# STRIVER DSA SHEET

1) Given an array nums with n objects colored red, white, or blue, sort them [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm)so that objects of the same color are adjacent, with the colors in the order red, white, and blue.

We will use the integers 0, 1, and 2 to represent the color red, white, and blue, respectivey.

You must solve this problem without using library's sort function.

**Input:** nums = [2,0,2,1,1,0]

**Output:** [0,0,1,1,2,2]

: void sortColors(vector<int>& nums) {

int low=0,mid=0,high=nums.size()-1;­­­

while(mid<=high)

{

if(nums[mid]==0)

{

swap(nums[mid],nums[low]);

low++,mid++;

}

else if(nums[mid]==1)

{

mid++;

}

else

{

swap(nums[mid],nums[high]);

high--;

}

}

2) Given an unsorted array of size n. Array elements are in the range from 1 to n. One number from set {1, 2, …n} is missing and one number occurs twice in the array. Find these two numbers.

**Examples:**

**Input:** arr[] = {3, 1, 3}

**Output:** Missing = 2, Repeating = 3

**Explanation:** In the array,

2 is missing and 3 occurs twice

: int \*findTwoElement(int \*arr, int n) {

// code here

int slow=arr[0];

int fast=arr[0];

do{

slow=arr[slow];

fast=arr[arr[fast]];

}while(slow!=fast);

fast=arr[0];

while(slow!=fast)

{

slow=arr[slow];

fast=arr[fast];

}

int sum=0;

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

{

sum+=arr[i];

}

int ans=sum-slow;

int ans2=0;

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

ans2+=arr[i];

int\*v=new int[2]; \*\*\*\*\*\*\*SOMETHING NEW\*\*\*\*\*\*\* RETURN POINTER\*\*\*\*\*\*\*\*\*

v[0]=slow;

v[1]=ans2-ans;

return v;

}

3)find duplicates in array

: int findDuplicate(vector<int>& nums) {

int slow=nums[0];

int fast=nums[0];

do

{

slow=nums[slow];

fast=nums[nums[fast]];

}while(slow!=fast);

fast=nums[0];

while(slow!=fast)

{

slow=nums[slow];

fast=nums[fast];

}

return slow;

}

4) Given two sorted arrays, we need to merge them in O((n+m)\*log(n+m)) time with O(1) extra space into a sorted array, when n is the size of the first array, and m is the size of the second array.

**Example:**

**Input:** ar1[] = {10};

ar2[] = {2, 3};

**Output:** ar1[] = {2}

ar2[] = {3, 10}

: void merge(long long arr1[], long long arr2[], int n, int m) \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*not accepting due to high time complexity(INSERTION SORT)\*\*\*\*\*\*\*\*\*\*\*\*

{

// code here

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

{

if(arr1[i]>arr2[0])

{

swap(arr1[i],arr2[0]);

sort(arr2,arr2+m);

}

}

}

\*\*\*\*\*\*\*\*\*Discussion solution\*\*\*\*\*\*\*\*

void merge(long long arr1[], long long arr2[], int n, int m)

{

// code here

int i=n-1;

int j=0;

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

if (arr1[i]>arr2[j]){

swap(arr1[i],arr2[j]);

i--;

j++;]

}

else{

j++;

}

}

sort(arr1,arr1+n);

sort(arr2,arr2+m);

}

5) Given an integer array nums, find the contiguous subarray (containing at least one number) which has the largest sum and return its sum.

A **subarray** is a **contiguous** part of an array.

**Example 1:**

**Input:** nums = [-1,-3,4,-1,2,1,-5,4]

**Output:** 6

**Explanation:** [4,-1,2,1] has the largest sum = 6.

(KADANE’S ALGORITHTM)

: int maxSubArray(vector<int>& nums) {

int sum=0;

int maximum=INT\_MIN;

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

{

sum+=nums[i];

maximum=max(maximum,sum);

if(sum<0)

sum=0;

}

return maximum;

}

6) Given an array of intervals where intervals[i] = [starti, endi], merge all overlapping intervals, and return an array of the non-overlapping intervals that cover all the intervals in the input.

**Example 1:**

**Input:** intervals = [[1,3],[2,6],[8,10],[15,18]]

**Output:** [[1,6],[8,10],[15,18]]

**Explanation:** Since intervals [1,3] and [2,6] overlaps, merge them into [1,6].

: vector<vector<int>> merge(vector<vector<int>>& intervals) {

vector<vector<int>>v;

if(intervals.size()==0)

return v;

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

vector<int>v1=intervals[0];

for(auto i:intervals)

{

if(i[0]<=v1[1])

v1[1]=max(i[1],v1[1]);

else

{

v.push\_back(v1);

v1=i;

}

}

v.push\_back(v1);

return v;

}

ANOTHER VARIATION OF SAME APPROACH

Given an array of intervals intervals where intervals[i] = [starti, endi], return *the minimum number of intervals you need to remove to make the rest of the intervals non-overlapping*.

**Example 1:**

**Input:** intervals = [[1,2],[2,3],[3,4],[1,3]]

**Output:** 1

**Explanation:** [1,3] can be removed and the rest of the intervals are non-overlapping.

: bool comp(vector<int>&a,vector<int>&b)

{

return a[1]<b[1];

}

class Solution {

public:

int eraseOverlapIntervals(vector<vector<int>>& intervals) {

int ans=-1;

sort(intervals.begin(),intervals.end(),comp);

vector<int>temp=intervals[0];

for(auto it:intervals)

{

if(temp[1]>it[0])

{

ans++;

}

else

temp=it;

}

return ans;

}

};

7) Given an m x n integer matrix matrix, if an element is 0, set its entire row and column to 0's, and return *the matrix*.

You must do it [in place](https://en.wikipedia.org/wiki/In-place_algorithm).

**Example 1:**



**Input:** matrix = [[1,1,1],[1,0,1],[1,1,1]]

**Output:** [[1,0,1],[0,0,0],[1,0,1]]

: void setZeroes(vector<vector<int>>& matrix) {

int col\_zero=1,row=matrix.size(),col=matrix[0].size();

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

{

if(matrix[i][0]==0)

col\_zero=0;

for(int j=1;j<col;j++)

{

if(matrix[i][j]==0)

matrix[i][0]=matrix[0][j]=0;

}

}

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

{

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

{

if(matrix[i][0]==0||matrix[0][j]==0)

matrix[i][j]=0;

}

if(col\_zero==0)

matrix[i][0]=0;

}

}

8) Given an integer numRows, return the first numRows of **Pascal's triangle**.

In **Pascal's triangle**, each number is the sum of the two numbers directly above it as shown:



**Example 1:**

**Input:** numRows = 5

**Output:** [[1],[1,1],[1,2,1],[1,3,3,1],[1,4,6,4,1]]

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

vector<vector<int>>r(numRows);

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

{

r[i].resize(i+1);

r[i][0]=r[i][i]=1;

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

{

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

}

}

return r;

}

9) Implement **next permutation**, which rearranges numbers 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).

The replacement must be [**in place**](http://en.wikipedia.org/wiki/In-place_algorithm) and use only constant extra memory.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [1,3,2]

:void nextPermutation(vector<int>& nums) {

int i,j;

int n=nums.size();

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

{

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

{

break;

}

}

if(i<0)

{

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

}

else

{

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

{

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

{

break;

}

}

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

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

}

}

EXTRA QUESTION[IF GIVEN QUESTION IS TALKING ABOUT N\*N MATRIX THEN NO NEED TO TAKE ANOTHER VECTOR MATRIX BUT IF QUESTION ASKS ABOUT N\*M THEN WE HAVE TO TAKE ANOTHER VECTOR MATRIX]

Given a 2D integer array matrix, return *the****transpose****of* matrix.

The **transpose** of a matrix is the matrix flipped over its main diagonal, switching the matrix's row and column indices.



**Example 1:**

**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[1,4,7],[2,5,8],[3,6,9]]

: vector<vector<int>> transpose(vector<vector<int>>& matrix) {

vector<vector<int>>ans(matrix[0].size(),vector<int>(matrix.size(),0));

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

{

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

{

ans[i][j]=matrix[j][i];

}

}

return ans;

}

10) You are given an n x n 2D matrix representing an image, rotate the image by **90** degrees (clockwise).

You have to rotate the image [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm), which means you have to modify the input 2D matrix directly. **DO NOT** allocate another 2D matrix and do the rotation.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [[7,4,1],[8,5,2],[9,6,3]]

: void rotate(vector<vector<int>>& matrix) {

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

{

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

{

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

}

}

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

{

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

}

}

Additional problem on sell-stock

Say you have an array, **A**, for which the **ith** element is the price of a given stock on day **i**.

Design an algorithm to find the maximum profit.

You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times).

However, you may not engage in multiple transactions at the same time (ie, you must sell the stock before you buy again).

**Input Format:** The first and the only argument is an array of integer, A.

**Output Format:** Return an integer, representing the maximum possible profit.

**Constraints:** 0 <= len(A) <= 1e5 1 <= A[i] <= 1e7 **Example:**

Input 1:

A = [1, 2, 3]

Output 1:

2

Explanation 1:

=> Buy a stock on day 0.

=> Sell the stock on day 1. (Profit +1)

=> Buy a stock on day 1.

=> Sell the stock on day 2. (Profit +1)

Overall profit = 2

: int Solution::maxProfit(const vector<int> &A) {

    if(A.size() == 0)

return 0;

    int profit = 0;

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

        if(A[i+1] > A[i])

profit += (A[i+1] - A[i]);

    }

    return profit;

}

11) sell stocks-1

You are given an array prices where prices[i] is the price of a given stock on the 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.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 5

**Explanation:** Buy on day 2 (price = 1) and sell on day 5 (price = 6), profit = 6-1 = 5.

Note that buying on day 2 and selling on day 1 is not allowed because you must buy before you sell.

: int maxProfit(vector<int>& prices) {

int profit=0;

int minprice=INT\_MAX;

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

{

minprice=min(minprice,prices[i]);

profit=max(profit,prices[i]-minprice);

}

return profit;

}

Sell-Stock-2

You are given an integer array prices where prices[i] is the price of a given stock on the ith day.

On each day, you may decide to buy and/or sell the stock. You can only hold **at most one** share of the stock at any time. However, you can buy it then immediately sell it on the **same day**.

Find and return *the****maximum****profit you can achieve*.

**Example 1:**

**Input:** prices = [7,1,5,3,6,4]

**Output:** 7

**Explanation:** Buy on day 2 (price = 1) and sell on day 3 (price = 5), profit = 5-1 = 4.

Then buy on day 4 (price = 3) and sell on day 5 (price = 6), profit = 6-3 = 3.

Total profit is 4 + 3 = 7.

: int maxProfit(vector<int>& prices) {

int profit=0;

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

{

profit+=max(prices[i]-prices[i-1],0);

}

return profit;

}

Sell -stocks-3

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete **at most two transactions**.

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [3,3,5,0,0,3,1,4]

**Output:** 6

**Explanation:** Buy on day 4 (price = 0) and sell on day 6 (price = 3), profit = 3-0 = 3.

Then buy on day 7 (price = 1) and sell on day 8 (price = 4), profit = 4-1 = 3.

: int maxProfit(vector<int>& prices) {

if(prices.size()==0)

return 0;

int buy1=INT\_MAX;

int profit1=INT\_MIN;

int buy2=INT\_MAX;

int profit2=INT\_MIN;

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

{

buy1=min(buy1,prices[i]);

profit1=max(profit1,prices[i]-buy1);

buy2=min(buy2,prices[i]-profit1);

profit2=max(profit2,prices[i]-buy2);

}

return profit2;

}

Sell-stock-4

You are given an integer array prices where prices[i] is the price of a given stock on the ith day, and an integer k.

Find the maximum profit you can achieve. You may complete at most k transactions.

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** k = 2, prices = [2,4,1]

**Output:** 2

**Explanation:** Buy on day 1 (price = 2) and sell on day 2 (price = 4), profit = 4-2 = 2.

: int maxProfit(int k, vector<int>& prices) {

if(k>=prices.size()/2)

{

int profit=0;

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

{

if(prices[i]>prices[i-1])

profit+=prices[i]-prices[i-1];

}

return profit;

}

vector<int>buy(k+1,INT\_MAX);

vector<int>sell(k+1,0);

for(auto it:prices)

{

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

{

buy[i]=min(buy[i],it-sell[i-1]);

sell[i]=max(sell[i],it-buy[i]);

}

}

return sell[k];

}

Sell-stock-5(cooldown)

You are given an array prices where prices[i] is the price of a given stock on the ith day.

Find the maximum profit you can achieve. You may complete as many transactions as you like (i.e., buy one and sell one share of the stock multiple times) with the following restrictions:

* After you sell your stock, you cannot buy stock on the next day (i.e., cooldown one day).

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [1,2,3,0,2]

**Output:** 3

**Explanation:** transactions = [buy, sell, cooldown, buy, sell]

: int maxProfit(vector<int>& prices) {

int buy=INT\_MAX;

int sell=0;

int prev\_sell=0;

int prev\_buy;

for(auto it:prices)

{

prev\_buy=buy;

buy=min(buy,it-prev\_sell);

prev\_sell=sell;

sell=max(sell,it-prev\_buy);

}

return sell;

}

SELL-STOCK WITH TRANSACTION FEE

You are given an array prices where prices[i] is the price of a given stock on the ith day, and an integer fee representing a transaction fee.

Find the maximum profit you can achieve. You may complete as many transactions as you like, but you need to pay the transaction fee for each transaction.

**Note:** You may not engage in multiple transactions simultaneously (i.e., you must sell the stock before you buy again).

**Example 1:**

**Input:** prices = [1,3,2,8,4,9], fee = 2

**Output:** 8

**Explanation:** The maximum profit can be achieved by:

- Buying at prices[0] = 1

- Selling at prices[3] = 8

- Buying at prices[4] = 4

- Selling at prices[5] = 9

The total profit is ((8 - 1) - 2) + ((9 - 4) - 2) = 8.

: int maxProfit(vector<int>& prices, int fee) {

int buy=INT\_MAX;

int sell=0;

for(auto it:prices)

{

buy=min(buy,it-sell);

sell=max(sell,it-buy-fee);

}

return sell;

}

MORE DETAILED EXPLAINATION OF SELL-STOCK PROBLEMS

//Buy and Sell Stock:1

int maxProfit(vector<int>& prices) {

int buy = INT\_MAX;

int sell = 0;

for (int price : prices) {

// the maximum profit if only one transaction is allowed

buy = min(buy, price);

sell = max(sell, price - buy);

}

return sell;

}

//Buy and Sell Stock:2

//Method 1:

int maxProfit2(vector<int>& prices) {

int maxprofit = 0;

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

if (prices[i] > prices[i - 1])

maxprofit += prices[i] - prices[i - 1];

}

return maxprofit;

}

//Method 2:

int maxProfit2(vector<int>& prices) {

int buy = INT\_MAX;

int sell = 0;

for (int price : prices) {

buy = min(buy, price-sell); //Use the previous profit for reinvetsment and minimize expenditure

sell = max(sell, price - buy);

}

return sell;

}

//Buy and Sell Stock:3

//Method 1:

int maxProfit3(int\* prices, int pricesSize){

int len = pricesSize;

if(len == 0)

return 0;

int min\_so\_far = prices[0];

int max\_so\_far = prices[len-1];

int max\_profit = 0;

int maxProf[len];

memset(maxProf,0,len\*sizeof(int));

for(int i=1;i<len; i++){

int profit = (prices[i] - min\_so\_far) ;

max\_profit = max\_profit > profit ? max\_profit : profit;

maxProf[i] = max\_profit;

min\_so\_far = min\_so\_far < prices[i] ? min\_so\_far : prices[i];

}

for(int i = len-2;i>0;i--){

int profit = (max\_so\_far - prices[i])+maxProf[i];

max\_profit = max\_profit > profit ? max\_profit : profit;

max\_so\_far = max\_so\_far > prices[i] ? max\_so\_far : prices[i];

}

return max\_profit;

}

//Method 2:

int maxProfit3(vector<int>& prices) {

int buy1 = INT\_MAX, buy2 = INT\_MAX;

int sell1 = 0, sell2 = 0;

for (int price : prices) {

// the maximum profit if only one transaction is allowed

buy1 = min(buy1, price);

sell1 = max(sell1, price - buy1);

// re-invest the gained profit in the second transaction

buy2 = min(buy2, price - sell1);

sell2 = max(sell2, price - buy2);

}

return sell2;

}

//Buy and Sell Stock:4

int maxProfit4(int k, vector<int>& prices) {

if (k >= prices.size() / 2) { // if k >= n/2, then you can make maximum number of transactions

int profit = 0;

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

if (prices[i] > prices[i - 1]) profit += prices[i] - prices[i - 1];

return profit;

}

vector<int> buy(k+1,INT\_MAX);

vector<int> sell(k + 1,0);

for (int price : prices) {

for (int i = 1; i <= k; i++) {

buy[i] = min(buy[i], price-sell[i - 1]);

sell[i] = max(sell[i], price-buy[i]);

}

}

return sell[k];

}

//Buy and sell stock with cooldown period

int maxProfit(vector<int>& prices) {

int buy(INT\_MAX), sell(0), prev\_sell(0), prev\_buy;

for (int price : prices) {

prev\_buy = buy;

buy = min(buy,price-prev\_sell);

prev\_sell = sell;

sell = max(sell,price-prev\_buy);

}

return sell;

}

//Buy and Sell with transaction fee

int maxProfit(vector<int>& prices, int fee) {

long buy(INT\_MAX), sell(0), prev\_sell(0), prev\_buy;

for (int price : prices) {

buy = min(buy,price-sell);

sell = max(sell,price-buy-fee);

}

return sell;

}

Variations on kadane’s algo

You are given an integer array nums. The **absolute sum** of a subarray [numsl, numsl+1, ..., numsr-1, numsr] is abs(numsl + numsl+1 + ... + numsr-1 + numsr).

Return *the****maximum****absolute sum of any****(possibly empty)****subarray of*nums.

Note that abs(x) is defined as follows:

* If x is a negative integer, then abs(x) = -x.
* If x is a non-negative integer, then abs(x) = x.

**Example 1:**

**Input:** nums = [1,-3,2,3,-4]

**Output:** 5

**Explanation:** The subarray [2,3] has absolute sum = abs(2+3) = abs(5) = 5.

: int maxAbsoluteSum(vector<int>& nums) {

int sum=0;

int mini=0;

int maxi=0;

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

{

sum+=nums[i];

mini=min(mini,sum);

maxi=max(maxi,sum);

}

return maxi-mini;

}

12)search in matrix

:\*\*\*\*\*\*\*\*GFG variant\*\*\*\*\*\*\*\*\*

Given an n x n matrix and a number x, find the position of x in the matrix if it is present in it. Otherwise, print “Not Found”. In the given matrix, every row and column is sorted in increasing order. The designed algorithm should have linear time complexity.

**Example:**

**Input:** mat[4][4] = { {10, 20, 30, 40},

{15, 25, 35, 45},

{27, 29, 37, 48},

{32, 33, 39, 50}};

x = 29

**Output:** Found at (2, 1)

**Explanation:** Element at (2,1) is 29

int matSearch (vector <vector <int>> &mat, int N, int M, int X)

{

// your code here

int i=0,j=M-1;

while(i<N&&j>=0)

{

if(mat[i][j]==X)

return 1;

if(mat[i][j]>X)

j--;

else

i++;

}

return 0;

}

\*\*\*\*\*\*\*\*LEETCODE variant\*\*\*\*\*\*\*\*

Write an efficient algorithm that searches for a value in an m x n matrix. This matrix has the following properties:

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

**Example 1:**



**Input:** matrix = [[1,3,5,7],[10,11,16,20],[23,30,34,60]], target = 3

**Output:** true

: bool searchMatrix(vector<vector<int>>& matrix, int target) {

if(matrix.size()==0)

return 0;

int row=matrix.size();

int col=matrix[0].size();

int lo=0;

int hi=(row\*col)-1;

while(lo<=hi)

{

int mid=hi-(hi-lo)/2; \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IMP\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

if(matrix[mid/col][mid%col]==target)

return 1;

if(matrix[mid/col][mid%col]<target)

{

lo=mid+1;

}

else

hi=mid-1;

return 0;

}

13) Given an array nums of size n, return *the majority element*.

The majority element is the element that appears more than ⌊n / 2⌋ times. You may assume that the majority element always exists in the array.

**Example 1:**

**Input:** nums = [3,2,3]

**Output:** 3

**Example 2:**

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

**Output:** 2 \*\*\*\*\*\*\*BOYER’S MOORE VOTING ALGO WORKS WELL(NO NEED OF SORTING )\*\*\*\*\*\*\*\*\*\*

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

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

int ans=nums[nums.size()/2];

int i;

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

{

if(nums[i]==ans)

break;

}

if(nums[i+nums.size()/2]==ans)

return ans;

return ans;

}

14) Implement [pow(x, n)](http://www.cplusplus.com/reference/valarray/pow/), which calculates x raised to the power n (i.e., xn).

**Example 1:**

**Input:** x = 2.00000, n = 10

**Output:** 1024.00000

:

double myPow(double x, int n) {

double ans=1.0;

long long nn=n;

if(nn<0)

nn=-1\*nn;

while(nn)

{

if(nn%2==1)

{

ans\*=x;

nn-=1;

}

else

{

x\*=x;

nn/=2;

}

}

if(n<0)

{

ans=1/ans;

}

return ans;

}

15) Given an integer array of size n, find all elements that appear more than ⌊ n/3 ⌋ times.

**Example 1:**

**Input:** nums = [3,2,3]

**Output:** [3]

\*\*\*\*\*\*\*\*\*\*\*BOYER MOORE’S VOTING ALGORITHM\*\*\*\*\*\*\*\*\*\*\*\*

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

int num1=-1,num2=-1,cnt1=0,cnt2=0;

int i;

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

{

if(nums[i]==num1)

cnt1++;

else if(nums[i]==num2)

cnt2++;

else if(cnt1==0)

{

num1=nums[i];

cnt1=1;

}

else if(cnt2==0)

{

num2=nums[i];

cnt2=1;

}

else

{

cnt1--;

cnt2--;

}

}

cnt1=0,cnt2=0;

vector<int>v;

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

{

if(nums[i]==num1)

cnt1++;

else if(nums[i]==num2)

cnt2++;

}

if(cnt1>=nums.size()/3+1)

v.push\_back(num1);

if(cnt2>=nums.size()/3+1)

v.push\_back(num2);

return v;

}

16) A robot is located at the top-left corner of a m x n grid (marked 'Start' in the diagram below).

The robot can only move either down or right at any point in time. The robot is trying to reach the bottom-right corner of the grid (marked 'Finish' in the diagram below).

How many possible unique paths are there?

[FIRST PROPOSE RECURSIVE APPROACH THEN GO TO DP WALA AND THEN FINALLY OPTIMIZE IT TO MATHEMATICAL WAY I.E USING COMBINATIONS]

**Example 1:**



**Input:** m = 3, n = 7

**Output:** 28

BRUTE-FORCE(RECURSIVE)

: int func(int a,int b,int c,int d)

{

if(a==c-1&&b==d-1)

return 1;

if(a>=c||b>=d)

return 0;

return func(a+1,b,c,d)+func(a,b+1,c,d);

}

int uniquePaths(int m, int n) {

return func(0,0,m,n);

}

MAKE IT DP WALA BY JUST INSETING DP VECTOR IN FUNCTION CALL AND IN FUNCTION AS WELL

OPTIMIZED

: int uniquePaths(int m, int n) {

int N=n+m-2;

int r=n-1;

double ans=1;

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

ans=ans\*(N-r+i)/i;

return ans;

}

17)Reverse a linked list

: ListNode\* reverseList(ListNode\* head) {

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

return head;

ListNode\*prev=NULL;

ListNode\*curr=head;

ListNode\*nextptr=NULL;

while(curr!=NULL)

{

nextptr=curr->next;

curr->next=prev;

prev=curr;

curr=nextptr;

}

return prev;

}

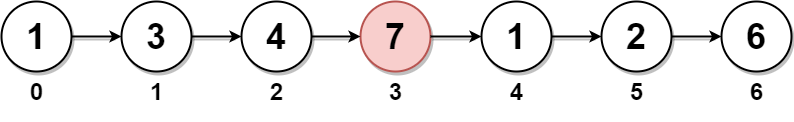
IMPORTANT

18)You are given the head of a linked list. **Delete** the **middle node**, and return *the* head *of the modified linked list*.

The **middle node** of a linked list of size n is the ⌊n / 2⌋th node from the **start** using **0-based indexing**, where ⌊x⌋ denotes the largest integer less than or equal to x.

* For n = 1, 2, 3, 4, and 5, the middle nodes are 0, 1, 1, 2, and 2, respectively.

**Example 1:**



**Input:** head = [1,3,4,7,1,2,6]

**Output:** [1,3,4,1,2,6]

**Explanation:**

The above figure represents the given linked list. The indices of the nodes are written below.

Since n = 7, node 3 with value 7 is the middle node, which is marked in red.

We return the new list after removing this node.

: ListNode\* deleteMiddle(ListNode\* head) {

if(!head->next)

return NULL;

ListNode\*slow=head;

ListNode\*fast=head->next->next;

while(fast&&fast->next)

{

slow=slow->next;

fast=fast->next->next;

}

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

return head;

}

19) 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.

**Example 1:**

**Input:** nums = [2,7,11,15], target = 9

**Output:** [0,1]

**Output:** Because nums[0] + nums[1] == 9, we return [0, 1].

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

vector<int>v;

unordered\_map<int,int>m;

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

{

if(m.find(target-nums[i])!=m.end())

{

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

v.push\_back(i);

return v;

}

else

m[nums[i]]=i;

}

return v;

}

20) Given an unsorted array of integers nums, return *the length of the longest consecutive elements sequence.*

You must write an algorithm that runs in O(n) time.

**Example 1:**

**Input:** nums = [100,4,200,1,3,2]

**Output:** 4

**Explanation:** The longest consecutive elements sequence is [1, 2, 3, 4]. Therefore its length is 4.

: int longestConsecutive(vector<int>& nums) {

unordered\_map<int,int>m;

int r=0;

for(auto i:nums)

{

if(m[i])

continue;

r=max(r,m[i]=m[i-m[i-1]]=m[i+m[i+1]]=m[i+1]+m[i-1]+1);

}

return r;

}

ANOTHER APPROACH

: int longestConsecutive(vector<int>& nums) {

unordered\_set<int>st;

for(auto it:nums)

{

st.insert(it);

}

int best=0;

for(auto it:st)

{

if(st.find(it-1)==st.end())

{

int m=it+1;

while(st.find(m)!=st.end())

m++;

best=max(best,m-it);

}

}

return best;

}

ANOTHER APPROACH

: int longestConsecutive(vector<int>& nums) {

if(nums.size()==0)

return 0;

int curr=1;

int maxi=1;

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

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

{

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

{

if(nums[i]==nums[i-1]+1)

curr+=1;

else

{

maxi=max(maxi,curr);

curr=1;

}

}

}

return max(curr,maxi);

}

21) Given an array having both positive and negative integers. The task is to compute the length of the largest subarray with sum 0.

**Example 1:**

**Input:**

N = 8

A[] = {15,-2,2,-8,1,7,10,23}

**Output:** 5

**Explanation:** The largest subarray with

sum 0 will be -2 2 -8 1 7.

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

{

// Your code here

unordered\_map<int,int>m;

int maxi=0;

int sum=0;

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

{

sum+=A[i];

if(sum==0)

maxi=i+1;

else

{

if(m.find(sum)!=m.end())

maxi=max(maxi,i-m[sum]);

else

m[sum]=i;

}

}

return maxi;

}

22) Given the head of a linked list, 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 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**.

**Do not modify** the linked list.

**Example 1:**



**Input:** head = [3,2,0,-4], pos = 1

**Output:** tail connects to node index 1

**Explanation:** There is a cycle in the linked list, where tail connects to the second node.

ListNode \*detectCycle(ListNode \*head) {

int pos;

cin>>pos;

if(!head&&!head->next)

return head;

ListNode\* slow=head;

ListNode\* fast=head;

bool flag=0;

while(slow&&fast&&fast->next)

{

slow=slow->next;

fast=fast->next->next;

if(slow==fast)

{

flag=1;

break;

}

}

if(flag==1)

{

slow=head;

while(slow!=fast)

{

slow=slow->next;

fast=fast->next;

}

return slow;

}

return NULL;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*PROOF FOR TWO POINTER APPOACH\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Distance travelled by slowPointer before meeting = x + y

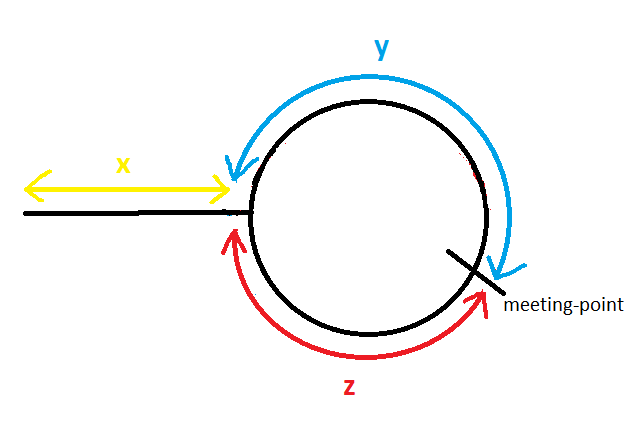
Distance travelled by fastPointer before meeting = (x + y + z) + y = x + 2y + z

Since fastPointer travels with double the speed of slowPointer, and time is constant for both when they reach the meeting point.

So by using simple speed, time and distance relation 2(x+y)= x+2y+z => x+2y+z = 2x+2y => x=z

Hence by moving slowPointer to start of linked list, and making both slowPointer and fastPointer to move one node at a time, they both have same distance to cover .

They will reach at the point where the loop starts in the linked list.

[](https://i.stack.imgur.com/7AUdb.png)

23) Given an array of integers arr[] and a number m, count the number of subarrays having XOR of their elements as m.  
**Examples:**

**Input :** arr[] = {4, 2, 2, 6, 4}, m = 6

**Output :** 4

**Explanation :** The subarrays having XOR of

their elements as 6 are {4, 2},

{4, 2, 2, 6, 4}, {2, 2, 6},

and {6}

: int XOR(vector<int>&v,int k)

{

unordered\_map<int,int­­­­>m;

int cnt=0;

int xor=0;

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

{

xor^=v[i];

if(xor==k)

cnt++;

if(m.find(xor^k)!=m.end())

cnt+=m[xor^k];

m[xor]=1;

}

return cnt;

}

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

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

**Explanation:** The answer is "abc", with the length of 3.

: int lengthOfLongestSubstring(string s) {

vector<int>m(256,-1);

int l=0,r=0;

int n=s.size();

int len=0;

while(r<n)

{

if(m[s[r]]!=-1)

l=max(l,m[s[r]]+1);

m[s[r]]=r;

len=max(len,r-l+1);

r++;

}

return len;

}

25) Merge two sorted linked lists and return it as a **sorted** list. The list should be made by splicing together the nodes of the first two lists.

**Example 1:**



**Input:** l1 = [1,2,4], l2 = [1,3,4]

**Output:** [1,1,2,3,4,4]

: ListNode\* mergeTwoLists(ListNode\* l1, ListNode\* l2) { \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*WITH EXTRA SPACE\*\*\*\*\*\*\*\*\*

ListNode\*dummy=new ListNode();

dummy->next=NULL;

ListNode\*temp=dummy;

while(l1&&l2)

{

if(l1->val<=l2->val)

{

temp->next=l1;

l1=l1->next;

temp=temp->next;

}

else

{

temp->next=l2;

l2=l2->next;

temp=temp->next;

}

}

while(l1)

{

temp->next=l1;

l1=l1->next;

temp=temp->next;

}

while(l2)

{

temp->next=l2;

l2=l2->next;

temp=temp->next;

}

temp=dummy->next;

return temp;

}

26) Given the heads of two singly linked-lists headA and headB, return *the node at which the two lists intersect*. If the two linked lists have no intersection at all, return null.

For example, the following two linked lists begin to intersect at node c1:



The test cases are generated such that there are no cycles anywhere in the entire linked structure.

**Note** that the linked lists must **retain their original structure** after the function returns.

: ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

if(!headA||!headB)

return NULL;

ListNode\*temp1=headA;

ListNode\*temp2=headB;

while(temp1!=temp2)

{

temp1=temp1==NULL?headB:temp1->next;

temp2=temp2==NULL?headA:temp2->next;

}

return temp2 (or temp1);

}

27) Given the head of a linked list, remove the nth node from the end of the list and return its head.

**Example 1:**



**Input:** head = [1,2,3,4,5], n = 2

**Output:** [1,2,3,5]

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

ListNode\*dummy=new ListNode();

dummy->next=head;

ListNode\*slow=dummy;

ListNode\*fast=dummy;

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

fast=fast->next;

while(fast->next)

{

slow=slow->next;

fast=fast->next;

}

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

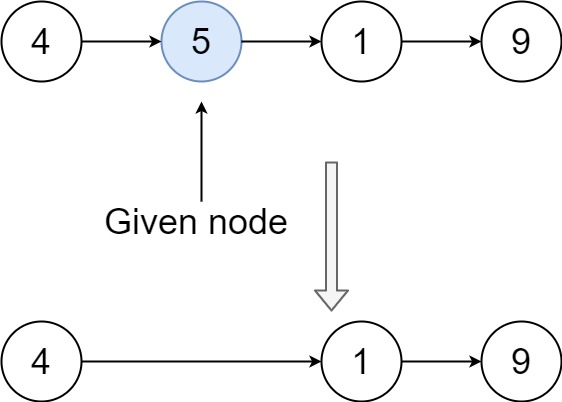
return dummy->next;

}

28) 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.

**Example 1:**



**Input:** head = [4,5,1,9], node = 5

**Output:** [4,1,9]

**Explanation:** You are given the second node with value 5, the linked list should become 4 -> 1 -> 9 after calling your function.

: void deleteNode(ListNode\* node) {

ListNode\*temp=node->next;

swap(node->val,node->next->val);

node->next=node->next->next;

delete temp;

}

29) You are given 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.

You may assume the two numbers do not contain any leading zero, except the number 0 itself.

**Example 1:**



**Input:** l1 = [2,4,3], l2 = [5,6,4]

**Output:** [7,0,8]

**Explanation:** 342 + 465 = 807.

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

ListNode\*dummy=new ListNode();

ListNode\*temp=dummy;

int carry=0;

while(l1||l2||carry)

{

int sum=0;

if(l1)

{

sum+=l1->val;

l1=l1->next;

}

if(l2)

{

sum+=l2->val;

l2=l2->next;

}

sum+=carry;

carry=sum/10;

ListNode\*node=new ListNode(sum%10);

temp->next=node;

temp=temp->next;

}

return dummy->next;

}

30) 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. Internally, pos is used to denote the index of the node that tail's next pointer is connected to. **Note that pos is not passed as a parameter**.

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

**Example 1:**



**Input:** head = [3,2,0,-4], pos = 1

**Output:** true

**Explanation:** There is a cycle in the linked list, where the tail connects to the 1st node (0-indexed).

: bool hasCycle(ListNode \*head) {

ListNode\*slow=head;

ListNode\*fast=head;

while(slow&&fast&&fast->next)

{

slow=slow->next;

fast=fast->next->next;

if(slow==fast)

return 1;

}

return 0;

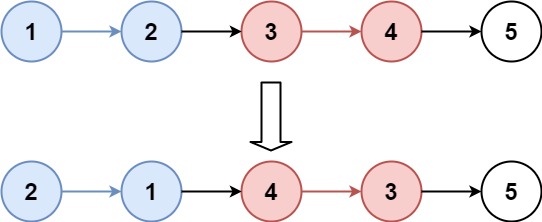
}

31) Given a linked list, reverse the nodes of a linked list *k* at a time and return its 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.

You may not alter the values in the list's nodes, only nodes themselves may be changed.

**Example 1:**



**Input:** head = [1,2,3,4,5], k = 2

**Output:** [2,1,4,3,5]

\*\*\*\*\*\*\*\*\*\*if all remaining nodes are not to reverse\*\*\*\*\*EX : 1 2 3 4 5 ans k=3 => 3 2 1 4 5

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

ListNode\*dummy=new ListNode();

dummy->next=head;

int cnt=0;

ListNode\*curr=dummy;

ListNode\*prev=dummy;

ListNode\*nex=dummy;

while(curr->next)

{

cnt++;

curr=curr->next;

}

while(cnt>=k)

{

curr=prev->next;

nex=curr->next;

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

{

curr->next=nex->next;

nex->next=prev->next;

prev->next=nex;

nex=curr->next;

}

prev=curr;

cnt-=k;

}

return dummy->next;

}

: ListNode\* reverseKGroup(ListNode\* head, int k) { \*\*\*\*\*\*\*\*\*\*if all remaining nodes also to be reverse\*\*\*\*\*EX : 1 2 3 4 5 ans k=3 => 3 2 1 5 4

if(head!=NULL&&k==1)

return head;

int cnt=k;

ListNode\*prev=NULL;

ListNode\*curr=head;

ListNode\*next=NULL;

while(curr!=NULL&&cnt--)

{

next=curr->next;

curr->next=prev;

prev=curr;

curr=next;

}

cnt=k;

if(next)

{

head->next=reverseKGroup(next,cnt);

}

return prev;

}

32) Given the head of a singly linked list, return true if it is a palindrome.

**Example 1:**



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

**Output:** true

: bool isPalindrome(ListNode\* head) {

ListNode\*temp=head;

stack<int>st;

while(temp!=NULL)

{

st.push(temp->val);

temp=temp->next;

}

while(head!=NULL)

{

int top=st.top();

st.pop();

if(head->val!=top)

return 0;

head=head->next;

}

return 1;

}

33) Given the head of a linked list, rotate the list to the right by k places.

**Example 1:**

**Input:** head = [1,2,3,4,5], k = 2

**Output:** [4,5,1,2,3]

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

if(!head||!head->next||k==0)

return head;

ListNode\*curr=head;

int cnt=1;

while(curr->next&&++cnt)

curr=curr->next;

curr->next=head;

k=k%cnt;

k=cnt-k;

while(k--)

curr=curr->next;

head=curr->next;

curr->next=NULL;

return head;

}

34)Given an array, rotate the array to the right by k steps, where k is non-negative.

**Example 1:**

**Input:** nums = [1,2,3,4,5,6,7], k = 3

**Output:** [5,6,7,1,2,3,4]

**Explanation:**

rotate 1 steps to the right: [7,1,2,3,4,5,6]

rotate 2 steps to the right: [6,7,1,2,3,4,5]

rotate 3 steps to the right: [5,6,7,1,2,3,4]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*IMP(MOST ASKED)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: void rotate(vector<int>& nums, int k) {

k%=nums.size();

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

reverse(nums.begin(),nums.begin()+k);

reverse(nums.begin()+k,nums.end());

}

35)

: Given a Linked List 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 next pointer.

**Example 1:**

**Input:**

5 -> 10 -> 19 -> 28

| | | |

7 20 22 35

| | |

8 50 40

| |

30 45

**Output:**  5-> 7-> 8- > 10 -> 19-> 20->

22-> 28-> 30-> 35-> 40-> 45-> 50.

**Explanation**:

The resultant linked lists has every

node in a single level.

(**Note:** | represents the bottom pointer.)

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

{

Node\*dummy=new Node(0);

Node\*temp=dummy;

while(a!=NULL&&b!=NULL)

{

if(a->data<b->data)

{

dummy->bottom=a;

a=a->bottom;

dummy=dummy->bottom;

}

else

{

dummy->bottom=b;

b=b->bottom;

dummy=dummy->bottom;

}

}

if(a!=NULL)

dummy->bottom=a;

else

dummy->bottom=b;

return temp->bottom;

}

Node \*flatten(Node \*root)

{

// Your code here

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

return root;

root->next=flatten(root->next);

root=merge(root,root->next);

return root;

}

36) Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

**Input:** nums = [-1,0,1,2,-1,-4]

**Output:** [[-1,-1,2],[-1,0,1]]

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

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

vector<vector<int>>v;

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

{

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

continue;

int l=i+1,r=nums.size()-1;

while(l<r)

{

int sum=nums[i]+nums[l]+nums[r];

if(sum>0)

r--;

else if(sum<0)

l++;

else

{

v.push\_back(vector<int>{nums[i],nums[l],nums[r]});

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

l++;

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

r--;

l++,r--;

}

}

}

return v;

}

37) Given an integer array nums sorted in **non-decreasing order**, remove the duplicates [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) such that each unique element appears only **once**. The **relative order** of the elements should be kept the **same**.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k after placing the final result in the first k slots of nums.

Do **not** allocate extra space for another array. You must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int[] expectedNums = [...]; // The expected answer with correct length

int k = removeDuplicates(nums); // Calls your implementation

assert k == expectedNums.length;

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

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

**Output:** 2, nums = [1,2,\_]

**Explanation:** Your function should return k = 2, with the first two elements of nums being 1 and 2 respectively.

It does not matter what you leave beyond the returned k (hence they are underscores).

: int removeDuplicates(vector<int>& nums) {

if(nums.size()==0)

return 0;

int i=0;

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

{

if(nums[j]!=nums[i])

i++;

nums[i]=nums[j];

}

return i+1;

}

38) Given a binary array nums, return *the maximum number of consecutive*1*'s in the array*.

**Example 1:**

**Input:** nums = [1,1,0,1,1,1]

**Output:** 3

**Explanation:** The first two digits or the last three digits are consecutive 1s. The maximum number of consecutive 1s is 3.

: int findMaxConsecutiveOnes(vector<int>& nums) {

int cnt=0,maximum=0;

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

{

if(nums[i]==1)

cnt++;

maximum=max(maximum,cnt);

if(nums[i]==0)

cnt=0;

}

return maximum;

}

39) Given n non-negative integers representing an elevation map where the width of each bar is 1, compute how much water it can trap after raining.

 \*\*\*\*\*\*\*\*\*RAINWATER TRAPPING PROBLEM\*\*\*\*\*\*\*\*\*\*\*(FAMOUS)\*\*\*\*\*\*\*

**Example 1:**



**Input:** height = [0,1,0,2,1,0,1,3,2,1,2,1]

**Output:** 6

**Explanation:** The above elevation map (black section) is represented by array [0,1,0,2,1,0,1,3,2,1,2,1]. In this case, 6 units of rain water (blue section) are being trapped.

: int trap(vector<int>& height) {

int n=height.size();

int l=0,r=n-1;

int maxl=0,maxr=0;

int ans=0;

while(l<=r)

{

if(height[l]<=height[r])

{

if(height[l]>=maxl)

maxl=height[l];

else

ans+=maxl-height[l];

l++;

}

else

{

if(height[r]>=maxr)

maxr=height[r];

else

ans+=maxr-height[r];

r--;

}

}

return ans;

}

40) Given a string S find all possible subsequences of the string in lexicographically-sorted order.

**Example 1:**

**Input :** str = "abc"

**Output:** a ab abc ac b bc c

**Explanation :** There are 7 substrings that

can be formed from abc.

: vector<string> AllPossibleStrings(string s){

// Code here

vector<string>v;

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

{

string str="";

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

{

if(i&(1<<j))

str+=s[j];

}

v.push\_back(str);

}

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

return v;

}

41) Given a list **arr** of **N** integers, print sums of all subsets in it.

**Example 1:**

**Input:**

N = 2

arr[] = {2, 3}

**Output:**

0 2 3 5

**Explanation:**

When no elements is taken then Sum = 0.

When only 2 is taken then Sum = 2.

When only 3 is taken then Sum = 3.

When element 2 and 3 are taken then

Sum = 2+3 = 5.

: void subsum(int ind,int sum,vector<int>&arr,int N,vector<int>&v)

{

if(ind==N)

{

v.push\_back(sum);

return;

}

subsum(ind+1,sum+arr[ind],arr,N,v);

subsum(ind+1,sum,arr,N,v);

}

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

{

// Write Your Code here

vector<int>v;

subsum(0,0,arr,N,v);

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

return v;

}

42) Given an integer array nums that may contain duplicates, return *all possible subsets (the power set)*.

The solution set **must not** contain duplicate subsets. Return the solution in **any order**.

**Example 1:**

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

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

: void func(int ind,vector<int>&nums,vector<int>&temp,vector<vector<int>>&v)

{

v.push\_back(temp);

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

{

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

continue;

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

func(i+1,nums,temp,v);

temp.pop\_back();

}

}

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

vector<vector<int>>v;

vector<int>temp;

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

func(0,nums,temp,v);

return v;

}

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

The **same** number may be chosen from candidates 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.

**Example 1:**

**Input:** candidates = [2,3,6,7], target = 7

**Output:** [[2,2,3],[7]]

**Explanation:**

2 and 3 are candidates, and 2 + 2 + 3 = 7. Note that 2 can be used multiple times.

7 is a candidate, and 7 = 7.

These are the only two combinations.

: void func(int ind,int target,vector<int>&candidates,vector<int>&temp,vector<vector<int>>&v)

{

if(ind==candidates.size())

{

if(target==0)

v.push\_back(temp);

return ;

}

if(candidates[ind]<=target)

{

temp.push\_back(candidates[ind]);

func(ind,target-candidates[ind],candidates,temp,v);

temp.pop\_back();

}

func(ind+1,target,candidates,temp,v);

}

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

vector<vector<int>>v;

vector<int>temp;

func(0,target,candidates,temp,v);

return v;

}

44) 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.

**Example 1:**

**Input:** candidates = [10,1,2,7,6,1,5], target = 8

**Output:**

[

[1,1,6],

[1,2,5],

[1,7],

[2,6]

]

: void func(int ind,int target,vector<int>&temp,vector<int>&candidates,vector<vector<int>>&v)

{

if(target==0)

{

v.push\_back(temp);

return;

}

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

{

if(i>ind&&candidates[i]==candidates[i-1])

continue;

if(candidates[i]>target)

break;

temp.push\_back(candidates[i]);

func(i+1,target-candidates[i],temp,candidates,v);

temp.pop\_back();

}

}

vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {

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

vector<vector<int>>v;

vector<int>temp;

func(0,target,temp,candidates,v);

return v;

}

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

A **palindrome** string is a string that reads the same backward as forward.

**Example 1:**

**Input:** s = "aab"

**Output:** [["a","a","b"],["aa","b"]]

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

vector<vector<string>>v;

if(s.empty())

return v;

vector<string>temp;

func(0,s,temp,v);

return v;

}

void func(int ind,string&s,vector<string>&temp,vector<vector<string>>&v)

{

if(ind==s.size())

{

v.push\_back(temp);

return;

}

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

{

if(is\_palindrome(s,ind,i))

{

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

func(i+1,s,temp,v);

temp.pop\_back();

}

}

}

bool is\_palindrome(string s,int l,int k)

{

while(l<=k)

{

if(s[l++]!=s[k--])

return 0;

}

return 1;

}

46) The set [1, 2, 3, ..., n] contains a total of n! unique permutations.

By listing and labeling all of the permutations in order, we get the following sequence for n = 3:

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

Given n and k, return the kth permutation sequence.

**Example 1:**

**Input:** n = 3, k = 3

**Output:** "213"

: string getPermutation(int n, int k) {

vector<int>v;

int fact=1;

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

{

fact=fact\*i;

v.push\_back(i);

}

v.push\_back(n);

k-=1;

string s="";

while(1)

{

s+=to\_string(v[k/fact]);

v.erase(v.begin()+k/fact);

if(v.size()==0)

break;

k%=fact;

fact=fact/v.size();

}

return s;

}

47) Given a non-negative integer **N**. The task is to check if N is a power of **2**. More formally, check if**N**can be expressed as **2x**for some **x.**

**Example 1:**

**Input:** N = 1

**Output:** true

**Explanation:**

1 is equal to 2 raised to 0 (20 = 1).

: bool isPowerofTwo(long long n){ \*\*\*\*\*\*\*\*using log method\*\*\*\*\*\*\*\*

Your code here

if(n==0)

return 0;

if(ceil(log2(n))==floor(log2(n)))

return 1;

else

return 0;

}

\*\*\*\*\*\*\*\*\*\*without using log method\*\*\*\*\*\*

**bool** isPowerOfTwo (**int** x)

{

    /\* First x in the below expression is for the case when x is 0 \*/

**return** x && (!(x&(x-1)));

}

48) You are given a number**N**. Find the **total count of set bits**for all numbers from 1 to N(both inclusive).  
  
**Example 1:**

**Input**: N = 4

**Output**: 5

**Explanation**:

For numbers from 1 to 4.

For 1: 0 0 1 = 1 set bits

For 2: 0 1 0 = 1 set bits

For 3: 0 1 1 = 2 set bits

For 4: 1 0 0 = 1 set bits

Therefore, the total set bits is 5.

: int countSetBits(int n)

{

// Your logic here

if(n==0)

return 0;

int x=power2(n);

return (x\*(1<<(x-1)))+(n-(1<<x)+1)+countSetBits(n-(1<<x));

}

int power2(int x)

{

int cnt=0;

while((1<<cnt)<=x)

cnt++;

return cnt-1;

}

49) There is **one** meeting room in a firm. There are **N** meetings in the form of **(start[i], end[i])** where **start[i]**is start time of meeting **i**and **end[i]**is finish time of meeting **i.**  
What is the **maximum** number of meetings that can be accommodated in the meeting room when only one meeting can be held in the meeting room at a particular time?

**Note:** Start time of one chosen meeting can't be equal to the end time of the other chosen meeting.

**Example 1:**

**Input:**

N = 6

start[] = {1,3,0,5,8,5}

end[] = {2,4,6,7,9,9}

**Output:**

4

**Explanation:**

Maximum four meetings can be held with

given start and end timings.

The meetings are - (1, 2),(3, 4), (5,7) and (8,9)

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

{

// Your code here

int cnt=1;

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 temp=v[0].first;

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

{

if(temp<v[i].second)

{

cnt++;

temp=v[i].first;

}

}

return cnt;

}

ANOTHER SAME PROBLEM

Given **N** activities with their start and finish day given in array **start[ ]** and **end[ ]**. Select the maximum number of activities that can be performed by a single person, assuming that a person can only work on a single activity at a given day.  
**Note :**Duration of the activity includes both starting and ending day.

**Example 1:**

**Input:**

N = 2

start[] = {2, 1}

end[] = {2, 2}

**Output:**

1

**Explanation:**

A person can perform only one of the

given activities.

: int activitySelection(vector<int> start, vector<int> end, int n)

{

int cnt=1;

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 temp=v[0].first;

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

{

if(temp<v[i].second)

{

cnt++;

temp=v[i].first;

}

}

return cnt;

}

50) Given arrival and departure times of all trains that reach a railway station. Find the minimum number of platforms required for the railway station so that no train is kept waiting.  
Consider that all the trains arrive on the same day and leave on the same day. Arrival and departure time can never be the same for a train but we can have arrival time of one train equal to departure time of the other. At any given instance of time, same platform can not be used for both departure of a train and arrival of another train. In such cases, we need different platforms**.**

**Example 1:**

**Input**: n = 6

arr[] = {0900, 0940, 0950, 1100, 1500, 1800}

dep[] = {0910, 1200, 1120, 1130, 1900, 2000}

**Output**: 3

**Explanation**:

Minimum 3 platforms are required to

safely arrive and depart all trains.

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

{

// Your code here

sort(arr,arr+n);

sort(dep,dep+n);

int ans=1,platform=1;

int i=1,j=0;

while(i<n&&j<n)

{

if(arr[i]<=dep[j])

{

platform++;

i++;

}

else if(arr[i]>dep[j])

{

platform--;

j++;

}

ans=max(ans,platform);

}

return ans;

}

51) Given an array nums of distinct integers, return *all the possible permutations*. You can return the answer in **any order**.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** [[1,2,3],[1,3,2],[2,1,3],[2,3,1],[3,1,2],[3,2,1]]

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

vector<vector<int>>v;

permutation(nums,0,v);

return v;

}

void permutation(vector<int>&nums,int begin,vector<vector<int>>&v)

{

if(begin>=nums.size())

{

v.push\_back(nums);

return;

}

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

{

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

permutation(nums,begin+1,v);

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

}

}

52)Given a matrix of integers **A** of size **N x M** in which each row is sorted.

Find an return the overall median of the 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 Solution::findMedian(vector<vector<int> > &A) {

    int l=0,r=pow(10,9),n=A.size(),mid,m=A[0].size(),c,q,p;

q=(n\*m)/2;

while(l<=r){

    mid=(l+r)/2;

    c=0;

    for(int i=0;i<n;i++)

        c+=lower\_bound(begin(A[i]),end(A[i]),mid)-begin(A[i]);

    if(c<=q){

        p=mid;

        l=mid+1;

    }

53) 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.

**Example 1:**

**Input:** s = "the sky is blue"

**Output:** "blue is sky the"

: string reverseWords(string s) {

string res="";

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

{

if(s[i]==' ')continue;

int pos=i;

while(i<s.size()&&s[i]!=' ')

i++;

if(res.size()>0)

res=' '+res;

res=s.substr(pos,i-pos)+res;

i--;

}

return res;

}

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

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

: **int expandfromcentre(string&s,int l,int r)**

**{**

**if(s.size()<1||left>right)**

**return 0;**

**while(l>=0&&r<s.size()&&s[l]==s[r])**

**{**

**r++;**

**l--;**

**}**

**return r-l-1;**

**}**

**string longestPalindrome(string s) {**

**string ans;**

**int len=0;**

**int start=0,end=0;**

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

**{**

**int len1=expandfromcentre(s,i,i);**

**int len2=expandfromcentre(s,i,i+1);**

**if(len<max(len1,len2))**

**{**

**len=max(len1,len2);**

**start=i-(len-1)/2;**

**}**

**}**

**return s.substr(start,len);**

**}**

**ANOTHER VARIATION ON SAME CONCEPT**

Given a string s, return *the number of****palindromic substrings****in it*.

A string is a **palindrome** when it reads the same backward as forward.

A **substring** is a contiguous sequence of characters within the string.

**Example 1:**

**Input:** s = "abc"

**Output:** 3

**Explanation:** Three palindromic strings: "a", "b", "c".

: void func(string s,int l,int r,int&ans)

{

while(l>=0&&r<s.size()&&s[l]==s[r])

{

ans++;

r++;

l--;

}

}

int countSubstrings(string s) {

int ans=0;

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

{

func(s,i,i,ans);

func(s,i,i+1,ans);

}

return ans;

}

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

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 1000

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given a roman numeral, convert it to an integer.

**Example 1:**

**Input:** s = "III"

**Output:** 3

: int romanToInt(string s) {

unordered\_map<char,int>mp={{'I',1},{'V',5},{'X',10},{'L',50},{'C',100},{'D',500},{'M',1000}};

int sum=mp[s.back()];

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

{

sum+=mp[s[i]]<mp[s[i+1]]?-mp[s[i]]:mp[s[i]];

}

return sum;

}

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

**Symbol** **Value**

I 1

V 5

X 10

L 50

C 100

D 500

M 10add00

For example, 2 is written as II in Roman numeral, just two one's added together. 12 is written as XII, which is simply X + II. The number 27 is written as XXVII, which is XX + V + II.

Roman numerals are usually written largest to smallest from left to right. However, the numeral for four is not IIII. Instead, the number four is written as IV. Because the one is before the five we subtract it making four. The same principle applies to the number nine, which is written as IX. There are six instances where subtraction is used:

* I can be placed before V (5) and X (10) to make 4 and 9.
* X can be placed before L (50) and C (100) to make 40 and 90.
* C can be placed before D (500) and M (1000) to make 400 and 900.

Given an integer, convert it to a roman numeral.

**Example 1:**

**Input:** num = 3

**Output:** "III"

: string intToRoman(int num) {

string res;

string arr[]={"M","CM","D","CD","C","XC","L","XL","X","IX","V","IV","I"};

int val[]={1000,900,500,400,100,90,50,40,10,9,5,4,1};

for(int i=0;num!=0;i++)

{

while(num>=val[i])

{

num-=val[i];

res+=arr[i];

}

}

return res;

}

57) 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 "".

**Example 1:**

**Input:** strs = ["flower","flow","flight"]

**Output:** "fl"

: string longestCommonPrefix(vector<string>& strs) {

if(strs.empty()) return "";

string prefix = strs[0];

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

{

while(strs[i].find(prefix) != 0)

{

prefix = prefix.substr(0, prefix.size()-1);

}

if(prefix.empty()) return "";

}

return prefix;

}

58) Find MSB in o(1)

: int findMSB(int n){

// Write your code here

int ans=1;

while(n!=1)

{

n/=2;

ans=ans\*2;

}

return ans;

}

59) Find the square of a number without using the multiplication and division operator

: int findSquare(int num)

{

int odd = 1;

int sq = 0;

// convert the number to positive if it is negative

num = abs(num);

// add odd numbers num times to result

while (num--)

{

sq = sq + odd;

odd = odd + 2;

}

return sq;

}

60) You are given a sorted array consisting of only integers where every element appears exactly twice, except for one element which appears exactly once.

Return *the single element that appears only once*.

Your solution must run in O(log n) time and O(1) space.

**Example 1:**

**Input:** nums = [1,1,2,3,3,4,4,8,8]

**Output:** 2

: int singleNonDuplicate(vector<int>& nums) {

int xorr=0;

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

{

xorr^=nums[i];

}

return xorr;

}

61) There is an integer array nums sorted in ascending order (with **distinct** values).

Prior to being passed to your function, nums is **possibly rotated** at an unknown pivot index k (1 <= k < nums.length) such that the resulting array is [nums[k], nums[k+1], ..., nums[n-1], nums[0], nums[1], ..., nums[k-1]] (**0-indexed**). For example, [0,1,2,4,5,6,7] might be rotated at pivot index 3 and become [4,5,6,7,0,1,2].

Given the array nums **after** the possible rotation and an integer target, return *the index of*target*if it is in*nums*, or*-1*if it is not in*nums.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [4,5,6,7,0,1,2], target = 0

**Output:** 4

: int search(vector<int>& nums, int target) {

int lo=0,hi=nums.size()-1;

while(lo<hi)

{

int mid=(lo+hi)/2;

if(nums[mid]>nums[hi])

lo=mid+1;

else

hi=mid;

}

int rot=lo;

lo=0,hi=nums.size()-1;

while(lo<=hi)

{

int mid=(lo+hi)/2;

int realmid=(rot+mid)%nums.size();

if(nums[realmid]==target)

return realmid;

if(nums[realmid]<target)

lo=mid+1;

else

hi=mid-1;

}

return -1;

}

ANOTHER VARIATION ON ROTATED ARRAY

Suppose an array of length n sorted in ascending order is **rotated** between 1 and n times. For example, the array nums = [0,1,2,4,5,6,7] might become:

* [4,5,6,7,0,1,2] if it was rotated 4 times.
* [0,1,2,4,5,6,7] if it was rotated 7 times.

Notice that **rotating** an array [a[0], a[1], a[2], ..., a[n-1]] 1 time results in the array [a[n-1], a[0], a[1], a[2], ..., a[n-2]].

Given the sorted rotated array nums of **unique** elements, return *the minimum element of this array*.

You must write an algorithm that runs in O(log n) time.

**Example 1:**

**Input:** nums = [3,4,5,1,2]

**Output:** 1

**Explanation:** The original array was [1,2,3,4,5] rotated 3 times.

: int findMin(vector<int>& nums) {

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

{

if(nums[i]>nums[i+1])

return nums[i+1];

}

return nums[0];

}

ANOTHER CODE VARIATION USING BINARY SEARCH

:int findMin(vector<int>& nums) {

int lo=0;

int hi=nums.size()-1;

while(lo<hi)

{

if(nums[lo]<nums[hi])

return nums[lo];

int mid=(lo+hi)/2;

if(nums[mid]>=nums[hi])

lo=mid+1;

else

{

hi=mid;

lo++;

}

}

return nums[lo];

}

62)stack implementation

:class{

Public;

Int arr[1000];

Int top=-1;

Void push(int x)

{

Arr[top++]=x;

}

Void pop()

{

Top--;

}

Int top()

{

Return arr[top-1];

}

Bool is\_empty()

{

If(top==-1)

Return 0;

Else

Return 1;

}

}

63) **Implement Queue Using Arrays**

:class queue{

Public:

Int arr[1000];

Int n=arr.size();

Int front=0;

Int rear=0;

Int size=0;

Void push(int x)

{

If(size==n)

Cout<<”not possible”;

Arr[rear%n]=x;

Rear++;

Size++;

}

Void pop()

{

If(size==0)

Cout<<”not possible”;

Arr[front%n]=-1;

Front++;

Size--;

}

Int top()

{

If(size==0)

Return -1;

Return Arr[front%n];

}

};

64) **Implement Stack using Queue (using single queue)**

: queue<int>q;

MyStack() {

}

void push(int x) {

q.push(x);

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

q.push(q.front());

q.pop();

}

}

void pop() {

q.pop();

}

int top() {

return q.top();

}

bool empty() {

return q.empty();

}

};

65)implement queue using stack

:void push(int x)

{

Input.push(x);

}

void pop()

{

If(!output.empty())

{

Output.pop();

Else

{

While(!input.empty())

{

Output.push(input.front());

Input.pop();

}

}

}

}

Int top()

{

If(!output.empty())

{

Return output.top();

Else

{

While(!input.empty())

{

Output.push(input.front())

Input.pop();

}

}

}

}

}

66) Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (push, peek, pop, and empty).

Implement the MyQueue class:

* void push(int x) Pushes element x to the back of the queue.
* int pop() Removes the element from the front of the queue and returns it.
* int peek() Returns the element at the front of the queue.
* boolean empty() Returns true if the queue is empty, false otherwise.

**Notes:**

* You must use **only** standard operations of a stack, which means only push to top, peek/pop from top, size, and is empty operations are valid.
* Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

**Example 1:**

**Input**

["MyQueue", "push", "push", "peek", "pop", "empty"]

[[], [1], [2], [], [], []]

**Output**

[null, null, null, 1, 1, false]

**Explanation**

MyQueue myQueue = new MyQueue();

myQueue.push(1); // queue is: [1]

myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)

myQueue.peek(); // return 1

myQueue.pop(); // return 1, queue is [2]

myQueue.empty(); // return false

: class MyQueue {

public:

stack<int>input;

stack<int>output;

MyQueue() {

}

void push(int x) {

input.push(x);

}

int pop() {

if((!output.empty()))

output.pop();

else

{

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

output.push(input.top());

input.pop();

}

}

return output.pop();

}

int peek() {

if(!output.empty())

return output.top();

else

{

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

{

output.push(input.top());

input.pop();

}

}

output.pop();

}

bool empty() {

}

};

/\*\*

\* Your MyQueue object will be instantiated and called as such:

\* MyQueue\* obj = new MyQueue();

\* obj->push(x);

\* int param\_2 = obj->pop();

\* int param\_3 = obj->peek();

\* bool param\_4 = obj->empty();

\*/

67) Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

**Example 1:**

**Input:** s = "()"

**Output:** true

: bool isValid(string s) {

stack<char>st;

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

{

if(s[i]=='['||s[i]=='('||s[i]=='{')

st.push(s[i]);

else

{

if(st.empty())

return 0;

char k=st.top();

st.pop();

if((s[i]==')'&&k=='(')||(s[i]==']'&&k=='[')||s[i]=='}'&&k=='{')

continue;

else

return 0;

}

}

if(st.empty())

return 1;

else

return 0;

}

68) Given a circular integer array nums (i.e., the next element of nums[nums.length - 1] is nums[0]), return *the****next greater number****for every element in* nums.

The **next greater number** of a number x is the first greater number to its traversing-order next in the array, which means you could search circularly to find its next greater number. If it doesn't exist, return -1 for this number.

**Example 1:**

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

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

Explanation: The first 1's next greater number is 2;

The number 2 can't find next greater number.

The second 1's next greater number needs to search circularly, which is also 2.

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

stack<int>st;

int n=nums.size();

vector<int>ans(n,-1);

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;

}

69) Given a stack, the task is to sort it such that the top of the stack has the greatest element.

**Example 1:**

**Input:**

Stack: 3 2 1

**Output:** 3 2 1

:

/\*The structure of the class is

class SortedStack{

public:

stack<int> s;

void sort();

};

\*/

/\* The below method sorts the stack s

you are required to complete the below method \*/

void insert(stack<int>&s,int xx)

{

if(s.size()==0||xx>s.top())

s.push(xx);

else

{

int x=s.top();

s.pop();

insert(s,xx);

s.push(x);

}

}

void reverse(stack<int>&s)

{

if(s.size()>0)

{

int a=s.top();

s.pop();

reverse(s);

insert(s,a);

}

}

void SortedStack :: sort()

{

//Your code here

reverse(s);

}

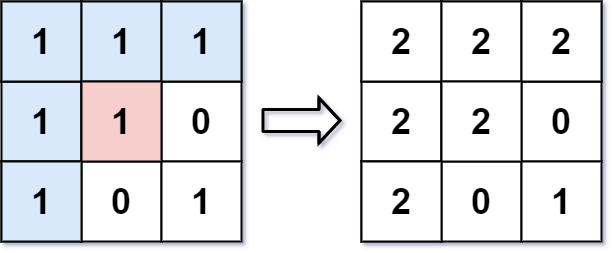
70) An image is represented by an m x n integer grid image where image[i][j] represents the pixel value of the image.

You are also given three integers sr, sc, and newColor. 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 newColor.

Return *the modified image after performing the flood fill*.

**Example 1:**



**Input:** image = [[1,1,1],[1,1,0],[1,0,1]], sr = 1, sc = 1, newColor = 2

**Output:** [[2,2,2],[2,2,0],[2,0,1]]

**Explanation:** From the center of the image with position (sr, sc) = (1, 1) (i.e., the red pixel), all pixels connected by a path of the same color as the starting pixel (i.e., the blue pixels) are colored with the new color.

Note the bottom corner is not colored 2, because it is not 4-directionally connected to the starting pixel.

: vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int newColor) {

int r=image.size();

int c=image[0].size();

vector<vector<int>>vis(r,vector<int>(c,0));

int oldcolor=image[sr][sc];

dfs(vis,image,sr,sc,r,c,oldcolor,newColor);

return image;

}

void dfs(vector<vector<int>>&vis,vector<vector<int>>&image,int sr,int sc,int x,int y,int oldcolor,int newcolor)

{

if(sr<0 || sc<0||sr>=x||sc>=y)

return;

if(vis[sr][sc]||image[sr][sc]!=oldcolor)

return;

vis[sr][sc]=1;

image[sr][sc]=newcolor;

dfs(vis,image,sr+1,sc,x,y,oldcolor,newcolor);

dfs(vis,image,sr-1,sc,x,y,oldcolor,newcolor);

dfs(vis,image,sr,sc+1,x,y,oldcolor,newcolor);

dfs(vis,image,sr,sc-1,x,y,oldcolor,newcolor);

}

71) You are given row x col grid representing a map where grid[i][j] = 1 represents land and grid[i][j] = 0 represents water.

Grid cells are connected **horizontally/vertically** (not diagonally). The grid is completely surrounded by water, and there is exactly one island (i.e., one or more connected land cells).

The island doesn't have "lakes", meaning the water inside isn't connected to the water around the island. One cell is a square with side length 1. The grid is rectangular, width and height don't exceed 100. Determine the perimeter of the island.

**Example 1:**



**Input:** grid = [[0,1,0,0],[1,1,1,0],[0,1,0,0],[1,1,0,0]]

**Output:** 16

**Explanation:** The perimeter is the 16 yellow stripes in the image above.

**Example 2:**

**Input:** grid = [[1]]

**Output:** 4

: int islandPerimeter(vector<vector<int>>& grid) {

int cnt=0;

int peri=0;

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

{

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

{

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

{

cnt++;

if(i!=0&&grid[i-1][j]==1)

peri++;

if(j!=0&&grid[i][j-1]==1)

peri++;

}

}

}

return 4\*cnt-2\*peri;

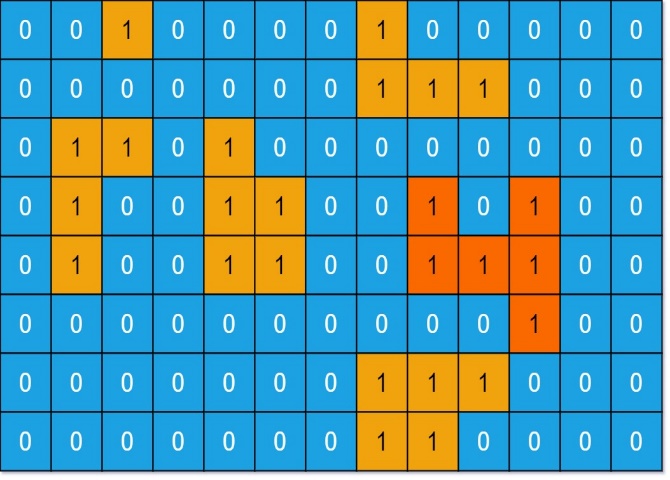
}

72) You are given an m x n binary matrix grid. An island is a group of 1's (representing land) connected **4-directionally** (horizontal or vertical.) You may assume all four edges of the grid are surrounded by water.

The **area** of an island is the number of cells with a value 1 in the island.

Return *the maximum****area****of an island in*grid. If there is no island, return 0.

**Example 1:**



**Input:** grid = [[0,0,1,0,0,0,0,1,0,0,0,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,1,1,0,1,0,0,0,0,0,0,0,0],[0,1,0,0,1,1,0,0,1,0,1,0,0],[0,1,0,0,1,1,0,0,1,1,1,0,0],[0,0,0,0,0,0,0,0,0,0,1,0,0],[0,0,0,0,0,0,0,1,1,1,0,0,0],[0,0,0,0,0,0,0,1,1,0,0,0,0]]

**Output:** 6

**Explanation:** The answer is not 11, because the island must be connected 4-directionally.

: int maxAreaOfIsland(vector<vector<int>>& grid) {

int maxland=0;

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

{

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

{

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

maxland=max(maxland,landarea(grid,i,j));

}

}

return maxland;

}

int landarea(vector<vector<int>>&grid,int i,int j)

{

if(i>=0&&i<grid.size()&&j>=0&&j<grid[0].size()&&grid[i][j]==1)

{

grid[i][j]=0;

return 1+landarea(grid,i+1,j)+landarea(grid,i-1,j)+landarea(grid,i,j-1)+landarea(grid,i,j+1);

}

return 0;

}

73) Given an m x n 2D binary grid grid which represents a map of '1's (land) and '0's (water), return *the number of islands*.

An **island** is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water.

**Example 1:**

**Input:** grid = [

["1","1","1","1","0"],

["1","1","0","1","0"],

["1","1","0","0","0"],

["0","0","0","0","0"]

]

**Output:** 1

**Example 2:**

**Input:** grid = [

["1","1","0","0","0"],

["1","1","0","0","0"],

["0","0","1","0","0"],

["0","0","0","1","1"]

]

**Output:** 3

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

int land=0;

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

{

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

{

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

{

dfs(grid,i,j);

++land;

}

}

}

return land;

}

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

{

if(i<0||i>=grid.size()||j<0||j>=grid[0].size())

return;

if(grid[i][j]=='2'||grid[i][j]=='0')

return;

grid[i][j]='2';

dfs(grid,i+1,j);

dfs(grid,i,j-1);

dfs(grid,i-1,j);

dfs(grid,i,j+1);

}

74) Given a string s, *find the first non-repeating character in it and return its index*. If it does not exist, return -1.

**Example 1:**

**Input:** s = "leetcode"

**Output:** 0

**Example 2:**

**Input:** s = "loveleetcode"

**Output:** 2

: int firstUniqChar(string s) {

vector<int>v(256,0);

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

{

v[s[i]-'a']++;

}

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

{

if(v[s[i]-'a']==1)

return i;

}

return -1;

}

75) Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

**Example 1:**

**Input:** root = [1,null,2,3]

**Output:** [1,3,2]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RECURSIVE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: vector<int> inorderTraversal(TreeNode\* root) {

vector<int>v;

inorder(root,v);

return v;

}

void inorder(TreeNode\*root,vector<int>&v)

{

if(!root)

return;

inorder(root->left,v);

v.push\_back(root->val);

inorder(root->right,v);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ITERATIVE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void inorder\_iterative(Root\*root)               //NON RECURSIVE INORDER

{                                                //O(n)  O(n)

    stack<Root\*>st;

    while(1)

    {

        while(root!=NULL)

        {

            st.push(root);

            root=root->left;

        }

        if(st.empty())

        break;

        root=st.top();

        st.pop();

        cout<<root->data<<" ";

        root=root->right;

    }

    while(!st.empty())

    st.pop();

}

76) Given the root of a binary tree, return *the preorder traversal of its nodes' values*.

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [1,2,3]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RECURSIVE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: vector<int> preorderTraversal(TreeNode\* root) {

vector<int>v;

preorder(root,v);

return v;

}

void preorder(TreeNode\*root,vector<int>&v)

{

if(!root)

return;

v.push\_back(root->val);

preorder(root->left,v);

preorder(root->right,v);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ITERATIVE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void preorder\_iterative(Root\*root)            //NON RECURSIVE PREORDER

{                                              //O(n)  O(n)

    stack<Root\*>st;

    while(1)

    {

        while(root!=NULL);

        {

            cout<<root->data<<" ";

            st.push(root);

            root=root->left;

        }

        if(st.empty())

        break;

        root=st.top();

        st.pop();

        root=root->right;

    }

    while(!st.empty())

    st.pop();

}

77) Given the root of a binary tree, return *the postorder traversal of its nodes' values*.

**Example 1:**



**Input:** root = [1,null,2,3]

**Output:** [3,2,1]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RECURSIVE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: vector<int> postorderTraversal(TreeNode\* root) {

vector<int>v;

postorder(root,v);

return v;

}

void postorder(TreeNode\*root,vector<int>&v)

{

if(!root)

return;

postorder(root->left,v);

postorder(root->right,v);

v.push\_back(root->val);

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ITERATIVE(USING TWO STACKS)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: vector<int> postorderTraversal(TreeNode\* root) {

vector<int>post;

if(!root)

return post;

stack<TreeNode\*>st1,st2;

st1.push(root);

while(!st1.empty())

{

root=st1.top();

st1.pop();

st2.push(root);

if(root->left!=NULL)

st1.push(root->left);

if(root->right!=NULL)

st1.push(root->right);

}

while(!st2.empty())

{

post.push\_back(st2.top()->val);

st2.pop();

}

return post;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ITERATIVE(USING SINGLE STACK)\*\*\*\*\*\*\*\*\*\*\*\*

:

|  |
| --- |
| vector<int> postorderTraversal(TreeNode\* root) { |
|  | vector<int> postorder; |
|  | if(root == NULL) return postorder; |
|  | stack<TreeNode\*> st1; |
|  | TreeNode\* current = root; |
|  | while(current != NULL || !st1.empty()) { |
|  | if(current != NULL){ |
|  | st1.push(current); |
|  | current = current->left; |
|  | }else{ |
|  | TreeNode\* temp = st1.top()->right; |
|  | if (temp == NULL) { |
|  | temp = st1.top(); |
|  | st1.pop(); |
|  | postorder.push\_back(temp->val); |
|  | while (!st1.empty() && temp == st1.top()->right) { |
|  | temp = st1.top(); |
|  | st1.pop(); |
|  | postorder.push\_back(temp->val); |
|  | } |
|  | } else { |
|  | current = temp; |
|  | } |
|  | } |
|  | } |
|  | return postorder; |

78) Level order Traversal

: void level\_order(Root\*root)                        //LEVEL ORDER TRAVERSAL(iteratively)

{                                                  //USING QUEUE

                                                   //O(n)   O(n)

    if(root==NULL)

    return;

    queue<Root\*>q;

    Root\*temp;

    q.push(root);

    while(!q.empty())

    {

        temp=q.front();

        q.pop();

        cout<<temp->data<<" ";

        if(temp->left!=NULL)

        q.push(temp->left);

        if(temp->right!=NULL)

        q.push(temp->right);

    }

    while(!q.empty())

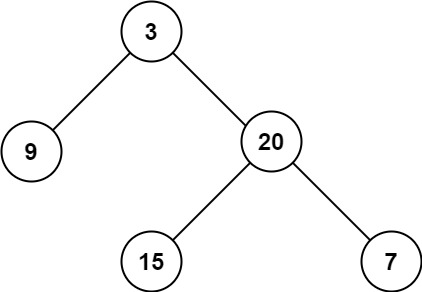
    q.pop();

}

78) Given the root of a binary tree, return *its maximum depth*.

A binary tree's **maximum depth(height)** is the number of nodes along the longest path from the root node down to the farthest leaf node.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** 3

: int maxDepth(TreeNode\* root) {

if(root==NULL)

return 0;

int l=maxDepth(root->left);

int r=maxDepth(root->right);

return 1+max(l,r);

}

FOR N-ARY TREE

Given a n-ary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value (See examples).*

**Example 1:**



**Input:** root = [1,null,3,2,4,null,5,6]

**Output:** 3

: int maxDepth(Node\* root) {

if(!root)

return 0;

int maxi=0;

for(auto it:root->children)

maxi=max(maxi,maxDepth(it));

return 1+maxi;

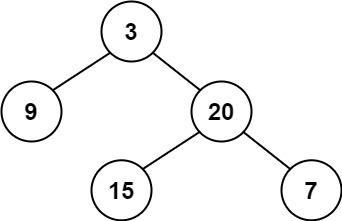
}

79) Given a binary tree, determine if it is height-balanced.

For this problem, a height-balanced binary tree is defined as:

a binary tree in which the left and right subtrees of *every* node differ in height by no more than 1.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** true

: bool isBalanced(TreeNode\* root) {

return dfs(root)!=-1;

}

int dfs(TreeNode\*root)

{

if(root==NULL)

return 0;

int l=dfs(root->left);

if(l==-1)

return -1;

int r=dfs(root->right);

if(r==-1)

return -1;

if(abs(l-r)>1)

return -1;

return 1+max(l,r);

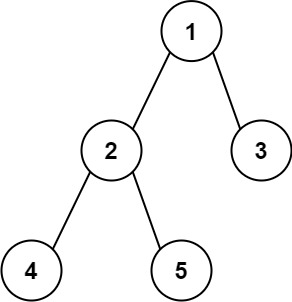
}

80) Given the root of a binary tree, return *the length of the****diameter****of the tree*.

The **diameter** of a binary tree is the **length** of the longest path between any two nodes in a tree. This path may or may not pass through the root.

The **length** of a path between two nodes is represented by the number of edges between them.

**Example 1:**



**Input:** root = [1,2,3,4,5]

**Output:** 3

**Explanation:** 3 is the length of the path [4,2,1,3] or [5,2,1,3].

: int diameterOfBinaryTree(TreeNode\* root) {

int diameter=0;

dfs(root,diameter);

return diameter;

}

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

{

if(!root)

return 0;

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

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

diameter=max(diameter,l+r);

return 1+max(l,r);

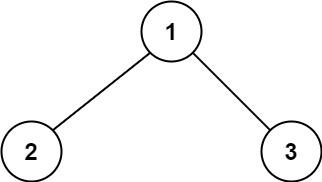
}

81) A **path** in a binary tree is a sequence of nodes where each pair of adjacent nodes in the sequence has an edge connecting them. A node can only appear in the sequence **at most once**. Note that the path does not need to pass through the root.

The **path sum** of a path is the sum of the node's values in the path.

Given the root of a binary tree, return *the maximum****path sum****of any path*.

**Example 1:**



**Input:** root = [1,2,3]

**Output:** 6

**Explanation:** The optimal path is 2 -> 1 -> 3 with a path sum of 2 + 1 + 3 = 6.

: int maxPathSum(TreeNode\* root) {

int maxsum=INT\_MIN;

dfs(root,maxsum);

return maxsum;

}

int dfs(TreeNode\*root,int&maxsum)

{

if(!root)

return 0;

int l=max(0,dfs(root->left,maxsum));

int r=max(0,dfs(root->right,maxsum));

maxsum=max(maxsum,l+r+root->val);

return root->val+max(l,r);

}

ANOTHER VARIATION OF IT

Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

A **leaf** is a node with no children.

**Example 1:**



**Input:** root = [5,4,8,11,null,13,4,7,2,null,null,null,1], targetSum = 22

**Output:** true

**Explanation:** The root-to-leaf path with the target sum is shown.

: bool hasPathSum(TreeNode\* root, int targetSum) {

if(!root)

return 0;

if(!root->left&&!root->right&&targetSum-root->val==0)

return 1;

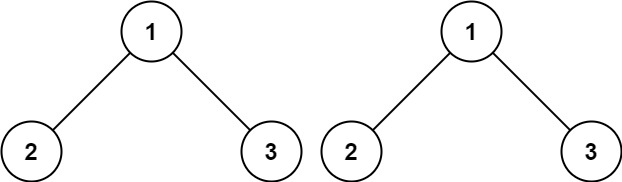
return hasPathSum(root->left,targetSum-root->val)||hasPathSum(root->right,targetSum-root->val);

}

82) Given the roots of two binary trees p and q, write a function to check if they are the same or not.

Two binary trees are considered the same if they are structurally identical, and the nodes have the same value.

**Example 1:**



**Input:** p = [1,2,3], q = [1,2,3]

**Output:** true

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

if(!p&&!q)

return 1;

if(!p||!q)

return 0;

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

}

83) Given the root of a binary tree, return *the zigzag level order traversal of its nodes' values*. (i.e., from left to right, then right to left for the next level and alternate between).

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[3],[20,9],[15,7]]

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

vector<vector<int>>v;

if(!root)

return v;

queue<TreeNode\*>q;

q.push(root);

bool l\_r=1;

while(!q.empty())

{

int size=q.size();

vector<int>v1(size);

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

{

TreeNode\*temp=q.front();

q.pop();

int index=l\_r?i:size-i-1;

v1[index]=temp->val;

if(temp->left)

q.push(temp->left);

if(temp->right)

q.push(temp->right);

}

l\_r=!l\_r;

v.push\_back(v1);

}

return v;

}

84) Given a Binary Tree, find its Boundary Traversal. The traversal should be in the following order:

1. **Left boundary nodes:** defined as the path from the root to the left-most node ie- the leaf node you could reach when you always travel preferring the left subtree over the right subtree.
2. **Leaf nodes:**All the leaf nodes except for the ones that are part of left or right boundary.
3. **Reverse right boundary nodes:** defined as the path from the right-most node to the root. The right-most node is the leaf node you could reach when you always travel preferring the right subtree over the left subtree. Exclude the root from this as it was already included in the traversal of left boundary nodes.

**Note:** If the root doesn't have a left subtree or right subtree, then the root itself is the left or right boundary.   
  
**Example 1:**

**Input:**

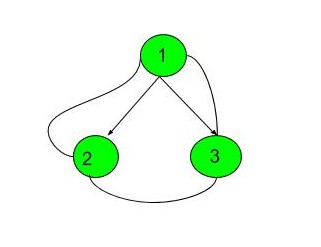
1

  / \

  2 3

**Output:** 1 2 3

**Explanation:**

****

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ISLEAF FUNCTION IS INCOMPLETE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: struct Node

{

int data;

Node\* left, \* right;

}; \*/

class Solution {

public:

vector <int> printBoundary(Node \*root)

{

//Your code here

vector<int>v;

if(!root)

return v;

if(!isLeaf(root))

v.push\_back(root->data);

left(root,v);

leaf(root,v);

right(root,v);

return v;

}

void left(Node\*root,vector<int>&v)

{

Node\*temp=root->left;

while(temp)

{

if(!isLeaf(temp))

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

if(temp->left)

temp=temp->left;

else

temp=temp->right;

}

}

void right(Node\*root,vector<int>&v)

{

Node\*temp=root->right;

vector<int>temp1;

{

if(!isLeaf(temp))

temp1.push\_back(temp->data);

if(temp->right)

temp=temp->right;

else

temp=temp->left;

}

for(int i=temp1.size()-1;i>=0;i--)

{

v.push\_back(temp1[i]);

}

}

void leaf(Node\*root,vector<int>&v)

{

if(isLeaf(root))

{

v.push\_back(root->data);

return;

}

if(root->left)

leaf(root->left,v);

if(root->right)

leaf(root->right,v);

}

void isLeaf(Node\*root)

{

if(!root)

return 0;

if(root)

}

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

**Example 1:**



**Input:** root = [1,2,2,3,4,4,3]

**Output:** true

: bool isSymmetric(TreeNode\* root) {

if(!root)

return 1;

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

}

bool dfs(TreeNode\*r1,TreeNode\*r2)

{

if(!r1&&!r2)

return 1;

if(!r1||!r2)

return 0;

return (r1->val==r2->val&&dfs(r1->left,r2->right)&&dfs(r1->right,r2->left));

}

86) Given a binary tree, find the lowest common ancestor (LCA) of two given nodes in the tree.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**



**Input:** root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1

**Output:** 3

**Explanation:** The LCA of nodes 5 and 1 is 3.

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

if(!root)

return root;

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

return root;

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

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

if(l&&r)

return root;

else

return l?l:r;

}

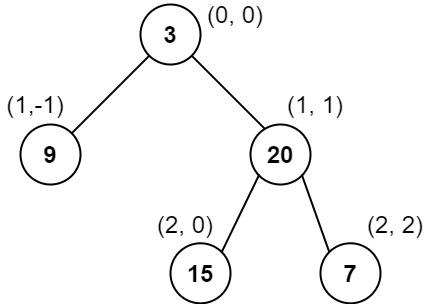
87) Given the root of a binary tree, calculate the **vertical order traversal** of the binary tree.

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*.

**Example 1:**



**Input:** root = [3,9,20,null,null,15,7]

**Output:** [[9],[3,15],[20],[7]]

**Explanation:**

Column -1: Only node 9 is in this column.

Column 0: Nodes 3 and 15 are in this column in that order from top to bottom.

Column 1: Only node 20 is in this column.

Column 2: Only node 7 is in this column.

USING BFS

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

map<int,map<int,multiset<int>>>m;

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

q.push({root,{0,0}});

while(!q.empty())

{

auto temp=q.front();

q.pop();

TreeNode\*temp1=temp.first;

int x=temp.second.first;

int y=temp.second.second;

m[x][y].insert(temp1->val);

if(temp1->left)

q.push({temp1->left,{x-1,y+1}});

if(temp1->right)

q.push({temp1->right,{x+1,y+1}});

}

vector<vector<int>>v;

for(auto p:m)

{

vector<int>v1;

for(auto temp2:p.second)

{

v1.insert(v1.end(),temp2.second.begin(),temp2.second.end());

}

v.push\_back(v1);

}

return v;

}

USING DFS

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

map<int,map<int,multiset<int>>>mp;

traverse(root,0,0,mp);

vector<vector<int>>ans;

for(auto it:mp)

{

vector<int>v1;

for(auto it1:it.second)

{

v1.insert(v1.end(),it1.second.begin(),it1.second.end());

}

ans.push\_back(v1);

}

return ans;

}

void traverse(TreeNode\*root,int x,int y,map<int,map<int,multiset<int>>>&mp)

{

if(root)

{

mp[x][y].insert(root->val);

traverse(root->left,x-1,y+1,mp);

traverse(root->right,x+1,y+1,mp);

}

}

88) You're given strings jewels representing the types of stones that are jewels, and stones representing the stones you have. Each character in stones is a type of stone you have. You want to know how many of the stones you have are also jewels.

Letters are case sensitive, so "a" is considered a different type of stone from "A".

**Example 1:**

**Input:** jewels = "aA", stones = "aAAbbbb"

**Output:** 3

: int numJewelsInStones(string jewels, string stones) {

int cnt=0;

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

{

cnt+=count(jewels.begin(),jewels.end(),stones[i]);

}

return cnt;

}

89) For a string sequence, a string word is **k-repeating** if word concatenated k times is a substring of sequence. The word's **maximum k-repeating value** is the highest value k where word is k-repeating in sequence. If word is not a substring of sequence, word's maximum k-repeating value is 0.

Given strings sequence and word, return *the****maximum k-repeating value****of word in sequence*.

**Example 1:**

**Input:** sequence = "ababc", word = "ab"

**Output:** 2

**Explanation:** "abab" is a substring in "ababc".

: **What is string::npos:**

* It is a [constant](https://www.geeksforgeeks.org/constants-in-c-cpp/) [static member](https://www.geeksforgeeks.org/static-data-members-c/) value with the highest possible value for an element of type [**size\_t**](https://www.geeksforgeeks.org/size_t-data-type-c-language/).
* It actually means until the end of the [string](https://www.geeksforgeeks.org/string-data-structure/).
* It is used as the value for a length parameter in the [string](https://www.geeksforgeeks.org/string-data-structure/)’s member functions.
* As a return value, it is usually used to indicate no matches.

: int maxRepeating(string sequence, string word) {

int cnt=0;

string s=word;

while(sequence.find(s)!=string::npos)

{

cnt++;

s+=word;

}

return cnt;

}

90) Given below is a binary tree. The task is to print the top view of binary tree. Top view of a binary tree is the set of nodes visible when the tree is viewed from the top. For the given below tree

       1  
    /     \  
   2       3  
  /  \    /   \  
4    5  6   7

Top view will be: 4 2 1 3 7  
**Note:**Return nodes from **leftmost**node to **rightmost**node.

**Example 1:**

**Input:**

  1

 /    \

2      3

**Output:** 2 1 3

: vector<int> topView(Node \*root)

{

//Your code here

vector<int>v;

map<int,int>m;

queue<pair<Node\*,int>>q;

q.push({root,0});

while(!q.empty())

{

auto it=q.front();

q.pop();

Node\*temp=it.first;

int x=it.second;

if(m.find(x)==m.end())

m[x]=temp->data;

if(temp->left)

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

if(temp->right)

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

}

for(auto it:m)

{

v.push\_back(it.second);

}

return v;

}

91) Given a binary tree, print the bottom view from left to right.  
A node is included in bottom view if it can be seen when we look at the tree from bottom.

                      20  
                    /    \  
                  8       22  
                /   \        \  
              5      3       25  
                    /   \        
                  10    14

For the above tree, the bottom view is 5 10 3 14 25.  
If there are **multiple**bottom-most nodes for a horizontal distance from root, then print the later one in level traversal. For example, in the below diagram, 3 and 4 are both the bottommost nodes at horizontal distance 0, we need to print 4.

                      20  
                    /    \  
                  8       22  
                /   \     /   \  
              5      3 4     25  
                     /    \        
                 10       14

For the above tree the output should be 5 10 4 14 25.

**Example 1:**

**Input:**

1

  / \

  3 2

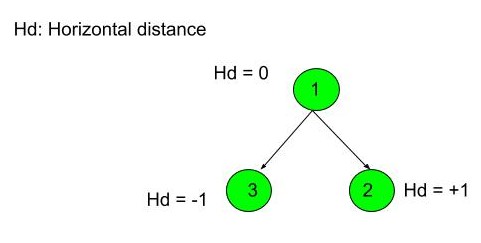
**Output:** 3 1 2

**Explanation:**

First case represents a tree with 3 nodes

and 2 edges where root is 1, left child of

1 is 3 and right child of 1 is 2.



Thus nodes of the binary tree will be

printed as such 3 1 2.

: vector <int> bottomView(Node \*root) {

// Your Code Here

vector<int>v;

map<int,int>m;

if(!root)

return v;

queue<pair<Node\*,int>>q;

q.push({root,0});

while(!q.empty())

{

auto it=q.front();

q.pop();

Node\*temp=it.first;

int x=it.second;

m[x]=temp->data;

if(temp->left)

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

if(temp->right)

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

}

for(auto it:m)

{

v.push\_back(it.second);

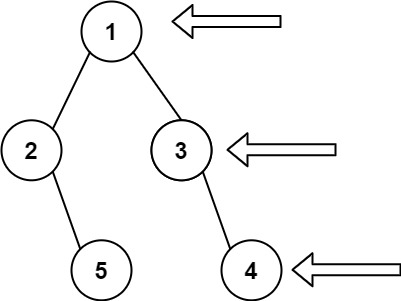
}

return v;

}

92) Given the root of a binary tree, imagine yourself standing on the **right side** of it, return *the values of the nodes you can see ordered from top to bottom*.

**Example 1:**



**Input:** root = [1,2,3,null,5,null,4]

**Output:** [1,3,4]

: vector<int> rightSideView(TreeNode\* root) {

vector<int>v;

rec(root,0,v);

return v;

}

void rec(TreeNode\*root,int level,vector<int>&v)

{

if(!root)

return ;

if(level==v.size())

v.push\_back(root->val);

rec(root->right,level+1,v);

rec(root->left,level+1,v);

}

93) Given a Binary Tree, print Left view of it. Left view of a Binary Tree is set of nodes visible when tree is visited from Left side. The task is to complete the function **leftView()**, which accepts root of the tree as argument.

Left view of following tree is 1 2 4 8.

          1  
       /     \  
     2        3  
   /     \    /    \  
  4     5   6    7  
   \  
     8

**Example 1:**

**Input:**

  1

 /  \

3    2

**Output:** 1 3

**Example 2:**

**Input:**

**Output:** 10 20 40

: void rec(Node\*root,int level,vector<int>&v)

{

if(!root)

return;

if(level==v.size())

v.push\_back(root->data);

rec(root->left,level+1,v);

rec(root->right,level+1,v);

}

vector<int> leftView(Node \*root)

{

// Your code here

vector<int>v;

rec(root,0,v);

return v;

}

94) Given a Binary Tree **A** containing **N** nodes.

You need to find the path from **Root** to a given node **B**.

**NOTE:**

* No two nodes in the tree have same data values.
* You can assume that **B** is present in the tree **A** and a path always exists.

**Problem Constraints**

1 <= N <= 105

1 <= Data Values of Each Node <= N

1 <= B <= N

**Input Format**

First Argument represents pointer to the root of binary tree **A**.

Second Argument is an integer **B** denoting the node number.

**Output Format**

Return an one-dimensional array denoting the path from **Root** to the node **B** in order.

**Example Input**

Input 1:

A =

1

/ \

2 3

/ \ / \

4 5 6 7

B = 5

:  bool rec(TreeNode\*a,int b,vector<int>&v)

 {

     if(!a)

     return 0;

     v.push\_back(a->val);

     if(a->val==b)

     return 1;

     if(rec(a->left,b,v)||rec(a->right,b,v))

     return 1;

     v.pop\_back();

     return 0;

 }

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

    vector<int>v;

    if(!A)

    return v;

    rec(A,B,v);

    return v;

}

95)CHILDREN SUM PROPERTY IN BINARY TREE

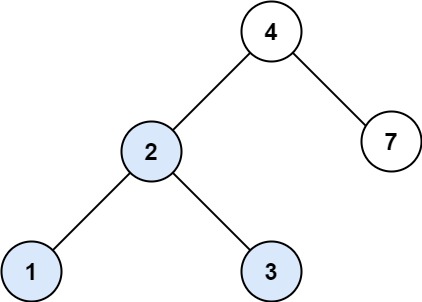
:

|  |
| --- |
| 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) root->data = child; |
|  | else { |
|  | 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 or root->right) root->data = tot; |
|  | } |
|  | void changeTree(BinaryTreeNode < int > \* root) { |
|  | reorder(root); |
|  | } |

96) You are given the root of a binary search tree (BST) and an integer val.

Find the node in the BST that the node's value equals val and return the subtree rooted with that node. If such a node does not exist, return null.

**Example 1:**



**Input:** root = [4,2,7,1,3], val = 2

**Output:** [2,1,3]

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

while(root&&root->val!=val)

{

root=(root->val>val)?root->left:root->right;

}

return root;

}

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

A **valid BST** is defined as follows:

* The left subtree of a node contains only nodes with keys **less than** the node's key.
* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.

**Example 1:**



**Input:** root = [2,1,3]

**Output:** true

: bool isValidBST(TreeNode\* root) {

return isVlidBST(root,LONG\_MIN,LONG\_MAX);

}

bool isVlidBST(TreeNode\*root,long mini,long maxi)

{

if(!root)

return 1;

if(mini>=root->val||maxi<=root->val)

return 0;

return isVlidBST(root->left,mini,root->val)&&isVlidBST(root->right,root->val,maxi);

}

98) Given a binary search tree (BST), find the lowest common ancestor (LCA) of two given nodes in the BST.

According to the [definition of LCA on Wikipedia](https://en.wikipedia.org/wiki/Lowest_common_ancestor): “The lowest common ancestor is defined between two nodes p and q as the lowest node in T that has both p and q as descendants (where we allow **a node to be a descendant of itself**).”

**Example 1:**



**Input:** root = [6,2,8,0,4,7,9,null,null,3,5], p = 2, q = 8

**Output:** 6

**Explanation:** The LCA of nodes 2 and 8 is 6.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*MYVERSION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

if(!root)

return root;

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

return root;

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

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

if(l&&r)

return root;

else

return l?l:r;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*NEW VESRION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

if(!root)

return root;

if(root->val>p->val&&root->val>q->val)

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

if(root->val<p->val&&root->val<q->val)

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

return root;

}

99) Given a positive integer **N**, find the last digit of the **Nth** term from the Fibonacci series.

\*\*\*\*\*\*\*AFTER EVERY 60 NUMBERS,LAST DIGITS GET

REPEATED\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*(IMP)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Example 1:**

**Input:**

N = 5

**Output:**

5

**Explanation:**

5th Fibonacci number is 5

: int fib(int N){

//code here

long long f[60]={0};

fibb(f,60);

return f[N%60];

}

long long fibb(long long f[],long long n)

{

f[0]=0;

f[1]=1;

for(long long i=2;i<=n;i++)

{

f[i]=(f[i-1]+f[i-2])%10;

}

return f[n];

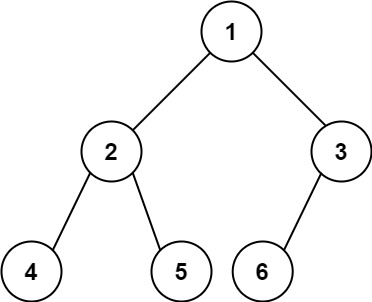
}

100) Given the root of a **complete** binary tree, return the number of the nodes in the tree.

According to [**Wikipedia**](http://en.wikipedia.org/wiki/Binary_tree#Types_of_binary_trees), every level, except possibly the last, is completely filled in a complete binary tree, and all nodes in the last level are as far left as possible. It can have between 1 and 2h nodes inclusive at the last level h.

Design an algorithm that runs in less than O(n) time complexity.

**Example 1:**



**Input:** root = [1,2,3,4,5,6]

**Output:** 6

: int countNodes(TreeNode\* root) {

if(!root)

return 0;

int l=lheight(root);

int r=rheight(root);

if(l==r)

return (1<<l)-1;

return 1+countNodes(root->left)+countNodes(root->right);

}

int lheight(TreeNode\*root)

{

int cnt=0;

while(root)

{

cnt++;

root=root->left;

}

return cnt;

}

int rheight(TreeNode\*root)

{

int cnt=0;

while(root)

{

cnt++;

root=root->right;

}

return cnt;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*WITH PREORDER AND POSTORDER,WE CANNOT CREATE UNIQUE BINARY TREE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

101) Given two integer arrays preorder and inorder where preorder is the preorder traversal of a binary tree and inorder is the inorder traversal of the same tree, construct and return *the binary tree*.

**Example 1:**



**Input:** preorder = [3,9,20,15,7], inorder = [9,3,15,20,7]

**Output:** [3,9,20,null,null,15,7]

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

map<int,int>m;

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

{

m[inorder[i]]=i;

}

TreeNode\*root=rec(preorder,0,preorder.size()-1,inorder,0,inorder.size()-1,m);

return root;

}

TreeNode\* rec(vector<int>&pre,int pres,int pree,vector<int>&in,int ins,int ine,map<int,int>&m)

{

if(ins>ine||pres>pree)

return NULL;

TreeNode\*root1=new TreeNode(pre[pres]);

int root\_pos=m[root1->val];

int lval=root\_pos-ins;

root1->left=rec(pre,pres+1,pres+lval,in,ins,root\_pos-1,m);

root1->right=rec(pre,pres+lval+1,pree,in,root\_pos+1,ine,m);

return root1;

}

102) Given two integer arrays inorder and postorder where inorder is the inorder traversal of a binary tree and postorder is the postorder traversal of the same tree, construct and return *the binary tree*.

**Example 1:**



**Input:** inorder = [9,3,15,20,7], postorder = [9,15,7,20,3]

**Output:** [3,9,20,null,null,15,7]

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

map<int,int>m;

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

{

m[inorder[i]]=i;

}

TreeNode\* root=rec(inorder,0,inorder.size()-1,postorder,0,postorder.size()-1,m);

return root;

}

TreeNode\* rec(vector<int>&in,int ins,int ine,vector<int>&post,int posts,int poste,map<int,int>&m)

{

if(posts>poste||ins>ine)

return NULL;

TreeNode\* root1=new TreeNode(post[poste]);

int root\_pos=m[post[poste]];

int lval=root\_pos-ins;

root1->left=rec(in,ins,root\_pos-1,post,posts,posts+lval-1,m);

root1->right=rec(in,root\_pos+1,ine,post,posts+lval,poste-1,m);

return root1;

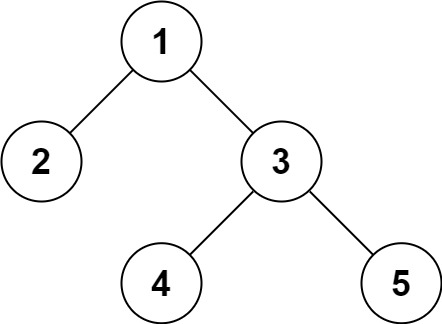
}

103) Serialization is the process of converting a data structure or object into a sequence of bits so that it can be stored in a file or memory buffer, or transmitted across a network connection link to be reconstructed later in the same or another computer environment.

Design an algorithm to serialize and deserialize a binary tree. There is no restriction on how your serialization/deserialization algorithm should work. You just need to ensure that a binary tree can be serialized to a string and this string can be deserialized to the original tree structure.

**Clarification:** The input/output format is the same as [how LeetCode serializes a binary tree](https://leetcode.com/faq/#binary-tree). You do not necessarily need to follow this format, so please be creative and come up with different approaches yourself.

**Example 1:**



**Input:** root = [1,2,3,null,null,4,5]

**Output:** [1,2,3,null,null,4,5]

: // Encodes a tree to a single string.

string serialize(TreeNode\* root) {

if(!root)

return "#";

return to\_string(root->val)+","+serialize(root->left)+","+serialize(root->right);

}

// Decodes your encoded data to tree.

TreeNode\* deserialize(string data) {

return mydes(data);

}

TreeNode\*mydes(string& s)

{

if(s[0]=='#')

{

if(s.size()>1)

s=s.substr(2);

return NULL;

}

else

{

TreeNode\* root1=new TreeNode(helper(s));

root1->left=mydes(s);

root1->right=mydes(s);

return root1;

}

}

int helper(string& s)

{

int pos=s.find(',');

int val=stoi(s.substr(0,pos));

s=s.substr(pos+1);

return val;

}

104)MORRIS INORDER TRAVERSAL

:

|  |
| --- |
| vector<int> inorderTraversal(TreeNode\* root) { |
|  | vector<int> inorder; |
|  |  |
|  | TreeNode\* cur = root; |
|  | while(cur != NULL) { |
|  | if(cur->left == NULL) { |
|  | inorder.push\_back(cur->val); |
|  | cur = cur->right; |
|  | } |
|  | else { |
|  | TreeNode\* prev = cur->left; |
|  | while(prev->right != NULL && prev->right != cur) { |
|  | prev = prev->right; |
|  | } |
|  |  |
|  | if(prev->right == NULL) { |
|  | prev->right = cur; |
|  | cur = cur->left; |
|  | } |
|  | else { |
|  | prev->right = NULL; |
|  | inorder.push\_back(cur->val); |
|  | cur = cur->right; |
|  | } |
|  | } |
|  | } |
|  | return inorder; |
|  | } |

105)MORRIS PREORDER TRAVERSAL

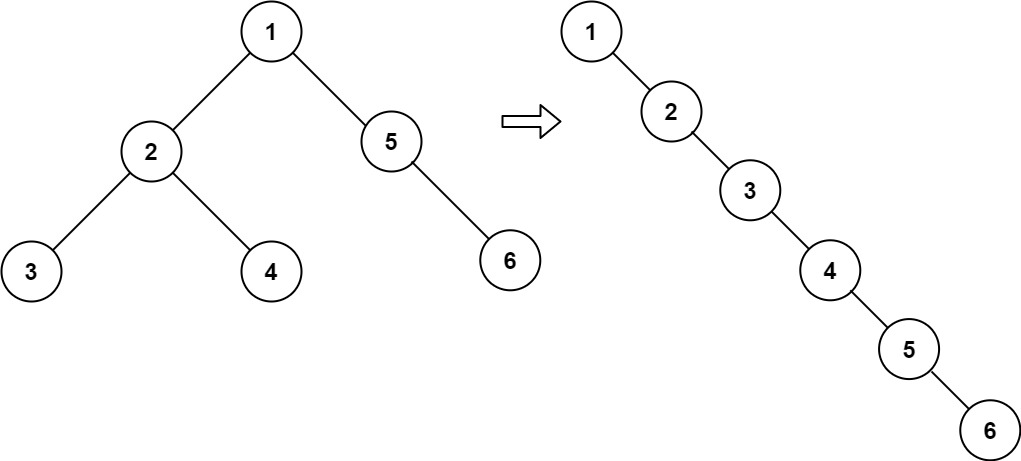
:

|  |
| --- |
| vector<int> preorderTraversal(TreeNode\* root) { |
|  | vector<int> preorder; |
|  |  |
|  | TreeNode\* cur = root; |
|  | while(cur != NULL) { |
|  | if(cur->left == NULL) { |
|  | preorder.push\_back(cur->val); |
|  | cur = cur->right; |
|  | } |
|  | else { |
|  | TreeNode\* prev = cur->left; |
|  | while(prev->right != NULL && prev->right != cur) { |
|  | prev = prev->right; |
|  | } |
|  |  |
|  | if(prev->right == NULL) { |
|  | prev->right = cur; |
|  | preorder.push\_back(cur->val); |
|  | cur = cur->left; |
|  | } |
|  | else { |
|  | prev->right = NULL; |
|  | cur = cur->right; |
|  | } |
|  | } |
|  | } |
|  | return preorder; |
|  | } |

106) Given the root of a binary tree, flatten the tree into a "linked list":

* The "linked list" should use the same TreeNode class where the right child pointer points to the next node in the list and the left child pointer is always null.
* The "linked list" should be in the same order as a [**pre-order traversal**](https://en.wikipedia.org/wiki/Tree_traversal#Pre-order,_NLR) of the binary tree.

**Example 1:**



**Input:** root = [1,2,5,3,4,null,6]

**Output:** [1,null,2,null,3,null,4,null,5,null,6]

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*RECURSIVE VERSION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: class Solution {

TreeNode\*prev=NULL;

public:

void flatten(TreeNode\* root) {

if(!root)

return;

flatten(root->right);

flatten(root->left);

root->right=prev;

root->left=NULL;

prev=root;

}

};

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*ITERATIVE VERSION WITH EXTRA SPACE\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

void flatten(TreeNode\* root) {

if(!root)

return;

stack<TreeNode\*>st;

st.push(root);

while(!st.empty())

{

TreeNode\*curr=st.top();

st.pop();

if(curr->right!=NULL)

st.push(curr->right);

if(curr->left!=NULL)

st.push(curr->left);

if(!st.empty())

{

curr->right=st.top();

}

curr->left=NULL;

}

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*USING LINKED LIST\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: void flatten(TreeNode\* root) {

TreeNode\*curr=root;

while(curr)

{

if(curr->left)

{

TreeNode\*prev=curr->left;

while(prev->right)

prev=prev->right;

prev->right=curr->right;

curr->right=curr->left;

}

curr->left=NULL;

curr=curr->right;

}

}

107) find ceil value in BST

:

|  |
| --- |
| int findCeil(BinaryTreeNode<int> \*root, int key){ |
|  |  |
|  | int ceil = -1; |
|  | while (root) { |
|  |  |
|  | if (root->data == key) { |
|  | ceil = root->data; |
|  | return ceil; |
|  | } |
|  |  |
|  | if (key > root->data) { |
|  | root = root->right; |
|  | } |
|  | else { |
|  | ceil = root->data; |
|  | root = root->left; |
|  | } |
|  | } |
|  | return ceil; |
|  | } |

108)find floor value in BST

:

|  |
| --- |
| int floorInBST(TreeNode<int> \* root, int key) |
|  | { |
|  | int floor = -1; |
|  | while (root) { |
|  |  |
|  | if (root->val == key) { |
|  | floor = root->val; |
|  | return floor; |
|  | } |
|  |  |
|  | if (key > root->val) { |
|  | floor = root->val; |
|  | root = root->right; |
|  | } |
|  | else { |
|  | root = root->left; |
|  | } |
|  | } |
|  | return floor; |
|  | } |

109) 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.

: Node\* connect(Node\* root) {

if(!root)

return root;

Node\*newroot=root;

while(root->left)

{

Node\*temp=root;

while(temp)

{

temp->left->next=temp->right;

if(temp->next)

temp->right->next=temp->next->left;

temp=temp->next;

}

root=root->left;

}

return newroot;

}

110) Given a sorted array. Convert it into a Height balanced Binary Search Tree (BST). Find the preorder traversal of height balanced BST. If there exist many such balanced BST consider the tree whose preorder is lexicographically smallest.  
Height balanced BST means a binary tree in which the depth of the left subtree and the right subtree of every node never differ by more than 1.

**Example 1:**

**Input:** nums = {1, 2, 3, 4}

**Output:** {2, 1, 3, 4}

**Explanation:**

The preorder traversal of the following

BST formed is {2, 1, 3, 4}:

  2

  / \

1 3

  \

  4

**Example 2:**

**Input:** nums = {1,2,3,4,5,6,7}

**Ouput:** {4,2,1,3,6,5,7}

**Explanation:**

The preorder traversal of the following

BST formed is {4,2,1,3,6,5,7} :

4

/ \

2 6

/ \ / \

1 3 5 7

: void bst(vector<int>&nums,vector<int>&v,int s,int e)

{

if(s<=e)

{

int mid=(s+e)/2;

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

bst(nums,v,s,mid-1);

bst(nums,v,mid+1,e);

}

}

vector<int> sortedArrayToBST(vector<int>& nums) {

// Code here

vector<int>v;

int start=0,end=nums.size()-1;

bst(nums,v,start,end);

return v;

}

111) There is BST given with root node with key part as integer only. You need to find the inorder successor and predecessor of a given key. In case, if the either of predecessor or successor is not found print -1.

**Input:**  
The first line of input contains an integer T denoting the number of test cases. Then T test cases follow. Each test case contains n denoting the number of edges of the BST. The next line contains the edges of the BST. The last line contains the key.

**Output:**  
Print the predecessor followed by successor for the given key. If the predecessor or successor is not found print -1.

**Constraints:**  
1<=T<=100  
1<=n<=100  
1<=data of node<=100  
1<=key<=100

**Example:  
Input:**  
2  
6  
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R  
65  
6  
50 30 L 30 20 L 30 40 R 50 70 R 70 60 L 70 80 R  
100

**Output:**  
60 70  
80 -1

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

{

// Your code goes here

if(!root)

return;

if(root->key==key)

{

if(root->left)

{

Node\*temp=root->left;

while(temp->right)

temp=temp->right;

pre=temp;

}

if(root->right)

{

Node\*temp=root->right;

while(temp->left)

temp=temp->left;

suc=temp;

}

}

else if(root->key>key)

{

suc=root;

findPreSuc(root->left,pre,suc,key);

}

else

{

pre=root;

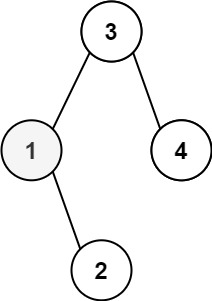
findPreSuc(root->right,pre,suc,key);

}

}

112) Given the root of a binary search tree, and an integer k, return the kth smallest value (***1-indexed***) of all the values of the nodes in the tree.

**Example 1:**



**Input:** root = [3,1,4,null,2], k = 1

**Output:** 1

: int kthSmallest(TreeNode\* root, int k) {

return find(root,k);

}

int find(TreeNode\*root,int&k)

{

if(root)

{

int x=find(root->left,k);

return !k?x:!--k?root->val:find(root->right,k);

}

return 0;

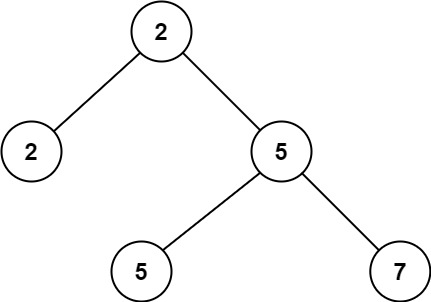
}

113) Given a non-empty special binary tree consisting of nodes with the non-negative value, where each node in this tree has exactly two or zero sub-node. If the node has two sub-nodes, then this node's value is the smaller value among its two sub-nodes. More formally, the property root.val = min(root.left.val, root.right.val) always holds.

Given such a binary tree, you need to output the **second minimum** value in the set made of all the nodes' value in the whole tree.

If no such second minimum value exists, output -1 instead.

**Example 1:**



**Input:** root = [2,2,5,null,null,5,7]

**Output:** 5

**Explanation:** The smallest value is 2, the second smallest value is 5.

: set<int>s;

int findSecondMinimumValue(TreeNode\* root) {

preorder(root);

return s.size()<=1?-1:\*(++s.begin());

}

void preorder(TreeNode\*root)

{

if(!root)

return;

s.insert(root->val);

preorder(root->left);

preorder(root->right);

}

114) Given a Binary search tree. Your task is to complete the function which will return the Kth largest element without doing any modification in Binary Search Tree.

**Example 1:**

**Input:**

**4**

  / \

2 9

k = 2

**Output: 4**

: int kthLargest(Node \*root, int K)

{

//Your code here

return find(root,K);

}

int find(Node\*root,int&k)

{

if(root)

{

int x=find(root->right,k);

return !k?x:!--k?root->data:find(root->left,k);

}

return 0;

}

115) Given a Binary Search Tree and a target sum. Check whether there's a pair of Nodes in the BST with value summing up to the target sum.

**Example 1:**

**Input:**

2

/ \

  1 3

sum = 5

**Output:** 1

**Explanation:**

Nodes with value 2 and 3 sum up to 5.

: int isPairPresent(struct Node \*root, int target)

{

//add code here.

unordered\_set<int>s;

return inorder(root,s,target);

}

int inorder(Node\* root,unordered\_set<int> &s,int target)

{

if(root==NULL)

return 0;

if(inorder(root->left,s,target)==1)

return 1;

if(s.find(target-root->data)!=s.end())

return 1;

else

s.insert(root->data);

return inorder(root->right,s,target);

}

116) Given an array of n numbers, give an algorithm for checking whether there are any duplicate elements in the array or no?

Let us assume that the array elements are positive numbers and all the elements are in the range 0 to n – 1. For each element A[i], go to the array element whose index is A[i]. That means select A[A[i]] and mark - A[A[i]] (negate the value at A[A[i]]). Continue this process until we encounter the element whose value is already negated. If one such element exists then we say duplicate elements exist in the given array. As an example, consider the array, A = {3,2,1,2,2,3}

Time Complexity: O(n). Since only one scan is required.

Space Complexity: O(1).

Notes: • This solution does not work if the given array is read only. • This solution will work only if all the array elements are positive. • If the elements range is not in 0 to n – 1 then it may give exception

Void checkduplicate(int arr[],int n)

{

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

{

If(arr[abs(arr[i])]>0)

{

Cout<<duplicates present;

return

}

Else

Arr[arr[i]]=-arr[arr[i]];

}

117) Given a binary tree. Find the size of its largest subtree that is a Binary Search Tree.  
**Note:**Here Size is equal to the number of nodes in the subtree.

**Example 1:**

**Input:**

  1

  / \

  4 4

  / \

  6 8

**Output:** 1

**Explanation:** There's no sub-tree with size

greater than 1 which forms a BST. All the

leaf Nodes are the BSTs with size equal

to 1.

: vector<int> solve(Node \*root){

if(!root) return {1,0,INT\_MAX,INT\_MIN};

// vector=>{bstornot,size,minval,maxval

if(!root->left and !root->right) return {1,1,root->data,root->data};

vector<int> l=solve(root->left);

vector<int> r=solve(root->right);

if(l[0] and r[0]){

if(root->data>l[3] and root->data<r[2]){

int x=l[2]; int y=r[3];

if(x==INT\_MAX) x=root->data;

if(y==INT\_MIN) y=root->data;

return {1,l[1]+r[1]+1,x,y};

}

}

return {0,max(l[1],r[1]),0,0};

}

int largestBst(Node \*root)

{

//Your code here

vector<int> ans=solve(root);

return ans[1];

}

118) Implement the BSTIterator class that represents an iterator over the [**in-order traversal**](https://en.wikipedia.org/wiki/Tree_traversal#In-order_(LNR)) 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.

**Example 1:**



**Input**

["BSTIterator", "next", "next", "hasNext", "next", "hasNext", "next", "hasNext", "next", "hasNext"]

[[[7, 3, 15, null, null, 9, 20]], [], [], [], [], [], [], [], [], []]

**Output**

[null, 3, 7, true, 9, true, 15, true, 20, false]

**Explanation**

BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]);

bSTIterator.next(); // return 3

bSTIterator.next(); // return 7

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 9

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 15

bSTIterator.hasNext(); // return True

bSTIterator.next(); // return 20

bSTIterator.hasNext(); // return False

: stack<TreeNode\*>s;

BSTIterator(TreeNode\* root) {

inorder(root);

}

void inorder(TreeNode\*root)

{

while(root)

{

s.push(root);

root=root->left;

}

}

int next() {

TreeNode\*top=s.top();

s.pop();

if(top->right)

inorder(top->right);

return top->val;

}

bool hasNext() {

return !s.empty();

}

119) Given an integer array nums and an integer k, return *the* kth *largest element in the array*.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

**Example 1:**

**Input:** nums = [3,2,1,5,6,4], k = 2

**Output:** 5

: int findKthLargest(vector<int>& nums, int k) {

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

return nums[nums.size()-k];

}

120) Given an integer array nums, return *the****third distinct maximum****number in this array. If the third maximum does not exist, return the****maximum****number*.

**Example 1:**

**Input:** nums = [3,2,1]

**Output:** 1

**Explanation:**

The first distinct maximum is 3.

The second distinct maximum is 2.

The third distinct maximum is 1.

: int thirdMax(vector<int>& nums) {

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

auto temp=unique(nums.begin(),nums.end());

nums.erase(temp,nums.end());

if(nums.size()<3)

return nums[nums.size()-1];

else

return nums[nums.size()-3];

}

121) Given an integer n, return *true if it is a power of two. Otherwise, return false*.

An integer n is a power of two, if there exists an integer x such that n == 2x.

**Example 1:**

**Input:** n = 1

**Output:** true

**Explanation:** 20 = 1

: bool isPowerOfTwo(int n) {

if(n<=0)

return 0;

return !(n&(n-1));

}

Another code

Bool is PoweOf Two(int n)

{

First n for checking for n=0;

Return n&&!(n&(n-1));

}

122) Given an integer n, return *true if it is a power of three. Otherwise, return false*.

An integer n is a power of three, if there exists an integer x such that n == 3x.

**Example 1:**

**Input:** n = 27

**Output:** true

bool isPowerOfThree(int n) {

return (n>0&&(int)pow(3,19)%n==0);

}

123) Given an integer n, return *true if it is a power of four. Otherwise, return false*.

An integer n is a power of four, if there exists an integer x such that n == 4x.

**Example 1:**

**Input:** n = 16

**Output:** true

: bool isPowerOfFour(int n) {

return (n>0&&(n&(n-1))==0&&((n-1)%3)==0);

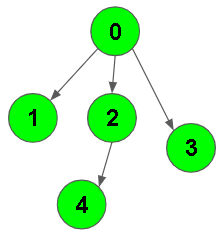
}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*GRAPH\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

124) Given a directed graph. The task is to do Breadth First Traversal of this graph starting from 0.  
**Note:**One can move from node u to node v only if there's an edge from u to v and find the BFS traversal of the graph starting from the 0th vertex, from left to right according to the graph. Also, you should only take nodes directly or indirectly connected from Node 0 in consideration.

**Example 1:**

**Input:**



**Output:** 0 1 2 3 4

**Explanation**:

0 is connected to 1 , 2 , 3.

2 is connected to 4.

so starting from 0, it will go to 1 then 2

then 3.After this 2 to 4, thus bfs will be

0 1 2 3 4.

: vector<int> bfsOfGraph(int V, vector<int> adj[]) {

// Code here

vector<int>bfs;

vector<int>vis(V,0);

queue<int>q;

q.push(0);

vis[0]=1;

while(!q.empty())

{

int temp=q.front();

q.pop();

bfs.push\_back(temp);

for(auto it:adj[temp])

{

if(!vis[it])

{

q.push(it);

vis[it]=1;

}

}

}

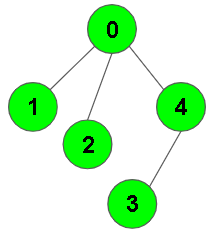
return bfs;

}

125) Given a connected undirected graph. Perform a Depth First Traversal of the graph.  
**Note:**Use recursive approach to find the DFS traversal of the graph starting from the 0th vertex from left to right according to the graph..

**Example 1:**

**Input:**



**Output:** 0 1 2 4 3

**Explanation**:

0 is connected to 1, 2, 4.

1 is connected to 0.

2 is connected to 0.

3 is connected to 4.

4 is connected to 0, 3.

so starting from 0, it will go to 1 then 2

then 4, and then from 4 to 3.

Thus dfs will be 0 1 2 4 3.

: void dfs(int node,vector<int>&vis,vector<int>adj[],vector<int>&dfs\_ans)

{

dfs\_ans.push\_back(node);

vis[node]=1;

for(auto it:adj[node])

{

if(!vis[it])

{

dfs(it,vis,adj,dfs\_ans);

}

}

}

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

// Code here

vector<int>dfs\_ans;

vector<int>vis(V,0);

dfs(0,vis,adj,dfs\_ans);

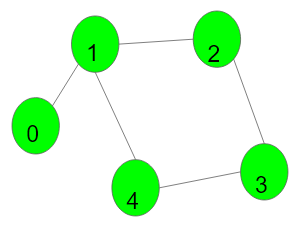
return dfs\_ans;

}

126) Given an undirected graph with V vertices and E edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**



**Output:** 1

**Explanation:** 1->2->3->4->1 is a cycle

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

: bool check(int node,vector<int>&vis,vector<int>adj[])

{

queue<pair<int,int>>q;

vis[node]=1;

q.push({node,-1});

while(!q.empty())

{

int parent=q.front().first;

int pre=q.front().second;

q.pop();

for(auto it:adj[parent])

{

if(!vis[it])

{

vis[it]=1;

q.push({it,parent});

}

else if(pre!=it)

return 1;

}

}

return 0;

}

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

// Code here

vector<int>vis(V,0);

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

{

if(!vis[i]&&check(i,vis,adj))

return 1;

}

return 0;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
|  |

:bool checkForCycle(int node, int parent, vector<int> &vis, vector<int> adj[]) {

vis[node] = 1;

for(auto it: adj[node]) {

if(!vis[it]) {

if(checkForCycle(it, node, vis, adj))

return true;

}

else if(it!=parent)

return true;

}

return false;

}

public:

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

vector<int> vis(V+1, 0);

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

if(!vis[i]) {

if(checkForCycle(i, -1, vis, adj)) return true;

}

}

return false;

}

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*REFER TO NEXT SECTION/REMAINING SECTION OF GRAPH\*\*\*\*\*\*\*\*\*\*\*\*

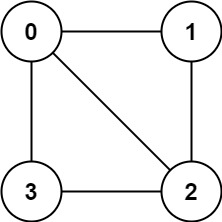
127) There is an **undirected** graph with n nodes, where each node is numbered between 0 and n - 1. You are given a 2D array graph, where graph[u] is an array of nodes that node u is adjacent to. More formally, for each v in graph[u], there is an undirected edge between node u and node v. The graph has the following properties:

* There are no self-edges (graph[u] does not contain u).
* There are no parallel edges (graph[u] does not contain duplicate values).
* If v is in graph[u], then u is in graph[v] (the graph is undirected).
* The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets A and B such that **every** edge in the graph connects a node in set A and a node in set B.

Return true*if and only if it is****bipartite***.

**Example 1:**

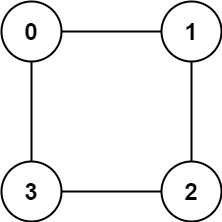


**Input:** graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

**Output:** false

**Explanation:** There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

**Example 2:**



**Input:** graph = [[1,3],[0,2],[1,3],[0,2]]

**Output:** true

**Explanation:** We can partition the nodes into two sets: {0, 2} and {1, 3}.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

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

int n=graph.size();

vector<int>vis(n,0);

queue<int>q;

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

{

if(vis[i])

continue;

vis[i]=1;

q.push(i);

while(!q.empty())

{

int temp=q.front();

for(auto it:graph[temp])

{

if(!vis[it])

{

vis[it]=-vis[temp];

q.push(it);

}

else if(vis[it]==vis[temp])

return 0;

}

q.pop();

}

}

return 1;

}

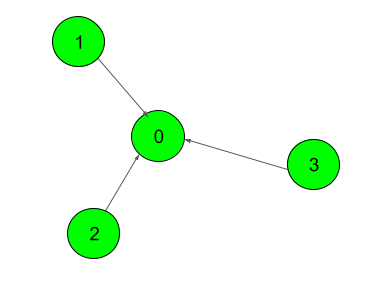
\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

|  |
| --- |
| bool bipartiteDfs(int node, vector<int> adj[], int color[]) { |
|  | for(auto it : adj[node]) { |
|  | if(color[it] == -1) { |
|  | color[it] = 1 - color[node]; |
|  | if(!bipartiteDfs(it, adj, color)) { |
|  | return false; |
|  | } |
|  | } else if(color[it] == color[node]) return false; |
|  | } |
|  | return true; |
|  | } |
|  | bool checkBipartite(vector<int> adj[], int n) { |
|  | int color[n]; |
|  | memset(color, -1, sizeof color); |
|  | for(int i = 0;i<n;i++) { |
|  | if(color[i] == -1) { |
|  | color[i] = 1; |
|  | if(!bipartiteDfs(i, adj, color)) { |
|  | return false; |
|  | } |
|  | } |
|  | } |
|  | return true; |
|  | } |

128) Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

**Example 1:**

**Input:**



**Output:**

1

**Explanation**:

The output 1 denotes that the order is

valid. So, if you have, implemented

your function correctly, then output

would be 1 for all test cases.

One possible Topological order for the

graph is 3, 2, 1, 0.

: 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);

}

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

{

// code here

stack<int>st;

vector<int>vis(V,0);

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

{

if(!vis[i])

dfs(i,vis,adj,st);

}

vector<int>topo;

while(!st.empty())

{

topo.push\_back(st.top());

st.pop();

}

return topo;

}

129) Given a grid consisting of '0's(Water) and '1's(Land). Find the number of islands.  
**Note:**An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically or diagonally i.e., in all 8 directions.

**Example 1:**

**Input:**

grid = {{0,1},{1,0},{1,1},{1,0}}

**Output:**

1

**Explanation:**

The grid is-

0 1

1 0

1 1

1 0

All lands are connected.

**Example 2:**

**Input:**

grid = {{0,1,1,1,0,0,0},{0,0,1,1,0,1,0}}

**Output:**

2

**Expanation:**

The grid is-

0 1 1 1 0 0 0

0 0 1 1 0 1 0

There are two islands one is colored in blue

and other in orange.

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

{

if(i<0||j<0||i>=n||j>=m||grid[i][j]=='2'||grid[i][j]=='0')

return;

grid[i][j]='2';

dfs(grid,i+1,j,n,m);

dfs(grid,i-1,j,n,m);

dfs(grid,i,j+1,n,m);

dfs(grid,i,j-1,n,m);

dfs(grid,i+1,j+1,n,m);

dfs(grid,i-1,j-1,n,m);

dfs(grid,i+1,j-1,n,m);

dfs(grid,i-1,j+1,n,m);

}

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

// Code here

int n=grid.size();

int m=grid[0].size();

int cnt=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);

cnt++;

}

}

}

return cnt;

}

130) Given an array of integers and a number K. Find the count of distinct elements in every window of size K in the array.

**Example 1:**

**Input:**

N = 7, K = 4

A[] = {1,2,1,3,4,2,3}

**Output:** 3 4 4 3

**Explanation:** Window 1 of size k = 4 is

1 2 1 3. Number of distinct elements in

this window are 3.

Window 2 of size k = 4 is 2 1 3 4. Number

of distinct elements in this window are 4.

Window 3 of size k = 4 is 1 3 4 2. Number

of distinct elements in this window are 4.

Window 4 of size k = 4 is 3 4 2 3. Number

of distinct elements in this window are 3.

: vector <int> countDistinct (int A[], int n, int k)

{

//code here.

unordered\_map<int,int>m;

vector<int>v;

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

{

m[A[i]]++;

}

v.push\_back(m.size());

int j=0;

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

{

m[A[j]]--;

m[A[i]]++;

if(m[A[j]]==0)

{

m.erase(A[j]);

}

j++;

v.push\_back(m.size());

}

return v;

}

131) Given two sorted arrays nums1 and nums2 of size m and n respectively, return **the median** of the two sorted arrays.

The overall run time complexity should be O(log (m+n)).

**Example 1:**

**Input:** nums1 = [1,3], nums2 = [2]

**Output:** 2.00000

**Explanation:** merged array = [1,2,3] and median is 2.

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

int n=nums1.size();

int m=nums2.size();

if(n<m) return findMedianSortedArrays(nums2,nums1);

int lo=0;

int hi=m\*2;

while(lo<=hi)

{

int mid2=(lo+hi)/2;

int mid1=n+m-mid2;

double l1=(mid1==0)?INT\_MIN:nums1[(mid1-1)/2];

double l2=(mid2==0)?INT\_MIN:nums2[(mid2-1)/2];

double r1=(mid1==n\*2)?INT\_MAX:nums1[mid1/2];

double r2=(mid2==m\*2)?INT\_MAX:nums2[mid2/2];

if(l1>r2)

lo=mid2+1;

else if(l2>r1)

hi=mid2-1;

else

return (max(l1,l2)+min(r1,r2))/2;

}

return -1;

}

C++ 11 introduced lambda expression to allow us write an inline function which can be used for short snippets of code that are not going to be reuse and not worth naming. In its simplest form lambda expression can be defined as follows:

***[ capture clause ] (parameters) -> return-type***

***{***

***definition of method***

***}***

Generally return-type in lambda expression are evaluated by compiler itself and we don’t need to specify that explicitly and -> return-type part can be ignored but in some complex case as in conditional statement, compiler can’t make out the return type and we need to specify that.

A lambda expression can have more power than an ordinary function by having access to variables from the enclosing scope. We can capture external variables from enclosing scope by three ways :  
      Capture by reference  
      Capture by value  
      Capture by both (mixed capture)

Syntax used for capturing variables :  
      [&] : capture all external variable by reference  
      [=] : capture all external variable by value  
      [a, &b] : capture a by value and b by reference

A lambda with empty capture clause [ ] can access only those variable which are local to it.  
Capturing ways are demonstrated below :

Lambda expression can work only on C++ 11 and after versions.

132) You are given an array of words where each word consists of lowercase English letters.

wordA is a **predecessor** of wordB if and only if we can insert **exactly one** letter anywhere in wordA **without changing the order of the other characters** to make it equal to wordB.

* For example, "abc" is a **predecessor** of "abac", while "cba" is not a **predecessor** of "bcad".

A **word chain**is a sequence of words [word1, word2, ..., wordk] with k >= 1, where word1 is a **predecessor** of word2, word2 is a **predecessor** of word3, and so on. A single word is trivially a **word chain** with k == 1.

Return *the****length****of the****longest possible word chain****with words chosen from the given list of*words.

**Example 1:**

**Input:** words = ["a","b","ba","bca","bda","bdca"]

**Output:** 4

**Explanation**: One of the longest word chains is ["a","ba","bda","bdca"].

: int longestStrChain(vector<string>& words) {

sort(words.begin(),words.end(),[](string a,string b){return a.size()<b.size();});

int ans=0;

map<string,int>dp;

for(auto it:words)

{

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

{

dp[it]=max(dp[it.substr(0,i)+it.substr(i+1)]+1,dp[it]);

}

ans=max(ans,dp[it]);

}

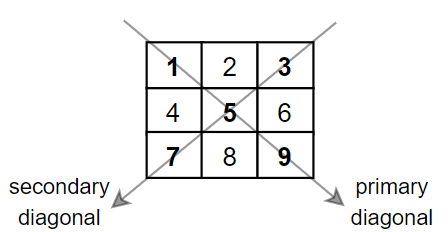
return ans;

}

133) Given a square matrix mat, return the sum of the matrix diagonals.

Only include­­ the sum of all the elements on the primary diagonal and all the elements on the secondary diagonal that are not part of the primary diagonal.

**Example 1:**



**Input:** mat = [[**1**,2,**3**],

  [4,**5**,6],

  [**7**,8,**9**]]

**Output:** 25

**Explanation:** Diagonals sum: 1 + 5 + 9 + 3 + 7 = 25

Notice that element mat[1][1] = 5 is counted only once.

: int diagonalSum(vector<vector<int>>& mat) {

int ans=0;

int n=mat.size();

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

{

if(i!=n-i-1)

ans+=mat[i][i]+mat[n-i-1][i];

else

ans+=mat[i][i];

}

return ans;

}

134) You are given an integer array nums. The unique elements of an array are the elements that appear **exactly once** in the array.

Return *the****sum****of all the unique elements of*nums.

**Example 1:**

**Input:** nums = [1,2,3,2]

**Output:** 4

**Explanation:** The unique elements are [1,3], and the sum is 4.

: int sumOfUnique(vector<int>& nums) {

int ans=0;

int arr[101]={};

for(auto it:nums)

{

arr[it]++;

}

for(int i=1;i<=100;i++)

{

if(arr[i]==1)

ans+=i;

}

return ans;

}

135) Given two string arrays word1 and word2, returntrue*if the two arrays****represent****the same string, and*false*otherwise.*

A string is **represented** by an array if the array elements concatenated **in order** forms the string.

**Example 1:**

**Input:** word1 = ["ab", "c"], word2 = ["a", "bc"]

**Output:** true

**Explanation:**

word1 represents string "ab" + "c" -> "abc"

word2 represents string "a" + "bc" -> "abc"

The strings are the same, so return true.

: bool arrayStringsAreEqual(vector<string>& word1, vector<string>& word2) {

string str1="";

for(int i=0;i<word1.size();i++)

{

str1+=word1[i];

}

string str2="";

for(int i=0;i<word2.size();i++)

{

str2+=word2[i];

}

int j=0;

for(int i=0;i<max(str1.size(),str2.size());i++)

{

if(str1[i]!=str2[j++])

return 0;

}

return 1;

}

136) Given a string s consisting of some words separated by some number of spaces, return *the length of the****last****word in the string.*

A **word** is a maximal substring consisting of non-space characters only.

**Example 1:**

**Input:** s = "Hello World"

**Output:** 5

**Explanation:** The last word is "World" with length 5.

: int lengthOfLastWord(string s) {

int l=0,r=s.size()-1;

while(r>=0&&s[r]==' ')

r--;

while(r>=0&&s[r]!=' ')

{

r--;

l++;

}

return l;

}

137) The **Fibonacci numbers**, commonly denoted F(n) form a sequence, called the **Fibonacci sequence**, such that each number is the sum of the two preceding ones, starting from 0 and 1. That is,

F(0) = 0, F(1) = 1

F(n) = F(n - 1) + F(n - 2), for n > 1.

Given n, calculate F(n).

**Example 1:**

**Input:** n = 2

**Output:** 1

**Explanation:** F(2) = F(1) + F(0) = 1 + 0 = 1.

### : **Solution 1 - Recursive Approach**

Time Complexity - O(2^N)  
Space Complexity - O(N) given the function call stack size

int fib(int N) {

if(N == 0) return 0;

if(N == 1) return 1;

return fib(N-1) + fib(N-2);

}

### **Solution 2 - Dynamic Programming Approach**

Use memoization to store perviously computed fibonacci values.  
Time Complexity - O(N)  
Space Complexity - O(N)

int fib(int N) {

if(N < 2)

return N;

int memo[N+1];

memo[0] = 0;

memo[1] = 1;

for(int i=2; i<=N; i++)

memo[i] = memo[i-1] + memo[i-2];

return memo[N];

}

### **Solution 3 - Imperative Approach (Bottom Up DP)**

With Imperative approach, we step through the loop and optimize the space by storing only two previous fibonacci values in two variables.  
Time Complexity - O(N)  
Space Complexity - O(1)

int fib(int N) {

if(N < 2)

return N;

int a = 0, b = 1, c = 0;

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

{

c = a + b;

a = b;

b = c;

}

return c;

}

### **Solution 4 - Binet's Nth-term Formula**

Using Binet's Formula for the Nth Fibonacci involves the usage of our golden section number **Phi**.  
**Phi** = ( sqrt(5) + 1 ) / 2  
Using approximation equation is good enough here, since we know N >= 0 && N <= 30, we can safely use the following rounded function  
Fib(N) = round( ( **Phi**^N ) / sqrt(5) )  
Full mathematical explanation of Binet's Formula [here](http://www.maths.surrey.ac.uk/hosted-sites/R.Knott/Fibonacci/fibFormula.html)  
Time Complexity - O(1)  
Space Complexity - O(1)

int fib(int N) {

double phi = (sqrt(5) + 1) / 2;

return round(pow(phi, N) / sqrt(5));

}

138) The Tribonacci sequence Tn is defined as follows:

T0 = 0, T1 = 1, T2 = 1, and Tn+3 = Tn + Tn+1 + Tn+2 for n >= 0.

Given n, return the value of Tn.

**Example 1:**

**Input:** n = 4

**Output:** 4

**Explanation:**

T\_3 = 0 + 1 + 1 = 2

T\_4 = 1 + 1 + 2 = 4

: int dp[38]={0};

int tribonacci(int n) {

if(n==0||n==1)

return n;

if(n==2)

return 1;

if(dp[n]!=0)

return dp[n];

return dp[n]=tribonacci(n-1)+tribonacci(n-2)+tribonacci(n-3);

}

139) Given an integer array nums, return the length of the longest strictly increasing subsequence.

A **subsequence** is a sequence that can be derived from an array by deleting some or no elements without changing the order of the remaining elements. For example, [3,6,2,7] is a subsequence of the array [0,3,1,6,2,2,7].

 ALL CODE VARIATIONS GET COVERED

**Example 1:**

**Input:** nums = [10,9,2,5,3,7,101,18]

**Output:** 4

**Explanation:** The longest increasing subsequence is [2,3,7,101], therefore the length is 4.

: int lengthOfLIS(vector<int>& nums) {

int n=nums.size();

vector<int>dp(n,1);

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

{

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

{

if(nums[i]>nums[j]&&dp[i]<dp[j]+1)

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

}

}

return \*max\_element(dp.begin(),dp.end());

}

\*\*\*\*\*\*\*\*\*\*\*NlogN\*\*\*\*\*\*

: int lengthOfLIS(vector<int>& nums) {

vector<int> res;

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

auto it = std::lower\_bound(res.begin(), res.end(), nums[i]);

if(it==res.end()) res.push\_back(nums[i]);

else \*it = nums[i];

}

return res.size();

}

\*\*\*\*\*using upperbound ans set\*\*\*\*\*\*\*\*\*\*\*\*\*

: class Solution {

public:

int lengthOfLIS(vector<int>& nums) {

set<int> s;

for (auto a : nums) {

if (s.find(a) != s.end()) continue;

s.insert(a);

auto it = s.upper\_bound(a);

if (it != s.end()) s.erase(it);

}

return s.size();

}

};

Indeed, geeksforgeeks provides a very detailed explanation, but maybe a short summary would help.

1. The algo is O(nlogn) because lower\_bound() is logarithmic on a sorted input. We keep our vector res sorted, so the search in dp is logarithmic.
2. Res is composed to be:

* sorted
* having a length of the longest found increasing sub-sequence  
  So it doesn't contain that subsequence. Only it's length is valid.

1. So what algo is doing? For each number we have 2 options:

* if it's the highest found value, we push it back, since a high value obviously makes our increasing sequence longer
* if it's not the highest found value, then it could be a nice start (or continuation) of a shorter sequence. And we keep that sequence in place. Merging all found and possible sequences into one dp

See how res is changing when we go through the numbers:

[1,2,7,8,3,4,5,9,0]

1 -> [1]

2 -> [1,2]

7 -> [1,2,7]

8 -> [1,2,7,8]

3 -> [1,2,3,8] // we replaced 7 with 3, since for the longest sequence we need only the last number and 1,2,3 is our new shorter sequence

4 -> [1,2,3,4] // we replaced 8 with 4, since the max len is the same but 4 has more chances for longer sequence

5 -> [1,2,3,4,5]

9 -> [1,2,3,4,5,9]

0 -> [0,2,3,4,5,9] // we replaced 1 with 0, so that it can become a new sequence

So in the end our res contains [0,2,3,4,5,9] which is not a found sequence, but it has the length of the valid answer = 6.

140) You are given weights and values of **N** items, put these items in a knapsack of capacity **W** to get the maximum total value in the knapsack. Note that we have only **one quantity of each item**.  
In other words, given two integer arrays **val[0..N-1]** and **wt[0..N-1]** which represent values and weights associated with **N** items respectively. Also given an integer W which represents knapsack capacity, find out the maximum value subset of **val[]** such that sum of the weights of this subset is smaller than or equal to **W.** You cannot break an item, **either pick the complete item or don’t pick it (0-1 property)**.

**Example 1:**

**Input:**

N = 3

W = 4

values[] = {1,2,3}

weight[] = {4,5,1}

**Output:** 3

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

{

// Your code here

int dp[1001][1001];

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

{

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

{

dp[i][j]=0;

}

}

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

{

for(int j=1;j<W+1;j++)

{

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];

}

141) Given an array of non-negative integers, and a value *sum*, determine if there is a subset of the given set with sum equal to given *sum*.

**Example 1:**

**Input**:

N = 6

arr[] = {3, 34, 4, 12, 5, 2}

sum = 9

**Output:** 1

**Explanation**: Here there exists a subset with

sum = 9, 4+3+2 = 9.

: bool isSubsetSum(int N, int arr[], int sum){

// code here

bool dp[N+1][sum];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

if(i==0&&j==0)

dp[i][j]=1;

}

}

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

{

for(int j=1;j<=sum;j++)

{

if(arr[i-1]<=j)

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

else

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

}

}

return dp[N][sum];

}

142) Given a **non-empty** array nums containing **only positive integers**, find if the array can be partitioned into two subsets such that the sum of elements in both subsets is equal.

**Example 1:**

**Input:** nums = [1,5,11,5]

**Output:** true

**Explanation:** The array can be partitioned as [1, 5, 5] and [11].

: bool subsum(vector<int>v,int sum)

{

bool dp[v.size()+1][sum+1];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

if(i==0&&j==0)

dp[i][j]=1;

}

}

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

{

for(int j=1;j<=sum;j++)

{

if(v[i-1]<=j)

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

else

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

}

}

return dp[v.size()][sum];

}

bool canPartition(vector<int>& nums) {

int sum=0;

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

{

sum+=nums[i];

}

if(sum%2!=0)

return 0;

else

return subsum(nums,sum/2);

}

143) Given an array **arr[]** of integers and an integer **sum**, the task is to count all subsets of the given array with a sum equal to a given **sum**.

Note: Answer can be very large, so, output answer modulo 109+7

**Example 1:**

**Input**: N = 6, arr[] = {2, 3, 5, 6, 8, 10}

sum = 10

**Output:** 3

**Explanation**: {2, 3, 5}, {2, 8}, {10}

: int perfectSum(int arr[], int n, int sum)

{

// Your code goes here

int dp[n+1][sum+1];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

dp[i][0]=1;

}

}

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

{

for(int j=1;j<=sum;j++)

{

if(arr[i-1]<=j)

dp[i][j]=dp[i-1][j-arr[i-1]]%1000000007+dp[i-1][j]%1000000007;

else

dp[i][j]=dp[i-1][j]%1000000007;

}

}

return dp[n][sum]%1000000007;

}

144) **Problem Description**

Given an integer array **A** containing **N** integers.

You need to divide the array **A** into two subsets S1 and S2 such that the absolute difference between their sums is minimum.

Find and return this **minimum possible absolute difference**.

**NOTE:**

 Subsets can contain elements from A in any order (not necessary to be contiguous).

 Each element of A should belong to any one subset S1 or S2, not both.

 It may be possible that one subset remains empty.

**Problem Constraints**

1 <= N <= 100

1 <= A[i] <= 100

**Input Format**

First and only argument is an integer array **A**.

**Output Format**

Return an integer denoting the minimum possible difference among the sums of two subsets.

**Example Input**

Input 1:

A = [1, 6, 11, 5]

**Example Output**

Output 1:

1

: int Solution::solve(vector<int> &A) {

    int sum=0;

    int n=A.size();

    for(int i=0;i<A.size();i++)

    {

        sum+=A[i];

    }

    int s=sum;

    sum=sum/2;

    int dp[n+1][sum+1];

    for(int i=0;i<=A.size();i++)

    {

        for(int j=0;j<=sum;j++)

        {

            if(i==0||j==0)

            dp[i][j]=0;

            dp[i][0]=1;

        }

    }

    int ans=0;

    for(int i=1;i<=A.size();i++)

    {

        for(int j=1;j<=sum;j++)

        {

            if(A[i-1]<=j)

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

            else

            dp[i][j]=dp[i-1][j];

            if(dp[i][j]==1)

            ans=j;

        }

    }

    return s-2\*ans;

}

145)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*TARGET SUM==COUNT # OF SUBSET WITH GIVEN DIFFERENCE\*\*\*\*\*\*\*\*\*\*

You are given an integer array nums and an integer target.

You want to build an **expression** out of nums by adding one of the symbols '+' and '-' before each integer in nums and then concatenate all the integers.

* For example, if nums = [2, 1], you can add a '+' before 2 and a '-' before 1 and concatenate them to build the expression "+2-1".

Return the number of different **expressions** that you can build, which evaluates to target.

**Example 1:**

**Input:** nums = [1,1,1,1,1], target = 3

**Output:** 5

**Explanation:** There are 5 ways to assign symbols to make the sum of nums be target 3.

-1 + 1 + 1 + 1 + 1 = 3

+1 - 1 + 1 + 1 + 1 = 3

+1 + 1 - 1 + 1 + 1 = 3

+1 + 1 + 1 - 1 + 1 = 3

+1 + 1 + 1 + 1 - 1 = 3

: int subsum(vector<int>v,int k)

{

int n=v.size();

int dp[n+1][k+1];

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

{

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

{

if(i==0)

dp[i][j]=0;

if(j==0)

dp[i][j]=1;

}

}

dp[0][0]=1;

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

{

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

{

if(v[i-1]<=j)

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

else

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

}

}

return dp[n][k];

}

int findTargetSumWays(vector<int>& nums, int target) {

int ans=0;

target=abs(target);

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

{

ans+=nums[i];

}

if(ans<target||(target+ans)%2!=0)

return 0;

int ans1=(target+ans)/2;

return subsum(nums,ans1);

}

146) Given a set of **N** items, each with a weight and a value, represented by the array **w[]** and **val[]** respectively. Also, a knapsack with weight limit **W**.  
The task is to fill the knapsack in such a way that we can get the maximum profit. Return the maximum profit.  
Note: Each item can be taken any number of times.

**Example 1:**

**Input:** N = 2, W = 3

val[] = {1, 1}

wt[] = {2, 1}

**Output:** 3

**Explanation:**

1.Pick the 2nd element thrice.

2.Total profit = 1 + 1 + 1 = 3. Also the total

 weight = 1 + 1 + 1 = 3 which is <= W.

: int knapSack(int N, int W, int val[], int wt[])

{

// code here

int dp[N+1][W+1];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

dp[i][0]=0;

}

}

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

{

for(int j=1;j<=W;j++)

{

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

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

else

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

}

}

return dp[N][W];

}

147) You are given an integer array coins representing coins of different denominations and an integer amount representing a total amount of money.

Return *the fewest number of coins that you need to make up that amount*. If that amount of money cannot be made up by any combination of the coins, return -1.

You may assume that you have an infinite number of each kind of coin.

**Example 1:**

**Input:** coins = [1,2,5], amount = 11

**Output:** 3

**Explanation:** 11 = 5 + 5 + 1

: int coinChange(vector<int>& coins, int amount) {

vector<int>dp(amount+1,amount+1);

dp[0]=0;

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

{

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

{

if(coins[j]<=i)

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

}

}

return dp[amount]>amount?-1:dp[amount];

}

HOUSE ROBBER MORE VARIATIONS

148) You are given an integer array nums. You want to maximize the number of points you get by performing the following operation any number of times:

* Pick any nums[i] and delete it to earn nums[i] points. Afterwards, you must delete **every** element equal to nums[i] - 1 and **every** element equal to nums[i] + 1.

Return *the****maximum number of points****you can earn by applying the above operation some number of times*.

**Example 1:**

**Input:** nums = [3,4,2]

**Output:** 6

**Explanation:** You can perform the following operations:

- Delete 4 to earn 4 points. Consequently, 3 is also deleted. nums = [2].

- Delete 2 to earn 2 points. nums = [].

You earn a total of 6 points.

: int deleteAndEarn(vector<int>& nums) {

int n=10001;

vector<int>bucket(n,0);

for(auto it:nums)

bucket[it]+=it;

vector<int>dp(n,0);

dp[0]=0;

dp[1]=bucket[1];

for(int i=2;i<n;i++)

dp[i]=max(bucket[i]+dp[i-2],dp[i-1]);

return dp[n-1];

}

Gfg variant

Find maximum possible stolen value from houses(STIKLER THIEF)

There are n houses build in a line, each of which contains some value in it. A thief is going to steal the maximal value of these houses, but he can’t steal in two adjacent houses because the owner of the stolen houses will tell his two neighbors left and right side. What is the maximum stolen value?

: int FindMaxSum(int arr[], int n)

{

// Your code here

if(n==0)

return 0;

if(n==1)

return arr[0];

if(n==2)

return max(arr[0],arr[1]);

int dp[n];

dp[0]=arr[0];

dp[1]=max(arr[0],arr[1]);

for(int i=2;i<n;i++)

dp[i]=max(arr[i]+dp[i-2],dp[i-1]);

return dp[n-1];

}

Leetcode variant

There is some frustration when people publish their perfect fine-grained algorithms without sharing any information abut how they were derived. This is an attempt to change the situation. There is not much more explanationpath but it's rather an example of higher level improvements. Converting a solution to the next step shouldn't be as hard as attempting to come up with perfect algorithm at first attempt.

This particular problem and most of others can be approached using the following sequence:

1. Find recursive relation
2. Recursive (top-down)
3. Recursive + memo (top-down)
4. Iterative + memo (bottom-up)
5. Iterative + N variables (bottom-up)

**Step 1.** Figure out recursive relation.  
A robber has 2 options: a) rob current house i; b) don't rob current house.  
If an option "a" is selected it means she can't rob previous i-1 house but can safely proceed to the one before previous i-2 and gets all cumulative loot that follows.  
If an option "b" is selected the robber gets all the possible loot from robbery of i-1 and all the following buildings.  
So it boils down to calculating what is more profitable:

* robbery of current house + loot from houses before the previous
* loot from the previous house robbery and any loot captured before that

rob(i) = Math.max( rob(i - 2) + currentHouseValue, rob(i - 1) )

**Step 2.** Recursive (top-down)  
Converting the recurrent relation from Step 1 shound't be very hard.

public int rob(int[] nums) {

return rob(nums, nums.length - 1);

}

private int rob(int[] nums, int i) {

if (i < 0) {

return 0;

}

return Math.max(rob(nums, i - 2) + nums[i], rob(nums, i - 1));

}

This algorithm will process the same i multiple times and it needs improvement. Time complexity: [to fill]

**Step 3.** Recursive + memo (top-down).

int[] memo;

public int rob(int[] nums) {

memo = new int[nums.length + 1];

Arrays.fill(memo, -1);

return rob(nums, nums.length - 1);

}

private int rob(int[] nums, int i) {

if (i < 0) {

return 0;

}

if (memo[i] >= 0) {

return memo[i];

}

int result = Math.max(rob(nums, i - 2) + nums[i], rob(nums, i - 1));

memo[i] = result;

return result;

}

Much better, this should run in O(n) time. Space complexity is O(n) as well, because of the recursion stack, let's try to get rid of it.

**Step 4.** Iterative + memo (bottom-up)

public int rob(int[] nums) {

if (nums.length == 0) return 0;

int[] memo = new int[nums.length + 1];

memo[0] = 0;

memo[1] = nums[0];

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

int val = nums[i];

memo[i+1] = Math.max(memo[i], memo[i-1] + val);

}

return memo[nums.length];

}

**Step 5.** Iterative + 2 variables (bottom-up)  
We can notice that in the previous step we use only memo[i] and memo[i-1], so going just 2 steps back. We can hold them in 2 variables instead. This optimization is met in Fibonacci sequence creation and some other problems [to paste links].

/\* the order is: prev2, prev1, num \*/

public int rob(int[] nums) {

if (nums.length == 0) return 0;

int prev1 = 0;

int prev2 = 0;

for (int num : nums) {

int tmp = prev1;

prev1 = Math.max(prev2 + num, prev1);

prev2 = tmp;

}

return prev1;

}

You are a professional robber planning to rob houses along a street. Each house has a certain amount of money stashed, the only constraint stopping you from robbing each of them is that adjacent houses have security systems connected and **it will automatically contact the police if two adjacent houses were broken into on the same night**.

Given an integer array nums representing the amount of money of each house, return *the maximum amount of money you can rob tonight****without alerting the police***.

**Example 1:**

**Input:** nums = [1,2,3,1]

**Output:** 4

: int rob(vector<int>& nums) {

if(nums.size()==0)

return 0;

int pre1=0;

int pre2=0;

for(auto it:nums)

{

int temp=pre1;

pre1=max(pre2+it,pre1);

pre2=temp;

}

return pre1;

}

149) Given an integer **x,** find the square root of x. If **x** is not a perfect square, then return floor(√x).

**Example 1:**

**Input:**

x = 5

**Output:** 2

**Explanation:** Since, 5 is not a perfect

square, floor of square\_root of 5 is 2.

: long long int floorSqrt(long long int x)

{

// Your code goes here

long long int ans=0;

for(long long int i=0;i\*i<=x;i++)

{

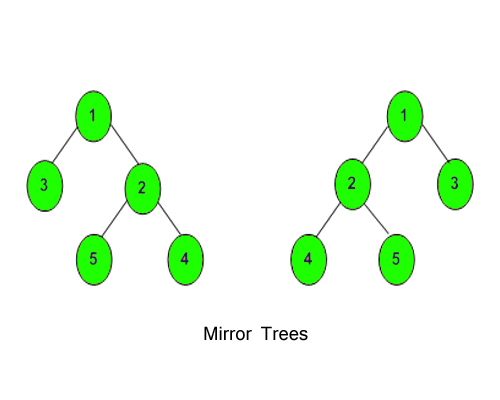
ans=i;

}

return ans;

}

LEETCODE VARIANT

150) Given a Binary Tree, convert it into its mirror.  


**Example 1:**

**Input:**

1

  / \

  2 3

**Output:** 2 1 3

**Explanation:** The tree is

   1   (mirror) 1

/  \   =>      /  \

3    2          2   3

The inorder of mirror is 2 1 3

: void mirror(Node\* node) {

// code here

if(!node)

return;

Node\*l1=node->left;

Node\*l2=node->right;

node->left=l2;

node->right=l1;

mirror(node->left);

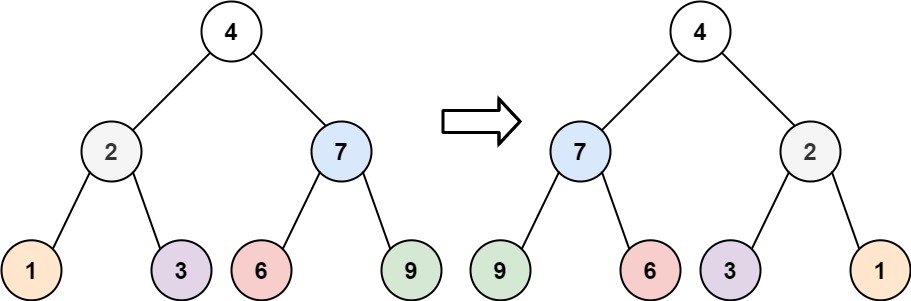
mirror(node->right);

}

LEETCODE VARIANT

: Given the root of a binary tree, invert the tree, and return *its root*.

**Example 1:**



**Input:** root = [4,2,7,1,3,6,9]

**Output:** [4,7,2,9,6,3,1]

### : TreeNode\* invertTree(TreeNode\* root) {

### if(root)

### {

### invertTree(root->left);

### invertTree(root->right);

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

### }

### return root;

### }

151) Given a linked list of **N** nodes where nodes can contain values **0s**, **1s,** and **2s**only. The task is to segregate **0s**, **1s,** and **2s** linked list such that all zeros segregate to head side, 2s at the end of the linked list, and 1s in the mid of 0s and 2s.

**Example 1:**

**Input:**

N = 8

value[] = {1,2,2,1,2,0,2,2}

**Output:** 0 1 1 2 2 2 2 2

**Explanation:** All the 0s are segregated

to the left end of the linked list,

2s to the right end of the list, and

1s in between.

### :

### 152) **2032. Two Out of Three**

Given three integer arrays nums1, nums2, and nums3, return a ***distinct*** array containing all the values that are present in ***at least two*** out of the three arrays. You may return the values in ***any*** order.

**Example 1:**

**Input:** nums1 = [1,1,3,2], nums2 = [2,3], nums3 = [3]

**Output:** [3,2]

**Explanation:** The values that are present in at least two arrays are:

- 3, in all three arrays.

:

vector<int> twoOutOfThree(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3) {

map<int,set<int>>v;

for(int i=0;i<nums1.size();i++)

v[nums1[i]].insert(1);

for(int i=0;i<nums2.size();i++)

v[nums2[i]].insert(2);

for(int i=0;i<nums3.size();i++)

v[nums3[i]].insert(3);

vector<int>ans;

for(auto it=v.begin();it!=v.end();it++)

{

if(it->second.size()>=2)

ans.push\_back(it->first);

}

return ans;

}153) Given two strings text1 and text2, return *the length of their longest****common subsequence****.*If there is no **common subsequence**, return 0.

A **subsequence** of a string is a new string generated from the original string with some characters (can be none) deleted without changing the relative order of the remaining characters.

* For example, "ace" is a subsequence of "abcde".

A **common subsequence** of two strings is a subsequence that is common to both strings.

**Example 1:**

**Input:** text1 = "abcde", text2 = "ace"

**Output:** 3

**Explanation:** The longest common subsequence is "ace" and its length is 3.

: int longestCommonSubsequence(string text1, string text2) {

int n=text1.size();

int m=text2.size();

int dp[n+1][m+1];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(text1[i-1]==text2[j-1])

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

else

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

}

}

return dp[n][m];

}

154) Given two strings str1 and str2, return *the shortest string that has both*str1*and*str2*as****subsequences***. If there are multiple valid strings, return **any** of them.

A string s is a **subsequence** of string t if deleting some number of characters from t (possibly 0) results in the string s.

**Example 1:**

**Input:** str1 = "abac", str2 = "cab"

**Output:** "cabac"

**Explanation:**

str1 = "abac" is a subsequence of "cabac" because we can delete the first "c".

str2 = "cab" is a subsequence of "cabac" because we can delete the last "ac".

The answer provided is the shortest such string that satisfies these properties.

: string shortestCommonSupersequence(string str1, string str2) {

int dp[1001][1001];

for(int i=0;i<str1.size();i++)

{

for(int j=0;j<str2.size();j++)

{

if(i==0||j==0)

dp[i][j]=0;

}

}

for(int i=1;i<=str1.size();i++)

{

for(int j=1;j<=str2.size();j++)

{

if(str1[i-1]==str2[j-1])

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

else

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

}

}

// return (str1.size()+str2.size()-dp[str1.size()][str2.size()]);

int i=str1.size(),j=str2.size();

string s="";

while(i>0&&j>0)

{

if(str1[i-1]==str2[j-1])

{

s.push\_back(str1[i-1]);

i--;

j--;

}

else if(dp[i][j-1]>dp[i-1][j])

{

s.push\_back(str2[j-1]);

j--;

}

else

{

s.push\_back(str1[i-1]);

i--;

}

}

while(i>0)

{

s.push\_back(str1[i-1]);

i--;

}

while(j>0)

{

s.push\_back(str2[j-1]);

j--;

}

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

return s;

}

155) Given two strings **str1** and **str2**. The task is to remove or insert the minimum number of characters from/in **str1** so as to transform it into **str2**. It could be possible that the same character needs to be removed/deleted from one point of str1 and inserted to some another point.

**Example 1:**

**Input:** str1 = "heap", str2 = "pea"

**Output:** 3

**Explanation:** 2 deletions and 1 insertion

**p** and **h** deleted from **heap**. Then, **p** is

inserted at the beginning One thing to

note, though **p** was required yet it was

removed/deleted first from its position

and then it is inserted to some other

position. Thus, **p** contributes one to the

**deletion\_count** and one to the

**insertion\_count**.

: int minOperations(string str1, string str2)

{

// Your code goes here

int n=str1.size();

int m=str2.size();

int dp[1001][1001];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(str1[i-1]==str2[j-1])

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

else

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

}

}

return (n-dp[n][m])+(m-dp[n][m]);

}

156) Given a string s, find *the longest palindromic****subsequence****'s length in* s.

A **subsequence** is a sequence that can be derived from another sequence by deleting some or no elements without changing the order of the remaining elements.

**Example 1:**

**Input:** s = "bbbab"

**Output:** 4

**Explanation:** One possible longest palindromic subsequence is "bbbb".

: int longestPalindromeSubseq(string s) {

int n=s.size();

int m=n;

int dp[1001][1001];

string s1=s;

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

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(s[i-1]==s1[j-1])

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

else

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

}

}

return dp[n][m];

}

157) Given a string of S as input. Your task is to write a program to remove or delete the minimum number of characters from the string so that the resultant string is a palindrome.  
**Note:** The order of characters in the string should be maintained.

**Example 1:**

**Input:** S= **"**aebcbda"

**Output:** 2

**Explanation**: Remove characters 'e'

and 'd'.

Length of string is inversely proportional to # of deletions

: int minimumNumberOfDeletions(string S) {

// code here

int n=S.size();

int m=n;

string s1=S;

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

int dp[1001][1001];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(s1[i-1]==S[j-1])

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

else

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

}

}

return n-dp[n][m];

}

158) Given a string str, find the length of the longest repeating subsequence such that it can be found twice in the given string. The two identified subsequences A and B can use the same ith character from string str if and only if that ith character has different indices in A and B.

**Example 1:**

**Input:**

str = "axxxy"

**Output:** 2

**Explanation:**

The given array with indexes looks like

a x x x y

0 1 2 3 4

The longest subsequence is "xx".

It appears twice as explained below.

**subsequence A**

x x

0 1 <-- index of subsequence A

------

1 2 <-- index of str

**subsequence B**

x x

0 1 <-- index of subsequence B

------

2 3 <-- index of str

We are able to use character 'x'

(at index 2 in str) in both subsequences

as it appears on index 1 in subsequence A

and index 0 in subsequence B.

: int LongestRepeatingSubsequence(string str){

// Code here

int n=str.size();

int m=n;

string s=str;

int dp[1001][1001];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(str[i-1]==s[j-1]&&i!=j)

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

else

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

}

}

return dp[n][m];

}

159) Given a sequence of matrices, find the most efficient way to multiply these matrices together. The efficient way is the one that involves the least number of multiplications.

The dimensions of the matrices are given in an array **arr[]** of size **N** (such that N = number of matrices + 1) where the **ith** matrix has the dimensions **(arr[i-1] x arr[i])**.

**Example 1:**

**Input:** N = 5

arr = {40, 20, 30, 10, 30}

**Output:** 26000

**Explaination:** There are 4 matrices of dimension

40x20, 20x30, 30x10, 10x30. Say the matrices are

named as A, B, C, D. Out of all possible combinations,

the most efficient way is (A\*(B\*C))\*D.

The number of operations are -

20\*30\*10 + 40\*20\*10 + 40\*10\*30 = 26000.

: int dp[101][101];

int solve(int arr[],int i,int j)

{

if(i>=j)

return 0;

if(dp[i][j]!=-1)

return dp[i][j];

int ans=INT\_MAX;

for(int k=i;k<=j-1;k++)

{

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

if(temp<ans)

ans=temp;

}

return dp[i][j]=ans;

}

int matrixMultiplication(int N, int arr[])

{

// code here

memset(dp,-1,sizeof(dp));

int ans=solve(arr,1,N-1);

return ans;

}

160) Given an array **Arr[]** that contains **N** integers (may be **positive**, **negative**or **zero**). Find the product of the maximum product subarray.

**Example 1:**

**Input:**

N = 5

Arr[] = {6, -3, -10, 0, 2}

**Output:** 180

**Explanation:** Subarray with maximum product

is [6, -3, -10] which gives product as 180.

: long long maxProduct(vector<int> arr, int n) {

long long maxp=arr[0];

long long minp=arr[0];

long long res=arr[0];

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

if(arr[i]<0)

swap(maxp,minp);

maxp=max((long long)arr[i],maxp\*arr[i]);

minp=min((long long)arr[i],minp\*arr[i]);

res=max(res,maxp);

}

return res;

}

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

A **palindrome** string is a string that reads the same backward as forward.

**Example 1:**

**Input:** s = "aab"

**Output:** [["a","a","b"],["aa","b"]]

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

vector<vector<string>>v;

if(s.empty())

return v;

vector<string>temp;

func(0,s,temp,v);

return v;

}

void func(int ind,string&s,vector<string>&temp,vector<vector<string>>&v)

{

if(ind==s.size())

{

v.push\_back(temp);

return;

}

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

{

if(is\_palindrome(s,ind,i))

{

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

func(i+1,s,temp,v);

temp.pop\_back();

}

}

}

bool is\_palindrome(string s,int l,int k)

{

while(l<=k)

{

if(s[l++]!=s[k--])

return 0;

}

return 1;

}

162) You are given **N** identical eggs and you have access to a **K**-floored building from **1** to **K**.

There exists a floor **f** where **0** <= **f** <= **K**such that any egg dropped at a floor higher than **f** will break, and any egg dropped **at or below**floor **f**will **not break**. There are few rules given below.

* An egg that survives a fall can be used again.
* A broken egg must be discarded.
* The effect of a fall is the same for all eggs.
* If the egg doesn't break at a certain floor, it will not break at any floor below.
* If the eggs breaks at a certain floor, it will break at any floor above.

Return the minimum number of moves that you need to determine with certainty what the value of **f** is.

For more description on this problem see [wiki page](http://en.wikipedia.org/wiki/Dynamic_programming#Egg_dropping_puzzle)

**Example 1:**

**Input:**

**N** = 1**, K** = 2

**Output:** 2

**Explanation:**

1. Drop the egg from floor 1. If it

  breaks, we know that f = 0.

2. Otherwise, drop the egg from floor 2.

  If it breaks, we know that f = 1.

3. If it does not break, then we know f = 2.

4. Hence, we need at minimum 2 moves to

  determine with certainty what the value of f is.

: int dp[201][201];

int solve(int n,int a)

{

if(n==1)

return a;

if(a==1||a==0)

return a;

if(dp[n][a]!=-1)

return dp[n][a];

int ans=INT\_MAX;

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

{

int temp=1+max(solve(n-1,i-1),solve(n,a-i));

ans=min(ans,temp);

}

return dp[n][a]=ans;

}

int eggDrop(int n, int k)

{

// your code here

memset(dp,-1,sizeof(dp));

return solve(n,k);

}

163) Given a string s and a dictionary of strings wordDict, return true if s can be segmented into a space-separated sequence of one or more dictionary words.

**Note** that the same word in the dictionary may be reused multiple times in the segmentation.

**Example 1:**

**Input:** s = "leetcode", wordDict = ["leet","code"]

**Output:** true

**Explanation:** Return true because "leetcode" can be segmented as "leet code".

: bool wordBreak(string s, vector<string>& wordDict) {

vector<bool>dp(s.size(),0);

dp[0]=1;

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

{

for(auto str:wordDict)

{

if(dp[i])

{

if(s.substr(i,str.size()).compare(str)==0)

dp[str.size()+i]=1;

}

}

}

return dp[s.size()];

}

164) You are given a **large integer** represented as an integer array digits, where each digits[i] is the ith digit of the integer. The digits are ordered from most significant to least significant in left-to-right order. The large integer does not contain any leading 0's.

Increment the large integer by one and return *the resulting array of digits*.

**Example 1:**

**Input:** digits = [1,2,3]

**Output:** [1,2,4]

**Explanation:** The array represents the integer 123.

Incrementing by one gives 123 + 1 = 124.

Thus, the result should be [1,2,4].

: vector<int> plusOne(vector<int>& digits) {

int n=digits.size();

for(int i=digits.size()-1;i>=0;i--)

{

if(++digits[i]%=10)

{

return digits;

}

}

digits[0]=1;

digits.push\_back(0);

return digits;

}

165) A number **N** is represented in Linked List such that each digit corresponds to a node in linked list. You need to add 1 to it.

**Example 1:**

**Input:**

LinkedList: 4->5->6

**Output:** 457

: Node\* addOne(Node \*head)

{

if(!head->next)

{

head->data+=1;

return head;

}

addOne(head->next);

int carry=0;

if(head->next->data>9)

{

carry=head->next->data/10;

int rem=head->next->data%10;

head->next->data=rem;

}

head->data+=carry;

return head;

}

166) You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1:**

**Input:** n = 2

**Output:** 2

**Explanation:** There are two ways to climb to the top.

1. 1 step + 1 step

2. 2 steps

: int climbStairs(int n) {

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

dp[0]=1;

dp[1]=2;

for(int i=2;i<n;i++)

{

dp[i]=dp[i-1]+dp[i-2];

}

return dp[n-1];

}

167) There are n pieces arranged in a line, and each piece is colored either by 'A' or by 'B'. You are given a string colors of length n where colors[i] is the color of the ith piece.

Alice and Bob are playing a game where they take **alternating turns** removing pieces from the line. In this game, Alice moves**first**.

* Alice is only allowed to remove a piece colored 'A' if **both its neighbors** are also colored 'A'. She is **not allowed** to remove pieces that are colored 'B'.
* Bob is only allowed to remove a piece colored 'B' if **both its neighbors** are also colored 'B'. He is **not allowed** to remove pieces that are colored 'A'.
* Alice and Bob **cannot** remove pieces from the edge of the line.
* If a player cannot make a move on their turn, that player **loses** and the other player **wins**.

Assuming Alice and Bob play optimally, return true*if Alice wins, or return*false*if Bob wins*.

**Example 1:**

**Input:** colors = "AAABABB"

**Output:** true

**Explanation:**

AAABABB -> AABABB

Alice moves first.

She removes the second 'A' from the left since that is the only 'A' whose neighbors are both 'A'.

Now it's Bob's turn.

Bob cannot make a move on his turn since there are no 'B's whose neighbors are both 'B'.

Thus, Alice wins, so return true.

: bool winnerOfGame(string colors) {

int ans=0,ans1=0;

if(colors.size()<=2)

return 0;

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

{

if(colors[i]==colors[i-1]&&colors[i]==colors[i+1])

{

if(colors[i]=='A')

ans++;

else

ans1++;

}

}

return ans>ans1;

}

168) Given an integer, write a function that calculates ⌈7n/8⌉ ([ceiling](http://en.wikipedia.org/wiki/Floor_and_ceiling_functions)of 7n/8) without using division and multiplication operators.

: **int** multiplyBySevenByEight(**int** n)

{

    // Note the inner bracket here. This is needed

    // because precedence of '-' operator is higher

    // than '<<'

**return** (n - (n >> 3));

}

The above method doesn’t always produce result same as “printf(“%u”, 7\*n/8)”. For example, the value of expression 7\*n/8 is 13 for n = 15, but above program produces 14. Below is modified version that always matches 7\*n/8. The idea is to first multiply the number with 7, then divide by 8 as it happens in expression 7\*n/8.

**int** multiplyBySevenByEight(unsigned **int** n)

{

    /\* Step 1) First multiply number by 7 i.e. 7n = (n << 3) -n

     \* Step 2) Divide result by 8 \*/

**return** ((n << 3) -n) >> 3;

}

169) Given an integer array nums and an integer k, return *the number of pairs* (i, j) *where* i < j *such that* |nums[i] - nums[j]| == k.

The value of |x| is defined as:

* x if x >= 0.
* -x if x < 0.

**Example 1:**

**Input:** nums = [1,2,2,1], k = 1

**Output:** 4

**Explanation:** The pairs with an absolute difference of 1 are:

- [**1**,**2**,2,1]

- [**1**,2,**2**,1]

- [1,**2**,2,**1**]

- [1,2,**2**,**1**]

: int countKDifference(vector<int>& nums, int k) {

int ans=0;

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

{

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

{

ans+=abs(nums[i]-nums[j])==k;

}

}

return ans;

}

170) An integer array original is transformed into a **doubled** array changed by appending **twice the value** of every element in original, and then randomly **shuffling** the resulting array.

Given an array changed, return original*if*changed*is a****doubled****array. If*changed*is not a****doubled****array, return an empty array. The elements in* original *may be returned in****any****order*.

**Example 1:**

**Input:** changed = [1,3,4,2,6,8]

**Output:** [1,3,4]

**Explanation:** One possible original array could be [1,3,4]:

- Twice the value of 1 is 1 \* 2 = 2.

- Twice the value of 3 is 3 \* 2 = 6.

- Twice the value of 4 is 4 \* 2 = 8.

Other original arrays could be [4,3,1] or [3,1,4].

: vector<int> findOriginalArray(vector<int>& changed) {

unordered\_map<int,int>m;

for(auto it:changed)

m[it]++;

vector<int>ans;

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

for(auto it:changed)

{

if(m[it]&&m[2\*it])

{

m[it]--;

m[2\*it]--;

ans.push\_back(it);

}

}

for(auto [a,b]:m)

if(b)

return {};

return ans;

}

171)You are given an integer array nums. In one operation, you can replace **any** element in nums with **any** integer.

nums is considered **continuous** if both of the following conditions are fulfilled:

* All elements in nums are **unique**.
* The difference between the **maximum** element and the **minimum** element in nums equals nums.length - 1.

For example, nums = [4, 2, 5, 3] is **continuous**, but nums = [1, 2, 3, 5, 6] is **not continuous**.

Return *the****minimum****number of operations to make*nums***continuous***.

**Example 1:**

**Input:** nums = [4,2,5,3]

**Output:** 0

**: int minOperations(vector<int>& nums) {**

**int i=0,j=0;**

**set<int>st(nums.begin(),nums.end());**

**vector<int>v(st.begin(),st.end());**

**for(int k=v.size();j<k;j++)**

**{**

**if(v[i]+nums.size()<=v[j])**

**i++;**

**}**

**return nums.size()-j+i;**

**}**

172) Given an array of integers nums, calculate the **pivot index** of this array.

The **pivot index** is the index where the sum of all the numbers **strictly** to the left of the index is equal to the sum of all the numbers **strictly** to the index's right.

If the index is on the left edge of the array, then the left sum is 0 because there are no elements to the left. This also applies to the right edge of the array.

Return *the****leftmost pivot index***. If no such index exists, return -1.

**Example 1:**

**Input:** nums = [1,7,3,6,5,6]

**Output:** 3

**Explanation:**

The pivot index is 3.

Left sum = nums[0] + nums[1] + nums[2] = 1 + 7 + 3 = 11

Right sum = nums[4] + nums[5] = 5 + 6 = 11

: int pivotIndex(vector<int>& nums) {

if(nums.size()==1)

return 1;

int ans=0;

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

{

ans+=nums[i];

}

int ans1=0;

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

{

ans-=nums[i];

if(ans==ans1)

return i;

ans1+=nums[i];

}

return -1;

}

173) Given an integer array nums, return *an array* answer *such that* answer[i] *is equal to the product of all the elements of* nums *except* nums[i].

The product of any prefix or suffix of nums is **guaranteed** to fit in a **32-bit** integer.

You must write an algorithm that runs in O(n) time and without using the division operation.

**Example 1:**

**Input:** nums = [1,2,3,4]

**Output:** [24,12,8,6]

: vector<int> productExceptSelf(vector<int>& nums) {

vector<int>v;

int ans=1;

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

{

ans\*=nums[i];

v.push\_back(ans);

}

ans=1;

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

{

v[i]=v[i-1]\*ans;;

ans\*=nums[i];

}

v[0]=ans;

return v;

}

174) Implement the RandomizedSet class:

* RandomizedSet() Initializes the RandomizedSet object.
* bool insert(int val) Inserts an item val into the set if not present. Returns true if the item was not present, false otherwise.
* bool remove(int val) Removes an item val from the set if present. Returns true if the item was present, false otherwise.
* int getRandom() Returns a random element from the current set of elements (it's guaranteed that at least one element exists when this method is called). Each element must have the **same probability** of being returned.

You must implement the functions of the class such that each function works in **average** O(1) time complexity.

**Example 1:**

**Input**

["RandomizedSet", "insert", "remove", "insert", "getRandom", "remove", "insert", "getRandom"]

[[], [1], [2], [2], [], [1], [2], []]

**Output**

[null, true, false, true, 2, true, false, 2]

**Explanation**

RandomizedSet randomizedSet = new RandomizedSet();

randomizedSet.insert(1); // Inserts 1 to the set. Returns true as 1 was inserted successfully.

randomizedSet.remove(2); // Returns false as 2 does not exist in the set.

randomizedSet.insert(2); // Inserts 2 to the set, returns true. Set now contains [1,2].

randomizedSet.getRandom(); // getRandom() should return either 1 or 2 randomly.

randomizedSet.remove(1); // Removes 1 from the set, returns true. Set now contains [2].

randomizedSet.insert(2); // 2 was already in the set, so return false.

randomizedSet.getRandom(); // Since 2 is the only number in the set, getRandom() will always return 2.

: class RandomizedSet {

public:

set<int>st;

RandomizedSet() {

}

bool insert(int val) {

if(st.find(val)!=st.end())

{

return 0;

}

else

{

st.insert(val);

return 1;

}

}

bool remove(int val) {

if(st.find(val)!=st.end())

{

st.erase(val);

return 1;

}

return 0;

}

int getRandom() {

auto it=st.begin();

advance(it,rand()%st.size());

return \*it;

}

};

/\*\*

\* Your RandomizedSet object will be instantiated and called as such:

\* RandomizedSet\* obj = new RandomizedSet();

\* bool param\_1 = obj->insert(val);

\* bool param\_2 = obj->remove(val);

\* int param\_3 = obj->getRandom();

\*/

174) Given an array of integers nums and an integer k, return *the total number of continuous subarrays whose sum equals to k*.

**Example 1:**

**Input:** nums = [1,1,1], k = 2

**Output:** 2

: int subarraySum(vector<int>& nums, int k) {

unordered\_map<int,int>m;

int sum=0;

int i=0;

int cnt=0;

while(i<nums.size())

{

sum+=nums[i];

if(sum==k)

cnt++;

if(m.find(sum-k)!=m.end())

cnt+=m[sum-k];

m[sum]++;

i++;

}

return cnt;

}

175) You are given an integer array nums and an integer x. In one operation, you can either remove the leftmost or the rightmost element from the array nums and subtract its value from x. Note that this **modifies** the array for future operations.

Return *the****minimum number****of operations to reduce*x *to****exactly*** 0 *if it is possible, otherwise, return*-1.

**Example 1:**

**Input:** nums = [1,1,4,2,3], x = 5

**Output:** 2

**Explanation:** The optimal solution is to remove the last two elements to reduce x to zero.

: int minOperations(vector<int>& nums, int x) {

int target=0;

for(auto it:nums)

target+=it;

if(target<x)

return -1;

target-=x;

int ans=-1,s=0,sum=0;

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

{

sum+=nums[i];

while(sum>target)

{

sum-=nums[s];

s++;

}

if(target==sum)

ans=max(ans,i-s+1);

}

return ans==-1?-1:nums.size()-ans;

}

176) Given an unsorted integer array nums, return the smallest missing positive integer.

You must implement an algorithm that runs in O(n) time and uses constant extra space.

**Example 1:**

**Input:** nums = [1,2,0]

**Output:** 3

: thief

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*DEATAILED BIT-MANIPULATION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Bit manipulation is the act of algorithmically manipulating bits or other pieces of data shorter than a word. Computer programming tasks that require bit manipulation include low-level device control, error detection and correction algorithms, data compression, encryption algorithms, and optimization. For most other tasks, modern programming languages allow the programmer to work directly with abstractions instead of bits that represent those abstractions. Source code that does bit manipulation makes use of the bitwise operations: AND, OR, XOR, NOT, and bit shifts.

Bit manipulation, in some cases, can obviate or reduce the need to loop over a data structure and can give many-fold speed ups, as bit manipulations are processed in parallel, but the code can become more difficult to write and maintain.

### **Details**

#### **Basics**

At the heart of bit manipulation are the bit-wise operators & (and), | (or), ~ (not) and ^ (exclusive-or, xor) and shift operators a << b and a >> b.

There is no boolean operator counterpart to bitwise exclusive-or, but there is a simple explanation. The exclusive-or operation takes two inputs and returns a 1 if either one or the other of the inputs is a 1, but not if both are. That is, if both inputs are 1 or both inputs are 0, it returns 0. Bitwise exclusive-or, with the operator of a caret, ^, performs the exclusive-or operation on each pair of bits. Exclusive-or is commonly abbreviated XOR.

* Set union A | B
* Set intersection A & B
* Set subtraction A & ~B
* Set negation ALL\_BITS ^ A or ~A
* Set bit A |= 1 << bit
* Clear bit A &= ~(1 << bit)
* Test bit (A & 1 << bit) != 0
* Extract last bit A&-A or A&~(A-1) or x^(x&(x-1))
* Remove last bit A&(A-1)
* Get all 1-bits ~0

#### **Examples**

Count the number of ones in the binary representation of the given number

int count\_one(int n) {

while(n) {

n = n&(n-1);

count++;

}

return count;

}

Is power of four (actually map-checking, iterative and recursive methods can do the same)

bool isPowerOfFour(int n) {

return !(n&(n-1)) && (n&0x55555555);

//check the 1-bit location;

}

#### **^ tricks**

Use ^ to remove even exactly same numbers and save the odd, or save the distinct bits and remove the same.

##### **Sum of Two Integers**

Use ^ and & to add two integers

int getSum(int a, int b) {

return b==0? a:getSum(a^b, (a&b)<<1); //be careful about the terminating condition;

}

##### **Missing Number**

Given an array containing n distinct numbers taken from 0, 1, 2, ..., n, find the one that is missing from the array. For example, Given nums = [0, 1, 3] return 2. (Of course, you can do this by math.)

int missingNumber(vector<int>& nums) {

int ret = 0;

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

ret ^= i;

ret ^= nums[i];

}

return ret^=nums.size();

}

#### **| tricks**

Keep as many 1-bits as possible

Find the largest power of 2 (most significant bit in binary form), which is less than or equal to the given number N.

long largest\_power(long N) {

//changing all right side bits to 1.

N = N | (N>>1);

N = N | (N>>2);

N = N | (N>>4);

N = N | (N>>8);

N = N | (N>>16);

return (N+1)>>1;

}

##### **Reverse Bits**

Reverse bits of a given 32 bits unsigned integer.

###### **Solution**

uint32\_t reverseBits(uint32\_t n) {

unsigned int mask = 1<<31, res = 0;

for(int i = 0; i < 32; ++i) {

if(n & 1) res |= mask;

mask >>= 1;

n >>= 1;

}

return res;

}

uint32\_t reverseBits(uint32\_t n) {

uint32\_t mask = 1, ret = 0;

for(int i = 0; i < 32; ++i){

ret <<= 1;

if(mask & n) ret |= 1;

mask <<= 1;

}

return ret;

}

#### **& tricks**

Just selecting certain bits

Reversing the bits in integer

x = ((x & 0xaaaaaaaa) >> 1) | ((x & 0x55555555) << 1);

x = ((x & 0xcccccccc) >> 2) | ((x & 0x33333333) << 2);

x = ((x & 0xf0f0f0f0) >> 4) | ((x & 0x0f0f0f0f) << 4);

x = ((x & 0xff00ff00) >> 8) | ((x & 0x00ff00ff) << 8);

x = ((x & 0xffff0000) >> 16) | ((x & 0x0000ffff) << 16);

##### **Bitwise AND of Numbers Range**

Given a range [m, n] where 0 <= m <= n <= 2147483647, return the bitwise AND of all numbers in this range, inclusive. For example, given the range [5, 7], you should return 4.

###### **Solution**

int rangeBitwiseAnd(int m, int n) {

int a = 0;

while(m != n) {

m >>= 1;

n >>= 1;

a++;

}

return m<<a;

}

##### **Number of 1 Bits**

Write a function that takes an unsigned integer and returns the number of ’1' bits it has (also known as the Hamming weight).

###### **Solution**

int hammingWeight(uint32\_t n) {

int count = 0;

while(n) {

n = n&(n-1);

count++;

}

return count;

}

int hammingWeight(uint32\_t n) {

ulong mask = 1;

int count = 0;

for(int i = 0; i < 32; ++i){ //31 will not do, delicate;

if(mask & n) count++;

mask <<= 1;

}

return count;

}

#### **Application**

##### **Repeated DNA Sequences**

All DNA is composed of a series of nucleotides abbreviated as A, C, G, and T, for example: "ACGAATTCCG". When studying DNA, it is sometimes useful to identify repeated sequences within the DNA. Write a function to find all the 10-letter-long sequences (substrings) that occur more than once in a DNA molecule.  
For example,  
Given s = "AAAAACCCCCAAAAACCCCCCAAAAAGGGTTT",  
Return: ["AAAAACCCCC", "CCCCCAAAAA"].

###### **Solution**

class Solution {

public:

vector<string> findRepeatedDnaSequences(string s) {

int sLen = s.length();

vector<string> v;

if(sLen < 11) return v;

char keyMap[1<<21]{0};

int hashKey = 0;

for(int i = 0; i < 9; ++i) hashKey = (hashKey<<2) | (s[i]-'A'+1)%5;

for(int i = 9; i < sLen; ++i) {

if(keyMap[hashKey = ((hashKey<<2)|(s[i]-'A'+1)%5)&0xfffff]++ == 1)

v.push\_back(s.substr(i-9, 10));

}

return v;

}

};

But the above solution can be invalid when repeated sequence appears too many times, in which case we should use unordered\_map<int, int> keyMap to replace char keyMap[1<<21]{0}here.

##### **Majority Element**

Given an array of size n, find the majority element. The majority element is the element that appears more than ⌊ n/2 ⌋ times. (bit-counting as a usual way, but here we actually also can adopt sorting and Moore Voting Algorithm)

###### **Solution**

int majorityElement(vector<int>& nums) {

int len = sizeof(int)\*8, size = nums.size();

int count = 0, mask = 1, ret = 0;

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

count = 0;

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

if(mask & nums[j]) count++;

if(count > size/2) ret |= mask;

mask <<= 1;

}

return ret;

}

##### **Single Number III**

Given an array of integers, every element appears three times except for one. Find that single one. (Still this type can be solved by bit-counting easily.) But we are going to solve it by digital logic design

###### **Solution**

//inspired by logical circuit design and boolean algebra;

//counter - unit of 3;

//current incoming next

//a b c a b

//0 0 0 0 0

//0 1 0 0 1

//1 0 0 1 0

//0 0 1 0 1

//0 1 1 1 0

//1 0 1 0 0

//a = a&~b&~c + ~a&b&c;

//b = ~a&b&~c + ~a&~b&c;

//return a|b since the single number can appear once or twice;

int singleNumber(vector<int>& nums) {

int t = 0, a = 0, b = 0;

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

t = (a&~b&~nums[i]) | (~a&b&nums[i]);

b = (~a&b&~nums[i]) | (~a&~b&nums[i]);

a = t;

}

return a | b;

}

;

##### **Maximum Product of Word Lengths**

Given a string array words, find the maximum value of length(word[i]) \* length(word[j]) where the two words do not share common letters. You may assume that each word will contain only lower case letters. If no such two words exist, return 0.

Example 1:  
Given ["abcw", "baz", "foo", "bar", "xtfn", "abcdef"]  
Return 16  
The two words can be "abcw", "xtfn".

Example 2:  
Given ["a", "ab", "abc", "d", "cd", "bcd", "abcd"]  
Return 4  
The two words can be "ab", "cd".

Example 3:  
Given ["a", "aa", "aaa", "aaaa"]  
Return 0  
No such pair of words.

###### **Solution**

Since we are going to use the length of the word very frequently and we are to compare the letters of two words checking whether they have some letters in common:

* using an array of int to pre-store the length of each word reducing the frequently measuring process;
* since int has 4 bytes, a 32-bit type, and there are only 26 different letters, so we can just use one bit to indicate the existence of the letter in a word.

int maxProduct(vector<string>& words) {

vector<int> mask(words.size());

vector<int> lens(words.size());

for(int i = 0; i < words.size(); ++i) lens[i] = words[i].length();

int result = 0;

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

for (char c : words[i])

mask[i] |= 1 << (c - 'a');

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

if (!(mask[i] & mask[j]))

result = max(result, lens[i]\*lens[j]);

}

return result;

}

#### **Attention**

* result after shifting left(or right) too much is undefined
* right shifting operations on negative values are undefined
* right operand in shifting should be non-negative, otherwise the result is undefined
* The & and | operators have lower precedence than comparison operators

### **Sets**

All the subsets  
A big advantage of bit manipulation is that it is trivial to iterate over all the subsets of an N-element set: every N-bit value represents some subset. Even better, if A is a subset of B then the number representing A is less than that representing B, which is convenient for some dynamic programming solutions.

It is also possible to iterate over all the subsets of a particular subset (represented by a bit pattern), provided that you don’t mind visiting them in reverse order (if this is problematic, put them in a list as they’re generated, then walk the list backwards). The trick is similar to that for finding the lowest bit in a number. If we subtract 1 from a subset, then the lowest set element is cleared, and every lower element is set. However, we only want to set those lower elements that are in the superset. So the iteration step is just i = (i - 1) & superset.

vector<vector<int>> subsets(vector<int>& nums) {

vector<vector<int>> vv;

int size = nums.size();

if(size == 0) return vv;

int num = 1 << size;

vv.resize(num);

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

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

if((1<<j) & i) vv[i].push\_back(nums[j]);

}

return vv;

}

Actually there are two more methods to handle this using recursion and iteration respectively.

### **Bitset**

A [bitset](http://www.cplusplus.com/reference/bitset/bitset/?kw=bitset) stores bits (elements with only two possible values: 0 or 1, true or false, ...).  
The class emulates an array of bool elements, but optimized for space allocation: generally, each element occupies only one bit (which, on most systems, is eight times less than the smallest elemental type: char).

// bitset::count

#include <iostream> // std::cout

#include <string> // std::string

#include <bitset> // std::bitset

int main () {

std::bitset<8> foo (std::string("10110011"));

std::cout << foo << " has ";

std::cout << foo.count() << " ones and ";

std::cout << (foo.size()-foo.count()) << " zeros.\n";

return 0;

}

177) Given two integers a and b, return *the sum of the two integers without using the operators* + *and* -.

**Example 1:**

**Input:** a = 1, b = 2

**Output:** 3

:

int getSum(int a, int b) {

int trigger=a&b;

int ret=a^b;

while(trigger)

{

int ans=(trigger&0xffffffff)<<1;

trigger=ans&ret;

ret=ret^ans;

}

return ret;

}

178) A **self-dividing number** is a number that is divisible by every digit it contains.

* For example, 128 is **a self-dividing number** because 128 % 1 == 0, 128 % 2 == 0, and 128 % 8 == 0.

A **self-dividing number** is not allowed to contain the digit zero.

Given two integers left and right, return *a list of all the****self-dividing numbers****in the range* [left, right].

**Example 1:**

**Input:** left = 1, right = 22

**Output:** [1,2,3,4,5,6,7,8,9,11,12,15,22]

: vector<int> selfDividingNumbers(int left, int right) {

vector<int>v;

for(int i=left;i<=right;i++)

{

int temp=i;

while(temp>0)

{

int digit=temp%10;

if(digit==0||i%digit!=0)

break;

temp/=10;

}

if(temp==0)

v.push\_back(i);

}

return v;

}

179) A [**perfect number**](https://en.wikipedia.org/wiki/Perfect_number) is a **positive integer** that is equal to the sum of its **positive divisors**, excluding the number itself. A **divisor** of an integer x is an integer that can divide x evenly.

Given an integer n, return true*if*n*is a perfect number, otherwise return*false.

**Example 1:**

**Input:** num = 28

**Output:** true

**Explanation:** 28 = 1 + 2 + 4 + 7 + 14

1, 2, 4, 7, and 14 are all divisors of 28.

: bool checkPerfectNumber(int num) {

return (num==6)||(num==28)||(num==496)|(num==8128)|(num==33550336);

}

180) Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**



**Input:** matrix = [[1,2,3],[4,5,6],[7,8,9]]

**Output:** [1,2,3,6,9,8,7,4,5]

: vector<int> spiralOrder(vector<vector<int>>& matrix) {

vector<int>v;

if(matrix.size()==0)

return v;

int n=matrix.size();

int m=matrix[0].size();

int left=0,right=m-1;

int top=0,bottom=n-1;

int direction=1;

while(left<=right&&top<=bottom)

{

if(direction==1)

{

for(int i=left;i<=right;i++)

v.push\_back(matrix[top][i]);

direction=2;

top++;

}

else if(direction==2)

{

for(int i=top;i<=bottom;i++)

v.push\_back(matrix[i][right]);

direction=3;

right--;

}

else if(direction==3)

{

for(int i=right;i>=left;i--)

v.push\_back(matrix[bottom][i]);

direction=4;

bottom--;

}

else if(direction==4)

{

for(int i=bottom;i>=top;i--)

v.push\_back(matrix[i][left]);

direction=1;

left++;

}

}

return v;

}

181) Given n non-negative integers a1, a2, ..., an, where each represents a point at coordinate (i, ai). n vertical lines are drawn such that the two endpoints of the line i is at (i, ai) and (i, 0). Find two lines, which, together with the x-axis forms a container, such that the container contains the most water.

**Notice** that you may not slant the container.

**Example 1:**



**Input:** height = [1,8,6,2,5,4,8,3,7]

**Output:** 49

**Explanation:** The above vertical lines are represented by array [1,8,6,2,5,4,8,3,7]. In this case, the max area of water (blue section) the container can contain is 49

: int maxArea(vector<int>& height) {

int l=0,r=height.size()-1;

int ans=0;

while(l<r)

{

int h=min(height[l],height[r]);

ans=max(ans,(r-l)\*h);

while(height[l]<=h&&l<r)

l++;

while(height[r]<=h&&l<r)

r--;

}

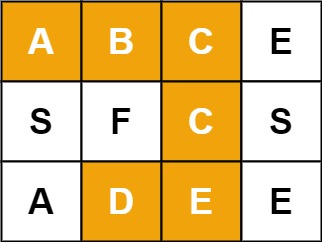
return ans;

}

182) Given an m x n grid of characters board and a string word, return true *if* word *exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

**Example 1:**



**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

: bool func(vector<vector<char>>&v,string word,int i,int j,int ind)

{

if(ind==word.size())

return 1;

if(i<0||j<0||i>v.size()-1||j>v[0].size()-1)

return 0;

if(v[i][j]!=word[ind])

return 0;

v[i][j]='\*';

bool ans=func(v,word,i+1,j,ind+1)||func(v,word,i-1,j,ind+1)||func(v,word,i,j-1,ind+1)||func(v,word,i,j+1,ind+1);

v[i][j]=word[ind];

return ans;

}

bool exist(vector<vector<char>>& board, string word) {

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

{

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

{

if(func(board,word,i,j,0))

return 1;

}

}

return 0;

}

183) Given an integer array nums of length n and an integer target, find three integers in nums such that the sum is closest to target.

Return *the sum of the three integers*.

You may assume that each input would have exactly one solution.

**Example 1:**

**Input:** nums = [-1,2,1,-4], target = 1

**Output:** 2

**Explanation:** The sum that is closest to the target is 2. (-1 + 2 + 1 = 2).

: int threeSumClosest(vector<int>& nums, int target) {

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

int n=nums.size(),ans=nums[0]+nums[1]+nums[2];

for(int i=0;i<n-2;i++)

{

int l=i+1,r=n-1;

while(l<r)

{

int sum=nums[i]+nums[l]+nums[r];

if(abs(ans-target)>abs(sum-target))

ans=sum;

if(ans==target)

break;

if(sum>target)

r--;

else

l++;

}

}

return ans;

}

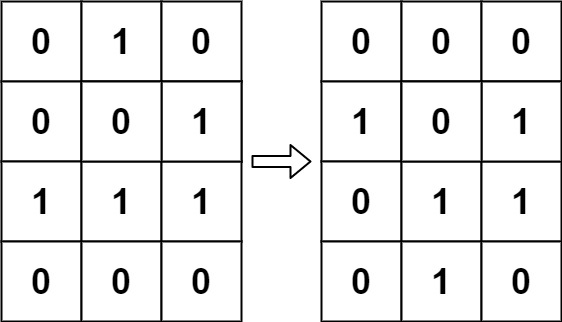
184) According to [Wikipedia's article](https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life): "The **Game of Life**, also known simply as **Life**, is a cellular automaton devised by the British mathematician John Horton Conway in 1970."

The board is made up of an m x n grid of cells, where each cell has an initial state: **live** (represented by a 1) or **dead** (represented by a 0). Each cell interacts with its [eight neighbors](https://en.wikipedia.org/wiki/Moore_neighborhood) (horizontal, vertical, diagonal) using the following four rules (taken from the above Wikipedia article):

1. Any live cell with fewer than two live neighbors dies as if caused by under-population.
2. Any live cell with two or three live neighbors lives on to the next generation.
3. Any live cell with more than three live neighbors dies, as if by over-population.
4. Any dead cell with exactly three live neighbors becomes a live cell, as if by reproduction.

The next state is created by applying the above rules simultaneously to every cell in the current state, where births and deaths occur simultaneously. Given the current state of the m x n grid board, return the next state.

**Example 1:**



**Input:** board = [[0,1,0],[0,0,1],[1,1,1],[0,0,0]]

**Output:** [[0,0,0],[1,0,1],[0,1,1],[0,1,0]]

: void func(vector<vector<int>>&board,int i,int j,int a,int b)

{

int n=board.size();

int m=board[0].size();

if(a>=n||b<0||b>=m)

return;

if(board[i][j]%2!=0)

board[a][b]+=2;

if(board[a][b]%2!=0)

board[i][j]+=2;

}

void gameOfLife(vector<vector<int>>& board) {

if(board.empty())

return;

int n=board.size();

int m=board[0].size();

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

{

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

{

func(board,i,j,i+1,j-1);

func(board,i,j,i+1,j);

func(board,i,j,i+1,j+1);

func(board,i,j,i,j+1);

if(board[i][j]>=5&&board[i][j]<=7)

board[i][j]=1;

else

board[i][j]=0;

}

}

}

184) Given an array of non-negative integers nums, you are initially positioned at the first index of the array.

Each element in the array represents your maximum jump length at that position.

Your goal is to reach the last index in the minimum number of jumps.

You can assume that you can always reach the last index.

**Example 1:**

**Input:** nums = [2,3,1,1,4]

**Output:** 2

**Explanation:** The minimum number of jumps to reach the last index is 2. Jump 1 step from index 0 to 1, then 3 steps to the last index.

: **1. Recursive solution but it will give TLE:**

class Solution {

public:

long long int jump(vector<int> nums , int curr, int dest)

{

if(curr>=dest) return 0;

long long int tmp=INT\_MAX;

//Try Every jump 1 to nums[curr] jump

//and find minimum steps need to reach to end

for(int i=1;i<=nums[curr];i++)

{

tmp=min(tmp,1+jump(nums,curr+i,dest));

}

return tmp;

}

int jump(vector<int>& nums) {

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

}

};

**Time Complexity: O(k^n)**, Where, **k** is max element of nums and **n** is size of nums.  
**Space Complexity: O(1).**  
Because every time inside the recursive function it is calling itself nums[i] times and this is happening for every element (i.e n elements).

**2. Memoization of recursive solution but it may give TLE.**

class Solution {

public:

long long int jump(vector<int> &nums , int curr, int dest,vector<int> &dp)

{

if(curr==dest) return 0;

if(dp[curr]!=-1) return dp[curr];

//cout<<curr<<" ";

long long int tmp=INT\_MAX;

for(int i=1;i<=nums[curr];i++)

{ if(i+curr>dest) break;

tmp=min(tmp,1+jump(nums,curr+i,dest,dp));

}

dp[curr]=tmp;

return tmp;

}

int jump(vector<int>& nums) {

vector<int> dp(nums.size(),-1);

return jump(nums,0,nums.size()-1,dp);

}

};

**Time Complexity: O(k \* n)**, Where, **k** is max element of nums and **n** is size of nums.  
**Space Complexity: O(n).**  
Because every time inside the recursive function it is calling itself nums[i] times and this is happening for every element (i.e n elements). But here we have stored previous calculated result, so there will be no repeation.

**3. O(n) Solution (accepted)**

**We run loop from 0 to size()-1 because we have to reach at last index, think just previous stage when we take jump and reach to last index (size()-1) the we have counted our jump previously and we reach to last index hence we end our journey.**

class Solution {

public:

int jump(vector<int>& nums) {

if(nums.size()<2) return 0; //base case

//initialize jump=1 , we are taking jump from 0th index to the range mxjump

//currjump, we can take jump from particular index

//mxjump , we cango up to maximum

// jump to count no. of jump

int jump=1,n=nums.size(),currjmp=nums[0],mxjmp=nums[0];

int i=0;

//till we reach last index, **NOTE:** Not necessary to cross last index

while(i<n-1)

{

mxjmp=max(mxjmp,i+nums[i]);

if(currjmp==i) //we have to take jump now because our currjump now ends.

{

jump++;//increment in jump

currjmp=mxjmp; //assign new maxjmp to currjmp

}

i++;

}

return jump;

}

};

**Time Complexity: O(n)**, Where, **n** is size of nums.

**185)** You are given an integer array nums. You are initially positioned at the array's **first index**, and each element in the array represents your maximum jump length at that position.

Return true*if you can reach the last index, or*false*otherwise*.

**Example 1:**

**Input:** nums = [2,3,1,1,4]

**Output:** true

**Explanation:** Jump 1 step from index 0 to 1, then 3 steps to the last index.

**: bool canJump(vector<int>& nums) {**

**int dis=0;**

**for(int i=0;i<=dis;i++)**

**{**

**dis=max(dis,i+nums[i]);**

**if(dis>=nums.size()-1)**

**return 1;**

**}**

**return 0;**

**}**

**186)** Given an integer array nums of length n where all the integers of nums are in the range [1, n] and each integer appears **once** or **twice**, return *an array of all the integers that appears****twice***.

You must write an algorithm that runs in O(n) time and uses only constant extra space.

**Example 1:**

**Input:** nums = [4,3,2,7,8,2,3,1]

**Output:** [2,3]

**:** **vector<int> findDuplicates(vector<int>& nums) {**

**vector<int>arr(1000000,0);**

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

**{**

**arr[nums[i]]++;**

**}**

**vector<int>v;**

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

**{**

**if(arr[i]>1)**

**v.push\_back(i);**

**}**

**return v;**

**}**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*ANOTHER APPROACH\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**: vector<int> findDuplicates(vector<int>& nums) {**

**vector<int>v;**

**if(nums.size()==0)**

**return v;**

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

**{**

**if(nums[abs(nums[i])-1]<0)**

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

**nums[abs(nums[i])-1]=-nums[abs(nums[i])-1];**

**}**

**return v;**

**}**

**187)** Given an array nums of n integers where nums[i] is in the range [1, n], return *an array of all the integers in the range* [1, n] *that do not appear in* nums.

**Example 1:**

**Input:** nums = [4,3,2,7,8,2,3,1]

**Output:** [5,6]

**:** **vector<int> findDisappearedNumbers(vector<int>& nums) {**

**vector<int>v;**

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

**{**

**int temp=nums[i];**

**temp=temp>0?temp:-temp;**

**if(nums[temp-1]>0)**

**nums[temp-1]\*=-1;**

**}**

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

**{**

**if(nums[i]>0)**

**v.push\_back(i+1);**

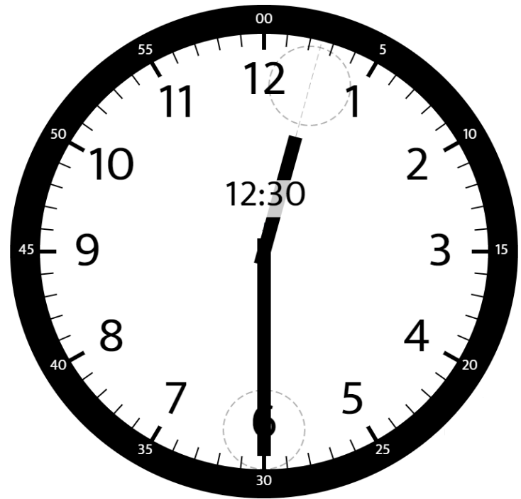
**}**

**return v;**

**}**

**188)** Given two numbers, hour and minutes. Return the smaller angle (in degrees) formed between the hour and the minute hand.

**Example 1:**

**Input:** hour = 12, minutes = 30

**Output:** 16

**: double angleClock(int hour, int minutes) {**

**double minute=minutes\*6;**

**double hr=(hour%12)\*30+minutes\*0.5;**

**double diff=abs(hr-minute);**

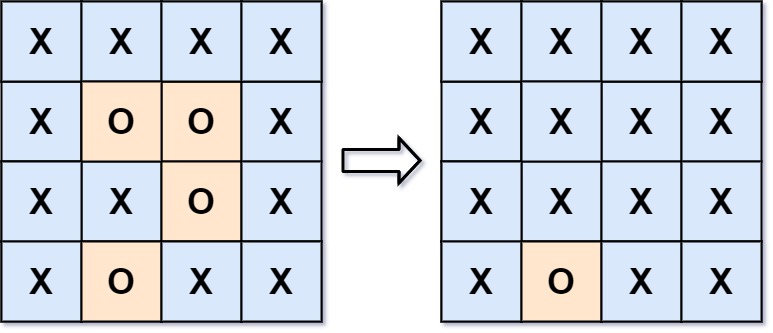
**return min(diff,360-diff);**

**}**

**189)** Given an m x n matrix board containing 'X' and 'O', *capture all regions that are 4-directionally surrounded by* 'X'.

A region is **captured** by flipping all 'O's into 'X's in that surrounded region.

**Example 1:**



**Input:** board = [["X","X","X","X"],["X","O","O","X"],["X","X","O","X"],["X","O","X","X"]]

**Output:** [["X","X","X","X"],["X","X","X","X"],["X","X","X","X"],["X","O","X","X"]]

**Explanation:** Surrounded regions should not be on the border, which means that any 'O' on the border of the board are not flipped to 'X'. Any 'O' that is not on the border and it is not connected to an 'O' on the border will be flipped to 'X'. Two cells are connected if they are adjacent cells connected horizontally or vertically.

**:**

**void dfs(vector<vector<char>>&board,int i,int j)**

**{**

**if(i<0||j<0||i>=board.size()||j>=board[0].size()||board[i][j]!='O')**

**return ;**

**if(board[i][j]=='O')**

**board[i][j]='#';**

**dfs(board,i+1,j);**

**dfs(board,i,j+1);**

**dfs(board,i-1,j);**

**dfs(board,i,j-1);**

**}**

**void solve(vector<vector<char>>& board) {**

**int n=board.size();**

**int m=board[0].size();**

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

**{**

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

**{**

**if((i==0||j==0||i==n-1||j==m-1)&&board[i][j]=='O')**

**dfs(board,i,j);**

**}**

**}**

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

**{**

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

**{**

**if(board[i][j]=='O')**

**board[i][j]='X';**

**if(board[i][j]=='#')**

**board[i][j]='O';**

**}**

**}**

**}**

**190)** You are given a list of songs where the ith song has a duration of time[i] seconds.

Return *the number of pairs of songs for which their total duration in seconds is divisible by* 60. Formally, we want the number of indices i, j such that i < j with (time[i] + time[j]) % 60 == 0.

**Example 1:**

**Input:** time = [30,20,150,100,40]

**Output:** 3

**Explanation:** Three pairs have a total duration divisible by 60:

(time[0] = 30, time[2] = 150): total duration 180

(time[1] = 20, time[3] = 100): total duration 120

(time[1] = 20, time[4] = 40): total duration 60

**: int numPairsDivisibleBy60(vector<int>& time) {**

**int cnt=0;**

**vector<int>v(60);**

**for(auto it:time)**

**{**

**cnt+=v[(60-it%60)%60];**

**v[it%60]+=1;**

**}**

**return cnt;**

**}**

**191)** Given an array nums of n integers, return *an array of all the****unique****quadruplets* [nums[a], nums[b], nums[c], nums[d]] such that:

* 0 <= a, b, c, d < n
* a, b, c, and d are **distinct**.
* nums[a] + nums[b] + nums[c] + nums[d] == target

You may return the answer in **any order**.

**Example 1:**

**Input:** nums = [1,0,-1,0,-2,2], target = 0

**Output:** [[-2,-1,1,2],[-2,0,0,2],[-1,0,0,1]]

**:** **vector<vector<int>> fourSum(vector<int>& nums, int target) {**

**sort(nums.begin(),nums.end());**

**int n=nums.size();**

**vector<vector<int>>v;**

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

**{**

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

**{**

**int l=j+1,r=n-1;**

**int sum=target-nums[i]-nums[j];**

**while(l<r)**

**{**

**if(nums[l]+nums[r]==sum)**

**{**

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

**l++,r--;**

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

**l++;**

**}**

**else if(nums[l]+nums[r]>sum)**

**r--;**

**else**

**l++;**

**}**

**while(j+1<n&&nums[j+1]==nums[j])**

**j++;**

**}**

**while(i+1<n&&nums[i+1]==nums[i])**

**i++;**

**}**

**return v;**

**}**

**192)** There are several cards **arranged in a row**, and each card has an associated number of points. The points are given in the integer array cardPoints.

In one step, you can take one card from the beginning or from the end of the row. You have to take exactly k cards.

Your score is the sum of the points of the cards you have taken.

Given the integer array cardPoints and the integer k, return the *maximum score* you can obtain.

**Example 1:**

**Input:** cardPoints = [1,2,3,4,5,6,1], k = 3

**Output:** 12

**Explanation:** After the first step, your score will always be 1. However, choosing the rightmost card first will maximize your total score. The optimal strategy is to take the three cards on the right, giving a final score of 1 + 6 + 5 = 12.

**: int maxScore(vector<int>& cardPoints, int k) {**

**int n=cardPoints.size();**

**vector<int>v(n+1);**

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

**v[i]=v[i-1]+cardPoints[i-1];**

**int ans=0;**

**for(int i=0;i<=k;i++)**

**ans=max(ans,v[i]+v[n]-v[n-k+i]);**

**return ans;**

**}**

**193)** Given an array **arr[]** denoting heights of **N** towers and a positive integer **K**, you **have to** modify the height of each tower either by increasing or decreasing them by **K** only **once**. After modifying, height should be a **non-negative** integer.   
Find out what could be the possible minimum difference of the height of shortest and longest towers after you have modified each tower.

A slight modification of the problem can be found [here](https://practice.geeksforgeeks.org/problems/minimize-the-heights-i/1/).

**Example 1:**

**Input:**

K = 2, N = 4

Arr[] = {1, 5, 8, 10}

**Output:**

5

**Explanation:**

The array can be modified as

{3, 3, 6, 8}. The difference between

the largest and the smallest is 8-3 = 5.

**:** **int getMinDiff(int arr[], int n, int k) {**

**// code here**

**sort(arr,arr+n);**

**int diff=arr[n-1]-arr[0];**

**int maxi,mini;**

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

**if(arr[i+1]<k){**

**continue;**

**}**

**else{**

**mini=min(arr[0]+k,arr[i+1]-k);**

**maxi=max(arr[i]+k,arr[n-1]-k);**

**diff=min(diff,maxi-mini);**

**}**

**}**

**return diff;**

**}**

**194)** Given two non-negative integers, num1 and num2 represented as string, return *the sum of* num1 *and* num2 *as a string*.

You must solve the problem without using any built-in library for handling large integers (such as BigInteger). You must also not convert the inputs to integers directly.

**Example 1:**

**Input:** num1 = "11", num2 = "123"

**Output:** "134"

**: string addStrings(string num1, string num2) {**

**int n=num1.size()-1;**

**int m=num2.size()-1;**

**int carry=0;**

**string ans="";**

**while(n>=0||m>=0||carry)**

**{**

**if(n>=0)**

**carry+=num1[n--]-'0';**

**if(m>=0)**

**carry+=num2[m--]-'0';**

**ans=to\_string(carry%10)+ans;**

**carry/=10;**

**}**

**return carry?"1"+ans:ans;**

**}**

**195)** Given a signed 32-bit integer x, return x*with its digits reversed*. If reversing x causes the value to go outside the signed 32-bit integer range [-231, 231 - 1], then return 0.

**Assume the environment does not allow you to store 64-bit integers (signed or unsigned).**

**Example 1:**

**Input:** x = 123

**Output:** 321

**: int reverse(int x) {**

**long long ans=0;**

**while(x)**

**{**

**ans=ans\*10+x%10;**

**x/=10;**

**}**

**return (ans<INT\_MIN||ans>INT\_MAX)?0:ans;;**

**}**

**CHECK IF NUMBER IS PALINDROME OR NOT?**

**:N=121 THEN CALCULATE ANS=ANS\*10+N%10; N/=10;**

**THEN CALCULATE 121-121(ANS) =>IF IT IS 0 THE N IS PALINDROME OTHERWISE NOT A PALINDROME**

**196)** Given two binary strings a and b, return *their sum as a binary string*.

**Example 1:**

**Input:** a = "11", b = "1"

**Output:** "100"

**:** **string addBinary(string a, string b) {**

**string s;**

**int n=a.size()-1;**

**int m=b.size()-1;**

**int carry=0;**

**while(n>=0||m>=0||carry==1)**

**{**

**carry+=n>=0?a[n--]-'0':0;**

**carry+=m>=0?b[m--]-'0':0;**

**s=char(carry%2+'0')+s;**

**carry/=2;**

**}**

**return s;**

**}**

**197)** Given an integer x, return true if x is palindrome integer.

An integer is a **palindrome** when it reads the same backward as forward. For example, 121 is palindrome while 123 is not.

**Example 1:**

**Input:** x = 121

**Output:** true

**: bool isPalindrome(int x) {**

**string s=to\_string(x);**

**if(x<0)**

**return 0;**

**int i=0;**

**int j=s.size()-1;**

**while(i<=j)**

**{**

**if(s[i]!=s[j])**

**return 0;**

**i++;**

**j--;**

**}**

**return 1;**

**}**

**198)** Given an integer array nums of size n, return *the minimum number of moves required to make all array elements equal*.

In one move, you can increment n - 1 elements of the array by 1.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** 3

**Explanation:** Only three moves are needed (remember each move increments two elements):

[1,2,3] => [2,3,3] => [3,4,3] => [4,4,4]

**: int minMoves(vector<int>& nums) {**

**int moves=0;**

**int minval=\*min\_element(nums.begin(),nums.end());**

**for(auto it:nums)**

**moves+=it-minval;**

**return moves;**

**}**

**199)** Write an algorithm to determine if a number n is happy.

A **happy number** is a number defined by the following process:

* Starting with any positive integer, replace the number by the sum of the squares of its digits.
* Repeat the process until the number equals 1 (where it will stay), or it **loops endlessly in a cycle** which does not include 1.
* Those numbers for which this process **ends in 1** are happy.

Return true *if* n *is a happy number, and* false *if not*.

**Example 1:**

**Input:** n = 19

**Output:** true

**Explanation:**

12 + 92 = 82

82 + 22 = 68

62 + 82 = 100

12 + 02 + 02 = 1

**: int next(int n)**

**{**

**int sum=0;**

**while(n!=0)**

**{**

**sum+=pow(n%10,2);**

**n/=10;**

**}**

**return sum;**

**}**

**bool isHappy(int n) {**

**int slow=next(n);**

**int fast=next(next(n));**

**while(slow!=fast)**

**{**

**slow=next(slow);**

**fast=next(next(fast));**

**}**

**return fast==1;**

**}**

**200)** Given an integer array nums, *find three numbers whose product is maximum and return the maximum product*.

**Example 1:**

**Input:** nums = [1,2,3]

**Output:** 6

**: int maximumProduct(vector<int>& nums) {**

**int n=nums.size();**

**sort(nums.begin(),nums.end());**

**int temp1=nums[0]\*nums[1]\*nums[n-1];**

**int temp2=nums[n-1]\*nums[n-2]\*nums[n-3];**

**return temp1>temp2?temp1:temp2;**

**}**

**201)** Given an integer columnNumber, return *its corresponding column title as it appears in an Excel sheet*.

For example:

A -> 1

B -> 2

C -> 3

...

Z -> 26

AA -> 27

AB -> 28

...

**Example 1:**

**Input:** columnNumber = 1

**Output:** "A"

**Example 2:**

**Input:** columnNumber = 28

**Output:** "AB"

**Example 3:**

**Input:** columnNumber = 701

**Output:** "ZY"

**Example 4:**

**Input:** columnNumber = 2147483647

**Output:** "FXSHRXW"

**:** **string convertToTitle(int columnNumber) {**

**int n=columnNumber;**

**return n==0?"":convertToTitle((n-1)/26)+(char)((n-1)%26+'A');**

**}**

**202)** You are given a binary array nums (**0-indexed**).

We define xi as the number whose binary representation is the subarray nums[0..i] (from most-significant-bit to least-significant-bit).

* For example, if nums = [1,0,1], then x0 = 1, x1 = 2, and x2 = 5.

Return *an array of booleans*answer*where*answer[i]*is*true*if*xi*is divisible by*5.

**Example 1:**

**Input:** nums = [0,1,1]

**Output:** [true,false,false]

**Explanation:** The input numbers in binary are 0, 01, 011; which are 0, 1, and 3 in base-10.

Only the first number is divisible by 5, so answer[0] is true.

**: vector<bool> prefixesDivBy5(vector<int>& nums) {**

**vector<bool>v;**

**int ans=0;**

**for(auto it:nums)**

**{**

**ans=((ans\*2)+it)%10;**

**v.push\_back(ans%5==0?1:0);**

**}**

**return v;**

**}**

**203)** There are n gas stations along a circular route, where the amount of gas at the ith station is gas[i].

You have a car with an unlimited gas tank and it costs cost[i] of gas to travel from the ith station to its next (i + 1)th station. You begin the journey with an empty tank at one of the gas stations.

Given two integer arrays gas and cost, return *the starting gas station's index if you can travel around the circuit once in the clockwise direction, otherwise return* -1. If there exists a solution, it is **guaranteed** to be **unique**

**Example 1:**

**Input:** gas = [1,2,3,4,5], cost = [3,4,5,1,2]

**Output:** 3

**Explanation:**

Start at station 3 (index 3) and fill up with 4 unit of gas. Your tank = 0 + 4 = 4

Travel to station 4. Your tank = 4 - 1 + 5 = 8

Travel to station 0. Your tank = 8 - 2 + 1 = 7

Travel to station 1. Your tank = 7 - 3 + 2 = 6

Travel to station 2. Your tank = 6 - 4 + 3 = 5

Travel to station 3. The cost is 5. Your gas is just enough to travel back to station 3.

Therefore, return 3 as the starting index.

**: int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {**

**int start=gas.size()-1;**

**int end=0;**

**int sum=gas[start]-cost[start];**

**while(start>end)**

**{**

**if(sum>=0)**

**{**

**sum+=gas[end]-cost[end];**

**end++;**

**}**

**else**

**{**

**--start;**

**sum+=gas[start]-cost[start];**

**}**

**}**

**return sum>=0?start:-1;**

**}**

**204)** A string is **good** if there are no repeated characters.

Given a string s​​​​​, return *the number of****good substrings****of length****three****in*s​​​​​​.

Note that if there are multiple occurrences of the same substring, every occurrence should be counted.

A **substring** is a contiguous sequence of characters in a string.

**Example 1:**

**Input:** s = "xyzzaz"

**Output:** 1

**Explanation:** There are 4 substrings of size 3: "xyz", "yzz", "zza", and "zaz".

The only good substring of length 3 is "xyz".

**:** **int countGoodSubstrings(string s) {**

**int ans=0;**

**if(s.size()<3)**

**return 0;**

**for(int i=0;i<s.size()-2;i++)**

**{**

**if((s[i]!=s[i+1])&&(s[i]!=s[i+2])&&(s[i+1]!=s[i+2]))**

**ans++;**

**}**

**return ans;**

**}**

**205)** The **pair sum** of a pair (a,b) is equal to a + b. The **maximum pair sum** is the largest **pair sum** in a list of pairs.

* For example, if we have pairs (1,5), (2,3), and (4,4), the **maximum pair sum** would be max(1+5, 2+3, 4+4) = max(6, 5, 8) = 8.

Given an array nums of **even** length n, pair up the elements of nums into n / 2 pairs such that:

* Each element of nums is in **exactly one** pair, and
* The **maximum pair sum**is **minimized**.

Return *the minimized****maximum pair sum****after optimally pairing up the elements*.

**Example 1:**

**Input:** nums = [3,5,2,3]

**Output:** 7

**Explanation:** The elements can be paired up into pairs (3,3) and (5,2).

The maximum pair sum is max(3+3, 5+2) = max(6, 7) = 7.

**:** **int minPairSum(vector<int>& nums) {**

**sort(nums.begin(),nums.end());**

**vector<int>v(nums.size()/2);**

**int i=0,j=nums.size()-1;**

**int ans=0;**

**int maxi=INT\_MIN;**

**while(i<j)**

**{**

**ans=nums[i]+nums[j];**

**maxi=max(maxi,ans);**

**i++;**

**j--;**

**}**

**return maxi;**

**}**

**206)** Given an integer array nums, move all 0's to the end of it while maintaining the relative order of the non-zero elements.

**Note** that you must do this in-place without making a copy of the array.

**Example 1:**

**Input:** nums = [0,1,0,3,12]

**Output:** [1,3,12,0,0]

**: void moveZeroes(vector<int>& nums) {**

**for(int last\_zero=0,curr=0;curr<nums.size();curr++)**

**{**

**if(nums[curr]!=0)**

**swap(nums[last\_zero++],nums[curr]);**

**}**

**}**

**207)** Given an integer array nums and an integer val, remove all occurrences of val in nums [**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm). The relative order of the elements may be changed.

Since it is impossible to change the length of the array in some languages, you must instead have the result be placed in the **first part** of the array nums. More formally, if there are k elements after removing the duplicates, then the first k elements of nums should hold the final result. It does not matter what you leave beyond the first k elements.

Return k after placing the final result in the first k slots of nums.

Do **not** allocate extra space for another array. You must do this by **modifying the input array**[**in-place**](https://en.wikipedia.org/wiki/In-place_algorithm) with O(1) extra memory.

**Custom Judge:**

The judge will test your solution with the following code:

int[] nums = [...]; // Input array

int val = ...; // Value to remove

int[] expectedNums = [...]; // The expected answer with correct length.

// It is sorted with no values equaling val.

int k = removeElement(nums, val); // Calls your implementation

assert k == expectedNums.length;

sort(nums, 0, k); // Sort the first k elements of nums

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

assert nums[i] == expectedNums[i];

}

If all assertions pass, then your solution will be **accepted**.

**Example 1:**

**Input:** nums = [3,2,2,3], val = 3

**Output:** 2, nums = [2,2,\_,\_]

**Explanation:** Your function should return k = 2, with the first two elements of nums being 2.

It does not matter what you leave beyond the returned k (hence they are underscores).

**:** **int removeElement(vector<int>& nums, int val) {**

**int cnt=0;**

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

**{**

**if(nums[i]==val)**

**cnt++;**

**else**

**nums[i-cnt]=nums[i];**

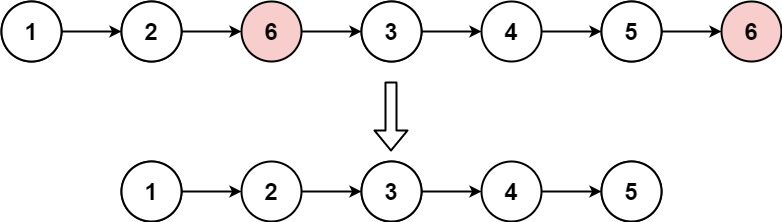
**}**

**return nums.size()-cnt;**

**}**

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

**Example 1:**



**Input:** head = [1,2,6,3,4,5,6], val = 6

**Output:** [1,2,3,4,5]

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

**if(head==NULL)**

**return NULL;**

**head->next=removeElements(head->next,val);**

**return head->val==val?head->next:head;**

**}**

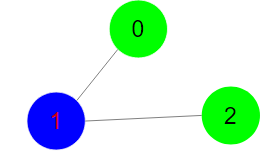
**GRAPH REMAINING**

**BIPARTITE=>THAT CAN BE COLOURED WITH 2 COLORS HAVING NO ADJACENT NODES SAME (OF SAME COLOR).**

**209)** Given an adjacency list of a graph**adj**of V no. of vertices having 0 based index. Check whether the graph is bipartite or not.

**Example 1:**

**Input:**

****

**Output:** 1

**Explanation:** The given graph can be colored

in two colors so, it is a bipartite graph.

**\*\*\*\*\*\*\*\*\*\*\*\*\*DFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**:** **bool dfs(int node, vector<int> adj[], int color[]){**

**for(auto it: adj[node]){**

**if(color[it]==-1){**

**color[it] =~color[node];**

**if(!dfs(it, adj, color)){**

**return false;**

**}**

**} else if(color[it]==color[node]){**

**return false;**

**}**

**}**

**return true;**

**}**

**bool isBipartite(int V, vector<int>adj[]){**

**// Code here**

**int color[V];**

**memset(color, -1, sizeof color);**

**for(int i=0; i<V; i++){**

**if(color[i]==-1){**

**color[i]=1;**

**if(!dfs(i, adj, color)){**

**return false;**

**}**

**}**

**}**

**return true;**

**}**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*BFS IMPLEMENTATION\*\*\*\*\*\*\*\*\*\*\*\***

**bool BFS(int node,vector<int>&vis,vector<int>adj[])**

**{**

**queue<int>q;**

**q.push(node);**

**vis[node]=1;**

**while(!q.empty())**

**{**

**int temp=q.front();**

**q.pop();**

**for(auto it:adj[temp])**

**{**

**if(vis[it]==-1)**

**{**

**vis[it]=~vis[temp];**

**q.push(it);**

**}**

**else if(vis[it]==vis[temp])**

**return 0;**

**}**

**}**

**return 1;**

**}**

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

**{**

**vector<int>vis(V,-1);**

**for(int i=0;i<V;i++)**

**{**

**if(vis[i]==-1)**

**{**

**if(!BFS(i,vis,adj))**

**return 0;**

**}**

**}**

**return 1;**

**}**

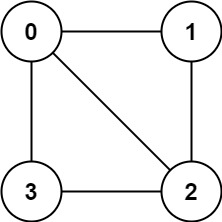
**210)** There is an **undirected** graph with n nodes, where each node is numbered between 0 and n - 1. You are given a 2D array graph, where graph[u] is an array of nodes that node u is adjacent to. More formally, for each v in graph[u], there is an undirected edge between node u and node v. The graph has the following properties:

* There are no self-edges (graph[u] does not contain u).
* There are no parallel edges (graph[u] does not contain duplicate values).
* If v is in graph[u], then u is in graph[v] (the graph is undirected).
* The graph may not be connected, meaning there may be two nodes u and v such that there is no path between them.

A graph is **bipartite** if the nodes can be partitioned into two independent sets A and B such that **every** edge in the graph connects a node in set A and a node in set B.

Return true*if and only if it is****bipartite***.

**Example 1:**



**Input:** graph = [[1,2,3],[0,2],[0,1,3],[0,2]]

**Output:** false

**Explanation:** There is no way to partition the nodes into two independent sets such that every edge connects a node in one and a node in the other.

**: bool bipartite(int node,vector<vector<int>>v,vector<int>&vis)**

**{**

**queue<int>q;**

**q.push(node);**

**vis[node]=1;**

**while(!q.empty())**

**{**

**int temp=q.front();**

**q.pop();**

**for(auto it:v[temp])**

**{**

**if(vis[it]==-1)**

**{**

**vis[it]=~vis[temp];**

**q.push(it);**

**}**

**else if(vis[it]==vis[temp])**

**return 0;**

**}**

**}**

**return 1;**

**}**

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

**vector<int>vis(graph.size(),-1);**

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

**{**

**if(vis[i]==-1)**

**{**

**if(!bipartite(i,graph,vis))**

**return 0;**

**}**

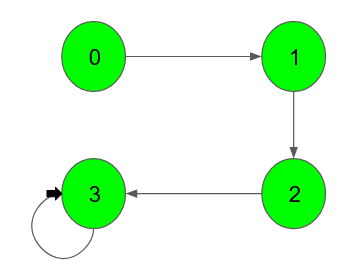
**}**

**return 1;**

**}**

**211)** Given a Directed Graph with **V** vertices (Numbered from **0** to **V-1**) and **E** edges, check whether it contains any cycle or not.  
**Example 1:**

**Input:**

****

**Output:** 1

**Explanation**: 3 -> 3 is a cycle

**:** **bool cycle(int node,vector<int>&vis,vector<int>&dfsvis,vector<int>adj[])**

**{**

**vis[node]=1;**

**dfsvis[node]=1;**

**for(auto it:adj[node])**

**{**

**if(!vis[it])**

**{**

**if(cycle(it,vis,dfsvis,adj))**

**return 1;**

**}**

**else if(dfsvis[it])**

**return 1;**

**}**

**dfsvis[node]=0;**

**return 0;**

**}**

**bool isCyclic(int V, vector<int> adj[]) {**

**// code here**

**vector<int>vis(V,0);**

**vector<int>dfsvis(V,0);**

**for(int i=0;i<V;i++)**

**{**

**if(!vis[i])**

**{**

**if(cycle(i,vis,dfsvis,adj))**

**return 1;**

**}**

**}**

**return 0;**

**}**

**\*\*\*\*\*\*\*\*\*\*\*ASKED IN CAMPUS RECRUITMENT\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

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

**Example 1:**

**Input:** s = "babad"

**Output:** "bab"

**Note:** "aba" is also a valid answer.

**: int expandfromcentre(string&s,int l,int r)**

**{**

**if(s.size()<1||left>right)**

**return 0;**

**while(l>=0&&r<s.size()&&s[l]==s[r])**

**{**

**r++;**

**l--;**

**}**

**return r-l-1;**

**}**

**string longestPalindrome(string s) {**

**string ans;**

**int len=0;**

**int start=0,end=0;**

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

**{**

**int len1=expandfromcentre(s,i,i);**

**int len2=expandfromcentre(s,i,i+1);**

**if(len<max(len1,len2))**

**{**

**len=max(len1,len2);**

**start=i-(len-1)/2;**

**}**

**}**

**return s.substr(start,len);**

**}**

**Topological Sort(only for directed acyclic graph[DAG])**

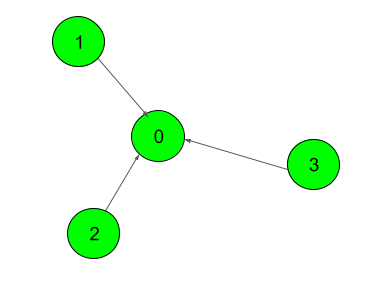
Topological sort is an ordering of vertices in a directed acyclic graph [DAG] in which each node comes before all nodes to which it has outgoing edges. As an example, consider the course prerequisite structure at universities. A directed edge (v ,w) indicates that course v must be completed before course w. Topological ordering for this example is the sequence which does not violate the prerequisite requirement. Every DAG may have one or more topological orderings. Topological sort is not possible if the graph has a cycle, since for two vertices v and w on the cycle, v precedes w and w precedes v. Topological sort has an interesting property. All pairs of consecutive vertices in the sorted order are connected by edges; then these edges form a directed Hamiltonian path [refer to Problems Section] in the DAG. If a Hamiltonian path exists, the topological sort order is unique. If a topological sort does not form a Hamiltonian path, DAG can have two or more topological orderings.

SIMPLY TOPO SORT MEANS LINEAR ORDERING OF VERTICES SUCH THAT IF THERE IS AN EDGE FROM U TO V

THEN U HAS TO APPEAR BEFORE V

**213)** Given a Directed Acyclic Graph (DAG) with V vertices and E edges, Find any Topological Sorting of that Graph.

**Example 1:**

**Input: **

**Output:**

1

**Explanation**:

The output 1 denotes that the order is

valid. So, if you have, implemented

your function correctly, then output

would be 1 for all test cases.

One possible Topological order for the

graph is 3, 2, 1, 0.

**\*\*\*\*\*\*\*USING DFS\*\*\*\*\*\*\*\*\*\*\***

**:** **void DFS(int node,vector<int>&vis,stack<int>&st,vector<int>adj[])**

**{**

**vis[node]=1;**

**for(auto it:adj[node])**

**{**

**if(!vis[it])**

**{**

**DFS(it,vis,st,adj);**

**vis[it]=1;**

**}**

**}**

**st.push(node);**

**}**

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

**{**

**// code here**

**vector<int>vis(V,0);**

**stack<int>st;**

**for(int i=0;i<V;i++)**

**{**

**if(!vis[i])**

**DFS(i,vis,st,adj);**

**}**

**vector<int>topo;**

**while(!st.empty())**

**{**

**int k=st.top();**

**st.pop();**

**topo.push\_back(k);**

**}**

**return topo;**

**}**

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*USING BFS(KAHN’S ALGO)\*\*\*\*\*\*\*\***

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

**{**

**// code here**

**queue<int>q;**

**vector<int>indegree(V,0);**

**for(int i=0;i<V;i++)**

**{**

**for(auto it:adj[i])**

**{**

**indegree[it]++;**

**}**

**}**

**for(int i=0;i<V;i++)**

**{**

**if(indegree[i]==0)**

**q.push(i);**

**}**

**vector<int>topo;**

**while(!q.empty())**

**{**

**int node=q.front();**

**q.pop();**

**topo.push\_back(node);**

**for(auto it:adj[node])**

**{**

**indegree[it]--;**

**if(indegree[it]==0)**

**q.push(it);**

**}**

**}**

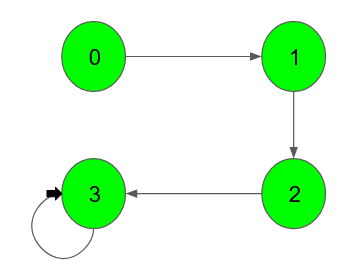
**return topo;**

**}**

**214)** Given a Directed Graph with **V** vertices (Numbered from **0** to **V-1**) and **E** edges, check whether it contains any cycle or not.

**Example 1:**

**Input:**

****

**Output:** 1

**Explanation**: 3 -> 3 is a cycle

**\*\*\*\*\*\*\*\*\*\*\*USING BFS(KAHN’S ALGO)\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**:** **bool isCyclic(int V, vector<int> adj[]) {**

**// code here**

**queue<int>q;**

**vector<int>indegree(V,0);**

**for(int i=0;i<V;i++)**

**{**

**for(auto it:adj[i])**

**{**

**indegree[it]++;**

**}**

**}**

**for(int i=0;i<V;i++)**

**{**

**if(indegree[i]==0)**

**q.push(i);**

**}**

**int cnt=0;**

**while(!q.empty())**

**{**

**int node=q.front();**

**q.pop();**

**cnt++;**

**for(auto it:adj[node])**

**{**

**indegree[it]--;**

**if(indegree[it]==0)**

**q.push(it);**

**}**

**}**

**if(cnt==V)**

**return 0;**

**return 1;**

**}**

**215)shortest distance from given source to any of the other node having no weigths on edges(so assume it as unit weight[1]).**

**Shortest path in unweighted graph(Acyclic)**

**:** **void bfs(vector<int>adj[],int V,int src)**

**{**

**vector<int>dis(V,INT\_MAX);**

**queue<int>q;**

**dis[src]=0;**

**q.push(src);**

**while(!q.empty())**

**{**

**int node=q.front();**

**q.pop();**

**for(auto it:adj[node])**

**{**

**If(dis[it]<dis[node]+1;**

**{**

**dis[it]=dis[node]+1;**

**q.push(it);**

**}**

**}**

**}**

**for(int i=0;i<V;i++)**

**cout<<dis[i]<<" ";**

**}**

**216)** Consider a directed graph whose vertices are numbered from 1 to n. There is an edge from a vertex i to a vertex j iff either j = i + 1 or j = 3 \* i. The task is to find the minimum number of edges in a path in G from vertex 1 to vertex n.

**Example 1:**

**Input:**

N = 9

**Output:**

2

**Explanation**:

9 -> 3 -> 1, so

number of steps are 2.

**:** **int minimumStep(int n){**

**//complete the function here**

**int count = 0;**

**while(n>1){**

**if(n%3==0){**

**n/=3;**

**}else{**

**n--;**

**}count++;**

**}return count;**

**}**

**217)Cycle in DAG(USING TOPO SORT)**

**#include<bits/stdc++.h>**

**using namespace std;**

**void toposort(int node,stack<int>&st,vector<pair<int,int>>adj[],vector<int>&vis)**

**{**

**vis[node]=1;**

**for(auto it:adj[node])**

**{**

**if(!vis[it.first])**

**{**

**toposort(it.first,st,adj,vis);**

**vis[it.first]=1;**

**}**

**}**

**st.push(node);**

**}**

**void shortestpath(int src,int n,vector<pair<int,int>>adj[])**

**{**

**vector<int>vis(n,0);**

**stack<int>st;**

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

**{**

**if(!vis[i])**

**{**

**toposort(i,st,adj,vis);**

**}**

**}**

**vector<int>dis(n,INT\_MAX);**

**dis[src]=0;**

**while(!st.empty())**

**{**

**int node=st.top();**

**st.pop();**

**if(dis[node]!=INT\_MAX)**

**{**

**for(auto it:adj[node])**

**{**

**dis[it.first]=min(it.first,dis[node]+it.second);**

**}**

**}**

**}**

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

**{**

**dis[i]==INT\_MAX?cout<<"inf"<<" ":cout<<dis[i]<<" ";**

**}**

**}**

**int main()**

**{**

**int n,m;**

**cin>>n>>m;**

**vector<pair<int,int>>adj[n];**

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

**{**

**int u,v,wt;**

**cin>>u>>v>>wt;**

**adj[u].push\_back({v,wt});**

**}**

**shortestpath(0,n,adj);**

**return 0;**

**}**

**218)** Given a weighted, undirected and connected graph of V vertices and E edges, Find the shortest distance of all the vertex's from the source vertex S.  
**Note:**The Graph doesn't contain any negative weight cycle.

**Example 1:**

**Input:**

**S** = 0

**Output:**

0 9

**Explanation**:

The source vertex is 0. Hence, the shortest

distance of node 0 is 0 and the shortest

distance from node 9 is 9 - 0 = 9.

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

**{**

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

**vector<int>distTo(V,INT\_MAX);**

**distTo[S]=0;**

**pq.push(make\_pair(0,S));**

**while(!pq.empty())**

**{**

**int dist=pq.top().first;**

**int prev=pq.top().second;**

**pq.pop();**

**for(auto ele:adj[prev])**

**{**

**int next=ele[0];**

**int nextDist=ele[1];**

**if(distTo[next]>distTo[prev]+nextDist)**

**{**

**distTo[next]=distTo[prev]+nextDist;**

**pq.push(make\_pair(distTo[next],next));**

**}**

**}**

**}**

**return distTo;**

**}**

**219)** Given a value N, find the number of ways to make change for N cents, if we have infinite supply of each of S = { S1, S2, .. , SM} valued coins.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*FLIPCART SDE INTERVIEW EXPERIENCE\*\*\*\*\*\*\*\*\*\*\*\*\*\*  
**Example 1:**

**Input:**

n = 4 , m = 3

S[] = {1,2,3}

**Output:** 4

**Explanation**: Four Possible ways are:

{1,1,1,1},{1,1,2},{2,2},{1,3}.

**:** **long long int count(int S[], int m, int n){**

**long long int dp[m+1][n+1];**

**for(int i=0; i<m+1; i++){**

**for(int j=0; j<n+1; j++){**

**if(j==0) dp[i][j]=1;**

**else if(i==0) dp[i][j]=0;**

**else if(S[i-1]<=j) dp[i][j]=dp[i-1][j]+dp[i][j-S[i-1]];**

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

**}**

**}**

**return dp[m][n];**

**}**

**220)** Given a number n, find the value of below expression:  
            f(n-1)\*f(n+1) - f(n)\*f(n)    where f(n) is nth Fibonacci. 

**Example 1:**

**Input :** n = 5

**Output:** -1

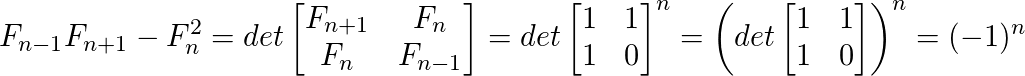
**Explanation:** f(4) = 3, f(5) = 5

f(6) = 8. 3 \* 8 - 5 \* 5 = -1

**:** Although the task is simple i.e. find n-1th, nth and (n+1)-th Fibonacci numbers. Evaluate the expression and display the result. But this can be done in O(1) time using [Cassini’s Identity](https://en.wikipedia.org/wiki/Cassini_and_Catalan_identities) which states that:

f(n-1)\*f(n+1) - f(n\*n) = (-1)^n

So, we don’t need to calculate any Fibonacci term,the only thing is to check whether n is even or odd.

**How does above formula work?**  
The formula is based on matrix representation of Fibonacci numbers.  
[](https://media.geeksforgeeks.org/wp-content/uploads/cassini-identity.png)

**:** **int evaluate\_exp(int n){**

**// Code here**

**return (n&1)?-1:1;**

**}**

**221)** You are given a string s. You need to reverse the string.

**Example 1:**

**Input:**

s = Geeks

**Output:** skeeG

**\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*USING RECURSION\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***

**:** **void func(string &s,int i)**

**{**

**if(i>=s.size()/2)**

**return ;**

**swap(s[i],s[s.size()-i-1]);**

**func(s,i+1);**

**}**

**string reverseWord(string str)**

**{**

**//Your code here**

**func(str,0);**

**return str;**

**}**

**222)** Consider a rat placed at **(0, 0)** in a square matrixof 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 rat cannot move to it while value 1 at a cell in the matrix represents that rat can be travel through it.  
**Note**: In a path, no cell can be visited more than one time.

**Example 1:**

**Input**:

N = 4

m[][] = {{1, 0, 0, 0},

{1, 1, 0, 1},

{1, 1, 0, 0},

{0, 1, 1, 1}}

**Output:**

DDRDRR DRDDRR

**Explanation**:

The rat can reach the destination at

(3, 3) from (0, 0) by two paths - DRDDRR

and DDRDRR, when printed in sorted order

we get DDRDRR DRDDRR.

**:** **void solve(vector<vector<int>> &m,vector<string> &ans,int i,int j,int n,string &str) {**

**if(i==n-1 && j==n-1) {**

**ans.push\_back(str);**

**return;**

**}**

**m[i][j] = 0;**

**if(i+1<n && m[i+1][j]==1) {**

**str.push\_back('D');**

**solve(m,ans,i+1,j,n,str);**

**str.pop\_back();**

**}**

**if(i-1>=0 && m[i-1][j]==1) {**

**str.push\_back('U');**

**solve(m,ans,i-1,j,n,str);**

**str.pop\_back();**

**}**

**if(j+1<n && m[i][j+1]==1) {**

**str.push\_back('R');**

**solve(m,ans,i,j+1,n,str);**

**str.pop\_back();**

**}**

**if(j-1>=0 && m[i][j-1]==1) {**

**str.push\_back('L');**

**solve(m,ans,i,j-1,n,str);**

**str.pop\_back();**

**}**

**m[i][j] = 1;**

**}**

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

**// Your code goes here**

**vector<string> ans;**

**if(m[0][0]==0) {**

**return ans;**

**}**

**string str;**

**solve(m,ans,0,0,n,str);**

**sort(ans.begin(),ans.end());**

**return ans;**

**}**

### **223) 2078. Two Furthest Houses With Different Colors**

There are n houses evenly lined up on the street, and each house is beautifully painted. You are given a **0-indexed** integer array colors of length n, where colors[i] represents the color of the ith house.

Return the ***maximum*** distance between ***two*** houses with ***different*** colors.

The distance between the ith and jth houses is abs(i - j), where abs(x) is the **absolute value** of x.

**Example 1:**

**Input:** colors = [**1**,1,1,**6**,1,1,1]

**Output:** 3

**Explanation:** In the above image, color 1 is blue, and color 6 is red.

The furthest two houses with different colors are house 0 and house 3.

House 0 has color 1, and house 3 has color 6. The distance between them is abs(0 - 3) = 3.

Note that houses 3 and 6 can also produce the optimal answer.

**:** **int maxDistance(vector<int>& colors) {**

**int i=0;**

**int j=colors.size()-1;**

**int maxi1=0;**

**int maxi2=0;**

**while(j>=0)**

**{**

**if(colors[i]!=colors[j])**

**{**

**maxi1=abs(i-j);**

**break;**

**}**

**j--;**

**}**

**i=0;**

**j=colors.size()-1;**

**while(i<colors.size())**

**{**

**if(colors[i]!=colors[j])**

**{**

**maxi2=abs(i-j);**

**break;**

**}**

**i++;**

**}**

**return max(maxi1,maxi2);**

**}**

**224** )

**2073. Time Needed to Buy Tickets**

There are n people in a line queuing to buy tickets, where the 0th person is at the **front** of the line and the (n - 1)th person is at the **back** of the line.

You are given a **0-indexed** integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes **exactly 1 second** to buy a ticket. A person can only buy **1 ticket at a time** and has to go back to **the end** of the line (which happens **instantaneously**) in order to buy more tickets. If a person does not have any tickets left to buy, the person will **leave**the line.

Return the ***time taken*** for the person at position k **(0-indexed)**to finish buying tickets.

**Example 1:**

**Input:** tickets = [2,3,2], k = 2

**Output:** 6

**Explanation:**

- In the first pass, everyone in the line buys a ticket and the line becomes [1, 2, 1].

- In the second pass, everyone in the line buys a ticket and the line becomes [0, 1, 0].

The person at position 2 has successfully bought 2 tickets and it took 3 + 3 = 6 seconds.

**:** **int timeRequiredToBuy(vector<int>& tickets, int k) {**

**int cnt=0;**

**while(tickets[k]!=0)**

**{**

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

**{**

**if(tickets[i]==0)**

**continue;**

**tickets[i]--;**

**cnt++;**

**if(tickets[k]==0)**

**return cnt;**

**}**

**}**

**return cnt;**

**}**

**225)** Given a **0-indexed** integer array nums, find the **leftmost** middleIndex (i.e., the smallest amongst all the possible ones).

A middleIndex is an index where nums[0] + nums[1] + ... + nums[middleIndex-1] == nums[middleIndex+1] + nums[middleIndex+2] + ... + nums[nums.length-1].

If middleIndex == 0, the left side sum is considered to be 0. Similarly, if middleIndex == nums.length - 1, the right side sum is considered to be 0.

Return *the****leftmost***middleIndex*that satisfies the condition, or*-1*if there is no such index*.

**Example 1:**

**Input:** nums = [2,3,-1,8,4]

**Output:** 3

**Explanation:**

The sum of the numbers before index 3 is: 2 + 3 + -1 = 4

The sum of the numbers after index 3 is: 4 = 4

**:** **int findMiddleIndex(vector<int>& nums) {**

**int n=nums.size();**

**vector<int>left(n);**

**vector<int>right(n);**

**left[0]=nums[0];**

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

**left[i]=nums[i]+left[i-1];**

**right[n-1]=nums[n-1];**

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

**right[i]=nums[i]+right[i+1];**

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

**{**

**if(left[i]==right[i])**

**return i;**

**}**

**return -1;**

**}**

**226)** You are given a rectangular cake of size h x w and two arrays of integers horizontalCuts and verticalCuts where:

* horizontalCuts[i] is the distance from the top of the rectangular cake to the ith horizontal cut and similarly, and
* verticalCuts[j] is the distance from the left of the rectangular cake to the jth vertical cut.

Return *the maximum area of a piece of cake after you cut at each horizontal and vertical position provided in the arrays* horizontalCuts *and* verticalCuts. Since the answer can be a large number, return this **modulo** 109 + 7.

**Example 1:**

**Input:** h = 5, w = 4, horizontalCuts = [1,2,4], verticalCuts = [1,3]

**Output:** 4

**Explanation:** The figure above represents the given rectangular cake. Red lines are the horizontal and vertical cuts. After you cut the cake, the green piece of cake has the maximum area.

EXPLANATION:

**Key idea**:  
If we need the area, we must think of lengths and breadths of each cake piece.

**Assume**  
LENGTH as total horizontal length of the original cake.  
HEIGHT as total vertical height of the original cake.  
H as number of horizontal cuts. HCUTS is array of horizontal cuts.  
Similarly, V as number of vertical cuts and VCUTS is an array of vertical cuts.

**Solution**

1. Lets only think of horizontal cuts. Each HCUTS[i] would create a piece with length LENGTH and height, say, heights[i].  
   As there are H cuts, there will be (H+1) pieces of length LENGTH.
2. Now each vertical cut VCUTS[i] will cut each of the horizontal pieces that we got in step 1.  
   We already know the height of each piece (step 1), now with each vertical cut, we will know the length of each piece as well.
3. Because we want the maximize the area, we must try to maximize the length and height of each piece.

**Algorithm**

1. Find heights of pieces if we only perform the horizontal cuts. Say this array is heights[].
2. Find lengths of pieces if we only perform the vertical cuts. Say this arrays is lengths[].
3. Find max of heights[] and lengths[].
4. Multiply those two max and take mod 10e7.
5. Return the answer

Is there anything that is still unclear? Let me know and I can elaborate.

**:** **int getmax(int h,vector<int>hi)**

**{**

**sort(hi.begin(),hi.end());**

**int lenght=hi.size();**

**int ans=max(h-hi[lenght-1],hi[0]);**

**for(int i=1;i<lenght;i++)**

**ans=max(ans,hi[i]-hi[i-1]);**

**return ans;**

**}**

**int maxArea(int h, int w, vector<int>& horizontalCuts, vector<int>& verticalCuts) {**

**return (int)((long)getmax(h,horizontalCuts)\*getmax(w,verticalCuts)%1000000007);**

**}**

**227)** Given an array of integers nums and an integer k, return *the number of****unique****k-diff pairs in the array*.

A **k-diff** pair is an integer pair (nums[i], nums[j]), where the following are true:

* 0 <= i < j < nums.length
* |nums[i] - nums[j]| == k

**Notice** that |val| denotes the absolute value of val.

**Example 1:**

**Input:** nums = [3,1,4,1,5], k = 2

**Output:** 2

**Explanation:** There are two 2-diff pairs in the array, (1, 3) and (3, 5).

Although we have two 1s in the input, we should only return the number of **unique** pairs.

**:** **int findPairs(vector<int>& nums, int k) {**

**if(k<0)**

**return 0;**

**unordered\_set<int>st;**

**unordered\_map<int,int>mp;**

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

**{**

**if(mp[nums[i]-k])**

**st.insert(nums[i]-k);**

**if(mp[nums[i]+k])**

**st.insert(nums[i]);**

**mp[nums[i]]+=1;**

**}**

**return st.size();**

**}**

**228)** Given an integer array nums and an integer k, return *the number of non-empty****subarrays****that have a sum divisible by*k.

A **subarray** is a **contiguous** part of an array.

**Example 1:**

**Input:** nums = [4,5,0,-2,-3,1], k = 5

**Output:** 7

**Explanation:** There are 7 subarrays with a sum divisible by k = 5:

[4, 5, 0, -2, -3, 1], [5], [5, 0], [5, 0, -2, -3], [0], [0, -2, -3], [-2, -3]

**: int subarraysDivByK(vector<int>& nums, int k) {**

**vector<int>v(k);**

**v[0]=1;**

**int ans=0;**

**int presum=0;**

**for(auto it:nums)**

**{**

**presum=(presum+it%k+k)%k;**

**ans+=v[presum]++;**

**}**

**return ans;**

**}**

**229)** Geek created a random series and given a name geek-onacci series. Given four integers **A, B, C, N**. A, B, C represents the first three numbers of geek-onacci series. Find the Nth number of the series. The nth number of geek-onacci series is a sum of the last three numbers (summation of N-1th, N-2th, and N-3th geek-onacci numbers)

**Input:**  
1. The first line of the input contains a single integer**T** denoting the number of test cases. The description of **T** test cases follows.  
2. The first line of each test case contains four space-separated integers **A, B, C,**and**N.**  
  
**Output:** For each test case, print **Nth**geek-onacci number  
  
**Constraints:**  
1. 1 <= T <= 3  
2. 1 <= A, B, C <= 100  
3. 4 <= N <= 10  
  
**Example:  
Input:**  
3  
1 3 2 4  
1 3 2 5  
1 3 2 6

**Output:**  
6  
11  
19

**:** **#include <iostream>**

**using namespace std;**

**int main() {**

**//code**

**int t;**

**cin>>t;**

**while(t--)**

**{**

**int a,b,c,n;**

**cin>>a>>b>>c>>n;**

**int sum=0;**

**for(int i=4;i<=n;i++)**

**{**

**sum=a+b+c;**

**a=b;**

**b=c;**

**c=sum;**

**}**

**cout<<c<<endl;**

**}**

**return 0;**

**}**

**230)** Print numbers from **1 to N** without the help of loops.

**Example 1:**

**Input:**

N = 10

**Output:** 1 2 3 4 5 6 7 8 9 10

**:** void printNos(int N)

{

//Your code here

if(N==0)

return;

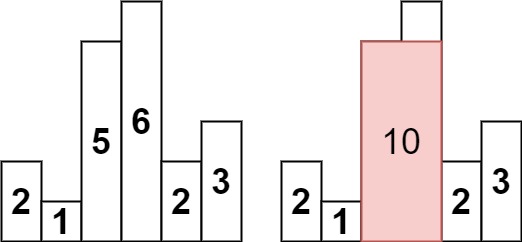
printNos(N-1);

cout<<N<<" ";

}

231) 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 the histogram*.

**Example 1:**



**Input:** heights = [2,1,5,6,2,3]

**Output:** 10

**Explanation:** The above is a histogram where width of each bar is 1.

The largest rectangle is shown in the red area, which has an area = 10 units.

: int largestRectangleArea(vector<int>& heights) {

heights.push\_back(0);

int n=heights.size();

int ans=0;

stack<int>st;

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

{

while(!st.empty()&&heights[st.top()]>heights[i])

{

int h=heights[st.top()];

st.pop();

int l=st.empty()?-1:st.top();

ans=max(ans,h\*(i-l-1));

}

st.push(i);

}

return ans;

}

232) Alice and Bob take turns playing a game, with Alice starting first.

Initially, there is a number n on the chalkboard. On each player's turn, that player makes a move consisting of:

* Choosing any x with 0 < x < n and n % x == 0.
* Replacing the number n on the chalkboard with n - x.

Also, if a player cannot make a move, they lose the game.

Return true *if and only if Alice wins the game, assuming both players play optimally*.

**Example 1:**

**Input:** n = 2

**Output:** true

**Explanation:** Alice chooses 1, and Bob has no more moves.

: bool divisorGame(int n) {

return n%2==0;

}

233) Given an integer n, return *an array*ans*of length*n + 1*such that for each*i(0 <= i <= n)*,*ans[i]*is the****number of***1***'s****in the binary representation of*i.

**Example 1:**

**Input:** n = 2

**Output:** [0,1,1]

**Explanation:**

0 --> 0

1 --> 1

2 --> 10

: vector<int> countBits(int n) {

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

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

ans[i]=ans[i&(i-1)]+1;

return ans;

}

234) A message containing letters from A-Z can be **encoded** into numbers using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To **decode** an encoded message, all the digits must be grouped then mapped back into letters using the reverse of the mapping above (there may be multiple ways). For example, "11106" can be mapped into:

* "AAJF" with the grouping (1 1 10 6)
* "KJF" with the grouping (11 10 6)

Note that the grouping (1 11 06) is invalid because "06" cannot be mapped into 'F' since "6" is different from "06".

Given a string s containing only digits, return *the****number****of ways to****decode****it*.

The answer is guaranteed to fit in a **32-bit** integer.

**Example 1:**

**Input:** s = "12"

**Output:** 2

**Explanation:** "12" could be decoded as "AB" (1 2) or "L" (12).

: int numDecodings(string s) {

int temp=1;

int ans;

for(int i=s.size()-1;i>=0;i--)

{

int curr=s[i]=='0'?0:temp;

if(i<s.size()-1&&(s[i]=='1'||s[i]=='2'&&s[i+1]<'7'))

curr+=ans;

ans=temp;

temp=curr;

}

return s.empty()?0:temp;

}

235) Given an m x n binary matrix filled with 0's and 1's, *find the largest square containing only* 1's *and return its area*.

**Example 1:**



**Input:** matrix = [["1","0","1","0","0"],["1","0","1","1","1"],["1","1","1","1","1"],["1","0","0","1","0"]]

**Output:** 4

: int maximalSquare(vector<vector<char>>& matrix) {

if(matrix.size()==0)

return 0;

int n=matrix.size();

int m=matrix[0].size();

int sz=0;

int pre;

vector<int>vis(m,0);

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

{

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

{

int temp=vis[j];

if(!i||!j||matrix[i][j]=='0')

vis[j]=matrix[i][j]-'0';

else

vis[j]=min(pre,min(vis[j],vis[j-1]))+1;

sz=max(sz,vis[j]);

pre=temp;

}

}

return sz\*sz;

}

236) Given an integer array nums, find a contiguous non-empty subarray within the array that has the largest product, and return *the product*.

It is **guaranteed** that the answer will fit in a **32-bit** integer.

A **subarray** is a contiguous subsequence of the array.

**Example 1:**

**Input:** nums = [2,3,-2,4]

**Output:** 6

**Explanation:** [2,3] has the largest product 6.

: int maxProduct(vector<int>& nums) {

int r=nums[0];

for(int i=1,mini=r,maxi=r;i<nums.size();i++)

{

if(nums[i]<0)

swap(mini,maxi);

mini=min(nums[i],mini\*nums[i]);

maxi=max(nums[i],maxi\*nums[i]);

r=max(r,maxi);

}

return r;

}

237) Given two integer arrays nums1 and nums2, return *the maximum length of a subarray that appears in****both****arrays*.

**Example 1:**

**Input:** nums1 = [1,2,3,2,1], nums2 = [3,2,1,4,7]

**Output:** 3

**Explanation:** The repeated subarray with maximum length is [3,2,1].

: int findLength(vector<int>& nums1, vector<int>& nums2) {

int n=nums1.size();

int m=nums2.size();

int ans=0;

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

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

{

for(int j=m-1;j>=0;j--)

{

ans=max(ans,dp[i][j]=nums1[i]==nums2[j]?1+dp[i+1][j+1]:0);

}

}

return ans;

}

238) Given an integer array nums and an integer k, return true *if*nums*has a continuous subarray of size****at least two****whose elements sum up to a multiple of* k*, or*false*otherwise*.

An integer x is a multiple of k if there exists an integer n such that x = n \* k. 0 is **always** a multiple of k.

**Example 1:**

**Input:** nums = [23,2,4,6,7], k = 6

**Output:** true

**Explanation:** [2, 4] is a continuous subarray of size 2 whose elements sum up to 6.

: bool checkSubarraySum(vector<int>& nums, int k) {

if(nums.size()<2)

return 0;

unordered\_map<int,int>mp;

mp[0]=-1;

int curr\_sum=0;

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

{

curr\_sum+=nums[i];

if(k!=0)

curr\_sum=curr\_sum%k;

if(mp.find(curr\_sum)!=mp.end())

{

if(i-mp[curr\_sum]>1)

return 1;

}

else

mp[curr\_sum]=i;

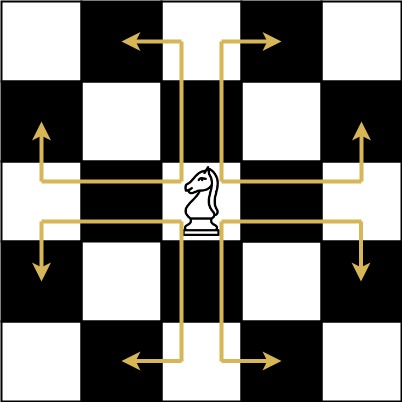
}

return 0;

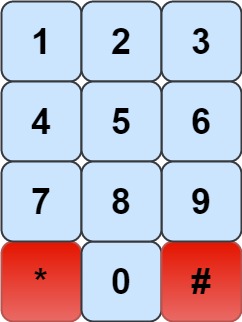
}

239) The chess knight has a **unique movement**, it may move two squares vertically and one square horizontally, or two squares horizontally and one square vertically (with both forming the shape of an **L**). The possible movements of chess knight are shown in this diagaram:

A chess knight can move as indicated in the chess diagram below:



We have a chess knight and a phone pad as shown below, the knight **can only stand on a numeric cell** (i.e. blue cell).



Given an integer n, return how many distinct phone numbers of length n we can dial.

You are allowed to place the knight **on any numeric cell** initially and then you should perform n - 1 jumps to dial a number of length n. All jumps should be **valid** knight jumps.

As the answer may be very large, **return the answer modulo** 109 + 7.

**Example 1:**

**Input:** n = 1

**Output:** 10

**Explanation:** We need to dial a number of length 1, so placing the knight over any numeric cell of the 10 cells is sufficient.

: int knightDialer(int n) {

if(n==1)

return 10;

int mod=1000000007;

vector<long long int>ans1(10,1);

vector<long long int>ans2(10);

ans1[5]=0;

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

{

ans2[0]=(ans1[4]+ans1[6])%mod;

ans2[1]=(ans1[6]+ans1[8])%mod;

ans2[2]=(ans1[7]+ans1[9])%mod;

ans2[3]=(ans1[4]+ans1[8])%mod;

ans2[4]=(ans1[0]+ans1[3]+ans1[9])%mod;

ans2[6]=(ans1[0]+ans1[1]+ans1[7])%mod;

ans2[7]=(ans1[2]+ans1[6])%mod;

ans2[8]=(ans1[1]+ans1[3])%mod;

ans2[9]=(ans1[2]+ans1[4])%mod;

for(int i=0;i<10;i++)

ans1[i]=ans2[i];

}

int sum=0;

for(int i=0;i<10;i++)

sum=(sum+ans1[i])%mod;

return sum;

}

240) Given a m \* n matrix of ones and zeros, return how many **square** submatrices have all ones.

**Example 1:**

**Input:** matrix =

[

  [0,1,1,1],

  [1,1,1,1],

  [0,1,1,1]

]

**Output:** 15

**Explanation:**

There are **10** squares of side 1.

There are **4** squares of side 2.

There is **1** square of side 3.

Total number of squares = 10 + 4 + 1 = **15**.

## **Explanation**

dp[i][j] means the size of biggest square with A[i][j] as bottom-right corner.  
dp[i][j] also means the number of squares with A[i][j] as bottom-right corner.

If A[i][j] == 0, no possible square.  
If A[i][j] == 1,  
we compare the size of square dp[i-1][j-1], dp[i-1][j] and dp[i][j-1].  
min(dp[i-1][j-1], dp[i-1][j], dp[i][j-1]) + 1 is the maximum size of square that we can find.

## **Complexity**

Time O(MN)  
Space O(1)

: int countSquares(vector<vector<int>>& matrix) {

int ans=0;

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

{

for(int j=0;j<matrix[0].size();ans+=matrix[i][j++])

{

if(matrix[i][j]&&i&&j)

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

}

}

return ans;

}

241) Given an array nums of integers, return the **length** of the longest arithmetic subsequence in nums.

Recall that a *subsequence* of an array nums is a list nums[i1], nums[i2], ..., nums[ik] with 0 <= i1 < i2 < ... < ik <= nums.length - 1, and that a sequence seq is *arithmetic* if seq[i+1] - seq[i] are all the same value (for 0 <= i < seq.length - 1).

**Example 1:**

**Input:** nums = [3,6,9,12]

**Output:** 4

**Explanation:**

The whole array is an arithmetic sequence with steps of length = 3.

: int longestArithSeqLength(vector<int>& nums) {

if(nums.size()==2)

return 2;

int n=nums.size();

vector<vector<int>>ans(n,vector<int>(1001,1));

int len=1;

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

{

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

{

int ind=nums[i]-nums[j]+500;

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

len=max(len,ans[i][ind]);

}

}

return len;

}

242) Given a string s, return true *if the*s*can be palindrome after deleting****at most one****character from it*.

**Example 1:**

**Input:** s = "aba"

**Output:** true

**Example 2:**

**Input:** s = "abca"

**Output:** true

**Explanation:** You could delete the character 'c'.

RECURSIVE WAY

: bool validPalindrome(string s) {

return valid(s,0,s.size()-1,1);

}

bool valid(string&s,int l,int r,int d)

{

if(l>=r)

return 1;

if(s[l]==s[r])

return valid(s,l+1,r-1,d);

else

return d>0&&(valid(s,l+1,r,d-1)||valid(s,l,r-1,d-1));

}

ITERATIVE WAY

: bool validPalindrome(string s) {

for(int i=0,j=s.size()-1;i<j;i++,j--)

{

if(s[i]!=s[j])

{

return valid(s,i+1,j)||valid(s,i,j-1);

}

}

return 1;

}

bool valid(string s,int l,int r)

{

for(int i=l,j=r;i<j;i++,j--)

{

if(s[i]!=s[j])

return 0;

}

return 1;

}

243) Implement [strStr()](http://www.cplusplus.com/reference/cstring/strstr/).

Return the index of the first occurrence of needle in haystack, or -1 if needle is not part of haystack.

**Clarification:**

What should we return when needle is an empty string? This is a great question to ask during an interview.

For the purpose of this problem, we will return 0 when needle is an empty string. This is consistent to C's [strstr()](http://www.cplusplus.com/reference/cstring/strstr/) and Java's [indexOf()](https://docs.oracle.com/javase/7/docs/api/java/lang/String.html#indexOf(java.lang.String)).

O(m\*(n-m+1)) with kmp but with kmp we got O(n)

**Example 1:**

**Input:** haystack = "hello", needle = "ll"

**Output:** 2

**Example 2:**

**Input:** haystack = "aaaaa", needle = "bba"

**Output:** -1

: int strStr(string haystack, string needle) {

int n=haystack.size();

int m=needle.size();

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

{

int j=0;

for(;j<m;j++)

{

if(haystack[i+j]!=needle[j])

break;

}

if(j==m)

return i;

}

return -1;

}

CRISP CODE BUT LITTLE CUMSY TO UNDERSTAND

int strStr(string haystack, string needle) {

int n=haystack.size();

int m=needle.size();

int p=0;

while(p+m-1<n)

{

if(haystack.substr(p,m)==needle)

return p;

while(p++ +m-1<n&&haystack[p]!=needle[0]);

}

return -1;

}

# The Knuth-Morris-Pratt Algorithm in my own words

DEC 13TH, 2009 | [COMMENTS](http://jakeboxer.com/blog/2009/12/13/the-knuth-morris-pratt-algorithm-in-my-own-words/#disqus_thread)

For the past few days, I’ve been reading various explanations of [the Knuth-Morris-Pratt string searching algorithms](http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm). For some reason, none of the explanations were doing it for me. I kept banging my head against a brick wall once I started reading “the prefix of the suffix of the prefix of the…”.

Finally, after reading the same paragraph of [CLRS](http://www.amazon.com/Introduction-Algorithms-Third-Thomas-Cormen/dp/0262033844/) over and over for about 30 minutes, I decided to sit down, do a bunch of examples, and diagram them out. I now understand the algorithm, and can explain it. For those who think like me, here it is in my own words. As a side note, I’m not going to explain why it’s more efficient than na”ive string matching; that’s explained perfectly well in a [multitude](http://en.wikipedia.org/wiki/Knuth-Morris-Pratt_algorithm) [of](http://www.ics.uci.edu/~eppstein/161/960227.html) [places](http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/StringMatch/kuthMP.htm). I’m going to explain exactly how it works, as my brain understands it.

## The Partial Match Table

The key to KMP, of course, is the partial match table. The main obstacle between me and understanding KMP was the fact that I didn’t quite fully grasp what the values in the partial match table really meant. I will now try to explain them in the simplest words possible.

Here’s the partial match table for the pattern “abababca”:

|  |  |
| --- | --- |
| 1  2  3 | char: | a | b | a | b | a | b | c | a |  index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  value: | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | |

If I have an eight-character pattern (let’s say “abababca” for the duration of this example), my partial match table will have eight cells. If I’m looking at the eighth and last cell in the table, I’m interested in the entire pattern (“abababca”). If I’m looking at the seventh cell in the table, I’m only interested in the first seven characters in the pattern (“abababc”); the eighth one (“a”) is irrelevant, and can go fall off a building or something. If I’m looking at the sixth cell of the in the table… you get the idea. Notice that I haven’t talked about what each cell means yet, but just what it’s referring to.

Now, in order to talk about the meaning, we need to know about **proper prefixes** and **proper suffixes**.

**Proper prefix**: All the characters in a string, with one or more cut off the end. “S”, “Sn”, “Sna”, and “Snap” are all the proper prefixes of “Snape”.

**Proper suffix**: All the characters in a string, with one or more cut off the beginning. “agrid”, “grid”, “rid”, “id”, and “d” are all proper suffixes of “Hagrid”.

With this in mind, I can now give the one-sentence meaning of the values in the partial match table:

**The length of the longest proper prefix in the (sub)pattern that matches a proper suffix in the same (sub)pattern.**

Let’s examine what I mean by that. Say we’re looking in the third cell. As you’ll remember from above, this means we’re only interested in the first three characters (“aba”). In “aba”, there are two proper prefixes (“a” and “ab”) and two proper suffixes (“a” and “ba”). The proper prefix “ab” does not match either of the two proper suffixes. However, the proper prefix “a” matches the proper suffix “a”. Thus, **the length of the longest proper prefix that matches a proper suffix**, in this case, is 1.

Let’s try it for cell four. Here, we’re interested in the first four characters (“abab”). We have three proper prefixes (“a”, “ab”, and “aba”) and three proper suffixes (“b”, “ab”, and “bab”). This time, “ab” is in both, and is two characters long, so cell four gets value 2.

Just because it’s an interesting example, let’s also try it for cell five, which concerns “ababa”. We have four proper prefixes (“a”, “ab”, “aba”, and “abab”) and four proper suffixes (“a”, “ba”, “aba”, and “baba”). Now, we have two matches: “a” and “aba” are both proper prefixes and proper suffixes. Since “aba” is longer than “a”, it wins, and cell five gets value 3.

Let’s skip ahead to cell seven (the second-to-last cell), which is concerned with the pattern “abababc”. Even without enumerating all the proper prefixes and suffixes, it should be obvious that there aren’t going to be any matches; all the suffixes will end with the letter “c”, and none of the prefixes will. Since there are no matches, cell seven gets 0.

Finally, let’s look at cell eight, which is concerned with the entire pattern (“abababca”). Since they both start and end with “a”, we know the value will be at least 1. However, that’s where it ends; at lengths two and up, all the suffixes contain a c, while only the last prefix (“abababc”) does. This seven-character prefix does not match the seven-character suffix (“bababca”), so cell eight gets 1.

## How to use the Partial Match Table

We can use the values in the partial match table to skip ahead (rather than redoing unnecessary old comparisons) when we find partial matches. The formula works like this:

If a partial match of length ***partial\_match\_length*** is found and *table[partial\_match\_length] > 1*, we may skip ahead *partial\_match\_length - table[partial\_match\_length - 1]* characters.

Let’s say we’re matching the pattern “abababca” against the text “bacbababaabcbab”. Here’s our partial match table again for easy reference:

|  |  |
| --- | --- |
| 1  2  3 | char: | a | b | a | b | a | b | c | a |  index: | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 |  value: | 0 | 0 | 1 | 2 | 3 | 4 | 0 | 1 | |

The first time we get a partial match is here:

|  |  |
| --- | --- |
| 1  2  3 | Bacbababaabcbab  |  Abababca |

This is a partial\_match\_length of 1. The value at table[partial\_match\_length - 1] (or table[0]) is 0, so we don’t get to skip ahead any. The next partial match we get is here:

|  |  |
| --- | --- |
| 1  2  3 | Bacbababaabcbab  |||||  Abababca |

This is a partial\_match\_length of 5. The value at table[partial\_match\_length - 1] (or table[4]) is 3. That means we get to skip ahead partial\_match\_length - table[partial\_match\_length - 1] (or 5 - table[4] or 5 - 3 or 2) characters:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // x denotes a skip  bacbababaabcbab  xx|||  abababca |

This is a partial\_match\_length of 3. The value at table[partial\_match\_length - 1] (or table[2]) is 1. That means we get to skip ahead partial\_match\_length - table[partial\_match\_length - 1] (or 3 - table[2] or 3 - 1 or 2) characters:

|  |  |
| --- | --- |
| 1  2  3  4  5 | // x denotes a skip  bacbababaabcbab  xx|  abababca |

At this point, our pattern is longer than the remaining characters in the text, so we know there’s no match.

## Conclusion

So there you have it. Like I promised before, it’s no exhaustive explanation or formal proof of KMP; it’s a walk through my brain, with the parts I found confusing spelled out in extreme detail. If you have any questions or notice something I messed up, please leave a comment; maybe we’ll all learn something.

CODE BASED ON KMP ALGO

int strStr(string haystack, string needle) {

if(needle.size()==0)

return 0;

int n=haystack.size();

int m=needle.size();

vector<int>lps=func(needle);

for(int i=0,j=0;i<n;)

{

if(haystack[i]==needle[j])

i++,j++;

if(j==m)

return i-j;

if(i<n&&haystack[i]!=needle[j])

j?j=lps[j-1]:i++;

}

return -1;

}

vector<int>func(string s)

{

int n=s.size();

vector<int>lps(n,0);

for(int i=1,len=0;i<n;)

{

if(s[i]==s[len])

lps[i++]=++len;

else if(len)

len=lps[len-1];

else

lps[i++]=0;

}

return lps;

}

244) Given a string s of '(' , ')' and lowercase English characters.

Your task is to remove the minimum number of parentheses ( '(' or ')', in any positions ) so that the resulting *parentheses string* is valid and return **any** valid string.

Formally, a *parentheses string* is valid if and only if:

* It is the empty string, contains only lowercase characters, or
* It can be written as AB (A concatenated with B), where A and B are valid strings, or
* It can be written as (A), where A is a valid string.

**Example 1:**

**Input:** s = "lee(t(c)o)de)"

**Output:** "lee(t(c)o)de"

**Explanation:** "lee(t(co)de)" , "lee(t(c)ode)" would also be accepted.

: string minRemoveToMakeValid(string s) {

stack<int>st;

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

{

if(s[i]=='(')

st.push(i);

if(s[i]==')')

{

if(!st.empty())

st.pop();

else

s[i]='\*';

}

}

while(!st.empty())

{

s[st.top()]='\*';

st.pop();

}

s.erase(remove(s.begin(),s.end(),'\*'),s.end());

return s;

}

ANOTHER APPPROACHs

: string minRemoveToMakeValid(string s) {

int opening=0;

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

{

if(s[i]=='(')

opening++;

else if(s[i]==')')

opening--;

if(opening<0)

s[i]='\*',opening++;

}

int closing=0;

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

{

if(s[i]=='(')

closing--;

else if(s[i]==')')

closing++;

if(closing<0)

s[i]='\*',closing++;

}

s.erase(remove(s.begin(),s.end(),'\*'),s.end());

return s;

}

245) Given an array nums. We define a running sum of an array as runningSum[i] = sum(nums[0]…nums[i]).

Return the running sum of nums.

**Example 1:**

**Input:** nums = [1,2,3,4]

**Output:** [1,3,6,10]

**Explanation:** Running sum is obtained as follows: [1, 1+2, 1+2+3, 1+2+3+4].

: vector<int> runningSum(vector<int>& nums) {

int sum=nums[0];

vector<int>ans;

for(auto it:nums)

ans.push\_back(it);

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

{

nums[i]=sum+nums[i];

sum+=ans[i];

}

return nums;

}

SHORT AND CRISP CODE

vector<int> runningSum(vector<int>& nums) {

vector<int>ans(nums.size());

ans[0]=nums[0];

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

{

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

}

return ans;

}

246) Given an integer array nums and an integer k, return true if there are two **distinct indices** i and j in the array such that nums[i] == nums[j] and abs(i - j) <= k.

**Example 1:**

**Input:** nums = [1,2,3,1], k = 3

**Output:** true

: bool containsNearbyDuplicate(vector<int>& nums, int k) {

unordered\_map<int,int>mp;

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

{

if(mp[nums[i]]!=0&&i-mp[nums[i]]<k)

return 1;

else

mp[nums[i]]=i+1;

}

return 0;

}

247) Given an array of strings strs, group **the anagrams** together. You can return the answer in **any order**.

An **Anagram** is a word or phrase formed by rearranging the letters of a different word or phrase, typically using all the original letters exactly once.

**Example 1:**

**Input:** strs = ["eat","tea","tan","ate","nat","bat"]

**Output:** [["bat"],["nat","tan"],["ate","eat","tea"]]

: vector<vector<string>> groupAnagrams(vector<string>& strs) {

unordered\_map<string,vector<string>>mp;

for(auto it:strs)

{

string temp=it;

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

mp[temp].push\_back(it);

}

vector<vector<string>>ans;

for(auto it:mp)

{

ans.push\_back(it.second);

}

return ans;

}

248) Given n pairs of parentheses, write a function to *generate all combinations of well-formed parentheses*.

**Example 1:**

**Input:** n = 3

**Output:** ["((()))","(()())","(())()","()(())","()()()"]

: vector<string> generateParenthesis(int n) {

vector<string>ans;

func(ans,"",n,n);

return ans;

}

void func(vector<string>&ans,string s,int l,int r)

{

if(l==0&&r==0)

{

ans.push\_back(s);

return ;

}

if(l>0)

func(ans,s+"(",l-1,r);

if(r>l)

func(ans,s+")",l,r-1);

}

249) You are given an integer array digits, where each element is a digit. The array may contain duplicates.

You need to find **all** the **unique** integers that follow the given requirements:

* The integer consists of the **concatenation** of **three** elements from digits in **any** arbitrary order.
* The integer does not have **leading zeros**.
* The integer is **even**.

For example, if the given digits were [1, 2, 3], integers 132 and 312 follow the requirements.

Return *a****sorted****array of the unique integers.*

**Example 1:**

**Input:** digits = [2,1,3,0]

**Output:** [102,120,130,132,210,230,302,310,312,320]

**Explanation:**

All the possible integers that follow the requirements are in the output array.

Notice that there are no **odd** integers or integers with **leading zeros**.

: vector<int> findEvenNumbers(vector<int>& digits) {

int n=digits.size();

unordered\_set<int>st;

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

{

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

{

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

{

if(i==j||j==k||i==k)

continue;

if(digits[i]==0)

continue;

if(digits[k]%2==1)

continue;

int x=digits[i]\*100+digits[j]\*10+digits[k];

st.insert(x);

}

}

}

vector<int>ans;

for(auto it:st)

{

ans.push\_back(it);

}

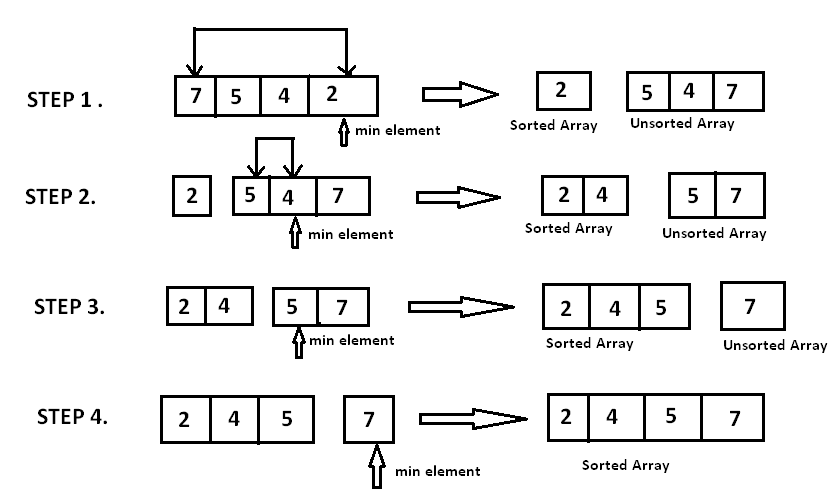
sort(ans.begin(),ans.end());

return ans;

}

MUST DO SORTING ALGO’S

Sorting is the rearrangement of elements in into a specific order.  
Sorting algorithms helps in making the solution easier and efficient.  
Some basic yet important sort algorithms, that one must know are as follows:

**1. Selection Sort:** It sorts an array by repeatedly finding the minimum element (considering decresing order) from unsorted part and putting it at the beginning.  


void selection\_sort(vector <int> arr, int n){

int min\_index=i;

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

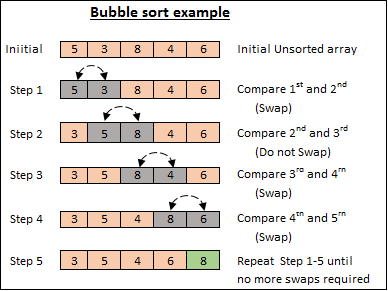
if(arr[j]<arr[min\_index])

min\_index=j;

}

swap(arr[i],arr[min\_index]);

}

**2. Bubble Sort:** It works by repeatedly swapping the adjacent wrong elements.  


void bubble\_sort(vector <int> arr, int n){

for(int itr=1;itr<n;itr++){

for(int j=0;j<(n-itr-);j++){

if(arr[j]>arr[j+1])

swap(arr[j],arr[j+1]);

}

}

}

**Recursive Bubble Sort:**

void bubble\_sort(vector <int> arr, int n){

if (n == 1) // Base case

return;

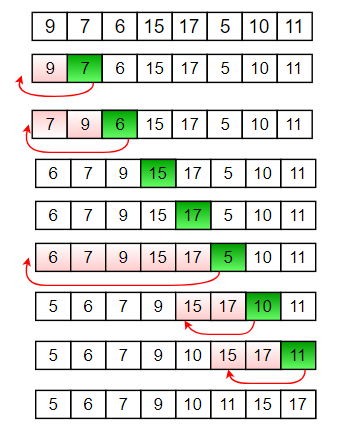
for (int i=0; i<n-1; i++)

if (arr[i] > arr[i+1])

swap(arr[i], arr[i+1]);

bubble\_sort(arr, n-1);

}

**3. Insertion Sort:** Values from unsorted part is picked and placed at the sorted position.  


void insertion\_sort(vector <int> arr, int n){

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

int e=arr[i];

int j=i-1;

while(j>=0 && arr[j]<arr[j]>e){

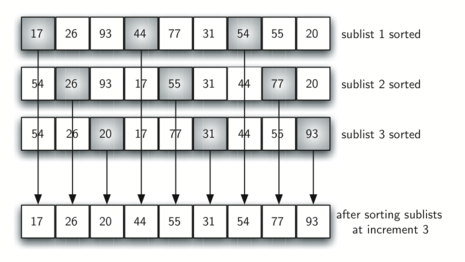
a[j+1]=a[j];

j=j-1;

}

a[j+1]=e;

}

**4. Shell Sort:** It is the variation of Insertion sort, as here the elements are moved far ahead.  


void shell\_sort(vector <int> arr, int n){

for (int gap = n/2; gap > 0; gap /= 2) {

for (int i = gap; i < n; i += 1) {

int temp = arr[i];

int j;

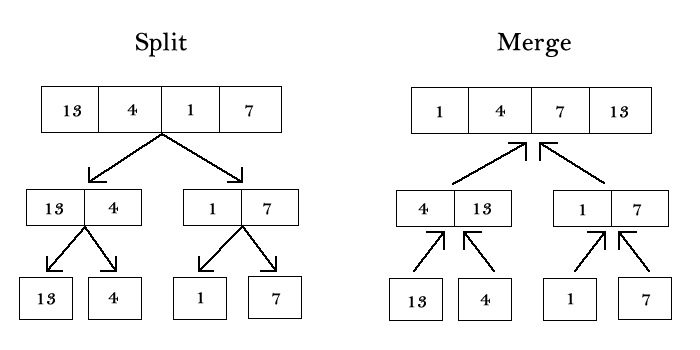
for (j = i; j >= gap && arr[j - gap] > temp; j -= gap)

arr[j] = arr[j - gap];

arr[j] = temp;

}

}

**5. Merge Sort:** It is based on **Divide and Conquer algorithm**. It divides the input array into two halves, calls itself for the two halves, and then merges the two sorted halves.  


void merge\_sort(int \*arr, int start, int end){

if(start==end)

return;

int mid=(start+end)/2;

merge\_sort(arr,start,mid);

merge\_sort(arr,mid+1,end);

merge(arr,start,end);

}

void merge(int \*arr, int start, int end){

int mid=(start+end)/2;

int i=start;

int j=mid+1;

int k=start;

int temp[100];

while(i<=mid && j<end){

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

temp[k++]=arr[i++];

else

temp[k++]=arr[j++];

}

while(i<=mid){

temp[k++]=arr[i++];

}

while(j<=end){

temp[k++]=arr[j++];

}

for(i=start;i<=end;i++)

arr[i]=temp[i];

**CODE 1):**

*class Solution  
{  
public:  
void merge(int arr[], int l, int m, int r)  
{  
int left=l,right=m+1;  
vector<int> temp;  
while(left<=m && right <=r){  
if(arr[left]>arr[right]){  
temp.push\_back(arr[right]);  
right++;  
}  
else{  
temp.push\_back(arr[left]);  
left++;  
}  
}  
while(left<=m){  
temp.push\_back(arr[left++]);  
}  
while(right<=r){  
temp.push\_back(arr[right++]);  
}  
for(int i=l;i<=r;i++){  
arr[i]=temp[i-l];  
}  
  
}  
public:  
void mergeSort(int arr[], int l, int r)  
{  
if(l == r) return;  
int m=(l+r)>>1;  
mergeSort(arr,l,m);  
mergeSort(arr,m+1,r);  
merge(arr,l,m,r);  
}  
};*

*CODE 2)*

*:class solution*

*{*

*public:*

*void merge(int arr[], int l, int m, int r)*

*{*

*// Your code here*

*int n1=m-l+1;*

*int n2=r-m;*

*int a[n1];*

*int b[n2];*

*for(int i=0;i<n1;i++){*

*a[i]=arr[l+i];*

*}*

*for(int j=0;j<n2;j++){*

*b[j]=arr[m+1+j];*

*}*

*int i=0,j=0,k=l;*

*while(i<n1 && j<n2){*

*if(a[i]<=b[j]){*

*arr[k++]=a[i++];*

*}else{*

*arr[k++]=b[j++];*

*}*

*}*

*while(i<n1){*

*arr[k++]=a[i++];*

*}*

*while(j<n2){*

*arr[k++]=b[j++];*

*}*

*}*

*public:*

*void mergeSort(int arr[], int l, int r)*

*{*

*//code here*

*if(l<r){*

*int mid=l+(r-l)/2;*

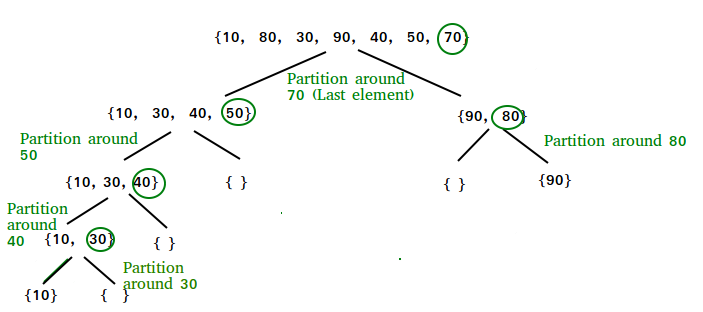
*mergeSort(arr,l,mid);*

*mergeSort(arr,mid+1,r);*

*merge(arr,l,mid,r);*

*}*

*}*

**6. Quick Sort:** It is alse based on **Divide and Conquer algorithm**. It picks an element as pivot and partitions the given array around the picked pivot.  


void quick\_sort(int \*arr, int start, int end){

if(start>=end)

return;

int p=partition(arr,start,end);

quick\_sort(arr,start,p-1);

quick\_sort(arr,p+1,end);

}

void partition(int \*arr,int start,int end){

int i=start-1;

int j=start;

int pivot=arr[end];

for(;j<=end-1;){

if(arr[j]<=pivot){

i++;

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

}

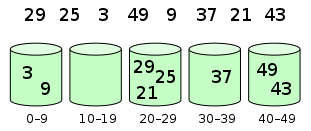
j=j+1;

}

swap(arr[i+1],arr[end]);

return i+1;

}

**7. Bucket Sort:** Bucket sort, or bin sort, is a sorting algorithm that works by distributing the elements of an array into a number of buckets.  


void bucket\_sort(int arr[], int n){

vector<int> a[101];

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

int a[i] = n \* arr[i];

a[a[i]].push\_back(arr[i]);

}

for(int i=100;i>=0;i--){

for(auto it:a)

cout<<it<<" ";

}

}

**8. Wave Sort:** Given an unsorted array of integers, sort the array into a wave like array. An array ‘arr[0..n-1]’ is sorted in wave form if arr[0] >= arr[1] <= arr[2] >= arr[3] <= arr[4] >= …..

void wave\_sort(int arr[], int n){

for(int i=0;i<n;i+=2){

if(i!=0 && arr[i]<arr[i-1])

swap(arr[i],arr[i-1]);

if(i!=(n-1) && arr[i]<arr[i+1])

swap(arr[i],arr[i+1]);

}

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

cout<<arr[i]<<" ";

}

}

**9. Single Pass (DNF):** Given an array arr[] consisting 0s, 1s and 2s. Sorting is required for this special case.

void dnf\_sort(int arr[], int n){

int low=0;

int high=n-1;

int mid=0;

while(mid<=high){

if(arr[mid]==0){

swap(arr[mid],arr[low]);

low++;

mid++;

}

if(arr[mid]==1)

mid++;

if(arr[mid]==2){

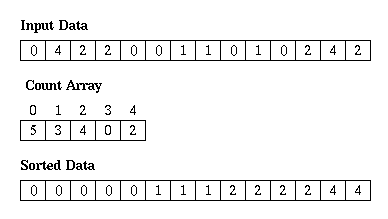
swap(arr[mid],arr[high]);

high--;

}

}

}

**10. Counting Sort:** It is a sorting technique based on keys between a specific range.  


void count\_sort(int arr[],int n){

int output[n];

int count[10];

memset(count, 0, sizeof(count));

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

++count[arr[i]];

for (i = 1; i<10;i++)

count[i] += count[i - 1];

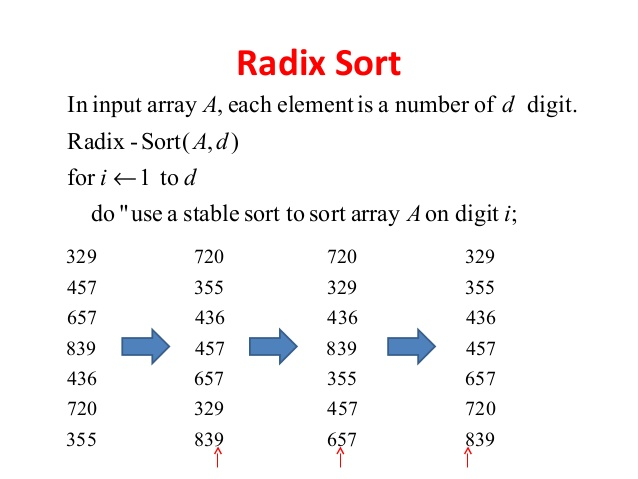
for (i = 0;i<n; ++i) {

output[count[arr[i]] - 1] = arr[i];

--count[arr[i]];

}

}

**11. Radix Sort:** The idea of it is to do digit by digit sort starting from least significant digit to most significant digit. It is upgraded form of counting sort.  


void count\_sort(int arr[],int n){

int output[n];

int count[10];

memset(count, 0, sizeof(count));

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

++count[(arr[i]/exp)%10];

for (i = 1; i<10;i++)

count[i] += count[i - 1];

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

output[count[(arr[i]/exp)%10]-1] = arr[i];

--count[(arr[i]/exp)%10];

}

}

void radix\_sort(int arr[],int n){

int max\_ele=INT\_MIN;

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

max\_ele=max(get\_max,arr[i]);

for(int exp=1;max\_ele/exp>=0;exp\*=10)

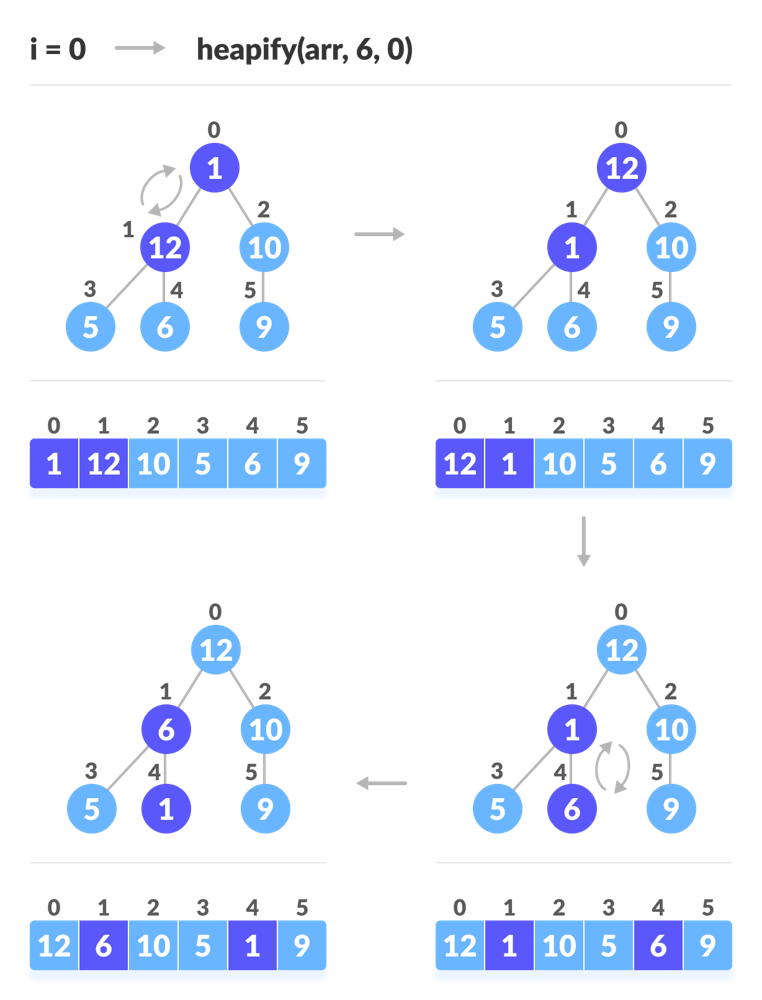
count\_sort(arr,n,exp);

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

cout<<arr[i]<<" ";

}

}

**12. Heap Sort:** It is a **comparison based sorting** technique based on Binary Heap data structure.  


int arr[10]= {0,1,3,17,2,30,7,25,19}

void heap\_sort(int idx){

int left=2\*idx;

int right=2\*idx+1;

int min\_idx=idx;

int last=arr.size()-1;

if(left <= last && compare(arr[left],arr[idx])

min\_idx=left;

if(right <= last && compare(arr[right],arr[idx])

min\_idx=right;

if(min\_idx!=idx){

swap(arr[idx],arr[min\_idx]);

heap\_sort(min\_idx);

}

bool compare(int a,int b){

if(minHeap)

return a<b;

else

return a>b;

}

**13. Tree Sort:** It is a sorting algorithm that is based on Binary Search Tree data structure.

void tree\_sort(node \*root, int arr[], int &index) {

if (root != NULL){

tree\_sort(root->left, arr, i);

arr[i++] = root->key;

tree\_sort(root->right, arr, i);

}

}

**14. Topoplogical Sort:** It is for Directed Acyclic Graph (DAG) only.

Graph:  
map<T,lis>l;

Using DFS

template <typename T>

void dfs(){

map<T,bool> visited;

list <T>ordering;

for(auto p : l){

T node=p.first;

visited[node]=fasle;

}

for(auto p : l){

T node=p.fisrt;

if(!visited[node])

dfs\_helper(node,visited,ordering);

}

for(auto node:ordering){

cout<<node<<" ";

}

}

void dfs\_helper(T src,map <T,bool> &visited,list<T> &ordering){

visited[src]=true;

for(T nbr : l[src]){

if(!visited[nbr]

dfs\_helper(nbr,visited,ordering);

}

ordering.push\_front(src);

return ;

}

Using BFS

void topological\_sort(){

int \*indegree=new int[V];

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

indegree[i]=0;

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

int x=p.first();

for(auto y:l[i])

indegree[y]++;

}

queue <int> q

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

if(inegree[i]==0)

q.push(i);

}

while(!q.empty()){

int node=q.front()'

cout<<node<<" ";

q.pop();

for(auto nbr:l[node]){

indegree[nbr]--;

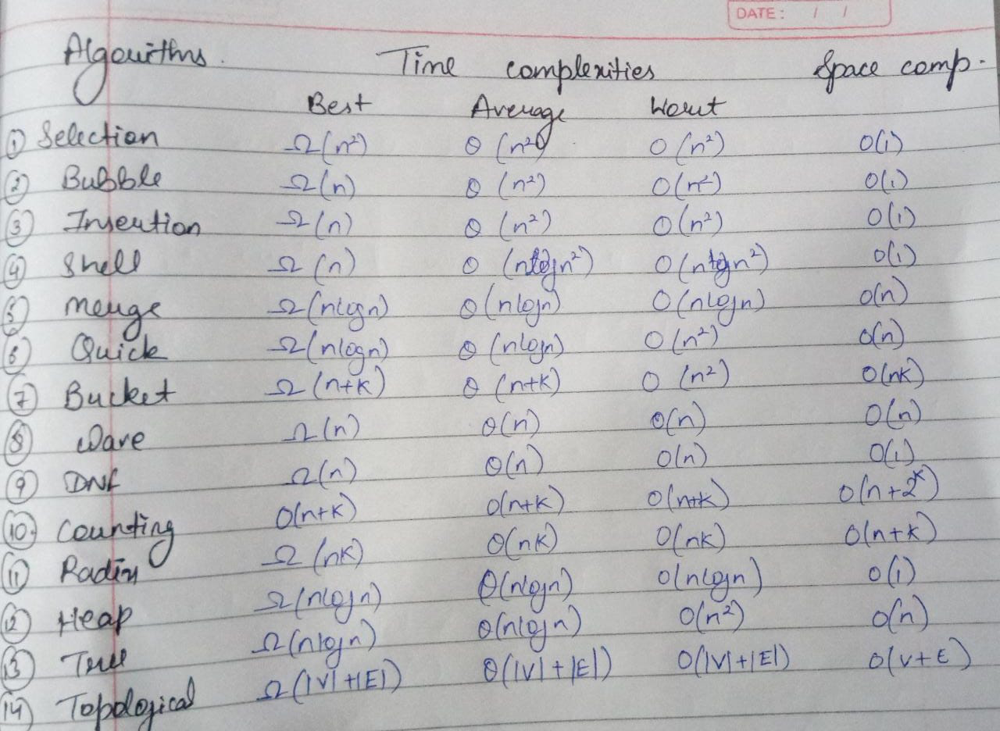
if(inegree[i]==0)

q.push(i);

}

}

}

**Time and Space Complexities of all the above algorithms:**  


BINARY ALGO USING RECURSION

int func(int arr[],int l,int r,int k)

{

if(r>=l)

{

int mid=l+(r-l)/2;

if(arr[mid]==k)

return mid;

if(arr[mid]>k)

return func(arr,l,mid-1,k);

return func(arr,mid+1,r,k);

}

return -1;

}

int binarysearch(int arr[], int n, int k){

// code here

return func(arr,0,n-1,k);

}

250) Given an integer n, return the number of prime numbers that are strictly less than n.

**Example 1:**

**Input:** n = 10

**Output:** 4

**Explanation:** There are 4 prime numbers less than 10, they are 2, 3, 5, 7.

: int countPrimes(int n) {

if(n==0||n==1)

return 0;

vector<int>sieve(n,1);

sieve[0]=0;

sieve[1]=0;

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

{

if(sieve[i])

{

for(int j=i\*i;j<n;j+=i)

{

sieve[j]=0;

}

}

}

return count(sieve.begin(),sieve.end(),1);

}

251) 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.

**Example 1:**

**Input:**

N = 4

Jobs = {(1,4,20),(2,1,10),(3,1,40),(4,1,30)}

**Output:**

2 60

**Explanation:**

Job1 and Job3 can be done with

maximum profit of 60 (20+40).

: static bool comp(Job a, Job b){

return a.profit > b.profit;

}

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

{

// your code here

sort(arr,arr+n,comp);

int day=0,profit=0;

bool done[n] = {0};

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

for(int j=min(n,arr[i].dead-1);j>=0;j--){

if(done[j]==false){

day+=1;

profit+=arr[i].profit;

done[j]=true;

break;

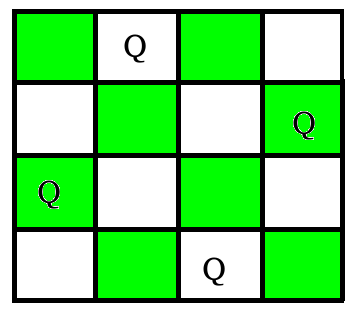
}

}

}

return {day,profit};

}

252) The n-queens puzzle is the problem of placing **n** queens on a (**n×n)** chessboard such that no two queens can attack each other.  
Given an integer n, find all distinct solutions to the n-queens puzzle. Each solution contains distinct board configurations of the n-queens’ placement, where the solutions are a permutation of [1,2,3..n] in increasing order, here the number in the *ith* place denotes that the *ith*-column queen is placed in the row with that number. For eg below figure represents a chessboard [3 1 4 2].  
  


**Example 1:**

**Input:**

1

**Output:**

[1]

**Explaination:**

Only one queen can be placed

in the single cell available.

**Example 2:**

**Input:**

4

**Output:**

[2 4 1 3 ] [3 1 4 2 ]

**Explaination:**

These are the 2 possible solutions.

: int arr[11];

// vector<int>temp;

bool place(int k, int i)

{

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

{

if(arr[j]==i || (abs(arr[j]-i) == abs(j-k)))

return false;

}

return true;

}

void check(int k ,int n, vector<vector<int>>&ans)

{

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

{

//memset(temp, 0, sizeof(temp));

vector<int>temp;

if(place(k, i))

{

arr[k]=i;

if(k==n)

{

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

{

temp.push\_back(arr[j]);

}

ans.push\_back(temp);

}

else

{

check(k+1, n, ans);

}

}

}

}

vector<vector<int>> nQueen(int n) {

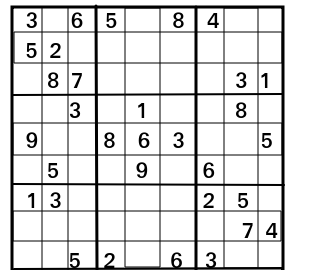
// code here

vector<vector<int>>ans;

check(1, n, ans);

return ans;

}

253) Given an incomplete Sudoku configuration in terms of a 9 x 9  2-D square matrix (grid[][]), the task to find a solved Sudoku. For simplicity, you may assume that there will be only one unique solution.  
  
**Sample Sudoku for you to get the logic for its solution:**  
  


**Example 1:**

**Input:**

grid[][] =

[[3 0 6 5 0 8 4 0 0],

[5 2 0 0 0 0 0 0 0],

[0 8 7 0 0 0 0 3 1 ],

[0 0 3 0 1 0 0 8 0],

[9 0 0 8 6 3 0 0 5],

[0 5 0 0 9 0 6 0 0],

[1 3 0 0 0 0 2 5 0],

[0 0 0 0 0 0 0 7 4],

[0 0 5 2 0 6 3 0 0]]

**Output:**

3 1 6 5 7 8 4 9 2

5 2 9 1 3 4 7 6 8

4 8 7 6 2 9 5 3 1

2 6 3 4 1 5 9 8 7

9 7 4 8 6 3 1 2 5

8 5 1 7 9 2 6 4 3

1 3 8 9 4 7 2 5 6

6 9 2 3 5 1 8 7 4

7 4 5 2 8 6 3 1 9

: bool is\_safe(int grid[9][9],int no,int i,int j) {

for(int x=0;x<9;x++) {

if(grid[i][x]==no || grid[x][j]==no) {

return false;

}

}

int sx = (i/3)\*3;

int sy = (j/3)\*3;

for(int x=sx;x<sx+3;x++) {

for(int y=sy;y<sy+3;y++) {

if(grid[x][y]==no) {

return false;

}

}

}

return true;

}

bool solve\_soduku(int grid[9][9],int i,int j) {

if(i==9) {

return true;

}

if(j==9) {

return solve\_soduku(grid,i+1,0);

}

if(grid[i][j]!=0) {

return solve\_soduku(grid,i,j+1);

}

for(int no=1;no<=9;no++) {

if(is\_safe(grid,no,i,j)) {

grid[i][j] = no;

if(solve\_soduku(grid,i,j+1)) {

return true;

}

}

}

grid[i][j] = 0;

return false;

}

bool SolveSudoku(int grid[N][N])

{

// Your code here

return solve\_soduku(grid,0,0);

}

//Function to print grids of the Sudoku.

void printGrid (int grid[N][N])

{

// Your code here

for(int i=0;i<9;i++) {

for(int j=0;j<9;j++) {

cout<<grid[i][j]<<" ";

}

}

}

254) Given an undirected graph and an integer **M**. The task is to 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. Here coloring of a graph means the assignment of colors to all vertices. Print 1 if it is possible to colour vertices and 0 otherwise.

**Example 1:**

**Input:**

N = 4

M = 3

E = 5

Edges[] = {(0,1),(1,2),(2,3),(3,0),(0,2)}

**Output:** 1

**Explanation:** It is possible to colour the

given graph using 3 colours.

: bool help(bool mat[101][101],int n,int c,vector<int> &col,int V){

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

if(i == n) continue;

if(mat[n][i] == 1 and col[i] == c )

return false;

}

return true;

}

bool coloring(bool mat[101][101],int n,vector<int> &col,int V,int m){

if( n == V ) { return true; }

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

if(help(mat,n,i,col,V) == true){

col[n] = i;

if(coloring(mat , n + 1 , col , V , m)) return true;

col[n] = -1;

}

}

return false;

}

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

{

vector<int> col(V,-1);

return coloring(graph,0,col,V,m);

}

255) Given two sorted arrays **arr1** and **arr2** of size **N** and **M** respectively and an element **K**. The task is to find the element that would be at the k’th position of the final sorted array.

**Example 1:**

**Input:**

arr1[] = {2, 3, 6, 7, 9}

arr2[] = {1, 4, 8, 10}

k = 5

**Output:**

6

**Explanation:**

The final sorted array would be -

1, 2, 3, 4, 6, 7, 8, 9, 10

The 5th element of this array is 6.

: int kthElement(int arr1[], int arr2[], int n, int m, int k)

{

vector<int>ans;

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

{

ans.push\_back(arr1[i]);

}

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

{

ans.push\_back(arr2[i]);

}

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

return ans[k-1];

}

ANOTHER APPROACH

: int kthElement(int arr1[], int arr2[], int n, int m, int k)

{

if(n==0)return arr2[k-1];

if(m==0)return arr1[k-1];

if(k==1)return min(arr1[0],arr2[0]);

int i=min(k/2,n);

int j=min(k/2,m);

if(arr1[i-1]<arr2[j-1])

return kthElement(arr1+i,arr2,n-i,m,k-i);

else

return kthElement(arr1,arr2+j,n,m-j,k-j);

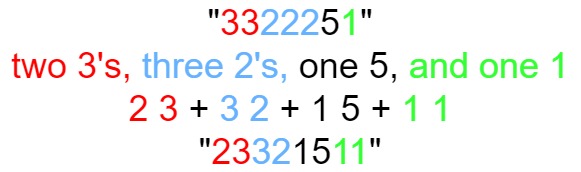
}

256) 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 groups so that each group is a contiguous section all of the **same character.** Then for each group, say the number of characters, then say the character. To convert the saying into a digit string, replace the counts with a number and concatenate every saying.

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*.

**Example 1:**

**Input:** n = 1

**Output:** "1"

**Explanation:** This is the base case.

**Example 2:**

**Input:** n = 4

**Output:** "1211"

**Explanation:**

countAndSay(1) = "1"

countAndSay(2) = say "1" = one 1 = "11"

countAndSay(3) = say "11" = two 1's = "21"

countAndSay(4) = say "21" = one 2 + one 1 = "12" + "11" = "1211": string countAndSay(int n) {

if(n==1)return "1";

if(n==2)return "11";

string str="11";

for(int i=3;i<=n;i++)

{

str+="#";

int cnt=1;

string str1="";

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

{

if(str[j]!=str[j-1])

{

str1+=cnt+'0';

str1+=str[j-1];

cnt=1;

}

else

cnt++;

}

str=str1;

}

return str;

}

ANOTHER APPROACH

: string countAndSay(int n) {

if(n==0)

return "";

string ans="1";

while(--n)

{

string curr="";

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

{

int cnt=1;

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

{

cnt++;

i++;

}

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

}

ans=curr;

}

return ans;

}

STREAM OF PROBLEMS ON NEXT MINIMUM ELEMENT CONCEPT

257) Given the array prices where prices[i] is the price of the ith item in a shop. There is a special discount for items in the shop, if you buy the ith item, then you will receive a discount equivalent to prices[j] where j is the **minimum** index such that j > i and prices[j] <= prices[i], otherwise, you will not receive any discount at all.

*Return an array where the ith element is the final price you will pay for the ith item of the shop considering the special discount.*

**Example 1:**

**Input:** prices = [8,4,6,2,3]

**Output:** [4,2,4,2,3]

**Explanation:**

For item 0 with price[0]=8 you will receive a discount equivalent to prices[1]=4, therefore, the final price you will pay is 8 - 4 = 4.

For item 1 with price[1]=4 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 4 - 2 = 2.

For item 2 with price[2]=6 you will receive a discount equivalent to prices[3]=2, therefore, the final price you will pay is 6 - 2 = 4.

For items 3 and 4 you will not receive any discount at all.

: vector<int> finalPrices(vector<int>& prices) {

vector<int>ans;

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

{

while(ans.size()&&prices[ans.back()]>=prices[i])

{

prices[ans.back()]-=prices[i];

ans.pop\_back();

}

ans.push\_back(i);

}

return prices;

}

258) You are given the head of a singly linked-list. The list can be represented as:

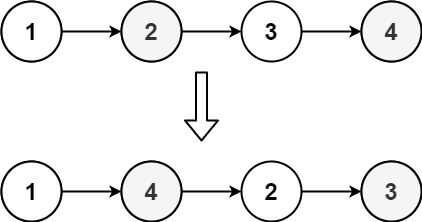
L0 → L1 → … → Ln - 1 → Ln

*Reorder the list to be on the following form:*

L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …

You may not modify the values in the list's nodes. Only nodes themselves may be changed.

**Example 1:**



**Input:** head = [1,2,3,4]

**Output:** [1,4,2,3]: void reorderList(ListNode\* head) {

void reorderList(ListNode\* head) {

ListNode\*temp;

while(head&&head->next&&head->next->next)

{

temp=head;

while(temp->next->next)

temp=temp->next;

temp->next->next=head->next;

head->next=temp->next;

temp->next=NULL;

head=head->next->next;

}

}

259) There is a biker going on a road trip. The road trip consists of n + 1 points at different altitudes. The biker starts his trip on point 0 with altitude equal 0.

You are given an integer array gain of length n where gain[i] is the **net gain in altitude** between points i​​​​​​ and i + 1 for all (0 <= i < n). Return *the****highest altitude****of a point.*

**Example 1:**

**Input:** gain = [-5,1,5,0,-7]

**Output:** 1

**Explanation:** The altitudes are [0,-5,-4,1,1,-6]. The highest is 1

: int largestAltitude(vector<int>& gain) {

int maxi=0;

int sum=0;

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

{

sum+=gain[i];

maxi=max(maxi,sum);

}

return maxi;

}

### 260) **1528. Shuffle String**

[My Submissions](https://leetcode.com/contest/weekly-contest-199/problems/shuffle-string/submissions/)[Back to Contest](https://leetcode.com/contest/weekly-contest-199/)

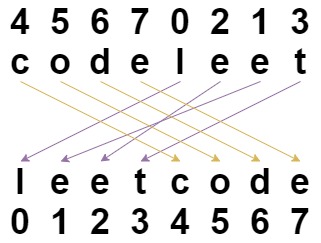
* **User Accepted:**6697
* **User Tried:**6862
* **Total Accepted:**6810
* **Total Submissions:**8084
* **Difficulty:Easy**

Given a string s and an integer array indices of the **same length**.

The string s will be shuffled such that the character at the ith position moves to indices[i] in the shuffled string.

Return the shuffled string.

**Example 1:**



**Input:** s = "codeleet", indices = [4,5,6,7,0,2,1,3]

**Output:** "leetcode"

**Explanation:** As shown, "codeleet" becomes "leetcode" after shuffling.

: string restoreString(string s, vector<int>& indices) {

vector<pair<int,char>>v;

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

{

v.push\_back({indices[i],s[i]});

}

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

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

{

s[i]=v[i].second;

}

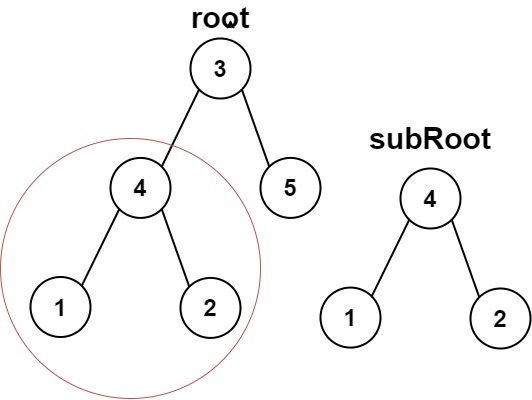
return s;

}

261) Given the roots of two binary trees root and subRoot, return true if there is a subtree of root with the same structure and node values of subRoot and false otherwise.

A subtree of a binary tree tree is a tree that consists of a node in tree and all of this node's descendants. The tree tree could also be considered as a subtree of itself.

**Example 1:**



**Input:** root = [3,4,5,1,2], subRoot = [4,1,2]

**Output:** true

: bool isSubtree(TreeNode\* root, TreeNode\* subRoot) {

if(!root)

return 0;

if(isSame(root,subRoot))

return 1;

return isSubtree(root->left,subRoot)||isSubtree(root->right,subRoot);

}

bool isSame(TreeNode\*root,TreeNode\*s)

{

if(!root&&!s)

return 1;

if(!root||!s)

return 0;

if(root->val!=s->val)

return 0;

return isSame(root->left,s->left)&&isSame(root->right,s->right);

}

TOP K FREQUENT ELEMENTS

262) Given an integer array nums and an integer k, return *the* k *most frequent elements*. You may return the answer in **any order**.

**Example 1:**

**Input:** nums = [1,1,1,2,2,3], k = 2

**Output:** [1,2]

: vector<int> topKFrequent(vector<int>& nums, int k) {

unordered\_map<int,int>mp;

for(auto it:nums)

{

mp[it]++;

}

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

for(auto it:mp)

{

pq.push({it.second,it.first});

}

vector<int>ans;

while(k--)

{

ans.push\_back(pq.top().second);

pq.pop();

}

return ans;

}

263) Given a non-negative integer x, compute and return *the square root of* x.

Since the return type is an integer, the decimal digits are **truncated**, and only **the integer part** of the result is returned.

**Note:**You are not allowed to use any built-in exponent function or operator, such as pow(x, 0.5) or x \*\* 0.5.

**Example 1:**

**Input:** x = 4

**Output:** 2

:

264)

: int mySqrt(int x) {

if(x==0||x==1)

return x;

int lo=0;

int hi=x/2;

int ans=0;

while(lo<=hi)

{

long long int mid=(lo+hi)/2;

if(mid\*mid==x)

return mid;

if(mid\*mid<x)

{

lo=mid+1;

ans=mid;

}

else

hi=mid-1;

}

return ans;

}

265) Given an array of integers nums which is sorted in ascending order, and an integer target, write a function to search target in nums. If target exists, then return its index. Otherwise, return -1.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [-1,0,3,5,9,12], target = 9

**Output:** 4

**Explanation:** 9 exists in nums and its index is 4

: int search(vector<int>& nums, int target) {

int lo=0;

int hi=nums.size()-1;

while(lo<=hi)

{

int mid=(lo+hi)/2;

if(nums[mid]==target)

return mid;

else if(nums[mid]<target)

lo=mid+1;

else

hi=mid-1;

}

return -1;

}

266) Given a m x n matrix grid which is sorted in non-increasing order both row-wise and column-wise, return *the number of****negative****numbers in* grid.

**Example 1:**

**Input:** grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]

**Output:** 8

**Explanation:** There are 8 negatives number in the matrix.

LARGE TC

: int countNegatives(vector<vector<int>>& grid) {

int ans=0;

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

{

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

{

if(grid[i][j]<0)

ans++;

}

}

return ans;

}

LOGN TC

static int countNegativesHelper(const vector<int>& vec) {

int low = 0;

int high = vec.size() - 1;

while (low <= high) {

int mid = low + (high - low) / 2;

if (vec[mid] >= 0) {

// discard everything before mid which is non-negative

low = mid + 1;

} else {

// discard everything after high which is negative

high = mid - 1;

}

}

// high is the index for the last non-negative number

// high + 1 is the number of non-negative numbers

// size - (high + 1) is the number of negative numbers

return vec.size() - high - 1;

}

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

int count = 0;

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

count += countNegativesHelper(grid[i]);

}

return count;

}

267) Let's call an array arr a **mountain** if the following properties hold:

* arr.length >= 3
* There exists some i with 0 < i < arr.length - 1 such that:
  + arr[0] < arr[1] < ... arr[i-1] < arr[i]
  + arr[i] > arr[i+1] > ... > arr[arr.length - 1]

Given an integer array arr that is **guaranteed** to be a mountain, return any i such that arr[0] < arr[1] < ... arr[i - 1] < arr[i] > arr[i + 1] > ... > arr[arr.length - 1].

**Example 1:**

**Input:** arr = [0,1,0]

**Output:** 1

: int peakIndexInMountainArray(vector<int>& arr) {

int ans=0;

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

{

if(arr[i]>arr[ans])

{

ans=i;

}

}

return ans;

}

268) A peak element is an element that is strictly greater than its neighbors.

Given an integer array nums, find a peak element, and return its index. If the array contains multiple peaks, return the index to **any of the peaks**.

You may imagine that nums[-1] = nums[n] = -∞.

You must write an algorithm that runs in O(log n) time.

**Example 1:**

**Input:** nums = [1,2,3,1]

**Output:** 2

**Explanation:** 3 is a peak element and your function should return the index number 2.

: int findPeakElement(vector<int>& nums) {

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

{

if(nums[i]<nums[i-1])

return i-1;;

}

return nums.size()-1;;

}

269) Given an array of integers nums sorted in non-decreasing order, find the starting and ending position of a given target value.

If target is not found in the array, return [-1, -1].

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [5,7,7,8,8,10], target = 8

**Output:** [3,4]

: vector<int> searchRange(vector<int>& nums, int target) {

int left\_occ=func(nums,target);

if(left\_occ==nums.size()||nums[left\_occ]!=target)

return {-1,-1};

return {left\_occ,func(nums,target+1,left\_occ)-1};

}

int func(vector<int>v,int target,int lo=0)

{

int hi=v.size()-1;

while(lo<=hi)

{

int mid=(lo+hi)>>1;

if(v[mid]<target)

lo=mid+1;

else

hi=mid-1;

}

return lo;

}

270) Given a binary array nums and an integer k, return *the maximum number of consecutive*1*'s in the array if you can flip at most* k 0's.

**Example 1:**

**Input:** nums = [1,1,1,0,0,0,1,1,1,1,0], k = 2

**Output:** 6

**Explanation:** [1,1,1,0,0,**1**,1,1,1,1,**1**]

Bolded numbers were flipped from 0 to 1. The longest subarray is underlined.

: int longestOnes(vector<int>& nums, int k) {

int i=0,j;

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

{

if(nums[j]==0)

k--;

if(k<0&&nums[i++]==0)

k++;

}

return j-i;

}

271) You are given an integer array heights representing the heights of buildings, some bricks, and some ladders.

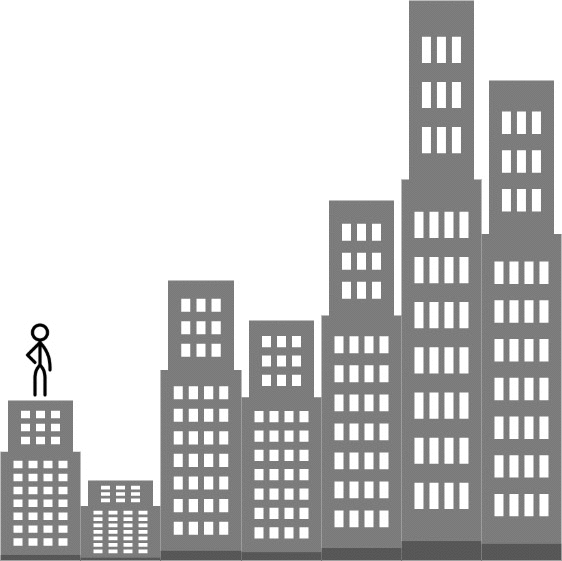
You start your journey from building 0 and move to the next building by possibly using bricks or ladders.

While moving from building i to building i+1 (**0-indexed**),

* If the current building's height is **greater than or equal** to the next building's height, you do **not** need a ladder or bricks.
* If the current building's height is **less than** the next building's height, you can either use **one ladder** or (h[i+1] - h[i]) **bricks**.

*Return the furthest building index (0-indexed) you can reach if you use the given ladders and bricks optimally.*

**Example 1:**



**Input:** heights = [4,2,7,6,9,14,12], bricks = 5, ladders = 1

**Output:** 4

**Explanation:** Starting at building 0, you can follow these steps:

- Go to building 1 without using ladders nor bricks since 4 >= 2.

- Go to building 2 using 5 bricks. You must use either bricks or ladders because 2 < 7.

- Go to building 3 without using ladders nor bricks since 7 >= 6.

- Go to building 4 using your only ladder. You must use either bricks or ladders because 6 < 9.

It is impossible to go beyond building 4 because you do not have any more bricks or ladders.

: int furthestBuilding(vector<int>& heights, int bricks, int ladders) {

priority\_queue<int>pq;

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

{

int d=heights[i+1]-heights[i];

if(d>0)

pq.push(-d);

if(pq.size()>ladders)

{

bricks+=pq.top();

pq.pop();

}

if(bricks<0)

return i;

}

return heights.size()-1;

}

272) Given a sorted array of distinct integers and a target value, return the index if the target is found. If not, return the index where it would be if it were inserted in order.

You must write an algorithm with O(log n) runtime complexity.

**Example 1:**

**Input:** nums = [1,3,5,6], target = 5

**Output:** 2

: int searchInsert(vector<int>& nums, int target) {

int lo=0;

int hi=nums.size()-1;

while(lo<=hi)

{

int mid=(lo+hi)/2;

if(nums[mid]==target)

return mid;

else if(nums[mid]>target)

hi=mid-1;

else

lo=mid+1;

}

return lo;

}

273) Given a **1-indexed** array of integers numbers that is already ***sorted in non-decreasing order***, find two numbers such that they add up to a specific target number. Let these two numbers be numbers[index1] and numbers[index2] where 1 <= index1 < index2 <= numbers.length.

Return*the indices of the two numbers,*index1*and*index2*,****added by one****as an integer array*[index1, index2]*of length 2.*

The tests are generated such that there is **exactly one solution**. You **may not** use the same element twice.

**Example 1:**

**Input:** numbers = [2,7,11,15], target = 9

**Output:** [1,2]

**Explanation:** The sum of 2 and 7 is 9. Therefore, index1 = 1, index2 = 2. We return [1, 2].

: vector<int> twoSum(vector<int>& numbers, int target) {

int l=0;

int r=numbers.size()-1;

while(l<r)

{

if(numbers[l]+numbers[r]==target)

return {l+1,r+1};

else if(numbers[l]+numbers[r]>target)

r--;

else

l++;

}

return {-1,-1};

}

274) You are a product manager and currently leading a team to develop a new product. Unfortunately, the latest version of your product fails the quality check. Since each version is developed based on the previous version, all the versions after a bad version are also bad.

Suppose you have n versions [1, 2, ..., n] and you want to find out the first bad one, which causes all the following ones to be bad.

You are given an API bool isBadVersion(version) which returns whether version is bad. Implement a function to find the first bad version. You should minimize the number of calls to the API.

**Example 1:**

**Input:** n = 5, bad = 4

**Output:** 4

**Explanation:**

call isBadVersion(3) -> false

call isBadVersion(5) -> true

call isBadVersion(4) -> true

Then 4 is the first bad version.

: int firstBadVersion(int n) {

if(!isBadVersion(n))

return n+1;

return firstBadVersion(n-1);

}

275) Given two strings s and t, return true*if*s*is a****subsequence****of*t*, or*false*otherwise*.

A **subsequence** of a string is a new string that is formed from the original string by deleting some (can be none) of the characters without disturbing the relative positions of the remaining characters. (i.e., "ace" is a subsequence of "abcde" while "aec" is not).

:DP WALA APPROACH

bool isSubsequence(string s, string t) {

int dp[s.size()+1][t.size()+1];

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

{

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

{

if(i==0||j==0)

dp[i][j]=0;

}

}

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

{

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

{

if(s[i-1]==t[j-1])

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

else

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

}

}

if(dp[s.size()][t.size()]==s.size())

return 1;

else

return 0;

}

2-P0INTER APPROACH

bool isSubsequence(string s, string t) {

int i=0;

int j=0;

while(i<s.size()&&j<t.size())

{

if(s[i]==t[j])

{

i++;

}

j++;

}

return i==s.size()?1:0;

}

276) You are given an integer array cost where cost[i] is the cost of ith step on a staircase. Once you pay the cost, you can either climb one or two steps.

You can either start from the step with index 0, or the step with index 1.

Return *the minimum cost to reach the top of the floor*.

**Example 1:**

**Input:** cost = [10,15,20]

**Output:** 15

**Explanation:** You will start at index 1.

- Pay 15 and climb two steps to reach the top.

The total cost is 15.

: int minCostClimbingStairs(vector<int>& cost) {

for(int i=2;i<cost.size();i++)

{

cost[i]+=min(cost[i-1],cost[i-2]);

}

return min(cost[cost.size()-1],cost[cost.size()-2]);

}

277) You are given an integer n. A **0-indexed** integer array nums of length n + 1 is generated in the following way:

* nums[0] = 0
* nums[1] = 1
* nums[2 \* i] = nums[i] when 2 <= 2 \* i <= n
* nums[2 \* i + 1] = nums[i] + nums[i + 1] when 2 <= 2 \* i + 1 <= n

Return*the****maximum****integer in the array*nums​​​.

**Example 1:**

**Input:** n = 7

**Output:** 3

**Explanation:** According to the given rules:

nums[0] = 0

nums[1] = 1

nums[(1 \* 2) = 2] = nums[1] = 1

nums[(1 \* 2) + 1 = 3] = nums[1] + nums[2] = 1 + 1 = 2

nums[(2 \* 2) = 4] = nums[2] = 1

nums[(2 \* 2) + 1 = 5] = nums[2] + nums[3] = 1 + 2 = 3

nums[(3 \* 2) = 6] = nums[3] = 2

nums[(3 \* 2) + 1 = 7] = nums[3] + nums[4] = 2 + 1 = 3

Hence, nums = [0,1,1,2,1,3,2,3], and the maximum is max(0,1,1,2,1,3,2,3) = 3.

: int getMaximumGenerated(int n) {

if(n==0||n==1)

return n;

int maxi=1;

int dp[n+1];

dp[0]=0;

dp[1]=1;

for(int i=2;i<=n;i++)

{

dp[i]=(i%2==0)?dp[i/2]:dp[i/2]+dp[i/2+1];

maxi=max(maxi,dp[i]);

}

return maxi;

}

(1<it)<=>(it%2)

278) Given a string s which consists of lowercase or uppercase letters, return *the length of the****longest palindrome*** that can be built with those letters.

Letters are **case sensitive**, for example, "Aa" is not considered a palindrome here.

**Example 1:**

**Input:** s = "abccccdd"

**Output:** 7

**Explanation:**

One longest palindrome that can be built is "dccaccd", whose length is 7.

: int longestPalindrome(string s) {

int freq[256]={};

for(auto it:s)

freq[it]++;

int oddgrp=0;

for(auto it:freq)

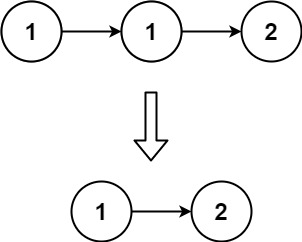
oddgrp+=(1&it);

return s.size()-oddgrp+(oddgrp>0);

}

279) Given the head of a sorted linked list, *delete all duplicates such that each element appears only once*. Return *the linked list****sorted****as well*.

**Example 1:**



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

**Output:** [1,2]

: ListNode\* deleteDuplicates(ListNode\* head) {

ListNode\*temp=head;

while(temp)

{

while(temp->next&&temp->val==temp->next->val)

temp->next=temp->next->next;

temp=temp->next;

}

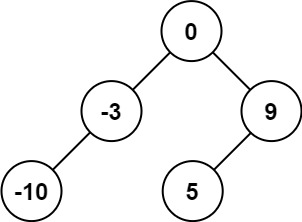
return head;

}

280) Given an integer array nums where the elements are sorted in **ascending order**, convert *it to a****height-balanced****binary search tree*.

A **height-balanced** binary tree is a binary tree in which the depth of the two subtrees of every node never differs by more than one.

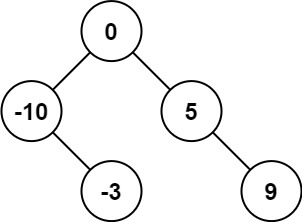
**Example 1:**



**Input:** nums = [-10,-3,0,5,9]

**Output:** [0,-3,9,-10,null,5]

**Explanation:** [0,-10,5,null,-3,null,9] is also accepted:



: TreeNode\*from\_middle(vector<int>&nums,int s,int e)

{

if(s>=e)

return NULL;

int mid\_index=(s+e)/2;

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

root->left=from\_middle(nums,s,mid\_index);

root->right=from\_middle(nums,mid\_index+1,e);

return root;

}

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

return from\_middle(nums,0,nums.size());

}

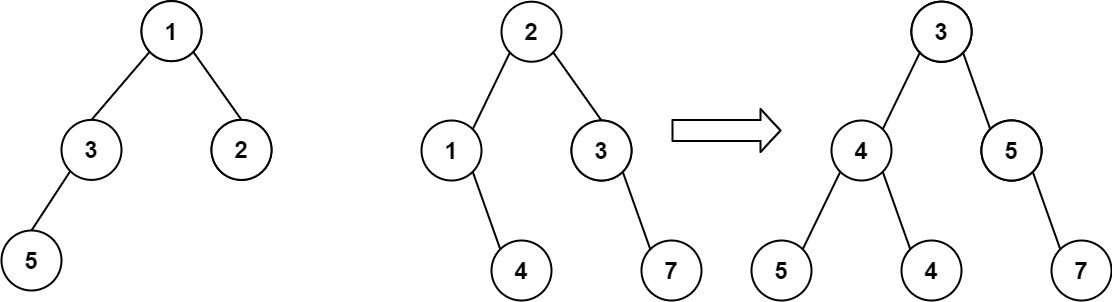
281) You are given two binary trees root1 and root2.

Imagine that when you put one of them to cover the other, some nodes of the two trees are overlapped while the others are not. You need to merge the two trees into a new binary tree. The merge rule is that if two nodes overlap, then sum node values up as the new value of the merged node. Otherwise, the NOT null node will be used as the node of the new tree.

Return *the merged tree*.

**Note:** The merging process must start from the root nodes of both trees.

**Example 1:**



**Input:** root1 = [1,3,2,5], root2 = [2,1,3,null,4,null,7]

**Output:** [3,4,5,5,4,null,7]

: TreeNode\* mergeTrees(TreeNode\* root1, TreeNode\* root2) {

if(!root1)

return root2;

if(!root2)

return root1;

TreeNode\*root=new TreeNode(root1->val+root2->val);

root->left=mergeTrees(root1->left,root2->left);

root->right=mergeTrees(root1->right,root2->right);

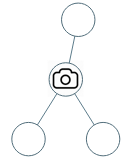
return root;

}

282) You are given the root of a binary tree. We install cameras on the tree nodes where each camera at a node can monitor its parent, itself, and its immediate children.

Return *the minimum number of cameras needed to monitor all nodes of the tree*.

**Example 1:**



**Input:** root = [0,0,null,0,0]

**Output:** 1

**Explanation:** One camera is enough to monitor all nodes if placed as shown.

## **Intuition:**

Consider a node in the tree.  
It can be covered by its parent, itself, its two children.  
Four options.

Consider the root of the tree.  
It can be covered by left child, or right child, or itself.  
Three options.

Consider one leaf of the tree.  
It can be covered by its parent or by itself.  
Two options.

If we set a camera at the leaf, the camera can cover the leaf and its parent.  
If we set a camera at its parent, the camera can cover the leaf, its parent and its sibling.

We can see that the second plan is always better than the first.  
Now we have only one option, set up camera to all leaves' parent.

Here is our greedy solution:

1. Set cameras on all leaves' parents, thenremove all covered nodes.
2. Repeat step 1 until all nodes are covered.

## **Explanation:**

Apply a recusion function dfs.  
Return 0 if it's a leaf.  
Return 1 if it's a parent of a leaf, with a camera on this node.  
Return 2 if it's coverd, without a camera on this node.

For each node,  
if it has a child, which is leaf (node 0), then it needs camera.  
if it has a child, which is the parent of a leaf (node 1), then it's covered.

If it needs camera, then res++ and we return 1.  
If it's covered, we return 2.  
Otherwise, we return 0.

: int res=0;

int minCameraCover(TreeNode\* root) {

return (func(root)<1?1:0)+res;

}

int func(TreeNode\*root)

{

if(!root)

return 2;

int l=func(root->left);

int r=func(root->right);

if(l==0||r==0)

{

res++;

return 1;

}

return l==1||r==1?2:0;

283) Given the coordinates of four points in 2D space p1, p2, p3 and p4, return true *if the four points construct a square*.

The coordinate of a point pi is represented as [xi, yi]. The input is **not** given in any order.

A **valid square** has four equal sides with positive length and four equal angles (90-degree angles).

**Example 1:**

**Input:** p1 = [0,0], p2 = [1,1], p3 = [1,0], p4 = [0,1]

**Output:** true

: bool validSquare(vector<int>& p1, vector<int>& p2, vector<int>& p3, vector<int>& p4) {

unordered\_set<int>st({d(p1,p2),d(p1,p3),d(p1,p4),d(p2,p3),d(p2,p4),d(p3,p4)});

return !st.count(0)&&st.size()==2;

}

int d(vector<int>&p1,vector<int>&p2)

{

return (p1[0]-p2[0])\*(p1[0]-p2[0])+(p1[1]-p2[1])\*(p1[1]-p2[1]);

}

284) You are given two positive integers n and k. A factor of an integer n is defined as an integer i where n % i == 0.

Consider a list of all factors of n sorted in **ascending order**, return *the*kth*factor* in this list or return -1 if n has less than k factors.

**Example 1:**

**Input:** n = 12, k = 3

**Output:** 3

**Explanation:** Factors list is [1, 2, 3, 4, 6, 12], the 3rd factor is 3.

: int kthFactor(int n, int k) {

for(int i=1;i<=n/2;i++)

{

if(n%i==0&&--k==0)

return i;

}

return k==1?n:-1;

}

285) Given an array of **distinct** integers arr, find all pairs of elements with the minimum absolute difference of any two elements.

Return a list of pairs in ascending order(with respect to pairs), each pair [a, b] follows

* a, b are from arr
* a < b
* b - a equals to the minimum absolute difference of any two elements in arr

**Example 1:**

**Input:** arr = [4,2,1,3]

**Output:** [[1,2],[2,3],[3,4]]

**Explanation:** The minimum absolute difference is 1. List all pairs with difference equal to 1 in ascending order.

: vector<vector<int>> minimumAbsDifference(vector<int>& arr) {

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

vector<vector<int>>ans;

int mini=INT\_MAX;

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

{

mini=min(mini,arr[i+1]-arr[i]);

}

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

{

if(arr[i+1]-arr[i]==mini)

ans.push\_back({arr[i],arr[i+1]});

}

return ans;

}

286) An **ugly number** is a positive integer whose prime factors are limited to 2, 3, and 5.

Given an integer n, return true *if* n *is an****ugly number***.

**Example 1:**

**Input:** n = 6

**Output:** true

**Explanation:** 6 = 2 × 3

: bool isUgly(int n) {

if(n<=0)

return 0;

while(n%2==0)

n/=2;

while(n%3==0)

n/=3;

while(n%5==0)

n/=5;

return n==1;

}

287) Given a grid of size m \* n, lets assume you are starting at (1,1) and your goal is to reach (m,n). At any instance, if you are on (x,y), you can either go to (x, y + 1) or (x + 1, y).

Now consider if some obstacles are added to the grids. How many unique paths would there be?  
An obstacle and empty space is marked as 1 and 0 respectively in the grid.

**Example :**  
There is one obstacle in the middle of a 3x3 grid as illustrated below.

[

[0,0,0],

[0,1,0],

[0,0,0]

]

The total number of unique paths is 2.

: int Solution::uniquePathsWithObstacles(vector<vector<int> > &A) {

    int r = A.size();

    int c = A[0].size();

    // If obstacle is at starting position

    if (A[0][0])

        return 0;

    //  Initializing starting position

    A[0][0] = 1;

    // first row all are '1' until obstacle

    for (int j = 1; j < c; j++) {

        if (A[0][j] == 0) {

            A[0][j] = A[0][j - 1];

        }

        else {

            // No ways to reach at this index

            A[0][j] = 0;

        }

    }

    // first column all are '1' until obstacle

    for (int i = 1; i < r; i++) {

        if (A[i][0] == 0) {

            A[i][0] = A[i - 1][0];

        }

        else {

            // No ways to reach at this index

            A[i][0] = 0;

        }

    }

    for (int i = 1; i < r; i++) {

        for (int j = 1; j < c; j++) {

            // If current cell has no obstacle

            if (A[i][j] == 0) {

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

            }

            else {

                // No ways to reach at this index

                A[i][j] = 0;

            }

        }

    }

    // returning the bottom right

    // corner of Grid

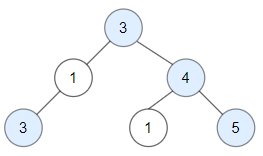
    return A[r - 1][c-1];

}

288) Given a binary tree root, a node *X* in the tree is named **good** if in the path from root to *X* there are no nodes with a value *greater than* X.

Return the number of **good** nodes in the binary tree.

**Example 1:**

****

**Input:** root = [3,1,4,3,null,1,5]

**Output:** 4

**Explanation:** Nodes in blue are **good**.

Root Node (3) is always a good node.

Node 4 -> (3,4) is the maximum value in the path starting from the root.

Node 5 -> (3,4,5) is the maximum value in the path

Node 3 -> (3,1,3) is the maximum value in the path.

: void dfs(TreeNode\*root,int maxi,int &good)

{

if(!root)

return ;

if(root->val>=maxi)

{

good++;

maxi=root->val;

}

dfs(root->left,maxi,good);

dfs(root->right,maxi,good);

}

int goodNodes(TreeNode\* root) {

int good=0;

int maxi=INT\_MIN;

dfs(root,maxi,good);

return good;

}

289) Given the head of a singly linked list, group all the nodes with odd indices together followed by the nodes with even indices, and return *the reordered list*.

The **first** node is considered **odd**, and the **second** node is **even**, and so on.

Note that the relative order inside both the even and odd groups should remain as it was in the input.

You must solve the problem in O(1) extra space complexity and O(n) time complexity.

**Example 1:**



**Input:** head = [1,2,3,4,5]

**Output:** [1,3,5,2,4]

: ListNode\* oddEvenList(ListNode\* head) {

if(!head||!head->next||!head->next->next)

return head;

ListNode\*odd=head;

ListNode\*even=head->next;

ListNode\*even\_start=head->next;

while(odd->next&&even->next)

{

odd->next=even->next;

even->next=odd->next->next;

odd=odd->next;

even=even->next;

}

odd->next=even\_start;

return head;

}

290) Given an array **arr[]** denoting heights of **N** towers and a positive integer **K**, you **have to** modify the height of each tower either by increasing or decreasing them by **K** only **once**. After modifying, height should be a **non-negative** integer.   
Find out what could be the possible minimum difference of the height of shortest and longest towers after you have modified each tower.

A slight modification of the problem can be found [here](https://practice.geeksforgeeks.org/problems/minimize-the-heights-i/1/).

**Example 1:**

**Input:**

K = 2, N = 4

Arr[] = {1, 5, 8, 10}

**Output:**

5

**Explanation:**

The array can be modified as

{3, 3, 6, 8}. The difference between

the largest and the smallest is 8-3 = 5.

: int getMinDiff(int arr[], int n, int k) {

// code here

sort(arr,arr+n);

int diff=arr[n-1]-arr[0];

int maxi,mini;

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

if(arr[i+1]<k){

continue;

}

else{

mini=min(arr[0]+k,arr[i+1]-k);

maxi=max(arr[i]+k,arr[n-1]-k);

diff=min(diff,maxi-mini);

}

}

return diff;

}

291) Given two arrays: **a1[0..n-1]** of size **n** and **a2[0..m-1]** of size **m**. Task is to check whether a2[] is a subset of a1[] or not. Both the arrays can be sorted or unsorted. It may be assumed that elements in both array are distinct.

**Example 1:**

**Input**:

a1[] = {11, 1, 13, 21, 3, 7}

a2[] = {11, 3, 7, 1}

**Output**:

Yes

**Explanation:**

a2[] is a subset of a1[]

: string isSubset(int a1[], int a2[], int n, int m) {

unordered\_map<int,int>mp;

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

mp[a1[i]]=1;

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

{

if(mp[a2[i]]>0)

mp[a2[i]]--;

else

return "No";

}

return "Yes";

}

292) Given an array arr of size n and an integer X. Find if there's a triplet in the array which sums up to the given integer X.

**Example 1:**

**Input**:

n = 6, X = 13

arr[] = [1 4 45 6 10 8]

**Output**:

1

**Explanation**:

The triplet {1, 4, 8} in

the array sums up to 13.

: bool find3Numbers(int A[], int n, int X)

{

//Your Code Here

sort(A, A+n);

for (int i = 0; i < n-3; i++)

{

int j = i+1;

int k = n - 1;

while (j < k)

{

int sum = A[i] + A[j] + A[k];

if (sum == X)

return true;

else if (sum > X)

k -= 1;

else if (sum < X)

j += 1;

}

}

return false;

}

293) Given an array **A[ ]** of positive integers of size **N**, where each value represents the number of chocolates in a packet. Each packet can have a variable number of chocolates. There are **M** students, the task is to distribute chocolate packets among **M** students such that :  
1. Each student gets **exactly** one packet.  
2. The difference between maximum number of chocolates given to a student and minimum number of chocolates given to a student is minimum.

**Example 1:**

**Input:**

N = 8, M = 5

A = {3, 4, 1, 9, 56, 7, 9, 12}

**Output:** 6

**Explanation:** The minimum difference between

maximum chocolates and minimum chocolates

is 9 - 3 = 6 by choosing following M packets :

{3, 4, 9, 7, 9}.

: long long findMinDiff(vector<long long> a, long long n, long long m){

//code

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

long long num= INT\_MAX;

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

int j= i+m-1;

if(a[j] - a[i] < num)

num =a[j]-a[i];

}

return num;

}

294) Given a Integer array A[] of n elements. Your task is to complete the function **PalinArray**. Which will return 1 if all the elements of the Array are palindrome otherwise it will return 0.

**Input:**  
The first line of input contains an integer denoting the no of test cases. Then T test cases follow. Each test case contains two lines. The first line of input contains an integer n denoting the size of the arrays. In the second line are N space separated values of the array A[].

**Output:**  
For each test case in a new line print the required result.

**Constraints:**  
1 <=T<= 50  
1 <=n<= 20  
1 <=A[]<= 10000

**Example:**  
**Input:**  
2  
5  
111 222 333 444 555  
3  
121 131 20

**Output:**  
1  
0

**Explanation:**  
For First test case.  
n=5;  
A[0] = 111    //which is a palindrome number.  
A[1] = 222   //which is a palindrome number.  
A[2] = 333   //which is a palindrome number.  
A[3] = 444  //which is a palindrome number.  
A[4] = 555  //which is a palindrome number.  
As all numbers are palindrome so This will return 1.

: int func(int n)

{

int r;

int ans=0;

while(n>0)

{

r=n%10;

ans=(ans\*10)+r;

n/=10;

}

return ans;

}

int PalinArray(int a[], int n)

{ //add code here.

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

{

if(func(a[i])!=a[i])

return 0;

}

return 1;

}

295) Given a binary array **A[]** of size **N**. The task is to arrange the array in increasing order.

**Note:** The binary array contains only 0  and 1.

**Example 1:**

**Input:**

N = 5

A[] = {1,0,1,1,0}

**Output:** 0 0 1 1 1

**Example 2:**

**Input:**

N = 10

A[] = {1,0,1,1,1,1,1,0,0,0}

**Output:** 0 0 0 0 1 1 1 1 1 1

**Your Task:**  
Complete the function **SortBinaryArray()**which takes given array as input and returns the sorted array.

**Expected Time Complexity:**O(N).  
**Expected Auxiliary Space:**O(1).

**Challenge:**Try doing it in one pass.

: vector<int> SortBinaryArray(vector<int> binArray)

{

// Your code goes here

int j=0;

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

{

if(binArray[i]==0)

{

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

j++;

}

}

return binArray;

}

296) Given a sorted and rotated array A of N distinct elements which is rotated at some point, and given an element key. The task is to find the index of the given element key in the array A.

**Example 1:**

**Input:**

N = 9

A[] = {5, 6, 7, 8, 9, 10, 1, 2, 3}

key = 10

**Output**:

5

**Explanation**: 10 is found at index 5.

**Example 2**:

**Input**:

N = 4

A[] = {3, 5, 1, 2}

key = 6

**Output**:

-1

**Explanation**: There is no element that has value 6.

**Your Task**:  
Complete the function search() which takes an array arr[] and start, end index of the array and the K as input parameters, and returns the answer.  
  
Can you solve it in expected time complexity?

**Expected Time Complexity**: O(log N).  
**Expected Auxiliary Space**: O(1).

: int search(int A[], int l, int h, int key){

//complete the function here

while(l <= h)

{

if(A[l]==key)

{

return l;

l++;

}

else

{

l++;

}

}

return -1;

}

297) Given a number N, swap the two nibbles in it and find the resulting number. 

**Example 1:**

**Input:**

N = 100

**Output:**

70

**Explanation:**

100 in binary is 01100100,

two nibbles are (0110) and (0100)

If we swap the two nibbles, we get

01000110 which is 70 in decimal

: int swapNibbles(unsigned char x){

// code here

unsigned char a=x<<4;

unsigned char b=x>>4;

return a+b;

}

298) Given a number **N,** check if a number is perfect or not. A number is said to be perfect if sum of all its factors excluding the number itself is equal to the number.

**Example 1:**

**Input:**

**N =** 6

**Output:**

1

**Explanation:**

Factors of 6 are 1, 2, 3 and 6.

Excluding 6 their sum is 6 which

is equal to N itself. So, it's a

Perfect Number.

: int isPerfectNumber(long long N)

{

// code here

if(N==1)

return 0;

int sum=1;

for(int i=2;i<=sqrt(N);i++){

if(N%i==0){

sum=sum+i;

if(i!=N/i)

sum=sum+(N/i);

}

}

if(sum==N)

return 1;

return 0;

}

299) Given two given numbers **a** and **b** where 1<=a<=b, find the number of perfect squares between a and b (a and b inclusive).

**Example 1:**

**Input:**

**a =** 9, **b =** 25

**Output:**

3

**Explanation:**

There are 3 perfect squares between 9

and 25. They are 9, 16, and 25.

: int numOfPerfectSquares(int a, int b) {

// code here

int cnt=0;

int ans=0;

for(int i=a;i<=b;i++)

{

ans=sqrt(i);

if(ans\*ans==i)

cnt++;

}

return cnt;

}

300) Given an array of N positive integers, find GCD of all the array elements.

**Example 1:**

**Input:** N = 3, arr[] = {2, 4, 6}

**Output:** 2

**Explanation:** GCD of 2,4,6 is 2.

: int gcd(int a, int b)

{

if (a == 0)

return b;

return gcd(b % a, a);

}

int gcd(int N, int arr[])

{

// Your code goes here

int result = arr[0];

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

{

result = gcd(arr[i], result);

if(result == 1)

{

return 1;

}

}

return result;

}

AMAZON SDE-II asked question

301)

* Explaination: The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula. Dist (n1, n2) = Dist (root, n1) + Dist (root, n2) - 2\*Dist (root, lca) 'n1' and 'n2' are the two given keys 'root' is root of given Binary Tree. 'lca' is lowest common ancestor of n1 and n2 Dist (n1, n2) is the distance between n1 and n2.
* The distance between two nodes can be obtained in terms of lowest common ancestor. Following is the formula. Dist (n1, n2) = Dist (root, n1) + Dist (root, n2) - 2\*Dist (root, lca) 'n1' and 'n2' are the two given keys 'root' is root of given Binary Tree. 'lca' is lowest common ancestor of n1 and n2 Dist (n1, n2) is the distance between n1 and n2.

: dist(a,b) = Dist(root , a)+Dist(root,b) - 2\* Dist(root , lca)

int find\_dist(Node \*root,int a)

{

if(root==NULL) return -1;

int dist=-1;

if(root->data==a ||

(dist=find\_dist(root->left,a)) >=0||

(dist=find\_dist(root->right,a))>=0

)

return dist+1;

return dist;

}

Node \*lca(Node \* root, int a,int b)

{

if(root==NULL)

return root;

if(root->data==a || root->data==b)

return root;

Node \*L=lca(root->left,a,b);

Node \*R=lca(root->right,a,b);

if(L!=NULL && R!=NULL)

return root;

if(L==NULL)

return R;

else

return L;

}

int findDist(Node\* root, int a, int b)

{

int n1=find\_dist(root , a);

int n2=find\_dist(root,b);

Node \*temp=lca(root,a,b);

int n3=find\_dist(root,temp->data);

return n1+n2-2\*(n3);

}