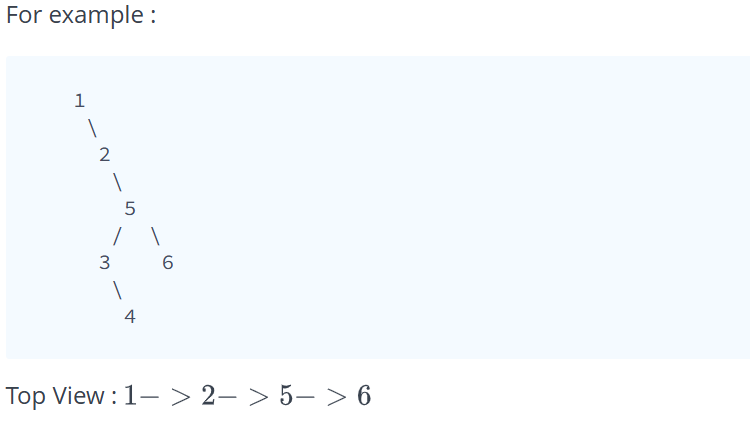
**30 DAYS CODING CHALLENGE:**

**DAY 15:**

**TOP VIEW :**

Given a pointer to the root of a binary tree, print the top view of the binary tree.

The tree as seen from the top the nodes, is called the top view of the tree.



**SOURCE CODE:**

void find(Node\* root, int pos, int &l, int &r) {

if (!root) return;

l = min(pos, l);

r = max(pos, r);

find(root->left, pos - 1, l, r);

find(root->right, pos + 1, l, r);

}

void topView(Node\* root) {

if (!root) return;

int l = 0, r = 0;

find(root, 0, l, r);

vector<int> ans(r - l + 1);

vector<bool> filled(r - l + 1, false);

queue<pair<Node\*, int>> q;

q.push({root, -l});

while (!q.empty()) {

auto [temp, pos] = q.front();

q.pop();

if (!filled[pos]) {

filled[pos] = true;

ans[pos] = temp->data;

}

if (temp->left){

q.push({temp->left, pos - 1});

}

if (temp->right){

q.push({temp->right, pos + 1});

}

}

for (int val : ans) {

cout << val << " ";

}

cout << endl;

}

**DRY RUN:**

**Example:**

1

/ \

2 3

\ \

4 5

/ / \

6 7 8

**Step-by-Step Execution**

1. **Find Leftmost and Rightmost Horizontal Positions (find Function)**
   * Starting from pos = 0 at root 1
   * Traverse left subtree:
     + 2 → pos = -1
     + 4 → pos = 0
     + 6 → pos = -1
   * Traverse right subtree:
     + 3 → pos = 1
     + 5 → pos = 2
     + 7 → pos = 1
     + 8 → pos = 3
   * **Leftmost l = -1, Rightmost r = 3**
   * Size of ans vector = r - l + 1 = 5 (Indexes 0 to 4)
2. **BFS Traversal (topView Function)**
   * **Queue initialized with:** {(1,1)}
   * **Processing nodes level-wise:**
     + **(1,1)** → First at index 1, store 1
       - Add left (2,0), right (3,2)
     + **(2,0)** → First at index 0, store 2
       - Add right (4,1)
     + **(3,2)** → First at index 2, store 3
       - Add right (5,3)
     + **(4,1)** → Index 1 already filled (skip)
       - Add left (6,0)
     + **(5,3)** → First at index 3, store 5
       - Add left (7,2), right (8,4)
     + **(6,0)** → Index 0 already filled (skip)
     + **(7,2)** → Index 2 already filled (skip)
     + **(8,4)** → First at index 4, store 8
3. **Final ans Array:** [2, 1, 3, 5, 8]

**DAY 16:**

**LIST TO BST:**

Given the head of a singly linked list where elements are sorted in ascending order, convert it to a height-balanced binary search tree.

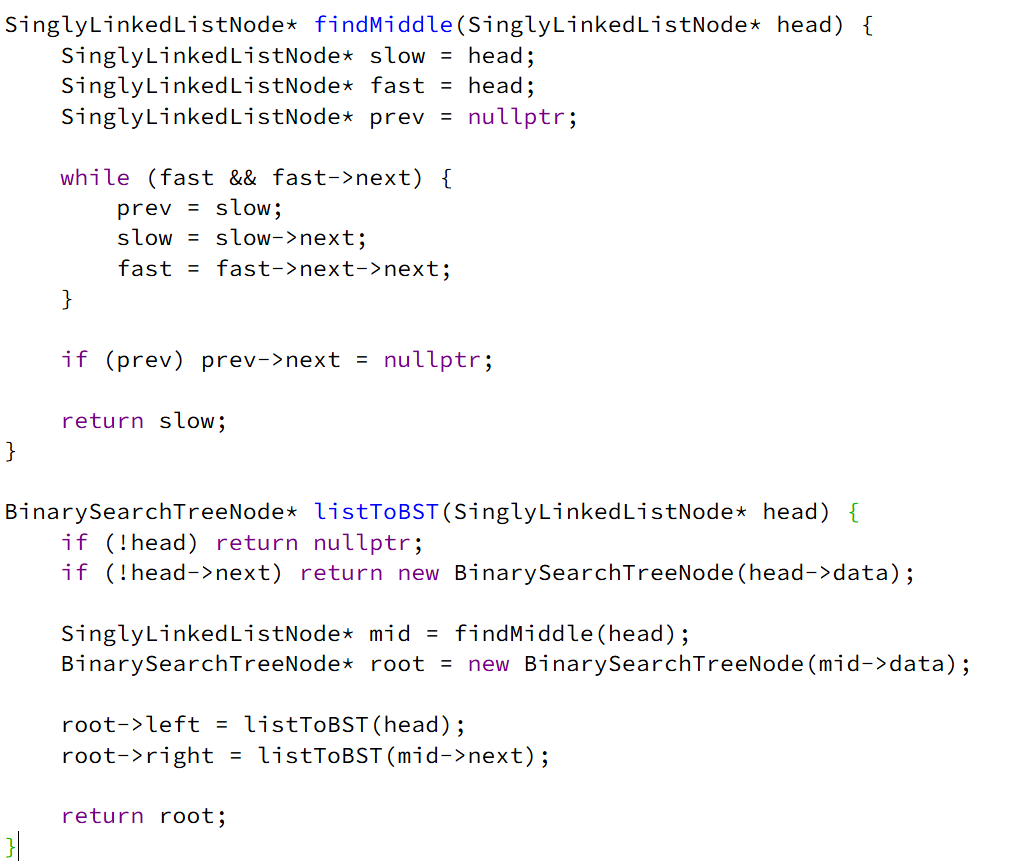
**Example:**

Input: head = [-10,-3,0,5,9]

Output: [0,-3,9,-10,null,5]

Explanation: One possible answer is [0,-3,9,-10,null,5], which represents the shown height balanced BST.

**SOURCE CODE:**



**DRY RUN:**

**Example:**

1 -> 2 -> 3 -> 4 -> 5

**Step 1: Find Middle of the List**

* **Initial Pointers**:
  + slow = 1, fast = 1, prev = nullptr
* **Iteration 1**:
  + Move slow to 2, fast to 3
  + prev = 1
* **Iteration 2**:
  + Move slow to 3, fast to 5
  + prev = 2
* **Stop Condition**: fast reaches the end.
* **Middle Element Found**: 3
* **Split the List**:
  + Left Part: 1 -> 2
  + Right Part: 4 -> 5

**Step 2: Construct BST**

* **Root Node**: 3
* **Recursively Call listToBST(head) on Left Part (1 -> 2)**
  + **Find Middle**: 2
  + **Split**:
    - Left: 1
    - Right: nullptr
  + **Create Nodes**:
    - 2 as Left Child of 3
    - 1 as Left Child of 2
* **Recursively Call listToBST(mid->next) on Right Part (4 -> 5)**
  + **Find Middle**: 5
  + **Split**:
    - Left: 4
    - Right: nullptr
  + **Create Nodes**:
    - 5 as Right Child of 3
    - 4 as Left Child of 5

**OUTPUT:**

**3**

**/ \**

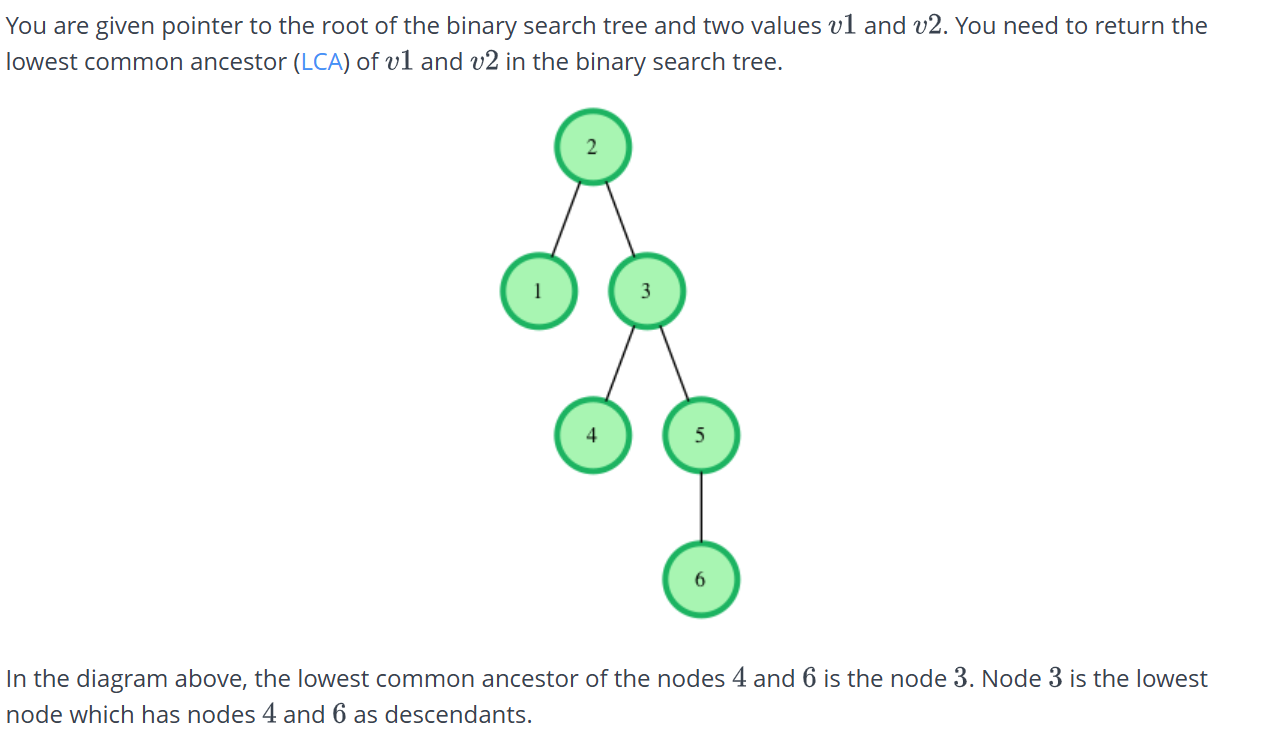
**2 5**

**/ /**

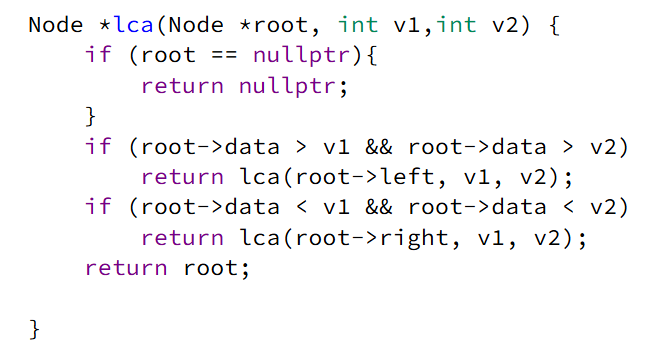
**1 4**

**DAY 17:**

**LOWEST COMMON ANCESTOR**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

5

/ \

3 8

/ \ / \

2 4 6 9

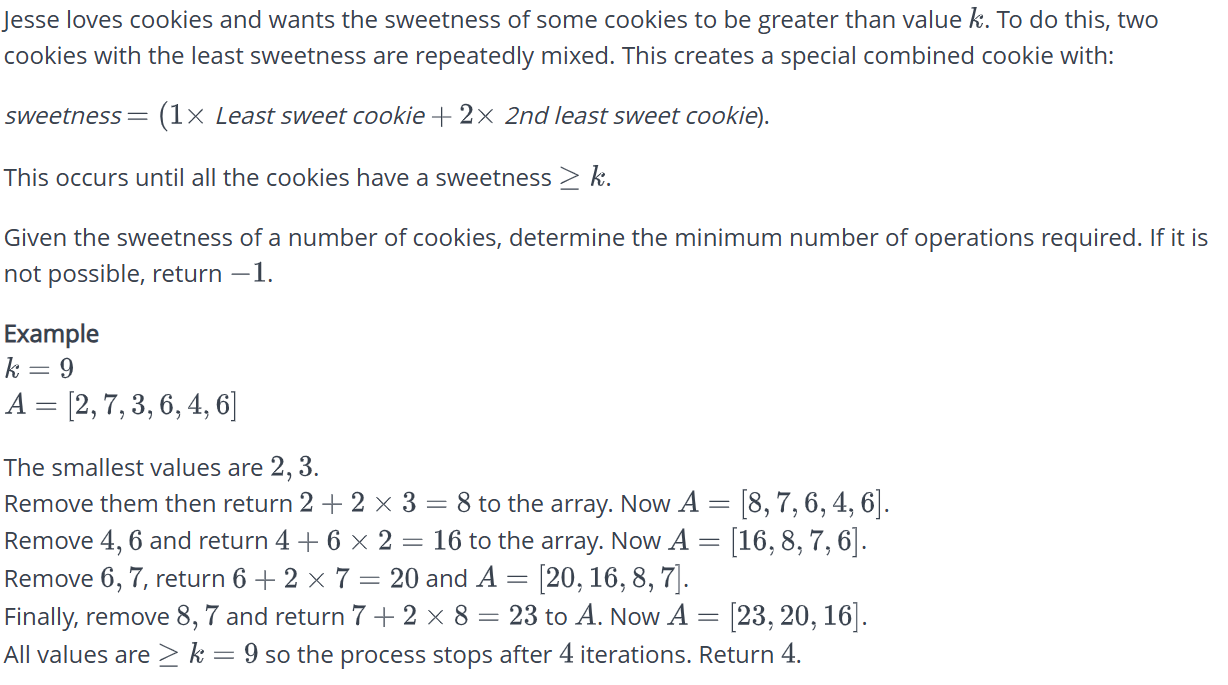
Find LCA of 4 and 6

**Initial Call:** lca(root = 5, v1 = 4, v2 = 6)

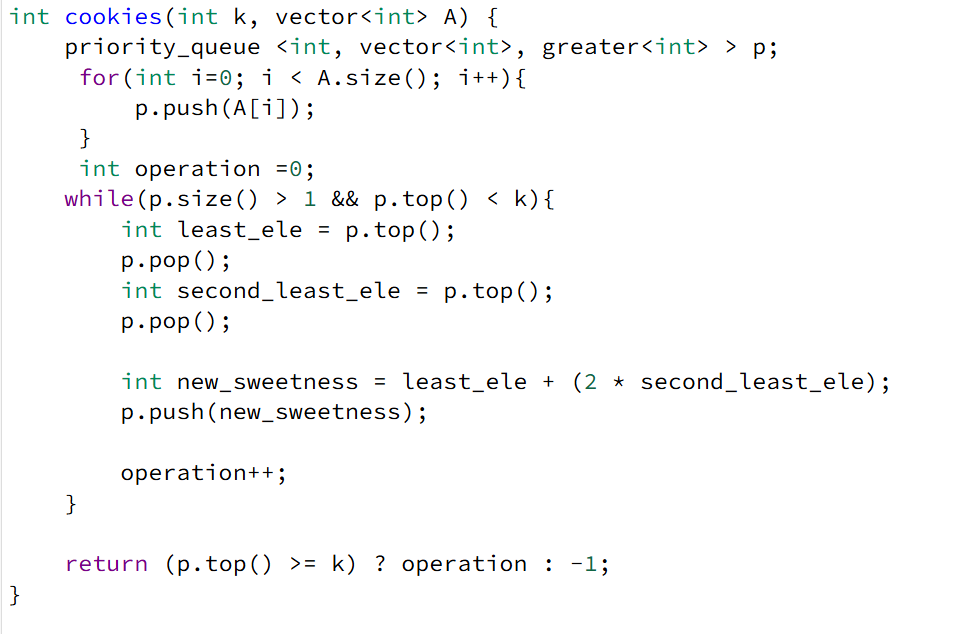
* root->data = 5
* v1 = 4, v2 = 6
* 4 < 5 and 6 > 5 → **Found LCA at 5**
* **Return 5**

**DAY 18:**

**JESSE AND COOKIES**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

k = 7

A = [1, 2, 3, 9, 10, 12]

**Initialize Min-Heap (priority\_queue)**

* Insert elements [1, 2, 3, 9, 10, 12]
* Heap now: [1, 2, 3, 9, 10, 12]

**First Iteration** (smallest elements: 1 and 2)

* Remove 1 and 2
* New cookie = 1 + (2 × 2) = 5
* Insert 5 into heap
* Heap now: [3, 5, 9, 10, 12]
* operation = 1

**Second Iteration** (smallest elements: 3 and 5)

* Remove 3 and 5
* New cookie = 3 + (2 × 5) = 13
* Insert 13 into heap
* Heap now: [9, 10, 12, 13]
* operation = 2

**Check Top Element (p.top() = 9)**

* All cookies now have sweetness **≥ 7**
* **Exit loop**

**Return operation = 2**

**OUTPUT:** **2**

**DAY 19:**

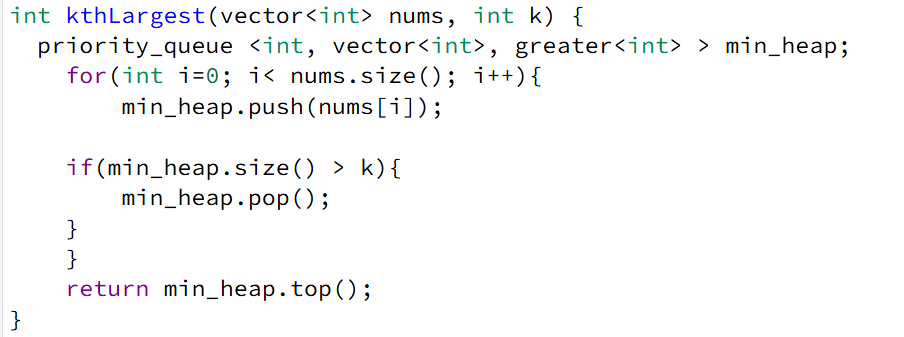
**KTH LARGEST ELEMENT:**

Given an integer array nums and an integer k, return the kth largest element in the array.

Note that it is the kth largest element in the sorted order, not the kth distinct element.

Can you solve it without sorting?

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

nums = [3, 2, 1, 5, 6, 4]

k = 2

**Initialize Min-Heap (priority\_queue)**

* min\_heap is created to store the largest k elements.

**Processing Elements One by One:**

* **Insert 3** → Heap: [3]
* **Insert 2** → Heap: [2, 3]
* **Insert 1** → Heap: [1, 3, 2] (Heap size > k, remove 1)
  + **After pop:** [2, 3]
* **Insert 5** → Heap: [2, 3, 5] (Heap size > k, remove 2)
  + **After pop:** [3, 5]
* **Insert 6** → Heap: [3, 5, 6] (Heap size > k, remove 3)
  + **After pop:** [5, 6]
* **Insert 4** → Heap: [4, 6, 5] (Heap size > k, remove 4)
  + **After pop:** [5, 6]

**OUTPUT:**

* The **top of the heap** is 5, which is the **2nd largest element**.

**DAY 20:**

**CONTAINER WITH MOST WATER:**

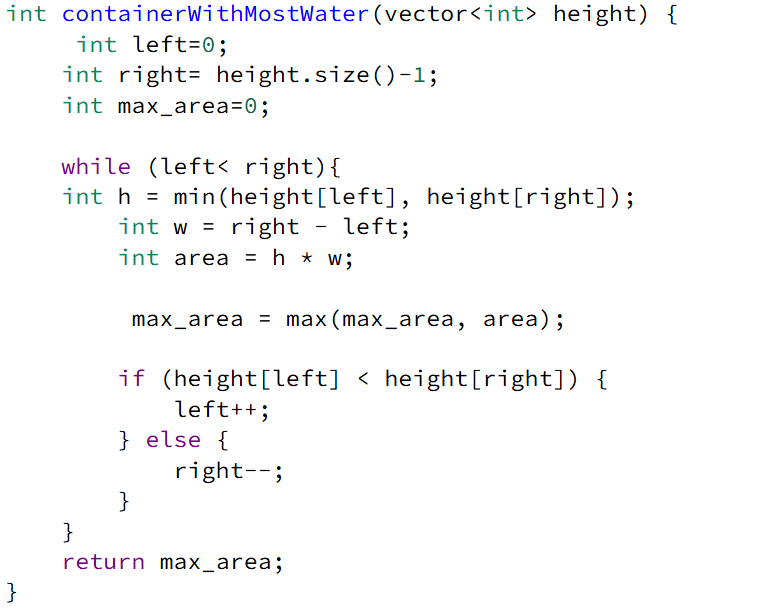
You are given an integer array height of length n. There are n vertical lines drawn such that the two endpoints of the ith line are (i, 0) and (i, height[i]).

Find two lines that together with the x-axis form a container, such that the container contains the most water.

Return the maximum amount of water a container can store.

Notice that you may not slant the container.

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

height = [1, 8, 6, 2, 5, 4, 8, 3, 7]

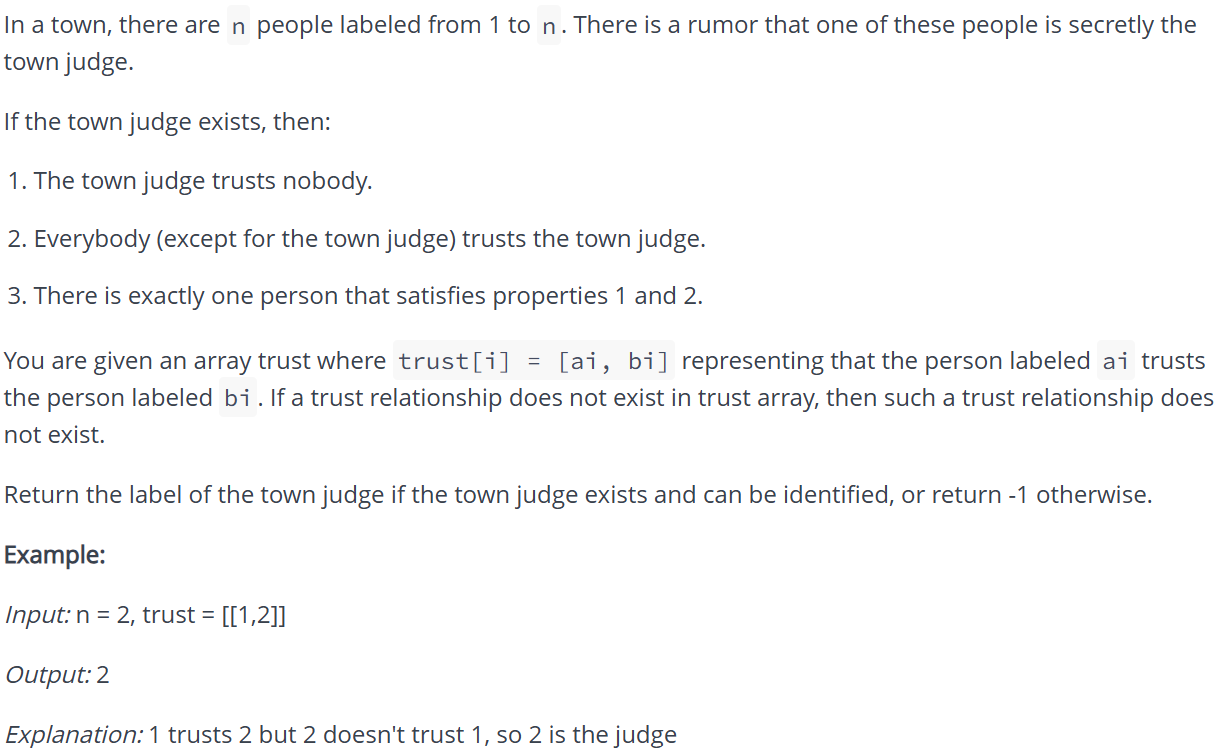
1. **Initialize:**
   * left = 0, right = 8, max\_area = 0
2. **Iteration 1:** (left = 0, right = 8)
   * h = min(1, 7) = 1
   * w = 8 - 0 = 8
   * area = 1 \* 8 = 8
   * **Update max\_area to 8**
   * **Move left to 1** (since height[0] < height[8])
3. **Iteration 2:** (left = 1, right = 8)
   * h = min(8, 7) = 7
   * w = 8 - 1 = 7
   * area = 7 \* 7 = 49
   * **Update max\_area to 49**
   * **Move right to 7** (since height[8] < height[1])
4. **Iteration 3:** (left = 1, right = 7)
   * h = min(8, 3) = 3
   * w = 7 - 1 = 6
   * area = 3 \* 6 = 18
   * max\_area remains 49
   * **Move right to 6**
5. **Iteration 4:** (left = 1, right = 6)
   * h = min(8, 8) = 8
   * w = 6 - 1 = 5
   * area = 8 \* 5 = 40
   * max\_area remains 49
   * **Move right to 5**
6. **Iteration 5:** (left = 1, right = 5)
   * h = min(8, 4) = 4
   * w = 5 - 1 = 4
   * area = 4 \* 4 = 16
   * max\_area remains 49
   * **Move right to 4**
7. **Iteration 6:** (left = 1, right = 4)
   * h = min(8, 5) = 5
   * w = 4 - 1 = 3
   * area = 5 \* 3 = 15
   * max\_area remains 49
   * **Move right to 3**
8. **Iteration 7:** (left = 1, right = 3)
   * h = min(8, 2) = 2
   * w = 3 - 1 = 2
   * area = 2 \* 2 = 4
   * max\_area remains 49
   * **Move right to 2**
9. **Iteration 8:** (left = 1, right = 2)
   * h = min(8, 6) = 6
   * w = 2 - 1 = 1
   * area = 6 \* 1 = 6
   * max\_area remains 49
   * **Move right to 1** (Loop ends)

### ****Final Output:****

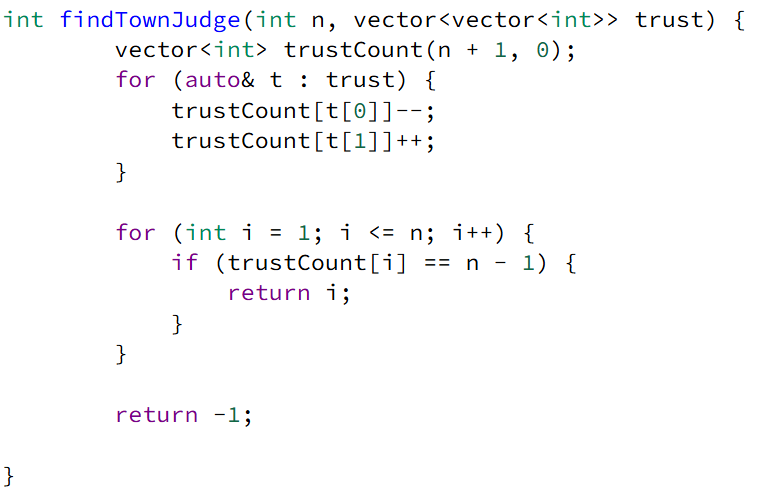
**containerWithMostWater([1, 8, 6, 2, 5, 4, 8, 3, 7]) → 49**

**DAY 21:**

**WHO’S THE TOWN JUDGE:**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

n = 3;

trust = {{1, 3}, {2, 3}};

**Step 1: Initialization**

* trustCount array of size n+1 = 4:

ini

CopyEdit

trustCount = [0, 0, 0, 0]

**Step 2: Process Trust Pairs**

* {1, 3} → Person 1 trusts 3
  + trustCount[1]-- → -1
  + trustCount[3]++ → 1

ini

CopyEdit

trustCount = [0, -1, 0, 1]

* {2, 3} → Person 2 trusts 3
  + trustCount[2]-- → -1
  + trustCount[3]++ → 2

ini

CopyEdit

trustCount = [0, -1, -1, 2]

**Step 3: Find the Judge**

* trustCount[1] = -1 →Not judge
* trustCount[2] = -1 → Not judge
* trustCount[3] = 2 == (n-1) → **Judge is 3**

**Final Output**

* **Return 3**

**DAY 22:**

**CLIMBING STAIRS:**

You are climbing a staircase. It takes n steps to reach the top.

Each time you can either climb 1 or 2 steps. In how many distinct ways can you climb to the top?

**Example 1:**

*Input*: n = 2

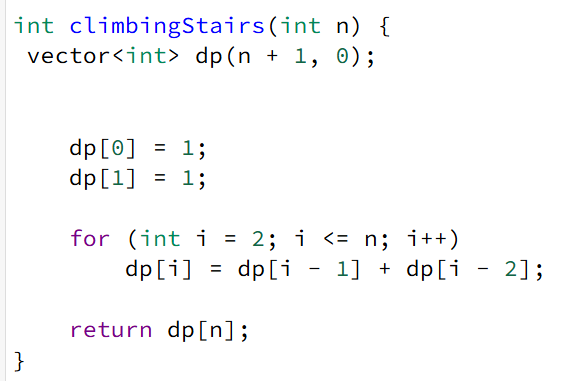
*Output*: 2

*Explanation*: There are two ways to climb to the top.

1 step + 1 step

2 steps

**SOURCE CODE:**

****

**DRY RUN :**

***Input:*** *n = 4****Output:*** *5****Explanation****: There are five ways to reach 4th stair: {1, 1, 1, 1}, {1, 1, 2}, {2, 1, 1}, {1, 2, 1} and {2, 2}.*

**Step 1: Initialization**

Input: n = 4

Create a dp array of size n + 1 (i.e., dp[5]), initialized to 0:

dp = [0, 0, 0, 0, 0]

**Set base cases:**

* dp[0] = 1 (1 way to stay at ground level)
* dp[1] = 1 (1 way to reach the first step)
* dp = [1, 1, 0, 0, 0]

**Step 2: Compute dp values using the loop**

**Iteration 1 (i = 2)**:

* Formula: dp[2] = dp[1] + dp[0]
* Calculation: dp[2] = 1 + 1 = 2
* Updated array:
* dp = [1, 1, 2, 0, 0]

**Iteration 2 (i = 3)**:

* Formula: dp[3] = dp[2] + dp[1]
* Calculation: dp[3] = 2 + 1 = 3
* Updated array:
* dp = [1, 1, 2, 3, 0]

**Iteration 3 (i = 4)**:

* Formula: dp[4] = dp[3] + dp[2]
* Calculation: dp[4] = 3 + 2 = 5
* Updated array:
* dp = [1, 1, 2, 3, 5]

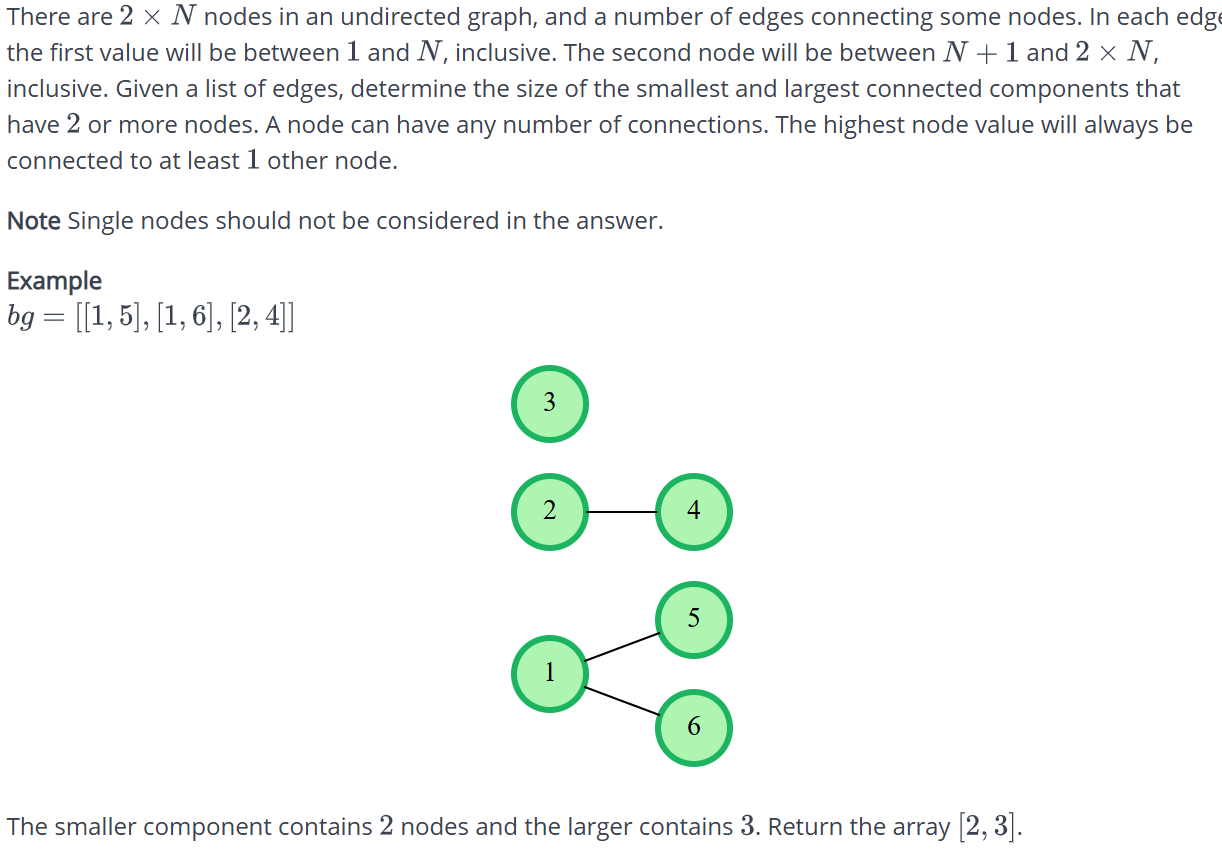
**Step 3: Final Output**

The number of ways to climb 4 steps is stored in dp[4] = 5.

**Return dp[4] = 5**.

**DAY 23:**

**COMPONENTS IN A GRAPH:**

****

**SOURCE CODE:**

unordered\_map<int, int> parent, size;

int find(int x) {

if (parent[x] != x) parent[x] = find(parent[x]);

return parent[x];

}

void unite(int x, int y) {

int rootX = find(x);

int rootY = find(y);

if (rootX != rootY) {

if (size[rootX] < size[rootY]) swap(rootX, rootY);

parent[rootY] = rootX;

size[rootX] += size[rootY];

}

}

void add(int x) {

if (parent.find(x) == parent.end()) {

parent[x] = x;

size[x] = 1;

}

}

vector<int> componentsInGraph(vector<vector<int>>& bg) {

unordered\_set<int> nodes;

for (auto& edge : bg) {

int u = edge[0], v = edge[1];

add(u);

add(v);

unite(u, v);

nodes.insert(u);

nodes.insert(v);

}

unordered\_map<int, int> componentSize;

for (int node : nodes) {

componentSize[find(node)]++;

}

int minSize = INT\_MAX, maxSize = INT\_MIN;

for (auto& [\_, size] : componentSize) {

minSize = min(minSize, size);

maxSize = max(maxSize, size);

}

return {minSize, maxSize};

}

**DRY RUN :**

**Example:**

bg = {{1, 2}, {2, 3}, {4, 5}, {6, 7}, {5, 6}}

#### ****Initial State:****

* parent = {} (stores parent of each node)
* size = {} (stores size of each component)

### ****Step-by-Step Dry Run****

#### ****Processing Edges:****

1. **Edge {1, 2}**
   * Add 1: parent[1] = 1, size[1] = 1
   * Add 2: parent[2] = 2, size[2] = 1
   * Unite 1 and 2:
     + find(1) = 1, find(2) = 2
     + parent[2] = 1, size[1] = 2
2. **Edge {2, 3}**
   * Add 3: parent[3] = 3, size[3] = 1
   * Unite 2 and 3:
     + find(2) = 1, find(3) = 3
     + parent[3] = 1, size[1] = 3
3. **Edge {4, 5}**
   * Add 4: parent[4] = 4, size[4] = 1
   * Add 5: parent[5] = 5, size[5] = 1
   * Unite 4 and 5:
     + find(4) = 4, find(5) = 5
     + parent[5] = 4, size[4] = 2
4. **Edge {6, 7}**
   * Add 6: parent[6] = 6, size[6] = 1
   * Add 7: parent[7] = 7, size[7] = 1
   * Unite 6 and 7:
     + find(6) = 6, find(7) = 7
     + parent[7] = 6, size[6] = 2
5. **Edge {5, 6}**
   * Unite 5 and 6:
     + find(5) = 4, find(6) = 6
     + Merge 6 into 4: parent[6] = 4, size[4] = 4

### ****Finding Component Sizes****

* **Final Components**:
  + {1, 2, 3} → Size 3
  + {4, 5, 6, 7} → Size 4
* **Minimum Size**: 3
* **Maximum Size**: 4