DSA MOST LIKELY QUESTIONS

1-Set Matrix zeros:

***Problem Statement:****Given a matrix if an element in the matrix is 0 then you will have to set its entire column and row to 0 and then return the matrix.*

void setZeros(vector<vector<int>> &m)

{

    map<int, int> row, col;

    for (int i = 0; i < m.size(); i++)

    {

        for (int j = 0; j < m[0].size(); j++)

        {

            if (m[i][j] == 0)

            {

                row[i]++;

                col[j]++;

            }

        }

    }

    for (auto x : row)

    {

        for (int i = 0; i < m[0].size(); i++)

        {

            m[x.first][i] = 0;

        }

    }

    for (auto x : col)

    {

        for (int i = 0; i < m.size(); i++)

        {

            m[i][x.first] = 0;

        }

    }

}

2-Pascals Triangle:

***Problem Statement:****Given an integer****N****, return the first****N****rows of Pascal’s triangle.*

vector<vector<int>> generate(int numRows)

{

    vector<vector<int>> ans(numRows);

    for (int i = 0; i < numRows; i++)

    {

        ans[i].resize(i + 1);

        ans[i][0] = 1;

        ans[i][i] = 1;

        for (int j = 1; j < i; j++)

        {

            ans[i][j] = ans[i - 1][j - 1] + ans[i - 1][j];

        }

    }

    return ans;

}

3-Next Permutation

***Problem Statement:****Given an array Arr[] of integers, rearrange the numbers of the given array into the lexicographically next greater permutation of numbers.*

*If such an arrangement is not possible, it must rearrange it as the lowest possible order (i.e., sorted in ascending order).*

void nextPermutation(vector<int> &nums)

{

    int i = nums.size() - 2;

    while (i >= 0 && nums[i + 1] <= nums[i])

    {

        i--;

    }

    if (i >= 0)

    {

        int j = nums.size() - 1;

        while (nums[j] <= nums[i])

        {

            j--;

        }

        swap(nums[i], nums[j]);

    }

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

}

4-Kadane’s Algorithm

***Problem Statement****: Given an integer array arr, find the contiguous sub-array (containing at least one number) which  
has the largest sum and return its sum and print the sub-array.*

int Kandane(vector<int> &nums)

{

    int ans = 0;

    int cur = 0;

    int mx = INT\_MIN;

    if (nums.size() == 1)

        return nums[0];

    for (auto x : nums)

    {

        cur += x;

        mx = max(mx, cur);

        if (cur > 0)

            ans = max(ans, cur);

        else

            cur = 0;

    }

    if (ans == 0)

        return mx;

    else

        return ans;

}

5-Sort an Array of 0’s 1’s & 2’s

***Problem Statement:****Given an array consisting of only 0s, 1s and 2s. Write a program to in-place sort the array without using inbuilt sort functions. ( Expected: Single pass-O(N) and constant space)*

void sortColors(vector<int> &nums)

{

    int i = 0;

    for (int v = 0; v <= 1; v++)

    {

        for (int j = 0; j < nums.size(); j++)

        {

            if (nums[j] == v)

            {

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

                i++;

            }

        }

    }

}

*6-Stock buy and sell problem*

**Problem Statement:** You are given an array of prices where prices[i] is the price of a given stock on an ith day. You want to maximize your profit by choosing a single day to buy one stock and choosing a different day in the future to sell that stock. Return the maximum profit you can achieve from this transaction. If you cannot achieve any profit, return 0.

int maxProfit(vector<int> &p)

{

    int curr = INT\_MAX;

    int ans = 0;

    for (int i = 0; i < p.size(); i++)

    {

        curr = min(curr, p[i]);

        int pro = p[i] - curr;

        ans = max(ans, pro);

    }

    return ans;

}

*7-Merge overlapping intervals*

***Problem Statement:****Given an array of intervals, merge all the overlapping intervals and return an array of non-overlapping intervals.*

vector<vector<int>> merge(vector<vector<int>> &inter)

{

    sort(inter.begin(), inter.end());

    vector<vector<int>> ans;

    ans.push\_back({inter[0][0], inter[0][1]});

    for (int i = 1; i < inter.size(); i++)

    {

        vector<int> bc = ans.back();

        if (inter[i][0] >= bc[0] && inter[i][0] <= bc[1])

        {

            ans.pop\_back();

            vector<int> temp = {min(bc[0], inter[i][0]), max(bc[1], inter[i][1])};

            ans.push\_back(temp);

        }

        else

        {

            ans.push\_back({inter[i][0], inter[i][1]});

        }

    }

    return ans;

}

*8-merge two sorted array without extra space*

***Problem statement:****Given two sorted arrays****arr1[]****and****arr2[]****of**sizes****n****and****m****in non-decreasing order. Merge them in sorted order. Modify arr1 so that it contains the first N elements and modify arr2 so that it contains the last M elements.*

void merge(vector<int>& nums1, int n, vector<int>& nums2, int m) {

int i=n-1,j=0;

while(i>=0 && j<m){

if(nums1[i]>nums2[j])

swap(nums1[i],nums2[j]),i--,j++;

else

break;

}

sort(nums1.begin(),nums1.begin()+n);

sort(nums2.begin(),nums2.end());

i=n;

j=0;

while(i<m+n)

{

nums1[i]=nums2[j];

i++;

j++;

}

}

*9-Find Duplicate*

***Problem Statement:****Given an array of N + 1 size, where each element is between 1 and N. Assuming there is only one duplicate number, your task is to find the duplicate number.*

int findDuplicate(vector<int> &nums)

{

    int fast = nums[0];

    int slow = nums[0];

    do

    {

        fast = nums[nums[fast]];

        slow = nums[slow];

    } while (slow != fast);

    fast = nums[0];

    while (slow != fast)

    {

        fast = nums[fast];

        slow = nums[slow];

    }

    return slow;

}

*10-Rotate Image by 90deg*

***Problem Statement:****Given a matrix, your task is to rotate the matrix 90 degrees clockwise*.

void rotate(vector<vector<int>> &mat)

{

    for (int i = 0; i < mat.size(); i++)

    {

        for (int j = i + 1; j < mat[0].size(); j++)

        {

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

        }

    }

    for (int i = 0; i < mat.size(); i++)

    {

        reverse(mat[i].begin(), mat[i].end());

    }

}

*11-Repeating and Missing Element*

***Problem Statement:****You are given a read-only array of N integers with values also in the range [1, N] both inclusive. Each integer appears exactly once except A which appears twice and B which is missing. The task is to find the repeating and missing numbers A and B where A repeats twice and B is missing*.

vector<int> Solution::repeatedNumber(const vector<int> &A)

{

    long long int len = A.size();

    long long int S = (len \* (len + 1)) / 2;

    long long int P = (len \* (len + 1) \* (2 \* len + 1)) / 6;

    long long int missingNumber = 0, repeating = 0;

    for (int i = 0; i < A.size(); i++)

    {

        S -= (long long int)A[i];

        P -= (long long int)A[i] \* (long long int)A[i];

    }

    missingNumber = (S + P / S) / 2;

    repeating = missingNumber - S;

    vector<int> ans;

    ans.push\_back(repeating);

    ans.push\_back(missingNumber);

    return ans;

}

*12-Count Inversions in an Array*

***Problem Statement:****Given an array of N integers, count the inversion of the array (using*[*merge-sort*](https://takeuforward.org/data-structure/merge-sort-algorithm/)*).What is an inversion of an array? Definition: for all i & j < size of array, if i < j then you have to find pair (A[i],A[j]) such that A[j] < A[i].*

long long merge(vector<long long> &arr, vector<long long> &temp, int l, int mid, int r)

{

    int inver = 0;

    int i = l;

    int j = mid;

    int k = l;

    while (i <= mid - 1 && j <= r)

    {

        if (arr[i] < arr[j])

        {

            temp[k++] = arr[i++];

        }

        else

        {

            temp[k++] = arr[j++];

            inver += mid - i;

        }

    }

    while (i <= mid - 1)

        temp[k++] = arr[i++];

while (j <= r)

        temp[k++] = arr[j++];

    for (int i = l; i <= r; i++)

        arr[i] = temp[i];

    return inver;

}

long long mergeSort(vector<long long> &arr, vector<long long> &temp, int l, int r)

{

    int inver = 0;

    if (l < r)

    {

        int mid = l + (r - l) / 2;

        inver += mergeSort(arr, temp, l, mid);

        inver += mergeSort(arr, temp, mid + 1, r);

        inver += merge(arr, temp, l, mid + 1, r);

    }

    return inver;

}

long long getInversions(vector<long long> &arr, int n)

{

    vector<long long> temp(n);

    return mergeSort(arr, temp, 0, n - 1);

}

*13-Search in a 2-D sorted Array*

***Problem Statement:****Given an m\*n 2D matrix and an integer, write a program to find if the given integer exists in the matrix*

*Given matrix has the following properties:*

*1-Integers in each row are sorted from left to right and the first integer of each row is greater than the last integer of the previous row*

bool searchMatrix(vector<vector<int>> &mat, int t)

{

    int m = mat.size();

    int n = mat[0].size();

    int r = 0, c = n - 1;

    bool found = false;

    while (r < m && c >= 0)

    {

        if (mat[r][c] > t)

            c--;

        else if (mat[r][c] < t)

            r++;

        else

        {

            found = true;

            break;

        }

    }

    return found;

}

*14-Power(x,n)*

***Problem Statement:****Given a double x and integer n, calculate x raised to power n. Basically Implement pow(x, n).*

double bin(double a, long b)

{

    double ans = 1;

    if (b < 0)

    {

        b = -b;

        a = 1 / a;

    }

    while (b)

    {

        if (b & 1)

        {

            ans \*= a;

        }

        a = (a \* a);

        b >>= 1;

    }

    return ans;

}

double myPow(double x, int n)

{

    double ans = bin(x, n);

    return ans;

}

*15-Find the majority element that occurs more than n/2 times*

***Problem Statement:****Given an array of****N integers****, write a program to return an element that occurs more than****N/2****times in the given array. You may consider that such an element always exists in the array.*

int majorityElement(vector<int> &nums)

{

    // Moore’s Voting Algorithm

    int cnt = 0;

    int ele = 0;

    for (auto x : nums)

    {

        if (cnt == 0)

        {

            ele = x;

        }

        if (x == ele)

            cnt++;

        else

            cnt--;

    }

    return ele;

}

*16- Find the majority element that occurs more than n/3 times*

***Problem Statement:****Given an array of N integers. Find the elements that appear more than****N/3****times in the array. If no such element exists, return an empty vector.*

vector<int> majorityElement(vector<int> &nums)

{

    int c1 = 0;

    int c2 = 0;

    int n1 = -1;

    int n2 = -1;

    int n = nums.size();

    for (auto x : nums)

    {

        if (x == n1)

            c1++;

        else if (x == n2)

            c2++;

        else if (c1 == 0)

        {

            c1 = 1;

            n1 = x;

        }

        else if (c2 == 0)

        {

            c2 = 1;

            n2 = x;

        }

        else

        {

            c1--;

            c2--;

        }

    }

    c1 = 0;

    c2 = 0;

    for (auto x : nums)

    {

        if (x == n1)

            c1++;

        else if (x == n2)

            c2++;

    }

    vector<int> ans;

    if (c1 > n / 3)

        ans.push\_back(n1);

    if (c2 > n / 3)

        ans.push\_back(n2);

    return ans;

}

*17-Count Unique Paths*

***Problem Statement:****Given a matrix****m X n****, count paths from left-top to the right bottom of a matrix with the constraints that from each cell you can either only move to the rightward direction or the downward direction. The test cases are generated so that the answer will be less than or equal to 2 \* 109.*

int Paths(int i, int j, int n, int m, vector<vector<int>> &dp)

{

    if (i > n - 1 || j > m - 1)

        return 0;

    if (dp[i][j] != -1)

        return dp[i][j];

    if (i == n - 1 && j == m - 1)

        return 1;

    int ans = Paths(i + 1, j, n, m, dp) + Paths(i, j + 1, n, m, dp);

    return dp[i][j] = ans;

}

int uniquePaths(int n, int m)

{

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

    return Paths(0, 0, n, m, dp);

}

*18-Count Reverse Pairs*

***Problem Statement:****Given an array of numbers, you need to return the count of reverse pairs.****Reverse Pairs****are those pairs where i<j and arr[i]>2\*arr[j].*

int Merge(int l, int mid, int r, vector<int> &v, vector<int> &temp)

{

    int total = 0;

    int j = mid + 1;

    for (int i = l; i <= mid; i++)

    {

        while (j <= r && v[i] > 2LL \* v[j])

        {

            j++;

        }

        total += (j - (mid + 1));

    }

    int i = l;

    j = mid + 1;

    int k = l;

    while (i <= mid && j <= r)

    {

        if (v[i] <= v[j])

        {

            temp[k++] = v[i++];

        }

        else

        {

            temp[k++] = v[j++];

        }

    }

    while (i <= mid)

        temp[k++] = v[i++];

    while (j <= r)

        temp[k++] = v[j++];

    for (int i = l; i <= r; i++)

        v[i] = temp[i];

    return total;

}

int mergeSort(int l, int r, vector<int> &v, vector<int> &temp)

{

    int pairs = 0;

    if (l < r)

    {

        int mid = l + (r - l) / 2;

        pairs += mergeSort(l, mid, v, temp);

        pairs += mergeSort(mid + 1, r, v, temp);

        pairs += Merge(l, mid, r, v, temp);

    }

    return pairs;

}

int ReversePairs(vector<int> &v, int n)

{

    vector<int> temp(n);

    return mergeSort(0, n - 1, v, temp);

}

*19-Two Sum*

*Given an array of integers nums and an integer target, return indices of the two numbers such that they add up to target. You may assume that each input would have exactly one solution, and you may not use the same element twice. You can return the answer in any order.*

vector<int> twoSum(vector<int> &nums, int target)

{

    vector<pair<int, int>> p;

    for (int i = 0; i < nums.size(); i++)

    {

        p.push\_back({nums[i], i});

    }

    vector<int> ans;

    sort(p.begin(), p.end());

    int i = 0, j = nums.size() - 1;

    while (i < j)

    {

        int sum = p[i].first + p[j].first;

        if (sum == target)

        {

            ans.push\_back(p[i].second);

            ans.push\_back(p[j].second);

            break;

        }

        else if (sum > target)

        {

            j--;

        }

        else

        {

            i++;

        }

    }

    return ans;

}

*20- 4-Sum*

***Problem Statement:****Given an array of N integers, your task is to find unique quads that add up to give a target value. In short, you need to return*an array of all the unique quadruplets*[arr[a], arr[b], arr[c], arr[d]] such that their sum is equal to a given*target*.*

vector<vector<int>> fourSum(vector<int> &nums, long long target)

{

    int n = nums.size();

    sort(nums.begin(), nums.end());

    set<vector<int>> sv;

    for (int i = 0; i < n; i++)

    {

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

        {

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

            {

                long long x = (long long)target -

                              (long long)nums[i] -

                              (long long)nums[j] - (long long)nums[k];

                if (binary\_search(nums.begin() + k + 1, nums.end(), x))

                {

                    vector<int> v;

                    v.push\_back(nums[i]);

                    v.push\_back(nums[j]);

                    v.push\_back(nums[k]);

                    v.push\_back(x);

                    sort(v.begin(), v.end());

                    sv.insert(v);

                }

            }

        }

    }

    vector<vector<int>> res(sv.begin(), sv.end());

    return res;

}

*21-Longest Consecutive Sequence in an Array*

***Problem Statement:****You are given an array of ‘N’ integers. You need to find the length of the longest sequence which contains the consecutive elements.*

int longestConsecutive(vector<int> &nums)

{

    set<int> st;

    for (auto x : nums)

        st.insert(x);

    int ans = 0;

    for (auto x : nums)

    {

        if (!st.count(x - 1))

        {

            int temp = 1;

            int curr = x;

            while (st.count(curr + 1))

            {

                temp++;

                curr++;

            }

            ans = max(ans, temp);

        }

    }

    return ans;

}

*22-Longest Subarray with Zero Sum*

***Problem Statement:****Given an array containing both positive and negative integers, we have to find the length of the longest subarray with the sum of all elements equal to zero.*

int maxLen(vector<int> &A, int n)

{

    unordered\_map<int, int> mp;

    int sum = 0;

    int ans = 0;

    for (int i = 0; i < n; i++)

    {

        sum += A[i];

        if (sum == 0)

            ans = max(ans, i + 1);

        else

        {

            if (mp.count(sum))

                ans = max(ans, i - mp[sum]);

            else

                mp[sum] = i;

        }

    }

    return ans;

}

*23-Count the no. of subarrays with given XOR ‘K’*

***Problem Statement:****Given an array of integers A and an integer B. Find the total number of subarrays having bitwise XOR of all elements equal to B.*

int solve(vector<int> &A, int B)

{

    unordered\_map<int, int> mp;

    mp[0] = 1;

    int ans = 0;

    int curr = 0;

    for (int i = 0; i < A.size(); i++)

    {

        curr ^= A[i];

        if (mp[curr ^ B])

            ans += mp[curr ^ B];

        mp[curr]++;

    }

    return ans;

}

*24-Longest substring without any repeating character*

***Problem Statement:****Given a String, find the length of longest substring without any repeating character*.

int lengthOfLongestSubstring(string s)

{

    unordered\_map<char, int> mp;

    int i = 0, j = 0;

    int ans = -1;

    while (j < s.length())

    {

        mp[s[j]]++;

        if (mp[s[j]] > 1)

        {

            ans = max(ans, j - i);

            while (i < j && mp[s[j]] != 1)

            {

                mp[s[i]]--;

                i++;

            }

        }

        j++;

    }

    ans = max(ans, j - i);

    return ans;

}

*25-Reverse a Linked List*

***Problem Statement:****Given the*head *of a singly linked list, write a program to reverse the linked list, and return*the head pointer to the reversed list*.*

ListNode \*reverseList(ListNode \*head)

{

    if (!head)

        return head;

    ListNode \*pre = NULL;

    while (head->next != NULL)

    {

        ListNode \*nxt = head->next;

        head->next = pre;

        pre = head;

        head = nxt;

    }

    head->next = pre;

    return head;

}

*26-Middle of a Linked List*

***Problem Statement:****Given the****head****of a singly linked list, return*the middle node of the linked list*. If there are two middle nodes, return the second middle node.*

ListNode \*middleNode(ListNode \*head)

{

    ListNode \*fast = head;

    ListNode \*slow = head;

    while (fast && fast->next)

    {

        fast = fast->next->next;

        slow = slow->next;

    }

    return slow;

}

*27-Merge Two Sorted Linked List*

***Problem Statement:****Given two singly linked lists that are sorted in increasing order of node values, merge two****sorted****linked lists and return them as a sorted list. The list should be made by splicing together the nodes of the first two lists.*

ListNode \*mergeTwoLists(ListNode \*l1, ListNode \*l2)

{

    if (!l1)

        return l2;

    if (!l2)

        return l1;

    if (l1->val > l2->val)

        swap(l1, l2);

    ListNode \*res = l1;

    while (l1 && l2)

    {

        ListNode \*temp = NULL;

        while (l1 && l1->val <= l2->val)

        {

            temp = l1;

            l1 = l1->next;

        }

        temp->next = l2;

        swap(l1, l2);

    }

    return res;

}

*28-Remove nth Node from the end of a LL*

***Problem Statement:****Given a*[*linked list*](https://takeuforward.org/linked-list/linked-list-introduction/)*, and a number N. Find the Nth node from the end of this linked list and delete it. Return the head of the new modified linked list.*

ListNode \*removeNthFromEnd(ListNode \*head, int n)

{

    ListNode \*start = new ListNode();

    start->next = head;

    ListNode \*fast = start;

    ListNode \*slow = start;

    for (int i = 1; i <= n; ++i)

        fast = fast->next;

    while (fast->next != NULL)

    {

        fast = fast->next;

        slow = slow->next;

    }

slow->next = slow->next->next;

    return start->next;

}

*29-Add two numbers represented as LL*

***Problem Statement****: Given the****heads****of two non-empty linked lists representing two non-negative integers. The digits are stored in****reverse order****, and each of their nodes contains a single digit. Add the two numbers and return the****sum****as a linked list.*

ListNode \*addTwoNumbers(ListNode \*l1, ListNode \*l2)

    {

        ListNode \*h1 = l1;

        ListNode \*h2 = l2;

        ListNode \*ans;

        int carry = 0;

        while (l1 && l2)

        {

            int sum = l1->val + l2->val + carry;

            carry = sum / 10;

            sum = sum % 10;

            l1->val = sum;

            l2->val = sum;

            l1 = l1->next;

            l2 = l2->next;

        }

        if (!l1 && l2)

        {

            while (l2)

            {

                int sum = l2->val + carry;

                carry = sum / 10;

                sum %= 10;

                l2->val = sum;

                l2 = l2->next;

            }

            if (carry)

            {

                ListNode \*node = new ListNode(carry);

                l2 = h2;

                while (l2->next)

                    l2 = l2->next;

                l2->next = node;

                l2->next->next = NULL;

            }

            ans = h2;

        }

        else if (l1 && !l2)

        {

            while (l1)

            {

                int sum = l1->val + carry;

                carry = sum / 10;

                sum %= 10;

                l1->val = sum;

                l1 = l1->next;

            }

            if (carry)

            {

                ListNode \*node = new ListNode(carry);

                l1 = h1;

                while (l1->next)

                    l1 = l1->next;

                l1->next = node;

                l1->next->next = NULL;

            }

            ans = h1;

        }

        else

        {

            if (carry)

            {

                ListNode \*node = new ListNode(carry);

                l1 = h1;

                while (l1->next)

                    l1 = l1->next;

                l1->next = node;

                l1->next->next = NULL;

            }

            ans = h1;

        }

        return ans;

    }

*30-Delete given node in a LL*

***Problem Statement:****Write a function to****delete a node****in a singly-linked list. You will****not****be given access to the head of the list instead, you will be given access to****the node to be deleted****directly. It is****guaranteed****that the node to be deleted is****not a tail node****in the list.*

void deleteNode(ListNode \*node)

{

    node->val = node->next->val;

    node->next = node->next->next;

    return;

}

*31-Find intersection of two LL*

***Problem Statement:****Given the heads of two singly*[*linked-lists*](https://takeuforward.org/linked-list/linked-list-introduction/)***headA****and****headB****, return****the node at which the two lists intersect****. If the two linked lists have no intersection at all, return****null****.*

ListNode \*getIntersectionNode(ListNode \*A, ListNode \*B)

{

    ListNode \*d1 = A;

    ListNode \*d2 = B;

    while (d1 != d2)

    {

        d1 = d1 == NULL ? B : d1 = d1->next;

        d2 = d2 == NULL ? A : d2 = d2->next;

    }

    return d1;

}

*32-Detect cycle in a LL*

***Problem Statement:****Given*head*, the head of a linked list, determine if the linked list has a cycle in it. There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer.*

*Return*true*if there is a cycle in the linked list. Otherwise, return*false*.*

bool hasCycle(ListNode \*head)

{

    ListNode \*fast = head;

    ListNode \*slow = head;

    if (!fast || !fast->next)

        return false;

    do

    {

        fast = fast->next->next;

        slow = slow->next;

    } while (fast != slow && fast && fast->next);

    if (fast == slow)

        return true;

    else

        return false;

}

*33-Reverse LL in groups of size k*

***Problem Statement:****Given the head of a linked list, reverse the nodes of the list k at a time, and return*the modified list*. k is a positive integer and is less than or equal to the length of the linked list. If the number of nodes is not a multiple of k then left-out nodes, in the end, should remain as it is.*

int Length(ListNode \*head)

{

    int len = 0;

    while (head)

    {

        len++;

        head = head->next;

    }

    return len;

}

ListNode \*reverseKGroup(ListNode \*head, int k)

{

    ListNode \*dummy = new ListNode(0);

    dummy->next = head;

    ListNode \*pre = dummy;

    ListNode \*cur;

    ListNode \*nex;

    int len = Length(head);

    while (len >= k)

    {

        cur = pre->next;

        nex = cur->next;

        for (int i = 1; i < k; i++)

        {

            cur->next = nex->next;

            nex->next = pre->next;

            pre->next = nex;

            nex = cur->next;

        }

        pre = cur;

        len -= k;

    }

    return dummy->next;

}

*34-Check for palindrome*

***Problem Statement:****Given the head of a singly linked list, return true if it is a palindrome.*

ListNode \*reverse(ListNode \*ptr)

{

    ListNode \*pre = NULL;

    ListNode \*nex = NULL;

    while (ptr != NULL)

    {

        nex = ptr->next;

        ptr->next = pre;

        pre = ptr;

        ptr = nex;

    }

    return pre;

}

bool isPalindrome(ListNode \*head)

{

    if (head == NULL || head->next == NULL)

        return true;

    ListNode \*slow = head;

    ListNode \*fast = head;

    while (fast->next != NULL && fast->next->next != NULL)

    {

        slow = slow->next;

        fast = fast->next->next;

    }

    slow->next = reverse(slow->next);

    slow = slow->next;

    ListNode \*dummy = head;

    while (slow != NULL)

    {

        if (dummy->val != slow->val)

            return false;

        dummy = dummy->next;

        slow = slow->next;

    }

    return true;

}

*35-Starting point of a loop of a LL*

***Problem Statement:****Given the head of a*[*linked list*](https://takeuforward.org/linked-list/linked-list-introduction/)*, return*the node where the cycle begins. If there is no cycle, return *null.*

*There is a cycle in a linked list if there is some node in the list that can be reached again by continuously following the next pointer. Internally, pos is used to denote the index of the node that the tail’s next pointer is connected to (0-indexed). It is -1 if there is no cycle. Note that pos is not passed as a parameter.*

ListNode \*detectCycle(ListNode \*head)

{

    ListNode \*fast = head;

    ListNode \*slow = head;

    if (!fast || !fast->next)

        return NULL;

    do

    {

        fast = fast->next->next;

        slow = slow->next;

    } while (fast != slow && fast && fast->next);

    if (fast != slow)

        return NULL;

    fast = head;

    while (fast != slow)

    {

        slow = slow->next;

        fast = fast->next;

    }

    return slow;

}

*36-Flattening a LL*

**Problem Statement:**Given a [Linked List](https://takeuforward.org/linked-list/linked-list-introduction/) of size N, where every node represents a sub-linked-list and contains two pointers:

(i) a next pointer to the next node,

(ii) a bottom pointer to a linked list where this node is head.

Each of the sub-linked-list is in sorted order.

Flatten the Link List such that all the nodes appear in a single level while maintaining the sorted order.

Note: The flattened list will be printed using the bottom pointer instead of the next pointer.

**Examples:**

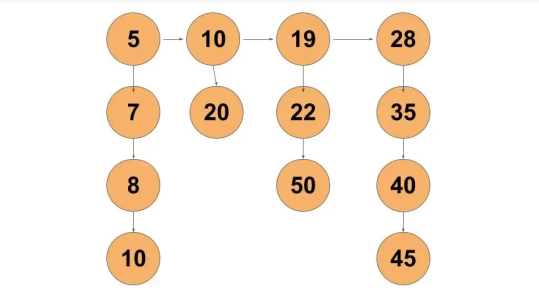
**Example 1:**

**Input:**

Number of head nodes = 4

Array holding length of each list with head and bottom = [4,2,3,4]

Elements of entire linked list = [5,7,8,30,10,20,19,22,50,28,35,40,45]

****

**Output:**

Flattened list = [5,7,8,10,19,20,22,28,30,35,40,45,50]

**Explanation:**

Flattened list is the linked list consisting entire elements of the given list in sorted order

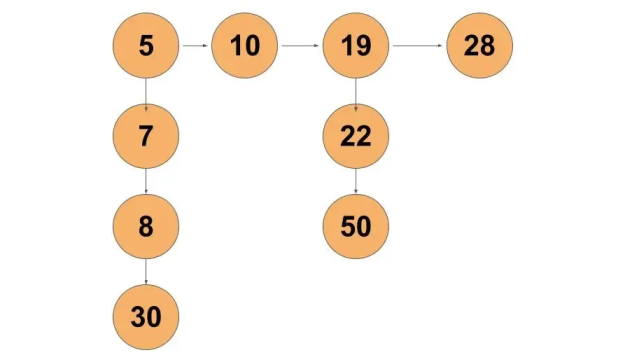
**Example 2:**

**Input:**

Number of head nodes = 4

Array holding length of each list with head and bottom = [4,1,3,1]

Elements of entire linked list = [5,7,8,30,10,19,22,50,28]

****

**Output:**

Flattened list = [5,7,8,10,19,22,28,30,50]

**Explanation:**

Flattened list is the linked list consisting entire elements of the given list in sorted order

Node \*merge(Node \*a, Node \*b)

{

    if (!a)

        return b;

    if (!b)

        return a;

    Node \*result;

    if (a->data <= b->data)

    {

        result = a;

        result->bottom = merge(a->bottom, b);

    }

    else

    {

        result = b;

        result->bottom = merge(a, b->bottom);

    }

    return result;

}

Node \*flatten(Node \*root)

{

    if (root == NULL || root->next == NULL)

        return root;

    return merge(root, flatten(root->next));

}

*37-Rotate a LL*

***Problem Statement:****Given the head of a*[*linked list*](https://takeuforward.org/linked-list/linked-list-introduction/)*, rotate the list to the right by k places.*

int length(ListNode \*head)

{

    int len = 0;

    while (head)

    {

        len++;

        head = head->next;

    }

    return len;

}

ListNode \*rotateRight(ListNode \*head, int k)

{

    if (!head || !head->next)

        return head;

    int len = length(head);

    k = k % len;

    k = len - k;

    k--;

    ListNode \*temp = head;

    while (k--)

    {

        temp = temp->next;

    }

    ListNode \*last = head;

    while (last->next)

        last = last->next;

    last->next = head;

    head = temp->next;

    temp->next = NULL;

    return head;

}

*38-Copy LL with Random pointers*

A linked list of length n is given such that each node contains an additional random pointer, which could point to any node in the list, or null.

Construct a [**deep copy**](https://en.wikipedia.org/wiki/Object_copying#Deep_copy) of the list. The deep copy should consist of exactly n **brand new** nodes, where each new node has its value set to the value of its corresponding original node. Both the next and random pointer of the new nodes should point to new nodes in the copied list such that the pointers in the original list and copied list represent the same list state. **None of the pointers in the new list should point to nodes in the original list**.

For example, if there are two nodes X and Y in the original list, where X.random --> Y, then for the corresponding two nodes x and y in the copied list, x.random --> y.

Return *the head of the copied linked list*.

The linked list is represented in the input/output as a list of n nodes. Each node is represented as a pair of [val, random\_index] where:

* val: an integer representing Node.val
* random\_index: the index of the node (range from 0 to n-1) that the random pointer points to, or null if it does not point to any node.

Your code will **only** be given the head of the original linked list.

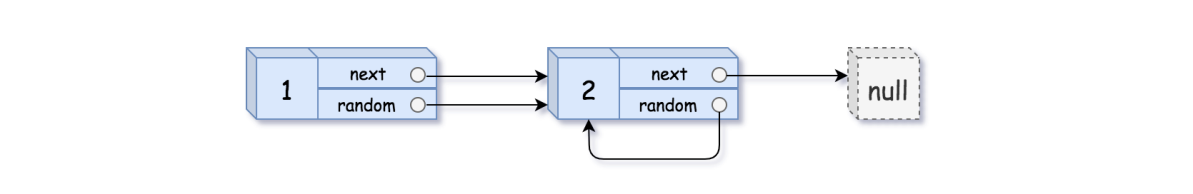
**Example 1:**



**Input:** head = [[7,null],[13,0],[11,4],[10,2],[1,0]]

**Output:** [[7,null],[13,0],[11,4],[10,2],[1,0]]

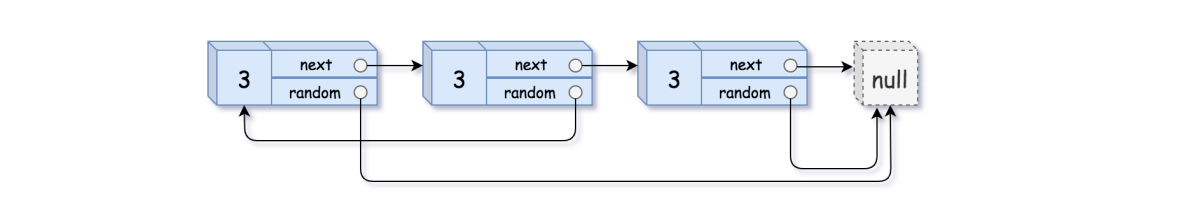
**Example 2:**



**Input:** head = [[1,1],[2,1]]

**Output:** [[1,1],[2,1]]

**Example 3:**

****

**Input:** head = [[3,null],[3,0],[3,null]]

**Output:** [[3,null],[3,0],[3,null]]

Node \*copyRandomList(Node \*head)

{

    Node \*iter = head;

    Node \*front = head;

    while (iter)

    {

        front = iter->next;

        Node \*copy = new Node(iter->val);

        iter->next = copy;

        copy->next = front;

        iter = front;

    }

    iter = head;

    while (iter)

    {

        if (iter->random)

            iter->next->random = iter->random->next;

        iter = iter->next->next;

    }

    iter = head;

    Node \*phead = new Node(0);

    Node \*copy = phead;

    while (iter)

    {

        front = iter->next->next;

        copy->next = iter->next;

        iter->next = front;

        copy = copy->next;

        iter = iter->next;

    }

    return phead->next;

}

*39-3 Sum*

***Problem Statement:****Given an array of N integers, your task is to find unique triplets that add up to give a sum of zero. In short, you need to return*an array of all the unique*triplets [arr[a], arr[b], arr[c]] such that i!=j, j!=k, k!=i, and their sum is equal to zero.*

vector<vector<int>> threeSum(vector<int> &nums)

{

    int n = nums.size();

    sort(nums.begin(), nums.end());

    vector<vector<int>> ans;

    for (int i = 0; i < n; i++)

    {

        if (nums[i] > 0)

            break;

        if (i > 0 && nums[i - 1] == nums[i])

            continue;

        int sum = -nums[i];

        int l = i + 1, r = n - 1;

        while (l < r)

        {

            int temp = nums[l] + nums[r];

            if (temp == sum)

            {

                vector<int> v;

                ans.push\_back({nums[i], nums[l], nums[r]});

                while (l < r && nums[l] == nums[l + 1])

                    l++;

                while (l < r && nums[r] == nums[r - 2])

                    r--;

                l++;

                r--;

            }

            else if (temp > sum)

                r--;

            else

                l++;

        }

    }

    return ans;

}

*40-Remove Duplicates in-place from a sorted array*

***Problem Statement:****Given an integer array sorted in non-decreasing order, remove the duplicates in place such that each unique element appears only once. The relative order of the elements should be kept the same.If there are k elements after removing the duplicates, then the first k elements of the array should hold the final result. It does not matter what you leave beyond the first k elements.*

***Note:****Return k after placing the final result in the first k slots of the array.*

int removeDuplicates(vector<int> &nums)

{

    int c = 1;

    int curr = nums[0];

    for (int i = 1; i < nums.size(); i++)

    {

        if (nums[i] != curr)

        {

            curr = nums[i];

            nums[c] = curr;

            c++;

        }

    }

    return c;

}

*41-Count Maximum consecutive ones in an array*

***Problem Statement:****Given an array that contains****only 1 and 0****return the count of****maximum consecutive****ones in the array.*

int findMaxConsecutiveOnes(vector<int> &nums)

{

    int ans = 0;

    int cur = 0;

    for (int i = 0; i < nums.size(); i++)

    {

        if (nums[i] == 1)

        {

            cur++;

        }

        else

        {

            ans = max(ans, cur);

            cur = 0;

        }

    }

    ans = max(ans, cur);

    return ans;

}

*42-N Meetings in one room*

***Problem Statement:****There is****one****meeting room in a firm. You are given two arrays, start and end each of size N.For an index ‘i’, start[i] denotes the starting time of the ith meeting while end[i]  will denote the ending time of the ith meeting. Find the maximum number of meetings that can be accommodated if only one meeting can happen in the room at a  particular time. Print the order in which these meetings will be performed.*

int maxMeetings(int start[], int end[], int n)

{

    vector<pair<int, int>> v;

    for (int i = 0; i < n; i++)

    {

        v.push\_back({end[i], start[i]});

    }

    sort(v.begin(), v.end());

    int limit = v[0].first;

    int ans = 1;

    for (int i = 1; i < n; i++)

    {

        if (v[i].second > limit)

        {

            limit = v[i].first;

            ans++;

        }

    }

    return ans;

}

***43-Minimum number of platforms required for trains***

***Problem Statement:****We are given two arrays that represent the arrival and departure times of trains that stop at the platform. We need to find the minimum number of platforms needed at the railway station so that no train has to wait.*

int findPlatform(int arr[], int dep[], int n)

{

    if (n == 1)

        return 1;

    vector<pair<int, int>> p;

    for (int i = 0; i < n; i++)

    {

        p.push\_back({arr[i], dep[i]});

    }

    sort(p.begin(), p.end());

    priority\_queue<int> pq;

    pq.push(-p[0].second);

    for (int i = 1; i < n; i++)

    {

        if (-pq.top() < p[i].first)

        {

            pq.pop();

            pq.push(-p[i].second);

        }

        else

            pq.push(-p[i].second);

    }

    return pq.size();

}

***44-Job sequencing problem***

***Problem Statement:****You are given a set of N jobs where each job comes with a****deadline****and****profit****. The profit can only be earned upon completing the job within its deadline. Find the****number of jobs****done and the****maximum profit****that can be obtained. Each job takes a****single unit****of time and only****one job****can be performed at a time.*

vector<int> JobScheduling(Job arr[], int n)

{

    vector<pair<int, int>> v;

    for (int i = 0; i < n; i++)

    {

        v.push\_back({arr[i].profit, arr[i].dead});

    }

    vector<int> pre(100005, 0);

    sort(v.begin(), v.end());

    for (int i = n - 1; i >= 0; i--)

    {

        int idx = v[i].second;

        if (pre[idx] == 0)

            pre[idx] = v[i].first;

        else

        {

            for (int j = idx - 1; j >= 1; j--)

            {

                if (pre[j] == 0)

                {

                    pre[j] = v[i].first;

                    break;

                }

            }

        }

    }

    int ans = 0;

    int cnt = 0;

    for (auto x : pre)

    {

        if (x > 0)

        {

            ans += x;

            cnt++;

        }

    }

    return {cnt, ans};

}

***45-Fractional Knapsack:***Greedy approach

***Problem Statement:****The weight of****N****items and their corresponding values are given. We have to put these items in a knapsack of weight****W****such that the****total value****obtained is****maximized.******Note:****We can either take the item as a whole or break it into smaller units.*

static bool comp(Item a, Item b)

{

    double r1 = (double)a.value / (double)a.weight;

    double r2 = (double)b.value / (double)b.weight;

    return r1 > r2;

}

double fractionalKnapsack(int W, Item arr[], int n)

{

    sort(arr, arr + n, comp);

    int cur = 0;

    double ans = 0;

    for (int i = 0; i < n; i++)

    {

        if (cur + arr[i].weight <= W)

        {

            ans += arr[i].value;

            W -= arr[i].weight;

        }

        else

        {

            int rem = min(W, arr[i].weight);

            ans += rem \* ((double)(arr[i].value) / (double)(arr[i].weight));

            break;

        }

    }

    return ans;

}

***46-Find the Minimum number of coins***

***Problem Statement****: Given a value V, if we want to make a change for V Rs, and we have an infinite supply of each of the denominations in Indian currency, i.e., we have an infinite supply of { 1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, what is the minimum number of coins and/or notes needed to make the change.*

int findMinimumCoins(int amount)

{

    vector<int> v = {1, 2, 5, 10, 20, 50, 100, 500, 1000};

    int ans = 0;

    for (int i = 8; i >= 0; i--)

    {

        if (amount >= v[i])

        {

            ans += amount / v[i];

            amount = amount % v[i];

            if (amount == 0)

                break;

        }

    }

    return ans;

}

***47-Sum of all Subsets***

***Problem Statement:****Given an array print all the sum of the subset generated from it, in the increasing order.*

void helper(vector<int> &arr, int i, vector<int> &ans, int sum)

{

    if (i >= arr.size())

    {

        ans.push\_back(sum);

        return;

    }

    helper(arr, i + 1, ans, sum + arr[i]);

    helper(arr, i + 1, ans, sum);

}

vector<int> subsetSums(vector<int> arr, int N)

{

    vector<int> ans;

    helper(arr, 0, ans, 0);

    return ans;

}

***48-Print all unique subsets***

***Problem Statement:****Given an array of integers that****may contain duplicates****the task is to return all possible subsets. Return only****unique subsets****and they can be in any order.*

void findSubsets(int ind, vector<int> &nums, vector<int> &ds, vector<vector<int>> &ans)

{

    ans.push\_back(ds);

    for (int i = ind; i < nums.size(); i++)

    {

        if (i != ind && nums[i] == nums[i - 1])

            continue;

        ds.push\_back(nums[i]);

        findSubsets(i + 1, nums, ds, ans);

        ds.pop\_back();

    }

}

public:

vector<vector<int>> subsetsWithDup(vector<int> &nums)

{

    vector<vector<int>> ans;

    vector<int> ds;

    sort(nums.begin(), nums.end());

    findSubsets(0, nums, ds, ans);

    return ans;

}

***49-Palindrome Partioning***

***Given a string s, partition s such that every substring of the partition is a palindrome . Return all possible palindrome partitioning of s.***

bool isPalindrome(string &s, int l, int r, vector<vector<int>> &dp)

{

    while (l < r)

    {

        if (s[l] == s[r])

        {

            l++;

            r--;

        }

        else

        {

            dp[l][r] = 0;

            return false;

        }

    }

    dp[l][r] = 1;

    return true;

}

void helper(string &s, int idx, vector<string> temp, vector<vector<string>> &ans, vector<vector<int>> &dp)

{

    if (idx == s.length())

    {

        ans.push\_back(temp);

        return;

    }

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

    {

        if (dp[idx][i] != -1)

        {

            if (dp[idx][i] == 1)

            {

                temp.push\_back(s.substr(idx, i - idx + 1));

                helper(s, i + 1, temp, ans, dp);

                temp.pop\_back();

            }

        }

        else

        {

            if (isPalindrome(s, idx, i, dp))

            {

                temp.push\_back(s.substr(idx, i - idx + 1));

                helper(s, i + 1, temp, ans, dp);

                temp.pop\_back();

            }

        }

    }

}

vector<vector<string>> partition(string s)

{

    vector<string> temp;

    vector<vector<string>> ans;

    vector<vector<int>> dp(s.length() + 1, vector<int>(s.length() + 1, -1));

    helper(s, 0, temp, ans, dp);

    return ans;

}

***50-Combination Sum-1***

**Problem Statement:**

*Given an array of distinct integers and a****target****, you have to return*the list of all unique combinations where the chosen numbers sum to *target*.*You may return the combinations in any order.*

*The same number may be chosen from the given array an unlimited number of times. Two combinations are unique if the frequency of at least one of the chosen numbers is different.*

*It is guaranteed that the number of unique combinations that sum up to****target****is less than****150****combinations for the given input.*

void findCombination(int ind, int target, vector<int> &arr, vector<vector<int>> &ans, vector<int> &ds)

{

    if (ind == arr.size())

    {

        if (target == 0)

        {

            ans.push\_back(ds);

        }

        return;

    }

    if (arr[ind] <= target)

    {

        ds.push\_back(arr[ind]);

        findCombination(ind, target - arr[ind], arr, ans, ds);

        ds.pop\_back();

    }

    findCombination(ind + 1, target, arr, ans, ds);

}

vector<vector<int>> combinationSum(vector<int> &candidates, int target)

{

    vector<vector<int>> ans;

    vector<int> ds;

    findCombination(0, target, candidates, ans, ds);

    return ans;

}

***51-Combination Sum-2***

***Problem Statement:****Given a collection of candidate numbers (candidates) and a target number (target), find all unique combinations in candidates where the candidate numbers sum to target. Each number in candidates may only be used once in the combination.*

***Note: The solution set must not contain duplicate combinations****.*

void helper(vector<int> &v, int idx, int t, vector<int> temp, vector<vector<int>> &ans)

{

    if (t == 0)

    {

        ans.push\_back(temp);

        return;

    }

    for (int i = idx; i < v.size(); i++)

    {

        if (i != idx && v[i] == v[i - 1])

            continue;

        if (v[i] <= t)

        {

            temp.push\_back(v[i]);

            helper(v, i + 1, t - v[i], temp, ans);

            temp.pop\_back();

        }

    }

}

vector<vector<int>> combinationSum2(vector<int> &v, int t)

{

    sort(v.begin(), v.end());

    vector<int> temp;

    vector<vector<int>> ans;

    helper(v, 0, t, temp, ans);

    return ans;

}

***52-Find Kth permutation of a Sequence***

***Problem Statement:****Given****N****and****K****, where N is the sequence of numbers from****1 to N([1,2,3….. N])****find the****Kth permutation sequence****.*

string getPermutation(int n, int k)

{

    string ans = "";

    vector<int> v;

    int f = 1;

    for (int i = 1; i < n; i++)

    {

        f \*= i;

        v.push\_back(i);

    }

    v.push\_back(n);

    k--;

    while (true)

    {

        ans += to\_string(v[k / f]);

        v.erase(v.begin() + k / f);

        if (v.size() == 0)

            break;

        k = k % f;

        f = f / v.size();

    }

    return ans;

}

***53-Print all permutations of a string***

***Problem Statement:****Given an array arr of distinct integers, print all permutations of String/Array.*

void helper(vector<int> &v, int idx, vector<vector<int>> &ans)

{

    if (idx >= v.size())

    {

        ans.push\_back(v);

        return;

    }

    for (int i = idx; i < v.size(); i++)

    {

        swap(v[i], v[idx]);

        helper(v, idx + 1, ans);

        swap(v[i], v[idx]);

    }

}

vector<vector<int>> permute(vector<int> &nums)

{

    vector<vector<int>> ans;

    helper(nums, 0, ans);

    return ans;

}

***54-N-Queen Problem***

***Problem Statement:****The n-queens is the problem of placing n queens on n × n chessboard such that no two queens can attack each other. Given an integer n, return all distinct solutions to the n -queens puzzle. Each solution contains a distinct boards configuration of the queen’s placement, where ‘Q’ and ‘.’ indicate queen and empty space respectively.*

void helper(int col, vector<string> &temp, vector<vector<string>> &ans, int n, vector<int> left, vector<int> lower, vector<int> upper)

{

    if (col == n)

    {

        ans.push\_back(temp);

        return;

    }

    for (int row = 0; row < n; row++)

    {

        if (left[row] == 0 && lower[row + col] == 0 && upper[n - 1 + col - row] == 0)

        {

            temp[row][col] = 'Q';

            left[row] = 1;

            lower[row + col] = 1;

            upper[n - 1 + col - row] = 1;

            helper(col + 1, temp, ans, n, left, lower, upper);

            temp[row][col] = '.';

            left[row] = 0;

            lower[row + col] = 0;

            upper[n - 1 + col - row] = 0;

        }

    }

}

vector<vector<string>> solveNQueens(int n)

{

    vector<vector<string>> ans;

    vector<string> temp;

    string s(n, '.');

    for (int i = 0; i < n; i++)

        temp.push\_back(s);

    vector<int> leftRow(n, 0), lowerDia(2 \* n - 1, 0), upperDia(2 \* n - 1, 0);

    helper(0, temp, ans, n, leftRow, lowerDia, upperDia);

    return ans;

}

***55-Sudoku Solver***

Given a 9×9 incomplete sudoku, solve it such that it becomes valid sudoku. Valid sudoku has the following properties.

         1. All the rows should be filled with numbers(1 – 9) exactly once.

         2. All the columns should be filled with numbers(1 – 9) exactly once.

         3. Each 3×3 submatrix should be filled with numbers(1 – 9) exactly once.

**Note**: Character **‘.’** indicates empty cell.

bool Valid(vector<vector<char>> &b, int row, int col, char ch)

{

    for (int i = 0; i < b.size(); i++)

    {

        if (b[i][col] == ch)

            return false;

        if (b[row][i] == ch)

            return false;

        if (b[3 \* (row / 3) + i / 3][3 \* (col / 3) + i % 3] == ch)

            return false;

    }

    return true;

}

bool helper(vector<vector<char>> &b)

{

    for (int i = 0; i < b.size(); i++)

    {

        for (int j = 0; j < b[0].size(); j++)

        {

            if (b[i][j] == '.')

            {

                for (char ch = '1'; ch <= '9'; ch++)

                {

                    if (Valid(b, i, j, ch))

                    {

                        b[i][j] = ch;

                        if (helper(b))

                            return true;

                        else

                            b[i][j] = '.';

                    }

                }

                return false;

            }

        }

    }

    return true;

}

void solveSudoku(vector<vector<char>> &board)

{

    helper(board);

}

***56-M Coloring Problem***

***Problem Statement:****Given an undirected graph and a number m, determine if the graph can be colored with at most m colors such that no two adjacent vertices of the graph are colored with the same color.*

bool Safe(bool graph[101][101], int node, int col, vector<int> &color, int n)

{

    for (int i = 0; i < n; i++)

    {

        if (i != node && graph[node][i] && color[i] == col)

            return false;

    }

    return true;

}

bool helper(bool graph[101][101], int node, vector<int> &color, int m, int n)

{

    if (node == n)

        return true;

    for (int i = 1; i <= m; i++)

    {

        if (Safe(graph, node, i, color, n))

        {

            color[node] = i;

            if (helper(graph, node + 1, color, m, n))

                return true;

            else

                color[node] = -1;

        }

    }

    return false;

}

bool graphColoring(bool graph[101][101], int m, int n)

{

    vector<int> color(n, -1);

    if (helper(graph, 0, color, m, n))

        return true;

    else

        return false;

}

***57-Rat in a Maze***

*Consider a rat placed at****(0, 0)****in a square matrix**of order****N \* N****. It has to reach the destination at****(N – 1, N – 1)****. Find all possible paths that the rat can take to reach from source to destination. The directions in which the rat can move are****‘U'(up)****,****‘D'(down)****,****‘L’ (left)****,****‘R’ (right)****. Value 0 at a cell in the matrix represents that it is blocked and the rat cannot move to it while value 1 at a cell in the matrix represents that rat can travel through it.*

***Note****: In a path, no cell can be visited more than one time.*

*Print the answer in lexicographical(sorted) order*

bool Path(vector<vector<int>> &m, int i, int j, int n)

{

    if (i >= 0 && i <= n - 1 && j >= 0 && j <= n - 1 && m[i][j] == 1)

        return true;

    else

        return false;

}

void helper(vector<vector<int>> &m, int n, int i, int j, string &temp, vector<string> &ans)

{

    if (i == n - 1 && j == n - 1)

    {

        ans.push\_back(temp);

        return;

    }

    m[i][j] = 0;

    if (Path(m, i + 1, j, n))

    {

        temp += "D";

        helper(m, n, i + 1, j, temp, ans);

        temp = temp.substr(0, temp.length() - 1);

    }

    if (Path(m, i - 1, j, n))

    {

        temp += "U";

        helper(m, n, i - 1, j, temp, ans);

        temp = temp.substr(0, temp.length() - 1);

    }

    if (Path(m, i, j + 1, n))

    {

        temp += "R";

        helper(m, n, i, j + 1, temp, ans);

        temp = temp.substr(0, temp.length() - 1);

    }

    if (Path(m, i, j - 1, n))

    {

        temp += "L";

        helper(m, n, i, j - 1, temp, ans);

        temp = temp.substr(0, temp.length() - 1);

    }

    m[i][j] = 1;

}

vector<string> findPath(vector<vector<int>> &m, int n)

{

    if (m[n - 1][n - 1] == 0 || m[0][0] == 0)

        return {{"-1"}};

    string temp = "";

    vector<string> ans;

    helper(m, n, 0, 0, temp, ans);

    return ans;

}

*58-Word Break-II*

*You are given a non-empty string S containing no spaces and a dictionary of non-empty strings (say the list of words). You are supposed to construct and return all possible sentences after adding spaces in the originally given string S', such that each word in a sentence exists in the given dictionary. Note: The same word in the dictionary can be used multiple times to make sentences.Assume that the dictionary does not contain duplicate words.*

void helper(string &s, int idx, string &temp, vector<string> &ans,

            unordered\_map<string, int> &mp)

{

    if (idx >= s.length())

    {

        ans.push\_back(temp);

        return;

    }

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

    {

        string st = s.substr(idx, i - idx + 1);

        string str = temp;

        if (mp[st] > 0)

        {

            temp += st + " ";

            helper(s, i + 1, temp, ans, mp);

            temp = str;

        }

    }

}

vector<string> wordBreak(string &s, vector<string> &dictionary)

{

    unordered\_map<string, int> mp;

    for (auto x : dictionary)

        mp[x]++;

    vector<string> ans;

    string temp = "";

    helper(s, 0, temp, ans, mp);

    return ans;

}

*59-Nth root of M*

***Problem Statement:****Given two numbers N and M, find the Nth root of M. The nth root of a number M is defined as a number X when raised to the power N equals M*

double findNthRootOfM(int n, int m)

{

    double start = 1, end = m;

    double eps = 0.00000001;

    while (end - start > eps)

    {

        double mid = (start + end) / 2;

        double mul = pow(mid, n);

        if (mul < m)

        {

            start = mid;

        }

        else

        {

            end = mid;

        }

    }

    return end;

}

*60-Matrix Median*

Given a matrix of integers **A** of size N x M in which each row is sorted. Find and return the overall median of matrix A. **NOTE**: No extra memory is allowed. **NOTE**: Rows are numbered from top to bottom and columns are numbered from left to right.

int findMedian(vector<vector<int>> &A)

{

    int n = A.size();

    int m = A[0].size();

    int minVal = INT\_MAX, maxVal = INT\_MIN;

    for (int i = 0; i < n; i++)

    {

        minVal = min(minVal, A[i][0]);

        maxVal = max(maxVal, A[i][m - 1]);

    }

    int desired = (n \* m + 1) / 2;

    while (minVal < maxVal)

    {

        int mid = minVal + (maxVal - minVal) / 2;

        int place = 0;

        for (int i = 0; i < n; i++)

        {

            place += upper\_bound(A[i].begin(), A[i].end(), mid) - A[i].begin();

        }

        if (place < desired)

        {

            minVal = mid + 1;

        }

        else

        {

            maxVal = mid;

        }

    }

    return minVal;

}

***61-Single Element in a Sorted array***

***Problem Statement:****Given a sorted array of N integers, where every element except one appears exactly twice and one element appears only once. Search Single Element in a sorted array.*

int singleNonDuplicate(vector<int>& nums) {

           int l=0,r=nums.size()-2;

           while(l<=r){

               int mid=(l+r)/2;

               if(mid%2==0)

               {

                   if(nums[mid]!=nums[mid+1])

                   r=mid-1;

                   else

                   l=mid+1;

               }

               else{

                   if(nums[mid]==nums[mid+1])

                   r=mid-1;

                   else

                   l=mid+1;

               }

           }

           return nums[l];

    }

62-Search in a Rotated Sorted Array

**Problem Statement:**There is an integer array **nums** sorted in ascending order (with distinct values). Given the array **nums** after the possible clockwise rotation and an integer **target**, return the **index** **of** **target** ifit is in nums, or -1 if it is not in nums. We need to search a given element in a rotated sorted array.

int search(vector<int>& nums, int t) {

       int l=0,r=nums.size()-1;

       while(l<=r){

           int mid=(l+r)/2;

           if(nums[mid]==t){

               return mid;

           }

           if(nums[mid]<nums[l]){

               if(t>=nums[mid] && t<=nums[r])

               l=mid+1;

               else

               r=mid-1;

           }

           else{

               if(t>=nums[l] && t<=nums[mid])

               r=mid-1;

               else

               l=mid+1;

           }

       }

       return -1;

    }

63-Median of two sorted arrays of different sizes

**Problem Statement:**Given **two sorted arrays** arr1 and arr2 of size m and n respectively, return the **median** of the two sorted arrays.

double findMedianSortedArrays(vector<int> &nums1, vector<int> &nums2)

{

    if (nums1.size() > nums2.size())

        return findMedianSortedArrays(nums2, nums1);

    int n1 = nums1.size();

    int n2 = nums2.size();

    int low = 0, high = n1;

    while (low <= high)

    {

        int cut1 = (low + high) / 2;

        int cut2 = (n1 + n2 + 1) / 2 - cut1;

        int l1 = cut1 == 0 ? INT\_MIN : nums1[cut1 - 1];

        int l2 = cut2 == 0 ? INT\_MIN : nums2[cut2 - 1];

        int r1 = cut1 == n1 ? INT\_MAX : nums1[cut1];

        int r2 = cut2 == n2 ? INT\_MAX : nums2[cut2];

        if (l1 <= r2 && l2 <= r1)

        {

            if ((n1 + n2) % 2 == 0)

                return (max(l1, l2) + min(r1, r2)) / 2.0;

            else

                return max(l1, l2);

        }

        else if (l1 > r2)

            high = cut1 - 1;

        else

            low = cut1 + 1;

    }

    return 0.0;

}

64-Kth Element of two sorted Array

***Problem Statement:****Given****two sorted arrays****of size****m****and****n****respectively, you are tasked with finding the element that would be at the****kth position****of the****final sorted array****.*

int kthElement(int nums1[], int nums2[], int n, int m, int k)

{

    if (n > m)

        return kthElement(nums2, nums1, m, n, k);

    int low = max(0, k - m), high = min(n, k);

    while (low <= high)

    {

        int cut1 = (low + high) / 2;

        int cut2 = k - cut1;

        int l1 = cut1 == 0 ? INT\_MIN : nums1[cut1 - 1];

        int l2 = cut2 == 0 ? INT\_MIN : nums2[cut2 - 1];

        int r1 = cut1 == n ? INT\_MAX : nums1[cut1];

        int r2 = cut2 == m ? INT\_MAX : nums2[cut2];

        if (l1 <= r2 && l2 <= r1)

            return max(l1, l2);

        else if (l1 > r2)

            high = cut1 - 1;

        else

            low = cut1 + 1;

    }

    return 0;

}

*65-Allocate minimum number of Pages*

**Problem Statement:**Given an array of integers A of size N and an integer B .The College library has N bags, the ith book has A[i] number of pages. You have to allocate books to B number of students so that the maximum number of pages allocated to a student is minimum.

Conditions given :

A book will be allocated to exactly one student.

Each student has to be allocated at least one book.

Allotment should be in contiguous order, for example, A student cannot be allocated book 1 and book 3, skipping book 2.

Calculate and return the **minimum possible number**. Return -1 if a valid assignment is not possible.

int isPossible(vector<int> &A, int pages, int students)

{

    int cnt = 0;

    int sumAllocated = 0;

    for (int i = 0; i < A.size(); i++)

    {

        if (sumAllocated + A[i] > pages)

        {

            cnt++;

            sumAllocated = A[i];

            if (sumAllocated > pages)

                return false;

        }

        else

        {

            sumAllocated += A[i];

        }

    }

    if (cnt < students)

        return true;

    return false;

}

int Solution::books(vector<int> &A, int B)

{

    if (B > A.size())

        return -1;

    int low = A[0];

    int high = 0;

    // to find minimum value and sum of all pages

    for (int i = 0; i < A.size(); i++)

    {

        high = high + A[i];

        low = min(low, A[i]);

    }

    int res = -1;

    // binary search

    while (low <= high)

    {

        int mid = (low + high) >> 1;

        if (isPossible(A, mid, B))

        {

            high = mid - 1, res = mid;

        }

        else

        {

            low = mid + 1;

        }

    }

    return res;

}

*66-Aggressive Cows*

**Problem Statement:** There is a new barn with N stalls and C cows. The stalls are located on a straight line at positions x1,….,xN (0 <= xi <= 1,000,000,000). We want to assign the cows to the stalls, such that the minimum distance between any two of them is as large as possible. What is the largest minimum distance?

bool isPossible(vl &v, ll n, ll cows, ll minDist)

{

    int cntCows = 1;

    int lastPlacedCow = v[0];

    for (int i = 1; i < n; i++)

    {

        if (v[i] - lastPlacedCow >= minDist)

        {

            cntCows++;

            lastPlacedCow = v[i];

        }

    }

    if (cntCows >= cows)

        return true;

    return false;

}

void solve()

{

    ll n, c;

    cin >> n >> c;

    vl v(n);

    vecIn(v);

    sort(all(v));

    int low = 1, high = v[n - 1] - v[0];

    while (low <= high)

    {

        ll mid = (low + high) >> 1;

        if (isPossible(v, n, c, mid))

        {

            low = mid + 1;

        }

        else

        {

            high = mid - 1;

        }

    }

    cout << high << "\n";

}

*67-* Max heap, Min Heap Implementation (Only for interviews)

#include <bits/stdc++.h>

using namespace std;

int left(int k)

{

    return 2 \* k + 1;

}

// Right child of the node.

int right(int k)

{

    return 2 \* k + 2;

}

// Returns the parent node.

int parent(int k)

{

    return (k - 1) / 2;

}

// Heapify the heap

int heapify(vector<int> &heap, int k, int &size)

{

    // Find the left child of the node.

    int l = left(k);

    // Find the right child of the node.

    int r = right(k);

    // Find the smallest element between the current node and its children.

    // Check if the left child is smallest.

    int smallest = k;

    if (l < size && heap[l] < heap[k])

    {

        smallest = l;

    }

    // Check if the right node is smallest then the previous smallest.

    if (r < size && heap[r] < heap[smallest])

    {

        smallest = r;

    }

    // If the smallest element is not in the current node.

    // We have to heapify the Heap to take that element to the top.

    if (smallest != k)

    {

        // Swap the values of current node and the smallest node value.

        int tempp = heap[k];

        heap[k] = heap[smallest];

        heap[smallest] = tempp;

        // Call the heapify function on smallest value node which now contains the value of parent node.

        heapify(heap, smallest, size);

    }

}

// Insert a val in the heap.

// Function contains heap array, val to inserted and the current size of the heap.

void insert(vector<int> &heap, int val, int &size)

{

    // Insert the val at the end of the heap.

    heap[size] = val;

    // If There is nore than 1 node in the Heap.

    // MinHeapify the heap by checking the val at its parent node.

    // Also do it until the heap property is not satisfied.

    int i = size;

    size += 1;

    while (i != 0 && heap[parent(i)] > heap[i])

    {

        // Swap the value of current node with its parent.

        swap(heap[i], heap[parent(i)]);

        // Check that if the parent element of current element is satisfying the heap property.

        i = parent(i);

    }

}

int extractMin(vector<int> &heap, int &size)

{

    // Check if the current node is the only node in the heap.

    if (size == 1)

    {

        size -= 1;

        return heap[0];

    }

    // Takeout the min value and remove it from the heap.

    int val = heap[0];

    // Put last node on the top of heap.

    heap[0] = heap[size - 1];

    // Decrease the size of heap as the minimum element is removed.

    size -= 1;

    // Heapify the heap to satisfy the heap property.

    heapify(heap, 0, size);

    return val;

}

// minHeap function which take size of Queries and Queries as Input.

// Returns an array out outputs depending on the query.

vector<int> minHeap(int n, vector<vector<int>> &q)

{

    int size = 0;

    vector<int> heap(n);

    // Define an array which stores the min elements.

    vector<int> ans;

    // For each query in the array Q.

    for (int i = 0; i < n; i++)

    {

        // If query is of type 1 then insert the value in the heap.

        // Else take min element from the heap and append it in the ans.

        if (q[i][0] == 0)

        {

            insert(heap, q[i][1], size);

        }

        else

        {

            ans.push\_back(extractMin(heap, size));

        }

    }

    // Return the ans array.

    return ans;

}

*68-Kth Largest Element*

**Problem Statement**: Given an unsorted array, print Kth Largest and Smallest Element from an unsorted array.

int partition(vector<int> &arr, int left, int right)

    {

        int pivot = arr[left];

        int l = left + 1;

        int r = right;

        while (l <= r)

        {

            if (arr[l] < pivot && arr[r] > pivot)

            {

                swap(arr[l], arr[r]);

                l++;

                r--;

            }

            if (arr[l] >= pivot)

            {

                l++;

            }

            if (arr[r] <= pivot)

            {

                r--;

            }

        }

        swap(arr[left], arr[r]);

        return r;

    }

    int findKthLargest(vector<int> &arr, int k)

    {

        int left = 0, right = arr.size() - 1, kth;

        while (1)

        {

            int idx = partition(arr, left, right);

            if (idx == k - 1)

            {

                kth = arr[idx];

                break;

            }

            if (idx < k - 1)

            {

                left = idx + 1;

            }

            else

            {

                right = idx - 1;

            }

        }

        return kth;

    }

*69-Implement Stack using Arrays*

#include <bits/stdc++.h>

using namespace std;

// Stack class.

class Stack

{

public:

    vector<int> st;

    int t;

    Stack(int capacity)

    {

        st.resize(capacity);

        t = -1;

    }

    void push(int num)

    {

        // Write your code here.

        if (t + 1 < st.size())

            st[t + 1] = num, t++;

    }

    int pop()

    {

        // Write your code here.

        if (t >= 0)

        {

            int x = st[t];

            t--;

            return x;

        }

        else

            return -1;

    }

    int top()

    {

        // Write your code here.

        if (t >= 0)

        {

            return st[t];

        }

        else

            return -1;

    }

    int isEmpty()

    {

        // Write your code here.

        if (t == -1)

            return 1;

        else

            return 0;

    }

    int isFull()

    {

        // Write your code here.

        if (t == st.size())

            return 1;

        else

            return 0;

    }

};

*70-Implement Queue using array*

#include <bits/stdc++.h>

using namespace std;

class Queue

{

public:

    int size;

    int \*arr;

    int fr, bc;

    Queue()

    {

        size = 1e5;

        arr = new int[size];

        fr = -1;

        bc = -1;

    }

    /\*----------------- Public Functions of Queue -----------------\*/

    bool isEmpty()

    {

        // Implement the isEmpty() function

        if ((fr == -1 && bc == -1) || fr > bc)

            return true;

        else

            return false;

    }

    void enqueue(int data)

    {

        // Implement the enqueue() function

        arr[bc + 1] = data;

        bc++;

        if (fr == -1)

            fr++;

    }

    int dequeue()

    {

        // Implement the dequeue() function

        if (fr <= bc && fr != -1)

        {

            int x = arr[fr];

            fr++;

            return x;

        }

        else

            return -1;

    }

    int front()

    {

        // Implement the front() function

        if (fr <= bc && fr != -1)

            return arr[fr];

        else

            return -1;

    }

};

*71-Balanced Paranthesis*

    bool isValid(string s) {

        stack<char> st;

        for(auto x:s){

            if(x=='(' || x=='{' || x=='[')

            st.push(x);

            else {

                if(!st.empty()){

                if(x==')' && st.top()=='(')

                st.pop();

                else if(x=='}' && st.top()=='{')

                st.pop();

                else if(x==']' && st.top()=='[')

                st.pop();

                else

                return false;

                }

                else

                return false;

            }

        }

        if(st.empty())

        return true;

        else

        return false;

    }

*72-Next Greater Element*

**Problem Statement:** Given a circular integer array **A**, return the next greater element for every element in A. The next greater element for an element x is the first element greater than x that we come across while traversing the array in a clockwise manner. If it doesn’t exist, return -1 for this element.

vector<int> nextGreaterElements(vector<int> &nums)

{

    int n = nums.size();

    vector<int> ans(n, -1);

    stack<int> st;

    for (int i = 2 \* n - 1; i >= 0; i--)

    {

        while (!st.empty() && st.top() <= nums[i % n])

            st.pop();

        if (i < n)

        {

            if (!st.empty())

                ans[i] = st.top();

        }

        st.push(nums[i % n]);

    }

    return ans;

}

*73-Sort a Stack using Recursion*

void insert(stack<int> &s,int temp){

    if(s.size()==0 || s.top()<=temp){

        s.push(temp);

        return;

    }

    int t=s.top();

    s.pop();

    insert(s,temp);

    s.push(t);

}

void sortStack(stack<int> &s)

{

    if(s.size()==1)

    return ;

    int temp=s.top();

    s.pop();

    sortStack(s);

    insert(s,temp);

}

*74-Next Smaller element*

Given an array, find the **nearest** smaller element G[i] for every element A[i] in the array such that the element has an **index smaller than i**.

More formally,

G[i] for an element A[i] = an element A[j] such that

j is maximum possible AND

j < i AND

A[j] < A[i]

Elements for which no smaller element exist, consider next smaller element as -1.

vector<int> Solution::prevSmaller(vector<int> &A)

{

    stack<int> s;

    vector<int> ans(A.size(), -1);

    for (int i = 0; i < A.size(); i++)

    {

        while (!s.empty() && s.top() >= A[i])

            s.pop();

        if (!s.empty())

            ans[i] = s.top();

        s.push(A[i]);

    }

    return ans;

}

*75-LRU Cache*

**Problem Statement: “**Design a data structure that follows the constraints of **Least Recently Used (LRU) cache**”.Implement the **LRUCache** class:

**LRUCache(int capacity)** we need to initialize the LRU cache with positive size **capacity**.

**int get(int key)** returns the value of the **key** if the key exists, otherwise return**-1**.

* **Void put(int key,int value),** Update the value of the**key** if the **key** exists. Otherwise, add the **key-value** pair to the cache.if the number of keys exceeds the **capacity** from this operation, evict the least recently used key.

The functions **get** and **put** must each run in**O(1)** average time complexity.

class LRUCache

{

public:

    class node

    {

    public:

        int key;

        int val;

        node \*next;

        node \*prev;

        node(int \_key, int \_val)

        {

            key = \_key;

            val = \_val;

        }

    };

    node \*head = new node(-1, -1);

    node \*tail = new node(-1, -1);

    int cap;

    unordered\_map<int, node \*> m;

    LRUCache(int capacity)

    {

        cap = capacity;

        head->next = tail;

        tail->prev = head;

    }

    void addnode(node \*newnode)

    {

        node \*temp = head->next;

        newnode->next = temp;

        newnode->prev = head;

        head->next = newnode;

        temp->prev = newnode;

    }

    void deletenode(node \*delnode)

    {

        node \*delprev = delnode->prev;

        node \*delnext = delnode->next;

        delprev->next = delnext;

        delnext->prev = delprev;

    }

    int get(int key\_)

    {

        if (m.find(key\_) != m.end())

        {

            node \*resnode = m[key\_];

            int res = resnode->val;

            m.erase(key\_);

            deletenode(resnode);

            addnode(resnode);

            m[key\_] = head->next;

            return res;

        }

        return -1;

    }

    void put(int key\_, int value)

    {

        if (m.find(key\_) != m.end())

        {

            node \*existingnode = m[key\_];

            m.erase(key\_);

            deletenode(existingnode);

        }

        if (m.size() == cap)

        {

            m.erase(tail->prev->key);

            deletenode(tail->prev);

        }

        addnode(new node(key\_, value));

        m[key\_] = head->next;

    }

};

*76-LFU Cache*

Design and implement a data structure for a [Least Frequently Used (LFU)](https://en.wikipedia.org/wiki/Least_frequently_used) cache.

Implement the LFUCache class:

* LFUCache(int capacity) Initializes the object with the capacity of the data structure.
* int get(int key) Gets the value of the key if the key exists in the cache. Otherwise, returns -1.
* void put(int key, int value) Update the value of the key if present, or inserts the key if not already present. When the cache reaches its capacity, it should invalidate and remove the **least frequently used** key before inserting a new item. For this problem, when there is a **tie** (i.e., two or more keys with the same frequency), the **least recently used** key would be invalidated.

To determine the least frequently used key, a **use counter** is maintained for each key in the cache. The key with the smallest **use counter** is the least frequently used key.

When a key is first inserted into the cache, its **use counter** is set to 1 (due to the put operation). The **use counter** for a key in the cache is incremented either a get or put operation is called on it.

The functions get and put must each run in O(1) average time complexity.

struct Node

{

    int key, value, cnt;

    Node \*next;

    Node \*prev;

    Node(int \_key, int \_value)

    {

        key = \_key;

        value = \_value;

        cnt = 1;

    }

};

struct List

{

    int size;

    Node \*head;

    Node \*tail;

    List()

    {

        head = new Node(0, 0);

        tail = new Node(0, 0);

        head->next = tail;

        tail->prev = head;

        size = 0;

    }

    void addFront(Node \*node)

    {

        Node \*temp = head->next;

        node->next = temp;

        node->prev = head;

        head->next = node;

        temp->prev = node;

        size++;

    }

    void removeNode(Node \*delnode)

    {

        Node \*delprev = delnode->prev;

        Node \*delnext = delnode->next;

        delprev->next = delnext;

        delnext->prev = delprev;

        size--;

    }

};

class LFUCache

{

    map<int, Node \*> keyNode;

    map<int, List \*> freqListMap;

    int maxSizeCache;

    int minFreq;

    int curSize;

public:

    LFUCache(int capacity)

    {

        maxSizeCache = capacity;

        minFreq = 0;

        curSize = 0;

    }

    void updateFreqListMap(Node \*node)

    {

        keyNode.erase(node->key);

        freqListMap[node->cnt]->removeNode(node);

        if (node->cnt == minFreq && freqListMap[node->cnt]->size == 0)

        {

            minFreq++;

        }

        List \*nextHigherFreqList = new List();

        if (freqListMap.find(node->cnt + 1) != freqListMap.end())

        {

            nextHigherFreqList = freqListMap[node->cnt + 1];

        }

        node->cnt += 1;

        nextHigherFreqList->addFront(node);

        freqListMap[node->cnt] = nextHigherFreqList;

        keyNode[node->key] = node;

    }

    int get(int key)

    {

        if (keyNode.find(key) != keyNode.end())

        {

            Node \*node = keyNode[key];

            int val = node->value;

            updateFreqListMap(node);

            return val;

        }

        return -1;

    }

    void put(int key, int value)

    {

        if (maxSizeCache == 0)

        {

            return;

        }

        if (keyNode.find(key) != keyNode.end())

        {

            Node \*node = keyNode[key];

            node->value = value;

            updateFreqListMap(node);

        }

        else

        {

            if (curSize == maxSizeCache)

            {

                List \*list = freqListMap[minFreq];

                keyNode.erase(list->tail->prev->key);

                freqListMap[minFreq]->removeNode(list->tail->prev);

                curSize--;

            }

            curSize++;

            // new value has to be added who is not there previously

            minFreq = 1;

            List \*listFreq = new List();

            if (freqListMap.find(minFreq) != freqListMap.end())

            {

                listFreq = freqListMap[minFreq];

            }

            Node \*node = new Node(key, value);

            listFreq->addFront(node);

            keyNode[key] = node;

            freqListMap[minFreq] = listFreq;

        }

    }

};

*77-Largest Rectangle in a Histogram*

**Problem Statement:** Given an array of integers heights representing the histogram’s bar height where the width of each bar is 1  return the area of the largest rectangle in histogram.

Sol-1

class Solution

{

public:

    int largestRectangleArea(vector<int> &h)

    {

        int n = h.size();

        vector<int> left(n), right(n);

        stack<int> s;

        for (int i = 0; i < n; i++)

        {

            while (!s.empty() && h[s.top()] >= h[i])

                s.pop();

            if (s.empty())

                left[i] = 0;

            else

                left[i] = s.top() + 1;

            s.push(i);

        }

        while (!s.empty())

            s.pop();

        for (int i = n - 1; i >= 0; i--)

        {

            while (!s.empty() && h[s.top()] >= h[i])

                s.pop();

            if (s.empty())

                right[i] = n - 1;

            else

                right[i] = s.top() - 1;

            s.push(i);

        }

        for (auto c : left)

            cout << c << " ";

        cout << "\n";

        for (auto c : right)

            cout << c << " ";

        cout << "\n";

        int ans = INT\_MIN;

        for (int i = 0; i < n; i++)

        {

            ans = max(ans, h[i] \* (right[i] - left[i] + 1));

        }

        return ans;

    }

};

*Sol-2*

class Solution {

public:

    int largestRectangleArea(vector<int>& h) {

        int n=h.size();

        stack<int> s;

        int ans=INT\_MIN;

        for(int i=0;i<=n;i++){

            while(!s.empty() && (i==n || h[s.top()]>=h[i]))

            {

                int x=h[s.top()];

                s.pop();

                int w;

                if(s.empty())

                w=i;

                else

                w=i-s.top()-1;

                ans=max(ans,x\*w);

            }

            s.push(i);

        }

        return ans;

    }

};

*78-Sliding Window Maximum*

**Problem Statement:** Given an array of integers arr, there is a sliding window of size k which is moving from the very left of the array to the very right. You can only see the k numbers in the window. Each time the sliding window moves right by one position. Return the ***max sliding window***.

class Solution

{

public:

    vector<int> maxSlidingWindow(vector<int> &nums, int k)

    {

        deque<int> d;

        vector<int> ans;

        for (int i = 0; i < nums.size(); i++)

        {

            if (!d.empty() && d.front() == i - k)

                d.pop\_front();

            while (!d.empty() && nums[d.back()] <= nums[i])

                d.pop\_back();

            d.push\_back(i);

            if (i >= k - 1)

                ans.push\_back(nums[d.front()]);

        }

        return ans;

    }

};

*79-Implement Min Stack*

**Problem Statement:** Implement Min Stack | O(2N) and O(N) Space Complexity. Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Sol-1 =O(2N)

class MinStack

{

public:

    stack<pair<int, int>> obj;

    MinStack() {}

    void push(int val)

    {

        if (obj.size() == 0)

            obj.push({val, val});

        else

        {

            auto x = obj.top();

            obj.push({val, min(val, x.second)});

        }

    }

    void pop()

    {

        obj.pop();

    }

    int top()

    {

        return obj.top().first;

    }

    int getMin()

    {

        return obj.top().second;

    }

};

*Sol-2=O(N)*

class MinStack

{

    stack<long long> st;

    long long mini;

public:

    MinStack()

    {

        while (st.empty() == false)

            st.pop();

        mini = INT\_MAX;

    }

    void push(int value)

    {

        long long val = value;

        if (st.empty())

        {

            mini = val;

            st.push(val);

        }

        else

        {

            if (val < mini)

            {

                st.push(2 \* val \* 1LL - mini);

                mini = val;

            }

            else

            {

                st.push(val);

            }

        }

    }

    void pop()

    {

        if (st.empty())

            return;

        long long el = st.top();

        st.pop();

        if (el < mini)

        {

            mini = 2 \* mini - el;

        }

    }

    int top()

    {

        if (st.empty())

            return -1;

        long long el = st.top();

        if (el < mini)

            return mini;

        return el;

    }

    int getMin()

    {

        return mini;

    }

};

*80-Rotten Oranges*

**Problem Statement**: You will be given an **m x n** grid, where each cell has the following values :

1. 2  –  represents a rotten orange
2. 1  –  represents a Fresh orange
3. 0  –  represents an Empty Cell

Every minute, if a Fresh Orange is adjacent to a Rotten Orange in 4-direction ( upward, downwards, right, and left ) it becomes Rotten.

Return the minimum number of minutes required such that none of the cells has a Fresh Orange. If it’s not possible, return **-1.**

class Solution

{

public:

    int orangesRotting(vector<vector<int>> &grid)

    {

        queue<pair<int, int>> q;

        int fresh = 0;

        int ans = -1;

        for (int i = 0; i < grid.size(); i++)

        {

            for (int j = 0; j < grid[0].size(); j++)

            {

                if (grid[i][j] == 2)

                    q.push({i, j});

                else if (grid[i][j] == 1)

                    fresh++;

            }

        }

        if (fresh == 0)

            return 0;

        while (!q.empty())

        {

            int n = q.size();

            ans++;

            while (n--)

            {

                auto x = q.front();

                q.pop();

                int row[] = {0, 1, -1, 0};

                int col[] = {1, 0, 0, -1};

                for (int i = 0; i < 4; i++)

                {

                    int r = x.first + row[i];

                    int c = x.second + col[i];

                    if (r >= 0 && r < grid.size() && c >= 0 && c < grid[0].size() &&

                        grid[r][c] == 1)

                    {

                        grid[r][c] = 2;

                        q.push({r, c});

                        fresh--;

                    }

                }

            }

        }

        if (fresh != 0)

            return -1;

        if (ans == -1)

            return ans;

        else if (ans != -1 && fresh == 0)

            ;

        return ans;

    }

};

*81-Reverse Words in a String*

*Given an input string s, reverse the order of the words. A word is defined as a sequence of non-space characters. The words in s will be separated by at least one space. Return a string of the words in reverse order concatenated by a single space. Note that s may contain leading or trailing spaces or multiple spaces between two words. The returned string should only have a single space separating the words. Do not include any extra spaces.*

class Solution

{

public:

    string reverseWords(string s)

    {

        string ans = "";

        string temp = "";

        s = s + " ";

        for (auto x : s)

        {

            if (x != ' ')

                temp += x;

            else if (x == ' ' && temp != "")

            {

                if (ans == "")

                    ans = temp;

                else

                    ans = temp + " " + ans;

                temp = "";

            }

        }

        return ans;

    }

};

*82-Longest Palindromic Substring*

*Given a string s, return the longest palindromic substring in s.*

class Solution

{

public:

    int LCS(string s, int l, int r)

    {

        while (l >= 0 && r <= s.length() - 1)

        {

            if (s[l] == s[r])

            {

                l--;

                r++;

            }

            else

                break;

        }

        return r - l - 1;

    }

    string longestPalindrome(string a)

    {

        int n = a.length();

        int ans = 0;

        int idx = 0;

        for (int i = 0; i < n; i++)

        {

            int l1 = LCS(a, i, i);

            int l2 = LCS(a, i, i + 1);

            int len = max(l1, l2);

            if (len > ans)

            {

                ans = len;

                idx = i - (len - 1) / 2;

            }

        }

        return a.substr(idx, ans);

    }

};

*83-Roman to Integer*

*Roman numerals are represented by seven different symbols: I, V, X, L, C, D and M*

class Solution

{

public:

    int romanToInt(string s)

    {

        unordered\_map<char, int> T = {{'I', 1}, {'V', 5}, {'X', 10},

        {'L', 50}, {'C', 100}, {'D', 500}, {'M', 1000}};

        int sum = T[s.back()];

        for (int i = s.length() - 2; i >= 0; --i)

        {

            if (T[s[i]] < T[s[i + 1]])

            {

                sum -= T[s[i]];

            }

            else

            {

                sum += T[s[i]];

            }

        }

        return sum;

    }

};

*84-Longest Common Prefix*

*Write a function to find the longest common prefix string amongst an array of strings. If there is no common prefix, return an empty string "".*

class Solution

{

public:

    string longestCommonPrefix(vector<string> &s)

    {

        sort(s.begin(), s.end());

        int n = s.size();

        string ans = "";

        int i = 0;

        while (true)

        {

            if (i < min(s[0].length(), s[n - 1].length()) && s[0][i] == s[n - 1][i])

                ans += s[0][i];

            else

                break;

            i++;

        }

        return ans;

    }

};

*85-String to Integer (ATOI)*

*Implement the myAtoi(string s) function, which converts a string to a 32-bit signed integer (similar to C/C++'s atoi function). The algorithm for myAtoi(string s) is as follows: Read in and ignore any leading whitespace. Check if the next character (if not already at the end of the string) is '-' or '+'. Read this character in if it is either. This determines if the final result is negative or positive respectively. Assume the result is positive if neither is present. Read in next the characters until the next non-digit character or the end of the input is reached. The rest of the string is ignored. Convert these digits into an integer (i.e. "123" -> 123, "0032" -> 32). If no digits were read, then the integer is 0. Change the sign as necessary (from step 2). If the integer is out of the 32-bit signed integer range [-231, 231 - 1], then clamp the integer so that it remains in the range. Specifically, integers less than -231 should be clamped to -231, and integers greater than 231 - 1 should be clamped to 231 - 1. Return the integer as the final result. Note: Only the space character ' ' is considered a whitespace character. Do not ignore any characters other than the leading whitespace or the rest of the string after the digits.*

class Solution

{

public:

    int myAtoi(string s)

    {

        const int len = s.size();

        if (len == 0)

        {

            return 0;

        }

        int index = 0;

        while (index < len && s[index] == ' ')

        {

            ++index;

        }

        bool isNegative = false;

        if (index < len)

        {

            if (s[index] == '-')

            {

                isNegative = true;

                ++index;

            }

            else if (s[index] == '+')

            {

                ++index;

            }

        }

        int result = 0;

        while (index < len && isDigit(s[index]))

        {

            int digit = s[index] - '0';

            if (result > (INT\_MAX / 10) || (result == (INT\_MAX / 10) && digit > 7))

            {

                return isNegative ? INT\_MIN : INT\_MAX;

            }

            result = (result \* 10) + digit;

            ++index;

        }

        return isNegative ? -result : result;

    }

private:

    bool isDigit(char ch)

    {

        return ch >= '0' && ch <= '9';

    }

};

*86-Index of first occurrence of a string*

*Given two strings ‘s’ and ‘h’, return the index of the first occurrence of ‘s’ in ‘h’, or -1 if ‘s’ is not part of ‘h’.*

class Solution

{

public:

    int strStr(string h, string s)

    {

        if (s.length() > h.length())

            return -1;

        string ans = "";

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

            ans += h[i];

        if (s == ans)

            return 0;

        else

        {

            for (int i = 0; i < h.length() - s.length() + 1; i++)

            {

                ans.erase(0, 1);

                ans += h[i + s.length()];

                if (ans == s)

                    return i + 1;

            }

            return -1;

        }

    }

};

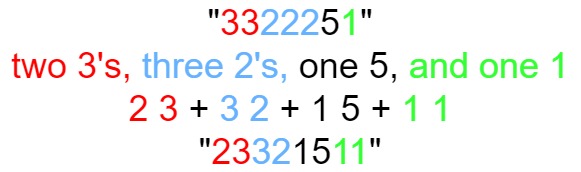
*87-Count and Say*

*The count-and-say sequence is a sequence of digit strings defined by the recursive formula:*

*countAndSay(1) = "1",*

*countAndSay(n) is the way you would "say" the digit string from countAndSay(n-1), which is then converted into a different digit string.To determine how you "say" a digit string, split it into the minimal number of substrings such that each substring contains exactly one unique digit. Then for each substring, say the number of digits, then say the digit. Finally, concatenate every said digit.*

*For example, the saying and conversion for digit string "3322251":*

**

*Given a positive integer n, return the nth term of the count-and-say sequence.*

class Solution

{

public:

    string countAndSay(int n)

    {

        if (n == 1)

            return "1";

        n--;

        string ans = "1";

        while (n--)

        {

            int i = 0;

            string temp = "";

            while (i < ans.length())

            {

                int cnt = 1;

                while (i < ans.length() - 1 && ans[i] == ans[i + 1])

                    i++, cnt++;

                temp += to\_string(cnt) + ans[i];

                i++;

            }

            ans = temp;

        }

        return ans;

    }

};

*88-Compare Version numbers*

*Given two version numbers, version1 and version2, compare them. Version numbers consist of one or more revisions joined by a dot '.'. Each revision consists of digits and may contain leading zeros. Every revision contains at least one character. Revisions are 0-indexed from left to right, with the leftmost revision being revision 0, the next revision being revision 1, and so on. For example 2.5.33 and 0.1 are valid version numbers. To compare version numbers, compare their revisions in left-to-right order. Revisions are compared using their integer value ignoring any leading zeros. This means that revisions 1 and 001 are considered equal. If a version number does not specify a revision at an index, then treat the revision as 0. For example, version 1.0 is less than version 1.1 because their revision 0s are the same, but their revision 1s are 0 and 1 respectively, and 0 < 1. Return the following: If version1 < version2, return -1. If version1 > version2, return 1. Otherwise, return 0.*

class Solution

{

public:

    int compareVersion(string v1, string v2)

    {

        int i = 0;

        int j = 0;

        int n1 = v1.size();

        int n2 = v2.size();

        int num1 = 0;

        int num2 = 0;

        while (i < n1 || j < n2)

        {

            while (i < n1 && v1[i] != '.')

            {

                num1 = num1 \* 10 + (v1[i] - '0');

                i++;

            }

            while (j < n2 && v2[j] != '.')

            {

                num2 = num2 \* 10 + (v2[j] - '0');

                j++;

            }

            if (num1 > num2)

                return 1;

            else if (num1 < num2)

                return -1;

            num1 = 0;

            num2 = 0;

            i++;

            j++;

        }

        return 0;

    }

};

*89-left View of BT*

vector<int> leftView(Node \*root)

{

    vector<int> ans;

    if (!root)

        return ans;

    queue<Node \*> q;

    q.push(root);

    while (!q.empty())

    {

        int n = q.size();

        ans.push\_back(q.front()->data);

        while (n--)

        {

            Node \*node = q.front();

            q.pop();

            if (node->left)

                q.push(node->left);

            if (node->right)

                q.push(node->right);

        }

    }

    return ans;

}

*90-Bottom View of BT*

class Solution

{

public:

    vector<int> bottomView(Node \*root)

    {

        if (!root)

            return {};

        map<int, int> mp;

        queue<pair<Node \*, int>> q;

        q.push({root, 0});

        while (!q.empty())

        {

            Node \*x = q.front().first;

            int d = q.front().second;

            q.pop();

            mp[d] = x->data;

            if (x->left)

                q.push({x->left, d - 1});

            if (x->right)

                q.push({x->right, d + 1});

        }

        vector<int> ans;

        for (auto x : mp)

            ans.push\_back(x.second);

        return ans;

    }

};

*91-Top View*

class Solution

{

public:

    vector<int> topView(Node \*root)

    {

        if (!root)

            return {};

        map<int, int> mp;

        queue<pair<Node \*, int>> q;

        q.push({root, 0});

        while (!q.empty())

        {

            Node \*x = q.front().first;

            int d = q.front().second;

            q.pop();

            if (!mp.count(d))

            {

                mp[d] = x->data;

            }

            if (x->left)

                q.push({x->left, d - 1});

            if (x->right)

                q.push({x->right, d + 1});

        }

        vector<int> ans;

        for (auto x : mp)

            ans.push\_back(x.second);

        return ans;

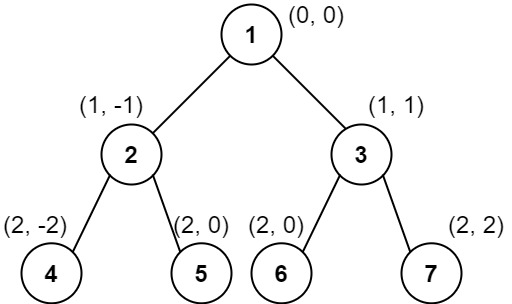
    }

};

*92-Vertical order traversal of a BT*

*Given the root of a binary tree, calculate the vertical order traversal of the binary tree.*

*For each node at position (row, col), its left and right children will be at positions (row + 1, col - 1) and (row + 1, col + 1) respectively. The root of the tree is at (0, 0). The vertical order traversal of a binary tree is a list of top-to-bottom orderings for each column index starting from the leftmost column and ending on the rightmost column. There may be multiple nodes in the same row and same column. In such a case, sort these nodes by their values. Return the vertical order traversal of the binary tree*

*.*

class Solution

{

public:

    vector<vector<int>> verticalTraversal(TreeNode \*root)

    {

        map<int, map<int, multiset<int>>> nodes;

        queue<pair<TreeNode \*, pair<int, int>>> todo;

        todo.push({root, {0, 0}});

        while (!todo.empty())

        {

            auto p = todo.front();

            todo.pop();

            TreeNode \*node = p.first;

            int x = p.second.first, y = p.second.second;

            nodes[x][y].insert(node->val);

            if (node->left)

            {

                todo.push({node->left, {x - 1, y + 1}});

            }

            if (node->right)

            {

                todo.push({node->right, {x + 1, y + 1}});

            }

        }

        vector<vector<int>> ans;

        for (auto p : nodes)

        {

            vector<int> col;

            for (auto q : p.second)

            {

                col.insert(col.end(), q.second.begin(), q.second.end());

            }

            ans.push\_back(col);

        }

        return ans;

    }

};

*93-Root to node path in BT*

void helper(TreeNode \*root, int t, vector<int> &ans, vector<int> temp)

{

    if (!root)

        return;

    if (root->val == t)

    {

        temp.push\_back(root->val);

        ans = temp;

        return;

    }

    temp.push\_back(root->val);

    helper(root->left, t, ans, temp);

    helper(root->right, t, ans, temp);

    temp.pop\_back();

}

vector<int> Solution::solve(TreeNode \*A, int B)

{

    vector<int> ans, temp;

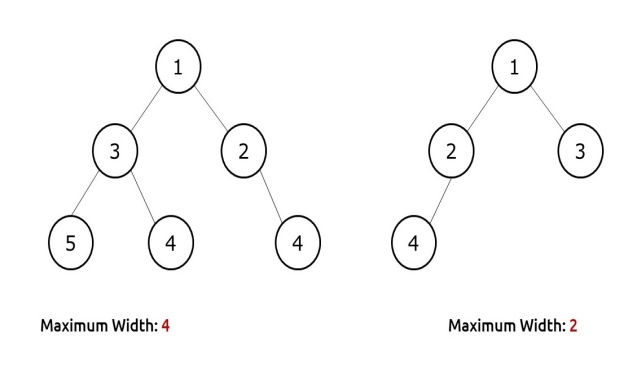
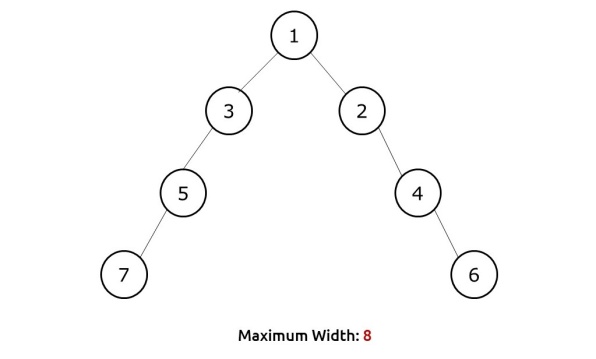
    helper(A, B, ans, temp);

    return ans;

}

*94-Max Width of a BT*

**Problem Statement:**Write a program to find the **Maximum Width of A Binary Tree**.

****

class Solution

{

public:

    long long widthOfBinaryTree(TreeNode \*root)

    {

        unsigned ans = 0;

        deque<pair<TreeNode \*, unsigned>> q;

        q.push\_back({root, 1});

        while (!q.empty())

        {

            int size = q.size();

            ans = max(ans, q.back().second - q.front().second + 1);

            for (int i = 1; i <= size; i++)

            {

                pair<TreeNode \*, unsigned> node = q.front();

                q.pop\_front();

                if (node.first->left)

                    q.push\_back({node.first->left, 2 \* node.second});

                if (node.first->right)

                    q.push\_back({node.first->right, 2 \* node.second + 1});

            }

        }

        return ans;

    }

};

*95-Level order traversal of BT*

class Solution

{

public:

    vector<vector<int>> levelOrder(TreeNode \*root)

    {

        vector<vector<int>> ans;

        if (!root)

            return ans;

        queue<TreeNode \*> q;

        q.push(root);

        while (!q.empty())

        {

            int size = q.size();

            vector<int> temp;

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

            {

                TreeNode \*node = q.front();

                temp.push\_back(node->val);

                q.pop();

                if (node->left)

                    q.push(node->left);

                if (node->right)

                    q.push(node->right);

            }

            ans.push\_back(temp);

            temp.clear();

        }

        return ans;

    }

};

*96-Height of a BT*

class Solution

{

public:

    int maxDepth(TreeNode \*root)

    {

        if (!root)

            return 0;

        return max(maxDepth(root->left), maxDepth(root->right)) + 1;

    }

};

*97-Diameter of a BT*

class Solution

{

public:

    int diameter(TreeNode \*root, int &dia)

    {

        if (!root)

            return 0;

        int l = diameter(root->left, dia);

        int r = diameter(root->right, dia);

        dia = max(dia, l + r);

        return 1 + max(l, r);

    }

    int diameterOfBinaryTree(TreeNode \*root)

    {

        if (!root)

            return 0;

        int dia = 0;

        diameter(root, dia);

        return dia;

    }

};

*98-Check if a BT is Height balanced or not*

class Solution

{

public:

    int balance(TreeNode \*root)

    {

        if (!root)

            return 0;

        int l = balance(root->left);

        if (l == -1)

            return -1;

        int r = balance(root->right);

        if (r == -1)

            return -1;

        if (abs(l - r) > 1)

            return -1;

        return 1 + max(l, r);

    }

    bool isBalanced(TreeNode \*root)

    {

        return balance(root) != -1;

    }

};

*99-LCA*

class Solution

{

public:

    TreeNode \*lowestCommonAncestor(TreeNode \*root, TreeNode \*p, TreeNode \*q)

    {

        if (!root || root == p || root == q)

            return root;

        TreeNode \*l = lowestCommonAncestor(root->left, p, q);

        TreeNode \*r = lowestCommonAncestor(root->right, p, q);

        if (l == NULL)

            return r;

        else if (r == NULL)

            return l;

        else

            return root;

    }

};

*100-Check if two trees are identical or not*

class Solution

{

public:

    bool isSameTree(TreeNode \*p, TreeNode \*q)

    {

        if (p && q)

        {

            if (p->val == q->val)

                return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);

            else

                return false;

        }

        else if (!p && !q)

            return true;

        else

            return false;

    }

};

*101-Zig-Zag Traversal of BT*

class Solution

{

public:

    vector<vector<int>> zigzagLevelOrder(TreeNode \*root)

    {

        vector<vector<int>> ans;

        if (root == NULL)

            return ans;

        queue<TreeNode \*> q;

        bool leftToRight = true;

        q.push(root);

        while (!q.empty())

        {

            int size = q.size();

            vector<int> row(size);

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

            {

                TreeNode \*front = q.front();

                q.pop();

                int index;

                if (leftToRight)

                    index = i;

                else

                    index = size - 1 - i;

                row[index] = front->val;

                if (front->left)

                {

                    q.push(front->left);

                }

                if (front->right)

                {

                    q.push(front->right);

                }

            }

            leftToRight = !leftToRight;

            ans.push\_back(row);

        }

        return ans;

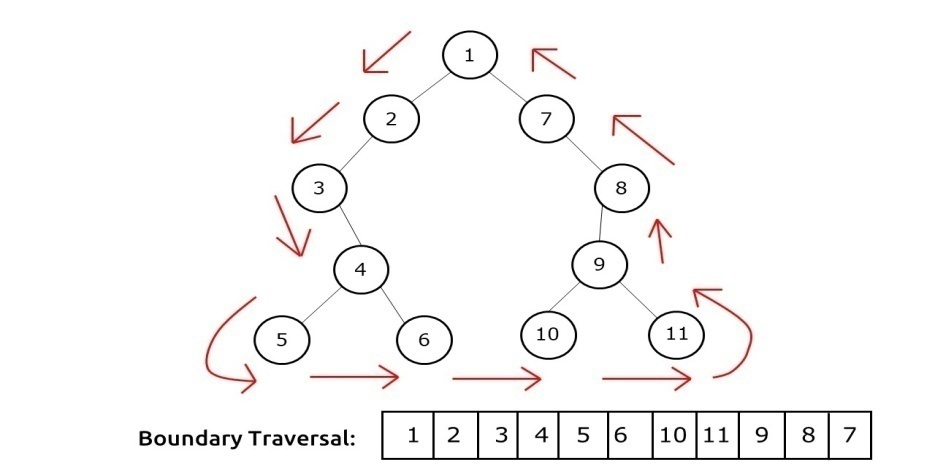
    }

};

*102-Boundary Traversal of a BT*

**Problem Statement: BoundaryTraversal of a binary tree**. Write a program for the Anti-Clockwise Boundary traversal of a binary tree.

**Example:**



#include <bits/stdc++.h>

bool isLeaf(TreeNode<int> \*root)

{

    if (root)

        return !root->left && !root->right;

    return false;

}

void leftview(TreeNode<int> \*root, vector<int> &ans)

{

    TreeNode<int> \*temp = root->left;

    while (temp)

    {

        if (!isLeaf(temp))

            ans.push\_back(temp->data);

        if (temp->left)

            temp = temp->left;

        else

            temp = temp->right;

    }

}

void leaf(TreeNode<int> \*root, vector<int> &ans)

{

    if (!root)

        return;

    if (isLeaf(root))

    {

        ans.push\_back(root->data);

        return;

    }

    leaf(root->left, ans);

    leaf(root->right, ans);

}

void rightview(TreeNode<int> \*root, vector<int> &ans)

{

    TreeNode<int> \*temp = root->right;

    stack<int> st;

    while (temp)

    {

        if (!isLeaf(temp))

            st.push(temp->data);

        if (temp->right)

            temp = temp->right;

        else

            temp = temp->left;

    }

    while (!st.empty())

    {

        ans.push\_back(st.top());

        st.pop();

    }

}

vector<int> traverseBoundary(TreeNode<int> \*root)

{

    vector<int> ans;

    if (!root)

        return ans;

    if (!isLeaf(root))

        ans.push\_back(root->data);

    leftview(root, ans);

    leaf(root, ans);

    rightview(root, ans);

    return ans;

}

*103-Maximum Sum Path in a BT*

*Write a program to find the maximum sum path in a binary tree. A path in a binary tree is a sequence of nodes where every adjacent pair of nodes are connected by an edge. A node can only appear in the sequence at most once. A path need not pass from the root. We need to find the path with the maximum sum in the binary tree.*

class Solution

{

public:

    int findMaxPathSum(TreeNode\*root, int &maxi)

    {

        if (root == NULL)

            return 0;

        int leftMaxPath = max(0, findMaxPathSum(root->left, maxi));

        int rightMaxPath = max(0, findMaxPathSum(root->right, maxi));

        maxi = max(maxi, (leftMaxPath + rightMaxPath) + root->val);

        return max(leftMaxPath, rightMaxPath) + root->val;

    }

    int maxPathSum(TreeNode \*root)

    {

        int maxi = INT\_MIN;

        findMaxPathSum(root, maxi);

        return maxi;

    }

};

*104-Construct BT from Preorder and Inorder*

class Solution

{

public:

    TreeNode \*helper(vector<int> &in, int inSt, int inEnd, vector<int> &pre, int preSt, int preEnd, unordered\_map<int, int> &mp)

    {

        if (inSt > inEnd || preSt > preEnd)

            return NULL;

        TreeNode \*root = new TreeNode(pre[preSt]);

        int pos = mp[root->val];

        int ele = pos - inSt;

        root->left = helper(in, inSt, pos - 1, pre, preSt + 1, preSt + ele, mp);

        root->right = helper(in, pos + 1, inEnd, pre, preSt + ele + 1, preEnd, mp);

        return root;

    }

    TreeNode \*buildTree(vector<int> &preorder, vector<int> &inorder)

    {

        unordered\_map<int, int> mp;

        for (int i = 0; i < inorder.size(); i++)

            mp[inorder[i]] = i;

        return helper(inorder, 0, inorder.size() - 1, preorder, 0, preorder.size(), mp);

    }

};

*105- Construct BT from Postorder and Inorder*

class Solution

{

public:

    TreeNode \*helper(vector<int> &in, int inSt, int inEnd, vector<int> &post,

    int posSt, int posEnd, unordered\_map<int, int> &mp)

    {

        if (inSt > inEnd || posSt > posEnd)

            return NULL;

        TreeNode \*root = new TreeNode(post[posEnd]);

        int pos = mp[root->val];

        int ele = pos - inSt;

        root->left = helper(in, inSt, pos - 1, post, posSt, posSt + ele - 1, mp);

        root->right = helper(in, pos + 1, inEnd, post, posSt + ele, posEnd - 1, mp);

        return root;

    }

    TreeNode \*buildTree(vector<int> &inorder, vector<int> &postorder)

    {

        if (postorder.size() != inorder.size())

            return NULL;

        unordered\_map<int, int> mp;

        for (int i = 0; i < inorder.size(); i++)

            mp[inorder[i]] = i;

        return helper(inorder, 0, inorder.size() - 1, postorder, 0,

                      postorder.size() - 1, mp);

    }

};

*106-Symmetric BT*

*Given the root of a binary tree, check whether it is a mirror of itself (i.e., symmetric around its center).*

**Example 1:**



class Solution

{

public:

    bool helper(TreeNode \*l, TreeNode \*r)

    {

        if (!l && !r)

            return true;

        if (!l || !r)

            return false;

        return l->val == r->val && helper(l->right, r->left) && helper(l->left, r->right);

    }

    bool isSymmetric(TreeNode \*root)

    {

        if (!root)

            return true;

        else

            return helper(root->left, root->right);

    }

};

*107-Flatten a BT to a Linked List*

class Solution

{

public:

    void preorder(TreeNode \*root, TreeNode \*&node)

    {

        if (!root)

            return;

        TreeNode \*left = root->left, \*right = root->right;

        if (!node)

        {

            node = root;

        }

        else

        {

            node->right = root;

            node->left = nullptr;

            node = node->right;

        }

        preorder(left, node);

        preorder(right, node);

    }

    void flatten(TreeNode \*root)

    {

        TreeNode \*node = nullptr;

        preorder(root, node);

    }

};

*108-Make Mirror of a BT*

class Solution

{

public:

    void helper(Node \*&root)

    {

        if (!root || (!root->left && !root->right))

            return;

        swap(root->left, root->right);

        helper(root->left);

        helper(root->right);

    }

    void mirror(Node \*node)

    {

        helper(node);

    }

};

# *109-Check For children Sum property*

**Problem Statement: Children Sum Property in a Binary Tree.** Write a program that converts any binary tree to one that follows the children sum property.

The children sum property is defined as, For every node of the tree, the value of a node is equal to the sum of values of its children(left child and right child).

**Note:**

* The node values can be increased by 1 any number of times but decrement of any node value is not allowed.
* A value for a NULL node can be assumed as 0.
* You are not allowed to change the structure of the given binary tree.

#include <bits/stdc++.h>

void reorder(BinaryTreeNode<int> \*root)

{

    if (root == NULL)

        return;

    int child = 0;

    if (root->left)

    {

        child += root->left->data;

    }

    if (root->right)

    {

        child += root->right->data;

    }

    if (child < root->data)

    {

        if (root->left)

            root->left->data = root->data;

        else if (root->right)

            root->right->data = root->data;

    }

    reorder(root->left);

    reorder(root->right);

    int tot = 0;

    if (root->left)

        tot += root->left->data;

    if (root->right)

        tot += root->right->data;

    if (root->left || root->right)

        root->data = tot;

}

void changeTree(BinaryTreeNode<int> \*root)

{

    reorder(root);

}

*110-Populating next Right pointer in each node*

*You are given a perfect binary tree where all leaves are on the same level, and every parent has two children. The binary tree has the following definition: struct Node { int val; Node \*left; Node \*right; Node \*next; } Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL. Initially, all next pointers are set to NULL. Example 1: Input: root = [1,2,3,4,5,6,7] Output: [1,#,2,3,#,4,5,6,7,#] Explanation: Given the above perfect binary tree (Figure A), your function should populate each next pointer to point to its next right node, just like in Figure B. The serialized output is in level order as connected by the next pointers, with '#' signifying the end of each level. Example 2: Input: root = [] Output: [] Constraints: The number of nodes in the tree is in the range [0, 212 - 1]. -1000 <= Node.val <= 1000 Follow-up: You may only use constant extra space. The recursive approach is fine. You may assume implicit stack space does not count as extra space for this problem.*



class Solution

{

public:

    Node \*connect(Node \*root)

    {

        if (!root)

            return root;

        queue<Node \*> q;

        q.push(root);

        while (!q.empty())

        {

            int n = q.size();

            while (n--)

            {

                Node \*node = q.front();

                q.pop();

                if (n != 0)

                    node->next = q.front();

                if (node->left)

                    q.push(node->left);

                if (node->right)

                    q.push(node->right);

            }

        }

        return root;

    }

};

*111-Search in a BST*

class Solution

{

public:

    void helper(TreeNode \*root, int val, TreeNode \*&ans)

    {

        if (!root)

            return;

        if (root->val == val)

        {

            ans = root;

            return;

        }

        if (root->val > val)

            helper(root->left, val, ans);

        else

            helper(root->right, val, ans);

    }

    TreeNode \*searchBST(TreeNode \*root, int val)

    {

        TreeNode \*ans = NULL;

        helper(root, val, ans);

        return ans;

    }

};

*112-Construct a BST*

class Solution

{

public:

    TreeNode \*Solve(vector<int> &a, int l, int r)

    {

        if (l > r)

            return nullptr;

        int mid = (l + r) / 2;

        TreeNode \*root = new TreeNode(a[mid]);

        root->left = Solve(a, l, mid - 1);

        root->right = Solve(a, mid + 1, r);

        return root;

    }

    TreeNode \*sortedArrayToBST(vector<int> &nums)

    {

        return Solve(nums, 0, nums.size() - 1);

    }

};

*113-Construct BST from preorder traversal array*

class Solution

{

public:

    TreeNode \*helper(vector<int> &pre, int st, int end)

    {

        if (st > end)

            return NULL;

        TreeNode \*root = new TreeNode(pre[st]);

        int k = st;

        for (int i = st + 1; i <= end; i++)

        {

            if (pre[st] > pre[i])

                k = i;

            else

                break;

        }

        root->left = helper(pre, st + 1, k);

        root->right = helper(pre, k + 1, end);

        return root;

    }

    TreeNode \*bstFromPreorder(vector<int> &preorder)

    {

        return helper(preorder, 0, preorder.size() - 1);

    }

};

*114-Check for BST*

class Solution

{

public:

    bool solve(TreeNode \*root, TreeNode \*min, TreeNode \*max)

    {

        if (!root)

            return 1;

        if (min && root->val <= min->val)

            return 0;

        if (max && root->val >= max->val)

            return 0;

        bool l = solve(root->left, min, root);

        bool r = solve(root->right, root, max);

        return l && r;

    }

    bool isValidBST(TreeNode \*root)

    {

        return solve(root, NULL, NULL);

    }

};

*115-LCA of Two nodes in a BST*

class Solution

{

public:

    TreeNode \*LCA(TreeNode \*root, TreeNode \*p, TreeNode \*q)

    {

        if (!root)

            return root;

        int rval = root->val;

        int pval = p->val;

        int qval = q->val;

        if ((rval >= pval && rval <= qval) || (rval >= qval && rval <= pval))

            return root;

        else if (rval > pval && rval > qval)

            return LCA(root->left, p, q);

        else

            return LCA(root->right, p, q);

    }

    TreeNode \*lowestCommonAncestor(TreeNode \*root, TreeNode \*p, TreeNode \*q)

    {

        return LCA(root, p, q);

    }

};

*116-Inorder Predecessor and Successor of a BST*

void findPreSuc(Node \*root, Node \*&pre, Node \*&suc, int key)

{

    pre = NULL;

    suc = NULL;

    Node \*temp = root;

    while (temp)

    {

        if (temp->key <= key)

            temp = temp->right;

        else

        {

            suc = temp;

            temp = temp->left;

        }

    }

    temp = root;

    while (temp)

    {

        if (temp->key >= key)

            temp = temp->left;

        else

        {

            pre = temp;

            temp = temp->right;

        }

    }

}

*117-Floor in a BST*

*You are given a BST of N nodes and a Value X.Find the greatest node value of BST which is smaller than or equal to X*

#include <bits/stdc++.h>

int floorInBST(TreeNode<int> \*root, int X)

{

    int ans = -1;

    while (root)

    {

        if (root->val > X)

            root = root->left;

        else

        {

            ans = root->val;

            root = root->right;

        }

    }

    return ans;

}

*118-Ceil in a BST*

*You are given a BST of N nodes and a Value X.Find the smallest node value of BST which Is greaterr than or equal to X*

#include <bits/stdc++.h>

int findCeil(BinaryTreeNode<int> \*node, int x)

{

    int ans = -1;

    while (node)

    {

        if (node->data < x)

            node = node->right;

        else

        {

            ans = node->data;

            node = node->left;

        }

    }

    return ans;

}

*119-K-smallest Element in BST*

class Solution

{

public:

    int ans;

    void inorder(TreeNode \*root, int &k)

    {

        if (!root)

            return;

        inorder(root->left, k);

        k--;

        if (k == 0)

        {

            ans = root->val;

            return;

        }

        inorder(root->right, k);

    }

    int kthSmallest(TreeNode \*root, int k)

    {

        if (!root)

            return 0;

        inorder(root, k);

        return ans;

    }

};

*120-Kth Largest Element in a BST*

class Solution

{

public:

    void inorder(Node \*root, int &ans, int &k)

    {

        if (!root)

            return;

        inorder(root->right, ans, k);

        k--;

        if (k == 0)

        {

            ans = root->data;

        }

        inorder(root->left, ans, k);

    }

    int kthLargest(Node \*root, int k)

    {

        int n = 0;

        inorder(root, n, k);

        return n;

    }

};

*121-Find Pair with a given sum in BST*

class Solution

{

public:

    bool search(TreeNode \*root, TreeNode \*cur, int k)

    {

        if (!root)

            return false;

        if (root->val == k && root != cur)

            return true;

        if (root->val > k)

            return search(root->left, cur, k);

        else

            return search(root->right, cur, k);

    }

    bool find(TreeNode \*root, TreeNode \*cur, int k)

    {

        if (!cur)

            return false;

        return search(root, cur, k - cur->val) || find(root, cur->left, k) || find(root, cur->right, k);

    }

    bool findTarget(TreeNode \*root, int k)

    {

        return find(root, root, k);

    }

};

*122-Binary Search Iterator*

*Implement the BSTIterator class that represents an iterator over the in-order traversal of a binary search tree (BST): BSTIterator(TreeNode root) Initializes an object of the BSTIterator class. The root of the BST is given as part of the constructor. The pointer should be initialized to a non-existent number smaller than any element in the BST. boolean hasNext() Returns true if there exists a number in the traversal to the right of the pointer, otherwise returns false. int next() Moves the pointer to the right, then returns the number at the pointer. Notice that by initializing the pointer to a non-existent smallest number, the first call to next() will return the smallest element in the BST. You may assume that next() calls will always be valid. That is, there will be at least a next number in the in-order traversal when next() is called.*

class BSTIterator

{

public:

    stack<TreeNode \*> st;

    BSTIterator(TreeNode \*root)

    {

        pushall(root);

    }

    int next()

    {

        TreeNode \*temp = st.top();

        st.pop();

        pushall(temp->right);

        return temp->val;

    }

    bool hasNext()

    {

        return st.size() > 0;

    }

    void pushall(TreeNode \*root)

    {

        if (!root)

            return;

        st.push(root);

        pushall(root->left);

    }

};

*123-Median of running stream of integers*

class MedianFinder

{

public:

    priority\_queue<long> small, large;

    MedianFinder() {}

    void addNum(int num)

    {

        small.push(num);

        large.push(-small.top());

        small.pop();

        while (small.size() < large.size())

        {

            small.push(-large.top());

            large.pop();

        }

    }

    double findMedian()

    {

        if (small.size() == large.size())

            return (small.top() - large.top()) / 2.0;

        else

            return small.top();

    }

};

*124-Kth Largest element in a stream*

class KthLargest

{

public:

    priority\_queue<int, vector<int>, greater<int>> pq;

    int size;

    KthLargest(int k, vector<int> nums)

    {

        size = k;

        for (int i = 0; i < nums.size(); i++)

        {

            pq.push(nums[i]);

            if (pq.size() > k)

                pq.pop();

        }

    }

    int add(int val)

    {

        pq.push(val);

        if (pq.size() > size)

            pq.pop();

        return pq.top();

    }

};

*125-Distinct numbers in a window*

vector<int> Solution::dNums(vector<int> &A, int B) {

    unordered\_map<int,int> mp;

    for(int i=0;i<B;i++)

    mp[A[i]]++;

    vector<int> ans;

    ans.push\_back(mp.size());

    for(int i=0;i<A.size()-B;i++){

        mp[A[i]]--;

        if(mp[A[i]]==0)

        mp.erase(A[i]);

        mp[A[i+B]]++;

        ans.push\_back(mp.size());

    }

    return ans;

}

*126-Flood Fill*

*An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image. You are also given three integers sr, sc, and color. You should perform a flood fill on the image starting from the pixel image[sr][sc]. To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color. Return the modified image after performing the flood fill.*

class Solution

{

public:

    bool IsValid(vector<vector<int>> &image, int i, int j, int n, int m, int ch)

    {

        if (i >= 0 && j >= 0 && i <= n - 1 && j <= m - 1 && image[i][j] == ch)

            return true;

        return false;

    }

    vector<vector<int>> floodFill(vector<vector<int>> &image, int sr, int sc, int col)

    {

        if (image[sr][sc] == col)

            return image;

        int n = image.size();

        int m = image[0].size();

        queue<pair<int, int>> q;

        q.push({sr, sc});

        int cl = image[sr][sc];

        image[sr][sc] = col;

        while (!q.empty())

        {

            auto u = q.front();

            q.pop();

            vector<int> r = {1, 0, -1, 0};

            vector<int> c = {0, 1, 0, -1};

            for (int i = 0; i < 4; i++)

            {

                int x = u.first + r[i];

                int y = u.second + c[i];

                if (IsValid(image, x, y, n, m, cl))

                    q.push({x, y}), image[x][y] = col;

            }

        }

        return image;

    }

};

*127-DFS*

class Solution

{

public:

    void DFS(int i, vector<int> adj[], vector<int> &dfs, vector<int> &Vis)

    {

        dfs.push\_back(i);

        Vis[i] = 1;

        for (auto x : adj[i])

        {

            if (!Vis[x])

                DFS(x, adj, dfs, Vis);

        }

    }

    vector<int> dfsOfGraph(int V, vector<int> adj[])

    {

        vector<int> dfs;

        vector<int> Vis(V + 1, 0);

        for (int i = 0; i < V; i++)

        {

            if (!Vis[i])

                DFS(i, adj, dfs, Vis);

        }

        return dfs;

    }

};

*128-Cycle in a directed graph using DFS*

class Solution

{

public:

    bool iscycle(vector<int> adj[], vector<int> &vis, int id)

    {

        if (vis[id] == 1)

            return true;

        if (vis[id] == 0)

        {

            vis[id] = 1;

            for (auto edge : adj[id])

            {

                if (iscycle(adj, vis, edge))

                    return true;

            }

        }

        vis[id] = 2;

        return false;

    }

    bool canFinish(int n, vector<vector<int>> &pre)

    {

        vector<int> adj[n];

        for (auto edge : pre)

            adj[edge[1]].push\_back(edge[0]);

        vector<int> vis(n, 0);

        for (int i = 0; i < n; i++)

        {

            if (iscycle(adj, vis, i))

                return false;

        }

        return true;

    }

};

*129-Number of islands*

**Problem Statement:** Given a boolean 2D matrix grid of size N x M. You have to find the number of distinct islands where a group of connected 1s (horizontally or vertically) forms an island. Two islands are considered to be distinct if and only if one island is equal to another (not rotated or reflected).

class Solution

{

public:

    void dfs(vector<vector<char>> &grid, int i, int j, int n, int m)

    {

        grid[i][j] = '0';

        int r[] = {1, 0, -1, 0};

        int c[] = {0, 1, 0, -1};

        for (int k = 0; k < 4; k++)

        {

            int x = i + r[k];

            int y = j + c[k];

            if (x >= 0 && x < n && y >= 0 && y < m && grid[x][y] == '1')

                dfs(grid, x, y, n, m);

        }

    }

    int numIslands(vector<vector<char>> &grid)

    {

        int n = grid.size();

        int m = grid[0].size();

        int ans = 0;

        for (int i = 0; i < n; i++)

        {

            for (int j = 0; j < m; j++)

            {

                if (grid[i][j] == '1')

                {

                    dfs(grid, i, j, n, m);

                    ans++;

                }

            }

        }

        return ans;

    }

};

*130-Bipartite check Using DFS*

class Solution

{

public:

    bool bipartite(int u, vector<vector<int>> &graph, vector<int> &vis, int col)

    {

        vis[u] = col;

        for (auto x : graph[u])

        {

            if (vis[x] == -1)

            {

                if (!bipartite(x, graph, vis, !col))

                    return false;

            }

            else if (vis[x] == vis[u])

                return false;

        }

        return true;

    }

    bool isBipartite(vector<vector<int>> &graph)

    {

        int n = graph.size();

        vector<int> vis(n + 1, -1);

        for (int i = 0; i < n; i++)

        {

            if (vis[i] == -1 && !bipartite(i, graph, vis, 0))

                return false;

        }

        return true;

    }

};

*131-Bipartite check using BFS*

class Solution

{

public:

    bool bipartite(int u, vector<vector<int>> &graph, vector<int> &vis)

    {

        queue<int> q;

        q.push(u);

        vis[u] = 1;

        while (!q.empty())

        {

            int x = q.front();

            q.pop();

            for (auto v : graph[x])

            {

                if (vis[v] == -1)

                {

                    q.push(v);

                    vis[v] = 1 - vis[x];

                }

                else if (vis[v] == vis[x])

                    return false;

            }

        }

        return true;

    }

    bool isBipartite(vector<vector<int>> &graph)

    {

        int n = graph.size();

        vector<int> vis(n + 1, -1);

        for (int i = 0; i < n; i++)

        {

            if (vis[i] == -1 && !bipartite(i, graph, vis))

            {

                return false;

            }

        }

        return true;

    }

};

*132-Topological sort using BFS (Kahn’s Algorithm)*

class Solution

{

public:

    vector<int> topoSort(int V, vector<int> adj[])

    {

        vector<int> inDegree(V, 0);

        for (int i = 0; i < V; i++)

        {

            for (auto x : adj[i])

            {

                inDegree[x]++;

            }

        }

        queue<int> q;

        for (int i = 0; i < V; i++)

        {

            if (inDegree[i] == 0)

                q.push(i);

        }

        vector<int> ans;

        while (!q.empty())

        {

            int node = q.front();

            q.pop();

            ans.push\_back(node);

            for (auto x : adj[node])

            {

                inDegree[x]--;

                if (inDegree[x] == 0)

                    q.push(x);

            }

        }

        return ans;

    }

};

*133-Topological sort using DFS*

class Solution

{

public:

    void dfs(int src, vector<int> adj[], vector<bool> &vis, stack<int> &st)

    {

        vis[src] = 1;

        for (auto x : adj[src])

        {

            if (!vis[x])

                dfs(x, adj, vis, st);

        }

        st.push(src);

    }

    vector<int> topoSort(int V, vector<int> adj[])

    {

        vector<bool> vis(V, false);

        stack<int> st;

        for (int i = 0; i < V; i++)

        {

            if (!vis[i])

                dfs(i, adj, vis, st);

        }

        vector<int> ans;

        while (!st.empty())

        {

            ans.push\_back(st.top());

            st.pop();

        }

        return ans;

    }

};

*134-Detect a cycle in a directed graph Using BFS(Kahn’s algorithm)*

class Solution

{

public:

    bool canFinish(int n, vector<vector<int>> &pre)

    {

        vector<int> adj[n];

        for (auto edge : pre)

            adj[edge[1]].push\_back(edge[0]);

        vector<int> inDegree(n, 0);

        for (int i = 0; i < n; i++)

        {

            for (auto x : adj[i])

            {

                inDegree[x]++;

            }

        }

        queue<int> q;

        for (int i = 0; i < n; i++)

        {

            if (inDegree[i] == 0)

                q.push(i);

        }

        vector<int> ans;

        while (!q.empty())

        {

            int node = q.front();

            q.pop();

            ans.push\_back(node);

            for (auto x : adj[node])

            {

                inDegree[x]--;

                if (inDegree[x] == 0)

                    q.push(x);

            }

        }

        if (ans.size() == n)

            return true;

        else

            return false;

    }

};

*135- Detect a cycle in an undirected graph Using BFS(Kahn’s algorithm)*

#include <bits/stdc++.h>

using namespace std;

class Solution

{

private:

    bool detect(int src, vector<int> adj[], int vis[])

    {

        vis[src] = 1;

        // store <source node, parent node>

        queue<pair<int, int>> q;

        q.push({src, -1});

        // traverse until queue is not empty

        while (!q.empty())

        {

            int node = q.front().first;

            int parent = q.front().second;

            q.pop();

            // go to all adjacent nodes

            for (auto adjacentNode : adj[node])

            {

                // if adjacent node is unvisited

                if (!vis[adjacentNode])

                {

                    vis[adjacentNode] = 1;

                    q.push({adjacentNode, node});

                }

                // if adjacent node is visited and is not it's own parent node

                else if (parent != adjacentNode)

                {

                    // yes it is a cycle

                    return true;

                }

            }

        }

        // there's no cycle

        return false;

    }

public:

    // Function to detect cycle in an undirected graph.

    bool isCycle(int V, vector<int> adj[])

    {

        // initialise them as unvisited

        int vis[V] = {0};

        for (int i = 0; i < V; i++)

        {

            if (!vis[i])

            {

                if (detect(i, adj, vis))

                    return true;

            }

        }

        return false;

    }

};

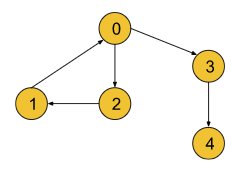
*136-Strongly connected components (Kosaraju’s algorithm)*

**Problem Statement:** Given a Directed Graph with V vertices (Numbered from 0 to V-1) and E edges, Find the number of strongly connected components in the graph.

A component is called a Strongly Connected Component(SCC) only if for every possible pair of vertices (u, v) inside that component, u is reachable from v and v is reachable from u.

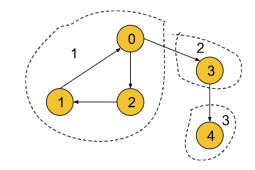
**Example 1**:

**Input Format:**



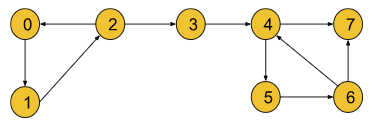
**Result:** 3

**Explanation:** Three strongly connected components are marked below:



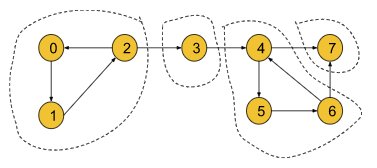
**Example 2**:

**Input Format:**



**Result:** 4

**Explanation:** Four strongly connected components are marked below:



#include <bits/stdc++.h>

using namespace std;

class Solution

{

private:

    void dfs(int node, vector<int> &vis, vector<int> adj[],

             stack<int> &st)

    {

        vis[node] = 1;

        for (auto it : adj[node])

        {

            if (!vis[it])

            {

                dfs(it, vis, adj, st);

            }

        }

        st.push(node);

    }

private:

    void dfs3(int node, vector<int> &vis, vector<int> adjT[])

    {

        vis[node] = 1;

        for (auto it : adjT[node])

        {

            if (!vis[it])

            {

                dfs3(it, vis, adjT);

            }

        }

    }

public:

    // Function to find number of strongly connected components in the graph.

    int kosaraju(int V, vector<int> adj[])

    {

        vector<int> vis(V, 0);

        stack<int> st;

        for (int i = 0; i < V; i++)

        {

            if (!vis[i])

            {

                dfs(i, vis, adj, st);

            }

        }

        vector<int> adjT[V];

        for (int i = 0; i < V; i++)

        {

            vis[i] = 0;

            for (auto it : adj[i])

            {

                // i -> it

                // it -> i

                adjT[it].push\_back(i);

            }

        }

        int scc = 0;

        while (!st.empty())

        {

            int node = st.top();

            st.pop();

            if (!vis[node])

            {

                scc++;

                dfs3(node, vis, adjT);

            }

        }

        return scc;

    }

};

int main()

{

    int n = 5;

    int edges[5][2] = {

        {1, 0}, {0, 2}, {2, 1}, {0, 3}, {3, 4}};

    vector<int> adj[n];

    for (int i = 0; i < n; i++)

    {

        adj[edges[i][0]].push\_back(edges[i][1]);

    }

    Solution obj;

    int ans = obj.kosaraju(n, adj);

    cout << "The number of strongly connected components is: " << ans << endl;

    return 0;

}

*137-* Dijkstra’s Algorithm

class Solution

{

public:

    vector<int> dijkstra(int V, vector<vector<int>> adj[], int S)

    {

        vector<int> ans(V, INT\_MAX);

        ans[S] = 0;

        priority\_queue<pair<int, int>> p;

        p.push({0, -S});

        while (!p.empty())

        {

            int node = -p.top().second;

            int d = -p.top().first;

            p.pop();

            for (auto arr : adj[node])

            {

                int x = arr[0];

                int dl = arr[1];

                if (dl + d < ans[x])

                {

                    ans[x] = dl + d;

                    p.push({-dl - d, -x});

                }

            }

        }

        return ans;

    }

};

*138-Bellman Ford Algorithm*

*When we have negative weights, dijkstra’s algorithm will get trapped in an infinite loop.*

class Solution

{

public:

    vector<int> bellman\_ford(int V, vector<vector<int>> &edges, int S)

    {

        vector<int> ans(V, 1e8);

        ans[S] = 0;

        for (int i = 1; i <= V - 1; i++)

        {

            for (auto x : edges)

            {

                int u = x[0];

                int v = x[1];

                int wt = x[2];

                if (ans[u] != 1e8 && ans[u] + wt < ans[v])

                {

                    ans[v] = ans[u] + wt;

                }

            }

        }

        for (auto x : edges)

        {

            int u = x[0];

            int v = x[1];

            int wt = x[2];

            if (ans[u] != 1e8 && ans[u] + wt < ans[v])

            {

                return {-1};

            }

        }

        return ans;

    }

};

*139-Floyd warshal algorithm*

*Problem Statement: The problem is to find the shortest distances between every pair of vertices in a given edge-weighted directed graph. The graph is represented as an adjacency matrix of size n\*n. Matrix[i][j] denotes the weight of the edge from i to j. If Matrix[i][j]=-1, it means there is no edge from i to j.*

class Solution

{

public:

    void shortest\_distance(vector<vector<int>> &mat)

    {

        int n = mat.size();

        for (int i = 0; i < n; i++)

        {

            for (int j = 0; j < n; j++)

            {

                if (mat[i][j] == -1)

                    mat[i][j] = 1e9;

                if (i == j)

                    mat[i][j] = 0;

            }

        }

        for (int k = 0; k < n; k++)

        {

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

            {

                for (int j = 0; j < n; j++)

                {

                    mat[i][j] = min(mat[i][j], mat[i][k] + mat[k][j]);

                }

            }

        }

        for (int i = 0; i < n; i++)

        {

            for (int j = 0; j < n; j++)

            {

                if (mat[i][j] == 1e9)

                    mat[i][j] = -1;

            }

        }

    }

};

*140-Prim’s Algorithm to find MST*

*Given a weighted, undirected, and connected graph of V vertices and E edges. The task is to find the sum of weights of the edges of the Minimum Spanning Tree. (Sometimes it may be asked to find the MST as well, where in the MST the edge-informations will be stored in the form {u, v}(u = starting node, v = ending node).)*

class Solution

{

public:

    int spanningTree(int V, vector<vector<int>> adj[])

    {

        priority\_queue<pair<int, int>> p;

        p.push({0, 0});

        int sum = 0;

        vector<bool> vis(V, false);

        while (!p.empty())

        {

            int node = -p.top().second;

            int wt = -p.top().first;

            p.pop();

            if (vis[node])

                continue;

            vis[node] = 1;

            sum += wt;

            for (auto x : adj[node])

            {

                int neigh = x[0];

                int wt = x[1];

                if (!vis[neigh])

                    p.push({-wt, -neigh});

            }

        }

        return sum;

    }

};

*141-Kruskal’s algorithm, to find MST*

class Solution

{

public:

    vector<int> par, size;

    int parent(int u)

    {

        if (par[u] == u)

            return u;

        else

            return par[u] = parent(par[u]);

    }

    void Union(int a, int b)

    {

        a = parent(a);

        b = parent(b);

        if (a != b)

        {

            if (size[a] < size[b])

                swap(a, b);

            par[b] = a;

            size[a] += size[b];

        }

    }

    void MakeUnion(int n)

    {

        par.resize(n + 1);

        size.resize(n + 1);

        for (int i = 0; i < n; i++)

            par[i] = i, size[i] = 1;

    }

    int spanningTree(int V, vector<vector<int>> adj[])

    {

        MakeUnion(V);

        vector<pair<int, pair<int, int>>> v;

        for (int i = 0; i < V; i++)

        {

            for (auto x : adj[i])

            {

                int k = x[0];

                int wt = x[1];

                v.push\_back({wt, {i, k}});

            }

        }

        sort(v.begin(), v.end());

        int sum = 0;

        for (auto x : v)

        {

            int wt = x.first;

            int u = x.second.first;

            int v = x.second.second;

            if (parent(u) != parent(v))

            {

                sum += wt;

                Union(u, v);

            }

        }

        return sum;

    }

};

*142-Max Product Subarray*

*Given an array that contains both negative and positive integers, find the maximum product subarray.*

class Solution

{

public:

    int maxProduct(vector<int> &nums)

    {

        int maxLeft = nums[0];

        int maxRight = nums[0];

        int prod = 1;

        bool zeroPresent = false;

        for (auto i : nums)

        {

            prod \*= i;

            if (i == 0)

            {

                zeroPresent = true;

                prod = 1;

                continue;

            }

            maxLeft = max(maxLeft, prod);

        }

        prod = 1;

        for (int j = nums.size() - 1; j >= 0; j--)

        {

            prod \*= nums[j];

            if (nums[j] == 0)

            {

                zeroPresent = true;

                prod = 1;

                continue;

            }

            maxRight = max(maxRight, prod);

        }

        if (zeroPresent)

            return max(max(maxLeft, maxRight), 0);

        return max(maxLeft, maxRight);

    }

};

*143-Longest increasing subsequence*

class Solution

{

public:

    int helper(vector<int> &nums, int idx, int prev, vector<vector<int>> &dp)

    {

        if (idx == nums.size())

            return 0;

        if (dp[idx][prev + 1] != -1)

            return dp[idx][prev + 1];

        int len = helper(nums, idx + 1, prev, dp);

        if (prev == -1 || nums[idx] > nums[prev])

            len = max(len, 1 + helper(nums, idx + 1, idx, dp));

        return dp[idx][prev + 1] = len;

    }

    int lengthOfLIS(vector<int> &nums)

    {

        int n = nums.size();

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

        return helper(nums, 0, -1, dp);

    }

};

*Better approach*

class Solution

{

public:

    int lengthOfLIS(vector<int> &nums)

    {

        int n = nums.size();

        vector<int> dp(n, 1);

        int ans = INT\_MIN;

        for (int i = 0; i < n; i++)

        {

            for (int j = 0; j < i; j++)

            {

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

                    dp[i] = max(dp[i], 1 + dp[j]);

            }

            ans = max(ans, dp[i]);

        }

        return ans;

    }

};

*144-Longest Common Subsequence*

*Memoized*

class Solution {

public:

    int helper(string a,int i,string b,int j,vector<vector<int>> &dp){

        if(i<0 || j<0)

        return 0;

        if(dp[i][j]!=-1)

        return dp[i][j];

        if(a[i]==b[j])

        return dp[i][j]=1+helper(a,i-1,b,j-1,dp);

        else

        return dp[i][j]=max(helper(a,i-1,b,j,dp),helper(a,i,b,j-1,dp));

    }

    int longestCommonSubsequence(string text1, string text2) {

        vector<vector<int>> dp(text1.size()+1,vector<int>(text2.size()+1,0));

        return helper(text1,text1.size()-1,text2,text2.size()-1,dp);

    }

};

*Tabulated*

class Solution

{

public:

    int longestCommonSubsequence(string text1, string text2)

    {

        int n = text1.size();

        int m = text2.size();

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

        for (int i = 0; i <= n; i++)

        {

            dp[i][0] = 0;

        }

        for (int i = 0; i <= m; i++)

        {

            dp[0][i] = 0;

        }

        for (int ind1 = 1; ind1 <= n; ind1++)

        {

            for (int ind2 = 1; ind2 <= m; ind2++)

            {

                if (text1[ind1 - 1] == text2[ind2 - 1])

                    dp[ind1][ind2] = 1 + dp[ind1 - 1][ind2 - 1];

                else

                    dp[ind1][ind2] = 0 + max(dp[ind1 - 1][ind2], dp[ind1][ind2 - 1]);

            }

        }

        return dp[n][m];

    }

};

*145-0-1 Knapsack*

*Memoized*

class Solution

{

public:

    int helper(int i, int wt[], int val[], int W, vector<vector<int>> &dp)

    {

        if (i < 0 || W <= 0)

            return 0;

        if (dp[i][W] != -1)

            return dp[i][W];

        if (wt[i] <= W)

            return dp[i][W] = max(val[i] + helper(i - 1, wt, val, W - wt[i], dp), helper(i - 1, wt, val, W, dp));

        else

            return dp[i][W] = helper(i - 1, wt, val, W, dp);

    }

    int knapSack(int W, int wt[], int val[], int n)

    {

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

        return helper(n - 1, wt, val, W, dp);

    }

};

*Tabulated*

class Solution

{

public:

    int knapSack(int W, int wt[], int val[], int n)

    {

        vector<vector<int>> dp(n + 1, vector<int>(W + 1, 0));

        for (int i = 1; i <= n; i++)

        {

            for (int j = 0; j <= W; j++)

            {

                if (i == 0 || j == 0)

                    dp[i][j] = 0;

                else if (wt[i - 1] <= j)

                {

                    dp[i][j] = max(val[i - 1] + dp[i - 1][j - wt[i - 1]], dp[i - 1][j]);

                }

                else

                    dp[i][j] = dp[i - 1][j];

            }

        }

        return dp[n][W];

    }

};

*146-* *Given two strings word1 and word2, return the minimum number of operations required to convert word1 to word2. You have the following three operations permitted on a word: Insert a character Delete a character Replace a character*

*Memoized*

class Solution

{

public:

    int helper(string s1, int i, string s2, int j, vector<vector<int>> &dp)

    {

        if (i < 0)

            return j + 1;

        if (j < 0)

            return i + 1;

        if (dp[i][j] != -1)

            return dp[i][j];

        if (s1[i] == s2[j])

            return dp[i][j] = 0 + helper(s1, i - 1, s2, j - 1, dp);

        else

            return dp[i][j] = 1 + min({helper(s1, i - 1, s2, j, dp), helper(s1, i, s2, j - 1, dp), helper(s1, i - 1, s2, j - 1, dp)});

    }

    int minDistance(string w1, string w2)

    {

        vector<vector<int>> dp(w1.size() + 1, vector<int>(w2.size(), -1));

        return helper(w1, w1.length() - 1, w2, w2.size() - 1, dp);

    }

};

*Tabulated*

class Solution

{

public:

    int minDistance(string w1, string w2)

    {

        vector<vector<int>> dp(w1.size() + 1, vector<int>(w2.size() + 1, 0));

        for (int i = 0; i <= w1.size(); i++)

            dp[i][0] = i;

        for (int i = 0; i <= w2.size(); i++)

            dp[0][i] = i;

        for (int i = 1; i <= w1.size(); i++)

        {

            for (int j = 1; j <= w2.size(); j++)

            {

                if (w1[i - 1] == w2[j - 1])

                    dp[i][j] = dp[i - 1][j - 1];

                else

                    dp[i][j] = 1 + min({dp[i - 1][j], dp[i][j - 1], dp[i - 1][j - 1]});

            }

        }

        return dp[w1.size()][w2.size()];

    }

};

*147-Maximum sum increasing subsequence*

*Memoized*

class Solution

{

public:

    int f(int arr[], int idx, int prev, int n, vector<vector<int>> &dp)

    {

        if (idx >= n)

            return 0;

        if (dp[idx][prev + 1] != -1)

            return dp[idx][prev + 1];

        if (prev == -1)

            return dp[idx][prev + 1] = max(arr[idx] + f(arr, idx + 1, idx, n, dp), f(arr, idx + 1, prev, n, dp));

        else if (arr[prev] < arr[idx])

            return dp[idx][prev + 1] = max(arr[idx] + f(arr, idx + 1, idx, n, dp), f(arr, idx + 1, prev, n, dp));

        else

            return dp[idx][prev + 1] = f(arr, idx + 1, prev, n, dp);

    }

    int maxSumIS(int arr[], int n)

    {

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

        return f(arr, 0, -1, n, dp);

    }

};

*Tabulated*

class Solution

{

public:

    int maxSumIS(int nums[], int n)

    {

        vector<int> dp(n);

        for (int i = 0; i < n; i++)

            dp[i] = nums[i];

        int ans = INT\_MIN;

        for (int i = 0; i < n; i++)

        {

            for (int j = 0; j < i; j++)

            {

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

                    dp[i] = max(dp[i], nums[i] + dp[j]);

            }

            ans = max(ans, dp[i]);

        }

        return ans;

    }

};

*148-Matrix chain Multiplication*

*Memoized*

class Solution

{

public:

    int f(int arr[], int i, int j, vector<vector<int>> &dp)

    {

        if (i == j)

            return 0;

        if (dp[i][j] != -1)

            return dp[i][j];

        int mn = INT\_MAX;

        for (int k = i; k < j; k++)

        {

            int temp = arr[i - 1] \* arr[k] \* arr[j] + f(arr, i, k, dp) + f(arr, k + 1, j, dp);

            mn = min(mn, temp);

        }

        return dp[i][j] = mn;

    }

    int matrixMultiplication(int N, int arr[])

    {

        vector<vector<int>> dp(N, vector<int>(N, -1));

        return f(arr, 1, N - 1, dp);

    }

};

*Tabulated*

class Solution

{

public:

    int matrixMultiplication(int N, int arr[])

    {

        vector<vector<int>> dp(N + 1, vector<int>(N + 1, 0));

        for (int i = N - 1; i >= 1; i--)

        {

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

            {

                if (i == j)

                    dp[i][j] = 0;

                else

                {

                    int mn = INT\_MAX;

                    for (int k = i; k < j; k++)

                    {

                        int temp = arr[i - 1] \* arr[k] \* arr[j] + dp[i][k] + dp[k + 1][j];

                        mn = min(mn, temp);

                    }

                    dp[i][j] = mn;

                }

            }

        }

        return dp[1][N - 1];

    }

};

*149-Minimum Path sum in a Grid*

We are given an “N\*M” matrix of integers. We need to find a path from the top-left corner to the bottom-right corner of the matrix, such that there is a minimum cost past that we select.

At every cell, we can move in only two directions: right and bottom. The cost of a path is given as the sum of values of cells of the given matrix.

*Memoized*

class Solution

{

public:

    int f(vector<vector<int>> &grid, int i, int j, vector<vector<int>> &dp)

    {

        int n = grid.size();

        int m = grid[0].size();

        if (i == n - 1 && j == m - 1)

        {

            return dp[i][j] = grid[i][j];

        }

        if (dp[i][j] != -1)

            return dp[i][j];

        int x1 = INT\_MAX, x2 = INT\_MAX;

        if (i + 1 < n)

            x1 = f(grid, i + 1, j, dp);

        if (j + 1 < m)

            x2 = f(grid, i, j + 1, dp);

        return dp[i][j] = min(x1, x2) + grid[i][j];

    }

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

    {

        vector<vector<int>> dp(grid.size(), vector<int>(grid[0].size(), -1));

        return f(grid, 0, 0, dp);

    }

};

*Tabulated*

class Solution {

public:

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

        int n = grid.size();

        int m = grid[0].size();

        vector<vector<int>> dp(n, vector<int>(m, 0));

        dp[0][0] = grid[0][0];

        for(int i=1; i<n; i++){

            dp[i][0] = dp[i-1][0] + grid[i][0];

        }

        for(int j=1; j<m; j++){

            dp[0][j] = dp[0][j-1] + grid[0][j];

        }

        for(int i=1; i<n; i++){

            for(int j=1; j<m; j++){

                dp[i][j] = min(dp[i-1][j], dp[i][j-1]) + grid[i][j];

            }

        }

        return dp[n-1][m-1];

    }

};

*150-Coin change problem*

*Memoized*

class Solution

{

public:

    int f(vector<int> coins, int amt, int idx, vector<vector<int>> &dp)

    {

        if (idx < 0 && amt > 0)

            return 1e9;

        if (amt == 0)

            return 0;

        if (dp[idx][amt] != -1)

            return dp[idx][amt];

        if (coins[idx] <= amt)

        {

            return dp[idx][amt] = min({1 + f(coins, amt - coins[idx], idx - 1, dp),

                   1 + f(coins, amt - coins[idx], idx, dp), f(coins, amt, idx - 1, dp)});

        }

        else

            return dp[idx][amt] = f(coins, amt, idx - 1, dp);

    }

    int coinChange(vector<int> &coins, int amount)

    {

        vector<vector<int>> dp(coins.size(), vector<int>(amount + 1, -1));

        int ans = f(coins, amount, coins.size() - 1, dp);

        return ans == 1e9 ? -1 : ans;

    }

};

*Tabulated*

class Solution

{

public:

    int coinChange(vector<int> &coins, int amount)

    {

        vector<vector<int>> dp(coins.size() + 1, vector<int>(amount + 1, -1));

        for (int i = 1; i <= amount; i++)

            dp[0][i] = 1e9;

        for (int i = 1; i <= coins.size(); i++)

        {

            for (int j = 1; j <= amount; j++)

            {

                if (coins[i - 1] <= j)

                {

                    dp[i][j] = min({1 + dp[i - 1][j - coins[i - 1]],

                                    1 + dp[i][j - coins[i - 1]], dp[i - 1][j]});

                }

                else

                    dp[i][j] = dp[i - 1][j];

            }

        }

        int ans = dp[coins.size()][amount];

        return ans == 1e9 ? -1 : ans + 1;

    }

};

*151-Partition equal subset sum*

*Memoized*

class Solution

{

public:

    bool f(vector<int> &nums, int idx, int sum, vector<vector<int>> &dp)

    {

        if (idx < 0 && sum)

            return false;

        if (sum == 0)

            return true;

        if (dp[idx][sum] != -1)

            return dp[idx][sum];

        if (nums[idx] <= sum)

            return dp[idx][sum] = f(nums, idx - 1, sum - nums[idx], dp) ||

                                  f(nums, idx - 1, sum, dp);

        else

            return dp[idx][sum] = f(nums, idx - 1, sum, dp);

    }

    bool canPartition(vector<int> &nums)

    {

        int sum = 0;

        for (auto x : nums)

            sum += x;

        if (sum & 1)

            return false;

        vector<vector<int>> dp(nums.size(), vector<int>(sum / 2 + 1, -1));

        return f(nums, nums.size() - 1, sum / 2, dp);

    }

};

*Tabulated*

class Solution

{

public:

    bool canPartition(vector<int> &nums)

    {

        int sum = 0;

        for (auto x : nums)

            sum += x;

        if (sum & 1)

            return false;

        vector<vector<int>> dp(nums.size() + 1, vector<int>(sum / 2 + 1, 0));

        for (int i = 0; i <= nums.size(); i++)

            dp[i][0] = 1;

        for (int i = 1; i <= nums.size(); i++)

        {

            for (int j = 0; j <= sum / 2; j++)

            {

                if (nums[i - 1] <= j)

                    dp[i][j] = dp[i - 1][j - nums[i - 1]] || dp[i - 1][j];

                else

                    dp[i][j] = dp[i - 1][j];

            }

        }

        return dp[nums.size()][sum / 2];

    }

};

*152-Minimum cost to cut a stick*

*Given a wooden stick of length n units. The stick is labelled from 0 to n. For example, a stick of length 6 is labelled as follows: Given an integer array cuts where cuts[i] denotes a position you should perform a cut at. You should perform the cuts in order, you can change the order of the cuts as you wish. The cost of one cut is the length of the stick to be cut, the total cost is the sum of costs of all cuts. When you cut a stick, it will be split into two smaller sticks (i.e. the sum of their lengths is the length of the stick before the cut). Please refer to the first example for a better explanation. Return the minimum total cost of the cuts.*

*Memoized*

class Solution

{

public:

    int f(vector<int> &cuts, int l, int r, vector<vector<int>> &dp)

    {

        if (r - l <= 1)

            return 0;

        if (dp[l][r] != -1)

            return dp[l][r];

        int mn = 1e9;

        for (int k = l + 1; k < r; k++)

        {

            mn = min(mn, cuts[r] - cuts[l] + f(cuts, l, k, dp) + f(cuts, k, r, dp));

        }

        return dp[l][r] = mn;

    }

    int minCost(int n, vector<int> &cuts)

    {

        cuts.push\_back(0);

        cuts.push\_back(n);

        sort(cuts.begin(), cuts.end());

        vector<vector<int>> dp(cuts.size(), vector<int>(cuts.size(), -1));

        return f(cuts, 0, cuts.size() - 1, dp);

    }

};

*Tabulated*

class Solution

{

public:

    int minCost(int n, vector<int> &cuts)

    {

        cuts.push\_back(0);

        cuts.push\_back(n);

        sort(cuts.begin(), cuts.end());

        vector<vector<int>> dp(cuts.size(), vector<int>(cuts.size(), 0));

        for (int l = cuts.size() - 1; l >= 0; l--)

        {

            for (int r = l + 2; r < cuts.size(); r++)

            {

                int mn = 1e9;

                for (int k = l + 1; k < r; k++)

                {

                    mn = min(mn, cuts[r] - cuts[l] + dp[l][k] + dp[k][r]);

                }

                dp[l][r] = mn;

            }

        }

        return dp[0][cuts.size() - 1];

    }

};

*153-Word Break*

*Memoized*

class Solution

{

public:

    bool f(string s, int i, int j, unordered\_map<string, int> &mp, vector<vector<int>> &dp)

    {

        if (i > j)

            return true;

        if (dp[i][j] != -1)

            return dp[i][j];

        for (int k = i; k <= j; k++)

        {

            string t = s.substr(i, k - i + 1);

            if (mp.count(t))

            {

                bool x = f(s, k + 1, j, mp, dp);

                if (x)

                    return dp[i][j] = x;

            }

        }

        return dp[i][j] = false;

    }

    int wordBreak(string A, vector<string> &B)

    {

        unordered\_map<string, int> mp;

        vector<vector<int>> dp(A.length(), vector<int>(A.length(), -1));

        for (auto x : B)

            mp[x]++;

        return f(A, 0, A.length() - 1, mp, dp);

    }

};

*154-Palindromic Partioning*

*Memoized*

class Solution

{

public:

    int f(string s, int i, int j, vector<vector<int>> &dp)

    {

        if (i >= j)

            return 0;

        if (dp[i][j] != -1)

            return dp[i][j];

        int mn = 1e9;

        for (int k = i; k <= j; k++)

        {

            string s1 = s.substr(i, k - i + 1);

            string s2 = s1;

            reverse(s1.begin(), s1.end());

            if (s1 == s2)

                mn = min(mn, 1 + f(s, k + 1, j, dp));

        }

        return dp[i][j] = mn;

    }

    int palindromicPartition(string str)

    {

        vector<vector<int>> dp(str.length() + 1, vector<int>(str.length() + 1, -1));

        return f(str, 0, str.size(), dp) - 1;

    }

};

Tabulated

155-Job sequencing

Given a set of **N** jobs where each **jobi** has a deadline and profit associated with it.

Each job takes **1** unit of time to complete and only one job can be scheduled at a time. We earn the profit associated with job if and only if the job is completed by its deadline.

Find the number of jobs done and the **maximum profit**.

**Note:**Jobs will be given in the form (Jobid, Deadline, Profit) associated with that Job.

class Solution

{

public:

    vector<int> JobScheduling(Job arr[], int n)

    {

        vector<pair<int, int>> v;

        for (int i = 0; i < n; i++)

        {

            v.push\_back({arr[i].profit, arr[i].dead});

        }

        vector<int> pre(100005, 0);

        sort(v.begin(), v.end());

        for (int i = n - 1; i >= 0; i--)

        {

            int idx = v[i].second;

            if (pre[idx] == 0)

                pre[idx] = v[i].first;

            else

            {

                for (int j = idx - 1; j >= 1; j--)

                {

                    if (pre[j] == 0)

                    {

                        pre[j] = v[i].first;

                        break;

                    }

                }

            }

        }

        int ans = 0;

        int cnt = 0;

        for (auto x : pre)

        {

            if (x > 0)

            {

                ans += x;

                cnt++;

            }

        }

        return {cnt, ans};

    }

};

156-Egg Dropping Problem

Memoized

class Solution

{

public:

    int f(int n, int k, vector<vector<int>> &dp)

    {

        if (n == 1 || k == 1 || k == 0)

            return k;

        if (dp[n][k] != -1)

            return dp[n][k];

        int mn = 1e9;

        for (int i = 1; i <= k; i++)

        {

            int x = 1 + max(f(n - 1, i - 1, dp), f(n, k - i, dp));

            mn = min(mn, x);

        }

        return dp[n][k] = mn;

    }

    int eggDrop(int n, int k)

    {

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

        return f(n, k, dp);

    }

};

Tabulated

class Solution

{

public:

    int eggDrop(int n, int k)

    {

        vector<vector<int>> dp(n + 1, vector<int>(k + 1, 0));

        for (int i = 1; i <= k; i++)

            dp[1][i] = i;

        for (int i = 1; i <= n; i++)

            dp[i][1] = 1;

        for (int i = 2; i <= n; i++)

        {

            for (int j = 2; j <= k; j++)

            {

                int mn = 1e9;

                for (int K = 1; K <= j; K++)

                {

                    int x = 1 + max(dp[i - 1][K - 1], dp[i][j - K]);

                    mn = min(mn, x);

                }

                dp[i][j] = mn;

            }

        }

        return dp[n][k];

    }

};