TECHNICAL TRAINING DSA - CODING PRACTICEP ROBLEMS

Name: Rishi Kumar S
Dept: CSBS
Date: 19-11-2024

**Problem 1: Minimum Path Sum**
Given a m x n grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.
Note: You can only move either down or right at any point in time.

**CODE:**

```
nt minPathSum(vector<vector<int>>& arr) {
int m=arr.size();
int n=arr[0].size();
vector<vector<int>> dp(m,vector<int>(n));
dp[0][0]=arr[0][0];
for(int i=1;i<n;i++){
dp[0][i]=arr[0][i]+dp[0][i-1];
}
for(int i=1;i<m;i++){
dp[i][0]=arr[i][0]+dp[i-1][0];
}

for(int i=1;i<m;i++){
for(int j=1;j<n;j++){
dp[i][j]=arr[i][j]+min(dp[i][j-1],dp[i-1][j]);
}
}

// for(auto x:dp){
// for(int y:x){
// cout<<y<<" ";
// }
// cout<<endl;
// }
return dp[m-1][n-1];
}
```

**Time Complexity:** O(n^2)
**Space Complexity:** O(n^2)

**Question 2: Validate binary search tree**

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

A valid BST is defined as follows:

The left
subtree
of a node contains only nodes with keys less than the node's key.

The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left and right subtrees must also be binary search trees.

**CODE:**

```
bool finder(TreeNode* root,long long min,long long max){
if(!root) return true;
if(root->val<=min || root->val>=max) return false;
return finder(root->left,min,root->val) && finder(root->right,root->val,max);
}

bool isValidBST(TreeNode* root) {
return finder(root,LONG_MIN,LONG_MAX);
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(n) //Recursion Stack

**Question 3: Next Permutation**

A permutation of an array of integers is an arrangement of its members into a sequence or linear order.

For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The next permutation of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the next permutation of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

For example, the next permutation of arr = [1,2,3] is [1,3,2].

Similarly, the next permutation of arr = [2,3,1] is [3,1,2].

While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.
Given an array of integers nums, find the next permutation of nums.
The replacement must be in place and use only constant extra memory.

**Code:**

```
void nextPermutation(vector<int>& nums) {
int n=nums.size();
if(n<2) return;
int i=n-2;
while(i>=0 && nums[i]>=nums[i+1]) i--;
if(i>=0){
int j=n-1;
while(j>=0 && nums[j]<=nums[i]){
j--;
}
swap(nums[i],nums[j]);
}
reverse(nums.begin()+i+1,nums.end());
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

**Question 4:Longest Substring Without Repeating Characters**

Given a string s, find the length of the longest substring without repeating characters.

**CODE:**

```
#include <iostream>
#include <unordered_set>
#include <string>
#include <algorithm>

class Solution {
public:
    int lengthOfLongestSubstring(std::string s) {
        int start = 0;
        int count = 0;
        for (int i = 1; i <= s.length(); ++i) {
```

```
            std::unordered_set<char> uniqueChars(s.begin() + start, s.begin()
+ i);

            if (uniqueChars.size() == i - start) {
                count = std::max(count, i - start); // Update count if we found a
longer substring
            } else {
                start++;
            }
        }
        return count;
    }
};

int main() {
    Solution sol;
    std::string s = "abcabcbb";
    std::cout << "Length of the longest substring without repeating
characters: "
              << sol.lengthOfLongestSubstring(s) << std::endl;
    return 0;
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(n)

**Question 5: Remove linked list elements**

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return the new head.

**CODE:**

```
ListNode* removeElements(ListNode* head, int val) {
ListNode* p1=head;
ListNode* newhead=new ListNode(0);
ListNode* newtemp=newhead;
while(p1!=nullptr){
if(p1->val!=val){
newtemp->next=new ListNode(p1->val);
newtemp=newtemp->next;
}
p1=p1->next;
}
```

```
return newhead->next;
}
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)

**Question 6: Palindrome linked list**

Given the head of a singly linked list, return true if it is a palindrome or false otherwise.

**CODE:**

```
ListNode* rev(ListNode* head){
ListNode* dummy=nullptr;
ListNode* newnode=nullptr;
while(head!=nullptr){
newnode=head->next;
head->next=dummy;
dummy=head;
head=newnode;
}
return dummy;
```

**Time Complexity:** O(n)
**Space Complexity:** O(1)