Course Name: Fundamentals of Programming & Data Science	**Course Code:** CMPE-112L
Assignment Type: Lab	**Dated:** 8th April 2024
Semester: 2nd	**Session:** 2023
Lab/Project/Assignment #: 9	**CLOs to be covered: CLO 3**
Lab Title: Searching and Sorting Algorithms	**Teacher Name: Engr.** Afeef Obaid

Lab Evaluation:

CLO 3	Adhere to plagiarism guidelines					
Levels (Marks)	**Level 1**	**Level 2**	**Level 3**	**Level 4**	**Level 5**	**Level 6**
(10)						
					Total	**/10**

Rubrics for Current Lab Evaluation

Scale	**Marks**	**Level**	**Rubric**
Excellent	**10**	L1	Submitted all lab tasks, BONUS task, have good understanding.
Very Good	**8**	L2	Submitted the lab tasks but have good understanding
Good	**6**	L3	Submitted the lab tasks but have weak understanding.
Basic	**4**	L4	Submitted the lab tasks but have no understanding.
Barely Acceptable	**2**	L5	Submitted only one lab task.
Not Acceptable	**0**	L6	Did not attempt

LAB No. 9

Lab Goals/Objectives:

By reading this manual, students will be able:

- To Become familiar with the Searching and Sorting algorithms in Python
- To learn about Bubble sort algorithm in Python
- To learn about Selection sort algorithm in Python
- To learn about Insertion sort algorithm in Python

Equipment Required: Computer system with Pycharm IDE and python package installed on it

Sorting Algorithms

A sorting algorithm is a method for reorganizing a large number of items into a specific order, such as alphabetical, highest-to-lowest value or shortest-to-longest distance. Sorting algorithms take lists of items as input data, perform specific operations on those lists and deliver ordered arrays as output. The many applications of sorting algorithms include organizing items by price on a retail website and determining the order of sites on a search engine results page.

There are various sorting algorithms that can be used to complete this operation. And, we can use any algorithm based on the requirement.

- Bubble Sort
- Insertion Sort
- Selection Sort

Bubble Sort

Bubble sort is a simple sorting algorithm that repeatedly steps through a list of elements, compares adjacent elements, and swaps them if they are in the wrong order. This process is repeated until the entire list is sorted. Let's dive into a detailed explanation of bubble sort along with a visualization.

Suppose we have an unsorted list of numbers: [5, 3, 8, 4, 2]. The goal is to sort this list in ascending order using the bubble sort algorithm.

Step 1: Starting from the beginning of the list, we compare the first element (5) with the second element (3). Since 5 is greater than 3, we swap them. The list now becomes [3, 5, 8, 4, 2].

Step 2: We move to the next pair of elements, which are 5 and 8. They are already in the correct order, so no swapping is needed. The list remains [3, 5, 8, 4, 2].

Step 3: Continuing this process, we compare 8 and 4. Since 8 is greater than 4, we swap them. The list becomes [3, 5, 4, 8, 2].

Step 4: We compare 8 and 2. Again, 8 is greater than 2, so we swap them. The list is now [3, 5, 4, 2, 8].

At this point, we have completed the first pass through the list, and the largest element (8) has moved to its correct position at the end of the list. Now, we need to repeat the same process for the remaining elements.

Step 5: We start the second pass by comparing 3 and 5. They are already in order, so we leave them as is. The list remains [3, 5, 4, 2, 8].

Step 6: We compare 5 and 4. Since 5 is greater than 4, we swap them. The list becomes [3, 4, 5, 2, 8].

Step 7: We compare 5 and 2. Again, 5 is greater than 2, so we swap them. The list is now [3, 4, 2, 5, 8].

Step 8: We compare 5 and 8. They are already in order, so we move on. The list remains [3, 4, 2, 5, 8].

After the second pass, the second largest element (5) has moved to its correct position. We continue this process for the remaining passes until the entire list is sorted.

Step 9: We compare 3 and 4. They are already in order, so we leave them as is. The list remains [3, 4, 2, 5, 8].

Step 10: We compare 4 and 2. Since 4 is greater than 2, we swap them. The list becomes [3, 2, 4, 5, 8].

Step 11: We compare 4 and 5. They are already in order, so we move on. The list remains [3, 2, 4, 5, 8].

Step 12: We compare 5 and 8. They are already in order, so we leave them as is. The list remains [3, 2, 4, 5, 8].

At this point, we have completed the third pass, and the third largest element (4) has moved to its correct position.

Step 13: We compare 3 and 2. Since 3 is greater than 2, we swap them. The list becomes [2, 3, 4, 5, 8].

Step 14: We compare 3 and 4. They are already in order, so we move on. The list remains [2, 3, 4, 5, 8].

Step 15: We compare 4 and 5. They are already in order, so we leave them as is. The list remains [2, 3, 4, 5, 8].

Step 16: We compare 5 and 8. They are already in order, so we move on. The list remains [2, 3, 4, 5, 8].

After the fourth pass, the fourth largest element (3) has moved to its correct position.

Step 17: We compare 2 and 3. They are already in order, so we leave them as is. The list remains [2, 3, 4, 5, 8].

Step 18: We compare 3 and 4. They are already in order, so we move on. The list remains [2, 3, 4, 5, 8].

Step 19: We compare 4 and 5. They are already in order, so we move on. The list remains [2, 3, 4, 5, 8].

Step 20: We compare 5 and 8. They are already in order, so we move on. The list remains [2, 3, 4, 5, 8].

After the fifth and final pass, the smallest element (2) has moved to its correct position. Now, the entire list is sorted.

The bubble sort algorithm works by repeatedly comparing adjacent elements and swapping them if they are in the wrong order. This process continues until the entire list is sorted. The name "bubble sort" comes from the way smaller elements gradually "bubble" their way to the beginning of the list.

Visualization:

Initially, we have the list: [5, 3, 8, 4, 2]

Pass 1:

[3, 5, 8, 4, 2]

[3, 5, 4, 8, 2]

[3, 5, 4, 2, 8]

Pass 2:

[3, 4, 5, 2, 8]

[3, 4, 2, 5, 8]

Pass 3:

[3, 2, 4, 5, 8]

Pass 4:

[2, 3, 4, 5, 8]

Pass 5:

[2, 3, 4, 5, 8]

The final sorted list is: [2, 3, 4, 5, 8]

In this visualization, each line represents a pass, and the elements being compared are highlighted. As the algorithm progresses, you can see the elements gradually being sorted and moving into their correct positions.

Bubble sort is a straightforward sorting algorithm, but it is not very efficient for large lists since it has an average and worst-case time complexity of O(n^2). However, it is often used in educational settings or for small lists due to its simplicity.

Selection Sort

Selection sort is a simple sorting algorithm that works by repeatedly finding the minimum element from the unsorted part of the list and placing it at the beginning. This process is repeated until the entire list is sorted. Let's delve into a detailed explanation of selection sort along with a visualization.

Suppose we have an unsorted list of numbers: [5, 3, 8, 4, 2]. The goal is to sort this list in ascending order using the selection sort algorithm.

Step 1: Initially, the entire list is unsorted. We start by finding the minimum element from the unsorted part of the list. In this case, the minimum element is 2. We swap it with the first element of the list. The list now becomes [2, 3, 8, 4, 5]

Step 2: The first element is now sorted, so we move to the next position and repeat the process. The minimum element from the remaining unsorted part is 3. We swap it with the second element. The list becomes [2, 3, 8, 4, 5]

Step 3: The first two elements are now sorted. We find the minimum element from the remaining unsorted part, which is 4. We swap it with the third element. The list becomes [2, 3, 4, 8, 5]

Step 4: We continue this process, finding the minimum element from the remaining unsorted part and placing it in the next position.

Step 5: The minimum element from the remaining unsorted part is 5. We swap it with the fourth element. The list becomes [2, 3, 4, 5, 8]

Finally, we have only one element left, which is already sorted.

After these steps, the entire list is sorted in ascending order: [2, 3, 4, 5, 8].

Visualization:

Initially, we have the list: [5, 3, 8, 4, 2]

Pass 1: We find the minimum element (2) and swap it with the first element.

[2, 3, 8, 4, 5]

Pass 2: We find the minimum element (3) from the remaining unsorted part and swap it with the second element.

[2, 3, 8, 4, 5]

Pass 3: We find the minimum element (4) from the remaining unsorted part and swap it with the third element.

[2, 3, 4, 8, 5]

Pass 4: We find the minimum element (5) from the remaining unsorted part and swap it with the fourth element.

[2, 3, 4, 5, 8]

Pass 5: The list is fully sorted.

In this visualization, each pass is represented by a step, and the minimum element being found and swapped is highlighted. As the algorithm progresses, you can see the minimum elements gradually moving to the beginning of the list, resulting in a sorted list.

Selection sort works by dividing the list into two parts: the sorted part and the unsorted part. In each pass, it finds the minimum element from the unsorted part and places it in the correct position in the sorted part. The algorithm continues this process until the entire list is sorted.

Selection sort has a time complexity of O(n^2) in all cases, making it less efficient than more advanced sorting algorithms for large lists. However, it has the advantage of simplicity and can be useful for small lists or as a building block in more complex sorting algorithms.

Insertion Sort

Insertion sort is a simple sorting algorithm that builds the final sorted list one element at a time. It works by iteratively inserting each unsorted element into its correct position within the sorted part of the list. Let's delve into a detailed explanation of insertion sort along with a visualization.

Suppose we have an unsorted list of numbers: [5, 3, 8, 4, 2]. The goal is to sort this list in ascending order using the insertion sort algorithm.

Step 1: Initially, the first element (5) is considered as the sorted part of the list. We then take the next element (3) and compare it to the elements in the sorted part. Since 3 is smaller than 5, we shift 5 to the right, making space for 3 to be inserted. The list becomes [3, 5, 8, 4, 2].

Step 2: The first two elements (3 and 5) are now sorted. We take the next element (8) and compare it with the elements in the sorted part. Since 8 is greater than 5, it remains in its position. The list remains [3, 5, 8, 4, 2].

Step 3: We take the next element (4) and compare it with the elements in the sorted part. 4 is smaller than 8, so we shift 8 to the right and 5 to the right, making space for 4 to be inserted. The list becomes [3, 4, 5, 8, 2].

Step 4: We take the next element (2) and compare it with the elements in the sorted part. 2 is smaller than 8, so we shift 8 to the right, 5 to the right, and 4 to the right, making space for 2 to be inserted. The list becomes [2, 3, 4, 5, 8].

At this point, we have completed the sorting process, and the entire list is sorted in ascending order: [2, 3, 4, 5, 8].

Visualization:

Initially, we have the list: [5, 3, 8, 4, 2]

Pass 1: Insert 3 into its correct position in the sorted part.

[3, 5, 8, 4, 2]

Pass 2: 8 is already in the correct position in the sorted part.

[3, 5, 8, 4, 2]

Pass 3: Insert 4 into its correct position in the sorted part.

[3, 4, 5, 8, 2]

Pass 4: Insert 2 into its correct position in the sorted part.

[2, 3, 4, 5, 8]

In this visualization, each pass is represented by a step, and the element being inserted is highlighted. As the algorithm progresses, you can see the sorted part gradually expanding and the unsorted elements being inserted into their correct positions.

Insertion sort works by dividing the list into two parts: the sorted part and the unsorted part. In each iteration, it takes an element from the unsorted part and inserts it into its correct position within the sorted part. The algorithm continues this process until the entire list is sorted.

Insertion sort has a time complexity of O(n^2) in the worst case, making it less efficient than more advanced sorting algorithms for large lists. However, it is efficient for small lists or partially sorted lists. Additionally, insertion sort has the advantage of being an in-place sorting algorithm, meaning it does not require extra memory space.

Linear or Sequential Search algorithm

Linear search, also known as sequential search, is a basic searching algorithm used to find a specific element in a list. It is called "linear" because it checks each element in the list sequentially until the target element is found or the end of the list is reached. While linear search is simple to understand and implement, it may not be the most efficient algorithm for large lists. However, it is still useful in situations where the list is small or unsorted.

Analysis:

In linear search algorithm the time taken to perform the search increases linearly with the size of the list. In the worst-case scenario, where the target element is not present in the list or is located at the last position, the algorithm will iterate through all the elements.

Advantages and Disadvantages:

Linear search has the following advantages:

Simplicity: The algorithm is straightforward to understand and implement.

Works on Unsorted Lists: Linear search can be applied to both sorted and unsorted lists.

However, it also has some disadvantages:

Efficiency: For large lists, linear search is not efficient compared to more advanced searching algorithms like binary search or hash-based searches.

Conclusion:

Linear search is a simple and easy-to-understand algorithm for searching elements in a list. It is effective for small or unsorted lists. However, for larger lists or situations where efficiency is crucial, other algorithms that offer better time complexity should be considered.

Binary Search algorithm

Binary search is a powerful searching algorithm used to find a specific element in a sorted list. It follows a divide-and-conquer approach by repeatedly dividing the search space in half until the target element is found or it is determined that the element does not exist in the list. Binary search is highly efficient and can drastically reduce the search time compared to linear search, especially for large sorted lists.

Analysis:

In binary search algorithm the search time grows logarithmically with the size of the list. In each iteration, the search space is divided by 2, leading to a significantly faster search compared to linear search.

Advantages and Disadvantages:

Binary search has the following advantages:

Efficiency: Binary search is highly efficient, especially for large sorted lists, as it reduces the search space in half with each iteration.

However, binary search also has some disadvantages as well:

Requirement of Sorted List: Binary search requires the list to be sorted beforehand. If the list is unsorted, it needs to be sorted first, which can be an additional time cost.

Limited Applicability: Binary search is not suitable for dynamically changing lists or situations where frequent insertions or deletions occur.

Conclusion:

Binary search is a powerful algorithm for searching elements in a sorted list. It offers a significant improvement in efficiency over linear search, especially for large lists. However, it requires the list to be sorted, and its applicability is limited in certain scenarios. Overall, binary search is an excellent choice when working with large sorted lists and can provide a significant performance boost.

Lab Tasks

- Write the python program for **Bubble Sort algorithm**. Also print the list in each iteration and total number of swaps which are done for sorting the list.
- Write the python program for **Selection Sort algorithm**. Also print the list in each iteration and total number of swaps which are done for sorting the list.
- Write the python program for **Insertion Sort algorithm**. Also print the list in each iteration and total number of swaps which are done for sorting the list.
- Write the python program for **Linear Search algorithm**. Also print the index of the element if it found during searching.
- Write the python program for **Binary Search algorithm**. Also print the index of the element if it found and total number of searches done.