Class 2

Detailed Notes

Problem 1: Corporate Flight Bookings

Link: Corporate Flight Bookings

Solution:

```cpp
class Solution {
public:
    vector<int> corpFlightBookings(vector<vector<int>>& bookings, int n) {
        vector<int> answer(n);
        for(int i = 0; i < bookings.size(); i++){
            int start = bookings[i][0], end = bookings[i][1], tickets =
bookings[i][2];
            answer[start-1] += tickets;
            if(end < n){
                answer[end] -= tickets;
            }
        }
        for(int i = 1; i < n; i++){
            answer[i] += answer[i-1];
        }
        return answer;
    }
};
```

Explanation:

The function implements a difference array approach to manage the bookings. For each booking, it adds the number of tickets to the start flight and subtracts the number from the flight following the end flight. This marks the start and end of a booking impact on the flights. The cumulative sum is then computed, resulting in the total number of bookings for each flight.

Why it works:

The difference array method efficiently updates ranges in an array, allowing to add or subtract a value over a range with only two operations, and later build the actual results with a single pass.

Time Complexity: O(n + b), where n is the number of flights, and b is the number of bookings.

Space Complexity: O(n), as it requires an array of size n to store the bookings impact.

Problem 2: Majority Element

Link: Majority Element

Solution:

```
class Solution{
public:
    int majorityElement(int a[], int size) {
        int candidate = a[0], count = 1;
        for(int i = 1; i < size; i++) {
            if(a[i] == candidate) {
                count++;
            } else {
                if(count == 0) {
                    candidate = a[i];
                    count = 1;
                } else {
                    count--;
                }
            }
        }
        count = 0;
        for(int i = 0; i < size; i++) {
            if(a[i] == candidate) {
                count++;
            }
        }
        if(count > size / 2) {
            return candidate;
        }
        return -1;
    }
};
```

Explanation:

The function finds the majority element (if it exists) in an array using Boyer-Moore Voting Algorithm. It first identifies a candidate through one pass where each element 'votes' for itself or against the current candidate. A second pass confirms whether this candidate indeed appears more than half the time.

Why it works:

The Boyer-Moore Voting Algorithm is efficient because it reduces the problem size by effectively pairing off different elements, knowing that the majority element will survive these reductions.

Time Complexity: O(n), since the array is traversed twice.

Space Complexity: O(1), as it uses a fixed amount of extra space.

Problem 3: Rain Water Trapped

Link: Rain Water Trapped

Solution:

```
class Solution {
public:
    int trap(vector<int>& height) {
        int tallest_height = height[0], tallest_index = 0;
        for(int i = 0; i < height.size(); i++){
            if(tallest_height < height[i]){
                tallest_height = height[i];
                tallest_index = i;
            }
        }
        int ans = 0, tallest_left = 0, tallest_right = 0;
        for(int i = height.size() - 1; i > tallest_index; i--){
            tallest_right = max(tallest_right, height[i]);
            ans += tallest_right - height[i];
        }
        for(int i = 0; i < tallest_index; i++){
            tallest_left = max(tallest_left, height[i]);
            ans += tallest_left - height[i];
        }
        return ans;
    }
};
```

Explanation:

This function calculates the amount of rainwater that can be trapped after raining. It first finds the tallest bar and then calculates trapped water on either side using the tallest bars seen from left to right and right to left.

Why it works:

Water trapping depends on the smallest height on either side of a bar, ensuring that each section only needs to consider one direction from the highest bar simplifies calculations.

Time Complexity: O(n), traverses the height array three times.

Space Complexity: O(1), only uses a few additional variables.

Problem 4: Spiral Order Matrix I

Link: [Spiral Order Matrix]

Solution:

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> ans;
        int upper = 0, lower = matrix.size() - 1, right = matrix[0].size()
- 1, left = 0;
        int direction = 0; // 0: left to right, 1: top to bottom, 2: right
to left, 3: bottom to top
        while(upper <= lower && right >= left) {
            if(direction == 0) {
                for(int i = left; i <= right; i++) {
                    ans.push_back(matrix[upper][i]);
                }
                upper++;
            } else if(direction == 1) {
                for(int i = upper; i <= lower; i++) {
                    ans.push_back(matrix[i][right]);
                }
                right--;
            } else if(direction == 2) {
                for(int i = right; i >= left; i--) {
                    ans.push_back(matrix[lower][i]);
                }
                lower--;
            } else {
                for(int i = lower; i >= upper; i--) {
                    ans.push_back(matrix[i][left]);
                }
                left++;
            }
            direction = (direction + 1) % 4;
        }
        return ans;
    }
};
```

Explanation:

The function generates a spiral order of a matrix by traversing its boundaries and gradually narrowing down the area to be traversed. It uses a direction variable to control the traversal direction, cycling through four possible directions.

Why it works:

By methodically adjusting boundaries and direction, the algorithm ensures every element is visited in the required spiral order without revisiting any element.

Time Complexity: O(m * n), where m is the number of rows, and n is the number of columns in the matrix, as each element is processed once.

Space Complexity: O(1), aside from the output list, the space used does not depend on the size of the matrix.