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.)

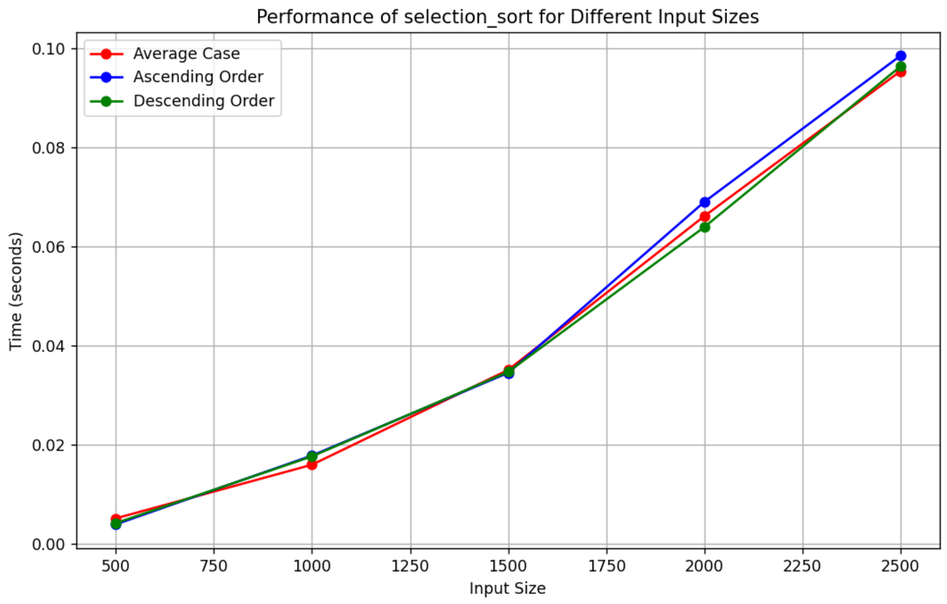
Selection sort:

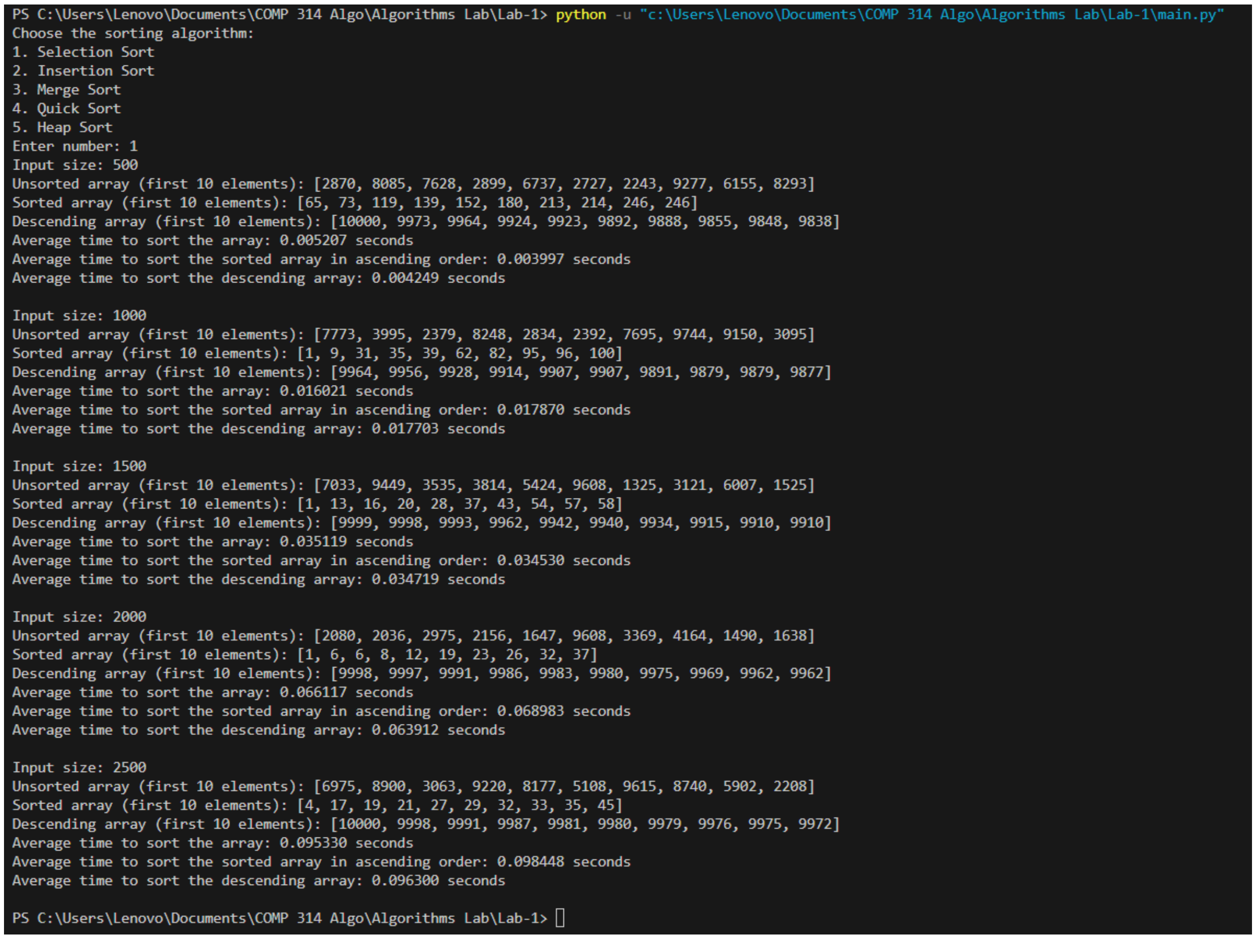
It sorts an array by repeatedly selecting the smallest element from the unsorted portion and swapping it with the first unsorted element. The list at any moment is divided into two sublists.

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:





```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> python -u "c:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1\main.py"
Choose the sorting algorithm:
1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort
Enter number: 1
Input size: 500
Unsorted array (first 10 elements): [2870, 8085, 7628, 2899, 6737, 2727, 2243, 9277, 6155, 8293]
Sorted array (first 10 elements): [65, 73, 119, 139, 152, 180, 213, 214, 246, 246]
Descending array (first 10 elements): [10000, 9973, 9964, 9924, 9923, 9892, 9888, 9855, 9848, 9838]
Average time to sort the array: 0.005207 seconds
Average time to sort the sorted array in ascending order: 0.003997 seconds
Average time to sort the descending array: 0.004249 seconds

Input size: 1000
Unsorted array (first 10 elements): [7773, 3995, 2379, 8248, 2834, 2392, 7695, 9744, 9150, 3095]
Sorted array (first 10 elements): [1, 9, 31, 35, 39, 62, 82, 95, 96, 100]
Descending array (first 10 elements): [9964, 9956, 9928, 9914, 9907, 9907, 9891, 9879, 9879, 9877]
Average time to sort the array: 0.016021 seconds
Average time to sort the sorted array in ascending order: 0.017870 seconds
Average time to sort the descending array: 0.017703 seconds

Input size: 1500
Unsorted array (first 10 elements): [7033, 9449, 3535, 3814, 5424, 9608, 1325, 3121, 6007, 1525]
Sorted array (first 10 elements): [1, 13, 16, 20, 28, 37, 43, 54, 57, 58]
Descending array (first 10 elements): [9999, 9998, 9993, 9962, 9942, 9940, 9934, 9915, 9910, 9910]
Average time to sort the array: 0.035119 seconds
Average time to sort the sorted array in ascending order: 0.034530 seconds
Average time to sort the descending array: 0.034719 seconds

Input size: 2000
Unsorted array (first 10 elements): [2080, 2036, 2975, 2156, 1647, 9608, 3369, 4164, 1490, 1638]
Sorted array (first 10 elements): [1, 6, 6, 8, 12, 19, 23, 26, 32, 37]
Descending array (first 10 elements): [9998, 9997, 9991, 9986, 9983, 9980, 9975, 9969, 9962, 9962]
Average time to sort the array: 0.066117 seconds
Average time to sort the sorted array in ascending order: 0.068983 seconds
Average time to sort the descending array: 0.063912 seconds

Input size: 2500
Unsorted array (first 10 elements): [6975, 8900, 3063, 9220, 8177, 5108, 9615, 8740, 5902, 2208]
Sorted array (first 10 elements): [4, 17, 19, 21, 27, 29, 32, 33, 35, 45]
Descending array (first 10 elements): [10000, 9998, 9991, 9987, 9981, 9980, 9979, 9976, 9975, 9972]
Average time to sort the array: 0.095330 seconds
Average time to sort the sorted array in ascending order: 0.098448 seconds
Average time to sort the descending array: 0.096300 seconds

PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> 
```

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(n^2). 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.

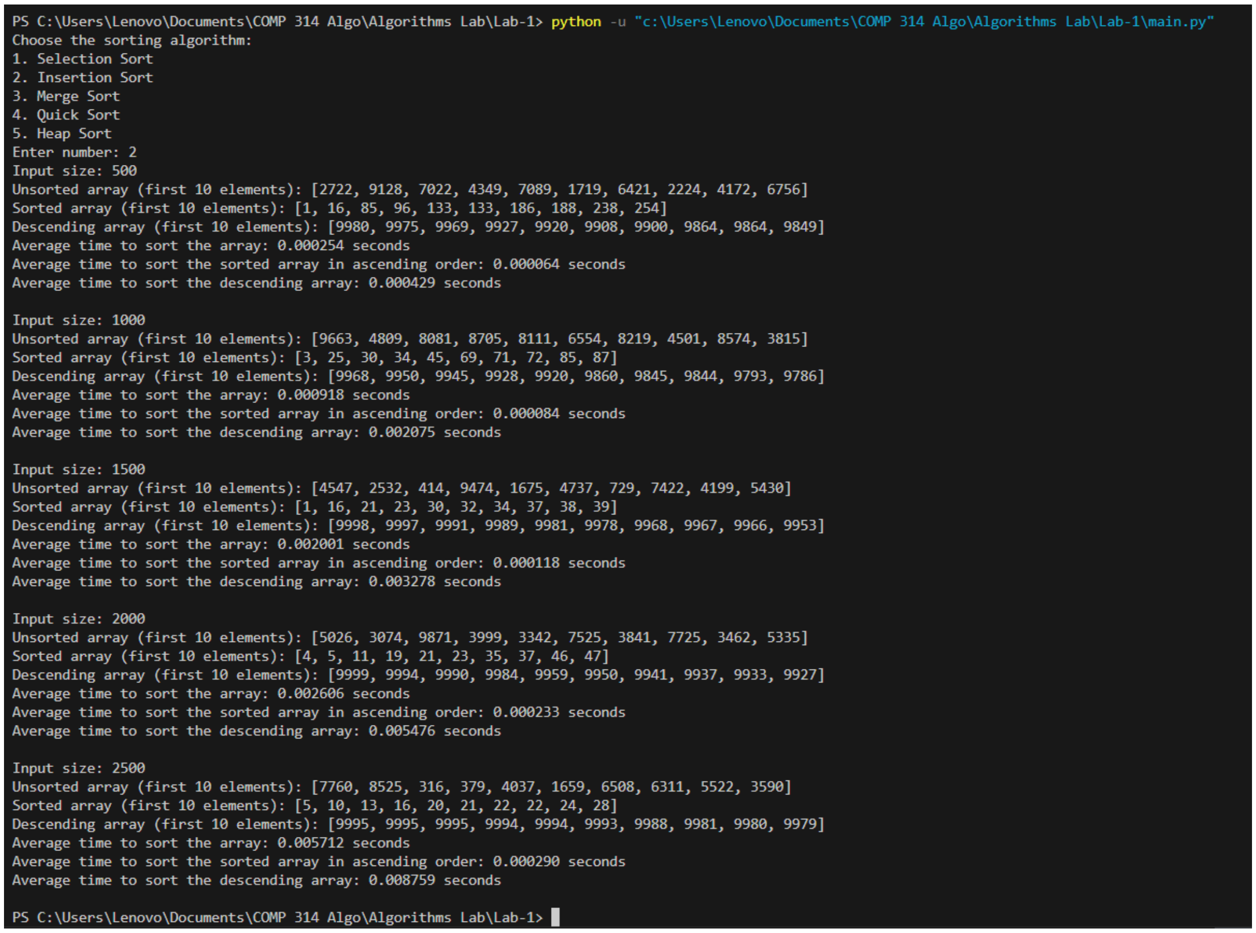
Insertion sort:

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.

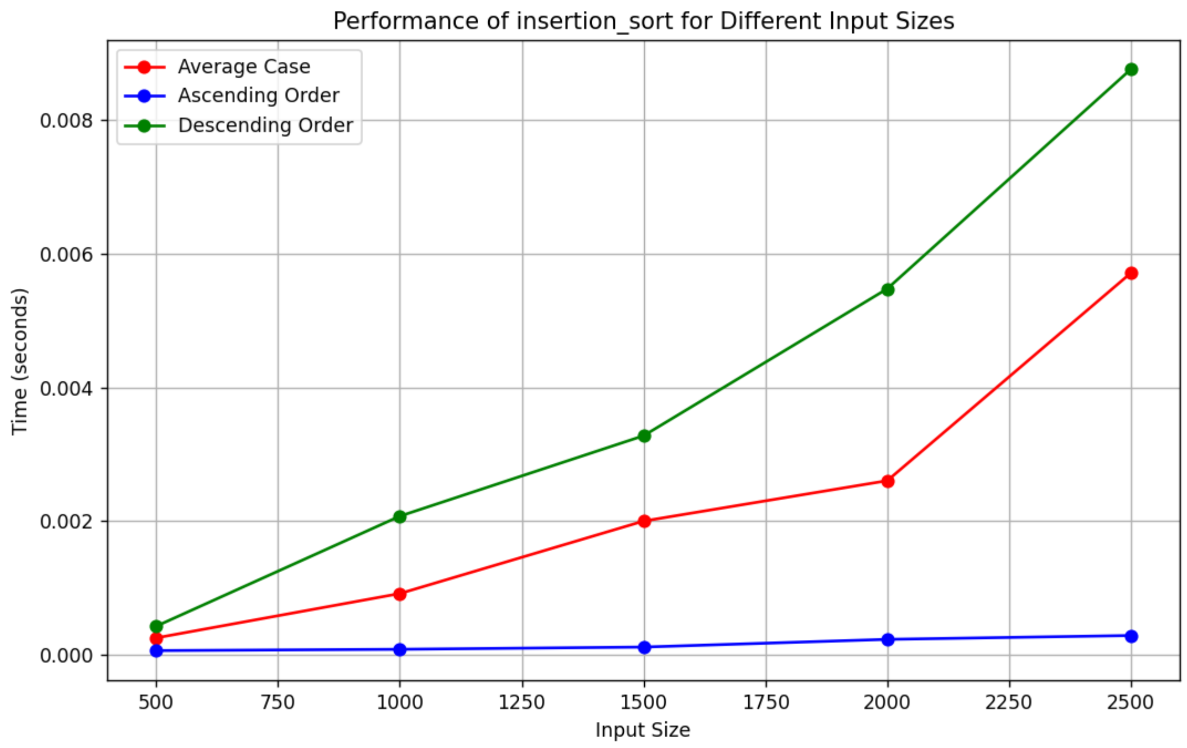
Steps:

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:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> python -u "c:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1\main.py"
Choose the sorting algorithm:
1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort
Enter number: 2
Input size: 500
Unsorted array (first 10 elements): [2722, 9128, 7022, 4349, 7089, 1719, 6421, 2224, 4172, 6756]
Sorted array (first 10 elements): [1, 16, 85, 96, 133, 133, 186, 188, 238, 254]
Descending array (first 10 elements): [9980, 9975, 9969, 9927, 9920, 9908, 9900, 9864, 9864, 9849]
Average time to sort the array: 0.000254 seconds
Average time to sort the sorted array in ascending order: 0.000064 seconds
Average time to sort the descending array: 0.000429 seconds

Input size: 1000
Unsorted array (first 10 elements): [9663, 4809, 8081, 8705, 8111, 6554, 8219, 4501, 8574, 3815]
Sorted array (first 10 elements): [3, 25, 30, 34, 45, 69, 71, 72, 85, 87]
Descending array (first 10 elements): [9968, 9950, 9945, 9928, 9920, 9860, 9845, 9844, 9793, 9786]
Average time to sort the array: 0.000918 seconds
Average time to sort the sorted array in ascending order: 0.000084 seconds
Average time to sort the descending array: 0.002075 seconds

Input size: 1500
Unsorted array (first 10 elements): [4547, 2532, 414, 9474, 1675, 4737, 729, 7422, 4199, 5430]
Sorted array (first 10 elements): [1, 16, 21, 23, 30, 32, 34, 37, 38, 39]
Descending array (first 10 elements): [9998, 9997, 9991, 9989, 9981, 9978, 9968, 9967, 9966, 9953]
Average time to sort the array: 0.002001 seconds
Average time to sort the sorted array in ascending order: 0.000118 seconds
Average time to sort the descending array: 0.003278 seconds

Input size: 2000
Unsorted array (first 10 elements): [5026, 3074, 9871, 3999, 3342, 7525, 3841, 7725, 3462, 5335]
Sorted array (first 10 elements): [4, 5, 11, 19, 21, 23, 35, 37, 46, 47]
Descending array (first 10 elements): [9999, 9994, 9990, 9984, 9959, 9950, 9941, 9937, 9933, 9927]
Average time to sort the array: 0.002606 seconds
Average time to sort the sorted array in ascending order: 0.000233 seconds
Average time to sort the descending array: 0.005476 seconds

Input size: 2500
Unsorted array (first 10 elements): [7760, 8525, 316, 379, 4037, 1659, 6508, 6311, 5522, 3590]
Sorted array (first 10 elements): [5, 10, 13, 16, 20, 21, 22, 22, 24, 28]
Descending array (first 10 elements): [9995, 9995, 9995, 9994, 9994, 9993, 9988, 9981, 9980, 9979]
Average time to sort the array: 0.005712 seconds
Average time to sort the sorted array in ascending order: 0.000290 seconds
Average time to sort the descending array: 0.008759 seconds

PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> 
```



Analysis:

We know that the time complexity for best case in insertion sort is O(n) and for average and worst case is O(n^2). 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:

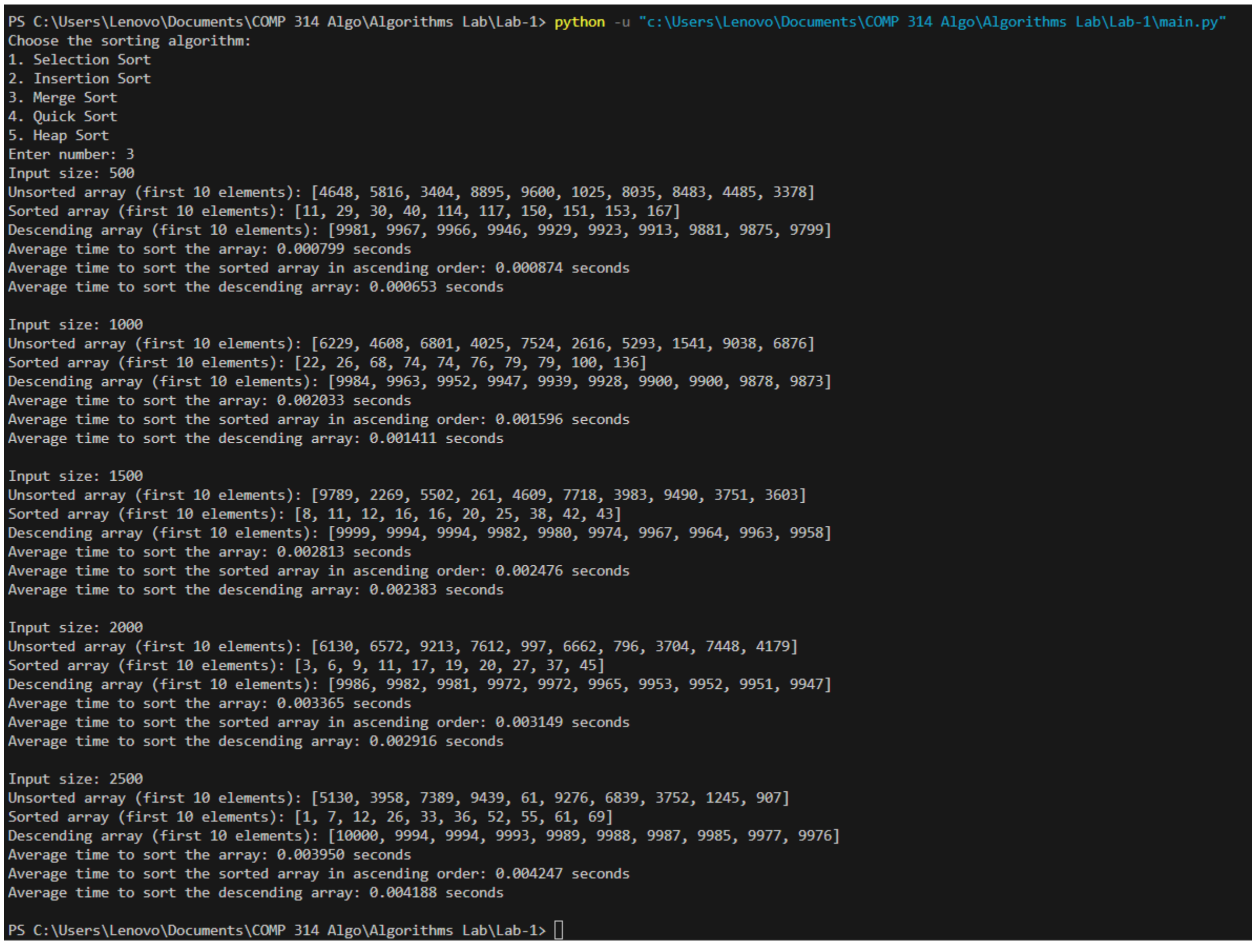
Merge sort uses divide-and-conquer strategy. The original problem is partitioned into simpler sub-problems, each subproblem is considered independently. Subdivision continues until sub problems obtained are simple.

Steps:

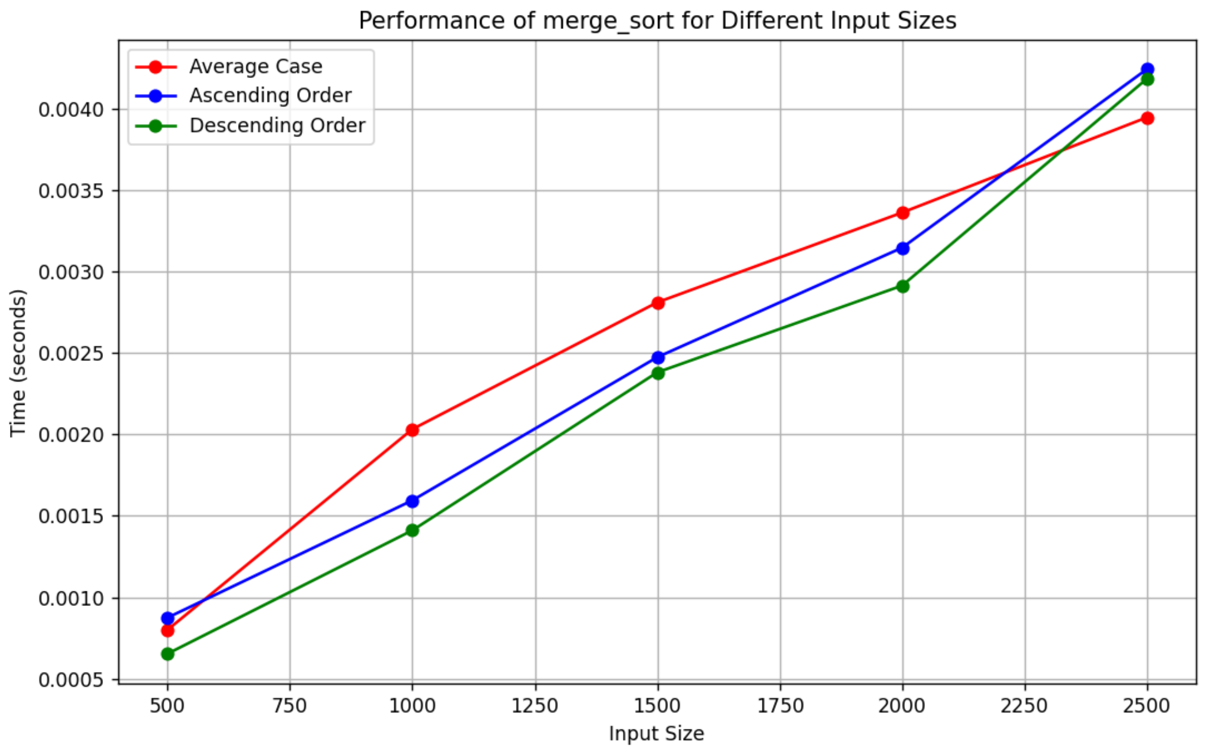
1. Divide: partition the list into two roughly equal parts, S1 and S2, called the left and the right sublists

2. Conquer: recursively sort S1 and S2
3. Combine: merge the sorted sublists

Output:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> python -u "c:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1\main.py"
Choose the sorting algorithm:
1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort
Enter number: 3
Input size: 500
Unsorted array (first 10 elements): [4648, 5816, 3404, 8895, 9600, 1025, 8035, 8483, 4485, 3378]
Sorted array (first 10 elements): [11, 29, 30, 40, 114, 117, 150, 151, 153, 167]
Descending array (first 10 elements): [9981, 9967, 9966, 9946, 9929, 9923, 9913, 9881, 9875, 9799]
Average time to sort the array: 0.000799 seconds
Average time to sort the sorted array in ascending order: 0.000874 seconds
Average time to sort the descending array: 0.000653 seconds

Input size: 1000
Unsorted array (first 10 elements): [6229, 4608, 6801, 4025, 7524, 2616, 5293, 1541, 9038, 6876]
Sorted array (first 10 elements): [22, 26, 68, 74, 74, 76, 79, 79, 100, 136]
Descending array (first 10 elements): [9984, 9963, 9952, 9947, 9939, 9928, 9900, 9900, 9878, 9873]
Average time to sort the array: 0.002033 seconds
Average time to sort the sorted array in ascending order: 0.001596 seconds
Average time to sort the descending array: 0.001411 seconds

Input size: 1500
Unsorted array (first 10 elements): [9789, 2269, 5502, 261, 4609, 7718, 3983, 9490, 3751, 3603]
Sorted array (first 10 elements): [8, 11, 12, 16, 16, 20, 25, 38, 42, 43]
Descending array (first 10 elements): [9999, 9994, 9994, 9982, 9980, 9974, 9967, 9964, 9963, 9958]
Average time to sort the array: 0.002813 seconds
Average time to sort the sorted array in ascending order: 0.002476 seconds
Average time to sort the descending array: 0.002383 seconds

Input size: 2000
Unsorted array (first 10 elements): [6130, 6572, 9213, 7612, 997, 6662, 796, 3704, 7448, 4179]
Sorted array (first 10 elements): [3, 6, 9, 11, 17, 19, 20, 27, 37, 45]
Descending array (first 10 elements): [9986, 9982, 9981, 9972, 9972, 9965, 9953, 9952, 9951, 9947]
Average time to sort the array: 0.003365 seconds
Average time to sort the sorted array in ascending order: 0.003149 seconds
Average time to sort the descending array: 0.002916 seconds

Input size: 2500
Unsorted array (first 10 elements): [5130, 3958, 7389, 9439, 61, 9276, 6839, 3752, 1245, 907]
Sorted array (first 10 elements): [1, 7, 12, 26, 33, 36, 52, 55, 61, 69]
Descending array (first 10 elements): [10000, 9994, 9994, 9993, 9989, 9988, 9987, 9985, 9977, 9976]
Average time to sort the array: 0.003950 seconds
Average time to sort the sorted array in ascending order: 0.004247 seconds
Average time to sort the descending array: 0.004188 seconds

PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1>
```



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:

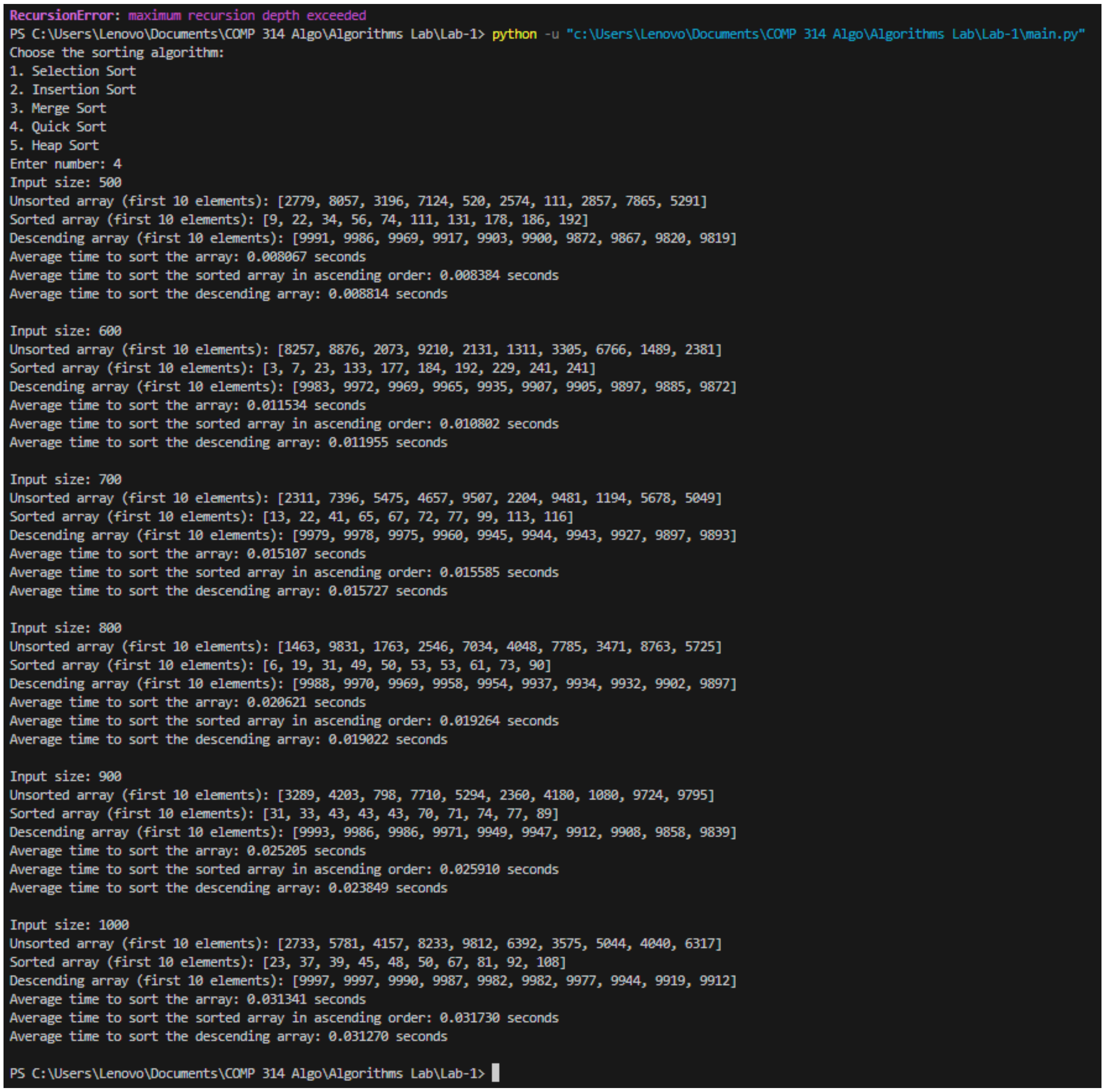
Like merge sort, quick sort also uses divide-and-conquer paradigm.

Steps:

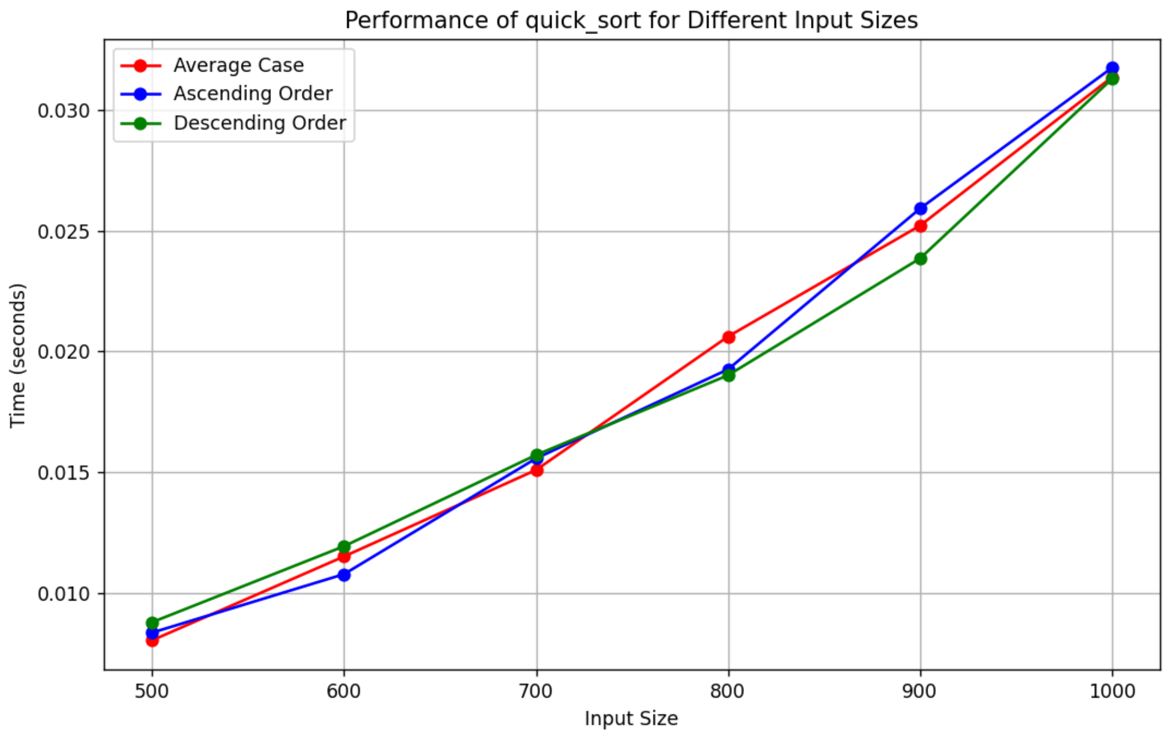
1. Divide: Select any element from the list. Call it the pivot. Then partition the list into two sublists such that all the elements in the left sublist are less than or equal to the pivot and those of the right sublist are greater than or equal to the pivot
2. Conquer: Recursively sort the two sublists

3. Combine: Since the subarrays are sorted in place, no work is needed to combine them: the entire list is now sorted.

Output:



```
RecursionError: maximum recursion depth exceeded
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> python -u "c:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1\main.py"
Choose the sorting algorithm:
1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort
Enter number: 4
Input size: 500
Unsorted array (first 10 elements): [2779, 8057, 3196, 7124, 520, 2574, 111, 2857, 7865, 5291]
Sorted array (first 10 elements): [9, 22, 34, 56, 74, 111, 131, 178, 186, 192]
Descending array (first 10 elements): [9991, 9986, 9969, 9917, 9903, 9900, 9872, 9867, 9820, 9819]
Average time to sort the array: 0.008067 seconds
Average time to sort the sorted array in ascending order: 0.008384 seconds
Average time to sort the descending array: 0.008814 seconds

Input size: 600
Unsorted array (first 10 elements): [8257, 8876, 2073, 9210, 2131, 1311, 3305, 6766, 1489, 2381]
Sorted array (first 10 elements): [3, 7, 23, 133, 177, 184, 192, 229, 241, 241]
Descending array (first 10 elements): [9983, 9972, 9969, 9965, 9935, 9907, 9905, 9897, 9885, 9872]
Average time to sort the array: 0.011534 seconds
Average time to sort the sorted array in ascending order: 0.010802 seconds
Average time to sort the descending array: 0.011955 seconds

Input size: 700
Unsorted array (first 10 elements): [2311, 7396, 5475, 4657, 9507, 2204, 9481, 1194, 5678, 5049]
Sorted array (first 10 elements): [13, 22, 41, 65, 67, 72, 77, 99, 113, 116]
Descending array (first 10 elements): [9979, 9978, 9975, 9960, 9945, 9944, 9943, 9927, 9897, 9893]
Average time to sort the array: 0.015107 seconds
Average time to sort the sorted array in ascending order: 0.015585 seconds
Average time to sort the descending array: 0.015727 seconds

Input size: 800
Unsorted array (first 10 elements): [1463, 9831, 1763, 2546, 7034, 4048, 7785, 3471, 8763, 5725]
Sorted array (first 10 elements): [6, 19, 31, 49, 50, 53, 53, 61, 73, 90]
Descending array (first 10 elements): [9988, 9970, 9969, 9958, 9954, 9937, 9934, 9932, 9902, 9897]
Average time to sort the array: 0.020621 seconds
Average time to sort the sorted array in ascending order: 0.019264 seconds
Average time to sort the descending array: 0.019022 seconds

Input size: 900
Unsorted array (first 10 elements): [3289, 4203, 798, 7710, 5294, 2360, 4180, 1080, 9724, 9795]
Sorted array (first 10 elements): [31, 33, 43, 43, 43, 70, 71, 74, 77, 89]
Descending array (first 10 elements): [9993, 9986, 9986, 9971, 9949, 9947, 9912, 9908, 9858, 9839]
Average time to sort the array: 0.025205 seconds
Average time to sort the sorted array in ascending order: 0.025910 seconds
Average time to sort the descending array: 0.023849 seconds

Input size: 1000
Unsorted array (first 10 elements): [2733, 5781, 4157, 8233, 9812, 6392, 3575, 5044, 4040, 6317]
Sorted array (first 10 elements): [23, 37, 39, 45, 48, 50, 67, 81, 92, 108]
Descending array (first 10 elements): [9997, 9997, 9990, 9987, 9982, 9982, 9977, 9944, 9919, 9912]
Average time to sort the array: 0.031341 seconds
Average time to sort the sorted array in ascending order: 0.031730 seconds
Average time to sort the descending array: 0.031270 seconds

PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1>
```



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 $O(n^2)$. Best case, $\Omega(n\log n)$, occurs when the pivot element divides the array into two equal halves. Average case, $\theta(n\log n)$ is obtained when the pivot divides the array into two parts, but not necessarily equal.

Heap sort:

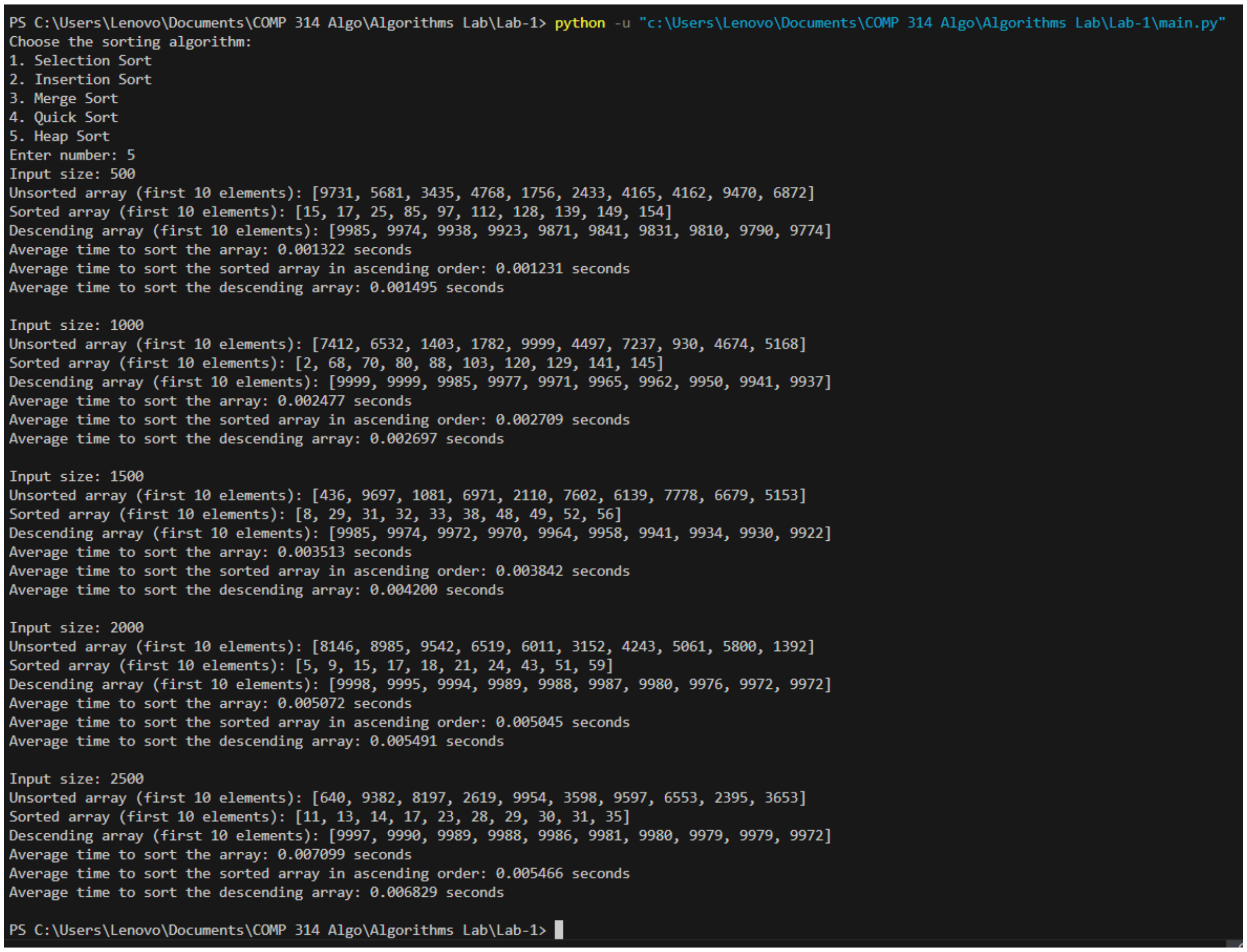
To implement the heap sort using a max-heap, we need two basic algorithms:

- 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.

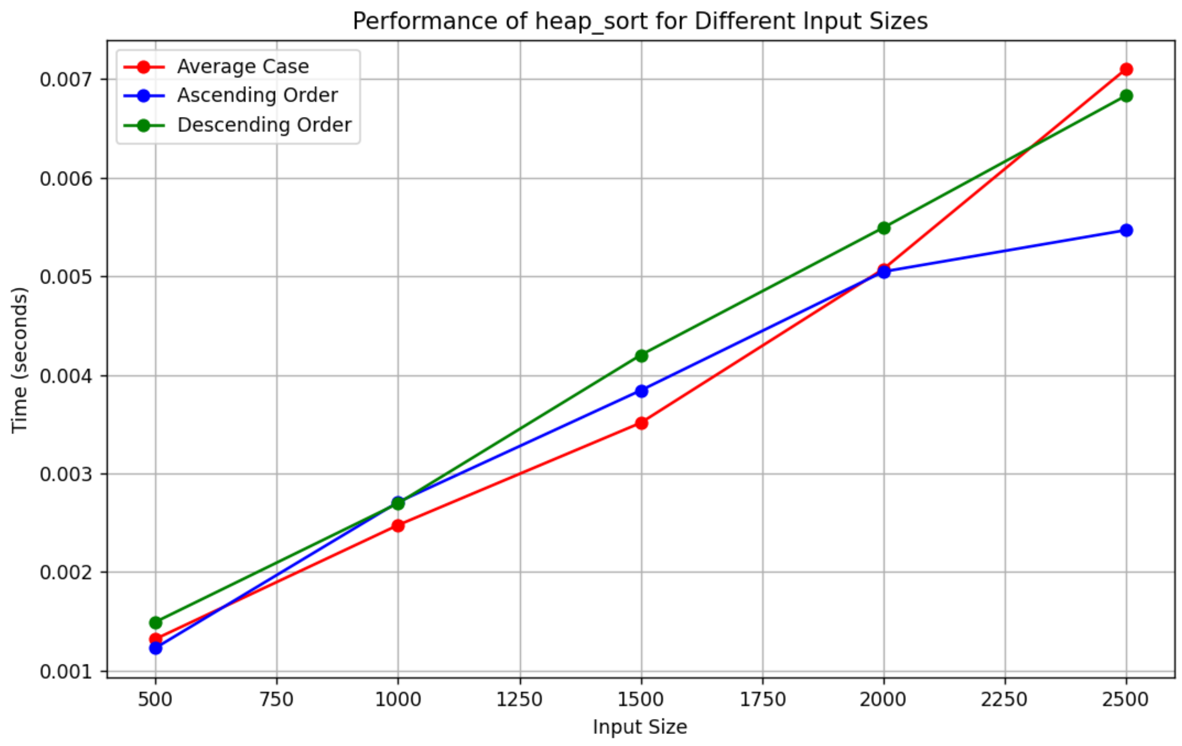
Steps:

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:



```
PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1> python -u "c:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1\main.py"
Choose the sorting algorithm:
1. Selection Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort
Enter number: 5
Input size: 500
Unsorted array (first 10 elements): [9731, 5681, 3435, 4768, 1756, 2433, 4165, 4162, 9470, 6872]
Sorted array (first 10 elements): [15, 17, 25, 85, 97, 112, 128, 139, 149, 154]
Descending array (first 10 elements): [9985, 9974, 9938, 9923, 9871, 9841, 9831, 9810, 9790, 9774]
Average time to sort the array: 0.001322 seconds
Average time to sort the sorted array in ascending order: 0.001231 seconds
Average time to sort the descending array: 0.001495 seconds

Input size: 1000
Unsorted array (first 10 elements): [7412, 6532, 1403, 1782, 9999, 4497, 7237, 930, 4674, 5168]
Sorted array (first 10 elements): [2, 68, 70, 80, 88, 103, 120, 129, 141, 145]
Descending array (first 10 elements): [9999, 9999, 9985, 9977, 9971, 9965, 9962, 9950, 9941, 9937]
Average time to sort the array: 0.002477 seconds
Average time to sort the sorted array in ascending order: 0.002709 seconds
Average time to sort the descending array: 0.002697 seconds

Input size: 1500
Unsorted array (first 10 elements): [436, 9697, 1081, 6971, 2110, 7602, 6139, 7778, 6679, 5153]
Sorted array (first 10 elements): [8, 29, 31, 32, 33, 38, 48, 49, 52, 56]
Descending array (first 10 elements): [9985, 9974, 9972, 9970, 9964, 9958, 9941, 9934, 9930, 9922]
Average time to sort the array: 0.003513 seconds
Average time to sort the sorted array in ascending order: 0.003842 seconds
Average time to sort the descending array: 0.004200 seconds

Input size: 2000
Unsorted array (first 10 elements): [8146, 8985, 9542, 6519, 6011, 3152, 4243, 5061, 5800, 1392]
Sorted array (first 10 elements): [5, 9, 15, 17, 18, 21, 24, 43, 51, 59]
Descending array (first 10 elements): [9998, 9995, 9994, 9989, 9988, 9987, 9980, 9976, 9972, 9972]
Average time to sort the array: 0.005072 seconds
Average time to sort the sorted array in ascending order: 0.005045 seconds
Average time to sort the descending array: 0.005491 seconds

Input size: 2500
Unsorted array (first 10 elements): [640, 9382, 8197, 2619, 9954, 3598, 9597, 6553, 2395, 3653]
Sorted array (first 10 elements): [11, 13, 14, 17, 23, 28, 29, 30, 31, 35]
Descending array (first 10 elements): [9997, 9990, 9989, 9988, 9986, 9981, 9980, 9979, 9979, 9972]
Average time to sort the array: 0.007099 seconds
Average time to sort the sorted array in ascending order: 0.005466 seconds
Average time to sort the descending array: 0.006829 seconds

PS C:\Users\Lenovo\Documents\COMP 314 Algo\Algorithms Lab\Lab-1>
```



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(n^2)	O(n^2)	O(n^2)
Insertion	O(n)	O(n^2)	O(n^2)
Merge	O(nlogn)	O(nlogn)	O(nlogn)
Quick	O(nlogn)	O(nlogn)	O(n^2)
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 $O(n^2)$. 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.