Array & Hashing (Easy–Medium)

Problem: Two Sum Variant

You are given an array of integers nums and an integer target.

Task

Return the indices of the two numbers such that they add up to target.

Example

Input

```
nums = [2, 7, 11, 15], target = 9
```

Output

```
[0, 1]
```

Constraints

1. Exactly one solution exists
2. Do not use the same element twice

What this tests

- Hash maps
- Time complexity optimization
- Edge-case handling

In [1]:
```
'''
big idea
    i need a hashmap that will store the first obsereved variable in the list
        hashmap = {number_in_list: its_index}
    i need an answer element
        finds the value that when subtracted from the target gives a value in the hashmap

            ans = target - current value being pointed
Process
    create an empty hashmap {key(number): value (index)}
    loop through the range of numbers that represent the indices of the values in the list
        create an answer variable (target - list[current_index])
        if the answer is in the hashmap
            return the [value represented by ans, the current_indes]
        else
            /-- because the current number has no answer value --/
            add the current number: current index into the hashmap
    Account for an edge case say there is no value (noting that the problem states that there is always a solution)
        this is just good practice
    return None (no solution is fount) /-- also breaks the function

'''

def twoSumVariant(nums: list[int], target: int):
    """
    Find the index of the two numbers that add up to the target from the nums array

    Not using the same element twice

    :param nums: an array of integers
    :type nums: list[int]
    :param target: the number target
    :type target: int
```

```
    """
    # track the last observed value than can add up to the target
    out = {}

    # find the current value that gives the target
    for i in range(len(nums)):
        ans = target - nums[i]
        # edge case: the two values have been found
        if ans in out:
            return [out[ans], i]
        # add the number that does not have a value (ans) to out
        else:
            out[nums[i]] = i
    return None

print(twoSumVariant([3, 2, 4], 6))
```

```
[1, 2]
```

In [2]:

```
a = [

twoSumVariant([2, 7, 11, 15], 9) == [0, 1],

twoSumVariant([3, 2, 4], 6) == [1, 2],

twoSumVariant([-1, -2, -3, -4, -5], -8) == [2, 4],

twoSumVariant([3, 3], 6) == [0, 1],

twoSumVariant([1, 2, 3], 7) is None,

twoSumVariant([0, 4, 3, 0], 0) == [0, 3]
]

print(a)
```

```
[True, True, True, True, True, True]
```

Strings & Stack (Medium)

Problem: Valid Parentheses

Given a string `s` containing only the characters `(, ), {, }, [, ],` determine if the input string is valid.

Rules

1. Open brackets must be closed by the same type
2. Brackets must be closed in the correct order

Example

Input:

```
"{[()]}"
```

Output:

```
true
```

Input:

```
"{[(])}"
```

Output:

```
false
```

What this tests

- Stack usage

- Pattern matching
- Clean logic

In [3]:
```
'''
main idea is to add opening brackets to the stack and await the closing bracket then remove the opening bracket fro

{[(<)]}

check = []
loop through the string
    push the opening brackets
    check becomes [{, [, (, < ]

    if the next variable is > pop the last element added
        in this case the next character is not > so return false

{[()]}

in this new case

loop to push the opening brackets
    check = [{, [, (]
    the loop then encounters )
    so pop ( from the check
        check = [{, []
        on the next run it encounters ]
    pop [
        check = [{]
        ...
    when check is empty the opening brackets have their closing bracket later in the code

'''

def validParenthesis(s :str):
    """
    check if a string is a valid bracket

    input s: a string
    type s: str
    """
```

```
    brackets = {
        "}": "{",
        "]": "[",
        ")":"(",
        ">":"<"
       }
    check = []
    # ideas
    # } ] ) > can not be added without the corresponding opening bracket
    # edge case: opening brackets are present
    for char in s:
        if char in "{[(<": # opening brackets can be added to the stack
            check.append(char)
        # edge case: the character is a closing bracket without the opening bracket\
        elif char in brackets:
            # empty stacks with the starting closing brackets are automatically false
            if not check or check[-1] != brackets[char]:
                return False
            check.pop()

    # if all the brackets are matched. The check stack is empty return true
    return len(check) == 0
```

In [4]:
```
b = [validParenthesis("()") is True,
     validParenthesis("({[]})") is True,
     validParenthesis("()[]{}") is True,
     validParenthesis("(]") is False,
     validParenthesis("(()") is False,
     validParenthesis("())") is False,
     validParenthesis("") is True,
     validParenthesis("(") is False
]

print(b)
```

```
[True, True, True, True, True, True, True, True]
```

Trees / Recursion (Medium)

Problem: Maximum Depth of a Binary Tree

Given the root of a binary tree, return its maximum depth.

Definition

The maximum depth is the number of nodes along the longest path from the root down to the farthest leaf.

Example

Input:

```
    3
   / \
  9  20
     / \
    15  7
```

Output:

```
3
```

In [5]:
```
'''
Idea
BFS - has an issue with too much memory consumption if the tree has too many subtrees

i need to inOrder DFS first so that i can find the max depth between left and right subtrees of the root node

loop through the tree as far left as possible adding the current depth. (left depth)
loop through the tree as far right as possible adding the current depth. (right depth)

get the max depth between left and right

edge cases: the tree is empty -> depth 0
            the tree has 1 element -> depth 1
```

```
current depth starts at 0 and increases from there if the (sub)tree has any node or child nodes

I need:
    1. a helper inOrder function. can be referenced in my notes on trees
    2. a max tree depth function. to run the inOrder function
'''

def maxTreeDepth(rootNode) -> int:
    return inOrder(rootNode, 0)

def inOrder(node, currentDepth) -> int:
    if node is None:
        return currentDepth  # Return the current depth when a leaf is reached
    leftDepth = inOrder(node.left, currentDepth + 1)
    rightDepth = inOrder(node.right, currentDepth + 1)
    return max(leftDepth, rightDepth)
```

In [6]:

```
class TreeNode:
    def __init__(self, val=0, left=None, right=None):
        self.val = val
        self.left = left
        self.right = right


def test_empty_tree():
    return maxTreeDepth(None) == 0

def test_single_node():
    root = TreeNode(1)
    return maxTreeDepth(root) == 1

def test_left_skewed_tree():
    # 1
    # |
    # 2
    # |
    # 3
    root = TreeNode(1,
        TreeNode(2,
```

```
            TreeNode(3)
        )
    )
    return maxTreeDepth(root) == 3

def test_right_skewed_tree():
    # 1
    #  \
    #   2
    #    \
    #     3
    root = TreeNode(1, None,
        TreeNode(2, None,
            TreeNode(3)
        )
    )
    return maxTreeDepth(root) == 3

def test_balanced_tree():
    #       1
    #      / \
    #     2   3
    #    / \
    #   4   5
    root = TreeNode(1,
        TreeNode(2,
            TreeNode(4),
            TreeNode(5)
        ),
        TreeNode(3)
    )
    return maxTreeDepth(root) == 3

def test_unbalanced_tree():
    #       1
    #      / \
    #     2   3
    #    /
    #   4
    #  /
    # 5
```

```
    root = TreeNode(1,
        TreeNode(2,
            TreeNode(4,
                TreeNode(5)
            )
        ),
        TreeNode(3)
    )
    return maxTreeDepth(root) == 4

[test_empty_tree(), test_balanced_tree(), test_left_skewed_tree(), test_right_skewed_tree(), test_single_node(), te
```

Out[6]: [True, True, True, True, True, True]