EXPERIMENT 1

Aim:

1. Write a program in C to perform linear and binary search.
2. Write a program in C to perform bubble sort, insertion sort and selection sort. Take the array size and array elements from user.
3. Write a program in C that obtains the minimum and maximum element from the array. Modify this program to give the second largest and second smallest element of the array.

Theory (Linear Search):

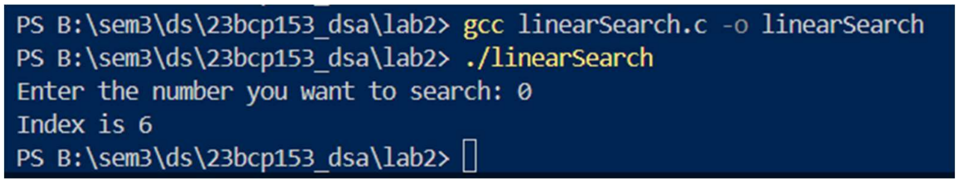
Linear search is a straightforward algorithm for finding a target value within an array. It involves sequentially checking each element from the beginning until the target value is found or the end of the array is reached. This method is simple but can be inefficient for large arrays, as it may require examining every element.

Program:

```
#include <stdio.h>

int linearSearch(int item, int size, int arr[]);

int main()
{
  //VedeshP
  // Linear search
  int item;
  int arr[] = {3,4,1,2,9,7,0,5,8,6};
  printf("Enter the number you want to search: ");
  scanf("%d", &item);
  int size = sizeof(arr) / sizeof(arr[0]);
  int index = linearSearch(item, size, arr);
  // printf("%d", size);

  if (index == -1)
  {
    printf("Number not found");
    return -1;
  }
  else
  {
    printf("Index is %d", index);
    return 0;
  }
}
```

```
int linearSearch(int item, int size, int arr[])
{
  for (int i = 0; i < size; i++)
  {
    if (arr[i] == item)
    {
      return i;
    }
  }

  return -1;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> gcc linearSearch.c -o linearSearch
PS B:\sem3\ds\23bcp153_dsa\lab2> ./linearSearch
Enter the number you want to search: 0
Index is 6
PS B:\sem3\ds\23bcp153_dsa\lab2>
```

Time Complexity:

- **Worst Case:** $O(n)$ – The target element is at the last position or not present, so every element in the array must be checked.
- **Average Case:** $O(n)$ – On average, the search will check half of the elements, which still results in linear time complexity.
- **Best Case:** $O(1)$ – The target element is at the first position, so only one comparison is needed.

Theory (Binary Search):

Binary search is an efficient algorithm for finding a target value in a sorted array. It works by repeatedly dividing the search interval in half. Starting with the entire array, the algorithm compares the target value to the middle element. If the target is equal to the middle element, the search is complete. If the target is less than the middle element, the search continues in the lower half; if greater, in the upper half. This process continues until the target is found or the interval is empty.

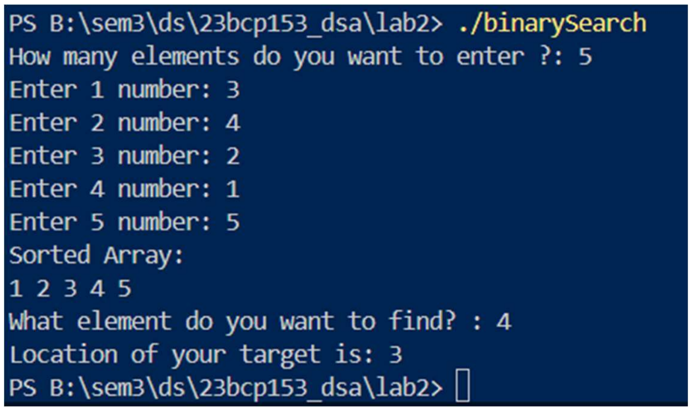
Program:

```
#include <stdio.h>

// Get time here too for complexity

int binarySearch(int arr[], int start, int end, int target);
void insertionSort(int arr[], int n);
```

```
int main(void)
{
    int n = 0;
    printf("How many elements do you want to enter ?: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);
    printf("Sorted Array:\n");
    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }
    printf("\n");

    int target;
    printf("What element do you want to find? : ");
    scanf("%d", &target);

    int location = binarySearch(arr, 0, n - 1, target);
    if (location == -1)
    {
        printf("Element not found!");
        return -1;
    }

    printf("Location of your target is: %d", location);

    return 0;
}
```

```
int binarySearch(int arr[], int start, int end, int target)
{

    while (start <= end)
    {
        int mid = (start + end) / 2;

        if (arr[mid] == target)
        {
            return mid;
        }
        else if (arr[mid] < target)
        {
            start = mid + 1;
        }
        else
        {
            end = mid - 1;
        }
    }

    return -1;
}

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

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

    return;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> ./binarySearch
How many elements do you want to enter ?: 5
Enter 1 number: 3
Enter 2 number: 4
Enter 3 number: 2
Enter 4 number: 1
Enter 5 number: 5
Sorted Array:
1 2 3 4 5
What element do you want to find? : 4
Location of your target is: 3
PS B:\sem3\ds\23bcp153_dsa\lab2>
```

Time Complexity:

Worst Case: $O(log\ n)$

- **Explanation:** In binary search, the array is divided in half with each iteration. Mathematically, this means the number of elements to check is reduced exponentially. After k iterations, the number of elements left to check is $\frac{n}{2^k}$. We stop when $\frac{n}{2^k}$ becomes 1, which means $2^k = n$. Taking the logarithm (base 2) of both sides, we get $k = \log_2(n)$. Thus, the time complexity is $O(log\ n)$.
- Because in this we do the iterations in this manner: $\frac{n}{2}, \frac{n}{4}, \frac{n}{8}, \ldots$
- When we are left with only one element we get: $\frac{n}{2^k} = 1$, therefore $\log\ (n)$

Average Case: $O(log\ n)$

- **Explanation:** On average, binary search will also perform log(n) comparisons. This is because, regardless of the position of the target value, the algorithm always divides the search space in half, leading to the same logarithmic growth rate.

Best Case: $O(1)$

- **Explanation:** The best case occurs when the target value is the middle element of the array in the first comparison. In this case, only one comparison is needed, resulting in constant time complexity, $O(1)$.

Theory (Bubble Sort):

Bubble sort is a simple sorting algorithm that repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order. The process is repeated for each element in the list until the entire list is sorted. In each pass through the list, the largest unsorted element "bubbles up" to its correct position. The algorithm continues to reduce the number of elements considered in each subsequent pass, effectively sorting the list from the end towards the beginning. The number of passes needed is one less than the number of elements in the list.

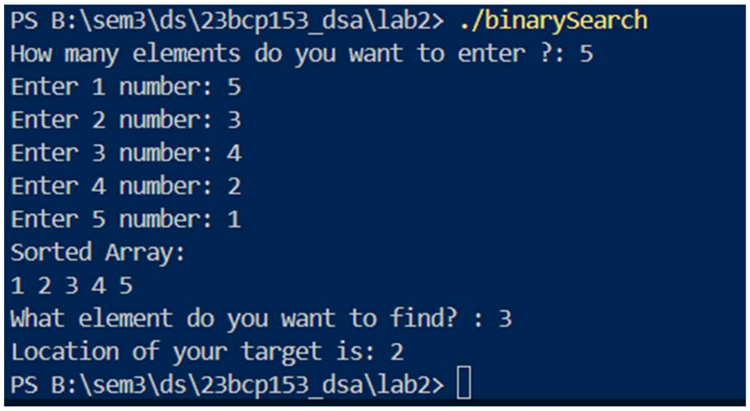
Program:

```
#include <stdio.h>

void bubbleSort(int arr[], int n);

int main(void)
{
   int n = 0;
   printf("How many elements do you want to enter ?: ");
   scanf("%d", &n);
   int arr[n];
```

```
    for (int i = 0; i < n; i++)
    {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    bubbleSort(arr, n);

    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

```
void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++)
    {
        for (int j = 0; j < n - i - 1; j++)
        {
            if (arr[j] > arr[j + 1])
            {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
            }
        }
    }
    return;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> ./binarySearch
How many elements do you want to enter ?: 5
Enter 1 number: 5
Enter 2 number: 3
Enter 3 number: 4
Enter 4 number: 2
Enter 5 number: 1
Sorted Array:
1 2 3 4 5
What element do you want to find? : 3
Location of your target is: 2
PS B:\sem3\ds\23bcp153_dsa\lab2>
```

Time Complexity:

Worst Case: $O(n^2)$

- **Explanation:** Occurs when the array is sorted in reverse order. The algorithm needs to perform a maximum number of comparisons and swaps, resulting in a quadratic time complexity.

Average Case: $O(n^2)$

- **Explanation:** On average, bubble sort performs a quadratic number of comparisons and swaps, as it does not have an efficient way to handle partially sorted arrays.

Best Case: $O(n)$

- **Explanation:** Occurs when the array is already sorted. With an optimized version of bubble sort that checks if any swaps were made in a pass, the algorithm can complete in linear time. If no swaps are needed during a pass, the algorithm terminates early.

Theory (Insertion Sort):

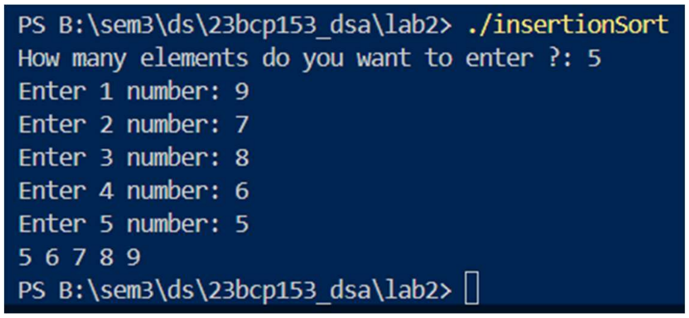
Insertion sort maintains a sorted section of the array and iterates through the unsorted section. For each element in the unsorted section, it compares the element with those in the sorted section and inserts it into the correct position within the sorted part. This process involves shifting elements in the sorted part to make space for the new element. The algorithm continues until all elements are sorted, resulting in a fully ordered array.

Program:

```
#include <stdio.h>

void insertionSort(int arr[], int n);

int main(void)
{
    int n = 0;
    printf("How many elements do you want to enter ?: ");
    scanf("%d", &n);
    int arr[n];
    for (int i = 0; i < n; i++)
    {
        printf("Enter %d number: ", i + 1);
        scanf("%d", &arr[i]);
    }

    insertionSort(arr, n);

    for (int i = 0; i < n; i++)
    {
        printf("%d ", arr[i]);
    }

    return 0;
}
```

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

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

    return;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> ./insertionSort
How many elements do you want to enter ?: 5
Enter 1 number: 9
Enter 2 number: 7
Enter 3 number: 8
Enter 4 number: 6
Enter 5 number: 5
5 6 7 8 9
PS B:\sem3\ds\23bcp153_dsa\lab2>
```

Time Complexity:

Worst Case: $O(n^2)$

- **Explanation:** Occurs when the array is sorted in reverse order. For each element, the algorithm might need to compare it with every element in the sorted portion, leading to quadratic time complexity.

Average Case: $O(n^2)$

- **Explanation:** On average, the algorithm performs a quadratic number of comparisons and shifts, as elements are not uniformly distributed, requiring significant comparisons and shifts.

Best Case: $O(n)$

- **Explanation:** Occurs when the array is already sorted. The algorithm only needs to make a single pass through the array, performing a constant amount of work for each element as no elements need to be shifted.

Theory (Selection Sort):

Selection sort is a simple sorting algorithm that sorts an array by repeatedly selecting the smallest (or largest) element from the unsorted portion and moving it to the end of the sorted portion. Here's a step-by-step explanation:

1. **Find the Minimum:** Start by finding the minimum element in the unsorted part of the array.
2. **Swap:** Swap this minimum element with the first element of the unsorted part. This places the minimum element in its correct position in the sorted part of the array.
3. **Move the Boundary:** The boundary between the sorted and unsorted parts of the array moves one element forward.
4. **Repeat:** Repeat the process for the remaining unsorted elements until the entire array is sorted.

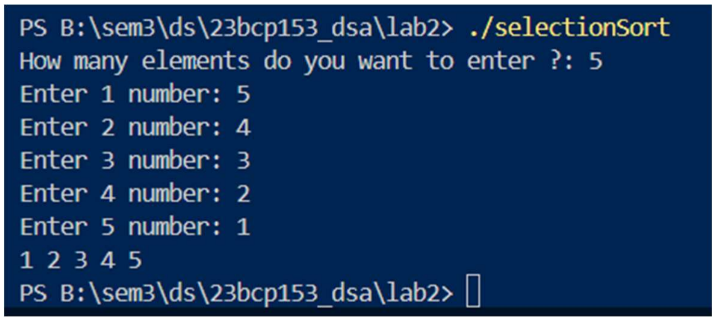
Program:

```
#include <stdio.h>

void selectionSort(int arr[], int n);

int main(void)
{
  int n = 0;
  printf("How many elements do you want to enter ?: ");
  scanf("%d", &n);
  int arr[n];
  for (int i = 0; i < n; i++)
  {
    printf("Enter %d number: ", i + 1);
    scanf("%d", &arr[i]);
  }

  selectionSort(arr, n);

  for (int i = 0; i < n; i++)
  {
    printf("%d ", arr[i]);
  }
```

```
    return 0;
}
```

```
void selectionSort(int arr[], int n)
{
    for (int i = 0; i < n; i++)
    {
        int min = i;
        for (int j = i; j < n; j++)
        {
            if (arr[j] < arr[min])
            {
                min = j;
            }
        }
        if (min != i)
        {
            int temp = arr[i];
            arr[i] = arr[min];
            arr[min] = temp;
        }
    }

    return;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> ./selectionSort
How many elements do you want to enter ?: 5
Enter 1 number: 5
Enter 2 number: 4
Enter 3 number: 3
Enter 4 number: 2
Enter 5 number: 1
1 2 3 4 5
PS B:\sem3\ds\23bcp153_dsa\lab2> 
```

Time Complexity:

Worst Case: $O(n^2)$

- **Explanation:** In every iteration, selection sort scans the unsorted portion of the array to find the minimum element. This scanning process involves O(n) comparisons for each of the n elements, resulting in a quadratic time complexity.

Average Case: $O(n^2)$

- **Explanation:** Regardless of the initial order of elements, selection sort always performs a fixed number of comparisons and swaps. The overall number of comparisons remains quadratic.

Best Case: $O(n^2)$

- **Explanation:** Even if the array is already sorted, selection sort still performs the same number of comparisons to find the minimum element for each iteration. Hence, the best case also has quadratic time complexity.

Theory (Minimum and Maximum elements in array):

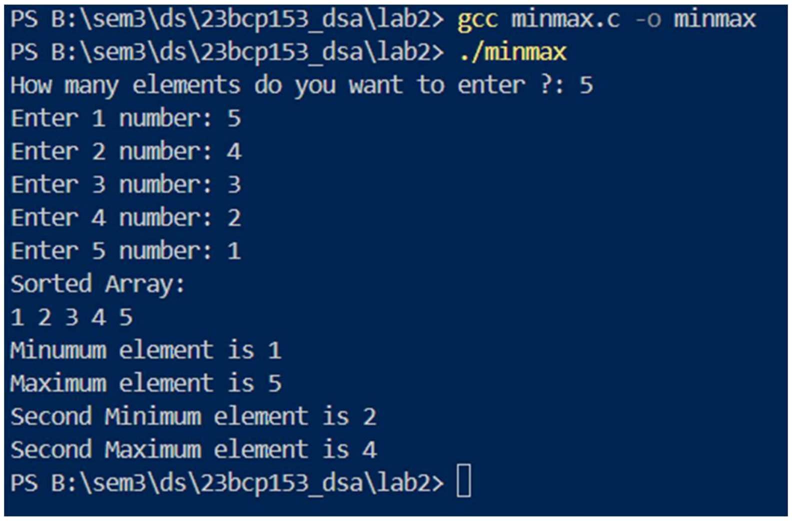
Using insertion sort to sort the array and then providing the first element, and the last element for the minimum element and the maximum element as the output respectively. Similarly providing 2nd element and 2nd last element for the 2nd minimum and 2nd maximum elements respectively.

Program:

```
#include <stdio.h>

void insertionSort(int arr[], int n);

int main(void)
{
   int n = 0;
   printf("How many elements do you want to enter ?: ");
   scanf("%d", &n);
   int arr[n];
   for (int i = 0; i < n; i++)
   {
      printf("Enter %d number: ", i + 1);
      scanf("%d", &arr[i]);
   }

   insertionSort(arr, n);
   printf("Sorted Array:\n");
   for (int i = 0; i < n; i++)
   {
      printf("%d ", arr[i]);
   }
   printf("\n");

   printf("Minumum element is %d\n", arr[0]);
   printf("Maximum element is %d\n", arr[n - 1]);
   printf("Second Minimum element is %d\n", arr[1]);
   printf("Second Maximum element is %d\n", arr[n - 2]);

   return 0;
}
```

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

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

   return;
}
```

Output:



```
PS B:\sem3\ds\23bcp153_dsa\lab2> gcc minmax.c -o minmax
PS B:\sem3\ds\23bcp153_dsa\lab2> ./minmax
How many elements do you want to enter ?: 5
Enter 1 number: 5
Enter 2 number: 4
Enter 3 number: 3
Enter 4 number: 2
Enter 5 number: 1
Sorted Array:
1 2 3 4 5
Minumum element is 1
Maximum element is 5
Second Minimum element is 2
Second Maximum element is 4
PS B:\sem3\ds\23bcp153_dsa\lab2>
```

Time Complexity:

Worst Case: $O(n^2)$

- **Explanation:** Occurs when the array is sorted in reverse order. For each element, the algorithm might need to compare it with every element in the sorted portion, leading to quadratic time complexity.

Average Case: $O(n^2)$

- **Explanation:** On average, the algorithm performs a quadratic number of comparisons and shifts, as elements are not uniformly distributed, requiring significant comparisons and shifts.

Best Case: $O(n)$

- **Explanation:** Occurs when the array is already sorted. The algorithm only needs to make a single pass through the array, performing a constant amount of work for each element as no elements need to be shifted.

Same as insertion sort as we are performing insertion sort here.