Class 11

Detailed Notes

Hiike

Problem 1: Subarray with 0 Sum

Link: Subarray with 0 Sum

Solution:

```
class Solution {
public:
    bool subArrayExists(int arr[], int n) {
        unordered_map<int, int> m;
        m[0]++;
        long long int sum = 0;
        for (int i = 0; i < n; i++) {
            sum += arr[i];
            if (m[sum] != 0) {
                return true;
            }
            m[sum]++;
        }
        return false;
    }
};
```

Explanation:

The solution uses a hash map to store the cumulative sum of elements as keys and their frequency as values. If the cumulative sum is repeated or becomes zero, it indicates the presence of a subarray with a sum of zero.

Why it works:

Using the hash map helps in efficiently checking for the presence of a subarray with zero sum by tracking the cumulative sums and their occurrences.

Time Complexity:

- Average case: O(n) due to the hash map operations.
- Worst case: O(n) if there are no collisions in the hash map.

Space Complexity:

- O(n) due to the hash map storing cumulative sums.

Problem 2: Consecutive Array Elements

Link: Consecutive Array Elements

Solution:

```
class Solution {
public:
    bool areConsecutives(long long arr[], int n) {
        unordered_map<long long, long long> m;
        long long smallest = INT_MAX;
        for (int i = 0; i < n; i++) {
            smallest = min(smallest, arr[i]);
            m[arr[i]]++;
            if (m[arr[i]] > 1) {
                return false;
            }
        }
        for (int i = smallest; i < smallest + n; i++) {
            if (m[i] == 0) {
                return false;
            }
        }
        return true;
    }
};
```

Explanation:

The solution uses a hash map to store the frequency of each element. It then checks if all numbers between the smallest element and the smallest element plus the array size are present in the hash map.

Why it works:

The hash map helps in efficiently tracking the presence of each element and ensures there are no duplicates.

Time Complexity:

- O(n) for inserting elements and checking their presence.

Space Complexity:

- O(n) for storing the elements in the hash map.

Problem 3: Subarray Sum Equals K

Link: Subarray Sum Equals K

Solution:

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<long long int, int> m;
        m[0] = 1;
        int ans = 0;
        long long sum = 0;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            ans += m[sum - k];
            m[sum]++;
        }
        return ans;
    }
};
```

Explanation:

The solution uses a hash map to store cumulative sums and their frequencies. For each element, it checks if sum - k exists in the map to find the subarrays with sum equal to k.

Why it works:

Using a hash map allows efficient lookup and insertion of cumulative sums, making it easy to find subarrays with the required sum.

Time Complexity:

- O(n) for iterating through the array and performing hash map operations.

Space Complexity:

- O(n) for storing cumulative sums in the hash map.

Problem 4: Anagram Check

Link: Anagram Check

Solution:

```
class Solution {
public:
    bool isAnagram(string a, string b) {
        if (a.size() != b.size()) {
            return false;
        }
        int freq[26] = {0};
        for (int i = 0; i < a.size(); i++) {
            freq[a[i] - 'a']++;
            freq[b[i] - 'a']--;
        }
        for (int i = 0; i < 26; i++) {
            if (freq[i] != 0) {
                return false;
            }
        }
        return true;
    }
};
```

Explanation:

The solution uses a frequency array to count the occurrences of each character in both strings. If the counts are identical, the strings are anagrams.

Why it works:

The frequency array allows efficient comparison of character counts between the two strings.

Time Complexity:

- O(n) for iterating through both strings (where n is the length of the strings).

Space Complexity:

- O(1) for the fixed-size frequency array.