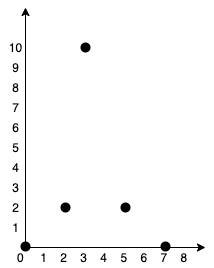
**CS2420 2021 Fall Final**

1 **(10 points)**. You are given an array points representing integer coordinates of some points on a 2D-plane, where points[i] = [xi, yi].

The cost of connecting two points [xi, yi] and [xj, yj] is the **manhattan distance** between them: |xi - xj| + |yi - yj|, where |val| denotes the absolute value of val.

Return *the minimum cost to make all points connected.* All points are connected if there is **exactly one** simple path between any two points.

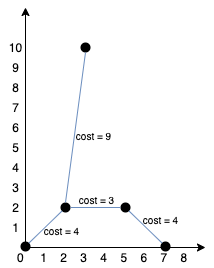
**Example 1:**



**Input:** points = [[0,0],[2,2],[3,10],[5,2],[7,0]]

**Output:** 20

**Explanation:**



We can connect the points as shown above to get the minimum cost of 20.

Notice that there is a unique path between every pair of points.

**Example 2:**

**Input:** points = [[3,12],[-2,5],[-4,1]]

**Output:** 18

**Example 3:**

**Input:** points = [[0,0],[1,1],[1,0],[-1,1]]

**Output:** 4

**Example 4:**

**Input:** points = [[-1000000,-1000000],[1000000,1000000]]

**Output:** 4000000

**Example 5:**

**Input:** points = [[0,0]]

**Output:** 0

**Constraints:**

* 1 <= points.length <= 1000
* -106 <= xi, yi <= 106
* All pairs (xi, yi) are distinct.

Finish the function below:

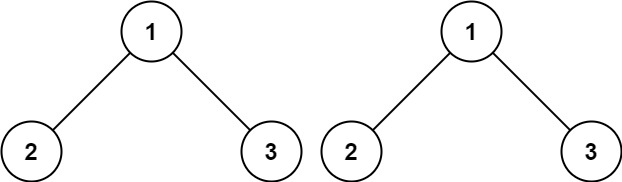
int minCostConnectPoints(vector<vector<int>>& points) {

}

2 **(10 points)**. Given the roots of two binary trees p and q, write a function to check if they are the same or not.

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

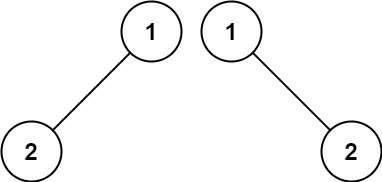
**Example 1:**



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

**Output:** true

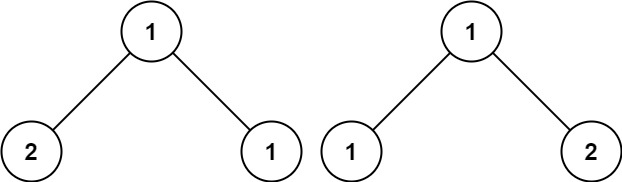
**Example 2:**



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

**Output:** false

**Example 3:**



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

**Output:** false

**Constraints:**

* The number of nodes in both trees is in the range [0, 100].
* -104 <= Node.val <= 104

Finish the function below:

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

}

3 **(10 points)**. Given the root of a binary tree, return *the inorder traversal of its nodes' values*.

**Example 1:**



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

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

**Example 2:**

**Input:** root = []

**Output:** []

**Example 3:**

**Input:** root = [1]

**Output:** [1]

**Example 4:**



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

**Output:** [2,1]

**Example 5:**



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

**Output:** [1,2]

**Constraints:**

* The number of nodes in the tree is in the range [0, 100].
* -100 <= Node.val <= 100

Finish your function below using recursion

void inorderTraversal(TreeNode\* root) {

}

4 **(10 points)**. Rewrite your recursive implementation of in-order traversal from the previous question (#3) iteratively. That is, your implementation cannot have a recursive call.

vector<int> inorderTraversalNonRecursive(TreeNode\* root) {

}

5 **(10 points)**. Given the root of a binary tree and an integer targetSum, return true if the tree has a **root-to-leaf** path such that adding up all the values along the path equals targetSum.

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

**Example 1:**



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

**Output:** true

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

**Example 2:**



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

**Output:** false

**Explanation:** There two root-to-leaf paths in the tree:

(1 --> 2): The sum is 3.

(1 --> 3): The sum is 4.

There is no root-to-leaf path with sum = 5.

**Example 3:**

**Input:** root = [], targetSum = 0

**Output:** false

**Explanation:** Since the tree is empty, there are no root-to-leaf paths.

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -1000 <= Node.val <= 1000
* -1000 <= targetSum <= 1000

Finish the function below:

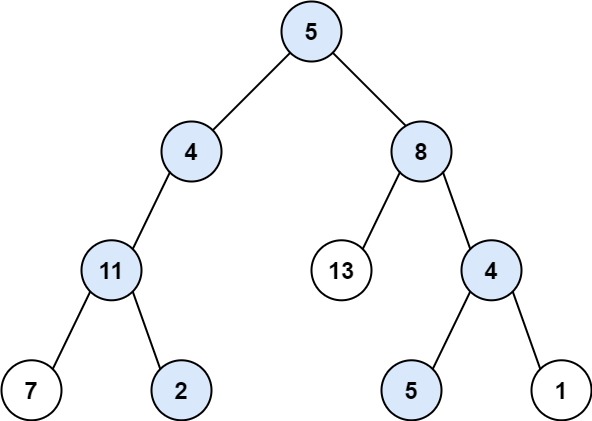
bool hasPathSum(TreeNode\* root, int targetSum) {

}

6 **(10 points)**. Similar to the previous question (#5), given the root of a binary tree and an integer targetSum, return *all****root-to-leaf****paths where the sum of the node values in the path equals*targetSum*. Each path should be returned as a list of the node****values****, not node references*.

A **root-to-leaf** path is a path starting from the root and ending at any leaf node. A **leaf** is a node with no children.

**Example 1:**



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

**Output:** [[5,4,11,2],[5,8,4,5]]

**Explanation:** There are two paths whose sum equals targetSum:

5 + 4 + 11 + 2 = 22

5 + 8 + 4 + 5 = 22

**Example 2:**



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

**Output:** []

**Example 3:**

**Input:** root = [1,2], targetSum = 0

**Output:** []

**Constraints:**

* The number of nodes in the tree is in the range [0, 5000].
* -1000 <= Node.val <= 1000
* -1000 <= targetSum <= 1000

Finish the function below:

vector<vector<int>> pathSum(TreeNode\* root, int targetSum) {

}

7 **(10 points)**. Given an integer array nums, return all the triplets [nums[i], nums[j], nums[k]] such that i != j, i != k, and j != k, and nums[i] + nums[j] + nums[k] == 0.

Notice that the solution set must not contain duplicate triplets.

**Example 1:**

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

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

**Example 2:**

**Input:** nums = []

**Output:** []

**Example 3:**

**Input:** nums = [0]

**Output:** []

**Constraints:**

* 0 <= nums.length <= 3000
* -105 <= nums[i] <= 105

Finish the function below:

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

}

8 **(10 points)**. Given two binary strings a and b, return *their sum as a binary string*.

**Example 1:**

**Input:** a = "11", b = "1"

**Output:** "100"

**Example 2:**

**Input:** a = "1010", b = "1011"

**Output:** "10101"

**Constraints:**

* 1 <= a.length, b.length <= 104
* a and b consist only of '0' or '1' characters.
* Each string does not contain leading zeros except for the zero itself.

Finish the function below:

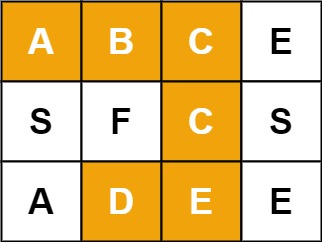
string addBinary(string a, string b) {

}

9 **(10 points)**. Given an m x n grid of characters board and a string word, return true *if* word *exists in the grid*.

The word can be constructed from letters of sequentially adjacent cells, where adjacent cells are horizontally or vertically neighboring. The same letter cell may not be used more than once.

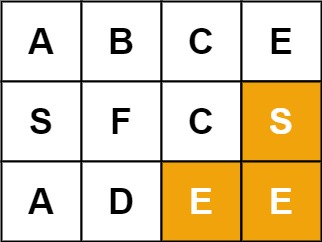
**Example 1:**



**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCCED"

**Output:** true

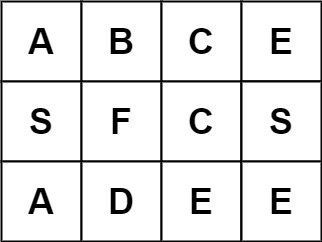
**Example 2:**



**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "SEE"

**Output:** true

**Example 3:**



**Input:** board = [["A","B","C","E"],["S","F","C","S"],["A","D","E","E"]], word = "ABCB"

**Output:** false

**Constraints:**

* m == board.length
* n = board[i].length
* 1 <= m, n <= 6
* 1 <= word.length <= 15
* board and word consists of only lowercase and uppercase English letters.

Finish the function below:

bool exist(vector<vector<char>>& board, string word) {

}

10 **(10 points)**. Given an array nums of size n, return *the majority element*.

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

**Example 1:**

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

**Output:** 3

**Example 2:**

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

**Output:** 2

**Constraints:**

* n == nums.length
* 1 <= n <= 5 \* 104
* -231 <= nums[i] <= 231 - 1

Finish the function below using linear time complexity and O(1) space

int majorityElement(vector<int>& nums) {

}