Day 134/180 Stack Problems

1:Backspace String Compare:

```
class Solution {
public:
    bool backspaceCompare(string s, string t) {
        stack<char> s1, s2; // Declare two stacks to process strings s and t

        // Process string s
        for(auto x : s) {
            if(x == '#') { // If the character is a backspace
                if(!s1.empty()) { // Check if stack s1 is not empty
                    s1.pop(); // Remove the top element (backspace)
                }
            } else { // If the character is not a backspace
                s1.push(x); // Push the character onto stack s1
            }
        }

        // Process string t
        for(auto x : t) {
            if(x == '#') { // If the character is a backspace
                if(!s2.empty()) { // Check if stack s2 is not empty
                    s2.pop(); // Remove the top element (backspace)
                }
            } else { // If the character is not a backspace
                s2.push(x); // Push the character onto stack s2
            }
        }
        // Compare the processed strings stored in stacks s1 and s2
        return s1 == s2; // Return true if both stacks are equal, false
otherwise
    }
};

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

2: Special Stack

```
import java.util.Stack; // Importing the Stack class from java.util package

class GfG {
    int minEle; // Variable to store the minimum element in the stack

    // Method to push an element onto the stack
    public void push(int a, Stack<Integer> s) {
        if (s.isEmpty()) { // If the stack is empty
            minEle = a; // Set the minimum element to the value being pushed
            s.push(a); // Push the value onto the stack
        } else {
            if (a < minEle) { // If the value being pushed is less than the
current minimum
                // Push 2*a - minEle onto the stack and update minEle to a
                s.push(2 * a - minEle);
                minEle = a;
            } else { // If the value being pushed is greater than or equal to the
current minimum
                s.push(a); // Simply push the value onto the stack
            }
        }
    }

    // Method to pop an element from the stack
    public int pop(Stack<Integer> s) {
        if (s.isEmpty()) { // If the stack is empty
            return -1; // Return -1 indicating underflow
        } else {
            int res; // Variable to store the popped value

            if (s.peek() < minEle) { // If the top element of the stack is less
than the current minimum
                // Set res to the current minimum and update minEle accordingly
                res = minEle;
                minEle = 2 * minEle - s.pop();
            } else { // If the top element of the stack is greater than or equal
to the current minimum
```

```
                // Simply pop the value from the stack and store it in res
                res = s.pop();
            }

            return res; // Return the popped value
        }
    }

    // Method to get the minimum element in the stack
    public int min(Stack<Integer> s) {
        if (s.isEmpty()) { // If the stack is empty
            return -1; // Return -1 indicating underflow
        }
        return minEle; // Return the current minimum element
    }

    // Method to check if the stack is full
    public boolean isFull(Stack<Integer> s, int n) {
        // Return true if the size of the stack is equal to n, otherwise return
false
        return s.size() == n;
    }

    // Method to check if the stack is empty
    public boolean isEmpty(Stack<Integer> s) {
        // Return true if the stack is empty, otherwise return false
        return s.isEmpty();
    }
}
```

3: Sort a Stack

```
void SortedStack::sort()
{
    // Create a new stack to hold sorted elements
    stack<int> s2;

    // Continue until the original stack is not empty
    while (!s.empty()) {
        // Get the top element of the original stack
        int t = s.top();
        s.pop();

        // Move elements from s2 to s until s2's top element is smaller
than t
        while (!s2.empty() && s2.top() > t) {
            s.push(s2.top()); // Move element from s2 to s
            s2.pop(); // Remove the moved element from s2
        }

        // Push the current element (t) into s2
        s2.push(t);
    }

    // Replace the original stack with the sorted stack (s2)
    s = s2;
}
```

4:Minimum Add to Make Parentheses Valid:

```
class Solution {
public:
    int minAddToMakeValid(string s) {
        stack<char> s1; // Create a stack to store opening parentheses '('
        int cnt = 0;    // Initialize a counter for unbalanced closing parentheses
')'
        // Iterate through each character in the string
        for(auto x:s){
            if(s1.empty()){ // If the stack is empty
                if(x == ')'){ // If the current character is ')'
                    cnt++;   // Increment the counter for unbalanced closing
parentheses
                } else {     // If the current character is '('
                    s1.push(x); // Push it onto the stack
                }
            }
            else { // If the stack is not empty
                if(x == ')'){ // If the current character is ')'
                    s1.pop(); // Pop the top element from the stack
                } else {     // If the current character is '('
                    s1.push(x); // Push it onto the stack
                }
            }
        }
        // After processing the entire string, the number of remaining elements in
the stack
        // represents the number of unbalanced opening parentheses '('
        // Add the count of unbalanced closing parentheses ')' and the size of the
stack
        // to get the total number of parentheses needed to make the string valid
        return cnt + s1.size();
    }
};
TC :- O(n), SC :- O(n)
```

(solve 4th problem in Constant Space).

```
class Solution {
public:
    int minAddToMakeValid(string s) {
        int open = 0, close = 0; // Variables to count the number of open and close parentheses
        int ans = 0; // Variable to store the minimum number of additions needed to make the string valid

        for(auto x:s){ // Iterate through each character in the string
            open += x == '('; // If the character is '(', increment the count of open parentheses
            close += x == ')'; // If the character is ')', increment the count of close parentheses

            if(close > open){ // If there are more close parentheses than open parentheses encountered so far
                open++; // Add an additional open parentheses to balance the count
                ans++; // Increment the answer count since an additional open parentheses is added
            }
        }

        // The difference between the count of open and close parentheses represents the number of missing parentheses to balance the string
        // Add this difference to the answer count to get the total minimum additions needed
        return ans + (open - close);
    }
};

TC :- O(n), SC :- O(1)
```