Solution Review: Sort Values in Stack

This review provides a detailed analysis of the three different ways to solve the 'Sort Values in a Stack' challenge.

We'll cover the following

Solution # 1: Using a Temporary Stack

main.py

Stack.py

```
"""
1. Use a second temp_stack.
2. Pop value from mainStack.
3. If the value is greater or equal to the top of temp_stack,
  then push the value in temp_stack
  else pop all values from temp_stack
      and push them in mainStack
      and in the end push value in temp_stack
4.repeat from step 2 till mainStack is not empty.
5. When mainStack will be empty,
    temp_stack will have sorted values in descending order.
```

```
    temp_stack will have sorted values in descending order.
6. Now transfer values from temp_stack to mainStack
    to make values sorted in ascending order.
"""

from Stack import MyStack

def sort_stack(stack):
    temp_stack = MyStack()
    while not stack.is_empty():
        value = stack.pop()
        # if value is not none and larger, push it at the top of temp
        if temp_stack.peek() is not None and value >= temp_stack.peek
            temp_stack.push(value)
        else:
            while not temp_stack.is_empty() and value < temp_stack.pe
                stack.push(temp_stack.pop())
            # place value as the smallest element in temp_stack
```

This solution takes an iterative approach towards the problem. We create a helper stack called `temp_stack`. Its job is to hold the elements of `stack` in descending order.

The main functionality of the algorithm lies in the nested while loops. In the outer loop, we pop elements out of `stack` until it is empty. As long as the popped value is larger than the top value in `temp_stack`, we can push it in.

The inner loop begins when `stack.pop()` gives us a `value` which is smaller than the top of `temp_stack`. All the elements (they are sorted) of `temp_stack` are shifted back to `stack` and the `value` is pushed into `temp_stack`. This ensures that the bottom of `temp_stack` always holds the smallest value from `stack`.

When `stack` becomes empty, all the elements are in `temp_stack` in descending order. Now we simply push them back into `stack`, resulting in the whole stack being sorted in ascending fashion.

Time Complexity

The outer and inner loops both traverse all the **n** elements of the stack. Hence, the time complexity is $O(n^2)$.

Solution # 2: Recursive Sort

main.py

Stack.py

```
# Recursive approach

from Stack import MyStack

def sort_stack(stack):
    if (not stack.is_empty()):
        # Pop the top element off the stack
        value = stack.pop()
        # Sort the remaining stack recursively
        sort_stack(stack)
        # Push the top element back into the sorted stack
        insert(stack, value)

def insert(stack, value):
    if (stack.is_empty() or value < stack.peek()):
        stack.push(value)
    else:
        temp = stack.pop()
        insert(stack, value)
        stack.push(temp)

if __name__ == "__main__" :
    # Creating and populating the stack
    stack_obj = MyStack()
    stack_obj.push(2)
    stack_obj.push(97)
    stack_obj.push(4)
    stack_obj.push(42)
    stack_obj.push(12)
    stack_obj.push(60)
    stack_obj.push(23)
    # Sorting the stack
    sort_stack(stack_obj)
```

```
    # Printing the sorted stack
    print("Stack after sorting")
    print([stack_obj.pop() for i in range(stack_obj.size())])
```

The idea is to pop out all the elements from the stack in one recursive call. Once the stack is empty, we will push back values in a sorted order using the `insert` function.

At each call, we receive a partially sorted stack in which we insert the `value` being popped out at that recursive call. If the `value` is smaller than the top element if the stack, we simply push it to the top.

Otherwise, we call `insert` again until we can find the appropriate place for the `value` in the stack.

Time Complexity

The `sortStack` function is recursively called on all **n** elements. In the worst case, there are **n** calls to `insert` for each element. This pushes the time complexity up to $O(n^2)$. However, unlike the first solution, we do not need to create another stack.

Solution # 3: Simple Sorting

main.py

Stack.py

```
# Using built in sort method for Python lists

from Stack import MyStack
```

```
def sort_stack(stack):
  stack.stack_list.sort(reverse=True)
  return stack

if __name__ == "__main__" :
    # Creating and populating the stack
    stack = MyStack()
    stack.push(2)
    stack.push(97)
    stack.push(4)
    stack.push(42)
    stack.push(12)
    stack.push(60)
    stack.push(23)
    # Sorting the stack
    stack = sort_stack(stack)
    # Printing the sorted stack
    print("Stack after sorting")
    print([stack.pop() for i in range(stack.size())])
```

This is the most obvious solution. Simply use your favorite sorting algorithm to sort the stack list and return it. We aren't focusing on this solution very much because although it's more optimal, it misses the point of using stacks to sort the stack.

Time Complexity

The time complexity of this solution is that of your sorting algorithm of choice, which is most likely $O(nlogn)$ or something better than quadratic time.

Back

Challenge 5: Sort Values in a Stack

Challenge 6: Evaluate Postfix Expressi...

Completed

Report an Issue