1. **Snake Game:**

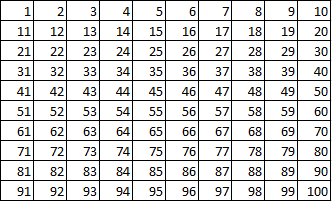
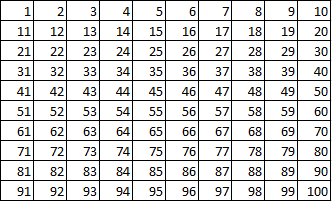
Snake is an old computer game. There are many versions, but all involve maneuvering a "snake" around the screen, trying to avoid running the snake into itself or an obstacle.

Simulate the game through the application, though it is very simplified version.

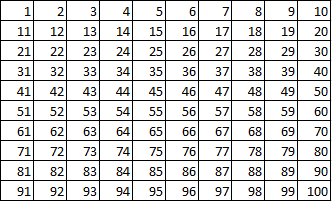
Consider a 10 X 10 board/matrix (refer the following diagram, where every integer value is indicating the cell number). The snake move Suppose the snake starts to move from cell number 15. The snake can move to 4 possible directions: north (N), east (E), south (S) and west (W). If the snake moves to N, then it actually moves to cell number 5. If it moves to S, then it actually moves to cell number 25. Similarly if the snake moves to E or W, which means snake is moving to cell number 26 or 24 respectively. So, with every move, the snake covers a cell and consequentially it leaves a trail along the path.

Now consider a sequence of moves, NEEESSWWNNN. That means the snake first moves from its initial position (cell number 15) to cell number 5, then takes a right turn and covers the cell number 6, 7 and 8. After that it takes a down turn and covers 18 and 28, before it takes a left turn and covers cell number 27 and 26. At last it takes again upward turn and covers cell number 16 first and meets the same trail, i.e., cell number 6. It means the snake has run on to itself. In this case, the snake made 9 successful moves and on the 10th move it runs into itself.

Again consider the sequence of moves, SSEEEEEEEE. That means from starting position (cell number 15, the snake moves downward till cell 35 and then takes a right turn to cover cell number 36 to 40, before it runs out of board. Before running out of board it made 7 moves.



Again consider sequence of moves: WWSSSEEEESSSS. That means the snake first made a left run from start position (cell number 15) and covered cell number14 and 13 and then a downward move and continued till cell number 43. Then it took a right turn to go till cell number 47 and again moved downwards and continued till cell number 87. This time the snake neither did run onto itself nor run out of board. It completed all the 13 moves successfully.



**Description:** Complete the given method to return the number of moves as well as the result of the moves (i.e., “SUCCESS” OR “RANOUTOFBOARD” OR “RANONTOITSELF”) for a given sequence of moves in string format.

**Input:** Start position as integer value and sequence of moves as string value

**Output:** Number of moves (made successfully or made before it ran out of board or before it ran onto itself) along with the result of the moves (i.e., “SUCCESS” OR “RANOUTOFBOARD” OR “RANONTOITSELF”).

|  |  |  |
| --- | --- | --- |
| UTC | Input | Output |
| UTC04\_01 | 53 and SSSSEEENNEEEEESS | {13, RANOUTOFBOARD } |
| UTC04\_02 | 3 and SSSSEESSSSWNNNNNNNN | {14, RANONTOITSELF} |
| UTC04\_03 | 15 and WWSSSEEEESSSS | {13, SUCCESS} |

1. Binary Tree

A binary tree is a tree data structure in which each node has at most two child nodes, usually distinguished as "left" and "right". Nodes with children are parent nodes, and child nodes may contain references to their parents. Outside the tree, there is often a reference to the "root" node (the ancestor of all nodes), if it exists. Any node in the data structure can be reached by starting at root node and repeatedly following references to either the left or right child.

Representing a binary tree:

A binary tree can be represented using a set of linked nodes. Each node contains a value and two links named left and right that reference the left child and right child.



Tree Traversal

Tree traversal is the process of visiting each node in the tree exactly once. There are several ways to traverse a tree:

 The inorder traversal follows the recursive visitation sequence: left - root - right subtree

 The postorder traversal follows the recursive visitation sequence: right subtree - left - root

 The preorder traversal follows the recursive visitation sequence: left subtree - root - right

For example, in the following tree:

 inorder is: {45, 55, 57, 60, 67, 100, 107 }

 postorder is: {45, 57, 55, 67, 107, 100, 60 }.

 preorder is: {60, 55, 45, 57, 100, 67, 107}

Problem Statement: Complete the functionalities of provided BinaryTree class to store the elements in a binary tree and return the data by traversing them using inorder, postorder and preorder

|  |  |  |
| --- | --- | --- |
| Test Case | Input | Output |
| 1 | Add elements {60, 55, 100, 57, 107,67,45}  BinaryTree.GetData("inorder") | {45, 55, 57, 60, 67, 100, 107 } |
| 2 | Add elements {5,2,8,7,9,10}  BinaryTree.GetData("inorder") | {2,5,7,8,9,10 } |
| 3 | Add elements: {60, 55, 100, 57, 107,67,45}  BinaryTree.GetData("postorder") | {45, 57, 55, 67, 107, 100, 60 } |
| 4 | Add elements: {60, 55, 100, 57, 107,67,45}  BinaryTree.GetData("preorder") | {60, 55, 45, 57, 100, 67, 107} |

1. **Co-Prime Numbers**

Two numbers are said to be co-primes, if the factors of the given two numbers have nothing in common except 1 (Factors of a number are nothing but the numbers between 1 and the number itself which can divide the given number evenly).

For an example, the numbers 6 and 25 have nothing but 1 in common, since factors of 6 is {1, 2, 3, 6} and factors of 25 are {1, 5, 25}. On comparing the factors of 6 and 25, it is evident that both numbers have only 1 as a common factor. So, 6 and 25 are co-primes. Again, 12 and 16 are not co-prime, since the factors of 12 are {1, 2, 3, 4, 6, 12} and the factors of 16 are {1, 2, 4, 8, 16}. On comparing the factors of 12 and 16, it is evident that both the numbers have {1, 2, 4} as common factors. So, they are not co-prime.

**Description**: Write a code logic in the given method to return a Boolean value, indicating whether the pair of two numbers, supplied in the method, is co-prime or not.

**Input**: Input is a pair of positive integers

**Output**: Output is a Boolean value indicating the given input is a co-prime pair or not.

|  |  |  |
| --- | --- | --- |
| Test case | Sample Input | Sample output |
| UTC01\_01 | 12, 4 | False |
| UTC01\_02 | 729, 33 | False |
| UTC01\_03 | 0, 0 | False |
| UTC01\_04 | 15, 22 | True |

Figure 1.1 – Co-prime Numbers

1. **Sum Of Digits In Array**

Consider an array which is collection of only positive integers. For an example, an array contains following numbers {11, 22, 33, 10, 23, 42, 53, 56 }. For any given number, one needs to check whether the supplied number 32 is equal to the summation of any two numbers present in the array or not. In this example, the summation of two elements 10 and 22, present in the array, is equal to the value 32, which is supplied by the user. Again, for a given value, say, 192, there are no two numbers present in the array, whose summation is equal to 192.

**Description:** Write code logic in the given method to find out whether a given number is equal to summation of any two numbers present in the array or not.

**Input:** An array of positive integers and any positive number.

**Output:** A Boolean value indicating whether summation of any two numbers, present in the array, is equal to the given positive number or not.

|  |  |  |
| --- | --- | --- |
| Test case | Sample Input | Sample output |
| UTC02\_01 | { 12, 14, 78, 23, 90, 1, 4, 8} and 26 | True |
| UTC02\_02 | Null and 6 | False |
| UTC02\_03 | { 0, 0, 0 } and 0 | True |
| UTC02\_04 | {77, 99, 65, 45, 52} and 177 | False |
| UTC02\_05 | { 1, 2, 3, 7, 4, 0} and 0 | False |

Figure 1.2 – Sum of Digits in Array