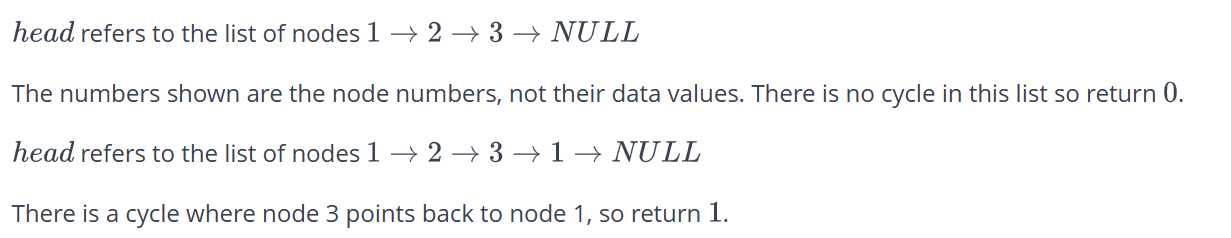
**30 DAYS CODING CHALLENGE:**

**DAY 8:**

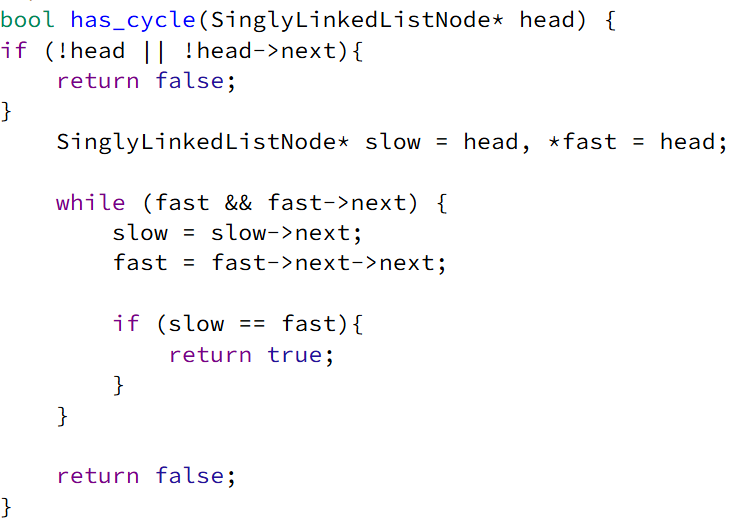
**CYCLE DETECTION:**

A linked list is said to contain a cycle if any node is visited more than once while traversing the list. Given a pointer to the head of a linked list, determine if it contains a cycle. If it does, return 1. Otherwise, return 0 .

**Example**



**SOURCE CODE:**

****

**DRY RUN:**

1 → 2 → 3 → 4

↑ |

└───┘ (Cycle from node 4 back to node 3)

 **Initialization:**

* slow = head (1), fast = head (1)

 **Iteration 1:**

* slow → 2, fast → 3
* slow ≠ fast → Continue

 **Iteration 2:**

* slow → 3, fast → 1 (due to cycle)
* slow ≠ fast → Continue

 **Iteration 3:**

* slow → 4, fast → 3
* slow ≠ fast → Continue

 **Iteration 4:**

* slow → 3, fast → 3
* slow == fast → **Cycle detected!**

**Final Output:**

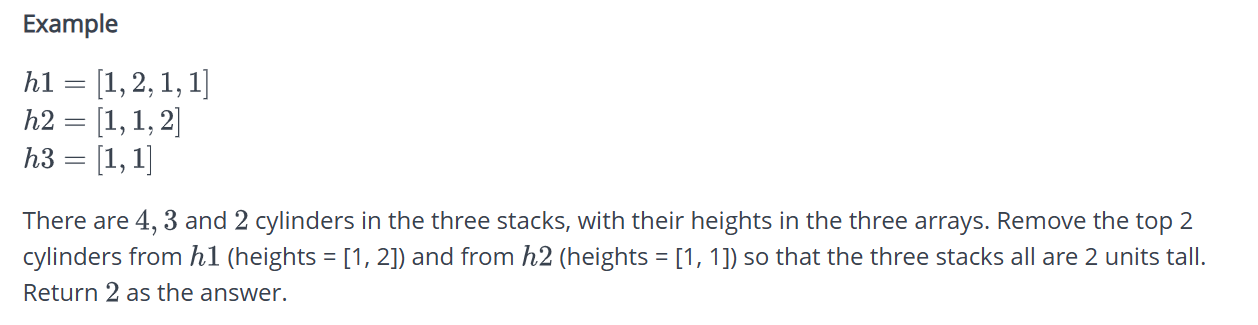
* **Return true**

**DAY 9:**

**EQUAL STACKS:**

You have three stacks of cylinders where each cylinder has the same diameter, but they may vary in height. You can change the height of a stack by removing and discarding its topmost cylinder any number of times.

Find the maximum possible height of the stacks such that all of the stacks are exactly the same height. This means you must remove zero or more cylinders from the top of zero or more of the three stacks until they are all the same height, then return the height.



**SOURCE CODE:**

int equalStacks(vector<int> h1, vector<int> h2, vector<int> h3) {

stack<int> s1,s2,s3;

int sum1=0, sum2=0 ,sum3=0;

for(int i= h1.size()-1; i>=0 ; i--){

sum1+= h1[i];

s1.push(sum1);

}

for(int i= h2.size()-1; i>=0 ; i--){

sum2+= h2[i];

s2.push(sum2);

}

for(int i= h3.size()-1; i>=0 ; i--){

sum3+= h3[i];

s3.push(sum3);

}

while (!s1.empty() && !s2.empty() && !s3.empty()) {

sum1 = s1.top();

sum2 = s2.top();

sum3 = s3.top();

if (sum1 == sum2 && sum2 == sum3){

return sum1;

}

if (sum1 >= sum2 && sum1 >= sum3) {

s1.pop();

}

else if (sum2 >= sum1 && sum2 >= sum3){

s2.pop();

}

else if (sum3 >= sum1 && sum3 >= sum2){

s3.pop();

}

}

return 0;

}

**DRY RUN:**

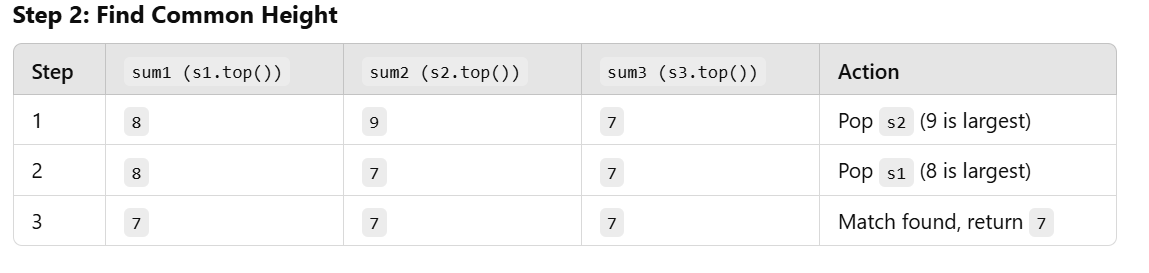
h1 = {3, 2, 1, 1, 1};

h2 = {4, 3, 2};

h3 = {1, 1, 4, 1};

**Step 1: Create Prefix Sum Stacks**

* **Stack s1 (Cumulative heights from bottom to top)**
  + {3, 5, 6, 7, 8}
* **Stack s2 (Cumulative heights from bottom to top)**
  + {4, 7, 9}
* **Stack s3 (Cumulative heights from bottom to top)**
  + {1, 2, 6, 7}



**Output:** 7

**DAY 10:**

**TIME NEEDED TO BUY TICKETS:**

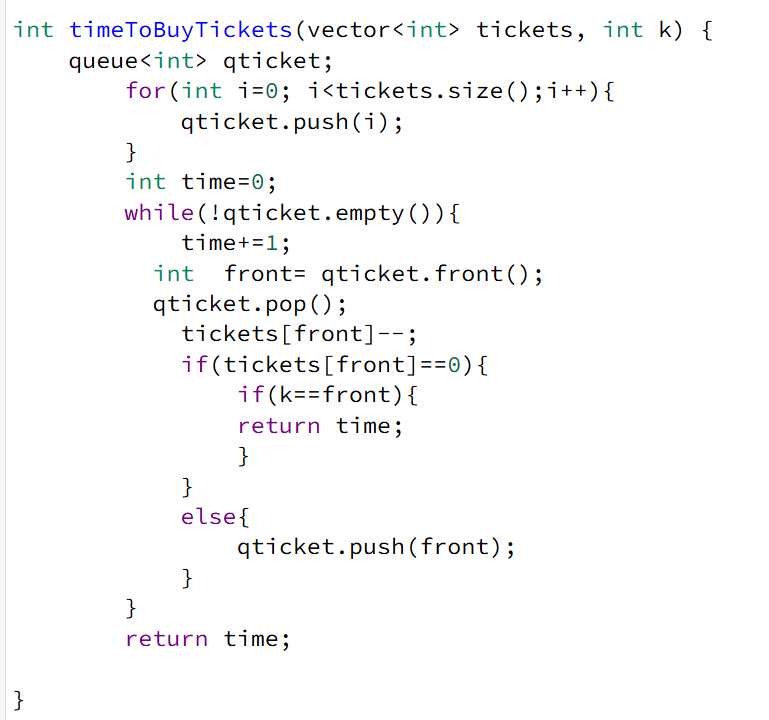
There are n people in a line queuing to buy tickets, where the 0th person is at the front of the line and the (n - 1)th person is at the back of the line.

You are given a 0-indexed integer array tickets of length n where the number of tickets that the ith person would like to buy is tickets[i].

Each person takes exactly 1 second to buy a ticket. A person can only buy 1 ticket at a time and has to go back to the end of the line (which happens instantaneously) in order to buy more tickets. If a person does not have any tickets left to buy, the person will leave the line.

Return the time taken for the person initially at position k (0-indexed) to finish buying tickets.

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

tickets = {2, 3, 2};

k = 2;

### ****Step 1: Initialize Queue****

* Queue stores indices: {0, 1, 2}
* time = 0

### ****Step 2: Process Queue Iteratively****

1. **First Iteration:**
   * front = 0, tickets[0] = 2 → 1
   * Re-enqueue 0
   * time = 1
2. **Second Iteration:**
   * front = 1, tickets[1] = 3 → 2
   * Re-enqueue 1
   * time = 2
3. **Third Iteration:**
   * front = 2, tickets[2] = 2 → 1
   * Re-enqueue 2
   * time = 3
4. **Fourth Iteration:**
   * front = 0, tickets[0] = 1 → 0
   * Remove 0 from queue
   * time = 4
5. **Fifth Iteration:**
   * front = 1, tickets[1] = 2 → 1
   * Re-enqueue 1
   * time = 5
6. **Sixth Iteration:**
   * front = 2, tickets[2] = 1 → 0
   * **Person at index k = 2 is done!**
   * **Return 6**

### ****Output:**** 6

**DAY 11:**

**BALANCED BRACKETS:**

A bracket is considered to be any one of the following characters: (, ), {, }, [, or ].

Two brackets are considered to be a *matched pair* if the an opening bracket (i.e., (, [, or {) occurs to the left of a closing bracket (i.e., ), ], or }) *of the exact same type*. There are three types of matched pairs of brackets: [], {}, and ().

A matching pair of brackets is *not balanced* if the set of brackets it encloses are not matched. For example, {[(])} is not balanced because the contents in between { and } are not balanced. The pair of square brackets encloses a single, unbalanced opening bracket, (, and the pair of parentheses encloses a single, unbalanced closing square bracket, ].

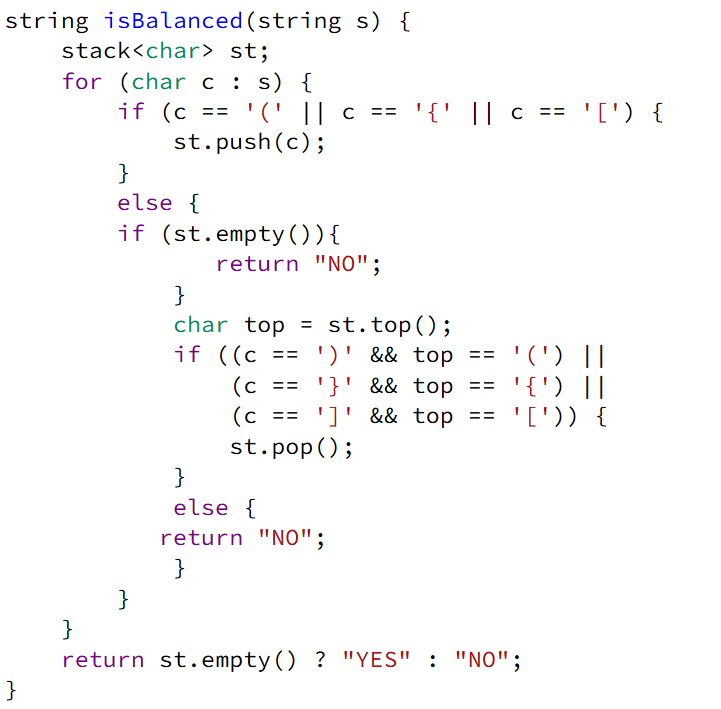
By this logic, we say a sequence of brackets is *balanced* if the following conditions are met:

It contains no unmatched brackets.

The subset of brackets enclosed within the confines of a matched pair of brackets is also a matched pair of brackets.

Given  strings of brackets, determine whether each sequence of brackets is balanced. If a string is balanced, return YES. Otherwise, return NO.

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

s = "{[()()]}"

 **Initialize an empty stack (st).**

 **Iterate through each character in s:**

* **First Character: {**
  + It is an opening bracket → Push { to the stack.
* **Second Character: [**
  + It is an opening bracket → Push [ to the stack.
* **Third Character: (**
  + It is an opening bracket → Push ( to the stack.
* **Fourth Character: )**
  + It is a closing bracket → Check top of stack (().
  + Matches → Pop ( from the stack.
* **Fifth Character: (**
  + It is an opening bracket → Push ( to the stack.
* **Sixth Character: )**
  + It is a closing bracket → Check top of stack (().
  + Matches → Pop ( from the stack.
* **Seventh Character: ]**
  + It is a closing bracket → Check top of stack ([).
  + Matches → Pop [ from the stack.
* **Eighth Character: }**
  + It is a closing bracket → Check top of stack ({).
  + Matches → Pop { from the stack.

 **Final Check:**

* Stack is empty, meaning all brackets matched correctly.

**OUTPUT:** Yes

**DAY 12:**

**JOSEPHUS PROBLEM:**

There are n friends that are playing a game. The friends are sitting in a circle and are numbered from 1 to n in clockwise order. More formally, moving clockwise from the ith friend brings you to the (i+1)th friend for 1 <= i < n, and moving clockwise from the nth friend brings you to the 1st friend.

The rules of the game are as follows:

Start at the 1st friend.

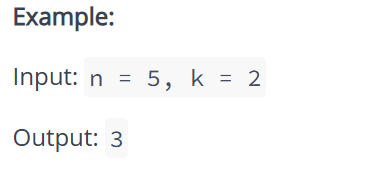
Count the next k friends in the clockwise direction including the friend you started at. The counting wraps around the circle and may count some friends more than once.

The last friend you counted leaves the circle and loses the game.

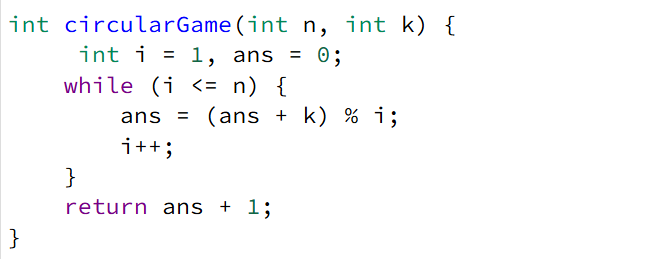
If there is still more than one friend in the circle, go back to step 2 starting from the friend immediately clockwise of the friend who just lost and repeat.

Else, the last friend in the circle wins the game.

Given the number of friends, n, and an integer k, return the winner of the game.

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

n = 5, k = 2

**Initialize variables:**

* i = 1, ans = 0

**First iteration (i = 1):**

* ans = (0 + 2) % 1 = 0
* Increment i → 2

**Second iteration (i = 2):**

* ans = (0 + 2) % 2 = 0
* Increment i → 3

**Third iteration (i = 3):**

* ans = (0 + 2) % 3 = 2
* Increment i → 4

**Fourth iteration (i = 4):**

* ans = (2 + 2) % 4 = 0
* Increment i → 5

**Fifth iteration (i = 5):**

* ans = (0 + 2) % 5 = 2
* Increment i → 6 (exit loop)

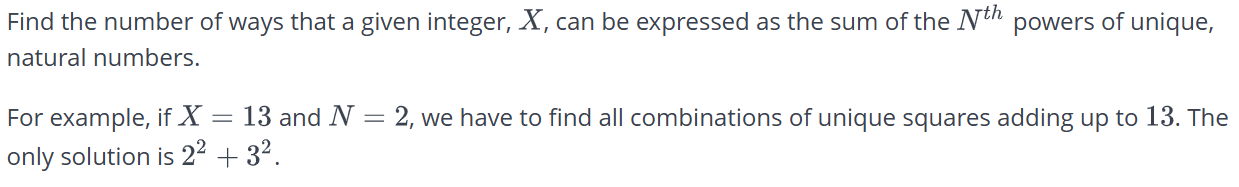
**Return result:**

* return ans + 1 = 2 + 1 = 3

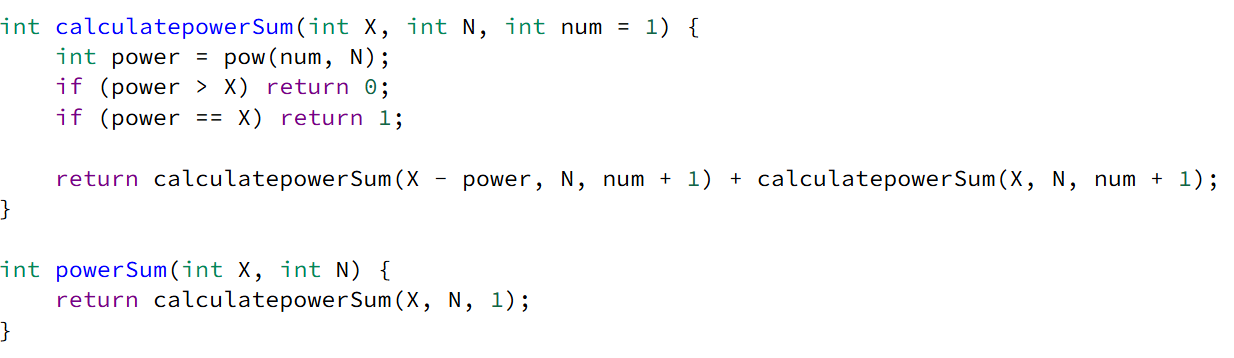
**Output:** 3

**DAY 13:**

**THE POWER SUM:**

****

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

X = 10, N = 2

 **Function Call:** powerSum(10, 2)

* Calls calculatePowerSum(10, 2, 1)

**First Call:** calculatePowerSum(10, 2, 1)

* power = 1^2 = 1
* Recursively calls:
  + calculatePowerSum(10 - 1, 2, 2)
  + calculatePowerSum(10, 2, 2)

**Second Call:** calculatePowerSum(9, 2, 2)

* power = 2^2 = 4
* Recursively calls:
  + calculatePowerSum(9 - 4, 2, 3)
  + calculatePowerSum(9, 2, 3)

**Third Call:** calculatePowerSum(5, 2, 3)

* power = 3^2 = 9 (exceeds 5, returns 0)
* Calls calculatePowerSum(5, 2, 4)

**Fourth Call:** calculatePowerSum(5, 2, 4)

* power = 4^2 = 16 (exceeds 5, returns 0)

**Backtrack to:** calculatePowerSum(9, 2, 3)

* power = 3^2 = 9
* Calls calculatePowerSum(9 - 9, 2, 4)

**Fifth Call:** calculatePowerSum(0, 2, 4)

* power == X, returns 1

**Backtrack to:** calculatePowerSum(10, 2, 2)

* Calls calculatePowerSum(10, 2, 3)

**Sixth Call:** calculatePowerSum(10, 2, 3)

* power = 3^2 = 9
* Calls calculatePowerSum(10 - 9, 2, 4)

**Seventh Call:** calculatePowerSum(1, 2, 4)

* power = 4^2 = 16 (exceeds 1, returns 0)

**DAY 14:**

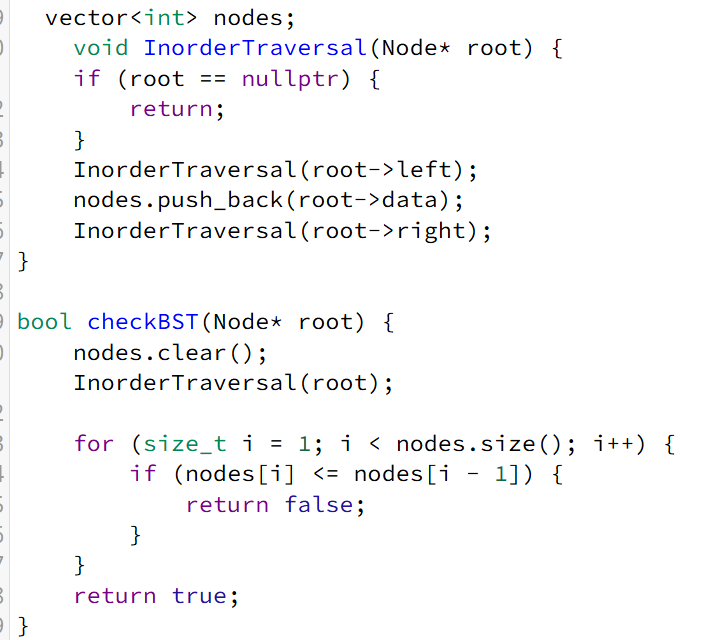
**IS THIS A BINARY SEARCH TREE:**

For the purposes of this challenge, we define a [binary tree](https://www.hackerrank.com/external_redirect?to=https://en.wikipedia.org/wiki/Binary_tree) to be a [binary search tree](https://www.hackerrank.com/external_redirect?to=https://en.wikipedia.org/wiki/Binary_search_tree) with the following ordering requirements:

* The data value of every node in a node's left subtree is *less than* the data value of that node.
* The data value of every node in a node's right subtree is *greater than* the data value of that node.

Given the root node of a binary tree, can you determine if it's also a binary search tree?

**SOURCE CODE:**

****

**DRY RUN:**

**Example:**

4

/ \

2 6

/ \ / \

1 3 5 7

**Step 1: Function Call**

* checkBST(root) is called with the root node (4).
* nodes.clear(); clears the global vector nodes.

**Step 2: Inorder Traversal Execution (InorderTraversal(root))**

* Recursive calls process the left subtree first, then root, then right subtree.
* Nodes visited in order: [1, 2, 3, 4, 5, 6, 7]

**Step 3: Checking if the Sequence is Strictly Increasing**

* Iterating over nodes vector:
  + 2 > 1
  + 3 > 2
  + 4 > 3
  + 5 > 4
  + 6 > 5
  + 7 > 6
* The sequence is strictly increasing → **BST is valid**.

**Output:**

* Returns **true**, confirming the tree is a BST.