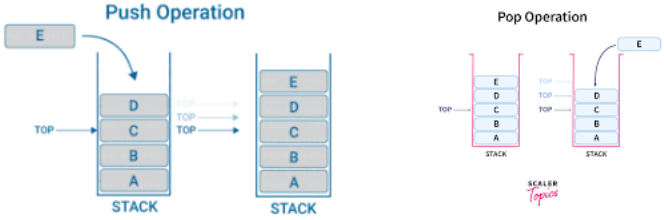
STACK

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) order, meaning that the last element added to the stack is the first one to be removed. Think of it as a stack of plate you can only add or remove plates from the top. In Python, a stack can be implemented using lists or the collections.

Stack Operations:
push(Data): Insert the element in the stack.
pop(): Remove and return the topmost element of the stack.
top(): Return the topmost element of the stack
size(): Return the number of remaining elements in the stack.
IfEmpty(): Returns TRUE if the stack is empty, else return FALSE



Example: Basic operations on Stack.

Code:

```
class Stack:
    def __init__(self):
        self.top = -1
        self.size = 1000
        self.arr = [0] * self.size

    def push(self, x: int) -> None:
        self.top += 1
        self.arr[self.top] = x

    def pop(self) -> int:
        x = self.arr[self.top]
        self.top -= 1
        return x

    def Top(self) -> int:
        return self.arr[self.top]
```

```
    def Size(self) -> int:
        return self.top + 1

if __name__ == "__main__":
    s = Stack()
    s.push(1)
    s.push(3)
    s.push(4)
    print("Top of stack is before deleting any element", s.Top())
    print("Size of stack before deleting any element", s.Size())
    print("The element deleted is", s.pop())
    print("Size of stack after deleting an element", s.Size())
    print("Top of stack after deleting an element", s.Top())
```

Output:

Top of stack is before deleting any element 4
Size of stack before deleting any element 3
The element deleted is 4
Size of stack after deleting an element 2
Top of stack after deleting an element 3

LeetCode Problems

Problem: Number of Students Unable to Eat Lunch

The school cafeteria offers circular and square sandwiches at lunch break, referred to by numbers 0 and 1 respectively. All students stand in a queue. Each student either prefers square or circular sandwiches.

The number of sandwiches in the cafeteria is equal to the number of students. The sandwiches are placed in a stack. At each step:

If the student at the front of the queue prefers the sandwich on the top of the stack, they will take it and leave the queue.
Otherwise, they will leave it and go to the queue's end.
This continues until none of the queue students want to take the top sandwich and are thus unable to eat.

You are given two integer arrays students and sandwiches where sandwiches[i] is the type of the ith sandwich in the stack (i = 0 is the top of the stack) and students[j] is the preference of the jth student in the initial queue (j = 0 is the front of the queue). Return the number of students that are unable to eat.

Example 1:

Input: students = [1,1,0,0], sandwiches = [0,1,0,1]
Output: 0

Code:

```
class Solution:
    def countStudents(self, students: List[int], sandwiches: List[int]) -> int:
```

```
    for i in sandwiches:
        if i in students:
            students.remove(i)
        else:
            break
    return len(students)
```

Explanation:

sandwiches = [1, 0, 0, 1]
students = [1, 1, 0, 0]

→ (i = 1):
1 in students: True
Remove 1 from students.
New state of students = [1, 0, 0]

→ (i = 0):
0 in students: True
Remove 0 from students.
New state of students = [1, 0]

→ (i = 0):
0 in students: True
Remove 0 from students.
New state of students = [1]

→ (i = 1):
1 in students: True
Remove 1 from students.
New state of students = []

Return the Result:
return len(students): len([]) is 0.

Output: 0

-→Time Complexity: O(n * m)
-→Space Complexity: O(n + m)

Problem: Final Prices with a Special Discount in a Shop

You are given an integer array prices where prices[i] is the price of the ith item in a shop.

There is a special discount for items in the shop. If you buy the ith item, then you will receive a discount equivalent to prices[j] where j is the minimum index such that j > i and prices[j] <= prices[i]. Otherwise, you will not receive any discount at all.

Return an integer array answer where answer[i] is the final price you will pay for the ith item of the shop, considering the special discount.

Example 1:

Input: prices = [8,4,6,2,3]
Output: [4,2,4,2,3]
Code:

```
class Solution:
    def finalPrices(self, prices: List[int]) -> List[int]:
        n = len(prices)
        answer = [0] * n

        for i in range(n):
            discount = 0
            for j in range(i + 1, n):
                if prices[j] <= prices[i]:
                    discount = prices[j]
                    break
            answer[i] = prices[i] - discount

        return answer
```

Explanation:

prices = [8, 4, 6, 2, 3]
n = 5
→ (i = 0):
prices[i] = 8
i + 1 (1) to n - 1 (4):
prices[1] = 4 (4 ≤ 8) -> discount = 4
answer[0] = 8 - 4 = 4

→ (i = 1):
prices[i] = 4
index i + 1 (2) to n - 1 (4):
prices[2] = 6 (6 > 4) -> continue
prices[3] = 2 (2 ≤ 4) -> discount = 2
answer[1] = 4 - 2 = 2

→ (i = 2):
prices[i] = 6
i + 1 (3) to n - 1 (4):
prices[3] = 2 (2 ≤ 6) -> discount = 2
answer[2] = 6 - 2 = 4

→ (i = 3):
prices[i] = 2
index i + 1 (4) to n - 1 (4):
prices[4] = 3 (3 > 2) -> no discount
answer[3] = 2 - 0 = 2

→ (i = 4):
prices[i] = 3
No elements to check after i + 1, so no discount
answer [4] = 3 - 0 = 3

Output: [4, 2, 4, 2, 3]

-→ Time complexity: $O(n^2)$.
-→ Space complexity: O(n)

Problem:

The next greater element of some element x in an array is the first greater element that is to the right of x in the same array.

You are given two distinct 0-indexed integer arrays nums1 and nums2, where nums1 is a subset of nums2.

For each 0 <= i < nums1.length, find the index j such that nums1[i] == nums2[j] and determine the next greater element of nums2[j] in nums2. If there is no next greater element, then the answer for this query is -1.

Return an array ans of length nums1.length such that ans[i] is the next greater element as described above.

Example 1:

Input: nums1 = [4,1,2], nums2 = [1,3,4,2]
Output: [-1, 3, -1]

Code:

```
class Solution:
    def nextGreaterElement(self, nums1: List[int], nums2: List[int]) -> List[int]:
        out = []
        for num in nums1:
            n = nums2.index(num)
            li = nums2[n+1:]
            found = False
            for nu in li:
                if nu > num:
                    out.append(nu)
                    found = True
                    break
```

```
            if not found:
                out.append(-1)
        return out
```

Explanation:

→ nums1 = 4
nums2 =2.
nums2[3:]: =sublist is [2].
Compare 4 with elements in sublist [2]: No element is greater than 4.
out = [-1].

→ nums1= 1
nums2: = 0.
Create nums2[1:]: sublist is [3, 4, 2].
Compare 1 with elements in sublist [3, 4, 2]: First element 3 is greater than 1.
out = [-1, 3].

→ nums1 = 2
nums2 = 3.
Create nums2[4:]: sublist is [] (empty).
No elements to compare with 2.
out is [-1, 3, -1].

Output: [-1, 3, -1]

-→ Time complexity: $O(n^2)$.
-→ Space complexity: O(n1 + n2)

Problem: Implement Stack using Queues

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (push, top, pop, and empty).
Implement the MyStack class:

- void push(int x) Pushes element x to the top of the stack.
- int pop() Removes the element on the top of the stack and returns it.
- int top() Returns the element on the top of the stack.
- boolean empty() Returns true if the stack is empty, false otherwise.

Input
["MyStack", "push", "push", "top", "pop", "empty"]
[[], [1], [2], [], [], []]
Output
[null, null, null, 2, 2, false]

Code:

```
from collections import deque
```

```
class MyStack:

    def __init__(self):
        self.q1 = deque()
        self.q2 = deque()

    def push(self, x: int) -> None:
        self.q1.append(x)
    def pop(self) -> int:
        # Move all elements except the last one from q1 to q2
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        # The last element in q1 is the top of the stack
        top_element = self.q1.popleft()
        # Swap the names of q1 and q2
        self.q1, self.q2 = self.q2, self.q1
        return top_element

    def top(self) -> int:
        # Move all elements except the last one from q1 to q2
        while len(self.q1) > 1:
            self.q2.append(self.q1.popleft())
        # Peek the last element in q1 which is the top of the stack
        top_element = self.q1[0]
        # Move it to q2
        self.q2.append(self.q1.popleft())
        # Swap the names of q1 and q2
        self.q1, self.q2 = self.q2, self.q1
        return top_element

    def empty(self) -> bool:
        return not self.q1
```

Explanation:

→ q1 and q2 = null. Output: null.

→ q1 = [1], q2 = []. Output: null.

→ q1 = [1, 2], q2 = []. Output: null

→ Moves elements from q1 to q2.
q1 = [2] and q2 = [1].
The top element is 2.

q1 = [] and q2 = [1, 2].
Swaps q1 and q2, resulting in q1 = [1, 2] and q2 = []. Output: 2.

→ Moves elements from q1 to q2 .
q1 = [2] and q2 = [1].
The top element 2
Swaps q1 and q2, resulting in q1 = [1] and q2 = []. Output: 2

→ Checks if q1 is empty.
State: q1 = [1], q2 = []. Output: false.

Output: [null, null, null, 2, 2, false]

-→ Time Complexity: O(n)
-→Space Complexity: O(n)

Problem: **Valid Parentheses**
Given a string s containing just the characters '(', ')', '{', '}', '[' and ']', determine if the input string is valid.

An input string is valid if:

Open brackets must be closed by the same type of brackets.
Open brackets must be closed in the correct order.
Every close bracket has a corresponding open bracket of the same type.

Example 1:

Input: s = "()"
Output: true

Code:

```
class Solution:
    def isValid(self, s: str) -> bool:
        st = []
        for it in s:
            if it == '(' or it == '{' or it == '[':
                st.append(it)
            else:
                if len(st) == 0:
                    return False
                ch = st[-1]
                st.pop()
                if (it == ')' and ch == '(') or (it == ']' and ch == '[') or (it == '}' and ch == '{'):
                    continue
                else:
```

```
                return False
        return len(st) == 0
```

Explanation:

→ s = "()". st = [].
→ st = ['(']
→ we check: if stack == empty. It is not empty,
We pop the top element from the stack, which is (.
We check if (matches with), which it does.
st = []
→ if stack == empty: return 'True'

Output: s = "()" is True

-→ Time Complexity: O(n)
-→Space Complexity: O(n)

Problem: Mini Stack

Design a stack that supports push, pop, top, and retrieving the minimum element in constant time.

Implement the MinStack class:

MinStack() initializes the stack object.
void push(int val) pushes the element val onto the stack.
void pop() removes the element on the top of the stack.
int top() gets the top element of the stack.
int getMin() retrieves the minimum element in the stack.
You must implement a solution with O(1) time complexity for each function.

Example 1:

Input
["MinStack","push","push","push","getMin","pop","top","getMin"]
[[], [-2], [0], [-3], [], [], [], []]

Code:

```
class MinStack:
    def __init__(self):
        self.stack = []
        self.mini = float('inf')
```

```
    def push(self, value: int) -> None:
        val = value
        if not self.stack:
            self.mini = val
            self.stack.append(val)
        else:
            if val < self.mini:
                self.stack.append(2 * val - self.mini)
                self.mini = val
            else:
                self.stack.append(val)

    def pop(self) -> None:
        if not self.stack:
            return
        el = self.stack.pop()
        if el < self.mini:
            self.mini = 2 * self.mini - el

    def top(self) -> int:
        if not self.stack:
            return -1
        el = self.stack[-1]
        if el < self.mini:
            return self.mini
        return el

    def getMin(self) -> int:
        return self.mini

# Your MinStack object will be instantiated and called as such:
# obj = MinStack()
# obj.push(val)
# obj.pop()
# param_3 = obj.top()
# param_4 = obj.getMin()
```

Explanation:

→ stack = [], mini = inf
→ push(-2), stack = [-2], mini = -2

→ push(0), stack = [-2, 0], mini = -2
→ push(-3), stack = [-2, 0, -4], mini = -3
→ getMin(), stack = [-2, 0, -4], mini = -3
→ pop(-4) , -4 < -3, mini = 2 * (-3) - (-4) = -6 - (-4) = -2.
stack = [-2, 0], mini = -2
→ top() = 0, stack = [-2, 0], mini = -2
→ getMin() = -2, stack = [-2, 0], mini = -2
Output: [null, null, null, null, -3, null, 0, -2]

-→ Time Complexity: O(n)
-→Space Complexity: O(n)

Challenges faced to solve stack leetcode problems:

When solving stack-related problems on LeetCode, especially those classified as easy and medium level, I encountered several challenges. Here are some common challenges.

→ **Understanding Problem Statements:**
Some problem descriptions can be tricky to interpret, especially if they involve complex scenarios where stack operations are crucial. Deciphering what the problem is asking for and identifying that a stack is the right data structure to use can be challenging.

→ **Identifying Edge Cases:**
Handling edge cases, such as empty inputs, single-element stacks, or unexpected operations, can be difficult. Missing these edge cases often leads to incorrect solutions or runtime errors.

→ **Balancing Time and Space Complexity:**
Finding an optimal solution that balances both time and space complexity is often challenging, encountering situations where a solution works but isn't efficient enough to pass all test cases due to time limits.

→ **Debugging Recursive Stack Solutions:**
When stacks are used in conjunction with recursion, debugging becomes more complex. Tracking the flow of recursive calls and understanding the state of the stack at each point can be difficult, leading to confusion and errors.

→ **Managing Stack Overflow:**
In problems where recursion or extensive stack operations are required, stack overflow can occur if the depth of recursion or number of operations exceeds the stack's capacity. Preventing and managing these situations requires careful planning and implementation.

→ **Implementing Custom Stack Operations:**

Some problems require to implement custom stack operations, such as a stack that supports additional operations like `getMin()` or `getMax()`. Balancing these operations with standard push and pop operations without increasing complexity can be tricky.