Solution Review: Next Greater Element Using a Stack

This review provides a detailed analysis of the different ways to solve the 'Next Greater Element Using a Stack' challenge.

We'll cover the following

Solution # 1: Brute Force#

main.py

Stack.py

```
from Stack import MyStack

def next_greater_element(lst):
    res = []
    # Iterate list
    for i in range(0, len(lst)):
        # initialise nextGreatest to -1
        next_greatest = -1
        for j in range(i+1, len(lst)):
            # Update nextGreatest when greater value found
            if lst[i] < lst[j]:
                next_greatest = lst[j]
                break
        # append next greatest
        res.append(next_greatest)
        print(str(lst[i]) + " -- " + str(next_greatest))
    return res

if __name__ == "__main__" :
    next_greater_element([4, 6, 3, 2, 8, 1, 9, 9, 9])
```

This solution iterates over the list for each element and prints the first element greater than that element.

Time Complexity#

The time complexity of this code is in $O(n^2)$ because the list is iterated once for each element.

Solution # 2: Stack Iteration#

main.py

Stack.py

```
from Stack import MyStack

def next_greater_element(lst):
    stack = MyStack()
    res = [-1] * len(lst)
    # Reverse iterate list
    for i in range(len(lst) - 1, -1, -1):
        ''' While stack has elements and current element is greater
        than top element, pop all elements '''
        while not stack.is_empty() and stack.peek() <= lst[i]:
            stack.pop()
        ''' If stack has an element, top element will be
        greater than ith element '''
        if not stack.is_empty():
            res[i] = stack.peek()
        # push in the current element in stack
        stack.push(lst[i])
    for i in range(len(lst)):
        print(str(lst[i]) + " -- " + str(res[i]))
    return res

if __name__ == "__main__" :
    nge = next_greater_element([4, 6, 3, 2, 8, 1, 9, 9, 9])
```

In this solution, we use our own `MyStack` and `res`, a Python list.

The outer `for` loop begins from the end of `lst`. In each iteration, the `top` of the stack is compared with the current element in `lst`. Whenever the current value in `lst` is larger than `top`, that value is the **next greater element** in the list.

The `top` is popped and the current element in the list is again compared with the new top of the stack in the inner while loop. This loop stops when the top of the stack is bigger than the element of the list or the stack becomes empty.

If the stack is empty, it implies that the current element in `lst` does not have a next greater element. Hence, its corresponding index in `res` would retain the value of `-1`.

If the stack is not empty, then the current element has a value larger than it. Hence, the `top` value will be assigned to the corresponding index in `res`.

Continuing this process, we end up with a list containing all the next greater elements for all indices of `lst`.

Time Complexity#

Since we make one iteration over the list of **n** elements, the algorithm will work in *O(n)*. This is a significant improvement over the brute force method's runtime complexity of $O(n^2)$.

Back
Challenge 7: Next Greater Element Usi...

Next
Challenge 8: Check Balanced Parenth...

Mark as Completed

Report an Issue