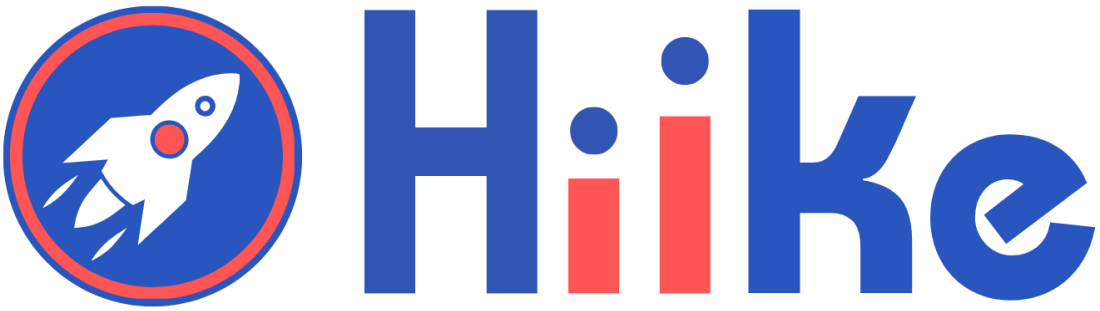
Class 4

Detailed Notes



Problem 1: Nth Catalan Number

Link: Nth Catalan Number

Solution:

```
class Solution {
public:
    int findCatalan(int n) {
        int arr[n+1] = {0};
        int m = 1000000007; // Modulo for large numbers
        arr[0] = arr[1] = 1; // Base cases
        for (int i = 2; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                arr[i] += ((long long)arr[j] * arr[i - j - 1]) % m;
                arr[i] %= m;
            }
        }
        return arr[n];
    }
};
```

Explanation:

The function calculates the nth Catalan number using dynamic programming. Catalan numbers are a sequence of natural numbers that have found applications in various combinatorial problems, often involving recursively defined objects. The solution involves calculating each Catalan number based on the sum of products of previous Catalan numbers.

Why it works:

The formula for Catalan numbers is a recursive one, where each number is computed based on the previous numbers in the sequence. This approach ensures all previous values are reused efficiently.

Time Complexity: O(n^2), as the calculation of each number involves a loop within a loop.

Space Complexity: O(n), due to the storage requirements for the dynamic programming array.

Problem 2: Count Pairs in Array Divisible by K

Link: [Count Pairs]

Solution:

```
class Solution {
public:
    long long countKdivPairs(int A[], int n, int K) {
        vector<int> vec(K, 0);
        for (int i = 0; i < n; i++) {
            vec[A[i] % K]++;
        }
        long long ans = 0;
        ans += ((long long)vec[0] * (vec[0] - 1)) / 2; // Pairs where both
numbers are divisible by K
        if (K % 2 == 0) { // Special case for even K
            for (int i = 1; i < K / 2; i++) {
                ans += vec[K - i] * vec[i];
            }
            ans += ((long long)vec[K / 2] * (vec[K / 2] - 1)) / 2;
        } else {
            for (int i = 1; i <= K / 2; i++) {
                ans += vec[K - i] * vec[i];
            }
        }
        return ans;
    }
};
```

Explanation:

The function counts pairs of integers in an array such that their sum is divisible by K. It uses a frequency array to count occurrences of each modulo K result. The pairs are then counted by matching complementary mod results that sum up to K.

Why it works:

The solution efficiently pairs elements by their modulo with K, ensuring that their sum is divisible by K without directly iterating over all pairs, thus reducing time complexity.

Time Complexity: O(n + K), where n is the size of the array and K for handling the modulo results.

Space Complexity: O(K), for storing the frequency of each remainder.

Problem 3: Diffk

Link: [Diffk on InterviewBit]

Solution:

```
int Solution::diffPossible(vector<int> &A, int B) {
    int p1 = A.size() - 1, p2 = A.size() - 2;
    while (p2 >= 0) {
        if (A[p1] - A[p2] == B) {
            return true;
        } else if (A[p1] - A[p2] > B) {
            p1--;
            if (p2 == p1) {
                p2--;
            }
        } else {
            p2--;
        }
    }
    return 0;
}
```

Explanation:

The function checks if there are two indices i and j in the array A such that A[i] - A[j] = B and i ≠ j. The approach involves two pointers moving from the end of the array, comparing differences.

Why it works:

This method efficiently finds the required pair by adjusting the pointers based on the difference comparison, reducing unnecessary comparisons.

Time Complexity: O(n), as each element is considered at most twice.

Space Complexity: O(1), using two pointers with no additional data structures.

Problem 4: Minimize the Absolute Difference

Link: [Minimize the Absolute Difference]

Solution:

```
int Solution::solve(vector<int> &A, vector<int> &B, vector<int> &C) {
    int p1 = 0, p2 = 0, p3 = 0;
    int ans = INT_MAX;
    A.push_back(INT_MAX);
    B.push_back(INT_MAX);
    C.push_back(INT_MAX);
    while (p1 < A.size() || p2 < B.size() || p3 < C.size()) {
        int a = A[p1], b = B[p2], c = C[p3];
        ans = min(ans, abs(max(max(a, b), c) - min(min(a, b), c)));
        if (a == INT_MAX && b == INT_MAX && c == INT_MAX) break;
        if (a <= b && a <= c) {
            p1++;
        } else if (b <= a && b <= c) {
            p2++;
        } else {
            p3++;
        }
    }
    return ans;
}
```

Explanation:

This solution aims to minimize the absolute difference between the maximum and minimum values taken from three arrays. It progresses through each array using pointers, selecting the smallest next element available among the current pointers to increment. This ensures that the difference is checked in a controlled manner, reducing the absolute difference progressively.

Why it works:

By always moving the pointer that points to the smallest value, the range is minimized iteratively, ensuring that the minimum possible difference is explored among combinations.

Time Complexity: O(n + m + o), where n, m, and o are the lengths of arrays A, B, and C respectively.

Space Complexity: O(1), no significant additional space is used aside from minor variables.

Problem 5: Smallest Sequence With Given Primes

Link: [Smallest Sequence With Given Primes]

Solution:

```
vector<int> Solution::solve(int A, int B, int C, int D) {
    vector<int> vec;
    vec.push_back(1);
    int p1 = 0, p2 = 0, p3 = 0;
    while (vec.size() <= D) {
        int min_num = min(min(A * vec[p1], B * vec[p2]), C * vec[p3]);
        vec.push_back(min_num);
        if (A * vec[p1] == min_num) p1++;
        if (B * vec[p2] == min_num) p2++;
        if (C * vec[p3] == min_num) p3++;
    }
    vec.erase(vec.begin());
    return vec;
}
```

Explanation:

This function generates the first D numbers in an increasing sequence where each number is either a multiple of A, B, or C. The approach uses a merging technique of three arithmetic progressions.

Why it works:

The approach guarantees that each subsequent number in the sequence is the smallest possible multiple of the three given numbers, ensuring no smaller number is missed.

Time Complexity: O(D), where D is the number of sequence elements to be produced.

Space Complexity: O(D), as it needs to store the sequence.

Problem 6: Container With Most Water

Link: [Container With Most Water]

Solution:

```
int Solution::maxArea(vector<int> &A) {
    int p1 = 0, p2 = A.size() - 1, ans = 0;
    while (p1 < p2) {
        ans = max(ans, min(A[p1], A[p2]) * (p2 - p1));
        if (A[p1] <= A[p2]) {
            p1++;
        } else {
            p2--;
        }
    }
    return ans;
}
```

Explanation:

The solution finds the maximum area formed between vertical lines (represented by array values) using a two-pointer approach. The pointers start at the ends of the array and move towards each other, always moving the pointer pointing to the shorter line to try and find a taller one which might increase the area.

Why it works:

This approach efficiently finds the maximum possible area by exploring potentially larger containers quickly, discarding pairs that cannot possibly exceed the current maximum due to the limitation of the shorter line.

Time Complexity: O(n), where n is the number of elements in array A.

Space Complexity: O(1), using two pointers without extra space.