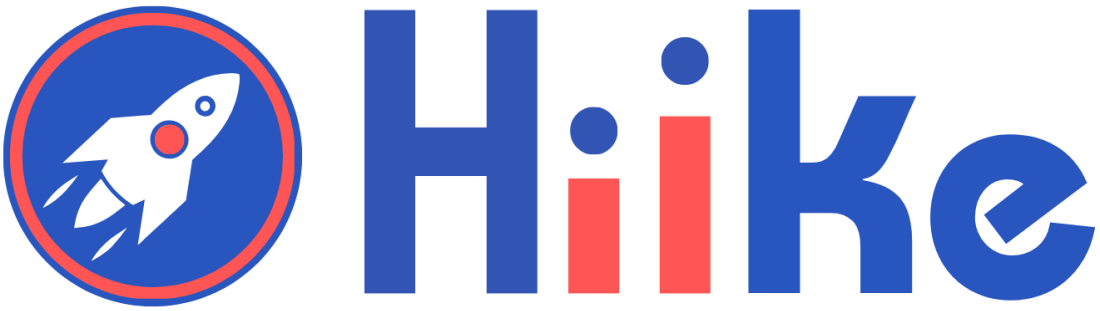
Class 10

Detailed Notes



Problem: Check if Two Strings are Anagram

Link: [Anagram]

Solution:

```
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 checks if two strings are anagrams by counting the frequency of each character in both strings and ensuring these frequencies match. The approach uses a single array to increment and decrement counts for characters from each string, respectively:

Why it works:

This method is effective because if the two strings are anagrams, all entries in the logfreqlog array will be zero after processing both strings. Any non-zero value indicates a mismatch in character frequency.

Time Complexity: O(n), where n is the length of the strings.

Space Complexity: O(1), since the space used does not scale with the size of the input, only a fixed-size array of 26 integers is used.

Problem: K Closest Points to Origin

Link: [K Closest Points to Origin]

Solution:

```
vector<vector<int>> kClosest(vector<vector<int>>& points, int k) {
    vector<pair<double, pair<int, int>>> temp;
    for(int i = 0; i < points.size(); i++) {
        temp.push_back(make_pair(sqrt(pow(points[i][0], 2) +
pow(points[i][1], 2)), make_pair(points[i][0], points[i][1])));
    }
    sort(temp.begin(), temp.end());
    vector<vector<int>> ans;
    for(int i = 0; i < k; i++) {
        vector<int> v = {temp[i].second.first, temp[i].second.second};
        ans.push_back(v);
    }
    return ans;
}
```

Explanation:

The solution computes the Euclidean distance of each point from the origin, sorts these points by distance, and selects the closest k

Why it works:
Sorting the points by distance ensures that the first k elements in the sorted list are the closest to the origin, providing a straightforward method to obtain the required points.

Time Complexity: O(n log n), due to the sorting step.

Space Complexity: O(n), for storing the distances and coordinates temporarily.

Problem: Sum of Subset Differences

Link: Sum of Subset Differences

Solution:

```
int sumDiff(int S[], int n) {
    int ans = 0;
    for(int i = 0; i < n; i++) {
        ans += (S[i] * (pow(2, i) - pow(2, n - i - 1)));
    }
    return ans;
}
```

Explanation:

The solution calculates the sum of all differences between subsets formed by n elements. It uses mathematical properties to compute contributions of each element to the sum based on its position

Why it works:

Each element's contribution to the total difference is determined by its position, exploiting combinatorial properties where each element appears in 2^{i} subsets as the last element, and is excluded from 2^{n-i-1} subsets formed by the remaining elements.

Time Complexity: O(n), for iterating through the array.

Space Complexity: O(1), as only a few additional variables are used.

Problem: Chocolate Distribution Problem

Link: [Chocolate Distribution Problem]

Solution:

```cpp
long long findMinDiff(vector<long long> a, long long n, long long m) {
    sort(a.begin(), a.end());
    long long ans = INT_MAX;
    for(int i = 0; (i + m - 1) < n; i++) {
        ans = min(ans, a[i + m - 1] - a[i]);
    }
    return ans;
}
```

Explanation:

This problem is solved by sorting the array of chocolates and then finding the minimum difference between the maximum and minimum number of chocolates in any subgroup of size m

Why it works:
Sorting the array allows the subgroups to be formed by consecutive elements, ensuring that the smallest difference between the maximum and minimum chocolates in a subgroup is found efficiently.

Time Complexity: O(n log n), mainly due to the sorting step.

Space Complexity: O(1), since the algorithm modifies the input array in place and uses only a few extra variables.

Problem 1: Counting Inversions

Link: [Counting Inversions]

Solution:

```
long long merge(vector<int> &arr,long long N,long long start,long long mid
, long long end){
    vector<long long> vec;
    int p1=start;
    int p2=mid+1;
    long long count=0;
    while(p1<=mid && p2<=end){
        if(arr[p1]<=arr[p2]){
            vec.push_back(arr[p1]);
            p1++;
        }
        else{
            count+=mid-p1+1;
            vec.push_back(arr[p2]);
            p2++;
        }
    }
    while(p1<=mid){
        vec.push_back(arr[p1]);
        p1++;
    }
    while(p2<=end){
        vec.push_back(arr[p2]);
        p2++;
    }
    for(int i=start;i<=end;i++){
        arr[i] = vec[i-start];
    }
    return count;
}

long long findinv(vector<int> &arr,long long N,long long start, long long
end){
    long long total=0;
```

```
    if(start<end){
        long long int mid = start+(end-start)/2;
        long long int t1 = findinv(arr,N,start,mid);
        long long int t2 = findinv(arr,N,mid+1,end);
        long long int t3 = merge(arr,N,start,mid,end);
        total+=(t1+t2+t3);
    }
    return total;
}

int Solution::countInversions(vector<int> &A) {
    return (int)findinv(A,A.size(),0,A.size()-1);
}
```

Explanation:

The solution uses a modified merge sort algorithm to count inversions in an array. An inversion is a pair of elements where the earlier element is greater than the later one. The findinv function merges two sorted halves of the array and counts how many inversions are there. The findinv function recursively splits the array and sums up the inversions from the left half, the right half, and the cross inversions between the two halves.

Why it Works:
This works because merge sort inherently splits the array into smaller subarrays, making it easier to count inversions in smaller chunks and then merge the results, maintaining the inversion count.

Time Complexity: O(n log n) - because the array is being divided into halves (log n) and the merge operation takes linear time (n).

Space Complexity: O(n) - due to the temporary array used during the merge process.

Problem 2: Counting Reverse Pairs

Link: [Reverse Pairs]

Solution:

```
class Solution {
public:
    int crossinv(vector<int> &num,int start,int mid,int end){
        vector<int> vec;
        int p1=start,p2=mid+1;
        int count=0;
        while(p1<=mid && p2<=end){
            if(num[p1]<=(long)2*num[p2]){
                p1++;
            }
            else{
                count+=mid-p1+1;
                p2++;
            }
        }
        p1=start;p2=mid+1;
        while(p1<=mid && p2<=end){
            if(num[p1]<=num[p2]){
                vec.push_back(num[p1++]);
            }
            else{
                vec.push_back(num[p2++]);
            }
        }
        while(p1<=mid){
            vec.push_back(num[p1++]);
        }
        while(p2<=end){
            vec.push_back(num[p2++]);
        }
        for(int i=start;i<=end;i++){
            num[i] = vec[i-start];
        }
        return count;
```

```
    }

    int doubleinvcount(vector<int> &num,int start,int end){
        int count=0;
        if(start<end){
            int mid = start + (end-start)/2;
            count+= doubleinvcount(num,start,mid);
            count+= doubleinvcount(num,mid+1,end);
            count+= crossinv(num,start,mid,end);
        }
        return count;
    }

    int reversePairs(vector<int> &nums) {
        int ans= doubleinvcount(nums,0,nums.size()-1);
        return ans;
    }
};
```

Explanation:

This solution also uses a modified merge sort to count reverse pairs, where a reverse pair is defined as (i, j) such that nums[i] > 2 * nums[j]. The crossinv function counts cross inversions between the two halves of the array. The logdoubleinvcountlog function recursively splits the array and sums up the counts from the left half, the right half, and the cross inversions.

Why it Works:

By breaking the problem into smaller subproblems, it efficiently counts reverse pairs through divide-and-conquer, ensuring that all pairs are counted correctly as the array is merged back together.

Time Complexity: O(n log n) - because of the merge sort's divide and conquer strategy.

Space Complexity: O(n) - due to the auxiliary array used in the merge process.

Problem 3: Sort Colors

Link: Sort Colors

Solution:

```
class Solution {
public:
    void sortColors(vector<int>& nums) {
        int s=-1,e=nums.size(),p1=0;
        while(p1<e){
            if(nums[p1]==0){
                s++;
                int temp = nums[s];
                nums[s]=nums[p1];
                nums[p1] = temp;
                p1--;
            }
            else if(nums[p1]==2){
                e--;
                int temp = nums[e];
                nums[e]=nums[p1];
                nums[p1] = temp;
                p1--;
            }
            p1++;
            if(p1<=s){
                p1=s+1;
            }
        }
    }
};
```

Explanation:

The solution sorts an array with three distinct values (0, 1, and 2) using a three-way partitioning algorithm similar to the Dutch national flag problem. It uses three pointers: s for tracking zeros, e for tracking twos, and p1 for current index traversal.

Why it Works:

This approach works because it efficiently segregates the three different values by maintaining and updating the pointers as it traverses the array. This ensures that the array is sorted in a single pass.

Time Complexity: O(n) - because the algorithm processes each element of the array once.

Space Complexity: O(1) - as it uses constant extra space.