Harshitha R AIDS A 22AD041 19/11/24

1. **Next Permutation**

A **permutation** of an array of integers is an arrangement of its members into a sequence or linear order.

* For example, for arr = [1,2,3], the following are all the permutations of arr: [1,2,3], [1,3,2], [2, 1, 3], [2, 3, 1], [3,1,2], [3,2,1].

The **next permutation** of an array of integers is the next lexicographically greater permutation of its integer. More formally, if all the permutations of the array are sorted in one container according to their lexicographical order, then the **next permutation** of that array is the permutation that follows it in the sorted container. If such arrangement is not possible, the array must be rearranged as the lowest possible order (i.e., sorted in ascending order).

* For example, the next permutation of arr = [1,2,3] is [1,3,2].
* Similarly, the next permutation of arr = [2,3,1] is [3,1,2].
* While the next permutation of arr = [3,2,1] is [1,2,3] because [3,2,1] does not have a lexicographical larger rearrangement.

Given an array of integers nums, *find the next permutation of* nums.

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

**Example 1:**

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

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

**Example 2:**

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

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

**Example 3:**

**Input:** nums = [1,1,5]

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

**Constraints:**

* 1 <= nums.length <= 100
* 0 <= nums[i] <= 100

**Code**

class Solution {

public void nextPermutation(int[] nums) {

int i = nums.length - 2;

while (i>=0 && nums[i] >= nums[i + 1]){

i--;

}

if (i != -1) {

int j= nums.length-1;

while (j>=0 && nums[i] >= nums[j]) {

j--;

}

swap(nums, i, j);

}

int start = i + 1;

int end = nums.length - 1;

while (start < end) {

swap(nums, start, end);

start++;

end--;

}

}

public static void swap(int[] nums, int a, int b) {

int temp = nums[a];

nums[a] = nums[b];

nums[b] = temp;

}

}



**Output**

**Time Complexity**

O(n)

1. Spiral Matrix

Given an m x n matrix, return *all elements of the* matrix *in spiral order*.

**Example 1:**

****

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

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

**Example 2:**

****

**Input:** matrix = [[1,2,3,4],[5,6,7,8],[9,10,11,12]]

**Output:** [1,2,3,4,8,12,11,10,9,5,6,7]

**Constraints:**

* m == matrix.length
* n == matrix[i].length
* 1 <= m, n <= 10
* -100 <= matrix[i][j] <= 100

**Code**

class Solution {

public List<Integer> spiralOrder(int[][] matrix) {

int top= 0;

int bottom = matrix.length-1;

int left = 0;

int right = matrix[0].length-1;

Integer[] arr = new Integer[matrix.length \* matrix[0].length];

int runningIndex= 0;

while(runningIndex!= arr.length){

for(int n = left; n <= right && (runningIndex!= arr.length); n++){

arr[runningIndex] = matrix[top][n];

runningIndex++;

}

for(int n = top; n <bottom && (runningIndex!= arr.length); n++){

arr[runningIndex] = matrix[n+1][right];

runningIndex++;

}

for(int n = right-1; n>= left && (runningIndex!= arr.length); n--){

arr[runningIndex]=matrix[bottom][n];

runningIndex++;

}

for(int n = bottom-1; n >= top+1 && (runningIndex!= arr.length); n--){

arr[runningIndex] = matrix[n][left];

runningIndex++;

}

top++;

bottom--;

left++;

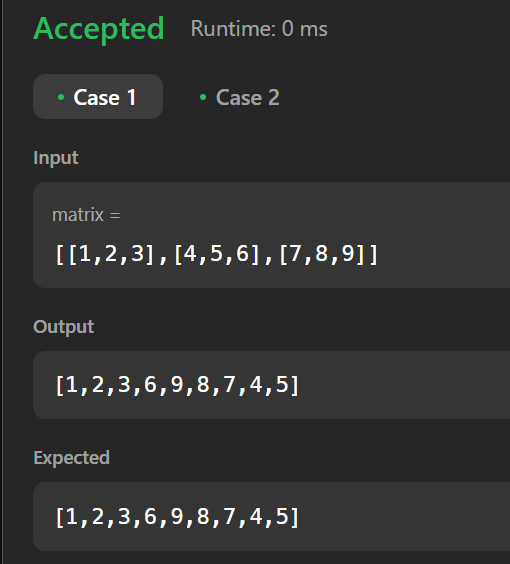
right--;

}

return Arrays.asList(arr);

}

}

**Output**

**Time Complexity**

O(n \* m)

1. Longest Substring without repeating characters

Given a string s, find the length of the **longest**

**substring**

without repeating characters.

**Example 1:**

**Input:** s = "abcabcbb"

**Output:** 3

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

**Example 2:**

**Input:** s = "bbbbb"

**Output:** 1

**Explanation:** The answer is "b", with the length of 1.

**Example 3:**

**Input:** s = "pwwkew"

**Output:** 3

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

Notice that the answer must be a substring, "pwke" is a subsequence and not a substring.

**Constraints:**

* 0 <= s.length <= 5 \* 104
* s consists of English letters, digits, symbols and spaces.

**Code**

import java.util.HashSet;

import java.util.Set;

class Solution {

public int lengthOfLongestSubstring(String s) {

int n=s.length();

Set<Character> charSet=new HashSet<>();

int l=0;

int r=0;

int maxLength=0;

while(r<n){

char currentChar=s.charAt(r);

if(!charSet.contains(currentChar))

{

charSet.add(currentChar);

r++;

maxLength=Math.max(maxLength,r-l);

}

else{

charSet.remove(s.charAt(l));

l++;

}

}

return maxLength;

}

public static void main(String[] args){

Solution solution=new Solution();

String s1="abcabcbb";

int result1=solution.lengthOfLongestSubstring(s1);

System.out.println(result1);

String s2="bbbbb";

int result2=solution.lengthOfLongestSubstring(s2);

System.out.println(result2);

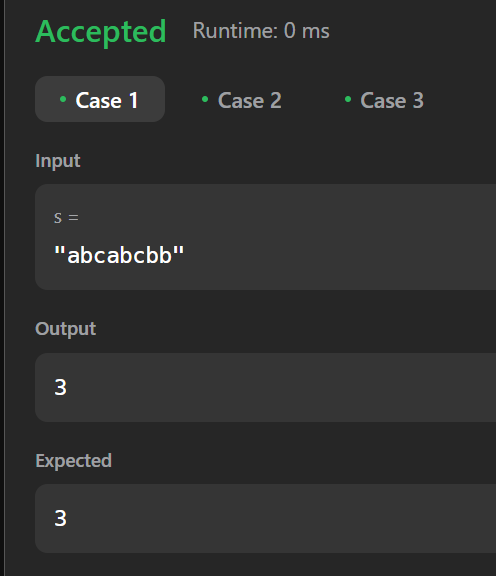
String s3="pwwkew";

int result3=solution.lengthOfLongestSubstring(s3);

System.out.println(result3);

}

}

**Output**

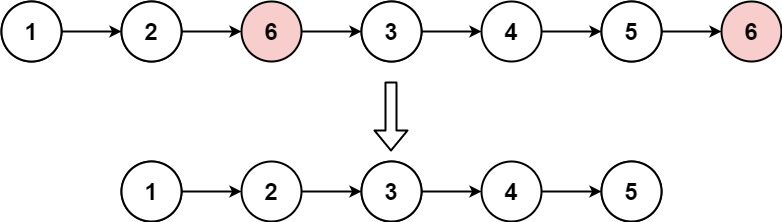
**Time Complexity**

O(n)

1. Remove linked list elements

Given the head of a linked list and an integer val, remove all the nodes of the linked list that has Node.val == val, and return *the new head*.

**Example 1:**

****

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

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

**Example 2:**

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

**Output:** []

**Example 3:**

**Input:** head = [7,7,7,7], val = 7

**Output:** []

**Constraints:**

* The number of nodes in the list is in the range [0, 104].
* 1 <= Node.val <= 50
* 0 <= val <= 50

**Code**

class Solution {

public ListNode removeElements(ListNode head, int val) {

ListNode temp = new ListNode(0) , curr = temp;

temp.next = head;

while(curr.next != null ){

if(curr.next.val == val) curr.next = curr.next.next;

else curr = curr.next;

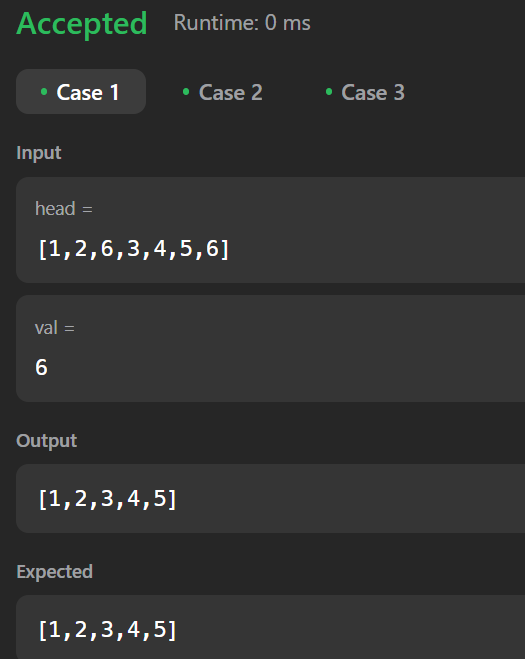
}

return temp.next;

}

}

**Output**



**Time Complexity**

O(n)

1. Palindrome Linked List

Given the head of a singly linked list, return true *if it is a*

*palindrome*

*or* false *otherwise*.

**Example 1:**

****

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

**Output:** true

**Example 2:**

****

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

**Output:** false

**Constraints:**

* The number of nodes in the list is in the range [1, 105].
* 0 <= Node.val <= 9

**Code**

class Solution {

public boolean isPalindrome(ListNode head) {

List<Integer> list = new ArrayList();

while(head != null) {

list.add(head.val);

head = head.next;

}

int left = 0;

int right = list.size()-1;

while(left < right && list.get(left) == list.get(right)) {

left++;

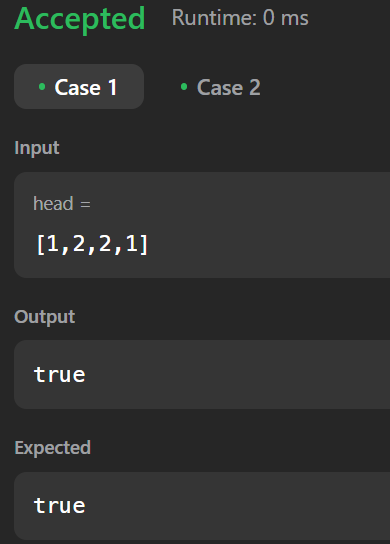
right--;

}

return left >= right;

}

}

**Output**

**Time Complexity**

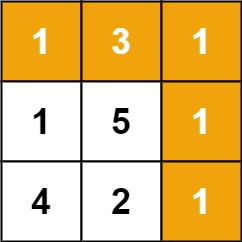
O(n)

1. Minimum path sum

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

**Constraints:**

* m == grid.length
* n == grid[i].length
* 1 <= m, n <= 200
* 0 <= grid[i][j] <= 200

**Code**

class Solution {

public int solve(int[][] dp, int[][] grid, int row, int col) {

if (row == 0 && col == 0) return grid[0][0];

if (row < 0 || col < 0) return Integer.MAX\_VALUE;

if (dp[row][col] != 0) return dp[row][col];

dp[row][col] = grid[row][col] + Math.min(solve(dp, grid, row - 1, col), solve(dp, grid, row, col - 1));

return dp[row][col];

}

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

int m = grid.length;

int n = grid[0].length;

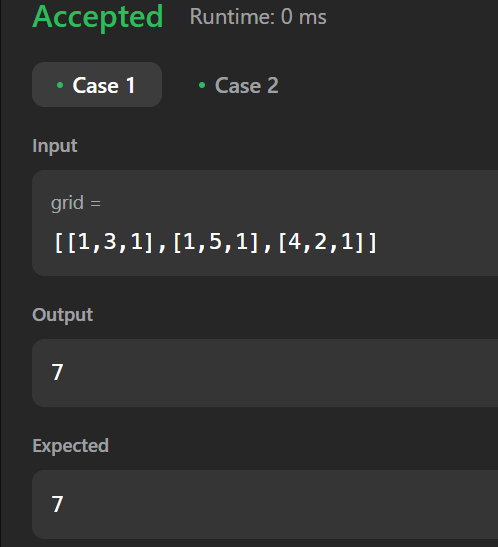
int[][] dp = new int[m + 1][n + 1];

return solve(dp, grid, m - 1, n - 1);

}

}

**Output**



**Time Complexity**

O(m \* n)

1. Validate binary search tree

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

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

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

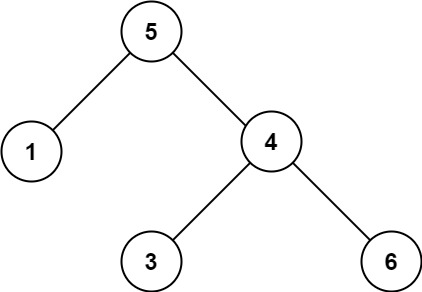
**Example 1:**

****

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

**Output:** true

**Example 2:**

****

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

**Output:** false

**Explanation:** The root node's value is 5 but its right child's value is 4.

**Constraints:**

* The number of nodes in the tree is in the range [1, 104].
* -231 <= Node.val <= 231 - 1

**Code**

class Solution {

public boolean isValidBST(TreeNode root) {

return valid(root, Long.MIN\_VALUE, Long.MAX\_VALUE);

}

private boolean valid(TreeNode node, long minimum, long maximum) {

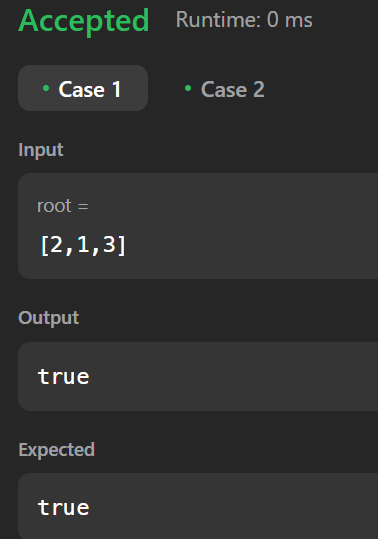
if (node == null) return true;

if (!(node.val > minimum && node.val < maximum)) return false;

return valid(node.left, minimum, node.val) && valid(node.right, node.val, maximum);

}

}

**Output**

**TIme Complexity**

O(n)

1. Word Ladder

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.

**Example 2:**

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

**Output:** 0

**Explanation:** The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

* 1 <= beginWord.length <= 10
* endWord.length == beginWord.length
* 1 <= wordList.length <= 5000
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the words in wordList are **unique**.

**Code**

class Solution {

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

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

Boolean isPresent = false;

wordSet.addAll(wordList);

for (String currWord : wordList) {

if (endWord.equals(currWord)) {

isPresent = true;

break;

}

}

if (!isPresent) return 0;

Queue<String> wordQueue = new LinkedList<>();

wordQueue.add(beginWord);

int distance = 0;

while (!wordQueue.isEmpty()) {

int size = wordQueue.size();

distance++;

while (size-- != 0) {

String currWord = wordQueue.poll();

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

char[] temp = currWord.toCharArray();

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

temp[i] = j;

String newWord = new String(temp);

if (newWord.equals(endWord)) return distance + 1;

if (wordSet.contains(newWord)) {

wordQueue.add(newWord);

wordSet.remove(newWord);

System.out.println(newWord);

}}}}}

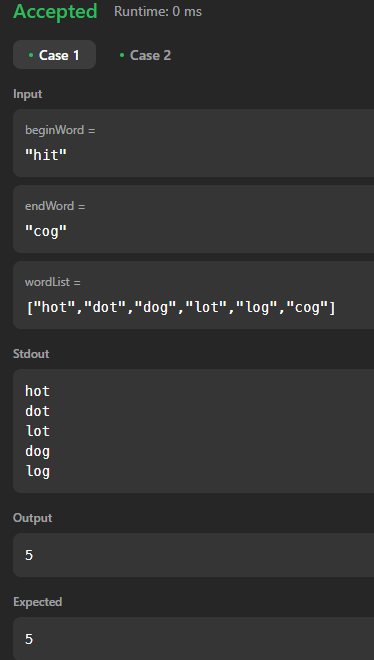
return 0;

}

}

**Time Complexity**

O(n \* m)

**Output**

1. Word Ladder - II

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"

**Example 2:**

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

**Output:** []

**Explanation:** The endWord "cog" is not in wordList, therefore there is no valid transformation sequence.

**Constraints:**

* 1 <= beginWord.length <= 5
* endWord.length == beginWord.length
* 1 <= wordList.length <= 500
* wordList[i].length == beginWord.length
* beginWord, endWord, and wordList[i] consist of lowercase English letters.
* beginWord != endWord
* All the words in wordList are **unique**.
* The **sum** of all shortest transformation sequences does not exceed 105.

**Code**

class Solution {

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

Map<String,Integer> hm = new HashMap<>();

List<List<String>> res = new ArrayList<>();

Queue<String> q = new LinkedList<>();

q.add(beginWord);

hm.put(beginWord,1);

HashSet<String> hs = new HashSet<>();

for(String w : wordList) hs.add(w);

hs.remove(beginWord);

while(!q.isEmpty()){

String word = q.poll();

if(word.equals(endWord)){

break;

}

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

int level = hm.get(word);

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

char[] replaceChars = word.toCharArray();

replaceChars[i] = ch;

String replaceString = new String(replaceChars);

if(hs.contains(replaceString)){

q.add(replaceString);

hm.put(replaceString,level+1);

hs.remove(replaceString);

}

}

}

}

if(hm.containsKey(endWord) == true){

List<String> seq = new ArrayList<>();

seq.add(endWord);

dfs(endWord,seq,res,beginWord,hm);

}

return res;

}

public void dfs(String word,List<String> seq,List<List<String>> res,String beginWord,Map<String,Integer> hm){

if(word.equals(beginWord)){

List<String> ref = new ArrayList<>(seq);

Collections.reverse(ref);

res.add(ref);

return;

}

int level = hm.get(word);

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

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

char replaceChars[] = word.toCharArray();

replaceChars[i] = ch;

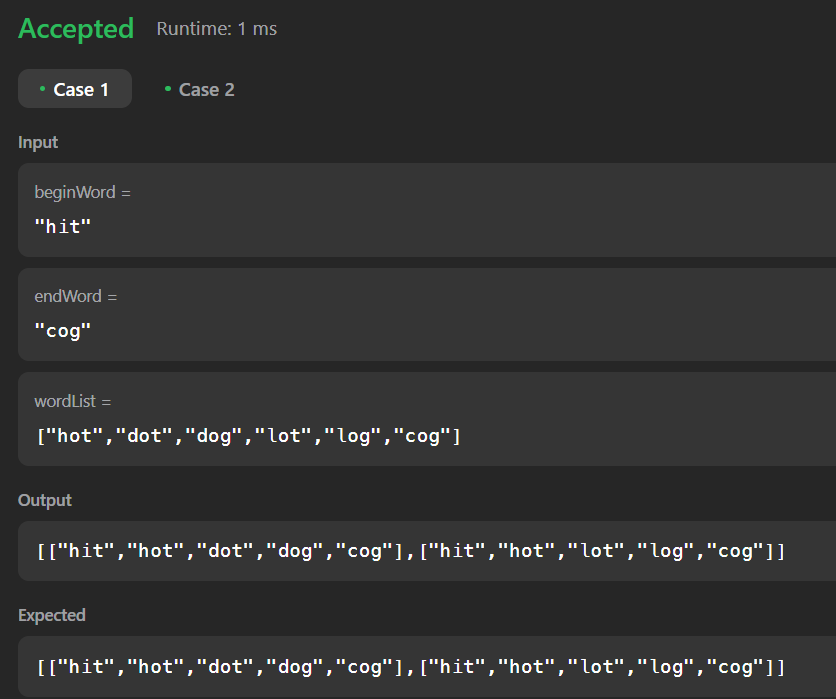
String replaceStr = new String(replaceChars);

if(hm.containsKey(replaceStr) && hm.get(replaceStr) == level-1){

seq.add(replaceStr);

dfs(replaceStr,seq,res,beginWord,hm);

seq.remove(seq.size()-1);}}}}}

**Output**

**Time Complexity**

O(m \* n)

1. Course Schedule

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.

**Example 2:**

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

**Output:** false

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

To take course 1 you should have finished course 0, and to take course 0 you should also have finished course 1. So it is impossible.

**Constraints:**

* 1 <= numCourses <= 2000
* 0 <= prerequisites.length <= 5000
* prerequisites[i].length == 2
* 0 <= ai, bi < numCourses
* All the pairs prerequisites[i] are **unique**.

**Code**

class Solution {

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

int counter = 0;

if (numCourses <= 0) {

return true;

}

int[] inDegree = new int[numCourses];

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

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

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

}

for (int[] edge : prerequisites) {

int parent = edge[1];

int child = edge[0];

graph.get(parent).add(child);

inDegree[child]++;

}

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

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

if (inDegree[i] == 0) {

sources.offer(i);

}

}

while (!sources.isEmpty()) {

int course = sources.poll();

counter++;

for (int child : graph.get(course)) {

inDegree[child]--;

if (inDegree[child] == 0) {

sources.offer(child);

}

}

}

return counter == numCourses;

}

}

**Output**

**Time Complexity**

O(v + e)

1. Valid Tic - Tac - Toe State

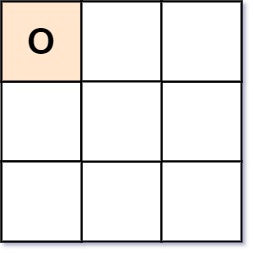
Given a Tic-Tac-Toe board as a string array board, return true if and only if it is possible to reach this board position during the course of a valid tic-tac-toe game.

The board is a 3 x 3 array that consists of characters ' ', 'X', and 'O'. The ' ' character represents an empty square.

Here are the rules of Tic-Tac-Toe:

* Players take turns placing characters into empty squares ' '.
* The first player always places 'X' characters, while the second player always places 'O' characters.
* 'X' and 'O' characters are always placed into empty squares, never filled ones.
* The game ends when there are three of the same (non-empty) character filling any row, column, or diagonal.
* The game also ends if all squares are non-empty.
* No more moves can be played if the game is over.

**Example 1:**

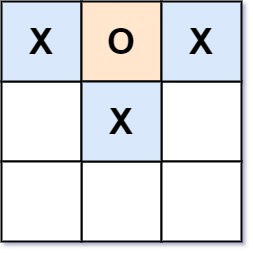
****

**Input:** board = ["O "," "," "]

**Output:** false

**Explanation:** The first player always plays "X".

**Example 2:**

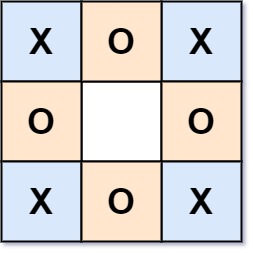
****

**Input:** board = ["XOX"," X "," "]

**Output:** false

**Explanation:** Players take turns making moves.

**Example 3:**

****

**Input:** board = ["XOX","O O","XOX"]

**Output:** true

**Constraints:**

* board.length == 3
* board[i].length == 3
* board[i][j] is either 'X', 'O', or ' '.

**Code**

class Solution {

public boolean validTicTacToe(String[] board) {

if (board == null || board.length == 0) {

return false;

}

int n = board.length;

int diag1 = 0;

int diag2 = 0;

int turnsDiff = 0;

boolean xWin = false;

boolean oWin = false;

char c;

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

int row = 0;

int col = 0;

for (int j = 0; j < n; j++) {

c = board[i].charAt(j);

if (c == 'X') {

row++;

turnsDiff++;

} else if (c == 'O') {

row--;

turnsDiff--;

}

c = board[j].charAt(i);

if (c == 'X') {

col++;

} else if (c == 'O') {

col--;

}

}

c = board[i].charAt(i);

if (c == 'X') {

diag1++;

} else if (c == 'O') {

diag1--;

}

c = board[i].charAt(n - 1 - i);

if (c == 'X') {

diag2++;

} else if (c == 'O') {

diag2--;

}

if (row == n || col == n || diag1 == n || diag2 == n) {

if (oWin) {

return false;

}

xWin = true;

}

if (row == -n || col == -n || diag1 == -n || diag2 == -n) {

if (xWin) {

return false;

}

oWin = true;

}

}

if (turnsDiff < 0 || turnsDiff > 1) {

return false;

}

if ((turnsDiff == 0 && xWin) || (turnsDiff == 1 && oWin)) {

return false;

}

return true;

}

}

**Output**

**Time Complexity**

O(n^2)