DSA Problems Summary with Stack – Approaches & Code

This comprehensive report covers essential stack-based data structure and algorithm problems, providing detailed explanations, approaches, and code implementations to help master stack concepts in competitive programming and technical interviews.

---

1. Backspace String Compare (LeetCode 844)

Problem Description

Given two strings `s` and `t`, return `true` if they are equal when both are typed into empty text editors where `'#'` means a backspace character that removes the previous character.

Example:

- Input: `s = "ab#c"`, `t = "ad#c"`
- Output: `true` (both become "ac")

Approach

The solution leverages a **stack data structure** to simulate the typing process:

1. **Character Processing**: Iterate through each character in the string
2. **Backspace Handling**: If the character is `'#'`, pop the last element from the stack (if not empty)
3. **Regular Character**: Push the character onto the stack
4. **Final Comparison**: Convert both stacks to strings and compare for equality

Time Complexity: O(n + m) where n and m are lengths of strings s and t
Space Complexity: O(n + m) for the stacks

C++ Implementation

```
class Solution {
public:
```

```
    bool backspaceCompare(string s, string t) {
        auto build = [](string str) {
            stack<char> st;
            for (char c : str) {
                if (c == '#') {
                    if (!st.empty()) st.pop();
                } else {
                    st.push(c);
                }
            }
            string result;
            while (!st.empty()) {
                result += st.top();
                st.pop();
            }
            reverse(result.begin(), result.end());
            return result;
        };
        return build(s) == build(t);
    }
};
```

---

2. Crawler Log Folder (LeetCode 1598)

Problem Description

A file system keeps a log of every folder change operation. Given a list of folder operations, determine how many steps you need to go back to reach the main folder after performing all operations.

Operations:

- `"../"` : Move to parent folder (go back one step)
- `"./"` : Remain in current folder
- `"x/"` : Move to folder named x

Approach

This problem can be solved efficiently using either a stack or a simple counter approach. The counter method is more space-efficient:

1. **Initialize Counter**: Start with depth = 0 (main folder)

2. **Process Operations**:
    - `"../"` : Decrease depth by 1 (minimum 0)
    - `"./"` : No change in depth
    - `"folder/"` : Increase depth by 1
3. **Return Final Depth**: The counter represents steps needed to return to main folder

Time Complexity: O(n) where n is the number of operations
Space Complexity: O(1) using counter approach

Optimized Python Implementation

```python
class Solution:
    def minOperations(self, logs: List[str]) -> int:
        depth = 0
        for log in logs:
            if log == "../":
                if depth > 0:
                    depth -= 1
            elif log != "./":
                depth += 1
        return depth
```

---

3. Baseball Game (LeetCode 682)

Problem Description

You are keeping score for a baseball game with the following operations:

- **Integer x**: Record a new score of x
- **"+"**: Record a new score that is the sum of the previous two scores
- **"D"**: Record a new score that is double the previous score
- **"C"**: Cancel the previous score, removing it from the record

Return the sum of all scores on the record after applying all operations.

Approach

Use a stack (or deque) to maintain the current valid scores:

1. **Process Each Operation**:

    - **Integer**: Push the score onto stack
    - **"+"**: Calculate sum of last two scores and push result
    - **"D"**: Double the last score and push result
    - **"C"**: Remove (pop) the last score
2. **Calculate Final Sum**: Sum all remaining scores in the stack

Time Complexity: O(n) where n is the number of operations
Space Complexity: O(n) for storing valid scores

Python Implementation

```python
from collections import deque

class Solution:
    def calPoints(self, operations: List[str]) -> int:
        stack = deque()
        for op in operations:
            if op == '+':
                stack.append(stack[-1] + stack[-2])
            elif op == 'D':
                stack.append(2 * stack[-1])
            elif op == 'C':
                stack.pop()
            else:
                stack.append(int(op))
        return sum(stack)
```

---

4. General Use Cases of Stack in DSA

When to Use Stack

Stacks follow the **Last In, First Out (LIFO)** principle, making them ideal for various algorithmic scenarios:

Use Case	Why Stack is Perfect	Example Problems
Balanced Parentheses	Match opening brackets with most recent closing ones	Valid Parentheses, Remove Invalid Parentheses
Expression Evaluation	Handle operator precedence and operand order	Infix to Postfix, Calculator

Use Case	Why Stack is Perfect	Example Problems
Recursion Simulation	Mimic the implicit call stack	Tree Traversal, Backtracking
DFS Traversal	Explore paths deeply before backtracking	Graph DFS, Maze Solving
Undo Operations	Reverse the most recent actions first	Text Editor, Game States
Next/Previous Element	Track recent elements for comparison	Next Greater Element, Stock Span
Backtracking Algorithms	Store and restore previous states	N-Queens, Sudoku Solver
Browser Navigation	Implement back/forward functionality	History Management

Key Stack Operations and Complexities

Operation	Time Complexity	Description
`push()`	O(1)	Add element to top
`pop()`	O(1)	Remove and return top element
`top()/peek()`	O(1)	View top element without removing
`empty()`	O(1)	Check if stack is empty
`size()`	O(1)	Get number of elements

Example: Valid Parentheses Implementation

Here's a classic example demonstrating stack usage for balanced parentheses:

```
def is_valid_parentheses(s):
    """
    Determine if parentheses in string are properly balanced.

    Args:
        s (str): String containing parentheses characters

    Returns:
        bool: True if balanced, False otherwise
    """
    stack = []
    mapping = {')': '(', ']': '[', '}': '{'}
```

```
    for char in s:
        if char in mapping.values():  # Opening bracket
            stack.append(char)
        elif char in mapping:  # Closing bracket
            if not stack or stack[-1] != mapping[char]:
                return False
            stack.pop()

    return not stack  # True if stack is empty (all matched)

# Example usage:
# is_valid_parentheses("()[]{}") → True
# is_valid_parentheses("([)]") → False
# is_valid_parentheses("{[()]}") → True
```

Advanced Stack Techniques

Monotonic Stack

Used for problems involving "next greater/smaller element":

- Maintain elements in increasing/decreasing order
- Pop elements that violate the monotonic property
- Useful for: Daily temperatures, Largest rectangle in histogram

Stack with Minimum

Implement a stack that supports getting minimum element in O(1):

- Use auxiliary stack to track minimums
- Applications: Design problems, optimization queries

Two Stacks Simulation

Implement queue using two stacks:

- One for enqueue operations
- One for dequeue operations
- Amortized O(1) for all operations

Problem-Solving Strategy

When encountering a new problem, consider using a stack if you need to:

1. **Process elements in reverse order** of their arrival
2. **Match pairs** (parentheses, tags, etc.)
3. **Implement undo functionality** or state restoration
4. **Simulate recursive behavior** iteratively
5. **Find nearest smaller/greater elements**
6. **Evaluate expressions** with operator precedence
7. **Implement depth-first traversal** algorithms

Understanding these patterns will help you quickly identify when a stack-based solution is appropriate and implement it efficiently.

This report serves as a comprehensive guide for mastering stack-based problem solving in data structures and algorithms. Practice these patterns regularly to build intuition for stack applications in competitive programming and technical interviews.