|  |  |  |  |
| --- | --- | --- | --- |
| **Academic Year** | **2025 - 26** | **Experiment No.** | **3** |
| **Course & Semester** | **S.E. – Sem. III** | **Subject Name** | **Analysis of Algorithm** |
| **Experiment Type** | **Software Performance** | **Subject Code** | **25PCC12CS05** |

|  |  |  |  |
| --- | --- | --- | --- |
|  | Atharva Dharmendra Jagtap | **Roll No.:** | 10937 |
| **Date of Performance:** |  | **Date of Submission:** |  |
| **LO Mapping** | 25PCC12CS05.1: Analyze the time and space complexity of algorithms.  25PCC12CS05.2: Apply divide and conquer strategy to solve a problem. | | |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| |  |  |  |  | | --- | --- | --- | --- | | **Indicator** | **Poor** | **Average** | **Good** | | Timeline Maintains submission deadline (3) | Submission not done (0) | One or More than One week late (1-2) | Maintains deadline (3) | | Completion and Organization (3) | N/A | Document is just acceptable (1-2) | Completed whole document and neatly organized (3) | | Program Performance (2) | Could not perform at all (0) | Implemented few parts (1) | Full implementation (2) | | Knowledge In depth knowledge of the Experiment (2) | Unable to answer questions (0) | Unable to answer few questions (1) | Able to answer all questions (2) | |
| **Assessment Marks:**   |  |  | | --- | --- | | Timeline |  | | Completion and Organization |  | | Program Performance |  | | Knowledge |  | |
| Total: (Out of 10) |
| Teacher’s Sign: Student Sign: |

**Experiment No. 3**

**AIM:** Implement and Analyze time and space complexity of finding minimum and maximum element of an array using divide and conquer strategy.

**THEORY:** The goal of this experiment is to find the **minimum and maximum elements** in an array using a **divide and conquer approach**, rather than scanning the entire array linearly.

In the **naive (brute-force)** method, we compare each element of the array to determine the minimum and maximum, which takes **O(n)** time with **2(n−1)** comparisons.

However, the **Divide and Conquer** approach reduces the number of comparisons by splitting the array into two halves, recursively finding the min and max of each half, and then combining the results with only two additional comparisons.  
This approach also uses **O(n)** time, but with fewer comparisons, i.e., approximately **3n/2 − 2** comparisons.

* **Time Complexity**: O(n)
* **Space Complexity**: O(log n) due to the recursive stack in divide and conquer.

This method is especially useful in systems with constraints on the number of comparisons or in recursive environments.

### ****ALGORITHM****:

#### ****1. Start****

* Begin the program.

#### ****2. Define a Function:****

* find\_min\_max(arr, low, high):
  + Takes the array and its starting (low) and ending (high) indices.
  + Returns a pair (min, max).

#### ****3. Base Cases:****

* **If low == high** (only one element):
  + Return (arr[low], arr[low]) → both min and max are the same.
* **If high == low + 1** (two elements):
  + Compare them and return the smaller as min and the larger as max.

#### ****4. Recursive Case:****

* Find the middle index: mid = (low + high) // 2
* Recursively find min and max for:
  + Left half: find\_min\_max(arr, low, mid)
  + Right half: find\_min\_max(arr, mid + 1, high)
* Combine the results:
  + Overall min = min(left\_min, right\_min)
  + Overall max = max(left\_max, right\_max)

#### ****5. Input the Array:****

* Read array elements and size n.

#### ****6. Call the Recursive Function:****

* Call find\_min\_max(arr, 0, n - 1).

#### ****7. Display the Result:****

* Print the minimum and maximum values returned by the function.

#### ****8. End****

* End the program.

**CODE:**

#include <stdio.h>

struct Pair

{

    int min;

    int max;

};

struct Pair find\_min\_max(int arr[], int low, int high)

{

    struct Pair result;

    if (low == high || low + 1 == high)

    {

        if (arr[low] <= arr[high])

        {

            result.min = arr[low];

            result.max = arr[high];

        }

        else

        {

            result.min = arr[high];

            result.max = arr[low];

        }

    }

    else

    {

        int mid;

        struct Pair beg, end;

        mid = low + (high - low) / 2;

        beg = find\_min\_max(arr, low, mid);

        end = find\_min\_max(arr, mid + 1, high);

        if (beg.min <= end.min)

            result.min = beg.min;

        else

            result.min = end.min;

        if (beg.max >= end.max)

            result.max = beg.max;

        else

            result.max = end.max;

    }

    return result;

}

int main()

{

    int i, n;

    printf("Enter number of elements: ");

    scanf("%d", &n);

    int arr[n];

    printf("Enter integer array elements:");

    for (i = 0; i < n; i++)

    {

        scanf("%d", &arr[i]);

    }

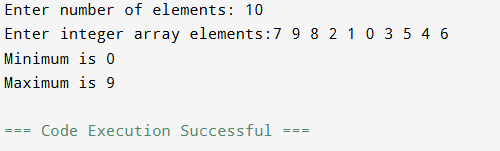
    struct Pair result = find\_min\_max(arr, 0, n - 1);

    printf("Minimum is %d\nMaximum is %d", result.min, result.max);

    return 0;

}

**OUTPUT:**

****

**POST LAB QUESTIONS**

1. If the input size is very small (e.g., 2 or 3 elements), is it still worth using divide and conquer? Why or why not?

**Ans.** No,it’s not worth it because of Overhead of recursion outweighs benefits as s simple linear scan is faster for such small arrays.

1. Can this method be extended to find the second smallest or second largest element? Explain.

**Ans.** Yes, but needs modification to tack two values: (min, second\_min) and (max, second\_max) during comparisons. The complexity remains O(n) but logic becomes more complex to handle ties and updates efficiently.

1. If the array has an odd number of elements, how does the algorithm handle the middle element? Explain with an example.

**Ans.** The algorithm includes the middle element naturally because of recursive splitting.

**Example:**

arr = [6, 2, 4, 7, 9] (n=5)

low=0, high=4 → mid=2

Left = [6, 2, 4]

Right = [7, 9]

Middle element arr[2] = 4 is in the left partition and processed normally.

**CONCLUSION:**

For very small inputs, the recursion overhead makes this approach inefficient. The algorithm can be adapted to find the second smallest or largest element, but it requires additional logic. When the array size is odd, the algorithm naturally splits it into two sub-arrays, with one containing the middle element, ensuring the final result remains accurate.