1. Insertion Sort :

Theory:

- Insertion Sort works similarly to how we sort playing cards in our hands. It builds the sorted array one element at a time by picking an element and inserting it into its correct position among the already sorted elements.

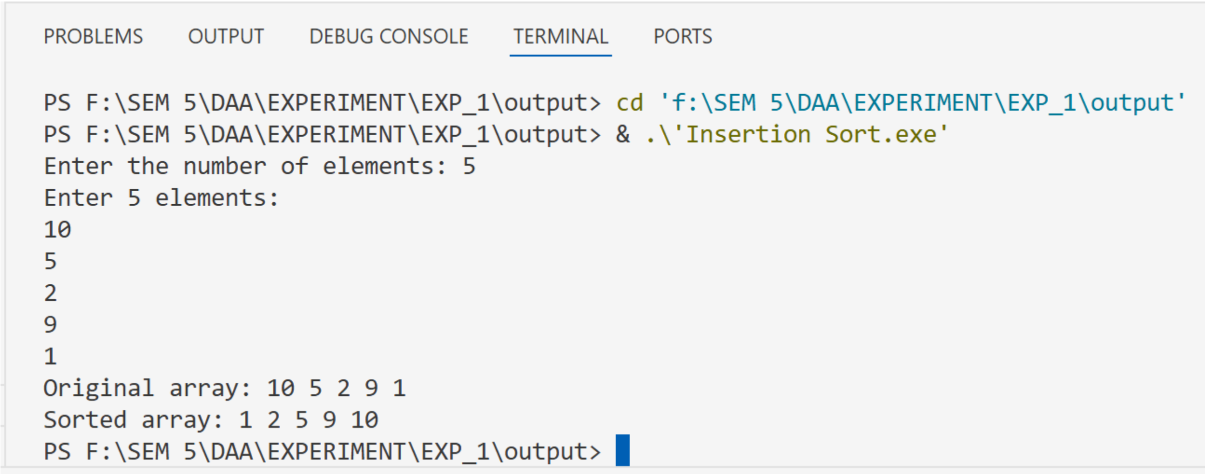
Programming Language: C++

Code:

```
#include <iostream>
using namespace std;
void insertionSort(int arr[], int n) {
    for(int i = 1; i < n; i++) {
        int current = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > current) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = current;
    }
}
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;

    int arr[n];
```

```
            cout << "Enter " << n << " elements:\n";
            for(int i = 0; i < n; i++) {
                cin >> arr[i];
            }
            cout << "Original array: ";
            for(int i = 0; i < n; i++) {
                cout << arr[i] << " ";
            }
            insertionSort(arr, n);
            cout << "\nSorted array: ";
            for(int i = 0; i < n; i++) {
                cout << arr[i] << " ";
            }
            return 0;
        }
```

Output:



```
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> cd 'f:\SEM 5\DAA\EXPERIMENT\EXP_1\output'
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> & .\'Insertion Sort.exe'
Enter the number of elements: 5
Enter 5 elements:
10
5
2
9
1
Original array: 10 5 2 9 1
Sorted array: 1 2 5 9 10
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output>
```

1. **Space complexity: O(1)**

   Justification: Insertion sort sorts the array in place and does not require any extra memory except for a few variables.

2. **Time complexity:**
   - **Best case time complexity:** O(n)
     Justification:
     When the array is already sorted, only one comparison is needed for each element.
   - **Worst case time complexity:** O(n^2)
     Justification:
     When the array is sorted in reverse order, each new element is compared with all sorted elements before it, resulting in n(n-1)/2 comparisons.

2. Bubble Sort:

Theory:

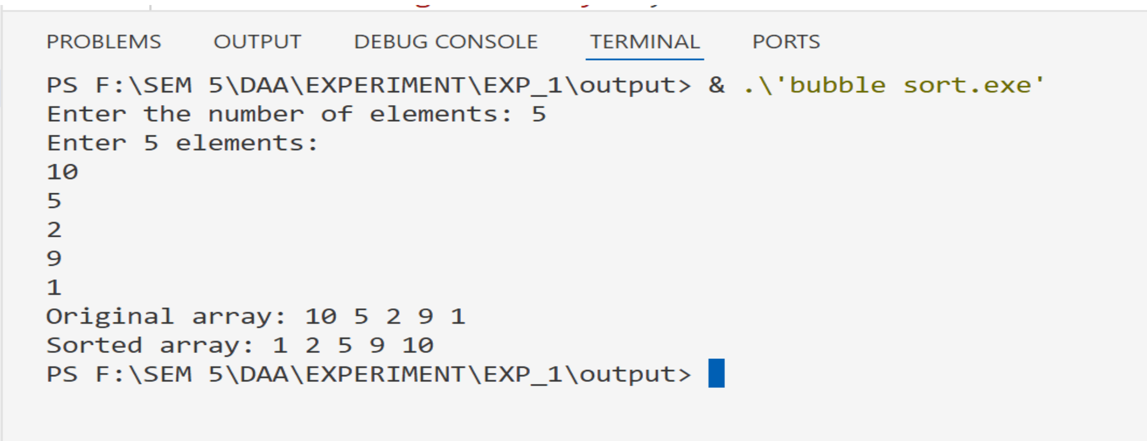
- Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. This process is repeated until the entire array is sorted. It "bubbles" the largest unsorted value to the end in each pass.

Programming Language: C++

Code:

```
#include <iostream>
using namespace std;
void insertionSort(int arr[], int n) {
    for(int i = 1; i < n; i++) {
        int current = arr[i];
        int j = i - 1;
        while(j >= 0 && arr[j] > current) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = current;
    }
}
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter " << n << " elements:\n";
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Original array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    insertionSort(arr, n);
    cout << "\nSorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output:



```
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> & .\'bubble sort.exe'
Enter the number of elements: 5
Enter 5 elements:
10
5
2
9
1
Original array: 10 5 2 9 1
Sorted array: 1 2 5 9 10
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> 
```

1. **Space complexity: 0(1)**

   Justification: Bubble Sort sorts in-place and uses only a constant amount of extra memory.

2. **Time complexity:**
   - **Best case time complexity:** 0(n)
     Justification: If the array is already sorted, only one pass is required with no swaps.
   - **Worst case time complexity:** 0(n^2)
     Justification: When the array is in reverse order, it requires n passes and n-1 comparisons each time, resulting in n(n-1)/2 comparisons.

3. Selection Sort:

Theory:

- Selection Sort divides the array into a sorted and an unsorted part. In each iteration, it finds the **minimum element** from the unsorted part and swaps it with the first unsorted element, growing the sorted portion one step at a time. It always makes the same number of comparisons, regardless of the input arrangement.

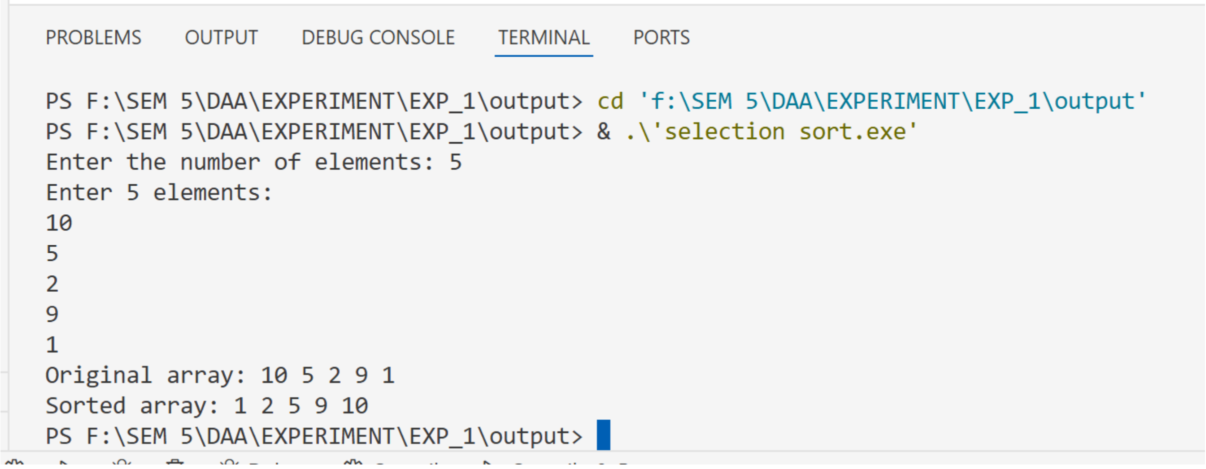
Programming Language: C++

Code:

```cpp
#include <iostream>

using namespace std;

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;

            }

        }

        if(minIndex != i) {

            swap(arr[i], arr[minIndex]);

        }

    }

}
```

```
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter " << n << " elements:\n";
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Original array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    selectionSort(arr, n);
    cout << "\nSorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    return 0;
}
```

Output:



```
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> cd 'f:\SEM 5\DAA\EXPERIMENT\EXP_1\output'
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> & .\'selection sort.exe'
Enter the number of elements: 5
Enter 5 elements:
10
5
2
9
1
Original array: 10 5 2 9 1
Sorted array: 1 2 5 9 10
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output>
```

1. **Space complexity: O(1)**

   Justification: Selection Sort does not require any extra storage; sorting happens within the array using a fixed number of variables.

2. **Time complexity:**
   - **Best case time complexity:** O(n^2)
     Justification: Unlike some other algorithms, Selection Sort always performs the same number of comparisons, even if the array is already sorted. It makes n(n-1)/2 comparisons in all cases.
   - **Worst case time complexity:** O(n^2)
     Justification: In the worst case (reverse order or unsorted array), the algorithm still performs the same fixed number of comparisons and a maximum number of swaps.

4. Counting Sort:

Theory:

- Counting Sort is a **non-comparison-based** sorting algorithm. It works by counting the number of occurrences of each unique element in the input array. It then uses this count to determine the position of each element in the sorted output. It is only efficient when the input values lie within a **small known range** (e.g., 0 to 100).

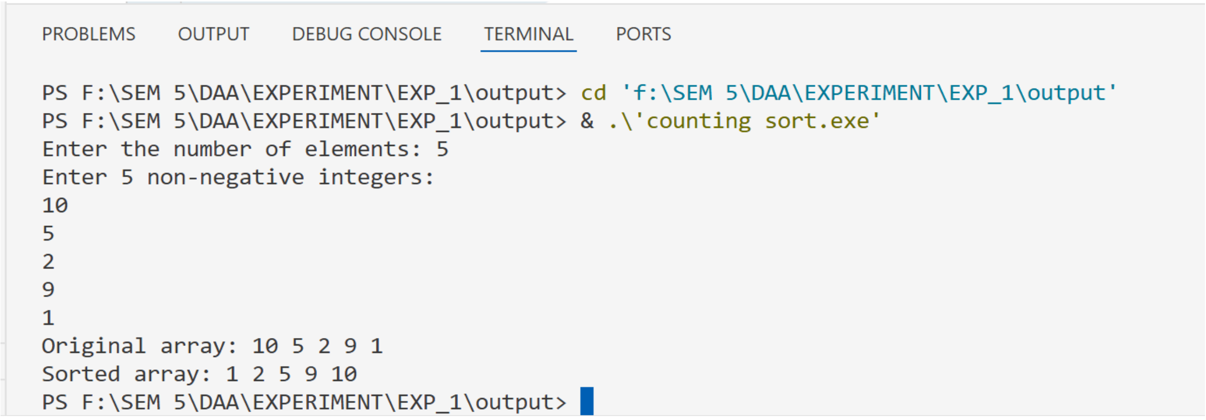
Programming Language: C++

Code:

```cpp
#include <iostream>
using namespace std;
void countingSort(int arr[], int n) {
    int maxElement = arr[0];
    for(int i = 1; i < n; i++) {
        if(arr[i] > maxElement)
            maxElement = arr[i];
    }
    int count[maxElement + 1] = {0};
    for(int i = 0; i < n; i++) {
        count[arr[i]]++;
    }
    int index = 0;
    for(int i = 0; i <= maxElement; i++) {
            while(count[i] > 0) {
                arr[index++] = i;
                count[i]--;
            }
        }
}
int main() {
    int n;
    cout << "Enter the number of elements: ";
    cin >> n;
    int arr[n];
    cout << "Enter " << n << " non-negative integers:\n";
    for(int i = 0; i < n; i++) {
        cin >> arr[i];
    }
    cout << "Original array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
    }
    countingSort(arr, n);
    cout << "\nSorted array: ";
    for(int i = 0; i < n; i++) {
        cout << arr[i] << " ";
```

```
            }

            return 0;

        }
```

Output:



```
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> cd 'f:\SEM 5\DAA\EXPERIMENT\EXP_1\output'
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output> & .\'counting sort.exe'
Enter the number of elements: 5
Enter 5 non-negative integers:
10
5
2
9
1
Original array: 10 5 2 9 1
Sorted array: 1 2 5 9 10
PS F:\SEM 5\DAA\EXPERIMENT\EXP_1\output>
```

1. **Space complexity: O(k)**

   Justification: Counting Sort uses extra space for the count array, where k is the range of the input values. So, space depends on both n and k, but auxiliary space is still linear.

2. **Time complexity:**

- **Best case time complexity:**O(n + k)
  Justification: All input values are scanned once to fill the count array, and again to build the sorted array. So, the time depends on n (input size) and k (range of values).
- **Worst case time complexity:** O(n + k)
  Justification: Even in the worst case, Counting Sort still performs only two passes over the array and one pass over the count array. However, if k is very large, performance drops.