Insertion Sort

Aim: The aim of this lab practical is to implement and analyze the efficiency of the Insertion Sort algorithm in sorting a collection of elements in ascending order.

Theory:

Insertion Sort is a simple and intuitive sorting algorithm that builds the final sorted array one element at a time. It is much less efficient on large lists than more advanced algorithms such as quicksort, heapsort, or merge sort. However, insertion sort has some advantages, such as simplicity and low memory usage, which make it useful for small datasets or partially sorted datasets.

Pseudocode:

```
for i = 1 to n-1:
   key = array[i]
   j = i - 1
   while j >= 0 and array[j] > key:
      array[j + 1] = array[j]
      j = j - 1
   array[j + 1] = key
```

Code:

```
#include <stdio.h>

void insertionSort(int arr[], int n) {
   for (int i = 1; i < n; i++) {
      int currentElement = arr[i];
      int j = i - 1;

      while (j >= 0 && currentElement < arr[j]) {
         arr[j + 1] = arr[j];
         j--;
      }

      arr[j + 1] = currentElement;
   }
}

int main() {
   int myArray[] = {12, 11, 13, 5, 6};

   int size = sizeof(myArray) / sizeof(myArray[0]);

   insertionSort(myArray, size);

   printf("Sorted array: ");
   for (int i = 0; i < size; i++) {
      printf("%d ", myArray[i]);
   }
   printf("\n");
```

```
    return 0;
}
```

Output:

```
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes$ gcc insertion_sort.c -o insertion_sort && ./insertion_sort
Sorted array: 5 6 11 12 13
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes$
```

Algorithm Analysis:

The Insertion Sort algorithm is a simple sorting algorithm that builds the final sorted array one element at a time. It is based on the idea of dividing the array into two portions: a sorted portion and an unsorted portion. The algorithm iterates through the unsorted portion, taking one element at a time and inserting it into its correct position in the sorted portion.

Algorithm Steps:

1. Initialization:
   - The algorithm begins by considering the first element of the array as a sorted sequence of one element.
   - The rest of the array is considered as the unsorted portion.

2. Iteration through Unsorted Portion:
   - Starting from the second element (index 1), the algorithm iterates through the unsorted portion of the array.
   - For each element in the unsorted portion, the algorithm compares it with the elements in the sorted portion to find its correct position.

3. Insertion:
   - The current element (let's call it the "key") is compared with the elements in the sorted portion.
   - The algorithm iterates backward through the sorted portion, shifting elements to the right until it finds the correct position for the key.
   - The key is then inserted into its correct position in the sorted portion.

4. Repeat:
- Steps 2 and 3 are repeated until the entire array is sorted.

Example:

Original Array: [5, 2, 4, 6, 1, 3]

Iteration 1: [2, 5, 4, 6, 1, 3]
Iteration 2: [2, 4, 5, 6, 1, 3]
Iteration 3: [2, 4, 5, 6, 1, 3]
Iteration 4: [1, 2, 4, 5, 6, 3]
Iteration 5: [1, 2, 3, 4, 5, 6]

Time Complexity:

Best Case:

- The best-case scenario occurs when the array is already sorted.

In this case, the inner while loop doesn't need to execute, and the time complexity is linear.

- Time Complexity: O(n)

Average Case:

- In the average case, each element in the unsorted portion may need to be compared and shifted.
- The nested while loop has a linear dependence on the input size.
- Time Complexity: O(n^2)

Worst Case:

- The worst-case scenario happens when the array is in reverse order.
- For each element in the unsorted portion, it has to be compared and shifted in the sorted portion.
- The nested while loop contributes to the quadratic time complexity.
- Time Complexity: O(n^2)

Space Complexity:

- Insertion Sort is an in-place sorting algorithm, meaning it sorts the array without requiring additional memory for a new sorted array.
- The space complexity is constant, as it only requires a small, fixed amount of additional memory for variables like the key and the index used in the loop.
- Space Complexity: O(1)

Selection Sort

Aim: The aim of this lab practical is to implement and analyze the efficiency of the Selection Sort algorithm in sorting a collection of elements in ascending order.

Theory:

Selection Sort is a simple and intuitive comparison-based sorting algorithm that divides the input array into two parts: a sorted portion and an unsorted portion. Similar to Insertion Sort, Selection Sort builds the final sorted array by iteratively selecting the smallest (or largest) element from the unsorted portion and swapping it with the first unsorted element. This process is repeated until the entire array is sorted.

Pseudocode:

```
for i = 0 to n-1:
   min_index = i
   for j = i+1 to n:
      if array[j] < array[min_index]:
         min_index = j
   swap(array[i], array[min_index])
```

Code:

```
#include <stdio.h>

void selectionSort(int arr[], int n)
{
   for (int i = 0; i < n - 1; i++)
   {
      int minIndex = i;

      for (int j = i + 1; j < n; j++)
      {
         if (arr[j] < arr[minIndex])
         {
            minIndex = j;
         }
      }

      int temp = arr[i];
      arr[i] = arr[minIndex];
      arr[minIndex] = temp;
   }
}

int main()
{
   int myArray[] = {64, 25, 12, 22, 11};

   int size = sizeof(myArray) / sizeof(myArray[0]);
```

```
    selectionSort(myArray, size);

    printf("Sorted array: ");
    for (int i = 0; i < size; i++)
    {
        printf("%d ", myArray[i]);
    }
    printf("\n");

    return 0;
}
```

Output:

```
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes$ gcc selection_sort.c -o selection_sort && ./selection_sort
Sorted array: 11 12 22 25 64
go-d-code@code-valley:~/Roger/College/DAA_Lab_Codes$
```

Algorithm Analysis:

Selection Sort is a simple comparison-based sorting algorithm that divides the input array into two portions: a sorted portion and an unsorted portion. The algorithm repeatedly selects the smallest (or largest, depending on the desired order) element from the unsorted portion and swaps it with the first unsorted element. This process is iteratively carried out until the entire array is sorted.

Algorithm Steps:

1. Initialization:
   - The algorithm starts with the entire array considered as the unsorted portion.
   - The first element of the unsorted portion is assumed to be the minimum (or maximum) element.

2. Iteration through Unsorted Portion:
   - The algorithm iterates through the unsorted portion to find the minimum (or maximum) element.
   - For each iteration, it compares the current element with the assumed minimum (or maximum) element.

3. Selection:
   - The minimum (or maximum) element found during the iteration is swapped with the first element of the unsorted portion.
   - This effectively expands the sorted portion and reduces the unsorted portion.

4. Repeat:
   - Steps 2 and 3 are repeated until the entire array is sorted.
   - In each iteration, the sorted portion grows, and the unsorted portion shrinks.

Pseudocode:

```
for i = 0 to n-1:
    min_index = i
    for j = i+1 to n:
        if array[j] < array[min_index]:
            min_index = j
    swap(array[i], array[min_index])
```

- Here, `n` is the number of elements in the array.

Example:

Original Array: [5, 2, 4, 6, 1, 3]

Iteration 1: [1, 2, 4, 6, 5, 3]
Iteration 2: [1, 2, 3, 6, 5, 4]
Iteration 3: [1, 2, 3, 4, 5, 6]

Time Complexity:

Best Case:

- The best-case scenario for Selection Sort occurs when the array is already sorted.
- Even in the best case, Selection Sort performs the same number of comparisons and swaps as in the average and worst cases.
- Time Complexity: O(n^2)

Average Case:

- In the average case, Selection Sort performs a quadratic number of comparisons and swaps.
- The nested loops contribute to the overall quadratic time complexity.
- Time Complexity: O(n^2)

Worst Case:

- The worst-case scenario for Selection Sort is when the array is in reverse order.
- Similar to the average case, the nested loops contribute to the quadratic time complexity.
- Time Complexity: O(n^2)

Space Complexity:

- Selection Sort is an in-place sorting algorithm, meaning it sorts the array without requiring additional memory for a new sorted array.
- The space complexity is constant, as it only requires a small, fixed amount of additional memory for variables like the minimum index.
- Space Complexity: O(1)