Experiment 5.1

Student Name: Aryan Mehta
UID: 22BCS11011
Branch: BE-CSE
Section/Group: IoT_641(A)
Semester: 6th
Date of Performance: 28/2/25
Subject Name: Advanced Programming-2
Subject Code: 22CSP-351

1. **Aim:** Find the k most frequent elements in an array. Use a hash map to count frequencies, then use a min-heap or bucket sort to extract the top k elements. The heap approach runs in O(n log k) time. Edge cases include k = 1 or all elements being unique.
2. **Objective:** Implement an algorithm to identify and return the k most frequently occurring elements within a given array.
3. **Implementation/Code:**

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int,int> mp;
        int n=nums.size();
        for(int i=0;i<n;i++){
            mp[nums[i]]++;
        }
        vector<pair<int,int>> freq(mp.begin(),mp.end());
        sort(freq.begin(), freq.end(), [](auto &a, auto &b) {
        return a.second > b.second;
    });
        vector<int> ans;

        for(int i=0;i<k;i++){
            ans.push_back(freq[i].first);
        }

        return ans;
    }
};
```

4. Output

5. Learning Outcomes

- Understand the concept of hash maps and min-heaps.
- Learnt how to implement the algorithm in a programming language.
- Learnt how to evaluate and compare the min-heap and bucket sort approaches for top-k extraction.
- Understand the importance of code efficiency, and how to create code that runs within the desired time constraints.

<u>Experiment 5.2</u>

1. **Aim**: Find the kth largest element in an array using a min-heap or Quickselect. The heap approach runs in O(n log k) time, while Quickselect runs in O(n) on average. Edge cases include k = 1 (max element) and k = n (min element).
2. **Objective**: Implement algorithms to efficiently find the kth largest element within a given array.

3. **Code**:

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
```

```
        priority_queue<int>max;
        int a=1;
       for(auto p:nums){
           max.push(p);
       }
       for(int i=0;i<nums.size();i++){
           if(a<k){
               max.pop();
               a++;
           }
       }
       return max.top();
    }
};
```

4. Output

5. Learning Outcomes:

- Understand the steps involved in both the min-heap and Quickselect algorithms for finding the kth largest element.
- Learn how to analyze and compare the time complexity of the min-heap (O(n log k)) and Quickselect (O(n) average) approaches.
- Implement both the min-heap and Quickselect algorithms in a programming language.

Experiment 5.3

1. **Aim**: Find the kth smallest element in a row- and column-sorted matrix. Use a min-heap (O(k log n)) or binary search on values (O(n log max-min)). Edge cases include k = 1 (smallest element) and k = n^2 (largest element).
2. **Objective**: Implement an algorithm to find the kth smallest element using binary search on values.

3. **Code**:

```
class Solution
{
public:
    int kthSmallest(vector<vector<int>>& matrix, int k)
    {
        int n = matrix.size();
        int le = matrix[0][0], ri = matrix[n - 1][n - 1];
        int mid = 0;
        while (le < ri)
        {
            mid = le + (ri-le)/2;
            int num = 0;
            for (int i = 0; i < n; i++)
            {
                int pos = upper_bound(matrix[i].begin(), matrix[i].end(), mid) -
matrix[i].begin();
                num += pos;
            }
            if (num < k)
            {
                le = mid + 1;
            }
            else
            {
                ri = mid;
            }
        }
        return le;
    }
};
```

4. **Output**

5. **Learning Outcomes:**
    - Understand the steps involved in both the min-heap and binary search algorithms.
    - Learn how to analyze and compare the time complexity.
    - Implement appropriate algorithm based on the problem's constraints.