Sort Stack Elements CodeStudio

You are given an unsorted stack of integers. Implement a function to sort the stack in ascending order. Write a function **sortStack** that takes the unsorted stack as input and arranges its elements in ascending order using a single approach.

Example: Consider an initial unsorted stack: 12, 23, 3, -4, 4, 66

Output: -4 3 4 12 23 66

Approach 1: Recursive function to sort the stack

This approach involves recursively sorting the given stack. The **sortStack** function pops the top element, recursively sorts the remaining stack, and then places the element in its correct sorted position using the **pushSorted** helper function.

pushSorted function:

- The **pushSorted** function assists in pushing a value in the correct sorted position within the stack. It compares the value with the top element and either pushes the value or makes a recursive call to continue pushing.

Time Complexity:

- The recursive approach involves repeatedly popping elements and making recursive calls.
- Since each element requires a constant amount of work (popping, pushing, and comparisons), the time complexity is linear.
- **Time Complexity**: **O(n^2), where n is the number of elements in the stack** (for each element, we may perform n/2 comparisons on average).

Space Complexity:

- The space complexity of the recursive approach is influenced by the maximum depth of the call stack due to recursive calls.
- In the worst case, the depth of the call stack can be **n** (number of elements in the stack).
- **Space Complexity**: **O(n), due to the call stack space required for recursion**.

Approach 2: Iterative function to sort the stack

In this approach, an auxiliary stack (**auxStack**) is used to assist in sorting the input stack. The main idea is to pop elements from the input stack and insert them in the correct position within the auxiliary stack.

Steps:

1. Pop an element from the input stack.
2. Compare the element with the top element of the auxiliary stack.
3. If the element is greater, transfer elements from the auxiliary stack to the input stack until the correct position is found.
4. Push the element into the auxiliary stack.
5. Repeat steps 1-4 until the input stack is empty.
6. Transfer the sorted elements from the auxiliary stack back to the input stack.

Time Complexity:

- The approach involves iterating over each element in the input stack and making comparisons.
- Each element insertion into the auxiliary stack takes linear time.
- The total time complexity is linear.
- **Time Complexity**: **O(n^2), where n is the number of elements in the stack.**

Space Complexity:

- The space complexity is influenced by the space required for the auxiliary stack (**auxStack**).
- In the worst case, the auxiliary stack may hold all the elements.
- **Space Complexity**: **O(n), due to the auxiliary stack space requirement.**