Stack and Queue

1. Valid Parentheses

Pattern: Stack & Monotonic Stack

Problem Statement

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.
3. Every close bracket has a corresponding open bracket of the same type.

Sample Input & Output

```
Input: "()"
Output: true
Explanation: Simple matching pair.
```

```
Input: "()[]{}"
Output: true
Explanation: Multiple independent valid pairs in order.
```

```
Input: "([)]"
Output: false
Explanation: Closing ')' appears before ']' but
'(' was opened before '[' - incorrect nesting.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def isValid(self, s: str) -> bool:
        # STEP 1: Initialize structures
        #   - Use a stack (list) to track expected closing brackets.
        #   - Map each opening bracket to its corresponding closing bracket.
        stack = []
        mapping = {
            '(': ')',
            '[': ']',
            '{': '}'
        }

        # STEP 2: Main loop / recursion
        #   - Iterate through each character in the string.
        #   - If it's an opening bracket, push its closing counterpart.
        #   - If it's a closing bracket, check if it matches the top.
        for char in s:
            if char in mapping:
                # Opening bracket: push expected closer
                stack.append(mapping[char])
            else:
                # Closing bracket: must match last expected closer
                if not stack or stack.pop() != char:
                    return False

        # STEP 3: Update state / bookkeeping
        #   - Already handled in loop via stack push/pop.

        # STEP 4: Return result
```

```
        #   - Valid only if all opened brackets were closed (stack empty).
        return len(stack) == 0

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.isValid("()[]{}") == True

    #  Test 2: Edge case - empty string
    assert sol.isValid("") == True

    #  Test 3: Tricky/negative - mismatched nesting
    assert sol.isValid("([)]") == False
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `s = "([)]"` step by step:

1. **Initialize**:
   - `stack = []`
   - `mapping = {'(': ')', '[': ']', '{': '}'}`
2. **Process `'('`**:
   - `'('` is in `mapping` → push `mapping['(']` → `')'`
   - `stack = [')']`
3. **Process `'['`**:
   - `'['` is in `mapping` → push `mapping['[']` → `']'`
   - `stack = [')', ']']`
4. **Process `')'`**:

- `')'` is **not** in `mapping` → it's a closer.
- Check: `stack` is not empty → pop last element → `']'`
- Compare: `']' != ')'` → **return `False` immediately**

Final output: `False`

Key Insight: The stack enforces **LIFO (Last-In-First-Out)** order, which matches the nested structure of valid parentheses. If a closer doesn't match the most recent opener's expected closer, the string is invalid.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the string once (`n` characters). Each stack operation (`append`, `pop`) is `O(1)`.

- **Space Complexity**: `O(n)`

  In the worst case (e.g., all opening brackets like `"((("`), the stack stores `n` elements. The `mapping` dict uses constant space.

2. Min Stack

Pattern: Stack & Monotonic Stack

Problem Statement

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 each function in $\mathbf{O(1)}$ time complexity.

Sample Input & Output

```
Input: ["MinStack","push","push","push","getMin","pop","top","getMin"]
       [[],[-2],[0],[-3],[],[],[],[]]
Output: [null,null,null,null,-3,null,0,-2]
Explanation:
  - push(-2): stack = [-2], min = -2
  - push(0):  stack = [-2,0], min = -2
  - push(-3): stack = [-2,0,-3], min = -3
  - getMin() → -3
  - pop() → removes -3
  - top() → 0
  - getMin() → -2
```

```
Input: ["MinStack","push","push","getMin"]
       [[],[5],[3],[]]
Output: [null,null,null,3]
```

```
Input: ["MinStack","push","getMin","pop","push","getMin"]
       [[],[-5],[],[],[10],[]]
Output: [null,null,-5,null,null,-5]
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class MinStack:
    def __init__(self):
        # STEP 1: Initialize two stacks
        #   - main_stack: holds all pushed values
        #   - min_stack: tracks current minimum at each state
```

```
        self.main_stack = []
        self.min_stack = []

    def push(self, val: int) -> None:
        # STEP 2: Push to main stack unconditionally
        self.main_stack.append(val)

        # STEP 3: Update min_stack with current minimum
        #   - If min_stack is empty, val is the new min
        #   - Else, compare val with current min (top of min_stack)
        current_min = val
        if self.min_stack:
            current_min = min(val, self.min_stack[-1])
        self.min_stack.append(current_min)

    def pop(self) -> None:
        # STEP 4: Pop from both stacks to maintain alignment
        #   - Each main_stack state has a corresponding min in min_stack
        self.main_stack.pop()
        self.min_stack.pop()

    def top(self) -> int:
        # STEP 5: Return top of main stack
        return self.main_stack[-1]

    def getMin(self) -> int:
        # STEP 6: Return top of min_stack (current min)
        return self.min_stack[-1]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = MinStack()

    #  Test 1: Normal case
    sol.push(-2)
    sol.push(0)
    sol.push(-3)
    assert sol.getMin() == -3
    sol.pop()
    assert sol.top() == 0
    assert sol.getMin() == -2
```

```python
    #  Test 2: Edge case - single element
    sol2 = MinStack()
    sol2.push(5)
    assert sol2.getMin() == 5
    sol2.pop()

    #  Test 3: Tricky/negative - repeated mins
    sol3 = MinStack()
    sol3.push(2)
    sol3.push(2)
    sol3.push(1)
    assert sol3.getMin() == 1
    sol3.pop()
    assert sol3.getMin() == 2  # min reverts correctly
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1** step by step:

1. `sol = MinStack()`
   - `main_stack = []`
   - `min_stack = []`
2. `sol.push(-2)`
   - `main_stack` becomes `[-2]`
   - `min_stack` is empty → push `-2`
   - `min_stack = [-2]`
3. `sol.push(0)`
   - `main_stack = [-2, 0]`
   - Current min = `min(0, -2) = -2` → push `-2`

- `min_stack = [-2, -2]`

4. **`sol.push(-3)`**
   - `main_stack = [-2, 0, -3]`
   - Current min = `min(-3, -2) = -3` → push `-3`
   - `min_stack = [-2, -2, -3]`
5. **`sol.getMin()`**
   - Return `min_stack[-1] = -3`
6. **`sol.pop()`**
   - Remove last from both:
     - `main_stack = [-2, 0]`
     - `min_stack = [-2, -2]`
7. **`sol.top()`**
   - Return `main_stack[-1] = 0`
8. **`sol.getMin()`**
   - Return `min_stack[-1] = -2`

Final state:
- `main_stack = [-2, 0]`
- `min_stack = [-2, -2]`

All operations in **O(1)** time!

Complexity Analysis

- **Time Complexity**: `O(1)`

  Every operation (`push`, `pop`, `top`, `getMin`) performs only a constant number of stack operations (append/pop/peek), each O(1).

- **Space Complexity**: `O(n)`

  We maintain two stacks of size `n` (number of elements pushed). In worst case, `min_stack` stores `n` values — e.g., when values are pushed in decreasing order.

3. Evaluate Reverse Polish Notation

Pattern: Stack & Monotonic Stack

Problem Statement

You are given an array of strings `tokens` that represents an arithmetic expression in **Reverse Polish Notation (RPN)**.

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

Valid operators are `+`, `-`, `*`, and `/`. Each operand may be an integer or another expression.

Note:
- Division between two integers always **truncates toward zero**.
- The given RPN expression is always valid. That means the expression would always evaluate to a result, and there will not be any division by zero.

Sample Input & Output

```
Input: ["2","1","+","3","*"]
Output: 9
Explanation: ((2 + 1) * 3) = 9
```

```
Input: ["4","13","5","/","+"]
Output: 6
Explanation: (4 + (13 / 5)) = 4 + 2 = 6 (truncated toward zero)
```

```
Input: ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
Output: 22
Explanation: Complicated nesting, but valid RPN.
```

LeetCode Editorial Solution + Inline Tests

```
from typing import List

class Solution:
    def evalRPN(self, tokens: List[str]) -> int:
        # STEP 1: Initialize stack
        #   - Stack holds operands (integers) as we process tokens.
        #   - Operators trigger popping last two operands.
        stack = []

        # STEP 2: Main loop over tokens
        #   - For each token:
        #       • If number → push to stack
        #       • If operator → pop two, apply op, push result
        for token in tokens:
            if token in "+-*/":
                # Pop right operand first, then left
                b = stack.pop()
                a = stack.pop()

                # STEP 3: Apply operation with truncation toward 0
                if token == "+":
                    res = a + b
                elif token == "-":
                    res = a - b
                elif token == "*":
                    res = a * b
                else:  # token == "/"
                    # Use int() to truncate toward zero (not //)
                    res = int(a / b)

                stack.append(res)
            else:
                # Convert string number to int and push
                stack.append(int(token))

        # STEP 4: Return result
        #   - Final result is the only element left in stack
        return stack[0]

# ------------------- INLINE TESTS -------------------
```

```python
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.evalRPN(["2","1","+","3","*"]) == 9

    #  Test 2: Edge case - division truncation
    assert sol.evalRPN(["4","13","5","/","+"]) == 6

    #  Test 3: Tricky/negative - complex nesting
    assert sol.evalRPN(
        ["10","6","9","3","+","-11","*","/","*","17","+","5","+"]
    ) == 22

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's walk through **Test 2**: `["4","13","5","/","+"]`

1. **Initialize**: `stack = []`
2. **Token "4"**: not an operator → push `4` → `stack = [4]`
3. **Token "13"**: push `13` → `stack = [4, 13]`
4. **Token "5"**: push `5` → `stack = [4, 13, 5]`
5. **Token "/"**: operator!
   - Pop `b = 5`, then `a = 13`
   - Compute `int(13 / 5) = int(2.6) = 2`
   - Push `2` → `stack = [4, 2]`
6. **Token "+"**: operator!

- Pop `b = 2`, then `a = 4`
- Compute `4 + 2 = 6`
- Push `6` → `stack = [6]`

7. **End**: return `stack[0] = 6`

Final output: `6`

Key Insight:
- RPN eliminates parentheses by ordering operands before operators.
- Stack naturally reverses operand order → pop **right** operand first!
- Division must **truncate toward zero**, so `int(a / b)` is safer than `a // b` (which floors for negatives).

Complexity Analysis

- **Time Complexity**: `O(n)`

  We process each token exactly once. Each operation (push/pop, arithmetic) is O(1).

- **Space Complexity**: `O(n)`

  In worst case (e.g., all numbers first), stack holds ~n/2 integers. Scales linearly with input size.

4. Basic Calculator

Pattern: Stack & Monotonic Stack

Problem Statement

> Given a string `s` representing a valid expression containing only non-negative integers, `+`, `-`, `*`, `/` operators, and empty spaces, evaluate the expression and return the result. The integer division should truncate toward zero.

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

Note: You are **not** allowed to use the built-in `eval()` function.

Sample Input & Output

```
Input: "3+2*2"
Output: 7
Explanation: Multiplication has higher precedence; 2*2 = 4, then 3+4 = 7.
```

```
Input: " 3/2 "
Output: 1
Explanation: Integer division truncates toward zero: 3/2 = 1.5 → 1.
```

```
Input: "14-3/2"
Output: 13
Explanation: 3/2 = 1, then 14 - 1 = 13.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def calculate(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - stack: holds numbers to sum at the end
        #   - num: accumulates current number from digits
```

```
        #   - op: tracks last seen operator (starts with '+')
        stack = []
        num = 0
        op = '+'  # Pretend expression starts with '+'

        # STEP 2: Main loop / recursion
        #   - Process each char; handle digits, spaces, operators
        for i, char in enumerate(s):
            if char.isdigit():
                num = num * 10 + int(char)

            # Trigger on operator or end of string
            if char in "+-*/" or i == len(s) - 1:
                # STEP 3: Update state / bookkeeping
                #   - Apply last operator to current num
                if op == '+':
                    stack.append(num)
                elif op == '-':
                    stack.append(-num)
                elif op == '*':
                    stack.append(stack.pop() * num)
                elif op == '/':
                    # Truncate toward zero (not floor division!)
                    prev = stack.pop()
                    if prev < 0:
                        stack.append(-(-prev // num))
                    else:
                        stack.append(prev // num)

                # Reset for next number
                num = 0
                op = char

        # STEP 4: Return result
        #   - Sum all values in stack (handles precedence via early eval)
        return sum(stack)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
```

```
    result1 = sol.calculate("3+2*2")
    print(f"Test 1: {result1}")  # Expected: 7

    #  Test 2: Edge case (division truncation)
    result2 = sol.calculate(" 3/2 ")
    print(f"Test 2: {result2}")  # Expected: 1

    #  Test 3: Tricky/negative (precedence + truncation)
    result3 = sol.calculate("14-3/2")
    print(f"Test 3: {result3}")  # Expected: 13
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `s = "14-3/2"` step by step.

Initial state:
- `stack = []`
- `num = 0`
- `op = '+'`

Step 1: `i=0`, `char='1'` → digit
- `num = 0*10 + 1 = 1`

Step 2: `i=1`, `char='4'` → digit
- `num = 1*10 + 4 = 14`

Step 3: `i=2`, `char='-'` → operator → **trigger evaluation**
- Last `op` is `'+'`, so push `num=14` → `stack = [14]`
- Reset: `num = 0`, `op = '-'`

Step 4: `i=3`, `char='3'` → digit
- `num = 0*10 + 3 = 3`

Step 5: `i=4`, `char='/'` → operator → **trigger evaluation**
- Last `op` is `'-'`, so push `-num = -3` → `stack = [14, -3]`
- Reset: `num = 0`, `op = '/'`

Step 6: `i=5`, `char='2'` → digit
- `num = 0*10 + 2 = 2`

Step 7: `i=5` is last index → **trigger final evaluation**
- Last `op` is `'/'`, pop `-3` → `prev = -3`
- Since `prev < 0`: compute `-(-(-3) // 2) = -(3 // 2) = -1`
- Push `-1` → `stack = [14, -1]`

Final step: `sum(stack) = 14 + (-1) = 13`

Output: `13`

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the string once (`n = len(s)`). Each character is processed in constant time. Stack operations (`append`, `pop`) are `O(1)`.

- **Space Complexity**: `O(n)`

  In the worst case (e.g., all additions: `"1+2+3+4"`), the stack stores every number → `O(n)` space. Note: actual numbers stored number of terms, which is `n`.

5. Basic Calculator II

Pattern: Stack & Monotonic Stack

Problem Statement

Given a string `s` which represents an expression, evaluate this expression and return its value.

The integer division should truncate toward zero.

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

Note: You are **not** allowed to use the built-in `eval()` function.

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

Sample Input & Output

```
Input: "3+2*2"
Output: 7
Explanation: Multiplication has higher precedence; 2*2=4, then 3+4=7.
```

```
Input: " 3/2 "
Output: 1
Explanation: Integer division truncates toward zero: 3/2 = 1.5 → 1.
```

```
Input: "14-3/2"
Output: 13
Explanation: 3/2 = 1, then 14 - 1 = 13.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def calculate(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - stack: holds numbers to sum at the end
        #   - num: current number being parsed
        #   - op: last seen operator (starts with '+')
        stack = []
        num = 0
        op = '+'

        # STEP 2: Main loop / recursion
        #   - iterate through each char; handle digits, spaces, ops
```

```
        for i, char in enumerate(s):
            if char.isdigit():
                num = num * 10 + int(char)

            # Process when we hit an operator or end of string
            if char in "+-*/" or i == len(s) - 1:
                # STEP 3: Update state / bookkeeping
                #   - apply last operator to current num
                if op == '+':
                    stack.append(num)
                elif op == '-':
                    stack.append(-num)
                elif op == '*':
                    stack.append(stack.pop() * num)
                elif op == '/':
                    # Truncate toward zero (not floor division!)
                    prev = stack.pop()
                    if prev < 0:
                        stack.append(-(-prev // num))
                    else:
                        stack.append(prev // num)

                # Reset for next number
                num = 0
                op = char

        # STEP 4: Return result
        #   - sum all values in stack (handles precedence via early
        #     resolution of * and /)
        return sum(stack)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.calculate("3+2*2") == 7, "Test 1 failed"
    print("  Test 1 passed: '3+2*2' → 7")

    #  Test 2: Edge case (division truncation)
    assert sol.calculate(" 3/2 ") == 1, "Test 2 failed"
    print("  Test 2 passed: ' 3/2 ' → 1")
```

```
#   Test 3: Tricky/negative (precedence + truncation)
assert sol.calculate("14-3/2") == 13, "Test 3 failed"
print("  Test 3 passed: '14-3/2' → 13")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `calculate("14-3/2")` step by step.

Initial state:
- `stack = []`
- `num = 0`
- `op = '+'`

Now iterate over each character:

1. **i=0, char='1'** → digit
   - `num = 0*10 + 1 = 1`
2. **i=1, char='4'** → digit
   - `num = 1*10 + 4 = 14`
3. **i=2, char='-'** → operator
   - Current `op` is `'+'`, so push `num=14` → `stack = [14]`
   - Reset: `num = 0`, `op = '-'`
4. **i=3, char='3'** → digit
   - `num = 0*10 + 3 = 3`
5. **i=4, char='/'** → operator
   - Current `op` is `'-'`, so push `-num = -3` → `stack = [14, -3]`
   - Reset: `num = 0`, `op = '/'`
6. **i=5, char='2'** → digit
   - `num = 0*10 + 2 = 2`

7. **i=5 is last index** → process final number
    - Current `op` is `'/'`
    - Pop last: `prev = -3`
    - Since `prev < 0`: compute `-(-(-3) // 2) = -(3 // 2) = -1`
    - Push `-1` → `stack = [14, -1]`
8. **Return sum(stack)** → `14 + (-1) = 13`

Final output: **13**

Key insight:
- We **defer addition/subtraction** by storing signed numbers.
- We **immediately resolve multiplication/division** using the last number in the stack.
- This avoids needing a full expression parser or operator precedence table.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We scan the string once (`n = len(s)`). Each character is processed in constant time. Stack operations are `O(1)` per number.

- **Space Complexity**: `O(n)`

  In the worst case (e.g., "1+2+3+4+..."), the stack stores every number. So space scales linearly with input size.

6. Decode String

Pattern: Stack & Monotonic Stack

Problem Statement

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. Note that `k` is guaranteed to be a positive integer.
You may assume that the input string is always valid; there are no extra white spaces, square brackets are well-formed, etc.
Furthermore, you may assume that the original data does not contain any digits and that digits are only for those repeat numbers, `k`.

Sample Input & Output

```
Input: "3[a]2[bc]"
Output: "aaabcbc"
Explanation: "a" repeated 3 times → "aaa", "bc" repeated 2 times → "bcbc",
concatenated → "aaabcbc"
```

```
Input: "3[a2[c]]"
Output: "accaccacc"
Explanation: Inner: "2[c]" → "cc"; then "a" + "cc" = "acc";
repeated 3 times → "accaccacc"
```

```
Input: "2[abc]3[cd]ef"
Output: "abcabccdcdcdef"
Explanation: Handles multiple adjacent encoded parts and trailing plain text.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def decodeString(self, s: str) -> str:
        # STEP 1: Initialize structures
```

```
        #   - stack: holds partial results and repeat counts
        #   - current_string: accumulates letters in current scope
        #   - current_num: builds the current repeat number digit by digit
        stack = []
        current_string = ""
        current_num = 0

        # STEP 2: Main loop / recursion
        #   - Process each char; digits build number, '[' pushes state,
        #     ']' pops and repeats, letters append to current_string
        for char in s:
            if char.isdigit():
                # Build multi-digit number (e.g., '12' from '1' then '2')
                current_num = current_num * 10 + int(char)
            elif char == '[':
                # Push current state before entering nested scope
                stack.append((current_string, current_num))
                current_string = ""
                current_num = 0
            elif char == ']':
                # Pop previous scope and repeat current_string
                prev_string, repeat = stack.pop()
                current_string = prev_string + current_string * repeat
            else:
                # Accumulate letters
                current_string += char

        # STEP 4: Return result
        #   - current_string holds fully decoded result after processing
        return current_string

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.decodeString("3[a]2[bc]") == "aaabcbc"

    #  Test 2: Edge case - nested brackets
    assert sol.decodeString("3[a2[c]]") == "accaccacc"

    #  Test 3: Tricky/negative - mixed with plain suffix
```

```
    assert sol.decodeString("2[abc]3[cd]ef") == "abcabccdcdcdef"

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `decodeString("3[a2[c]]")` step by step:

1. **Initial state**:
   `stack = []`, `current_string = ""`, `current_num = 0`
2. **Read '3'** → digit:
   `current_num = 0*10 + 3 = 3`
3. **Read '['** → push state:
   Push `("", 3)` to stack.
   Reset: `current_string = ""`, `current_num = 0`
4. **Read 'a'** → letter:
   `current_string = "a"`
5. **Read '2'** → digit:
   `current_num = 0*10 + 2 = 2`
6. **Read '['** → push state:
   Push `("a", 2)` to stack.
   Reset: `current_string = ""`, `current_num = 0`
7. **Read 'c'** → letter:
   `current_string = "c"`
8. **Read ']'** → pop and repeat:
   Pop `("a", 2)` → `current_string = "a" + "c" * 2 = "acc"`
9. **Read ']'** → pop and repeat:
   Pop `("", 3)` → `current_string = "" + "acc" * 3 = "accaccacc"`
10. **Return**: `"accaccacc"`

Key insight: The stack saves the outer context so inner brackets can be resolved first (LIFO), then merged outward.

Complexity Analysis

- **Time Complexity**: `O(n * m)`

  `n` = length of input string. In worst case (e.g., `k[...]` deeply nested), each character may be copied up to `m` times, where `m` is the max repetition depth product. However, for typical inputs, it's effectively linear in output size.

- **Space Complexity**: `O(n + d)`

  `n` for the stack (stores strings and numbers), `d` = max nesting depth (recursion simulated via stack). In worst case, stack holds partial strings proportional to input size.

7. Asteroid Collision

Pattern: Stack & Monotonic Stack

Problem Statement

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

For each asteroid, the absolute value represents its size, and the sign represents its direction (`positive` meaning right, `negative` meaning left).

All asteroids move at the same speed.

Find out the state of the asteroids after all collisions. If two asteroids meet, the smaller one will explode. If both are the same size, both will explode. Two asteroids moving in the same direction will never meet.

Return the resulting array of asteroids.

Sample Input & Output

```
Input: [5, 10, -5]
Output: [5, 10]
Explanation: The 10 and -5 collide, -5 explodes. 5 and 10 never meet.
```

```
Input: [8, -8]
Output: []
Explanation: Both asteroids collide and annihilate each other.
```

```
Input: [-2, -1, 1, 2]
Output: [-2, -1, 1, 2]
Explanation: Left-moving (-2, -1) and right-moving (1, 2) never collide.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def asteroidCollision(self, asteroids: List[int]) -> List[int]:
        # STEP 1: Initialize stack to simulate surviving asteroids
        #   - Stack holds asteroids moving right (positive) or left
        #     that haven't been destroyed yet.
        stack = []

        for asteroid in asteroids:
            # STEP 2: Handle collisions only when:
            #   - Current asteroid is moving left (negative)
            #   - Top of stack is moving right (positive)
            while (stack and
                   stack[-1] > 0 and
                   asteroid < 0):
                # STEP 3: Compare sizes to decide outcome
                if stack[-1] < -asteroid:
                    # Top asteroid explodes; continue checking
                    stack.pop()
                    continue
                elif stack[-1] == -asteroid:
                    # Both explode
```

```
                    stack.pop()
                    break
                else:
                    # Current asteroid explodes; do nothing
                    break
            else:
                # STEP 4: Add asteroid if it survives all collisions
                #   - 'else' triggers if while loop didn't break
                stack.append(asteroid)

        # STEP 5: Return final state
        return stack

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    result1 = sol.asteroidCollision([5, 10, -5])
    print(f"Test 1: {result1}")  # Expected: [5, 10]

    #  Test 2: Edge case - mutual destruction
    result2 = sol.asteroidCollision([8, -8])
    print(f"Test 2: {result2}")  # Expected: []

    #  Test 3: Tricky/negative - no collisions
    result3 = sol.asteroidCollision([-2, -1, 1, 2])
    print(f"Test 3: {result3}")  # Expected: [-2, -1, 1, 2]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `asteroidCollision([5, 10, -5])` step by step.

Initial state: `stack = []`

Step 1: Process `asteroid = 5`
- `stack` is empty → skip `while` loop
- Enter `else` → `stack.append(5)`
- **State**: `stack = [5]`

Step 2: Process `asteroid = 10`
- `stack[-1] = 5 > 0`, but `asteroid = 10 > 0` → no collision
- Skip `while` loop → `else` → `stack.append(10)`
- **State**: `stack = [5, 10]`

Step 3: Process `asteroid = -5`
- Enter `while` loop because: - `stack` not empty - `stack[-1] = 10 > 0` - `asteroid = -5 < 0`
- Compare: `10` vs `|-5| = 5` → `10 > 5` - Current asteroid (`-5`) explodes → `break` out of loop
- **Do not append** `-5` - **State**: `stack = [5, 10]`

Final return: `[5, 10]`

> **Key insight**: Only right-moving (`+`) followed by left-moving (`-`) causes collisions. Stack naturally preserves order and enables backtracking to resolve chain reactions.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each asteroid is pushed and popped at most once. The `while` loop may seem nested, but total operations are bounded by `2n`.

- **Space Complexity**: `O(n)`

  In worst case (e.g., all asteroids moving right), stack stores all `n` elements. Output list also counts toward space.

8. Largest Rectangle in Histogram

Pattern: Stack & Monotonic Stack

Problem Statement

Given an array of integers `heights` representing the heights of bars in a histogram, return the area of the largest rectangle that can be formed within the histogram.

Each bar has width `1`. The rectangle must be aligned with the histogram bars (i.e., you cannot rotate or skew it).

Sample Input & Output

```
Input: [2,1,5,6,2,3]
Output: 10
Explanation: The largest rectangle spans bars with
heights 5 and 6 (width=2, height=5 → area=10).
```

```
Input: [2,4]
Output: 4
Explanation: Either bar alone gives area 2 or 4; max is 4.
```

```
Input: [1]
Output: 1
Explanation: Only one bar → area = 1×1 = 1.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def largestRectangleArea(self, heights: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - Use a stack to store indices of bars in increasing height.
        #   - Append 0 to handle remaining bars in stack at the end.
        stack = []
        max_area = 0
        # Add sentinel to trigger final pops
        heights.append(0)

        # STEP 2: Main loop / recursion
        #   - For each bar, if current height < stack top,
        #     we calculate area with stack top as smallest bar.
        for i in range(len(heights)):
            # Maintain monotonic increasing stack
            while stack and heights[i] < heights[stack[-1]]:
                h = heights[stack.pop()]
                # Width = i if stack empty (covers from start),
                # else i - stack[-1] - 1
                w = i if not stack else i - stack[-1] - 1
                max_area = max(max_area, h * w)
            stack.append(i)

        # STEP 3: Update state / bookkeeping
        #   - The sentinel (0) ensures all bars are popped and evaluated.

        # STEP 4: Return result
        #   - max_area holds the largest rectangle found.
        return max_area

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.largestRectangleArea([2,1,5,6,2,3]) == 10

    #  Test 2: Edge case - single bar
    assert sol.largestRectangleArea([1]) == 1
```

```
    #   Test 3: Tricky/negative - descending then ascending
    assert sol.largestRectangleArea([5,4,3,2,1]) == 9  # 3*3 from middle
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `heights = [2,1,5,6,2,3]` → becomes `[2,1,5,6,2,3,0]` after sentinel.

Initial state:
- `stack = []`
- `max_area = 0`

---

i = 0 (`height = 2`)
- Stack empty → push `0`
- `stack = [0]`

i = 1 (`height = 1`)
- `1 < heights[0]=2` → pop 0:
- `h = 2`
- Stack empty → `w = 1`
- `area = 2*1 = 2` → `max_area = 2`
- Push `1`
- `stack = [1]`

i = 2 (`height = 5`)
- `5 >= heights[1]=1` → push 2
- `stack = [1,2]`

i = 3 (`height = 6`)
- `6 >= 5` → push 3
- `stack = [1,2,3]`

i = 4 (`height = 2`)
- `2 < heights[3]=6` → pop 3:
- `h = 6`, `w = 4 - 2 - 1 = 1` → area = 6 → `max_area = 6`
- Now `2 < heights[2]=5` → pop 2:
- `h = 5`, `w = 4 - 1 - 1 = 2` → area = 10 → `max_area = 10`

- Now `2 >= heights[1]=1` → push `4`
- `stack = [1,4]`

i = 5 (`height = 3`)
- `3 >= 2` → push `5`
- `stack = [1,4,5]`

i = 6 (`height = 0`, sentinel)
- Pop `5`: `h=3`, `w=6-4-1=1` → area=3
- Pop `4`: `h=2`, `w=6-1-1=4` → area=8
- Pop `1`: `h=1`, stack empty → `w=6` → area=6
- `max_area` remains **10**

Final return: **10**

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each bar is pushed and popped **at most once**, so total operations are linear in `n`.

- **Space Complexity**: `O(n)`

  The stack can hold up to `n` indices in the worst case (e.g., strictly increasing heights). The input is modified in-place with one extra element (sentinel), which is acceptable per LeetCode conventions.

9. Trapping Rain Water

Pattern: Stack & Monotonic Stack

Problem Statement

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.

Sample Input & Output

```
Input: height = [0,1,0,2,1,0,1,3,2,1,2,1]
Output: 6
Explanation: The above elevation map traps 6 units of rain water.
```

```
Input: height = [3,0,2]
Output: 2
Explanation: Water is trapped between the 3 and 2.
```

```
Input: height = [2,0,2]
Output: 2
Explanation: Symmetric case - water fills the dip fully.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def trap(self, height: List[int]) -> int:
        # STEP 1: Initialize structures
        #   - Use a stack to store indices of bars in decreasing order
        #     (monotonic decreasing stack). This helps find bounded
        #     depressions where water can collect.
        stack = []
        total_water = 0

        # STEP 2: Main loop / recursion
        #   - Traverse each bar. When we find a bar taller than the
        #     one at the top of the stack, we may trap water.
        for i, h in enumerate(height):
            # While stack isn't empty and current bar is taller
            # than the bar at the top of the stack:
            while stack and h > height[stack[-1]]:
                # STEP 3: Update state / bookkeeping
                #   - Pop the bottom of the potential "valley"
```

```
                bottom = stack.pop()
                # If stack is empty, no left boundary → stop
                if not stack:
                    break
                # Distance between current bar and new top of stack
                distance = i - stack[-1] - 1
                # Height of trapped water = min(left, right) - bottom
                bounded_height = min(h, height[stack[-1]]) - height[bottom]
                total_water += distance * bounded_height
            # Push current index onto stack
            stack.append(i)

        # STEP 4: Return result
        #   - total_water already accounts for all trapped units
        return total_water

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.trap([0,1,0,2,1,0,1,3,2,1,2,1]) == 6

    #  Test 2: Edge case - no water trapped
    assert sol.trap([1,2,3,4,5]) == 0

    #  Test 3: Tricky/negative - symmetric dip
    assert sol.trap([2,0,2]) == 2

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace `trap([2, 0, 2])` step by step:

1. **Initialize**:

- `stack = []`
- `total_water = 0`

2. **i = 0, h = 2**
   - Stack is empty → push index `0`
   - `stack = [0]`
3. **i = 1, h = 0**
   - `0 <= height[0] (2)` → no while loop
   - Push index `1`
   - `stack = [0, 1]`
4. **i = 2, h = 2**
   - Enter while loop: `2 > height[1] = 0` → true
     - Pop `bottom = 1`
     - Stack not empty → `stack = [0]`
     - `distance = 2 - 0 - 1 = 1`
     - `bounded_height = min(2, height[0]=2) - height[1]=0 = 2`
     - Add `1 * 2 = 2` to `total_water` → now `2`
   - Now check while again: `2 > height[0] = 2`? No → exit
   - Push index `2` → `stack = [0, 2]`
5. **Loop ends** → return `total_water = 2`

Final Output: `2`
Key Insight: The monotonic stack lets us find "containers" bounded by higher bars on left and right. Each time we pop, we compute the water trapped **above** that popped bar.

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each bar is pushed and popped at most once → total operations 2n → linear.

- **Space Complexity**: `O(n)`

  In worst case (strictly decreasing heights), stack stores all `n` indices.

10. Daily Temperatures

Pattern: Stack & Monotonic Stack

Problem Statement

Given an array of integers `temperatures` representing the daily temperatures, return an array `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, put `0` instead.

Sample Input & Output

```
Input: [73,74,75,71,69,72,76,73]
Output: [1,1,4,2,1,1,0,0]
Explanation: On day 0 (73°F), the next warmer day is day 1 →
wait 1 day.
```

```
Input: [30,40,50,60]
Output: [1,1,1,0]
Explanation: Each day is followed immediately by a warmer day,
except the last.
```

```
Input: [30,30,30,30]
Output: [0,0,0,0]
Explanation: No warmer day ever occurs → all zeros.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        # STEP 1: Initialize structures
        #   - Use a stack to store indices of days with unresolved warmer days.
        #   - Result array initialized to 0 handles default (no warmer day).
        n = len(temperatures)
        answer = [0] * n
        stack = []  # stores indices; maintains decreasing temp order

        # STEP 2: Main loop / recursion
        #   - For each day, check if current temp resolves previous days.
        #   - Invariant: stack holds indices with temps in decreasing order.
        for i in range(n):
            # STEP 3: Update state / bookkeeping
            # While current temp > temp at top of stack, we found a warmer day.
            # Pop and compute wait days = i - popped_index.
            while stack and temperatures[i] > temperatures[stack[-1]]:
                prev_index = stack.pop()
                answer[prev_index] = i - prev_index
            # Push current index to resolve later
            stack.append(i)

        # STEP 4: Return result
        # Already filled with 0s for unresolved days→no extra handling needed.
        return answer

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.dailyTemperatures([73,74,75,71,69,72,76,73])==[1,1,4,2,1,1,0,0]

    #  Test 2: Edge case - strictly increasing
    assert sol.dailyTemperatures([30,40,50,60]) == [1,1,1,0]
```

```
    #   Test 3: Tricky/negative - all equal
    assert sol.dailyTemperatures([30,30,30,30]) == [0,0,0,0]

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `temperatures = [73, 74, 75, 71, 69, 72, 76, 73]`.

Initial state:
- `answer = [0,0,0,0,0,0,0,0]`
- `stack = []`

---

i = 0 (temp = 73)
- Stack empty → push 0
- `stack = [0]`

i = 1 (temp = 74)
- Compare 74 > 73 (temp at `stack[-1] = 0`) → **YES**
- Pop 0 → `answer[0] = 1 - 0 = 1`
- Push 1
- `answer = [1,0,0,0,0,0,0,0]`
- `stack = [1]`

i = 2 (temp = 75)
- 75 > 74 → pop 1 → `answer[1] = 2 - 1 = 1`
- Push 2
- `answer = [1,1,0,0,0,0,0,0]`
- `stack = [2]`

i = 3 (temp = 71)
- 71 < 75 → no pop
- Push 3
- `stack = [2,3]`

i = 4 (temp = 69)
- 69 < 71 → no pop
- Push 4
- `stack = [2,3,4]`

i = 5 (temp = 72)
- 72 > 69 → pop 4 → `answer[4] = 5 - 4 = 1`
- Now top = 3 (temp=71), 72 > 71 → pop 3 → `answer[3] = 5 - 3 = 2`
- Now top = 2 (temp=75), 72 < 75 → stop
- Push 5
- `answer = [1,1,0,2,1,0,0,0]`
- `stack = [2,5]`

i = 6 (temp = 76)
- 76 > 72 → pop 5 → `answer[5] = 6 - 5 = 1`
- 76 > 75 → pop 2 → `answer[2] = 6 - 2 = 4`
- Stack empty → push 6
- `answer = [1,1,4,2,1,1,0,0]`
- `stack = [6]`

i = 7 (temp = 73)
- 73 < 76 → no pop
- Push 7
- `stack = [6,7]`

Loop ends. Remaining indices (6,7) stay 0 → correct.

Final output: `[1,1,4,2,1,1,0,0]`

Complexity Analysis

- **Time Complexity**: `O(n)`

  Each index is pushed and popped **at most once**. The inner `while` loop may seem nested, but total operations are bounded by `2n` → linear.

- **Space Complexity**: `O(n)`

  The `stack` can hold up to `n` indices in worst case (e.g., strictly decreasing temps). The `answer` array is output space and typically not counted, but if included, still `O(n)`.

11. Longest Valid Parentheses

Pattern: Stack & Monotonic Stack

Problem Statement

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

A valid parentheses substring is a contiguous part of the string where every opening parenthesis `'('` has a matching closing `')'` in the correct order.

Sample Input & Output

```
Input: "(()"
Output: 2
Explanation: The longest valid substring is "()", length = 2.
```

```
Input: ")()())"
Output: 4
Explanation: The valid substrings are "()()" → length 4.
```

```
Input: ""
Output: 0
Explanation: Empty string has no valid parentheses.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def longestValidParentheses(self, s: str) -> int:
        # STEP 1: Initialize structures
        #   - Use a stack to store indices of '(' and unmatched ')'
        #   - Push -1 as base for length calculation (before start)
        stack = [-1]
        max_len = 0

        # STEP 2: Main loop / recursion
        #   - Iterate over each character with its index
        for i, char in enumerate(s):
            if char == '(':
                # Push index of '(' for future matching
                stack.append(i)
            else:
                # char is ')'
                stack.pop()  # Remove top (either '(' or base)

                if not stack:
                    # No matching '(': current ')' is unmatched
                    # Push current index as new base for future
                    stack.append(i)
                else:
                    # Valid substring ends at i
                    # Length = i - stack[-1] (last unmatched index)
                    current_len = i - stack[-1]
                    max_len = max(max_len, current_len)

        # STEP 4: Return result
        #   - max_len already handles empty or all-invalid cases
        return max_len

# ------------------ INLINE TESTS ------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.longestValidParentheses("(()") == 2, "Test 1 failed"

    #  Test 2: Edge case
```

```
    assert sol.longestValidParentheses("") == 0, "Test 2 failed"

    #   Test 3: Tricky/negative
    assert sol.longestValidParentheses(")()())") == 4, "Test 3 failed"

    print("  All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `s = ")()())"` step by step.

Initial state:
- `stack = [-1]`
- `max_len = 0`

Step 1: `i=0`, `char=')'`
- Pop `-1` → stack becomes `[]`
- Stack is empty → push `0` as new base
- `stack = [0]`, `max_len = 0`

Step 2: `i=1`, `char='('`
- Push `1`
- `stack = [0, 1]`, `max_len = 0`

Step 3: `i=2`, `char=')'`
- Pop `1` → stack = `[0]`
- Stack not empty → `current_len = 2 - 0 = 2`
- `max_len = max(0, 2) = 2`

Step 4: `i=3`, `char='('`
- Push `3`
- `stack = [0, 3]`, `max_len = 2`

Step 5: `i=4`, `char=')'`
- Pop `3` → stack = `[0]`

- `current_len = 4 - 0 = 4`
- `max_len = max(2, 4) = 4`

Step 6: `i=5`, `char=')'`
- Pop `0` → stack = `[]`
- Stack empty → push `5`
- `stack = [5]`, `max_len = 4`

Final return: `4`

Key insight: The stack always holds indices of **unmatched** parentheses. The gap between current index and the last unmatched index gives valid length.

Complexity Analysis

- **Time Complexity**: `O(n)`

  We iterate through the string exactly once. Each character is pushed and popped at most once → total operations 2n.

- **Space Complexity**: `O(n)`

  In worst case (e.g., all `'('`), the stack stores all `n` indices.

12. Implement Queue using Stacks

Pattern: Queue & Variants

Problem Statement

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `pop`, `peek`, and `empty`).

Implement the `MyQueue` class:
- `void push(int x)` — Pushes element x to the back of the queue.
- `int pop()` — Removes the element from the front of the queue and returns it.
- `int peek()` — Returns the element at the front of the queue.
- `boolean empty()` — Returns `true` if the queue is empty, `false` otherwise.

Notes:
- You must use **only** standard stack operations: `push`, `pop`, `top`, and `empty`.
- Stacks may not be natively available; you can use Python lists with `append()` and `pop()` as stack operations.

Sample Input & Output

```
Input: ["MyQueue", "push", "push", "peek", "pop", "empty"]
       [[], [1], [2], [], [], []]
Output: [null, null, null, 1, 1, false]
Explanation:
  - After pushing 1 and 2, the queue is [1, 2] (front to back).
  - peek() returns 1 (front element).
  - pop() removes and returns 1.
  - empty() returns false since 2 remains.
```

```
Input: ["MyQueue", "push", "empty"]
       [[], [5], []]
Output: [null, null, false]
Explanation: Queue has one element → not empty.
```

```
Input: ["MyQueue", "empty", "push", "pop", "empty"]
       [[], [], [10], [], []]
Output: [null, true, null, 10, true]
Explanation: Starts empty → push 10 → pop returns 10 → now empty again.
```

LeetCode Editorial Solution + Inline Tests

```python
class MyQueue:
    def __init__(self):
        # STEP 1: Initialize two stacks
        #   - 'in_stack' handles incoming pushes
        #   - 'out_stack' handles outgoing pops/peeks
        self.in_stack = []
        self.out_stack = []

    def push(self, x: int) -> None:
        # STEP 2: Push to in_stack directly
        #   - No need to rebalance now; defer cost to pop/peek
        self.in_stack.append(x)

    def pop(self) -> int:
        # STEP 3: Ensure out_stack has front element
        #   - If out_stack is empty, transfer all from in_stack
        self._transfer_if_needed()
        # Pop from out_stack (which holds oldest elements)
        return self.out_stack.pop()

    def peek(self) -> int:
        # STEP 4: Same as pop but don't remove element
        self._transfer_if_needed()
        return self.out_stack[-1]

    def empty(self) -> bool:
        # STEP 5: Queue is empty only if both stacks are empty
        return not self.in_stack and not self.out_stack

    def _transfer_if_needed(self):
        # Helper: Move elements from in_stack to out_stack
        #   - Only when out_stack is empty
        #   - Reverses order so oldest becomes top of out_stack
        if not self.out_stack:
            while self.in_stack:
                self.out_stack.append(self.in_stack.pop())

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = MyQueue()
```

```python
    #  Test 1: Normal case
    sol.push(1)
    sol.push(2)
    assert sol.peek() == 1
    assert sol.pop() == 1
    assert sol.empty() == False

    #  Test 2: Edge case - empty queue
    q2 = MyQueue()
    assert q2.empty() == True
    q2.push(5)
    assert q2.empty() == False

    #  Test 3: Tricky/negative - interleaved ops
    q3 = MyQueue()
    assert q3.empty() == True
    q3.push(10)
    assert q3.pop() == 10
    assert q3.empty() == True
    q3.push(20)
    q3.push(30)
    assert q3.peek() == 20
    assert q3.pop() == 20
    assert q3.pop() == 30
    assert q3.empty() == True

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 3** step by step:

1. `q3 = MyQueue()`
    - `in_stack = []`, `out_stack = []`
    - State: both stacks empty.

2. `q3.empty()` → `True`
    - Checks `not [] and not []` → `True`.
3. `q3.push(10)`
    - Appends `10` to `in_stack`.
    - State: `in_stack = [10]`, `out_stack = []`.
4. `q3.pop()`
    - Calls `_transfer_if_needed()`:
        - `out_stack` is empty → transfer from `in_stack`.
        - `in_stack.pop()` → `10`, append to `out_stack`.
        - Now: `in_stack = []`, `out_stack = [10]`.
    - Then `out_stack.pop()` → returns `10`.
    - Final state: both stacks empty again.
5. `q3.empty()` → `True`
    - Confirmed.
6. `q3.push(20)` → `in_stack = [20]`
7. `q3.push(30)` → `in_stack = [20, 30]`
    - Note: order in `in_stack` is `[20, 30]` (20 pushed first).
8. `q3.peek()`
    - `_transfer_if_needed()`: `out_stack` is empty → transfer:
        - Pop `30` from `in_stack` → push to `out_stack` → `out_stack = [30]`
        - Pop `20` → push → `out_stack = [30, 20]`
        - Now `in_stack = []`, `out_stack = [30, 20]`
    - `out_stack[-1]` is `20` → correct front of queue.
9. `q3.pop()` → **returns** `20`
    - `out_stack` becomes `[30]`.
10. `q3.pop()` → **returns** `30`

- `out_stack` becomes `[]`.

11. `q3.empty()` → `True`

Key Insight: The two-stack design **delays reversal** until needed, making `push` O(1) amortized and `pop/peek` O(1) amortized.

Complexity Analysis

- **Time Complexity**: `O(1)` amortized per operation

  Each element is pushed and popped **at most twice** (once in each stack). Over `n` operations, total work is `O(n)` → average `O(1)` per op. Worst-case single `pop` or `peek` is `O(n)`, but amortized it's constant.

- **Space Complexity**: `O(n)`

  We store up to `n` elements across both stacks. No extra space beyond input size.

13. Design Hit Counter

Pattern: Queue & Variants

Problem Statement

Design a hit counter which counts the number of hits received in the past 5 minutes (i.e., the last 300 seconds).

Your system should accept a timestamp parameter (in seconds granularity), and you may assume that calls are being made to the system in chronological order (i.e., timestamp is monotonically increasing). Several hits may arrive roughly at the same time.

Implement the `HitCounter` class:
- `HitCounter()` Initializes the object of the hit counter.
- `void hit(int timestamp)` Records a hit at the given timestamp.
- `int getHits(int timestamp)` Returns the number of hits in the past 5 minutes from the given timestamp (i.e., `[timestamp - 299, timestamp]`).

Note: The interval is **inclusive** and exactly 300 seconds long (e.g., at `t=300`, valid hits are from `1` to `300`).

Sample Input & Output

```
Input: ["HitCounter", "hit", "hit", "hit", "getHits", "hit", "getHits"]
       [[], [1], [2], [3], [4], [300], [300]]
Output: [null, null, null, null, 3, null, 4]
Explanation: At timestamp 4, hits at 1,2,3 are within
[4-299= -295 → 4] → all valid.
At timestamp 300, hits at 1,2,3,300 are in [1,300] → all valid.
```

```
Input: ["HitCounter", "hit", "getHits", "getHits"]
       [[], [1], [300], [301]]
Output: [null, null, 1, 0]
Explanation: At t=301, window is [2,301]; hit at t=1 is expired.
```

```
Input: ["HitCounter", "hit", "hit", "getHits"]
       [[], [1], [1], [1]]
Output: [null, null, null, 2]
Explanation: Multiple hits at same timestamp are allowed.
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import deque
from typing import List

class HitCounter:
    def __init__(self):
        # STEP 1: Initialize structures
        #   - Use deque to store timestamps of hits
        #   - Deque allows O(1) popleft (FIFO) for expired hits
        self.hits = deque()
```

```python
    def hit(self, timestamp: int) -> None:
        # STEP 2: Record new hit
        #   - Append current timestamp to right end
        self.hits.append(timestamp)

    def getHits(self, timestamp: int) -> int:
        # STEP 3: Remove expired hits
        #   - Expired: timestamp <= (current - 300)
        #   - Because window is [timestamp - 299, timestamp]
        #     → anything <= timestamp - 300 is too old
        while self.hits and self.hits[0] <= timestamp - 300:
            self.hits.popleft()

        # STEP 4: Return result
        #   - Length of deque = valid hits in last 300 sec
        return len(self.hits)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = HitCounter()

    #  Test 1: Normal case
    sol.hit(1)
    sol.hit(2)
    sol.hit(3)
    assert sol.getHits(4) == 3, f"Expected 3, got {sol.getHits(4)}"

    #  Test 2: Edge case - expiration
    sol.hit(300)
    assert sol.getHits(300) == 4, f"Expected 4, got {sol.getHits(300)}"
    assert sol.getHits(301) == 3, f"Expected 3, got {sol.getHits(301)}"

    #  Test 3: Tricky/negative - multiple hits at same time
    counter2 = HitCounter()
    counter2.hit(1)
    counter2.hit(1)
    assert counter2.getHits(1) == 2, f"Expected 2, got {counter2.getHits(1)}"

    print(" All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 2** step-by-step:

1. **Initialize**: `sol = HitCounter()`
   → `self.hits = deque([])` (empty)
2. **Call `sol.hit(1)`**
   → Append `1` → `deque([1])`
3. **Call `sol.hit(2)`**
   → Append `2` → `deque([1, 2])`
4. **Call `sol.hit(3)`**
   → Append `3` → `deque([1, 2, 3])`
5. **Call `sol.hit(300)`**
   → Append `300` → `deque([1, 2, 3, 300])`
6. **Call `sol.getHits(300)`**
   - Compute window: `300 - 300 = 0` → remove timestamps 0
   - `self.hits[0] = 1` → $1 > 0$ → nothing removed
   - Return `len(deque) = 4`
7. **Call `sol.getHits(301)`**
   - Compute window: `301 - 300 = 1` → remove timestamps 1
   - `self.hits[0] = 1` → 1 1 → **remove it**
     → deque becomes `[2, 3, 300]`
   - Now `self.hits[0] = 2` → $2 > 1$ → stop
   - Return `len = 3`

Final state: `deque([2, 3, 300])`
Key insight: We lazily remove expired hits **only when querying**, not on every `hit()` — efficient!

Complexity Analysis

- **Time Complexity**:
  - `hit()`: `O(1)` — simple append
  - `getHits()`: **Amortized** `O(1)`
    > Each hit is added once and removed once. Over many calls, total work is linear in number of hits. Worst-case single call is `O(n)`, but amortized it's constant.
- **Space Complexity**: `O(n)`
  > In worst case (e.g., all hits within last 300 sec), we store all `n` timestamps in the deque.

14. Maximum Frequency Stack

Pattern: Queue & Variants

Problem Statement

Implement `FreqStack`, a class which simulates a stack-like data structure that supports the following operations:
- `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.

You may assume that `pop()` is never called on an empty stack.

Sample Input & Output

```
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:
After pushes: stack = [5,7,5,7,4,5] (left = bottom)
```

```
Frequencies: 5→3, 7→2, 4→1
1st pop → 5 (freq=3, topmost among max freq)
2nd pop → 7 (now 5→2, 7→2; 7 is closer to top than the other 5)
3rd pop → 5
4th pop → 4
```

```
Input: ["FreqStack", "push", "push", "pop", "pop"]
       [[], [10], [10], [], []]
Output: [null, null, null, 10, 10]
Explanation: Both have same frequency; last-in wins.
```

```
Input: ["FreqStack", "push", "pop"]
       [[], [42], []]
Output: [null, null, 42]
Explanation: Single element edge case.
```

LeetCode Editorial Solution + Inline Tests

```python
from collections import defaultdict

class FreqStack:
    def __init__(self):
        # STEP 1: Initialize structures
        #   - freq: tracks current frequency of each value
        #   - group: maps frequency -> stack of values with that freq
        #   - max_freq: current maximum frequency in the stack
        self.freq = defaultdict(int)
        self.group = defaultdict(list)
        self.max_freq = 0

    def push(self, val: int) -> None:
        # STEP 2: Update frequency of val
        self.freq[val] += 1
        f = self.freq[val]

        # STEP 3: Add val to its frequency group
        #   - This maintains insertion order within same frequency
```

```
        self.group[f].append(val)

        # STEP 4: Update global max frequency if needed
        if f > self.max_freq:
            self.max_freq = f

    def pop(self) -> int:
        # STEP 5: Pop from the stack of max frequency
        val = self.group[self.max_freq].pop()

        # STEP 6: Decrement frequency of popped value
        self.freq[val] -= 1

        # STEP 7: If max_freq group is now empty, reduce max_freq
        if not self.group[self.max_freq]:
            self.max_freq -= 1

        return val

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = FreqStack()

    #  Test 1: Normal case
    sol.push(5)
    sol.push(7)
    sol.push(5)
    sol.push(7)
    sol.push(4)
    sol.push(5)
    assert sol.pop() == 5  # most frequent (3x), topmost
    assert sol.pop() == 7  # now 5:2, 7:2 → 7 closer to top
    assert sol.pop() == 5
    assert sol.pop() == 4

    #  Test 2: Edge case - single element
    sol2 = FreqStack()
    sol2.push(42)
    assert sol2.pop() == 42

    #  Test 3: Tricky/negative - tie-breaking by recency
    sol3 = FreqStack()
```

```
sol3.push(10)
sol3.push(20)
sol3.push(10)
sol3.push(20)
# freq: 10→2, 20→2; top is 20
assert sol3.pop() == 20  # same freq, but 20 is newer
assert sol3.pop() == 10  # now both freq=1, but 10 is top
print("  All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1** step by step:

1. `push(5)`
   - `freq[5]` becomes 1
   - `group[1] = [5]`
   - `max_freq = 1`
2. `push(7)`
   - `freq[7] = 1`
   - `group[1] = [5, 7]`
   - `max_freq` still 1
3. `push(5)`
   - `freq[5] = 2`
   - `group[2] = [5]`
   - `max_freq = 2`
4. `push(7)`

- `freq[7] = 2`
- `group[2] = [5, 7]`
- `max_freq = 2`

5. **`push(4)`**
   - `freq[4] = 1`
   - `group[1] = [5, 7, 4]`
   - `max_freq` unchanged (2)
6. **`push(5)`**
   - `freq[5] = 3`
   - `group[3] = [5]`
   - `max_freq = 3`

Now call **`pop()`**: - Pop from `group[3]` → `val = 5`
- `freq[5]` becomes 2
- `group[3]` is now empty → `max_freq = 2`
- Return `5`

Next **`pop()`**: - Pop from `group[2]` → `val = 7` (last in group[2])
- `freq[7] = 1`
- `group[2] = [5]` → not empty, so `max_freq` stays 2
- Return `7`

Next **`pop()`**: - Pop from `group[2]` → `val = 5`
- `freq[5] = 1`
- `group[2]` now empty → `max_freq = 1`
- Return `5`

Final **`pop()`**: - Pop from `group[1]` → `val = 4` (last in group[1] = [5,7,4])
- `freq[4] = 0`
- `group[1] = [5,7]` → not empty
- Return `4`

This matches expected output: `[5,7,5,4]`.

Complexity Analysis

- **Time Complexity**: `O(1)` per operation

  Both `push` and `pop` use only dictionary lookups, list appends, and pops — all O(1) average time in Python. No loops or searches.

- **Space Complexity**: `O(N)`

  We store each pushed value once in `group` and track its frequency in `freq`, where `N` is total number of operations. In worst case, all values are distinct → `O(N)` space.

15. Backspace String Compare

Pattern: Queue & Variants

Problem Statement

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, the text will continue empty.

Sample Input & Output

```
Input: s = "ab#c", t = "ad#c"
Output: true
Explanation: Both s and t become "ac".
```

```
Input: s = "ab##", t = "c#d#"
Output: true
Explanation: Both become "" (empty string).
```

```
Input: s = "a#c", t = "b"
Output: false
Explanation: s becomes "c", but t remains "b".
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def backspaceCompare(self, s: str, t: str) -> bool:
        # STEP 1: Initialize structures
        #   - Use lists as stacks to simulate typing with backspace
        #   - Each '#' pops the last character (if any)

        def build(string: str) -> List[str]:
            stack = []
            for char in string:
                if char != '#':
                    stack.append(char)
                elif stack:  # only pop if not empty
                    stack.pop()
            return stack

        # STEP 2: Main loop / recursion
        #   - Process both strings independently using helper
        #   - Compare final stacks for equality

        # STEP 3: Update state / bookkeeping
        #   - No global state; each string processed cleanly

        # STEP 4: Return result
        #   - Direct comparison of resulting character lists
        return build(s) == build(t)

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()
```

```python
    #  Test 1: Normal case
    assert sol.backspaceCompare("ab#c", "ad#c") == True

    #  Test 2: Edge case (both become empty)
    assert sol.backspaceCompare("ab##", "c#d#") == True

    #  Test 3: Tricky/negative (different results)
    assert sol.backspaceCompare("a#c", "b") == False

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**: `s = "ab#c"`, `t = "ad#c"`

Step 1: Call `build("ab#c")`

- `stack = []`
- `char = 'a'` → not `#` → `stack = ['a']`
- `char = 'b'` → not `#` → `stack = ['a', 'b']`
- `char = '#'` → is `#` and stack not empty → pop → `stack = ['a']`
- `char = 'c'` → not `#` → `stack = ['a', 'c']`
- Return `['a', 'c']`

Step 2: Call `build("ad#c")`

- `stack = []`
- `char = 'a'` → `stack = ['a']`
- `char = 'd'` → `stack = ['a', 'd']`
- `char = '#'` → pop → `stack = ['a']`
- `char = 'c'` → `stack = ['a', 'c']`
- Return `['a', 'c']`

Step 3: Compare results

- `['a', 'c'] == ['a', 'c']` → `True`

Final Output: `True`

Key Insight:
Instead of simulating backspaces in-place (which is messy), we **reconstruct** what the final string *should* look like using a stack. Each `#` removes the last typed character—exactly what a stack's `pop()` does. This is a classic **stack simulation** pattern, often grouped under "Queue & Variants" because stacks are LIFO queues.

Complexity Analysis

- **Time Complexity**: `O(n + m)`
  > We traverse each string once (`n = len(s)`, `m = len(t)`). Each character is pushed or popped at most once → linear time.
- **Space Complexity**: `O(n + m)`
  > In worst case (no `#`), we store all characters of both strings in stacks. If many `#`, space reduces, but worst-case scales with input size.