Day 1 / 100 :

Topic - Array

1 Problem statement: Majority element (easy)

Given an array nums of size n, return the majority element.
The majority element is the element that appears more than [n / 2] times. You may assume that the majority element always exists in the array.

Example 1:

Input: nums = [3,2,3]
Output: 3
Example 2:

Input: nums = [2,2,1,1,1,2,2]
Output: 2

Solutions :

Approach 1 - sorting

Explanation:

1. The code begins by sorting the array nums in non-decreasing order using the sort function from the C++ Standard Library. This rearranges the elements such that identical elements are grouped together.
2. Once the array is sorted, the majority element will always be present at index n/2, where n is the size of the array.
3. This is because the majority element occurs more than n/2 times, and when the array is sorted, it will occupy the middle position.
4. The code returns the element at index n/2 as the majority element.
5. The time complexity of this approach is O(n log n) since sorting an array of size n takes **O(n log n)** time.

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        sort(nums.begin(), nums.end());
        int n = nums.size();
        return nums[n/2];
    }
};
```

Approach 2 - Hash Map

Explanation:

1. The code begins by initializing a hash map m to store the count of occurrences of each element.
2. It then iterates through the array nums using a for loop.
3. For each element nums[i], it increments its count in the hash map m by using the line m[nums[i]]++;.
    - If nums[i] is encountered for the first time, it will be added to the hash map with a count of 1.
    - If nums[i] has been encountered before, its count in the hash map will be incremented by 1.
4. After counting the occurrences of each element, the code updates n to be n/2, where n is the size of the array. This is done to check if an element occurs more than n/2 times, which is the criteria for being the majority element.
5. The code then iterates through the key-value pairs in the hash map using a range-based for loop.
    - For each key-value pair (x.first, x.second), it checks if the count x.second is greater than n.
    - If the count is greater than n, it means that x.first occurs more than n/2 times, so it returns x.first as the majority element.
6. If no majority element is found in the hash map, the code returns 0 as the default value.
    - Note that this will only occur if the input array nums is empty or does not have a majority element.
7. The time complexity of this approach **is O(n)** because it iterates through the array once to count the occurrences and then iterates through the hash map, which has a maximum size of the number of distinct elements in the array.

```cpp
class Solution {
public:
    int majorityElement(vector<int>& nums) {
      int n = nums.size();
      unordered_map<int,int> map;
```

```
        for(int i=0;i<nums.size();i++){
            map[nums[i]]++;
        }
        n=n/2;
        for(auto x:map){
            if(x.second>n){
                return x.first;
            }
        }
        return 0;
    }
};
```

Approach 3 - Moore Voting Algorithm

Explanation:

The Moore Voting Algorithm is an efficient approach to find the majority element in an array. It utilizes the fact that the majority element occurs more than [n / 2] times, implying that it will dominate the array.

The algorithm works as follows:

1. Initialize two variables, candidate and count, to store the potential majority element and its count, respectively.
2. Iterate through the array:
    - If the count variable is 0, update the candidate to the current element and set the count to 1.
    - If the current element is the same as the candidate, increment the count by 1.
    - If the current element is different from the candidate, decrement the count by 1.
3. After the iteration, the candidate variable will hold the majority element.

The time complexity of the Moore's Voting Algorithm is **O(n)** since it traverses the array once.

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {
        int candidate = 0;
        int count = 0;

        for(int num : nums) {
            if(count == 0) {
                candidate = num;
                count = 1;
            } else if(candidate == num) {
                count++;
            } else {
                count--;
            }
        }

        return candidate;
    }
};
```

- In this implementation, I initialize the candidate and count variables to 0. Then, I iterate through the given array and update the candidate and count based on the Moore Voting Algorithm's rules. Finally, I return the candidate as the majority element.

- The Moore Voting Algorithm is an elegant and efficient solution to the majority element problem, requiring only a single pass through the array. It's a great technique to have in your problem-solving toolkit!

2 Problem statement: **Majority Element II** (Medium)

Given an integer array of size n, find all elements that appear more than [n/3] times.
Example 1:

Input: nums = [3,2,3]
Output: [3]

Example 2:

Input: nums = [1]
Output: [1]

Solutions :

Approach 1 - Hasp map

The given problem requires finding all elements in the array that appear more than [n/3] times. To solve this, we can use a hash map to store the count of each element in the array. **(logic is similar to previous question)**

The implementation provided in the solution code follows this approach, where the hash map map is used to store the element-count pairs. Finally, the elements that satisfy the count condition are returned as the result.

```
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        unordered_map<int,int> map;
        vector<int>result;
        int n= nums.size()/3;
        for(int i=0;i<nums.size();i++){
            map[nums[i]]++;
        }
        for(auto x:map){
            if(x.second>n){
                result.push_back(x.first);
            }
        }
        return result;
    }
};
```

Approach 2 - Moore Voting Algorithm

The optimized Moore Voting Algorithm uses two variables, candidate1 and candidate2, and two counters, count1 and count2, to find potential majority elements in a single pass through the array. Then, it verifies if either candidate1 or candidate2 appear more than [n/3] times, where n is the array size. The space complexity is **O(1)**, and the time complexity is **O(n)**.

We'll iterate through the array and perform the following checks:

- If the current number is the same as candidate1, we'll increment count1.
- If the current number is the same as candidate2, we'll increment count2.
- If count1 is 0, we'll update candidate1 to the current number and set count1 to 1.
- If count2 is 0, we'll update candidate2 to the current number and set count2 to 1.
- If none of the above conditions match, we'll decrement both count1 and count2.

```
class Solution {
public:
    vector<int> majorityElement(vector<int>& nums) {
        int candidate1 = 0, candidate2 = 0;
        int count1 = 0, count2 = 0;

        // Step 1: Finding potential candidates
        for (int num : nums) {
            if (num == candidate1) {
                count1++;
            } else if (num == candidate2) {
                count2++;
            } else if (count1 == 0) {
                candidate1 = num;
                count1 = 1;
            } else if (count2 == 0) {
                candidate2 = num;
                count2 = 1;
            } else {
                count1--;
                count2--;
            }
        }
        // Step 2: Counting occurrences of potential candidates
        count1 = 0;
        count2 = 0;
        for (int num : nums) {
            if (num == candidate1) {
                count1++;
            } else if (num == candidate2) {
                count2++;
            }
        }

        // Step 3: Checking if candidates appear more than n/3 times
        int n = nums.size();
        vector<int> result;
        if (count1 > n / 3) {
            result.push_back(candidate1);
        }
        if (count2 > n / 3 && candidate1 != candidate2) {
            result.push_back(candidate2);
        }

        return result;
    }
};
```

3️⃣Problem statement: **Maximum Length of a Concatenated String with Unique Characters** (Medium)

You are given an array of strings arr. A string s is formed by the concatenation of a subsequence of arr that has unique characters.

Return the maximum possible length of s.

A subsequence is an array that can be derived from another array by deleting some or no elements without changing the order of the remaining elements.

Example 1:

Input: arr = ["un","iq","ue"]
Output: 4
Explanation: All the valid concatenations are:
- ""
- "un"
- "iq"
- "ue"
- "uniq" ("un" + "iq")
- "ique" ("iq" + "ue")
Maximum length is 4.

Solutions :

Approach 1 - Brute Force

1. It first constructs bit masks for each string in the 'arr' vector to track unique characters.
2. It then iterates over all possible subsets of the 'arr' vector using a bitmask.
3. For each subset, it checks if the included strings have any overlapping characters based on the bit masks. If there are no overlaps, it calculates the length of the concatenated string for the current subset.
4. It keeps track of the maximum length found among all valid subsets and returns it as the result.
5. The time complexity of this approach is exponential, and the space complexity is linear.

Although this approach is straightforward, it may not be efficient for large inputs due to the exponential time complexity.

```
class Solution {
public:
    int maxLength(vector<string>& arr) {
        const size_t n = arr.size();
        vector<int> masks(n);
        for (int i = 0; i < n; i++) {
            for (char c : arr[i]) {
                if (masks[i] & (1 << (c - 'a'))) { /* has repeated
letters so cannot contribute */
                    arr[i] = "";
                    masks[i] = 0;
                    break;
                }
                masks[i] |= (1 << (c - 'a'));
            }
        }

        int ans = 0;
        for (int i = 0; i < (1 << n); i++) {

            int mask = 0; bool ok = true; int tot = 0;
            for (int j = 0; ok && j < n; j++) {
                if (i & (1 << j)) {
                    if (mask & masks[j]) ok = false;
                    mask |= masks[j];
                    tot += arr[j].size();
                }
            }
            if (ok) ans = max(ans, tot);
        }

        return ans;
    }
};
```

Approach 2 - Backtracking

To solve this problem, we can use a backtracking approach. We'll iterate through each string in the 'arr' array and try to include it in the concatenation or exclude it. We'll keep track of the maximum length found so far.

```cpp
class Solution {
public:
    int maxi=0;
    bool checkRepeat(string s){
        int check=0;
        for(auto it:s){
            int i=it-'a';
            if((check & 1<<i))
                return true;
            check^=1<<i;
        }
        return false;
    }
    void maxPossibleLength(int ind,int size,string s,vector<string>&
arr){
        if(checkRepeat(s))
            return;
        if(ind==size){
            int n=s.size();
            maxi=max(maxi,n);
            return;
        }
        maxPossibleLength(ind+1,size,s+arr[ind],arr);
        maxPossibleLength(ind+1,size,s,arr);
    }
    int maxLength(vector<string>& arr) {
        int size=arr.size();
        maxPossibleLength(0,size,"",arr);
        return maxi;
    }
};
```

1. We define a helper function called 'checkRepeat' that checks if a given string 's' contains any repeated characters. It uses a bitwise representation of a set to efficiently check for repetition.

2. The main function 'maxLength' takes the input array 'arr' and performs the following steps:
    - Initialize 'maxi' to 0, which will store the maximum length found so far.
    - Call the helper function 'maxPossibleLength' with initial parameters: index (ind) as 0, size as the length of 'arr', current concatenation (s) as an empty string, and the 'arr' array.
    - Return the value of 'maxi' as the result.
3. The helper function 'maxPossibleLength' takes the current index (ind), size, current concatenation (s), and the 'arr' array as parameters.
    - First, it checks if the current concatenation 's' contains repeated characters using the 'checkRepeat' function. If it does, it returns and backtracks.
    - If the index (ind) reaches the size, it means we have considered all strings in 'arr'. We calculate the length of 's' and update 'maxi' if the length is greater than 'maxi'.
    - Otherwise, we have two choices:
        - Include the string 'arr[ind]' in the concatenation by calling 'maxPossibleLength' recursively with the next index (ind+1) and the updated concatenation (s+arr[ind]).
        - Exclude the string 'arr[ind]' from the concatenation by calling 'maxPossibleLength' recursively with the next index (ind+1) and the same concatenation (s).
        - After backtracking, the function returns.
4. Finally, we call the 'maxLength' function with the initial input array 'arr'.

The time complexity of this solution depends on the **number of possible concatenations**, which can be exponential in the worst case. The space complexity is **O(1)**, as we use recursion and maintain a 'maxi' variable.