Given a m x 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.

**Example 1:**



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

**Output:** 7

**Explanation:** Because the path 1 → 3 → 1 → 1 → 1 minimizes the sum.

**Example 2:**

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

**Output:** 12

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example grid

int[][] grid = {

{1, 3, 1},

{1, 5, 1},

{4, 2, 1}

};

System.out.println("Input Grid:");

printGrid(grid);

// Find the minimum path sum

int result = solution.minPathSum(grid);

System.out.println("Minimum Path Sum: " + result);

}

// Utility method to print a 2D grid

public static void printGrid(int[][] grid) {

for (int[] row : grid) {

for (int val : row) {

System.out.print(val + " ");

}

System.out.println();

}

}

}

class Solution {

public int minPathSum(int[][] grid) {

int m = grid.length;

int n = grid[0].length;

// Create a DP array to store minimum path sums

int[][] dp = new int[m][n];

// Initialize the starting point

dp[0][0] = grid[0][0];

// Fill the first row

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

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

}

// Fill the first column

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

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

}

// Fill the rest of the DP table

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

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

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

}

}

// Return the result at the bottom-right corner

return dp[m - 1][n - 1];

}

}

Given the root of a binary tree, determine if it is a valid binary search tree (BST).

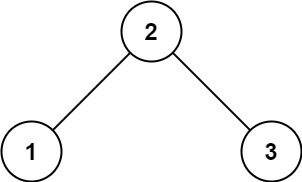
A **valid BST** is defined as follows:

* The left

subtree

 of a node contains only nodes with keys **less than** the node's key.

* The right subtree of a node contains only nodes with keys **greater than** the node's key.
* Both the left and right subtrees must also be binary search trees.



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

**Output:** true

class TreeNode {

int val;

TreeNode left;

TreeNode right;

TreeNode() {}

TreeNode(int val) { this.val = val; }

TreeNode(int val, TreeNode left, TreeNode right) {

this.val = val;

this.left = left;

this.right = right;

}

}

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example 1: A valid BST

TreeNode validBST = new TreeNode(2, new TreeNode(1), new TreeNode(3));

System.out.println("Is the tree a valid BST? " + solution.isValidBST(validBST));

// Example 2: An invalid BST

TreeNode invalidBST = new TreeNode(5, new TreeNode(1), new TreeNode(4, new TreeNode(3), new TreeNode(6)));

System.out.println("Is the tree a valid BST? " + solution.isValidBST(invalidBST));

}

}

class Solution {

public boolean isValidBST(TreeNode root) {

if (root == null) return true;

return dfs(root, null, null);

}

private boolean dfs(TreeNode root, Integer min, Integer max) {

if (root == null) return true;

// Validate the current node

if ((min != null && root.val <= min) || (max != null && root.val >= max)) {

return false;

}

// Recursively validate the left and right subtrees

return dfs(root.left, min, root.val) && dfs(root.right, root.val, max);

}

}

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

* Every adjacent pair of words differs by a single letter.
* Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
* sk == 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.

import java.util.\*;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example input

String beginWord = "hit";

String endWord = "cog";

List<String> wordList = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");

System.out.println("Begin Word: " + beginWord);

System.out.println("End Word: " + endWord);

System.out.println("Word List: " + wordList);

// Find the length of the shortest transformation sequence

int result = solution.ladderLength(beginWord, endWord, wordList);

System.out.println("Length of shortest transformation sequence: " + result);

}

}

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

// Add all words to a set for quick lookup

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return 0; // If endWord is not in the list, no transformation is possible

}

// Initialize a queue for BFS

Queue<Pair> queue = new LinkedList<>();

queue.offer(new Pair(beginWord, 1)); // Start with beginWord and step count of 1

while (!queue.isEmpty()) {

Pair current = queue.poll();

String word = current.word;

int steps = current.steps;

// Try all possible transformations of the current word

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

char[] charArray = word.toCharArray();

for (char c = 'a'; c <= 'z'; c++) {

charArray[i] = c;

String nextWord = new String(charArray);

// If the transformed word is endWord, return the steps

if (nextWord.equals(endWord)) {

return steps + 1;

}

// If the transformed word is in the wordSet, add it to the queue

if (wordSet.contains(nextWord)) {

queue.offer(new Pair(nextWord, steps + 1));

wordSet.remove(nextWord); // Remove from set to mark as visited

}

}

}

}

return 0; // If no transformation sequence is found

}

}

// Helper class to store word and step count

class Pair {

String word;

int steps;

Pair(String word, int steps) {

this.word = word;

this.steps = steps;

}

}

A **transformation sequence** from word beginWord to word endWord using a dictionary wordList is a sequence of words beginWord -> s1 -> s2 -> ... -> sk such that:

* Every adjacent pair of words differs by a single letter.
* Every si for 1 <= i <= k is in wordList. Note that beginWord does not need to be in wordList.
* sk == endWord

Given two words, beginWord and endWord, and a dictionary wordList, return all the **shortest transformation sequences** from beginWord to endWord, or an empty list if no such sequence exists. Each sequence should be returned as a list of the words [beginWord, s1, s2, ..., sk].

**Example 1:**

**Input:** beginWord = "hit", endWord = "cog", wordList = ["hot","dot","dog","lot","log","cog"]

**Output:** [["hit","hot","dot","dog","cog"],["hit","hot","lot","log","cog"]]

**Explanation:** There are 2 shortest transformation sequences:

"hit" -> "hot" -> "dot" -> "dog" -> "cog"

"hit" -> "hot" -> "lot" -> "log" -> "cog"

import java.util.\*;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example input

String beginWord = "hit";

String endWord = "cog";

List<String> wordList = Arrays.asList("hot", "dot", "dog", "lot", "log", "cog");

System.out.println("Begin Word: " + beginWord);

System.out.println("End Word: " + endWord);

System.out.println("Word List: " + wordList);

// Find the length of the shortest transformation sequence

int result = solution.ladderLength(beginWord, endWord, wordList);

System.out.println("Length of shortest transformation sequence: " + result);

}

}

class Solution {

public int ladderLength(String beginWord, String endWord, List<String> wordList) {

// Add all words to a set for quick lookup

Set<String> wordSet = new HashSet<>(wordList);

if (!wordSet.contains(endWord)) {

return 0; // If endWord is not in the list, no transformation is possible

}

// Initialize a queue for BFS

Queue<Pair> queue = new LinkedList<>();

queue.offer(new Pair(beginWord, 1)); // Start with beginWord and step count of 1

while (!queue.isEmpty()) {

Pair current = queue.poll();

String word = current.word;

int steps = current.steps;

// Try all possible transformations of the current word

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

char[] charArray = word.toCharArray();

for (char c = 'a'; c <= 'z'; c++) {

charArray[i] = c;

String nextWord = new String(charArray);

// If the transformed word is endWord, return the steps

if (nextWord.equals(endWord)) {

return steps + 1;

}

// If the transformed word is in the wordSet, add it to the queue

if (wordSet.contains(nextWord)) {

queue.offer(new Pair(nextWord, steps + 1));

wordSet.remove(nextWord); // Remove from set to mark as visited

}

}

}

}

return 0; // If no transformation sequence is found

}

}

// Helper class to store word and step count

class Pair {

String word;

int steps;

Pair(String word, int steps) {

this.word = word;

this.steps = steps;

}

}

There are a total of numCourses courses you have to take, labeled from 0 to numCourses - 1. You are given an array prerequisites where prerequisites[i] = [ai, bi] indicates that you **must** take course bi first if you want to take course ai.

* For example, the pair [0, 1], indicates that to take course 0 you have to first take course 1.

Return true if you can finish all courses. Otherwise, return false.

**Example 1:**

**Input:** numCourses = 2, prerequisites = [[1,0]]

**Output:** true

**Explanation:** There are a total of 2 courses to take.

To take course 1 you should have finished course 0. So it is possible.

import java.util.\*;

public class Main {

public static void main(String[] args) {

Solution solution = new Solution();

// Example input

int numCourses = 4;

int[][] prerequisites = {

{1, 0},

{2, 1},

{3, 2}

};

System.out.println("Number of Courses: " + numCourses);

System.out.println("Prerequisites: " + Arrays.deepToString(prerequisites));

// Determine if all courses can be finished

boolean result = solution.canFinish(numCourses, prerequisites);

System.out.println("Can all courses be finished? " + result);

}

}

class Solution {

public boolean canFinish(int numCourses, int[][] prerequisites) {

// Build the adjacency list and compute in-degrees

List<List<Integer>> graph = new ArrayList<>();

int[] inDegree = new int[numCourses];

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

graph.add(new ArrayList<>());

}

for (int[] prereq : prerequisites) {

int course = prereq[0];

int pre = prereq[1];

graph.get(pre).add(course);

inDegree[course]++;

}

// Initialize the queue with courses having in-degree of 0

Queue<Integer> queue = new LinkedList<>();

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

if (inDegree[i] == 0) {

queue.offer(i);

}

}

int processedCourses = 0;

while (!queue.isEmpty()) {

int current = queue.poll();

processedCourses++;

for (int neighbor : graph.get(current)) {

inDegree[neighbor]--;

if (inDegree[neighbor] == 0) {

queue.offer(neighbor);

}

}

}

return processedCourses == numCourses;

}

}