Stack

1. Implement Queue using Stacks (Easy)

The task is to implement a **queue** using two **stacks**.
A **queue** follows the **First In First Out (FIFO)** principle: the element inserted first is the one removed first.
A **stack** follows the **Last In First Out (LIFO)** principle: the last element inserted is the one removed first.

We need to create a `MyQueue` class that supports the following methods:

1. `push(x: int)` - Pushes element `x` to the end of the queue.
2. `pop()` - Removes and returns the element at the front of the queue.
3. `peek()` - Returns the element at the front of the queue without removing it.
4. `empty()` - Returns `true` if the queue is empty, `false` otherwise.

Key Insight

To simulate queue operations using **two stacks**, we will:

1. Use `stack_in` for incoming elements.
2. Use `stack_out` to reverse the order of elements whenever we need to **pop** or **peek**.

```
class MyQueue:

    def __init__(self):
        # Initialize two stacks
        self.stack_in = []
        self.stack_out = []

    def push(self, x: int) -> None:
        # Push the new element onto the input stack
        self.stack_in.append(x)
```

```
    def pop(self) -> int:
        # If the output stack is empty,
        # transfer elements from input stack to output stack
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())

        # Pop the top element from output stack and return it
        return self.stack_out.pop()

    def peek(self) -> int:
        # If the output stack is empty,
        # transfer elements from input stack to output stack
        if not self.stack_out:
            while self.stack_in:
                self.stack_out.append(self.stack_in.pop())

        # Return the top element from output stack without removing it
        return self.stack_out[-1]

    def empty(self) -> bool:
        # The queue is empty if both stacks are empty
        return not self.stack_in and not self.stack_out
```

```
# Create an instance of MyQueue
queue = MyQueue()

# Push elements into the queue
queue.push(1)
queue.push(2)
queue.push(3)

# Peek should return the first element entered (1)
print(queue.peek())  # Output: 1

# Pop should remove and return the first element entered (1)
print(queue.pop())   # Output: 1

# Check if the queue is empty
print(queue.empty()) # Output: False
```

```python
# Pop remaining elements
print(queue.pop())   # Output: 2
print(queue.pop())   # Output: 3

# Now the queue should be empty
print(queue.empty()) # Output: True
```

```
1
1
False
2
3
True
```

Initialization

- Two stacks, `stack_in` and `stack_out`, are initialized.
- `stack_in` handles **push** operations.
- `stack_out` is used for **pop** and **peek** operations after reversing the order of elements.

Push Operation

- Elements are directly added to `stack_in`.

Pop/Peek Operations

1. If `stack_out` is empty, all elements from `stack_in` are moved to `stack_out`. This reverses the order, so the oldest element is on top of `stack_out`.
2. For `pop`, remove and return the top element of `stack_out`.
3. For `peek`, return the top element of `stack_out` without removing it.

Empty Check

- The queue is empty only if **both** `stack_in` and `stack_out` are empty.

Input

```
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
```

Output

```
[null, null, null, 1, 1, false]
```

Explanation

1. `MyQueue` initializes the queue.
   Output: `null`.
2. `push(1)` inserts `1` into the queue.
   Output: `null`.
3. `push(2)` inserts `2` into the queue.
   Output: `null`.
4. `peek()` returns the front element, which is `1`.
   Output: `1`.
5. `pop()` removes and returns the front element, which is `1`.
   Output: `1`.
6. `empty()` checks if the queue is empty. It returns `false` since `2` is still in the queue.
   Output: `false`.

2. Valid Parentheses (Easy)

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

An input string is valid if:
1. Open brackets must be closed by the same type of brackets.
2. Open brackets must be closed in the correct order.

Example 1:

Input: `s = "()"`
Output: `true`

Example 2:

Input: `s = "()[]{}"`
Output: `true`

Example 3:

Input: `s = "(]"`
Output: `false`

Solution Explanation

The problem requires us to check for matching pairs of parentheses while also ensuring proper nesting. A stack is an ideal data structure for this because of its **Last-In-First-Out (LIFO)** property.

Algorithm

1. Use a stack to keep track of opening brackets.
2. Traverse the string `s` character by character:
    - If the character is an **opening bracket** (`'('`, `'{'`, `'['`), push it onto the stack.
    - If the character is a **closing bracket** (`')'`, `'}'`, `']'`):
        - Check if the stack is empty. If it is, return `false` (there is no corresponding opening bracket).
        - Otherwise, pop the top element from the stack and check if it matches the current closing bracket. If not, return `false`.
3. After traversing the string, the stack should be empty. If not, return `false` (there are unmatched opening brackets).

```
def isValid(s: str) -> bool:
    # A dictionary to map closing brackets
    # to their corresponding opening brackets
    bracket_map = {')': '(', '}': '{', ']': '['}
    # Stack to keep track of opening brackets
    stack = []

    # Traverse each character in the string
    for char in s:
        if char in bracket_map:  # If it's a closing bracket
            # Check the top of the stack (last opened bracket
```

```
            # Pop if stack isn't empty, else use dummy '#'
            top_element = stack.pop() if stack else '#'
            # Check if the popped element matches the expected opening bracket
            if bracket_map[char] != top_element:
                return False
        else:
            # Push opening brackets onto the stack
            stack.append(char)

    # If stack is empty at the end, all brackets are matched
    return not stack

s = "([{}])"
print(isValid(s))  # Output: True
```

```
True
```

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

1. Stack: `[]`
2. Process `'('` → Stack: `['(']`
3. Process `'['` → Stack: `['(', '[']`
4. Process `'{'` → Stack: `['(', '[', '{']`
5. Process `'}'` → Match found → Pop `'{'` → Stack: `['(', '[']`
6. Process `']'` → Match found → Pop `'['` → Stack: `['(']`
7. Process `')'` → Match found → Pop `'('` → Stack: `[]`

Since the stack is empty, the string is valid.

3. Min Stack (Medium)

Design a stack that supports the following operations in constant time: 1. **push(x)**: Push element x onto the stack. 2. **pop()**: Removes the element on the top of the stack. 3. **top()**: Get the top element. 4. **getMin()**: Retrieve the minimum element in the stack.

Example Input and Output

Input:

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

Output:

```
[null, null, null, null, -3, null, 0, -2]
```

Explanation:

- `MinStack` initializes the stack.
- `push(-2)` adds -2 to the stack.
- `push(0)` adds 0 to the stack.
- `push(-3)` adds -3 to the stack.
- `getMin()` returns -3 as it is the smallest element in the stack.
- `pop()` removes the top element (-3).
- `top()` returns the top element (0).
- `getMin()` now returns -2, the smallest element in the stack.

```python
class MinStack:
    def __init__(self):
        # Main stack to store elements
        self.stack = []
        # Helper stack to track the minimum element at each level
        self.min_stack = []

    def push(self, val: int) -> None:
        # Push the value onto the main stack
        self.stack.append(val)
        # Push the minimum value onto the min stack
        # If the min_stack is empty, push the current value
        # Otherwise, push the smaller value between the current value and
        # the top of the min_stack
        if not self.min_stack:
            self.min_stack.append(val)
        else:
            self.min_stack.append(min(val, self.min_stack[-1]))
```

```
    def pop(self) -> None:
        # Pop the top element from both the main stack and the min stack
        if self.stack:
            self.stack.pop()
            self.min_stack.pop()

    def top(self) -> int:
        # Return the top element of the main stack
        if self.stack:
            return self.stack[-1]
        return None  # Return None if the stack is empty (edge case)

    def getMin(self) -> int:
        # Return the top element of the min stack,
        # which is the current minimum
        if self.min_stack:
            return self.min_stack[-1]
        return None  # Return None if the min stack is empty (edge case)

# Example Usage
operations = [
    "MinStack", "push", "push", "push", "getMin", "pop", "top", "getMin"
]
values = [[], [-2], [0], [-3], [], [], [], []]
output = []

# Simulate the operations
min_stack = None
for op, val in zip(operations, values):
    if op == "MinStack":
        min_stack = MinStack()
        output.append(None)
    elif op == "push":
        min_stack.push(val[0])
        output.append(None)
    elif op == "pop":
        min_stack.pop()
        output.append(None)
    elif op == "top":
        output.append(min_stack.top())
    elif op == "getMin":
        output.append(min_stack.getMin())
```

```
print(output)  # Output: [None, None, None, None, -3, None, 0, -2]
```

```
[None, None, None, None, -3, None, 0, -2]
```

Key Points:

1. **Two Stacks Approach**:
   - **Main stack** (`stack`) is used to store all elements.
   - **Min stack** (`min_stack`) tracks the smallest element up to the current point.
2. **Push Operation**:
   - Always maintain the smallest value in the `min_stack` by comparing the new value with the top of `min_stack`.
3. **Pop Operation**:
   - Remove elements from both `stack` and `min_stack` simultaneously to keep them synchronized.
4. **Top Operation**:
   - Directly retrieve the top of `stack`.
5. **GetMin Operation**:
   - The top of `min_stack` always holds the current minimum.

Dry Run for the Sample Input

Input:

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

Execution:

1. `MinStack`: Initialize `stack = []`, `min_stack = []`.
   - Output: `None`.
2. `push(-2)`:
   - Add -2 to `stack`: `stack = [-2]`.

- Add -2 to `min_stack`: `min_stack = [-2]`.
- Output: `None`.

3. `push(0)`:
   - Add 0 to `stack`: `stack = [-2, 0]`.
   - Add the smaller of 0 and `min_stack[-1]` (-2): `min_stack = [-2, -2]`.
   - Output: `None`.
4. `push(-3)`:
   - Add -3 to `stack`: `stack = [-2, 0, -3]`.
   - Add the smaller of -3 and `min_stack[-1]` (-2): `min_stack = [-2, -2, -3]`.
   - Output: `None`.
5. `getMin()`:
   - Return `min_stack[-1]`: -3.
   - Output: `-3`.
6. `pop()`:
   - Remove -3 from `stack`: `stack = [-2, 0]`.
   - Remove -3 from `min_stack`: `min_stack = [-2, -2]`.
   - Output: `None`.
7. `top()`:
   - Return `stack[-1]`: 0.
   - Output: `0`.
8. `getMin()`:
   - Return `min_stack[-1]`: -2.
   - Output: `-2`.

Final Output:

```
[None, None, None, None, -3, None, 0, -2]
```

4. Evaluate Reverse Polish Notation (Medium)

You are given an array of strings `tokens` that represents an arithmetic expression in a reverse Polish notation.

Evaluate the expression. Return an integer that represents the value of the expression.

Note: - The valid operators are `+`, `-`, `*`, and `/`. - Each operand may be an integer or another expression. - Division between two integers should truncate toward zero. - The given `tokens` are always valid. Each token is either an operator or an integer. There is no need to validate the input.

Example 1:

Input:

```
tokens = ["2", "1", "+", "3", "*"]
```

Output:

```
9
```

Explanation:

```
((2 + 1) * 3) = 9
```

Example 2:

Input:

```
tokens = ["4", "13", "5", "/", "+"]
```

Output:

```
6
```

Explanation:

```
(4 + (13 / 5)) = 6
```

Example 3:

Input:

```
tokens = ["10", "6", "9", "3", "/", "-", "*", "11", "+", "-"]
```

Output:

```
22
```

Explanation:

```
(((10 * (6 / 3)) - 9) + 11) = 22
```

Constraints:

1. `1 <= tokens.length <= 10`
2. `tokens[i]` is either an operator (`"+", "-", "*", "/"`) or an integer in the range `[-200, 200]`.

```python
def evalRPN(tokens):
    # Stack to keep track of operands
    stack = []

    # Define a helper function to perform operations
    def operate(op1, op2, operator):
        if operator == "+":
            return op1 + op2
        elif operator == "-":
            return op1 - op2
        elif operator == "*":
            return op1 * op2
        elif operator == "/":
            # Truncate division towards zero
            return int(op1 / op2)

    # Iterate over the tokens
    for token in tokens:
        if token in "+-*/":
            # If the token is an operator,
            # pop the top two elements from the stack
            op2 = stack.pop()  # Second operand
            op1 = stack.pop()  # First operand
            # Perform the operation and push the result back onto the stack
```

```
            result = operate(op1, op2, token)
            stack.append(result)
        else:
            # If the token is a number,
            # convert it to an integer and push onto the stack
            stack.append(int(token))

    # The final result will be the last element in the stack
    return stack.pop()

# Example usage:
tokens = ["2", "1", "+", "3", "*"]
print(evalRPN(tokens))
```

```
9
```

Step-by-Step Walkthrough for Input `tokens = ["2", "1", "+", "3", "*"]`:

1. Initialize an empty stack: `stack = []`.
2. Process each token:
   - `"2"`: Push `2` onto the stack → `stack = [2]`.
   - `"1"`: Push `1` onto the stack → `stack = [2, 1]`.
   - `"+"`: Pop `1` and `2`, perform `2 + 1 = 3`, push `3` → `stack = [3]`.
   - `"3"`: Push `3` onto the stack → `stack = [3, 3]`.
   - `"*"`: Pop `3` and `3`, perform `3 * 3 = 9`, push `9` → `stack = [9]`.
3. Return the top of the stack: `9`.

Sample Output for Test Cases

Test Case 1:

Input:

```
tokens = ["2", "1", "+", "3", "*"]
```

Output:

```
9
```

Test Case 2:

Input:

```
tokens = ["4", "13", "5", "/", "+"]
```

Output:

```
6
```

Test Case 3:

Input:

```
tokens = ["10", "6", "9", "3", "/", "-", "*", "11", "+", "-"]
```

Output:

```
22
```

5. Decode String (Medium)

Problem:
Given an encoded string, return its decoded string.
The encoding rule is: `k[encoded_string]`, where the `encoded_string` inside the square brackets is being repeated exactly `k` times. You may assume that the input string is always valid; no extra white spaces, square brackets are well-formed, etc.

Constraints: - `1 <= s.length <= 30` - `s` consists of lowercase English letters, digits, and square brackets `'['` and `']'`. - `s` is guaranteed to be a valid encoding that follows the described rules.

Example

Input:
`s = "3[a]2[bc]"`
Output:
`"aaabcbc"`

Explanation:
Decode the string `"3[a]2[bc]"` as follows: - `"3[a]"` means `"a"` repeated 3 times, so it becomes

"aaa". - "2[bc]" means "bc" repeated 2 times, so it becomes "bcbc". - Combine these to get "aaabcbc".

Input:
`s = "3[a2[c]]"`
Output:
"accaccacc"

Explanation:
- "a2[c]" means "a" followed by "c" repeated 2 times, so it becomes "acc". - "3[acc]" means "acc" repeated 3 times, so it becomes "accaccacc".

```
def decodeString(s):
    # Initialize two stacks: one for numbers and another for strings
    num_stack = []  # To store repeat counts
    str_stack = []  # To store temporary strings

    # A temporary variable to keep track of the current string being built
    current_str = ""
    current_num = 0

    for char in s:
        if char.isdigit():
            # Build the current number (handles multiple digits)
            current_num = current_num * 10 + int(char)
        elif char == '[':
            # Push the number and current string to their respective stacks
            num_stack.append(current_num)
            str_stack.append(current_str)

            # Reset temporary variables
            current_str = ""
            current_num = 0
        elif char == ']':
            # Pop the repeat count and previous string
            repeat_count = num_stack.pop()
            previous_str = str_stack.pop()

            # Decode the current string and append to the previous string
            current_str = previous_str + current_str * repeat_count
        else:
            # Build the current string with characters
            current_str += char
```

```
    # The final decoded string is in current_str
    return current_str

# Sample test case
s = "3[a2[c]]"
print(decodeString(s))  # Output: "accaccacc"
```

```
accaccacc
```

Solution: Decoding using Stacks

Here's a step-by-step solution with detailed comments:

Explanation of Code

1. **Stacks for Numbers and Strings:**
   - Use `num_stack` to store numbers (e.g., `3`, `2` in the input).
   - Use `str_stack` to store intermediate strings before encountering a closing bracket `]`.
2. **Processing Characters:**
   - If the character is a digit, build the number (e.g., `3` or `12`).
   - If the character is `[`, push the `current_num` and `current_str` onto their respective stacks and reset them.
   - If the character is `]`, pop from both stacks, repeat the `current_str` by the popped number, and concatenate it with the popped string.
   - Otherwise, append the character to `current_str`.
3. **Final Output:**
   - Once the loop ends, `current_str` contains the fully decoded string.

Sample Output:
For `s = "3[a2[c]]"`, the function prints:
`"accaccacc"`

6. Asteroid Collision (Medium)

We are given an array `asteroids` of integers representing asteroids in a row.

- For each asteroid:
  - The absolute value represents its size.
  - The sign represents its direction:
    - Positive: moving to the right.
    - Negative: moving to the left.

Asteroids move at the same speed. If two asteroids collide, the smaller one explodes. If both are the same size, both explode. Two asteroids moving in the same direction never meet.

Return an array of the asteroids after all collisions.

Sample Input

Input:

```
asteroids = [5, 10, -5]
```

Output:

```
[5, 10]
```

Explanation:

- The `5` and `10` move to the right; no collision.
- The `10` and `-5` collide. The `10` survives as it is larger.

Input:

```
asteroids = [8, -8]
```

Output:

```
[]
```

Explanation:

- The `8` and `-8` collide. Both explode since they are the same size.

Input:

```
asteroids = [10, 2, -5]
```

Output:

```
[10]
```

Explanation:

- The `2` and `-5` collide. The `-5` survives and moves left. It doesn't affect `10`.

How It Works

1. **Initialization**: We use a stack to keep track of active asteroids moving right.
   - Positive asteroids are pushed onto the stack.
   - Negative asteroids may trigger collisions.
2. **Collision Logic**:
   - A collision occurs only if:
     - The stack is non-empty.
     - The top of the stack is positive, and the current asteroid is negative.
   - Depending on their sizes:
     - If the top asteroid is smaller, it explodes (popped from the stack).
     - If the sizes are equal, both explode.
     - If the top asteroid is larger, the current asteroid explodes.
3. **Push Non-Colliding Asteroids**:
   - If no collision occurs, push the current asteroid onto the stack.
4. **Result**:
   - The stack contains the final state of the asteroids.

Sample Output

For the test case `asteroids = [5, 10, -5]`, the printed output will be:

```
Final state of asteroids: [5, 10]
```

```
from typing import List

def asteroidCollision(asteroids: List[int]) -> List[int]:
    """
    This function simulates asteroid collisions and returns the
    state of asteroids after all collisions.
    """
    stack = []  # A stack to keep track of asteroids

    for asteroid in asteroids:
        # Process current asteroid
        while stack and asteroid < 0 < stack[-1]:
            # Collision: Compare top of stack (moving right) with
            # current asteroid (moving left)
            if stack[-1] < -asteroid:
                # Top of stack explodes (smaller size)
                stack.pop()
                continue
            elif stack[-1] == -asteroid:
                # Both explode if equal size
                stack.pop()
            break  # Current asteroid explodes, exit the loop
        else:
            # No collision, push the asteroid onto the stack
            stack.append(asteroid)

    return stack

# Sample Test Case
asteroids = [5, 10, -5]
result = asteroidCollision(asteroids)
print("Final state of asteroids:", result)
```

```
Final state of asteroids: [5, 10]
```

7. Maximum Frequency Stack (Hard)

Design a stack-like data structure to perform the following operations efficiently.

Implement the `FreqStack` class:

- `FreqStack()`: Initializes the object.
- `push(int val)`: Pushes an integer `val` onto the stack.
- `pop()`: Removes and returns the most frequent element in the stack.
  - If there is a tie for the most frequent element, the element closest to the top of the stack is removed and returned.

Example:

Input:

```
["FreqStack", "push", "push", "push", "push", "push", "push", "pop", "pop", "pop", "pop"]
[[], [5], [7], [5], [7], [4], [5], [], [], [], []]
```

Output:

```
[null, null, null, null, null, null, null, 5, 7, 5, 4]
```

Explanation:

```
FreqStack freqStack = new FreqStack();
freqStack.push(5); // The stack is [5]
freqStack.push(7); // The stack is [5, 7]
freqStack.push(5); // The stack is [5, 7, 5]
freqStack.push(7); // The stack is [5, 7, 5, 7]
freqStack.push(4); // The stack is [5, 7, 5, 7, 4]
freqStack.push(5); // The stack is [5, 7, 5, 7, 4, 5]
freqStack.pop();   // Returns 5, as 5 is the most frequent. The stack becomes [5, 7, 5, 7, 4]
freqStack.pop();   // Returns 7, as 7 is the most frequent. The stack becomes [5, 7, 5, 4].
freqStack.pop();   // Returns 5. The stack becomes [5, 7, 4].
freqStack.pop();   // Returns 4. The stack becomes [5, 7].
```

```python
from collections import defaultdict

class FreqStack:
    def __init__(self):
        # Dictionary to count the frequency of each element
        self.freq = defaultdict(int)
        # Dictionary to store stacks for each frequency
        self.group = defaultdict(list)
        # Variable to keep track of the maximum frequency
        self.max_freq = 0

    def push(self, val: int) -> None:
        # Increment the frequency of the value
        self.freq[val] += 1
        # Update the maximum frequency if needed
        self.max_freq = max(self.max_freq, self.freq[val])
        # Add the value to the stack corresponding to its frequency
        self.group[self.freq[val]].append(val)

    def pop(self) -> int:
        # Get the most frequent element (top of the stack of max_freq)
        val = self.group[self.max_freq].pop()
        # Decrease its frequency count
        self.freq[val] -= 1
        # If the stack for the current max frequency is empty,
        # decrement max_freq
        if not self.group[self.max_freq]:
            self.max_freq -= 1
        return val

# Sample Test Case
# Initialize the FreqStack
freqStack = FreqStack()

# Perform operations
freqStack.push(5)  # Stack: [5]
freqStack.push(7)  # Stack: [5, 7]
freqStack.push(5)  # Stack: [5, 7, 5]
freqStack.push(7)  # Stack: [5, 7, 5, 7]
freqStack.push(4)  # Stack: [5, 7, 5, 7, 4]
freqStack.push(5)  # Stack: [5, 7, 5, 7, 4, 5]
print(freqStack.pop())# O/p:5 (Most frequent:5, Stack becomes [5, 7, 5, 7, 4])
```

```
print(freqStack.pop())# O/p:7 (Most frequent:7, Stack becomes [5, 7, 5, 4])
print(freqStack.pop())# O/p:5 (Most frequent:5, Stack becomes [5, 7, 4])
print(freqStack.pop())# O/p:4 (Most frequent:4, Stack becomes [5, 7])
```

```
5
7
5
4
```

8. Daily Temperatures (Medium)

Given a list of daily temperatures `temperatures`, return a list `answer` such that `answer[i]` is the number of days you have to wait after the `i-th` day to get a warmer temperature. If there is no future day for which this is possible, keep `answer[i] == 0` instead.

Example:

Input:

```
temperatures = [73, 74, 75, 71, 69, 72, 76, 73]
```

Output:

```
[1, 1, 4, 2, 1, 1, 0, 0]
```

Explanation:

- On day 0, the next warmer temperature is on day 1 (1 day away).
- On day 1, the next warmer temperature is on day 2 (1 day away).
- On day 2, the next warmer temperature is on day 6 (4 days away).
- On day 3, the next warmer temperature is on day 5 (2 days away).
- On day 4, the next warmer temperature is on day 5 (1 day away).
- On day 5, the next warmer temperature is on day 6 (1 day away).
- On day 6, there is no warmer temperature in the future.
- On day 7, there is no warmer temperature in the future.

```python
def dailyTemperatures(temperatures):
    # Initialize the result list with zeros,
    # as the default value when there's no warmer day is 0
    n = len(temperatures)
    answer = [0] * n

    # A stack to keep track of indices of days.
    # It will store indices where we haven't found a warmer day yet.
    stack = []

    # Traverse the list of temperatures
    for current_day, current_temp in enumerate(temperatures):
        # While stack is not empty and the current day's temperature is
        # greater than the temperature of the day at the stack's top
        while stack and temperatures[stack[-1]] < current_temp:
            prev_day = stack.pop()  # Get the index of the previous day
            answer[prev_day] = current_day - prev_day

        # Add the current day's index to the stack
        stack.append(current_day)

    return answer

# Example usage with the given test case
temperatures = [73, 74, 75, 71, 69, 72, 76, 73]
print(dailyTemperatures(temperatures))
```

```
[1, 1, 4, 2, 1, 1, 0, 0]
```

9. Largest Rectangle in Histogram (Hard)

Given an array of integers `heights` where each element represents the height of a bar in a histogram, find the area of the largest rectangle that can be formed within the histogram.

Example

Input:

```
heights = [2, 1, 5, 6, 2, 3]
```

Output:

```
10
```

Explanation: The largest rectangle is formed by the bars with heights `[5, 6]`, which gives an area of `5 * 2 = 10`.

Constraints:

- `1 <= heights.length <= 10^5`
- `0 <= heights[i] <= 10^4`

```python
from typing import List

class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        n = len(heights)
        max_area = 0
        stack = []  # Stack to keep track of indices of bars

        for i in range(n + 1):
            # Treat the end of the array as a new bar of height 0
            h = 0 if i == n else heights[i]

            # If the current bar is lower than the last one in the stack,
            # pop from the stack
            while stack and heights[stack[-1]] > h:
                # Height of the bar to calculate area with
                current_height = heights[stack.pop()]
                # Calculate the width of the rectangle with the popped height
                left_boundary = -1 if not stack else stack[-1]
                # Current index minus the index after popped
                width = i - left_boundary - 1
                # Update max area if needed
                max_area = max(max_area, current_height * width)

            # Push the current index to the stack
            stack.append(i)
```

```
        return max_area

# Example usage:
sol = Solution()
example_input = [2, 1, 5, 6, 2, 3]
largest_area = sol.largestRectangleArea(example_input)
print("Largest rectangle area is:", largest_area)  # Output should be 10
```

```
Largest rectangle area is: 10
```

10. Trapping Rain Water (Hard)

Given `n` non-negative integers representing an elevation map where the width of each bar is `1`, compute how much water it can trap after raining.

Constraints:

- `n == height.length`
- `1 <= n <= 2 * 10^4`
- `0 <= height[i] <= 10^5`

Example 1:

Input:

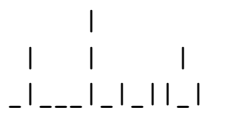
```
height = [0,1,0,2,1,0,1,3,2,1,2,1]
```

Output:

```
6
```

Explanation:

The elevation map is represented as follows:



The trapped water is highlighted between the bars.

Example 2:

Input:

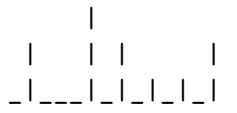
```
height = [4,2,0,3,2,5]
```

Output:

```
9
```

Explanation:

The elevation map looks like this:



A total of 9 units of water are trapped between the bars.

Example 3:

Input:

```
height = [1,0,0,0,1]
```

Output:

3

```
class Solution:
    def trap(self, height: List[int]) -> int:
        # If height array is empty or has less than 3 bars,
        # no water can be trapped
        if not height or len(height) < 3:
            return 0

        # Initialize two pointers for traversing the height list
        left = 0
        right = len(height) - 1

        # Variables to keep track of the maximum height observed
        # so far from the left and the right
        left_max = 0
        right_max = 0

        # Variable to accumulate the total amount of trapped water
        trapped_water = 0

        # Traverse the height array from both ends using two pointers
        while left < right:
            # If the current left bar is lower than the current right bar
            if height[left] < height[right]:
                # If the left bar is higher than the maximum height
                # seen so far from the left, update left_max
                if height[left] >= left_max:
                    left_max = height[left]
                else:
                    # Water can be trapped, equal to the difference between
                    # left_max and current left bar height
                    trapped_water += left_max - height[left]

                # Move the left pointer to the right
                left += 1
            else:
                # If the right bar is higher than the maximum height seen
                # so far from the right, update right_max
                if height[right] >= right_max:
```

```
                    right_max = height[right]
                else:
                    # Water can be trapped, equal to the difference between
                    # right_max and current right bar height
                    trapped_water += right_max - height[right]

                # Move the right pointer to the left
                right -= 1

        # Return the total amount of trapped water
        return trapped_water

# Example test case
test_case = [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]
solution = Solution()
print(f'Trapped water for test case {test_case}: {solution.trap(test_case)}')
```

```
Trapped water for test case [0, 1, 0, 2, 1, 0, 1, 3, 2, 1, 2, 1]: 6
```

11. Backspace String Compare (Easy)

Given two strings `s` and `t`, return `true` if they are equal when both are typed into empty text editors. `#` means a backspace character.

Note that after backspacing an empty text editor, the text will continue to be empty.

Example 1:

Input:

```
s = "ab#c", t = "ad#c"
```

Output:

```
true
```

Explanation: Both strings become "ac".

Example 2:

Input:

`s = "ab##", t = "c#d#"`

Output:

`true`

Explanation: Both strings become “ “.

```python
class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        # This helper function processes the input string
        # and simulates the effect of backspaces by using a stack.
        def build(string) -> str:
            stack = []  # Initialize an empty stack to store characters.
            for char in string:
                if char != '#':
                    # If the character is not a backspace,
                    # push it onto the stack.
                    stack.append(char)
                elif stack:
                    # If the character is a backspace and
                    # the stack is not empty,
                    # pop the top element from the stack.
                    stack.pop()
            # Join the characters in the stack to form the resultant string.
            return "".join(stack)

        # Compare the processed versions of both strings.
        return build(s) == build(t)

# Test case to demonstrate the functionality
if __name__ == "__main__":
    solution = Solution()

    # Example test
    s = "ab#c"
    t = "ad#c"
    print(solution.backspaceCompare(s, t))  # Output: True
```

`True`

12. Longest Valid Parentheses (Hard)

Given a string containing just the characters `'('` and `')'`, find the length of the longest valid (well-formed) parentheses substring.

Example 1:

Input:
`s = "(()"`

Output:
`2`

Explanation:
The longest valid parentheses substring is "()"

Example 2:

Input:
`s = ")()())"`

Output:
`4`

Explanation:
The longest valid parentheses substring is `"()()"`.

Example 3:

Input:
`s = ""`

Output:
`0`

Explanation:
There are no valid parentheses substrings.

Constraints:

- `0 <= s.length <= 3 * 10^4`
- `s[i]` is either `'('` or `')'`.

```
class Solution:
    def longestValidParentheses(self, s: str) -> int:
        """
        This function finds the length of the longest valid
        (well-formed) parentheses substring within a given string `s`.

        Args:
        s (str): The input string consisting of '(' and ')'.

        Returns:
        int: The length of the longest valid parentheses substring.
        """

        # Initialize max_length to keep track of the
        # longest valid parentheses length found.
        max_length = 0

        # Initialize a stack with a base index of -1.
        # This helps with calculating the length of valid parentheses.
        # It's used to handle edge cases such as "()".
        stack = [-1]

        # Iterate over the string with both index and character
        for i, char in enumerate(s):
            if char == "(":
                # If the character is '(', push its index onto the stack.
                stack.append(i)
            else:
                # If it's ')', pop the topmost element from the stack.
                stack.pop()

                if not stack:
                    # If the stack is empty after popping, push the
                    # current index as a base for future calculations.
                    stack.append(i)
                else:
                    # If stack is not empty, calculate the valid
```

```
                    # substring length by subtracting the current index
                    # with the top of the stack.
                    # Update max_length if the calculated length is greater.
                    max_length = max(max_length, i - stack[-1])

        # Return the maximum length of valid parentheses found
        return max_length

# Test case
solution = Solution()
test_string = "(()"
print(solution.longestValidParentheses(test_string))  # Expected output: 2
```

2

13. Basic Calculator (Hard)

Implement a basic calculator to evaluate a simple expression string.

The expression string may contain the following components: - Non-negative integers. - `+`, `-`, `(`, and `)` operators. - Empty spaces.

You should return the result of the evaluated expression.

Note: - You may assume that the given expression is always valid. - Do not use the `eval` built-in library function.

Example 1:

Input:
`"1 + 1"`

Output:
2

Example 2:

Input:
`" 2-1 + 2 "`

Output:
3

Example 3:

Input:
`"(1+(4+5+2)-3)+(6+8)"`

Output:
23

Constraints:

- The length of the input string is in the range `[1, 3 * 10^5]`.
- The input string consists of digits, `+`, `-`, `(`, `)`, and spaces.

```python
class Solution:
    def calculate(self, s: str) -> int:
        # Stack for storing the result and sign temporarily
        # when encountering parentheses
        stack = []

        # Variables to hold the working result, current number, and sign
        result = 0
        number = 0
        sign = 1

        # Iterate over each character in the input string
        for c in s:
            # If the character is a digit, build the current number
            if c.isdigit():
                number = number * 10 + int(c)

            # If the character is a '+', complete the current operation
            # using the current number and reset for the next number
            elif c == '+':
                result += number * sign
                sign = 1  # Reset the sign to positive
                number = 0  # Reset the current number

            # If the character is a '-', complete the current operation
            # using the current number and reset for the next number
            elif c == '-':
                result += number * sign
                sign = -1  # Change the sign to negative for the next number
```

```
                number = 0  # Reset the current number

            # If the character is '(', push the result and the
            # sign onto the stack and reset them for a new sub-expression
            elif c == '(':
                stack.append(result)
                stack.append(sign)
                result = 0
                sign = 1

            # If the character is ')', complete the current sub-expression:
            #1. Apply the last number with the current sign,
            #2. Pop the sign from the stack and apply it to the current result
            #3. Pop the result from the stack and add it to the current result
            elif c == ')':
                result += number * sign
                number = 0
                result *= stack.pop()  # Apply the sign
                result += stack.pop()  # Add to the previous result

        # After the loop, there might be a number left to be added to result
        result += number * sign

        return result

# Test case
solution = Solution()
test_expression = "(1+(4+5+2)-3)+(6+8)"
print("Expression:", test_expression)
print("Calculated Result:", solution.calculate(test_expression))
```

```
Expression: (1+(4+5+2)-3)+(6+8)
Calculated Result: 23
```

14. Basic Calculator II (Hard)

Implement a basic calculator to evaluate a simple expression string.

The expression string contains only **non-negative integers**, `+`, `-`, `*`, `/` operators, and empty spaces .
The integer division should truncate toward zero.

You may assume that the given input is always valid.
All intermediate results will be in the range of `[-2^31, 2^31 - 1]`.

Constraints:

- `1 <= s.length <= 3 * 10^5`
- `s` consists of digits, `+`, `-`, `*`, `/`, and spaces `' '`.
- `s` is a valid expression.

Examples:

Example 1:

Input:
`s = "3+2*2"`

Output:
`7`

Example 2:

Input:
`s = " 3/2 "`

Output:
`1`

Example 3:

Input:
`s = " 3+5 / 2 "`

Output:
`5`

Note:

- Do not use the `eval()` function or any similar built-in methods for evaluation.

```python
class Solution:
    def calculate(self, s: str) -> int:
        # Initialize a stack to keep track of numbers
        stack = []

        # Variable to store the current number being processed
        num = 0

        # Variable to store the last operation encountered, initialized to '+'
        last_oper = '+'

        # Add a '+' at the end of the string to handle the last number
        s += '+'

        # Iterate over each character in the string
        for char in s:
            # If the character is a digit, build the current number
            if char.isdigit():
                num = num * 10 + int(char)

            # If the character is an operator, process the
            # previous number and operator
            elif char in '+-*/':
                # Process the current number based on the last operation
                if last_oper == '+':
                    stack.append(num)
                elif last_oper == '-':
                    stack.append(-num)
                elif last_oper == '*':
                    stack.append(stack.pop() * num)
                elif last_oper == '/':
                    # Use integer division in Python3
                    stack.append(int(stack.pop() / num))

                # Update the last operator to the current operator
                last_oper = char

                # Reset current number for the next parse
                num = 0

        # Return the sum of numbers in the stack
        return sum(stack)
```

```
# Test case
solution = Solution()
result = solution.calculate("3+2*2")
print(result)  # Expected output: 7
```

7