19/11/24

1)Minimum path sum

class Solution {

public:

    int solver(int i,int j,vector<vector<int>> &dp,vector<vector<int>> &grid,int tot,int m,int n){

        if(i>=m || j>=n){

            return 0;

        }if(dp[i][j]!=-1){

            return dp[i][j];

        }int left = solver

    }

    int minPathSum(vector<vector<int>>& grid) {

        int m = grid.size();

        int n = grid[0].size();

        vector<vector<int>> dp(m,vector<int> (n,-1));

        return solver()

    }

};

2)validate binary search tree

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

bool solver(TreeNode\* node, TreeNode\* &prev){

if (node == NULL) return true;

if (!solver(node->left, prev)) return false;

if (prev != NULL && prev->val >= node->val) return false;

prev = node;

return solver(node->right, prev);

}

bool isValidBST(TreeNode\* root) {

TreeNode\* prev = NULL;

return solver(root,prev);

}

}

3)word ladder

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

Set<String> set = new HashSet<>(wordList);

if(!set.contains(endWord)) return 0;

Queue<String> queue = new LinkedList<>();

queue.add(beginWord);

Set<String> visited = new HashSet<>();

queue.add(beginWord);

int changes = 1;

while(!queue.isEmpty()){

int size = queue.size();

for(int i = 0; i < size; i++){

String word = queue.poll();

if(word.equals(endWord)) return changes;

for(int j = 0; j < word.length(); j++){

for(int k = 'a'; k <= 'z'; k++){

char arr[] = word.toCharArray();

arr[j] = (char) k;

String str = new String(arr);

if(set.contains(str) && !visited.contains(str)){

queue.add(str);

visited.add(str);

}

}

}

}

++changes;

}

return 0;

}

}

4)Word ladder 2

class Solution {

public:

bool able(string s,string t){

int c=0;

for(int i=0;i<s.length();i++)

c+=(s[i]!=t[i]);

return c==1;

}

void bfs(vector<vector<int>> &g,vector<int> parent[],int n,int start,int end){

vector <int> dist(n,1005);

queue <int> q;

q.push(start);

parent[start]={-1};

dist[start]=0;

while(!q.empty()){

int x=q.front();

q.pop();

for(int u:g[x]){

if(dist[u]>dist[x]+1){

dist[u]=dist[x]+1;

q.push(u);

parent[u].clear();

parent[u].push\_back(x);

}

else if(dist[u]==dist[x]+1)

parent[u].push\_back(x);

}

}

}

void shortestPaths(vector<vector<int>> &Paths, vector<int> &path, vector<int> parent[],int node){

if(node==-1){

// as parent of start was -1, we've completed the backtrack

Paths.push\_back(path);

return ;

}

for(auto u:parent[node]){

path.push\_back(u);

shortestPaths(Paths,path,parent,u);

path.pop\_back();

}

}

vector<vector<string>> findLadders(string beginWord, string endWord, vector<string>& wordList) {

// start and end are indices of beginWord and endWord

int n=wordList.size(),start=-1,end=-1;

vector<vector<string>> ANS;

for(int i=0;i<n;i++){

if(wordList[i]==beginWord)

start=i;

if(wordList[i]==endWord)

end=i;

}

// if endWord doesn't exist, return empty list

if(end==-1)

return ANS;

// if beginWord doesn't exist, add it in start of WordList

if(start==-1){

wordList.emplace(wordList.begin(),beginWord);

start=0;

end++;

n++;

}

// for each word, we're making adjency list of neighbour words (words that can be made with one letter change)

// Paths will store all the shortest paths (formed later by backtracking)

vector<vector<int>> g(n,vector<int>()),Paths;

// storing possible parents for each word (to backtrack later), path is the current sequence (while backtracking)

vector<int> parent[n],path;

// creating adjency list for each pair of words in the wordList (including beginword)

for(int i=0;i<n-1;i++)

for(int j=i+1;j<n;j++)

if(able(wordList[i],wordList[j])){

g[i].push\_back(j);

g[j].push\_back(i);

}

bfs(g,parent,n,start,end);

// backtracking to make shortestpaths

shortestPaths(Paths,path,parent,end);

for(auto u:Paths){

vector <string> now;

for(int i=0;i<u.size()-1;i++)

now.push\_back(wordList[u[i]]);

reverse(now.begin(),now.end());

now.push\_back(wordList[end]);

ANS.push\_back(now);

}

return ANS;

}

};

5) Course schedule

class Solution {

public:

bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {

int counter = 0;

if (numCourses <= 0) {

return true;

}

// Initialize inDegree and graph

vector<int> inDegree(numCourses, 0);

vector<vector<int>> graph(numCourses);

// Build the graph and update inDegree for each node

for (const auto& edge : prerequisites) {

int parent = edge[1];

int child = edge[0];

graph[parent].push\_back(child);

inDegree[child]++;

}

// Initialize the queue with courses having no prerequisites (inDegree =

// 0)

queue<int> sources;

for (int i = 0; i < numCourses; i++) {

if (inDegree[i] == 0) {

sources.push(i);

}

}

// Process nodes with no prerequisites

while (!sources.empty()) {

int course = sources.front(); // dequeue

sources.pop();

counter++;

// Process all the children of the current course

for (int child : graph[course]) {

inDegree[child]--;

if (inDegree[child] == 0) {

sources.push(child); // enqueue child if inDegree becomes 0

}

}

}

// If we processed all courses, return true

return counter == numCourses;

}

};

6)Next permutation

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int n=nums.size();

int i,j;

for(i=n-2;i>=0;i--){

if(nums[i]<nums[i+1]){

break;

}

}if(i>=0){

for(j=n-1;j>i;j--){

if(nums[i]<nums[j]){

swap(nums[i],nums[j]);

break;

}

}

}reverse(nums.begin()+i+1,nums.end());

}

};

7)Spiral matrix

class Solution {

public:

vector<int> spiralOrder(vector<vector<int>>& mat) {

int ilim = mat.size();

int jlim = mat[0].size();

int cou = 0;

vector<int> ans;

while(cou < ilim - cou && cou < jlim - cou) {

for(int j = cou; j < jlim - cou; j++) {

ans.push\_back(mat[cou][j]);

}

for(int i = cou + 1; i < ilim - cou; i++) {

ans.push\_back(mat[i][jlim - cou - 1]);

}

if(ilim - cou - 1 > cou) {

for(int j = jlim - cou - 2; j >= cou; j--) {

ans.push\_back(mat[ilim - cou - 1][j]);

}

}

if(jlim - cou - 1 > cou) {

for(int i = ilim - cou - 2; i > cou; i--) {

ans.push\_back(mat[i][cou]);

}

}

cou++;

}

return ans;

}

};

8)Longest substring without rep chars

class Solution {

public:

int lengthOfLongestSubstring(string s) {

int left = 0;

int right = 0;

int ans = 0;

map<char,int> mp;

while(right<s.size()){

if(mp[s[right]]!=0){

left = max(left,mp[s[right]]);

}mp[s[right]] = right+1;

// cout<<left<<" "<<right<<" ";

ans = max(ans,right-left+1);

// cout<<ans<<endl;

right++;

}

return ans;

}

};

9) remove linked list elements

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* removeElements(ListNode\* head, int val) {

ListNode \*temp = new ListNode(0);

temp->next = head;

ListNode \*curr = temp;

while(curr->next != NULL ){

if(curr->next->val == val) curr->next = curr->next->next;

else curr = curr->next;

}

return temp->next;

}

};

10)Palndromic linked list

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

bool isPalindrome(ListNode\* head) {

stack<int> st;

ListNode\* temp = head;

while(temp!=NULL){

st.push(temp->val);

temp = temp->next;

}temp = head;

while(temp!=NULL){

if(st.top()!=temp->val){

return false;

}st.pop();

temp = temp->next;

}return true;

}

};