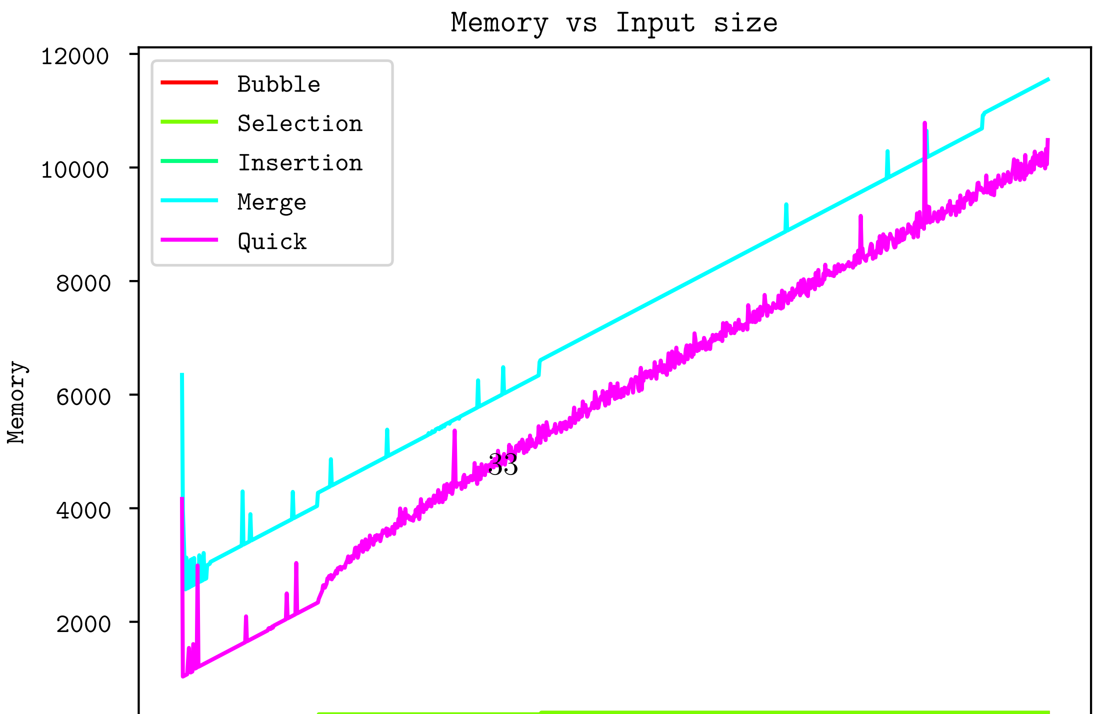


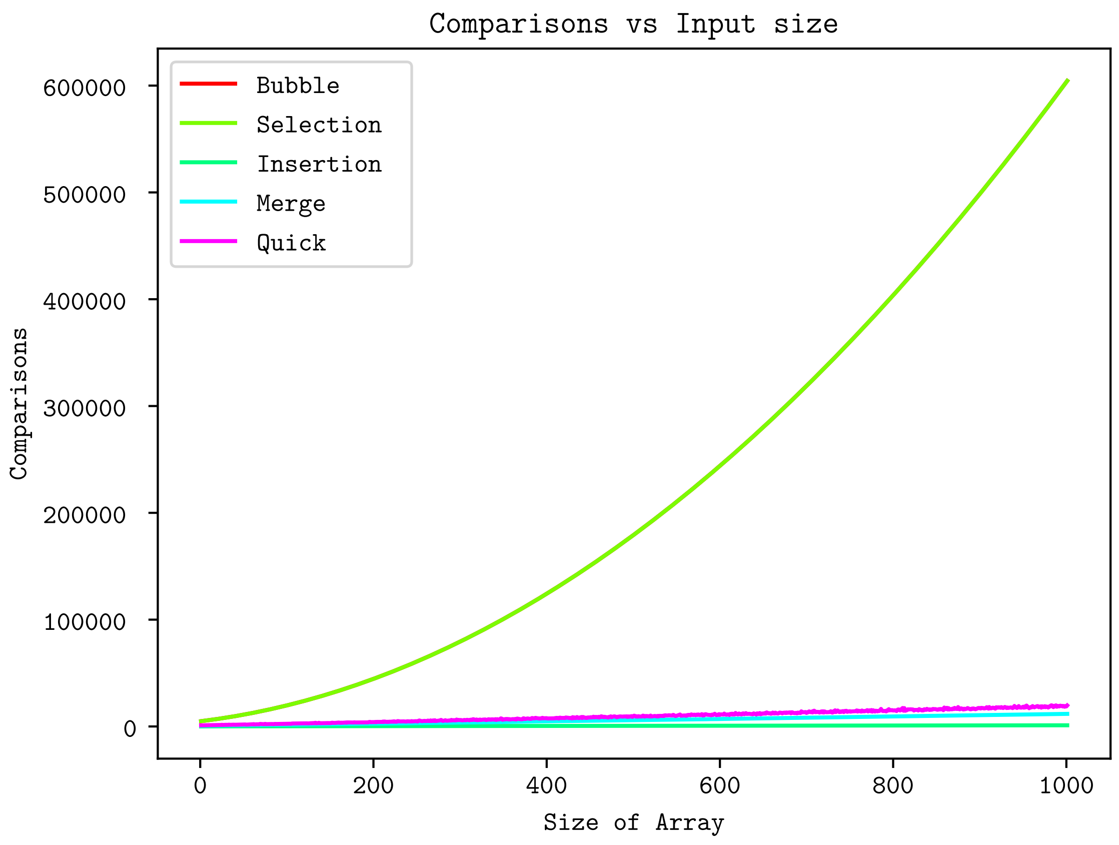
Insights

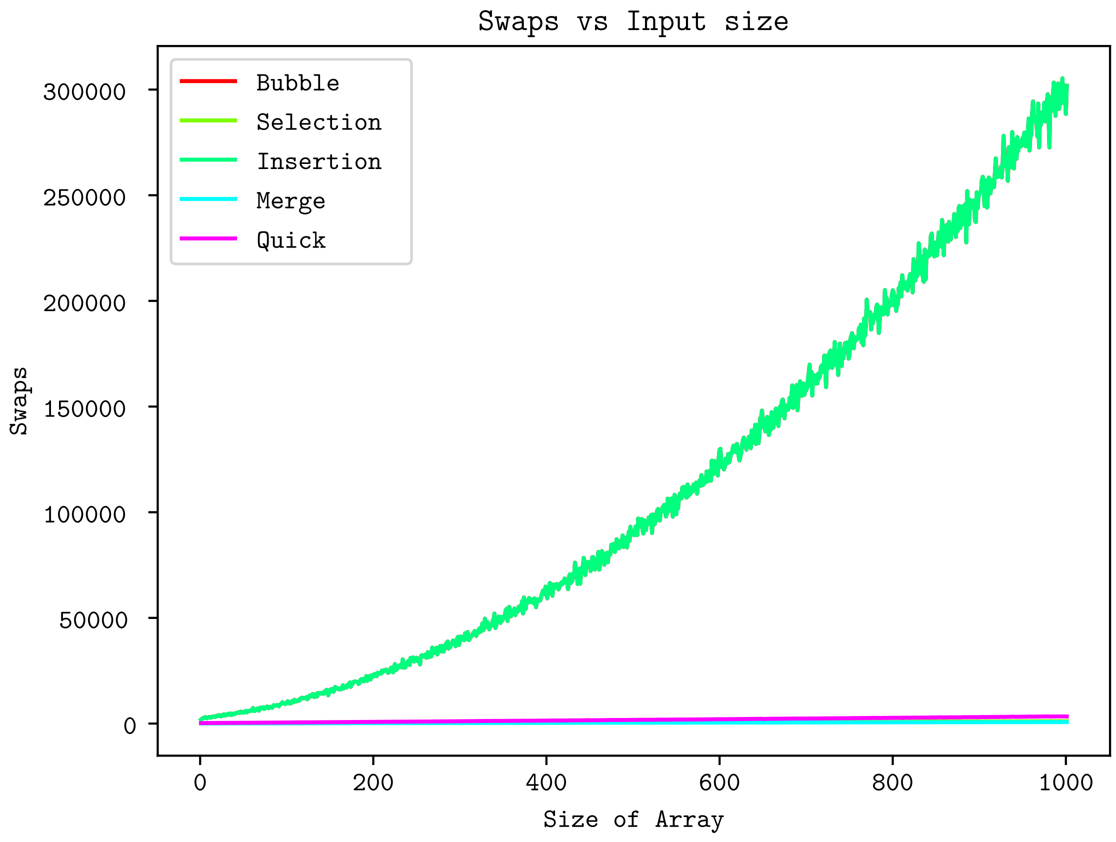
The best and average case time complexity is $O(n \log n)$ and the worst case complexity is $O(n2)$.

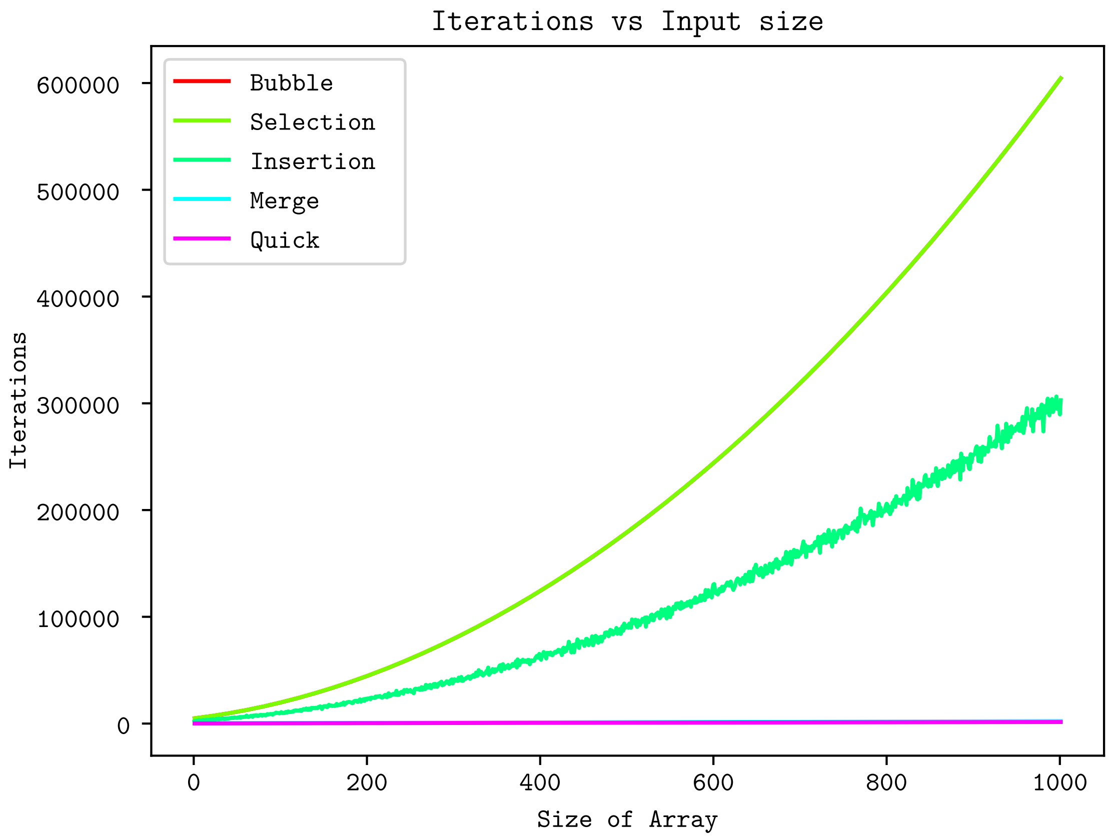
2 Comparison






Comparisons vs Input size
Bubble
Selection
Insertion
Merge
Quick
600000
500000
400000
300000
200000
100000
0
Comparisons
0
200
400
600
800
1000
Size of Array


Swaps vs Input size
Bubble
Selection
Insertion
Merge
Quick
Swaps
Size of Array
300000
250000
200000
150000
100000
50000
0
0
200
400
600
800
1000


Iterations vs Input size
Bubble
Selection
Insertion
Merge
Quick
600000
500000
400000
300000
200000
100000
0
Iterations
0
200
400
600
800
1000
Size of Array