1522. Diameter of N-Ary Tree Premium

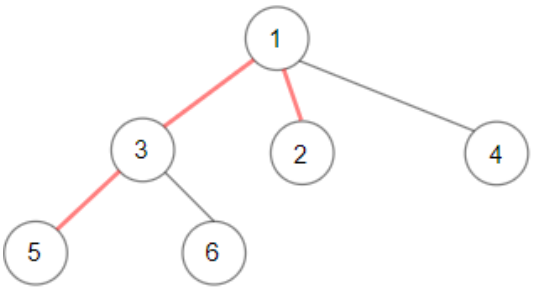
Medium Topics Companies Hint

Given a `root` of an `N-ary tree`, you need to compute the length of the diameter of the tree.

The diameter of an N-ary tree is the length of the **longest** path between any two nodes in the tree. This path may or may not pass through

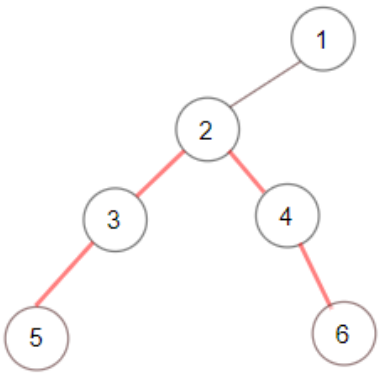
(*Nary-Tree input serialization is represented in their level order traversal, each group of children is separated by the null value.)*

Example 1:



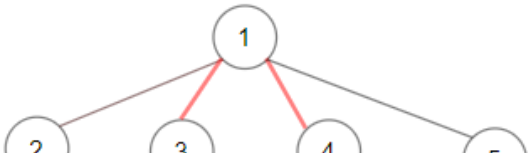

```
Input: root = [1,null,3,2,4,null,5,6]
Output: 3
Explanation: Diameter is shown in red color.
```

Example 2:




```
Input: root = [1,null,2,null,3,4,null,5,null,6]
Output: 4
```

Example 3:



Code

Java
Auto

```java
// Definition for a Node.
class Node {
    public int val;
    public List<Node> children;
    public Node() {
        children = new ArrayList<Node>();
    }
    public Node(int _val) {
        val = _val;
        children = new ArrayList<Node>();
    }

    public Node(int _val,ArrayList<Node> _children) {
        val = _val;
        children = _children;
    }
};
*/

class Solution {

// In this solution, we are exploring all nodes of the input N-Ary Tree using Depth-First Search.

// Length of the longest path passing through a node = 1st Max Height among N children + 2nd Max Height among N childr

// Thus, Diameter of the entire tree = Max(Length of the longest path passing through each node)

    public int diameter(Node root) {
        if (root == null || root.children.size() == 0) {
            return 0;
        }

        int[] maxDiameter = new int[1];
        diameterHelper(root, maxDiameter);
        return maxDiameter[0];
    }

    private int diameterHelper(Node root, int[] maxDiameter) {
        if (root.children.size() == 0) {
            return 0;
        }

        // Setting below maximums to -1 helps in the case if there is only one child
        // node of this root node.
        int maxHeight1 = -1;
        int maxHeight2 = -1;
```

Restored from Cloud

Testcase
Test Result

Case 1
Case 2
Case 3

```
[1,null,3,2,4,null,5,6]
```

Code

Java
Auto

```java
class Solution {

// In this solution, we are exploring all nodes of the input N-Ary Tree using Depth-First Search.

// Length of the longest path passing through a node = 1st Max Height among N children + 2nd Max Height among N childr

// Thus, Diameter of the entire tree = Max(Length of the longest path passing through each node)

    public int diameter(Node root) {
        if (root == null || root.children.size() == 0) {
            return 0;
        }

        int[] maxDiameter = new int[1];
        diameterHelper(root, maxDiameter);
        return maxDiameter[0];
    }

    private int diameterHelper(Node root, int[] maxDiameter) {
        if (root.children.size() == 0) {
            return 0;
        }

        // Setting below maximums to -1 helps in the case if there is only one child
        // node of this root node.
        int maxHeight1 = -1;
        int maxHeight2 = -1;

        for (Node child : root.children) {
            int childHeight = diameterHelper(child, maxDiameter);
            if (childHeight > maxHeight1) {
                maxHeight2 = maxHeight1;
                maxHeight1 = childHeight;
            } else if (childHeight > maxHeight2) {
                maxHeight2 = childHeight;
            }
        }

        maxDiameter[0] = Math.max(maxDiameter[0], maxHeight1 + maxHeight2 + 2);
        return maxHeight1 + 1;
    }

}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

[1,null,3,2,4,null,5,6]

1570. Dot Product of Two Sparse Vectors Premium

Medium Topics Companies Hint

Given two sparse vectors, compute their dot product.

Implement class `SparseVector`:

- `SparseVector(nums)` Initializes the object with the vector `nums`
- `dotProduct(vec)` Compute the dot product between the instance of *SparseVector* and `vec`

A **sparse vector** is a vector that has mostly zero values, you should store the sparse vector **efficiently** and compute the dot product betwee

Follow up: What if only one of the vectors is sparse?

Example 1:

```
Input: nums1 = [1,0,0,2,3], nums2 = [0,3,0,4,0]
Output: 8
Explanation: v1 = SparseVector(nums1) , v2 = SparseVector(nums2)
v1.dotProduct(v2) = 1*0 + 0*3 + 0*0 + 2*4 + 3*0 = 8
```

Example 2:

```
Input: nums1 = [0,1,0,0,0], nums2 = [0,0,0,0,2]
Output: 0
Explanation: v1 = SparseVector(nums1) , v2 = SparseVector(nums2)
v1.dotProduct(v2) = 0*0 + 1*0 + 0*0 + 0*0 + 0*2 = 0
```

Example 3:

```
Input: nums1 = [0,1,0,0,2,0,0], nums2 = [1,0,0,0,3,0,4]
Output: 6
```

Constraints:

- `n == nums1.length == nums2.length`
- `1 <= n <= 10^5`
- `0 <= nums1[i], nums2[i] <= 100`

Seen this question in a real interview before? 1/4

Yes No

Accepted 232.5K Submissions 258.5K Acceptance Rate 89.9%

Code

Java
Auto

```java
    private TreeMap<Integer, Integer> map = new TreeMap<>();

    public TreeMap<Integer, Integer> getMap() {
      return map;
    }

    SparseVector(int[] nums) {
      for (int i=0; i<nums.length; i++) {
        map.put(i, nums[i]);
      }
    }

  // Return the dotProduct of two sparse vectors
    public int dotProduct(SparseVector vec) {
      TreeMap<Integer, Integer> map1 = getMap();
      TreeMap<Integer, Integer> map2 = vec.getMap();
      if (map1.size() == 0 || map2.size() == 0) return 0;
      int ret = 0;

      var it1 = map1.keySet().iterator();
      var it2 = map1.keySet().iterator();
      int key1 = it1.next();
      int key2 = it2.next();

      while(true) {
        if (key1 == key2) {
          ret += map1.get(key1) * map2.get(key2);
          if (!it1.hasNext()) break;
          key1 = it1.next();
          if (!it2.hasNext()) break;
          key2 = it2.next();
          continue;
        }

        if (key1 < key2) {
          if (!it1.hasNext()) break;
          key1 = it1.next();
          continue;
        }

        if (!it2.hasNext()) break;
        key2 = it2.next();

      }
      return ret;
    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

[1,0,0,2,3]

[0,3,0,4,0]

Code

Java
Auto

```
}

// Your SparseVector object will be instantiated and called as such:
// SparseVector v1 = new SparseVector(nums1);
// SparseVector v2 = new SparseVector(nums2);
// int ans = v1.dotProduct(v2);


// class SparseVector {

//     List<int[]> list;

//     SparseVector(int[] nums) {
//         list = new ArrayList<>();

//         for(int i=0; i<nums.length; i++){
//             if(nums[i] != 0)
//                 list.add(new int[]{i, nums[i]});
//         }
//     }

//  // Return the dotProduct of two sparse vectors
//     public int dotProduct(SparseVector vec) {
//         int dotProduct = 0;
//         int p = 0, q = 0;

//         while(p < list.size() && q < vec.list.size()) {

//             if(list.get(p)[0] == vec.list.get(q)[0]){
//                 dotProduct += list.get(p)[1] * vec.list.get(q)[1];
//                 p++;
//                 q++;
//             }
//             else if(list.get(p)[0] < vec.list.get(q)[0])
//                 p++;
//             else
//                 q++;
//         }


//         return dotProduct;
//     }
// }
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

```
[1,0,0,2,3]

[0,3,0,4,0]
```

1586. Binary Search Tree Iterator II Premium

Medium Topics Companies Hint

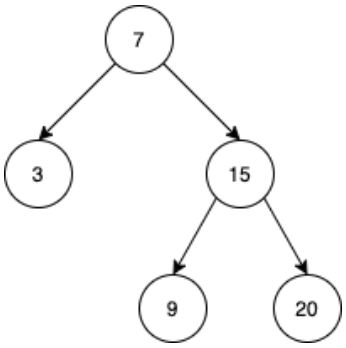
Implement the `BSTIterator` class that represents an iterator over the **in-order traversal** of a binary search tree (BST):

- `BSTIterator(TreeNode root)` Initializes an object of the `BSTIterator` class. The `root` of the BST is given as part of the constructor. Th be initialized to a non-existent number smaller than any element in the BST.
- `boolean hasNext()` Returns `true` if there exists a number in the traversal to the right of the pointer, otherwise returns `false`.
- `int next()` Moves the pointer to the right, then returns the number at the pointer.
- `boolean hasPrev()` Returns `true` if there exists a number in the traversal to the left of the pointer, otherwise returns `false`.
- `int prev()` Moves the pointer to the left, then returns the number at the pointer.

Notice that by initializing the pointer to a non-existent smallest number, the first call to `next()` will return the smallest element in the BST.

You may assume that `next()` and `prev()` calls will always be valid. That is, there will be at least a next/previous number in the in-order tra `next()` / `prev()` is called.

Example 1:




```
Input
["BSTIterator", "next", "next", "prev", "next", "hasNext", "next", "next", "next", "hasNext", "hasPr
"prev"]
[[[7, 3, 15, null, null, 9, 20]], [null], [null], [null], [null], [null], [null], [null], [null], [r
[null], [null]]
Output
[null, 3, 7, 3, 7, true, 9, 15, 20, false, true, 15, 9]

Explanation
// The underlined element is where the pointer currently is.
BSTIterator bSTIterator = new BSTIterator([7, 3, 15, null, null, 9, 20]); // state is _ [3, 7, 9, 15
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 3
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 7
bSTIterator.prev(); // state becomes [3, 7, 9, 15, 20], return 3
bSTIterator.next(); // state becomes [3, 7, 9, 15, 20], return 7
bSTIterator.hasNext(); // return true
```

```
    Stack<TreeNode> nextStack = new Stack<>();
    Stack<TreeNode> prevStack = new Stack<>();
    Set<TreeNode> visited = new HashSet<>();

    public BSTIterator(TreeNode root) {
        fillInNextStack(root);
    }

    public boolean hasNext() {
        return !nextStack.isEmpty();
    }

    public int next() {
        TreeNode node = nextStack.pop();
        if(!visited.contains(node) && node.right != null) {
            fillInNextStack(node.right);
        }
        visited.add(node);
        prevStack.push(node);
        return node.val;
    }

    public boolean hasPrev() {
        return !prevStack.isEmpty() && prevStack.size() > 1;
    }

    public int prev() {
        nextStack.push(prevStack.pop());
        return prevStack.peek().val;
    }

    private void fillInNextStack(TreeNode node) {
        while(node != null) {
            nextStack.push(node);
            node = node.left;
        }
    }
}
```

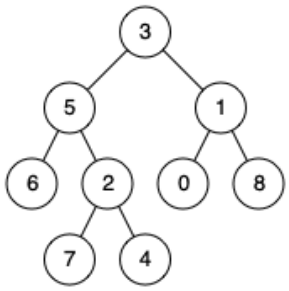
1644. Lowest Common Ancestor of a Binary Tree II Premium

Medium Topics Companies Hint

Given the `root` of a binary tree, return *the lowest common ancestor (LCA) of two given nodes,* `p` *and* `q`. If either node `p` or `q` **does not exi** `null`. All values of the nodes in the tree are **unique**.

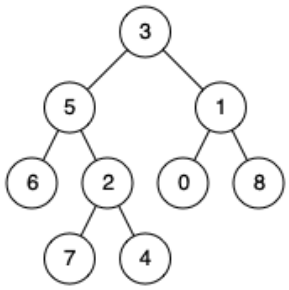
According to the **definition of LCA on Wikipedia**: "The lowest common ancestor of two nodes `p` and `q` in a binary tree `T` is the lowest n and `q` as **descendants** (where we allow **a node to be a descendant of itself**)". A **descendant** of a node `x` is a node `y` that is on the path some leaf node.

Example 1:



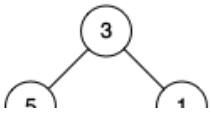

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5. A node can be a descendant of itself according to the de
LCA.
```

Example 3:



Code

Java
Auto

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    boolean foundP = false;
    boolean foundQ = false;

    public TreeNode lowestCommonAncestor(TreeNode root, TreeNode p, TreeNode q) {
      TreeNode ret = lca(root, p, q);
      if (!foundP || !foundQ) return null;
      return ret;
    }
    public TreeNode lca(TreeNode root, TreeNode p, TreeNode q) {
      if(root==null)    return root;
      TreeNode left=lca(root.left, p, q);
      TreeNode right=lca(root.right, p, q);

      if (root == p) {
        foundP = true;
        return root;
      }
      if (root == q) {
        foundQ = true;
        return root;
      }

      return left == null ? right : right == null ? left : root;
    }

}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

[3,5,1,6,2,0,8,null,null,7,4]

5

1650. Lowest Common Ancestor of a Binary Tree III Premium

Medium Topics Companies Hint

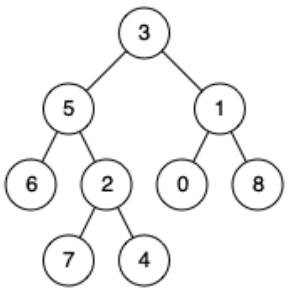
Given two nodes of a binary tree `p` and `q`, return *their lowest common ancestor (LCA)*.

Each node will have a reference to its parent node. The definition for `Node` is below:

```
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node parent;
}
```

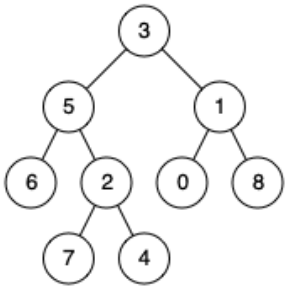
According to the **definition of LCA on Wikipedia**: "The lowest common ancestor of two nodes p and q in a tree T is the lowest node that h
descendants (where we allow **a node to be a descendant of itself**)."

Example 1:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 1
Output: 3
Explanation: The LCA of nodes 5 and 1 is 3.
```

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 4
Output: 5
Explanation: The LCA of nodes 5 and 4 is 5 since a node can be a descendant of itself according to t
definition
```

Description
Solutions
Submissions
Editorial

Code

Java
Auto

```java
/*
// Definition for a Node.
class Node {
    public int val;
    public Node left;
    public Node right;
    public Node parent;
};
*/

class Solution {
    public Node lowestCommonAncestor(Node p, Node q) {
      Node a = p, b = q;
      while (a != b) {
        a = a == null? q : a.parent;
        b = b == null? p : b.parent;
      }
      return a;
    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

[3,5,1,6,2,0,8,null,null,7,4]

5

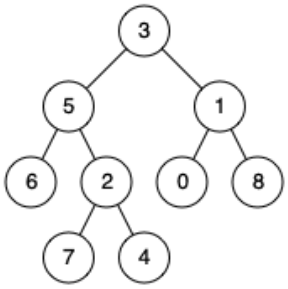
1740. Find Distance in a Binary Tree Premium

Medium Topics Companies Hint

Given the root of a binary tree and two integers `p` and `q`, return *the **distance** between the nodes of value* `p` *and value* `q` *in the tree.*

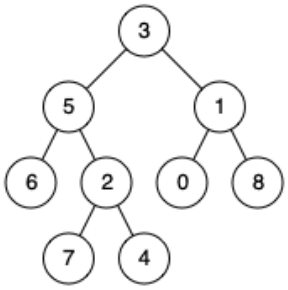
The **distance** between two nodes is the number of edges on the path from one to the other.

Example 1:



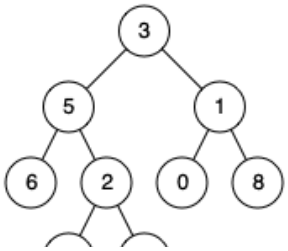

```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 0
Output: 3
Explanation: There are 3 edges between 5 and 0: 5-3-1-0.
```

Example 2:




```
Input: root = [3,5,1,6,2,0,8,null,null,7,4], p = 5, q = 7
Output: 2
Explanation: There are 2 edges between 5 and 7: 5-2-7.
```

Example 3:



Code

Java Auto

```java
/**
 * Definition for a binary tree node.
 * public 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;
 *     }
 * }
 */
class Solution {
    class Distance {
      Distance(int toP, int toQ, int pToQ) {
        this.toP = toP;
        this.toQ = toQ;
        this.pToQ = pToQ;
      }
      int toP;
      int toQ;
      int pToQ;
    };

    public int findDistance(TreeNode root, int p, int q) {
      Distance ret = processNode(root, p, q);
      return ret.pToQ;
    }

    private Distance processNode(TreeNode node, int p, int q) {
      int toP = Integer.MAX_VALUE;
      int toQ = Integer.MAX_VALUE;
      int pToQ = Integer.MAX_VALUE;
      if (node == null) {
        return new Distance(toP, toQ, pToQ);
      }

      Distance distLeft = processNode(node.left, p, q);
      if (distLeft.pToQ < Integer.MAX_VALUE ) {
        return new Distance(-1, -1, distLeft.pToQ);
      }
      Distance distRight = processNode(node.right, p, q);
      if (distRight.pToQ < Integer.MAX_VALUE ) {
        return new Distance(-1, -1, distRight.pToQ);
```

Saved to cloud

Testcase Test Result

Case 1 Case 2 Case 3

root =

[3,5,1,6,2,0,8,null,null,7,4]

Code

Java
Auto

```java

    public int findDistance(TreeNode root, int p, int q) {
      Distance ret = processNode(root, p, q);
      return ret.pToQ;
    }

    private Distance processNode(TreeNode node, int p, int q) {
      int toP = Integer.MAX_VALUE;
      int toQ = Integer.MAX_VALUE;
      int pToQ = Integer.MAX_VALUE;
      if (node == null) {
        return new Distance(toP, toQ, pToQ);
      }

      Distance distLeft = processNode(node.left, p, q);
      if (distLeft.pToQ < Integer.MAX_VALUE ) {
        return new Distance(-1, -1, distLeft.pToQ);
      }
      Distance distRight = processNode(node.right, p, q);
      if (distRight.pToQ < Integer.MAX_VALUE ) {
        return new Distance(-1, -1, distRight.pToQ);
      }

      if (node.val == p) {
        toP = 0;
      } else {
        toP = Math.min(distLeft.toP, distRight.toP);
        if (toP < Integer.MAX_VALUE )
          toP++;
      }


      if (node.val == q) {
        toQ = 0;
      } else {
        toQ = Math.min(distLeft.toQ, distRight.toQ);
        if (toQ < Integer.MAX_VALUE )
          toQ++;
      }

      if (toP < Integer.MAX_VALUE && toQ < Integer.MAX_VALUE) {
        pToQ = toP + toQ;
      }

      return new Distance(toP, toQ, pToQ);

    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

root =

[3,5,1,6,2,0,8,null,null,7,4]

1762. Buildings With an Ocean View Premium

Medium Topics Companies Hint

There are `n` buildings in a line. You are given an integer array `heights` of size `n` that represents the heights of the buildings in the line.

The ocean is to the right of the buildings. A building has an ocean view if the building can see the ocean without obstructions. Formally, a b ocean view if all the buildings to its right have a **smaller** height.

Return a list of indices **(0-indexed)** of buildings that have an ocean view, sorted in increasing order.

Example 1:

```
Input: heights = [4,2,3,1]
Output: [0,2,3]
Explanation: Building 1 (0-indexed) does not have an ocean view because building 2 is taller.
```

Example 2:

```
Input: heights = [4,3,2,1]
Output: [0,1,2,3]
Explanation: All the buildings have an ocean view.
```

Example 3:

```
Input: heights = [1,3,2,4]
Output: [3]
Explanation: Only building 3 has an ocean view.
```

Constraints:

- `1 <= heights.length <= 10^5`
- `1 <= heights[i] <= 10^9`

Seen this question in a real interview before? 1/4

Yes No

Accepted 190.2K Submissions 239.6K Acceptance Rate 79.4%

Topics

Companies

Hint 1

Code

Java
Auto

```java
class Solution {
    public int[] findBuildings(int[] heights) {
          Deque<Integer> stack = new ArrayDeque<>();
       // Assume that the first building can see the ocean
          stack.push(0);
       // Walk through list of buildings
          for(int i = 1; i<heights.length;i++){
         // If the height of the current building is taller than whats in the stack
         // it needs to be the first building in the stack
              while(!stack.isEmpty() && heights[i] >= heights[stack.peek()]){
                  stack.pop();
              }
           // We know that we have the next tallest building in the input array
                  stack.push(i);
          }

       //Our stack now contains only the buildings that have a view of the ocean and we need to return it in the approp
          int[] result = new int[stack.size()];
          int n = stack.size();
          for(int i = n-1; i>=0; i--){
              result[i] = stack.pop();
          }
          return result;
    }
}

    // public int[] findBuildings(int[] heights) {
    //     List<Integer> ls = new ArrayList<>();
    //     int last = Integer.MIN_VALUE;
    //     for (int i = heights.length - 1; i >= 0; i--) {
    //         if (heights[i] > last) {
    //             ls.add(i);
    //             last = heights[i];
    //         }
    //     }

    //     int index = 0;
    //     int[] res = new int[ls.size()];
    //     for (int i = ls.size() - 1; i >= 0; i--)
    //         res[index++] = ls.get(i);

    //     return res;
    // }
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

heights =

[4,2,3,1]

1778. Shortest Path in a Hidden Grid Premium

Medium Topics Companies Hint

This is an **interactive problem**.

There is a robot in a hidden grid, and you are trying to get it from its starting cell to the target cell in this grid. The grid is of size `m x n`, an grid is either empty or blocked. It is **guaranteed** that the starting cell and the target cell are different, and neither of them is blocked.

You want to find the minimum distance to the target cell. However, you **do not know** the grid's dimensions, the starting cell, nor the target allowed to ask queries to the `GridMaster` object.

Thr `GridMaster` class has the following functions:

- `boolean canMove(char direction)` Returns `true` if the robot can move in that direction. Otherwise, it returns `false`.
- `void move(char direction)` Moves the robot in that direction. If this move would move the robot to a blocked cell or off the grid, the **ignored**, and the robot will remain in the same position.
- `boolean isTarget()` Returns `true` if the robot is currently on the target cell. Otherwise, it returns `false`.

Note that `direction` in the above functions should be a character from `{'U','D','L','R'}`, representing the directions up, down, left, ar

Return *the **minimum distance** between the robot's initial starting cell and the target cell. If there is no valid path between the cells, return* `-1`

Custom testing:

The test input is read as a 2D matrix `grid` of size `m x n` where:

- `grid[i][j] == -1` indicates that the robot is in cell `(i, j)` (the starting cell).
- `grid[i][j] == 0` indicates that the cell `(i, j)` is blocked.
- `grid[i][j] == 1` indicates that the cell `(i, j)` is empty.
- `grid[i][j] == 2` indicates that the cell `(i, j)` is the target cell.

There is exactly one `-1` and `2` in `grid`. Remember that you will **not** have this information in your code.

Example 1:

```
Input: grid = [[1,2],[-1,0]]
Output: 2
Explanation: One possible interaction is described below:
The robot is initially standing on cell (1, 0), denoted by the -1.
- master.canMove('U') returns true.
- master.canMove('D') returns false.
- master.canMove('L') returns false.
- master.canMove('R') returns false.
- master.move('U') moves the robot to the cell (0, 0).
- master.isTarget() returns false.
- master.canMove('U') returns false.
- master.canMove('D') returns true.
```

Java | DFS+BFS with explanations

Lasgana
162 3394 Mar 05, 2021

Intuition

Since the problem is asking for the `minimal distance` between start and target, it is natural to want to apply BFS directly to solve the problem. After all, if you are able to locate target with a simple BFS search, you would get the distance for free right away.

However, the issue is that we are not able explore the grid freely and are reliant on the robot pointer's `canMove()`. If we insist on using BFS, we would need to move the robot pointer back and forth between each `queue.poll()`. It is managable for a small grid, but we will run into TLE when the grid becomes bigger.

So we fall back to DFS to locate `target`. And once we locate `target`, we would still need to apply BFS to get minimal distance between the two nodes. Again, the same problem applies, as we know using robot pointer to run BFS will cause TLE.

The way out here is to explore the whole map while running DFS. If we know how the grid looks, we do not need the robot pointer. To do that, we do not return `target` immediately even if we found it while traversing during DFS, and we keep exploring the 4 directions for each intermediate cell.

Implementation Notes

- We create a board of `[1000][1000]` and start from the very middle of it, so the board is big enough to accommodate any input combinations.

```
class Solution {

    private static final int BLOCKED = -1;
    private static final int UNEXPLORED = 0;
    private static final int PATH = 1;
    private static final int TARGET = 2;
    private static final int START = 3;

    public int findShortestPath(GridMaster master) {
        int m = 500;
        int[][] board = new int[m*2][m*2];
        int[] start = new int[] {m, m};
        board[m][m] = START;
        int[] target = findTarget(master, board, m, m);
        return target == null ? -1 : getDistance(board, target, start);
```

```java
private static final int BLOCKED = -1;
private static final int UNEXPLORED = 0;
private static final int PATH = 1;
private static final int TARGET = 2;
private static final int START = 3;

public int findShortestPath(GridMaster master) {
    int m = 500;
    int[][] board = new int[m*2][m*2];
    int[] start = new int[] {m, m};
    board[m][m] = START;
    int[] target = findTarget(master, board, m, m);
    return target == null ? -1 : getDistance(board, target, start);
}

// dfs
private int[] findTarget(GridMaster master, int[][] board, int r, int c) {
    if (master.isTarget()) {
        board[r][c] = TARGET;
        return new int[]{r,c};
    }

    int[] res = null;
    for (Direction d : Direction.values()) {
        int r2 = r + d.n[0];
        int c2 = c + d.n[1];
        if (board[r2][c2] == UNEXPLORED) {
            if (master.canMove(d.c)) {
                master.move(d.c);
                board[r2][c2] = PATH;
                int[] target = findTarget(master, board, r2, c2);
                if (target != null)
                    res = target;
                master.move(d.o);
            } else {
                board[r2][c2] = BLOCKED;
            }
        }
    }
    return res;
}
```

```
// bfs
private int getDistance(int[][] board, int[] target, int[] start) {
    Queue<int[]> queue = new ArrayDeque<>();
    queue.add(target);
    int distance = 1;
    while (!queue.isEmpty()) {
        int size = queue.size();
        for (int i = 0; i < size; i++) {
            int[] cur = queue.poll();
            for (Direction d : Direction.values()) {
                int r2 = cur[0] + d.n[0];
                int c2 = cur[1] + d.n[1];
                if (board[r2][c2] == START)
                    return distance;
                else if (board[r2][c2] == PATH) {
                    board[r2][c2] = BLOCKED;
                    queue.add(new int[]{r2,c2});
                }
            }
        }
        distance++;
    }

    return distance;
}

enum Direction {
    UP('U', 'D', new int[]{0,1}),
    DOWN('D', 'U', new int[]{0,-1}),
    LEFT('L', 'R', new int[]{-1,0}),
    RIGHT('R', 'L', new int[]{1,0});

    public char c;
    public char o;
    public int[] n;

    Direction(char c, char o, int[] n) {
        this.c = c;
        this.o = o;
        this.n = n;
    }
}
```

```
                            queue.add(new int[]{r2,c2});
                        }
                    }
                }
                distance++;
            }

            return distance;
        }

        enum Direction {
            UP('U', 'D', new int[]{0,1}),
            DOWN('D', 'U', new int[]{0,-1}),
            LEFT('L', 'R', new int[]{-1,0}),
            RIGHT('R', 'L', new int[]{1,0});

            public char c;
            public char o;
            public int[] n;

            Direction(char c, char o, int[] n) {
                this.c = c;
                this.o = o;
                this.n = n;
            }
        }
    }
```

Next
[Python] DSF to explore the graph and BFS to find minimum distance

Comments (13)

Sort by: Best

Type comment here... (Markdown supported)

Preview Comment

chenzuojing

Mar 14, 2021

1810. Minimum Path Cost in a Hidden Grid Premium

Medium Topics Companies Hint

This is an **interactive problem**.

There is a robot in a hidden grid, and you are trying to get it from its starting cell to the target cell in this grid. The grid is of size `m x n`, an blocked. It is **guaranteed** that the starting cell and the target cell are different, and neither of them is blocked.

Each cell has a **cost** that you need to pay each time you **move** to the cell. The starting cell's cost is **not** applied before the robot moves.

You want to find the minimum total cost to move the robot to the target cell. However, you **do not know** the grid's dimensions, the startin allowed to ask queries to the `GridMaster` object.

The `GridMaster` class has the following functions:

- `boolean canMove(char direction)` Returns `true` if the robot can move in that direction. Otherwise, it returns `false`.
- `int move(char direction)` Moves the robot in that direction and returns the cost of moving to that cell. If this move would move the r the move will be **ignored**, the robot will remain in the same position, and the function will return `-1`.
- `boolean isTarget()` Returns `true` if the robot is currently on the target cell. Otherwise, it returns `false`.

Note that `direction` in the above functions should be a character from `{'U','D','L','R'}`, representing the directions up, down, left, ar

Return *the **minimum total cost** to get the robot from its initial starting cell to the target cell. If there is no valid path between the cells, return*

Custom testing:

The test input is read as a 2D matrix `grid` of size `m x n` and four integers `r1`, `c1`, `r2`, and `c2` where:

- `grid[i][j] == 0` indicates that the cell `(i, j)` is blocked.
- `grid[i][j] >= 1` indicates that the cell `(i, j)` is empty and `grid[i][j]` is the **cost** to move to that cell.
- `(r1, c1)` is the starting cell of the robot.
- `(r2, c2)` is the target cell of the robot.

Remember that you will **not** have this information in your code.

Example 1:

```
Input: grid = [[2,3],[1,1]], r1 = 0, c1 = 1, r2 = 1, c2 = 0
Output: 2
Explanation: One possible interaction is described below:
The robot is initially standing on cell (0, 1), denoted by the 3.
- master.canMove('U') returns false.
- master.canMove('D') returns true.
- master.canMove('L') returns true.
- master.canMove('R') returns false.
- master.move('L') moves the robot to the cell (0, 0) and returns 2.
- master.isTarget() returns false.
```

59 1430 Mar 31, 2021

Same as 1778, use dfs to explore the grid first. Then use PQ to find the shortest path.

```
int[][] dxys = new int[][] {{-1, 0}, {1, 0}, {0, -1}, {0, 1}};
char[] dirs = new char[] {'U', 'D', 'L', 'R'};

public int findShortestPath(GridMaster master) {
    int[][] grid = new int[200][200];
    for (int i = 0; i < 200; i++) {
        for (int j = 0; j < 200; j++) {
            grid[i][j] = -1;
        }
    }
    grid[99][99] = 0;
    int[] target = new int[] {-100, -100};
    fillGrid(master, 99, 99, grid, target);
    boolean[][] visited = new boolean[200][200];
    PriorityQueue<int[]> pq = new PriorityQueue<>((a, b) -> {
        if (a[2] == b[2]) {
            return 0;
        }
        return a[2] < b[2] ? -1 : 1;
    });
    pq.offer(new int[] {99, 99, 0});
    while (!pq.isEmpty()) {
        int[] cur = pq.poll();
        int row = cur[0];
        int col = cur[1];
        int cost = cur[2];
        if (row == target[0] && col == target[1]) {
            return cost;
        }
        if (visited[row][col]) {
            continue;
        }
        visited[row][col] = true;
        for (int[] dxy : dxys) {
            int nextRow = row + dxy[0];
            int nextCol = col + dxy[1];
            if (nextRow < 0 || nextRow >= 200 || nextCol < 0 || nextCol >= 200 || vis
                continue;
            }
```

```
                }
                if (visited[row][col]) {
                    continue;
                }
                visited[row][col] = true;
                for (int[] dxy : dxys) {
                    int nextRow = row + dxy[0];
                    int nextCol = col + dxy[1];
                    if (nextRow < 0 || nextRow >= 200 || nextCol < 0 || nextCol >= 200 || vis
                        continue;
                    }
                    int nextCost = cost + grid[nextRow][nextCol];
                    pq.offer(new int[] {nextRow, nextCol, nextCost});
                }
            }
            return -1;
        }

        private void fillGrid(GridMaster master, int row, int col, int[][] grid, int[] target

            if (master.isTarget()) {
                target[0] = row;
                target[1] = col;
            }

            for (int i = 0; i < 4; i++) {
                char ch = dirs[i];
                int[] dxy = dxys[i];
                int nr = row + dxy[0];
                int nc = col + dxy[1];
                if (master.canMove(ch) && grid[nr][nc] == -1) {
                    int val = master.move(ch);
                    grid[nr][nc] = val;
                    fillGrid(master, nr, nc, grid, target);
                    if (i == 0 || i == 1) {
                        master.move(dirs[1 - i]);
                    } else {
                        master.move(dirs[5 - i]);
                    }
                }
            }
        }
```

1826. Faulty Sensor Premium

Easy Topics Companies Hint

An experiment is being conducted in a lab. To ensure accuracy, there are **two** sensors collecting data simultaneously. You are given two arra
`sensor1[i]` and `sensor2[i]` are the i^{th} data points collected by the two sensors.

However, this type of sensor has a chance of being defective, which causes **exactly one** data point to be dropped. After the data is droppec
dropped data are **shifted** one place to the left, and the last data point is replaced with some **random value**. It is guaranteed that this randc
dropped value.

- For example, if the correct data is `[1,2,3,4,5]` and `3` is dropped, the sensor could return `[1,2,4,5,7]` (the last position can be **any** va

We know that there is a defect in **at most one** of the sensors. Return *the sensor number (`1` or `2`) with the defect. If there is **no defect** in eith
determine the defective sensor, return* `-1`.

Example 1:

```
Input: sensor1 = [2,3,4,5], sensor2 = [2,1,3,4]
Output: 1
Explanation: Sensor 2 has the correct values.
The second data point from sensor 2 is dropped, and the last value of sensor 1 is replaced by a 5.
```

Example 2:

```
Input: sensor1 = [2,2,2,2,2], sensor2 = [2,2,2,2,5]
Output: -1
Explanation: It is impossible to determine which sensor has a defect.
Dropping the last value for either sensor could produce the output for the other sensor.
```

Example 3:

```
Input: sensor1 = [2,3,2,2,3,2], sensor2 = [2,3,2,3,2,7]
Output: 2
Explanation: Sensor 1 has the correct values.
The fourth data point from sensor 1 is dropped, and the last value of sensor 1 is replaced by a 7.
```

Constraints:

- `sensor1.length == sensor2.length`
- `1 <= sensor1.length <= 100`
- `1 <= sensor1[i], sensor2[i] <= 100`

Seen this question in a real interview before? 1/4

Code

Java
Auto

```java
class Solution {

// Think about the normal case
// 1: [3 2]
// 2: [2 5]
// We can find 2 is defect.
// That is because 2 == 2 && 3 != 5.

// And think about the more advance case.
// 1: [2 3 2]
// 2: [3 2 7]
// In this case 2 == 2 && 3 == 3, we can't distinguish which is the defect.
// We skip the first time comparation, and continue to next time comparation. In the next time comparation, then we ca

    public int badSensor(int[] sensor1, int[] sensor2) {
      for (int i = 0; i < sensor1.length - 1; ++i) {
        if (sensor1[i] == sensor2[i]) {
          continue;
        }
        if (sensor1[i] == sensor2[i+1] && sensor2[i] != sensor1[i+1]) {
          return 1;
        }
        if (sensor2[i] == sensor1[i+1] && sensor1[i] != sensor2[i+1]) {
          return 2;
        }

      }
      return -1;

    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

Description Solutions Submissions Editorial

1868. Product of Two Run-Length Encoded Arrays Premium

Medium Topics Companies Hint

Run-length encoding is a compression algorithm that allows for an integer array `nums` with many segments of **consecutive repeated** nun smaller) 2D array `encoded`. Each `encoded[i] = [val_i, freq_i]` describes the `i^th` segment of repeated numbers in `nums` where `val_i` is th

- For example, `nums = [1,1,1,2,2,2,2,2]` is represented by the **run-length encoded** array `encoded = [[1,3],[2,5]]`. Another way to five `2`'s".

The **product** of two run-length encoded arrays `encoded1` and `encoded2` can be calculated using the following steps:

1. **Expand** both `encoded1` and `encoded2` into the full arrays `nums1` and `nums2` respectively.
2. Create a new array `prodNums` of length `nums1.length` and set `prodNums[i] = nums1[i] * nums2[i]`.
3. **Compress** `prodNums` into a run-length encoded array and return it.

You are given two **run-length encoded** arrays `encoded1` and `encoded2` representing full arrays `nums1` and `nums2` respectively. Both `nums` Each `encoded1[i] = [val_i, freq_i]` describes the `i^th` segment of `nums1`, and each `encoded2[j] = [val_j, freq_j]` describes the `j^th` se

Return *the **product** of* `encoded1` *and* `encoded2`.

Note: Compression should be done such that the run-length encoded array has the **minimum** possible length.

Example 1:

```
Input: encoded1 = [[1,3],[2,3]], encoded2 = [[6,3],[3,3]]
Output: [[6,6]]
Explanation: encoded1 expands to [1,1,1,2,2,2] and encoded2 expands to [6,6,6,3,3,3].
prodNums = [6,6,6,6,6,6], which is compressed into the run-length encoded array [[6,6]].
```

Example 2:

```
Input: encoded1 = [[1,3],[2,1],[3,2]], encoded2 = [[2,3],[3,3]]
Output: [[2,3],[6,1],[9,2]]
Explanation: encoded1 expands to [1,1,1,2,3,3] and encoded2 expands to [2,2,2,3,3,3].
prodNums = [2,2,2,6,9,9], which is compressed into the run-length encoded array [[2,3],[6,1],[9,2]].
```

Constraints:

- `1 <= encoded1.length, encoded2.length <= 10^5`
- `encoded1[i].length == 2`
- `encoded2[j].length == 2`
- `1 <= val_i, freq_i <= 10^4` for each `encoded1[i]`.
- `1 <= val_j, freq_j <= 10^4` for each `encoded2[j]`.
- The full arrays that [illegible] represent are the same length

Code

Java
Auto

```java
class Solution {
    public List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {
        int i1 = 0, i2 = 0;
        int f1 = 0, f2 = 0;
        int p = 0;
        int len1 = encoded1.length, len2 = encoded2.length;
        List<List<Integer>> ret = new ArrayList<>();
        while (i1 < len1 || i2 < len2) {
            f1 = encoded1[i1][1];
            f2 = encoded2[i2][1];
            p = encoded1[i1][0] * encoded2[i2][0];
            int newF;
            if (f1 == f2) {
                newF = f1;
                i1++;
                i2++;
            } else if (f1 < f2) {
                newF = f1;
                encoded2[i2][1] = f2 - f1;
                i1++;
            } else {
                newF = f2;
                encoded1[i1][1] = f1 - f2;
                i2++;
            }
            if (ret.size() > 0 && ret.get(ret.size() - 1).get(0) == p ) {
              int oldF = ret.get(ret.size() - 1).get(1);
              ret.get(ret.size() - 1).set(1, oldF + newF);
            } else {
              ret.add(Arrays.asList(p, newF));
            }

        }
        return ret;

    }


}



//out of memory
    // public List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {
    //   var data1 = decode(encoded1);
    //   var data2 = decode(encoded2);
    //   var productData = product(data1, data2);
    //   return encode(productData);
    // }

    // private List<Integer> decode(int[][] encoded) {
    //   List<Integer> ret = new ArrayList<>();
    //   for (var item : encoded) {
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2

Code

Java
Auto

```
    }


}



//out of memory
    // public List<List<Integer>> findRLEArray(int[][] encoded1, int[][] encoded2) {
    //   var data1 = decode(encoded1);
    //   var data2 = decode(encoded2);
    //   var productData = product(data1, data2);
    //   return encode(productData);
    // }

    // private List<Integer> decode(int[][] encoded) {
    //   List<Integer> ret = new ArrayList<>();
    //   for (var item : encoded) {
    //     for (int i = 0; i < item[1]; i++) {
    //       ret.add(item[0]);
    //     }
    //   }
    //   return ret;
    // }

    // private List<List<Integer>> encode(List<Integer> data) {
    //   List<List<Integer>> ret = new ArrayList<>();
    //   int idx = 0;
    //   while (idx < data.size()) {
    //     int val = data.get(idx);
    //     int cnt = 1;
    //     while ( idx + 1 < data.size() && data.get(idx + 1) == val) {
    //       idx++;
    //       cnt++;
    //     }
    //     ret.add(Arrays.asList(val, cnt));
    //     idx++;
    //   }
    //   return ret;
    // }

    // private List<Integer> product(List<Integer> l1, List<Integer> l2) {
    //   List<Integer> ret = new ArrayList<>();
    //   for (int i = 0; i < l1.size(); i++) {
    //     ret.add(l1.get(i) * l2.get(i));
    //   }
    //   return ret;
    // }
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2

1891. Cutting Ribbons Premium

Medium Topics Companies Hint

You are given an integer array `ribbons`, where `ribbons[i]` represents the length of the `i^th` ribbon, and an integer `k`. You may cut any of segments of **positive integer** lengths, or perform no cuts at all.

- For example, if you have a ribbon of length `4`, you can:
  - Keep the ribbon of length `4`,
  - Cut it into one ribbon of length `3` and one ribbon of length `1`,
  - Cut it into two ribbons of length `2`,
  - Cut it into one ribbon of length `2` and two ribbons of length `1`, or
  - Cut it into four ribbons of length `1`.

Your goal is to obtain `k` ribbons of all the **same positive integer length**. You are allowed to throw away any excess ribbon as a result of cu

Return *the **maximum** possible positive integer length that you can obtain `k` ribbons of, or `0` if you cannot obtain `k` ribbons of the same leng*

Example 1:

```
Input: ribbons = [9,7,5], k = 3
Output: 5
Explanation:
- Cut the first ribbon to two ribbons, one of length 5 and one of length 4.
- Cut the second ribbon to two ribbons, one of length 5 and one of length 2.
- Keep the third ribbon as it is.
Now you have 3 ribbons of length 5.
```

Example 2:

```
Input: ribbons = [7,5,9], k = 4
Output: 4
Explanation:
- Cut the first ribbon to two ribbons, one of length 4 and one of length 3.
- Cut the second ribbon to two ribbons, one of length 4 and one of length 1.
- Cut the third ribbon to three ribbons, two of length 4 and one of length 1.
Now you have 4 ribbons of length 4.
```

Example 3:

```
Input: ribbons = [5,7,9], k = 22
Output: 0
Explanation: You cannot obtain k ribbons of the same positive integer length.
```

Constraints:

Code

Java
Auto

```java
class Solution {
    public int maxLength(int[] ribbons, int k) {
        int l = 1;
        int r = (int) 1e5 + 1;
        while (l < r) {
            int mid = l + ((r - l) >> 1);

            if (!isCutPossible(ribbons, mid, k)) {
                r = mid;
            } else {
                l = mid + 1;
            }
        }
        return l - 1;
    }

   public boolean isCutPossible(int[] ribbons, int length, int k) {
        int count = 0;
        for (int ribbon: ribbons) {
            count += (ribbon / length);
        } // I could've written an early 'return' here to save some computation, but for me, the more "if", the more l
        return count >= k;
    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3

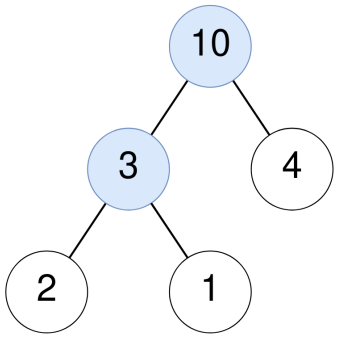
1973. Count Nodes Equal to Sum of Descendants Premium

Medium Topics Companies Hint

Given the `root` of a binary tree, return *the number of nodes where the value of the node is equal to the **sum** of the values of its descendants*.

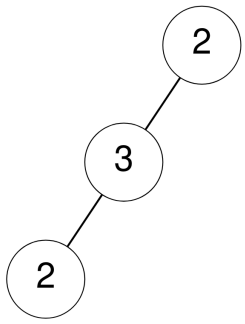
A **descendant** of a node `x` is any node that is on the path from node `x` to some leaf node. The sum is considered to be `0` if the node has

Example 1:




```
Input: root = [10,3,4,2,1]
Output: 2
Explanation:
For the node with value 10: The sum of its descendants is 3+4+2+1 = 10.
For the node with value 3: The sum of its descendants is 2+1 = 3.
```

Example 2:




```
Input: root = [2,3,null,2,null]
Output: 0
Explanation:
No node has a value that is equal to the sum of its descendants.
```

Example 3:

Code

Java
Auto

```java
/**
 * Definition for a binary tree node.
 * public 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;
 *     }
 * }
 */
class Solution {
    public int equalToDescendants(TreeNode root) {
      int[] ret = new int[1];
      sumOfNode(root, ret);
      return ret[0];
    }

    private int sumOfNode(TreeNode node, int[] ret) {
      if (node == null) return 0;
      int leftSum = sumOfNode(node.left, ret);
      int rightSum = sumOfNode(node.right, ret);
      if (node.val == leftSum + rightSum) {
        ret[0]++;
      }
      return node.val + leftSum + rightSum;
    }
}
```

Saved to cloud

Testcase
Test Result

Case 1
Case 2
Case 3