Practical 10: General Problem

Aim

To design and solve given problems using different algorithmic approaches and analyse their complexity.

Algorithm

1. Take an input ***array A*** of length ***n***.
2. Initialize two variables ***left*** and ***right*** to represent the first and last indices of the array, respectively.
3. While **left is less than right**, repeat the following steps:
4. Calculate the **middle index** mid as the average of left and right.
5. If the element at A[mid] is greater than the element at A[mid + 1], **discard the right half** of the array by setting right = mid.
6. Otherwise, **discard the left half** of the array by setting left = mid + 1.
7. **Return the index *left***, which is the index of the peak element in the array.

Program

```
public class Main {
   public static void main(String[] args) {
        int[] A = {1, 3, 5, 7, 8, 6, 4, 2};
        int n = A.length;
        int left = 0, right = n - 1;

        // Continue until left and right indices become same
        while (left < right) {
           int mid = (left + right) / 2; // Middle index
           if (A[mid] > A[mid + 1]) {
              right = mid; // Discard right half of array & use left half
           } else {
              left = mid + 1; // Discard left half of array & use right half
           }
        }
        System.out.printf("Peak Element is %d at index %d",A[left] , left); // Return index of the peak element
      }
}
```

Output:

```
Enter length of array: 8
Enter Element 0 : 1
Enter Element 1 : 3
Enter Element 2 : 5
Enter Element 3 : 7
Enter Element 4 : 8
Enter Element 5 : 6
Enter Element 6 : 4
Enter Element 7 : 2
Peak Element is 8 at index 4
```

Analysis of Algorithm

Time Complexity:

The time complexity of the binary search algorithm for finding the peak element in a unimodal array is ***O(log n)***, where **n** is the size of the input array. This is because in each iteration of the while loop, the search range is **reduced by half**. Since the algorithm only needs to search a fraction of the array, it is much faster than searching the entire array, which would take ***O(n)*** time. Therefore, the algorithm is efficient and has a logarithmic time complexity.

Space Complexity:

The space complexity of the algorithm is ***O(1)*** because it only uses a constant amount of extra space to store the **left**, **right**, and **mid** indices. The algorithm does not create any additional arrays or data structures that grow with the size of the input array, so its space complexity does not depend on the size of the input. Therefore, the algorithm is space-efficient and does not use excessive memory.