**=============== DYNAMIC PROGRAMMING ===============**

Dynamic programming works by breaking down a problem into smaller subproblems, solving each subproblem independently, and using the solutions to these subproblems to construct the overall solution. The solutions to the subproblems are stored in a table or array(memoization) or in a bottom-up manner(tabulation) to avoid redundant computation.

**Recursion** is used to automate a function, whereas dynamic programming is an optimization technique used to solve problems.

**Recursive** functions recognize when they are needed, execute themselves, then stop working. When the function identifies the moment it is needed, it calls itself and is executed; this is called a recursive case. As a result, the function must stop once the task is completed, known as the base case.

**Dynamic programming** recognizes the problem and divides it into sub-problems in order to solve the whole scene. After solving these sub-problems, or variables, the programmer must establish a mathematical relationship between them. These solutions and results are stored as algorithms, so they can be accessed in the future without having to solve the whole problem again.

## **Techniques to solve Dynamic Programming Problems:**

## **1. Top-Down(Memoization):**

Break down the given problem in order to begin solving it. If you see that the problem has already been solved, return the saved answer. If it hasn’t been solved, solve it and save it. This is usually easy to think of and very intuitive, This is referred to as Memoization.

**2. Bottom-Up(Tabulation):**

Analyse the problem and see in what order the subproblems are solved, and work your way up from the trivial subproblem to the given problem. This process ensures that the subproblems are solved before the main problem. This is referred to as Tabulation.

**=================== Binary Search Tree ===================**

A **Binary Search Tree (BST)** is a node-based binary tree data structure which has the following properties:

* The left subtree of a node contains only nodes with keys lesser than the node’s key.
* The right subtree of a node contains only nodes with keys greater than the node’s key.
* The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.

The major advantage of binary search trees over other data structures is that the related sorting algorithms and search algorithms such as in-order traversal can be very efficient.

**Traversal**

There are several ways to traverse a BST:

* **In-order Traversal**: Visit the left subtree, the root, and then the right subtree.
* **Pre-order Traversal**: Visit the root, the left subtree, and then the right subtree.
* **Post-order Traversal**: Visit the left subtree, the right subtree, and then the root.

Traversal operations are used to print out the data in a sorted manner and to analyse the tree structure.

**Time Complexity**

The average time complexity for the operations of search, insertion, and deletion in a BST is **O(log n)**, where **n** is the number of nodes in the tree. However, in the worst case, these complexities can degrade to **O(n)**, particularly in the case of an unbalanced tree.

**CODE:**

package practice;

import java.util.Scanner;

public class BinarySearchTree{

class Node{

int data;

Node left,right;

Node(int key){

this.data = key;

left = right = null;

}

}

Node root;

BinarySearchTree()

{

root = null;

}

void insert(int key) {

root = insertRec(root,key);

}

Node insertRec(Node root, int key) {

if(root==null) {

root = new Node(key);

return root;

}

if(root.data<key) {

root.right = insertRec(root.right,key);

}

else if(root.data>key) {

root.left = insertRec(root.left,key);

}

return root;

}

void inorder() {

inorderRec(root);

}

void inorderRec(Node root) {

if(root!=null) {

inorderRec(root.left);

System.out.println(root.data);

inorderRec(root.right);}

}

boolean search(Node root, int key) {

return searchRec(root,key);

}

boolean searchRec(Node root, int key) {

if(root==null || root.data==key)

return root!=null;

if(root.data<key)

return searchRec(root.right,key);

return searchRec(root.left,key);

}

void depthFirstSearch(Node root) {

System.out.println("Depth First Search (Pre-order Traversal):");

dfsRec(root);

}

void dfsRec(Node root) {

if(root==null)

return;

System.out.println(root.data);

dfsRec(root.left);

dfsRec(root.right);

}

public static void main(String[] args) {

BinarySearchTree bst = new BinarySearchTree();

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

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

int x = sc.nextInt();

bst.insert(x);

}

bst.inorder();

}

}