Week-2

Archana Goli

Tree:

A tree is a type of hierarchical data structure where nodes are connected by edges. The topmost node is called the root node. All other nodes of it branch out from a root node. There can be zero or more child nodes for every node.

Components of a tree:

Root node: topmost node in a tree.

Parent node: A node which has one or more child nodes.
Child node: A node that descends from another node.
Leaf Node: A node which does not have any child nodes.

Internal node: A node which has atleast one child node. This includes all the nodes except the root and leaf node.
Subtree: A portion of tree that consists of node and all its descendants.

Binary Tree:

A Binary tree is a hierarchical data structure which has atmost 2 children. One left child and one right child.

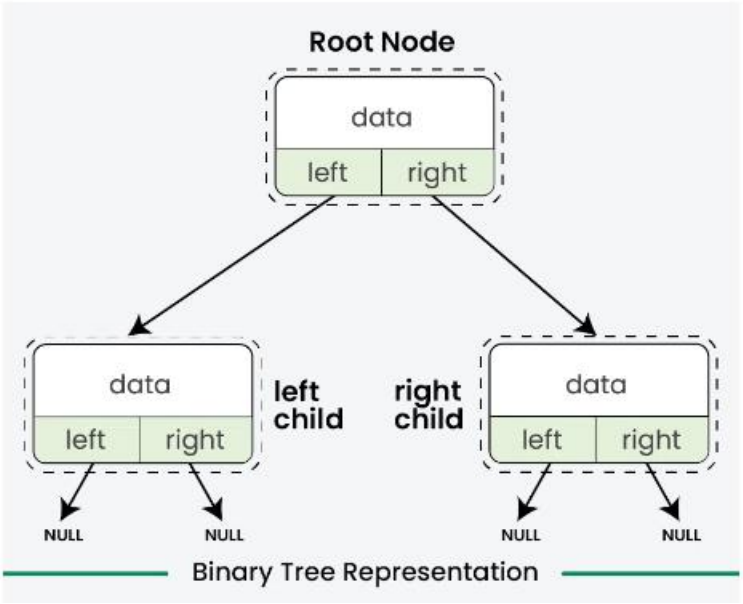
The topmost node in a binary tree is called the root node, and the bottom most nodes are called the leaf nodes.

Representation of a Binary tree:

Each node has 3 parts

- The data
- Pointer to the left node

- Pointer to the right node



Depth of a node: The number of edges from the specific node to the root node.

Height of Binary tree: The number of nodes from the deepest leaf node to the root node.

Properties of Binary Tree:

- In a binary tree, 2L is the maximum number of nodes at level L.
- In a binary tree of height H, the maximum number of nodes is 2H – 1.
- In a binary tree, the total number of leaf nodes equals the sum of the nodes with two children plus one.
- The lowest height or lowest number of levels in a Binary Tree with N nodes is equal to Log2(N+1).
- There are at least | Log2L |+ 1 levels in a binary tree with L leaves.

Types of Binary Trees:

Full Binary Tree: There are either 0 or 2 children for each node.

Complete Binary Tree: All levels are fully filled except possibly the last level, which is filled from left to right.

Perfect Binary Tree: Every leaf node is at the same level, and every internal node has two children.

Balanced Binary Tree: This tree has a maximum height difference of one between each node's left and right subtrees.

Operations on Binary Tree:

Traversal in Binary Tree:

Traversal in Binary Tree involves visiting all the nodes of the binary tree. Tree Traversal algorithms can be classified broadly into two categories, DFS and BFS:

Depth-First Search (DFS) algorithms: DFS explores as far down a branch as possible before backtracking. It is implemented using recursion. The main traversal methods in DFS for binary trees are:

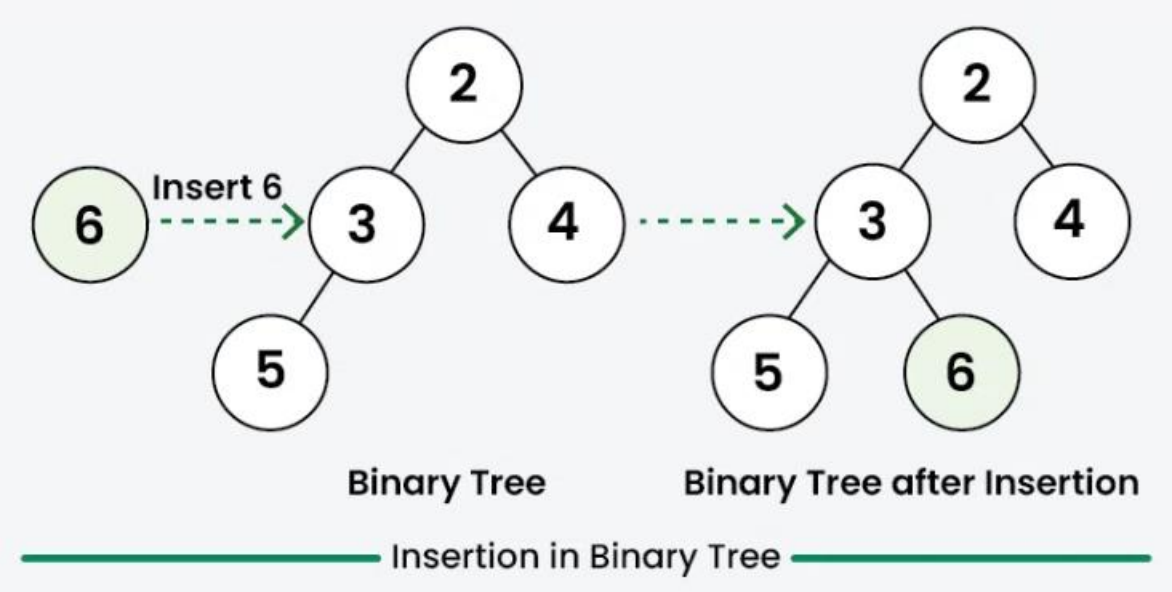
- **Preorder Traversal (current-left-right):** Node visited first followed by left sub tree and then right sub tree.
- **Inorder Traversal (left-current-right):** Left subtree is visited first then followed by the node and then the right subtree.
- **Postorder Traversal (left-right-current):** Left subtree is visited first, then followed by the right sub tree and then the node.

Breadth-First Search (BFS) algorithms: BFS investigates every node at the current depth before going on to the nodes at the subsequent depth level. Usually, a queue is used to accomplish that.

Insertion in Binary Tree:

In a binary tree, inserting elements entails adding a new node. We don't have to think about where values should go because binary trees don't have a set order for their nodes.

The process of insertion:
First Node: if tree is empty, we create the root node
Subsequent Nodes: We search for an empty space at the root level and proceed level by level with each additional element.
Insert Position: We search every node for a left or right empty child. The new node is placed there once we locate an empty space, usually starting from the left. The left child is automatically verified for insertion first.



Searching in Binary Tree:

In binary tree, searching involves finding a node that contains a value. We can use any traversal method since binary tree does not follow any specific order.

Common search methods:

Depth-First Search (DFS): Start at the root and explore as deep as possible before backtracking.

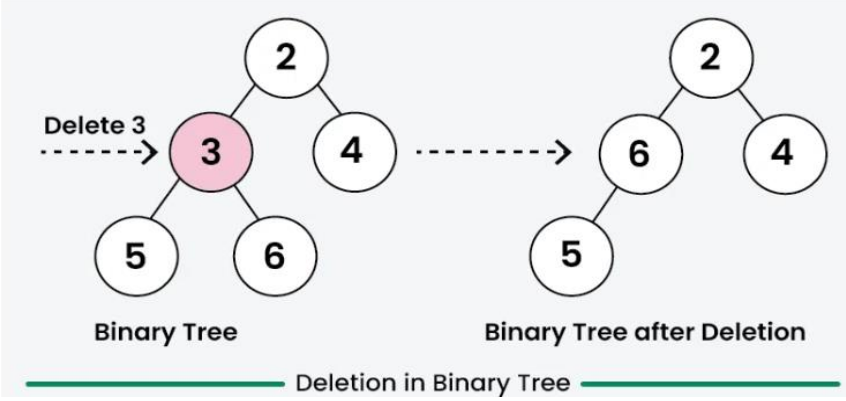
Breadth-First Search (BFS): Explore all nodes at the current level before moving to the next.

Deletion in Binary Tree:

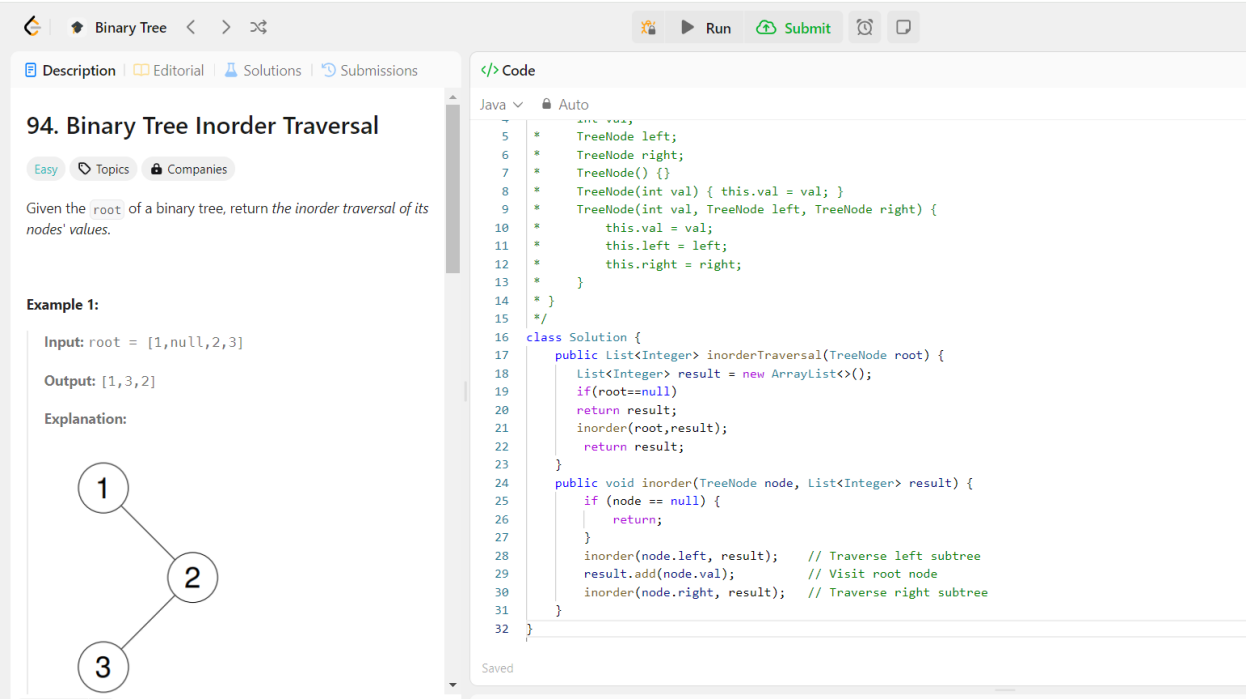
In a binary tree, deleting a node entails taking out that particular node while maintaining the tree's structure.

Procedures for removing:
find the node: Utilizing any traversal method, look for the node you wish to delete.
Replace the node: When the node is located, substitute its value with the value of the last node in the tree, the rightmost leaf.
Delete the last node: After replacing, take off the leaf node farthest to the right.
In this manner the structure of the tree is preserved. Furthermore, to prevent problems, deal with unusual instances such as deleting from an empty tree.

The primary objective is to guarantee that the tree maintains its correct structure even after being deleted.



Binary Tree Inorder Traversal:



Since the root is the starting point of a tree, the inorderTraversal() method takes the root node as an argument.

List result is initialized to store the values of visited nodes in the order.

If root is null then empty result list is returned.

The inorder() helper function is invoked by passing the root node and list to handle recursive traversal.

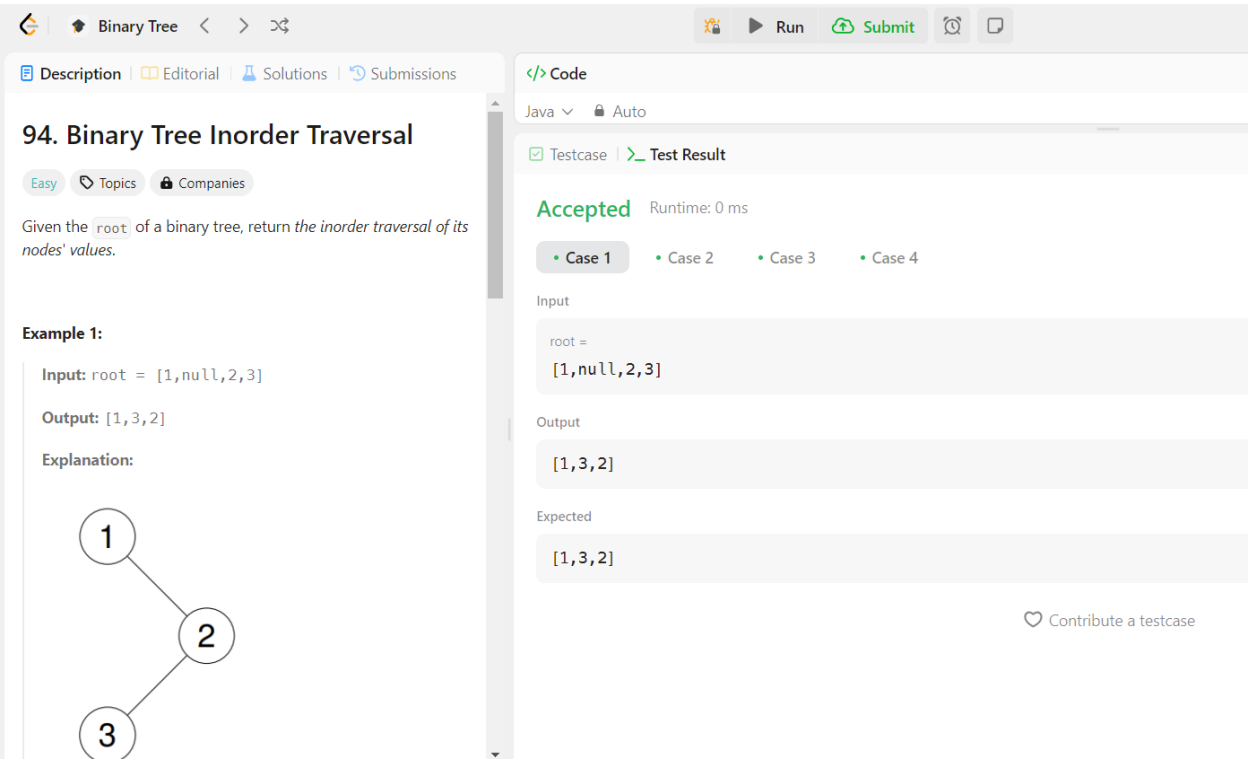
It checks if the current node is null, if it is null then it returns immediately.

It recursively calls itself to traverse the left subtree first.

After returning from the left sub tree, it adds current nodes value to the result list.

Finally, it recursively calls itself to traverse the right subtree.

Time complexity: The time complexity is O(n) since each node is visited once.



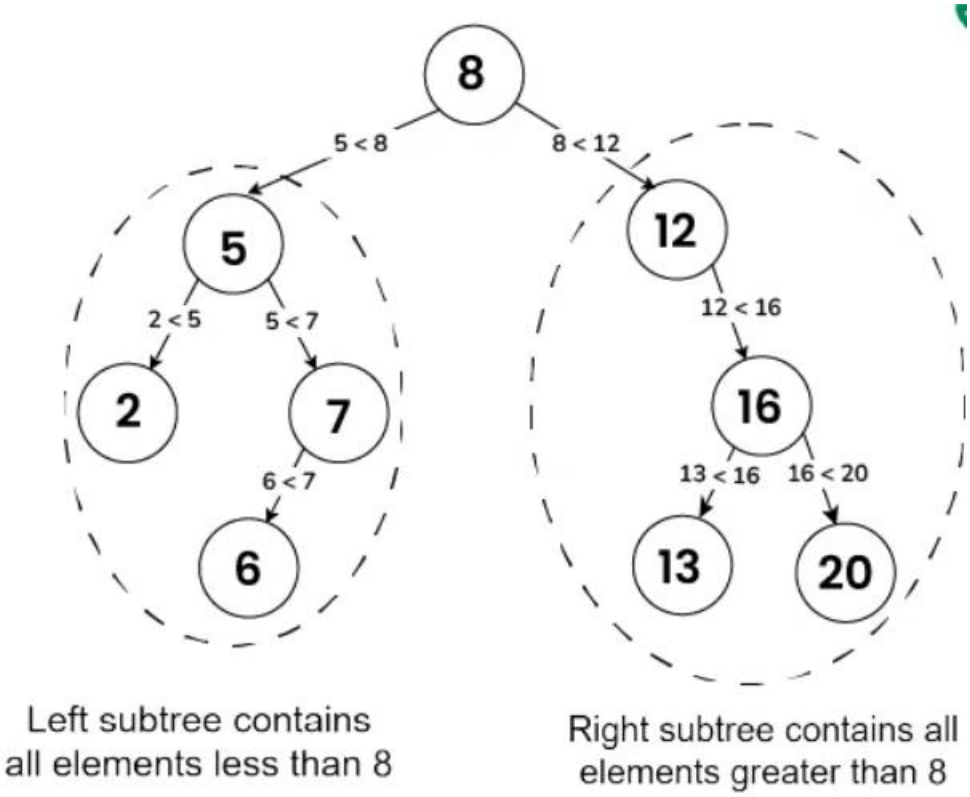
Binary Search Tree:

A data structure called a Binary Search Tree (BST) is used to store and arrange data in a sorted fashion.

It follows all the rules of Binary tree and also follows some additional features like

- The left child of any node contains values less than the parent node.
- The right child contains values greater than the parent node.

This structure ensures that operations like searching, insertion, and deletion can be performed efficiently, typically in O(log n) time if the tree is balanced. The sorted nature of BSTs allows quick lookups and easy navigation through the tree.



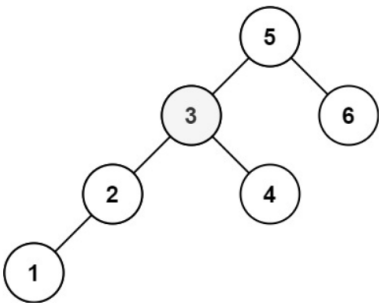
Binary Tree

Run Submit

Description | Editorial | Solutions | Submissions

```
Output: 1
```

Example 2:




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

Constraints:

- The number of nodes in the tree is `n`.
- `1 <= k <= n <= 10^4`

Code

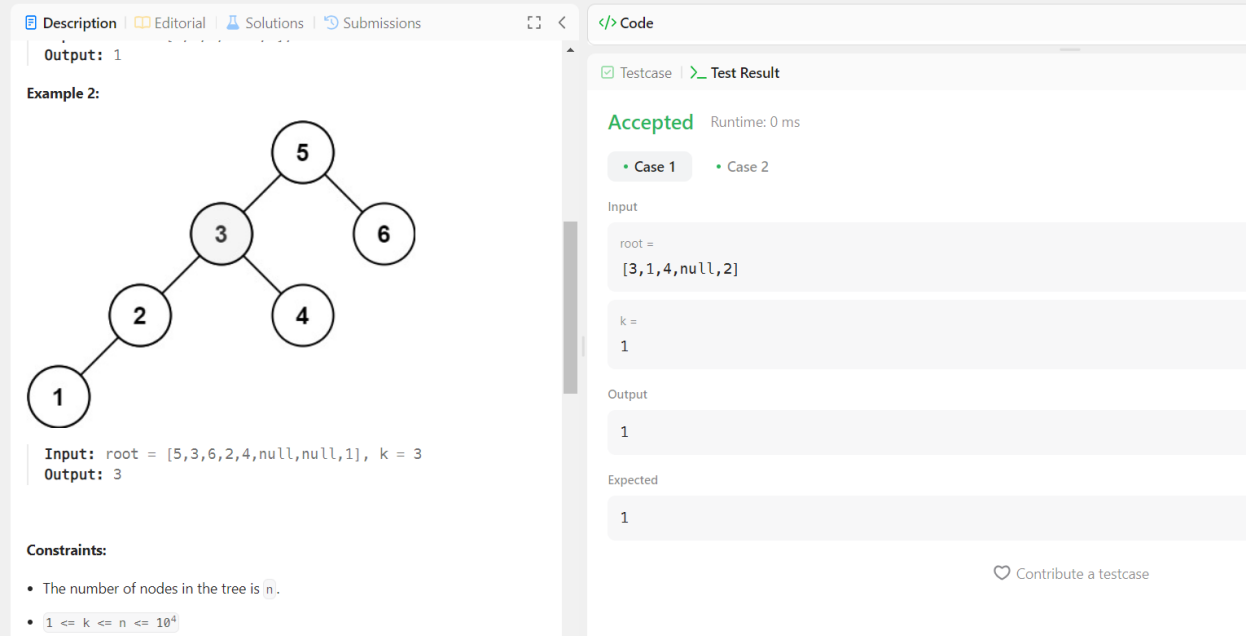
Java | Auto

```
9   *     TreeNode(int val, TreeNode left, TreeNode right) {
10  *         this.val = val;
11  *         this.left = left;
12  *         this.right = right;
13  *     }
14  * }
15  */
16  class Solution {
17      int count=0;
18      int min=-1;
19      public int kthSmallest(TreeNode root, int k) {
20          inorder(root,k);
21          return min;
22      }
23      public void inorder(TreeNode node, int k)
24      {
25          if(node==null)
26            return;
27          inorder(node.left,k);
28          count++;
29          if(k==count)
30          {
31                  min=node.val;
32                  return ;
33          }
34          inorder(node.right,k);
35      }
36  }
```

Saved

Using inorder traversal, which traverses nodes in ascending order, the method is intended to get the kth smallest element in a Binary Search Tree (BST). It keeps track of the number of nodes visited by maintaining a count variable. The kth smallest element has been located when count == k, at which point the value of the current node is set to min. In certain circumstances, efficiency is increased because the traverse ends when the kth element is located.

Time Complexity: In the worst scenario, O(n) is used, where n is the entire number of nodes (or big k if the tree is skewed).
O(k) is the best case scenario, particularly if k is small and the traverse ends early.



Advantages of Binary Trees:

Effective Search: A unique kind of binary tree called a Binary Search Tree (BST) makes it possible to do insertion, deletion, and searching in an effective manner. Compared to linked lists and arrays, where the average search time is O(log n) if balanced, each node can have up to two child nodes, leading to significantly quicker search times.

Memory Efficient: Binary trees often use less memory than more intricate tree

structures, in part because they only allow a maximum of two offspring per node, which promotes efficient memory utilization.

Simplicity: Because each node in a binary tree has a maximum of two children, traversal, insertion, and deletion operations are made simple, making binary trees easy to construct and comprehend.

Disadvantages of Binary Trees:

Restricted Structure: Binary trees are limited to a maximum of two child nodes per parent node. This can be restrictive in some applications that require more than two child nodes per node; in those situations, binary trees are not appropriate.

Unbalanced Trees: Search operations perform less effectively in binary trees if one subtree is noticeably larger than the other. Performance can suffer greatly from improper balancing, like with AVL or Red-Black trees.

Space Inefficiency: Binary trees take more memory to store two child pointers per node than arrays or linked lists do. This might result in memory overhead, particularly for large trees.

Dynamic Programming:

Dynamic Programming (DP) is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler sub problems. By solving each sub problem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

How does it works?

Sub problem Identification: Break the primary issue down into more manageable, stand-alone sub problems.
Store Solutions: Resolve every sub-problem, then record the outcome in a table or

array.
Build Up Solutions: Assemble the primary problem's solution using the previously saved solutions.
Prevent Redundancy: DP reduces computing time by guaranteeing that each sub problem is solved only once by storing solutions.

1. Top-Down Approach (Memoization): This method, sometimes referred to as memoization, begins with the ultimate solution and iteratively dissects it into more manageable subproblems. We keep the solutions to the solved subproblems in a memoization table to prevent duplication of computations.

Let's dissect. Top-down methodology:

iteratively divides the ultimate solution into smaller subproblems starting with the final solution.
keeps track of subproblem solutions in a table to prevent repeating computations.
Appropriate when there are a lot of subproblems and a high percentage of reuse.

2. Bottom-Up technique (Tabulation): The bottom-up technique, sometimes referred to as tabulation, begins with the tiniest subproblems and works its way up to the ultimate answer. To save repeating computations, we keep the outcomes of subproblems that have been solved in a table.

Let’s breakdown Bottom-up approach:

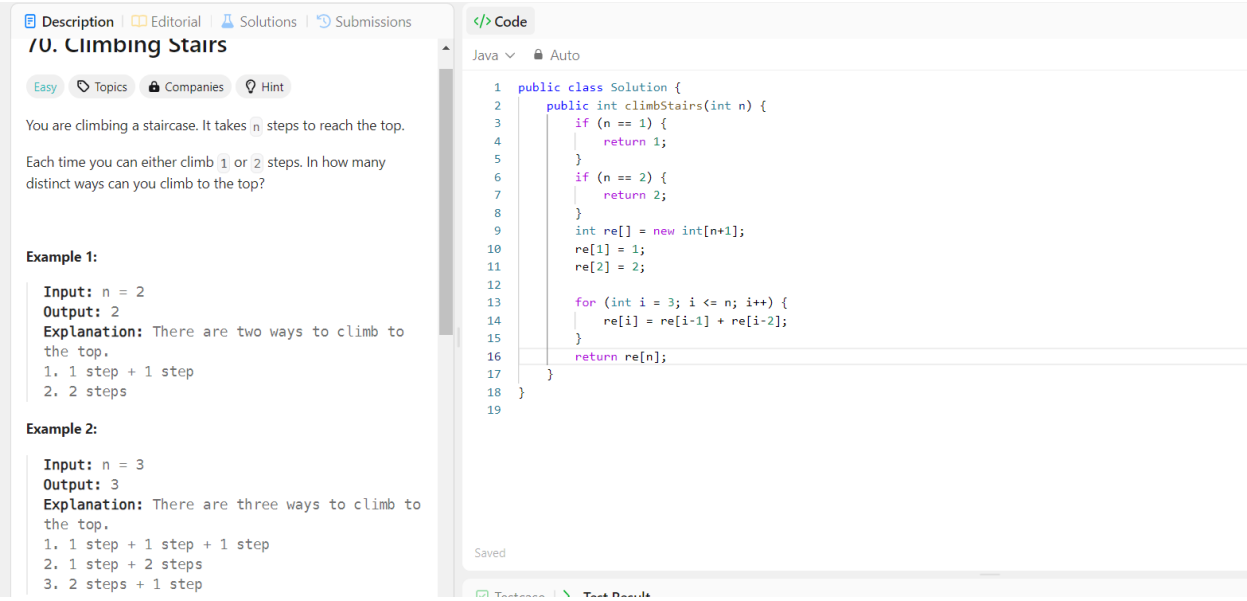
works up to the final solution by starting with the smallest subproblems.
fills a table with bottom-up solutions to subproblems.
Appropriate in situations where there are few subproblems and the best solution may be computed straight from the smaller subproblem solutions.

Dynamic Programming's (DP) benefits

There are several benefits to dynamic programming, such as:

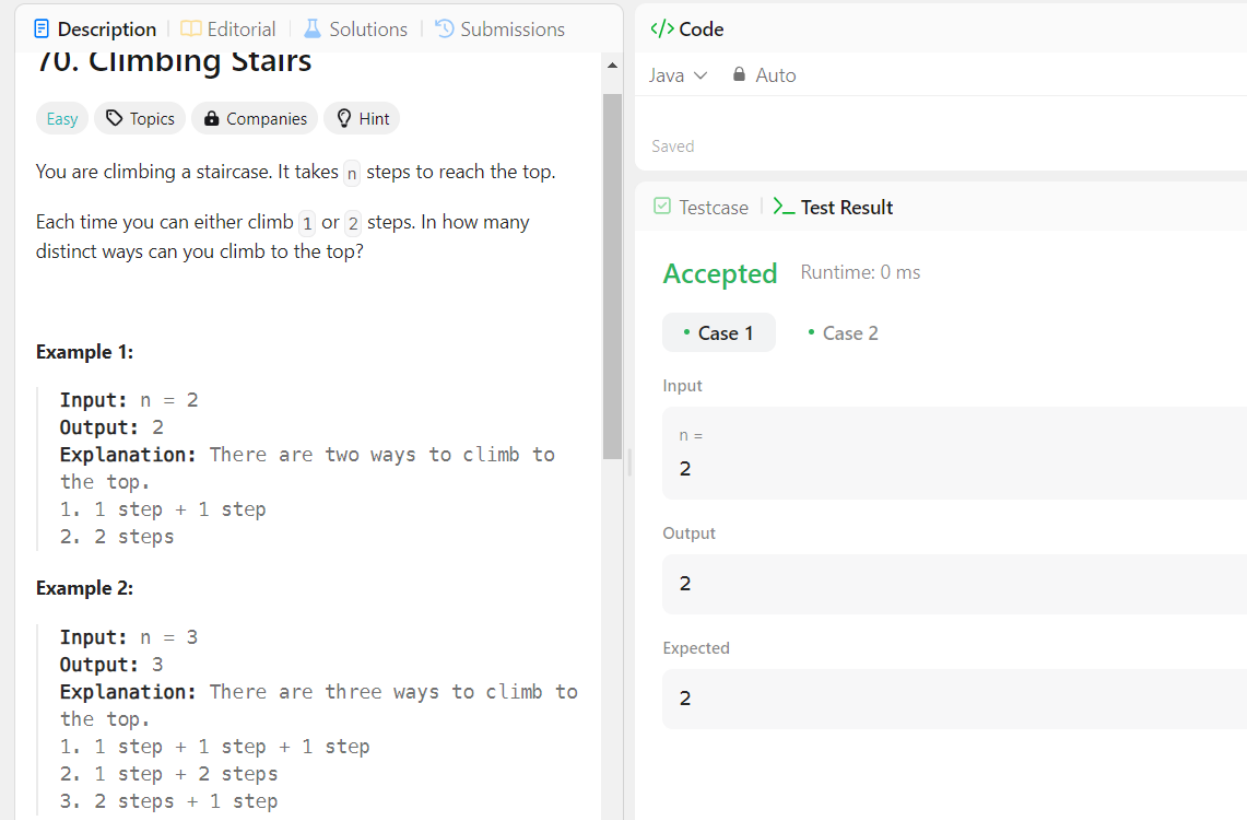
saves a lot of time by avoiding the need to compute the same subproblems more than once.
guarantees that all potential combinations are taken into account in order to get the best answer.
divides complicated issues into more manageable, smaller subproblems.

Climbing Stairs:

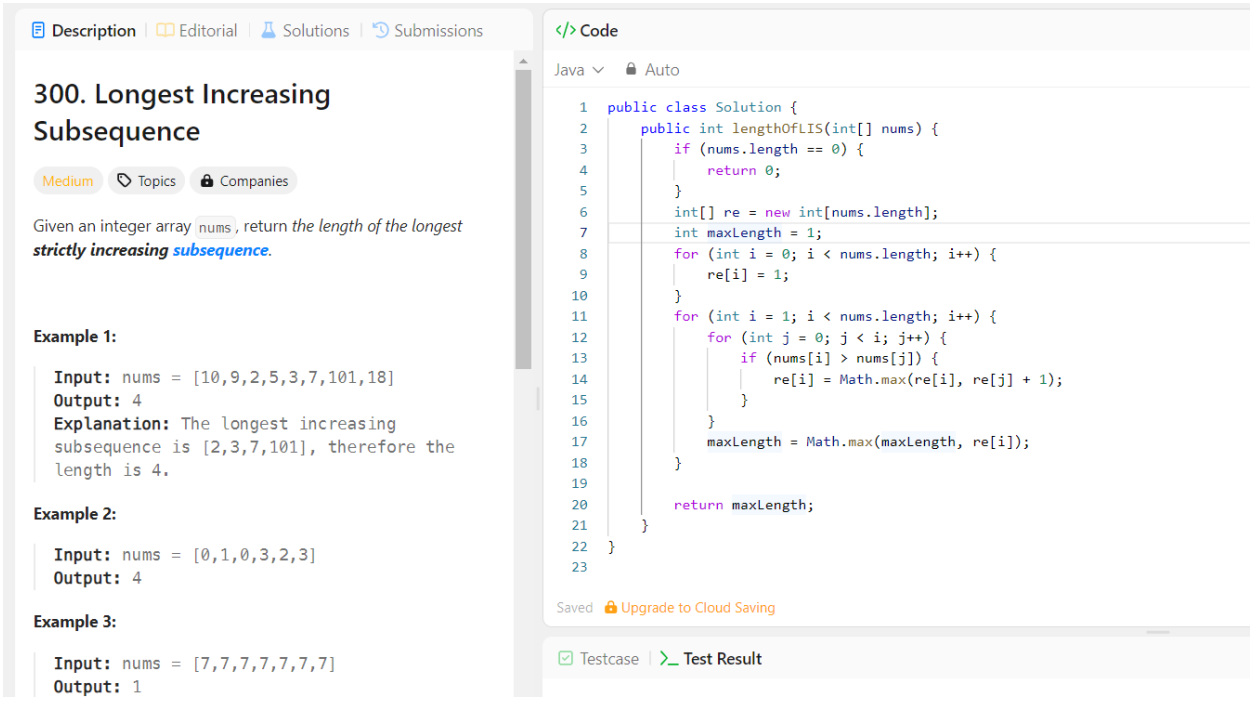


Since these are straightforward scenarios with known results, the method first handles the base cases where n == 1 or n == 2. It returns 1 or 2 directly in these cases. To store the number of ways to reach each step, an array called re[] is formed with a size of n + 1. Initial values for re[1] = 1 and re[2] = 2 are set to reflect the base cases. Beginning at i = 3, the loop adds the methods to reach the previous two stages to get the number of ways to reach each step: re[i] = re[i-1] + re[i-2]. The outcome is finally stored in re[n], and it is given back as the total number of unique ways to climb the staircase.

Since the loop executes once for every step from 3 to n, the time complexity is O(n). Because intermediate results are stored in an array called re[], the space complexity is O(n).



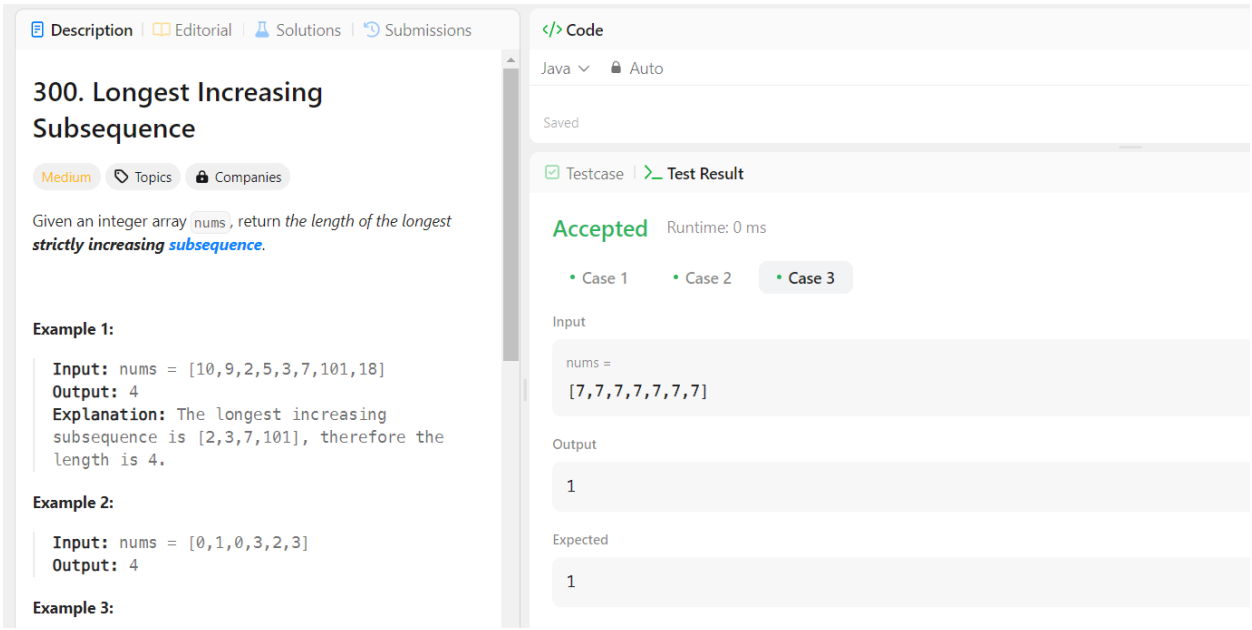
Longest Increasing Subsequence:



Dynamic programming is used to calculate the length of the longest subsequence where each successive number is greater than the preceding one in order to solve the longest increasing subsequence (LIS) problem.

As the minimal LIS ending at any index is at least the element itself, it begins by initializing an array re and setting each element to 1. The inner loop verifies all earlier items nums[j] (where j < i) for each element nums[i], while the outer loop iterates through each element of nums. Re[i] is updated to the greatest value between its current value and re[j] + 1 if nums[i] is greater than nums[j], indicating that nums[i] can expand the subsequence ending at nums[j]. The maxLength variable, which represents the length of the LIS, is set to the greatest value in the re array once all items have been checked.

O(n 2) is the time complexity because of the nested loops.


Description
Editorial
Solutions
Submissions
300. Longest Increasing Subsequence
Medium
Topics
Companies
Given an integer array nums, return the length of the longest strictly increasing subsequence.
Example 1:
Input: nums = [10,9,2,5,3,7,101,18]
Output: 4
Explanation: The longest increasing subsequence is [2,3,7,101], therefore the length is 4.
Example 2:
Input: nums = [0,1,0,3,2,3]
Output: 4
Example 3:
Code
Java
Auto
Saved
Testcase
Test Result
Accepted
Runtime: 0 ms
Case 1
Case 2
Case 3
Input
nums =
[7,7,7,7,7,7,7]
Output
1
Expected
1