

tDSA:

[Math Logics:](#)

[Lagrange's Four Square Theorem](#)

[Sorting Algorithms:](#)

[Graph Algorithms' Time Complexity](#)

[Backtracking:](#)

[Printing all solutions in N-Queen Problem](#)

[Word Break Problem using Backtracking](#)

[Remove Invalid Parentheses](#)

[Sudoku Solver](#)

[M-Coloring Problem](#)

[Print all Palindromic Partitions of a String using Backtracking](#)

[Subset Sum Problem](#)

[The Knight's tour problem](#)

[Tug of War](#)

[Find shortest safe route in a path with landmines](#)

[Combinational Sum](#)

[Find Maximum number possible by doing at-most K swaps](#)

[Print all permutations of a string](#)

[Find if there is a path of more than k length from a source](#)

[Longest Possible Route in a Matrix with Hurdles](#)

[Print all possible paths from top left to bottom right of a mXn matrix](#)

[Partition of a set into K subsets with equal sum](#)

[Find the K-th Permutation Sequence of first N natural numbers](#)

[Greedy:](#)

[N meetings in one room](#)

[Job Sequencing Problem](#)

[Fractional Kancsak Problem:](#)

[Greedy Algorithm to find Minimum number of Coins or](#)

[Coin Change – Minimum Coins to Make Sum](#)

[Maximum trains for which stoppage can be provided](#)

[Minimum Platforms Required for Given Arrival and Departure Times](#)

[Buy Maximum Stocks if i stocks can be bought on i-th day](#)

[Min and Max Costs to Buy All Candies](#)

[Minimize Cash Flow among a given set of friends who have borrowed money from each other](#)

[Dynamic Programming:](#)

[0 - 1 Knapsack Problem](#)

[Edit Distance](#)

[Partition Equal Subset Sum](#)

[Longest Common Subsequence](#)

[Longest Common Substring](#)

[Coin Change \(Calculate total subsets to form sum\)](#)

[Pascal's Triangle:](#)

[Catalean Number](#)

[Gold Mine Problem:](#)

[Subset Sum Problem](#)

[Find the first non-repeating character from a stream of characters](#)

[Stack and Queues](#)

[Design a Stack that supports getMin\(\) in O\(1\) time and O\(1\) extra space.](#)

[Next Greater Element](#)

[The Celebrity Problem](#)

[Arithmetic Expression Using Stack](#)

[Evaluate a Postfix Expression](#)

[Reverse a stack using recursion](#)

[Sort A Stack Using Recursion:](#)

[Arrays](#)

[Minimum swaps to sort an array](#)

[Sort an array of 0s, 1s and 2s](#)

[Move all negative elements to end without disturbing order:](#)

[Kadane's Algorithm \(Longest Contiguous Subarray\)](#)

[Minimize the Heights: add or subtract k to every element](#)

[Min Jumps to Read End](#)

[Find the duplicate number:](#)

[Merge Without using extra space:](#)

[Merge Intervals:](#)

[Best time to buy and sell stocks](#)

[Next Permutation:](#)

[Count Inversion in an array:](#)

[Count All Pairs with given sum:\(2sum Problem\)](#)

[Common elements in 3 sorted array:](#)

[Subarray With Sum 0 ★](#)

[Maximum product subarray:](#)

[Longest Consecutive Subsequence:](#)

[Buy and Sell stocks at max two transactions:](#)

[Find whether an array is subset of another array](#)

[3 Sum – Triplet Sum in Array](#)

[Trapping Rain Water](#)

[Chocolate Distribution Problem](#)

[Smallest subarray with sum greater than a given value](#)

[Three way partitioning](#)

[Minimum swaps required to bring all elements less than or equal to k together](#)

[Find minimum number of merge operations to make an array palindrome](#)

[Strings:](#)

[Check if given strings are rotations of each other or not](#)

[Count and Say:](#)

[Longest Palindrome String:](#)

[Rearrange Positive and Negative numbers alternatively:](#)

[Find the factorial of large number:](#)

[Find all elements that appear more than n/k time in aray of size n, and k is number](#)

[Buy And Sell Stocks \(unlimited number of transaction\)](#)

[Write a Program to check whether a string is a valid shuffle of two strings or not](#)

[Solve using unordered_map](#)

[Find Longest Common Subsequence in String](#)

[Print all subsequencs of a string](#)

[Print all permutations of String / Array](#)

[Split the binary string into substrings with equal number of 0s and 1s](#)

[Word Wrap Problem:](#)

[Parenthesis Problem](#)

[Word Break Problem: ★](#)

[Rabin Karp Algorithm for Pattern Searching](#)

[Convert a sentence into its equivalent mobile numeric keypad sequence](#)

[Minimum number of bracket reversals needed to make an expression balanced.](#)

[Count All Palindromic Subsequence in a given String.](#)

[Count of number of given string in 2D character array](#)

[Word Search in a 2D Grid of characters](#)

[Converting Roman Numerals to Decimal](#)

[Longest Common Prefix](#)

[Find the first repeated word in string.](#)

[Minimum number of swaps for bracket balancing.](#)

[Write a program tofind the smallest window that contains all characters of string itself.](#)

[Rearrange characters in a string such that no two adjacent are same](#)

[Minimum characters to be added at front to make string palindrome](#)

[Given a sequence of words, print all anagrams together](#)

[Smallest window in a string containing all the characters of another string](#)

[Recursively remove all adjacent duplicates](#)

[Recursively print all sentences that can be formed from list of word lists](#)

[Transform One String to Another using Minimum Number of Given Operation](#)

[Check if two given strings are isomorphic to each other](#)

[Function to find Number of customers who could not get a computer](#)

[String matching where one string contains wildcard characters](#)

[Linked List:](#)

[Reverse a Linked List:](#)
[Reverse a Linked List in a given group size:](#)
[Detect Loop in a Linked List:](#)
[Delete Loop in a Linked List:](#)
[Find the starting point of the Loop:](#)
[Remove Duplicated from Sorted Linked List:](#)
[Remove Duplicates from unsorted Linked List:](#)
[Bring Last element to first in Linked List:](#)
[Add “1” to number represent by linked list:](#)
[Add Two Numbers Represented by Linked List:](#)
[Intersection of two sorted Linked List:](#)
[Find the middle element of the linked list:](#)
[Check if linked list is circular:](#)
[Nth Node of a Linked List form End:](#)
[Split a Circular linked list into two halves.](#)
[Write a Program to check whether the Singly Linked list is a palindrome or not.](#)
[Deletion from a Circular Linked List](#)
[Reverse a Doubly Linked List:](#)
[Find pairs with given sum in doubly linked list](#)
[Presum, Two Pointers, Array Misc](#)
[Subarray sums divided by K:](#)
[Searching and Sorting:](#)
[Find first and last positions of an element in a sorted array](#)
[Search in rotated sorted array:](#)
[Find the square root:](#)
[Optimum location of point to minimize total distance](#)
[Find the repeating and the missing](#)
[find majority element](#)
[Searching in an array where adjacent differ by at most k](#)
[find a pair with a given difference](#)
[find four elements that sum to a given value \(4 Sum\)](#)
[maximum sum such that no 2 elements are adjacent](#)
[Count triplet with sum smaller than a given value](#)
[Product array Puzzle](#)
[Sort array according to count of set bits](#)
[minimum no. of swaps required to sort the array](#)
[Weighted Job Scheduling in O\(n Log n\) time](#)
[Smallest number with atleastn trailing zeroes infactorial](#)
[The Painter’s Partition Problem](#)
[Binary Trees:](#)
[Inorder Traversal:](#)

[Preorder Traversal:](#)
[Post Order Traversal:](#)
[Morris Inorder Traversal](#)
[Morris Preorder Traversal:](#)
[Level Order Traversal of BT:](#)
[Height of Binary Tree:](#)
[Left View of Binary Tree](#)
[Bottom View of the Binary Tree:](#)
[Top View of the Binary Tree:](#)
[Vertical Order Traversal:](#)
[Serialize and Deserialize a Binary Tree](#)
[ZigZag Tree Traversal of a Binary Tree](#)
[Check if a tree is balanced or not](#)
[Diagonal Traversal of a Binary tree](#)
[Boundary traversal of a Binary tree](#)
[Construct Binary Tree from String with Bracket Representation](#)
[Convert Binary tree into Sum tree](#)
[Construct Tree from given Inorder and Preorder traversals](#)
[Find minimum swaps required to convert a Binary tree into BST](#)
[Check if Binary tree is Sum tree or not](#)
[Check if all leaf nodes are at same level or not](#)
[Check if a Binary Tree contains duplicate subtrees of size 2 or more \[IMP \]](#)
[Check if 2 trees are mirror or not](#)
[Sum of Nodes on the Longest path from root to leaf node](#)
[Check if given graph is tree or not. \[IMP \]](#)
[Find Largest subtree sum in a tree](#)
[Maximum Sum of nodes in Binary tree such that no two are adjacent](#)
[Print all "K" Sum paths in a Binary tree](#)
[Find LCA in a Binary tree](#)
[Find distance between 2 nodes in a Binary tree](#)
[Kth Ancestor of node in a Binary tree](#)
[Find all Duplicate subtrees in a Binary tree \[IMP \]](#)
[Tree Isomorphism Problem](#)
[Binary Search Tree](#)
[Find a value in a BST](#)
[Deletion of a node in a BST](#)
[Insertion of a node in a BST](#)
[Find inorder successor and inorder predecessor in a BST](#)
[Check if a tree is a BST or not](#)
[Populate Inorder Successor for all nodes](#)
[LCA in BST – Lowest Common Ancestor in Binary Search Tree](#)

[Construct BST from Preorder Traversal](#)
[Binary Tree to Binary Search Tree Conversion](#)
[Balance a Binary Search Tree](#)
[Merge Two Balanced Binary Search Trees](#)
[K'th Largest element in BST using constant extra space](#)
[Find Kth smallest element in a BST](#)
[Count pairs from two BSTs whose sum is equal to a given value x](#)
[Find median of BST](#)
[Count BST nodes that lie in a given range](#)
[Replace every element with the least greater element on its right](#)
[Given n appointments, find all conflicting appointments](#)
[Check if an array can be Preorder of a BST](#)
[Check whether BST contains Dead End or not](#)
[Largest BST Subtree – Simple Implementation](#)
[Flatten BST to sorted list | Increasing order](#)

Graphs:

[Clone the Graph:](#)
[DFS:](#)
[BFS:](#)
[BFS Detect Cycle in Undirected Graph:](#)
[DFS Detect a Cycle in Undirected Graph:](#)
[DFS Find Cycle Directed Graph:](#)
[BFS Directed Detect Cycle:](#)
[Kruskal MST:](#)
[Prim's MST Minimum Spanning Tree:](#)
[Single Source Dijkstra's Algorithm:](#)
[Topological Sort:](#)
[Rat in a MAZE:](#)
[Steps by Knight:](#)
[Flood Fill Algo:](#)
[Word Ladder ★](#)
[Minimum time taken by each job to be completed given by a Directed Acyclic Graph](#)
[Check if it is possible to finish all task from given dependencies \(Course Schedule I\)](#)
[Number is Islands:](#)
[Given a sorted Dictionary of an Alien Language, find order of characters](#)
[Making wired Connections](#)
[Total number of Spanning Trees in a Graph](#)
[Implement Bellman Ford Algorithm](#)
[Floyd Warshall Algorithm](#)
[Travelling Salesman Problem](#)
[Graph Colouring Problem](#)

[Snake and Ladder Problem](#)

[Bridges in a graph](#) 

[Articulation Points \(or Cut Vertices\) in a Graph](#)

[Count Strongly connected Components\(Kosaraju Algo\)](#)

[Matrix](#)

[Rotate the matrix by 90 Degrees](#)

[Spiral Traversal of matrix:](#)

[Search in 2d Matrix:](#)

[Median in a rowwise sorted Matrix!](#)

[Find the row with maximum number of 1s](#)

[Print all elements in sorted order from row and column wise sorted matrix](#)

[Maximal Rectangle](#)

[Find a specific pair in matrix:](#)

[Kth smallest element in a row-wise and column-wise sorted 2D array](#)

[Common elements in all rows of a given matrix](#)

[Bit Manipulation:](#)

[Concepts:](#)

[Count Set Bits in an Integer](#)

[Bit Difference of Two Number - count the number of bits needed to be flipped to convert A to B.](#)

[Find the two non-repeating elements in an array of repeating elements/ Unique Numbers 2](#)

[Find position of set bit:](#)

[Copy set bits in a range](#)

[Divide two integers without using multiplication and division operator](#)

[Calculate square of a number without using *, / and pow\(\)](#)

[Power Set](#)

[Count total set bits in first N Natural Numbers \(all numbers from 1 to N\)](#)

[Trie:](#)

[Implement a Trie From Scratch:](#)

[Find shortest unique prefix for every word in a given list | Set 1 \(Using Trie\)](#)

[Word Break Problem | Trie Solution](#)

[Implement a Phone Directory](#)

[Unique rows in boolean matrix](#)

[General Questions for GS:](#)

[Find the total lattice points on the circumference of a circle. Ref:](#)

[First non-repeating character of given string](#)

[Arrange given numbers to form the biggest number](#)

[Min flip to make binary string alternate](#)

[Maximum possible stolen value from houses.](#)

[Water Jug Puzzle](#)

[Round table coin game](#)

[First Missing Positive](#)
[Longest Substring Without Repeating Characters](#)
[Coin Change: Min coins to form a sum](#)
[Find Median from Running Data Stream](#)
[Kth Largest Element in an Array](#)
[Minimum Window Substring](#)
[Regular Expression Matching](#)
[Burst Balloons | Partition DP | DP 51](#)
[Trapping Rain Water](#)
[Spiral Matrix](#)
[Fraction to Recurring Decimal'](#)
[High Five](#)
[IP Frequency:](#)
[Minimum Path Sum:](#)
[Number of Sub-arrays of Size K and Average Greater than or Equal to Threshold](#)
[Car Pooling](#)
[Median of two sorted arrays](#)
[String Compression](#)
[Robot Bounded In Circle](#)
[Climbing Stairs](#)
[Coin Change](#)
[Coin Change II - Total number of solutions:](#)
[Sliding Window Maximum](#)
[Swim in Rising Water](#)
[Min Cost to Connect Points](#)
[Network Delay Time](#)
[Time Based Key-Value Store](#)
[Task Scheduler](#)
[Rotting Fruit](#)
[Islands and Treasure](#)
[Pacific Atlantic Water Flow](#)
[Counting Bits](#)
[Reverse Bits](#)
[Min Cost Climbing Stairs](#)
[House Robber:](#)
[House Robber II](#)
[Meeting Rooms II](#)
[Decode Ways](#)
[Maximum Product Subarray:](#)
[Unique Paths:](#)
[Best Time to Buy and Sell Stock with Cooldown](#)

[Target Sum](#)

[Interleaving String](#)

[Longest Increasing Path in Matrix](#)

[Distinct Subsequences](#)

[Burst Balloons](#)

[Jump Game](#)

[Gas Station](#)

[Hand of Straights](#)

Math Logics:

Check if dddd....n! Times (given n and d) is divisible by 7

if n>5 or d==9 or (n>2 and d%3==0):

ans.append(9)

Input : 8955795758

Output : Divisible by 7

Explanation:

We express the number in terms of triplets of digits as follows.

(008)(955)(795)(758)

Now, $758 - 795 + 955 - 8 = 910$, which is divisible by 7

Input : 100000000000

Output : Not Divisible by 7

Explanation:

We express the number in terms of triplets of digits as follows.

(100)(000)(000)(000)

Now, $000 - 000 + 000 - 100 = -100$, which is not divisible by 7

Lagrange's Four Square Theorem

Lagrange's Four Square Theorem states that **every natural number can be expressed as the sum of four integer squares**. That is, for any natural number n , there exist integers a, b, c, d such that:

$$n = a^2 + b^2 + c^2 + d^2$$

```
int minSquares(int n) {

    // Case 1: ans = 1 if the number itself is a
    // perfect square
    if (isSquare(n)) {
        return 1;
    }

    // Case 2: ans = 2 if the number is a sum of
    // two perfect square
    // we check for each i if  $n - (i * i)$  is a perfect
    // square
    for (int i = 1; i * i <= n; i++) {
        if (isSquare(n - (i * i))) {
            return 2;
        }
    }

    // Case 4: ans = 4 if the number is a sum of
    // four perfect square
    // possible if the number is representable in the form
    //  $4^a (8^b + 7)$ .
    while (n > 0 && n % 4 == 0) {
        n /= 4;
    }
    if (n % 8 == 7) {
        return 4;
    }

    // since all the other cases have been evaluated,
    // the answer can only then be 3 if the program
    // reaches here
    return 3;
}
```

Legendre proved that a number n can be written as a sum of three squares if and only if $n \neq 4^a(8m + 7)$ for some integers a, m . That is, numbers of the form $4^a(8m + 7)$ cannot be written as the sum of three squares.

For example:

- $7 = 4^0(8 \times 0 + 7)$ cannot be written as a sum of three squares.
- $28 = 4^1(8 \times 0 + 7)$ also cannot be written as a sum of three squares.

However, Lagrange's Four Square Theorem asserts that **every** number can be written as a sum of four squares, even those that fail Legendre's condition.

Sorting Algorithms:

Comparison Based : [Selection Sort](#), [Bubble Sort](#), [Insertion Sort](#), [Merge Sort](#), [Quick Sort](#), [Heap Sort](#), [Cycle Sort](#), [3-way Merge Sort](#)

Non Comparison Based : [Counting Sort](#), [Radix Sort](#), [Bucket Sort](#), [TimSort](#), [Comb Sort](#), [Pigeonhole Sort](#)

Hybrid Sorting Algorithms : [IntroSort](#), [Tim Sort](#)

Algorithm	Best Time Complexity	Average Time Complexity	Worst Time Complexity	Worst Space Complexity
Linear Search	$O(1)$	$O(n)$	$O(n)$	$O(1)$
Binary Search	$O(1)$	$O(\log n)$	$O(\log n)$	$O(1)$
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Bucket Sort	$O(n+k)$	$O(n+k)$	$O(n^2)$	$O(n)$
Radix Sort	$O(nk)$	$O(nk)$	$O(nk)$	$O(n+k)$
Tim Sort	$O(n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Shell Sort	$O(n)$	$O((n \log(n))^2)$	$O((n \log(n))^2)$	$O(1)$

◆ Stable Sorting Algorithms

Stable sorting algorithms preserve the relative order of records with equal keys.

Algorithm	Time Complexity (Best / Avg / Worst)	Space Complexity	Stable	Notes
Bubble Sort	$O(n) / O(n^2) / O(n^2)$	$O(1)$	<input checked="" type="checkbox"/>	Simple, educational
Insertion Sort	$O(n) / O(n^2) / O(n^2)$	$O(1)$	<input checked="" type="checkbox"/>	Efficient for small or nearly sorted data
Merge Sort	$O(n \log n) / O(n \log n) / O(n \log n)$	$O(n)$	<input checked="" type="checkbox"/>	Recursive, divide-and-conquer
Tim Sort	$O(n) / O(n \log n) / O(n \log n)$	$O(n)$	<input checked="" type="checkbox"/>	Used in Python's and Java's standard libraries
Counting Sort	$O(n + k) / O(n + k) / O(n + k)$	$O(k)$	<input checked="" type="checkbox"/>	Non-comparison-based; good for small range of integers
Radix Sort	$O(nk) / O(nk) / O(nk)$	$O(n + k)$	<input checked="" type="checkbox"/>	Works on integers/strings; uses counting sort internally
Bucket Sort	$O(n + k) / O(n + k) / O(n^2)$	$O(n + k)$	<input checked="" type="checkbox"/>	Stable if underlying sort is stable

◆ Unstable Sorting Algorithms

Unstable algorithms **may** reorder equal elements.

Algorithm	Time Complexity (Best / Avg / Worst)	Space Complexity	Stable	Notes
Selection Sort	$O(n^2) / O(n^2) / O(n^2)$	$O(1)$	✗	Simple but inefficient
Heap Sort	$O(n \log n) / O(n \log n) / O(n \log n)$	$O(1)$	✗	Uses binary heap
Quick Sort	$O(n \log n) / O(n \log n) / O(n^2)$	$O(\log n)$	✗	Very efficient; divide-and-conquer
Shell Sort	$O(n \log n) / \text{depends} / O(n^2)$	$O(1)$	✗	Gap-based variation of insertion sort
Tree Sort	$O(n \log n) / O(n \log n) / O(n^2)$	$O(n)$	✗	Uses BST; stability depends on tree implementation

Graph Algorithms' Time Complexity

Algorithm	Time Complexity
Depth-First Search (DFS)	$O(V + E)$
Breadth-First Search (BFS)	$O(V + E)$
Dijkstra's Algorithm	$O(V^2) \text{ or } O((V + E) \log V)$
Bellman-Ford Algorithm	$O(V * E)$
Floyd-Warshall Algorithm	$O(V^3)$
Prim's Algorithm	$O(V^2) \text{ or } O((V + E) \log V)$
Kruskal's Algorithm	$O(E \log V)$
Topological Sorting	$O(V + E)$

Kosaraju's Algorithm	$O(V + E)$
Tarjan's Algorithm	$O(V + E)$
Ford-Fulkerson Algorithm	$O(\text{max_flow} * E)$
Edmonds-Karp Algorithm	$O(V * E^2)$
Dinic's Algorithm	$O(V^2 * E)$
Union-Find Algorithm	$O(E \log V)$
Hierholzer's Algorithm	$O(V + E)$
A Algorithm*	$O((V + E) \log V)$

Backtracking:

Printing all solutions in N-Queen Problem

[Expected Approach] – Using Backtracking with Pruning – $O(n!)$ Time and $O(n)$ Space

```

// Utility function for solving the N-Queens
// problem using backtracking.
void nQueenUtil(int j, int n, vector<int> &board, vector<bool> &rows,
    vector<bool> &diag1, vector<bool> &diag2, vector<vector<int>> &res) {

    if (j > n) {
        // A solution is found
        res.push_back(board);
        return;
    }
    for (int i = 1; i <= n; ++i) {
        if (!rows[i] && !diag1[i + j] && !diag2[i - j + n]) {

            // Place queen
            rows[i] = diag1[i + j] = diag2[i - j + n] = true;
            board.push_back(i);

            // Recurse to the next column
            nQueenUtil(j + 1, n, board, rows, diag1, diag2, res);

            // Remove queen (backtrack)
            board.pop_back();
            rows[i] = diag1[i + j] = diag2[i - j + n] = false;
        }
    }
}

// Solves the N-Queens problem and returns
// all valid configurations.
vector<vector<int>> nQueen(int n) {
    vector<vector<int>> res;
    vector<int> board;

    // Rows occupied
    vector<bool> rows(n + 1, false);

    // Major diagonals (row + j) and Minor diagonals (row - col + n)
    vector<bool> diag1(2 * n + 1, false);
    vector<bool> diag2(2 * n + 1, false);

    // Start solving from the first column
    nQueenUtil(1, n, board, rows, diag1, diag2, res);
}

```

```

        return res;
    }

int main() {
    int n = 4;
    vector<vector<int>> res = nQueen(n);

    for (int i = 0; i < res.size(); i++) {
        cout << "[";
        for (int j = 0; j < n; ++j) {
            cout << res[i][j];
            if (j != n - 1)
                cout << " ";
        }
        cout << "]\n";
    }
    return 0;
}

```

Word Break Problem using Backtracking

Given a non-empty sequence s and a dictionary dict[] containing a list of non-empty words, the task is to return all possible ways to break the sentence in individual dictionary words.

Note: The same word in the dictionary may be reused multiple times while breaking.

Examples:

Input: s = “catsanddog” , dict = [“cat”, “cats”, “and”, “sand”, “dog”]

Output:

“cats and dog”

“cat sand dog”

Explanation: The string is split into above 2 ways, in each way all are valid dictionary words.

Input: s = “pineapplepenapple” , dict = [“apple”, “pen”, “applepen”, “pine”, “pineapple”]

Output:

“pine apple pen apple”

“pineapple pen apple”

“pine applepen apple”

Explanation: The string is split into above 3 ways, in each way all are valid dictionary words.

```

// Helper function to perform backtracking
void wordBreakHelper(string& s, unordered_set<string>& dictSet,
                     string& curr, vector<string>& res,
                     int start) {

    // If start reaches the end of the string,
    // save the result
    if (start == s.length()) {
        res.push_back(curr);
        return;
    }

    // Try every possible substring from the current index
    for (int end = start + 1; end <= s.length(); ++end) {
        string word = s.substr(start, end - start);

        // Check if the word exists in the dictionary
        if (dictSet.count(word)) {
            string prev = curr;

            // Append the word to the current sentence
            if (!curr.empty()) {
                curr += " ";
            }
            curr += word;

            // Recurse for the remaining string
            wordBreakHelper(s, dictSet, curr, res, end);

            // Backtrack to restore the current sentence
            curr = prev;
        }
    }
}

// Main function to generate all possible sentence breaks
vector<string> wordBreak(string s, vector<string>& dict) {

    // Convert dictionary vector
    // to an unordered set
    unordered_set<string>
        dictSet(dict.begin(), dict.end());
}

```

```

vector<string> res;
string curr;

wordBreakHelper(s, dictSet, curr, res, 0);

return res;
}

int main() {

string s = "ilikesamsungmobile";
vector<string> dict = {"i", "like", "sam", "sung",
                      "samsung", "mobile", "ice",
                      "and", "cream", "icecream",
                      "man", "go", "mango"};

vector<string> result = wordBreak(s, dict);

for (string sentence : result) {
    cout << sentence << endl;
}

return 0;
}

```

Remove Invalid Parentheses

Given a string s that contains parentheses and letters, remove the minimum number of invalid parentheses to make the input string valid.

Return a list of unique strings that are valid with the minimum number of removals. You may return the answer in any order.

Example 1:

Input: $s = "())()$ "

Output: $["(())()", "()()()"]$

Example 2:

Input: s = "(a)())()"
Output: ["(a())()", "(a)()"]
Example 3:

Input: s = ")(")
Output: [""]

```
class Solution {
public:
    vector<string> removeInvalidParentheses(string s) {
        unordered_set<string> result;
        int left_removed = 0;
        int right_removed = 0;
        for(auto c : s) {
            if(c == '(') {
                ++left_removed;
            }
            if(c == ')') {
                if(left_removed != 0) {
                    --left_removed;
                }
                else {
                    ++right_removed;
                }
            }
        }
        helper(s, 0, left_removed, right_removed, 0, "", result);
        return vector<string>(result.begin(), result.end());
    }
private:
    void helper(string s, int index, int left_removed, int right_removed,
    int pair, string path, unordered_set<string>& result) {
        if(index == s.size()) {
            if(left_removed == 0 && right_removed == 0 && pair == 0) {
                result.insert(path);
            }
            return;
        }
        if(s[index] != '(' && s[index] != ')') {
            helper(s, index + 1, left_removed, right_removed, pair, path +
            s[index], result);
        }
        else {
```

```

        if(s[index] == '(') {
            if(left_removed > 0) {
                helper(s, index + 1, left_removed - 1, right_removed,
pair, path, result);
            }
            helper(s, index + 1, left_removed, right_removed, pair + 1,
path + s[index], result);
        }
        if(s[index] == ')') {
            if(right_removed > 0) {
                helper(s, index + 1, left_removed, right_removed - 1,
pair, path, result);
            }
            if(pair > 0) {
                helper(s, index + 1, left_removed, right_removed, pair
- 1, path + s[index], result);
            }
        }
    }
};


```

BFS:

```

// method checks if character is parenthesis(open
// or closed)
bool isParenthesis(char c)
{
    return ((c == '(') || (c == ')'));
}

// method returns true if string contains valid
// parenthesis
bool isValidString(string str)
{
    int cnt = 0;
    for (int i = 0; i < str.length(); i++)
    {
        if (str[i] == '(')
            cnt++;
        else if (str[i] == ')')
            cnt--;
    }
}

```

```

        if (cnt < 0)
            return false;
    }
    return (cnt == 0);
}

// method to remove invalid parenthesis
void removeInvalidParenthesis(string str)
{
    if (str.empty())
        return ;

    // visit set to ignore already visited string
    unordered_set<string> visit;

    // queue to maintain BFS
    queue<string> q;
    string temp;
    bool level;

    // pushing given string as starting node into queue
    q.push(str);
    visit.insert(str);
    while (!q.empty())
    {
        str = q.front();  q.pop();
        if (isValidString(str))
        {
            cout << str << endl;

            // If answer is found, make level true
            // so that valid string of only that level
            // are processed.
            level = true;
        }
        if (level)
            continue;
        for (int i = 0; i < str.length(); i++)
        {
            if (!isParenthesis(str[i]))
                continue;

```

```

        // Removing parenthesis from str and
        // pushing into queue,if not visited already
        temp = str.substr(0, i) + str.substr(i + 1);
        if (visit.find(temp) == visit.end())
        {
            q.push(temp);
            visit.insert(temp);
        }
    }
}

```

Sudoku Solver

```

class Solution {
public:
    bool isValid(vector<vector<char>>& board, int row, int col, char c){
        for(int i = 0; i < 9; i++) {
            if(board[i][col] == c)
                return false;

            if(board[row][i] == c)
                return false;

            if(board[3 * (row / 3) + i / 3][3 * (col / 3) + i % 3] == c)
                return false;
        }
        return true;
    }
    bool solve(vector<vector<char>> &b){
        for(int i=0;i<b.size();i++){
            for(int j=0;j<b.size();j++){
                if(b[i][j]=='.'){
                    for(char c='1';c<='9';c++){
                        if(isValid(b,i,j,c)){
                            b[i][j]=c;

```

```

        if(solve(b))
            return true;
        else
            b[i][j]='.';
        }
    }
    return false;
}

}
return true;
}
void solveSudoku(vector<vector<char>>& board) {
    solve(board);
}

};

```

M-Coloring Problem

Given an undirected graph and a number m, the task is to color the given graph with at most m colors such that no two adjacent vertices of the graph are colored with the same color

Note: Here coloring of a graph means the assignment of colors to all vertices

Below is an example of a graph that can be colored with 3 different colors:

Input: graph = {0, 1, 1, 1},
 {1, 0, 1, 0},
 {1, 1, 0, 1},
 {1, 0, 1, 0}

Output: Solution Exists: Following are the assigned colors: 1 2 3 2

Explanation: By coloring the vertices with following colors,
 adjacent vertices does not have same colors

Input: graph = {1, 1, 1, 1},
 {1, 1, 1, 1},
 {1, 1, 1, 1},
 {1, 1, 1, 1}

Output: Solution does not exist

Explanation: No solution exists

```
// C++ program for solution of M
// Coloring problem using backtracking

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if
the current color assignment
is safe for vertex v i.e. checks
whether the edge exists or not
(i.e, graph[v][i]==1). If exist
then checks whether the color to
be filled in the new vertex(c is
sent in the parameter) is already
used by its adjacent
vertices(i-->adj vertices) or
not (i.e, color[i]==c) */
bool isSafe(int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;

    return true;
}

/* A recursive utility function
to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[],
                      int v)
{
    /* base case: If all vertices are
       assigned a color then return true */
}
```

```

if (v == V)
    return true;

/* Consider this vertex v and
try different colors */
for (int c = 1; c <= m; c++) {

    /* Check if assignment of color
    c to v is fine*/
    if (isSafe(v, graph, color, c)) {
        color[v] = c;

        /* recur to assign colors to
        rest of the vertices */
        if (graphColoringUtil(graph, m, color, v + 1)
            == true)
            return true;

        /* If assigning color c doesn't
        lead to a solution then remove it */
        color[v] = 0;
    }
}

/* If no color can be assigned to
this vertex then return false */
return false;
}

/* This function solves the m Coloring
problem using Backtracking. It mainly
uses graphColoringUtil() to solve the
problem. It returns false if the m
colors cannot be assigned, otherwise
return true and prints assignments of
colors to all vertices. Please note
that there may be more than one solutions,
this function prints one of the
feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0.
}

```

```

// This initialization is needed
// correct functioning of isSafe()
int color[V];
for (int i = 0; i < V; i++)
    color[i] = 0;

// Call graphColoringUtil() for vertex 0
if (graphColoringUtil(graph, m, color, 0) == false) {
    cout << "Solution does not exist";
    return false;
}

// Print the solution
printSolution(color);
return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    cout << "Solution Exists:"
        << " Following are the assigned colors"
        << "\n";
    for (int i = 0; i < V; i++)
        cout << " " << color[i] << " ";

    cout << "\n";
}

// Driver code
int main()
{
    /* Create following graph and test
    whether it is 3 colorable
    (3)---(2)
    | / |
    | / |
    | / |
    (0)---(1)
    */
    bool graph[V][V] = {
        { 0, 1, 1, 1 },

```

```

        { 1, 0, 1, 0 },
        { 1, 1, 0, 1 },
        { 1, 0, 1, 0 },
    };

    // Number of colors
    int m = 3;

    // Function call
    graphColoring(graph, m);
    return 0;
}

// This code is contributed by Shivani

```

Print all Palindromic Partitions of a String using Backtracking

Input: nitin
Output: n i t i n
n i t i n
nitin

Input: geeks
Output: g e e k s
g ee k s

```

bool checkPalindrome(string& s)
{
    int n = s.size();
    int i = 0, j = n - 1;
    while (i < j) {
        if (s[i] != s[j])
            return false;
        i++;
        j--;
    }
    return true;
}

```

```

}

// Recursive function which takes starting index idx
// and generates all substrings starting at idx.
// If substring generated is palindrome it adds to
// current list and makes a recursive call for
// remaining string.
void Partition(vector<vector<string> >& res, string& s,
               int idx, vector<string>& curr)
{
    // If we reach the end of string at the current list
    // to the result.
    if (idx == s.size()) {
        res.push_back(curr);
        return;
    }
    // Stores the current substring.
    string t;
    for (int i = idx; i < s.size(); i++) {
        t.push_back(s[i]);

        // Check whether the string is palindrome is
        // not.
        if (checkPalindrome(t)) {

            // Adds the string to current list
            curr.push_back(t);

            // Recursive call for the remaining string
            Partition(res, s, i + 1, curr);

            // Remove the string from the current
            // string.
            curr.pop_back();
        }
    }
}

```

Subset Sum Problem

Given an array arr[] of non-negative integers and a value sum, the task is to check if there is a subset of the given array whose sum is equal to the given sum.

Examples:

Input: arr[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True

Explanation: There is a subset (4, 5) with sum 9.

Using Top-Down DP (Memoization) – O(sum*n) Time and O(sum*n) Space

```
// Recursive function to check if a subset
// with the given sum exists
bool isSubsetSumRec(vector<int>& arr, int n, int sum,
                     vector<vector<int>> &memo) {

    // If the sum is zero, we found a subset
    if (sum == 0)
        return 1;

    // If no elements are left
    if (n <= 0)
        return 0;

    // If the value is already
    // computed, return it
    if (memo[n][sum] != -1)
        return memo[n][sum];

    // If the last element is greater than
    // the sum, ignore it
    if (arr[n - 1] > sum)
        return memo[n][sum] = isSubsetSumRec(arr, n - 1, sum, memo);
    else {

        // Include or exclude the last element
        return memo[n][sum] = isSubsetSumRec(arr, n - 1, sum, memo) ||
                           isSubsetSumRec(arr, n - 1, sum - arr[n - 1],
                                         memo);
    }
}

// Function to initiate the subset sum check
bool isSubsetSum(vector<int>&arr, int sum) {
```

```

int n = arr.size();

vector<vector<int>> memo(n + 1, vector<int>(sum + 1, -1));
return isSubsetSumRec(arr, n, sum, memo);
}

```

Using Bottom-Up DP (Tabulation) – O(sum*n) Time and O(sum*n) Space

```

// Function to check if there is a subset of arr[]
// with sum equal to the given sum using tabulation with vectors
bool isSubsetSum(vector<int> &arr, int sum) {
    int n = arr.size();

    // Create a 2D vector for storing results
    // of subproblems
    vector<vector<bool>> dp(n + 1, vector<bool>(sum + 1, false));

    // If sum is 0, then answer is true (empty subset)
    for (int i = 0; i <= n; i++)
        dp[i][0] = true;

    // Fill the dp table in bottom-up manner
    for (int i = 1; i <= n; i++) {

        for (int j = 1; j <= sum; j++) {
            if (j < arr[i - 1]) {

                // Exclude the current element
                dp[i][j] = dp[i - 1][j];
            }
            else {

                // Include or exclude
                dp[i][j] = dp[i - 1][j]
                || dp[i - 1][j - arr[i - 1]];
            }
        }
    }

    return dp[n][sum];
}

```

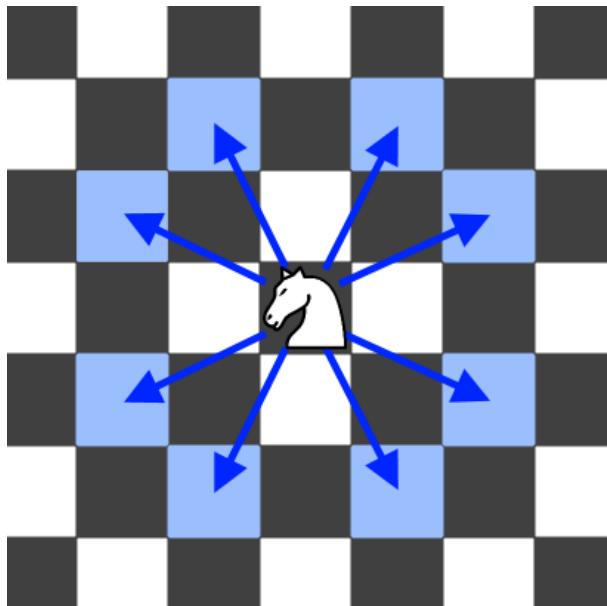
The Knight's tour problem

There is a knight on an $n \times n$ chessboard. In a valid configuration, the knight starts **at the top-left cell** of the board and visits every cell on the board **exactly once**.

You are given an $n \times n$ integer matrix grid consisting of distinct integers from the range $[0, n * n - 1]$ where $\text{grid}[\text{row}][\text{col}]$ indicates that the cell (row, col) is the $\text{grid}[\text{row}][\text{col}]^{\text{th}}$ cell that the knight visited. The moves are **0-indexed**.

Return true if grid represents a valid configuration of the knight's movements or false otherwise.

Note that a valid knight move consists of moving two squares vertically and one square horizontally, or two squares horizontally and one square vertically. The figure below illustrates all the possible eight moves of a knight from some cell.



Example 1:

0	11	16	5	20
17	4	19	10	15
12	1	8	21	6
3	18	23	14	9
24	13	2	7	22

Input: grid = [[0,11,16,5,20],[17,4,19,10,15],[12,1,8,21,6],[3,18,23,14,9],[24,13,2,7,22]]

Output: true

Explanation: The above diagram represents the grid. It can be shown that it is a valid configuration.

```
class Solution {
public:
    bool checkValidGrid(vector<vector<int>>& grid) {
        int n = grid.size();
        int r = -1, c = -1;
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (grid[i][j] == 0) {
                    r = i;
                    c = j;
                    goto Rahul;
                }
            }
        }
    }
Rahul:
    vector<pair<int, int>> dir = {{-2, 1}, {-1, 2}, {1, 2}, {2, 1}, {2, -1}, {1, -2},
{-1, -2}, {-2, -1}};
    int cn = 0;
    auto dfs = [&](this auto self, int i, int j)->void {
        for (auto [dx, dy]: dir) {
            int x = i + dx;
            int y = j + dy;
            if (x >= 0 && x < n && y >= 0 && y < n && grid[x][y] == cn+1) {
                cn++;
                self(x, y);
                return;
            }
        }
    };
};
```

```

dfs(0,0);
if (grid[0][0] != 0) {
    return false;
}
if (cn == n * n-1) {
    return true;
}
return false;
};


```

Tug of War

Given a set of n integers, divide the set in two subsets of $n/2$ sizes each such that the absolute difference of the sum of two subsets is as minimum as possible. If n is even, then sizes of two subsets must be strictly $n/2$ and if n is odd, then size of one subset must be $(n-1)/2$ and size of other subset must be $(n+1)/2$.

```

// function that tries every possible solution by calling itself
recursively
void TOWUtil(int* arr, int n, bool* curr_elements, int
no_of_selected_elements,
            bool* soln, int* min_diff, int sum, int curr_sum, int
curr_position)
{
    // checks whether the it is going out of bound
    if (curr_position == n)
        return;

    // checks that the numbers of elements left are not less than the
    // number of elements required to form the solution
    if ((n/2 - no_of_selected_elements) > (n - curr_position))
        return;

    // consider the cases when current element is not included in the
    // solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements,
            soln, min_diff, sum, curr_sum, curr_position+1);

    // add the current element to the solution
    curr_elements[curr_position] = true;
    curr_sum += arr[curr_position];
    if (curr_sum > sum - curr_sum)
        min_diff[0] = abs(sum - curr_sum);
    else
        min_diff[0] = min(min_diff[0], abs(sum - curr_sum));
}


```

```

no_of_selected_elements++;
curr_sum = curr_sum + arr[curr_position];
curr_elements[curr_position] = true;

// checks if a solution is formed
if (no_of_selected_elements == n/2)
{
    // checks if the solution formed is better than the best solution
so far
    if (abs(sum/2 - curr_sum) < *min_diff)
    {
        *min_diff = abs(sum/2 - curr_sum);
        for (int i = 0; i<n; i++)
            soln[i] = curr_elements[i];
    }
}
else
{
    // consider the cases where current element is included in the
solution
    TOWUtil(arr, n, curr_elements, no_of_selected_elements, soln,
            min_diff, sum, curr_sum, curr_position+1);
}

// removes current element before returning to the caller of this
function
curr_elements[curr_position] = false;
}

// main function that generate an arr
void tugOfWar(int *arr, int n)
{
    // the boolean array that contains the inclusion and exclusion of an
element
    // in current set. The number excluded automatically form the other set
bool* curr_elements = new bool[n];

    // The inclusion/exclusion array for final solution
bool* soln = new bool[n];

    int min_diff = INT_MAX;

    int sum = 0;
}

```

```

for (int i=0; i<n; i++)
{
    sum += arr[i];
    curr_elements[i] = soln[i] = false;
}

// Find the solution using recursive function TOWUtil()
TOWUtil(arr, n, curr_elements, 0, soln, &min_diff, sum, 0, 0);

// Print the solution
cout << "The first subset is: ";
for (int i=0; i<n; i++)
{
    if (soln[i] == true)
        cout << arr[i] << " ";
}
cout << "\nThe second subset is: ";
for (int i=0; i<n; i++)
{
    if (soln[i] == false)
        cout << arr[i] << " ";
}
}

```

Find shortest safe route in a path with landmines

Given a path in the form of a rectangular matrix having few landmines arbitrarily placed (marked as 0), calculate length of the shortest safe route possible from any cell in the first column to any cell in the last column of the matrix. We have to avoid landmines and their four adjacent cells (left, right, above and below) as they are also unsafe. We are allowed to move to only adjacent cells which are not landmines. i.e. the route cannot contains any diagonal moves.

```

// C++ program to find shortest safe Route in
// the matrix with landmines
#include <bits/stdc++.h>
using namespace std;

#define R 12

```

```

#define C 10

struct Key{
    int x,y;
    Key(int i,int j){ x=i;y=j;};
};

// These arrays are used to get row and column
// numbers of 4 neighbours of a given cell
int rowNum[] = { -1, 0, 0, 1 };
int colNum[] = { 0, -1, 1, 0 };

// A function to check if a given cell (x, y) is
// a valid cell or not
bool isValid(int x, int y)
{
    if (x < R && y < C && x >= 0 && y >= 0)
        return true;

    return false;
}

// A function to mark all adjacent cells of
// landmines as unsafe. Landmines are shown with
// number 0
void findShortestPath(int mat[R][C]){
    int i,j;

    for (i = 0; i < R; i++)
    {
        for (j = 0; j < C; j++)
        {
            // if a landmine is found
            if (mat[i][j] == 0)
            {
                // mark all adjacent cells
                for (int k = 0; k < 4; k++)
                    if (isValid(i + rowNum[k], j + colNum[k]))
                        mat[i + rowNum[k]][j + colNum[k]] = -1;
            }
        }
    }
    // mark all found adjacent cells as unsafe
}

```

```

for (i = 0; i < R; i++)
{
    for (j = 0; j < C; j++)
    {
        if (mat[i][j] == -1)
            mat[i][j] = 0;
    }
}

int dist[R][C];

for(i=0;i<R;i++){
    for(j=0;j<C;j++)
        dist[i][j] = -1;
}
queue<Key> q;

for(i=0;i<R;i++){
    if(mat[i][0] == 1){
        q.push(Key(i,0));
        dist[i][0] = 0;
    }
}

while(!q.empty()){
    Key k = q.front();
    q.pop();

    int d = dist[k.x][k.y];

    int x = k.x;
    int y = k.y;

    for (int k = 0; k < 4; k++) {
        int xp = x + rowNum[k];
        int yp = y + colNum[k];
        if(isValid(xp,yp) && dist[xp][yp] == -1 && mat[xp][yp] ==
1){
            dist[xp][yp] = d+1;
            q.push(Key(xp,yp));
        }
    }
}

```

```

// stores minimum cost of shortest path so far
int ans = INT_MAX;
// start from first column and take minimum
for(i=0;i<R;i++){
    if(mat[i][C-1] == 1 && dist[i][C-1] != -1){
        ans = min(ans,dist[i][C-1]);
    }
}

// if destination can be reached
if(ans == INT_MAX)
    cout << "NOT POSSIBLE\n";

else// if the destination is not reachable
    cout << "Length of shortest safe route is " << ans << endl;
}

// Driver code
int main(){

    // input matrix with landmines shown with number 0
    int mat[R][C] =
    {
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 0, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 0, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 0, 1, 1, 1, 1 },
        { 1, 0, 1, 1, 1, 1, 1, 1, 1, 0, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1 },
        { 0, 1, 1, 1, 1, 0, 1, 1, 1, 1, 1 }
    };
    // find shortest path
    findShortestPath(mat);
}

```

Combinational Sum

Given an array of distinct integers arr[] and an integer target, the task is to find a list of all unique combinations of array where the sum of chosen element is equal to target.

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

```
class Solution {  
public:  
    vector<vector<int>> ans;  
    void solve(vector<int> &c, int t, vector<int> &curr, int idx){  
        if(t<0) return;  
        if(t==0){  
            ans.push_back(curr);return;  
        }  
        for(int i=idx;i<c.size();i++){  
            curr.push_back(c[i]);  
            solve(c, t-c[i], curr, i);  
            curr.pop_back();  
        }  
    }  
    vector<vector<int>> combinationSum(vector<int>& c, int t) {  
        vector<int> curr;  
        solve(c, t, curr, 0);  
        return ans;  
    }  
};
```

Find Maximum number possible by doing at-most K swaps

Given a string s and an integer k, the task is to find the maximum possible number by performing swap operations on the digits of s at most k times.

Examples:

Input: s = “7599”, k = 2

Output: 9975

Explanation: Swap 9 with 5 so number becomes 7995, Swap 9 with 7 so number becomes 9975

Swap the Max Digit – O((n^2)*k) Time and O(k) Space

The idea is to recursively iterate through the string, finding the maximum digit for the current position, and swapping it with a larger digit later in the string. The algorithm uses a backtracking technique, meaning after each swap, it recurses to explore further swaps and then undoes the swap (backtracks). This process continues until no swaps are left or the maximum number is achieved.

```
// Function to keep the maximum result
void match(string &curr, string &result) {

    // If current number is larger, update result
    if (curr > result) {
        result = curr;
    }
}

// Function to set highest possible digits at given index
void setDigit(string &s, int index, string &res, int k) {

    // Base case: If no swaps left or index reaches
    // the last character, update result
    if (k == 0 || index == s.size() - 1) {
        match(s, res);
        return;
    }

    int maxDigit = 0;

    // Finding maximum digit for placing at given index
    for (int i = index; i < s.size(); i++) {
        maxDigit = max(maxDigit, s[i] - '0');
    }

    // If the digit at current index is already max
    if (s[index] - '0' == maxDigit) {
        setDigit(s, index + 1, res, k);
        return;
    }

    // Try swapping with the maximum digit found
    for (int i = index + 1; i < s.size(); i++) {
```

```

// If max digit is found at current position
if (s[i] - '0' == maxDigit) {

    // Swap to get the max digit at the required index
    swap(s[index], s[i]);

    // Call the recursive function to set the next digit
    setDigit(s, index + 1, res, k - 1);

    // Backtrack: swap the digits back
    swap(s[index], s[i]);
}
}

// Function to find the largest number after k swaps
string findMaximumNum(string s, int k) {
    string res = s;
    setDigit(s, 0, res, k);

    // Returning the result
    return res;
}

```

Print all permutations of a string

```

// Recursive function to generate
// all permutations of string s
void recurPermute(int index, string &s,
                  vector<string> &ans) {

    // Base Case
    if (index == s.size()) {
        ans.push_back(s);
        return;
    }

    // Swap the current index with all
    // possible indices and recur

```

```

        for (int i = index; i < s.size(); i++) {
            swap(s[index], s[i]);
            recurPermute(index + 1, s, ans);
            swap(s[index], s[i]);
        }
    }

// Function to find all unique permutations
vector<string> findPermutation(string &s) {

    // Stores the final answer
    vector<string> ans;

    recurPermute(0, s, ans);

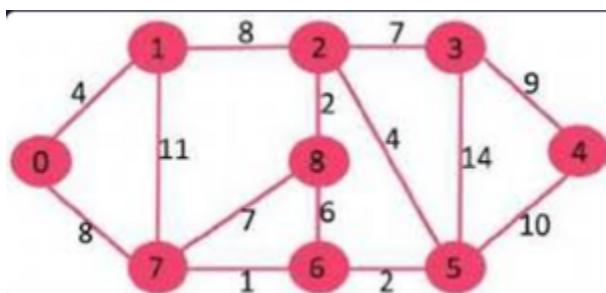
    // sort the resultant vector
    sort(ans.begin(), ans.end());

    return ans;
}

```

Find if there is a path of more than k length from a source

Given a graph, a source vertex in the graph and a number k, find if there is a simple path (without any cycle) starting from given source and ending at any other vertex such that the distance from source to that vertex is atleast 'k' length.



Input : Source s = 0, k = 58

Output : True

```
There exists a simple path 0 -> 7 -> 1  
-> 2 -> 8 -> 6 -> 5 -> 3 -> 4  
Which has a total distance of 60 km which  
is more than 58.
```

```
Input : Source s = 0, k = 62  
Output : False
```

```
In the above graph, the longest simple  
path has distance 61 (0 -> 7 -> 1-> 2  
-> 3 -> 4 -> 5-> 6 -> 8, so output  
should be false for any input greater  
than 61.
```

Time Complexity: The time complexity of this algorithm is $O((V-1)!)$ where V is the number of vertices.

Auxiliary Space: $O(V)$

```
// Program to find if there is a simple path with  
// weight more than k  
#include<bits/stdc++.h>  
using namespace std;  
  
// iPair ==> Integer Pair  
typedef pair<int, int> iPair;  
  
// This class represents a dipathted graph using  
// adjacency list representation  
class Graph  
{  
    int V;      // No. of vertices  
  
    // In a weighted graph, we need to store vertex  
    // and weight pair for every edge  
    list< pair<int, int> > *adj;  
    bool pathMoreThanKUtil(int src, int k, vector<bool> &path);  
  
public:  
    Graph(int V); // Constructor  
  
    // function to add an edge to graph  
    void addEdge(int u, int v, int w);
```

```

    bool pathMoreThanK(int src, int k);
};

// Returns true if graph has path more than k length
bool Graph::pathMoreThanK(int src, int k)
{
    // Create a path array with nothing included
    // in path
    vector<bool> path(V, false);

    // Add source vertex to path
    path[src] = 1;

    return pathMoreThanKUtil(src, k, path);
}

// Prints shortest paths from src to all other vertices
bool Graph::pathMoreThanKUtil(int src, int k, vector<bool> &path)
{
    // If k is 0 or negative, return true;
    if (k <= 0)
        return true;

    // Get all adjacent vertices of source vertex src and
    // recursively explore all paths from src.
    list<iPair>::iterator i;
    for (i = adj[src].begin(); i != adj[src].end(); ++i)
    {
        // Get adjacent vertex and weight of edge
        int v = (*i).first;
        int w = (*i).second;

        // If vertex v is already there in path, then
        // there is a cycle (we ignore this edge)
        if (path[v] == true)
            continue;

        // If weight of is more than k, return true
        if (w >= k)
            return true;

        // Else add this vertex to path
        path[v] = true;
    }
}

```

```

        // If this adjacent can provide a path longer
        // than k, return true.
        if (pathMoreThanKUtil(v, k-w, path))
            return true;

        // Backtrack
        path[v] = false;
    }

    // If no adjacent could produce longer path, return
    // false
    return false;
}

// Allocates memory for adjacency list
Graph::Graph(int V)
{
    this->V = V;
    adj = new list<iPair> [V];
}

// Utility function to an edge (u, v) of weight w
void Graph::addEdge(int u, int v, int w)
{
    adj[u].push_back(make_pair(v, w));
    adj[v].push_back(make_pair(u, w));
}

// Driver program to test methods of graph class
int main()
{
    // create the graph given in above figure
    int V = 9;
    Graph g(V);

    // making above shown graph
    g.addEdge(0, 1, 4);
    g.addEdge(0, 7, 8);
    g.addEdge(1, 2, 8);
    g.addEdge(1, 7, 11);
    g.addEdge(2, 3, 7);
    g.addEdge(2, 8, 2);
}

```

```

g.addEdge(2, 5, 4);
g.addEdge(3, 4, 9);
g.addEdge(3, 5, 14);
g.addEdge(4, 5, 10);
g.addEdge(5, 6, 2);
g.addEdge(6, 7, 1);
g.addEdge(6, 8, 6);
g.addEdge(7, 8, 7);

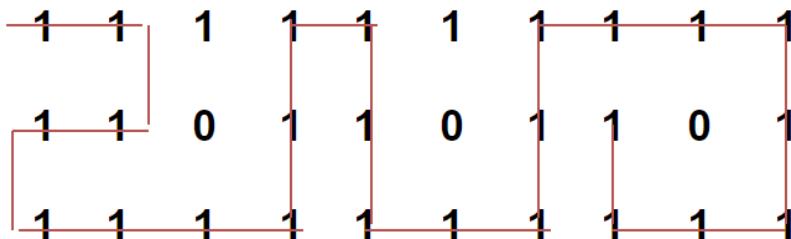
int src = 0;
int k = 62;
g.pathMoreThanK(src, k)? cout << "Yes\n" :
                           cout << "No\n";

k = 60;
g.pathMoreThanK(src, k)? cout << "Yes\n" :
                           cout << "No\n";

return 0;
}

```

Longest Possible Route in a Matrix with Hurdles



```

#include <iostream>
#include <vector>
using namespace std;

// Function for finding the longest path
// 'ans' is -1 if we can't reach
// 'cur' is the number of steps we have traversed
int findLongestPath(vector<vector<int> >& mat, int i, int j,
                     int di, int dj, int n, int m,
                     int cur = 0, int ans = -1)

```

```

{
    // If we reach the destination
    if (i == di && j == dj) {
        // If current path steps are more than previous path
        // steps
        if (cur > ans)
            ans = cur;
        return ans;
    }

    //if the source or destination is a hurdle itself
    if(mat[i][j]==0 || mat[di][dj]==0) return;

    // Mark as visited
    mat[i][j] = 0;

    // Checking if we can reach the destination going right
    if (j != m - 1 && mat[i][j + 1] > 0)
        ans = findLongestPath(mat, i, j + 1, di, dj, n, m,
                              cur + 1, ans);

    // Checking if we can reach the destination going down
    if (i != n - 1 && mat[i + 1][j] > 0)
        ans = findLongestPath(mat, i + 1, j, di, dj, n, m,
                              cur + 1, ans);

    // Checking if we can reach the destination going left
    if (j != 0 && mat[i][j - 1] > 0)
        ans = findLongestPath(mat, i, j - 1, di, dj, n, m,
                              cur + 1, ans);

    // Checking if we can reach the destination going up
    if (i != 0 && mat[i - 1][j] > 0)
        ans = findLongestPath(mat, i - 1, j, di, dj, n, m,
                              cur + 1, ans);

    // Marking visited to backtrack
    mat[i][j] = 1;

    // Returning the answer we got so far
    return ans;
}

```

```

int main()
{
    vector<vector<int>> mat = { { 1, 1, 1, 1 },
                                { 1, 1, 0, 1 },
                                { 1, 1, 1, 1 } };

    // Find the longest path with source (0, 0) and
    // destination (2, 3)
    int result = findLongestPath(mat, 0, 0, 2, 3,
                                mat.size(), mat[0].size());
    cout << result << endl;

    return 0;
}

```

Print all possible paths from top left to bottom right of a mXn matrix

Given a 2D matrix of dimension $m \times n$, the task is to print all the possible paths from the top left corner to the bottom right corner in a 2D matrix with the constraints that from each cell you can either move to right or down only.

Examples :

Input: [[1,2,3],
 [4,5,6]]

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

```

// Function to find all possible path in matrix from top
// left cell to bottom right cell
void findPaths(vector<vector<int>>& arr, vector<int>& path,
               int i, int j)
{

    // if the bottom right cell, print the path
    if (i == M - 1 && j == N - 1) {

```

```

        path.push_back(arr[i][j]);
        printPath(path);
        path.pop_back();
        return;
    }

    // Boundary cases: In case if we reach out of the matrix
    if (i < 0 || i >= M || j < 0 || j >= N) {
        return;
    }

    // Include the current cell in the path
    path.push_back(arr[i][j]);

    // Move right in the matrix
    if (j + 1 < N) {
        findPaths(arr, path, i, j + 1);
    }

    // Move down in the matrix
    if (i + 1 < M) {
        findPaths(arr, path, i + 1, j);
    }

    // Backtrack: Remove the current cell from the current
    // path
    path.pop_back();
}

// Driver code
int main()
{
    // Input matrix
    vector<vector<int>> arr
        = { { 1, 2, 3 }, { 4, 5, 6 }, { 7, 8, 9 } };

    // To store the path
    vector<int> path;

    // Starting cell `(0, 0)` cell
    int i = 0, j = 0;

    M = arr.size();
}

```

```

N = arr[0].size();

// Function call
findPaths(arr, path, i, j);

return 0;
}

```

Partition of a set into K subsets with equal sum

Given an integer array arr[] and an integer k, the task is to check if it is possible to divide the given array into k non-empty subsets of equal sum such that every array element is part of a single subset.

Examples:

Input: arr[] = [2, 1, 4, 5, 6], k = 3

Output: true

Explanation: Possible subsets of the given array are [2, 4], [1, 5] and [6]

Input: arr[] = [2, 1, 5, 5, 6], k = 3

Output: false

Explanation: It is not possible to divide above array into 3 parts with equal sum.

```

bool isKPartitionPossible(vector<int> &arr, vector<int> &subsetSum,
                         vector<bool> &taken, int target, int k,
                         int n, int currIdx, int limitIdx) {

    // If the current subset sum matches the target
    if (subsetSum[currIdx] == target) {

        // If all but one subset are filled, the
        // last subset is guaranteed to work
        if (currIdx == k - 2)
            return true;
        return isKPartitionPossible(arr, subsetSum, taken,
                                   target, k, n, currIdx + 1, n - 1);
    }
}

```

```

// Try including each element in the current subset
for (int i = limitIdx; i >= 0; i--) {

    // Skip if the element is already used
    if (taken[i])
        continue;
    int temp = subsetSum[currIdx] + arr[i];
    if (temp <= target) {

        // Only proceed if it doesn't exceed the target
        taken[i] = true;
        subsetSum[currIdx] += arr[i];
        if (isKPartitionPossible(arr, subsetSum, taken,
                               target, k, n, currIdx, i - 1))
            return true;

        // Backtrack
        taken[i] = false;
        subsetSum[currIdx] -= arr[i];
    }
}
return false;
}

bool isKPartitionPossible(vector<int> &arr, int k) {

    int n = arr.size(), sum = accumulate(arr.begin(), arr.end(), 0);

    // If only one subset is needed, it's always possible
    if (k == 1)
        return true;

    // Check if partition is impossible
    if (n < k || sum % k != 0)
        return false;

    int target = sum / k;
    vector<int> subsetSum(k, 0);
    vector<bool> taken(n, false);

    // Initialize first subset with the last element
    subsetSum[0] = arr[n - 1];
}

```

```

    taken[n - 1] = true;

    // Recursively check for partitions
    return isKPartitionPossible(arr, subsetSum, taken,
                                target, k, n, 0, n - 1);
}

```

Find the K-th Permutation Sequence of first N natural numbers

```

// Function to find the index of number
// at first position of
// kth sequence of set of size n
// precalculated factorials
int fact[10] = {1,1,2,6,24,120,720,5040,40320,362880};

// recursively insert numbers in to string
void permutation(int n, int k, set<int>&nums, string &str)
{
    // base case n==0 then no numbers to process
    if(n==0) return;

    int val;

    // base case k=1 then add numbers from begin
    // base case k=0 then next numbers to be added will be in reverse
from rbegin
    // k<=fact[n-1] then add the begin number
    if(k<=1 || k<=fact[n-1])
    {
        val = k==0 ? *nums.rbegin() : *nums.begin();
    }
    else
    {
        // calculate number of values cover k => k/fact[n-1]
        // so next value index => k/fact[n-1]
        int index = k/fact[n-1];
        k = k %fact[n-1];    // remaining permutations

        // also if k%fact[n-1] == 0 then kth permutation covered by
value is in index-1
    }
}

```

```

        // EX: [2,3] n=2, k=2 => index = k/fact[n-1] = 2/1 = 2
        // as k%fact[n-1] => 2%1 = 0, so decrease index to 1
        // so we take the value 3 as next value

        if(k==0)index--;

        // value taken
        val = *next(nums.begin(),index);
    }

    // add value to the string and remove from set
    str+= to_string(val);
    nums.erase(val);

    // decrement n in each step
    return permutation(n-1,k,nums,str);
}

string getPermutation(int n, int k) {

    // insert numbers 1 to N in to set
    set<int>nums;
    for(int i=1;i<=n;i++)nums.insert(i);

    // resulting string
    string str = "";

    permutation(n,k,nums,str);

    return str;
}

// Driver code
int main()
{
    int n = 3, k = 4;

    string kth_perm_seq
        = getPermutation(n, k);

    cout << kth_perm_seq << endl;
}

```

```
    return 0;
}
```

Greedy:

N meetings in one room

You are given timings of n meetings in the form of (start[i], end[i]) where start[i] is the start time of meeting i and end[i] is the finish time of meeting i. Return the maximum number of meetings that can be accommodated in a single meeting room, when only one meeting can be held in the meeting room at a particular time.

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

Sorting pair in increasing order of their ending time of meeting

```
class Solution
{
public:
    //Function to find the maximum number of meetings that can
    //be performed in a meeting room.
    bool static cap(pair<int,int> a,pair<int,int> b){
        if(a.second < b.second)  return true;
        return false;
    }
}
```

```
}

int maxMeetings(int start[], int end[], int n)
{
    // Your code here
    vector<pair<int,int>> vp;
    for(int i=0;i<n;i++){
        vp.push_back({start[i],end[i]});

    }
    sort(vp.begin(),vp.end(), cap);
    int ans = 1;
    int limit = vp[0].second;
    for(int i=1;i<n;i++){
        if(vp[i].first > limit ) {
            ans++;
            limit = vp[i].second;
        }
    }
    return ans;
}
};
```

Job Sequencing Problem

You are given three arrays: `id`, `deadline`, and `profit`, where each job is associated with an ID, a deadline, and a profit. Each job takes 1 unit of time to complete, and only one job can be scheduled at a time. You will earn the profit associated with a job only if it is completed by its deadline.

Your task is to find:

- The maximum number of jobs that can be completed within their deadlines.
- The total maximum profit earned by completing those jobs.

```
// code here

vector<pair<int,int>> vp;
for(int i=0;i<id.size();i++){
    vp.push_back({ded[i], pro[i]});
}
sort(vp.begin(),vp.end(), comp);
vector<bool> vis(id.size()+1, 0);
int ans=0, cnt=0;
for(int i=0;i<vp.size();i++){
    // cout<<vp[i].second<<" "<<vp[i].first<<endl;
    int temp = vp[i].first;
    while(temp--){
        if(vis[temp]==0){
            ans++;
            vis[temp]=1;
            cnt+=vp[i].second;
            break;
        }
    }
    // cout<<endl;
    return {ans, cnt};
}
};
```

Fractional Kancsak Problem:

Given two arrays, **val[]** and **wt[]**, representing the values and weights of items, and an integer **capacity** representing the maximum weight a knapsack can hold, determine the maximum total value that can be achieved by putting items in the knapsack. You are allowed to break items into fractions if necessary. Return the maximum value as a double, rounded to 6 decimal places.

Examples :

Input: val[] = [60, 100, 120], wt[] = [10, 20, 30], capacity = 50

Output: 240.000000

Explanation: Take the item with value 60 and weight 10, value 100 and weight 20 and split the third item with value 120 and weight 30, to fit it into weight 20. so it becomes $(120/30) \times 20 = 80$, so the total value becomes $60 + 100 + 80.0 = 240.0$. Thus, total maximum value of item we can have is 240.00 from the given capacity of sack.

```
//class implemented
/*
struct Item{
    int value;
    int weight;
};

class Solution
{
public:
    bool static cap (Item a,Item b){
        return (double)a.value/(double)a.weight
        >(double)b.value/(double)b.weight;
    }
    //Function to get the maximum total value in the knapsack.
    double fractionalKnapsack(int W, Item arr[], int n)
    {
        // Your code here
        sort(arr,arr+n,cap);
        double ans =0;
        for(int i=0;i<n;i++){
            if(arr[i].weight < W){
                ans+= (double)arr[i].value;
                W-=arr[i].weight;
            }
        }
        return ans;
    }
}
```

```

        }
    else{
        if(W!=0){
            double add = W * (arr[i].value/(double)arr[i].weight);
            ans+=add;
        }
        break;
    }
}
return ans;

}

};


```

Greedy Algorithm to find Minimum number of Coins or

Coin Change – Minimum Coins to Make Sum

Given a value of V Rs and an infinite supply of each of the denominations {1, 2, 5, 10, 20, 50, 100, 500, 1000} valued coins/notes, The task is to find the minimum number of coins and/or notes needed to make the change?

Greedy Approach:

Sort the array of coins in decreasing order.

Initialize ans vector as empty.

Find the largest denomination that is smaller than remaining amount and while it is smaller than the remaining amount:

Add found denomination to ans. Subtract value of found denomination from amount.

If amount becomes 0, then print ans.

Using Recursion – O(n^{sum}) Time and O(sum) Space

```

class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> minCoins(amount + 1, amount + 1);
        minCoins[0] = 0;
    }
};


```

```

        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.size(); j++) {
                if (i - coins[j] >= 0) {
                    minCoins[i] = min(minCoins[i], 1 + minCoins[i - coins[j]]);
                }
            }
        }

        return minCoins[amount] != amount + 1 ? minCoins[amount] : -1;
    }
};


```

```

// C++ program to find minimum of coins
// to make a given change sum
#include<bits/stdc++.h>
using namespace std;

int minCoinsRecur(int i, int sum, vector<int> &coins, vector<vector<int>>
&memo) {

    // base case
    if (sum == 0) return 0;
    if (sum < 0 || i == coins.size()) return INT_MAX;

    if (memo[i][sum] != -1) return memo[i][sum];

    int take = INT_MAX;

    // take a coin only if its value
    // is greater than 0.
    if (coins[i] > 0) {
        take = minCoinsRecur(i, sum - coins[i], coins, memo);
        if (take != INT_MAX) take++;
    }
    // not take the coins
    int noTake = minCoinsRecur(i+1, sum, coins, memo);

    return memo[i][sum] = min(take, noTake);
}

int minCoins(vector<int> &coins, int sum) {

```

```

vector<vector<int>> memo(coins.size(), vector<int>(sum+1, -1));
int res = minCoinsRecur(0, sum, coins, memo);
return res!=INT_MAX?res:-1;
}

int main() {
    vector<int> coins = {9, 6, 5, 1};
    int sum = 19;
    cout << minCoins(coins, sum);
    return 0;
}

```

Maximum trains for which stoppage can be provided

```

// CPP to design platform for maximum stoppage
#include <bits/stdc++.h>
using namespace std;

// number of platforms and trains
#define n 2
#define m 5

// function to calculate maximum trains stoppage
int maxStop(int arr[][3])
{
    // declaring vector of pairs for platform
    vector<pair<int, int> > vect[n + 1];

    // Entering values in vector of pairs
    // as per platform number
    // make departure time first element
    // of pair
    for (int i = 0; i < m; i++)
        vect[arr[i][2]].push_back(
            make_pair(arr[i][1], arr[i][0]));

    // sort trains for each platform as per
    // dept. time
    for (int i = 0; i <= n; i++)
        sort(vect[i].begin(), vect[i].end());
}

```

```

// perform activity selection approach
int count = 0;
for (int i = 0; i <= n; i++) {
    if (vect[i].size() == 0)
        continue;

    // first train for each platform will
    // also be selected
    int x = 0;
    count++;
    for (int j = 1; j < vect[i].size(); j++) {
        if (vect[i][j].second >=
                        vect[i][x].first) {
            x = j;
            count++;
        }
    }
}
return count;
}

// driver function
int main()
{
    int arr[m][3] = { 1000, 1030, 1,
                      1010, 1020, 1,
                      1025, 1040, 1,
                      1130, 1145, 2,
                      1130, 1140, 2 };
    cout << "Maximum Stopped Trains = "
          << maxStop(arr);
    return 0;
}

```

Minimum Platforms Required for Given Arrival and Departure Times

```
// Program to find minimum number of platforms
```

```
// required on a railway station
#include <bits/stdc++.h>
using namespace std;

// Returns minimum number of platforms required
int findPlatform(int arr[], int dep[], int n)
{
    // Sort arrival and departure arrays
    sort(arr, arr + n);
    sort(dep, dep + n);

    // plat_needed indicates number of platforms
    // needed at a time
    int plat_needed = 1, result = 1;
    int i = 1, j = 0;

    // Similar to merge in merge sort to process
    // all events in sorted order
    while (i < n && j < n) {
        // If next event in sorted order is arrival,
        // increment count of platforms needed
        if (arr[i] <= dep[j]) {
            plat_needed++;
            i++;
        }

        // Else decrement count of platforms needed
        else if (arr[i] > dep[j]) {
            plat_needed--;
            j++;
        }

        // Update result if needed
        if (plat_needed > result)
            result = plat_needed;
    }

    return result;
}

// Driver code
int main()
{
```

```

    int arr[] = { 100, 300, 500 };
    int dep[] = { 900, 400, 600 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << findPlatform(arr, dep, n);
    return 0;
}

```

```

// C++ program to implement the above approach
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum number of platforms
// required
int findPlatformOptimized(int arr[], int dep[], int n)
{
    int count = 0, maxPlatforms = 0;

    // Find the maximum departure time
    int maxDepartureTime = dep[0];
    for (int i = 1; i < n; i++) {
        maxDepartureTime = max(maxDepartureTime, dep[i]);
    }

    // Create a vector to store the count of trains at each
    // time
    vector<int> v(maxDepartureTime + 2, 0);

    // Increment the count at the arrival time and decrement
    // at the departure time
    for (int i = 0; i < n; i++) {
        v[arr[i]]++;
        v[dep[i] + 1]--;
    }

    // Iterate over the vector and keep track of the maximum
    // sum seen so far
    for (int i = 0; i <= maxDepartureTime + 1; i++) {
        count += v[i];
        maxPlatforms = max(maxPlatforms, count);
    }
}

```

```
        return maxPlatforms;
    }

// Driver Code
int main()
{
    int arr[] = { 100, 300, 600 };
    int dep[] = { 900, 400, 500 };
    int n = sizeof(arr) / sizeof(arr[0]);
    cout << findPlatformOptimized(arr, dep, n);
    return 0;
}
```

Buy Maximum Stocks if i stocks can be bought on i-th day

```

        return ans;
    }

// Driven Program
int main()
{
    int price[] = { 10, 7, 19 };
    int n = sizeof(price)/sizeof(price[0]);
    int k = 45;

    cout << buyMaximumProducts(n, k, price) << endl;

    return 0;
}

```

Min and Max Costs to Buy All Candies

In a candy store, there are N different types of candies available and the prices of all the N different types of candies are provided. There is also an attractive offer by the candy store. We can buy a single candy from the store and get at most K other candies (all are different types) for free.

Find the minimum amount of money we have to spend to buy all the N different candies.
 Find the maximum amount of money we have to spend to buy all the N different candies.
 In both cases, we must utilize the offer and get the maximum possible candies back. If k or more candies are available, we must take k candies for every candy purchase. If less than k candies are available, we must take all candies for a candy purchase.

Examples:

Input : price[] = {3, 2, 1, 4}, k = 2

Output : Min = 3, Max = 7

Explanation :

Since k is 2, if we buy one candy we can take atmost two more for free. So in the first case we buy the candy which costs 1 and take candies worth 3 and 4 for free, also you buy candy worth 2 as well. So min cost = $1 + 2 = 3$.

In the second case we buy the candy which costs 4 and take candies worth 1 and 2 for free, also We buy candy worth 3 as well. So max cost = $3 + 4 = 7$.

Input: n = 5, k = 4, candies[] = {3 2 1 4 5}

Output: 1 5

Explanation: For minimum cost buy the candy with the cost 1 and get all the other candies for free. For maximum cost buy the candy with the cost 5 and get all other candies for free.

First Sort the given array.

For finding minimum amount : Start purchasing candies from starting and reduce k free candies from last with every single purchase.

For finding maximum amount : Start purchasing candies from the end and reduce k free candies from starting in every single purchase.

```
#include <bits/stdc++.h>
using namespace std;

// Function to find the minimum and maximum amount
// to buy all candies
vector<int> findMinAndMax(vector<int>& arr, int k) {

    // Sort the array to arrange candies by price
    sort(arr.begin(), arr.end());

    // Calculate minimum cost by traversing
    // from beginning
    int n = arr.size();
    int minCost = 0;
    for (int i = 0; i < n; i++) {
        minCost += arr[i];
        n -= k;
    }

    // calculate maximum cost by traversing
    // from end
    int index = 0, maxCost = 0;
    for (int i = arr.size() - 1; i >= index; i--) {
        maxCost += arr[i];
        index += k;
    }

    return {minCost, maxCost};
}

int main() {
    vector<int> arr = {3, 2, 1, 4};
    int k = 2;
    vector<int> res = findMinAndMax(arr, k);
```

```

    cout << res[0] << " " << res[1] << endl;
    return 0;
}

```

Minimize Cash Flow among a given set of friends who have borrowed money from each other

/

```

/ C++ program to find maximum cash flow among a set of
// persons
#include <bits/stdc++.h>
using namespace std;

// Number of persons (or vertices in the graph)
#define N 3

// A utility function that returns index of minimum value in
// arr[]
int getMin(int arr[])
{
    int minInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns index of maximum value in
// arr[]
int getMax(int arr[])
{
    int maxInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

```

```

// A utility function to return minimum of 2 values
int minOf2(int x, int y) { return (x < y) ? x : y; }

// amount[p] indicates the net amount to be credited/debited
// to/from person 'p'
// If amount[p] is positive, then i'th person will amount[i]
// If amount[p] is negative, then i'th person will give
// -amount[i]
void minCashFlowRec(int amount[])
{
    // Find the indexes of minimum and maximum values in
    // amount[] amount[mxCredit] indicates the maximum
    // amount to be given
    //           (or credited) to any person .
    // And amount[mxDabit] indicates the maximum amount to
    // be taken
    //           (or debited) from any person.
    // So if there is a positive value in amount[], then
    // there must be a negative value
    int mxCredit = getMax(amount), mxDebit = getMin(amount);

    // If both amounts are 0, then all amounts are settled
    if (amount[mxCredit] == 0 && amount[mxDabit] == 0)
        return;

    // Find the minimum of two amounts
    int min = minOf2(-amount[mxDabit], amount[mxCredit]);
    amount[mxCredit] -= min;
    amount[mxDabit] += min;

    // If minimum is the maximum amount to be
    cout << "Person " << mxDebit << " pays " << min
        << " to "
        << "Person " << mxCredit << endl;

    // Recur for the amount array. Note that it is
    // guaranteed that the recursion would terminate as
    // either amount[mxCredit] or amount[mxDabit] becomes 0
    minCashFlowRec(amount);
}

// Given a set of persons as graph[] where graph[i][j]

```

```

// indicates the amount that person i needs to pay person j,
// this function finds and prints the minimum cash flow to
// settle all debts.
void minCashFlow(int graph[][N])
{
    // Create an array amount[], initialize all value in it
    // as 0.
    int amount[N] = { 0 };

    // Calculate the net amount to be paid to person 'p',
    // and stores it in amount[p]. The value of amount[p]
    // can be calculated by subtracting debts of 'p' from
    // credits of 'p'
    for (int p = 0; p < N; p++)
        for (int i = 0; i < N; i++)
            amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

// Driver program to test above function
int main()
{
    // graph[i][j] indicates the amount that person i needs
    // to pay person j
    int graph[N][N] = {
        { 0, 1000, 2000 },
        { 0, 0, 5000 },
        { 0, 0, 0 },
    };

    // Print the solution
    minCashFlow(graph);
    return 0;
}

```

```

// C++ program to find maximum cash flow among a set of
// persons
#include <bits/stdc++.h>
using namespace std;

// Number of persons (or vertices in the graph)

```

```

#define N 3

// A utility function that returns index of minimum value in
// arr[]
int getMin(int arr[])
{
    int minInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] < arr[minInd])
            minInd = i;
    return minInd;
}

// A utility function that returns index of maximum value in
// arr[]
int getMax(int arr[])
{
    int maxInd = 0;
    for (int i = 1; i < N; i++)
        if (arr[i] > arr[maxInd])
            maxInd = i;
    return maxInd;
}

// A utility function to return minimum of 2 values
int minOf2(int x, int y) { return (x < y) ? x : y; }

// amount[p] indicates the net amount to be credited/debited
// to/from person 'p'
// If amount[p] is positive, then i'th person will amount[i]
// If amount[p] is negative, then i'th person will give
// -amount[i]
void minCashFlowRec(int amount[])
{
    // Find the indexes of minimum and maximum values in
    // amount[] amount[mxCredit] indicates the maximum
    // amount to be given
    // (or credited) to any person .
    // And amount[mxD debit] indicates the maximum amount to
    // be taken
    // (or debited) from any person.
    // So if there is a positive value in amount[], then
    // there must be a negative value
}

```

```

int mxCredit = getMax(amount), mxDebit = getMin(amount);

// If both amounts are 0, then all amounts are settled
if (amount[mxCredit] == 0 && amount[mxDebit] == 0)
    return;

// Find the minimum of two amounts
int min = minOf2(-amount[mxDebit], amount[mxCredit]);
amount[mxCredit] -= min;
amount[mxDebit] += min;

// If minimum is the maximum amount to be
cout << "Person " << mxDebit << " pays " << min
    << " to "
    << "Person " << mxCredit << endl;

// Recur for the amount array. Note that it is
// guaranteed that the recursion would terminate as
// either amount[mxCredit] or amount[mxDebit] becomes 0
minCashFlowRec(amount);
}

// Given a set of persons as graph[] where graph[i][j]
// indicates the amount that person i needs to pay person j,
// this function finds and prints the minimum cash flow to
// settle all debts.
void minCashFlow(int graph[][N])
{
    // Create an array amount[], initialize all value in it
    // as 0.
    int amount[N] = { 0 };

    // Calculate the net amount to be paid to person 'p',
    // and stores it in amount[p]. The value of amount[p]
    // can be calculated by subtracting debts of 'p' from
    // credits of 'p'
    for (int p = 0; p < N; p++)
        for (int i = 0; i < N; i++)
            amount[p] += (graph[i][p] - graph[p][i]);

    minCashFlowRec(amount);
}

```

```
// Driver program to test above function
int main()
{
    // graph[i][j] indicates the amount that person i needs
    // to pay person j
    int graph[N][N] = {
        { 0, 1000, 2000 },
        { 0, 0, 5000 },
        { 0, 0, 0 },
    };

    // Print the solution
    minCashFlow(graph);
    return 0;
}
```

Dynamic Programming:

0 - 1 Knapsack Problem

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 dont pick it (0-1 property).

Example 1:

Input:

N = 3

W = 4

values[] = {1,2,3}

weight[] = {4,5,1}

Output: 3

Explanation: Choose the last item that weighs 1 unit and holds a value of 3.

Example 2:

Input:

N = 3

W = 3

values[] = {1,2,3}

weight[] = {4,5,6}

Output: 0

Explanation: Every item has a weight exceeding the knapsack's capacity (3).

```
class Solution
{
public:
    //Function to return max value that can be put in knapsack of capacity
    W.
    int knapSack(int W, int wt[], int val[], int n)
    {
        // Your code here
        vector<vector<int>>v(n+1,vector<int>(W+1,0));
        vector<pair<int,int>>vp;
        for(int i=0;i<n;i++){
            vp.push_back({wt[i],val[i]});
        }
        sort(vp.begin(),vp.end());
        reverse(vp.begin(),vp.end());
        for(int i=1;i<=n;i++){
            for(int j=1;j<=W;j++){
                if(j-wt[i]<0) v[i][j]=v[i-1][j];
                else v[i][j]=max(v[i-1][j],v[i-1][j-wt[i]]+val[i]);
            }
        }
        return v[n][W];
    }
};
```

```

        vp.push_back({wt[i],val[i]});
    }
sort(vp.begin(), vp.end());
for(int i=0;i<=n;i++){
    for(int j=1;j<=W;j++){
        if(i==0 or j==0){
            v[i][j]=0;
            continue;
        }
        if(j<vp[i-1].first){
            v[i][j] = v[i-1][j];
        }
        else{
            v[i][j] = max(v[i-1][j], vp[i-1].second +
v[i-1][max(j-vp[i-1].first,0)]);
        }
    }
    return v[n][W];
}
};


```

Edit Distance

```

class Solution {
public:
    int editDistance(string s, string t) {
        // Code here
        int sn = s.size(), tn = t.size();
        vector<vector<int>> v(tn+1, vector<int>(sn+1,0));
        for(int i=0;i<=tn;i++){
            for(int j=0;j<=sn;j++){
                if(i == 0 and j == 0) v[i][j]=0;
                if(i == 0) v[i][j] = j;
                else if(j == 0) v[i][j] = i;
                else{
                    if(t[i-1] == s[j-1]){
                        v[i][j] = v[i-1][j-1];
                    }
                    else{
                        v[i][j] = min(v[i-1][j], v[i][j-1]);
                        v[i][j]++;
                    }
                }
            }
        }
        return v[tn][sn];
    }
};


```

```

        }
    else{
        v[i][j] = min(v[i][j-1], min(v[i-1][j],
v[i-1][j-1])) +1;
    }
}
return v[tn][sn];
}
};


```

Partition Equal Subset Sum

```

class Solution{
public:
    int equalPartition(int N, int nums[])
    {
        // code here

        long long sum = 0;

        for(int i = 0; i < N; i++)
            sum+=nums[i];

        long long expectedSum = sum/2;

        if(sum & 0x01)
            return false;

        for(int i = 0; i < N; i++)
        {
            if(expectedSum < nums[i])
                return false;
        }

        bool dp[N+1][expectedSum+1];

        for(int i = 0; i < N+1; i++)
        {
            for(int j = 0; j < expectedSum + 1; j++)

```

```

    {
        if(i == 0 && j == 0)
            dp[i][j] = true;
        else if(i==0)
            dp[i][j] = false;
        else if(j==0)
            dp[i][j] = true;
        else
        {
            dp[i][j] = dp[i-1][j];

            if(j>=nums[i-1])
            {
                dp[i][j] = dp[i][j] || dp[i-1][j-nums[i-1]];
            }
        }
    }

    return dp[N][expectedSum];
}
};


```

Longest Common Subsequence

```

// function to find longest common subsequence

class Solution {
public:
    // Function to find the length of longest common subsequence in two
    strings.
    int lcs(int n, int m, string s1, string s2) {
        // your code here
        vector<vector<int>>v(n+1,vector<int>(m+1,0));
        int ans =0;
        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                // if(i==0 or j==0) v[i][j] = 0;
                if(s1[i-1]==s2[j-1]){

```

```

        v[i][j]=v[i-1][j-1] +1;
    }else{
        v[i][j]=max(v[i-1][j],v[i][j-1]);
    }
    ans = max(ans, v[i][j]);
}
}
return ans;
}
};


```

Longest Common Substring

```

class Solution {
public:
    // Function to find the length of longest common subsequence in two
    strings.
    int lcs(int n, int m, string s1, string s2) {
        // your code here
        vector<vector<int>>v(n+1,vector<int>(m+1,0));
        int ans =0;
        for(int i=1;i<=n;i++){
            for(int j=1;j<=m;j++){
                // if(i==0 or j==0) v[i][j] = 0;
                if(s1[i-1]==s2[j-1]){
                    v[i][j]=v[i-1][j-1] +1;
                }else{
                    v[i][j]=0;
                }
                ans = max(ans, v[i][j]);
            }
        }
        return ans;
    }
};


```

Coin Change (Calculate total subsets to form sum)

```
int change(int sum, vector<int>& coins) {
```

```

int n = coins.size();
// 2d dp array where n is the number of coin
// denominations and sum is the target sum
vector<vector<int>> dp(n + 1, vector<int>(sum + 1, 0));

// Represents the base case where the target sum is 0,
// and there is only one way to make change: by not
// selecting any coin
dp[0][0] = 1;
for (int i = 1; i <= n; i++) {
    for (int j = 0; j <= sum; j++) {

        // Add the number of ways to make change without
        // using the current coin,
        dp[i][j] += dp[i - 1][j];

        if ((j - coins[i - 1]) >= 0) {

            // Add the number of ways to make change
            // using the current coin
            dp[i][j] += dp[i][j - coins[i - 1]];
        }
    }
}
return dp[n][sum];
}

```

Pascal's Triangle:

```

class Solution {
public:
    vector<vector<int>> generate(int n) {
        vector<vector<int>> ans;
        for(int i=0;i<n;i++){
            vector<int> v;
            for(int j=0;j<=i;j++){
                if(j==0 || j==i) v.push_back(1);
                else{
                    v.push_back(ans[i-1][j]+ans[i-1][j-1]);
                }
            }
            ans.push_back(v);
        }
        return ans;
    }
};

```

```

        }
        ans.push_back(v);
    }
    return ans;
}

```

Catalean Number

- Given an integer N, the task is to find the number of binary strings of size 2^N such that each string has exactly N 1's and each prefix of the string has more than or an equal number of 1's than 0's.
- Given a number n, return the number of ways you can draw n chords in a circle with $2 \times n$ points such that no 2 chords intersect.
- Count the number of expressions containing n pairs of parentheses that are correctly matched. For $n = 3$, possible expressions are $((())), ()((), ()()()$.

Note: The answer can be very large. So, output answer modulo $10^9 + 7$

```

int prefixStrings(int N)
{
    // Your code goes here
    long long mod = 1e9+7;
    vector<long long> v(N+1, 0);
    v[0]=v[1]=1;

```

```
for(int i=2;i<=N;i++){
    for(int j=0;j<i;j++){
        v[i]+=v[j]*v[i-j-1];
        v[i]%=mod;
    }
}
return v[N];
}
```

Gold Mine Problem:

Given a gold mine called M of ($n \times m$) dimensions. Each field in this mine contains a positive integer which is the amount of gold in tons. Initially the miner can start from any row in the first column. From a given cell, the miner can move

1. to the cell diagonally up towards the right
2. to the right
3. to the cell diagonally down towards the right

Find out maximum amount of gold which he can collect until he can no longer move.

Example 1:

Input: $n = 3, m = 3$

$M = \{\{1, 3, 3\},$
 $\{2, 1, 4\},$
 $\{0, 6, 4\}\};$

Output: 12

Explanation:

The path is $\{(1,0) \rightarrow (2,1) \rightarrow (2,2)\}$

```
class Solution{
public:
    int maxGold(int m, int n, vector<vector<int>> mat)
    {
        // code here
        vector<vector<int>> v = mat;
        for(int j=n-2;j>=0;j--){
            for(int i=0;i<m;i++){
                if(i-1>=0 and i+1<=m-1){
                    v[i][j] += max({v[i-1][j+1], v[i][j+1], v[i+1][j+1]});
                }
                else if(i-1 < 0 and i+1>=m){
                    v[i][j] += v[i][j+1];
                }else if(i-1<0){
                    v[i][j] += max(v[i][j+1], v[i+1][j+1]);
                }else{
                    v[i][j] += max(v[i][j+1], v[i-1][j+1]);
                }
            }
        }
        int ans = v[0][0];
        for(int i=0;i<m;i++){
            ans = max(ans, v[i][0]);
        }
        return ans;
    }
};
```

Subset Sum Problem

Given an array $arr[]$ of non-negative integers and a value sum , the task is to check if there is a subset of the given array whose sum is equal to the given sum.

Naive:

```
//C++ implementation for subset sum
// problem using recursion
#include <bits/stdc++.h>
using namespace std;
```

```

// Function to check if there is a subset
// with the given sum using recursion
bool isSubsetSumRec(vector<int>& arr, int n, int sum) {

    // Base Cases
    if (sum == 0)
        return true;
    if (n == 0)
        return false;

    // If last element is greater than sum,
    // then ignore it
    if (arr[n - 1] > sum)
        return isSubsetSumRec(arr, n - 1, sum);

    // Check if sum can be obtained by including
    // or excluding the last element
    return isSubsetSumRec(arr, n - 1, sum)
        || isSubsetSumRec(arr, n - 1, sum - arr[n - 1]);
}

bool isSubsetSum(vector<int>& arr, int sum) {
    return isSubsetSumRec(arr, arr.size(), sum);
}

int main() {

    vector<int> arr = {3, 34, 4, 12, 5, 2};
    int sum = 9;

    if (isSubsetSum(arr, sum))
        cout << "True" << endl;
    else
        cout << "False" << endl;

    return 0;
}

```

Using Top-Down DP (Memoization) – O(sum*n) Time and O(sum*n) Space

```

//C++ implementation for subset sum
// problem using memoization

```

```

#include <bits/stdc++.h>
using namespace std;

// Recursive function to check if a subset
// with the given sum exists
bool isSubsetSumRec(vector<int>& arr, int n, int sum,
                     vector<vector<int>> &memo) {

    // If the sum is zero, we found a subset
    if (sum == 0)
        return 1;

    // If no elements are left
    if (n <= 0)
        return 0;

    // If the value is already
    // computed, return it
    if (memo[n][sum] != -1)
        return memo[n][sum];

    // If the last element is greater than
    // the sum, ignore it
    if (arr[n - 1] > sum)
        return memo[n][sum] = isSubsetSumRec(arr, n - 1, sum, memo);
    else {

        // Include or exclude the last element
        return memo[n][sum] = isSubsetSumRec(arr, n - 1, sum, memo) ||
                           isSubsetSumRec(arr, n - 1, sum - arr[n - 1],
                                         memo);
    }
}

// Function to initiate the subset sum check
bool isSubsetSum(vector<int>&arr, int sum) {
    int n = arr.size();

    vector<vector<int>> memo(n + 1, vector<int>(sum + 1, -1));
    return isSubsetSumRec(arr, n, sum, memo);
}

int main() {

```

```

vector<int>arr = {1, 5, 3, 7, 4};
int sum = 12;

if (isSubsetSum(arr, sum)) {
    cout << "True" << endl;
}
else {
    cout << "False" << endl;
}

return 0;
}

```

Using Bottom-Up DP (Tabulation) – O(sum*n) Time and O(sum*n) Space

The approach is similar to the previous one. just instead of breaking down the problem recursively, we **iteratively** build up the solution by calculating in bottom-up manner.

*So we will create a 2D array of size $(n + 1) * (sum + 1)$ of type boolean. The state $dp[i][j]$ will be true if there exists a subset of elements from $arr[0 \dots i]$ with $sum = j$.*

The dynamic programming relation is as follows:

```

if (arr[i-1] > j)
    dp[i][j] = dp[i-1][j]
else
    dp[i][j] = dp[i-1][j] OR dp[i-1][j-arr[i-1]]

```

```

//C++ implementation for subset sum
// problem using tabulation
#include <bits/stdc++.h>
using namespace std;

// Function to check if there is a subset of arr[]
// with sum equal to the given sum using tabulation with vectors
bool isSubsetSum(vector<int> &arr, int sum) {
    int n = arr.size();

    // Create a 2D vector for storing results
    // of subproblems
    vector<vector<bool>> dp(n + 1, vector<bool>(sum + 1, false));

```

```

// If sum is 0, then answer is true (empty subset)
for (int i = 0; i <= n; i++)
    dp[i][0] = true;

// Fill the dp table in bottom-up manner
for (int i = 1; i <= n; i++) {

    for (int j = 1; j <= sum; j++) {
        if (j < arr[i - 1]) {

            // Exclude the current element
            dp[i][j] = dp[i - 1][j];
        }
        else {

            // Include or exclude
            dp[i][j] = dp[i - 1][j]
            || dp[i - 1][j - arr[i - 1]];
        }
    }
}

return dp[n][sum];
}

int main() {

    vector<int> arr = {3, 34, 4, 12, 5, 2};
    int sum = 9;

    if (isSubsetSum(arr, sum))
        cout << "True" << endl;
    else
        cout << "False" << endl;

    return 0;
}

```

```

// C++ Program for Space Optimized Dynamic Programming
// Solution to Subset Sum Problem
#include <bits/stdc++.h>

```

```

using namespace std;

// Returns true if there is a subset of arr[]
// with sum equal to given sum
bool isSubsetSum(vector<int> arr, int sum) {
    int n = arr.size();
    vector<bool> prev(sum + 1, false), curr(sum + 1);

    // Mark prev[0] = true as it is true
    // to make sum = 0 using 0 elements
    prev[0] = true;

    // Fill the subset table in
    // bottom up manner
    for (int i = 1; i <= n; i++) {
        for (int j = 0; j <= sum; j++) {
            if (j < arr[i - 1])
                curr[j] = prev[j];
            else
                curr[j] = (prev[j] || prev[j - arr[i - 1]]);
        }
        prev = curr;
    }
    return prev[sum];
}

int main() {
    vector<int> arr = {3, 34, 4, 12, 5, 2};
    int sum = 9;
    if (isSubsetSum(arr, sum) == true)
        cout << "True";
    else
        cout << "False";
    return 0;
}

```

Time Complexity: $O(\text{sum} * \text{n})$, where n is the size of the array.

Auxiliary Space: $O(\text{sum})$, as the size of the 1-D array is $\text{sum}+1$.

Find the first non-repeating character from a stream of characters

```

class Solution {
public:

```

```

int firstUniqChar(string s) {
    vector<int> freq(27,-1);
    int n = s.size();
    for(int i=0;i<n;i++){
        freq[s[i]-'a']+=1;
    }

    int idx = 0;
    for(int i=0;i<n;i++){
        int val = s[i]-'a';
        if(freq[val]==0){
            return i;
        }
    }

    return -1;
}
};

```

```

char nonRep(const string& s) {
    vector<int> vis(MAX_CHAR, -1);
    for (int i = 0; i < s.length(); ++i) {
        int index = s[i] - 'a';
        if (vis[index] == -1) {

            // Store the index when character is first seen
            vis[index] = i;
        } else {

            // Mark character as repeated
            vis[index] = -2;
        }
    }

    int idx = -1;

    // Find the smallest index of the non-repeating characters
    for (int i = 0; i < MAX_CHAR; ++i) {
        if (vis[i] >= 0 && (idx == -1 || vis[i] < vis[idx])) {
            idx = i;
        }
    }
}

```

```
    }
    return (idx == -1) ? '$' : s[vis[idx]];
}
```

Stack and Queues

Design a Stack that supports getMin() in O(1) time and O(1) extra space.

Follow the given steps to implement the stack operations:

Push(x): Insert x at the top of the stack

If the stack is empty, insert x into the stack and make minEle equal to x.

If the stack is not empty, compare x with minEle. Two cases arise:

If x is greater than or equal to minEle, simply insert x.

If x is less than minEle, insert $(2*x - \text{minEle})$ into the stack and make minEle equal to x.

For example, let the previous minEle be 3. Now we want to insert 2. We update minEle as 2 and insert $2*2 - 3 = 1$ into the stack

Pop(): Removes an element from the top of the stack

Remove the element from the top. Let the removed element be y. Two cases arise:

If y is greater than or equal to minEle, the minimum element in the stack is still minEle.

If y is less than minEle, the minimum element now becomes $(2*\text{minEle} - y)$, so update ($\text{minEle} = 2*\text{minEle} - y$). This is where we retrieve the previous minimum from the current minimum and its value in the stack.

For example, let the element to be removed be 1 and minEle be 2. We remove 1 and update minEle as $2*2 - 1 = 3$

```
// C++ program to implement a stack that supports
// getMinimum() in O(1) time and O(1) extra space.
#include <bits/stdc++.h>
using namespace std;

// A user defined stack that supports getMin() in
// addition to push() and pop()
struct MyStack {
```

```
stack<int> s;
int minEle;

// Prints minimum element of MyStack
void getMin()
{
    if (s.empty())
        cout << "Stack is empty\n";

    // variable minEle stores the minimum element
    // in the stack.
    else
        cout << "Minimum Element in the stack is: "
            << minEle << "\n";
}

// Prints top element of MyStack
void peek()
{
    if (s.empty()) {
        cout << "Stack is empty ";
        return;
    }

    int t = s.top(); // Top element.

    cout << "Top Most Element is: ";

    // If t < minEle means minEle stores
    // value of t.
    (t < minEle) ? cout << minEle : cout << t;
}

// Remove the top element from MyStack
void pop()
{
    if (s.empty()) {
        cout << "Stack is empty\n";
        return;
    }

    cout << "Top Most Element Removed: ";
    int t = s.top();
```

```
s.pop();

// Minimum will change as the minimum element
// of the stack is being removed.
if (t < minEle) {
    cout << minEle << "\n";
    minEle = 2 * minEle - t;
}

else
    cout << t << "\n";
}

// Removes top element from MyStack
void push(int x)
{
    // Insert new number into the stack
    if (s.empty()) {
        minEle = x;
        s.push(x);
        cout << "Number Inserted: " << x << "\n";
        return;
    }

    // If new number is less than minEle
    else if (x < minEle) {
        s.push(2 * x - minEle);
        minEle = x;
    }

    else
        s.push(x);

    cout << "Number Inserted: " << x << "\n";
}
};

// Driver Code
int main()
{
    MyStack s;

    // Function calls
```

```

    s.push(3);
    s.push(5);
    s.getMin();
    s.push(2);
    s.push(1);
    s.getMin();
    s.pop();
    s.getMin();
    s.pop();
    s.peek();

    return 0;
}

```

Next Greater Element

Using a stack

```

// A Stack based C++ program to find next
// greater element for all array elements.
#include <bits/stdc++.h>
using namespace std;

/* prints element and NGE pair for all
elements of arr[] of size n */
void printNGE(int arr[], int n)
{
    stack<int> s;

    /* push the first element to stack */
    s.push(arr[0]);

    // iterate for rest of the elements
    for (int i = 1; i < n; i++) {

        if (s.empty()) {
            s.push(arr[i]);
            continue;

```

```

    }

    /* if stack is not empty, then
       pop an element from stack.
       If the popped element is smaller
       than next, then
       a) print the pair
       b) keep popping while elements are
       smaller and stack is not empty */
    while (s.empty() == false && s.top() < arr[i]) {
        cout << s.top() << " --> " << arr[i] << endl;
        s.pop();
    }

    /* push next to stack so that we can find
       next greater for it */
    s.push(arr[i]);
}

/* After iterating over the loop, the remaining
   elements in stack do not have the next greater
   element, so print -1 for them */
while (s.empty() == false) {
    cout << s.top() << " --> " << -1 << endl;
    s.pop();
}
}

/* Driver code */
int main()
{
    int arr[] = { 11, 13, 21, 3 };
    int n = sizeof(arr) / sizeof(arr[0]);
    printNGE(arr, n);
    return 0;
}

```

The Celebrity Problem

[Naive Approach] – Using Adjacency List – O(n^2) Time and O(n) Space

Analyze the problem using [Graphs](#). Initialize **indegree** and **outdegree** of every vertex as 0. If A knows B, draw a **directed edge** from A to B, increase indegree of B and outdegree of A by 1. Construct all possible edges of the graph for every possible pair $[i, j]$. There are $N C_2$ pairs. If a celebrity is present in the party, there will be one node in the graph which has the outdegree as zero and indegree equals to $N-1$.

Follow the steps below to solve the problem:

- Create two arrays **indegree** and **outdegree**, to store the indegree and outdegree
- Run a nested loop, the outer loop from 0 to n and inner loop from **0 to n**.
- For every pair i, j check if i knows j then increase the outdegree of i and indegree of j .
- For every pair i, j check if j knows i then increase the outdegree of j and indegree of i .
- Run a loop from **0 to n** and find the id where the indegree is **n-1** and outdegree is **0**.

```
bool knows(int a, int b, vector<vector<int> >& matrix)
{
    return matrix[a][b];
}

// Returns -1 if celebrity
// is not present. If present,
// returns id (value from 0 to n-1).
int findCelebrity(int n, vector<vector<int> >& matrix)
{
    // the graph needs not be constructed
    // as the edges can be found by
    // using knows function

    // degree array;
    int indegree[n] = { 0 }, outdegree[n] = { 0 };

    // query for all edges
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; j++) {
            int x = knows(i, j, matrix);

            // set the degrees
            outdegree[i] += x;
            indegree[j] += x;
        }
    }
}
```

```

    // find a person with indegree n-1
    // and out degree 0
    for (int i = 0; i < n; i++)
        if (indegree[i] == n - 1 && outdegree[i] == 0)
            return i;

    return -1;
}

```

[Efficient Approach] Using Stack- O(n) Time and O(n) Space

```

int findCelebrity(int n, vector<vector<int> >& matrix)
{

    stack<int> s;
    // Celebrity
    int C;

    // Push everybody to stack
    for (int i = 0; i < n; i++)
        s.push(i);

    // Extract top 2

    // Find a potential celebrity
    while (s.size() > 1) {
        int A = s.top();
        s.pop();
        int B = s.top();
        s.pop();
        if (knows(A, B, matrix)) {
            s.push(B);
        }
        else {
            s.push(A);
        }
    }

    // Potential candidate?
    C = s.top();
}

```

```

s.pop();

// Check if C is actually
// a celebrity or not
for (int i = 0; i < n; i++) {
    // If any person doesn't
    // know 'C' or 'C' doesn't
    // know any person, return -1
    if ((i != C)
        && (knows(C, i, matrix)
            || !knows(i, C, matrix)))
        return -1;
}

return C;
}

```

[Expected Approach] Using Two Pointers – O(n) Time and O(1) Space

Follow the steps below to solve the problem:

Create two indices i and j, where i = 0 and j = n-1

Run a loop until i is less than j.

Check if i knows j, then i can't be a celebrity. so increment i, i.e. i++

Else j cannot be a celebrity, so decrement j, i.e. j–

Assign i as the celebrity candidate

Now at last check whether the candidate is actually a celebrity by re-running a loop from 0 to n-1 and constantly checking if the candidate knows a person or if there is a candidate who does not know the candidate.

Then we should return -1. else at the end of the loop, we can be sure that the candidate is actually a celebrity.

```

int celebrity(vector<vector<int> >& matrix, int n)
{
    // This function returns the celebrity
    // index 0-based (if any)

    int i = 0, j = n - 1;
    while (i < j) {
        if (matrix[j][i] == 1) // j knows i
            j--;

```

```

        else // j doesnt know i so i cant be celebrity
            i++;
    }
    // i points to our celebrity candidate
    int candidate = i;

    // Now, all that is left is to check that whether
    // the candidate is actually a celebrity i.e: he is
    // known by everyone but he knows no one
    for (i = 0; i < n; i++) {
        if (i != candidate) {
            if (matrix[i][candidate] == 0
                || matrix[candidate][i] == 1)
                return -1;
        }
    }
    // if we reach here this means that the candidate
    // is really a celebrity
    return candidate;
}

```

Arithmetic Expression Using Stack

<https://www.geeksforgeeks.org/arithmetic-expression-evaluation/>

Evaluate a Postfix Expression

```

class Solution {
public:
    int do_op(char op, int a, int b){
        switch(op){
            case '+':
                return a+b;
                break;
            case '-':
                return a-b;
                break;
        }
    }
};

```

```

        case '*':
            return a*b;
            break;
        case '/':
            return a/b;
            break;
    }
}

// Function to evaluate a postfix expression.
int evaluatePostfix(string &s) {
    // Your code here
    stack<int> stk;

    int ans=0;
    for(int i=0;i<s.size();i++){
        if(isdigit(s[i])){
            stk.push(s[i]-'0');
        }
        else{
            int A=stk.top();
            stk.pop();
            int B=stk.top();
            stk.pop();
            // cout<<A<<B<<endl;
            int ans = do_op(s[i], B, A);
            stk.push(ans);
        }
    }
    return stk.top();
}

};


```

Reverse a stack using recursion

```

// C++ code to reverse a
// stack using recursion
#include <bits/stdc++.h>
using namespace std;


```

```
// Below is a recursive function
// that inserts an element
// at the bottom of a stack.
void insert_at_bottom(stack<int>& st, int x)
{
    if (st.size() == 0) {
        st.push(x);
    }
    else {

        // All items are held in Function Call
        // Stack until we reach end of the stack
        // When the stack becomes empty, the
        // st.size() becomes 0, the above if
        // part is executed and the item is
        // inserted at the bottom

        int a = st.top();
        st.pop();
        insert_at_bottom(st, x);

        // push allthe items held in
        // Function Call Stack
        // once the item is inserted
        // at the bottom
        st.push(a);
    }
}

// Below is the function that
// reverses the given stack using
// insert_at_bottom()
void reverse(stack<int>& st)
{
    if (st.size() > 0) {

        // Hold all items in Function
        // Call Stack until we
        // reach end of the stack
        int x = st.top();
        st.pop();
        reverse(st);
        insert_at_bottom(st, x);
    }
}
```

```
// Insert all the items held
// in Function Call Stack
// one by one from the bottom
// to top. Every item is
// inserted at the bottom
insert_at_bottom(st, x);
}

return;
}

// Driver Code
int main()
{
    stack<int> st, st2;
    // push elements into
    // the stack
    for (int i = 1; i <= 4; i++) {
        st.push(i);
    }

    st2 = st;

    cout << "Original Stack" << endl;
    // printing the stack after reversal
    while (!st2.empty()) {
        cout << st2.top() << " ";
        st2.pop();
    }
    cout<<endl;

    // function to reverse
    // the stack
    reverse(st);
    cout << "Reversed Stack" << endl;
    // printing the stack after reversal
    while (!st.empty()) {
        cout << st.top() << " ";
        st.pop();
    }
    return 0;
}
```

Sort A Stack Using Recursion:

```
void sortedInsert(struct stack** s, int x)
{
    // Base case: Either stack is empty or newly inserted
    // item is greater than top (more than all existing)
    if (isEmpty(*s) or x > top(*s)) {
        push(s, x);
        return;
    }

    // If top is greater, remove the top item and recur
    int temp = pop(s);
    sortedInsert(s, x);

    // Put back the top item removed earlier
    push(s, temp);
}

// Function to sort stack
void sortStack(struct stack** s)
{
    // If stack is not empty
    if (!isEmpty(*s)) {
        // Remove the top item
        int x = pop(s);

        // Sort remaining stack
        sortStack(s);

        // Push the top item back in sorted stack
        sortedInsert(s, x);
    }
}
```

Arrays

Minimum swaps to sort an array

By Swapping Elements to Correct Positions – O(nlogn) Time and O(n) Space

The idea is to use a hash map to store each element of the given array along with its index. We also create a temporary array that stores all the elements of the input array in sorted order. As we traverse the input array, if the current element $\text{arr}[i]$ is not in its correct position, we swap it with the element that should be at i i.e., $\text{temp}[i]$. After this, we increment the swap count and update the indices in the hash map accordingly.

```
int minSwaps(vector<int> &arr) {

    // Temporary array to store elements in sorted order
    vector<int> temp(arr.begin(), arr.end());
    sort(temp.begin(), temp.end());

    // Hashing elements with their correct positions
    unordered_map<int, int> pos;
    for(int i = 0; i < arr.size(); i++)
        pos[arr[i]] = i;

    int swaps = 0;
    for(int i = 0; i < arr.size(); i++) {
        if(temp[i] != arr[i]) {

            // Index of the element that should be at index i.
            int ind = pos[temp[i]];
            swap(arr[i], arr[ind]);

            // Update the indices in the hashmap
            pos[arr[i]] = i;
            pos[arr[ind]] = ind;

            swaps++;
        }
    }
    return swaps;
}
```

Using Cycle Detection – O(nlogn) Time and O(n) Space

This approach uses cycle detection method to find out the minimum number of swaps required to sort the array. If an element is not in its correct position, it indicates that it is a part of a cycle with one or more other elements that also need to be moved. For example, if element A is in the position of element B, and element B is in the position of element C, and so on, until it comes back to A, it forms a cycle. And to sort the elements in the cycle, we need $\text{cycleSize} - 1$ swaps, as each swap places one element in its correct position, and the last element will automatically be in its correct place.

```
int minSwaps(vector<int> &arr) {
    int n = arr.size();

    // Array to Keep track of those elements
    // who already been included in the cycle
    bool vis[n] = {0};

    // Hashing elements with their original positions
    unordered_map<int, int> pos;
    for (int i = 0; i < n; i++)
        pos[arr[i]] = i;

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

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

        // Already a part of another cycle Or
        // in its correct position
        if (vis[i] || pos[arr[i]] == i)
            continue;

        int j = i, cycleSize = 0;

        // We make a cycle until it comes
        // back to first element again.
        while (!vis[j]) {
            vis[j] = true;

            // move to next element of the cycle
            j = pos[arr[j]];
            cycleSize++;
        }
    }
}
```

```

        // Update answer by adding current cycle.
        if (cycleSize > 0) {
            swaps += (cycleSize - 1);
        }
    }
    return swaps;
}

```

Sort an array of 0s, 1s and 2s

```

void sort012(int a[], int n)
{
    // code here
    int l=0,m=0,h=n-1;
    while(m<=h){
        if(a[m] == 0){
            swap(a[l],a[m]);
            l++,m++;
        }
        else if(a[m] == 1){
            m++;
        }
        else if(a[m] == 2){
            swap(a[m],a[h]);
            h--;
        }
    }
}

```

Move all negative elements to end without disturbing order:

```

void segregateElements(int arr[],int n)
{
    // Your code goes here
    vector<int> v;
    int l=0;
    for(int i=0 ;i<n;i++){
        if(arr[i] < 0){

```

```

        v.push_back(arr[i]);

    }

    else{
        swap(arr[i],arr[l]);
        l++;
    }
}
for(int x=0;x<v.size();x++){
    arr[l++] = v[x];
}
}

```

Kadane's Algorithm (Longest Contiguous Subarray)

```

long long maxSubarraySum(int arr[], int n){

    // Your code here
    long long max_so_far = 0, curr_max = 0;
    for(int i=0;i<n;i++){
        if(curr_max+ arr[i]>0){
            curr_max+=arr[i];
        }else{
            curr_max=0;
        }
        max_so_far = max(max_so_far, curr_max);

    }
    if(max_so_far == 0){
        max_so_far = arr[0];
        for(int i=0;i<n;i++){
            if(max_so_far < arr[i])
                max_so_far = arr[i];
        }
    }
    return max_so_far;
}

```

Minimize the Heights: add or subtract k to every element

```
class Solution {
public:
    int getMinDiff(int arr[], int n, int k) {
        // code here
        sort(arr,arr+n);
        int ans = arr[n-1]-arr[0];
        int mi,mx;
        for(int i=1;i<n;i++){
            mx = max(arr[n-1]-k,arr[i-1]+k);
            mi = min(arr[0]+k,arr[i]-k);
            if(mi<0)continue;
            ans = min(ans,mx-mi);
        }
        return ans;
    }
};
```

Min Jumps to Read End

```
class Solution:
    def jump(self, nums: List[int]) -> int:
        n = len(nums)
        ans = 0
        i = 0
        if n == 1:
            return 0
        while i < n-1:
            curr = i + nums[i]
            if curr >= n-1:
                return ans + 1
            if not nums[i]:
                return -1
            nxt = 0
            mx = 0
            for j in range(i+1, min(curr+1, n), 1):
                if j + nums[j] > mx:
                    mx = j + nums[j]
```

```
        nxt = j
    i = nxt
    ans += 1
return ans
```

```
// C++ program for Minimum number
// of jumps to reach end
#include <bits/stdc++.h>
using namespace std;

int min(int x, int y) { return (x < y) ? x : y; }

// Returns minimum number of jumps
// to reach arr[n-1] from arr[0]
int minJumps(int arr[], int n)
{
    // jumps[n-1] will hold the result
    int* jumps = new int[n];
    int i, j;

    if (n == 0 || arr[0] == 0)
        return INT_MAX;

    jumps[0] = 0;

    // Find the minimum number of jumps to reach arr[i]
    // from arr[0], and assign this value to jumps[i]
    for (i = 1; i < n; i++) {
        jumps[i] = INT_MAX;
        for (j = 0; j < i; j++) {
            if (i <= j + arr[j] && jumps[j] != INT_MAX) {
                jumps[i] = min(jumps[i], jumps[j] + 1);
                break;
            }
        }
    }
    return jumps[n - 1];
}
```

```

// Driver code
int main()
{
    int arr[] = { 1, 3, 5, 8, 9, 2, 6, 7, 6, 8, 9 };
    int size = sizeof(arr) / sizeof(int);
    cout << "Minimum number of jumps to reach end is "
         << minJumps(arr, size);
    return 0;
}

// This code is contributed by rathbhupendra

```

Find the duplicate number:

Given an array of integers `nums` containing $n + 1$ integers where each integer is in the range $[1, n]$ inclusive.

There is only one repeated number in `nums`, return this repeated number.

You must solve the problem without modifying the array `nums` and uses only constant extra space.

Example 1:

Input: `nums` = [1,3,4,2,2]

Output: 2

Example 2:

Input: `nums` = [3,1,3,4,2]

Output: 3

Example 3:

Input: `nums` = [3,3,3,3,3]

Output: 3

```

class Solution {
public:
    int findDuplicate(std::vector<int>& nums) {
        int left = 1;

```

```

int right = nums.size() - 1;

while (left < right) {
    int mid = left + (right - left) / 2;
    int count = 0;

    // Count the numbers less than or equal to mid
    for (int num : nums) {
        if (num <= mid) {
            count++;
        }
    }

    // If count is greater than mid, the duplicate lies in the left half
    if (count > mid) {
        right = mid;
    } else { // Otherwise, it lies in the right half
        left = mid + 1;
    }
}

return left;
}
};

```

Merge Without using extra space:

```

class Solution{
public:
    //Function to merge the arrays.
    void merge(long long arr1[], long long arr2[], int n, int m)
    {
        // code here
        int i=n-1,j=0;
        while(i>=0 and j<m){
            if(arr1[i]>arr2[j]){
                swap(arr1[i],arr2[j]);
                i--;
                j++;
            }
        }
    }
};

```

```

        }
        else break;
    }
    sort(arr1,arr1+n);
    sort(arr2,arr2+m);
}
};
```

Merge Intervals:

```

class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>> & intervals) {
        sort(intervals.begin(), intervals.end());

        int n = intervals.size();

        vector<vector<int>> ans;
        vector<int> curr_interval;
        ans.push_back(intervals[0]);
        for(int i=1;i<n;i++){
            //case 1 overlapping case enc of prev > begin of curr
            if(ans.back()[1]>=intervals[i][0]){
                ans.back()[1]=max(ans.back()[1], intervals[i][1]);
            }
            else{
                ans.push_back(intervals[i]);
            }
        }

        return ans;
    }
};
```

Best time to buy and sell stocks

```
class Solution {
```

```

public:
    int maxProfit(vector<int>& a) {
        int ans = 0;
        int mn = a[0];
        for(int i=0;i<a.size();i++){
            mn = min(mn,a[i]);
            ans = max(ans, a[i]-mn);
        }
        return ans;
    }
};

```

Next Permutation:

The steps are the following:

Find the break-point, i : Break-point means the first index i from the back of the given array where $\text{arr}[i]$ becomes smaller than $\text{arr}[i+1]$.

For example, if the given array is $\{2,1,5,4,3,0,0\}$, the break-point will be index 1(0-based indexing). Here from the back of the array, index 1 is the first index where $\text{arr}[1]$ i.e. 1 is smaller than $\text{arr}[i+1]$ i.e. 5.

To find the break-point, using a loop we will traverse the array backward and store the index i where $\text{arr}[i]$ is less than the value at index $(i+1)$ i.e. $\text{arr}[i+1]$.

If such a break-point does not exist i.e. if the array is sorted in decreasing order, the given permutation is the last one in the sorted order of all possible permutations. So, the next permutation must be the first i.e. the permutation in increasing order.

So, in this case, we will reverse the whole array and will return it as our answer.

If a break-point exists:

Find the smallest number i.e. $> \text{arr}[i]$ and in the right half of index i (i.e. from index $i+1$ to $n-1$) and swap it with $\text{arr}[i]$.

Reverse the entire right half(i.e. from index $i+1$ to $n-1$) of index i . And finally, return the array.

```

#include <bits/stdc++.h>
using namespace std;

vector<int> nextGreaterPermutation(vector<int> &A) {
    int n = A.size(); // size of the array.

    // Step 1: Find the break point:
    int ind = -1; // break point

```

```

for (int i = n - 2; i >= 0; i--) {
    if (A[i] < A[i + 1]) {
        // index i is the break point
        ind = i;
        break;
    }
}

// If break point does not exist:
if (ind == -1) {
    // reverse the whole array:
    reverse(A.begin(), A.end());
    return A;
}

// Step 2: Find the next greater element
//           and swap it with arr[ind]:

for (int i = n - 1; i > ind; i--) {
    if (A[i] > A[ind]) {
        swap(A[i], A[ind]);
        break;
    }
}

// Step 3: reverse the right half:
reverse(A.begin() + ind + 1, A.end());

return A;
}

```

Count Inversion in an array:

```

int merge(vector<int> &arr, int low, int mid, int high) {
    vector<int> temp; // temporary array
    int left = low;      // starting index of left half of
arr
    int right = mid + 1; // starting index of right half of
arr

```

```
//Modification 1: cnt variable to count the pairs:  
int cnt = 0;  
  
//storing elements in the temporary array in a sorted  
manner//  
  
while (left <= mid && right <= high) {  
    if (arr[left] <= arr[right]) {  
        temp.push_back(arr[left]);  
        left++;  
    }  
    else {  
        temp.push_back(arr[right]);  
        cnt += (mid - left + 1); //Modification 2  
        right++;  
    }  
}  
  
// if elements on the left half are still left //  
  
while (left <= mid) {  
    temp.push_back(arr[left]);  
    left++;  
}  
  
// if elements on the right half are still left //  
while (right <= high) {  
    temp.push_back(arr[right]);  
    right++;  
}  
  
// transferring all elements from temporary to arr //  
for (int i = low; i <= high; i++) {  
    arr[i] = temp[i - low];
```

```

    }

    return cnt; // Modification 3
}

int mergeSort(vector<int> &arr, int low, int high) {
    int cnt = 0;
    if (low >= high) return cnt;
    int mid = (low + high) / 2 ;
    cnt += mergeSort(arr, low, mid); // left half
    cnt += mergeSort(arr, mid + 1, high); // right half
    cnt += merge(arr, low, mid, high); // merging sorted
halves
    return cnt;
}

int numberofInversions(vector<int>&a, int n) {

    // Count the number of pairs:
    return mergeSort(a, 0, n - 1);
}

```

Count All Pairs with given sum:(2sum Problem)

Given an array of N integers, and an integer K, find the number of pairs of elements in the array whose sum is equal to K.

Example 1:

Input:

N = 4, K = 6

arr[] = {1, 5, 7, 1}

Output: 2

Explanation:

$\text{arr}[0] + \text{arr}[1] = 1 + 5 = 6$
and $\text{arr}[1] + \text{arr}[3] = 5 + 1 = 6$.

```
class Solution{
public:
    int getPairsCount(int arr[], int n, int k) {
        // code here
        unordered_map<int,int> m;
        int ans =0;
        for(int i=0;i<n;i++){
            ans += m[k-arr[i]];
            m[arr[i]]++;
        }
        return ans;
    }
};
```

Common elements in 3 sorted array:

```
class Solution
{
public:
    vector <int> commonElements (int A[], int B[], int C[], int n1, int
n2, int n3)
    {
        //code here.
        vector<int> ans;
        int i,j,k;
        i=j=k=0;
        while(i<n1 and j<n2 and k<n3){
            if(A[i]==B[j] and B[j]==C[k]){
                if(!ans.empty() and ans.back() == A[i]){
                    i++,j++,k++;
                    continue;
                }
                ans.push_back(A[i]);
                i++,j++,k++;
            }
        }
    }
};
```

```

        else if(A[i]<B[j]) i++;
        else if(B[j]<C[k]) j++;
        else k++;
    }
    return ans;
}
};

```

Subarray With Sum 0 ⭐

The idea is to iterate through the array and for every element $arr[i]$, calculate the sum of elements from 0 to i (this can simply be done as $sum += arr[i]$). If the current sum has been seen before, then there must be a zero-sum subarray. Hashing is used to store the sum values so that sum can be stored quickly and find out whether the current sum is seen before or not.

```

class Solution{
public:
//Complete this function
//Function to check whether there is a subarray present with 0-sum or
not.
bool subArrayExists(int arr[], int n)
{
    //Your code here
    unordered_map<int,int> m;
    int sum=0;
    for(int i=0;i<n;i++){
        sum += arr[i];
        if(arr[i]==0 or sum==0) return true;
        if(m[sum]) return true;
        m[sum]++;
    }
    return false;
}
};

```

Maximum product subarray:

```

int maxSubarrayProduct(int arr[], int n)
{
    // max positive product
    // ending at the current position
    int max_ending_here = arr[0];

    // min negative product ending
    // at the current position
    int min_ending_here = arr[0];

    // Initialize overall max product
    int max_so_far = arr[0];
    /* Traverse through the array.
    the maximum product subarray ending at an index
    will be the maximum of the element itself,
    the product of element and max product ending previously
    and the min product ending previously. */
    for (int i = 1; i < n; i++) {
        int temp = max({ arr[i], arr[i] * max_ending_here,
                         arr[i] * min_ending_here });

        min_ending_here
            = min({ arr[i], arr[i] * max_ending_here,
                     arr[i] * min_ending_here });
        max_ending_here = temp;
        max_so_far = max(max_so_far, max_ending_here);
    }
    return max_so_far;
}

```

Longest Consecutive Subsequence:

Given an array of positive integers. Find the length of the longest sub-sequence such that elements in the subsequence are consecutive integers, the consecutive numbers can be in any order.

Example 1:

Input:

N = 7

a[] = {2,6,1,9,4,5,3}

Output:

6

Explanation:

The consecutive numbers here
are 1, 2, 3, 4, 5, 6. These 6
numbers form the longest consecutive
subsequence.

```
class Solution{
public:
    // arr[] : the input array
    // N : size of the array arr[]

    //Function to return length of longest subsequence of consecutive
    integers.
    int findLongestConseqSubseq(int arr[], int n)
    {
        //Your code here
        sort(arr,arr+n);
        int ans =1,j=0,g=1;
        for(int i=0;i<n-1;i++){
            if(arr[i]+1==arr[i+1]){

                g++;
            }
            else if(arr[i]==arr[i+1]){
                continue;
            }
            else{
                ans = max(ans,g);
                g=1;
            }
        }
        ans = max(ans,g);
        return ans;
    }
};
```

Buy ans Sell stocks at max two transactions:

Better Approach – O(n) Time and O(n) Space:

The idea is to store the maximum possible profit for every index.

- 1) Create a table profit[0..n-1] and initialize all values in it 0.
- 2) Traverse price[] from right to left and update profit[i] such that profit[i] stores maximum profit achievable from one transaction in subarray price[i..n-1]
- 3) Traverse price[] from left to right starting from second element, keep track of the minimum price, and for index i, find the maximum profit achievable from 0 to i using the idea of the single transaction algorithm that is used in step 1. Now we have profit of one transaction from 0 to i in current loop, and from i to n-1 in profit in profit array, using these two we find the current profit with at most two transactions. And also update result if the current value is more.

price = [10, 30, 5, 50, 60]

We compute profit from right to left, starting from the second last

profit = [55, 55, 55, 10, 0]

Now we traverse price from left, starting from second element, we compute current profit using profit[i] and minimum price so far. We also update result if we get more value.

Below are the implementations of the above approach.

```
int maxProfit(vector<int>& price) {  
    int n = price.size();  
  
    // Create profit vector and initialize it as 0  
    vector<int> profit(n, 0);  
  
    // Get the maximum profit with only one transaction  
    // allowed. After this loop, profit[i] contains  
    // maximum profit from price[i..n-1] using at most  
    // one transaction.  
    int max_price = price[n - 1];  
    for (int i = n - 2; i >= 0; i--) {  
  
        // max_price has maximum of price[i..n-1]  
        max_price = max(max_price, price[i]);  
  
        // Update profit[i]  
        profit[i] = max(profit[i + 1], max_price - price[i]);  
    }  
  
    // Variable to store the maximum profit using two transactions  
    int res = 0;
```

```

int min_price = price[0];
for (int i = 1; i < n; i++) {

    // min_price is the minimum price in price[0..i]
    min_price = min(min_price, price[i]);

    // Calculate the maximum profit by adding
    // the profit of the first transaction
    res = max(res, profit[i] + (price[i] - min_price));
}

return res;
}

```

Find whether an array is subset of another array

Method 1:

Naiv approach $n*m$ TC

Method 2:

Using Sorting and two pointers to compare arrays

Method 3:

Using hashing

```

bool isSubsetUsingHashing(const vector<int>& arr1, const vector<int>& arr2) {

    // Create a hash set and insert all elements of arr1
    unordered_set<int> hashSet(arr1.begin(), arr1.end());

    // Check each element of arr2 in the hash set
    for (int num : arr2) {
        if (hashSet.find(num) == hashSet.end()) {
            return false;
        }
    }

    // If all elements of arr2 are found in the hash set
    return true;
}

```

3 Sum – Triplet Sum in Array

Method 1: Naive N^3 TC

Efficeint Mehtod 2: Hash Set N*n

```
bool find3Numbers(vector<int>& arr, int sum)
{
    int n = arr.size();

    // Fix the first element as arr[i]
    for (int i = 0; i < n - 2; i++) {

        // Create a set to store potential second elements
        // that complement the desired sum
        unordered_set<int> s;

        // Calculate the current sum needed to reach the
        // target sum
        int curr_sum = sum - arr[i];

        // Iterate through the subarray arr[i+1..n-1] to find
        // a pair with the required sum
        for (int j = i + 1; j < n; j++) {

            // Calculate the required value for the second
            // element
            int required_value = curr_sum - arr[j];

            // Check if the required value is present in the
            // set
            if (s.find(required_value) != s.end()) {

                // Triplet is found; print the triplet
                // elements
                cout << "Triplet is " << arr[i] << ", "
                    << arr[j] << ", " << required_value;
                return true;
            }

            // Add the current element to the set for future
            // complement checks
            s.insert(arr[j]);
        }
    }

    // If no triplet is found, return false
    return false;
}
```

```
}
```

Efficeint Mehtod 3: Sorted n Two Pointer

```
bool find3Numbers(vector<int>& arr, int sum)
{
    int n = arr.size();

    // Sort the elements
    sort(arr.begin(), arr.end());

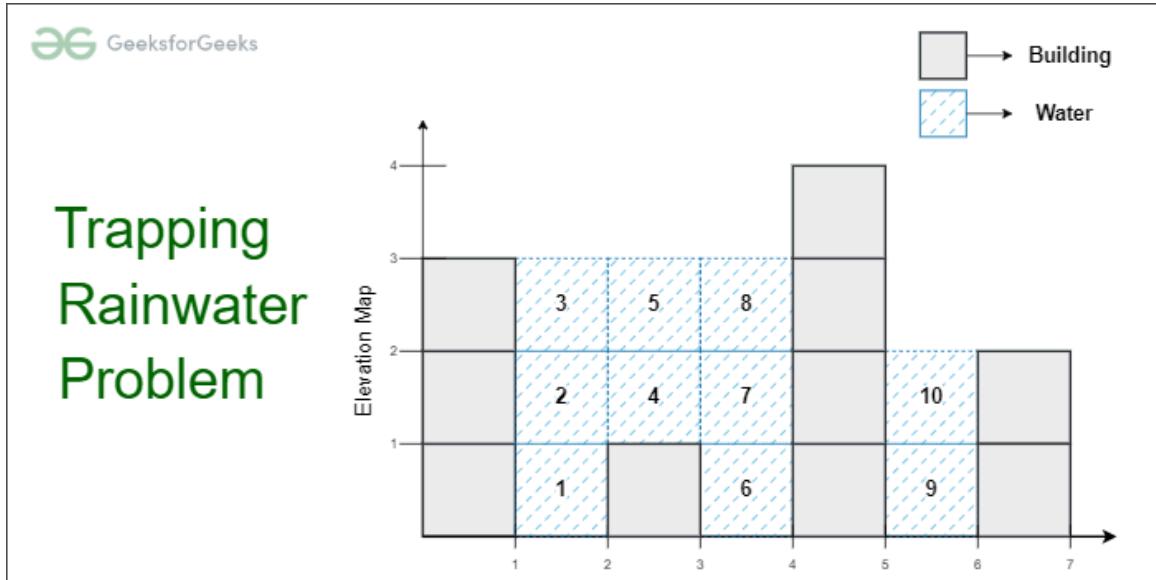
    // Fix the first element one by one
    // and find the other two elements
    for (int i = 0; i < n - 2; i++) {

        // To find the other two elements, start two index
        // variables from two corners of the array and move
        // them toward each other
        int l = i + 1; // index of the first element
                        // in the remaining elements
        int r = n - 1; // index of the last element

        while (l < r) {
            int curr_sum = arr[i] + arr[l] + arr[r];
            if (curr_sum == sum) {
                cout << "Triplet is " << arr[i] << ", "
                    << arr[l] << ", " << arr[r];
                return true;
            } else if (curr_sum < sum)
                l++;
            else // curr_sum > sum
                r--;
        }
    }

    // If we reach here, then no triplet was found
    return false;
}
```

Trapping Rain Water



Naive Algorithm:

Observations:

The basic intuition of the problem is as follows:

- An element of the array can store water if there are higher bars on the left and the right.
- The amount of water to be stored in every position can be found by finding the heights of the higher bars on the left and right sides.
- The total amount of water stored is the summation of the water stored in each index.
- No water can be filled if there is no boundary on both sides.

Brute Force – $O(n^2)$ Time and $O(1)$ Space:

Traverse every array element and find the highest bars on the left and right sides. Take the smaller of two heights. The difference between the smaller height and the height of the current element is the amount of water that can be stored in this array element.

```
int maxWater(vector<int>& arr)
{
    int res = 0;
```

```

// For every element of the array
for (int i = 1; i < arr.size() - 1; i++) {

    // Find the maximum element on its left
    int left = arr[i];
    for (int j = 0; j < i; j++)
        left = max(left, arr[j]);

    // Find the maximum element on its right
    int right = arr[i];
    for (int j = i + 1; j < arr.size(); j++)
        right = max(right, arr[j]);

    // Update the maximum water
    res += (min(left, right) - arr[i]);
}

return res;
}

```

Computing prefix and suffix max for every index – O(n) Time and O(n) Space

In the previous approach, for every element we needed to calculate the highest element on the left and on the right.

So, to reduce the time complexity:

- *For every element we first calculate and store the highest bar on the left and on the right (say stored in arrays **left[]** and **right[]**).*
- *Then iterate the array and use the calculated values to find the amount of water stored in this index,*
*which is the same as (**min(left[i], right[i]) – arr[i]**)*

```

int findWater(vector<int>& arr)
{
    int n = arr.size();

    // Left[i] contains height of tallest bar to the
    // left of i'th bar including itself
    vector<int> left(n);

```

```

// Right[i] contains height of tallest bar to
// the right of i'th bar including itself
vector<int> right(n);

// Initialize result
int res = 0;

// Fill left array
left[0] = arr[0];
for (int i = 1; i < n; i++)
    left[i] = max(left[i - 1], arr[i]);

// Fill right array
right[n - 1] = arr[n - 1];
for (int i = n - 2; i >= 0; i--)
    right[i] = max(right[i + 1], arr[i]);

// Calculate the accumulated water element by element
for (int i = 1; i < n - 1; i++) {
    int minOf2 = min(left[i - 1], right[i + 1]);
    if (minOf2 > arr[i])
        res += minOf2 - arr[i];
}
}

return res;
}

```

Best Approach – Using Two Pointers – O(n) Time and O(1) Space:

The approach is mainly based on the following facts:

1. If we consider a subarray $\text{arr}[\text{left} \dots \text{right}]$, we can decide the amount of water either for $\text{arr}[\text{left}]$ or $\text{arr}[\text{right}]$ if we know the left max (max element in $\text{arr}[0 \dots \text{left}-1]$) and right max (max element in $\text{arr}[\text{right}+1 \dots n-1]$).
2. If left max is less than the right max, then we can decide for $\text{arr}[\text{left}]$. Else we can decide for $\text{arr}[\text{right}]$
3. If we decide for $\text{arr}[\text{left}]$, then the amount of water would be left max – $\text{arr}[\text{left}]$

How does this work? Let us consider the case when left max is less than the right max. For $\text{arr}[\text{left}]$, we know left max for it and we also know that the right max for it would not be less than left max because we already have a greater value in $\text{arr}[\text{right} \dots n-1]$. Any other value from $\text{left}+1$ to $\text{right}-1$ would anyways be more than left max.

```
int maxWater(vector<int> &arr) {

    int left = 1;
    int right = arr.size() - 2;

    // lMax : Maximum in subarray arr[0..left-1]
    // rMax : Maximum in subarray arr[right+1..n-1]
    int lMax = arr[left - 1];
    int rMax = arr[right + 1];

    int res = 0;
    while (left <= right) {

        // If rMax is smaller, then we can decide the amount of water for
        arr[right]
        if (rMax <= lMax) {

            // Add the water for arr[right]
            res += max(0, rMax - arr[right]);

            // Update right max
            rMax = max(rMax, arr[right]);
        }

        // Update right pointer as we have decided the amount of water for
        this
    }
}
```

```

        right -= 1;
    } else {
        // Add the water for arr[left]
        res += max(0, lMax - arr[left]);

        // Update left max
        lMax = max(lMax, arr[left]);

        // Update left pointer as we have decided water for this
        left += 1;
    }
}
return res;
}

```

Chocolate Distribution Problem

Given an array of N integers 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 such that:

Each student gets one packet.

The difference between the number of chocolates in the packet with maximum chocolates and the packet with minimum chocolates given to the students is minimum.

Input : arr[] = {7, 3, 2, 4, 9, 12, 56} , m = 3

Output: Minimum Difference is 2

Explanation:

We have seven packets of chocolates and we need to pick three packets for 3 students

If we pick 2, 3 and 4, we get the minimum difference between maximum and minimum packet sizes.

Input : arr[] = {3, 4, 1, 9, 56, 7, 9, 12} , m = 5

Output: Minimum Difference is 6

Input : arr[] = {12, 4, 7, 9, 2, 23, 25, 41, 30, 40, 28, 42, 30, 44, 48, 43, 50} , m = 7

Output: Minimum Difference is 10

```

int findMinDiff(int arr[], int n, int m)
{
    // if there are no chocolates or number

```

```

// of students is 0
if (m == 0 || n == 0)
    return 0;

// Sort the given packets
sort(arr, arr + n);

// Number of students cannot be more than
// number of packets
if (n < m)
    return -1;

// Largest number of chocolates
int min_diff = INT_MAX;

// Find the subarray of size m such that
// difference between last (maximum in case
// of sorted) and first (minimum in case of
// sorted) elements of subarray is minimum.

for (int i = 0; i + m - 1 < n; i++) {
    int diff = arr[i + m - 1] - arr[i];
    if (diff < min_diff)
        min_diff = diff;
}
return min_diff;
}

```

Smallest subarray with sum greater than a given value

Given an array arr[] of integers and a number x, the task is to find the smallest subarray with a sum greater than the given value.

Method 1: Naive Approach: Use two nested loops

Method 2: Optimal Apporach:

```

int smallestSubWithSum(int arr[], int n, int x)
{

```

```

// Initialize current sum and minimum length
int curr_sum = 0, min_len = n + 1;

// Initialize starting and ending indexes
int start = 0, end = 0;
while (end < n) {
    // Keep adding array elements while current sum
    // is smaller than or equal to x
    while (curr_sum <= x && end < n)
        curr_sum += arr[end++];

    // If current sum becomes greater than x.
    while (curr_sum > x && start < n) {
        // Update minimum length if needed
        if (end - start < min_len)
            min_len = end - start;

        // remove starting elements
        curr_sum -= arr[start++];
    }
}
return min_len;
}

```

Three way partitioning

Given an array of size n and a range [a, b]. The task is to partition the array around the range such that the array is divided into three parts.

- 1) All elements smaller than a come first.
- 2) All elements in range a to b come next.
- 3) All elements greater than b appear in the end.

The individual elements of three sets can appear in any order. You are required to return the modified array.

USE SORTING OF Array 0s 1s and 2s

```

void threeWayPartition(int arr[], int n, int lowVal,
                      int highVal)
{
    // Initialize next available positions for
    // smaller (than range) and greater elements

```

```

int start = 0, end = n - 1;

// Traverse elements from left
for (int i = 0; i <= end;) {
    // If current element is smaller than
    // range, put it on next available smaller
    // position.
    if (arr[i] < lowVal) {
        // if i and start are same in that case we can't
        // swap swap only if i is greater than start
        if (i == start) {
            start++;
            i++;
        }
        else
            swap(arr[i++], arr[start++]);
    }

    // If current element is greater than
    // range, put it on next available greater
    // position.
    else if (arr[i] > highVal)
        swap(arr[i], arr[end--]);

    else
        i++;
}
}

```

Minimum swaps required to bring all elements less than or equal to k together

Given an array of n positive integers and a number k. Find the minimum number of swaps required to bring all the numbers less than or equal to k together.

```

int minSwap(int arr[], int n, int k)
{
    // Find the number of elements <= k
    int good = 0;

```

```

for (int i = 0; i < n; i++) {
    if (arr[i] <= k)
        good += 1;
}

// Initialize min swaps with good as the max possible
// value
int bad = 0, minSwaps = good;
for (int i = 0; i < n; i++) {
    // If the current element > k, then increment the
    // count of bad elements in the current sliding
    // window
    if (arr[i] > k) {
        bad += 1;
    }
    // If we complete the first sliding window,
    // calculate min swaps
    if (i == good - 1) {
        minSwaps = min(minSwaps, bad);
    }
    else if (i >= good) {
        // Exclude the elements from the start of the
        // sliding window to maintain its size as 'good'
        if (arr[i - good] > k)
            bad -= 1;
        // For every sliding window of size 'good', find
        // the minimum swaps required
        minSwaps = min(minSwaps, bad);
    }
}
// Return the minimum swaps
return minSwaps;
}

```

Find minimum number of merge operations to make an array palindrome

```

int findMinOps(int arr[], int n)
{
    int ans = 0; // Initialize result

    // Start from two corners

```

```

for (int i=0, j=n-1; i<=j;)
{
    // If corner elements are same,
    // problem reduces arr[i+1..j-1]
    if (arr[i] == arr[j])
    {
        i++;
        j--;
    }

    // If left element is greater, then
    // we merge right two elements
    else if (arr[i] > arr[j])
    {
        // need to merge from tail.
        j--;
        arr[j] += arr[j+1];
        ans++;
    }

    // Else we merge left two elements
    else
    {
        i++;
        arr[i] += arr[i-1];
        ans++;
    }
}

return ans;
}

```

Strings:

Check if given strings are rotations of each other or not

```

bool areRotations(string str1, string str2)
{
    /* Check if sizes of two strings are same */

```

```

if (str1.length() != str2.length())
    return false;

string temp = str1 + str1;
return (temp.find(str2) != string::npos);
}

```

Count and Say:

```

class Solution {
public:
    string solve(string s){
        int c=1,i=0;
        string ans = "", t = "";
        for(i=0;i<s.size()-1;i++){
            if(s[i]==s[i+1]){
                c++;
            }
            else{
                ans += to_string(c) + s[i];
                c=1;
            }
        }
        if(c){
            ans += to_string(c) + s[i];
        }
        cout<<ans<<endl;
        return ans;
    }
    string countAndSay(int n) {
        if(n==1) return "1";
        string ans = "1";
        for(int i=1;i<n;i++) ans = solve(ans);
        return ans;
    }
};

```

Longest Palindrome String:

```
Class Solution {
public:
    int lenOfPalin(string &s, int a, int b) {
        int cnt=0;
        for(int i=a, j=b; j<s.size() && i>=0; --i, ++j)
        {
            if(s[i]!=s[j])
                return cnt;
            ++cnt;
        }
        return cnt;
    }

    string longestPalindrome(string s) {
        int ans = 0, n=s.size(), idx = -1, len = 0;
        for(int i=0; i<n; ++i) {
            len = 2*lenOfPalin(s, i, i+1); //Even case
            if(ans < len) {
                ans = len;
                idx = i-len/2+1;
            }
            len = 2*lenOfPalin(s, i-1, i+1)+1; //Odd case
            if(ans < len) {
                ans = len;
                idx = i-len/2;
            }
        }
        return string(s.begin()+idx, s.begin()+idx+ans);
    }
};
```

simpler version

```
class Solution {
public:
    string longestPalindrome(string s) {
        int resLen = 0, resIdx = 0;

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

```

        // odd length
        int l = i, r = i;
        while (l >= 0 && r < s.size() &&
               s[l] == s[r]) {
            if (r - l + 1 > resLen) {
                resIdx = l;
                resLen = r - l + 1;
            }
            l--;
            r++;
        }

        // even length
        l = i;
        r = i + 1;
        while (l >= 0 && r < s.size() &&
               s[l] == s[r]) {
            if (r - l + 1 > resLen) {
                resIdx = l;
                resLen = r - l + 1;
            }
            l--;
            r++;
        }
    }

    return s.substr(resIdx, resLen);
}
};


```

Rearrange Positive and Negative numbers alternatively:

```

void rearrange(vector<int>& arr) {
    vector<int> pos, neg;

    // Separate positive and negative numbers
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] >= 0)
            pos.push_back(arr[i]);
        else
            neg.push_back(arr[i]);
    }
}


```

```

}

int posIdx = 0, negIdx = 0;
int i = 0;

// Place positive and negative numbers alternately
// in the original array
while (posIdx < pos.size() && negIdx < neg.size()) {
    if (i % 2 == 0)
        arr[i++] = pos[posIdx++];
    else
        arr[i++] = neg[negIdx++];
}

// Append remaining positive numbers (if any)
while (posIdx < pos.size())
    arr[i++] = pos[posIdx++];

// Append remaining negative numbers (if any)
while (negIdx < neg.size())
    arr[i++] = neg[negIdx++];
}

```

Find the factorial of large number:

```

class Solution {
public:
    vector<int> factorial(int N){
        // code here
        vector<int>ans(1,1);
        while(N>1){
            int val=0;
            for(int i=0;i<ans.size();i++){
                val+=ans[i]*N;
                ans[i]=val%10;
                val/=10;
            }
            while(val){
                ans.push_back(val%10);
                val/=10;
            }
        }
    }
}

```

```

        N--;
    }
    reverse(ans.begin(),ans.end());
    return ans;
}
};

```

Find all elements that appear more than n/k time in array of size n , and k is number

```

// A C++ program to print elements with count more than n/k
#include <iostream>
using namespace std;

// A structure to store an element and its current count
struct eleCount {
    int e; // Element
    int c; // Count
};

// Prints elements with more
// than n/k occurrences in arr[]
// of size n. If there are no
// such elements, then it prints
// nothing.
void moreThanNdk(int arr[], int n, int k)
{
    // k must be greater than
    // 1 to get some output
    if (k < 2)
        return;

    /* Step 1: Create a temporary
    array (contains element
    and count) of size k-1.
    Initialize count of all
    elements as 0 */
    struct eleCount temp[k - 1];
    for (int i = 0; i < k - 1; i++)

```

```

temp[i].c = 0;

/* Step 2: Process all
elements of input array */
for (int i = 0; i < n; i++) {
    int j;

    /* If arr[i] is already present in
the element count array,
then increment its count
*/
    for (j = 0; j < k - 1; j++) {
        if (temp[j].e == arr[i]) {
            temp[j].c += 1;
            break;
        }
    }

    /* If arr[i] is not present in temp[] */
    if (j == k - 1) {
        int l;

        /* If there is position available
in temp[], then place arr[i] in
the first available position and
set count as 1*/
        for (l = 0; l < k - 1; l++) {
            if (temp[l].c == 0) {
                temp[l].e = arr[i];
                temp[l].c = 1;
                break;
            }
        }
    }

    /* If all the position in the
temp[] are filled, then decrease
count of every element by 1 */
    if (l == k - 1)
        for (l = 0; l < k - 1; l++)
            temp[l].c -= 1;
    }
}

```

```

/*Step 3: Check actual counts of
* potential candidates in temp[]*/
for (int i = 0; i < k - 1; i++) {
    // Calculate actual count of elements
    int ac = 0; // actual count
    for (int j = 0; j < n; j++)
        if (arr[j] == temp[i].e)
            ac++;

    // If actual count is more than n/k,
    // then print it
    if (ac > n / k)
        cout << "Number:" << temp[i].e
            << " Count:" << ac << endl;
}
}

/* Driver code */
int main()
{
    int arr1[] = { 4, 5, 6, 7, 8, 4, 4 };
    int size = sizeof(arr1) / sizeof(arr1[0]);
    int k = 3;
    moreThanNdK(arr1, size, k);

    return 0;
}

```

Buy And Sell Stocks (unlimited number of transaction)

```

class Solution {
public:
    int maxProfit(vector<int>& prc) {
        int n = prc.size();
        //vector<int>pro(n,0);
        int mt = prc[0];
        int ans = 0;
        int p = 0;
        for(int i=0;i <n;i++){

```

```

        mt = min(mt, prc[i]);
        //cout<< "p = "<<p<<" prci: "<<prc[i]<<" and mt: "<<mt<<endl;
        if(p>=prc[i]-mt){ // p:4 > p:2
            ans+=p;
            p=0;
            mt = prc[i];
        }
        if(prc[i]>=mt){
            p = prc[i]-mt;
        }else{
            ans+=p;
            p=0;
            mt=prc[i];
        }
    }
    if(p!=0) ans += p;
    return ans;
}
};


```

Write a Program to check whether a string is a valid shuffle of two strings or not

Given two strings **str1** and **str2**, and a third-string **shuffle**, determine if **shuffle** is a valid shuffle of **str1** and **str2**, where a valid shuffle contains all characters from **str1** and **str2** occurring the same number of times, regardless of order. Print “**YES**” if valid, and “**NO**” otherwise.

Solve using `unordered_map`

```

bool validShuffle(string str1, string str2, string shuffle)
{
    // n1 = size of str1, n2 = size of str2
    int n1 = str1.size();
    int n2 = str2.size();

    // n = size of string shuffle
    int n = shuffle.size();

```

```
// Its obvious if the no. of char in
// shuffle are more or less than the
// length of str1 and str2 then it
// won't be a valid shuffle
if (n != n1 + n2)
    return false;

// We use an unordered map to keep
// track of frequency of
// each character.
unordered_map<int, int> freq;

// Count frequency of each char
// in str1
for (int i = 0; i < n1; i++)
    freq[str1[i]]++;

// Count frequency of each char
// in str2
for (int i = 0; i < n2; i++)
    freq[str2[i]]++;

// If any of the char is not found in
// the map, then its not a
// valid shuffle.
for (int i = 0; i < n; i++) {
    if (freq.find(shuffle[i]) != freq.end())
        freq[shuffle[i]]--;
    else
        return false;
}

// Checks whether all the elements's frequency in
// hashmap becomes 0
for (auto it : freq) {
    if (it.second != 0) {
        return false;
    }
}
return true;
}
```

Find Longest Common Subsequence in String

```
// C++ program to find the longest repeating
// subsequence
#include <iostream>
#include <string>
using namespace std;

// This function mainly returns LCS(str, str)
// with a condition that same characters at
// same index are not considered.
int findLongestRepeatingSubSeq(string str)
{
    int n = str.length();

    // Create and initialize DP table
    int dp[n+1][n+1];
    for (int i=0; i<=n; i++)
        for (int j=0; j<=n; j++)
            dp[i][j] = 0;

    // Fill dp table (similar to LCS loops)
    for (int i=1; i<=n; i++)
    {
        for (int j=1; j<=n; j++)
        {
            // If characters match and indexes are
            // not same
            if (str[i-1] == str[j-1] && i != j)
                dp[i][j] = 1 + dp[i-1][j-1];

            // If characters do not match
            else
                dp[i][j] = max(dp[i][j-1], dp[i-1][j]);
        }
    }
    return dp[n][n];
}

// Driver Program
int main()
{
    string str = "aabb";
```

```

        cout << "The length of the largest subsequence that"
            " repeats itself is : "
            << findLongestRepeatingSubSeq(str);
    return 0;
}

```

Print all subsequences of a string

Pick and Don't pick method

```

#include <bits/stdc++.h>
using namespace std;

// Find all subsequences recursively
void printSubRec(string s, string curr)
{
    // Base Case : s is empty, print
    // current subsequence
    if (s.empty()) {
        cout << curr << endl;
        return;
    }

    // curr is passed with including
    // the first character of the string
    printSubRec(s.substr(1), curr + s[0]);

    // curr is passed without including
    // the first character of the string
    printSubRec(s.substr(1), curr);
}

// Wrapper method for printSubRec
void printSubs(string s)
{
    string curr = "";
    printSubRec(s, curr);
}

```

```

// Driver code
int main()
{
    string s = "ab";
    printSubs(s);
    return 0;
}

```

Incremental approach

```

#include <bits/stdc++.h>
using namespace std;

// s : Stores input string
// n : Length of s.
// curr : Stores current permutation
// index : Index in current permutation, curr
void printSubSeqRec(string s, int n, int index = -1,
                     string curr = "") {
    // base case
    if (index == n)
        return;

    // Print the current subsequence (including empty)
    cout << curr << "\n";

    for (int i = index + 1; i < n; i++) {
        curr += s[i];
        printSubSeqRec(s, n, i, curr);

        // backtracking
        curr = curr.erase(curr.size() - 1);
    }
    return;
}

// Generates power set in lexicographic order.
void printSubSeq(string s) {
    printSubSeqRec(s, s.size());
}

// Driver code

```

```
int main() {
    string s = "ab";
    printSubSeq(s);
    return 0;
}
```

Bit Manipulation Approach:

```
#include <bits/stdc++.h>
using namespace std;

// Function to print all the power set
void printPowerSet(string &s) {
    int n = pow(2, s.size());

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

            // Check if jth bit in the counter is set
            if (counter & (1 << j))
                cout << s[j];
        }
        cout << endl;
    }
}

/* Driver code */
int main() {
    string s = "ab";
    printPowerSet(s);
    return 0;
}
```

Print all permutations of String / Array

Follow the given steps to solve the problem:

- Create a function **permute()** with parameters as input string and starting index of the string
- Call this function with values input string and starting index, **idx** as 0
 - In this function, if idx becomes size of the string then print the same string
 - Else run a for loop from idx to size – 1 and swap the current element in the for loop with the s[idx]
 - Then again call this same function by increasing the value of idx by 1
 - After that again swap the previously swapped values to initiate backtracking

```
#include <bits/stdc++.h>
using namespace std;

// Function to print permutations of the string
// This function takes two parameters:
// 1. String
// 2. Starting index of the string.
void permuteRec(string& s, int idx)
{
    // Base case
    if (idx == s.size() - 1) {
        cout << s << endl;
        return;
    }

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

        // Swapping
        swap(s[idx], s[i]);

        // First idx+1 characters fixed
        permuteRec(s, idx + 1);

        // Backtrack
        swap(s[idx], s[i]);
    }
}

// Wrapper function
void permute(string& s) {
```

```

        permuteRec(s, 0);
    }

int main(){
    string s = "ABC";
    permute(s);
    return 0;
}

```

Split the binary string into substrings with equal number of 0s and 1s

Given a binary string str of length N, the task is to find the maximum count of consecutive substrings str can be divided into such that all the substrings are balanced i.e. they have equal number of 0s and 1s. If it is not possible to split str satisfying the conditions then print -1.

Example:

Input: str = “0100110101”

Output: 4

The required substrings are “01”, “0011”, “01” and “01”.

Input: str = “0111100010”

Output: 3

Input: str = “0000000000”

Output: -1

```

int maxSubStr(string str, int n)
{
    // To store the count of 0s and 1s
    int count0 = 0, count1 = 0;

    // To store the count of maximum
    // substrings str can be divided into
    int cnt = 0;
    for (int i = 0; i < n; i++) {
        if (str[i] == '0') {
            count0++;

```

```

    }
    else {
        count1++;
    }
    if (count0 == count1) {
        cnt++;
    }
}

// It is not possible to
// split the string
if (count0!=count1) {
    return -1;
}

return cnt;
}

```

Word Wrap Problem:

Given an array `nums[]` of size `n`, where `nums[i]` denotes the number of characters in one word. Let `K` be the limit on the number of characters that can be put in one line (line width). Put line breaks in the given sequence such that the lines are printed neatly.

Assume that the length of each word is smaller than the line width. When line breaks are inserted there is a possibility that extra spaces are present in each line. The extra spaces include spaces put at the end of every line except the last one.

You have to minimize the following total cost where total cost = Sum of cost of all lines, where cost of line is = (Number of extra spaces in the line)².

Example 1:

Input: `nums = {3,2,2,5}`, `k = 6`

Output: 10

Explanation: Given a line can have 6 characters,

Line number 1: From word no. 1 to 1

Line number 2: From word no. 2 to 3

Line number 3: From word no. 4 to 4

So total cost = $(6-3)^2 + (6-2-2-1)^2 = 3^2 + 1^2 = 10$.

As in the first line word length = 3 thus

extra spaces = $6 - 3 = 3$ and in the second line there are two word of length 2 and there already 1 space between two word thus extra spaces $= 6 - 2 - 2 - 1 = 1$. As mentioned in the problem description there will be no extra spaces in the last line. Placing first and second word in first line and third word on second line would take a cost of $02 + 42 = 16$ (zero spaces on first line and $6-2 = 4$ spaces on second), which isn't the minimum possible cost.

Word Wrap Problem with Memoization:

The problem can be solved using a divide and conquer (recursive) approach. The algorithm for the same is mentioned below:

- We recur for each word starting with first word, and remaining length of the line (initially k).
- The last word would be the base case: We check if we can put it on same line:
 - if yes, then we return cost as 0.
 - if no, we return cost of current line based on its remaining length.
- For non-last words, we have to check if it can fit in the current line:
 - If yes, then we have two choices i.e. whether to put it in same line or next line.
 - if we put it on next line: **cost1 = square(remLength) + cost of putting word on next line.**
 - if we put it on same line: **cost2 = cost of putting word on same line.**
 - return **min(cost1, cost2)**
 - If no, then we have to put it on next line and return cost of putting word on next line
- Use memoization table of size n (number of words) * k (line length), to keep track of already visited positions.

Below is the implementation of above approach:

```
#include <bits/stdc++.h>
using namespace std;

int solveWordWrapUsingMemo(int words[], int n, int length,
                           int wordIndex, int remLength,
                           vector<vector<int>> memo);

int square(int n) { return n * n; }

int solveWordWrapUtil(int words[], int n, int length,
                      int wordIndex, int remLength,
                      vector<vector<int>> memo)
{
    // base case for last word
```

```

        if (wordIndex == n - 1) {
            memo[wordIndex][remLength]
                = words[wordIndex] < remLength
                    ? 0
                    : square(remLength);
            return memo[wordIndex][remLength];
        }

        int currWord = words[wordIndex];
        // if word can fit in the remaining line
        if (currWord < remLength) {
            return min(solveWordWrapUsingMemo(
                words, n, length, wordIndex + 1,
                remLength == length
                    ? remLength - currWord
                    : remLength - currWord - 1,
                memo),

                square(remLength)
                + solveWordWrapUsingMemo(
                    words, n, length, wordIndex + 1,
                    length - currWord, memo));
        }
        else {
            // if word is kept on next line
            return square(remLength)
                + solveWordWrapUsingMemo(
                    words, n, length, wordIndex + 1,
                    length - currWord, memo);
        }
    }

    int solveWordWrapUsingMemo(int words[], int n, int length,
                                int wordIndex, int remLength,
                                vector<vector<int> > memo)
    {
        if (memo[wordIndex][remLength] != -1) {
            return memo[wordIndex][remLength];
        }

        memo[wordIndex][remLength] = solveWordWrapUtil(
            words, n, length, wordIndex, remLength, memo);
        return memo[wordIndex][remLength];
    }
}

```

```

}

int solveWordWrap(int words[], int n, int k)
{
    vector<vector<int>> memo(n, vector<int>(k + 1, -1));

    return solveWordWrapUsingMemo(words, n, k, 0, k, memo);
}
int main()
{
    int words[] = { 3, 2, 2, 5 };
    int n = sizeof(words) / sizeof(words[0]);
    int k = 6;

    cout << solveWordWrap(words, n, k);
    return 0;
}
/* This Code is contributed by Tapesh (tapeshdua420) */

```

Parenthesis Problem

```

class Solution {
public:

bool isParenthesisBalanced(string& s) {
    // code here
    stack<char> stk;
    for(int i=0;i<s.size();i++){
        if(s[i] == '{' or s[i]=='(' or s[i]== '['){
            stk.push(s[i]);
        }else if(s[i]=='}'){
            if (stk.size()==0) return false;
            if(stk.top() == '{'){
                stk.pop();
            }else
                return false;
        }else if(s[i]==')'){

```

```

        if (stk.size()==0) return false;
        if(stk.top() == '('){
            stk.pop();
        }else
            return false;
    }else if(s[i]==')'){
        if (stk.size()==0) return false;
        if(stk.top() == '['){
            stk.pop();
        }else
            return false;
    }
}
if (stk.size()==0) return true;
return false;
}
};

```

Word Break Problem:

Recursion

```

class Solution
{
public:
    bool check(string s, map<string,bool> mp){
        //base case
        if(s.empty()) return true;

        int n = s.size();

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

            string prefix = s.substr(0,i);

            if(mp[prefix]==1 && check(s.substr(i), mp) == true)
                return true;

        }

        return false;
    }
};

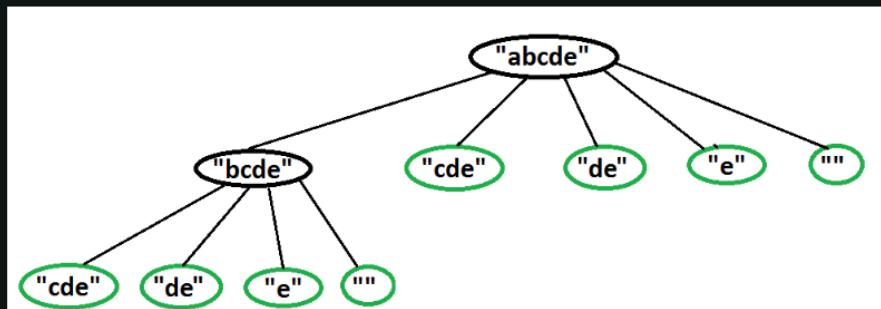
```

```

    }
    int wordBreak(int n, string s, vector<string> &d) {
        //code here
        map<string, bool> mp;
        for(auto x: d){
            mp[x]=1;
        }
        return true ==check(s, mp)? 1: 0;
    }
};
```

Using DP:

Why Dynamic Programming? The above problem exhibits overlapping sub-problems. For example, see the following partial recursion tree for string “abcde” in the worst case.



Partial recursion tree for input string "abcde". The subproblems encircled with green color are overlapping subproblems

```

// Returns true if dictionary d contains the given word
bool isInDict(const string &word, unordered_set<string>& d) {
    return d.find(word) != d.end();
}

// Returns true if a given string can be broken into
// dictionary d words, else false
bool wordBreak(const string &str, unordered_set<string>& d) {

    int n = str.size();

    // dp[i] is going to be true if the prefix of
    // length i can be broken into words.
    vector<bool> dp(n + 1, false);
```

```

dp[0] = true; // Answe is true for an empty string

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

    // Check for smaller strings and if we find
    // a j such that a prefix of length j can be
    // broken and str[j..i-1] is also a valid word
    for (int j = 0; j < i; j++) {
        if (dp[j] && isInDict(str.substr(j, i - j), d)) {
            dp[i] = true;
            break;
        }
    }
    return dp[n];
}

```

Optimized DP

```

// Returns true if dictionary d contains the given word
bool isInDict(const string &word, unordered_set<string> &d){
    return d.find(word) != d.end();
}

bool wordBreak(const string &s, unordered_set<string> &d)
{
    int n = s.size();
    if (n == 0)
        return true;

    vector<bool> dp(n + 1, 0);
    dp[0] = true;

    // matched_len stores lengths of
    // prefixes for which we have got
    // the result as true.
    vector<int> matched_len;
    matched_len.push_back(0);

    for (int i = 1; i <= n; i++)
    {
        int msize = matched_len.size();

```

```

// Consider all already matched lengths
// and check only for remaining strings
// after these already matched
for (int j = msize - 1; j >= 0; j--)
{
    int len = matched_len[j];

    // rem is substring starting from
    // matched_len[j] and of length
    // i - matched_len[j]
    string rem = s.substr(len, i - len);

    // If the remaining substring is
    // also a word
    if (isInDict(rem, d))
    {
        dp[i] = 1;
        matched_len.push_back(i);
        break;
    }
}
return dp[n];
}

```

Rabin Karp Algorithm for Pattern Searching

<https://www.geeksforgeeks.org/rabin-karp-algorithm-for-pattern-searching/>

Like the Naive Algorithm, the Rabin-Karp algorithm also check every substring. But unlike the Naive algorithm, the Rabin Karp algorithm matches the **hash value** of the **pattern** with the **hash value** of the current substring of **text**, and if the **hash values** match then only it starts matching individual characters. So Rabin Karp algorithm needs to calculate hash values for the following strings.

- **Pattern itself**
- **All the substrings of the text of length m which is the size of pattern.**

```
/* Following program is a C++ implementation of Rabin Karp
```

```

Algorithm given in the CLRS book */
#include <bits/stdc++.h>
using namespace std;

// d is the number of characters in the input alphabet
#define d 256

/* pat -> pattern
   txt -> text
   q -> A prime number
*/
void search(char pat[], char txt[], int q)
{
    int M = strlen(pat);
    int N = strlen(txt);
    int i, j;
    int p = 0; // hash value for pattern
    int t = 0; // hash value for txt
    int h = 1;

    // The value of h would be "pow(d, M-1)%q"
    for (i = 0; i < M - 1; i++)
        h = (h * d) % q;

    // Calculate the hash value of pattern and first
    // window of text
    for (i = 0; i < M; i++) {
        p = (d * p + pat[i]) % q;
        t = (d * t + txt[i]) % q;
    }

    // Slide the pattern over text one by one
    for (i = 0; i <= N - M; i++) {

        // Check the hash values of current window of text
        // and pattern. If the hash values match then only
        // check for characters one by one
        if (p == t) {
            /* Check for characters one by one */
            for (j = 0; j < M; j++) {
                if (txt[i + j] != pat[j]) {
                    break;
                }
            }
        }
    }
}

```

```

    }

    // if p == t and pat[0...M-1] = txt[i, i+1,
    // ...i+M-1]

    if (j == M)
        cout << "Pattern found at index " << i
        << endl;
}

// Calculate hash value for next window of text:
// Remove leading digit, add trailing digit
if (i < N - M) {
    t = (d * (t - txt[i] * h) + txt[i + M]) % q;

    // We might get negative value of t, converting
    // it to positive
    if (t < 0)
        t = (t + q);
}
}

/* Driver code */
int main()
{
    char txt[] = "GEEKS FOR GEEKS";
    char pat[] = "GEEK";

    // we mod to avoid overflowing of value but we should
    // take as big q as possible to avoid the collision
    int q = INT_MAX;

    // Function Call
    search(pat, txt, q);
    return 0;
}

```

Convert a sentence into its equivalent mobile numeric keypad sequence

```
// C++ implementation to convert a
// sentence into its equivalent
// mobile numeric keypad sequence
#include <bits/stdc++.h>
using namespace std;

// Function which computes the sequence
string printSequence(string arr[], string input)
{
    string output = "";

    // length of input string
    int n = input.length();
    for (int i = 0; i < n; i++) {
        // Checking for space
        if (input[i] == ' ')
            output = output + "0";

        else {
            // Calculating index for each
            // character
            int position = input[i] - 'A';
            output = output + arr[position];
        }
    }

    // Output sequence
    return output;
}

// Driver Code
int main()
{
    // storing the sequence in array
```

```

string str[]
= { "2", "22", "222", "3", "33", "333", "4",
    "44", "444", "5", "55", "555", "6", "66",
    "666", "7", "77", "777", "7777", "8", "88",
    "888", "9", "99", "999", "9999" };

string input = "GEEKSFORGEEKS";
cout << printSequence(str, input);
return 0;
}

```

Minimum number of bracket reversals needed to make an expression balanced.

```

int countRev (string s)
{
    // your code here
    int c =0, g=0;
    stack<char> st;
    if(s.size()%2==1) return -1;
    for(int i=0;i<s.size();i++){
        if(s[i]=='{'){
            st.push(s[i]);
        }else if(st.empty()){
            c++;
        }else if(st.size()>0){
            st.pop();
        }
        // cout<<st.size()<< " "<<endl;
    }

    g = st.size();
    // cout<<g<<" "<<c<<" \n";
    int a =-1;
    a = c/2;
    a+= g/2;
    c%=2;
    g%=2;
    a = a+g+c;
    // if(a==0) return -1;
    return a;
}

```

Count All Palindromic Subsequence in a given String.

```
#include <bits/stdc++.h>
using namespace std;

int countPS(string &s) {
    int n = s.size();
    vector<vector<int>> dp(n, vector<int>(n, 0));

    // Every single character is a palindrome,
    // so initialize diagonal elements
    for (int i = 0; i < n; i++) {
        dp[i][i] = 1;
    }

    // Fill the table for substrings of length greater than 1
    for (int length = 2; length <= n; length++) {
        for (int i = 0; i <= n - length; i++) {
            int j = i + length - 1;
            if (s[i] == s[j]) {
                dp[i][j] = dp[i+1][j] + dp[i][j-1] + 1;
            } else {
                dp[i][j] = dp[i+1][j] + dp[i][j-1] - dp[i+1][j-1];
            }
        }
    }

    return dp[0][n-1];
}

int main(){
    string s = "geeksforgeeks";
    cout << countPS(s);
}
```

Count of number of given string in 2D character array

Algorithm :

- **Step 1–** Take each row and convert it into string and apply KMP to find no of occurrence of given str in string(for left to right) and its reverse(for right to left).
- **Step 2–** sum occurrence for each row.
- **Step 3–** Apply same for each column.
- **Step 6–** return final answer as summation for column and rows.

Word Search in a 2D Grid of characters

Naive:

Time Complexity: $O(m*n*k)$, where **m** is the number of rows, **n** is the number of columns and **k** is the length of word.

Auxiliary Space: $O(k)$, recursion stack space.

```
// C++ program to search a word in a 2D grid
#include <bits/stdc++.h>
using namespace std;

// This function checks if the given
// coordinate is valid
bool validCoord(int x, int y, int m, int n) {
    if (x>=0 && x<m && y>=0 && y<n)
        return true;
    return false;
}

// This function searches for the given word
// in a given direction from the coordinate.
bool findWord(int index, string word, vector<vector<char>> &grid,
              int x, int y, int dirX, int dirY) {

    // if word has been found
    if (index == word.length()) return true;

    // if the current coordinate is
    // valid and characters match, then
    // check the next index
    if (validCoord(x, y, grid.size(), grid[0].size())
        && word[index] == grid[x][y])
        return findWord(index+1, word, grid, x+dirX,
```

```

        y+dirY, dirX, dirY);

    return false;
}

// This function calls search2D for each coordinate
vector<vector<int>>searchWord(vector<vector<char>>grid,
                                string word){
    int m = grid.size();
    int n = grid[0].size();

    vector<vector<int>>ans;

    // x and y are used to set the direction in which
    // word needs to be searched.
    vector<int>x = { -1, -1, -1, 0, 0, 1, 1, 1 };
    vector<int>y = { -1, 0, 1, -1, 1, -1, 0, 1 };

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

            // Search in all 8 directions
            for (int k=0; k<8; k++) {

                // If word is found, then append the
                // coordinates into answer and break.
                if (findWord(0, word, grid, i, j, x[k], y[k])) {
                    ans.push_back({i,j});
                    break;
                }
            }
        }
    }

    return ans;
}

void printResult(vector<vector<int>> ans) {
    for (int i=0; i<ans.size(); i++) {
        cout << "{" << ans[i][0] << "," << ans[i][1] << "}" << " ";
    }
    cout<<endl;
}

int main() {
    vector<vector<char>> grid = {{'a','b','a','b'},
                                    {'a','b','e','b'},
                                    {'e','b','e','b'}};

    string word = "abe";

    vector<vector<int>> ans = searchWord(grid, word);
}

```

```
    printResult(ans);
}
```

Better Approach:

Time complexity: $O(m \cdot n \cdot k)$, where **m** is the number of rows, **n** is the number of columns and **k** is the length of word.
Auxiliary Space: $O(1)$.

```
// C++ program to search a word in a 2D grid
#include <bits/stdc++.h>
using namespace std;

// This function searches for the given word
// in all 8 directions from the coordinate.
bool search2D(vector<vector<char>> grid, int row, int col, string word) {
    int m = grid.size();
    int n = grid[0].size();

    // return false if the given coordinate
    // does not match with first index char.
    if (grid[row][col] != word[0])
        return false;

    int len = word.size();

    // x and y are used to set the direction in which
    // word needs to be searched.
    vector<int>x = { -1, -1, -1, 0, 0, 1, 1, 1 };
    vector<int>y = { -1, 0, 1, -1, 1, -1, 0, 1 };

    // This loop will search in all the 8 directions
    // one by one. It will return true if one of the
    // directions contain the word.
    for (int dir = 0; dir < 8; dir++) {

        // Initialize starting point for current direction
        int k, currX = row + x[dir], currY = col + y[dir];

        // First character is already checked, match remaining
        // characters
        for (k = 1; k < len; k++) {

            // break if out of bounds
            if (currX >= m || currX < 0 || currY >= n || currY < 0)
                break;

            if (grid[currX][currY] != word[k])

```

```

        break;

        // Moving in particular direction
        currX += x[dir], currY += y[dir];
    }

    // If all character matched, then value of must
    // be equal to length of word
    if (k == len)
        return true;
}

// if word is not found in any direction,
// then return false
return false;
}

// This function calls search2D for each coordinate
vector<vector<int>>searchWord(vector<vector<char>>grid, string word){
    int m = grid.size();
    int n = grid[0].size();

    vector<vector<int>>ans;

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

            // if the word is found from this coordinate,
            // then append it to result.
            if(search2D(grid, i, j, word)){
                ans.push_back({i, j});
            }
        }
    }

    return ans;
}

void printResult(vector<vector<int>> ans) {
    for (int i=0; i<ans.size(); i++) {
        cout << "{" << ans[i][0] << "," << ans[i][1] << "}" << " ";
    }
    cout<<endl;
}

int main() {
    vector<vector<char>> grid =
{{'a','b','a','b'},{'a','b','e','b'},{'e','b','e','b'}};
    string word = "abe";

    vector<vector<int>> ans = searchWord(grid, word);
}

```

```
    printResult(ans);
}
```

Converting Roman Numerals to Decimal

```
#include <bits/stdc++.h>
using namespace std;

// This function returns value of a Roman symbol
int value(char r)
{
    if (r == 'I')
        return 1;
    if (r == 'V')
        return 5;
    if (r == 'X')
        return 10;
    if (r == 'L')
        return 50;
    if (r == 'C')
        return 100;
    if (r == 'D')
        return 500;
    if (r == 'M')
        return 1000;
    return -1;
}

// Returns decimal value of roman numeral
int romanToDecimal(string& str)
{
    int res = 0; // Initialize result

    for (int i = 0; i < str.length(); i++) {
        // Get value of current symbol
        int s1 = value(str[i]);

        // Compare with the next symbol if it exists
        if (i + 1 < str.length()) {
            int s2 = value(str[i + 1]);

            // If current value is greater or equal, add it
            // to result
            if (s1 >= s2) {
                res += s1;
            }
            else {
```

```

        // Else, add the difference and skip next
        // symbol
        res += (s2 - s1);
        i++;
    }
}
else {
    res += s1;
}
}

return res;
}

// Driver code
int main()
{
    string str = "IX";
    cout << romanToDecimal(str) << endl;
    return 0;
}

```

```

#include <bits/stdc++.h>
using namespace std;

int romanToDecimal(string& str) {

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

    // Initialize result
    int sum = 0;
    for (int i = 0; i < str.length(); i++) {

        // If the current value is less than the next value, subtract current from next
        // and add to sum
        if (i + 1 < str.length() && romanMap[str[i]] < romanMap[str[i + 1]]) {
            sum += romanMap[str[i + 1]] - romanMap[str[i]];

            // Skip the next symbol
            i++;
        } else {

            // Otherwise, add the current value to sum
            sum += romanMap[str[i]];
        }
    }
}

```

```

        }
    }

    return sum;
}

// Driver code
int main() {

    string str = "IX";
    cout << romanToDecimal(str) << endl;
    return 0;
}

```

Longest Common Prefix

```

class Solution {
public:
    string longestCommonPrefix(vector<string>& strs) {
        int n = strs.size();
        string ans = "";
        int min_size_str=2000;
        int i,j;
        for(i=0;i<n;i++){
            int ss = strs[i].size();
            if(ss< min_size_str){
                min_size_str = ss;
            }
        }
        cout<<min_size_str<<" min size str\n";
        if(min_size_str ==0){
            return ans;
        }
        if(n==0){
            return ans;
        }
        if(n==1){
            return strs[0];
        }
        cout<<"ha last stage\n";
        for(i=0;i<min_size_str;i++){
            bool yes = true;
            for(j=1;j<n;j++){
                if(strs[j-1][i]!=strs[j][i]){
                    yes = false;
                    return ans;
                }
            }
        }
    }
}

```

```

        if(yes){
            ans+=strs[0][i];
        }

    }
    return ans;
}
};

```

Number of flips to make binary string alternate

```

int minFlips (string s)
{
    // your code here

    int n = s.size();
    string s1="",s2="";
    int c1=0,c2=0;
    for(int i=0;i<n;i++){
        //creating first string
        if(i%2) s1+="1";
        else s1+="0";

        //creating second string
        if(i%2) s2+="0";
        else s2+="1";

        if(s[i]!=s1[i])c1++;
        if(s[i]!=s2[i])c2++;
    }
    return min(c1,c2);
}

```

Find the first repeated word in string.

```

// Cpp program to find first repeated word in a string
#include<bits/stdc++.h>
using namespace std;
void solve(string s)
{
    unordered_map<string,int> mp; // to store occurrences of word
    string t="",ans="";
    // traversing from back makes sure that we get the word which repeats first as ans
    for(int i=s.length()-1;i>=0;i--)
    {
        // if char present , then add that in temp word string t
        if(s[i]!=' ')
        {
            t+=s[i];

```

```

    }
    // if space is there then this word t needs to stored in map
    else
    {
        mp[t]++;
        // if that string t has occurred previously then it is a possible ans
        if(mp[t]>1)
            ans=t;
        // set t as empty for again new word
        t="";
    }

}

// first word like "he" needs to be mapped
mp[t]++;
if(mp[t]>1)
    ans=t;

if(ans!="")
{
    // reverse ans string as it has characters in reverse order
    reverse(ans.begin(),ans.end());
    cout<<ans<<'\n';
}
else
    cout<<"No Repetition\n";
}

int main()
{
    string u="Ravi had been saying that he had been there";
    string v="Ravi had been saying that";
    string w="he had had he";
    solve(u);
    solve(v);
    solve(w);

    return 0;
}

```

Minimum number of swaps for bracket balancing.

```

class Solution {
public:

```

```

int minimumNumberOfSwaps(string& s) {
    // code here
    // given ]]]][[
    //pick the first [ and count how many ] are there before [
    // if pos is 1 and before bracs are 3
    // then pos i should have i-1 closing bracks before it
    // before bracs are i.e. is the count of swaps
    int n = s.size();
    int ans=0, curr =1, ps=0, ctr=0;
    for(int i=0;i<n;i++){
        if(s[i]==']')ctr++;
        else if(s[i]=='['){
            if(ctr>=curr){
                ans += ctr-curr +1;
            }
            curr++;
        }
    }
    return ans;
}

```

Write a program to find the smallest window that contains all characters of string itself.

```

class Solution{
public:
    int findSubString(string str)
    {
        // Your code goes here
        map<char, int> mp,test;
        int n = str.size();
        for(int i=0;i<n;i++){
            mp[str[i]]++;
        }
        int total_diff_letters = mp.size();
        int i=0,j=0;
        int curr_window_size=0;
        int ans = INT_MAX;

        while(i<=j and j<=n){
            if(test.size()<mp.size()){
                test[str[j]]++;
                j++;
            }
        }
    }
};

```

```

    }else{
        ans = min(ans, j-i);
        test[str[i]]--;

        if(test[str[i]]==0){
            auto itr = test.find(str[i]);
            test.erase(itr);
        }
        i++;
    }
}
return ans;
};


```

Rearrange characters in a string such that no two adjacent are same

```

// C++ code for the above approach

#include <bits/stdc++.h>
using namespace std;

char getMaxCountChar(vector<int>& count)
{
    int max = 0;
    char ch;
    for (int i = 0; i < 26; i++) {
        if (count[i] > max) {
            max = count[i];
            ch = 'a' + i;
        }
    }

    return ch;
}

string rearrangeString(string S)
{
    int n = S.size();
    if (n == 0)
        return "";
    vector<int> count(26, 0);
    for (auto& ch : S)

```

```

        count[ch - 'a']++;

    char ch_max = getMaxCountChar(count);
    int maxCount = count[ch_max - 'a'];

    // check if the result is possible or not
    if (maxCount > (n + 1) / 2)
        return "";

    string res(n, ' ');
    int ind = 0;

    // filling the most frequently occurring char in the
    // even indices
    while (maxCount) {
        res[ind] = ch_max;
        ind = ind + 2;
        maxCount--;
    }

    count[ch_max - 'a'] = 0;

    // now filling the other Chars, first
    // filling the even positions and then
    // the odd positions
    for (int i = 0; i < 26; i++) {

        while (count[i] > 0) {

            ind = (ind >= n) ? 1 : ind;
            res[ind] = 'a' + i;
            ind += 2;
            count[i]--;
        }
    }

    return res;
}

// Driver's code
int main()
{
    string str = "bbbbaa";

    // Function call
    string res = rearrangeString(str);
    if (res == "")
        cout << "Not possible" << endl;
    else
        cout << res << endl;

    return 0;
}

```

```
}
```

Minimum characters to be added at front to make string palindrome

Start by checking if the entire string is a palindrome. If it is, then we don't need to add any letters. If the entire string isn't a palindrome, we then check the next longest prefix, which is the string without its last letter. We keep checking shorter prefixes until we find the longest one that is a palindrome. As soon as we find a prefix which is also a palindrome, we return the number of remaining characters as the minimum number of characters to be added to the front to make the string palindrome.

```
// C++ program for counting minimum character to be
// added at front to make string palindrome
#include <iostream>
using namespace std;

// Function to check if the substring s[i...j] is a palindrome
bool isPalindrome(string &s, int i, int j) {
    while (i < j) {

        // If characters at the ends are not equal, it's not a palindrome
        if (s[i] != s[j]) {
            return false;
        }
        i++;
        j--;
    }
    return true;
}

int minChar(string &s) {
    int cnt = 0;
    int i = s.size() - 1;

    // Iterate from the end of the string, checking for the longest
    // palindrome starting from the beginning
    while (i >= 0 && !isPalindrome(s, 0, i)) {

        i--;
        cnt++;
    }

    return cnt;
}

int main() {
    string s = "AACECAAAA";
    cout << minChar(s);
```

```
    return 0;
}
```

```
// C++ program for getting minimum character to be
// added at front to make string palindrome

#include <algorithm>
#include <iostream>
#include <vector>
using namespace std;

vector<int> computeLPSArray(string &pat) {
    int n = pat.length();
    vector<int> lps(n);

    // lps[0] is always 0
    lps[0] = 0;
    int len = 0;

    // loop calculates lps[i] for i = 1 to M-1
    int i = 1;
    while (i < n) {

        // If the characters match, increment len
        // and set lps[i]
        if (pat[i] == pat[len]) {
            len++;
            lps[i] = len;
            i++;
        }

        // If there is a mismatch
        else if (pat[i] != pat[len]) {

            // If len is not zero, update len to
            // the last known prefix length
            if (len != 0) {
                len = lps[len - 1];
            }

            // No prefix matches, set lps[i] to 0
            else {
                lps[i] = 0;
                i++;
            }
        }
    }
    return lps;
}
```

```

// Method returns minimum character to be added at
// front to make string palindrome
int minChar(string &s) {
    int n = s.length();
    string rev = s;
    reverse(rev.begin(), rev.end());

    // Get concatenation of string, special character
    // and reverse string
    s = s + "$" + rev;

    // Get LPS array of this concatenated string
    vector<int> lps = computeLPSArray(s);

    // By subtracting last entry of lps vector from
    // string length, we will get our result
    return (n - lps.back());
}

int main() {
    string s = "AAECECAAAA";
    cout << minChar(s);
    return 0;
}

```

Given a sequence of words, print all anagrams together

Given an array of strings, return all groups of strings that are anagrams. The groups must be created in order of their appearance in the original array. Look at the sample case for clarification.

Note: The final output will be in **lexicographic** order.

Examples:

Input: arr[] = ["act", "god", "cat", "dog", "tac"]

Output: [[["act", "cat", "tac"], ["god", "dog"]]]

Explanation: There are 2 groups of anagrams "god", "dog" make group 1. "act", "cat", "tac" make group 2.

```

class Solution {
public:

```

```

vector<vector<string>> anagrams(vector<string>& arr) {
    // code here
    int n = arr.size();
    vector<vector<string>> ans,vb;
    map<string, int> mp;
    map<string, vector<string> > as;
    for(int i=0;i<n;i++){
        string s = arr[i];
        sort(s.begin(), s.end());
        as[s].push_back(arr[i]);
    }
    for(auto x: as){
        // sort(x.begin(),x.end());
        ans.push_back(x.second);
    }
    for(auto x: ans){
        //sort(x.begin(),x.end());
        vb.push_back(x);
    }
    sort(vb.begin(),vb.end());
    return vb;
}
};
```

Smallest window in a string containing all the characters of another string

```

#include <bits/stdc++.h>
using namespace std;

string smallestWindow(string s, string p) {
    int len1 = s.length();
    int len2 = p.length();

    // Check if string's length is less than P's length
    if (len1 < len2) return "-1";

    // Initialize hash arrays for P and string S
    vector<int> hashP(256, 0);
    vector<int> hashS(256, 0);

    // Store occurrence of characters of P
    for (int i = 0; i < len2; i++) hashP[p[i]]++;

    int start = 0, start_idx = -1, min_len = INT_MAX;
```

```

// Count of characters matched
int count = 0;

// Start traversing the string
for (int j = 0; j < len1; j++) {
    // Count occurrence of characters of string S
    hashS[s[j]]++;

    // If S's char matches with P's char, increment count
    if (hashP[s[j]] != 0 && hashS[s[j]] <= hashP[s[j]]) {
        count++;
    }

    // If all characters are matched
    if (count == len2) {
        // Try to minimize the window
        while (hashS[s[start]] > hashP[s[start]]
               || hashP[s[start]] == 0) {
            if (hashS[s[start]] > hashP[s[start]]) {
                hashS[s[start]]--;
            }
            start++;
        }

        // Update window size
        int len = j - start + 1;
        if (min_len > len) {
            min_len = len;
            start_idx = start;
        }
    }
}

// If no window found
if (start_idx == -1) return "-1";

// Return the substring starting from start_idx
// and length min_len
return s.substr(start_idx, min_len);
}

int main() {
    string s = "timetopractice";
    string p = "toc";

```

```

    string result = smallestWindow(s, p);
    cout << result;

    return 0;
}

```

Recursively remove all adjacent duplicates

```

#include <bits/stdc++.h>
using namespace std;

// Helper function to remove adjacent duplicates
void remove_util(string &str, int n) {

    // Get the length of the string
    int len = str.length();

    // Index to store the result string
    int k = 0;

    // Iterate over the string to remove adjacent
    // duplicates
    for (int i = 0; i < n; i++) {

        // Check if the current character is the same
        // as the next one
        if (i < n - 1 && str[i] == str[i + 1]) {
            // Skip all the adjacent duplicates
            while (i < n - 1 && str[i] == str[i + 1]) {
                i++;
            }
        } else {
            // If not a duplicate, store the character
            str[k++] = str[i];
        }
    }

    // Remove the remaining characters from the
    // original string
    str.erase(k);

    // If any adjacent duplicates were removed,
    // recursively check for more
    if (k != n)

```

```

        remove_util(str, k);
    }

// Function to initiate the removal of adjacent
// duplicates
string rremove(string s) {

    // Call the helper function
    remove_util(s, s.length());

    // Return the modified string
    return s;
}

int main() {
    string s = "geeksforgeek";
    cout << rremove(s) << endl;
    return 0;
}

```

Recursively print all sentences that can be formed from list of word lists

```

class Solution{
public:
    vector<vector<string>>ans;
    void solve(int i,vector<vector<string>>&list,int n,string s){
        if(i>n-1){
            vector<string>out;
            out.push_back(s);
            ans.push_back(out);
            return;
        }
        for(auto val:list[i]){
            if(i==n-1){
                solve(i+1,list,n,s+val);
            }
            else{
                solve(i+1,list,n,s+val+" ");
            }
        }
    }
    vector<vector<string>> sentences(vector<vector<string>>&list){
        //Write your code here
    }
}

```

```

        int n=list.size();

        solve(0,list,n,"");
        return ans;
    }

};

```

Transform One String to Another using Minimum Number of Given Operation

```

#include <bits/stdc++.h>
using namespace std;

int transform(string A, string B)
{
    if (A.length() != B.length()) {
        return -1;
    }

    // create a map to store the frequency of characters in string A
    unordered_map<char, int> m;
    int n = A.length();
    for (int i = 0; i < n; i++) {
        if (m.count(A[i])) // if the character already exists in the map
            m[A[i]]++;      // increment its frequency
        else
            m[A[i]] = 1;     // add the character to the map with a frequency of 1
    }

    // subtract the frequency of characters in string B from the map
    for (int i = 0; i < n; i++) {
        if (m.count(B[i]))
            m[B[i]]--;
    }

    // check if all the frequencies in the map are 0, indicating equal frequency of
    // characters in both strings
    for (auto it : m) {
        if (it.second != 0) // if frequency is not zero
            return -1;      // strings cannot be transformed into each other, return -1
    }

    // calculate the minimum number of operations required to transform string A into string
    B
    int i = n - 1, j = n - 1;
    int res = 0;

```

```

while (i >= 0 && j >= 0) {
    while (i >= 0 && A[i] != B[j]) {
        res++; // increment the number of operations required
        i--; // move the pointer i to the left
    }
    i--;
    j--;
}
return res; // returning result
}

// Driver code
int main()
{
    string A = "EACBD";
    string B = "EABCD";

    cout << "Minimum number of operations required is " << transform(A, B) << endl;
    return 0;
}

```

Check if two given strings are isomorphic to each other

Given two strings s1 and s2, check if these two strings are isomorphic to each other.

If the characters in s1 can be changed to get s2, then two strings, s1 and s2, are isomorphic. A character must be completely swapped out for another character while maintaining the order of the characters. A character may map to itself, but no two characters may map to the same character.

Examples:

Input: s1 = "aab", s2 = "xxy"

Output: true

Explanation: There are two different characters in aab and xxy, i.e a and b with frequency 2 and 1 respectively.

```

class Solution {
public:
    // Function to check if two strings are isomorphic.
    bool areIsomorphic(string &s1, string &s2) {

        // Your code here
        int n1=s1.size(),n2=s2.size();
        map<char,char> mp;
        unordered_set<char> st;

```

```

if(n1!=n2) return false;

for(int i=0;i<n1;i++){

    //case1: new char
    if(mp.find(s1[i]) == mp.end()){
        if(st.find(s2[i])==st.end()){
            mp[s1[i]]=s2[i];
            st.insert(s2[i]);
        }
    }

    else{
        return false;
    }

}else{
    //if char passed and mapped not same
    if(mp[s1[i]]!=s2[i]){
        return false;
    }
}

}

return true;
};

}

```

Function to find Number of customers who could not get a computer

Write a function “runCustomerSimulation” that takes following two inputs

An integer ‘n’: total number of computers in a cafe and a string:

A sequence of uppercase letters ‘seq’: Letters in the sequence occur in pairs. The first occurrence indicates the arrival of a customer; the second indicates the departure of that same customer.

A customer will be serviced if there is an unoccupied computer. No letter will occur more than two times.

Customers who leave without using a computer always depart before customers who are currently using the computers. There are at most 20 computers per cafe.

For each set of input the function should output a number telling how many customers, if any walked away without using a computer. Return 0 if all the customers were able to use a computer.

runCustomerSimulation (2, “ABBAJKZKZ”) should return 0

runCustomerSimulation (3, “GACCBDDBAGEE”) should return 1 as ‘D’ was not able to get any computer

runCustomerSimulation (3, “GACCBGDDBAEE”) should return 0

runCustomerSimulation (1, “ABCBCA”) should return 2 as ‘B’ and ‘C’ were not able to get any computer.

runCustomerSimulation(1, “ABCBCADEED”) should return 3 as ‘B’, ‘C’ and ‘E’ were not able to get any computer.

```
// C++ program to find number of customers who couldn't get a resource.
```

```

#include<iostream>
#include<cstring>
using namespace std;

#define MAX_CHAR 26

// n is number of computers in cafe.
// 'seq' is given sequence of customer entry, exit events
int runCustomerSimulation(int n, const char *seq)
{
    // seen[i] = 0, indicates that customer 'i' is not in cafe
    // seen[1] = 1, indicates that customer 'i' is in cafe but
    //           computer is not assigned yet.
    // seen[2] = 2, indicates that customer 'i' is in cafe and
    //           has occupied a computer.
    char seen[MAX_CHAR] = {0};

    // Initialize result which is number of customers who could
    // not get any computer.
    int res = 0;

    int occupied = 0; // To keep track of occupied computers

    // Traverse the input sequence
    for (int i=0; seq[i]; i++)
    {
        // Find index of current character in seen[0...25]
        int ind = seq[i] - 'A';

        // If First occurrence of 'seq[i]'
        if (seen[ind] == 0)
        {
            // set the current character as seen
            seen[ind] = 1;

            // If number of occupied computers is less than
            // n, then assign a computer to new customer
            if (occupied < n)
            {
                occupied++;

                // Set the current character as occupying a computer
                seen[ind] = 2;
            }
        }

        // Else this customer cannot get a computer,
        // increment result
        else
            res++;
    }

    // If this is second occurrence of 'seq[i]'
}

```

```

        else
        {
            // Decrement occupied only if this customer
            // was using a computer
            if (seen[ind] == 2)
                occupied--;
            seen[ind] = 0;
        }
    }
    return res;
}

// Driver program
int main()
{
    cout << runCustomerSimulation(2, "ABBAJJKZKZ") << endl;
    cout << runCustomerSimulation(3, "GACCBDBBAGEE") << endl;
    cout << runCustomerSimulation(3, "GACCBGDDBAEE") << endl;
    cout << runCustomerSimulation(1, "ABCBCA") << endl;
    cout << runCustomerSimulation(1, "ABCBCADEED") << endl;
    return 0;
}

```

String matching where one string contains wildcard characters

* --> This character in string wild can be replaced by any sequence of characters, it can also be replaced by an empty string.

? --> This character in string wild can be replaced by any one character.

```

class Solution{
public:

    bool match(string wild, string pattern) {
        int n = wild.size();
        int m = pattern.size();

        int i = 0, j = 0;
        int checkpointWild = -1, checkpointPattern = -1;

        while (j < m) {
            if (i < n && (wild[i] == pattern[j] || wild[i] == '?')) {
                i++;
                j++;
            } else if (i < n && wild[i] == '*') {
                checkpointWild = i;
                checkpointPattern = j;

```

```

        i++;
    } else if (checkpointWild != -1) {
        // Backtrack to the last '*' checkpoint
        i = checkpointWild + 1;
        j = checkpointPattern + 1;
        checkpointPattern++;
    } else {
        return false;
    }
}

while (i < n && wild[i] == '*') {
    i++;
}

return i == n && j == m;
}
};

```

Linked List:

Reverse a Linked List:

```

Node* reverseIterative(Node *head) {
    Node* prev = NULL, *next = NULL, *curr = head;
    while (curr != NULL) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    head = prev;
    return head;
}

```

```

Node* reverseRecursive(Node* head) {
    if (head == NULL || head->next == NULL)
        return head;

    Node* reverse = reverseRecursive(head->next);
    head->next->next = head;
    head->next = NULL;
    return reverse;
}

```

Reverse a LInked List in a given group size:

Version 1:

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

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

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

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

```

Node* reverseInGroupSize(Node* head, int k) {
    int c = 0;
    if (!head) return NULL;
    Node* prev = NULL, *next = NULL, *curr = head;
    while (curr != NULL && c < k) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
        c++;
    }
    if (next != NULL)
        head->next = reverseInGroupSize(next, k);
    return prev;
}

```

Version 2:

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

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

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

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

```
class Solution {
    bool sizze(ListNode* h,int k){
        int c=0;
        while(h!=NULL and c<k){
            c++;
            h=h->next;
        }
        if(c>=k) return true;
        return false;
    }

public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        int c = 0;
        if (!head) return NULL;
        ListNode* prev = NULL, *next = NULL, *curr = head;
        if(!sizze(head,k)) return head;
        while (curr != NULL && c < k) {
            next = curr->next;
            curr->next = prev;
            prev = curr;
            curr = next;
            c++;
        }
        if (next != NULL)
            head->next = reverseKGroup(next, k);
        return prev;
    }
};
```

Detect Loop in a Linked List:

Floyd's - Cycle finding Algorithm

```
bool detect_loop_F(Node *h) {
    Node *f = h, *s = h;
    while (s && f && f->next) {
```

```

        f = f->next->next;
        s = s->next;
        if (f == s) {
            return true;
        }
    }
    return false;
}

```

Using undoreder_set

```

bool detectLoopInListViaSet(Node* head) {
    unordered_set<Node*> s;
    while (head != NULL) {
        if (s.find(head) == s.end()) {
            s.insert(head);
            head = head->next;
        }
        else
            return true;
    }
    return false;
}

```

Delete Loop in a Linked List:

Solution 1:

```

void loopDetDelBrute(node * head) {
    unordered_set < node * > uns;

    node * prev = NULL;
    node * current = head;
    while (current != NULL) {
        if (uns.find(current) != uns.end()) { // loop detected
            prev -> next = NULL; // breaking the loop
            return;
        }
    }
}

```

```

    }
    uns.insert(current);

    prev = current;
    current = current -> next;
}
}

```

Solution: 2 (optimal approach)

```

void loop_detection_deletion(node * head) {

    // Floyd Loop Detection Algorithm
    node * slow = head, * fast = head;
    while (fast != NULL && fast -> next != NULL) {
        slow = slow -> next;
        fast = fast -> next -> next;
        if (slow == fast) break; // first meeting point
    }
    if (slow != fast) return;

    // Loop Removal Algorithm
    slow = head;
    // locating the starting node of the loop
    while (slow -> next != fast -> next) {
        slow = slow -> next;
        fast = fast -> next;
    }
    // terminating the loop
    fast -> next = NULL;
}

```

Find the starting point of the Loop:

```
//method 1
Node* findLoopStartingPointMethod1(Node* h) {
    if (!h || !(h->next)) return NULL;
    Node* f = h, *s = h;
    while (s and f and f->next) {
        s = s->next;
        f = f->next->next;
        if (s == f) break;
    }
    if (s != f) return NULL;
    s = h;
    while (s != f) {
        s = s->next;
        f = f->next;
    }
    return s;
}
//Method 2
Node* findLoopStartingPointMethod2(Node* h) {
    if (!h or !(h->next)) return NULL;
    unordered_set<Node*> s;
    while (h != NULL) {
        if (s.find(h) != s.end())
            return h;
        else {
            s.insert(h);
            h = h->next;
        }
    }
    return NULL;
}
```

Remove Duplicated from Sorted Linked List:

```
Node* removeDuplicatesSorted(Node* h) {
    Node* c = h, *p;
    while (c->next != NULL) {
```

```

        if (c->data != c->next->data)
            c = c->next;
        else {
            p = c->next->next;
            free(c->next);
            c->next = p;
        }
    }
    return h;
}

```

Remove Duplicates from unsorted Linked List:

Solution: 1

```

Node* removeDuplicatesUnsorted(Node* h) {
    if (!h) return h;
    Node* p1, * p2, * dup;
    p1 = h;
    while (p1 != NULL and p1->next != NULL) {
        p2 = p1;
        while (p2->next != NULL) {
            if (p1->data == p2->next->data) {
                dup = p2->next;
                p2->next = p2->next->next;
                delete(dup);
            }
            else {
                p2 = p2->next;
            }
        }
        p1 = p1->next;
    }
    return h;
}

```

Solution: 2

```

Node* removeDuplicatesUnsorted(Node* h){
    unordered_set<int> s;
    Node* p1 = h,*p2;

```

```

while(p1){
    if(s.find(p1->data) != s.end()){
        p2->next = p1->next;
    }else{
        s.insert(p1->data);
        p2 = p1;
    }
    p1 = p1->next;
}
return h;
}

```

Bring Last element to first in Linked List:

```

Node* moveLastToFront(Node* h) {
    if (!h || !(h->next)) return h;
    Node* p1 = h, *last = h;
    while (p1->next != NULL) {
        last = p1;
        p1 = p1->next;
    }
    last->next = NULL;
    h = pushFront(h, p1->data);
    delete(p1);
    return h;
}

```

Add “1” to number represent by linked list:

```

Node* addOneToLinkedList(Node* h) {
    h = reverseIterative(h);
    int carry = 1;
    Node* p1 = h, *last;
    while (p1 != NULL) {
        carry = (p1->data + carry);
        p1->data = carry % 10;
        carry = (carry > 9) ? 1 : 0;
    }
}

```

```

        last = p1;
        p1 = p1->next;
    }
    cout << endl;
    if (carry) {
        Node* t = new Node();
        t->data = carry;
        last->next = t;
    }
    h = reverseIterative(h);
    return h;
}

```

Add Two Numbers Represented by Linked List:

```

Node* addTwoList(Node* h1, Node* h2) {
    h1 = reverseIterative(h1);
    h2 = reverseIterative(h2);
    int carry = 0;
    Node* p1 = h1, *last1, *last2, *p2 = h2;
    Node* sum;
    while (p1 != NULL and p2 != NULL) {
        carry = p1->data + p2->data + carry;
        sum = pushFront(sum, carry % 10);
        carry = carry > 9 ? carry / 10 : 0;
        last1 = p1;
        last2 = p2;
        p1 = p1->next;
        p2 = p2->next;
    }
    while (p1 != NULL) {
        carry = p1->data + carry;
        sum = pushFront(sum, carry % 10);
        carry = carry > 9 ? carry / 10 : 0;
        last1 = p1;
        p1 = p1->next;
    }
    while (p2 != NULL) {
        carry = p2->data + carry;
        sum = pushFront(sum, carry % 10);
    }
}

```

```

        carry = carry > 9 ? carry / 10 : 0;
        last2 = p2;
        p2 = p2->next;
    }
    if (carry) {
        sum = pushFront(sum, carry);
    }
    sum = reverseRecursive(sum);
    return sum;
}

```

Intersection of two sorted Linked List:

```

Node* findIntersection(Node* h1, Node* h2)
{
    Node* ans = new Node(0);
    Node* s = ans;
    Node* i = h1, *j = h2;
    while ( i and j) {
        if (i->data == j->data) {
            Node* temp = new Node(i->data);
            ans->next = temp;
            ans = ans->next;
            i = i->next;
            j = j->next;
        }
        else if (i->data < j->data) {
            i = i->next;
        }
        else {
            j = j->next;
        }
    }

    return s->next;
}

```

Find the middle element of the linked list:

```
Node* middleNode(Node* head) {  
    Node* s = head;  
    Node* f = head;  
    while (f and f->next) {  
        s = s->next;  
        f = f->next->next;  
    }  
    return s;  
}
```

Check if linked list is circular:

```
bool isCircular(struct Node *head) {  
    //code here  
    Node* s = head, *f = head;  
    while (f and f->next) {  
        s = s->next;  
        f = f->next->next;  
        if (s == f) return true;  
    }  
    return false;  
}
```

Nth Node of a Linked List from End:

```
class Solution{  
public:  
    int getNthFromLast(Node *head, int n)  
    {  
        // Your code here  
        int size =0;  
        Node*h = head;  
        while(h){  
            size++;  
            h=h->next;
```

```

    }
    if(size<n) return -1;
    int ans;
    size = size-n+1;
    while(size--){
        ans = head->data;
        head = head->next;
    }
    return ans;
}
};

```

Split a Circular linked list into two halves.

```

// C++ Program to split a circular linked list
// into two halves
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node *next;
    Node (int new_value){
        data = new_value;
        next = nullptr;
    }
};

// Function to split a list (starting with head)
// into two lists.
pair<Node*, Node*> splitList(Node *head) {
    Node *slow = head;
    Node *fast = head;

    if(head == nullptr)
        return {nullptr, nullptr};

    // For odd nodes, fast->next is head and
    // for even nodes, fast->next->next is head
    while(fast->next != head &&

```

```

        fast->next->next != head) {
    fast = fast->next->next;
    slow = slow->next;
}

// If there are even elements in list
// then move fast
if(fast->next->next == head)
    fast = fast->next;

// Set the head pointer of first half
Node* head1 = head;

// Set the head pointer of second half
Node* head2 = slow->next;

// Make second half circular
fast->next = slow->next;

// Make first half circular
slow->next = head;

return {head1, head2};
}

void printList(Node *head) {
    Node *curr = head;
    if(head != nullptr) {
        do {
            cout << curr->data << " ";
            curr = curr->next;
        } while(curr != head);
        cout << endl;
    }
}

int main() {

    Node *head = new Node(1);
    Node *head1 = nullptr;
    Node *head2 = nullptr;

    // Created linked list will be 1->2->3->4
}

```

```

head->next = new Node(2);
head->next->next = new Node(3);
head->next->next->next = new Node(4);
head->next->next->next->next = head;

pair<Node*, Node*> result = splitList(head);
head1 = result.first;
head2 = result.second;

printList(head1);
printList(head2);

return 0;
}

```

Write a Program to check whether the Singly Linked list is a palindrome or not.

[Naive Approach] Using Stack – O(n) Time and O(n) Space

[Expected Approach] Using Iterative Method – O(n) Time and O(1) Space

The approach involves reversing the second half of the linked list starting from the middle. After reversing, traverse from the head of the list and the head of the reversed second half simultaneously, comparing the node values. If all corresponding nodes have equal values, the list is a palindrome.

```

// C++ program to check if a linked list is palindrome
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;
    Node(int d) {
        data = d;
        next = nullptr;
    }
};

// Function to reverse a linked list

```

```

Node* reverse(Node* head) {
    Node* prev = nullptr;
    Node* curr = head;
    Node* next;

    while (curr) {
        next = curr->next;
        curr->next = prev;
        prev = curr;
        curr = next;
    }
    return prev;
}

// Function to check if two lists are identical
bool isIdentical(Node* n1, Node* n2) {
    for (; n1 && n2; n1 = n1->next, n2 = n2->next)
        if (n1->data != n2->data)
            return 0;

    // returning 1 if data at all nodes are equal.
    return 1;
}

// Function to check whether the list is palindrome
bool isPalindrome(Node* head) {
    if (!head || !head->next)
        return true;

    // Initialize slow and fast pointers
    Node* slow = head;
    Node* fast = head;

    // Move slow to the middle of the list
    while (fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }

    // Split the list and reverse the second half
    Node* head2 = reverse(slow->next);
    slow->next = nullptr; // End the first half
}

```

```

// Check if the two halves are identical
bool ret = isIdentical(head, head2);

// Restore the original list
head2 = reverse(head2);
slow->next = head2;

return ret;
}

int main() {

// Linked list : 1->2->3->2->1
Node head(1);
head.next = new Node(2);
head.next->next = new Node(3);
head.next->next->next = new Node(2);
head.next->next->next->next = new Node(1);

bool result = isPalindrome(&head);

if (result)
    cout << "true\n";
else
    cout << "false\n";

return 0;
}

```

Deletion from a Circular Linked List

There are three main ways to delete a node from circular linked list:

- Deletion at the beginning
- Deletion at specific position
- Deletion at the end

Reverse a Doubly Linked List:

```
Node *reverse(Node *head) {

    // If the list is empty or has only one node,
    // return the head as is
    if (head == nullptr || head->next == nullptr)
        return head;

    Node *prevNode = NULL;
    Node *currNode = head;

    // Traverse the list and reverse the links
    while (currNode != nullptr) {

        // Swap the next and prev pointers
        prevNode = currNode->prev;
        currNode->prev = currNode->next;

        currNode->next = prevNode;

        // Move to the next node in the original list
        // (which is now previous due to reversal)
        currNode = currNode->prev;
    }

    // The final node in the original list
    // becomes the new head after reversal
    return prevNode->prev;
}
```

Find pairs with given sum in doubly linked list

Using two points like two sum problem:

```
// C++ program to find a pair with given sum target.
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
```

```

int data;
Node *next, *prev;

Node(int value) {
    data = value;
    next = nullptr;
    prev = nullptr;
}
};

// Function to find pairs in the doubly
// linked list whose sum equals the given value x
vector<pair<int, int>> findPairsWithGivenSum(Node *head, int target) {

    vector<pair<int, int>> res;

    // Set two pointers, first to the beginning of DLL
    // and second to the end of DLL.
    Node *first = head;
    Node *second = head;

    // Move second to the end of the DLL
    while (second->next != nullptr)
        second = second->next;

    // To track if we find a pair or not
    bool found = false;

    // The loop terminates when two pointers
    // cross each other (second->next == first),
    // or they become the same (first == second)
    while (first != second && second->next != first) {

        // If the sum of the two nodes is equal to x, print the pair
        if ((first->data + second->data) == target) {
            found = true;
            res.push_back({first->data, second->data});

            // Move first in forward direction
            first = first->next;

            // Move second in backward direction
            second = second->prev;
        }
    }
}

```

```

    }
    else {
        if ((first->data + second->data) < target)
            first = first->next;
        else
            second = second->prev;
    }
}

// If no pair is found
return res;
}

int main() {

    // Create a doubly linked list: 1 <-> 2 <-> 4 <-> 5
    Node *head = new Node(1);
    head->next = new Node(2);
    head->next->prev = head;
    head->next->next = new Node(4);
    head->next->next->prev = head->next;
    head->next->next->next = new Node(5);
    head->next->next->next->prev = head->next->next;

    int target = 7;
    vector<pair<int, int>> pairs = findPairsWithGivenSum(head, target);

    if (pairs.empty()) {
        cout << "No pairs found." << endl;
    }
    else {
        for (auto &pair : pairs) {
            cout << pair.first << " " << pair.second << endl;
        }
    }
    return 0;
}

```

Presum, Two Pointers, Array Misc

Subarray sums divided by K:

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]

```
class Solution {
public:
    int subarraysDivByK(vector<int>& nums, int k) {
        int n = nums.size(), i, j, ans=0;
        unordered_map<int,int> map;
        //for subarrays starting with indeox 0
        map[0]=1;
        int sum=0;
        for(i=0;i<n;i++){
            sum += nums[i];

            int mod = sum % k;
            if(mod<0) mod += k;

            if(map.find(mod)!=map.end()){
                ans += map[mod];
                map[mod] +=1;
            }else{
                map[mod]=1;
            }
        }
        return ans;
    }
};
```

Searching and Sorting:

Find first and last positions of an element in a sorted array

```
Class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int x) {
        int n = nums.size();
        int m, l=0, h=n-1;
        int a=-1, b=-1;
        while(l<=h){
            m = (l+h)/2;
            if(nums[m]==x){
                a=m;
                h=m-1;
            }else if(nums[m]>x){
                h=m-1;
            }else {
                l=m+1;
            }
        }
        l=0, h=n-1;
        while(l<=h){
            m = (l+h)/2;
            if(nums[m]==x){
                b=m;
                l=m+1;
            }else if(nums[m]>x){
                h=m-1;
            }else {
                l=m+1;
            }
        }
        return {a,b};
    };
};
```

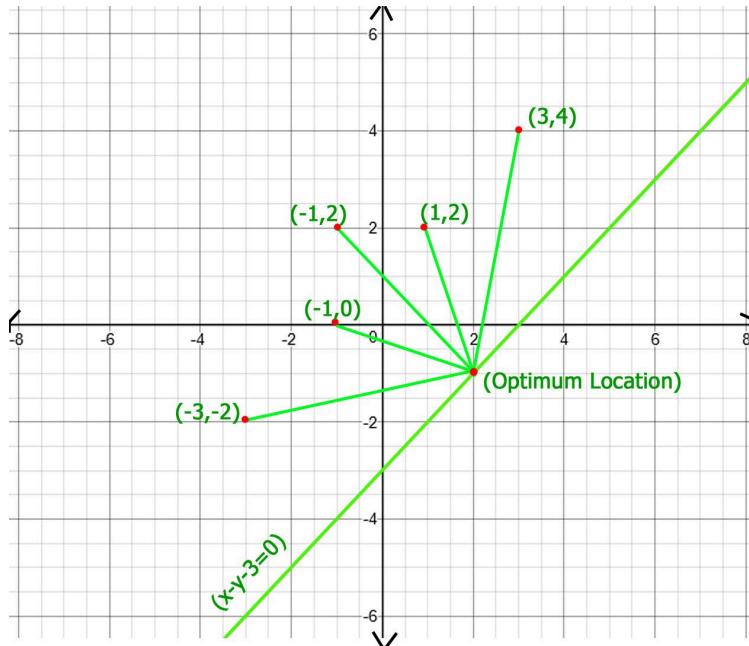
Search in rotated sorted array:

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int ans = -1, l=0, r=nums.size()-1, m;
        while(l<=r){
            m = (l+r)/2;
            if(nums[m]==target) return m;
            if(nums[l]<=nums[m]){
                if(nums[l]<=target and target<=nums[m]) r=m-1;
                else l = m+1;
            }else if(nums[m]<=nums[r]){
                if(nums[m]<=target and target<=nums[r]) l = m+1;
                else r = m -1;
            }
        }
        return ans;
    }
};
```

Find the square root:

```
class Solution {
public:
    int mySqrt(int x) {
        if(!x) return 0;
        long long le=1, r=INT_MAX;
        long long m;
        while(1){
            m = (r+le) / 2 ;
            if(m*m> x){
                r = m-1;
            }else{
                if((m+1)*(m+1) > x) return m;
                le = m+1;
            }
        }
    }
};
```

Optimum location of point to minimize total distance



```
// C/C++ program to find optimum location and total cost
#include <bits/stdc++.h>
using namespace std;
#define sq(x) ((x) * (x))
#define EPS 1e-6
#define N 5

// structure defining a point
struct point {
    int x, y;
    point() {}
    point(int x, int y)
        : x(x)
        , y(y)
    {}
};

// structure defining a line of ax + by + c = 0 form
struct line {
    int a, b, c;
    line(int a, int b, int c)
        : a(a)
        , b(b)
        , c(c)
    {}
};
```

```

    {
}
};

// method to get distance of point (x, y) from point p
double dist(double x, double y, point p)
{
    return sqrt(sq(x - p.x) + sq(y - p.y));
}

/* Utility method to compute total distance all points
   when choose point on given line has x-coordinate
   value as X */
double compute(point p[], int n, line l, double X)
{
    double res = 0;

    // calculating Y of chosen point by line equation
    double Y = -1 * (l.c + l.a * X) / l.b;
    for (int i = 0; i < n; i++)
        res += dist(X, Y, p[i]);

    return res;
}

// Utility method to find minimum total distance
double findOptimumCostUtil(point p[], int n, line l)
{
    double low = -1e6;
    double high = 1e6;

    // loop until difference between low and high
    // become less than EPS
    while ((high - low) > EPS) {
        // mid1 and mid2 are representative x co-ordinates
        // of search space
        double mid1 = low + (high - low) / 3;
        double mid2 = high - (high - low) / 3;

        //
        double dist1 = compute(p, n, l, mid1);
        double dist2 = compute(p, n, l, mid2);

        // if mid2 point gives more total distance,
        // skip third part
        if (dist1 < dist2)
            high = mid2;

        // if mid1 point gives more total distance,
        // skip first part
        else
            low = mid1;
    }
}

```

```

    }

    // compute optimum distance cost by sending average
    // of low and high as X
    return compute(p, n, l, (low + high) / 2);
}

// method to find optimum cost
double findOptimumCost(int points[N][2], line l)
{
    point p[N];

    // converting 2D array input to point array
    for (int i = 0; i < N; i++)
        p[i] = point(points[i][0], points[i][1]);

    return findOptimumCostUtil(p, N, l);
}

// Driver code to test above method
int main()
{
    line l(1, -1, -3);
    int points[N][2] = {
        { -3, -2 }, { -1, 0 }, { -1, 2 }, { 1, 2 }, { 3, 4 }
    };
    cout << findOptimumCost(points, l) << endl;
    return 0;
}

```

Find the repeating and the missing

Calculate the sum of the first size natural numbers. Traverse the array. While traversing, use the absolute value of every element as an index and make the value at this index negative to mark it visited and subtract this value from the missing variable. If something is already marked negative, then this is the repeating element. After traversing, the ‘missing’ variable holds the value of the missing element.

```
#include <bits/stdc++.h>
using namespace std;
```

```
void printTwoElements(vector<int>& arr)
{
    int n = arr.size();
    int missing = (n * (n + 1)) / 2;
    cout << "Repeating ";
```

```

        for (int i = 0; i < n; i++) {
            if (arr[abs(arr[i]) - 1] > 0)
            {
                arr[abs(arr[i]) - 1] = -arr[abs(arr[i]) - 1];
                missing -= abs(arr[i]); // subtract unique elements
            }
            else
                cout << abs(arr[i]) << "\n";
        }

        cout << "Missing " << missing;
    }

    /* Driver code */
    int main()
    {
        vector<int> arr = { 7, 3, 4, 5, 5, 6, 2 };
        printTwoElements(arr);
    }

```

find majority element

Given an array arr. Find the majority element in the array. If no majority exists, return -1.
A majority element in an array is an element that appears strictly more than $\text{arr.size()}/2$ times in the array.

```

// C++ program to find Majority
// element in an array

#include <bits/stdc++.h>
using namespace std;

// Function to find the Majority element in an array using Boyer-Moore
// Voting Algorithm
// It returns -1 if there is no majority element
int majorityElement(const vector<int>& arr) {
    int n = arr.size();
    int candidate = -1;
    int count = 0;

```

```

// Find a candidate
for (int num : arr) {
    if (count == 0) {
        candidate = num;
        count = 1;
    } else if (num == candidate) {
        count++;
    } else {
        count--;
    }
}

// Validate the candidate
count = 0;
for (int num : arr) {
    if (num == candidate) {
        count++;
    }
}

// If count is greater than n / 2, return the candidate; otherwise,
return -1
if (count > n / 2) {
    return candidate;
} else {
    return -1;
}
}

int main() {
    vector<int> arr = {1, 1, 2, 1, 3, 5, 1};
    cout << majorityElement(arr) << endl;
    return 0;
}

```

Searching in an array where adjacent differ by at most k

```

// C++ program to search an element in an array
// where difference between adjacent elements is atmost k
#include<bits/stdc++.h>

```

```

using namespace std;

// x is the element to be searched in arr[0..n-1]
// such that all elements differ by at-most k.
int search(int arr[], int n, int x, int k)
{
    // Traverse the given array starting from
    // leftmost element
    int i = 0;
    while (i < n)
    {
        // If x is found at index i
        if (arr[i] == x)
            return i;

        // Jump the difference between current
        // array element and x divided by k
        // We use max here to make sure that i
        // moves at-least one step ahead.
        i = i + max(1, abs(arr[i]-x)/k);
    }

    cout << "number is not present!";
    return -1;
}

// Driver program to test above function
int main()
{
    int arr[] = {2, 4, 5, 7, 7, 6};
    int x = 6;
    int k = 2;
    int n = sizeof(arr)/sizeof(arr[0]);
    cout << "Element " << x << " is present at index "
        << search(arr, n, x, k);
    return 0;
}

```

find a pair with a given difference

```
// C++ program to find a pair with the given difference
#include <bits/stdc++.h>
using namespace std;

// The function assumes that the array is sorted
bool findPair(int arr[], int size, int n)
{
    unordered_map<int, int> mpp;
    for (int i = 0; i < size; i++) {
        mpp[arr[i]]++;

        // Check if any element whose frequency
        // is greater than 1 exist or not for n == 0
        if (n == 0 && mpp[arr[i]] > 1)
            return true;
    }

    // Check if difference is zero and
    // we are unable to find any duplicate or
    // element whose frequency is greater than 1
    // then no such pair found.
    if (n == 0)
        return false;

    for (int i = 0; i < size; i++) {
        if (mpp.find(n + arr[i]) != mpp.end()) {
            cout << "Pair Found: (" << arr[i] << ", "
                << n + arr[i] << ")";
            return true;
        }
    }

    cout << "No Pair found";
    return false;
}

// Driver program to test above function
int main()
{
    int arr[] = { 1, 8, 30, 40, 100 };
    int size = sizeof(arr) / sizeof(arr[0]);
    int n = -60;
    findPair(arr, size, n);
```

```
    return 0;
}
```

find four elements that sum to a given value (4 Sum)

Sorting and Two Pointer – O(n^3) Time and O(1) Space

1. Sort the array
2. Generate all pairs. For every pair, find the remaining two elements using [two pointer technique](#).

```
// C++ Program to find all Distinct Quadruplets with given
// Sum in an Array using Two Pointer Technique

#include <iostream>
#include <vector>
#include <algorithm>
using namespace std;

// Function to find quadruplets that sum to the target
vector<vector<int>> fourSum(vector<int>& arr, int target) {
    vector<vector<int>> res;
    int n = arr.size();

    // Sort the array
    sort(arr.begin(), arr.end());

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

        // Skip duplicates for i
        if (i > 0 && arr[i] == arr[i - 1]) continue;

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

            // Skip duplicates for j
            if (j > i + 1 && arr[j] == arr[j - 1]) continue;

            int k = j + 1, l = n - 1;

            // Two pointers approach
            while (k < l) {
```

```

        int sum = arr[i] + arr[j] + arr[k] + arr[l];
        if (sum == target) {
            res.push_back({arr[i], arr[j], arr[k], arr[l]});
            k++;
            l--;
            // Skip duplicates for k and l
            while (k < l && arr[k] == arr[k - 1]) k++;
            while (k < l && arr[l] == arr[l + 1]) l--;
        } else if (sum < target) {
            k++;
        } else {
            l--;
        }
    }
}
return res;
}

int main() {
    vector<int> arr = {0, 1, 0, 2, 1, 2, 2};
    int target = 3;
    vector<vector<int>> ans = fourSum(arr, target);
    for (const auto& v : ans) {
        for (int x : v) {
            cout << x << " ";
        }
        cout << endl;
    }
    return 0;
}

```

maximum sum such that no 2 elements are adjacent

[Naive Approach] Using Recursion- O(2^n) Time and O(n) Space

Pick the current element and skip the element just before it.

Skip the current element and move to the element just before it.

So, the recurrence relation will be:

$$\text{maxSumRec}(n) = \max(\text{arr}[n - 1] + \text{maxSumRec}(n - 2),$$

maxSumRec(n - 1)),

where **maxSumRec(n)** returns the maximum sum if n elements are left.

```
// C++ Program to find maximum sum with no two adjacent using Recursion

#include <iostream>
#include <vector>
using namespace std;

// Calculate the maximum Sum value recursively
int maxSumRec(vector<int> &arr, int n) {

    // If no elements are left, return 0.
    if (n <= 0) return 0;

    // If only 1 element is left, pick it.
    if (n == 1) return arr[0];

    // Two Choices: pick the nth element and do not pick the nth element
    int pick = arr[n - 1] + maxSumRec(arr, n - 2);
    int notPick = maxSumRec(arr, n - 1);

    // Return the max of two choices
    return max(pick, notPick);
}

// Function to calculate the maximum Sum value
int maxSum(vector<int>& arr) {
    int n = arr.size();

    // Call the recursive function for n elements
    return maxSumRec(arr, n);
}

int main() {
    vector<int> arr = {6, 7, 1, 3, 8, 2, 4};
    cout << maxSum(arr);
    return 0;
}
```

Expected Approach 1: Using Tabulation DP

```
#include <iostream>
```

```

#include <vector>
using namespace std;

// Function to calculate the maximum Sum value using bottom-up DP
int maxSum(vector<int>& arr) {
    int n = arr.size();

    // Create a dp array to store the maximum sum at each element
    vector<int> dp(n+1, 0);

    // Base cases
    dp[0] = 0;
    dp[1] = arr[0];

    // Fill the dp array using the bottom-up approach
    for (int i = 2; i <= n; i++)
        dp[i] = max(arr[i - 1] + dp[i - 2], dp[i - 1]);

    return dp[n];
}

int main() {
    vector<int> arr = {6, 7, 1, 3, 8, 2, 4};
    cout << maxSum(arr) << endl;
    return 0;
}

```

[Expected Approach 2] Space-Optimized DP – O(n) Time and O(1) Space

```

#include <iostream>
#include <vector>
using namespace std;

// Function to calculate the maximum Sum value
int maxSum(vector<int> &arr) {
    int n = arr.size();

    if (n == 0)
        return 0;
    if (n == 1)
        return arr[0];

```

```

// Set previous 2 values
int secondLast = 0, last = arr[0];

// Compute current value using previous two values
// The final current value would be our result
int res;
for (int i = 1; i < n; i++) {
    res = max(arr[i] + secondLast, last);
    secondLast = last;
    last = res;
}
return res;
}

int main() {
    vector<int> arr = {6, 7, 1, 3, 8, 2, 4};
    cout << maxSum(arr) << endl;
    return 0;
}

```

Count triplet with sum smaller than a given value

- 1) Sort the input array in increasing order.
- 2) Initialize result as 0.
- 3) Run a loop from $i = 0$ to $n-2$. An iteration of this loop finds all triplets with $\text{arr}[i]$ as first element.
 - a) Initialize other two elements as corner elements of subarray $\text{arr}[i+1..n-1]$, i.e., $j = i+1$ and $k = n-1$
 - b) Move j and k toward each other until they meet, i.e., while ($j < k$),
 - (i) If $\text{arr}[i] + \text{arr}[j] + \text{arr}[k] \geq \text{sum}$
then $k--$
// Else for current i and j , there can $(k-j)$ possible third elements
// that satisfy the constraint.
 - (ii) Else Do $\text{ans} += (k - j)$ followed by $j++$

```

// C++ program to count triplets with sum smaller than a given value
#include<bits/stdc++.h>
using namespace std;

int countTriplets(int arr[], int n, int sum)
{

```

```

// Sort input array
sort(arr, arr+n);

// Initialize result
int ans = 0;

// Every iteration of loop counts triplet with
// first element as arr[i].
for (int i = 0; i < n - 2; i++)
{
    // Initialize other two elements as corner elements
    // of subarray arr[j+1..k]
    int j = i + 1, k = n - 1;

    // Use Meet in the Middle concept
    while (j < k)
    {
        // If sum of current triplet is more or equal,
        // move right corner to look for smaller values
        if (arr[i] + arr[j] + arr[k] >= sum)
            k--;

        // Else move left corner
        else
        {
            // This is important. For current i and j, there
            // can be total k-j third elements.
            ans += (k - j);
            j++;
        }
    }
}

return ans;
}

// Driver program
int main()
{
    int arr[] = {5, 1, 3, 4, 7};
    int n = sizeof arr / sizeof arr[0];
    int sum = 12;
    cout << countTriplets(arr, n, sum) << endl;
    return 0;
}

```

```
}
```

Product array Puzzle

Given an array arr[] of n integers, construct a Product Array prod[] (of the same size) such that prod[i] is equal to the product of all the elements of arr[] except arr[i].

Algorithm:

1. Iterate through the array.
2. Find the product of all elements except if it is a 0.
3. If the element is 0 increment the count.
4. Iterate through the array again.
5. If the count of 0's is greater than 2, product of all numbers will be 0.
6. If only one 0 then all the other elements product will be 0, other than that of 0 this will be equal to product of the array.
7. If no zeros are found then product of each element is product divided by itself.

```
#include <iostream>
#include <vector>
using namespace std;

// Function to calculate the product of all
// elements except the current element
vector<int> productExceptSelf(vector<int>& nums, int n) {
    int c = 0;
    int prod = 1;
    vector<int> res(n, 0);

    // Calculate product of all non-zero elements and count zeros
    for (int num : nums) {
        if (num == 0)
            c++;
        else
            prod *= num;
    }

    if (c == 0)
        return res;
    else if (c == 1)
        res[0] = prod;
    else
        for (int i = 0; i < n; i++)
            res[i] = prod / nums[i];
}

int main() {
    vector<int> nums = {1, 2, 3, 4, 5};
    int n = nums.size();
    vector<int> res = productExceptSelf(nums, n);
    for (int num : res)
        cout << num << " ";
}
```

```
// Generate the result array
for (int i = 0; i < n; i++) {
    if (c > 1)
        res[i] = 0;
    else if (c == 1) {
        if (nums[i] == 0)
            res[i] = prod;
        else
            res[i] = 0;
    }
    else {
        res[i] = prod / nums[i];
    }
}

return res;
}

int main() {
    vector<int> nums = {10, 3, 5, 6, 2};
    vector<int> result = productExceptSelf(nums, nums.size());

    for (int val : result) {
        cout << val << " ";
    }
    cout << endl;

    return 0;
}
```

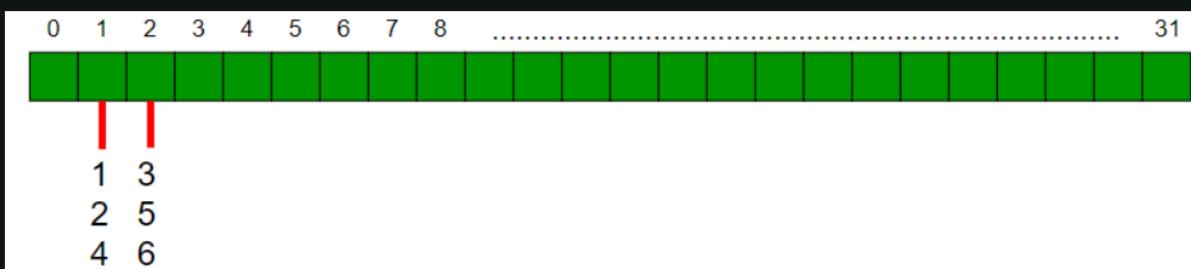
Sort array according to count of set bits

This problem can be solved in $O(n)$ time. The idea is similar to counting sort.

Note: There can be a minimum 1 set-bit and only a maximum of 31set-bits in an integer.

Steps (assuming that an integer takes 32 bits):

1. Create a vector “count” of size 32. Each cell of count i.e., $\text{count}[i]$ is another vector that stores all the elements whose set-bit-count is i
2. Traverse the array and do the following for each element:
 1. Count the number set-bits of this element. Let it be ‘setbitcount’
 2. $\text{count}[\text{setbitcount}].\text{push_back}(\text{element})$
3. Traverse ‘count’ in reverse fashion(as we need to sort in non-increasing order) and modify the array.



```
// C++ program to sort an array according to
// count of set bits using std::sort()
#include <bits/stdc++.h>
using namespace std;

// a utility function that returns total set bits
// count in an integer
int countBits(int a)
{
    int count = 0;
    while (a)
    {
        if (a & 1)
            count+= 1;
        a = a>>1;
    }
    return count;
}

// Function to sort according to bit count
```

```

// This function assumes that there are 32
// bits in an integer.
void sortBySetBitCount(int arr[],int n)
{
    vector<vector<int> > count(32);
    int setbitcount = 0;
    for (int i=0; i<n; i++)
    {
        setbitcount = countBits(arr[i]);
        count[setbitcount].push_back(arr[i]);
    }

    int j = 0; // Used as an index in final sorted array

    // Traverse through all bit counts (Note that we
    // sort array in decreasing order)
    for (int i=31; i>=0; i--)
    {
        vector<int> v1 = count[i];
        for (int i=0; i<v1.size(); i++)
            arr[j++] = v1[i];
    }
}

// Utility function to print an array
void printArr(int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

// Driver Code
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortBySetBitCount(arr, n);
    printArr(arr, n);
    return 0;
}

```

minimum no. of swaps required to sort the array

The minimum number of swaps required to sort an array using a greedy algorithm:

To solve the problem follow the below idea:

While iterating over the array, check the current element, and if not in the correct place, replace that element with the index of the element which should have come to this place greedily which will give the optimal answer

Follow the below steps to solve the problem:

- Create a new array and copy the elements of the input array
- Sort the new array and declare a variable ans equal to 0
- Run a for loop to traverse the elements
 - If the current element in the sorted array is not equal to the one in the input array then increase the ans by 1
 - And swap the current element, with the required element at this index
- Return ans

```
// C++ program to sort an array according to
// count of set bits using std::sort()
#include <bits/stdc++.h>
using namespace std;

// a utility function that returns total set bits
// count in an integer
int countBits(int a)
{
    int count = 0;
    while (a)
    {
        if (a & 1)
            count+= 1;
        a = a>>1;
    }
}
```

```

    }
    return count;
}

// Function to sort according to bit count
// This function assumes that there are 32
// bits in an integer.
void sortBySetBitCount(int arr[],int n)
{
    vector<vector<int> > count(32);
    int setbitcount = 0;
    for (int i=0; i<n; i++)
    {
        setbitcount = countBits(arr[i]);
        count[setbitcount].push_back(arr[i]);
    }

    int j = 0; // Used as an index in final sorted array

    // Traverse through all bit counts (Note that we
    // sort array in decreasing order)
    for (int i=31; i>=0; i--)
    {
        vector<int> v1 = count[i];
        for (int i=0; i<v1.size(); i++)
            arr[j++] = v1[i];
    }
}

// Utility function to print an array
void printArr(int arr[], int n)
{
    for (int i=0; i<n; i++)
        cout << arr[i] << " ";
}

// Driver Code
int main()
{
    int arr[] = {1, 2, 3, 4, 5, 6};
    int n = sizeof(arr)/sizeof(arr[0]);
    sortBySetBitCount(arr, n);
    printArr(arr, n);
}

```

```
    return 0;
}
```

Weighted Job Scheduling in O(n Log n) time

Given n jobs where every job has the following properties:

Start Time

Finish Time

Profit or Value Associated

The task is to find the maximum profit in scheduling jobs such that no two jobs overlap.

```
// C++ program to implement
// Weighted Job Scheduling
#include <bits/stdc++.h>
using namespace std;

// Function to find the closest next job.
int findNextJob(int i, vector<vector<int>> &jobs) {
    int end = jobs[i][1];
    int ans = jobs.size();
    int s = i + 1, e = jobs.size() - 1;
    while (s <= e) {
        int mid = s + (e - s) / 2;
        if (jobs[mid][0] >= end) {
            ans = mid;
            e = mid - 1;
        } else {
            s = mid + 1;
        }
    }
    return ans;
}

int maxProfitRecur(int i, vector<vector<int>> &jobs, vector<int> &memo) {
    if (i == jobs.size())
        return 0;

    // If value is memoized
    if (memo[i] != -1)
```

```

        return memo[i];

    // Find the next closest job if
    // current job is chosen.
    int next = findNextJob(i, jobs);

    // Find maximum profit if current job
    // is chosen.
    int take = jobs[i][2] + maxProfitRecur(next, jobs, memo);

    // Find maximum profit if current
    // job is skipped.
    int noTake = maxProfitRecur(i + 1, jobs, memo);

    return memo[i] = max(take, noTake);
}

int maxProfit(vector<vector<int>> &jobs) {
    int n = jobs.size();

    // Sort the jobs in increasing order
    // of start time and end time.
    sort(jobs.begin(), jobs.end());

    // Array for memoization.
    vector<int> memo(n, -1);
    return maxProfitRecur(0, jobs, memo);
}

int main() {
    vector<vector<int>> jobs =
        {{1, 2, 50}, {3, 5, 20}, {6, 19, 100}, {2, 100, 200}};
    cout << maxProfit(jobs);
    return 0;
}

```

Smallest number with atleastn trailing zeroes infactorial

```

class Solution
{

```

```

public:
    int res(int a){
        int b=0;
        while(a>4 && a%5==0){

            b++;
            a/=5;

        }
        return b;
    }

    int findNum(int n)
    {
        //complete the function here
        int g=0;
        for(int i=5;i<=INT_MAX;i+=5){
            g += res(i);
            if(g>=n) return i;
        }
        return -1;
    }
};

```

The Painter's Partition Problem

Given an array **arr[]** and **k**, where the array represents the **boards** and each element of the given array represents the **length of each board**. **k** numbers of **painters** are available to **paint** these boards. Consider that each unit of a board takes **1 unit of time** to paint. The task is to find the **minimum time** to get this job done by **painting all the boards** under the constraint that any painter will only paint the **continuous sections of boards**. say **board [2, 3, 4]** or only **board [1]** or nothing but **not board [2, 4, 5]**.

Examples:

Input: arr[] = [5, 10, 30, 20, 15], k = 3

Output: 35

Explanation: The most optimal way will be: Painter 1 allocation : [5,10], Painter 2 allocation : [30], Painter 3 allocation : [20, 15], Job will be done when all painters finish i.e. at time = max(5 + 10, 30, 20 + 15) = 35

Input: arr[] = [10, 20, 30, 40], k = 2

Output: 60

Explanation: The most optimal way to paint: Painter 1 allocation : [10, 20, 30], Painter 2 allocation : [40], Job will be complete at time = 60

Naive Approach – Using recursion

A brute force solution is to consider **all possible sets of contiguous partitions** and calculate the **maximum sum** partition in each case and return the **minimum** of all these cases.

In this problem, we need to minimize the maximum time taken by any painter to complete their assigned boards. We have k painters who can work simultaneously. The recursive approach explores every possible way of dividing the boards among the painters and then selects the division that minimizes the maximum time.

The recursive function tries assigning boards from index $curr$ to $last\ index$ to each painter, then recursively computes the minimum time for the remaining boards and painters.

Recurrence Relation:

Let $\text{minTime}(curr, k)$ represent the **minimum time** to paint the boards from $curr$ to the **end** with k remaining painters. The recurrence relation is:

- $\text{minTime}(curr, k) = \min \{\max\{\text{sum}(curr, i) + \text{minTime}(i + 1, k - 1)\}\}$

Where:

$\text{sum}(curr, i)$ is the **total time** for the **current painter** to paint boards from index $curr$ to i where i can range from $curr$ to $n-1$.

$\text{minTime}(i + 1, k - 1)$ is the **recursive call** for the remaining boards and painters.

Base Cases:

No boards left ($curr \geq n$): Return 0, as there's nothing to paint.

No painters left ($k == 0$): Return **infinity** as it's infeasible to paint with no painters.

```
// C++ program to find the minimum time for
// painter's partation problem using recursion.

#include <bits/stdc++.h>
using namespace std;

int minimizeTime(int curr, int n, vector<int> &arr, int k) {

    // If all boards are painted, return 0
    if (curr >= n)
        return 0;

    // If no painters are left
    if (k == 0)
        return INT_MAX;

    // Current workload for this painter
```

```

int currSum = 0;

// Result to store the minimum possible time
int res = INT_MAX;

// Divide the boards among painters starting from curr
for (int i = curr; i < n; i++) {
    currSum += arr[i];

    // Find the maximum time if we assign arr[curr..i] to
    // this painter
    int remTime = minimizeTime(i + 1, n, arr, k - 1);
    int remaining = max(currSum, remTime);

    // Update the result
    res = min(res, remaining);
}

return res;
}

int minTime(vector<int> &arr, int k) {
    int n = arr.size();
    return minimizeTime(0, n, arr, k);
}

int main() {
    vector<int> arr = {10, 10, 10, 10};
    int k = 2;
    int res = minTime(arr, k);
    cout << res << endl;

    return 0;
}

```

Using Memoization – $O(n \cdot n \cdot k)$ Time and $O(n \cdot k)$ Space

If we closely observe the recursive function `minTime()`, it exhibits the property of overlapping subproblems, where the **same subproblem** is solved repeatedly in different recursive paths. This redundancy can be avoided by applying **memoization**. Since the changing parameters in the recursive calls are the **current index** (`curr`) and the number of **remaining painters** (`k`), we can utilize a **2D array** of size $n \times (k+1)$ to store the results. We initialize this array with `-1` to signify that a value has not been computed yet, thus preventing redundant calculations.

```
// C++ program to find the minimum time for
// painter's partition problem using memoization.

#include <bits/stdc++.h>
using namespace std;

int minimizeTime(int curr, int n, vector<int> &arr, int k,
                 vector<vector<int>> &memo) {

    // If all boards are painted, return 0
    if (curr >= n)
        return 0;

    // If no painters are left
    if (k == 0)
        return INT_MAX;

    // Check if the result is already computed and stored in memo table
    // If so, return the stored result to avoid recomputation
    if (memo[curr][k] != -1)
        return memo[curr][k];

    // Current workload for this painter
    int currSum = 0;

    // Result to store the minimum possible time
    int res = INT_MAX;

    // Divide the boards among painters starting from curr
    for (int i = curr; i < n; i++) {
        currSum += arr[i];

        // Find the maximum time if we assign arr[curr..i] to
        // this painter
    }
}
```

```
    int remTime = minimizeTime(i + 1, n, arr, k - 1, memo);
    int remaining = max(currSum, remTime);

    // Update the result
    res = min(res, remaining);
}

// Store the computed result in the memo table
// This helps avoid redundant calculations in future calls
return memo[curr][k] = res;
}

int minTime(vector<int> &arr, int k) {
    int n = arr.size();

    // Initialize memoization table with -1
    // (indicating no result computed yet)
    vector<vector<int>> memo(n, vector<int>(k + 1, -1));
    return minimizeTime(0, n, arr, k, memo);
}

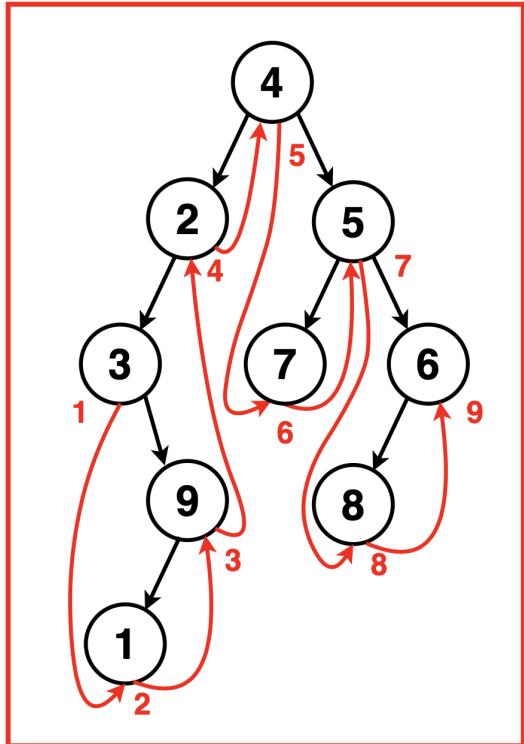
int main() {

    vector<int> arr = {10, 10, 10, 10};
    int k = 2;
    int res = minTime(arr, k);
    cout << res << endl;

    return 0;
}
```

Binary Trees:

Inorder Traversal:

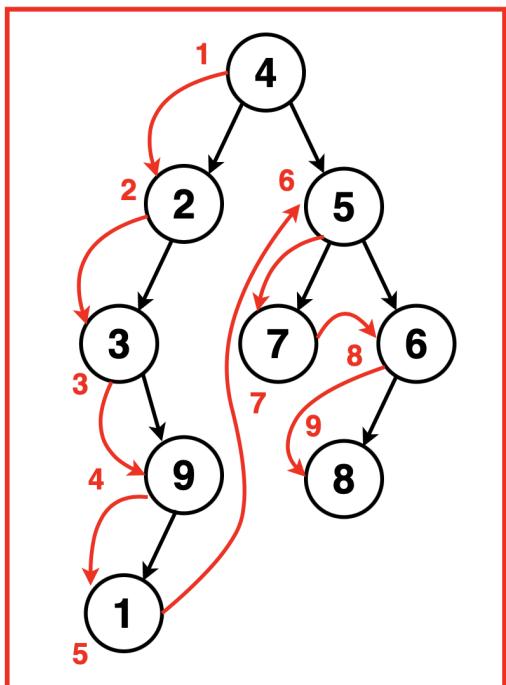


Output: [3, 1, 9, 2, 4, 7, 5, 8, 6]

```
void inorder(Node* root, vector<int> &arr){  
    // If the current node is NULL  
    // (base case for recursion), return  
    if(root == nullptr){  
        return;  
    }  
    // Recursively traverse the left subtree  
    inorder(root->left, arr);  
    // Push the current node's  
    // value into the vector  
    arr.push_back(root->data);  
    // Recursively traverse  
    // the right subtree  
    inorder(root->right, arr);
```

```
}
```

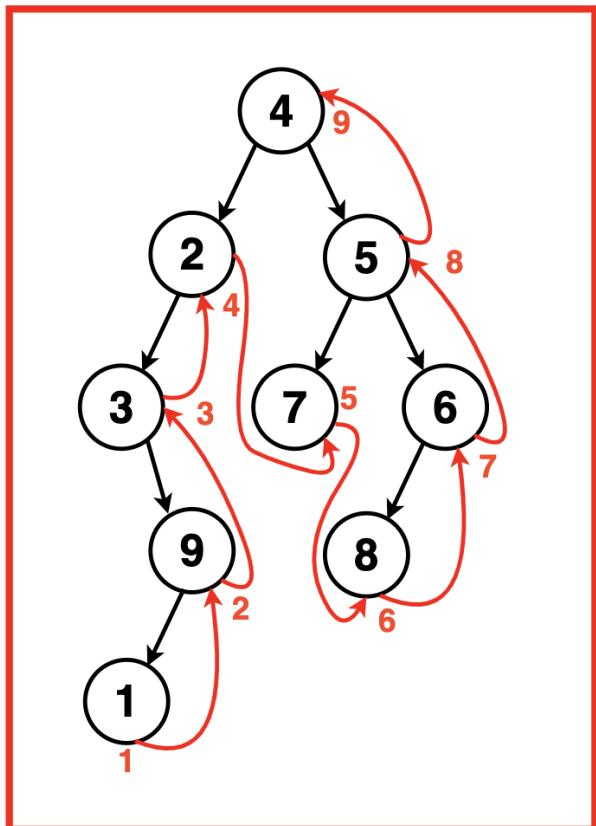
Preorder Traversal:



```
void preorder(Node* root, vector<int> &arr){  
    // If the current node is NULL  
    // (base case for recursion), return  
    if(root == nullptr){  
        return;  
    }  
    // Push the current node's  
    // value into the vector  
    arr.push_back(root->data);  
    // Recursively traverse  
    // the left subtree  
    preorder(root->left, arr);  
    // Recursively traverse  
    // the right subtree
```

```
    preorder(root->right, arr);  
}
```

Post Order Traversal:



```
void postorder(Node* root, vector<int>& arr){  
    // Base case: if root is null, return  
    if(root==NULL){  
        return;  
    }  
    // Traverse left subtree  
    postorder(root->left, arr);  
    // Traverse right subtree  
    postorder(root->right, arr);  
    // Visit the node  
}
```

```

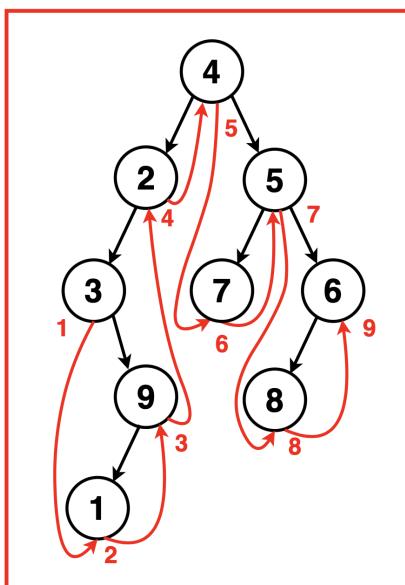
    // (append node's data to the array)
    arr.push_back(root->data);
}

```

Morris Inorder Traversal

Problem Statement: Given a Binary Tree, implement Morris Inorder Traversal and return the array containing its inorder sequence.

Morris Inorder Traversal is a tree traversal algorithm aiming to achieve a space complexity of O(1) without recursion or an external data structure. The algorithm should efficiently visit each node in the binary tree in inorder sequence, printing or processing the node values as it traverses, without using a stack or recursion.



```

// Function to perform iterative Morris
// inorder traversal of a binary tree
vector<int> getInorder(TreeNode* root) {
    // Vector to store the
    // inorder traversal result
    vector<int> inorder;
    // Pointer to the current node,
    // starting from the root

```

```
TreeNode* cur = root;

// Loop until the current
// node is not NULL
while (cur != NULL) {
    // If the current node's
    // left child is NULL
    if (cur->left == NULL) {
        // Add the value of the current
        // node to the inorder vector
        inorder.push_back(cur->val);
        // Move to the right child
        cur = cur->right;
    } else {
        // If the left child is not NULL,
        // find the predecessor (rightmost node
        // in the left subtree)
        TreeNode* prev = cur->left;
        while (prev->right && prev->right != cur) {
            prev = prev->right;
        }

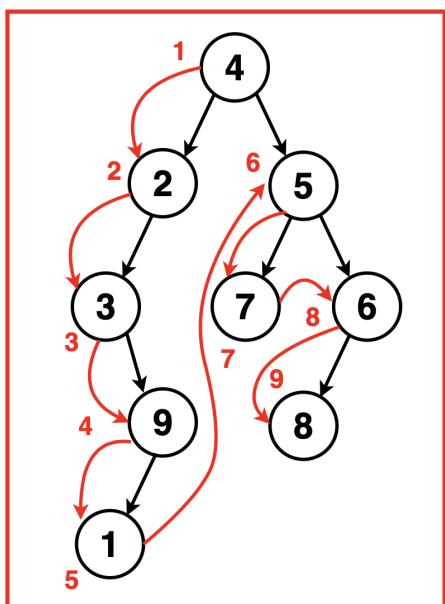
        // If the predecessor's right child
        // is NULL, establish a temporary link
        // and move to the left child
        if (prev->right == NULL) {
            prev->right = cur;
            cur = cur->left;
        } else {
            // If the predecessor's right child
            // is already linked, remove the link,
            // add current node to inorder vector,
            // and move to the right child
            prev->right = NULL;
            inorder.push_back(cur->val);
        }
    }
}
```

```
        cur = cur->right;  
    }  
}  
  
// Return the inorder  
// traversal result  
return inorder;  
}
```

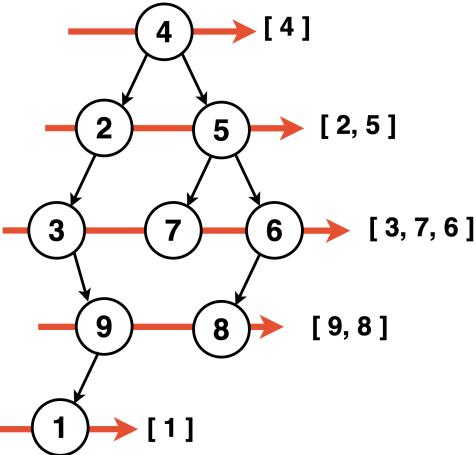
Morris Preorder Traversal:

Problem Statement: Given a Binary Tree, implement Morris Preorder Traversal and return the array containing its preorder sequence.

Morris Preorder Traversal is a tree traversal algorithm aiming to achieve a space complexity of $O(1)$ without recursion or an external data structure. The algorithm should efficiently visit each node in the binary tree in preorder sequence, printing or processing the node values as it traverses, without using a stack or recursion.



Level Order Traversal of BT:



level order traversal : [[4], [2, 5], [3, 7, 6], [9, 8], [1]]

```
vector<vector<int>> levelOrder(TreeNode* root) {
    // Create a 2D vector to store levels
    vector<vector<int>> ans;
    if (root == nullptr) {
        // If the tree is empty,
        // return an empty vector
        return ans;
    }

    // Create a queue to store nodes
    // for level-order traversal
    queue<TreeNode*> q;
    // Push the root node to the queue
    q.push(root);

    while (!q.empty()) {
        // Get the size of the current level
```

```
int size = q.size();
// Create a vector to store
// nodes at the current level
vector<int> level;

for (int i = 0; i < size; i++) {
    // Get the front node in the queue
    TreeNode* node = q.front();
    // Remove the front node from the queue
    q.pop();
    // Store the node value
    // in the current level vector
    level.push_back(node->val);

    // Enqueue the child nodes if they exist
    if (node->left != nullptr) {
        q.push(node->left);
    }
    if (node->right != nullptr) {
        q.push(node->right);
    }
}
// Store the current level
// in the answer vector
ans.push_back(level);
}

// Return the level-order
// traversal of the tree
return ans;
}
```

Height of Binary Tree:

```

int maxDepth(Node* root){
    // If the root is NULL
    // (empty tree), depth is 0
    if(root == NULL){
        return 0;
    }

    // Recursive call to find the
    // maximum depth of the left subtree
    int lh = maxDepth(root->left);

    // Recursive call to find the
    // maximum depth of the right subtree
    int rh = maxDepth(root->right);

    // Return the maximum depth of the
    // tree, adding 1 for the current node
    return 1 + max(lh, rh);
}

```

Left View of Binary Tree

```

class Solution {
public:
    // Function to return the Right view of the binary tree
    vector<int> rightsideView(Node* root){
        // Vector to store the result
        vector<int> res;

        // Call the recursive function
        // to populate the right-side view
        recursionRight(root, 0, res);
    }
};

```

```
    return res;
}

// Function to return the Left view of the binary tree
vector<int> leftsideView(Node* root){
    // Vector to store the result
    vector<int> res;

    // Call the recursive function
    // to populate the left-side view
    recursionLeft(root, 0, res);

    return res;
}

private:
    // Recursive function to traverse the
    // binary tree and populate the left-side view
    void recursionLeft(Node* root, int level, vector<int>&
res){
        // Check if the current node is NULL
        if(root == NULL){
            return;
        }

        // Check if the size of the result vector
        // is equal to the current level
        if(res.size() == level){
            // If equal, add the value of the
            // current node to the result vector
            res.push_back(root->data);
        }

        // Recursively call the function for the
        // left child of the current node
        recursionLeft(root->left, level + 1, res);
    }
}
```

```
// left child with an increased level
recursionLeft(root->left, level + 1, res);

// Recursively call the function for the
// right child with an increased level
recursionLeft(root->right, level + 1, res);
}

// Recursive function to traverse the
// binary tree and populate the right-side view
void recursionRight(Node* root, int level, vector<int>
&res){
    // Check if the current node is NULL
    if(root == NULL){
        return;
    }

    // Check if the size of the result vector
    // is equal to the current level
    if(res.size() == level){
        // If equal, add the value of the
        // current node to the result vector
        res.push_back(root->data);

        // Recursively call the function for the
        // right child with an increased level
        recursionRight(root->right, level + 1, res);

        // Recursively call the function for the
        // left child with an increased level
        recursionRight(root->left, level + 1, res);
    }
}
};
```

Bottom View of the Binary Tree:

```
vector<int> bottomView(Node* root){
    // Vector to store the result
    vector<int> ans;

    // Check if the tree is empty
    if(root == NULL){
        return ans;
    }

    // Map to store the bottom view nodes
    // based on their vertical positions
    map<int, int> mpp;

    // Queue for BFS traversal, each
    // element is a pair containing node
    // and its vertical position
    queue<pair<Node*, int>> q;

    // Push the root node with its vertical
    // position (0) into the queue
    q.push({root, 0});

    // BFS traversal
    while(!q.empty()){
        // Retrieve the node and its vertical
        // position from the front of the queue
        auto it = q.front();
        q.pop();
        Node* node = it.first;
        int line = it.second;
```

```

        // Update the map with the node's data
        // for the current vertical position
        mpp[line] = node->data;

        // Process left child
        if(node->left != NULL){
            // Push the left child with a decreased
            // vertical position into the queue
            q.push({node->left, line - 1});
        }

        // Process right child
        if(node->right != NULL){
            // Push the right child with an increased
            // vertical position into the queue
            q.push({node->right, line + 1});
        }
    }

    // Transfer values from the
    // map to the result vector
    for(auto it : mpp){
        ans.push_back(it.second);
    }

    return ans;
}

```

Top View of the Binary Tree:

```

vector<int> topView(Node* root){
    // Vector to store the result
    vector<int> ans;

```

```

// Check if the tree is empty
if(root == NULL){
    return ans;
}

// Map to store the top view nodes
// based on their vertical positions
map<int, int> mpp;

// Queue for BFS traversal, each element
// is a pair containing node
// and its vertical position
queue<pair<Node*, int>> q;

// Push the root node with its vertical
// position (0) into the queue
q.push({root, 0});

// BFS traversal
while(!q.empty()){
    // Retrieve the node and its vertical
    // position from the front of the queue
    auto it = q.front();
    q.pop();
    Node* node = it.first;
    int line = it.second;

    // If the vertical position is not already
    // in the map, add the node's data to the map
    if(mpp.find(line) == mpp.end()){
        mpp[line] = node->data;
    }

    // Process left child
}

```

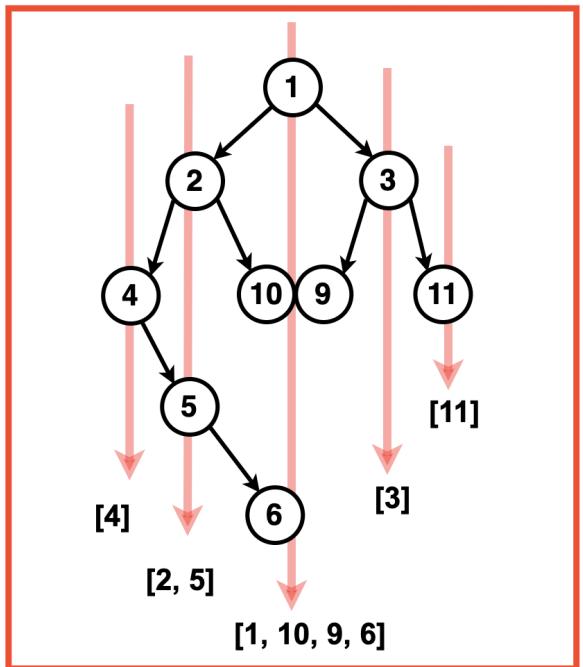
```
if(node->left != NULL){
    // Push the left child with a decreased
    // vertical position into the queue
    q.push({node->left, line - 1});
}

// Process right child
if(node->right != NULL){
    // Push the right child with an increased
    // vertical position into the queue
    q.push({node->right, line + 1});
}

// Transfer values from the
// map to the result vector
for(auto it : mpp){
    ans.push_back(it.second);
}

return ans;
}
```

Vertical Order Traversal:



```
vector<vector<int>> findVertical(Node* root){  
    // Map to store nodes based on  
    // vertical and level information  
    map<int, map<int, multiset<int>>> nodes;  
  
    // Queue for BFS traversal, each  
    // element is a pair containing node  
    // and its vertical and level information  
    queue<pair<Node*, pair<int, int>>> todo;  
  
    // Push the root node with initial vertical  
    // and level values (0, 0)  
    todo.push({root, {0, 0}});  
  
    // BFS traversal
```

```

while(!todo.empty()){
    // Retrieve the node and its vertical
    // and level information from
    // the front of the queue
    auto p = todo.front();
    todo.pop();
    Node* temp = p.first;

    // Extract the vertical and level information
    // x -> vertical
    int x = p.second.first;
    // y -> level
    int y = p.second.second;

    // Insert the node value into the
    // corresponding vertical and level
    // in the map
    nodes[x][y].insert(temp->data);

    // Process left child
    if(temp->left){
        todo.push({
            temp->left,
            {
                // Move left in
                // terms of vertical
                x-1,
                // Move down in
                // terms of level
                y+1
            }
        });
    }

    // Process right child
}

```

```

        if(temp->right){
            todo.push({
                temp->right,
                {
                    // Move right in
                    // terms of vertical
                    x+1,
                    // Move down in
                    // terms of level
                    y+1
                }
            });
        }

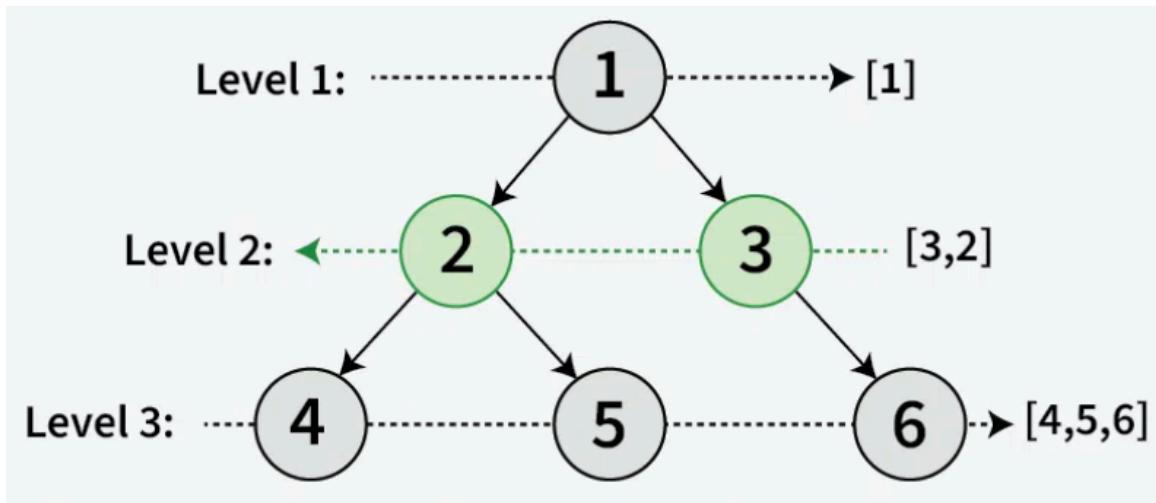
        // Prepare the final result vector
        // by combining values from the map
        vector<vector<int>> ans;
        for(auto p: nodes){
            vector<int> col;
            for(auto q: p.second){
                // Insert node values
                // into the column vector
                col.insert(col.end(), q.second.begin(),
q.second.end());
            }
            // Add the column vector
            // to the final result
            ans.push_back(col);
        }
        return ans;
    }
}

```

Serialize and Deserialize a Binary Tree

<https://www.geeksforgeeks.org/serialize-deserialize-binary-tree/>

ZigZag Tree Traversal of a Binary Tree



Using two stacks – O(n) Time and O(n) Space

The idea is to use two stacks. One stack will be used to traverse the current level and the other stack will be used to store the nodes of next level.

Step by step implementation:

- Initialize two stacks, say **currentLevel** and **nextLevel**. Push the **root** node into the **currentLevel**. Initialize a boolean value **leftToRight** which will indicate the direction of traversal (initially set to **true**).
- Follow until **currentLevel** stack is not empty:
 - Initialize a variable, say **size** which represents the size of current level. Start traversing till **size != 0**.
 - Pop from the **currentLevel** stack and push the nodes value into the **ans**.
 - If current level order is from **left to right**, then **push** the left child node into **nextLevel** stack. Then push the **right child** node.(stack is **last-in first-out** structure, so in next level traversal, right child node will be popped first).
 - If the current level is from **right to left**, then push the right child first. Then push the left child.
 - swap the **currentLevel** and **nextLevel** stack. Also, flip the value of **leftToRight**.
- return **ans** vector whose elements follow ZigZag Tree Traversal.

```
// function to print the zigzag traversal
vector<int> zigZagTraversal(Node* root) {

    vector<int> ans;

    // if null then return
    if (root == nullptr)
        return ans;

    // declare two stacks
    stack<Node*> currentLevel;
    stack<Node*> nextLevel;

    // push the root
    currentLevel.push(root);

    // check if stack is empty
    bool leftToRight = true;
```

```

while (!currentLevel.empty()) {

    int size = currentLevel.size();

    while (size--) {

        // pop out of stack
        struct Node* curr = currentLevel.top();
        currentLevel.pop();

        // push the current node
        ans.push_back(curr->data);

        // store data according to current
        // order.
        if (leftToRight) {
            if (curr->left)
                nextLevel.push(curr->left);
            if (curr->right)
                nextLevel.push(curr->right);
        }
        else {
            if (curr->right)
                nextLevel.push(curr->right);
            if (curr->left)
                nextLevel.push(curr->left);
        }
    }

    leftToRight = !leftToRight;
    swap(currentLevel, nextLevel);
}

return ans;
}

```

Using recursion – O(n) Time and O(n) Space

The idea is to traverse the binary tree using preorder traversal. For each level, push the node values into a 2d array (where rows in array will represent the level and node values corresponding to level as the column). Push the level order array into the resultant array one by one by checking the odd and even indexed levels. For odd indexed levels, push the nodes normally. For even indexed levels, push the nodes in reverse order.

```
// Function to calculate height of tree
int treeHeight(Node *root) {
    if (!root)
        return 0;
    int lHeight = treeHeight(root->left);
    int rHeight = treeHeight(root->right);
    return max(lHeight, rHeight) + 1;
}

// Function to store the zig zag order traversal
// of tree in a list recursively
void zigZagTraversalRecursion
(Node *root, int height, vector<vector<int>> &level) {

    if (root == nullptr)
        return;

    level[height].push_back(root->data);

    zigZagTraversalRecursion(root->left, height + 1, level);
    zigZagTraversalRecursion(root->right, height + 1, level);
}

// Function to traverse tree in zig zag order
vector<int> zigZagTraversal(Node *root) {
    int height = treeHeight(root);

    // This array will store tree
    // nodes at their corresponding levels
    vector<vector<int>> level(height);
    vector<int> ans;

    zigZagTraversalRecursion(root, 0, level);

    // Push each level array into answer
    for (int i = 0; i < height; i++) {

        // For odd levels,
```

```

        // push from left to right
        if (i % 2 == 0) {
            for (int j = 0; j < level[i].size(); j++)
                ans.push_back(level[i][j]);
        }

        // For even levels,
        // push from right to left
        else {
            for (int j = level[i].size() - 1; j >= 0; j--)
                ans.push_back(level[i][j]);
        }
    }

    return ans;
}

```

Using Using Deque – O(n) Time and O(n) Space Deque – O(n) Time and O(n) Space

```

vector<int> zigZagTraversal(Node* root) {

    // base case
    if (root == nullptr)
        return {};

    deque<Node*> dq;
    dq.push_front(root);

    vector<int> ans;

    bool leftToRight = true;

    while (!dq.empty()) {
        int size = dq.size();

        // Traverse in level order
        while (size--) {
            Node* curr;

            // If direction is from left
            // to right, pop from front
            if (leftToRight) {
                curr = dq.front();

```

```

        dq.pop_front();
    }

    // else pop from back
    else {
        curr = dq.back();
        dq.pop_back();
    }

    // Push current node to result
    ans.push_back(curr->data);

    // If direction is from left to
    // right, push the left node and
    // right node into the back of dq.
    if (leftToRight) {
        if (curr->left != nullptr)
            dq.push_back(curr->left);
        if (curr->right != nullptr)
            dq.push_back(curr->right);
    }

    // else push the right node and left
    // node into the front of the dq.
    else {
        if (curr->right != nullptr)
            dq.push_front(curr->right);
        if (curr->left != nullptr)
            dq.push_front(curr->left);
    }
}

// Flip the direction
leftToRight = !leftToRight;
}

return ans;
}

```

Check if a tree is balanced or not

[Naive Approach] Using Top Down Recursion – O(n^2) Time and O(h) Space

```
bool isBalanced(Node* root) {

    // If tree is empty then return true
    if (root == NULL)
        return true;

    // Get the height of left and right sub trees
    int lHeight = height(root->left);
    int rHeight = height(root->right);

    // If absolute height difference is greater than 1
    // Then return false
    if (abs(lHeight - rHeight) > 1)
        return false;

    // Recursively check the left and right subtrees
    return isBalanced(root->left) && isBalanced(root->right);
}
```

[Expected Approach] Using Bottom Up Recursion – O(n) Time and O(h) Space

```
# Definition for a binary tree node.
# class TreeNode:
#     def __init__(self, val=0, left=None, right=None):
#         self.val = val
#         self.left = left
#         self.right = right

class Solution:
    def isBalanced(self, root: Optional[TreeNode]) -> bool:
        if not root:
            return True
        return True if self.check(root) > 0 else False

    def check(self, root):

        if not root:
            return 0

        lh = self.check(root.left)
        rh = self.check(root.right)
```

```

if lh === -1 or rh === -1 or abs(lh - rh) > 1:
    return -1

return max(lh, rh) + 1

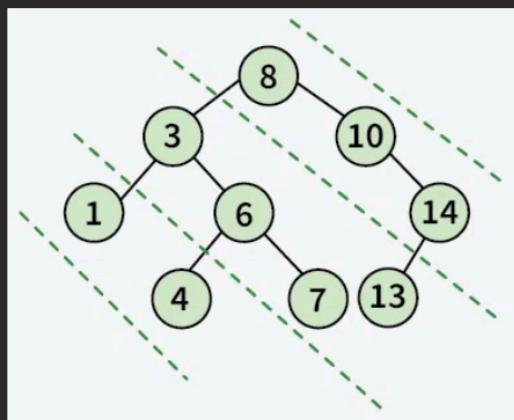
```

Diagonal Traversal of a Binary tree

Given a Binary Tree, the task is to print the diagonal traversal of the binary tree.

Note: If the diagonal element are present in two different subtrees, then left subtree diagonal element should be taken first and then right subtree.

Input:



Output: 8 10 14 3 6 7 13 1 4

Explanation: The above is the diagonal elements in a binary tree that belong to the same line.

```

// Recursive function to print diagonal view
void diagonalRecur(Node *root, int level,
                    unordered_map<int, vector<int>> &levelData) {

    // Base case
    if (root == nullptr)
        return;

```

```

// Append the current node into hash map.
levelData[level].push_back(root->data);

// Recursively traverse the left subtree.
diagonalRecur(root->left, level + 1, levelData);

// Recursively traverse the right subtree.
diagonalRecur(root->right, level, levelData);
}

// function to print diagonal view
vector<int> diagonal(Node *root) {
    vector<int> ans;

    // Create a hash map to store each
    // node at its respective level.
    unordered_map<int, vector<int>> levelData;
    diagonalRecur(root, 0, levelData);

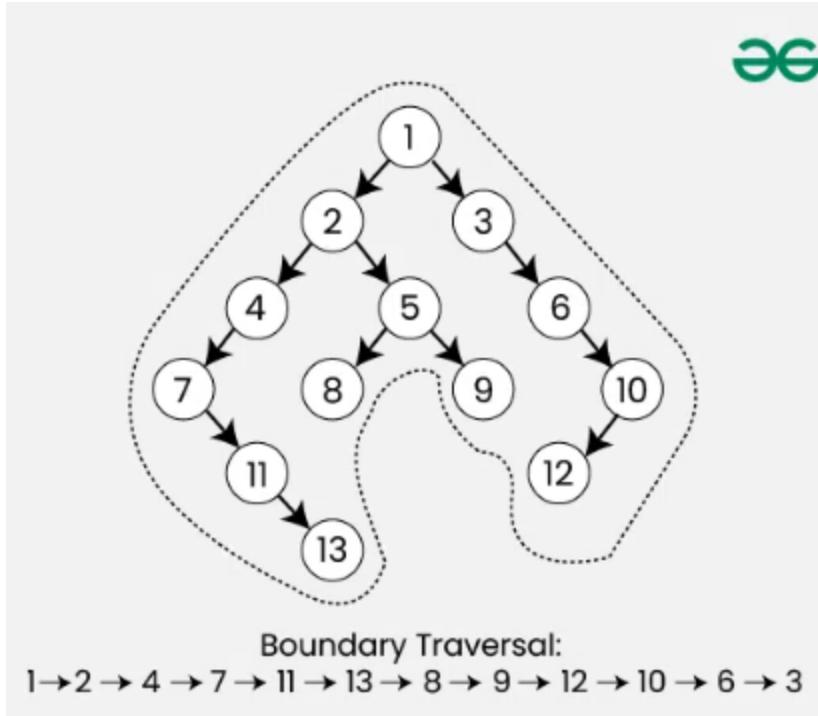
    int level = 0;

    // Insert into answer level by level.
    while (levelData.find(level) != levelData.end()) {
        vector<int> v = levelData[level];
        for (int j = 0; j < v.size(); j++) {
            ans.push_back(v[j]);
        }
        level++;
    }

    return ans;
}

```

Boundary traversal of a Binary tree



[Approach – 1] Using Recursion – O(n) Time and O(h) Space

The idea is to traverse the boundary of the binary tree in three parts:

Collect Left Boundary Nodes: Collects all nodes from the root's left child, excluding leaf nodes, until a leaf is reached.

Collect Leaf Nodes: Using recursion traverse the tree and collect all leaf nodes in the result.

Collect Right Boundary Nodes: Collects all nodes from the root's right child, excluding leaf nodes, in reverse order.

By combining these parts, we achieve the desired boundary traversal. Each part is collected using recursive functions for left boundary, leaf nodes, and right boundary traversal.

```

bool isLeaf(Node *node) {
    return node->left == nullptr && node->right == nullptr;
}

// Function to collect left boundary nodes
// (top-down order)
void collectBoundaryLeft(Node* root, vector<int>& res) {
    // exclude leaf node
    if (root == nullptr || isLeaf(root))
        return;

    res.push_back(root->data);
}

```

```

    if (root->left)
        collectBoundaryLeft(root->left, res);

    else if (root->right)
        collectBoundaryLeft(root->right, res);
}

// Function to collect all leaf nodes
void collectLeaves(Node *root, vector<int> &res) {
    if (root == nullptr)
        return;

    // Add leaf nodes
    if (isLeaf(root)) {
        res.push_back(root->data);
        return;
    }

    collectLeaves(root->left, res);
    collectLeaves(root->right, res);
}

// Function to collect right boundary nodes
// (bottom-up order)
void collectBoundaryRight(Node* root, vector<int>& res) {
    // exclude leaf nodes
    if (root == nullptr || isLeaf(root))
        return;

    if (root->right)
        collectBoundaryRight(root->right, res);

    else if (root->left)
        collectBoundaryRight(root->left, res);

    res.push_back(root->data);
}

// Function to find Boundary Traversal of Binary Tree
vector<int> boundaryTraversal(Node *root) {
    vector<int> res;

    if (!root)
        return res;

    collectBoundaryLeft(root, res);
    collectLeaves(root, res);
    collectBoundaryRight(root, res);
}

```

```

// Add root data if it's not a leaf
if (!isLeaf(root))
    res.push_back(root->data);

// Collect left boundary
collectBoundaryLeft(root->left, res);

// Collect leaf nodes
collectLeaves(root, res);

// Collect right boundary
collectBoundaryRight(root->right, res);

return res;
}

```

[Approach – 2] Using Iteration and Morris Traversal – O(n) Time and O(1) Space

```

bool isLeaf(Node *node) {
    return node->left == nullptr && node->right == nullptr;
}

// Function to collect the left boundary nodes
void collectBoundaryLeft(Node *root, vector<int> &res) {
    if (root == nullptr)
        return;

    Node *curr = root;
    while (!isLeaf(curr)) {
        res.push_back(curr->data);

        if (curr->left)
            curr = curr->left;
        else
            curr = curr->right;
    }
}

// Function to collect the leaf nodes using Morris Traversal
void collectLeaves(Node* root, vector<int>& res) {
    Node* current = root;

```

```

while (current) {
    if (current->left == nullptr) {

        // If it's a leaf node
        if (current->right == nullptr)
            res.push_back(current->data);

        current = current->right;
    }
    else {

        // Find the inorder predecessor
        Node* predecessor = current->left;
        while (predecessor->right && predecessor->right != current) {
            predecessor = predecessor->right;
        }

        if (predecessor->right == nullptr) {
            predecessor->right = current;
            current = current->left;
        }
        else {
            // If it's predecessor is a leaf node
            if (predecessor->left == nullptr)
                res.push_back(predecessor->data);

            predecessor->right = nullptr;
            current = current->right;
        }
    }
}

// Function to collect the right boundary nodes
void collectBoundaryRight(Node *root, vector<int> &res) {
    if (root == nullptr)
        return;

    Node *curr = root;
    vector<int> temp;
    while (!isLeaf(curr)) {

        temp.push_back(curr->data);

```

```

        if (curr->right)
            curr = curr->right;
        else
            curr = curr->left;
    }
    for (int i = temp.size() - 1; i >= 0; --i)
        res.push_back(temp[i]);
}

// Function to perform boundary traversal
vector<int> boundaryTraversal(Node *root) {
    vector<int> res;

    if (!root)
        return res;

    // Add root data if it's not a leaf
    if (!isLeaf(root))
        res.push_back(root->data);

    // Collect left boundary
    collectBoundaryLeft(root->left, res);

    // Collect leaf nodes
    collectLeaves(root, res);

    // Collect right boundary
    collectBoundaryRight(root->right, res);

    return res;
}

```

Construct Binary Tree from String with Bracket Representation

[Naive Approach] Using Recursion – O(n^2) time and O(n) space:

The idea is to recursively parse the string by first extracting the root value (which could be multiple digits) and then finding matching pairs of parentheses that enclose the left and right subtrees.

For each subtree enclosed in parentheses, we find its corresponding closing parenthesis using a counter (incrementing for '(' and decrementing for ')'), which helps us identify the correct substring for the left child.

Once we have the left subtree's closing index, we can process the right subtree starting two positions after (to skip the closing parenthesis and opening parenthesis of right subtree).

This process continues recursively until we either encounter an empty substring (base case) or process all characters, effectively building the tree from root to leaves.

```
// function to return the index of close parenthesis
int findIndex(string str, int i, int j) {
    if (i > j)
        return -1;

    int cnt = 0;

    for (int k=i; k<=j; k++) {
        if (str[k]=='(') cnt++;
        else if (str[k]==')') {
            cnt--;
            if (cnt == 0) return k;
        }
    }

    return -1;
}

// function to construct tree from string
Node* constructTreeRecur(string str, int i, int j) {

    // Base case
    if (i > j)
        return nullptr;

    int val = 0;

    // In case the value is having more than 1 digit
    while(i <= j && str[i] >= '0' && str[i] <= '9') {
        val *= 10;
        val += (str[i] - '0');
        i++;
    }

    // new root
    Node* root = new Node(val);
```

```

int index = -1;

// if next char is '(' find the index of
// its complement ')'
if (i <= j && str[i] == '(')
    index = findIndex(str, i, j);

// if index found
if (index != -1) {

    // call for left subtree
    root->left = constructTreeRecur(str, i + 1, index - 1);

    // call for right subtree
    root->right
        = constructTreeRecur(str, index + 2, j - 1);
}

return root;
}

Node* treeFromString(string str) {
    Node* root = constructTreeRecur(str, 0, str.length() - 1);
    return root;
}

```

[Expected Approach] Using Pre-Order Traversal – O(n) time and O(n) space

The idea is to traverse the string sequentially using a single index passed by reference, constructing the tree in a preorder manner (root-left-right) where we first extract the root value by parsing consecutive digits, then recursively build the left subtree if we encounter an opening parenthesis, and finally build the right subtree if another opening parenthesis exists after completing the left subtree construction.

Step by step approach:

- Maintain a single index that traverses through the string from left to right.
- When encountering digits, parse them to form the complete node value and create a new node with the parsed value.
- If an opening parenthesis '(' is found, increment index and recursively build left subtree. After left subtree is built and index is incremented past closing parenthesis, check for another opening parenthesis.
- If second opening parenthesis exists, increment index and recursively build right subtree.
- Return the constructed node which serves as root for its subtree

```

// Recursive function to construct
// tree using preorder traversal.
Node* preOrder(int &i, string &s) {

    // If substring is empty, return null.
    if (s[i]==')') return nullptr;

    // Find the value of root node.
    int val = 0;
    while (i<s.length() && s[i]!='(' && s[i]!=')') {
        int digit = s[i]-'0';
        i++;
        val = val*10+digit;
    }

    // Create the root node.
    Node* root = new Node(val);

    // If left subtree exists
    if (i<s.length() && s[i]=='(') {
        i++;
        root->left = preOrder(i, s);
        i++;
    }

    // If right subtree exists
    if (i<s.length() && s[i]==')') {
        i++;
        root->right = preOrder(i, s);
        i++;
    }

    // Return the root node.
    return root;
}

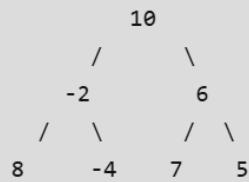
// function to construct tree from string
Node *treeFromString(string s){

    int i = 0;
    return preOrder(i, s);
}

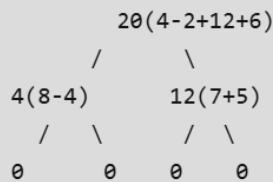
```

Convert Binary tree into Sum tree

For example, the following tree



should be changed to



```
// Convert a given tree to a tree where
// every node contains sum of values of
// nodes in left and right subtrees in the original tree
int toSumTree(node *Node)
{
    // Base case
    if(Node == NULL)
        return 0;

    // Store the old value
    int old_val = Node->data;

    // Recursively call for left and
    // right subtrees and store the sum as
    // old value of this node
    Node->data = toSumTree(Node->left) + toSumTree(Node->right);

    // Return the sum of values of nodes
    // in left and right subtrees and
    // old_value of this node
    return Node->data + old_val;
}
```

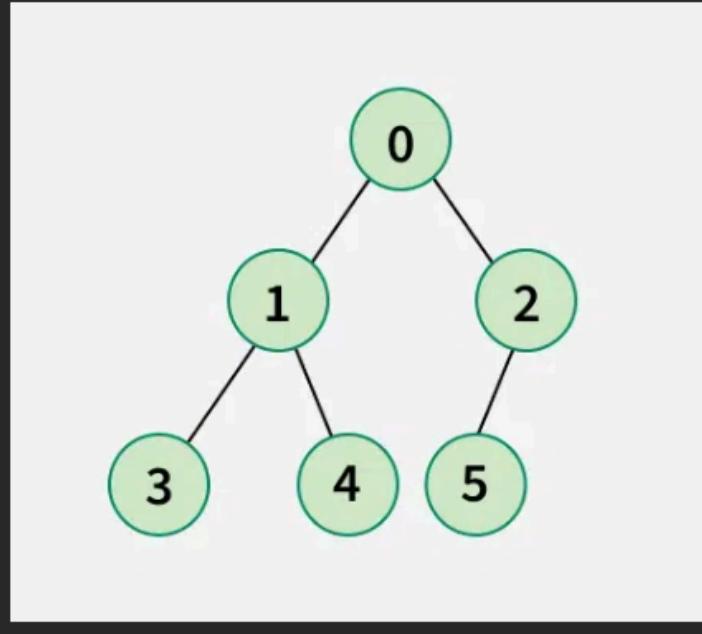
```
// Convert a given tree to a tree where
// every node contains sum of values of
// nodes in left and right subtrees in the original tree
int toSumTree(node* Node)
{
    //check for the base condition
    if (Node != NULL) {
        //recurse the left subtree
        int l = toSumTree(Node->left);
        //recurse the right subtree
        int r = toSumTree(Node->right);
        //storing the temp value of root value
        int temp = Node->data;
        //update the root node
        Node->data = l + r;
        //return the recurse value
        return temp + l + r;
    }
    else
        return 0;
}
```

Construct Tree from given Inorder and Preorder traversals

Input: $\text{inorder}[] = [3, 1, 4, 0, 5, 2]$, $\text{preorder}[] = [0, 1, 3, 4, 2, 5]$

Output: $[0, 1, 2, 3, 4, 5]$

Explanation: The tree will look like:



[Naive Approach] Using Pre-order traversal – $O(n^2)$ Time and $O(h)$ Space

```
// Function to find the index of an element in the array.
int search(vector<int> &inorder, int value, int left, int right) {

    for (int i = left; i <= right; i++) {
        if (inorder[i] == value)
            return i;
    }
    return -1;
}

// Recursive function to build the binary tree.
Node *buildTreeRecur(vector<int> &inorder, vector<int> &preorder,
                     int &preIndex, int left, int right) {

    // For empty inorder array, return null
    if (left > right)
        return nullptr;

    int rootValue = preorder[preIndex];
    int rootIndex = search(inorder, rootValue, left, right);
    preIndex++;

    Node *root = new Node(rootValue);

    root->left = buildTreeRecur(inorder, preorder, preIndex, left, rootIndex - 1);
    root->right = buildTreeRecur(inorder, preorder, preIndex, rootIndex + 1, right);

    return root;
}
```

```

        int rootVal = preorder[preIndex];
        preIndex++;

        // create the root Node
        Node *root = new Node(rootVal);

        // find the index of Root element in the in-order array.
        int index = search(inorder, rootVal, left, right);

        // Recursively create the left and right subtree.
        root->left = buildTreeRecur(inorder, preorder, preIndex, left, index - 1);
        root->right = buildTreeRecur(inorder, preorder, preIndex, index + 1,
                                      right);

        return root;
    }

    // Function to construct tree from its inorder and preorder traversals
    Node *buildTree(vector<int> &inorder, vector<int> &preorder) {

        int preIndex = 0;
        Node *root = buildTreeRecur(inorder, preorder, preIndex, 0, preorder.size() - 1);

        return root;
    }
}

```

[Expected Approach] Using Pre-order traversal and Hash map – O(n) Time and O(n) Space

The idea is similar to first approach, but instead of linearly searching the in-order array for each node we can use hashing. Map the values of in-order array to its indices. This will reduce the searching complexity from O(n) to O(1).

```

// Recursive function to build the binary tree.
Node *buildTreeRecur(unordered_map<int,int> &mp, vector<int> &preorder,
                     int &preIndex, int left, int right) {

    // For empty inorder array, return null
    if (left > right)
        return nullptr;

    int rootVal = preorder[preIndex];

```

```
preIndex++;

// create the root Node
Node *root = new Node(rootVal);

// find the index of Root element in the in-order array.
int index = mp[rootVal];

// Recursively create the left and right subtree.
root->left = buildTreeRecur(mp, preorder, preIndex, left, index - 1);
root->right = buildTreeRecur(mp, preorder, preIndex, index + 1, right);

return root;
}

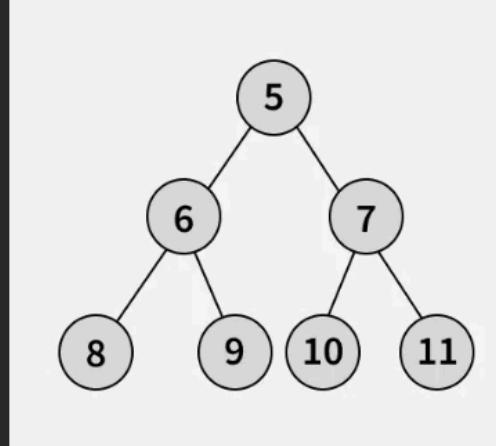
// Function to construct tree from its inorder and preorder traversals
Node *buildTree(vector<int> &inorder, vector<int> &preorder) {

    // Hash map that stores index of a root element in inorder array
    unordered_map<int,int> mp;
    for (int i = 0; i < inorder.size(); i++)
        mp[inorder[i]] = i;

    int preIndex = 0;
    Node *root = buildTreeRecur(mp, preorder, preIndex, 0, inorder.size() - 1);

    return root;
}
```

Find minimum swaps required to convert a Binary tree into BST

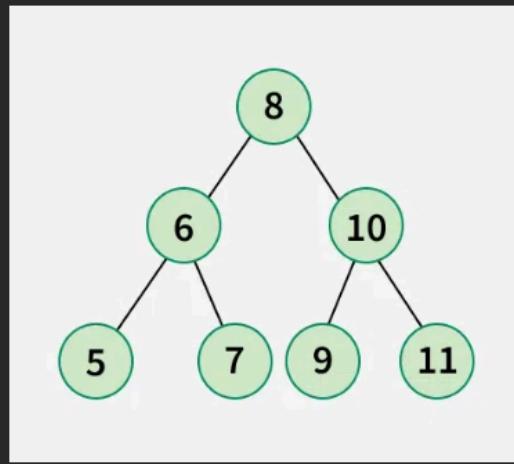


Swap 1: Swap node 8 with node 5.

Swap 2: Swap node 9 with node 10.

Swap 3: Swap node 10 with node 7.

So, minimum 3 swaps are required to obtain the below binary search tree:



The idea is to use the fact that inorder traversal of Binary Search Tree is in increasing order of their value.

So, find the inorder traversal of the Binary Tree and store it in the array and try to sort the array. The minimum number of swap required to get the array sorted will be the answer.

```
// Function to perform inorder traversal of the binary tree
// and store it in vector v
void inorder(vector<int>& arr, vector<int>& inorderArr, int index) {
```

```

int n = arr.size();

// If index is out of bounds, return
if (index >= n)
    return;

// Recursively visit left subtree
inorder(arr, inorderArr, 2 * index + 1);

// Store current node value in vector
inorderArr.push_back(arr[index]);

// Recursively visit right subtree
inorder(arr, inorderArr, 2 * index + 2);
}

// Function to calculate minimum swaps
// to sort inorder traversal
int minSwaps(vector<int>& arr) {
    int n = arr.size();
    vector<int> inorderArr;

    // Get the inorder traversal of the binary tree
    inorder(arr, inorderArr, 0);

    // Create an array of pairs to store value
    // and original index
    vector<pair<int, int>> t(inorderArr.size());
    int ans = 0;

    // Store the value and its index
    for (int i = 0; i < inorderArr.size(); i++)
        t[i] = {inorderArr[i], i};

    // Sort the pair array based on values
    // to get BST order
    sort(t.begin(), t.end());

    // Find minimum swaps by detecting cycles

```

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

    // If the element is already in the
    // correct position, continue
    if (i == t[i].second)
        continue;

    // Otherwise, perform swaps until the element
    // is in the right place
    else {

        // Swap elements to correct positions
        swap(t[i].first, t[t[i].second].first);
        swap(t[i].second, t[t[i].second].second);
    }

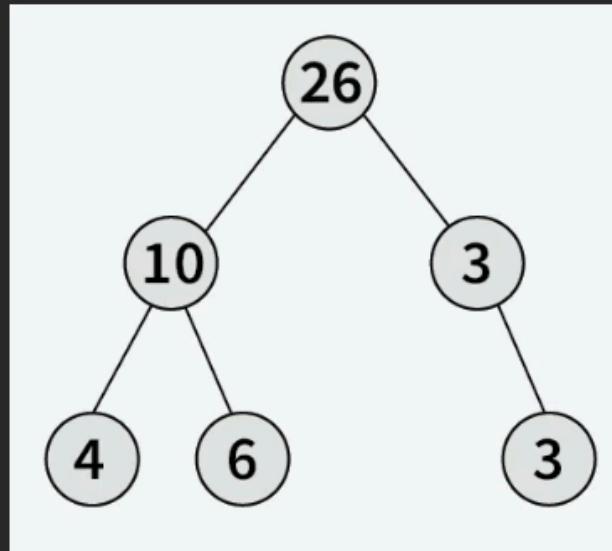
    // Check if the element is still not
    // in the correct position
    if (i != t[i].second)
        --i;

    // Increment swap count
    ans++;
}

return ans;
}
```

Check if Binary tree is Sum tree or not

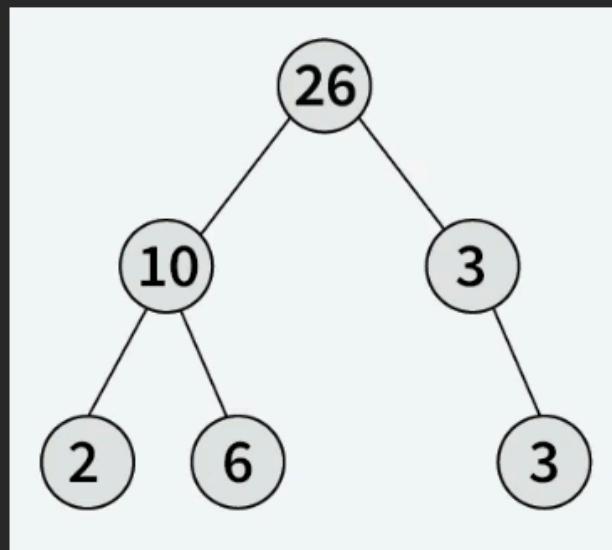
Input:



Output: True

Explanation: The above tree follows the property of Sum Tree.

Input:



Output: False



[Expected Approach] Calculating left and right subtree sum directly – O(n) Time and O(h) Space:

```

bool isSumTree(Node* root) {
    int ls, rs;

    // If node is NULL or it's a leaf node then
    // return true
    if(root == nullptr || isLeaf(root))
        return true;

    // If the left subtree and right subtree are sum trees,
    // then we can find subtree sum in O(1).
    if( isSumTree(root->left) && isSumTree(root->right)) {

        // Get the sum of nodes in left subtree
        if(root->left == nullptr)
            ls = 0;
        else if(isLeaf(root->left))
            ls = root->left->data;
        else
            ls = 2 * (root->left->data);

        // Get the sum of nodes in right subtree
        if(root->right == nullptr)
            rs = 0;
        else if(isLeaf(root->right))
            rs = root->right->data;
        else
            rs = 2 * (root->right->data);

        // If root's data is equal to sum of nodes in left
        // and right subtrees then return true else return false
        return(root->data == ls + rs);
    }

    // if either of left or right subtree is not
    // sum tree, then return false.
    return false;
}

```

[Alternate Approach] Using post order traversal – O(n) Time and O(h) Space:

```
// returns sum if tree is SumTree
// else return -1
int isSumTree(Node* root) {

    if(root == nullptr)
        return 0;

    // If node is leaf node, return its value.
    if (root->left == nullptr && root->right == nullptr)
        return root->data;

    // Calculate left subtree sum
    int ls = isSumTree(root->left);

    // if left subtree is not sum tree,
    // return -1.
    if(ls == -1)
        return -1;

    // Calculate right subtree sum
    int rs = isSumTree(root->right);

    // if right subtree is not sum tree,
    // return -1.
    if(rs == -1)
        return -1;

    if(ls + rs == root->data)
        return ls + rs + root->data;
    else
        return -1;

}
```

Check if all leaf nodes are at same level or not

Method1: Time Complexity: The function does a simple traversal of the tree, so the complexity is O(n).

Auxiliary Space: O(H) for call stack, where H is height of tree

```
/* Recursive function which checks whether
all leaves are at same level */
bool checkUtil(struct Node *root,
```

```

        int level, int *leafLevel)
{
    // Base case
    if (root == NULL) return true;

    // If a leaf node is encountered
    if (root->left == NULL &&
        root->right == NULL)
    {
        // When a leaf node is found
        // first time
        if (*leafLevel == 0)
        {
            *leafLevel = level; // Set first found leaf's level
            return true;
        }

        // If this is not first leaf node, compare
        // its level with first leaf's level
        return (level == *leafLevel);
    }

    // If this node is not leaf, recursively
    // check left and right subtrees
    return checkUtil(root->left, level + 1, leafLevel) &&
           checkUtil(root->right, level + 1, leafLevel);
}

/* The main function to check
if all leafs are at same level.
It mainly uses checkUtil() */
bool check(struct Node *root)
{
    int level = 0, leafLevel = 0;
    return checkUtil(root, level, &leafLevel);
}

```

Method 2 (Iterative):

```

// return true if all leaf nodes are
// at same level, else false

```

```
int checkLevelLeafNode(Node* root)
{
    if (!root)
        return 1;

    // create a queue for level order traversal
    queue<Node*> q;
    q.push(root);

    int flag = 0;

    // traverse until the queue is empty
    while (!q.empty()) {
        int n = q.size();

        // traverse for complete level
        for (int i = 1; i <= n; i++) {
            Node* temp = q.front();
            q.pop();

            // check for left child
            if (temp->left) {
                q.push(temp->left);
            }

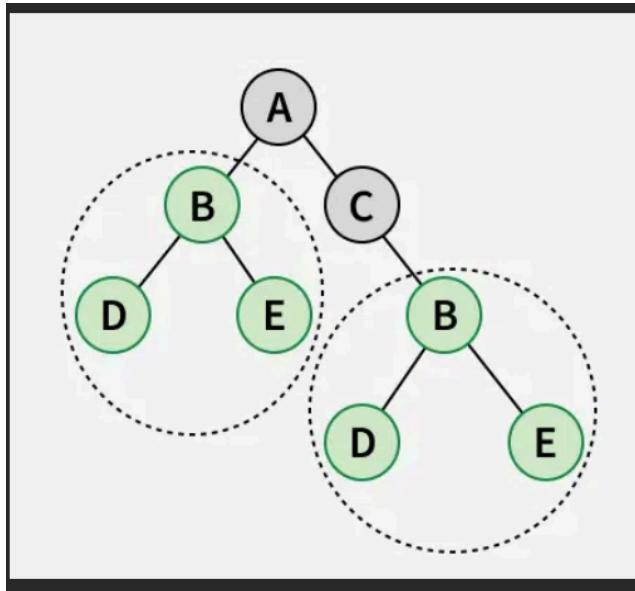
            // check for right child
            if (temp->right) {
                q.push(temp->right);
            }

            // check for leaf node
            if (temp->left == NULL && temp->right == NULL)
                flag = 1;
        }

        // check if there exist any further levels.
        if (flag && !q.empty())
            return 0;
    }

    return 1;
}
```

Check if a Binary Tree contains duplicate subtrees of size 2 or more [IMP]



[Naive Approach] Generating All Subtrees – O(n^2) Time and O(n) Space

```
// Function which generates all the subtrees of size>2
// and store its serialized form in map.
string dupSubRecur(Node *root, unordered_map<string, int> &map) {

    // For null nodes,
    if (root == nullptr) return "N";

    // For leaf nodes, return its value in string.
    if (root->left==nullptr && root->right==nullptr) {
        return to_string(root->data);
    }

    // Process the left and right subtree.
    string left = dupSubRecur(root->left, map);
    string right = dupSubRecur(root->right, map);

    // Generate the serialized form.
    string curr = "";
    curr += to_string(root->data);
    curr += '*';
    curr += left;
    curr += '*';
    curr += right;
}
```

```

curr += right;

// Store the subtree in map.
map[curr]++;
return curr;
}

int dupSub(Node *root) {
    unordered_map<string,int> map;

    // Generate all the subtrees.
    dupSubRecur(root, map);

    // Check for all subtrees.
    for (auto p: map) {

        // If subtree is duplicate.
        if (p.second>1) {
            return 1;
        }
    }

    return 0;
}

```

[Expected Approach] Using Hash Set – O(n) Time and O(n) Space

The idea is to use a hash set to store the subtrees in Serialized String Form. For a given subtree of size greater than 1, if its equivalent serialized string already exists, then return true. If all subtrees are unique, return false.

Step by step approach:

- To identify **duplicate subtrees** efficiently, we only need to check subtrees of **size 2 or 3**, as any larger duplicate subtree would already contain **smaller** duplicate subtrees. This reduces the **time complexity** of concatenating strings to **O(1)** by focusing only on nodes where both children are either leaf nodes or one is a leaf and the other is null.
- Use a **hash set**, say **s** to track **unique** subtree structures and an **answer variable ans** (initially set to false).
- Define a function **dupSubRecur** that takes a node and returns a string.
 - If the **node** is null, return “N”.
 - If it’s a **leaf node**, return its value as a string.
 - For internal nodes, first process the **left and right** subtrees. If either subtree string is **empty**, return an empty string. Otherwise, **concatenate** the current node with its left and right subtree strings. If this concatenated string is in the **hash set**, set **ans** to **true**; if not, insert it.
- Return an empty string for each processed subtree, as upper subtrees don’t need further processing.

```
// Function which checks all the subtree of size 2 or
// 3 if they are duplicate.
string dupSubRecur(Node *root, unordered_set<string> &s,
                    int &ans) {

    // For null nodes,
    if (root == nullptr) return "N";

    // For leaf nodes, return its value in string.
    if (root->left==nullptr && root->right==nullptr) {
        return to_string(root->data);
    }

    string curr = "";
    curr += to_string(root->data);
```

```

// Process the left and right subtree.
string left = dupSubRecur(root->left, s, ans);
string right = dupSubRecur(root->right, s, ans);

// If the node is parent to 2
// leaf nodes, or 1 leaf node and 1
// null node, then concatenate the strings
if (left != "" && right != "") {
    curr += "*";
    curr += left;
    curr += "*";
    curr += right;
}

// Otherwise, there is no need
// to process this node.
else {
    return "";
}

// If this subtree string is already
// present in set, set ans to 1.
if (s.find(curr) != s.end()) {
    ans = 1;
}

// Else add this string to set.
else {
    s.insert(curr);
}

return "";
}

int dupSub(Node *root) {
    int ans = 0;
    unordered_set<string> s;

    dupSubRecur(root, s, ans);

    return ans;
}

```

Check if 2 trees are mirror or not

Recursive:

```
// Function to check if two roots are mirror images
bool areMirrors(Node* root1, Node* root2) {

    // If both roots are empty, they are mirrors
    if (root1 == nullptr && root2 == nullptr)
        return true;

    // If only one root is empty, they are not mirrors
    if (root1 == nullptr || root2 == nullptr)
        return false;

    // Check if the root data is the same and
    // if the left subtree of root1 is a mirror
    // of the right subtree of root2 and vice versa
    return (root1->data == root2->data) &&
           areMirrors(root1->left, root2->right) &&
           areMirrors(root1->right, root2->left);
}
```

Iterative:

The idea is to check if two binary trees are mirrors using two stacks to simulate recursion. Nodes from each tree are pushed onto the stacks in a way that compares the left subtree of one tree with the right subtree of the other, and vice versa. This approach systematically compares nodes while maintaining their mirrored structure, ensuring the trees are symmetric relative to their root. Please refer to Iterative method to check if two trees are mirror of each other for implementation.

Sum of Nodes on the Longest path from root to leaf node

Expected Approach – 1] Using Recursive – O(n) Time and O(h) Space

```
void sumOfRootToLeaf(Node* root, int sum, int len,
                      int& maxLen, int& maxSum) {

    // Base case: if the current node is null
    if (!root) {
```

```

// Checking if the current path has a longer length
// and update maxLen and maxSum accordingly
if (len > maxLen) {
    maxLen = len;
    maxSum = sum;
}
// If the lengths are equal, check if the current sum is
// greater and update maxSum if necessary
else if (len == maxLen && sum > maxSum) {
    maxSum = sum;
}
return;
}

// Recursively calculating the sum of
// the left and right subtrees
sumOfRootToLeaf(root->left, sum + root->data, len + 1, maxLen, maxSum);
sumOfRootToLeaf(root->right, sum + root->data, len + 1, maxLen, maxSum);
}

// Function to calculate the sum of the longest root to leaf path
int sumOfLongRootToLeafPath(Node* root) {

    // Base case: if the tree is empty
    if (!root) return 0;

    // Initializing the variables to store the maximum length and sum
    int maxSum = INT_MIN, maxLen = 0;

    // Calling the utility function
    sumOfRootToLeaf(root, 0, 0, maxLen, maxSum);

    // Returning the maximum sum
    return maxSum;
}

```

[Expected Approach – 2] Using level order traversal – O(n) Time and O(n) Space

```

// Function to calculate the sum of the longest root to
// leaf path using level order traversal
int sumOfLongRootToLeafPath(Node* root) {

    // Base case: if the tree is empty
    if (!root) return 0;

```

```

// Initialize variables to store
// e maximum length and sum
int maxSum = 0;
int maxLen = 0;

// Queue for level order traversal, storing pairs of
// nodes and their corresponding sum and length
queue<pair<Node*, pair<int, int>>> q;

// Starting with the root node, initial
// sum, and length
q.push({root, {root->data, 1}});

while (!q.empty()) {
    auto front = q.front();
    q.pop();

    Node* node = front.first;
    int sum = front.second.first;
    int len = front.second.second;

    // If it's a leaf node, check if we need
    // to update maxLen and maxSum
    if (!node->left && !node->right) {
        if (len > maxLen) {
            maxLen = len;
            maxSum = sum;
        } else if (len == maxLen && sum > maxSum) {
            maxSum = sum;
        }
    }

    // Push left and right children
    // into the queue
    if (node->left) {
        q.push({node->left, {sum + node->left->data, len + 1}});
    }
    if (node->right) {
        q.push({node->right, {sum + node->right->data, len + 1}});
    }
}

return maxSum;
}

```

Check if given graph is tree or not. [IMP]

Approach 1:

An undirected graph is a tree if it has the following properties.

There is no cycle.

The graph is connected.

```
// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V; // No. of vertices
    list<int> *adj; // Pointer to an array for adjacency lists
    bool isCyclicUtil(int v, bool visited[], int parent);
public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isTree(); // returns true if graph is tree
};

Graph::Graph(int V)
{
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive function that uses visited[] and parent to
// detect cycle in subgraph reachable from vertex v.
bool Graph::isCyclicUtil(int v, bool visited[], int parent)
```

```

{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for
        // that adjacent
        if (!visited[*i])
        {
            if (isCyclicUtil(*i, visited, v))
                return true;
        }

        // If an adjacent is visited and not parent of current
        // vertex, then there is a cycle.
        else if (*i != parent)
            return true;
    }
    return false;
}

// Returns true if the graph is a tree, else false.
bool Graph::isTree()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
    bool *visited = new bool[V];
    for (int i = 0; i < V; i++)
        visited[i] = false;

    // The call to isCyclicUtil serves multiple purposes.
    // It returns true if graph reachable from vertex 0
    // is cyclic. It also marks all vertices reachable
    // from 0.
    if (isCyclicUtil(0, visited, -1))
        return false;

    // If we find a vertex which is not reachable from 0
    // (not marked by isCyclicUtil()), then we return false
    for (int u = 0; u < V; u++)
        if (!visited[u])
            return false;
}

```

```

        return true;
    }

// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.isTree()? cout << "Graph is Tree\n":
                  cout << "Graph is not Tree\n";

    return 0;
}

```

Approach 2:

However if we observe carefully the definition of tree and its structure we will deduce that if a graph is connected and has $n - 1$ edges exactly then the graph is a tree.

```

// A C++ Program to check whether a graph is tree or not
#include<iostream>
#include <list>
#include <limits.h>
using namespace std;

// Class for an undirected graph
class Graph
{
    int V; // No. of vertices
    int E; // No. of edges
    list<int> *adj; // Pointer to an array for adjacency lists
    void dfsTraversal(int v, bool visited[], int parent);

```

```

public:
    Graph(int V); // Constructor
    void addEdge(int v, int w); // to add an edge to graph
    bool isConnected(); // returns true if graph is connected
    bool isTree(); // returns true if the graph is tree
};

Graph::Graph(int V)
{
    E = 0;
    this->V = V;
    adj = new list<int>[V];
}

void Graph::addEdge(int v, int w)
{
    E++; // increase the number of edges
    adj[v].push_back(w); // Add w to v's list.
    adj[w].push_back(v); // Add v to w's list.
}

// A recursive dfs function that uses visited[] and parent to
// traverse the graph and mark visited[v] to true for visited nodes
void Graph::dfsTraversal(int v, bool visited[], int parent)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all the vertices adjacent to this vertex
    list<int>::iterator i;
    for (i = adj[v].begin(); i != adj[v].end(); ++i)
    {
        // If an adjacent is not visited, then recur for
        // that adjacent
        if (!visited[*i])
        {
            dfsTraversal(*i, visited, v);
        }
    }
}

// Returns true if the graph is connected, else false.
bool Graph::isConnected()
{
    // Mark all the vertices as not visited and not part of
    // recursion stack
}

```

```

bool *visited = new bool[V];
for (int i = 0; i < V; i++)
    visited[i] = false;

// Performing DFS traversal of the graph and marking
// reachable vertices from 0 to true
dfsTraversal(0, visited, -1);

// If we find a vertex which is not reachable from 0
// (not marked by dfsTraversal()), then we return false
// since graph is not connected
for (int u = 0; u < V; u++)
    if (!visited[u])
        return false;

// since all nodes were reachable so we returned true and
// and hence graph is connected
return true;
}

bool Graph::isTree()
{
    // as we proved earlier if a graph is connected and has
    // V - 1 edges then it is a tree i.e. E = V - 1
    return isConnected() and E == V - 1;
}

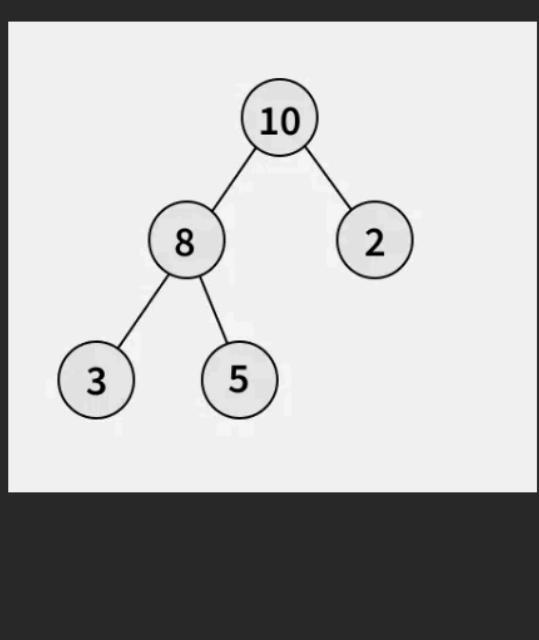
// Driver program to test above functions
int main()
{
    Graph g1(5);
    g1.addEdge(1, 0);
    g1.addEdge(0, 2);
    g1.addEdge(0, 3);
    g1.addEdge(3, 4);
    g1.isTree()? cout << "Graph is Tree\n":
                cout << "Graph is not Tree\n";

    Graph g2(5);
    g2.addEdge(1, 0);
    g2.addEdge(0, 2);
    g2.addEdge(2, 1);
    g2.addEdge(0, 3);
    g2.addEdge(3, 4);
    g2.isTree()? cout << "Graph is Tree\n":
                cout << "Graph is not Tree\n";
}

```

```
    return 0;  
}
```

Find Largest subtree sum in a tree



[Expected Approach – 1] Using Recursion – O(n) Time and O(h) Space

```
// Helper function to find largest  
// subtree sum recursively.  
int findLargestSubtreeSumUtil(Node* root, int& ans) {  
  
    // If current node is null then  
    // return 0 to parent node.  
    if (root == nullptr)  
        return 0;  
  
    // Subtree sum rooted at current node.  
    int currSum = root->data +  
        findLargestSubtreeSumUtil(root->left, ans)  
        + findLargestSubtreeSumUtil(root->right, ans);  
  
    // Update answer if current subtree  
    // sum is greater than answer so far.
```

```

ans = max(ans, currSum);

// Return current subtree sum to
// its parent node.
return currSum;
}

// Function to find largest subtree sum.
int findLargestSubtreeSum(Node* root) {

    // If tree does not exist,
    // then answer is 0.
    if (root == nullptr)
        return 0;

    // Variable to store maximum subtree sum.
    int ans = INT_MIN;

    // Call to recursive function to
    // find maximum subtree sum.
    findLargestSubtreeSumUtil(root, ans);

    return ans;
}

```

[Expected Approach – 2] Using BFS – O(n) Time and O(n) Space

```

int findLargestSubtreeSum(Node* root) {

    // Base case when tree is empty
    if (root == nullptr)
        return 0;

    int ans = INT_MIN;

    queue<Node*> q;

    // Vector of Vector for storing
    // nodes at a particular level
    vector<vector<Node*>> levels;

    // Map for storing sum of subtree
    // rooted at a particular node
    unordered_map<Node*, int> subtreeSum;

```

```

// Push root to the queue
q.push(root);

while (!q.empty()) {

    int n = q.size();

    vector<Node*> level;

    while (n--) {
        Node* node = q.front();

        // Push current node to current
        // level vector
        level.push_back(node);

        // Add left & right child of node
        // in the queue
        if (node->left)
            q.push(node->left);
        if (node->right)
            q.push(node->right);

        q.pop();
    }

    // add current level to levels
    // vector
    levels.push_back(level);
}

// Traverse all levels from bottom
// most level to top most level
for (int i = levels.size() - 1; i >= 0; i--) {

    for (auto e : levels[i]) {

        // add value of current node
        subtreeSum[e] = e->data;

        // If node has left child, add the subtree sum
        // of subtree rooted at left child
        if (e->left)
            subtreeSum[e] += subtreeSum[e->left];

        // If node has right child, add the subtree sum
    }
}

```

```

    // of subtree rooted at right child
    if (e->right)
        subtreeSum[e] += subtreeSum[e->right];

    // update ans to maximum of ans and sum of
    // subtree rooted at current node
    ans = max(ans, subtreeSum[e]);
}
}

return ans;
}

```

Maximum Sum of nodes in Binary tree such that no two are adjacent

Using Recursion – O(2^n) Time and O(n) Space

```

// Utility method to return the maximum sum
// rooted at the node 'curr'
int getMaxSumUtil(Node* node) {
    if (node == nullptr) {

        // If the node is null, the sum is 0
        return 0;
    }

    // Calculate the maximum sum including the
    // current node
    int incl = node->data;

    // If the left child exists, include its contribution
    if (node->left) {
        incl += getMaxSumUtil(node->left->left) +
            getMaxSumUtil(node->left->right);
    }

    // If the right child exists, include its contribution
    if (node->right) {
        incl += getMaxSumUtil(node->right->left) +
            getMaxSumUtil(node->right->right);
    }
}

```

```

// Calculate the maximum sum excluding
// the current node
int excl = 0;
if (node->left) {
    excl += getMaxSumUtil(node->left);
}
if (node->right) {
    excl += getMaxSumUtil(node->right);
}

// The result for the current node is the
// maximum of including or excluding it
return max(incl, excl);
}

int getMaxSum(Node* root) {

    // If the tree is empty, the maximum sum is 0
    if (root == nullptr) {
        return 0;
    }

    // Call the utility function to compute the
    // maximum sum for the entire tree
    return getMaxSumUtil(root);
}

```

Using pair – O(n) Time and O(h) Space

```

pair<int, int> maxSumHelper(Node* root) {
    if (root == nullptr) {
        pair<int, int> sum(0, 0);
        return sum;
    }
    pair<int, int> sum1 = maxSumHelper(root->left);
    pair<int, int> sum2 = maxSumHelper(root->right);
    pair<int, int> sum;

    // This node is included (Left and right children
    // are not included)
    sum.first = sum1.second + sum2.second + root->data;

    // This node is excluded (Either left or right
    // child is included)
    sum.second = max(sum1.first, sum1.second)

```

```

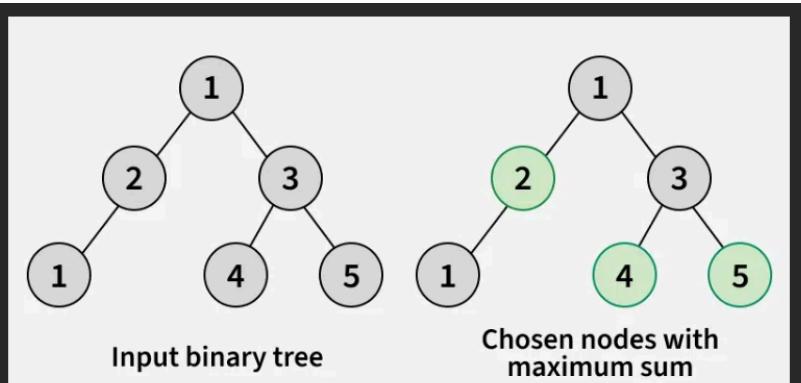
        + max(sum2.first, sum2.second);

    return sum;
}

// Returns maximum sum from subset of nodes
// of binary tree under given constraints
int getMaxSum(Node* root) {
    pair<int, int> res = maxSumHelper(root);
    return max(res.first, res.second);
}

```

Using Top-Down DP (Memoization) – O(n) Time and O(n) Space



*The naive approach leads to **recalculating** results for the same nodes multiple times. For example, if we **include** the root node, we **recursively compute** the sum for its **grandchildren** (**nodes 4 and 5**). But if we **exclude** the root, we compute the sum for its children, and **node 3** also computes the sum for its children (**4 and 5 again**).*

*To avoid this redundancy, we use **memoization**:*

- We store the result of each node in a [hashmap](#).
- When a node's value is needed again, we directly **return** it from the map instead of recalculating.

```

// Utility method to return the maximum sum rooted
// at the node 'node'
int getMaxSumUtil(Node* node, unordered_map<Node*, int>& memo) {
    if (node == nullptr) {

        // If the node is null, the sum is 0
        return 0;
    }

```

```

// If the result is already computed, return it from memo
if (memo.find(node) != memo.end()) {
    return memo[node];
}

// Calculate the maximum sum including the current node
int incl = node->data;

// If the left child exists, include its grandchildren
if (node->left) {
    incl += getMaxSumUtil(node->left->left, memo) +
        getMaxSumUtil(node->left->right, memo);
}

// If the right child exists, include its grandchildren
if (node->right) {
    incl += getMaxSumUtil(node->right->left, memo) +
        getMaxSumUtil(node->right->right, memo);
}

// Calculate the maximum sum excluding the current node
int excl = getMaxSumUtil(node->left, memo) +
    getMaxSumUtil(node->right, memo);

// Store the result in memo and return
// the maximum of incl and excl
memo[node] = max(incl, excl);
return memo[node];
}

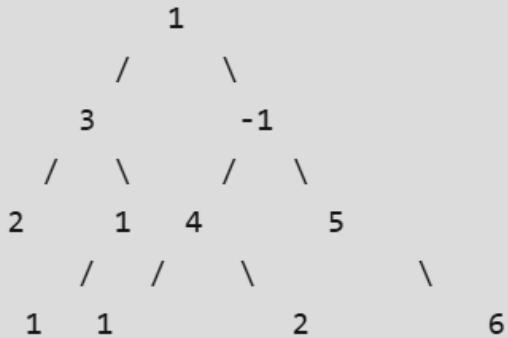
// Function to compute the maximum
// sum of non-adjacent nodes
int getMaxSum(Node* root) {
    unordered_map<Node*, int> memo;
    return getMaxSumUtil(root, memo);
}

```

Print all "K" Sum paths in a Binary tree

```
Input : k = 5
```

```
Root of below binary tree:
```



```
Output :
```

```
3 2
3 1 1
1 3 1
4 1
1 -1 4 1
-1 4 2
5
1 -1 5
```

```
// This function prints all paths that have sum k
void printKPathUtil(Node* root, vector<int>& path, int k)
{
    // empty node
    if (!root)
        return;

    // add current node to the path
    path.push_back(root->data);

    // check if there's any k sum path
    // in the left sub-tree.
    printKPathUtil(root->left, path, k);

    // check if there's any k sum path
```

```

// in the right sub-tree.
printKPathUtil(root->right, path, k);

// check if there's any k sum path that
// terminates at this node
// Traverse the entire path as
// there can be negative elements too
int f = 0;
for (int j = path.size() - 1; j >= 0; j--) {
    f += path[j];

    // If path sum is k, print the path
    if (f == k)
        printVector(path, j);
}

// Remove the current element from the path
path.pop_back();
}

// A wrapper over printKPathUtil()
void printKPath(Node* root, int k)
{
    vector<int> path;
    printKPathUtil(root, path, k);
}

```

Find LCA in a Binary tree

Using Arrays to Store Paths of Nodes from Root – O(n) Time and O(n) Space

```

// Function to find path from root to given node.
bool findPath(Node* root, vector<Node*>& path, int k) {

    // base case
    if (root == nullptr)
        return false;

    // Store current node value in the path.
    path.push_back(root);

    // If node value is equal to k, or

```

```

// if node exist in left subtree or
// if node exist in right subtree return true
if (root->data == k ||
    findPath(root->left, path, k) ||
    findPath(root->right, path, k))
    return true;

// else remove root from path and return false
path.pop_back();
return false;
}

// Returns LCA of two nodes.
Node* lca(Node* root, int n1, int n2) {

    // to store paths to n1 and n2 from the root
    vector<Node*> path1, path2;

    // Find paths from root to n1 and
    // root to n2. If either
    // n1 or n2 is not present, return nullptr
    if (!findPath(root, path1, n1) ||
        !findPath(root, path2, n2))
        return nullptr;

    // Compare the paths to get the first
    // different value
    int i;
    for (i = 0; i < path1.size()
                     && i < path2.size(); i++) {
        if (path1[i] != path2[i])
            return path1[i-1];
    }

    // if both the datas are same, return last node
    return path1[i-1];
}

```

Expected Approach] Using Single Traversal – O(n) Time and O(h) Space

```

// Function to find LCA of two keys.
Node* lca(Node* root, int n1, int n2) {

    if (!root)
        return nullptr;

```

```

// If either key matches with root data, return root
if (root->data == n1 || root->data == n2)
    return root;

// Look for datas in left and right subtrees
Node* leftLca = lca(root->left, n1, n2);
Node* rightLca = lca(root->right, n1, n2);

// If both of the above calls return Non-NULL, then one
// data is present in one subtree and the other is present
// in the other, so this node is the LCA
if (leftLca && rightLca)
    return root;

// Otherwise check if left subtree or right subtree is
// LCA
return leftLca ? leftLca : rightLca;
}

```

Alternate Approach – O(n) Time and O(h) Space

```

// Function to find LCA of two datas.
Node* findLca(Node* root, int n1, int n2) {

    if (!root)
        return nullptr;

    // If either data matches with root data, return root
    if (root->data == n1 || root->data == n2)
        return root;

    // Look for datas in left and right subtrees
    Node* leftLca = findLca(root->left, n1, n2);
    Node* rightLca = findLca(root->right, n1, n2);

    // If both of the above calls return Non-NULL, then one
    // data is present in one subtree and the other is present
    // in the other, so this node is the LCA
    if (leftLca && rightLca)
        return root;

    // Otherwise check if left subtree or right subtree is
    // LCA
}

```

```

        return leftLca ? leftLca : rightLca;
    }

// Returns true if key k is present in tree rooted with root
bool checkIfPresent(Node* root, int k) {

    // Base Case
    if (root == nullptr)
        return false;

    // If data is present at root, or in left subtree or
    // right subtree, return true;
    if (root->data == k || checkIfPresent(root->left, k) ||
        checkIfPresent(root->right, k))
        return true;

    // Else return false
    return false;
}

// function to check if keys are present
// in the tree and returns the lca.
Node* lca(Node* root, int n1, int n2) {

    // Return LCA only if both n1 and n2 are
    // present in tree
    if (checkIfPresent(root, n1) && checkIfPresent(root, n2))
        return findLca(root, n1, n2);

    // Else return nullptr
    return nullptr;
}

```

Application of Lowest Common Ancestor(LCA)

We use LCA to find the shortest distance between pairs of nodes in a tree: the shortest distance from n1 to n2 can be computed as the distance from the LCA to n1, plus the distance from the LCA to n2.

Find distance between 2 nodes in a Binary tree

Using LCA and Path Length – O(n) Time and O(h) Space

```

// Function to find the level of a node
int findLevel(Node *root, int k, int level) {
    if (root == nullptr)
        return -1;
    if (root->data == k)
        return level;

    // Recursively call function on left child
    int leftLevel = findLevel(root->left, k, level + 1);

    // If node is found on left, return level
    // Else continue searching on the right child
    if (leftLevel != -1) {
        return leftLevel;
    }
    else {
        return findLevel(root->right, k, level + 1);
    }
}

// Function to find the lowest common ancestor
// and calculate distance between two nodes
Node *findLcaAndDistance(Node *root, int a, int b, int &d1,
                           int &d2, int &dist, int lvl) {
    if (root == nullptr)
        return nullptr;

    if (root->data == a) {

        // If first node found, store level and
        // return the node
        d1 = lvl;
        return root;
    }
    if (root->data == b) {

        // If second node found, store level and
        // return the node
        d2 = lvl;
        return root;
    }

    // Recursively call function on left child

```

```

Node *left = findLcaAndDistance
            (root->left, a, b, d1, d2, dist, lvl + 1);

// Recursively call function on right child
Node *right = findLcaAndDistance
            (root->right, a, b, d1, d2, dist, lvl + 1);

if (left != nullptr && right != nullptr) {

    // If both nodes are found in different
    // subtrees, calculate the distance
    dist = d1 + d2 - 2 * lvl;
}

// Return node found or nullptr if not found
if (left != nullptr) {
    return left;
}
else {
    return right;
}
}

// Function to find distance between two nodes
int findDist(Node *root, int a, int b) {
    int d1 = -1, d2 = -1, dist;

    // Find lowest common ancestor and calculate distance
    Node *lca = findLcaAndDistance(root, a, b, d1, d2, dist, 1);

    if (d1 != -1 && d2 != -1) {

        // Return the distance if both
        // nodes are found
        return dist;
    }

    if (d1 != -1) {

        // If only first node is found, find
        // distance to second node
        dist = findLevel(lca, b, 0);
        return dist;
    }
}

```

```

    }

    if (d2 != -1) {

        // If only second node is found, find
        // distance to first node
        dist = findLevel(lca, a, 0);
        return dist;
    }

    // Return -1 if both nodes not found
    return -1;
}

```

Using LCA – O(n) Time and O(h) Space

```

// Function to find the Lowest Common Ancestor
// (LCA) of two nodes
Node* LCA(Node* root, int n1, int n2) {
    if (root == nullptr)
        return root;

    if (root->data == n1 || root->data == n2)
        return root;

    Node* left = LCA(root->left, n1, n2);
    Node* right = LCA(root->right, n1, n2);

    if (left != nullptr && right != nullptr)
        return root;

    if (left == nullptr && right == nullptr)
        return nullptr;

    return (left != nullptr) ? LCA(root->left, n1, n2)
                           : LCA(root->right, n1, n2);
}

// Returns level of key k if it is present in tree,
// otherwise returns -1
int findLevel(Node* root, int k, int level) {
    if (root == nullptr)
        return -1;

```

```

    if (root->data == k)
        return level;

    int left = findLevel(root->left, k, level + 1);
    if (left == -1)
        return findLevel(root->right, k, level + 1);

    return left;
}

// Function to find distance between two
// nodes in a binary tree
int findDistance(Node* root, int a, int b) {
    Node* lca = LCA(root, a, b);

    int d1 = findLevel(lca, a, 0);
    int d2 = findLevel(lca, b, 0);

    return d1 + d2;
}

```

Using LCA (one pass) – O(n) Time and O(h) Space

```

// Function that calculates distance between two nodes.
// It returns a pair where the first element indicates
// whether n1 or n2 is found and the second element
// is the distance from the current node.
pair<bool, int> calculateDistance(Node* root, int n1,
                                    int n2, int& distance) {
    if (!root) return {false, 0};

    // Recursively calculate the distance in
    // the left and right subtrees
    pair<bool, int> left =
        calculateDistance(root->left, n1, n2, distance);
    pair<bool, int> right =
        calculateDistance(root->right, n1, n2, distance);

    // Check if the current node is either n1 or n2
    bool current = (root->data == n1 || root->data == n2);
    if (current) {
        distance = 0;
        return {true, 0};
    }

    if (left.first) {
        if (right.first) {
            distance = left.second + right.second + 1;
            return {true, distance};
        }
        else {
            distance = left.second + 1;
            return {true, distance};
        }
    }
    else if (right.first) {
        distance = right.second + 1;
        return {true, distance};
    }
    else {
        distance = max(left.second, right.second) + 1;
        return {false, distance};
    }
}

```

```

// If current node is one of n1 or n2 and
// we found the other in a subtree, update distance
if (current && (left.first || right.first)) {
    distance = max(left.second, right.second);
    return {false, 0};
}

// If left and right both returned true,
// root is the LCA and we update the distance
if (left.first && right.first) {
    distance = left.second + right.second;
    return {false, 0};
}

// If either left or right subtree contains n1 or n2,
// return the updated distance
if (left.first || right.first || current) {
    return {true, max(left.second, right.second) + 1};
}

// If neither n1 nor n2 exist in the subtree
return {false, 0};
}

// The function that returns distance between n1 and n2.
int findDistance(Node* root, int n1, int n2) {
    int distance = 0;
    calculateDistance(root, n1, n2, distance);
    return distance;
}

```

Kth Ancestor of node in a Binary tree

```

/ function to generate array of ancestors
void generateArray(Node *root, int ancestors[])
{
    // There will be no ancestor of root node
    ancestors[root->data] = -1;

```

```

// level order traversal to
// generate 1st ancestor
queue<Node*> q;
q.push(root);

while(!q.empty())
{
    Node* temp = q.front();
    q.pop();

    if (temp->left)
    {
        ancestors[temp->left->data] = temp->data;
        q.push(temp->left);
    }

    if (temp->right)
    {
        ancestors[temp->right->data] = temp->data;
        q.push(temp->right);
    }
}

// function to calculate Kth ancestor
int kthAncestor(Node *root, int n, int k, int node)
{
    // create array to store 1st ancestors
    int ancestors[n+1] = {0};

    // generate first ancestor array
    generateArray(root,ancestors);

    // variable to track record of number of
    // ancestors visited
    int count = 0;

    while (node!=-1)
    {
        node = ancestors[node];
        count++;

        if(count==k)

```

```

        break;
    }

    // print Kth ancestor
    return node;
}

```

Method 2: In this method first we will get an element whose ancestor has to be searched and from that node, we will decrement count one by one till we reach that ancestor node.
for example –

```

// Program to find kth ancestor
bool ancestor(struct node* root, int item, int &k)
{
    if(root == NULL)
        return false;

    // Element whose ancestor is to be searched
    if(root->data == item)
    {
        //reduce count by 1
        k = k-1;
        return true;
    }
    else
    {

        // Checking in left side
        bool flag = ancestor(root->left,item,k);
        if(flag)
        {
            if(k == 0)
            {

                // If count = 0 i.e. element is found
                cout<<"["<<root->data<<"] ";
                return false;
            }
        }
    }
}

```

```

        // ancestor we are searching for
        // so decrement count
        k = k-1;
        return true;
    }

    // Similarly Checking in right side
    bool flag2 = ancestor(root->right,item,k);
    if(flag2)
    {
        if(k == 0)
        {
            cout<<"["<<root->data<<"] ";
            return false;
        }
        k = k-1;
        return true;
    }
}
}

```

Method 3: Iterative Approach

The basic idea behind the iterative approach is to traverse the binary tree from the root node and keep track of the path from the root to the target node using a stack. Once we find the target node, we pop elements from the stack and add their values to a vector until we reach the k th ancestor or the stack becomes empty

Follow the Steps below to implement the above idea:

1. Initialize a stack to keep track of the path from the root to the target node, and a vector to store the ancestors.
2. Traverse the binary tree from the root node using a while loop.
3. If the current node is not NULL, push it onto the stack and move to its left child.
4. If the current node is NULL, pop the top element from the stack. If the top element is the target node, break out of the loop. Otherwise, move to its right child.
5. If the target node is not found, return -1.
6. Pop elements from the stack and add their values to the ancestors vector until we reach the k th ancestor or the stack becomes empty.
7. If the stack becomes empty before we reach the k th ancestor, return -1.
8. Return the value of the k th ancestor

```

// Function to find the kth ancestor of the given node using iterative
approach
int kthAncestor(TreeNode* root, int node, int k) {
    // Initialize a stack to keep track of the path from the root to the
target node
    stack<TreeNode*> s;
    vector<int> ancestors;
    bool found = false;

    // Traverse the binary tree from the root node
    while (root != NULL || !s.empty()) {
        // If the current node is not NULL, push it onto the stack and move
to its left child
        if (root != NULL) {
            s.push(root);
            root = root->left;
        }
        // If the current node is NULL, pop the top element from the stack
        // If the top element is the target node, break out of the loop
        // Otherwise, move to its right child
        else {
            TreeNode* temp = s.top();
            s.pop();
            if (temp->val == node) {
                found = true;
                break;
            }
            if (temp->right != NULL) {
                root = temp->right;
            }
        }
    }

    // If the target node is not found, return -1
    if (!found) {
        return -1;
    }

    // Pop elements from the stack and add their values to the ancestors
vector
    // until we reach the kth ancestor or the stack becomes empty
    while (!s.empty() && k > 0) {
        TreeNode* temp = s.top();

```

```

        s.pop();
        ancestors.push_back(temp->val);
        k--;
    }

    // If the stack becomes empty before we reach the kth ancestor, return
-1
    if (k > 0) {
        return -1;
    }

    // Return the value of the kth ancestor
    return ancestors.back();
}

```

Find all Duplicate subtrees in a Binary tree [IMP]

```

string inorder(Node* node, unordered_map<string, int>& m)
{
    if (!node)
        return "";

    string str = "(";
    str += inorder(node->left, m);
    str += to_string(node->data);
    str += inorder(node->right, m);
    str += ")";

    // Subtree already present (Note that we use
    // unordered_map instead of unordered_set
    // because we want to print multiple duplicates
    // only once, consider example of 4 in above
    // subtree, it should be printed only once.
    if (m[str] == 1)
        cout << node->data << " ";

    m[str]++;
}

return str;

```

```

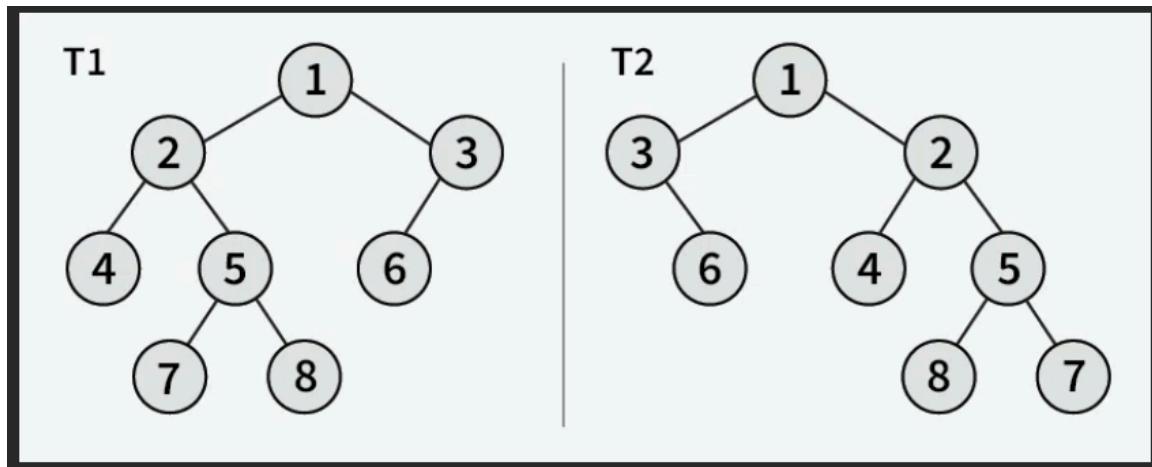
}

// Wrapper over inorder()
void printAllDups(Node* root)
{
    unordered_map<string, int> m;
    inorder(root, m);
}

/* Helper function that allocates a
new node with the given data and
NULL left and right pointers. */
Node* newNode(int data)
{
    Node* temp = new Node;
    temp->data = data;
    temp->left = temp->right = NULL;
    return temp;
}

```

Tree Isomorphism Problem



[Expected Approach – 1] Using Recursion – O(n) Time and O(n) Space

```

// Function to check if two trees are isomorphic
bool isIsomorphic(Node* root1, Node* root2) {

```

```

// Both roots are NULL, trees are isomorphic
// by definition
if (root1 == nullptr && root2 == nullptr) {
    return true;
}

// Exactly one of the root1 and root2 is NULL,
// trees not isomorphic
if (root1 == nullptr || root2 == nullptr) {
    return false;
}

// If the data doesn't match, trees
// are not isomorphic
if (root1->data != root2->data) {
    return false;
}

// Check if the trees are isomorphic by
// considering the two cases:
// Case 1: The subtrees have not been flipped
// Case 2: The subtrees have been flipped
return (isIsomorphic(root1->left, root2->left) &&
        isIsomorphic(root1->right, root2->right)) ||
       (isIsomorphic(root1->left, root2->right) &&
        isIsomorphic(root1->right, root2->left));
}

int main() {

    // Representation of input binary tree 1
    //      1
    //     / \
    //    2   3
    //   / \
    //  4   5
    //   / \
    //  7   8
    Node* root1 = new Node(1);
    root1->left = new Node(2);
    root1->right = new Node(3);
    root1->left->left = new Node(4);
    root1->left->right = new Node(5);
}

```

```

root1->left->right->left = new Node(7);
root1->left->right->right = new Node(8);

// Representation of input binary tree 2
//      1
//     / \
//    3   2
//   /   / \
//  6   4   5
//        / \
//       8   7

Node* root2 = new Node(1);
root2->left = new Node(3);
root2->right = new Node(2);
root2->left->left = new Node(6);
root2->right->left = new Node(4);
root2->right->right = new Node(5);
root2->right->right->left = new Node(8);
root2->right->right->right = new Node(7);

if (isIsomorphic(root1, root2)) {
    cout << "True\n";
}
else {
    cout << "False\n";
}

return 0;
}

```

[Expected Approach – 2] Using Iteration – O(n) Time and O(n) Space

To solve the question mentioned above we traverse both the trees iteratively using level order traversal and store the levels in a queue data structure. There are following two conditions at each level

The value of nodes has to be the same.

The number of nodes at each level should be the same.

Binary Search Tree

Find a value in a BST

```
// function to search a key in a BST
Node* search(Node* root, int key) {

    // Base Cases: root is null or key
    // is present at root
    if (root == NULL || root->key == key)
        return root;

    // Key is greater than root's key
```

```

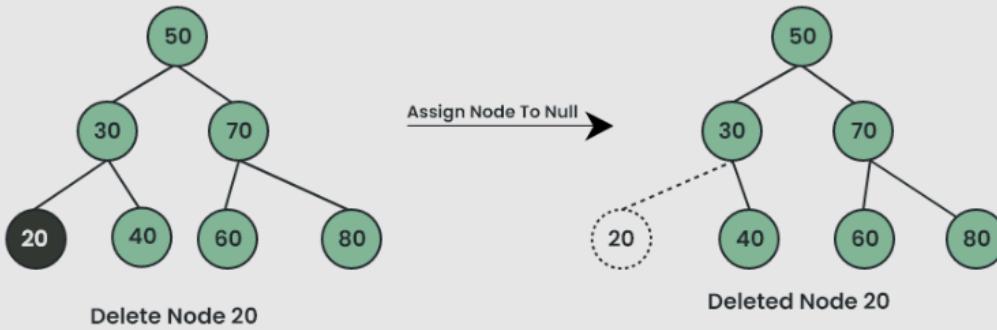
if (root->key < key)
    return search(root->right, key);

// Key is smaller than root's key
return search(root->left, key);
}

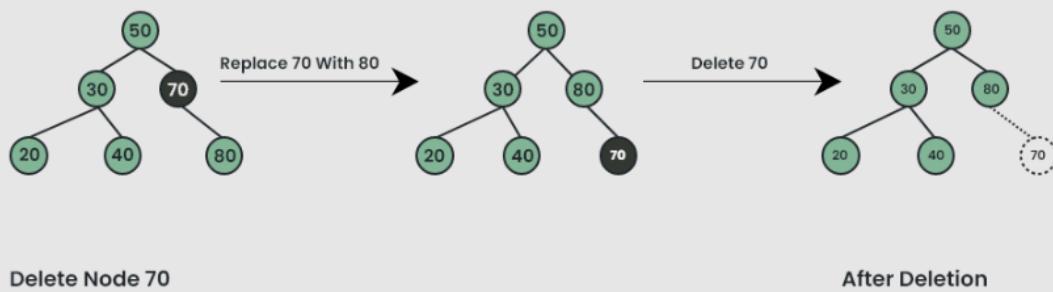
```

Deletion of a node in a BST

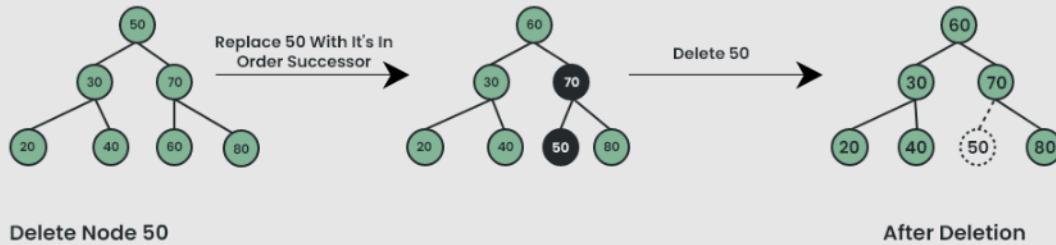
Case 1: Delete A Leaf Node In BST



Case 2: Delete A Node With Single Child In BST



Case 3 : Delete A Node With Both Children In BST



```
// Note that it is not a generic inorder  
// successor function. It mainly works  
// when right child is not empty which is  
// the case we need in BST delete  
Node* getSuccessor(Node* curr){  
    curr = curr->right;  
    while (curr != nullptr && curr->left != nullptr)  
        curr = curr->left;  
    return curr;  
}  
  
// This function deletes a given key x from  
// the give BST and returns modified root of  
// the BST (if it is modified)  
Node* delNode(Node* root, int x){  
  
    // Base case  
    if (root == nullptr)  
        return root;  
  
    // If key to be searched is in a subtree  
    if (root->key > x)  
        root->left = delNode(root->left, x);  
    else if (root->key < x)  
        root->right = delNode(root->right, x);  
  
    // If root matches with the given key  
    else {  
  
        // Cases when root has 0 children  
        // or only right child  
    }  
}
```

```

    if (root->left == nullptr) {
        Node* temp = root->right;
        delete root;
        return temp;
    }

    // When root has only left child
    if (root->right == nullptr) {
        Node* temp = root->left;
        delete root;
        return temp;
    }

    // When both children are present
    Node* succ = getSuccessor(root);
    root->key = succ->key;
    root->right = delNode(root->right, succ->key);
}
return root;
}

```

Insertion of a node in a BST

Find inorder successor and inorder predecessor in a BST

You are given root node of the BST and an integer key. You need to find the in-order successor and predecessor of the given key. If either predecessor or successor is not found, then set it to NULL.

Note:- In an inorder traversal the number just smaller than the target is the predecessor and the number just greater than the target is the successor.

```

class Solution
{
public:

    void inorder(Node* root, vector<int> &v){
        if(!root) return;

        inorder(root->left, v);
        v.push_back(root->key);
        inorder(root->right, v);
    }

    int bs(vector<int> v, int t){
        int l = 0, r = v.size() - 1;
        int res = -1;
        while(l <= r){
            int m = (l+r)/2;
            if(v[m] == t) return m;
            if(t < v[m]) {
                r = m - 1;
            }
            else{
                res = l;
                l = m + 1;
            }
        }
        return res;
    }

    Node* bst(Node* r, int t){
        if(!r || r->key == t)
            return r;

        if(r->key < t)
            return bst(r->right, t);

        return bst(r->left, t);
    }

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

```

```

// Your code goes here
vector<int> v;
inorder(root, v);
int i = bs(v, key);
pre == NULL;
suc = NULL;
if(v[i]!=key){
    if(i+1<v.size()) suc = bst(root, v[i+1]);
    pre = bst(root, v[i]);
}
else{
    if(i+1<v.size()) suc = bst(root, v[i+1]);
    if(i-1>=0) pre = bst(root, v[i-1]);
}
}

};


```

Check if a tree is a BST or not

[Approach – 1] Using specified range of Min and Max Values – O(n) Time and O(h) Space

```

// Helper function to check if a tree is BST within a given range
bool isBSTUtil(Node* node, int min, int max) {
    if (node == nullptr)
        return true;

    // If the current node's data
    // is not in the valid range, return false
    if (node->data < min || node->data > max)
        return false;

    // Recursively check the left and
    // right subtrees with updated ranges
    return isBSTUtil(node->left, min, node->data - 1) &&
           isBSTUtil(node->right, node->data + 1, max);
}

// Function to check if the entire binary tree is a BST

```

```

bool isBST(Node* root) {
    return isBSTUtil(root, INT_MIN, INT_MAX);
}

```

[Approach – 2] Using Inorder Traversal – O(n) Time and O(h) Space

The idea is to use inorder traversal of a binary search tree, in which the output values are sorted in ascending order. After generating the inorder traversal of the given binary tree, we can check if the values are sorted or not.

Note: We can avoid the use of an Auxiliary Array. While doing In-Order traversal, we can keep track of previously visited value. If the value of the currently visited node is less than the previous value, then the tree is not BST.

[Approach – 3] Using Morris Traversal – O(n) Time and O(1) Space

The idea is to use Morris Traversal for checking if a binary tree is a Binary Search Tree (BST) without using extra space for storing the inorder traversal.

```

// Function to check if the binary tree is a BST using Morris Traversal
bool isBST(Node* root) {
    Node* curr = root;
    Node* pre = nullptr;
    int prevValue = INT_MIN;

    while (curr != nullptr) {
        if (curr->left == nullptr) {

            // Process curr node
            if (curr->data <= prevValue) {

                // Not in ascending order
                return false;
            }
            prevValue = curr->data;
            curr = curr->right;
        } else {

            // Find the inorder predecessor of curr
            pre = curr->left;
            while (pre->right != nullptr && pre->right != curr) {
                pre = pre->right;
            }
        }
    }
}

```

```

        if (pre->right == nullptr) {

            // Create a temporary thread to the curr node
            pre->right = curr;
            curr = curr->left;
        } else {

            // Remove the temporary thread
            pre->right = nullptr;

            // Process the curr node
            if (curr->data <= prevValue) {

                // Not in ascending order
                return false;
            }
            prevValue = curr->data;
            curr = curr->right;
        }
    }

    return true;
}

```

Populate Inorder Successor for all nodes

Given a Binary Tree where each node has the following structure, write a function to populate the next pointer for all nodes. The next pointer for every node should be set to point to in-order successor.

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

```

class node {
public:
    int data;
    node* left;
}

```

```

    node* right;
    node* next;
};

/* Set next of p and all descendants of p
by traversing them in reverse Inorder */
void populateNext(node* p)
{
    // The first visited node will be the
    // rightmost node next of the rightmost
    // node will be NULL
    static node* next = NULL;

    if (p) {
        // First set the next pointer
        // in right subtree
        populateNext(p->right);

        // Set the next as previously visited
        // node in reverse Inorder
        p->next = next;

        // Change the prev for subsequent node
        next = p;

        // Finally, set the next pointer in
        // left subtree
        populateNext(p->left);
    }
}

/* UTILITY FUNCTIONS */
/* Helper function that allocates a new
node with the given data and NULL left
and right pointers. */
node* newnode(int data)
{
    node* Node = new node();
    Node->data = data;
    Node->left = NULL;
    Node->right = NULL;
    Node->next = NULL;
}

```

```

        return (Node);
    }

// Driver Code
int main()
{
    /* Constructed binary tree is
       10
      / \
     8  12
      /
     3
    */
    node* root = newnode(10);
    root->left = newnode(8);
    root->right = newnode(12);
    root->left->left = newnode(3);

    // Populates nextRight pointer in all nodes
    populateNext(root);

    // Let us see the populated values
    node* ptr = root->left->left;
    while (ptr) {
        // -1 is printed if there is no successor
        cout << "Next of " << ptr->data << " is "
            << (ptr->next ? ptr->next->data : -1) << endl;
        ptr = ptr->next;
    }

    return 0;
}

```

LCA in BST – Lowest Common Ancestor in Binary Search Tree

Using BST Properties (Recursive Approach) – O(h) Time and O(h) Space

In a Binary search tree, while traversing the tree from top to bottom the first node which lies in between the two numbers n1 and n2 is the LCA of the nodes, i.e. the first node n with the lowest depth which lies in between n1 and n2 ($n_1 \leq n \leq n_2$, assuming $n_1 < n_2$).

So just recursively traverse the BST , if node's value is greater than both n1 and n2 then our LCA lies in the left side of the node, if it is smaller than both n1 and n2, then LCA lies on the right side. Otherwise, the root is LCA (assuming that both n1 and n2 are present in BST).

```
// Function to find LCA of nodes n1 and n2, assuming
// both are present in the BST
Node* LCA(Node* root, Node* n1, Node* n2) {

    if (root == nullptr)
        return nullptr;

    // If both n1 and n2 are smaller than
    // root, go to left subtree
    if (root->data > n1->data && root->data > n2->data)
        return LCA(root->left, n1, n2);

    // If both n1 and n2 are greater than
    // root, go to right subtree
    if (root->data < n1->data && root->data < n2->data)
        return LCA(root->right, n1, n2);

    // If nodes n1 and n2 are on the opposite sides,
    // then root is the LCA
    return root;
}
```

Using BST Properties (Iterative Method) – O(h) Time and O(1) Space

The auxiliary space in the above method can be optimized by eliminating recursion. Below is the iterative implementation of this approach.

```
// Function to find LCA of n1 and n2, assuming
// that both nodes n1 and n2 are present in BST
Node* LCA(Node* root, Node* n1, Node* n2) {

    while (root != nullptr) {
```

```

    // If both n1 and n2 are smaller than root,
    // then LCA lies in left
    if (root->data > n1->data && root->data > n2->data)
        root = root->left;

    // If both n1 and n2 are greater than root,
    // then LCA lies in right
    else if (root->data < n1->data && root->data < n2->data)
        root = root->right;

    // Else Ancestor is found
    else
        break;
}

return root;
}

```

Construct BST from Preorder Traversal

Naive – One by One Insert

1. Create an empty BST
 2. Traverse through the given pre array and one by one insert every item into the BST
- Time Complexity: O(n²)

Better – Find the first Greater than Root

```

// A recursive function to construct Full BST from pre[]
// without passing preIndex.
Node* constructUtil(vector<int>& pre, int low, int high) {
    if (low > high)
        return nullptr;

    // Create the root node with the first element
    // in the current range
    Node* root = new Node(pre[low]);

```

```

if (low == high)
    return root;

// Find the first element greater than root to
// divide the array
int i;
for (i = low + 1; i <= high; ++i)
    if (pre[i] > root->data)
        break;

// Recursively construct left and right subtrees
root->left = constructUtil(pre, low + 1, i - 1);
root->right = constructUtil(pre, i, high);

return root;
}

// Constructs the BST from the preorder traversal
Node* construct(vector<int>& pre) {
    return constructUtil(pre, 0, pre.size() - 1);
}

// Utility function to print inorder traversal of the BST
void printInorder(Node* root) {
    if (root == nullptr)
        return;
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

// Driver code
int main() {
    vector<int> pre = {10, 5, 1, 7, 40, 50};
    Node* root = construct(pre);
    printInorder(root);
    return 0;
}

```

Time Complexity: $O(n^2)$

Auxiliary Space: $O(n)$

Efficient – Pass Range in Recursion

The trick is to set a range {min .. max} for every node. We initialize range as [-inf, +inf]. We begin with the first element of the preorder traversal, create a node with the given key. Now moving forward, we set the range as [-inf, key] for left subtree and [key, inf] for right subtree.

Follow the below steps to solve the problem:

- Initialize the range as {-inf , +inf}
- The first node will definitely be in range, so create a root node.
- To construct the left subtree, set the range as {-inf, root.key} and for right subtree as [root.key, +inf]

Below is the implementation of the above approach:

```
// A recursive function to construct BST from pre[].
// idx is used to keep track of index in pre[].
Node *constructUtil(vector<int> &pre, int &idx,
                     int min, int max)
{
    if (idx >= pre.size())
        return nullptr;

    int key = pre[idx];
    if (key <= min || key >= max)
        return nullptr;

    // If current element of pre[] is in range,
    // then only it is part of the current subtree
    Node *root = new Node(key);
    idx++;

    // All nodes in range {min .. key}
    // go to the left subtree
    if (idx < pre.size())
        root->left = constructUtil(pre, idx, min, key);

    // All nodes in range {key .. max}
    // go to the right subtree
    if (idx < pre.size())
        root->right = constructUtil(pre, idx, key, max);

    return root;
}
```

```

// The main function to construct BST from
// given preorder traversal.
Node *constructTree(vector<int> &pre)
{
    int idx = 0;
    return constructUtil(pre, idx, INT_MIN, INT_MAX);
}

// A utility function to print inorder
// traversal of a Binary Tree
void inorder(Node *node)
{
    if (node == nullptr)
        return;
    inorder(node->left);
    cout << node->data << " ";
    inorder(node->right);
}

// Driver code
int main()
{
    vector<int> pre = {10, 5, 1, 7, 40, 50};
    Node *root = constructTree(pre);
    inorder(root);
    return 0;
}

```

Binary Tree to Binary Search Tree Conversion

The idea is to recursively traverse the binary tree and store the nodes in an array. Sort the array, and perform in-order traversal of the tree and update the value of each node to the corresponding value in tree.

```

// Inorder traversal to store the nodes in a vector
void inorder(Node* root, vector<int>& nodes) {
    if (root == nullptr) {
        return;
    }
    inorder(root->left, nodes);
    nodes.push_back(root->data);
    inorder(root->right, nodes);
}

```

```
    nodes.push_back(root->data);
    inorder(root->right, nodes);
}

// Inorder traversal to convert tree
// to BST.
void constructBST(Node* root, vector<int> nodes, int& index) {
    if (root == nullptr) return;

    constructBST(root->left, nodes, index);

    // Update root value
    root->data = nodes[index++];

    constructBST(root->right, nodes, index);
}

// Function to convert a binary tree to a binary search tree
Node* binaryTreeToBST(Node* root) {
    vector<int> nodes;
    inorder(root, nodes);

    // sort the nodes
    sort(nodes.begin(), nodes.end());

    int index = 0;
    constructBST(root, nodes, index);
    return root;
}

// Function to print the inorder traversal of a binary tree
void printInorder(Node* root) {
    if (root == NULL) {
        return;
    }
    printInorder(root->left);
    cout << root->data << " ";
    printInorder(root->right);
}

int main() {

    // Creating the tree
```

```

//          10
//          /   \
//          2     7
//         /   \
//        8     4
Node* root = new Node(10);
root->left = new Node(2);
root->right = new Node(7);
root->left->left = new Node(8);
root->left->right = new Node(4);

Node* ans = binaryTreeToBST(root);
printInorder(ans);

return 0;
}

```

Time Complexity: O(nlogn), for sorting the array.

Auxiliary Space: O(n), for storing nodes in an array.

Balance a Binary Search Tree

The idea is to store the elements of the tree in an array using inorder traversal. Inorder traversal of a BST produces a sorted array. Once we have a sorted array, recursively construct a balanced BST by picking the middle element of the array as the root for each subtree.

Time Complexity: O(n)

Auxiliary space: O(n)

```

// Inorder traversal to store elements of the
// tree in sorted order
void storeInorder(Node* root, vector<int>& nodes) {
    if (root == nullptr)
        return;

    // Traverse the left subtree
    storeInorder(root->left, nodes);

    // Store the node data
    nodes.push_back(root->data);

    // Traverse the right subtree
    storeInorder(root->right, nodes);
}

```

```

        storeInorder(root->right, nodes);
    }

// Function to build a balanced BST from a sorted array
Node* buildBalancedTree(vector<int>& nodes, int start, int end) {

    // Base case
    if (start > end)
        return nullptr;

    // Get the middle element and make it the root
    int mid = (start + end) / 2;
    Node* root = new Node(nodes[mid]);

    // Recursively build the left and right subtrees
    root->left = buildBalancedTree(nodes, start, mid - 1);
    root->right = buildBalancedTree(nodes, mid + 1, end);

    return root;
}

// Function to balance a BST
Node* balanceBST(Node* root) {
    vector<int> nodes;

    // Store the nodes in sorted order
    storeInorder(root, nodes);

    // Build the balanced tree from the sorted nodes
    return buildBalancedTree(nodes, 0, nodes.size() - 1);
}

```

Merge Two Balanced Binary Search Trees

Method 1 (Insert elements of the first tree to the second):

Method 2 (Merge Inorder Traversals):

Do inorder traversal of the first tree and store the traversal in one temp array arr1[]. This step takes O(m) time.

Do inorder traversal of the second tree and store the traversal in another temp array arr2[]. This step takes $O(n)$ time.

The arrays created in steps 1 and 2 are sorted arrays. Merge the two sorted arrays into one array of size $m + n$. This step takes $O(m+n)$ time.

Construct a balanced tree from the merged array using the technique discussed in this post.

This step takes $O(m+n)$ time.

K'th Largest element in BST using constant extra space

The idea is to use Reverse Morris Traversal which is based on Threaded Binary Trees.

Threaded binary trees use the NULL pointers to store the successor and predecessor information which helps us to utilize the wasted memory by those NULL pointers.

The special thing about Morris traversal is that we can do Inorder traversal without using stack or recursion which saves us memory consumed by stack or recursion call stack. Reverse Morris traversal is just the reverse of Morris traversal which is majorly used to do Reverse Inorder traversal with constant $O(1)$ extra memory consumed as it does not uses any Stack or Recursion.

To find Kth largest element in a Binary search tree, the simplest logic is to do reverse inorder traversal and while doing reverse inorder traversal simply keep a count of number of Nodes visited. When the count becomes equal to k, we stop the traversal and print the data. It uses the fact that reverse inorder traversal will give us a list sorted in descending order.

```
// Function to perform Morris Traversal and
// return kth largest element
int kthLargest(Node* root, int k {

    // return -1 if root is null
    if (root == nullptr) return -1;

    Node* curr = root;
    int cnt = 0;

    while (curr != nullptr) {

        // if right tree does not exists,
```

```

// then increment the count, check
// count==k. Otherwise,
// set curr = curr->left
if (curr->right == nullptr) {
    cnt++;

    // return current Node
    // if cnt == k.
    if (cnt == k)
        return curr->data;

    curr = curr->left;
}
else {
    Node* succ = curr->right;

    // find the inorder successor
    while (succ->left != nullptr &&
           succ->left != curr) {
        succ = succ->left;
    }

    // create a linkage between succ and
    // curr
    if (succ->left == nullptr) {
        succ->left = curr;
        curr = curr->right;
    }

    // if succ->left = curr, it means
    // we have processed the right subtree,
    // and we can process curr node
    else {
        cnt++;

        // remove the link
        succ->left = nullptr;

        // return current Node
        // if cnt == k.
        if (cnt == k)
            return curr->data;
    }
}

```

```

        curr = curr->left;
    }
}

return -1;
}

int main() {

// Create a hard coded tree.
//           20
//         /   \
//        8     22
//       /   \
//      4   12
//     /   \
//    10   14

Node* root = new Node(20);
root->left = new Node(8);
root->right = new Node(22);
root->left->left = new Node(4);
root->left->right = new Node(12);
root->left->right->left = new Node(10);
root->left->right->right = new Node(14);

int k = 3;

cout << kthLargest(root, k) << endl;

return 0;
}

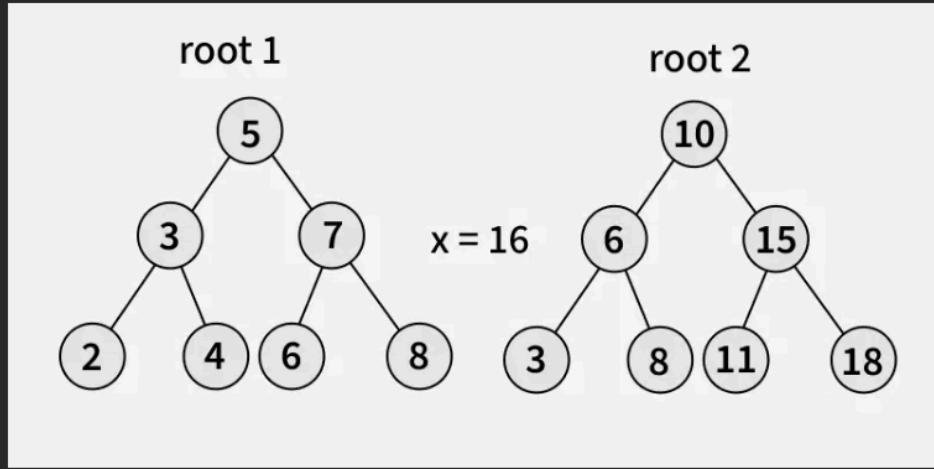
```

Find Kth smallest element in a BST

Same as above

Count pairs from two BSTs whose sum is equal to a given value x

Input :



Output: 3

Explanation: The pairs are: (5, 11), (6, 10) and (8, 8) whose sum is equal to x.

[Naive Approach] Using Recursive Method – $O(n_1 * n_2)$ Time and $O(h_1 + h_2)$ Space
The idea is traverse the first BST. For each node, find the value of $(x - \text{node})$ in the second BST. If the value exists, then increment the count.

```
bool findVal(Node* root, int x) {  
    if (root == nullptr) return false;  
  
    if (root->data == x) return true;  
    else if (root->data < x)  
        return findVal(root->right, x);  
    else  
        return findVal(root->left, x);  
}  
  
// Function to count pairs with sum equal to x  
int countPairs(Node* root1, Node* root2, int x) {  
  
    // base case  
    if (root1 == nullptr) return 0;  
  
    int ans = 0;  
  
    // If pair (root1.data, x-root1.data) exists,  
}
```

```

// then increment the ans.
if (findVal(root2, x-root1->data))
    ans++;

// Recursively check for left and right subtree.
ans += countPairs(root1->left, root2, x);
ans += countPairs(root1->right, root2, x);

return ans;
}

int main() {

    // BST1
    //   2
    // / \
    // 1  3
    Node* root1 = new Node(2);
    root1->left = new Node(1);
    root1->right = new Node(3);

    // BST2
    //   5
    // / \
    // 4  6
    Node* root2 = new Node(5);
    root2->left = new Node(4);
    root2->right = new Node(6);

    int x = 6;
    cout << countPairs(root1, root2, x);

    return 0;
}

```

[Expected Approach] Using Iterative method – $O(n_1 + n_2)$ Time and $O(h_1 + h_2)$ Space
The idea is to traverse the first BST from smallest to largest value iterative inorder traversal and traverse the second BST from largest to smallest value (reverse in-order). If the sum of nodes is equal to x , then increment the pair count and move both nodes to the next nodes. If the value is less than x , then move the node of first BST. Otherwise, move the node of second BST to the next node.

```
// Function to count pairs with sum equal to x
int countPairs(Node* root1, Node* root2, int x) {

    // if either of the tree is empty
    if (root1 == nullptr || root2 == nullptr)
        return 0;

    // stack 'st1' used for the inorder
    // traversal of BST 1
    // stack 'st2' used for the reverse
    // inorder traversal of BST 2
    stack<Node*> st1, st2;
    Node* top1, *top2;

    int count = 0;

    // the loop will break when either of two
    // traversals gets completed
    while (1) {

        // to find next node in inorder
        // traversal of BST 1
        while (root1 != nullptr) {
            st1.push(root1);
            root1 = root1->left;
        }

        // to find next node in reverse
        // inorder traversal of BST 2
        while (root2 != nullptr) {
            st2.push(root2);
            root2 = root2->right;
        }

        // if either gets empty then corresponding
        // tree traversal is completed
        if (st1.empty() || st2.empty())
            break;

        top1 = st1.top();
        top2 = st2.top();

        // if the sum of the node's is equal to 'x'
```

```

if ((top1->data + top2->data) == x) {

    count++;

    // pop nodes from the respective stacks
    st1.pop();
    st2.pop();

    // insert next possible node in the
    // respective stacks
    root1 = top1->right;
    root2 = top2->left;
}

// move to next possible node in the
// inorder traversal of BST 1
else if ((top1->data + top2->data) < x) {
    st1.pop();
    root1 = top1->right;
}

// move to next possible node in the
// reverse inorder traversal of BST 2
else {
    st2.pop();
    root2 = top2->left;
}

return count;
}

```

Find median of BST

Median Of BST using Morris Inorder Traversal:

The idea is based on K'th smallest element in BST using O(1) Extra Space.

The task is very simple if we are allowed to use extra space but Inorder to traversal using recursion and stack both use Space which is not allowed here. So, the solution is to do Morris Inorder traversal as it doesn't require extra space.

```
int countNodes(struct Node *root)
{
    struct Node *current, *pre;

    // Initialise count of nodes as 0
    int count = 0;

    if (root == NULL)
        return count;

    current = root;
    while (current != NULL)
    {
        if (current->left == NULL)
        {
            // Count node if its left is NULL
            count++;

            // Move to its right
            current = current->right;
        }
        else
        {
            /* Find the inorder predecessor of current */
            pre = current->left;

            while (pre->right != NULL &&
                   pre->right != current)
                pre = pre->right;

            /* Make current as right child of its
               inorder predecessor */
            if (pre->right == NULL)
            {
                pre->right = current;
                current = current->left;
            }

            /* Revert the changes made in if part to
               restore the original tree structure */
            if (current->left == pre)
                current->left = pre->right;
            else
                current->right = pre->right;
        }
    }
}
```

```

        restore the original tree i.e., fix
        the right child of predecessor */
    else
    {
        pre->right = NULL;

        // Increment count if the current
        // node is to be visited
        count++;
        current = current->right;
    } /* End of if condition pre->right == NULL */
} /* End of if condition current->left == NULL*/
} /* End of while */

return count;
}

/* Function to find median in O(n) time and O(1) space
   using Morris Inorder traversal*/
int findMedian(struct Node *root)
{
    if (root == NULL)
        return 0;

    int count = counNodes(root);
    int currCount = 0;
    struct Node *current = root, *pre, *prev;

    while (current != NULL)
    {
        if (current->left == NULL)
        {
            // count current node
            currCount++;

            // check if current node is the median
            // Odd case
            if (count % 2 != 0 && currCount == (count+1)/2)
                return current->data;

            // Even case
            else if (count % 2 == 0 && currCount == (count/2)+1)

```

```

        return (prev->data + current->data)/2;

    // Update prev for even no. of nodes
    prev = current;

    //Move to the right
    current = current->right;
}
else
{
    /* Find the inorder predecessor of current */
    pre = current->left;
    while (pre->right != NULL && pre->right != current)
        pre = pre->right;

    /* Make current as right child of its inorder predecessor */
    if (pre->right == NULL)
    {
        pre->right = current;
        current = current->left;
    }

    /* Revert the changes made in if part to restore the original
       tree i.e., fix the right child of predecessor */
    else
    {
        pre->right = NULL;

        prev = pre;

        // Count current node
        currCount++;

        // Check if the current node is the median
        if (count % 2 != 0 && currCount == (count+1)/2 )
            return current->data;

        else if (count%2==0 && currCount == (count/2)+1)
            return (prev->data+current->data)/2;

        // update prev node for the case of even
        // no. of nodes
        prev = current;
}

```

```

        current = current->right;

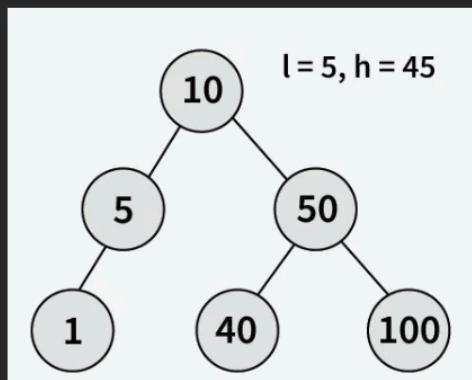
    } /* End of if condition pre->right == NULL */
} /* End of if condition current->left == NULL*/
} /* End of while */
}

```

Count BST nodes that lie in a given range

[Expected Approach] Using Recursion – O(n) Time and O(h) Space

The idea is traverse the given binary search tree starting from root. For every node check if this node lies in range, if yes, then add 1 to result and recursively check for both of its children. If current node is smaller than low value of range, then recur for right child, else recur for left child.



Output: 3

Explanation: There are three nodes in range $[5, 45] = 5, 10 \text{ and } 40$.

```

// Returns count of nodes in BST in range [l, h]
int getCount(Node *root, int l, int h) {

    // Base case
    if (root == nullptr) return 0;

    // If current node is in range, then
    // include it in count and recur for
    // left and right children of it
    if (root->data <= h && root->data >= l)
        return 1 + getCount(root->left, l, h) +
               getCount(root->right, l, h);
}

```

```

    // If current node is smaller than low,
    // then recur for right child
    else if (root->data < l)
        return getCount(root->right, l, h);

    // Else recur for left child
    else return getCount(root->left, l, h);
}

```

Replace every element with the least greater element on its right

Given an array of integers, replace every element with the least greater element on its right side in the array. If there are no greater elements on the right side, replace it with -1.

Examples:

Input: [8, 58, 71, 18, 31, 32, 63, 92,
43, 3, 91, 93, 25, 80, 28]
Output: [18, 63, 80, 25, 32, 43, 80, 93,
80, 25, 93, -1, 28, -1, -1]

Time and space complexity: We insert each element in our set and find upper bound for each element using a loop so its time complexity is $O(n \log n)$. We are storing each element in our set so space complexity is $O(n)$

```

void solve(vector<int>& arr)
{
    set<int> s;
    for (int i = arr.size() - 1; i >= 0;
         i--) { // traversing the array backwards
        s.insert(arr[i]); // inserting the element into set
        auto it
            = s.upper_bound(arr[i]); // finding upper bound
        if (it == s.end())
            arr[i] = -1; // if upper_bound does not exist
                           // then -1
    }
}

```

```

        else
            arr[i] = *it; // if upper_bound exists, lets
                           // take it
    }
}

```

Given n appointments, find all conflicting appointments

Examples:

Input: appointments[] = { {1, 5} {3, 7}, {2, 6}, {10, 15}, {5, 6}, {4, 100} }

Output: Following are conflicting intervals

- [3,7] Conflicts with [1,5]
- [2,6] Conflicts with [1,5]
- [5,6] Conflicts with [3,7]
- [4,100] Conflicts with [1,5]

```

// C++ program to print all conflicting appointments in a
// given set of appointments
#include <bits/stdc++.h>
using namespace std;

// Structure to represent an interval
struct Interval
{
    int low, high;
};

// Structure to represent a node in Interval Search Tree
struct ITNode
{
    Interval *i; // 'i' could also be a normal variable
    int max;
    ITNode *left, *right;
};

// A utility function to create a new Interval Search Tree Node
ITNode * newNode(Interval i)
{

```

```

ITNode *temp = new ITNode;
temp->i = new Interval(i);
temp->max = i.high;
temp->left = temp->right = NULL;
return temp;
};

// A utility function to insert a new Interval Search Tree
// Node. This is similar to BST Insert. Here the low value
// of interval is used to maintain BST property
ITNode *insert(ITNode *root, Interval i)
{
    // Base case: Tree is empty, new node becomes root
    if (root == NULL)
        return newNode(i);

    // Get low value of interval at root
    int l = root->i->low;

    // If root's low value is smaller, then new interval
    // goes to left subtree
    if (i.low < l)
        root->left = insert(root->left, i);

    // Else, new node goes to right subtree.
    else
        root->right = insert(root->right, i);

    // Update the max value of this ancestor if needed
    if (root->max < i.high)
        root->max = i.high;

    return root;
}

// A utility function to check if given two intervals overlap
bool doOverlap(Interval i1, Interval i2)
{
    if (i1.low < i2.high && i2.low < i1.high)
        return true;
    return false;
}

```

```

// The main function that searches a given interval i
// in a given Interval Tree.
Interval *overlapSearch(ITNode *root, Interval i)
{
    // Base Case, tree is empty
    if (root == NULL) return NULL;

    // If given interval overlaps with root
    if (doOverlap(*(root->i), i))
        return root->i;

    // If left child of root is present and max of left child
    // is greater than or equal to given interval, then i may
    // overlap with an interval in left subtree
    if (root->left != NULL && root->left->max >= i.low)
        return overlapSearch(root->left, i);

    // Else interval can only overlap with right subtree
    return overlapSearch(root->right, i);
}

// This function prints all conflicting appointments in a given
// array of appointments.
void printConflicting(Interval appt[], int n)
{
    // Create an empty Interval Search Tree, add first
    // appointment
    ITNode *root = NULL;
    root = insert(root, appt[0]);

    // Process rest of the intervals
    for (int i=1; i<n; i++)
    {
        // If current appointment conflicts with any of the
        // existing intervals, print it
        Interval *res = overlapSearch(root, appt[i]);
        if (res != NULL)
            cout << "[" << appt[i].low << "," << appt[i].high
                << "] Conflicts with [" << res->low << ","
                << res->high << "]\n";

        // Insert this appointment
        root = insert(root, appt[i]);
    }
}

```

```

        }
    }

// Driver program to test above functions
int main()
{
    // Let us create interval tree shown in above figure
    Interval appt[] = { {1, 5}, {3, 7}, {2, 6}, {10, 15},
                        {5, 6}, {4, 100} };
    int n = sizeof(appt)/sizeof(appt[0]);
    cout << "Following are conflicting intervals\n";
    printConflicting(appt, n);
    return 0;
}

```

Check if an array can be Preorder of a BST

[Expected Approach – 1] Using Stack – O(n) Time and O(n) Space

The idea is to use a stack. This problem is similar to Next (or closest) Greater Element problem. Here we find the next greater element and after finding next greater, if we find a smaller element, then return false.

```

bool canRepresentBST(vector<int> &pre) {

    stack<int> s;

    // Initialize current root as minimum possible
    // value
    int root = INT_MIN;

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

        // If we find a node who is on right side
        // and smaller than root, return false
        if (pre[i] < root)
            return false;

        // If pre[i] is in right subtree of stack top,

```

```

    // Keep removing items smaller than pre[i]
    // and make the last removed item as new
    // root.
    while (!s.empty() && s.top() < pre[i]) {
        root = s.top();
        s.pop();
    }

    // At this point either stack is empty or
    // pre[i] is smaller than root, push pre[i]
    s.push(pre[i]);
}
return true;
}

```

[Expected Approach – 2] Without Using Stack – O(n) Time and O(n) Space

We can check if the given preorder traversal is valid or not for a BST without using stack. The idea is to use the similar concept of Building a BST using narrowing bound algorithm. We will recursively visit all nodes, but we will not build the nodes. In the end, if the complete array is not traversed, then that means that array can not represent the preorder traversal of any binary tree.

```

void buildBSTHelper(int& preIndex, int n, vector<int> &pre,
                    int min, int max) {

    // If we have processed all elements, return
    if (preIndex >= n)
        return;

    // If the current element lies between min and max,
    // it can be part of the BST
    if (min <= pre[preIndex] && pre[preIndex] <= max) {

        // Treat the current element as the root of
        // this subtree
        int rootData = pre[preIndex];
        preIndex++;

        buildBSTHelper(preIndex, n, pre, min, rootData);
        buildBSTHelper(preIndex, n, pre, rootData, max);
    }
}

```

```
    }

}

bool canRepresentBST(vector<int> &arr) {

    // Set the initial min and max values
    int min = INT_MIN, max = INT_MAX;

    // Start from the first element in
    // the array
    int preIndex = 0;
    int n = arr.size();

    buildBSTHelper(preIndex, n, arr, min, max);

    // If all elements are processed, it means the
    // array represents a valid BST
    return preIndex == n;
}
```

Check whether BST contains Dead End or not

```
Input :      8
           /   \
         5     9
        /   \
       2     7
      /
     1
```

Output : Yes

Explanation : Node "1" is the dead End because
after that we cant insert any element.

```
Input :      8
           /   \
         7     10
        /   /   \
       2   9   13
```

Output : Yes

Explanation : We can't insert any element at
node 9.

```
// Returns true if there is a dead end in tree,
// else false.
bool isDeadEndUtil(Node *root, int low, int high) {
    // Base case
    if (root == NULL)
        return false;
    // Check if current node falls within the range
    if (root->data >= low && root->data <= high)
        return true;
    // Recur for left and right subtrees
    return isDeadEndUtil(root->left, low, root->data - 1) ||
    isDeadEndUtil(root->right, root->data + 1, high);
}
bool isDeadEnd(Node *root) {
    return isDeadEndUtil(root, 1, INT_MAX);
```

```
}
```

```
void findallNodes(Node* root,map<int,int> &allnodes)
{
    if(root == NULL)
        return ;

    allnodes[root->data] = 1;
    findallNodes(root->left,allnodes);
    findallNodes(root->right,allnodes);
}

bool check(Node* root,map<int,int> &allnodes)
{
    if(root == NULL)
        return false;

    if(root->left == NULL and root->right == NULL)
    {
        int pre = root->data - 1;
        int next = root->data + 1;

        if(allnodes.find(pre) != allnodes.end() and allnodes.find(next) != allnodes.end())
            return true;
    }

    return check(root->left,allnodes) or check(root->right,allnodes);
}

bool isDeadEnd(Node *root)
{
    // Base case
    if (root == NULL)
        return false ;
    map<int,int> allnodes;
    // adding 0 for handling the exception of node having data = 1
    allnodes[0] = 1;
    findallNodes(root,allnodes);

    return check(root,allnodes);
}
```

Largest BST Subtree – Simple Implementation

Given a Binary Tree, write a function that returns the size of the largest subtree which is also a Binary Search Tree (BST). If the complete Binary Tree is BST, then return the size (number of nodes) of the whole tree.

Input:

```
 5
 / \
 2   4
 / \
1   3
```

Output: 3

The following subtree is the maximum size BST subtree

```
 2
 / \
1   3
```

Input:

```
 50
 / \
30   60
/ \ / \
5 20 45 70
       / \
      65  80
```

Output: 5

The following subtree is the maximum size BST subtree

```
 60
 / \
45  70
 / \
65  80
```

```
// Returns a vector of size 3: {min, max, size of largest BST}
// Vector {INT_MAX, INT_MIN, 0} is returned for an empty tree
vector<int> largestBSTUtil(Node *root) {
    if (!root)
        return {INT_MAX, INT_MIN, 0};

    if (!root->left && !root->right)
        return {root->data, root->data, 1};

    vector<int> left = largestBSTUtil(root->left);
    vector<int> right = largestBSTUtil(root->right);

    vector<int> ans(3);

    // If the current subtree rooted at root is a BST
    if (left[1] < root->data && right[0] > root->data) {
        ans[0] = min(left[0], root->data);
```

```

        ans[1] = max(right[1], root->data);
        ans[2] = left[2] + right[2] + 1;
        return ans;
    }

    // If the subtree is not a BST
    ans[0] = INT_MIN;
    ans[1] = INT_MAX;
    ans[2] = max(left[2], right[2]);

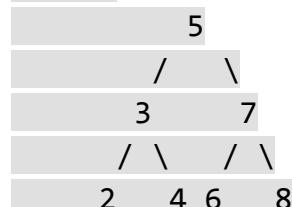
    return ans;
}

```

Flatten BST to sorted list | Increasing order

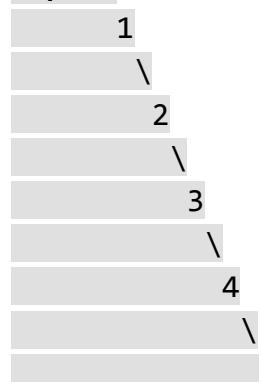
Given a binary search tree, the task is to flatten it to a sorted list. Precisely, the value of each node must be lesser than the values of all the nodes at its right, and its left node must be NULL after flattening. We must do it in $O(H)$ extra space where 'H' is the height of BST.

Input:



Output: 2 3 4 5 6 7 8

Input:



Output: 1 2 3 4 5

Approach: A simple approach will be to recreate the BST from its in-order traversal. This will take O(N) extra space where N is the number of nodes in BST.

Another Approach:

```
// Function to print flattened
// binary Tree
void print(node* parent)
{
    node* curr = parent;
    while (curr != NULL)
        cout << curr->data << " ", curr = curr->right;
}

// Function to perform in-order traversal
// recursively
void inorder(node* curr, node*& prev)
{
    // Base case
    if (curr == NULL)
        return;
    inorder(curr->left, prev);
    prev->left = NULL;
    prev->right = curr;
    prev = curr;
    inorder(curr->right, prev);
}

// Function to flatten binary tree using
// level order traversal
node* flatten(node* parent)
{
    // Dummy node
    node* dummy = new node(-1);

    // Pointer to previous element
    node* prev = dummy;

    // Calling in-order traversal
    inorder(parent, prev);

    prev->left = NULL;
    prev->right = NULL;
    node* ret = dummy->right;
```

```
// Delete dummy node
delete dummy;
return ret;
}

// Dri
```

Graphs:

Clone the Graph:

```
/*
// Definition for a Node.
class Node {
public:
    int val;
    vector<Node*> neighbors;
    Node() {
        val = 0;
        neighbors = vector<Node*>();
    }
    Node(int _val) {
        val = _val;
        neighbors = vector<Node*>();
    }
    Node(int _val, vector<Node*> _neighbors) {
        val = _val;
        neighbors = _neighbors;
    }
};
*/
class Solution {
public:
    Node* copy(Node* original, vector<Node*> &vis ){
        if(vis[original->val]==NULL){
            Node *n = new Node;
            n->val = original->val;
            vis[n->val]=n;
            for(auto x: original->neighbors){
                Node* k =copy(x,vis);
                n->neighbors.push_back(k);
            }
            return n;
        }
        else{
            return vis[original->val];
        }
    }
};
```

```
        }
    }

Node* cloneGraph(Node* node) {
    if(node==NULL) return node;
    Node* ans;
    vector<Node*> vis(101,NULL);
    ans=copy(node,vis);
    return ans;
}

};
```

```
class Solution {
public:
    Node* cloneGraph(Node* node) {
        map<Node*, Node*> oldToNew;
        return dfs(node, oldToNew);
    }

    Node* dfs(Node* node, map<Node*, Node*>& oldToNew) {
        if (node == nullptr) {
            return nullptr;
        }

        if (oldToNew.count(node)) {
            return oldToNew[node];
        }

        Node* copy = new Node(node->val);
        oldToNew[node] = copy;

        for (Node* nei : node->neighbors) {
            copy->neighbors.push_back(dfs(nei, oldToNew));
        }

        return copy;
    }
};
```

DFS:

```
#include<bits/stdc++.h>
using namespace std;

typedef long long ll;

void dfs(vector<ll> adj[], bool vis[], ll s) {
    vis[s] = true;
    cout << s << " ";
    for (ll i = 0; i < adj[s].size(); i++) {
        if (vis[adj[s][i]] == false) {
            dfs(adj, vis, adj[s][i]);
        }
    }
}
/*
    you can also write
    ll n = size of list
;    ll visited[n]={};
    vector<ll> list[n];
    void dfs(int s){
        // s =source
        if(visited[s]) return;
        visited[s]=1;
        for(auto u: list[s]){
            dfs(s);
        }
    }
*/
int main() {
    //initialize
    ll n = 7;
    vector<ll> adj[n];

    //add edge adjacency list
    adj[0].push_back(4);
    adj[0].push_back(1);
    adj[2].push_back(1);
    adj[2].push_back(3);
    adj[5].push_back(4);
    adj[6].push_back(4);
```

```

adj[4].push_back(0);
adj[1].push_back(0);
adj[1].push_back(2);
adj[3].push_back(2);
adj[4].push_back(5);
adj[4].push_back(6);

//s=source,vis[] = visited node array
ll s = 0;
bool vis[n];
memset(vis, false, sizeof(vis));
vis[s] = true;

cout << "Source is: " << s << "\n";

dfs(adj, vis, s);

//for disconnected components
for (ll i = 0; i < n; i++) {
    if (vis[i] == false) {
        dfs(adj, vis, i);
    }
}

return 0;
}

```

BFS:

```

void bfs(vector<ll> adj[], bool vis[], ll s) {
    queue<ll> q;
    q.push(s);
    while (!q.empty()) {
        ll x = q.front();
        cout << x << " ";
        q.pop();
        for (ll i = 0; i < adj[x].size(); i++) {
            if (vis[adj[x][i]] == false) {
                vis[adj[x][i]] = true;
                q.push(adj[x][i]);
            }
        }
    }
}

```

```
        }
    }
}
```

BFS Detect Cycle in Undirected Graph:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool detect(int src, vector<int> adj[], int vis[]) {
        vis[src] = 1;
        // store <source node, parent node>
        queue<pair<int,int>> q;
        q.push({src, -1});
        // traverse until queue is not empty
        while(!q.empty()) {
            int node = q.front().first;
            int parent = q.front().second;
            q.pop();

            // go to all adjacent nodes
            for(auto adjacentNode: adj[node]) {
                // if adjacent node is unvisited
                if(!vis[adjacentNode]) {
                    vis[adjacentNode] = 1;
                    q.push({adjacentNode, node});
                }
                // if adjacent node is visited and is not it's
                // own parent node
                else if(parent != adjacentNode) {
                    // yes it is a cycle

```

```

        return true;
    }
}
// there's no cycle
return false;
}
public:
// Function to detect cycle in an undirected graph.
bool isCycle(int V, vector<int> adj[]) {
    // initialise them as unvisited
    int vis[V] = {0};
    for(int i = 0;i<V;i++) {
        if(!vis[i]) {
            if(detect(i, adj, vis)) return true;
        }
    }
    return false;
}
};

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}

```

DFS Detect a Cycle in Undirected Graph:

```
#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfs(int node, int parent, int vis[], vector<int>
adj[]) {
        vis[node] = 1;
        // visit adjacent nodes
        for(auto adjacentNode: adj[node]) {
            // unvisited adjacent node
            if(!vis[adjacentNode]) {
                if(dfs(adjacentNode, node, vis, adj) == true)
                    return true;
            }
            // visited node but not a parent node
            else if(adjacentNode != parent) return true;
        }
        return false;
    }
public:
    // Function to detect cycle in an undirected graph.
    bool isCycle(int V, vector<int> adj[]) {
        int vis[V] = {0};
        // for graph with connected components
        for(int i = 0;i<V;i++) {
            if(!vis[i]) {
                if(dfs(i, -1, vis, adj) == true) return true;
            }
        }
        return false;
    }
};
```

```

int main() {

    // V = 4, E = 2
    vector<int> adj[4] = {{}, {2}, {1, 3}, {2}};
    Solution obj;
    bool ans = obj.isCycle(4, adj);
    if (ans)
        cout << "1\n";
    else
        cout << "0\n";
    return 0;
}

```

DFS Find Cycle Directed Graph:

```

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[],
    int pathVis[]) {
        vis[node] = 1;
        pathVis[node] = 1;

        // traverse for adjacent nodes
        for (auto it : adj[node]) {
            // when the node is not visited
            if (!vis[it]) {
                if (dfsCheck(it, adj, vis, pathVis) ==
true)
                    return true;
            }
        }
    }
}

```

```

        // if the node has been previously visited
        // but it has to be visited on the same path
        else if (pathVis[it]) {
            return true;
        }
    }

    pathVis[node] = 0;
    return false;
}
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int pathVis[V] = {0};

        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                if (dfsCheck(i, adj, vis, pathVis) ==
true) return true;
            }
        }
        return false;
    }
};

int main() {

    // V = 11, E = 11;
    vector<int> adj[11] = {{}, {2}, {3}, {4, 7}, {5}, {6},
{}, {5}, {9}, {10}, {8}};
    int V = 11;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);
}

```

```
if (ans)0.          cout << "True\n";
else
    cout << "False\n";

return 0;
}

#include <bits/stdc++.h>
using namespace std;

class Solution {
private:
    bool dfsCheck(int node, vector<int> adj[], int vis[],
int pathVis[]) {
        vis[node] = 1;
        pathVis[node] = 1;

        // traverse for adjacent nodes
        for (auto it : adj[node]) {
            // when the node is not visited
            if (!vis[it]) {
                if (dfsCheck(it, adj, vis, pathVis) ==
true)
                    return true;
            }
            // if the node has been previously visited
            // but it has to be visited on the same path
            else if (pathVis[it]) {
                return true;
            }
        }

        pathVis[node] = 0;
        return false;
    }
}
```

```
public:
    // Function to detect cycle in a directed graph.
    bool isCyclic(int V, vector<int> adj[]) {
        int vis[V] = {0};
        int pathVis[V] = {0};

        for (int i = 0; i < V; i++) {
            if (!vis[i]) {
                if (dfsCheck(i, adj, vis, pathVis) ==
true) return true;
            }
        }
        return false;
    }
};

int main() {

    // V = 11, E = 11;
    vector<int> adj[11] = {{}, {2}, {3}, {4, 7}, {5}, {6},
{}, {5}, {9}, {10}, {8}};
    int V = 11;
    Solution obj;
    bool ans = obj.isCyclic(V, adj);

    if (ans)
        cout << "True\n";
    else
        cout << "False\n";

    return 0;
}
```

BFS Directed Detect Cycle:

```
bool isCycle(vector<ll> adj[], ll n) {
    //cout << "count";
    vector<ll> incomingEdges(n + 1, 0);

    for (ll i = 0; i < n; i++) {
        for (ll j = 0; j < adj[i].size(); j++)
            incomingEdges[adj[i][j]]++;
    }
    queue<ll> q;
    for (ll i = 0; i <= n; i++) {
        if (incomingEdges[i] == 0) {
            q.push(i);
            cout << "i " << i << endl;
        }
    }
    ll count = 0;
    // cout << "count";
    vector<ll> sequence;
    while (!q.empty()) {
        count++;
        int u = q.front();
        q.pop();
        sequence.push_back(u);
        for (ll i = 0; i < adj[u].size(); i++) {
            if (--incomingEdges[adj[u][i]] == 0) {
                q.push(i);
            }
        }
        cout << u << " ";
    }

    if (count != n)
        return true;
    else return false;
}
```

Kruskal MST:

How to find MST using Kruskal's algorithm?

Below are the steps for finding MST using Kruskal's algorithm:

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are $(V-1)$ edges in the spanning tree.

```
// C++ program for the above approach

#include <bits/stdc++.h>
using namespace std;

// DSU data structure
// path compression + rank by union
class DSU {
    int* parent;
    int* rank;

public:
    DSU(int n)
    {
        parent = new int[n];
        rank = new int[n];

        for (int i = 0; i < n; i++) {
            parent[i] = -1;
            rank[i] = 1;
        }
    }

    // Find function
    int find(int i)
    {
        if (parent[i] == -1)
            return i;

        return parent[i] = find(parent[i]);
    }

    // Union function
    void unite(int x, int y)
    {
```

```

        int s1 = find(x);
        int s2 = find(y);

        if (s1 != s2) {
            if (rank[s1] < rank[s2]) {
                parent[s1] = s2;
            }
            else if (rank[s1] > rank[s2]) {
                parent[s2] = s1;
            }
            else {
                parent[s2] = s1;
                rank[s1] += 1;
            }
        }
    }

class Graph {
    vector<vector<int>> edgelist;
    int V;

public:
    Graph(int V) { this->V = V; }

    // Function to add edge in a graph
    void addEdge(int x, int y, int w)
    {
        edgelist.push_back({ w, x, y });
    }

    void kruskals_mst()
    {
        // Sort all edges
        sort(edgelist.begin(), edgelist.end());

        // Initialize the DSU
        DSU s(V);
        int ans = 0;
        cout << "Following are the edges in the "
             "constructed MST"
             << endl;
        for (auto edge : edgelist) {

```

```

        int w = edge[0];
        int x = edge[1];
        int y = edge[2];

        // Take this edge in MST if it does
        // not forms a cycle
        if (s.find(x) != s.find(y)) {
            s.unite(x, y);
            ans += w;
            cout << x << " -- " << y << " == " << w
                << endl;
        }
    }
    cout << "Minimum Cost Spanning Tree: " << ans;
}
};

// Driver code
int main()
{
    Graph g(4);
    g.addEdge(0, 1, 10);
    g.addEdge(1, 3, 15);
    g.addEdge(2, 3, 4);
    g.addEdge(2, 0, 6);
    g.addEdge(0, 3, 5);

    // Function call
    g.kruskals_mst();

    return 0;
}

```

Prim's MST Minimum Spanning Tree:

The algorithm starts with an empty spanning tree. The idea is to maintain two sets of vertices. The first set contains the vertices already included in the MST, and the other set contains the vertices not yet included. At every step, it considers all the edges that connect the two sets and

picks the minimum weight edge from these edges. After picking the edge, it moves the other endpoint of the edge to the set containing MST.

Step 1: Determine an arbitrary vertex as the starting vertex of the MST.

Step 2: Follow steps 3 to 5 till there are vertices that are not included in the MST (known as fringe vertex).

Step 3: Find edges connecting any tree vertex with the fringe vertices.

Step 4: Find the minimum among these edges.

Step 5: Add the chosen edge to the MST if it does not form any cycle.

Step 6: Return the MST and exit

How to implement Prim's Algorithm?

Follow the given steps to utilize the Prim's Algorithm mentioned above for finding MST of a graph:

Create a set mstSet that keeps track of vertices already included in MST.

Assign a key value to all vertices in the input graph. Initialize all key values as INFINITE. Assign the key value as 0 for the first vertex so that it is picked first.

While mstSet doesn't include all vertices

Pick a vertex u that is not there in mstSet and has a minimum key value.

Include u in the mstSet.

Update the key value of all adjacent vertices of u. To update the key values, iterate through all adjacent vertices.

For every adjacent vertex v, if the weight of edge u-v is less than the previous key value of v, update the key value as the weight of u-v.

```
// A C++ program for Prim's Minimum
// Spanning Tree (MST) algorithm. The program is
// for adjacency matrix representation of the graph

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 5

// A utility function to find the vertex with
// minimum key value, from the set of vertices
```

```

// not yet included in MST
int minKey(int key[], bool mstSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;

    return min_index;
}

// A utility function to print the
// constructed MST stored in parent[]
void printMST(int parent[], int graph[V][V])
{
    cout << "Edge \tWeight\n";
    for (int i = 1; i < V; i++)
        cout << parent[i] << " - " << i << " \t"
            << graph[i][parent[i]] << " \n";
}

// Function to construct and print MST for
// a graph represented using adjacency
// matrix representation
void primMST(int graph[V][V])
{
    // Array to store constructed MST
    int parent[V];

    // Key values used to pick minimum weight edge in cut
    int key[V];

    // To represent set of vertices included in MST
    bool mstSet[V];

    // Initialize all keys as INFINITE
    for (int i = 0; i < V; i++)
        key[i] = INT_MAX, mstSet[i] = false;

    // Always include first 1st vertex in MST.
    // Make key 0 so that this vertex is picked as first
}

```

```

// vertex.
key[0] = 0;

// First node is always root of MST
parent[0] = -1;

// The MST will have V vertices
for (int count = 0; count < V - 1; count++) {

    // Pick the minimum key vertex from the
    // set of vertices not yet included in MST
    int u = minKey(key, mstSet);

    // Add the picked vertex to the MST Set
    mstSet[u] = true;

    // Update key value and parent index of
    // the adjacent vertices of the picked vertex.
    // Consider only those vertices which are not
    // yet included in MST
    for (int v = 0; v < V; v++)

        // graph[u][v] is non zero only for adjacent
        // vertices of m mstSet[v] is false for vertices
        // not yet included in MST Update the key only
        // if graph[u][v] is smaller than key[v]
        if (graph[u][v] && mstSet[v] == false
            && graph[u][v] < key[v])
            parent[v] = u, key[v] = graph[u][v];
}

// Print the constructed MST
printMST(parent, graph);
}

// Driver's code
int main()
{
    int graph[V][V] = { { 0, 2, 0, 6, 0 },
                        { 2, 0, 3, 8, 5 },
                        { 0, 3, 0, 0, 7 },
                        { 6, 8, 0, 0, 9 },
                        { 0, 5, 7, 9, 0 } };
}

```

```

    // Print the solution
    primMST(graph);

    return 0;
}

// This code is contributed by rathbhupendra

```

Single Source Dijkstra's Algorithm:

The idea is to generate a SPT (shortest path tree) with a given source as a root. Maintain an Adjacency Matrix with two sets,

one set contains vertices included in the shortest-path tree,

other set includes vertices not yet included in the shortest-path tree.

At every step of the algorithm, find a vertex that is in the other set (set not yet included) and has a minimum distance from the source.

Algorithm :

- Create a set **sptSet** (shortest path tree set) that keeps track of vertices included in the shortest path tree, i.e., whose minimum distance from the source is calculated and finalized. Initially, this set is empty.
- Assign a distance value to all vertices in the input graph. Initialize all distance values as **INFINITE** . Assign the distance value as 0 for the source vertex so that it is picked first.
- While **sptSet** doesn't include all vertices
 - Pick a vertex **u** that is not there in **sptSet** and has a minimum distance value.
 - Include **u** to **sptSet** .
 - Then update the distance value of all adjacent vertices of **u** .
 - To update the distance values, iterate through all adjacent vertices.
 - For every adjacent vertex **v**, if the sum of the distance value of **u** (from source) and weight of edge **u-v** , is less than the distance value of **v** , then update the distance value of **v** .

Note: We use a boolean array **sptSet[]** to represent the set of vertices included in **SPT** . If a value **sptSet[v]** is true, then vertex **v** is included in **SPT** , otherwise not. Array **dist[]** is used to store the shortest distance values of all vertices.

```

// C++ program for Dijkstra's single source shortest path
// algorithm. The program is for adjacency matrix
// representation of the graph
#include <iostream>
using namespace std;
#include <limits.h>

// Number of vertices in the graph
#define V 9

// A utility function to find the vertex with minimum
// distance value, from the set of vertices not yet included
// in shortest path tree
int minDistance(int dist[], bool sptSet[])
{
    // Initialize min value
    int min = INT_MAX, min_index;

    for (int v = 0; v < V; v++)
        if (sptSet[v] == false && dist[v] <= min)
            min = dist[v], min_index = v;

    return min_index;
}

// A utility function to print the constructed distance
// array
void printSolution(int dist[])
{
    cout << "Vertex \t Distance from Source" << endl;
    for (int i = 0; i < V; i++)
        cout << i << " \t\t\t\t" << dist[i] << endl;
}

// Function that implements Dijkstra's single source
// shortest path algorithm for a graph represented using
// adjacency matrix representation
void dijkstra(int graph[V][V], int src)
{
    int dist[V]; // The output array. dist[i] will hold the
                 // shortest
                 // distance from src to i

```

```

bool sptSet[V]; // sptSet[i] will be true if vertex i is
                 // included in shortest
// path tree or shortest distance from src to i is
// finalized

// Initialize all distances as INFINITE and stpSet[] as
// false
for (int i = 0; i < V; i++)
    dist[i] = INT_MAX, sptSet[i] = false;

// Distance of source vertex from itself is always 0
dist[src] = 0;

// Find shortest path for all vertices
for (int count = 0; count < V - 1; count++) {
    // Pick the minimum distance vertex from the set of
    // vertices not yet processed. u is always equal to
    // src in the first iteration.
    int u = minDistance(dist, sptSet);

    // Mark the picked vertex as processed
    sptSet[u] = true;

    // Update dist value of the adjacent vertices of the
    // picked vertex.
    for (int v = 0; v < V; v++)

        // Update dist[v] only if is not in sptSet,
        // there is an edge from u to v, and total
        // weight of path from src to v through u is
        // smaller than current value of dist[v]
        if (!sptSet[v] && graph[u][v]
            && dist[u] != INT_MAX
            && dist[u] + graph[u][v] < dist[v])
            dist[v] = dist[u] + graph[u][v];
    }

    // print the constructed distance array
    printSolution(dist);
}

// driver's code

```

```
int main()
{
    /* Let us create the example graph discussed above */
    int graph[V][V] = { { 0, 4, 0, 0, 0, 0, 0, 8, 0 },
                        { 4, 0, 8, 0, 0, 0, 0, 11, 0 },
                        { 0, 8, 0, 7, 0, 4, 0, 0, 2 },
                        { 0, 0, 7, 0, 9, 14, 0, 0, 0 },
                        { 0, 0, 0, 9, 0, 10, 0, 0, 0 },
                        { 0, 0, 4, 14, 10, 0, 2, 0, 0 },
                        { 0, 0, 0, 0, 0, 2, 0, 1, 6 },
                        { 8, 11, 0, 0, 0, 0, 1, 0, 7 },
                        { 0, 0, 2, 0, 0, 0, 6, 7, 0 } };

    // Function call
    dijkstra(graph, 0);

    return 0;
}

// This code is contributed by shivanisinghss2110
```

Topological Sort:

Advantages of Topological Sort:

- Helps in scheduling tasks or events based on dependencies.
- Detects cycles in a directed graph.
- Efficient for solving problems with precedence constraints.

Disadvantages of Topological Sort:

- Only applicable to directed acyclic graphs (DAGs), not suitable for cyclic graphs.
- May not be unique, multiple valid topological orderings can exist.

Applications of Topological Sort:

- Task scheduling and project management.
- In software deployment tools like [Makefile](#).
- Dependency resolution in package management systems.
- Determining the order of compilation in software build systems.
- Deadlock detection in operating systems.
- Course scheduling in universities.
- It is used to find [shortest paths in weighted directed acyclic graphs](#)

```
// BFS Using indegrees:  
void bfs_topological(vector<ll> adj[], vector<ll> &indegree) {  
    queue<ll> q;  
    for (ll i = 0; i < indegree.size(); i++) {  
        if (indegree[i] == 0) q.push(i);  
    }  
    while (!q.empty()) {  
        ll x = q.front();  
        q.pop();  
        cout << x << " ";  
        for (ll i = 0; i < adj[x].size(); i++) {  
            indegree[adj[x][i]]--;  
            if (indegree[adj[x][i]] == 0) {  
                q.push(adj[x][i]);  
            }  
        }  
    }  
}
```

Algorithm for Topological Sorting using DFS:

Here's a step-by-step algorithm for topological sorting using Depth First Search (DFS):

- Create a graph with **n** vertices and **m**-directed edges.
- Initialize a stack and a visited array of size **n**.
- For each unvisited vertex in the graph, do the following:
 - Call the DFS function with the vertex as the parameter.
 - In the DFS function, mark the vertex as visited and recursively call the DFS function for all unvisited neighbors of the vertex.
 - Once all the neighbors have been visited, push the vertex onto the stack.
- After all, vertices have been visited, pop elements from the stack and append them to the output list until the stack is empty.
- The resulting list is the topologically sorted order of the graph.

```
#include <bits/stdc++.h>
using namespace std;

// Function to perform DFS and topological sorting
void topologicalSortUtil(int v, vector<vector<int> >& adj,
                         vector<bool>& visited,
                         stack<int>& Stack)
{
    // Mark the current node as visited
    visited[v] = true;

    // Recur for all adjacent vertices
    for (int i : adj[v]) {
        if (!visited[i])
            topologicalSortUtil(i, adj, visited, Stack);
    }

    // Push current vertex to stack which stores the result
    Stack.push(v);
}

// Function to perform Topological Sort
void topologicalSort(vector<vector<int> >& adj, int V)
{
    stack<int> Stack; // Stack to store the result
    vector<bool> visited(V, false);
```

```

// Call the recursive helper function to store
// Topological Sort starting from all vertices one by
// one
for (int i = 0; i < V; i++) {
    if (!visited[i])
        topologicalSortUtil(i, adj, visited, Stack);
}

// Print contents of stack
while (!Stack.empty()) {
    cout << Stack.top() << " ";
    Stack.pop();
}
}

int main()
{
    // Number of nodes
    int V = 4;

    // Edges
    vector<vector<int> > edges
        = { { 0, 1 }, { 1, 2 }, { 3, 1 }, { 3, 2 } };

    // Graph represented as an adjacency list
    vector<vector<int> > adj(V);

    for (auto i : edges) {
        adj[i[0]].push_back(i[1]);
    }

    cout << "Topological sorting of the graph: ";
    topologicalSort(adj, V);

    return 0;
}

```

Rat in a MAZE:

Consider a rat placed at (0, 0) in a square matrix mat of order $n \times n$. 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 travel through it.

Note: In a path, no cell can be visited more than one time. If the source cell is 0, the rat cannot move to any other cell. In case of no path, return an empty list. The driver will output "-1" automatically.

```
class Solution {
public:
    int dx[4] = {1, -1, 0, 0};
    int dy[4] = {0, 0, -1, 1};
    string ch = "DULR";
    bool isValid(int r, int c, int n, vector<vector<int>> vis,
    vector<vector<int>> &g){
        return r>=0 && r<n && c>=0 && c<n && g[r][c]==1;
    }
    void solve(int r,int c, int n, vector<vector<int>> &g,
    vector<vector<int>>&vis, vector<string> &ans, string &s){
        if(r==n-1 && c==n-1){
            // cout<<"s2: "<<s<<endl;
            ans.push_back(s);
            return;
        }
        if(!isValid(r,c,n,vis, g)) return;
        g[r][c]=0;
        for(int i=0;i<4;i++){
            int x = r + dx[i];
            int y = c + dy[i];
            s = s + ch[i];
            // cout<<x<<y<<endl;
            solve(x,y,n,g,vis,ans,s);
            s = s.substr(0, s.size()-1);
            // cout<<"s2: "<<s<<endl;
        }
        g[r][c]=1;
    }
    vector<string> findPath(vector<vector<int>> &g) {
        // Your code goes here
    }
}
```

```

int n = g.size();

vector<string> ans;
if(g[0][0]==0 || g[n-1][n-1]==0) return ans;
vector<vector<int>> vis(n, vector<int>(n,0));
string s = "";
solve(0,0,n,g,vis,ans,s);
return ans;
}
};


```

Steps by Knight:

Given a square chessboard, the initial position of Knight and position of a target. Find out the minimum steps a Knight will take to reach the target position.

Note:

The initial and the target position coordinates of Knight have been given according to 1-base indexing.

```

int dx[8] = { -2, 2, -1, -1, -2, 2, 1, 1};
int dy[8] = {1, 1, -2, 2, -1, -1, -2, 2};

class Solution
{
public:
    //Function to find out minimum steps Knight needs to reach target
    position.
    bool isValid(int i, int j, int &N, vector<vector<bool>> &visited) {
        if (i < 0 || j < 0 || i >= N || j >= N || visited[i][j])
            return false;
        return true;
    }

    int minStepToReachTarget(vector<int>&KnightPos,
    vector<int>&TargetPos, int N)
    {


```

```

// Code here
queue<pair<int, pair<int, int>>> q; //cost,i,j
vector<vector<bool>> visited(N, vector<bool>(N, false));

int desx = TargetPos[0] - 1;
int desy = TargetPos[1] - 1;

int i = KnightPos[0] - 1;
int j = KnightPos[1] - 1;

q.push({0, {i, j}}); //cost,i,j
visited[i][j] = true;

while (!q.empty()) {
    auto z = q.front();
    q.pop();

    int cost = z.first;
    int x = z.second.first;
    int y = z.second.second;

    if (x == desx && y == desy)
        return cost;

    for (int ii = 0; ii < 8; ii++) {
        int tx = x + dx[ii];
        int ty = y + dy[ii];

        if (isValid(tx, ty, N, visited)) {
            q.push({cost + 1, {tx, ty}});
            visited[tx][ty] = true;
        }
    }
}

return -1;
}

```

Flood Fill Algo:

An image is represented by an $m \times n$ integer grid image where $\text{image}[i][j]$ represents the pixel value of the image.

You are also given three integers sr , sc , and color. You should perform a flood fill on the image starting from the pixel $\text{image}[sr][sc]$.

To perform a flood fill, consider the starting pixel, plus any pixels connected 4-directionally to the starting pixel of the same color as the starting pixel, plus any pixels connected 4-directionally to those pixels (also with the same color), and so on. Replace the color of all of the aforementioned pixels with color.

Return the modified image after performing the flood fill.

```
class Solution {
public:
    int dx[4]={1,-1,0,0};
    int dy[4]={0,0,-1,1};
    bool isval(int r,int c,int n,int m){
        return r>=0 and r<n and c>=0 and c<m;
    }
    void dfs(int r,int c, vector<vector<int>>&g, vector<vector<bool>>&vis, int col,int xxx){
        vis[r][c]=1;
        if(g[r][c]!=col) return;
        g[r][c]=xxx;
        int n = g.size();
        int m = g[0].size();
        for(int i=0;i<4;i++){
            int x = r+dx[i];
            int y = c+dy[i];
            if(isval(x,y,n,m)){
                if(!vis[x][y]){
                    dfs(x,y,g,vis,col,xxx);
                }
            }
        }
    }
    vector<vector<int>> floodFill(vector<vector<int>>& image, int sr, int sc, int c) {
        int n = image.size();
        int m = image[0].size();
        vector<vector<bool>> v(n,vector<bool>(m,0));
        dfs(sr,sc,image,v,image[sr][sc],c);
        return image;
    }
};
```

```
    }
};
```

Word Ladder ★

A **transformation sequence** from word `beginWord` to word `endWord` using a dictionary `wordList` is a sequence of words `beginWord` → s_1 → s_2 → ... → s_k such that:

- Every adjacent pair of words differs by a single letter.
- Every s_i for $1 \leq i \leq k$ is in `wordList`. Note that `beginWord` does not need to be in `wordList`.
- $s_k = endWord$

Given two words, `beginWord` and `endWord`, and a dictionary `wordList`, return *the number of words in the shortest transformation sequence* from `beginWord` to `endWord`, or `0` if no such sequence exists.

Example 1:

Input: `beginWord = "hit", endWord = "cog", wordList = ["hot", "dot", "dog", "lot", "log", "cog"]`

Output: 5

Explanation: One shortest transformation sequence is "hit" → "hot" → "dot" → "dog" → "cog", which is 5 words long.

Example 2:

Input: `beginWord = "hit", endWord = "cog", wordList = ["hot", "dot", "dog", "lot", "log"]`

Output: 0

Explanation: The `endWord` "cog" is not in `wordList`, therefore there is no valid transformation sequence.

```
class Solution {
public:
    int ladderLength(string bw, string ew, vector<string>& wl) {
        unordered_set<string> s;
        for(auto x: wl) s.insert(x);
        if(s.find(ew)==s.end()) return 0;
        unordered_set<string> vis;
        vis.insert(bw);
        queue<string> q;
        q.push(bw);
        int ans = 1;
        while(!q.empty()){
            int sz = q.size();
```

```

        for(int i=0;i<sz;i++){
            string word = q.front();
            q.pop();
            if(word == ew) return ans;
            cout<<"actual word: "<<word<<" ";
            for(int j=0;j<word.size();j++){
                string wrd = word;
                for(int k = 'a'; k<= 'z';k++){
                    wrd[j] = (char)k ;
                    cout<<wrd<<" ";
                    if(s.find(wrd)!=s.end() and
vis.find(wrd)==vis.end()){
                        q.push(wrd);
                        vis.insert(wrd);
                    }
                }
            cout<<endl;
        }
        ans++;
    }
    return 0;
}
};


```

Minimum time taken by each job to be completed given by a
Directed Acyclic Graph

<https://www.geeksforgeeks.org/minimum-time-taken-by-each-job-to-be-completed-given-by-a-directed-acyclic-graph/>

```

//{ Driver Code Starts
//Initial Template for C++

#include<bits/stdc++.h>
using namespace std;

// } Driver Code Ends
//User function Template for C++

```

```

class Solution{
public:
    vector<int> minimumTime(int n,vector<vector<int>> &adj,int m)
    {
        // code here
        vector<int> indeg(n+1,0);
        vector<vector<int>> al(n+1);
        for(int i=0;i<m;i++){
            indeg[adj[i][1]]++;
            al[adj[i][0]].push_back(adj[i][1]);
        }
        queue<int>q;
        for(int i=1;i<=n;i++){
            if(indeg[i]==0) q.push(i);
        }
        vector<int> ans(n);
        int lvl = 1;
        while(!q.empty()){
            int sz = q.size();
            for(int k=0;k<sz;k++){
                int x = q.front();
                ans[x-1]=lvl;
                q.pop();
                for(int i=0;i<al[x].size();i++){
                    indeg[al[x][i]]-=1;
                    if(indeg[al[x][i]]==0){
                        q.push(al[x][i]);
                    }
                }
            }
            lvl+=1;
        }
        return ans;
    }
};

```

Check if it is possible to finish all task from given dependencies (Course Schedule I)

<https://www.geeksforgeeks.org/find-whether-it-is-possible-to-finish-all-tasks-or-not-from-given-dependencies/>

```
// CPP program to check whether we can finish all
// tasks or not from given dependencies.
#include <bits/stdc++.h>
using namespace std;

// Returns adjacency list representation from a list
// of pairs.
vector<unordered_set<int>> make_graph(int numTasks,
                                             vector<pair<int, int>>& prerequisites)
{
    vector<unordered_set<int>> graph(numTasks);
    for (auto pre : prerequisites)
        graph[pre.second].insert(pre.first);
    return graph;
}

// A DFS based function to check if there is a cycle
// in the directed graph.
bool dfs_cycle(vector<unordered_set<int>>& graph, int node,
               vector<bool>& onpath, vector<bool>& visited)
{
    if (visited[node])
        return false;
    onpath[node] = visited[node] = true;
    for (int neigh : graph[node])
        if (onpath[neigh] || dfs_cycle(graph, neigh, onpath, visited))
            return true;
    return onpath[node] = false;
}

// Main function to check whether possible to finish all tasks or not
bool canFinish(int numTasks, vector<pair<int, int>>& prerequisites)
{
    vector<unordered_set<int>> graph = make_graph(numTasks,
prerequisites);
```

```

vector<bool> onpath(numTasks, false), visited(numTasks, false);
for (int i = 0; i < numTasks; i++)
    if (!visited[i] && dfs_cycle(graph, i, onpath, visited))
        return false;
return true;
}

int main()
{
    int numTasks = 4;

    vector<pair<int, int> > prerequisites;

    // for prerequisites: [[1, 0], [2, 1], [3, 2]]

    prerequisites.push_back(make_pair(1, 0));
    prerequisites.push_back(make_pair(2, 1));
    prerequisites.push_back(make_pair(3, 2));
    if (canFinish(numTasks, prerequisites)) {
        cout << "Possible to finish all tasks";
    }
    else {
        cout << "Impossible to finish all tasks";
    }

    return 0;
}

```

Number of Islands:

```

class Solution {
public:
    int dx[4]={1,-1,0,0};
    int dy[4]={0,0,-1,1};
    bool isval(int r,int c,int n,int m){
        return r>=0 && r<n && c>=0 && c<m;
    }
    void dfs(int r,int c,vector<vector<char>> &g){
        int n = g.size();
        int m = g[0].size();

```

```

// vis[r][c]=1;
g[r][c]='x';
for(int i=0;i<4;i++){
    int x = r+dx[i];
    int y = c+dy[i];
    if(isval(x,y,n,m)){
        if(g[x][y]=='1')
            dfs(x,y,g);
    }
}
int numIslands(vector<vector<char>>& g) {

    int n = g.size();
    int m = g[0].size();
    int ans =0;
    for(int i=0;i<n;i++){
        for(int j=0;j<m;j++){
            if(g[i][j]=='1'){
                dfs(i,j,g);
                ans++;
            }
        }
    }
    return ans;
}
};

```

Given a sorted Dictionary of an Alien Language, find order of characters

Given a sorted dictionary of an alien language having N words and k starting alphabets of standard dictionary. Find the order of characters in the alien language.

Note: Many orders may be possible for a particular test case, thus you may return any valid order and output will be 1 if the order of string returned by the function is correct else 0 denoting incorrect string returned.

Examples :

Input: n = 5, k = 4, dict = {"baa", "abcd", "abca", "cab", "cad"}

Output: 1

Explanation: Here order of characters is 'b', 'd', 'a', 'c' Note that words are sorted and in the given language "baa" comes before "abcd", therefore 'b' is before 'a' in output.
Similarly we can find other orders.

```
class Solution {
public:
    map<int, vector<int>> adj;
    map<int, int> id;
    unordered_set<int> aps;

    void compare(string a, string b){
        int as = a.size();
        int bs = b.size();
        for(int i=0;i<min(as,bs);i++){
            if(a[i]!=b[i]){
                adj[a[i]-'a'].push_back(b[i]-'a');
                id[b[i]-'a']++;
                aps.insert(a[i]-'a');
                aps.insert(b[i]-'a');
                return;
            }
        }
    }

    string findOrder(string dict[], int n, int k) {
        // code here
        for(int i=0;i<n-1;i++){
            compare(dict[i],dict[i+1]);
        }
        queue<int> q;
        for(auto gg: aps){
            if(id[gg] == 0){
                q.push(gg);
            }
        }
        string ans = "";
        while(!q.empty()){
            int x = q.front();
            q.pop();
            ans += char(97 + x);
            for(int i=0;i<adj[x].size();i++){
                id[adj[x][i]]--;
                if(id[adj[x][i]]==0){

```

```

        q.push(adj[x][i]);
    }
}
return ans;
}
};

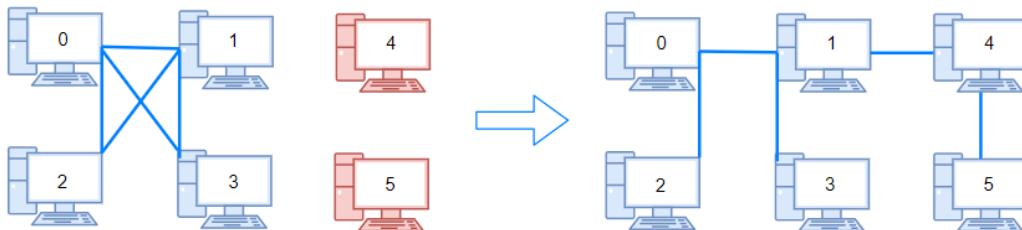
```

Making wired Connections

There are n computers numbered from 0 to $n - 1$ connected by ethernet cables connections forming a network where $\text{connections}[i] = [a_i, b_i]$ represents a connection between computers a_i and b_i . Any computer can reach any other computer directly or indirectly through the network.

You are given an initial computer network connections. You can extract certain cables between two directly connected computers, and place them between any pair of disconnected computers to make them directly connected.

Return the minimum number of times you need to do this in order to make all the computers connected. If it is not possible, return -1.



```

class Solution {
private:
    void dfs(vector<vector<int>> &adj, vector<bool> &visited, int src)
    {
        visited[src] = true;
        for(int i : adj[src])
            if(!visited[i])
                dfs(adj, visited, i);
    }
public:

```

```

int makeConnected(int n, vector<vector<int>>& connections) {

    if(connections.size() < n - 1)
        return -1;
    vector<vector<int>> adj(n);
    for(auto v : connections)
    {
        adj[v[0]].push_back(v[1]);
        adj[v[1]].push_back(v[0]);
    }
    vector<bool> visited(n, false);
    int components = 0;
    for(int i=0; i<n; i++)
        if(!visited[i])
        {
            dfs(adj, visited, i);
            components++;
        }
    return components - 1;
}
};

```

Total number of Spanning Trees in a Graph

If a graph is a complete graph with n vertices, then total number of spanning trees is $n(n-2)$ where n is the number of nodes in the graph. In complete graph, the task is equal to counting different labeled trees with n nodes for which have Cayley's formula.

What if graph is not complete?

Follow the given procedure:

STEP 1: Create Adjacency Matrix for the given graph.

STEP 2: Replace all the diagonal elements with the degree of nodes. For eg. element at (1,1) position of adjacency matrix will be replaced by the degree of node 1, element at (2,2) position of adjacency matrix will be replaced by the degree of node 2, and so on.

STEP 3: Replace all non-diagonal 1's with -1.

STEP 4: Calculate co-factor for any element.

STEP 5: The cofactor that you get is the total number of spanning tree for that graph.

Implement Bellman Ford Algorithm

Given a weighted graph with V vertices and E edges, and a source vertex src , find the shortest path from the source vertex to all vertices in the given graph. If a vertex cannot be reached from source vertex, mark its distance as 108.

```
Note: If a graph contains negative weight cycle, return -1.

vector<int> bellmanFord(int V, vector<vector<int>>& edges, int src) {

    // Initially distance from source to all
    // other vertices is not known(Infinte).
    vector<int> dist(V, 1e8);
    dist[src] = 0;

    // Relaxation of all the edges  $V$  times, not  $(V - 1)$  as we
    // need one additional relaxation to detect negative cycle
    for (int i = 0; i < V; i++) {
        for (vector<int> edge : edges) {
            int u = edge[0];
            int v = edge[1];
            int wt = edge[2];
            if (dist[u] != 1e8 && dist[u] + wt < dist[v]) {

                // If this is the  $V$ th relaxation, then there is
                // a negative cycle
                if(i == V - 1)
                    return {-1};

                // Update shortest distance to node v
                dist[v] = dist[u] + wt;
            }
        }
    }

    return dist;
}
```

Best Case: $O(E)$, when distance array after 1st and 2nd relaxation are same , we can simply stop further processing.

Average Case: $O(V^*E)$

Worst Case: $O(V^*E)$

Floyd Warshall Algorithm

Suppose we have a graph $\text{graph}[][]$ with V vertices from 0 to $V-1$. Now we have to evaluate a $\text{dist}[][]$ where $\text{dist}[i][j]$ represents the shortest path between vertex i to j .

Let us assume that vertices i to j have intermediate nodes. The idea behind Floyd Warshall algorithm is to treat each and every vertex k from 0 to $V-1$ as an intermediate node one by one. When we consider the vertex k , we must have considered vertices from 0 to $k-1$ already. So we use the shortest paths built by previous vertices to build shorter paths with vertex k included.

The following figure shows the above optimal substructure property in Floyd Warshall algorithm:

```
// C++ Program for Floyd Warshall Algorithm
#include <bits/stdc++.h>
using namespace std;

// Solves the all-pairs shortest path
// problem using Floyd Warshall algorithm
void floydWarshall(vector<vector<int>> &graph) {
    int V = graph.size();

    // Add all vertices one by one to
    // the set of intermediate vertices.
    for (int k = 0; k < V; k++) {

        // Pick all vertices as source one by one
        for (int i = 0; i < V; i++) {

            // Pick all vertices as destination
            // for the above picked source
            for (int j = 0; j < V; j++) {

                // If vertex k is on the shortest path from
                // i to j, then update the value of graph[i][j]

                if ((graph[i][j] == -1 ||
```

```

        graph[i][j] > (graph[i][k] + graph[k][j]))
        && (graph[k][j] != -1 && graph[i][k] != -1))
        graph[i][j] = graph[i][k] + graph[k][j];
    }
}
}

int main() {

vector<vector<int>> graph = {
    {0, 4, -1, 5, -1},
    {-1, 0, 1, -1, 6},
    {2, -1, 0, 3, -1},
    {-1, -1, 1, 0, 2},
    {1, -1, -1, 4, 0}
};

floydWarshall(graph);
for(int i = 0; i<graph.size(); i++) {
    for(int j = 0; j<graph.size(); j++) {
        cout<<graph[i][j]<<" ";
    }
    cout<<endl;
}
return 0;
}
}

```

Travelling Salesman Problem

Exploring All Permutations – $O(n!)$ Time and $O(n)$ Space

Using Recursion – $O(n!)$ Time and $O(n)$ Space

The idea behind this approach is to use two parameters: curr, which denotes the currently visited node, and mask, which represents the set of all visited nodes using bitmasking. These two parameters are sufficient to define the state at any given point in the problem.

Let $tsp(curr, mask)$ be the function that calculates the minimum cost to visit all cities, where mask represents the set of cities that have been visited, and curr denotes the current city being visited.

The recurrence relation for the TSP problem is defined as:

$tsp(\text{curr}, \text{mask}) = \min(\text{cost}[\text{curr}][\text{i}] + \text{tsp}(\text{i}, \text{mask} \mid (1 \ll \text{i})))$, for all cities i that have not been visited yet.

where:

curr is the current city in the tour.

mask represents the cities that have already been visited.

$\text{cost}[\text{curr}][\text{i}]$ is the cost to travel from city curr to city i .

$\text{tsp}(\text{i}, \text{mask} \mid (1 \ll \text{i}))$ represents the cost of visiting the remaining cities in the new mask (after visiting city i), and continuing the tour from city i .

Base Case: The base case occurs when all cities have been visited, which can be identified when:

$\text{mask} == ((1 \ll n) - 1)$

In this case, the function simply returns the cost of traveling back to the starting city (city 0):

$\text{tsp}(\text{curr}, \text{mask}) = \text{cost}[\text{curr}][0]$

```
int totalCost(int mask, int pos, int n, vector<vector<int>> &cost) {

    // Base case: if all cities are visited, return the
    // cost to return to the starting city (0)
    if (mask == (1 << n) - 1) {
        return cost[pos][0];
    }

    int ans = INT_MAX;

    // Try visiting every city that has not been visited yet
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) == 0) {

            // If city i is not visited, visit it and update the mask
            ans = min(ans, cost[pos][i] +
                      totalCost((mask | (1 << i)), i, n, cost));
        }
    }

    return ans;
}
```

```

int tsp(vector<vector<int>> &cost) {
    int n = cost.size();

    // Start from city 0, and only city 0 is visited initially (mask = 1)
    return totalCost(1, 0, n, cost);
}

int main() {

    vector<vector<int>> cost = {{0, 10, 15, 20},
                                {10, 0, 35, 25},
                                {15, 35, 0, 30},
                                {20, 25, 30, 0}};

    int res = tsp(cost);
    cout << res << endl;

    return 0;
}

```

Using Top-Down DP (Memoization) – O($n^*n^*2^n$) Time and O(n^*2^n) Space

```

int totalCost(int mask, int curr, int n,
              vector<vector<int>> &cost, vector<vector<int>> &memo) {

    // Base case: if all cities are visited, return the
    // cost to return to the starting city (0)
    if (mask == (1 << n) - 1) {
        return cost[curr][0];
    }

    if (memo[curr][mask] != -1)
        return memo[curr][mask];

    int ans = INT_MAX;

    // Try visiting every city that has not been visited yet
    for (int i = 0; i < n; i++) {
        if ((mask & (1 << i)) == 0) {

            // If city i is not visited, visit it and update

```

```

        // the mask
        ans = min(ans, cost[curr][i] +
                    totalCost((mask | (1 << i)), i, n, cost, memo));
    }
}

return memo[curr][mask] = ans;
}

int tsp(vector<vector<int>> &cost) {
    int n = cost.size();
    vector<vector<int>> memo(n, vector<int>(1 << n, -1));

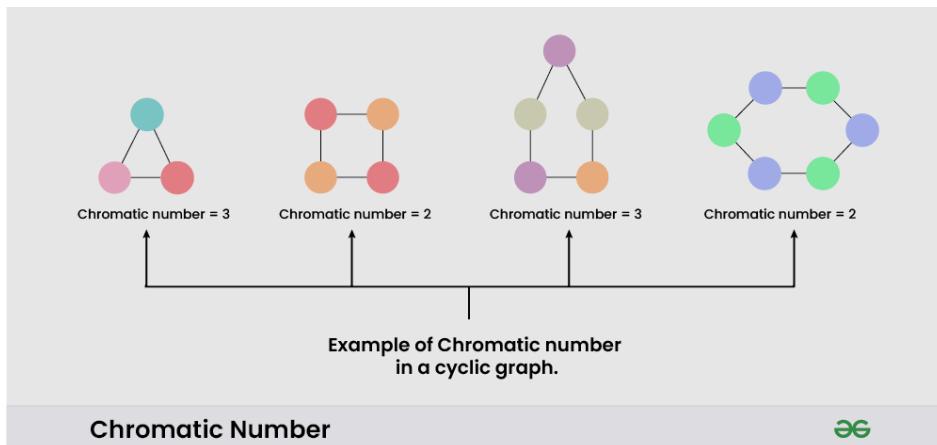
    // Start from city 0, and only city 0 is visited
    // initially (mask = 1)
    return totalCost(1, 0, n, cost,
                     memo);
}

```

Graph Colouring Problem

Chromatic Number:

The minimum number of colors needed to color a graph is called its chromatic number. For example, the following can be colored a minimum of 2 colors.



Chromatic Number

36

The problem of finding a chromatic number of a given graph is NP-complete.

Graph coloring problem is both, a decision problem as well as an optimization problem.

A decision problem is stated as, “With given M colors and graph G, whether a such color scheme is possible or not?”.

The optimization problem is stated as, “Given M colors and graph G, find the minimum number of colors required for graph coloring.”

Follow the given steps to solve the problem:

- Create a recursive function that takes the graph, current index, number of vertices, and color array.
- If the current index is equal to the number of vertices. Print the color configuration in the color array.
- Assign a color to a vertex from the range (1 to m).
 - For every assigned color, check if the configuration is safe, (i.e. check if the adjacent vertices do not have the same color) and recursively call the function with the next index and number of vertices else return **false**
 - If any recursive function returns **true** then break the loop and return true
 - If no recursive function returns **true** then return **false**

```
// C++ program for solution of M
// Coloring problem using backtracking

#include <bits/stdc++.h>
using namespace std;

// Number of vertices in the graph
#define V 4

void printSolution(int color[]);

/* A utility function to check if
   the current color assignment
   is safe for vertex v i.e. checks
   whether the edge exists or not
   (i.e, graph[v][i]==1). If exist
   then checks whether the color to
   be filled in the new vertex(c is
   sent in the parameter) is already
   used by its adjacent
   vertices(i-->adj vertices) or
   not (i.e, color[i]==c) */
bool isSafe(int v, bool graph[V][V], int color[], int c)
{
    for (int i = 0; i < V; i++)
        if (graph[v][i] && c == color[i])
            return false;
```

```

        return true;
    }

/* A recursive utility function
to solve m coloring problem */
bool graphColoringUtil(bool graph[V][V], int m, int color[],
                      int v)
{
    /* base case: If all vertices are
       assigned a color then return true */
    if (v == V)
        return true;

    /* Consider this vertex v and
       try different colors */
    for (int c = 1; c <= m; c++) {

        /* Check if assignment of color
           c to v is fine*/
        if (isSafe(v, graph, color, c)) {
            color[v] = c;

            /* recur to assign colors to
               rest of the vertices */
            if (graphColoringUtil(graph, m, color, v + 1)
                == true)
                return true;

            /* If assigning color c doesn't
               lead to a solution then remove it */
            color[v] = 0;
        }
    }

    /* If no color can be assigned to
       this vertex then return false */
    return false;
}

/* This function solves the m Coloring
problem using Backtracking. It mainly

```

```

uses graphColoringUtil() to solve the
problem. It returns false if the m
colors cannot be assigned, otherwise
return true and prints assignments of
colors to all vertices. Please note
that there may be more than one solutions,
this function prints one of the
feasible solutions.*/
bool graphColoring(bool graph[V][V], int m)
{
    // Initialize all color values as 0.
    // This initialization is needed
    // correct functioning of isSafe()
    int color[V];
    for (int i = 0; i < V; i++)
        color[i] = 0;

    // Call graphColoringUtil() for vertex 0
    if (graphColoringUtil(graph, m, color, 0) == false) {
        cout << "Solution does not exist";
        return false;
    }

    // Print the solution
    printSolution(color);
    return true;
}

/* A utility function to print solution */
void printSolution(int color[])
{
    cout << "Solution Exists:"
        << " Following are the assigned colors"
        << "\n";
    for (int i = 0; i < V; i++)
        cout << " " << color[i] << " ";

    cout << "\n";
}

// Driver code
int main()

```

```

{
    /* Create following graph and test
       whether it is 3 colorable
    (3)---(2)
      |   / |
      |   / |
      | /  |
    (0)---(1)
*/
bool graph[V][V] = {
    { 0, 1, 1, 1 },
    { 1, 0, 1, 0 },
    { 1, 1, 0, 1 },
    { 1, 0, 1, 0 },
};

// Number of colors
int m = 3;

// Function call
graphColoring(graph, m);
return 0;
}

```

Snake and Ladder Problem

Given a snake and ladder board, find the minimum number of dice throws required to reach the destination or last cell from the source or 1st cell. Basically, the player has total control over the outcome of the dice throw and wants to find out the minimum number of throws required to reach the last cell.

If the player reaches a cell which is the base of a ladder, the player has to climb up that ladder and if reaches a cell is the mouth of the snake, and has to go down to the tail of the snake without a dice throw.

```

// This function returns minimum number of dice throws
// required to Reach last cell from 0'th cell in a snake and
// ladder game. move[] is an array of size N where N is no.
// of cells on board If there is no snake or ladder from
// cell i, then move[i] is -1 Otherwise move[i] contains

```

```

// cell to which snake or ladder at i takes to.
int getMinDiceThrows(int move[], int N)
{
    // The graph has N vertices. Mark all the vertices as
    // not visited
    bool* visited = new bool[N];
    for (int i = 0; i < N; i++)
        visited[i] = false;

    // Create a queue for BFS
    queue<queueEntry> q;

    // Mark the node 0 as visited and enqueue it.
    visited[0] = true;
    queueEntry s
        = { 0, 0 }; // distance of 0't vertex is also 0
    q.push(s); // Enqueue 0'th vertex

    // Do a BFS starting from vertex at index 0
    queueEntry qe; // A queue entry (qe)
    while (!q.empty()) {
        qe = q.front();
        int v = qe.v; // vertex no. of queue entry

        // If front vertex is the destination vertex,
        // we are done
        if (v == N - 1)
            break;

        // Otherwise dequeue the front vertex and enqueue
        // its adjacent vertices (or cell numbers reachable
        // through a dice throw)
        q.pop();
        for (int j = v + 1; j <= (v + 6) && j < N; ++j) {
            // If this cell is already visited, then ignore
            if (!visited[j]) {
                // Otherwise calculate its distance and mark
                // it as visited
                queueEntry a;
                a.dist = (qe.dist + 1);
                visited[j] = true;

                // Check if there a snake or ladder at 'j'

```

```

        // then tail of snake or top of ladder
        // become the adjacent of 'i'
        if (move[j] != -1)
            a.v = move[j];
        else
            a.v = j;
        q.push(a);
    }
}

// We reach here when 'qe' has last vertex
// return the distance of vertex in 'qe'
return qe.dist;
}

// Driver program to test methods of graph class
int main()
{
    // Let us construct the board given in above diagram
    int N = 30;
    int moves[N];
    for (int i = 0; i < N; i++)
        moves[i] = -1;

    // Ladders
    moves[2] = 21;
    moves[4] = 7;
    moves[10] = 25;
    moves[19] = 28;

    // Snakes
    moves[26] = 0;
    moves[20] = 8;
    moves[16] = 3;
    moves[18] = 6;

    cout << "Min Dice throws required is "
         << getMinDiceThrows(moves, N);
    return 0;
}

```

Bridges in a graph

Given an undirected Graph, The task is to find the Bridges in this Graph.

An edge in an undirected connected graph is a bridge if removing it disconnects the graph. For a disconnected undirected graph, the definition is similar, a bridge is an edge removal that increases the number of disconnected components.

Naive Approach: Below is the idea to solve the problem:

One by one remove all edges and see if the removal of an edge causes a disconnected graph.

Time Complexity: $O(E^*(V+E))$ for a graph represented by an adjacency list.

Auxiliary Space: $O(V+E)$

Find Bridges in a graph using Tarjan's Algorithm.

Before heading towards the approach understand which edge is termed as bridge. Suppose there exists a edge from $u \rightarrow v$, now after removal of this edge if v can't be reached by any other edges then $u \rightarrow v$ edge is bridge. Our approach is based on this intuition, so take time and grasp it.

ALGORITHM: –

To implement this algorithm, we need the following data structures –

`visited[]` = to keep track of the visited vertices to implement DFS

`disc[]` = to keep track when for the first time that particular vertex is reached

`low[]` = to keep track of the lowest possible time by which we can reach that vertex 'other than parent' so that if edge from parent is removed can the particular node can be reached other than parent.

We will traverse the graph using DFS traversal but with slight modifications i.e. while traversing we will keep track of the parent node by which the particular node is reached because we will update the $\text{low}[\text{node}] = \min(\text{low}[\text{all its adjacent node except parent}])$ hence we need to keep track of the parent.

While traversing adjacent nodes let 'v' of a particular node let 'u', then 3 cases arise –

1. v is parent of u then,

skip that iteration.

2. v is visited then,

update the low of u i.e. $\text{low}[u] = \min(\text{low}[u], \text{disc}[v])$ this arises when a node can be visited by more than one node, but low is to keep track of the lowest possible time so we will update it.

3. v is not visited then,

call the DFS to traverse ahead

now update the $\text{low}[u] = \min(\text{low}[u], \text{low}[v])$ as we know v can't be parent cause we have handled that case first.

now check if ($\text{low}[v] > \text{disc}[u]$) i.e. the lowest possible time to reach 'v' is greater than 'u' this means we can't reach 'v' without 'u' so the edge $u \rightarrow v$ is a bridge.

Below is the implementation of the above approach:

```
class Solution {
private:
    int timer = 1;
    void dfs(int node, int parent, vector<int> &vis,
             vector<int> adj[], int tin[], int low[],
             vector<vector<int>> &bridges) {
        vis[node] = 1;
        tin[node] = low[node] = timer;
        timer++;
        for (auto it : adj[node]) {
            if (it == parent) continue;
            if (vis[it] == 0) {
                dfs(it, node, vis, adj, tin, low, bridges);
                low[node] = min(low[it], low[node]);
                // node --- it
                if (low[it] > tin[node]) {
                    bridges.push_back({it, node});
                }
            }
            else {
                low[node] = min(low[node], low[it]);
            }
        }
    }
public:
    vector<vector<int>> criticalConnections(int n,
```

```

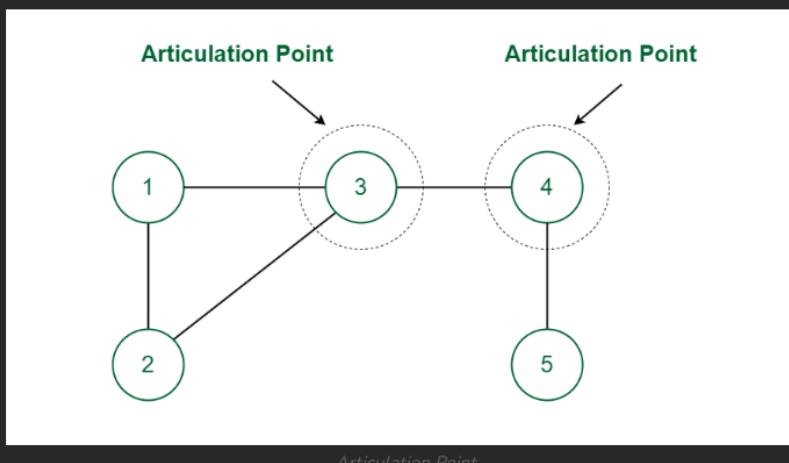
vector<vector<int>>& connections) {
    vector<int> adj[n];
    for (auto it : connections) {
        int u = it[0], v = it[1];
        adj[u].push_back(v);
        adj[v].push_back(u);
    }
    vector<int> vis(n, 0);
    int tin[n];
    int low[n];
    vector<vector<int>> bridges;
    dfs(0, -1, vis, adj, tin, low, bridges);
    return bridges;
}
};


```

Articulation Points (or Cut Vertices) in a Graph

What is Articulation Point? A vertex v is an **articulation point** (also called cut vertex) if removing v increases the number of connected components. Articulation points represent vulnerabilities in a connected network – single points whose failure would split the network into 2 or more components. They are useful for designing reliable networks.

Examples:



[Naive Approach] – $O(V * (V + E))$ Time and $O(V)$ Space

A simple approach is to one by one remove all vertices and see if removal of a vertex causes disconnected graph. To do so, iterate over all the vertices and for every vertex do the following:

Remove v from graph

See if the graph remains connected (We can either use BFS or DFS)

Add v back to the graph

Expected Approach] – Using Tarjan's Algorithm – $O(V + E)$ Time and $O(V)$ Space

The idea is to use DFS (Depth First Search). In DFS, follow vertices in a tree form called the DFS tree. In the DFS tree, a vertex u is the parent of another vertex v, if v is discovered by u.

In DFS tree, a vertex u is an articulation point if one of the following two conditions is true.

u is the root of the DFS tree and it has at least two children.

u is not the root of the DFS tree and it has a child v such that no vertex in the subtree rooted with v has a back edge to one of the ancestors in DFS tree of u.

```
#include <bits/stdc++.h>
using namespace std;

// helper function to perform dfs and find the articulation points
void findPoints(vector<vector<int>> &adj, int u, vector<int> &visited,
                vector<int> &disc, vector<int> &low, int &time,
                int parent, vector<int> &isAP) {

    // Count of children in DFS Tree
    int children = 0;

    // Mark the current node as visited
    visited[u] = 1;

    // Initialize discovery time and low value
    disc[u] = low[u] = ++time;

    // Go through all vertices adjacent to this
    for (auto v : adj[u]) {
```

```

// If v is not visited yet, then make it a child of u
// in DFS tree and recur for it
if (!visited[v]) {
    children++;
    findPoints(adj, v, visited, disc, low, time, u, isAP);

    // Check if the subtree rooted with v has
    // a connection to one of the ancestors of u
    low[u] = min(low[u], low[v]);

    // If u is not root and low value of one of
    // its child is more than discovery value of u.
    if (parent != -1 && low[v] >= disc[u])
        isAP[u] = 1;
}

// Update low value of u for parent function calls.
else if (v != parent)
    low[u] = min(low[u], disc[v]);
}

// If u is root of DFS tree and has two or more children.
if (parent == -1 && children > 1)
    isAP[u] = 1;
}

// Function to find Articulation Points in the graph
vector<int> articulationPoints(vector<vector<int>> adj) {

    int V = adj.size();

    // to store the articulation points
    vector<int> res;

    // to stores discovery times of visited vertices
    vector<int> disc(V, 0);

    // to store earliest visited vertex (the vertex with minimum
    // discovery time) that can be reached from subtree
    vector<int> low(V);

    // to keep track of visited vertices
    vector<int> visited(V, 0);
}

```

```

// to mark the articulation points
vector<int> isAP(V, 0);

// to store time and parent node
int time = 0, par = -1;

// Adding this loop so that the code works
// even if we are given disconnected graph
for (int u = 0; u < V; u++)
    if (!visited[u])
        findPoints(adj, u, visited, disc, low,
                   time, par, isAP);

// storing the articulation points
for (int u = 0; u < V; u++)
    if (isAP[u] == true)
        res.push_back(u);

// if no points are found, return -1
if(res.empty())
    return {-1};

return res;
}

// Utility function to add an edge
void addEdge(vector<vector<int>> &adj, int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main() {
    int V = 5;

    // create adjacency list
    vector<vector<int>> adj(V);
    addEdge(adj, 0, 1);
    addEdge(adj, 1, 4);
    addEdge(adj, 2, 4);
    addEdge(adj, 3, 4);
    addEdge(adj, 2, 3);
    vector<int> ans = articulationPoints(adj);
}

```

```
    for(auto i:ans) {
        cout<<i<<" ";
    }
    return 0;
}
```

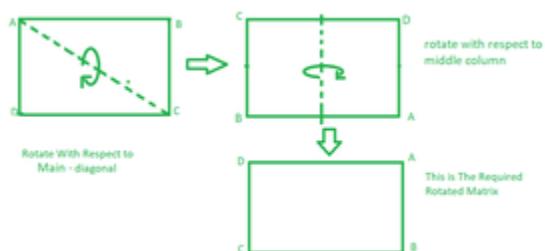
Count Strongly connected Components(Kosaraju Algo)

Strongly Connected Component is a subset of vertices where every vertex in the subset is reachable from every other vertex in the same subset by traversing the directed edges. Finding the SCCs of a graph can provide important insights into the structure and connectivity of the graph, with applications in various fields such as social network analysis, web crawling, and network routing

Matrix

Rotate the matrix by 90 Degrees

90 degree right:



```
void rotate(int arr[N][N])
{
    // First rotation
    // with respect to main diagonal
```

```

for(int i = 0; i < N; ++i)
{
    for(int j = 0; j < i; ++j)
    {
        int temp = arr[i][j];
        arr[i][j] = arr[j][i];
        arr[j][i] = temp;
    }
}

// Second rotation
// with respect to middle column
for(int i = 0; i < N; ++i)
{
    for(int j = 0; j < N / 2; ++j)
    {
        int temp = arr[i][j];
        arr[i][j] = arr[i][N - j - 1];
        arr[i][N - j - 1] = temp;
    }
}
}

```

90 degree left:

```

void rotate(int arr[N][N])
{

    // First rotation
    // with respect to main diagonal
    for(int i = 0; i < N; i++)
    {
        for(int j = 0; j < N - i; j++)
        {
            int temp = arr[i][j];
            arr[i][j] = arr[N - 1 - j][N - 1 - i];
            arr[N - 1 - j][N - 1 - i] = temp;
        }
    }
    // Second rotation
}

```

```

// with respect to middle column
for(int i = 0; i < N; ++i)
{
    for(int j = 0; j < N / 2; ++j)
    {
        int temp = arr[i][j];
        arr[i][j] = arr[i][N - j - 1];
        arr[i][N - j - 1] = temp;
    }
}

```

Method 3: By rotating cyclic rings in clockwise direction:

```

// Function to rotate the matrix 90 degree clockwise
void rotate90Clockwise(int a[N][N])
{

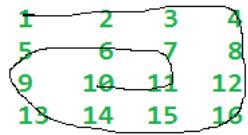
    // Traverse each cycle
    for (int i = 0; i < N / 2; i++) {
        for (int j = i; j < N - i - 1; j++) {

            // Swap elements of each cycle
            // in clockwise direction
            int temp = a[i][j];
            a[i][j] = a[N - 1 - j][i];
            a[N - 1 - j][i] = a[N - 1 - i][N - 1 - j];
            a[N - 1 - i][N - 1 - j] = a[j][N - 1 - i];
            a[j][N - 1 - i] = temp;
        }
    }
}

```

Spiral Traversal of matrix:

Input:



Output:

1 2 3 4 8 12 16 15 14 13 9 5 6 7 11 10

```
class Solution {  
public:  
    vector<int> spirallyTraverse(vector<vector<int>>& matrix) {  
        vector<int> result;  
        if (matrix.empty() || matrix[0].empty()) return result;  
  
        int top = 0, bottom = matrix.size() - 1;  
        int left = 0, right = matrix[0].size() - 1;  
  
        while (top <= bottom && left <= right) {  
            // Traverse top row  
            for (int i = left; i <= right; i++)  
                result.push_back(matrix[top][i]);  
            top++;  
  
            // Traverse right column  
            for (int i = top; i <= bottom; i++)  
                result.push_back(matrix[i][right]);  
            right--;  
  
            if (top <= bottom) {  
                // Traverse bottom row  
                for (int i = right; i >= left; i--)  
                    result.push_back(matrix[bottom][i]);  
                bottom--;  
            }  
        }  
    }  
};
```

```

        if (left <= right) {
            // Traverse left column
            for (int i = bottom; i >= top; i--)
                result.push_back(matrix[i][left]);
            left++;
        }
    }

    return result;
}
};

```

Search in 2d Matrix:

You are given an $m \times n$ integer matrix `matrix` with the following two properties:

- Each row is sorted in non-decreasing order.
- The first integer of each row is greater than the last integer of the previous row.

Given an integer `target`, return `true` if `target` is in `matrix` or `false` otherwise.

You must write a solution in $O(\log(m * n))$ time complexity.

Example 1:

1	3	5	7
10	11	16	20
23	30	34	60

```

class Solution {
public:

```

```

bool searchMatrix(vector<vector<int>>& m, int t) {
    int x=m.size(),y=m[0].size(),i,j;
    if(m[0][0]>t || m[x-1][y-1]<t) return false;
    int hr=x-1,lr=0,lc=0,hc=y-1;
    int mr=(hr+lr)/2,mc=(lc+hc)/2;
    while(lr<=hr){
        mr=(hr+lr)/2;
        cout<<lr<<" "<<mr<<" "<<hr<<endl;
        if(t==m[hr][0]){
            mr=hr;
            break;
        }
        if(mr+1<=x-1 and t>=m[mr][0] and t<=m[mr+1][0]){
            cout<<"condition 1"<<endl;
            break;
        }
        else if(t<m[mr][0]){
            cout<<"condition 2"<<endl;
            hr=mr-1;
            continue;
        }
        else{
            cout<<"condition 3"<<endl;
            lr=mr+1;
        }
    }
    cout<<"mr is "<<mr<<endl;
    while(lc<=hc){
        mc = (lc+hc)/2;
        cout<<lc<<" "<<mc<<" "<<hc<<endl;
        if(m[mr][mc]==t) return true;
        if(m[mr][mc]>t){
            hc=mc-1;
            cout<<"condition 4"<<endl;
            continue;
        }
        else{
            cout<<"condition 5"<<endl;
            lc=mc+1;
            continue;
        }
    }
    return false;
}

```

```
    }
};
```

Median in a rowwise sorted Matrix!

```
class Solution{
public:
void mergeArrays(vector<int>& a, vector<int>& b,
                  vector<int>& c) {
    int i = 0, j = 0, k = 0;
    int n1 = a.size();
    int n2 = b.size();
    c.resize(n1 + n2);

    // Traverse both arrays
    while (i < n1 && j < n2) {
        if (a[i] < b[j])
            c[k++] = a[i++];
        else
            c[k++] = b[j++];
    }

    // Store remaining elements of a
    while (i < n1)
        c[k++] = a[i++];

    // Store remaining elements of b
    while (j < n2)
        c[k++] = b[j++];
}

// This function takes a vector of vectors as an argument and
// All arrays are assumed to be sorted. It merges them
// together and returns the final sorted output.
void mergeKArrays(vector<vector<int>>& arr, int lo, int hi,
                  vector<int>& res) {

    // If one array is in range
    if (lo == hi) {
        res = arr[lo];
```

```

        return;
    }

    // If only two arrays are left, merge them
    if (hi - lo == 1) {
        mergeArrays(arr[lo], arr[hi], res);
        return;
    }

    // Calculate mid point
    int mid = (lo + hi) / 2;

    // Divide the array into halves
    // Output arrays
    vector<int> out1, out2;
    mergeKArrays(arr, lo, mid, out1);
    mergeKArrays(arr, mid + 1, hi, out2);

    // Merge the output arrays
    mergeArrays(out1, out2, res);
}

int median(vector<vector<int>> &g, int R, int C){
    // code here      ]
    vector<int> res;
    mergeKArrays(g, 0, g.size() - 1, res);
    return res[(R*C)/2];
}
};

```

Find the row with maximum number of 1s

[Naive Approach] Row-wise traversal – O(M*N) Time and O(1) Space:

[Better Approach] Using Binary Search – O(M * logN) Time O(1) Space:

[Expected Approach] Traversal from top-right to outside the grid – O(M + N) Time and O(1) Space:

```

#include <bits/stdc++.h>
using namespace std;

```

```

// Function to find the index of first instance
// of 1 in a boolean array arr[]
int first(vector<bool>& arr, int low, int high) {
    int idx = -1;
    while (low <= high) {

        // Get the middle index
        int mid = low + (high - low) / 2;

        // If the element at mid is 1, then update mid as
        // starting index of 1s and search in the left half
        if (arr[mid] == 1) {
            idx = mid;
            high = mid - 1;
        }

        // If the element at mid is 0, then search in the
        // right half
        else {
            low = mid + 1;
        }
    }
    return idx;
}

// Function that returns index of row
// with maximum number of 1s.
int rowWithMax1s(vector<vector<bool>>& mat) {
    // Initialize max values
    int max_row_index = -1, max = -1;
    int R = mat.size();
    int C = mat[0].size();

    // Traverse for each row and count number of 1s
    // by finding the index of first 1
    for (int i = 0; i < R; i++) {
        int index = first(mat[i], 0, C - 1);
        if (index != -1 && C - index > max) {
            max = C - index;
            max_row_index = i;
        }
    }
}

```

```

        return max_row_index;
    }

// Driver Code
int main() {
    vector<vector<bool>> mat = { { 0, 0, 0, 1 },
                                { 0, 1, 1, 1 },
                                { 1, 1, 1, 1 },
                                { 0, 0, 0, 0 } };

    cout << rowWithMax1s(mat);

    return 0;
}

```

```

int rowWithMax1s(vector<vector<bool>>& mat) {
    int maxRow = -1, row = 0;
    int R = mat.size();
    int C = mat[0].size();
    int col = C - 1;

    // Move till we are inside the matrix
    while (row < R && col >= 0) {
        // If the current value is 0, move down to the next row
        if (mat[row][col] == 0) {
            row += 1;
        }
        // Else if the current value is 1, update ans and
        // move to the left column
        else {
            maxRow = row;
            col -= 1;
        }
    }
    return maxRow;
}

```

Print all elements in sorted order from row and column wise sorted matrix

Use merging k sorted arraya

```
// C++ program to merge k sorted arrays of size n each.
#include<iostream>
#include<climits>
using namespace std;

#define N 4

// A min heap node
struct MinHeapNode
{
    int element; // The element to be stored
    int i; // index of the row from which the element is taken
    int j; // index of the next element to be picked from row
};

// Prototype of a utility function to swap two min heap nodes
void swap(MinHeapNode *x, MinHeapNode *y);

// A class for Min Heap
class MinHeap
{
    MinHeapNode *harr; // pointer to array of elements in heap
    int heap_size; // size of min heap
public:
    // Constructor: creates a min heap of given size
    MinHeap(MinHeapNode a[], int size);

    // to heapify a subtree with root at given index
    void MinHeapify(int );

    // to get index of left child of node at index i
    int left(int i) { return (2*i + 1); }

    // to get index of right child of node at index i
    int right(int i) { return (2*i + 2); }
}
```

```

// to get the root
MinHeapNode getMin() { return harr[0]; }

// to replace root with new node x and heapify() new root
void replaceMin(MinHeapNode x) { harr[0] = x; MinHeapify(0); }
};

// This function prints elements of a given matrix in non-decreasing
// order. It assumes that ma[][] is sorted row wise sorted.
void printSorted(int mat[][]) {
    // Create a min heap with k heap nodes. Every heap node
    // has first element of an array
    MinHeapNode *harr = new MinHeapNode[N];
    for (int i = 0; i < N; i++)
    {
        harr[i].element = mat[i][0]; // Store the first element
        harr[i].i = i; // index of row
        harr[i].j = 1; // Index of next element to be stored from row
    }
    MinHeap hp(harr, N); // Create the min heap

    // Now one by one get the minimum element from min
    // heap and replace it with next element of its array
    for (int count = 0; count < N*N; count++)
    {
        // Get the minimum element and store it in output
        MinHeapNode root = hp.getMin();

        cout << root.element << " ";

        // Find the next element that will replace current
        // root of heap. The next element belongs to same
        // array as the current root.
        if (root.j < N)
        {
            root.element = mat[root.i][root.j];
            root.j += 1;
        }
        // If root was the last element of its array
        else root.element = INT_MAX; //INT_MAX is for infinite

        // Replace root with next element of array
    }
}

```

```

        hp.replaceMin(root);
    }
}

// FOLLOWING ARE IMPLEMENTATIONS OF STANDARD MIN HEAP METHODS
// FROM CORMEN BOOK
// Constructor: Builds a heap from a given array a[] of given size
MinHeap::MinHeap(MinHeapNode a[], int size)
{
    heap_size = size;
    harr = a; // store address of array
    int i = (heap_size - 1)/2;
    while (i >= 0)
    {
        MinHeapify(i);
        i--;
    }
}

// A recursive method to heapify a subtree with root at given index
// This method assumes that the subtrees are already heapified
void MinHeap::MinHeapify(int i)
{
    int l = left(i);
    int r = right(i);
    int smallest = i;
    if (l < heap_size && harr[l].element < harr[i].element)
        smallest = l;
    if (r < heap_size && harr[r].element < harr[smallest].element)
        smallest = r;
    if (smallest != i)
    {
        swap(&harr[i], &harr[smallest]);
        MinHeapify(smallest);
    }
}

// A utility function to swap two elements
void swap(MinHeapNode *x, MinHeapNode *y)
{
    MinHeapNode temp = *x; *x = *y; *y = temp;
}

```

```

// driver program to test above function
int main()
{
int mat[N][N] = { {10, 20, 30, 40},
                  {15, 25, 35, 45},
                  {27, 29, 37, 48},
                  {32, 33, 39, 50},
                };
printSorted(mat);
return 0;
}

```

Time complexity: $O(N^2 \log N)$.

Auxiliary Space: $O(N)$

Maximal Rectangle

Given a rows x cols binary matrix filled with 0's and 1's, find the largest rectangle containing only 1's and return its area.

1	0	1	0	0
1	0	1	1	1
1	1	1	1	1
1	0	0	1	0

Logic to solve question is that: you first take a row (from top) and calculate maximum histogram area and then take another row place below the previous row and now you've new histogram, so then calculate max area under histogram and so on till last row and the maximum area under histogram among those will be the answer.

Sub problems are: nearest small value left to array element

```

class Solution {
public:

```

```

int largestRectangleArea(vector<int> heights) {
    // left
    // right
    // right-left-1
    // heights[i]*width
    // max area
    // yahi cheeze karni NSR-NSL-1
    vector<int> v;
    vector<int> v1;
    stack<pair<int, int>> lst;

    int pseudo = -1;

    for (int i = 0; i < heights.size(); i++) {
        if (lst.size() == 0) {
            v.push_back(pseudo);
        } else if (lst.size() > 0 && lst.top().first < heights[i]) {
            v.push_back(lst.top().second);
        } else if (lst.size() > 0 && lst.top().first >= heights[i]) {
            while (lst.size() > 0 && lst.top().first >= heights[i]) {
                lst.pop();
            }
            if (lst.size() == 0)
                v.push_back(pseudo);
            else
                v.push_back(lst.top().second);
        }
        lst.push({heights[i], i});
    }
    stack<pair<int, int>> rst;
    int pseudo1 = heights.size();

    for (int i = heights.size() - 1; i >= 0; i--) {
        if (rst.size() == 0) {
            v1.push_back(pseudo1);
        } else if (rst.size() > 0 && rst.top().first < heights[i]) {
            v1.push_back(rst.top().second);
        } else if (rst.size() > 0 && rst.top().first >= heights[i]) {
            while (rst.size() > 0 && rst.top().first >= heights[i]) {
                rst.pop();
            }
            if (rst.size() == 0)
                v1.push_back(pseudo1);
        }
    }
}

```

```

        else
            v1.push_back(rst.top().second);
    }
    rst.push({heights[i], i});
}
reverse(v1.begin(), v1.end());
int width[heights.size() + 1];
int maxi = -1;
int area;
for (int i = 0; i < heights.size(); i++) {
    width[i] = v1[i] - v[i] - 1;
}

for (int i = 0; i < heights.size(); i++) {
    area = heights[i] * width[i];
    maxi = max(maxi, area);
}
return maxi;
}
vector<vector<int>> convertCharToIntBinaryArray(const
vector<vector<char>>& matrix) {
    vector<vector<int>> matrix1;
    for (const auto& row : matrix) {
        vector<int> intRow;
        for (char element : row) {
            intRow.push_back(element - '0');
        }
        matrix1.push_back(intRow);
    }
    return matrix1;
}
int maximalRectangle(vector<vector<char>>& matrix) {
    //n number of maximum histogram problems
    //1). convert char to integer matrix
    //2). largest rectangle problem
    //3). calculate maximum area for each histogram
    //4). return maximum one
    vector<vector<int>> matrix1 = convertCharToIntBinaryArray(matrix);
    int row = matrix1.size();
    int col = matrix1[0].size();
    vector<int> v2;
    for (int j=0;j<col;j++)
    {

```

```
v2.push_back(matrix1[0][j]);
}
int maxwell=largestRectangleArea(v2);

for(int i=1;i<row;i++)
{
    for(int j=0;j<col;j++)
    {
        if(matrix1[i][j]==0)
            v2[j]=0;
        else
            v2[j]=v2[j]+matrix1[i][j];
    }
    maxwell=max(maxwell,largestRectangleArea(v2));
}
return maxwell;
};

};
```

Find a specific pair in matrix:

Given an $n \times n$ matrix $\text{mat}[n][n]$ of integers, find the maximum value of $\text{mat}(c, d) - \text{mat}(a, b)$ over all choices of indexes such that both $c > a$ and $d > b$.

Brute force

```
    return maxValue;
}
```

An efficient solution uses extra space. We pre-process the matrix such that index(i, j) stores max of elements in matrix from (i, j) to (N-1, N-1) and in the process keeps on updating maximum value found so far. We finally return the maximum value.

Time complexity: $O(N^2)$.

Auxiliary Space: $O(N^2)$

```
// The function returns maximum value A(c,d) - A(a,b)
// over all choices of indexes such that both c > a
// and d > b.
int findMaxValue(int mat[][][N])
{
    //stores maximum value
    int maxValue = INT_MIN;

    // maxArr[i][j] stores max of elements in matrix
    // from (i, j) to (N-1, N-1)
    int maxArr[N][N];

    // last element of maxArr will be same's as of
    // the input matrix
    maxArr[N-1][N-1] = mat[N-1][N-1];

    // preprocess last row
    int maxv = mat[N-1][N-1]; // Initialize max
    for (int j = N - 2; j >= 0; j--)
    {
        if (mat[N-1][j] > maxv)
            maxv = mat[N - 1][j];
        maxArr[N-1][j] = maxv;
    }

    // preprocess last column
    maxv = mat[N - 1][N - 1]; // Initialize max
    for (int i = N - 2; i >= 0; i--)
    {
        if (mat[i][N - 1] > maxv)
            maxv = mat[i][N - 1];
```

```

        maxArr[i][N - 1] = maxv;
    }

    // preprocess rest of the matrix from bottom
    for (int i = N-2; i >= 0; i--)
    {
        for (int j = N-2; j >= 0; j--)
        {
            // Update maxValue
            if (maxArr[i+1][j+1] - mat[i][j] >
                maxValue)
                maxValue = maxArr[i + 1][j + 1] - mat[i][j];

            // set maxArr (i, j)
            maxArr[i][j] = max(mat[i][j],
                                max(maxArr[i][j + 1],
                                    maxArr[i + 1][j]));
        }
    }

    return maxValue;
}

```

An **optimal approach** is with **space complexity O(N)**.

Instead of using the maxArr matrix, we can use two separate vectors (temp1 and temp2) to get maxArr[i+1][j] and maxArr[i][j+1] values.

```

int findMaxValue(int mat[][])
{
    vector<int> temp1(N), temp2(N);
    temp1[N - 1] = mat[N - 1][N - 1];

    // Fill temp1
    for (int j = N - 2; j >= 0; j--)
        temp1[j] = max(temp1[j + 1], mat[N - 1][j]);

    // stores maximum value
    int maxValue = INT_MIN;

    // Iterate over the remaining rows

```

```

for (int i = N - 2; i >= 0; i--) {
    // Initialize the last element of temp2
    temp2[N - 1] = max(temp1[N - 1], mat[i][N - 1]);
    for (int j = N - 2; j >= 0; j--) {
        // update temp2 and maxValue
        maxValue
            = max(maxValue, temp1[j + 1] - mat[i][j]);
        temp2[j] = max(
            { mat[i][j], temp1[j], temp2[j + 1] });
    }
    // Set temp1 to temp2 for the next iteration
    temp1 = temp2;
}

// Return the maximum value
return maxValue;
}

```

Kth smallest element in a row-wise and column-wise sorted 2D array

```

greater than k
//thus top of priority queue is kth smallest element in matrix

priority_queue<int> maxH;
if(k==1)
    return matrix[0][0];

for(i=0;i<n;i++)
{
    for(j=0;j<n;j++)
    {
        maxH.push(matrix[i][j]);
        if(maxH.size()>k)
            maxH.pop();
    }
}

return maxH.top();

```

```
}
```

Common elements in all rows of a given matrix

```
void printCommonElements(int mat[M][N])
{
    unordered_map<int, int> mp;

    // initialize 1st row elements with value 1
    for (int j = 0; j < N; j++)
        mp[(mat[0][j])] = 1;

    // traverse the matrix
    for (int i = 1; i < M; i++)
    {
        for (int j = 0; j < N; j++)
        {
            // If element is present in the map and
            // is not duplicated in current row.
            if (mp[(mat[i][j])] == i)
            {
                // we increment count of the element
                // in map by 1
                mp[(mat[i][j])] = i + 1;

                // If this is last row
                if (i==M-1 && mp[(mat[i][j])]==M)
                    cout << mat[i][j] << " ";
            }
        }
    }
}
```

Bit Manipulation:

Concepts:

- I believe it is a trick to figure out if n is a power of 2. ($n == (n \& -n)$) IFF n is a power of 2 (1,2,4,8).
 - // Get its last set bit
diff &= -diff;
-

Now, lets look at how negation works with binary numbers. Generally, we store integers like 1 or 2 as 32 or 64 bit numbers. However, for simplicity, lets pretend we're only working with 8 bits--the principles are the same. So 11, for example, would look like:

00001011

One simple way to do negation is so-called "one's complement". To make a number negative, all we do is flip all the bits. So -11 would be:

11110100

If this was actually the method we used, then $n \& -n$ would always be 0. Since we complemented the number, not a single bit remains unchanged!

However, in reality, we use a slightly different method to do negation: "two's complement". To get -11, we would first flip all the bits and then add 1 to the result. So we would get:

11110101

Count Set Bits in an Integer

Method 1

```
unsigned int countSetBits(unsigned int n)
{
    unsigned int count = 0;
```

```
while (n) {
    count += n & 1;
    n >>= 1;
}
return count;
}
```

Method 2

```
// recursive function to count set bits
int countSetBits(int n)
{
    // base case
    if (n == 0)
        return 0;
    else
        return 1 + countSetBits(n & (n - 1));
}
```

Method 3: Brian Kernighan's Algorithm:

Subtracting 1 from a decimal number flips all the bits after the rightmost set bit(which is 1) including the rightmost set bit.

for example :

10 in binary is 00001010

9 in binary is 00001001

8 in binary is 00001000

7 in binary is 00000111

So if we subtract a number by 1 and do it bitwise & with itself ($n \& (n-1)$), we unset the rightmost set bit. If we do $n \& (n-1)$ in a loop and count the number of times the loop executes, we get the set bit count.

The beauty of this solution is the number of times it loops is equal to the number of set bits in a given integer.

```
1 Initialize count: = 0
2 If integer n is not zero
    (a) Do bitwise & with (n-1) and assign the value back to n
        n: = n&(n-1)
    (b) Increment count by 1
    (c) go to step 2
3 Else return count
```

```
n = 9 (1001)
```

```
count = 0
```

Since $9 > 0$, subtract by 1 and do bitwise & with $(9-1)$

```
n = 9&8 (1001 & 1000)
```

```
n = 8
```

```
count = 1
```

Since $8 > 0$, subtract by 1 and do bitwise & with $(8-1)$

```
n = 8&7 (1000 & 0111)
```

```
n = 0
```

```
count = 2
```

Since $n = 0$, return count which is 2 now.

```
public:
    unsigned int countSetBits(int n)
    {
        unsigned int count = 0;
        while (n) {
            n &= (n - 1);
            count++;
        }
    }
```

```
        }
        return count;
    }
};
```

Bit Difference of Two Number - count the number of bits needed to be flipped to convert A to B.

```
class Solution{
public:
    // Function to find number of bits needed to be flipped to convert A to
B
    int countBitsFlip(int a, int b){

        // Your logic here
        int g = (a&b)^(~a|~b), c=0;
        while(g){
            c += g&1;
            g>>=1;
        }
        return c;
    }
};
```

Find the two non-repeating elements in an array of repeating elements/ Unique Numbers 2

Non Repeating Numbers using XOR:

All the bits that are set in xor will be set in one non-repeating element (x or y) and not in others. So if we take any set bit of xor and divide the elements of the array in two sets – one set of elements with same bit set and another set with same bit not set. By doing so, we will get x in one set and y in another set. Now if we do XOR of all the elements in the first set, we will get the first non-repeating element, and by doing same in other sets we will get the second non-repeating element.

We have the array: [2, 4, 7, 9, 2, 4]

- $XOR = 2 \wedge 4 \wedge 7 \wedge 9 \wedge 2 \wedge 4 = 2 \wedge 2 \wedge 4 \wedge 4 \wedge 7 \wedge 9 = 0 \wedge 0 \wedge 7 \wedge 9 = 7 \wedge 9 = 14$
- The rightmost set bit in binary representation of 14 is at position 1 (from the right).
- Divide the elements into two groups based on the rightmost set bit.
 - Group 1 (rightmost bit set at position 1): [2, 7, 2]
 - Group 2 (rightmost bit not set at position 1): [4, 9, 4]
- XOR all elements in Group 1 to find one non-repeating element.
 - Non-repeating element 1 = $2 \wedge 7 \wedge 2 = 7$
- XOR all elements in Group 2 to find the other non-repeating element.
 - Non-repeating element 2 = $4 \wedge 9 \wedge 4 = 9$
- The two non-repeating elements are 7 and 9,

```
vector<int> get2NonRepeatingNos(vector<int>& nums)
{
    // Pass 1:
    // Get the XOR of the two numbers we need to find
    long long int diff = 0;
    for (auto i : nums) {
        diff = i ^ diff;
    }

    // Get its last set bit
    diff &= -diff;

    // Pass 2:
    vector<int> rets = {
        0, 0
    }; // this vector stores the two numbers we will return
    for (int num : nums) {
        if ((num & diff) == 0) { // the bit is not set
            rets[0] ^= num;
        }
        else { // the bit is set
            rets[1] ^= num;
        }
    }
}
```

```

    // Ensure the order of the returned numbers is
    // consistent
    if (rets[0] > rets[1]) {
        swap(rets[0], rets[1]);
    }

    return rets;
}

```

Find position of set bit:

```

class Solution {
public:
    int findPosition(int N) {
        // code here
        int a = N&-N;
        if(a!=N){
            return -1;
        }
        if( N==0 ) return -1;

        int ans=0;
        while(N>0){
            ans++;
            N = N>>1;
        }
        return ans;
    }
};

```

Copy set bits in a range

Given two numbers x and y, and a range [l, r] where $1 \leq l, r \leq 32$. The task is consider set bits of y in range [l, r] and set these bits in x also.

Examples :

Input : x = 10, y = 13, l = 2, r = 3

Output : x = 14

Binary representation of 10 is 1010 and
that of y is 1101. There is one set bit
in y at 3'rd position (in given range).
After we copy this bit to x, x becomes 1110
which is binary representation of 14.

Input : x = 8, y = 7, l = 1, r = 2
Output : x = 11

Iteration Naive Approach - O(r) TC

```
void copySetBits(unsigned &x, unsigned y,
                 unsigned l, unsigned r)
{
    // l and r must be between 1 to 32
    // (assuming ints are stored using
    // 32 bits)
    if (l < 1 || r > 32)
        return ;

    // Traverse in given range
    for (int i=l; i<=r; i++)
    {
        // Find a mask (A number whose
        // only set bit is at i'th position)
        int mask = 1 << (i-1);

        // If i'th bit is set in y, set i'th
        // bit in x also.
        if (y & mask)
            x = x | mask;
    }
}
```

Method 2 (Copy all bits using one bit mask) - O(1) TC

```
oid copySetBits(unsigned &x, unsigned y,
                unsigned l, unsigned r)
```

```

{
    // l and r must be between 1 to 32
    if (l < 1 || r > 32)
        return ;

    // get the length of the mask
    int maskLength = (1l<<(r-l+1)) - 1;

    // Shift the mask to the required position
    // "&" with y to get the set bits at between
    // l ad r in y
    int mask = ((maskLength)<<(l-1)) & y ;
    x = x | mask;
}

```

Divide two integers without using multiplication and division operator

dividend = 30 divisor = 4

$$30 = \frac{4 \times 2^2}{30 - (4 \ll 2)} + \frac{4 \times 2^1}{14 - (4 \ll 1)} + \frac{4 \times 2^0}{6 - (4 \ll 0)} + 2$$

$$\begin{aligned} &= 30 - 16 \\ &= 14 \end{aligned}$$

$$\begin{aligned} &= 14 - 8 \\ &= 6 \end{aligned}$$

$$\begin{aligned} &= 6 - 4 \\ &= 2 \end{aligned}$$

This can be easily done by iterating on the bit position $i = 30$ to 0. For each bit position i , where $(\text{divisor} \ll i)$ is less than dividend, set the i th bit of the quotient and subtract $(\text{divisor} \ll i)$ from the dividend. After iterating over all the bits, return the quotient with the corresponding sign.

```

// Function to divide a by b and
// return floor value it
long long divide(long long a, long long b) {

```

```

// Handle overflow
if(a == INT_MIN && b == -1)
    return INT_MAX;

// The sign will be negative only if sign of
// divisor and dividend are different
int sign = ((a < 0) ^ (b < 0)) ? -1 : 1;

// remove sign of operands
a = abs(a);
b = abs(b);

// Initialize the quotient
long long quotient = 0;

// Iterate from most significant bit to
// least significant bit
for (int i = 31; i >= 0; --i) {

    // Check if (divisor << i) <= dividend
    if ((b << i) <= a) {
        a -= (b << i);
        quotient |= (1LL << i);
    }
}

return sign * quotient;
}

```

Calculate square of a number without using *, / and pow()

We can do it in O(Logn) time using bitwise operators. The idea is based on the following fact.

```

square(n) = 0 if n == 0
if n is even
    square(n) = 4*square(n/2)
if n is odd
    square(n) = 4*square(floor(n/2)) + 4*floor(n/2) + 1

```

Examples

```

square(6) = 4*square(3)
square(3) = 4*(square(1)) + 4*1 + 1 = 9
square(7) = 4*square(3) + 4*3 + 1 = 4*9 + 4*3 + 1 = 49

```

```

int square(int n)
{
    // Base case
    if (n == 0)
        return 0;

    // Handle negative number
    if (n < 0)
        n = -n;

    // Get floor(n/2) using right shift
    int x = n >> 1;

    // If n is odd
    if (n & 1)
        return ((square(x) << 2) + (x << 2) + 1);
    else // If n is even
        return (square(x) << 2);
}

```

Method 2: breaking num in 2's form

```

int square(int num)

```

```

{
    // handle negative input
    if (num < 0)
        num = -num;

    // Initialize power of 2 and result
    int power = 0, result = 0;
    int temp = num;

    while (temp) {
        if (temp & 1) {
            // result=result+(num*(2^power))
            result += (num << power);
        }
        power++;

        // temp=temp/2
        temp = temp >> 1;
    }

    return result;
}

```

Power Set

Given a string s of length n , find all the possible non-empty subsequences of the string s in lexicographically-sorted order.

```

vector<string> allPossibleStrings(string &s) {
    int n = s.size();
    vector<string> res;

    // Iterate through all subsets (represented by 0 to  $2^n - 1$ )
    for (int i = 0; i < (1 << n); i++) {
        string sub = "";
        for (int j = 0; j < n; j++) {
            if (i & (1 << j)) sub += s[j];
        }
        res.push_back(sub);
    }
}

```

```
        return res;
    }
```

Count total set bits in first N Natural Numbers (all numbers from 1 to N)

Naive Method (Count one by One)

Time Complexity: $O(N \log(N))$, where N is the given integer and $\log(N)$ time is used for the binary conversion of the number.

Auxiliary Space: $O(1)$.

```
/* Function to get no of set bits in binary
representation of positive integer n */
int countSetBits(int n)
{
    int count = 0;
    while (n) {
        count += n & 1;
        n >>= 1;
    }
    return count;
}

// Count total set bits in all
// numbers from 1 to n
int countAllSetBits(int n)
{
    int res = 0;
    for (int i=1; i<=n; i++)
        res += countSetBits(i);
    return res;
}
```

Better Approach: Time Complexity: $O(k \cdot n)$
where k = number of bits to represent number n
 $k \leq 64$

If we observe bits from rightmost side at distance i than bits get inverted after 2^i position in vertical sequence.

for example $n = 5$;

```
0 = 0000
1 = 0001
2 = 0010
3 = 0011
4 = 0100
5 = 0101
```

Observe the right most bit ($i = 0$) the bits get flipped after ($2^0 = 1$)

Observe the 3rd rightmost bit ($i = 2$) the bits get flipped after ($2^2 = 4$)

So, We can count bits in vertical fashion such that at i 'th right most position bits will be get flipped after 2^i iteration;

```
int countSetBits(int n)
{
    int i = 0;

    // ans store sum of set bits from 0 to n
    int ans = 0;

    // while n greater than equal to  $2^i$ 
    while ((1 << i) <= n) {

        // This k will get flipped after
        //  $2^i$  iterations
        bool k = 0;

        // change is iterator from  $2^i$  to 1
        int change = 1 << i;

        // This will loop from 0 to n for
        // every bit position
        for (int j = 0; j <= n; j++) {

            ans += k;

            if (change == 1) {
                k = !k; // When change = 1 flip the bit
                change = 1 << i; // again set change to  $2^i$ 
            }
            else {
```

```

        change--;
    }
}

// increment the position
i++;
}

return ans;
}

```

Tricky Method

If the input number is of the form $2^b - 1$ e.g., 1, 3, 7, 15.. etc, the number of set bits is $b * 2^{(b-1)}$. This is because for all the numbers 0 to $(2^b)-1$, if you complement and flip the list you end up with the same list (half the bits are on, half off).

If the number does not have all set bits, then some position m is the position of leftmost set bit. The number of set bits in that position is $n - (1 << m) + 1$.

The remaining set bits are in two parts:

- 1) The bits in the $(m-1)$ positions down to the point where the leftmost bit becomes 0, and
- 2) The $2^{(m-1)}$ numbers below that point, which is the closed form above.

An easy way to look at it is to consider the number 6:

0 0 0
0 0 1
0 1 0
0 1 1
- --
1 0 0
1 0 1

1|1 0

The leftmost set bit is in position 2 (positions are considered starting from 0). If we mask that off what remains is 2 (the “1 0” in the right part of the last row.) So the number of bits in the 2nd position (the lower left box) is 3 (that is, $2 + 1$). The set bits from 0-3 (the upper right box above) is $2 * 2^{(2-1)} = 4$. The box in the lower right is the remaining bits we haven’t yet counted, and is the number of set bits for all the numbers up to 2 (the value of the last entry in the lower right box) which can be figured recursively.

```
// The main recursive function used by countSetBits()
unsigned int _countSetBits(unsigned int n, int m);

// Returns count of set bits present in
// all numbers from 1 to n
unsigned int countSetBits(unsigned int n)
{
    // Get the position of leftmost set
    // bit in n. This will be used as an
    // upper bound for next set bit function
    int m = getLeftmostBit(n);

    // Use the position
    return _countSetBits(n, m);
}

unsigned int _countSetBits(unsigned int n, int m)
{
    // Base Case: if n is 0, then set bit
    // count is 0
    if (n == 0)
        return 0;

    /* get position of next leftmost set bit */
    m = getNextLeftmostBit(n, m);
```

```

// If n is of the form 2^x-1, i.e., if n
// is like 1, 3, 7, 15, 31, .. etc,
// then we are done.
// Since positions are considered starting
// from 0, 1 is added to m
if (n == ((unsigned int)1 << (m + 1)) - 1)
    return (unsigned int)(m + 1) * (1 << m);

// update n for next recursive call
n = n - (1 << m);
return (n + 1) + countSetBits(n) + m * (1 << (m - 1));
}

```

Trie:

Implement a Trie From Scratch:

```

#include <iostream>
#include <vector>
#include <string>
using namespace std;

struct TrieNode {

    // Array for child nodes of each node
    TrieNode* child[26];

    // for end of word
    bool wordEnd;

    TrieNode() {
        wordEnd = false;
        for (int i = 0; i < 26; i++) {
            child[i] = nullptr;
        }
    }
}

```

```
};

// Method to insert a key into the Trie
void insertKey(TrieNode* root, const string& key) {

    // Initialize the curr pointer with the root node
    TrieNode* curr = root;

    // Iterate across the length of the string
    for (char c : key) {

        // Check if the node exists for the
        // current character in the Trie
        if (curr->child[c - 'a'] == nullptr) {

            // If node for current character does
            // not exist then make a new node
            TrieNode* newNode = new TrieNode();

            // Keep the reference for the newly
            // created node
            curr->child[c - 'a'] = newNode;
        }

        // Move the curr pointer to the
        // newly created node
        curr = curr->child[c - 'a'];
    }

    // Mark the end of the word
    curr->wordEnd = true;
}

// Method to search a key in the Trie
bool searchKey(TrieNode* root, const string& key) {

    if (root == nullptr) {
        return false;
    }

    // Initialize the curr pointer with the root node
    TrieNode* curr = root;
```

```

// Iterate across the length of the string
for (char c : key) {

    // Check if the node exists for the
    // current character in the Trie
    if (curr->child[c - 'a'] == nullptr)
        return false;

    // Move the curr pointer to the
    // already existing node for the
    // current character
    curr = curr->child[c - 'a'];
}

// Return true if the word exists
// and is marked as ending
return curr->wordEnd;
}

int main() {

    // Create an example Trie
    TrieNode* root = new TrieNode();
    vector<string> arr =
        {"and", "ant", "do", "geek", "dad", "ball"};
    for (const string& s : arr) {
        insertKey(root, s);
    }

    // One by one search strings
    vector<string> searchKeys = {"do", "gee", "bat"};
    for (string& s : searchKeys) {
        cout << "Key : " << s << "\n";
        if (searchKey(root, s))
            cout << "Present\n";
        else
            cout << "Not Present\n";
    }

    return 0;
}

```

Find shortest unique prefix for every word in a given list | Set 1 (Using Trie)

A **Simple Solution** is to consider every prefix of every word (starting from the shortest to largest), and if a prefix is not prefix of any other string, then print it.

An **Efficient Solution** is to use [Trie](#). The idea is to maintain a count in every node. Below are steps.

- 1) Construct a [Trie](#) of all words. Also maintain frequency of every node (Here frequency is number of times node is visited during insertion). Time complexity of this step is $O(N)$ where N is total number of characters in all words.
- 2) Now, for every word, we find the character nearest to the root with frequency as 1. The prefix of the word is path from root to this character. To do this, we can traverse Trie starting from root. For every node being traversed, we check its frequency. If frequency is one, we print all characters from root to this node and don't traverse down this node.

Time complexity if this step also is $O(N)$ where N is total number of characters in all words.

```
// C++ program to print all prefixes that
// uniquely represent words.
#include<bits/stdc++.h>
using namespace std;

#define MAX 256

// Maximum length of an input word
#define MAX_WORD_LEN 500

// Trie Node.
struct trieNode
{
```

```

        struct trieNode *child[MAX];
        int freq; // To store frequency
    };

// Function to create a new trie node.
struct trieNode *newTrieNode(void)
{
    struct trieNode *newNode = new trieNode;
    newNode->freq = 1;
    for (int i = 0; i<MAX; i++)
        newNode->child[i] = NULL;
    return newNode;
}

// Method to insert a new string into Trie
void insert(struct trieNode *root, string str)
{
    // Length of the URL
    int len = str.length();
    struct trieNode *pCrawl = root;

    // Traversing over the length of given str.
    for (int level = 0; level<len; level++)
    {
        // Get index of child node from current character
        // in str.
        int index = str[level];

        // Create a new child if not exist already
        if (!pCrawl->child[index])
            pCrawl->child[index] = newTrieNode();
        else
            (pCrawl->child[index]->freq)++;

        // Move to the child
        pCrawl = pCrawl->child[index];
    }
}

// This function prints unique prefix for every word stored
// in Trie. Prefixes one by one are stored in prefix[].
// 'ind' is current index of prefix[]
void findPrefixesUtil(struct trieNode *root, char prefix[],
```

```

        int ind)
{
    // Corner case
    if (root == NULL)
        return;

    // Base case
    if (root->freq == 1)
    {
        prefix[ind] = '\0';
        cout << prefix << " ";
        return;
    }

    for (int i=0; i<MAX; i++)
    {
        if (root->child[i] != NULL)
        {
            prefix[ind] = i;
            findPrefixesUtil(root->child[i], prefix, ind+1);
        }
    }
}

// Function to print all prefixes that uniquely
// represent all words in arr[0..n-1]
void findPrefixes(string arr[], int n)
{
    // Construct a Trie of all words
    struct trieNode *root = newTrieNode();
    root->freq = 0;
    for (int i = 0; i<n; i++)
        insert(root, arr[i]);

    // Create an array to store all prefixes
    char prefix[MAX_WORD_LEN];

    // Print all prefixes using Trie Traversal
    findPrefixesUtil(root, prefix, 0);
}

// Driver function.
int main()

```

```

{
    string arr[] = {"zebra", "dog", "duck", "dove"};
    int n = sizeof(arr)/sizeof(arr[0]);
    findPrefixes(arr, n);

    return 0;
}

```

Word Break Problem | Trie Solution

```

// A DP and Trie based program to test whether
// a given string can be segmented into
// space separated words in dictionary
#include <iostream>
using namespace std;

const int ALPHABET_SIZE = 26;

// trie node
struct TrieNode {
    struct TrieNode* children[ALPHABET_SIZE];

    // isEndOfWord is true if the node represents
    // end of a word
    bool isEndOfWord;
};

// Returns new trie node (initialized to NULLs)
struct TrieNode* getNode(void)
{
    struct TrieNode* pNode = new TrieNode;

    pNode->isEndOfWord = false;

    for (int i = 0; i < ALPHABET_SIZE; i++)
        pNode->children[i] = NULL;

    return pNode;
}

```

```

// If not present, inserts key into trie
// If the key is prefix of trie node, just
// marks leaf node
void insert(struct TrieNode* root, string key)
{
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            pCrawl->children[index] = getNode();

        pCrawl = pCrawl->children[index];
    }

    // mark last node as leaf
    pCrawl->isEndOfWord = true;
}

// Returns true if key presents in trie, else
// false
bool search(struct TrieNode* root, string key)
{
    struct TrieNode* pCrawl = root;

    for (int i = 0; i < key.length(); i++) {
        int index = key[i] - 'a';
        if (!pCrawl->children[index])
            return false;

        pCrawl = pCrawl->children[index];
    }

    return (pCrawl != NULL && pCrawl->isEndOfWord);
}

// returns true if string can be segmented into
// space separated words, otherwise returns false
bool wordBreak(string str, TrieNode* root)
{
    int size = str.size();

```

```

// Base case
if (size == 0)
    return true;

// Try all prefixes of lengths from 1 to size
for (int i = 1; i <= size; i++) {
    // The parameter for search is str.substr(0, i)
    // str.substr(0, i) which is prefix (of input
    // string) of length 'i'. We first check whether
    // current prefix is in dictionary. Then we
    // recursively check for remaining string
    // str.substr(i, size-i) which is suffix of
    // length size-i
    if (search(root, str.substr(0, i))
        && wordBreak(str.substr(i, size - i), root))
        return true;
}

// If we have tried all prefixes and none
// of them worked
return false;
}

// Driver program to test above functions
int main()
{
    string dictionary[]
        = { "mobile", "samsung", "sam", "sung", "ma\n",
            "mango", "icecream", "and", "go", "i",
            "like", "ice",     "cream" };
    int n = sizeof(dictionary) / sizeof(dictionary[0]);
    struct TrieNode* root = getNode();

    // Construct trie
    for (int i = 0; i < n; i++)
        insert(root, dictionary[i]);

    wordBreak("ilikesamsung", root) ? cout << "Yes\n"
                                    : cout << "No\n";
    wordBreak("iiiiiiii", root) ? cout << "Yes\n"
                                : cout << "No\n";
    wordBreak("", root) ? cout << "Yes\n" : cout << "No\n";
    wordBreak("ilikelikeimangoiii", root) ? cout << "Yes\n"

```

```

        : cout <<
"No\n";
    wordBreak("samsungandmango", root) ? cout << "Yes\n"
                                    : cout << "No\n";
    wordBreak("samsungandmangok", root) ? cout << "Yes\n"
                                    : cout <<
"No\n";
    return 0;
}

```

Implement a Phone Directory

Input:

```

n = 3
contact[] = {"geekistest", "geeksforgeeks",
"geeksfortest"}
s = "geeips"

```

Output:

```

geekistest geeksforgeeks geeksfortest
geekistest geeksforgeeks geeksfortest
geekistest geeksforgeeks geeksfortest
geekistest
0
0

```

Explanation: By running the search query on contact list for "g" we get: "geekistest", "geeksforgeeks" and "geeksfortest".

By running the search query on contact list for "ge" we get: "geekistest" "geeksforgeeks" and "geeksfortest".

By running the search query on contact list for "gee" we get: "geekistest" "geeksforgeeks" and "geeksfortest".

By running the search query on contact list for "geei" we get: "geekistest".

No results found for "geeip", so print "0".

No results found for "geeips", so print "0".

```

// C++ Program to Implement a Phone
// Directory Using Trie Data Structure

```

```

#include <bits/stdc++.h>
using namespace std;

struct TrieNode {
    // Each Trie Node contains a Map 'child'
    // where each alphabet points to a Trie
    // Node.
    // We can also use a fixed size array of
    // size 256.
    unordered_map<char, TrieNode*> child;

    // 'isLast' is true if the node represents
    // end of a contact
    bool isLast;

    // Default Constructor
    TrieNode()
    {
        // Initialize all the Trie nodes with NULL
        for (char i = 'a'; i <= 'z'; i++)
            child[i] = NULL;

        isLast = false;
    }
};

// Making root NULL for ease so that it doesn't
// have to be passed to all functions.
TrieNode* root = NULL;

// Insert a Contact into the Trie
void insert(string s)
{
    int len = s.length();

    // 'itr' is used to iterate the Trie Nodes
    TrieNode* itr = root;
    for (int i = 0; i < len; i++) {
        // Check if the s[i] is already present in
        // Trie
        TrieNode* nextNode = itr->child[s[i]];
        if (nextNode == NULL) {
            // If not found then create a new TrieNode

```

```

nextNode = new TrieNode();

        // Insert into the Map
        itr->child[s[i]] = nextNode;
    }

    // Move the iterator('itr') ,to point to next
    // Trie Node
    itr = nextNode;

    // If its the last character of the string 's'
    // then mark 'isLast' as true
    if (i == len - 1)
        itr->isLast = true;
    }
}

// This function simply displays all dictionary words
// going through current node. String 'prefix'
// represents string corresponding to the path from
// root to curNode.
void displayContactsUtil(TrieNode* curNode, string prefix)
{
    // Check if the string 'prefix' ends at this Node
    // If yes then display the string found so far
    if (curNode->isLast)
        cout << prefix << endl;

    // Find all the adjacent Nodes to the current
    // Node and then call the function recursively
    // This is similar to performing DFS on a graph
    for (char i = 'a'; i <= 'z'; i++) {
        TrieNode* nextNode = curNode->child[i];
        if (nextNode != NULL)
            displayContactsUtil(nextNode, prefix + (char)i);
    }
}

// Display suggestions after every character enter by
// the user for a given query string 'str'
void displayContacts(string str)
{
    TrieNode* prevNode = root;

```

```

string prefix = "";
int len = str.length();

// Display the contact List for string formed
// after entering every character
int i;
for (i = 0; i < len; i++) {
    // 'prefix' stores the string formed so far
    prefix += (char)str[i];

    // Get the last character entered
    char lastChar = prefix[i];

    // Find the Node corresponding to the last
    // character of 'prefix' which is pointed by
    // prevNode of the Trie
    TrieNode* curNode = prevNode->child[lastChar];

    // If nothing found, then break the loop as
    // no more prefixes are going to be present.
    if (curNode == NULL) {
        cout << "\nNo Results Found for " << prefix
            << "\n";
        i++;
        break;
    }

    // If present in trie then display all
    // the contacts with given prefix.
    cout << "\nSuggestions based on " << prefix
        << "are ";
    displayContactsUtil(curNode, prefix);

    // Change prevNode for next prefix
    prevNode = curNode;
}

// Once search fails for a prefix, we print
// "Not Results Found" for all remaining
// characters of current query string "str".
for (; i < len; i++) {
    prefix += (char)str[i];
}

```

```

        cout << "\nNo Results Found for " << prefix << "\n";
    }
}

// Insert all the Contacts into the Trie
void insertIntoTrie(string contacts[], int n)
{
    // Initialize root Node
    root = new TrieNode();

    // Insert each contact into the trie
    for (int i = 0; i < n; i++)
        insert(contacts[i]);
}

// Driver program to test above functions
int main()
{
    // Contact list of the User
    string contacts[] = { "gforgeeks", "geeksquiz" };

    // Size of the Contact List
    int n = sizeof(contacts) / sizeof(string);

    // Insert all the Contacts into Trie
    insertIntoTrie(contacts, n);

    string query = "gekk";

    // Note that the user will enter 'g' then 'e', so
    // first display all the strings with prefix as 'g'
    // and then all the strings with prefix as 'ge'
    displayContacts(query);

    return 0;
}

```

Unique rows in boolean matrix

```

class Solution
{
public:
// #define MAX 1000
vector<vector<int>> uniqueRow(int M[MAX][MAX],int row,int col)
{
    //Your code here
    set<vector<int> >ans;

    vector<vector<int>> gg;
    for(auto x: ans) gg.push_back(x);
    for(int i=0;i<row;i++){
        vector<int> ins;
        for(int j=0;j<col;j++){
            ins.push_back(M[i][j]);
        }
        if(ans.find(ins)==ans.end()){
            ans.insert(ins);
            gg.push_back(ins);
        }
    }

    return gg;
}
};

```

General Questions for GS:

Find the total lattice points on the circumference of a circle. Ref:

<https://www.geeksforgeeks.org/circle-lattice-points/>

Given a circle of radius r in 2-D with origin or $(0, 0)$ as center. The task is to find the total lattice points on circumference. Lattice Points are points with coordinates as integers in 2-D space.

Example:

```
int countLattice(int r)
{
    if (r <= 0)
        return 0;

    // Initialize result as 4 for (r, 0), (-r, 0),
    // (0, r) and (0, -r)
    int result = 4;

    // Check every value that can be potential x
    for (int x=1; x<r; x++)
    {
        // Find a potential y
        int ySquare = r*r - x*x;
        int y = sqrt(ySquare);

        // checking whether square root is an integer
        // or not. Count increments by 4 for four
        // different quadrant values
        if (y*y == ySquare)
            result += 4;
    }

    return result;
}
```

First non-repeating character of given string

Input: s = "geeksforgeeks"

Output: 'f'

Explanation: 'f' is the first character in the string which does not repeat.

Input: s = "racecar"

Output: 'e'

Explanation: 'e' is the only character in the string which does not repeat.

Input: "aabbcce"

Output: '\$'

Explanation: All the characters in the given string are repeating.

```
const int MAX_CHAR = 26;

char nonRep(const string& s) {
    vector<int> vis(MAX_CHAR, -1);
    for (int i = 0; i < s.length(); ++i) {
        int index = s[i] - 'a';
        if (vis[index] == -1) {

            // Store the index when character is first seen
            vis[index] = i;
        } else {

            // Mark character as repeated
            vis[index] = -2;
        }
    }

    int idx = -1;

    // Find the smallest index of the non-repeating characters
    for (int i = 0; i < MAX_CHAR; ++i) {
        if (vis[i] >= 0 && (idx == -1 || vis[i] < vis[idx])) {
            idx = i;
        }
    }
    return (idx == -1) ? '$' : s[vis[idx]]; // notice here
```

```
}
```

Arrange given numbers to form the biggest number

Input: arr[] = [3, 30, 34, 5, 9]

Output: "9534330"

Explanation: Given numbers are [3, 30, 34, 5, 9], the arrangement

"9534330" gives the largest value.

Input: arr[] = [54, 546, 548, 60]

Output: "6054854654"

Explanation: Given numbers are [54, 546, 548, 60], the arrangement

"6054854654" gives the largest value.

Input: arr[] = [3, 4, 6, 5, 9]

Output: "96543"

Explanation: Given numbers are [3, 4, 6, 5, 9], the arrangement "96543"

gives the largest value.

```
static bool comp(string a, string b){  
    string na = a+b;  
    string nb = b+a;  
    return na < nb;  
}  
string findLargest(vector<int> &arr) {  
    // code here  
  
    int n = arr.size();  
    vector<string> vs(n);  
    bool all_zeros = true;  
    for(int i=0;i<n;i++){  
        vs[i]=to_string(arr[i]);
```

```
        if(arr[i]!=0) all_zeros = false;
    }
    if(all_zeros) return "0";
    sort(vs.rbegin(),vs.rend(), comp);
    string ans = "";
    for(auto x: vs){
        // cout<<x<<" ";
        ans += x;
    }
    return ans;
}
```

Min flip to make binary string alternate

<https://www.geeksforgeeks.org/number-flips-make-binary-string-alternate/>

Maximum possible stolen value from houses.

<https://www.geeksforgeeks.org/find-maximum-possible-stolen-value-houses/>

Water Jug Puzzle

Round table coin game

<https://www.geeksforgeeks.org/puzzle-round-table-coin-game/>

First Missing Positive

Example 1:

Input: nums = [1,2,0]

Output: 3

Explanation: The numbers in the range [1,2] are all in the array.

Example 2:

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

Output: 2

Explanation: 1 is in the array but 2 is missing.

Example 3:

Input: nums = [7,8,9,11,12]

Output: 1

Explanation: The smallest positive integer 1 is missing.

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        int n = nums.size();

        for (int i = 0; i < n; i++) {
            while (
                nums[i] > 0 && nums[i] <= n &&
                nums[nums[i] - 1] != nums[i]
            ) {
                swap(nums[i], nums[nums[i] - 1]);
            }
        }

        for (int i = 0; i < n; i++) {
            if (nums[i] != i + 1)
                return i + 1;
        }

        return n + 1;
    }
};
```

Longest Substring Without Repeating Characters

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        int maxLength = 0;
        int left = 0;
        unordered_map<char, int> count;

        for (int right = 0; right < s.length(); right++) {
            char c = s[right];
            count[c] = count[c] + 1;

            while (count[c] > 1) {
                char leftChar = s[left];
                count[leftChar] = count[leftChar] - 1;
                left++;
            }

            maxLength = max(maxLength, right - left + 1);
        }

        return maxLength;
    }
};
```

Coin Change: Min coins to form a sum

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<int> minCoins(amount + 1, amount + 1);
        minCoins[0] = 0;

        for (int i = 1; i <= amount; i++) {
            for (int j = 0; j < coins.size(); j++) {
                if (i - coins[j] >= 0) {
                    minCoins[i] = min(minCoins[i], 1 + minCoins[i - coins[j]]);
                }
            }
        }

        return minCoins[amount];
    }
};
```

```

        }

        return minCoins[amount] != amount + 1 ? minCoins[amount] : -1;
    }
};
```

Find Median from Running Data Stream

*The median of an array occurs at the center of sorted array, so the idea is to store the current elements in two nearly equal parts. A **max heap** (left half) stores the smaller elements, ensuring the largest among them is at the top, while a **min heap** (right half) stores the larger elements, keeping the smallest at the top.*

For each new element:

1. *It is first added to the max heap.*
2. *The max heap's top element is moved to the min heap to maintain order.*
3. *If the min heap has more elements than the max heap, its top element is moved back to ensure balance.*

This keeps both halves nearly equal in size, differing by at most one element. If the heaps are balanced, the median is the average of their root values; otherwise, it is the root of the heap with more elements.

```
// Function to find the median of a stream of data
vector<double> getMedian(vector<int> &arr) {

    // Max heap to store the smaller half of numbers
    priority_queue<int> leftMaxHeap;
```

```

// Min heap to store the greater half of numbers
priority_queue<int, vector<int>, greater<int>> rightMinHeap;

vector<double> res;

for (int i = 0; i < arr.size(); i++) {
    // Insert new element into max heap
    leftMaxHeap.push(arr[i]);

    // Move the top of max heap to min heap to maintain order
    int temp = leftMaxHeap.top();
    leftMaxHeap.pop();
    rightMinHeap.push(temp);

    // Balance heaps if min heap has more elements
    if (rightMinHeap.size() > leftMaxHeap.size()) {
        temp = rightMinHeap.top();
        rightMinHeap.pop();
        leftMaxHeap.push(temp);
    }

    // Compute median based on heap sizes
    double median;
    if (leftMaxHeap.size() != rightMinHeap.size())
        median = leftMaxHeap.top();
    else
        median = (double)(leftMaxHeap.top() + rightMinHeap.top()) / 2;

    res.push_back(median);
}

return res;
}

```

Kth Largest Element in an Array

The QuickSelect algorithm is an efficient method to find the k-th smallest (or largest) element in an unordered list without sorting the entire list. It works similarly to the QuickSort algorithm but only recurses into one half of the data.

Key Data Structures: List/Array: We use Python's built-in list for this approach. The algorithm modifies the list in place. **Pivot:** An element chosen from the list, around which the list gets partitioned. **Step-by-step Breakdown:** Initialization:

Set the left boundary to the beginning of the list and the right boundary to the end of the list.
Pivot Selection:

Randomly select a pivot index between the left and right boundaries. **Partitioning:**

Move all elements smaller than the pivot to its left and all larger elements to its right. Return the final position of the pivot after the partitioning. **Check Pivot Position:**

If the position of the pivot is the desired k-th largest index, return the pivot. If the pivot's position is greater than the desired index, adjust the right boundary and repeat. If the pivot's position is lesser than the desired index, adjust the left boundary and repeat. **Result:**

The function will eventually return the k-th largest element in the original list. Example: Let's walk through the QuickSelect algorithm using the list `nums = [3,2,1,5,6,4]` and (`k = 2`) to find the 2nd largest element.

Initial List: [3,2,1,5,6,4]

Iteration 1: Chosen pivot: 3 (at index 0) After partitioning, the list becomes: [2, 1, 3, 5, 6, 4] The new pivot index is 2. Since we're looking for the 2nd largest element (index 4 in 0-indexed list), and the current pivot index is less than this, we know the desired element is to the right of the current pivot. **Iteration 2:** Chosen pivot: 6 (at index 4) After partitioning, the list becomes: [2, 1, 3, 4, 5, 6] The new pivot index is 4, which matches our target index for the 2nd largest element. **Result:** The 2nd largest element in the list is 5.

This example demonstrates the behavior of the QuickSelect algorithm. By iteratively selecting a pivot and partitioning the list around that pivot, it efficiently narrows down the search space until it locates the kth largest element.

Complexity: Time Complexity:

Best and Average Case: $O(N)$

Worst Case: $O(N^2)$

The average performance is linear. However, in the worst case (very rare, especially with randomized pivot), the algorithm can degrade to $O(N^2)$.

Space Complexity: $O(1)$

The space used is constant. The algorithm modifies the original list in place and doesn't utilize any significant additional data structures. The recursive stack calls (in the worst case) are also bounded by the depth of the list, making it $O(\log N)$, but this is typically considered as $O(1)$ space complexity in QuickSelect. Performance: This solution is efficient for larger lists, especially when the pivot is chosen randomly, which greatly reduces the chance of the worst-case scenario. The QuickSelect algorithm allows for finding the desired element without sorting the entire list, making it faster than the sorting approach for large datasets.

```
class Solution {
public:
    int findKthLargest(std::vector<int>& nums, int k) {
        int left = 0, right = nums.size() - 1;
        while (true) {
            int pivot_index = rand() % (right - left + 1) + left;
            int new_pivot_index = partition(nums, left, right, pivot_index);
            if (new_pivot_index == nums.size() - k) {
                return nums[new_pivot_index];
            } else if (new_pivot_index > nums.size() - k) {
                right = new_pivot_index - 1;
            } else {
                left = new_pivot_index + 1;
            }
        }
    }

private:
    int partition(std::vector<int>& nums, int left, int right, int pivot_index) {
        int pivot = nums[pivot_index];
        std::swap(nums[pivot_index], nums[right]);
        int stored_index = left;
        for (int i = left; i < right; i++) {
            if (nums[i] < pivot) {
                std::swap(nums[i], nums[stored_index]);
                stored_index++;
            }
        }
        std::swap(nums[right], nums[stored_index]);
        return stored_index;
    }
};
```

Minimum Window Substring

Given two strings s and t of lengths m and n respectively, return the minimum window substring of s such that every character in t (including duplicates) is included in the window. If there is no such substring, return the empty string "".

The testcases will be generated such that the answer is unique.

Example 1:

Input: s = "ADOBECODEBANC", t = "ABC"

Output: "BANC"

Explanation: The minimum window substring "BANC" includes 'A', 'B', and 'C' from string t.

Example 2:

Input: s = "a", t = "a"

Output: "a"

Explanation: The entire string s is the minimum window.

Example 3:

Input: s = "a", t = "aa"

Output: ""

Explanation: Both 'a's from t must be included in the window.

Since the largest window of s only has one 'a', return empty string.

```
class Solution {
public:
    string minWindow(string s, string t) {
        unordered_map<char,int> mp;
        for(auto ch:t)
        {
            mp[ch]++;
        }
        int dist=mp.size() ;
        unordered_map<char,int> window;
        int count = 0 , ll = 0 , rr = 0 ;
        int l = 0 , r = 0 , ans = INT_MAX ;
        while(r<s.length())
        {
            window[s[r]]++ ;
            if(mp.count(s[r]) and mp[s[r]]==window[s[r]])
            {
                count++;
            }
        }
    }
};
```

```

    r++;
    while(count == dist and l < r)
    {
        if(ans > r-1)
        {
            ans= r - 1 ;
            ll = l ;
            rr = r ;
        }
        window[s[1]]-- ;
        if(mp.count(s[1]) and window[s[1]] < mp[s[1]])
        {
            count--;
        }
        l++;
    }
    return s.substr(ll,rr-l);
}
};

```

Regular Expression Matching

Given an input string s and a pattern p , implement regular expression matching with support for '.' and '*' where:

'.' Matches any single character.

'*' Matches zero or more of the preceding element.

The matching should cover the entire input string (not partial).

Example 1:

Input: $s = "aa"$, $p = "a"$

Output: false

Explanation: "a" does not match the entire string "aa".

Example 2:

Input: $s = "aa"$, $p = "a^*$

Output: true

Explanation: "*" means zero or more of the preceding element, 'a'. Therefore, by repeating 'a' once, it becomes "aa".

Example 3:

Input: s = "ab", p = ".*"

Output: true

Explanation: ".*" means "zero or more (*) of any character (.)".

```
class Solution {
public:

    string S;
    string P;

    bool solver(int i, int j) {

        if (i >= S.size() && j >= P.size()) {
            // both out of bounds, that means matched
            return true;
        }
        if (j >= P.size()) {
            // only pattern out bounds, that means some characters are still
            remaining to be matched
            // so not possible to match
            return false;
        }

        bool matched = (i < S.size()) && (S[i] == P[j] || P[j] == '.');

        if (j+1 < P.size() && P[j+1] == '*') {
            // you have a * next in the pattern

            // don't repeat and move ahead and recur
            bool dontChoose = solver(i, j+2);

            // repeat and recur
            // can choose only if the character matches
            bool choose = matched && solver(i+1, j);

            return dontChoose || choose;
        }

        if (matched) {
            // character matched but no * in the pattern
            // increment both
            return solver(i+1, j+1);
        }
    }
}
```

```

    }

    return false;
}

bool isMatch(string s, string p) {
    S = s;
    P = p;

    return solver(0, 0);
}
};


```

Solution 2:

Create a boolean 2D dp array of size $(n + 1) * (m + 1)$. Please note that the range of values in the recursion goes from 0 to text length (or n) and 0 to pattern length (or m)
 $dp[i][j]$ is going to be true if first i characters of text match with first j characters of pattern.
If both strings are empty, then it's a match, thus, $dp[0][0] = \text{true}$.
For other cases, we simply follow the above recursive solution.

```

bool isMatch(string t, string p) {
    int n = t.size();
    int m = p.size();

    // DP table where dp[i][j] means whether first i characters in t
    // match the first j characters in p
    vector<vector<bool>> dp(n + 1, vector<bool>(m + 1, false));

    // Empty pattern matches empty text
    dp[0][0] = true;

    // Deals with patterns like a*, a*b*, a*b*c* etc, where '*'
    // can eliminate preceding character
    for (int j = 1; j <= m; ++j) {
        if (p[j - 1] == '*' && j > 1) {
            dp[0][j] = dp[0][j - 2];
        }
    }

    // Fill the table
    for (int i = 1; i <= n; ++i) {
        for (int j = 1; j <= m; ++j) {
            if (t[i - 1] == p[j - 1]) {
                dp[i][j] = dp[i - 1][j - 1];
            } else if (p[j - 1] == '*') {
                dp[i][j] = dp[i][j - 1] || dp[i - 1][j];
            }
        }
    }
}


```

```

for (int j = 1; j <= m; ++j) {

    // Characters match
    if (p[j - 1] == '.' || t[i - 1] == p[j - 1]) {
        dp[i][j] = dp[i - 1][j - 1];
    }

    else if (p[j - 1] == '*' && j > 1) {

        // Two cases:
        // 1. '*' represents zero occurrence of the preceding
character
        // 2. '*' represents one or more occurrence of the
preceding character
        dp[i][j] = dp[i][j - 2] ||
                    (dp[i - 1][j] && (p[j - 2] == t[i - 1] || p[j -
2] == '.'));
    }
}

return dp[n][m];
}

```

```

class Solution:
    def isMatch(self, s: str, p: str) -> bool:
        m, n = len(s), len(p)
        cache = {}

        def dfs(i, j):
            if j == n:
                return i == m
            if (i, j) in cache:
                return cache[(i, j)]

            match = i < m and (s[i] == p[j] or p[j] == ".")
            if (j + 1) < n and p[j + 1] == "*":
                cache[(i, j)] = (dfs(i, j + 2) or # don't use *
                                (match and dfs(i + 1, j))) # use *

        return cache[(i, j)]

```

```

if match:
    cache[(i, j)] = dfs(i + 1, j + 1)
    return cache[(i, j)]

cache[(i, j)] = False
return False

return dfs(0, 0)

```

Burst Balloons | Partition DP | DP 51

You are given n balloons, indexed from 0 to n - 1. Each balloon is painted with a number on it represented by an array. You are asked to burst all the balloons.

If you burst the ith balloon, you will get $\text{arr}[i - 1] * \text{arr}[i] * \text{arr}[i + 1]$ coins. If $i - 1$ or $i + 1$ goes out of the array's bounds, then treat it as if there is a balloon with a 1 painted on it.

Return the maximum coins you can collect by bursting the balloons wisely.

Example 1:

Input: N = 4, array[] = {3, 1, 5, 8}

Output: 167

Explanation:

First, we will burst the second balloon with the value 1. Coins = $3 * 1 * 5 = 15$.

Second, we will burst the balloon with the value 5. Coins = $3 * 5 * 8 = 120$.

Third, we will burst the balloon with the value 3. Coins = $1 * 3 * 8 = 24$.

Fourth, we will burst the balloon with the value 8. Coins = $1 * 8 * 1 = 8$.

So, the total number of coins we can collect is 167. This is the maximum number of coins we can collect.

```

// Recursive function to calculate the maximum coins obtained
int maxCoinsHelper(int i, int j, vector<int> &nums, vector<vector<int>>
&dp) {
    if (i > j) return 0;
    if (dp[i][j] != -1) return dp[i][j];
    int maxCoins = INT_MIN;

    // Iterate through each possible balloon to burst last
    for (int k = i; k <= j; k++) {
        // Calculate the coins obtained by bursting the k-th balloon last
        int coins = nums[i - 1] * nums[k] * nums[j + 1];

        // Recursively calculate the maximum coins for the remaining
        balloons
        int remainingCoins = maxCoinsHelper(i, k - 1, nums, dp) +
maxCoinsHelper(k + 1, j, nums, dp);

        // Update the maximum coins
        maxCoins = max(maxCoins, coins + remainingCoins);
    }

    return dp[i][j] = maxCoins;
}

// Function to calculate the maximum coins obtained
int maxCoins(vector<int> &nums) {
    int n = nums.size();

    // Add 1 to the beginning and end of the nums array
    nums.insert(nums.begin(), 1);
    nums.push_back(1);

    // Create a DP array for memoization
    vector<vector<int>> dp(n + 2, vector<int>(n + 2, -1));

    // Call the helper function to compute the maximum coins
    return maxCoinsHelper(1, n, nums, dp);
}

```

Trapping Rain Water

Brute force:

```
int trap(vector<int>& height) {
    if (height.empty()) {
        return 0;
    }
    int n = height.size();
    int res = 0;

    for (int i = 0; i < n; i++) {
        int leftMax = height[i];
        int rightMax = height[i];

        for (int j = 0; j < i; j++) {
            leftMax = max(leftMax, height[j]);
        }
        for (int j = i + 1; j < n; j++) {
            rightMax = max(rightMax, height[j]);
        }

        res += min(leftMax, rightMax) - height[i];
    }
    return res;
}
```

using stack optimal:

```
class Solution {
public:
    int trap(vector<int>& height) {
        stack<int> stk;
        int res = 0;
        int n = height.size();
        if(n ==0) return 0;

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

```

        while(!stk.empty() and height[i]>=
height[stk.top()]){
            int m = height[stk.top()];
            stk.pop();
            if(!stk.empty()){
                int right = height[i];
                int left = height[stk.top()];
                int h = min(right, left)-m;
                int width = i - stk.top() - 1;
                res += h * width;
            }
        }
        stk.push(i);
    }
    return res;
}

};


```

Optimal Two Pointer

```

class Solution {
public:
    int trap(vector<int>& height) {
        int n = height.size();
        int left = 0, right=n-1;
        int leftmx = height[left], rightmx = height[right];
        int res = 0;
        while(left<right){
            if(leftmx >= rightmx){
                right--;
                rightmx = max(rightmx, height[right]);
            }
            else{
                left++;
                leftmx = max(leftmx, height[left]);
            }
            res += rightmx - height[left];
        }
        return res;
    }
};


```

```

        res += rightmx - height[right];
    }else{
        left++;
        leftmx = max(leftmx, height[left]);
        res += leftmx - height[left];
    }
}

return res;
}

};

}

```

Spiral Matrix

Fraction to Recurring Decimal'

```

class Solution {
public:
    string fractionToDecimal(int numerator, int denominator) {
        if(!numerator) return "0";
        string ans = "";
        if (numerator > 0 ^ denominator > 0) ans += '-';
        long num = labs(numerator), den = labs(denominator);
        long q = num / den;
        long r = num % den;
        ans += to_string(q);

        if(r == 0) return ans;

        ans += '.';
        unordered_map<long, int> mp;

```

```

while(r != 0){
    if(mp.find(r) != mp.end()){
        int pos = mp[r];
        ans.insert(pos, "(");
        ans += ')';
        break;
    }
    else{
        mp[r] = ans.length();
        r *= 10;
        q = r / den;
        r = r % den;
        ans += to_string(q);
    }
}
return ans;
};


```

High Five

```

#include <iostream>
#include <vector>
#include <map>
#include <queue> // For std::priority_queue
#include <algorithm> // For std::sort (if needed, but map handles sorting by key)

// Structure to represent a student's score
struct StudentScore {
    int id;
    int score;
};

int main() {
    // Example input data (replace with actual input mechanism)
    std::vector<std::pair<int, int>> items = {
        {1, 90}, {1, 80}, {1, 70}, {1, 100}, {1, 95}, {1, 60},
        {2, 85}, {2, 92}, {2, 78}, {2, 95}, {2, 88},
        {3, 70}, {3, 75}, {3, 80}
    };
}

```

```

// Use std::map to store scores for each student.
// The value is a min-priority queue to easily keep track of the top 5
scores.
// A min-priority queue (std::greater<int>) will keep the smallest
element at the top,
// so when its size exceeds 5, we pop the smallest (which is the 6th
largest, so to speak).
std::map<int, std::priority_queue<int, std::vector<int>,
std::greater<int>>> student_top_scores;

for (const auto& item : items) {
    int student_id = item.first;
    int score = item.second;

    if (student_top_scores[student_id].size() < 5) {
        student_top_scores[student_id].push(score);
    } else {
        // If the current score is greater than the smallest of the top
5,
        // remove the smallest and add the new score.
        if (score > student_top_scores[student_id].top()) {
            student_top_scores[student_id].pop();
            student_top_scores[student_id].push(score);
        }
    }
}

std::vector<std::pair<int, int>> result;

// Iterate through the map to calculate averages
// std::map iterates in ascending order of keys (student IDs),
// so the result will naturally be sorted by student ID.
for (auto const& [student_id, scores_pq] : student_top_scores) {
    int total_marks = 0;
    int num_scores_counted = 0;

    // Create a copy of the priority queue to sum up the elements
    // without modifying the original (which is 'const&').
    // Alternatively, you can iterate by value: 'for (auto&
[student_id, scores_pq] : student_top_scores)'
    // if you don't mind scores_pq being modified.
    std::priority_queue<int, std::vector<int>, std::greater<int>>
temp_pq = scores_pq;

```

```

        while (!temp_pq.empty()) {
            total_marks += temp_pq.top();
            temp_pq.pop();
            num_scores_counted++;
        }

        // Handle the case where num_scores_counted might be zero (though
        // unlikely with the current logic if there's an entry)
        int average = (num_scores_counted > 0) ? (total_marks /
        num_scores_counted) : 0;
        result.push_back({student_id, average});
    }

    // Print the result (for verification)
    std::cout << "Student Top Five Averages:\n";
    for (const auto& p : result) {
        std::cout << "ID: " << p.first << ", Average: " << p.second <<
    "\n";
}

return 0;
}

```

IP Frequency:

```

#include <iostream>
#include <fstream>
#include <unordered_map>
#include <vector>
#include <algorithm>

std::string getMostFrequentIPs(const std::string& filename) {
    std::ifstream file(filename);
    if (!file.is_open()) {
        return "Error: Cannot open file.";
    }

    std::unordered_map<std::string, int> freq;
    std::string ip;

```

```
int maxFreq = 0;

// Count frequency of each IP
while (std::getline(file, ip)) {
    if (ip.empty()) continue;
    int count = ++freq[ip];
    if (count > maxFreq) {
        maxFreq = count;
    }
}

file.close();

// Collect IPs with max frequency
std::vector<std::string> result;
for (const auto& [key, val] : freq) {
    if (val == maxFreq) {
        result.push_back(key);
    }
}

std::sort(result.begin(), result.end());

// Join result with comma
std::string output;
for (size_t i = 0; i < result.size(); ++i) {
    output += result[i];
    if (i != result.size() - 1) {
        output += ",";
    }
}

return output;
}

int main() {
    std::string filename = "ip_list.txt"; // Change this to your actual
file path
    std::string result = getMostFrequentIPs(filename);
    std::cout << result << std::endl;
    return 0;
}
```

Minimum Path Sum:

Given a $m \times n$ grid filled with non-negative numbers, find a path from top left to bottom right, which minimizes the sum of all numbers along its path.

Note: You can only move either down or right at any point in time.

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int n=grid.size(),m=grid[0].size();
        for(int i=0;i<n;i++){
            for(int j=0;j<m;j++){
                if(i==0 and j!=0) grid[i][j]+=grid[i][j-1];
                if(j==0 and i!=0) grid[i][j]+=grid[i-1][j];
                if(i!=0 and j!=0)
                    grid[i][j]+=min(grid[i-1][j],grid[i][j-1]);
            }
        }
        return grid[n-1][m-1];
    }
};
```

Number of Sub-arrays of Size K and Average Greater than or Equal to Threshold

```
int numOfSubarrays(vector<int>& arr, int k, int t) {
    int n = arr.size();
    int s = 0;
    for(int i=0;i<k;i++) s += arr[i];
    int cnt = 0;
    if(s/k >= t) cnt++;
    for(int i=k;i<n;i++){
        s += arr[i];
        s -= arr[i-k];
        if(s/k >= t) cnt++;
    }
}
```

```
    return cnt;
}
```

Car Pooling

There is a car with capacity empty seats. The vehicle only drives east (i.e., it cannot turn around and drive west).

You are given the integer capacity and an array trips where trips[i] = [numPassenger_i, from_i, to_i] indicates that the *i*th trip has numPassenger_i passengers and the locations to pick them up and drop them off are from_i and to_i respectively. The locations are given as the number of kilometers due east from the car's initial location.

Return true if it is possible to pick up and drop off all passengers for all the given trips, or false otherwise.

```
class Solution {
public:
    bool carPooling(vector<vector<int>>& t, int c) {
        int n = t.size();
        vector<int> v(1006,0);
        for(int i=0;i<n;i++){
            v[t[i][1]]+=t[i][0];
            v[t[i][2]]-=t[i][0];
        }
        if(v[0]>c) return 0;
        for(int i=1;i<1005;i++){
            v[i]+=v[i-1];
            if(v[i]>c) return 0;
        }
        return true;
    }
};
```

Median of two sorted arrays

```
class Solution {
```

```

public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size()) {
            return findMedianSortedArrays(nums2, nums1);
        }

        int len1 = nums1.size(), len2 = nums2.size();
        int left = 0, right = len1;

        while (left <= right) {
            int part1 = (left + right) / 2;
            int part2 = (len1 + len2 + 1) / 2 - part1;

            int maxLeft1 = (part1 == 0) ? INT_MIN : nums1[part1 - 1];
            int minRight1 = (part1 == len1) ? INT_MAX : nums1[part1];
            int maxLeft2 = (part2 == 0) ? INT_MIN : nums2[part2 - 1];
            int minRight2 = (part2 == len2) ? INT_MAX : nums2[part2];

            if (maxLeft1 <= minRight2 && maxLeft2 <= minRight1) {
                if ((len1 + len2) % 2 == 0) {
                    return (max(maxLeft1, maxLeft2) + min(minRight1, minRight2)) / 2.0;
                } else {
                    return max(maxLeft1, maxLeft2);
                }
            } else if (maxLeft1 > minRight2) {
                right = part1 - 1;
            } else {
                left = part1 + 1;
            }
        }

        return 0.0;
    }
};

```

String Compression

```

class Solution {
public:
    int compress(vector<char>& chars) {
        vector<char> ans;
        int n = chars.size();

        int curr_cnt = 1;
        char curr_char = chars[0];
        int tot = 0;

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

```

```
        if(curr_char == chars[i]){
            curr_cnt += 1;
        }else{
            ans.push_back(curr_char);
            tot += 1;
            curr_char = chars[i];
            if(curr_cnt > 1){
                int t = curr_cnt;
                string gg = "";
                while(t){
                    gg = to_string(t%10) + gg;
                    t/=10;
                }
                for(char ch: gg){
                    ans.push_back(ch);
                }
            }
            curr_cnt = 1;
        }
    }
    ans.push_back(curr_char);
    curr_char = chars[n-1];
    tot += 1;
    if(curr_cnt > 1){
        int t = curr_cnt;
        string gg = "";
        while(t){
            gg = to_string(t%10) + gg;
            t/=10;
        }
        for(char ch: gg){
            ans.push_back(ch);
        }
    }
    chars = ans;
    return ans.size();
}
};
```

Robot Bounded In Circle

On an infinite plane, a robot initially stands at $(0, 0)$ and faces north. Note that:

The north direction is the positive direction of the y-axis.

The south direction is the negative direction of the y-axis.

The east direction is the positive direction of the x-axis.

The west direction is the negative direction of the x-axis.

The robot can receive one of three instructions:

"G": go straight 1 unit.

"L": turn 90 degrees to the left (i.e., anti-clockwise direction).

"R": turn 90 degrees to the right (i.e., clockwise direction).

The robot performs the instructions given in order, and repeats them forever.

Return true if and only if there exists a circle in the plane such that the robot never leaves the circle.

```
class Solution {
public:
    bool isRobotBounded(string instructions) {
        vector<vector<int>> dir = {{0,1}, {-1, 0}, {0, -1}, {1,0}};
        int i = 0;
        int x = 0;
        int y = 0;

        for(int s = 0; s < instructions.size(); s++){
            if(instructions.at(s) == 'L'){
                i = (i + 1) % 4;
            }
            else if(instructions.at(s) == 'R'){
                i = (i + 3) % 4;
            }
            else{
                x = x + dir[i][0];
                y = y + dir[i][1];
            }
        }
        return x == 0 && y == 0 || i != 0;
    }
}
```

```
};
```

Climbing Stairs

There are n stairs, and a person standing at the bottom wants to climb stairs to reach the top. The person can climb either 1 stair or 2 stairs at a time, the task is to count the number of ways that a person can reach at the top.

```
int countWays(int n) {  
    vector<int> dp(n + 1, 0);  
  
    // Base cases  
    dp[0] = 1;  
    dp[1] = 1;  
  
    for (int i = 2; i <= n; i++)  
        dp[i] = dp[i - 1] + dp[i - 2];  
  
    return dp[n];  
}
```

```
int countWays(int n) {  
  
    // variable prev1, prev2 to store the  
    // values of last and second last states  
    int prev1 = 1;  
    int prev2 = 1;  
  
    for (int i = 2; i <= n; i++) {  
        int curr = prev1 + prev2;  
        prev2 = prev1;  
        prev1 = curr;  
    }  
  
    // In last iteration final value  
    // of curr is stored in prev.  
    return prev1;  
}
```

Coin Change

```
class Solution {
public:
    unordered_map<int, int> memo;
    int dfs(int amount, vector<int>& coins) {
        if (amount == 0) return 0;
        if (memo.find(amount) != memo.end())
            return memo[amount];

        int res = INT_MAX;
        for (int coin : coins) {
            if (amount - coin >= 0) {
                int result = dfs(amount - coin, coins);
                if (result != INT_MAX) {
                    res = min(res, 1 + result);
                }
            }
        }

        memo[amount] = res;
        return res;
    }

    int coinChange(vector<int>& coins, int amount) {
        int minCoins = dfs(amount, coins);
        return minCoins == INT_MAX ? -1 : minCoins;
    }
};
```

Coin Change || - Total number of solutions:

```

        return dfs(0, amount, coins, memo);
    }

    int dfs(int i, int a, vector<int>& coins, vector<vector<int>>& memo) {
        if (a == 0) return 1;
        if (i >= coins.size()) return 0;
        if (memo[i][a] != -1) return memo[i][a];

        int res = 0;
        if (a >= coins[i]) {
            res = dfs(i + 1, a, coins, memo);
            res += dfs(i, a - coins[i], coins, memo);
        }
        memo[i][a] = res;

        return res;
    }
};

```

Sliding Window Maximum

You are given an array of integers `nums` and an integer `k`. There is a sliding window of size `k` that starts at the left edge of the array. The window slides one position to the right until it reaches the right edge of the array.

Return a list that contains the maximum element in the window at each step.

```

class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        priority_queue<pair<int, int>> heap;
        vector<int> output;
        for (int i = 0; i < nums.size(); i++) {
            heap.push({nums[i], i});
            if (i >= k - 1) {
                while (heap.top().second <= i - k) {
                    heap.pop();
                }
                output.push_back(heap.top().first);
            }
        }
    }
};

```

```
        }
        return output;
    }
};
```

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        int n = nums.size();
        vector<int> output(n - k + 1);
        deque<int> q;
        int l = 0, r = 0;

        while (r < n) {
            while (!q.empty() && nums[q.back()] < nums[r]) {
                q.pop_back();
            }
            q.push_back(r);

            if (l > q.front()) {
                q.pop_front();
            }

            if ((r + 1) >= k) {
                output[l] = nums[q.front()];
                l++;
            }
            r++;
        }

        return output;
    }
};
```

Swim in Rising Water

You are given a square 2-D matrix of distinct integers grid where each integer $\text{grid}[i][j]$ represents the elevation at position (i, j) .

Rain starts to fall at time = 0, which causes the water level to rise. At time t, the water level across the entire grid is t.

You may swim either horizontally or vertically in the grid between two adjacent squares if the original elevation of both squares is less than or equal to the water level at time t.

Starting from the top left square (0, 0), return the minimum amount of time it will take until it is possible to reach the bottom right square (n - 1, n - 1).

Example 1:

0	1
2	3

Input: grid = [[0,1],[2,3]]

Output: 3

Explanation: For a path to exist to the bottom right square `grid[1][1]` the water elevation must be at least 3. At time `t = 3`, the water level is 3.

Example 2:

0	1	2	10
9	14	4	13
12	3	8	15
11	5	7	6

Input: grid = [
 [0,1,2,10],
 [9,14,4,13],
 [12,3,8,15],
 [11,5,7,6]]
]

Output: 8

Explanation: The water level must be at least 8 to reach the bottom right square. The path is [0, 1, 2, 4, 8, 7, 6].

Using Dijkstra's Algorithm:

```
class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int N = grid.size();
        set<pair<int, int>> visit;
        priority_queue<vector<int>, vector<vector<int>>, greater<>> minHeap;
        vector<vector<int>> directions = {
```

```

    {0, 1}, {0, -1}, {1, 0}, {-1, 0}
};

minHeap.push({grid[0][0], 0, 0});
visit.insert({0, 0});

while (!minHeap.empty()) {
    auto curr = minHeap.top();
    minHeap.pop();
    int t = curr[0], r = curr[1], c = curr[2];
    if (r == N - 1 && c == N - 1) {
        return t;
    }
    for (const auto& dir : directions) {
        int neiR = r + dir[0], neiC = c + dir[1];
        if (neiR < 0 || neiC < 0 || neiR == N ||
            neiC == N || visit.count({neiR, neiC})) {
            continue;
        }
        visit.insert({neiR, neiC});
        minHeap.push({
            max(t, grid[neiR][neiC]), neiR, neiC
        });
    }
}
return N * N;
};

```

Kruskal's Algo:

```

class DSU {
    vector<int> Parent, Size;
public:
    DSU(int n) : Parent(n + 1), Size(n + 1, 1) {
        for (int i = 0; i <= n; i++) Parent[i] = i;
    }

    int find(int node) {
        if (Parent[node] != node)
            Parent[node] = find(Parent[node]);
        return Parent[node];
    }
};

```

```

    }

bool unionSets(int u, int v) {
    int pu = find(u), pv = find(v);
    if (pu == pv) return false;
    if (Size[pu] < Size[pv]) swap(pu, pv);
    Size[pu] += Size[pv];
    Parent[pv] = pu;
    return true;
}

bool connected(int u, int v) {
    return find(u) == find(v);
}
};

class Solution {
public:
    int swimInWater(vector<vector<int>>& grid) {
        int N = grid.size();
        DSU dsu(N * N);
        vector<tuple<int, int, int>> positions;
        for (int r = 0; r < N; r++)
            for (int c = 0; c < N; c++)
                positions.emplace_back(grid[r][c], r, c);

        sort(positions.begin(), positions.end());
        vector<pair<int, int>> directions = {
            {0, 1}, {1, 0}, {0, -1}, {-1, 0}
        };

        for (auto& [t, r, c] : positions) {
            for (auto& [dr, dc] : directions) {
                int nr = r + dr, nc = c + dc;
                if (nr >= 0 && nr < N && nc >= 0 &&
                    nc < N && grid[nr][nc] <= t) {
                    dsu.unionSets(r * N + c, nr * N + nc);
                }
            }
            if (dsu.connected(0, N * N - 1)) return t;
        }
        return N * N;
    }
}

```

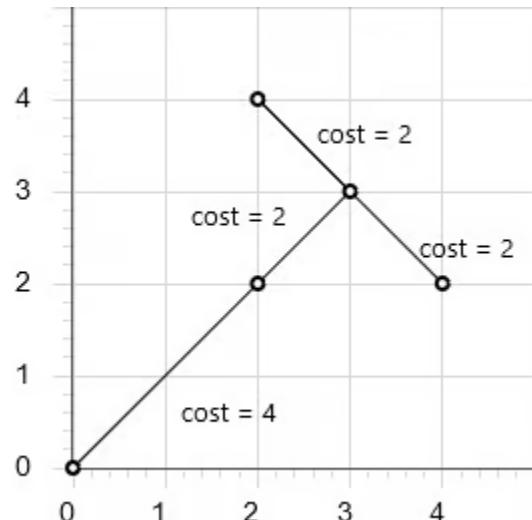
```
};
```

Min Cost to Connect Points

You are given a 2-D integer array points, where $\text{points}[i] = [x_i, y_i]$. Each $\text{points}[i]$ represents a distinct point on a 2-D plane.

The cost of connecting two points $[x_i, y_i]$ and $[x_j, y_j]$ is the manhattan distance between the two points, i.e. $|x_i - x_j| + |y_i - y_j|$.

Return the minimum cost to connect all points together, such that there exists exactly one path between each pair of points.



Input: $\text{points} = [[0,0],[2,2],[3,3],[2,4],[4,2]]$

Output: 10

Brute force: Kruskal's Algo:

- Time complexity: $O(n^2 \log n)$
- Space complexity: $O(n^2)$

```
class DSU {  
public:  
    vector<int> Parent, Size;
```

```

DSU(int n) : Parent(n + 1), Size(n + 1, 1) {
    for (int i = 0; i <= n; ++i) Parent[i] = i;
}

int find(int node) {
    if (Parent[node] != node) {
        Parent[node] = find(Parent[node]);
    }
    return Parent[node];
}

bool unionSets(int u, int v) {
    int pu = find(u), pv = find(v);
    if (pu == pv) return false;
    if (Size[pu] < Size[pv]) swap(pu, pv);
    Size[pu] += Size[pv];
    Parent[pv] = pu;
    return true;
}
};

class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {
        int n = points.size();
        DSU dsu(n);
        vector<array<int, 3>> edges;

        for (int i = 0; i < n; ++i) {
            for (int j = i + 1; j < n; ++j) {
                int dist = abs(points[i][0] - points[j][0]) +
                           abs(points[i][1] - points[j][1]);
                edges.push_back({dist, i, j});
            }
        }

        sort(edges.begin(), edges.end());
        int res = 0;

        for (auto& [dist, u, v] : edges) {
            if (dsu.unionSets(u, v)) {
                res += dist;
            }
        }
    }
};

```

```
        }
    }
    return res;
}
};
```

Prim's Algorithm (Optimal)

```
class Solution {
public:
    int minCostConnectPoints(vector<vector<int>>& points) {
        int n = points.size(), node = 0;
        vector<int> dist(n, 100000000);
        vector<bool> visit(n, false);
        int edges = 0, res = 0;

        while (edges < n - 1) {
            visit[node] = true;
            int nextNode = -1;
            for (int i = 0; i < n; i++) {
                if (visit[i]) continue;
                int curDist = abs(points[i][0] - points[node][0]) +
                    abs(points[i][1] - points[node][1]);
                dist[i] = min(dist[i], curDist);
                if (nextNode == -1 || dist[i] < dist[nextNode]) {
                    nextNode = i;
                }
            }
            res += dist[nextNode];
            node = nextNode;
            edges++;
        }
        return res;
    }
};
```

Network Delay Time

You are given a network of n directed nodes, labeled from 1 to n . You are also given times, a list of directed edges where $\text{times}[i] = (u_i, v_i, t_i)$.

u_i is the source node (an integer from 1 to n)

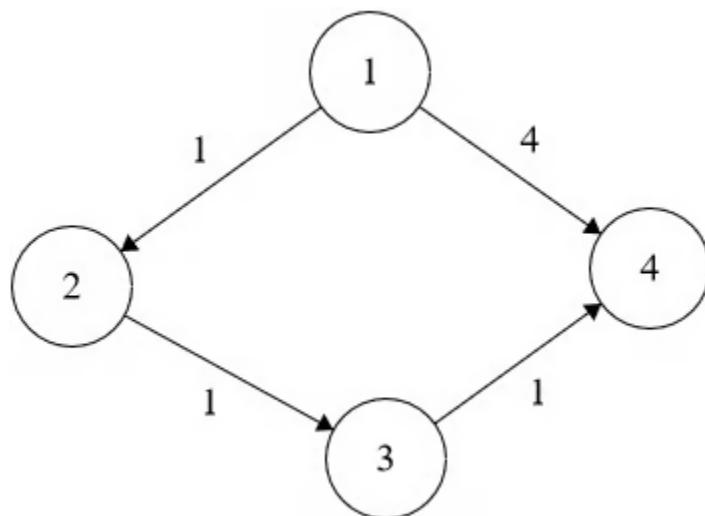
v_i is the target node (an integer from 1 to n)

t_i is the time it takes for a signal to travel from the source to the target node (an integer greater than or equal to 0).

You are also given an integer k , representing the node that we will send a signal from.

Return the minimum time it takes for all of the n nodes to receive the signal. If it is impossible for all the nodes to receive the signal, return -1 instead.

Example 1:



Input: $\text{times} = [[1,2,1],[2,3,1],[1,4,4],[3,4,1]]$, $n = 4$, $k = 1$

Output: 3

```
class Solution {
public:
    int networkDelayTime(vector<vector<int>>& times, int n, int k) {
        vector<bool> vis(n+1, false);
        unordered_map<int, vector<pair<int,int>>> adj;
        for(auto it: times){
            adj[it[0]].push_back({it[1],it[2]});
        }
    }
};
```

```

        priority_queue<vector<int>, vector<vector<int>>,
greater<vector<int>>> pq;
    vector<int> dis(n+1, INT_MAX);
pq.push({0, k});
dis[k]=0;
dis[0]=0;
while(!pq.empty()){
    auto it = pq.top();
    pq.pop();
    int node = it[1];
    int time = it[0];
    cout<<"node: "<<node<<" time: "<<time<<endl;
    for(auto n_node: adj[node]){
        int edg_time = n_node.second;
        int edg_node = n_node.first;
        if(time + edg_time < dis[edg_node]){
            dis[edg_node]=time+edg_time;
            pq.push({dis[edg_node], edg_node});
        }
    }
}
for(auto it: dis){
    cout<<it<<" ";
}
int ans = *max_element(dis.begin(), dis.end());
return ans == INT_MAX ? -1 : ans;
}

};

```

Time Based Key-Value Store

Solved

Implement a time-based key-value data structure that supports:

Storing multiple values for the same key at specified time stamps

Retrieving the key's value at a specified timestamp

Implement the TimeMap class:

TimeMap() Initializes the object.

void set(String key, String value, int timestamp) Stores the key key with the value value at the given time timestamp.

String get(String key, int timestamp) Returns the most recent value of key if set was previously called on it and the most recent timestamp for that key prev_timestamp is less than or equal to the given timestamp (prev_timestamp <= timestamp). If there are no values, it returns "".

Note: For all calls to set, the timestamps are in strictly increasing order.

Example 1:

Input:

```
["TimeMap", "set", ["alice", "happy", 1], "get", ["alice", 1], "get", ["alice", 2], "set", ["alice", "sad", 3], "get", ["alice", 3]]
```

Output:

```
[null, null, "happy", "happy", null, "sad"]
```

Explanation:

```
TimeMap timeMap = new TimeMap();
timeMap.set("alice", "happy", 1); // store the key "alice" and value "happy" along with timestamp = 1.
timeMap.get("alice", 1); // return "happy"
timeMap.get("alice", 2); // return "happy", there is no value stored for timestamp 2, thus we return the value at timestamp 1.
timeMap.set("alice", "sad", 3); // store the key "alice" and value "sad" along with timestamp = 3.
timeMap.get("alice", 3); // return "sad"
```

```
class TimeMap {
public:
    unordered_map<string, vector<pair<int, string>>> keystore;
    TimeMap() {

    }
}
```

```

void set(string key, string value, int timestamp) {
    keystore[key].push_back({timestamp, value});
}

string get(string key, int timestamp) {
    auto value = keystore[key];
    int left = 0, right = value.size()-1;
    string ans = "";
    while(left<=right){
        int mid = (left+right)/2;
        if(value[mid].first <= timestamp){
            ans = value[mid].second;
            left = mid +1;
        }else{
            right = mid -1;
        }
    }
    return ans;
}
};

```

Task Scheduler

You are given an array of CPU tasks, each labeled with a letter from A to Z, and a number n. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of at least n intervals between two tasks with the same label.

Return the minimum number of CPU intervals required to complete all tasks.

Example 1:

Input: tasks = ["A","A","A","B","B","B"], n = 2

Output: 8

Explanation: A possible sequence is: A -> B -> idle -> A -> B -> idle -> A -> B.

After completing task A, you must wait two intervals before doing A again. The same applies to task B. In the 3rd interval, neither A nor B can be done, so you idle. By the 4th interval, you can do A again as 2 intervals have passed.

Example 2:

Input: tasks = ["A","C","A","B","D","B"], n = 1

Output: 6

Explanation: A possible sequence is: A -> B -> C -> D -> A -> B.

With a cooling interval of 1, you can repeat a task after just one other task.

```
class Solution {
public:
    int leastInterval(vector<char>& tasks, int n) {
        unordered_map<int,int> mp;
        for(auto task: tasks){
            mp[task]++;
        }
        int max_F = 0;
        for(auto task: mp){

            max_F = max(max_F, task.second);
        }
        int ele_w_mf = 0;
        for(auto task: mp){
            if(task.second == max_F) ele_w_mf++;
        }

        int ans = (((max_F-1) * n ) + max_F + ele_w_mf - 1);
    }
}
```

```

        int ts = tasks.size();
        ans = max(ans, ts);
        return ans;
    }
};

```

Rotting Fruit

You are given a 2-D matrix grid. Each cell can have one of three possible values:

0 representing an empty cell

1 representing a fresh fruit

2 representing a rotten fruit

Every minute, if a fresh fruit is horizontally or vertically adjacent to a rotten fruit, then the fresh fruit also becomes rotten.

Return the minimum number of minutes that must elapse until there are zero fresh fruits remaining. If this state is impossible within the grid, return -1.

Example 1:

Minute 0	Minute 1	Minute 2	Minute 3	Minute 4
🍌 🍌 🍃	🍌 🍌 🍃	🍌 🍌 🍃	🍌 🍄 🍄	🔥 🔥 🍃
🍌 🍌 🍃	🍌 🍃 🔥	🔥 🔥 🔥	🔥 🔥 🔥	🔥 🔥 🔥
🍌 🔥	🔥 🔥	🔥 🔥 🔥	🔥 🔥 🔥	🔥 🔥 🔥

Input: grid = [[1,1,0],[0,1,1],[0,1,2]]

Output: 4

Example 2:

Input: grid = [[1,0,1],[0,2,0],[1,0,1]]

Output: -1

```
class Solution {
```

```

public:
    int orangesRotting(vector<vector<int>>& grid) {
        queue<pair<int, int>> q;
        int fresh = 0;
        int time = 0;

        for (int r = 0; r < grid.size(); r++) {
            for (int c = 0; c < grid[0].size(); c++) {
                if (grid[r][c] == 1) {
                    fresh++;
                }
                if (grid[r][c] == 2) {
                    q.push({r, c});
                }
            }
        }

        vector<pair<int, int>> directions = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}};
        while (fresh > 0 && !q.empty()) {
            int length = q.size();
            for (int i = 0; i < length; i++) {
                auto curr = q.front();
                q.pop();
                int r = curr.first;
                int c = curr.second;

                for (const auto& dir : directions) {
                    int row = r + dir.first;
                    int col = c + dir.second;
                    if (row >= 0 && row < grid.size() &&
                        col >= 0 && col < grid[0].size() &&
                        grid[row][col] == 1) {
                        grid[row][col] = 2;
                        q.push({row, col});
                        fresh--;
                    }
                }
            }
            time++;
        }
        return fresh == 0 ? time : -1;
    }
}

```

```
};
```

Islands and Treasure

You are given a $m \times n$ $m \times n$ 2D grid initialized with these three possible values:

-1 - A water cell that can not be traversed.

0 - A treasure chest.

INF - A land cell that can be traversed. We use the integer $2^{31} - 1 = 2147483647$ to represent INF.

Fill each land cell with the distance to its nearest treasure chest. If a land cell cannot reach a treasure chest then the value should remain INF.

Assume the grid can only be traversed up, down, left, or right.

Modify the grid in-place.

Example 1:

Input: [
[2147483647,-1,0,2147483647],
[2147483647,2147483647,2147483647,-1],
[2147483647,-1,2147483647,-1],
[0,-1,2147483647,2147483647]
]

Output: [
[3,-1,0,1],
[2,2,1,-1],
[1,-1,2,-1],
[0,-1,3,4]
]

```
class Solution {  
public:  
    void islandsAndTreasure(vector<vector<int>>& grid) {  
        int n = grid.size(), m = grid[0].size();
```

```

vector<vector<int>> vis(n, vector<int>(m, 0));
int inf = 2147483647;
queue<pair<int,int>> q;
vector<vector<int>> dir = {{1,0},{0,1},{-1,0},{0,-1}};

for(int i=0;i<n;i++){
    for(int j=0;j<m;j++){
        if(grid[i][j]==0){
            q.push({i,j});
        }
    }
}
int dist = 1;
while(!q.empty()){
    int qs = q.size();
    for(int i=0;i<qs;i++){
        auto curr = q.front();
        int x = curr.first;
        int y = curr.second;

        q.pop();
        vis[x][y]=1;
        for(int j=0;j<4;j++){
            int nx = x + dir[j][0];
            int ny = y + dir[j][1];
            if(nx >=0 && nx < n && ny >=0 && ny < m &&
vis[nx][ny]==0 && grid[nx][ny]!=-1 && grid[nx][ny] == inf){
                q.push({nx,ny});
                grid[nx][ny] = dist;
            }
        }
    }
    dist+=1;
}

```

```
    }
};
```

Pacific Atlantic Water Flow

You are given a rectangular island heights where $\text{heights}[r][c]$ represents the height above sea level of the cell at coordinate (r, c) .

The islands borders the Pacific Ocean from the top and left sides, and borders the Atlantic Ocean from the bottom and right sides.

Water can flow in four directions (up, down, left, or right) from a cell to a neighboring cell with height equal or lower. Water can also flow into the ocean from cells adjacent to the ocean.

Find all cells where water can flow from that cell to both the Pacific and Atlantic oceans. Return it as a 2D list where each element is a list $[r, c]$ representing the row and column of the cell. You may return the answer in any order.

```
class Solution {
    vector<pair<int, int>> directions = {{1, 0}, {-1, 0},
                                              {0, 1}, {0, -1}};
public:
    vector<vector<int>> pacificAtlantic(vector<vector<int>>& heights) {
        int ROWS = heights.size(), COLS = heights[0].size();
        vector<vector<bool>> pac(ROWS, vector<bool>(COLS, false));
        vector<vector<bool>> atl(ROWS, vector<bool>(COLS, false));

        for (int c = 0; c < COLS; ++c) {
            dfs(0, c, pac, heights);
            dfs(ROWS - 1, c, atl, heights);
        }
        for (int r = 0; r < ROWS; ++r) {
            dfs(r, 0, pac, heights);
            dfs(r, COLS - 1, atl, heights);
        }

        vector<vector<int>> res;
```

```

        for (int r = 0; r < ROWS; ++r) {
            for (int c = 0; c < COLS; ++c) {
                if (pac[r][c] && atl[r][c]) {
                    res.push_back({r, c});
                }
            }
        }
        return res;
    }

private:
    void dfs(int r, int c, vector<vector<bool>>& ocean,
    vector<vector<int>>& heights) {
        ocean[r][c] = true;
        for (auto [dr, dc] : directions) {
            int nr = r + dr, nc = c + dc;
            if (nr >= 0 && nr < heights.size() &&
                nc >= 0 && nc < heights[0].size() &&
                !ocean[nr][nc] && heights[nr][nc] >= heights[r][c]) {
                dfs(nr, nc, ocean, heights);
            }
        }
    }
};

```

Counting Bits

Given an integer n, count the number of 1's in the binary representation of every number in the range [0, n].

Return an array output where output[i] is the number of 1's in the binary representation of i.

Example 1:

Input: n = 4

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

Explanation:

0 -> 0

```
1 -> 1
2 -> 10
3 -> 11
4 -> 100
```

Constraints:

$0 \leq n \leq 1000$

```
class Solution {
public:
    vector<int> countBits(int n) {
        vector<int> dp(n + 1);
        for (int i = 1; i <= n; i++) {
            dp[i] = dp[i >> 1] + (i & 1);
        }
        return dp;
    }
};
```

Reverse Bits

Given a 32-bit unsigned integer n , reverse the bits of the binary representation of n and return the result.

Example 1:

Input: $n = 00000000000000000000000000001010$

Output: 2818572288 (10101000000000000000000000000000)

```
class Solution {
public:
    uint32_t reverseBits(uint32_t n) {
        uint32_t res = 0;
        for(int i=0;i<32;i++){
            res = (res << 1) | (n & 1);
            n = n>>1;
        }
        return res;
    }
};
```

```
};
```

Min Cost Climbing Stairs

You are given an array of integers cost where cost[i] is the cost of taking a step from the ith floor of a staircase. After paying the cost, you can step to either the (i + 1)th floor or the (i + 2)th floor.

You may choose to start at the index 0 or the index 1 floor.

Return the minimum cost to reach the top of the staircase, i.e. just past the last index in cost.

```
class Solution {  
public:  
    int minCostClimbingStairs(vector<int>& cost) {  
        int n = cost.size();  
        vector<int> dp(n, 0);  
  
        dp[0]=cost[0];  
        dp[1]=cost[1];  
        for(int i=2;i<n;i++){  
            dp[i]= min(dp[i-1], dp[i-2]) + cost[i];  
        }  
  
        return min(dp[n-2], dp[n-1]);  
    }  
};
```

House Robber:

```
class Solution:  
    def rob(self, nums: List[int]) -> int:  
  
        n = len(nums)
```

```

# base cases
if n == 2:
    return max(nums[0], nums[1])
if n == 1:
    return nums[0]

dp = [0]*n
dp[0] = nums[0]
dp[1] = max(nums[0], nums[1])
for i in range(2, n):
    dp[i] = max(dp[i-1], nums[i]+dp[i-2])

return dp[n-1]

```

House Robber II

House in circles

```

class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() == 1) return nums[0];
        return max(getMax(nums, 0, nums.size() - 2), getMax(nums, 1,
nums.size() - 1));
    }

private:
    int getMax(vector<int>& nums, int start, int end) {
        int prevRob = 0, maxRob = 0;

        for (int i = start; i <= end; ++i) {
            int temp = max(maxRob, prevRob + nums[i]);
            prevRob = maxRob;
            maxRob = temp;
        }

        return maxRob;
    }
}

```

```
};
```

Meeting Rooms II

Given an array of meeting time interval objects consisting of start and end times $[[\text{start}_1, \text{end}_1], [\text{start}_2, \text{end}_2], \dots]$ ($\text{start}_i < \text{end}_i$), find the minimum number of days required to schedule all meetings without any conflicts.

Note: $(0,8),(8,10)$ is not considered a conflict at 8.

Example 1:

Input: intervals = [(0,40),(5,10),(15,20)]

Output: 2

```
/**  
 * Definition of Interval:  
 * class Interval {  
 * public:  
 *     int start, end;  
 *     Interval(int start, int end) {  
 *         this->start = start;  
 *         this->end = end;  
 *     }  
 * }  
 */  
  
class Solution {  
public:  
    int minMeetingRooms(vector<Interval>& intervals) {  
        vector<pair<int, int>> time;  
        for (const auto& i : intervals) {  
            time.push_back({i.start, 1});  
            time.push_back({i.end, -1});  
        }  
  
        sort(time.begin(), time.end(), [](auto& a, auto& b) {  
            return a.first == b.first ? a.second < b.second : a.first <
```

```

        b.first;
    });

    int res = 0, count = 0;
    for (const auto& t : time) {
        count += t.second;
        res = max(res, count);
    }
    return res;
}
};


```

Decode Ways

A string consisting of uppercase english characters can be encoded to a number using the following mapping:

'A' -> "1"

'B' -> "2"

...

'Z' -> "26"

To decode a message, digits must be grouped and then mapped back into letters using the reverse of the mapping above. There may be multiple ways to decode a message. For example, "1012" can be mapped into:

"JAB" with the grouping (10 1 2)

"JL" with the grouping (10 12)

The grouping (1 01 2) is invalid because 01 cannot be mapped into a letter since it contains a leading zero.

Given a string s containing only digits, return the number of ways to decode it. You can assume that the answer fits in a 32-bit integer.

Example 1:

Input: s = "12"

Output: 2

Explanation: "12" could be decoded as "AB" (1 2) or "L" (12).

Example 2:

Input: s = "01"

Output: 0

```
class Solution:
    def numDecodings(self, s: str) -> int:
        self.dp = [-1] * (len(s)+1)
        return self.dfs(0, s)

    def dfs(self, i, s):
        if i >= len(s):
            return 1

        if s[i] == "0":
            return 0

        if self.dp[i] != -1:
            return self.dp[i]

        res = 0

        if i < len(s)-1 and (s[i] == '1' or (s[i]=='2' and s[i+1]<'7')):
            res = self.dfs(i+2, s)

        res += self.dfs(i+1, s)
        self.dp[i] = res
        return res
```

Maximum Product Subarray:

```
class Solution:
```

```

def maxProduct(self, nums: List[int]) -> int:
    res = nums[0]
    curMin, curMax = 1, 1

    for num in nums:
        tmp = curMax * num
        curMax = max(num * curMax, num * curMin, num)
        curMin = min(tmp, num * curMin, num)
        res = max(res, curMax)
    return res

```

```

class Solution:
    def maxProduct(self, nums: List[int]) -> int:
        n, res = len(nums), nums[0]
        prefix = suffix = 0

        for i in range(n):
            prefix = nums[i] * (prefix or 1)
            suffix = nums[n - 1 - i] * (suffix or 1)
            res = max(res, max(prefix, suffix))
        return res

```

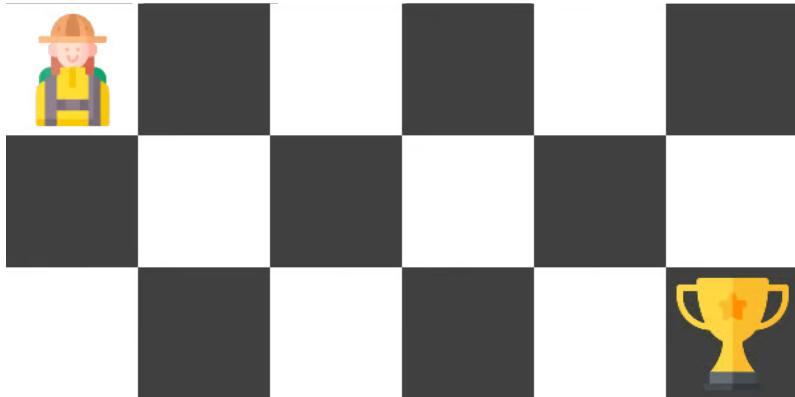
Unique Paths:

There is an $m \times n$ grid where you are allowed to move either down or to the right at any point in time.

Given the two integers m and n , return the number of possible unique paths that can be taken from the top-left corner of the grid ($\text{grid}[0][0]$) to the bottom-right corner ($\text{grid}[m - 1][n - 1]$).

You may assume the output will fit in a 32-bit integer.

Example 1:



Using maths:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        if m == 1 or n == 1:
            return 1
        if m < n:
            m, n = n, m

        res = j = 1
        for i in range(m, m + n - 1):
            res *= i
            res //= j
            j += 1

        return res
```

Top Down Memo:

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        memo = [[-1] * n for _ in range(m)]
        def dfs(i, j):
            if i == (m - 1) and j == (n - 1):
                return 1
            if i >= m or j >= n:
                return 0
            if memo[i][j] != -1:
                return memo[i][j]

            memo[i][j] = dfs(i, j + 1) + dfs(i + 1, j)

        return memo[0][0]
```

```
        return memo[i][j]

    return dfs(0, 0)
```

Bottom Up Tab

```
class Solution:
    def uniquePaths(self, m: int, n: int) -> int:
        dp = [[0] * (n + 1) for _ in range(m + 1)]
        dp[m - 1][n - 1] = 1

        for i in range(m - 1, -1, -1):
            for j in range(n - 1, -1, -1):
                dp[i][j] += dp[i + 1][j] + dp[i][j + 1]

        return dp[0][0]
```

Best Time to Buy and Sell Stock with Cooldown

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

You may buy and sell one NeetCoin multiple times with the following restrictions:

After you sell your NeetCoin, you cannot buy another one on the next day (i.e., there is a cooldown period of one day).

You may only own at most one NeetCoin at a time.

You may complete as many transactions as you like.

Return the maximum profit you can achieve.

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

Output: 6

```
class Solution:
    def maxProfit(self, prices: List[int]) -> int:
        dp = {} # key=(i, buying) val=max_profit

        def dfs(i, buying):
```

```

        if i >= len(prices):
            return 0
        if (i, buying) in dp:
            return dp[(i, buying)]

        cooldown = dfs(i + 1, buying)
        if buying:
            buy = dfs(i + 1, not buying) - prices[i]
            dp[(i, buying)] = max(buy, cooldown)
        else:
            sell = dfs(i + 1, not buying) + prices[i]
            dp[(i, buying)] = max(sell, cooldown)
        return dp[(i, buying)]

    return dfs(0, True)

```

Target Sum

You are given an array of integers `nums` and an integer `target`.

For each number in the array, you can choose to either add or subtract it to a total sum.

For example, if `nums` = [1, 2], one possible sum would be "+1-2=-1".

If `nums`=[1,1], there are two different ways to sum the input numbers to get a sum of 0: "+1-1" and "-1+1".

Return the number of different ways that you can build the expression such that the total sum equals target.

Example 1:

Input: `nums` = [2,2,2], `target` = 2

Output: 3

Explanation: There are 3 different ways to sum the input numbers to get a sum of 2.

+2 +2 -2 = 2
+2 -2 +2 = 2
-2 +2 +2 = 2

```
class Solution:
```

```

def findTargetSumWays(self, nums: List[int], target: int) -> int:

    dp = {}
    def dfs(i, total):
        if i == len(nums):
            return target == total
        if (i, total) in dp:
            return dp[(i, total)]

        dp[(i, total)] = dfs(i+1, total-nums[i]) + dfs(i+1,
total+nums[i])
        return dp[(i, total)]

    return dfs(0, 0)

```

Interleaving String

You are given three strings s_1 , s_2 , and s_3 . Return true if s_3 is formed by interleaving s_1 and s_2 together or false otherwise.

Interleaving two strings s and t is done by dividing s and t into n and m substrings respectively, where the following conditions are met

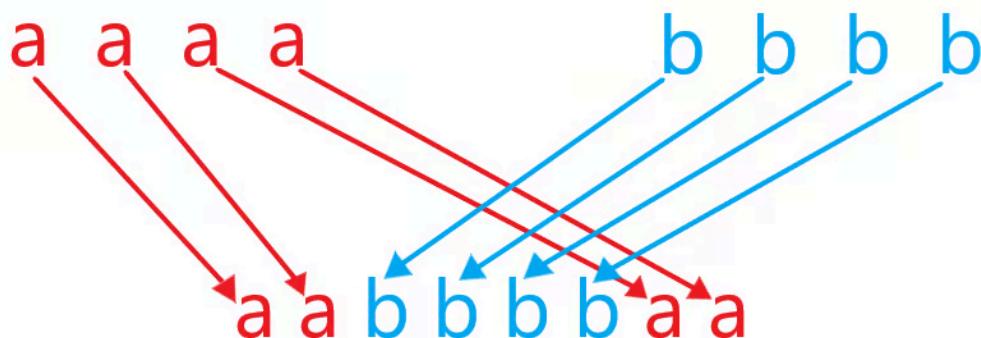
$|n - m| \leq 1$, i.e. the difference between the number of substrings of s and t is at most 1.

$s = s_1 + s_2 + \dots + s_n$

$t = t_1 + t_2 + \dots + t_m$

Interleaving s and t is $s_1 + t_1 + s_2 + t_2 + \dots$ or $t_1 + s_1 + t_2 + s_2 + \dots$

You may assume that s_1 , s_2 and s_3 consist of lowercase English letters.



Input: s1 = "aaaa", s2 = "bbbb", s3 = "aabbbbbaa"

Output: true

Input: s1 = "", s2 = "", s3 = ""

Output: true

Input: s1 = "abc", s2 = "xyz", s3 = "abxzcy"

Output: false

```
class Solution:
    def isInterleave(self, s1: str, s2: str, s3: str) -> bool:
        if len(s1) + len(s2) != len(s3):
            return False

        dp = []
        def dfs(i, j, k):
            if k == len(s3):
                return (i == len(s1)) and (j == len(s2))
            if (i, j) in dp:
                return dp[(i, j)]

            res = False
            if i < len(s1) and s1[i] == s3[k]:
                res = dfs(i + 1, j, k + 1)
            if not res and j < len(s2) and s2[j] == s3[k]:
                res = dfs(i, j + 1, k + 1)

            dp[(i, j)] = res
            return res

        return dfs(0, 0, 0)
```

Longest Increasing Path in Matrix

You are given a 2-D grid of integers matrix, where each integer is greater than or equal to 0.

Return the length of the longest strictly increasing path within matrix.

From each cell within the path, you can move either horizontally or vertically. You may not move diagonally.

5	5	3
2	3	6
1	1	1

Input: matrix = [[5,5,3],[2,3,6],[1,1,1]]

Output: 4

```
class Solution {
public:
    vector<vector<int>> directions = {{-1, 0}, {1, 0},
                                         {0, -1}, {0, 1}};
    vector<vector<int>> dp;

    int dfs(vector<vector<int>>& matrix, int r, int c, int prevVal) {
        int ROWS = matrix.size(), COLS = matrix[0].size();
        if (r < 0 || r >= ROWS || c < 0 ||
            c >= COLS || matrix[r][c] <= prevVal) {
            return 0;
        }
        if (dp[r][c] != -1) return dp[r][c];

        int res = 1;
        for (vector<int> d : directions) {
            res = max(res, 1 + dfs(matrix, r + d[0],
                                   c + d[1], matrix[r][c]));
        }
        dp[r][c] = res;
        return res;
    }

    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int ROWS = matrix.size(), COLS = matrix[0].size();
        dp = vector<vector<int>>(ROWS, vector<int>(COLS, -1));
    }
}
```

```

int LIP = 0;

for (int r = 0; r < ROWS; r++) {
    for (int c = 0; c < COLS; c++) {
        LIP = max(LIP, dfs(matrix, r, c, INT_MIN));
    }
}
return LIP;
};

}

```

Topological Sort (Kahn's Algorithm)

```

class Solution {
public:
    int longestIncreasingPath(vector<vector<int>>& matrix) {
        int ROWS = matrix.size(), COLS = matrix[0].size();
        vector<vector<int>> indegree(ROWS, vector<int>(COLS, 0));
        vector<vector<int>> directions = {{-1, 0}, {1, 0},
                                            {0, -1}, {0, 1}};

        for (int r = 0; r < ROWS; ++r) {
            for (int c = 0; c < COLS; ++c) {
                for (auto& d : directions) {
                    int nr = r + d[0], nc = c + d[1];
                    if (nr >= 0 && nr < ROWS && nc >= 0 &&
                        nc < COLS && matrix[nr][nc] < matrix[r][c]) {
                        indegree[r][c]++;
                    }
                }
            }
        }

        queue<pair<int, int>> q;
        for (int r = 0; r < ROWS; ++r) {
            for (int c = 0; c < COLS; ++c) {
                if (indegree[r][c] == 0) {
                    q.push({r, c});
                }
            }
        }
    }
}

```

```

int LIS = 0;
while (!q.empty()) {
    int size = q.size();
    for (int i = 0; i < size; ++i) {
        auto [r, c] = q.front();
        q.pop();
        for (auto& d : directions) {
            int nr = r + d[0], nc = c + d[1];
            if (nr >= 0 && nr < ROWS && nc >= 0 &&
                nc < COLS && matrix[nr][nc] > matrix[r][c]) {
                if (--indegree[nr][nc] == 0) {
                    q.push({nr, nc});
                }
            }
        }
    }
    LIS++;
}
return LIS;
};

```

Distinct Subsequences

You are given two strings s and t, both consisting of english letters.

Return the number of distinct subsequences of s which are equal to t.

Example 1:

Input: s = "caaat", t = "cat"

Output: 3

Explanation: There are 3 ways you can generate "cat" from s.

(c)aa(at)
(c)a(a)a(t)
(ca)aa(t)

TDM:

```

class Solution:
    def numDistinct(self, s: str, t: str) -> int:
        if len(t) > len(s):
            return 0

        dp = {}
        def dfs(i, j):
            if j == len(t):
                return 1
            if i == len(s):
                return 0
            if (i, j) in dp:
                return dp[(i, j)]

            res = dfs(i + 1, j)
            if s[i] == t[j]:
                res += dfs(i + 1, j + 1)
            dp[(i, j)] = res
            return res

        return dfs(0, 0)

```

BUT:

Burst Balloons

You are given an array of integers nums of size n. The ith element represents a balloon with an integer value of nums[i]. You must burst all of the balloons.

If you burst the ith balloon, you will receive nums[i - 1] * nums[i] * nums[i + 1] coins. If i - 1 or i + 1 goes out of bounds of the array, then assume the out of bounds value is 1.

Return the maximum number of coins you can receive by bursting all of the

Burst Balloons

You are given an array of integers nums of size n. The ith element represents a balloon with an

integer value of $\text{nums}[i]$. You must burst all of the balloons.

If you burst the i th balloon, you will receive $\text{nums}[i - 1] * \text{nums}[i] * \text{nums}[i + 1]$ coins. If $i - 1$ or $i + 1$ goes out of bounds of the array, then assume the out of bounds value is 1.

Return the maximum number of coins you can receive by bursting all of the balloons.

Example 1:

Input: $\text{nums} = [4,2,3,7]$

Output: 143

Explanation:

$\text{nums} = [4,2,3,7] \rightarrow [4,3,7] \rightarrow [4,7] \rightarrow [7] \rightarrow []$

$\text{coins} = 4*2*3 + 4*3*7 + 1*4*7 + 1*7*1 = 143$

```
class Solution:
    def maxCoins(self, nums: List[int]) -> int:

        arr = [1] + nums + [1]
        dp = {}

        def dfs(i, j):
            if i > j:
                return 0
            if (i,j) in dp:
                return dp[(i,j)]
            mx = 0
            for k in range(i, j+1):
                coins = arr[i-1] * arr[k] * arr[j+1]
                remcoin = dfs(i, k-1) + dfs(k+1, j)
                mx = max(mx, coins + remcoin)

            dp[(i,j)] = mx
            return mx

    return dfs(1,len(nums))
```

Jump Game

Find whether can one reach to end (n-1)

```
class Solution:
    def canJump(self, nums: List[int]) -> bool:
        goal = len(nums)-1

        for i in range(len(nums)-2, -1, -1):
            if i + nums[i] >= goal:
                goal = i

        return goal == 0
```

Gas Station

There are n gas stations along a circular route. You are given two integer arrays gas and cost where:

gas[i] is the amount of gas at the ith station.

cost[i] is the amount of gas needed to travel from the ith station to the (i + 1)th station. (The last station is connected to the first station)

You have a car that can store an unlimited amount of gas, but you begin the journey with an empty tank at one of the gas stations.

Return the starting gas station's index such that you can travel around the circuit once in the clockwise direction. If it's impossible, then return -1.

It's guaranteed that at most one solution exists.

Example 1:

Input: gas = [1,2,3,4], cost = [2,2,4,1]

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 0. Your tank = $4 - 1 + 1 = 4$

Travel to station 1. Your tank = $4 - 2 + 2 = 4$

Travel to station 2. Your tank = $4 - 2 + 3 = 5$

Travel to station 3. Your tank = $5 - 4 + 4 = 5$

```
class Solution:
    def canCompleteCircuit(self, gas: List[int], cost: List[int]) -> int:
        res = 0
        t_gas = 0
        c_gas = 0

        for i in range(len(gas)):
            costs = gas[i] - cost[i]
            t_gas += costs
            c_gas += costs

            if c_gas < 0:
                c_gas = 0
                res = i + 1

        return -1 if t_gas < 0 else res
```

Hand of Straights

Alice has some number of cards and she wants to rearrange the cards into groups so that each group is of size `groupSize`, and consists of `groupSize` consecutive cards.

Given an integer array `hand` where `hand[i]` is the value written on the `i`th card and an integer `groupSize`, return true if she can rearrange the cards, or false otherwise.

Example 1:

Input: `hand = [1,2,3,6,2,3,4,7,8]`, `groupSize = 3`

Output: true

Explanation: Alice's hand can be rearranged as [1,2,3],[2,3,4],[6,7,8]

ordered Set:

```
from collections import defaultdict

class Solution:
    def isNStraightHand(self, hand, gs):
        n = len(hand)
        if n % gs != 0:
            return False

        mp = defaultdict(int)
        for g in hand:
            mp[g] += 1

        k = gs
        t = n // gs

        while t > 0:
            val = min(mp.keys()) # equivalent to mp.begin()->first

            for i in range(k):
                if mp[val] <= 0:
                    return False
                mp[val] -= 1
                if mp[val] == 0:
                    del mp[val]
            val += 1

            t -= 1

        return True
```

using unordered map

```
class Solution {
public:
    bool isNStraightHand(vector<int>& hand, int groupSize) {
        if (hand.size() % groupSize != 0) return false;

        unordered_map<int, int> count;
        for (int num : hand) count[num]++;
    }
}
```

```

        for (int num : hand) {
            int start = num;
            while (count[start - 1] > 0) start--;
            while (start <= num) {
                while (count[start] > 0) {
                    for (int i = start; i < start + groupSize; i++) {
                        if (count[i] == 0) return false;
                        count[i]--;
                    }
                }
                start++;
            }
        }
        return true;
    }
};

```

Google Interview Questions:

3 Sum - Find All Triplets with Zero Sum

```

vector<vector<int>> triplet(int n, vector<int> &arr) {
    vector<vector<int>> ans;
    sort(arr.begin(), arr.end());
    for (int i = 0; i < n; i++) {
        //remove duplicates:
        if (i != 0 && arr[i] == arr[i - 1]) continue;

        //moving 2 pointers:
        int j = i + 1;
        int k = n - 1;
        while (j < k) {
            int sum = arr[i] + arr[j] + arr[k];
            if (sum < 0) {
                j++;
            }
            else if (sum > 0) {
                k--;
            }
            else {
                ans.push_back({arr[i], arr[j], arr[k]});
                j++;
                k--;
            }
        }
    }
    return ans;
}

```

```
        }
    else {
        vector<int> temp = {arr[i], arr[j], arr[k]};
        ans.push_back(temp);
        j++;
        k--;
        //skip the duplicates:
        while (j < k && arr[j] == arr[j - 1]) j++;
        while (j < k && arr[k] == arr[k + 1]) k--;
    }
}
return ans;
}
```