**Kathmandu University**

**Department of Computer Science and Engineering**

**Dhulikhel, Kavre**

**Lab 1**

**COMP 314**

**(CE-III/II)**

**Submitted by**

**Puspa Hari Ghimire (19)**

**Submitted to**

**Dr. Prakash Poudyal**

**Department of Computer Science and Engineering**

**Submission Date:**

**January 27, 2025**

**Lab 1: Implementation of Sorting Algorithms**

The main idea behind implementing these algorithms is to compare the time complexity and analyze the best, worst and average case of each algorithm. In this lab, we have taken 5 different sorting algorithms. Array sizes are 500, 1000, 1500, 2000 and 2500 except for Quicksort. For each array size, we have generated random numbers from 1 to 10,000 and performed these algorithms 30 times to get the average time required to sort the array, ascending array and descending array. (for ex. Selection sort is performed 30 times to sort an array of size 500 for 3 cases viz. given array, ascending array and descending array. Similarly, it is done for other remaining sizes.)

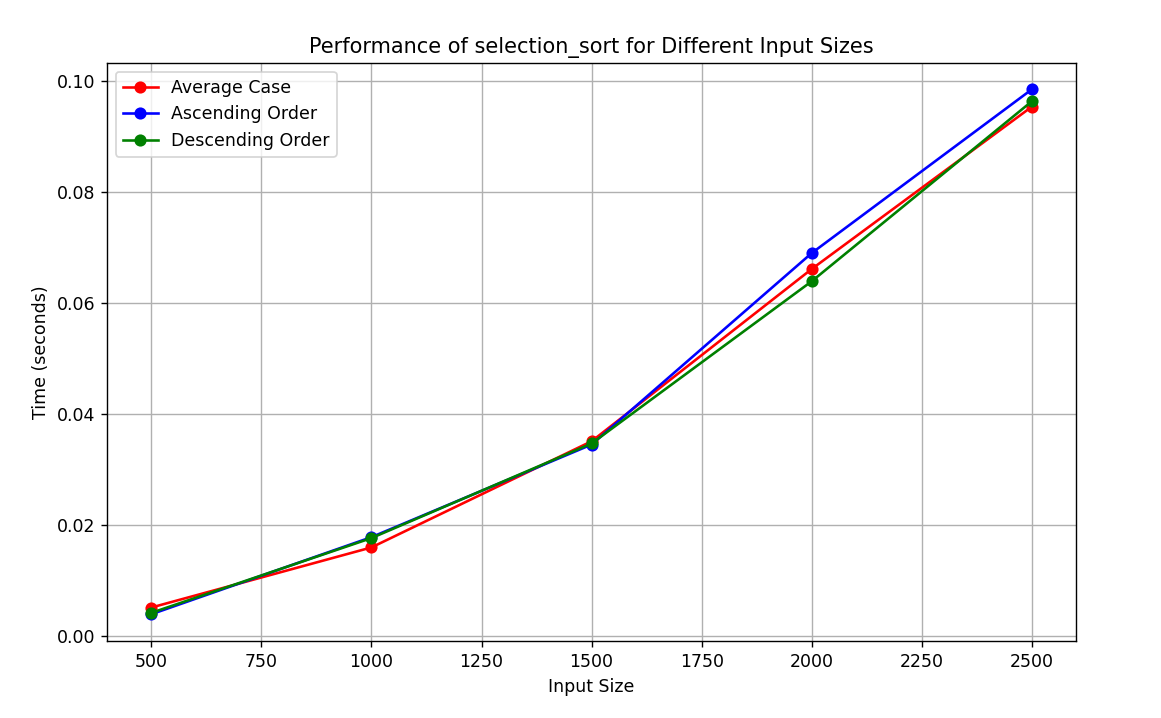
**Selection sort:**

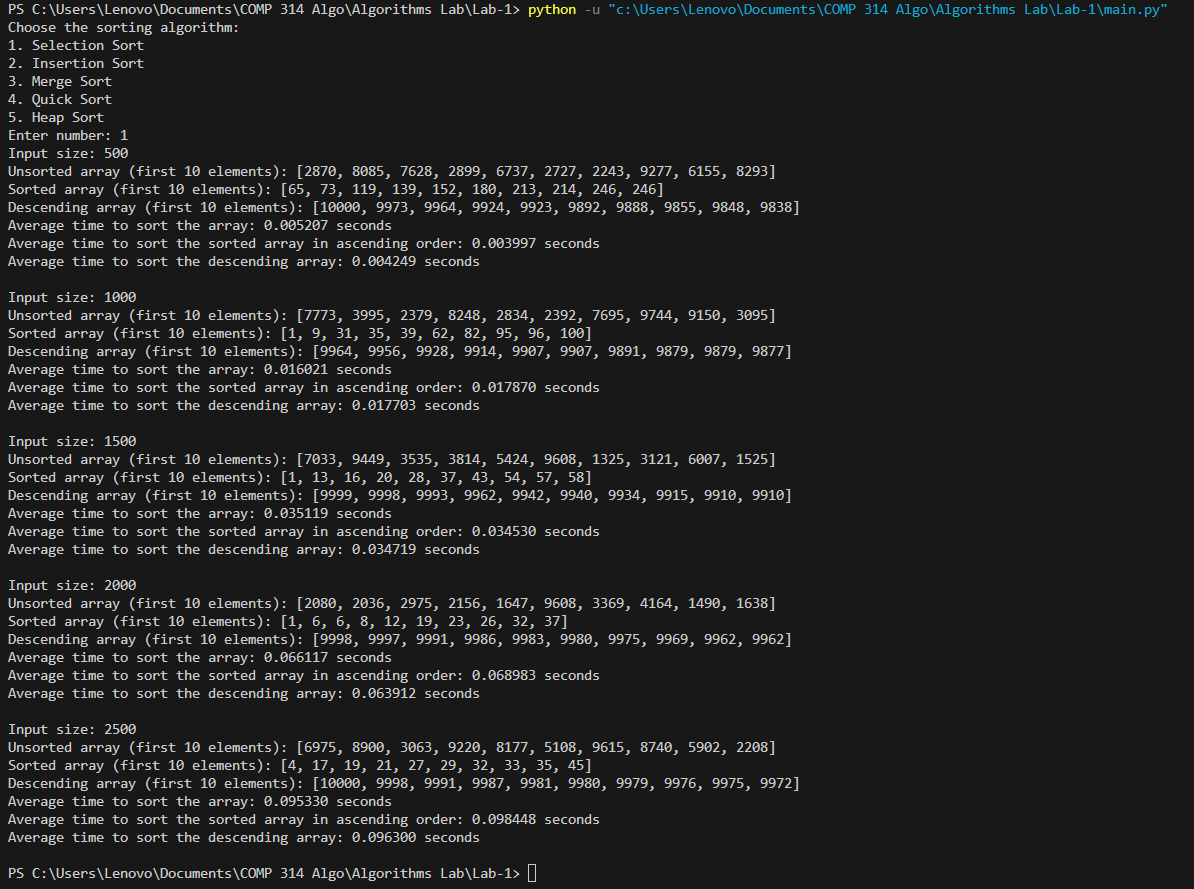
It sorts an array by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. The list at any moment is divided into two sublists.

Steps:

1. Select the smallest element from the unsorted sublist and exchange it with the element at the beginning of the unsorted data.
2. Repeat Step 1 until there is no element in the unsorted sublist.

**Output:**





**Analysis:**  
From the above graph, we can see that, selection sort performs similar is all three cases. The time complexity of average, best and worst case for this algorithm is O(n2). Even when the array is already sorted, the time taken is similar to that of descending array.We can verify it from our obtained graph.

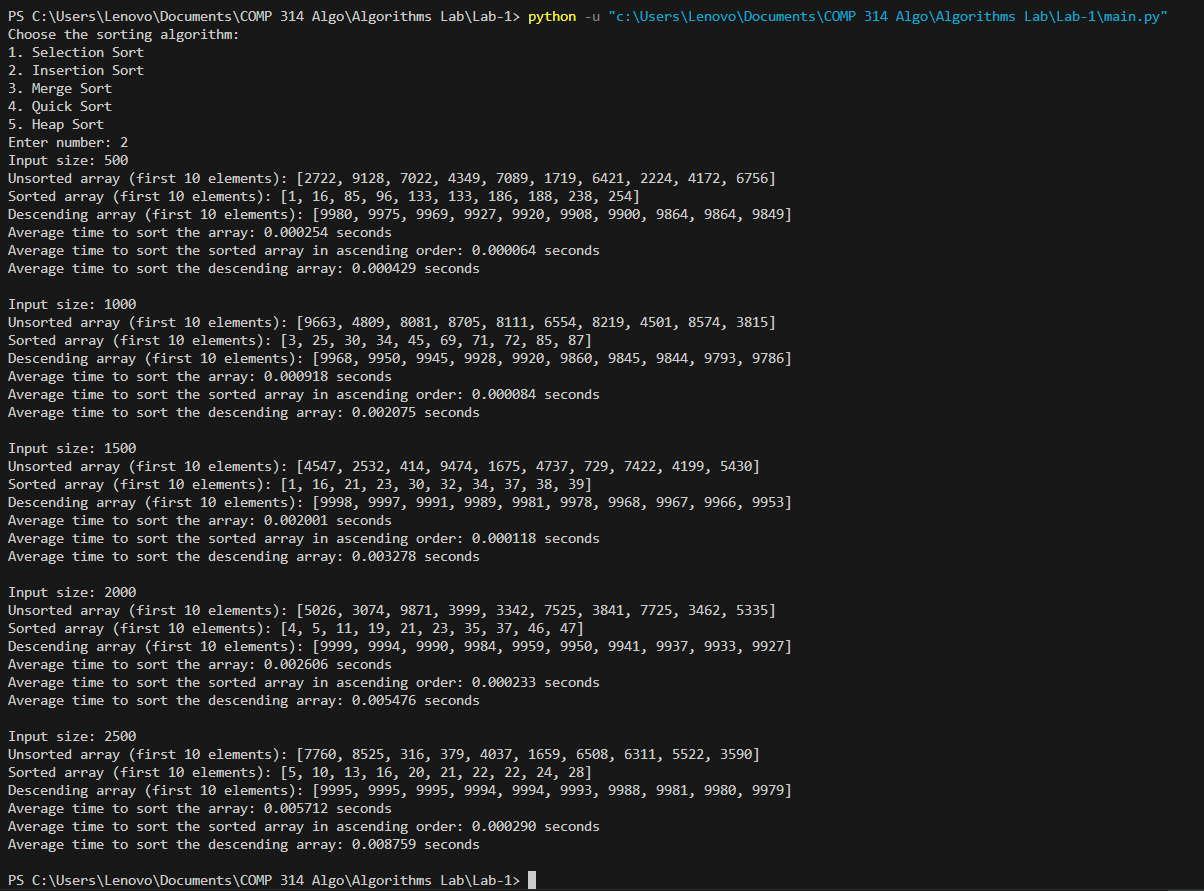
**Insertion sort:**

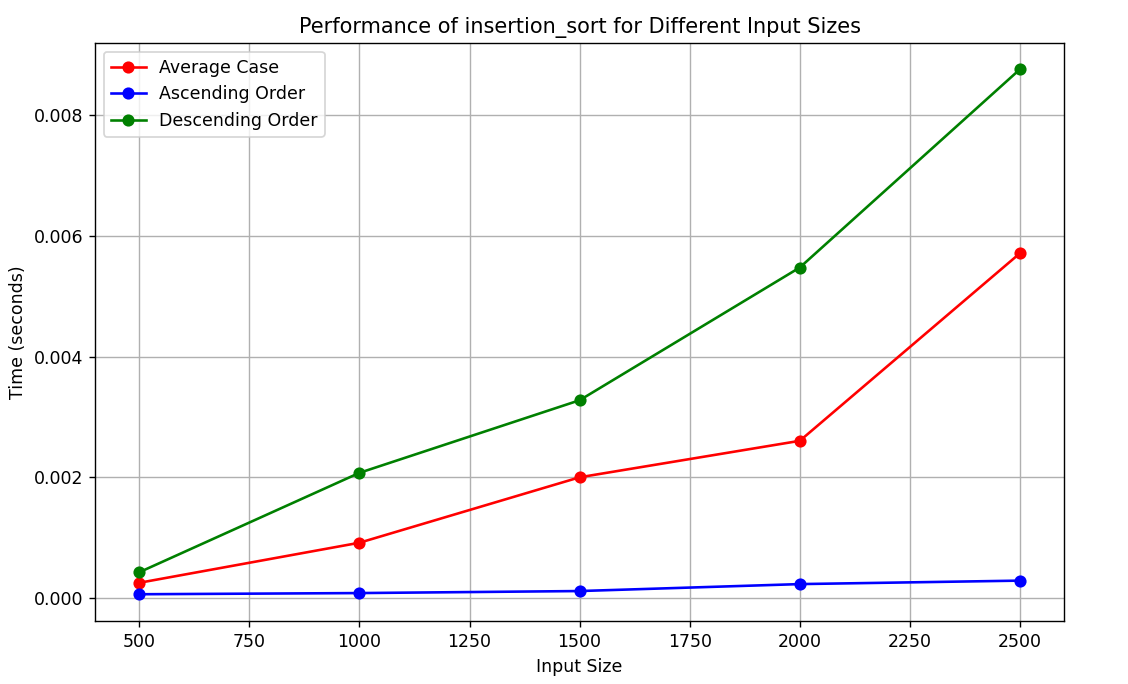
One of the most common sorting techniques used by card players. As they pick up each card, they insert it into the proper sequence in their hand.

Steps:

1. We start with second element of the array as first element in the array is assumed to be sorted.
2. Compare second element with the first element and check if the second element is smaller then swap them.
3. Move to the third element and compare it with the first two elements and put at its correct position
4. Repeat until the entire array is sorted.

**Output:**





**Analysis:**

We know that the time complexity for best case in insertion sort is O(n) and for average and worst case is O(n2). From the above graph, we can see that the worst case (descending array), the time complexity is similar to average case. But when the sorted array (ascending array) is given, the performance is very fast compared to other cases.

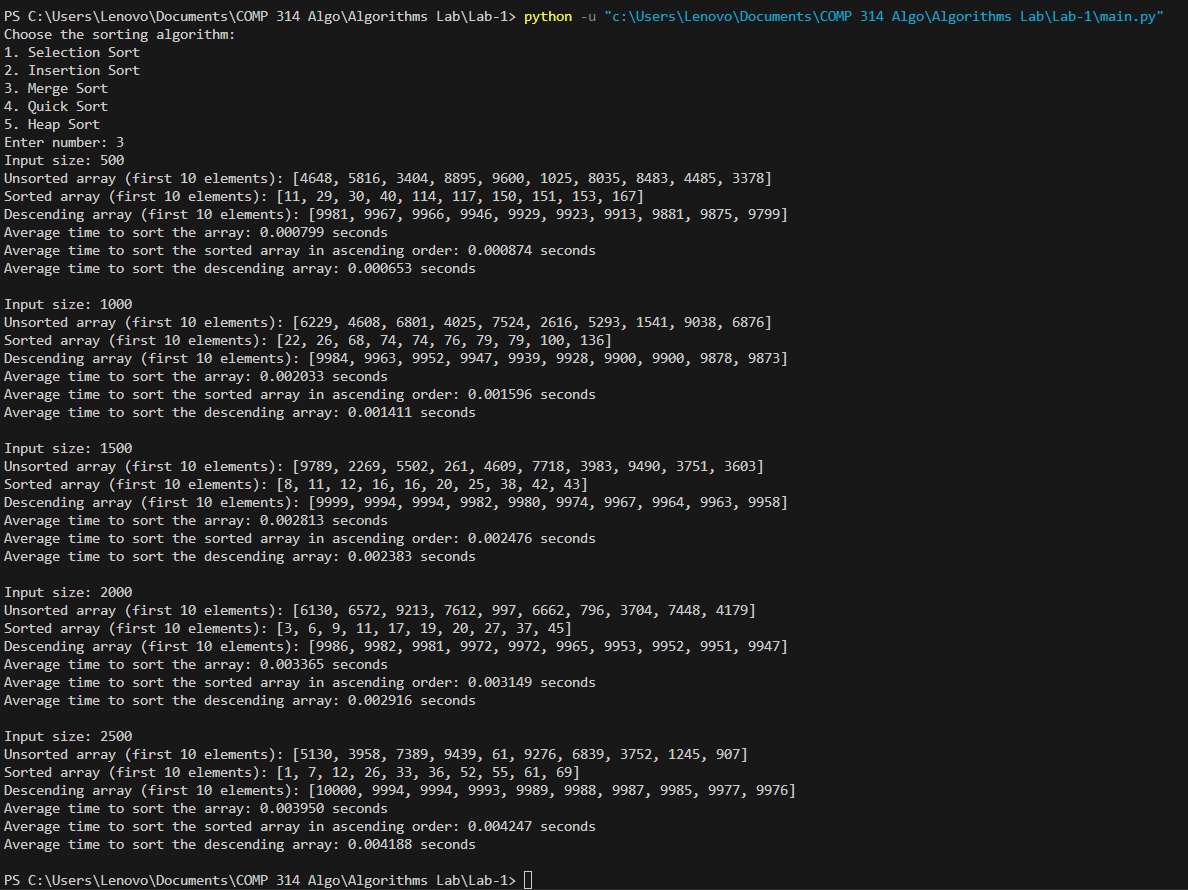
**Merge sort:**

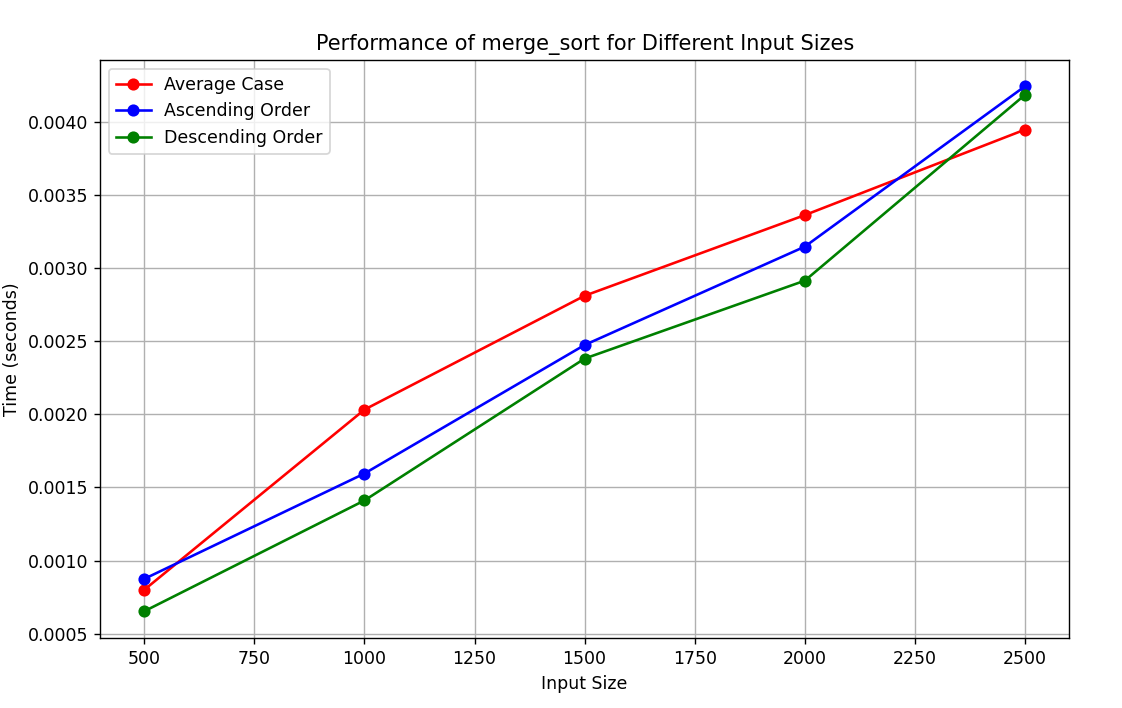
Merge sort uses divide-and-conquer strategy. The original problem is partitioned into simpler sub-problems, each subproblem is considered independently. Subdivision continues until sub problems obtained are simple.

Steps:

1. Divide: partition the list into two roughly equal parts, S1 and S2, called the left and the right sublists
2. Conquer: recursively sort S1 and S2
3. Combine: merge the sorted sublists

**Output:**





**Analysis:**

Time complexity for every case in merge sort is O(nlogn). From the graph, we see the trend that every case has similar time complexity for each input sizes. Even though array is already sorted, it still takes same time as descending array.

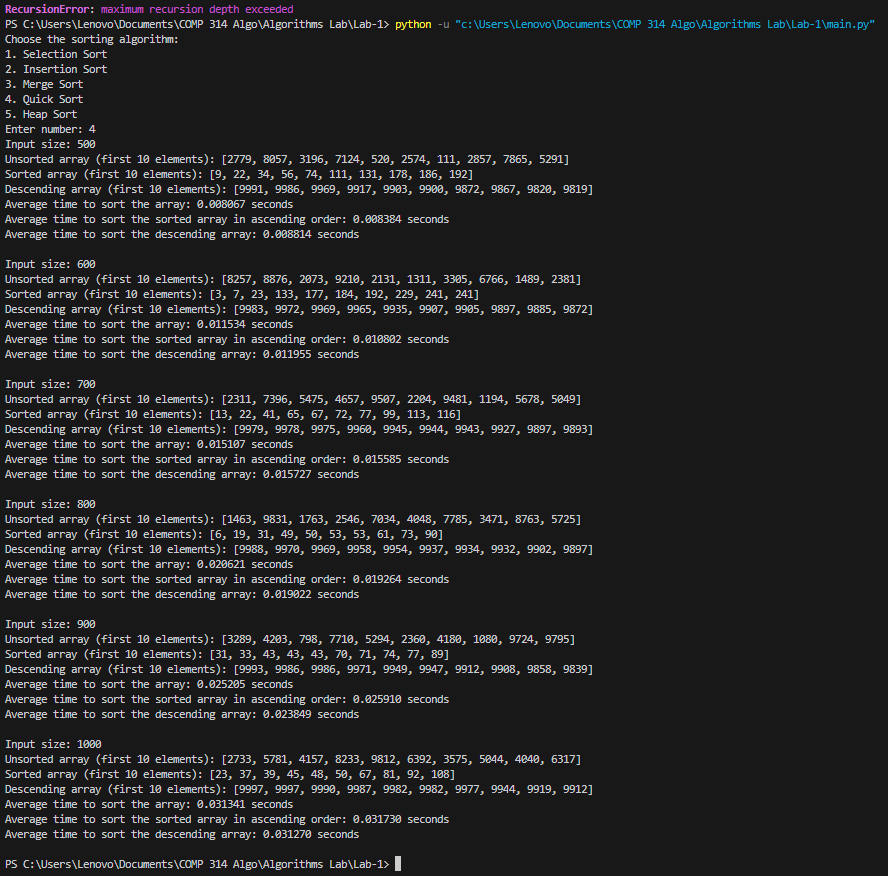
**Quick sort:**

Like merge sort, quick sort also uses divide-and-conquer paradigm.

Steps:

1. Divide: Select any element from the list. Call it the pivot. Then partition the list into two sublists such that all the elements in the left sublist are less than or equal to the pivot and those of the right sublist are greater than or equal to the pivot
2. Conquer: Recursively sort the two sublists
3. Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire list is now sorted.

**Output:**





**Analysis:**

In this implementation of quicksort, we have taken the pivot element as the last element of the array. The array sizes are 500, 600, 700, 800, 900 and 1000. We took smaller sizes because using size >1000 exceeded the maximum recursion depth that Python provides. Pivot is always the greatest element in ascending array and smallest in descending array. Doing so, the pivot element doesn’t guarantee the division of array into two equal halves so our time complexity is given in graph for the worst case which is O(n2). Best case, Ω(nlogn), occurs when the pivot element divides the array into two equal halves. Average case, θ(nlogn) is obtained when the pivot divides the array into two parts, but not necessarily equal.

**Heap sort:**

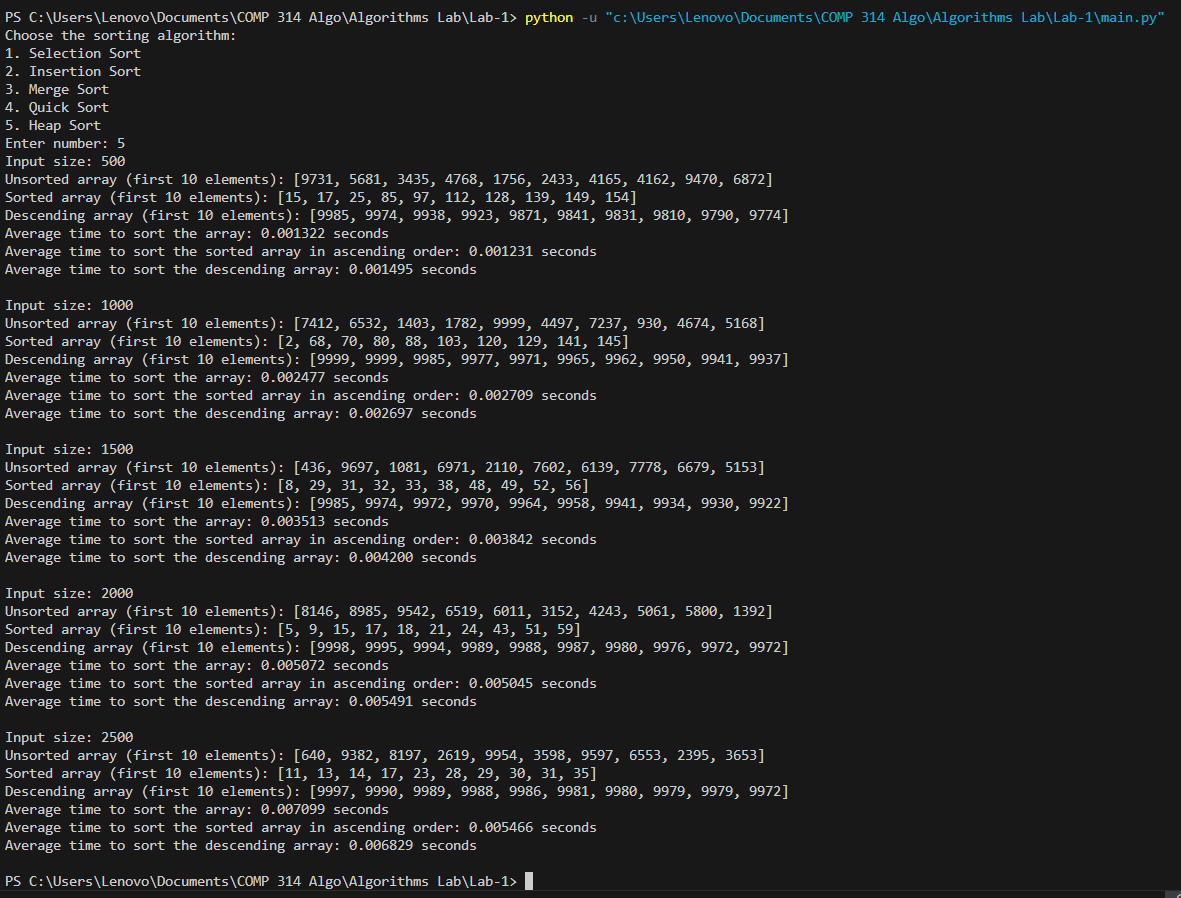
To implement the heap sort using a max-heap, we need two basic algorithms:

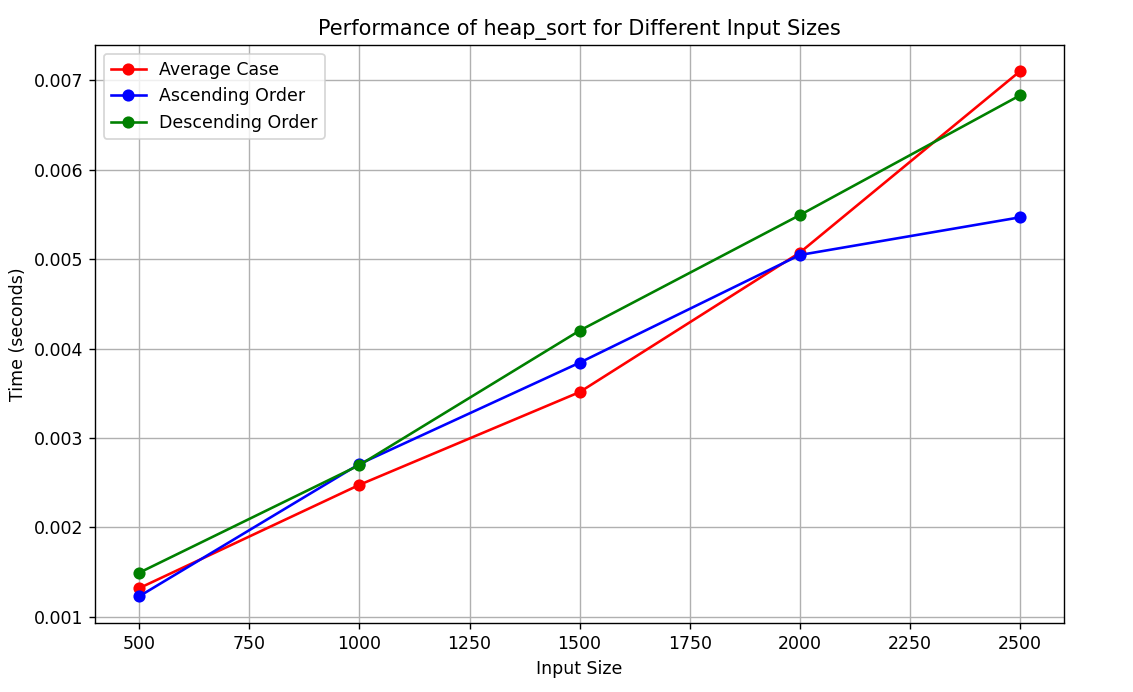
* Max-heapify: Maintains the max-heap property by pushing the root down the tree until it is in its correct position in the heap.
* Build-max-heap: Produces a max-heap from an unordered input array.

Steps:

1. Convert the array into a max heap
2. Find the largest element of the list (i.e., the root of the heap) and then place it at the end of the list. Decrement the heap size by 1 and readjust the heap
3. Repeat Step 2 until the unsorted list is empty

**Output:**





**Analysis:**

Heap sort has a time complexity of O(nlogn) in all cases. We can see from the above graph that the time complexity for given array, ascending array and descending array is much similar. Among all the sorting algorithms we implemented, heap sort has consistent time complexity which is verified from the graph.

**Conclusion:**

|  |  |  |  |
| --- | --- | --- | --- |
| Sorting algorithms | Best case | Average case | Worst case |
| Selection | O(n2) | O(n2) | O(n2) |
| Insertion | O(n) | O(n2) | O(n2) |
| Merge | O(nlogn) | O(nlogn) | O(nlogn) |
| Quick | O(nlogn) | O(nlogn) | O(n2) |
| Heap | O(nlogn) | O(nlogn) | O(nlogn) |

In conclusion, the implementation of these sorting algorithms viz. Selection, Insertion, Merge, Quick, and Heap sort gave us a good look at how different sorting methods work and how they perform.

Each algorithm has its own pros and cons. Insertion and Selection sort are simple and work fine for small lists, but they slow down with larger ones because their time complexity is O(n²). Merge and Heap sort are more efficient for larger datasets with a time complexity of O(nlogn), making them faster for bigger tasks. Quick sort tends to be the fastest in practice but choosing of the right pivot is very crucial.

Overall, the best sorting algorithm depends on the specific needs of the problem, like the size of the data, whether stability is important, or memory limits. Understanding these sorting methods helps in choosing the right one for different situations.