Blind 75:
https://leetcode.com/discuss/general-discussion/460599/blind-75-leetcode-questions

– Viet Dang

1. Leet Code Problem: Add two Integer (if they fit)

C/C++

```
#include <iostream>
#include <vector>
#include <unordered_map>

using namespace std;

class Solution {
public:
        vector<int> twoSum(vector<int>& nums, int target) {
        unordered_map<int, int> seen;

                for (int i = 0; i < nums.size(); i++) {
                        int complement = target - nums[i];

                                if (seen.count(complement)){

                                }

                        seen[nums[i]] = i;
                }
        return {};
        }
};
```

2. Leet Code Problem: Best TIme to Buy and Sell Stock

C/C++

```
#include <iostream>
#include <vector>

class Solution {
public:
        int maxProfit(vector<int>& prices) {
        int maxProfit = 0;
        int minPrice = prices[0];

        for(int i = 1; i < prices.size(); i++){
                minPrice = min(minPrice, prices[i]);
                maxProfit = max(maxProfit, prices[i] - minPrice);
        }

        return maxProfit;
        }
};
```

3. Leet Code Problem: Contains Duplicate

C/C++

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

class Solution {
        public: bool containsDuplicate(vector<int>& nums) {
                unordered_map<int, int> count;
                for(int num : nums) {
                        count[num]++;
                        if(count[num] > 1)
                        { return true; }
                }
        return false;
        }
};
```

4. Leet Code Problem: Product of Array Except Self

C/C++

```
class Solution {

public:
        vector<int> productExceptSelf(vector <int>&nums) {
                int prefix = 1;
```

```
                for (int i = 0; i < nums.size(); i++) {
                        result[i] = prefix;
                        prefix *= nums[i];
                }

                int postfix = 1;
                for (int i = nums.size(); i>=0; i--) {
                        result[i] *= postfix;
                        postfix *= nums[i];
                }
        }
}
```

5. Leet Code Problem: Maximum Subarray

C/C++

```
class Solution {
public:
        int maxSubArray(vector<int>& nums) {
                int maxSum = INT_MIN;
                int currSum = 0;
                for (int i = 0; i < nums.size(); i++) {
                        currSum += nums[i];

                        if(currSum > maxSum) {
                                maxSum = currSum;
                        }
                        if(currSum < 0) {
                                currSum = 0;
                        }
                }
        return maxSum;
        }
}
```

6. Maximum Product Subarray

Unset

Given an integer array nums, find a

subarray
 that has the largest product, and return *the product*.
The test cases are generated so that the answer will fit in a **32-bit** integer.

C/C++

```
class Solution {
public:
        int maxProduct(std::vector<int>& nums) {
                if (nums.empty()) {
```

```
                        return 0;
                }

                int max_product_ending = nums[0];
                int min_product_ending = nums[0];
                int max_product = nums[0];

                for (int i = 1; i < nums.size(); i++) {
                        if (nums[i] < 0) {
                                std::swap(max_product_ending_here, min_product_ending_here);
                        }

                        max_product_ending_here = std::max(nums[i], max_product_ending_here *
                nums[i];
                        min_product_ending_here = std::max(nums[i], min_product_ending_here *
                nums[i];

                        max_product = std::max(max_product, max_product_ending_here);
                }

        }
        return max_product;
}
```

7. Find Minimum in Rotated Sorted Array

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

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

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

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

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

Example 1:

```
Input: nums = [3,4,5,1,2]
Output: 1
Explanation: The original array was [1,2,3,4,5] rotated 3 times.
```

Example 2:

```
Input: nums = [4,5,6,7,0,1,2]
Output: 0
Explanation: The original array was [0,1,2,4,5,6,7] and it was rotated
4 times.
```

Example 3:

```
Input: nums = [11,13,15,17]
Output: 11
Explanation: The original array was [11,13,15,17] and it was rotated 4
times.
```

C/C++

```
class Solution {
public:
        int findMIn(vector<int>& nums) {
                int left = 0;
                int right = nums.size() - 1;

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

                        if (nums[mid] > nums[right]) {
                        left = mid + 1;
                        }
                        else {
                                right = mid;
                        }
                }
        return nums[left];
        }
};
```

8. Search inside a loop

C/C++

```
class Solution {
public:
        int search(vector<int>& nums, int target) {
                for (int i = 0; i < nums.size(); i++) {
                        if (nums[i] == target) {
                                return i;
                        }
                }
        return -1;
        }
}
```

9. 3 Sum

C/C++

```
class Solution {
public:
        vector<vector<int>> threeSum(vector<int>& nums) {
                vector<vector<int>> triplets;
                sort(nums.begin(), nums.end());

                for (int i=0; i < nums.size()-2; i++) {
                        if (i>0 && nums[i]== nums[i-1]) {
                                continue;
                        }

                        int j = i + 1;
                        int k = nums.size()-1;

                                while(j<k) {
                                        int sum = nums[i] + nums[j] + nums[k];
                                        if (sum == 0) {
                                                triplets.push_back({nums[i], nums[j],
                                        nums[k]});
                                        j++;

                                        while (nums[j] == nums[j-1] && j < k) { j++;}
                                        }
                                }
                                else if (sum < 0) { // Negative Summen, Skip
                                j++;
                                }

                                else {
                                        k--;
                                }

                }
        return triplets;
        }
};
```

10. Leetcode Problem: Container with most Water

C/C++

```
class Solution{
public:
        int maxArea(vector<int>& height) {
                int maxArea = 0;
                int left = 0;
```

```
                int right = height.size() - 1;

                while (left < right) {
                        int width = right - left;
                        int h = min(height[left], height[right]);
                        int area = width * h;
                        maxArea = max(maxArea, area);

                        if(height[left] < height[right]) {
                                left++;
                        }
                        else {
                        right --;
                        }
                }
        return maxArea;
        }

}
```

11. Leetcode Problem: Add 2 Numbers without using + and -

C/C++

```
class Solution {
public:
        int getSum(int a, int b) {
                while (b!=0) {
                int carry = a & b;
                a = a ^ b;
                b = carry << 1;
                }
        return a;
        }

}
```

12. Leetcode Problem: Hamming Weight Calculation

C/C++

```
class Solution {
public:
 int hammingWeight(uint32_t n) {

  int count = 0;

  while(n) {
   count += n & 1;
   n >>= 1;
  }

  return count;

 }
};
```

13. Leetcode Problem: Counting Bits

C/C++

```
class Solution {
public:
        vector<int> countBits(int n){
                vector<int> ans(n+1);
                for (int i = 0; i <= n; i++) {
                        int count = 0;
                        int num = i;
                        while(num) {
                                count += num & 1;
                                num >>= 1;
                        }
                        ans[i] = count;
                }
        return ans;
        }
}
```

Example 1:

```
Input: n = 2
Output: [0,1,1]
Explanation:
0 --> 0
1 --> 1
2 --> 10
```

Example 2:

```
Input: n = 5
Output: [0,1,1,2,1,2]
Explanation:
0 --> 0
1 --> 1
2 --> 10
3 --> 11
4 --> 100
5 --> 101
```

14. LeetCode Problem: Missing Number

C/C++

```
class Solution {
public:
        int missingNumber(vector<int>& nums) {
        int missing = 0;

        for (int i = 0; i < nums.size(); ++i) {
                missing ^= i;
                missing ^= nums[i];
                }

                missing ^= nums.size();

                return missing;
        }
}
```

15. LeetCode Problem: Reverse Number bits

C/C++

```
class Solution {
public:
        uint32_t reverseBits(uint32_t n) {
        uint32_t result = 0;
        for (int i = 0; i<32; i++) {
                result <<= 1;            // result wird von rechts nach links gefüllt
                result |= (n & 1);       // Bitmaskierung n&1 Auswahl des Bits
        n >>= 1;
        }
        return result;
        }
};
```

Python

```
for bit in bits:
        pulse = np.zeros(sps)
        pulse[0] = bits * 2 - 1
        pulse_train = np.concatenate ((pulse_train, pulse))
```

Dynamic Programming

16. LeetCode Problem: climbStairs

C/C++

```
class Solution {
public:
        int climbStairs(int n) {
        if (n <= 2) {
                return n;
```

```
        }

        int prev1 = 1, prev2 = 2;

        for (int i = 3; i <= n; ++i) {
                int current = prev1 + prev2;
                prev1 = prev2;
                prev2 = current;
        }

        return prev2;
        }
};
```

17. LeetCode Problem: Longest Increasing subsequence:

Runtime efficient:

```
C/C++
class Solution {
public:
        int solve (vector<int>&a, int n, vector<int>&dp){
                if(n-1 == 0) return 1;
                int val = 0;
                if (dp[n-1] != -1) return dp[n-1];
                for (int i = 0; i < n-1; ++i) {
                        int x = solve(a, i+1, dp);
                        if (a[n-1] > a [i]) val = max(val,x);
                }
                return dp[n-1] = 1 + val;
        }

        int binarySearchMethod (vector<int>&a, int n) {
                vector<int> ans;
                ans.push_back(a[0]);
                for (int i = 0; i < n; ++i) {
                        if(ans[ans.size()-1] < a[i]) ans.push_back(a[i]));
                        else {
                                int target = a[i];
                                int start = 0, end = ans.size()-1, mid;
                                while (start < end) {
                                        mid = (start + end)/2;
                                        if      (ans[mid]< target) start = mid+1;
                                        else if(ans[mid] > target) end = mid;
                                        else {    end = mid;

                                        break;
                                        }
                                }
                        ans [end] = target;
                        }
                }
        return ans.size();
        }

}
```

oder:

C/C++

```
class Solution {
public:
        int legnthOfLIS( vector<int>& nums ) {
                int n = nums.size();
                if (n == 0) {
                        return 0;
                }

                std::vector<int> dp( n , 1 );

                for (int i = 1; i < n; ++i) {
                        for (int j = 0; j < i; ++j) {
                                if ( nums[i] > nums[j] ) {
                                        dp[i] = std::max( dp[i] , dp[j] + 1 );
                                }
                        }
                }
        return *std::max_element() dp.begin() , dp.end() };
};
```

18. Leetcode Problem: Subsequence of 2 strings

C/C++

```
class Solution {
public:
        int longestCommonSubsequence(string text1, string text2) {
                int m = text1.size();
                int n = text2.size();
        std::vector< std::vector<int >> dp( m + 1 , std::vector<int>(n + 1, 0));

        for (int i; i <= m; ++i) {
                for (int j; j <= n; ++j) {
                        if(text1[i - 1] == text2[j - 1]) {
                                dp[i][j] = dp [i-1] [j-1] + 1;
                        }
                        else {
                                dp[i][j] = std::max(dp[i-1][j], dp [i][j-1]);
                        }
                }

        }
        return dp[m][n];
        }
}
```

19. Leetcode Problem: Word Break

C/C++

```
class Solution {
public:
        bool wordBreak(string s, vector<string>& wordDict) {
        int n = s.length();
        std::unordered_set<std::string> wordSet (wordDict.begin(), wordDict.end());
```

```
        std::vector<bool> dp(n + 1, false);
        dp[0] = true;

        for (int i = 1; i <= n; ++i) {
                for (int j = 1; j < i; ++j) {
                        if (dp[j] && wordSet.count(s.substr(j, u-j))) {
                                dp[i] = true;
                                break;
                        }
                }
        }

        return dp[n];
        }
}
```

20. Leetcode Combination Sum

C/C++

```
class Solution {
public:
        int combinationSum4(vector<int>& nums, int target) {
        unordered_map<int, long long> dp;
        dp[0] = 1;

        for (int i= 1; i<= target; i++) {
                for (int num: nums) {}
                        if(i - num >= 0) {
                                if (dp [i] < numeric_limits<int>::max()-dp[i-num])
                                { dp [i] += dp[i-num];
                                }
                        }
                }
        return dp[target];
        }
}
```

21. Leetcode House Robber

C/C++

```
class Solution {
public:
        int rob(vector<int>& nums) {
        int n = nums.size();

        if(n == 0) {
                return 0;
        } else if (n == 1){
                return nums[0];
        }
        // dp[i] represents the maximum amount of money that can be robbed up to house i
        vector<int> dp(n,0);
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);

        for (int i = 2; i < n; ++i) {
```

```
                dp[i] = max (dp[i - 1], dp[i - 2] + nums[i]);
        }
        return dp[n-1];
        }

}
```

22. Leetcode: House Robber 2

C/C++

```
class Solution {
public:
        int rob(std::vector<int>& nums) {
        int n = nums.size();
        if (n == 0) {
        return 0;
        }
        else if (n == 1){
        return nums[0];
        }

        int result1 = robHelper(nums, 0, n - 2);
        int result2 = robHelper(nums, 1, n - 1);

        return std::max(result1, result2);
        }
private:
        int robHelper(const std::vector<int>& nums, int start, int end) {
        int prevMax = 0;
        int currMax = 0;
        for (int i = start; i <= end; ++i) {
                int temp = currMax;
                currMax = std::max(prevMax + nums [i], currMax);
                prevMax = temp;
        }
        return currMax;
        }
};
```

23. Leetcode Decode Ways

C/C++

```
class Solution {
public:
        int numDecodings(std::string s) {
                int n = s.size();

                if (n == 0 || s[0] == '0') {
                return 0;
                }

                std::vector<int> dp (n + 1, 0);
                dp [0] = 1;

                for (int i = 1; i <= n; ++i) {
                // Single digit Decoding
                if (s[i - 1] != '0') {
                        dp[i] += dp [i - 1];
```

```
                }

                // Double Digit Decoding
                if (i > 1 && s[i - 2] != '0' && stoi(s.substr(i - 2, 2)) <= 26) {
                        dp[i] += dp[i - 2];
                }
        }
        return dp[n];
        }
};
```

24. Leetcode Unique Paths

C/C++

```
class Solution{

public:
        int helper(int i, int j, vector<vector<int>> & dp){
                if(i == 0|| j == 0){
                        return 1;
                }
                if (dp[i][j] != -1)
                return dp[i][j];

                int x = helper(i-1, j, dp);
                int y = helper(i, j-1,dp);
                dp[i][j] = x + y;
                return dp[i][j];
        }

        int uniquePaths(int m, int n) {
                vector<vector<int>> dp(m, vector<int>(n, -1));
                return helper(m-1,n-1,dp);
        }
};
```

25. Leetcode: Jump Game

C/C++

```
class Solution {
public:
        bool canJump(vector<int>& nums) {
        int n = nums.size();
        int maxReach = 0;

        for(int i = 0; i < n; ++i) {
        if (i > maxReach) {
                return false;
        }

        maxReach = std::max(maxReach, i + nums[i]);

        if (maxReach >= n-1) {
                return true;
        }
        }
        return false;
        }
```

```
}
```

26. Leetcode Clone Graph

C/C++

```
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 {
private:
        std::unordered_map<Node*, Node*> visited;
public:
        Node* cloneGraph(Node* node) {
        if (node == nullptr) {
                return nullptr;
        }

        if (visited.find(node) != visited.end()) {
                return visited[node];
        }

        Node * cloneNode = new Node (node->val);

        visited[node] = cloneNode;

        for (Node* neighbor : node->neighbors) {
                cloneNode->neighbors.push_back(cloneGraph(neighbor));
        }
        return cloneNode;
        }
}
```

C/C++

```
class Solution {
public:
        void dfs(Node* node, Node* copy, vector<Node*> &vis)
```

```
        {
                vis(copy->val) = copy;

                for(auto &x:node->neighbors) {
                        if (vis[x->val] == NULL){
                                Node* newNode= new Node(x->val);

                                (copy->neighbors).push_back(NewNode);

                                dfs(x, NewNode,vis);
                        }
                        else

                                (copy->neighbors).push_back(vis[x->val]);
                }
        }

        Node* cloneGraph (Node* node) {

                if (node == NULL)
                        return NULL;

                vector<Node*> vis(1000, NULL);

                Node* copy = new Node(node->val);

                dfs(node, copy, vis);

                return copy;

        }
};
```

27. Leetcode Course Schedule

C/C++

```
class Solution {
public:
        bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {
        vector<vector<int>> v(numCourses);
        vector<int>degree(numCourses);

        for (int i = 0; i < prerequisites.size(); i++) {
                v[prerequisites[i][1]].push_back(prerequisites[i][0]);
                degree[prerequisites[i][0]++];
        }

        queue<int>q;
        for (int i = 0; i < numCourses; i++) {
                if(!degree[i])
                        q.push(i);
        }

        while(!q.empty()) {
                int cur = q.front();
                q.pop();
                for (int i = 0; i < v[cur].size(); i++) {
                        if(--degree[v[cur][i]] == 0)
                                q.push(v[cur][i]);
                }
        }

        for (int i = 0; i < numCourses; i++) {
                if (degree[i])
                        return false;
        }
        return true;
        }
};
```

28. Pacific Atlantic Water Flow

C/C++

```
class Solution {
public:
   std::vector<std::vector<int>> pacificAtlantic(std::vector<std::vector<int>>& heights) {
       std::vector<std::vector<int>> result;

       if (heights.empty() || heights[0].empty()) {
           return result;
       }

       int m = heights.size();
       int n = heights[0].size();

       // Create two 2D vectors to mark cells reachable from Pacific and Atlantic Oceans
       std::vector<std::vector<bool>> canReachPacific(m, std::vector<bool>(n, false));
       std::vector<std::vector<bool>> canReachAtlantic(m, std::vector<bool>(n, false));

       // DFS to mark cells reachable from Pacific Ocean
       for (int i = 0; i < m; ++i) {
           dfs(heights, canReachPacific, i, 0);
       }
       for (int j = 0; j < n; ++j) {
           dfs(heights, canReachPacific, 0, j);
       }

       // DFS to mark cells reachable from Atlantic Ocean
       for (int i = 0; i < m; ++i) {
           dfs(heights, canReachAtlantic, i, n - 1);
       }
       for (int j = 0; j < n; ++j) {
           dfs(heights, canReachAtlantic, m - 1, j);
       }

       // Find cells that can flow into both oceans
       for (int i = 0; i < m; ++i) {
           for (int j = 0; j < n; ++j) {
               if (canReachPacific[i][j] && canReachAtlantic[i][j]) {
                   result.push_back({i, j});
               }
           }
       }

       return result;
   }

private:
```

```
    void dfs(const std::vector<std::vector<int>>& heights,
             std::vector<std::vector<bool>>& canReach,
             int row, int col) {
        // Mark the current cell as reachable
        canReach[row][col] = true;

        static const std::vector<std::pair<int, int>> directions = {{0, 1}, {1, 0}, {0, -1}, {-1,
0}};

        // DFS to adjacent cells with equal or lower height
        for (const auto& dir : directions) {
            int newRow = row + dir.first;
            int newCol = col + dir.second;

            if (newRow >= 0 && newRow < heights.size() && newCol >= 0 && newCol < heights[0].size()
&&
                !canReach[newRow][newCol] && heights[newRow][newCol] >= heights[row][col]) {
                dfs(heights, canReach, newRow, newCol);
            }
        }
    }
};
```

29. Number of Islands

C/C++

```
class Solution {
public:
        int numIslands(vector<vector<char>>& grid) {
                if (grid.empty() || grid[0].empty()) {
                        return 0;
                }

                int m = grid.size();
                int n = grid[0].size();
                int numIslands = 0;

                // Peform DFS to find and mark connected islands

                for (int i = 0; i < m; ++i) {
                        for(int j = 0; j < n; ++j) {
                                if (grid[i][j] == '1') {
                                        numIslands++;
                                        dfs(grid, i, j);
                                }
                        }
                }
        return numIslands;
        }
```

```
private:
        void dfs(vector<vector<char>>& grid, int row, int col) {
                int m = grid.size();
                int n = grid[0].size();
                // Mark the current land cell is visited

                grid[row][col] = '0';

                // Define the four possible directions to move (up, down, left, right)
                static const vector<pair<int,int>> directions = {{-1,0}, {1,0}, {0,-1}, {0,-1}};

                for (const auto& dir : directions) {
                        int newRow = row + dir.first;
                        int newCol = col + dir.second;

                        // Check bounds and continue DFS if the neighboring cell is land
                        if (newRow >= 0 && newRow < m && newCol >= 0 && newCol < n &&
grid[newRow][newCol] == 1) {
                                dfs(grid, newRow, newCol);
                        }
                }
        }
};
```

30. LongestSequence

C/C++

```
class Solution {
public:
        int longestConsecutive(vector<int>& nums) {
        if(nums.size() == 0) return 0;
        sort(nums.begin(), nums.end());

        int ans = 1, start = nums[0], last = nums[0];
        int curr = nums[0];

        for (int i = 1; i < nums.size(); i++) {
                if (nums[i] == curr+1 || nums[i] == curr) {
                        curr = nums[i];
                        last = nums[i];
                }

                else {
                        start = nums[i];
                        curr = nums[i];
                }
                ans = max(ans, last-start+1);
        }
        return ans;
        }
};
```

31. Insert Interval

C/C++

```
class Solution {
public:
        vector<vector<inter>> insert(vector<vector<int>>& intervals, vector<int>& newInterval) {
        vector<vector<int>> result;
        int n = intervals.size();
        int i = 0;

        while (i < n && intervals[i][0] < newInterval[0]){
                newInterval[0] = std::min(newInterval[0], intervals[i][0]);
                newInterval[1] = std::max(newInterval[1], intervals[i][1]);
                i++;
        }

        result.push_back(newInterval);
        while(i < n) {
                result.push_back(intervals[i]);
                i++;
        }

        return result;

        }
};
```

32. Merge Intervals

C/C++

```
class Solution {
public:
        vector<vector<int>> merge(vector<vector<int>>& intervals) {
                if (intervals.empty()) {
                return {};
        }

        sort(intervals.begin(), intervals.end(), [](const auto& a, const auto& b) {return {};
        }

        vector<vector<int>> result;
        result.push_back(intervals[0]);

        for (int i; i < intervals.size(); ++i) {
                if (result.back()[1] >= intervals[i][0]) {
                        result.back()[1] = max(result.back()[1], intervals[i][1]);
                }
                else {
                        result.push_back(intervals[i]);

                }
        }
        return result;
        }

}
```

33. Non-overlapping Intervals

C/C++

```
class Soltuon {
public:
        int eraseOverlapIntervals(vector<vector<int>>& intervals) {
                if(intervals.empty()) {
                return 0;
                }

                sort(intervals.begin(), intervals.end(), [](const auto& a, const auto & b)
        {return a[1] < b[1];});

                int count = 1;
                int end = intervals[0][1];

                for (int i = 1; i < intervals.size(); ++i) {
                if (intervals[i][0] < end) {
                                continue;
                        }
                        end = intervals[i][1];
                        count++;
                }
                return intervals.size() - count;
        }

};
```

34. Leetcode Reverse Linked List

C/C++

```
#include <iostream>

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

class Solution {
public:
  ListNode* reverseList(ListNode* head) {
    // Base case: an empty list or a single-node list
    if (head == nullptr || head->next == nullptr) {
      return head;
    }

    // Recursive case: reverse the rest of the list
    ListNode* reversedList = reverseList(head->next);

    // Adjust the next pointer of the current node
    head->next->next = head;
    head->next = nullptr;

    return reversedList;
  }
};
```

35. Detect Cycle in a Linked List

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode(int x) : val(x), next(NULL) {}
* };
*/
class Solution {
public:
   bool hasCycle(ListNode* head) {
       if (head == nullptr || head->next == nullptr) {
           return false;
       }

       ListNode* slow = head;
       ListNode* fast = head;

       while (fast != nullptr && fast->next != nullptr) {
           slow = slow->next;
```

```
            fast = fast->next->next;

            // If the pointers meet, there is a cycle
            if (slow == fast) {
                return true;
            }
        }

        // No cycle found
        return false;
    }
};
```

36. Merge two Sorted Lists

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode() : val(0), next(nullptr) {}
*     ListNode(int x) : val(x), next(nullptr) {}
*     ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode dummy(0);
        ListNode* current = &dummy;

        while (list1 != nullptr && list2 != nullptr) {
            if (list1->val < list2->val) {
                current->next = list1;
                list1 = list1->next;
            } else {
                current->next = list2;
                list2 = list2->next;
            }
            current = current->next;
        }

        // Append the remaining nodes of list1 or list2
        if (list1 != nullptr) {
```

```
            current->next = list1;
        } else {
            current->next = list2;
        }

        return dummy.next;
    }
};
```

37. merge K Sorted Lists

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode() : val(0), next(nullptr) {}
*     ListNode(int x) : val(x), next(nullptr) {}
*     ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/

struct CompareNodes {
    bool operator()(const ListNode* a, const ListNode* b) const {
        return a->val > b->val;
    }
};
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, CompareNodes> minHeap;

        // Push the head of each linked list to the min heap
        for (ListNode* list : lists) {
            if (list != nullptr) {
                minHeap.push(list);
            }
        }

        ListNode dummy(0);
        ListNode* current = &dummy;

        while (!minHeap.empty()) {
```

```
            ListNode* smallest = minHeap.top();
            minHeap.pop();

            current->next = smallest;
            current = current->next;

            if (smallest->next != nullptr) {
                minHeap.push(smallest->next);
            }
        }

        return dummy.next;
    }
};
```

38. Remove Nth Node from End of List

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode() : val(0), next(nullptr) {}
*     ListNode(int x) : val(x), next(nullptr) {}
*     ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode dummy(0);
        dummy.next = head;
        ListNode* fast = &dummy;
        ListNode* slow = &dummy;

        // Move fast pointer n+1 nodes ahead
        for (int i = 0; i <= n; ++i) {
            fast = fast->next;
        }

        // Move both pointers until fast reaches the end
        while (fast != nullptr) {
            fast = fast->next;
            slow = slow->next;
```

```
        }

        // Remove the nth node
        ListNode* toRemove = slow->next;
        slow->next = toRemove->next;
        delete toRemove;

        return dummy.next;
    }
};
```

39. Reorder List

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode() : val(0), next(nullptr) {}
*     ListNode(int x) : val(x), next(nullptr) {}
*     ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/

class Solution {
public:
    void reorderList(ListNode* head) {
        if (head == nullptr || head->next == nullptr) {
            return;
        }

        // Step 1: Find the middle of the linked list
        ListNode* middle = findMiddle(head);

        // Step 2: Reverse the second half of the linked list
        ListNode* reversedSecondHalf = reverseList(middle->next);
        middle->next = nullptr; // Break the link to the second half

        // Step 3: Merge the first half and the reversed second half
        mergeLists(head, reversedSecondHalf);
    }

private:
    ListNode* findMiddle(ListNode* head) {
```

```
        ListNode* slow = head;
        ListNode* fast = head;

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

        return slow;
    }

    ListNode* reverseList(ListNode* head) {
        ListNode* prev = nullptr;
        ListNode* current = head;

        while (current != nullptr) {
            ListNode* nextNode = current->next;
            current->next = prev;
            prev = current;
            current = nextNode;
        }

        return prev;
    }

    void mergeLists(ListNode* list1, ListNode* list2) {
        while (list2 != nullptr) {
            ListNode* nextNode1 = list1->next;
            ListNode* nextNode2 = list2->next;

            list1->next = list2;
            list2->next = nextNode1;

            list1 = nextNode1;
            list2 = nextNode2;
        }
    }
};
```

40. Set Matrix Zeroes

C/C++

```
class Solution {
public:
```

```cpp
    void setZeroes(std::vector<std::vector<int>>& matrix) {
        int m = matrix.size();
        int n = matrix[0].size();

        bool firstRowZero = false;
        bool firstColZero = false;

        // Step 1: Mark rows and columns to be set to 0
        for (int i = 0; i < m; ++i) {
            for (int j = 0; j < n; ++j) {
                if (matrix[i][j] == 0) {
                    if (i == 0) {
                        firstRowZero = true;
                    }
                    if (j == 0) {
                        firstColZero = true;
                    }
                    matrix[i][0] = matrix[0][j] = 0;
                }
            }
        }

        // Step 2: Set marked rows and columns to 0
        for (int i = 1; i < m; ++i) {
            for (int j = 1; j < n; ++j) {
                if (matrix[i][0] == 0 || matrix[0][j] == 0) {
                    matrix[i][j] = 0;
                }
            }
        }

        // Set first row to 0 if needed
        if (firstRowZero) {
            for (int j = 0; j < n; ++j) {
                matrix[0][j] = 0;
            }
        }

        // Set first column to 0 if needed
        if (firstColZero) {
            for (int i = 0; i < m; ++i) {
                matrix[i][0] = 0;
            }
        }
    }
};
```

41. Spiral Matrix:

C/C++

```
class Solution {
public:
   std::vector<int> spiralOrder(std::vector<std::vector<int>>& matrix) {
       std::vector<int> result;
       if (matrix.empty() || matrix[0].empty()) {
           return result;
       }

       int m = matrix.size();
       int n = matrix[0].size();
       int top = 0, bottom = m - 1, left = 0, right = n - 1;

       while (top <= bottom && left <= right) {
           // Move right
           for (int j = left; j <= right; ++j) {
               result.push_back(matrix[top][j]);
           }
           top++;

           // Move down
           for (int i = top; i <= bottom; ++i) {
               result.push_back(matrix[i][right]);
           }
           right--;

           // Move left
           if (top <= bottom) {
               for (int j = right; j >= left; --j) {
                   result.push_back(matrix[bottom][j]);
               }
               bottom--;
           }

           // Move up
           if (left <= right) {
               for (int i = bottom; i >= top; --i) {
                   result.push_back(matrix[i][left]);
               }
               left++;
           }
       }
```

```
        return result;
    }
};
```

42. Rotate Image

C/C++

```
class Solution {
public:
    void rotate(std::vector<std::vector<int>>& matrix) {
        int n = matrix.size();

        // Rotate layer by layer
        for (int layer = 0; layer < n / 2; ++layer) {
            int first = layer;
            int last = n - 1 - layer;

            for (int i = first; i < last; ++i) {
                int offset = i - first;

                // Save top
                int top = matrix[first][i];

                // Move left to top
                matrix[first][i] = matrix[last - offset][first];

                // Move bottom to left
                matrix[last - offset][first] = matrix[last][last - offset];

                // Move right to bottom
                matrix[last][last - offset] = matrix[i][last];

                // Move top to right
                matrix[i][last] = top;
            }
        }
    }
};
```

43. Word Search

C/C++

```
#include <iostream>
#include <vector>

class Solution {
public:
  bool exist(std::vector<std::vector<char>>& board, std::string word) {
    if (board.empty() || board[0].empty() || word.empty()) {
      return false;
    }

    int m = board.size();
    int n = board[0].size();

    for (int i = 0; i < m; ++i) {
      for (int j = 0; j < n; ++j) {
        if (search(board, word, i, j, 0)) {
          return true;
        }
      }
    }

    return false;
  }

private:
  bool search(std::vector<std::vector<char>>& board, const std::string& word, int i, int j, int index) {
    if (index == word.size()) {
      return true; // Found the word
    }

    int m = board.size();
    int n = board[0].size();

    if (i < 0 || i >= m || j < 0 || j >= n || board[i][j] != word[index]) {
      return false; // Out of bounds or mismatched character
    }

    // Mark the current cell as visited
    char original = board[i][j];
    board[i][j] = '.';

    // Explore in all four directions
    bool found = search(board, word, i + 1, j, index + 1) ||
          search(board, word, i - 1, j, index + 1) ||
          search(board, word, i, j + 1, index + 1) ||
          search(board, word, i, j - 1, index + 1);

    // Revert the change (backtrack)
    board[i][j] = original;

    return found;
  }
};
```

44. Longest Substring Without Repeating Characters:

C/C++

```
class Solution {
```

```
public:
    int lengthOfLongestSubstring(std::string s) {
        std::unordered_map<char, int> charIndexMap;
        int maxLength = 0;
        int start = 0;

        for (int end = 0; end < s.size(); ++end) {
            char currentChar = s[end];

            if (charIndexMap.find(currentChar) != charIndexMap.end() && charIndexMap[currentChar]
>= start) {
                // If the character is repeated and its last occurrence is within the current
substring
                start = charIndexMap[currentChar] + 1;
            }

            charIndexMap[currentChar] = end;
            maxLength = std::max(maxLength, end - start + 1);
        }

        return maxLength;
    }
};
```

45. Longest Repeating Character Replacement

C/C++

```
class Solution {
public:
    int characterReplacement(std::string s, int k) {
        int maxLength = 0;
        int maxCount = 0;
        int start = 0;
        std::vector<int> charCount(26, 0);

        for (int end = 0; end < s.size(); ++end) {
            charCount[s[end] - 'A']++;
            maxCount = std::max(maxCount, charCount[s[end] - 'A']);

            // If the window size is greater than the maximum count + k, shrink the window
            if (end - start + 1 - maxCount > k) {
                charCount[s[start] - 'A']--;
                start++;
            }

            maxLength = std::max(maxLength, end - start + 1);
```

```
        }

        return maxLength;
    }
};
```

46. Minimum Window Substring

C/C++

```
class Solution {
public:
  std::string minWindow(std::string s, std::string t) {
    std::unordered_map<char, int> charCountT;
    for (char c : t) {
      charCountT[c]++;
    }

    int left = 0, right = 0;
    int minLen = INT_MAX;
    int minStart = 0;
    int requiredChars = t.size();
    std::unordered_map<char, int> charCountWindow;

    while (right < s.size()) {
      // Expand the window
      if (charCountT.find(s[right]) != charCountT.end()) {
        charCountWindow[s[right]]++;
        if (charCountWindow[s[right]] <= charCountT[s[right]]) {
          requiredChars--;
        }
      }

      // Shrink the window from the left
      while (requiredChars == 0) {
        int currentLen = right - left + 1;
        if (currentLen < minLen) {
          minLen = currentLen;
          minStart = left;
        }

        if (charCountT.find(s[left]) != charCountT.end()) {
          charCountWindow[s[left]]--;
          if (charCountWindow[s[left]] < charCountT[s[left]]) {
            requiredChars++;
          }
        }

        left++;
      }

      right++;
    }

    return (minLen == INT_MAX) ? "" : s.substr(minStart, minLen);
  }
};
```

47. Valid Anagram

C/C++

```
class Solution {
public:
  bool isAnagram(std::string s, std::string t) {
    if (s.size() != t.size()) {
      return false;
    }

    std::unordered_map<char, int> charCount;

    // Count characters in string s
    for (char c : s) {
      charCount[c]++;
    }

    // Update character counts based on string t
    for (char c : t) {
      if (charCount.find(c) == charCount.end() || charCount[c] <= 0) {
        return false; // Character not present in s or counts don't match
      }
      charCount[c]--;
    }

    return true;
  }
};
```

48. Group Anagrams

C/C++

```
class Solution {
public:
   std::vector<std::vector<std::string>> groupAnagrams(std::vector<std::string>& strs) {
       std::unordered_map<std::string, std::vector<std::string>> anagramGroups;

       for (const std::string& str : strs) {
           std::string sortedStr = str;
           std::sort(sortedStr.begin(), sortedStr.end());

           // Use sorted string as a key in the hash table
           anagramGroups[sortedStr].push_back(str);
       }

       // Convert the hash table values to the final result
       std::vector<std::vector<std::string>> result;
```

```
        for (const auto& group : anagramGroups) {
            result.push_back(group.second);
        }

        return result;
    }
};
```

49. Valid Parenthesis

C/C++

```
class Solution {
public:
  bool isValid(std::string s) {
    std::stack<char> brackets;
    std::unordered_map<char, char> bracketPairs = {
      {')', '('},
      {'}', '{'},
      {']', '['}
    };

    for (char c : s) {
      if (bracketPairs.find(c) != bracketPairs.end()) {
        // If the current character is a closing bracket
        char topElement = brackets.empty() ? '#' : brackets.top();
        brackets.pop();

        if (topElement != bracketPairs[c]) {
          return false; // Mismatched opening bracket
        }
      } else {
        // If the current character is an opening bracket, push onto the stack
        brackets.push(c);
      }
    }

    return brackets.empty(); // If the stack is empty, all brackets were matched
  }
};
```

50. Valid Palindrom

C/C++

```
class Solution {
public:
    bool isPalindrome(std::string s) {
        int left = 0, right = s.size() - 1;
```

```
        while (left < right) {
            // Skip non-alphanumeric characters from both ends
            while (left < right && !isalnum(s[left])) {
                left++;
            }

            while (left < right && !isalnum(s[right])) {
                right--;
            }

            // Compare characters (ignoring case)
            if (tolower(s[left]) != tolower(s[right])) {
                return false;  // Mismatched characters
            }

            left++;
            right--;
        }

        return true;  // All characters matched
    }
};
```

51. Longest Palindromic Substring

C/C++

```
class Solution {
public:
  std::string longestPalindrome(std::string s) {
    int n = s.size();
    if (n <= 1) {
      return s; // Trivial case: empty string or single character is a palindrome
    }

    // Initialize a 2D table to store palindrome information
    std::vector<std::vector<bool>> dp(n, std::vector<bool>(n, false));

    // All substrings of length 1 are palindromes
    for (int i = 0; i < n; ++i) {
      dp[i][i] = true;
    }

    int start = 0; // Start index of the longest palindromic substring
    int maxLength = 1; // Length of the longest palindromic substring

    // Check all substrings of length 2
    for (int i = 0; i < n - 1; ++i) {
      if (s[i] == s[i + 1]) {
        dp[i][i + 1] = true;
        start = i;
```

```
        maxLength = 2;
      }
    }

    // Check substrings of length 3 or more
    for (int len = 3; len <= n; ++len) {
      for (int i = 0; i <= n - len; ++i) {
        int j = i + len - 1;

        // Check if the current substring is a palindrome
        if (s[i] == s[j] && dp[i + 1][j - 1]) {
          dp[i][j] = true;

          // Update start index and maxLength
          start = i;
          maxLength = len;
        }
      }
    }

    return s.substr(start, maxLength);
  }
};
```

52. Palindromic Substrings

C/C++

```
class Solution {
public:
   int countSubstrings(std::string s) {
       int n = s.size();
       if (n <= 1) {
           return n;  // Trivial case: empty string or single character is a palindrome
       }

       // Initialize a 2D table to store palindrome information
       std::vector<std::vector<bool>> dp(n, std::vector<bool>(n, false));

       int count = 0;  // Number of palindromic substrings

       // All substrings of length 1 are palindromes
       for (int i = 0; i < n; ++i) {
           dp[i][i] = true;
           count++;
       }

       // Check all substrings of length 2
       for (int i = 0; i < n - 1; ++i) {
           if (s[i] == s[i + 1]) {
               dp[i][i + 1] = true;
```

```
                count++;
            }
        }

        // Check substrings of length 3 or more
        for (int len = 3; len <= n; ++len) {
            for (int i = 0; i <= n - len; ++i) {
                int j = i + len - 1;

                // Check if the current substring is a palindrome
                if (s[i] == s[j] && dp[i + 1][j - 1]) {
                    dp[i][j] = true;
                    count++;
                }
            }
        }

        return count;
    }
};
```

53. maximum Depth of Binary Tree

C/C++

```
struct TreeNode{
         int val;
         TreeNode* left;
         TreeNode* right;
         TreeNode(int x): val(x), left(nullptr), right(nullptr) {};
};


class Solution {
public:
         int maxDepth(TreeNode* root) {
         if (root == nullptr) {
                  return 0;
         }

         int leftDepth = maxDepth(root->left);
         int rightDepth = maxDepth(root->right);

         return std::max(leftDepth, rightDepth);
};
```

C/C++

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
  int maxDepth(TreeNode* root) {
    vector<vector<int>> ans;
    if (!root) {
      return ans.size();
    }
    queue<TreeNode*> q;
    q.push(root);

    while (!q.empty()) {
    int nodesAtCurrentLevel = q.size();
    vector<int> temp; // Create a single vector for the entire level

    while (nodesAtCurrentLevel--) {
      TreeNode* currNode = q.front();
      temp.push_back(currNode->val);
      q.pop();

      if (currNode->left) {
        q.push(currNode->left);
      }
      if (currNode->right) {
        q.push(currNode->right);
      }
    }

    ans.push_back(temp); // Push the vector for the entire level
  }
  return ans.size();

  }
};
```

54. Same tree:

C/C++

```
struct TreeNode {
        int val;
        TreeNode* left;
        TreeNode* right;
        TreeNode (int x) : val(x), left(nullptr), right(nullptr);
}

class Solution{
public:

        bool isSameTree(TreeNode* p, TreeNode* q) {
```

```
        // Base cases

        if (p == nullptr && q == nullptr) {
                return true;
        }
        else if (p == nullptr || q == nullptr){
                return false;
        }

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

55. Invert/Flip Binary Tree:

C/C++

```
struct TreeNode {
        int val;
        TreeNode* left;
        TreeNode* right;
        TreeNode (int x) : val(x), left(nullptr), right(nullptr);
}

class Solution {
public:
        TreeNode* invertTree(TreeNode* root) {
                if (root == nullptr){
                return nullptr;
                }

                TreeNode* temp = root -> left;
                root->left = invertTree(root->right);
                root->right = invertTree(temp);
        return root;
        }
};
```

Run-Time Optimized

C/C++

```
class Solution {
public:
        TreeNode* invertTree(TreeNode* root){
        if(root == nullptr) {return nullptr;}
         invertTree(root->left);
         invertTree(root->right);

        TreeNode* temp = root->left;
        root->left = root->right;
        root->right = temp;
        return root;
}
};
```

56. Binary Tree Maximum Path Sum

C/C++

```
class Solution {
public:
   int maxPathSum(TreeNode* root) {
       int maxSum = INT_MIN; // Initialize with the smallest possible integer

       maxPathSumHelper(root, maxSum);

       return maxSum;
   }

private:
   int maxPathSumHelper(TreeNode* root, int& maxSum) {
       if (root == nullptr) {
           return 0; // Base case: empty tree has sum 0
       }

       // Calculate the maximum path sum that includes the current node
       int leftSum = std::max(0, maxPathSumHelper(root->left, maxSum));
       int rightSum = std::max(0, maxPathSumHelper(root->right, maxSum));

       // Update the maximum path sum considering the current node
       maxSum = std::max(maxSum, leftSum + rightSum + root->val);

       // Return the maximum path sum that extends from the current node to one of its children
       return std::max(leftSum, rightSum) + root->val;
   }
};
```

Run-Time Optimized:

C/C++

```
class Solution{
public:
        int solve(TreeNode* root; int &maxi)
        {
                if(root == NULL) return 0;
                int l= solve (root->left, maxi);
                int r= solve (root->right, maxi);
                if(l<0) l=0;
                if(r<0) r=0;
                maxi = max(maxi, l+r+root->val) ;
                return maxi(l,r) + root-val;
        }
        int maxPathSum(TreeNode* root) {
                int maxi= INT_MIN;
                solve (root,maxi);
                return maxi;
                }
```

```
};
```

57. Binary Tree Level Order Traversal

C/C++

```
class Solution {
public:
        std::vector<vector<int>>  levelOrder(TreeNode* root) {
        vector<vector<int>> result;
        if (root == nullptr) {
                return result;
        }

        std::queue<TreeNode*> q;
        q.push(root);

        while(!q.empty()) {
                int levelSize = q.size();
                vector<int> currentLevel;

        for (int i = 0; i < levelSize; ++i) {
                TreeNode* node = q.front();
                q.pop();
                currentLevel.push_back(node->val);

        if ( node -> left) {q.push(node->left);}
        if ( node -> right) {q.push(node->right);}
        }
        result.push_back(currentLevel);
        }
        return result;
        }
};
```

58. Serialize and Deserialize Binary Tree

C/C++

```
#include <iostream>
#include <sstream>
#include <string>

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

class Codec {
public:
  // Encodes a tree to a single string.
  std::string serialize(TreeNode* root) {
    std::ostringstream ss;
    serializeHelper(root, ss);
    return ss.str();
  }
```

```
  // Decodes your encoded data to tree.
  TreeNode* deserialize(std::string data) {
    std::istringstream ss(data);
    return deserializeHelper(ss);
  }

private:
  void serializeHelper(TreeNode* root, std::ostringstream& ss) {
    if (root == nullptr) {
      ss << "null "; // Represent null nodes with "null"
    } else {
      ss << root->val << ' ';
      serializeHelper(root->left, ss);
      serializeHelper(root->right, ss);
    }
  }

  TreeNode* deserializeHelper(std::istringstream& ss) {
    std::string token;
    ss >> token;

    if (token == "null") {
      return nullptr; // Null node encountered
    } else {
      TreeNode* root = new TreeNode(std::stoi(token));
      root->left = deserializeHelper(ss);
      root->right = deserializeHelper(ss);
      return root;
    }
  }
};

// Example usage:
// TreeNode* root = ...; // Your binary tree
// Codec codec;
// std::string serialized = codec.serialize(root);
// TreeNode* deserialized = codec.deserialize(serialized);
```

59. Subtree of Another Tree

C/C++

```
#include <iostream>
#include <unordered_map>

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

class Solution {
public:
  bool isSubtree(TreeNode* root, TreeNode* subRoot) {
    if (root == nullptr) {
      return false; // Empty tree, no subtree can match
    }
```

```
    if (isIdentical(root, subRoot)) {
      return true; // Found a matching subtree
    }

    // Recursively check in the left and right subtrees
    return isSubtree(root->left, subRoot) || isSubtree(root->right, subRoot);
  }

private:
  bool isIdentical(TreeNode* node1, TreeNode* node2) {
    // Helper function to check if two trees are identical
    if (node1 == nullptr && node2 == nullptr) {
      return true; // Both nodes are null, considered identical
    }

    if (node1 == nullptr || node2 == nullptr) {
      return false; // One node is null, the other is not; considered different
    }

    return (node1->val == node2->val) &&
       isIdentical(node1->left, node2->left) &&
       isIdentical(node1->right, node2->right);
  }
};
```

60. Construct Binary Tree from Preorder and Inorder Traversal

C/C++

```
class Solution {
public:
        TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        for (int i = 0; i < inorder.size(); ++i) {
                inorderMap[inOrder[i]] = i;
        }

        return buildTreeHelper(preOrder, inorder, 0, 0, inorder.size() - 1);
        }

private:
        unordered_map <int, int> inorderMap;
        TreeNode* buildTreeHelper(vector<int>& preOrder, vector<int>& inorder, int preStart,
inStart, int inEnd) {
        if (preStart > preorder.size() - 1 || inStart > inEnd) {
                return nullptr;
                }
        int rootValue = preorder[preStart];
        TreeNode* root = new TreeNode(rootValue);

        int inIndex = inorderMap[rootValue];

        root->left = buildTreeHelper(preOrder, inorder, preStart + 1, inStart, inIndex - 1);
        root->right = buildTreeHelper(preOrder, inorder, preStart + 1, inStart, inIndex + 1,
        inEnd);

        return root;
        }
```

```
}
```

61. Validate Binary Search Tree

C/C++

```
class Solution {
public:
        bool isValidBST(TreeNode* root) {
        return isValidBSTHelper(root, std::numeric_limits<long>::min(),
std::numeric_limits<long>::max());
        }
private:
        bool isValidBSTHelper(TreeNode* root, long lower, long upper) {
        if(root == nullptr) {
                return true;
        }

        if(root->val <= lower || root->val >= upper) {
                return false;
        }

        return isValidBSTHelper(root->left, lower, root->val) && isValidBSTHelper(root->right,
root->val, upper);
        }
};
```

62. Kth Smallest Element in a BST

C/C++

```
class Solution {
public:
  int kthSmallest(TreeNode* root, int k) {
    int count = 0;
    int result = 0;

    std::stack<TreeNode*> st;

    while (root != nullptr || !st.empty()) {
      while (root != nullptr) {
        st.push(root);
        root = root->left;
      }

      root = st.top();
      st.pop();

      // Process the current node
      count++;
      if (count == k) {
        result = root->val;
        break;
      }
```

```
      root = root->right;
    }

    return result;
  }
};
```

63. Lowest Common Ancestor of BST

C/C++

```
class Solution {
public:
  TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if (root == nullptr) {
      return nullptr; // Empty tree, no LCA
    }

    int rootVal = root->val;
    int pVal = p->val;
    int qVal = q->val;

    if (pVal < rootVal && qVal < rootVal) {
      return lowestCommonAncestor(root->left, p, q); // Both nodes are in the left subtree
    } else if (pVal > rootVal && qVal > rootVal) {
      return lowestCommonAncestor(root->right, p, q); // Both nodes are in the right subtree
    } else {
      return root; // Found the LCA
    }
  }
};
```

64. Implement Trie (Prefix Tree)

C/C++

```
struct Node {
    bool completeWord;
    Node *next[26] {};
    Node(): completeWord(false) {}
};

class Trie {
private:
    Node *root;

    int characterPos(char c) {
        return c - 97;
    }

public:
    Trie() {
```

```
        ios_base::sync_with_stdio(false); cin.tie(NULL);
        root = new Node;
    }

    void insert(string word) {
        Node *node = root;

        for (char c:word) {
           int pos = characterPos(c);
           if (node->next[pos] == nullptr)
               node->next[pos] = new Node;
           node = node->next[pos];
        }

        node->completeWord = true;
    }

    bool search(string word) {
        Node *node = root;
        for (char c:word) {
            int pos = characterPos(c);
            if (node->next[pos] == nullptr)
                return false;
            node = node->next[pos];
        }

        return node->completeWord;
    }

    bool startsWith(string prefix) {
        Node *node = root;
        for (char c:prefix) {
            int pos = characterPos(c);
            if (node->next[pos] == nullptr)
                return false;
            node = node->next[pos];
        }

        return true;
    }
};
```

65. Add and Search Word

C/C++

```
class TrieNode {
public:
   std::unordered_map<char, TrieNode*> children;
   bool isEndOfWord;

   TrieNode() : isEndOfWord(false) {}
};

class WordDictionary {
```

```
private:
    TrieNode* root;

public:
    WordDictionary() {
        root = new TrieNode();
    }

    void addWord(const std::string& word) {
        TrieNode* node = root;
        for (char ch : word) {
            if (node->children.find(ch) == node->children.end()) {
                node->children[ch] = new TrieNode();
            }
            node = node->children[ch];
        }
        node->isEndOfWord = true;
    }

    bool search(const std::string& word) {
        return searchHelper(root, word, 0);
    }

private:
    bool searchHelper(TrieNode* node, const std::string& word, int index) {
        if (index == word.length()) {
            return node != nullptr && node->isEndOfWord;
        }

        char ch = word[index];
        if (ch == '.') {
            for (const auto& child : node->children) {
                if (searchHelper(child.second, word, index + 1)) {
                    return true;
                }
            }
            return false;
        } else {
            if (node->children.find(ch) == node->children.end()) {
                return false;
            }
            return searchHelper(node->children[ch], word, index + 1);
        }
    }
};
```

66. Word Search II

C/C++

```
class TrieNode {
public:
   TrieNode* child[26] = {NULL};
   bool taken=false;
   bool isEnd=false;
   TrieNode() {};
   void add(string str) {
       auto cur = this;
       for(auto c : str) {
           if(!cur->child[c-'a']) cur->child[c-'a'] = new TrieNode();
           cur = cur->child[c-'a'];
       }
       cur->isEnd = true;
   };
};


class Solution {
private:
   void helper(vector<vector<char>>& board, int i, int j, string sofar, TrieNode* node,
vector<string>& res) {
       char c= board[i][j];
       if(node->child[c-'a'] == NULL) return;
       board[i][j] = '.';
       sofar += c;
       node = node->child[c-'a'];
       if(node->isEnd && !node->taken) { res.push_back(sofar); node->taken = true; }
       int di[4]={0,1,0,-1}, dj[4]={1,0,-1,0};
       for(int k=0; k<4; ++k) {
           int i1=i+di[k], j1=j+dj[k];
           if(i1<0 || i1>=board.size() || j1<0 || j1>=board[0].size() || board[i1][j1] == '.')
continue;
           helper(board, i1, j1, sofar, node, res);
       }
       board[i][j] = c;
   }

public:
   vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {
       TrieNode root;
       for(auto w:words) root.add(w);
       int m = board.size(), n = board[0].size();
       vector<string> res;
       for(int i=0; i<m; ++i)
           for(int j=0; j<n; ++j) {
```

```
                helper(board, i, j, "", &root, res);
            }
        return res;
    }
};
```

67. Merge K Sorted Lists

C/C++

```
/**
* Definition for singly-linked list.
* struct ListNode {
*     int val;
*     ListNode *next;
*     ListNode() : val(0), next(nullptr) {}
*     ListNode(int x) : val(x), next(nullptr) {}
*     ListNode(int x, ListNode *next) : val(x), next(next) {}
* };
*/
// Definition for singly-linked list.

class CompareNodes {
public:
    bool operator()(const ListNode* a, const ListNode* b) const {
        return a->val > b->val;
    }
};

class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        priority_queue<ListNode*, vector<ListNode*>, CompareNodes> minHeap;

        // Push the heads of all linked lists into the min-heap
        for (ListNode* list : lists) {
            if (list) {
                minHeap.push(list);
            }
        }

        // Dummy node to simplify code
        ListNode* dummy = new ListNode(0);
        ListNode* current = dummy;

        // Process the min-heap until it's empty
```

```
        while (!minHeap.empty()) {
            // Pop the smallest element from the min-heap
            ListNode* smallest = minHeap.top();
            minHeap.pop();

            // Add the smallest element to the result list
            current->next = smallest;
            current = current->next;

            // Move to the next element in the popped list
            if (smallest->next) {
                minHeap.push(smallest->next);
            }
        }

        return dummy->next;
    }
};

// Helper function to create a linked list from a vector
ListNode* createLinkedList(const vector<int>& values) {
    ListNode* dummy = new ListNode(0);
    ListNode* current = dummy;

    for (int value : values) {
        current->next = new ListNode(value);
        current = current->next;
    }

    return dummy->next;
}

// Helper function to print a linked list
void printLinkedList(ListNode* head) {
    while (head) {
        cout << head->val << " ";
        head = head->next;
    }
    cout << endl;
}
```

68. Top K Frequent Elements

C/C++

```cpp
class Solution {
public:
   vector<int> topKFrequent(vector<int>& nums, int k) {
       unordered_map<int, int> frequencyMap;
       for (int num : nums) {
           frequencyMap[num]++;
       }

       // Custom comparator for the max heap
       auto compare = [&](const int& a, const int& b) {
           return frequencyMap[a] < frequencyMap[b];
       };

       // Max heap to keep track of the k most frequent elements
       priority_queue<int, vector<int>, decltype(compare)> maxHeap(compare);

       // Populate the max heap with unique elements
       for (const auto& entry : frequencyMap) {
           maxHeap.push(entry.first);
       }

       // Extract the k most frequent elements from the heap
       vector<int> result;
       for (int i = 0; i < k; ++i) {
           result.push_back(maxHeap.top());
           maxHeap.pop();
       }

       return result;
   }
};
```

69. Find Median from Data Stream

C/C++

```cpp
class MedianFinder {
public:
  priority_queue<int> maxHeap; // Max heap for the smaller half
  priority_queue<int, vector<int>, greater<int>> minHeap; // Min heap for the larger half

  MedianFinder() {}

  void addNum(int num) {
    if (maxHeap.empty() || num <= maxHeap.top()) {
      maxHeap.push(num);
    } else {
```

```
      minHeap.push(num);
    }

    // Balance the heaps
    if (maxHeap.size() > minHeap.size() + 1) {
      minHeap.push(maxHeap.top());
      maxHeap.pop();
    } else if (minHeap.size() > maxHeap.size()) {
      maxHeap.push(minHeap.top());
      minHeap.pop();
    }
  }

  double findMedian() {
    if (maxHeap.size() == minHeap.size()) {
      return (maxHeap.top() + minHeap.top()) / 2.0;
    } else {
      return maxHeap.top();
    }
  }
};
```

70. Alien Dictionary (Premium)

C/C++

```
class Solution {
public:
    string alienOrder(vector<string>& words) {
        unordered_map<char, unordered_set<char>> graph;
        unordered_map<char, int> inDegree;

        // Initialize inDegree for all characters
        for (string word : words) {
            for (char ch : word) {
                inDegree[ch] = 0;
            }
        }

        // Build the graph and calculate inDegree
        for (int i = 0; i < words.size() - 1; ++i) {
            string word1 = words[i];
            string word2 = words[i + 1];

            int minLength = min(word1.length(), word2.length());

            for (int j = 0; j < minLength; ++j) {
                char ch1 = word1[j];
                char ch2 = word2[j];

                if (ch1 != ch2) {
                    if (!graph[ch1].count(ch2)) {
                        graph[ch1].insert(ch2);
                        inDegree[ch2]++;
                    }
                    break; // No need to check further characters
                }
            }
```

```
        }

        // Topological sort using BFS
        queue<char> q;
        for (auto entry : inDegree) {
            if (entry.second == 0) {
                q.push(entry.first);
            }
        }

        string result;
        while (!q.empty()) {
            char current = q.front();
            q.pop();
            result += current;

            for (char neighbor : graph[current]) {
                inDegree[neighbor]--;
                if (inDegree[neighbor] == 0) {
                    q.push(neighbor);
                }
            }
        }

        // Check if the graph is valid (no cycle)
        if (result.length() != inDegree.size()) {
            return "";
        }

        return result;
    }
};
```

71. Graph Valid Tree (Premium)

C/C++

```
#include <iostream>
#include <vector>
#include <unordered_set>

using namespace std;

class Solution {
public:
  bool validTree(int n, vector<vector<int>>& edges) {
    vector<unordered_set<int>> adjList(n);

    // Build the adjacency list
    for (const auto& edge : edges) {
      adjList[edge[0]].insert(edge[1]);
      adjList[edge[1]].insert(edge[0]);
    }

    vector<bool> visited(n, false);

    // Check for cycle using DFS
    if (hasCycle(adjList, visited, 0, -1)) {
```

```
      return false;
    }

    // Check if all nodes are connected
    for (bool visitStatus : visited) {
      if (!visitStatus) {
        return false;
      }
    }

    return true;
  }

private:
  bool hasCycle(const vector<unordered_set<int>>& adjList, vector<bool>& visited, int node, int parent)
{
    visited[node] = true;

    for (int neighbor : adjList[node]) {
      if (!visited[neighbor]) {
        if (hasCycle(adjList, visited, neighbor, node)) {
          return true;
        }
      } else if (neighbor != parent) {
        return true; // Found a cycle
      }
    }

    return false;
  }
};

int main() {
  Solution solution;

  int n = 5;
  vector<vector<int>> edges = {{0, 1}, {0, 2}, {0, 3}, {1, 4}};

  bool result = solution.validTree(n, edges);

  cout << "Is the graph a valid tree? " << (result ? "Yes" : "No") << endl;

  return 0;
}
```

C/C++

```
class Solution {
public:
        bool validTree(int n, vector<vector<int>>& edges) {
        vector<unordered_set<int>> adjList(n);

// Build the adjacency list

        for (const auto& edge: edges) {
                adjList[edge[0]].insert(edge[1]);
                adjList[edge[1]].insert(edge[0]);
        }
        vector <bool> visited(n, false);
```

```
        if(hasCycle(adjList, visisted, 0, -1)) {
                return false;
        }

        for (bool visitStatus : visited) {
                if (!visitStatus) {
                        return false;
                }
        }

        return true;
        }

private:
        bool hasCycle (const vector<unordered_set<int>>& adjList, vector<bool>& visited, int node,
int parent) {
        visited[node] = true;

        for (int neighbor : adjList[node]) {
                if(!visited[neighbor]) {
                        if (hasCycle(adjList, visited, neighbor, node)) {
                        return true;
                }
                } else if (neighbor != parent) {
                        return true;
                }
        }
        return false;
        }
};
```

72. Number of Connected Components in an Undirected Graph (Premium)

C/C++

```
class Solution {
public:
        int countComponents(int n, vector<vector<int>>& edges) {
        vector<int> parent(n, 1);

        int components = n;

        for (const auto& edge : edges) {
                int root1 = find(parent, edge[0];
                int root2 = find(parent, edge[1];

        if (root1 != root2) {
                parent[root1] = root2;
                components--;
        }
        }
        return components;
        }
private:
        int find(vector<int>& parent, int node) {
        while (parent[node] != -1) {
                node = parent[node];
```

```
        }
        return node;
        }
};
```

73. Meeting Rooms (Premium)

C/C++

```
class Solution {
public:
  bool canAttendMeetings(vector<vector<int>>& intervals) {
    // Sort intervals based on the start time
    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b) {
      return a[0] < b[0];
    });

    // Check for overlapping intervals
    for (int i = 1; i < intervals.size(); ++i) {
      if (intervals[i][0] < intervals[i - 1][1]) {
        return false; // Overlapping intervals
      }
    }

    return true;
  }
};
```

74. Meeting Rooms II (Premium)

C/C++

```
class Solution {
public:
  int minMeetingRooms(vector<vector<int>>& intervals) {
    if (intervals.empty()) {
      return 0;
    }

    // Sort intervals based on start time
    sort(intervals.begin(), intervals.end(), [](const vector<int>& a, const vector<int>& b) {
      return a[0] < b[0];
    });

    // Min heap to track end times of ongoing meetings
    priority_queue<int, vector<int>, greater<int>> minHeap;

    // Add the end time of the first meeting
    minHeap.push(intervals[0][1]);

    // Iterate through the remaining meetings
    for (int i = 1; i < intervals.size(); ++i) {
      if (intervals[i][0] >= minHeap.top()) {
        // The current meeting can reuse a room, update the end time
        minHeap.pop();
```

```
      }

      // Add the end time of the current meeting
      minHeap.push(intervals[i][1]);
    }

    // The size of the min heap represents the number of meeting rooms required
    return minHeap.size();
  }
};
```

75. Encode and Decode Strings (Premium)

C/C++

```
class Codec {
public:
  // Encodes a list of strings to a single string.
  string encode(vector<string>& strs) {
    string result;
    for (const string& str : strs) {
      result += to_string(str.length()) + "#" + str;
    }
    return result;
  }

  // Decodes a single string to a list of strings.
  vector<string> decode(string s) {
    vector<string> result;
    int i = 0;
    while (i < s.length()) {
      // Find the delimiter "#"
      int delimiterIndex = s.find("#", i);
      // Extract the length of the next string
      int length = stoi(s.substr(i, delimiterIndex - i));
      // Extract the string using substr
      result.push_back(s.substr(delimiterIndex + 1, length));
      // Move to the next position after the extracted string
      i = delimiterIndex + 1 + length;
    }
    return result;
  }
};
```