Monotonic Stacks: Understanding the Concept and Implementation

Introduction

A **monotonic stack** is a stack that maintains elements in a strictly increasing or decreasing order. Instead of storing the values directly, we typically store their **indices** to efficiently retrieve information about their relative positions.

Monotonic stacks are particularly useful for solving problems that require finding the **next greater/smaller element** in a list, as they allow processing in **O(n) time complexity**, significantly improving upon the naive **O(n^2) approach**.

Key Properties of Monotonic Stacks

- **Store indices, not actual values**
- **Maintain order:**
    1. **Increasing Stack** (used for Next Greater Element): The top always holds the smallest unprocessed value.
    2. **Decreasing Stack** (used for Next Smaller Element): The top always holds the largest unprocessed value.
- **Processing Steps:**
    1. Iterate through all elements.
    2. Pop elements from the stack that do not maintain the order property.
    3. Update results for popped elements.
    4. Push the current index onto the stack.
    5. After the iteration, any remaining elements in the stack do not have a next greater/smaller element.

Example Problem: Next Greater Element (NGE)

Given an array `{4,5,1,3,9,7,5}`, find the **Next Greater Element (NGE)** for each element. The NGE of an element is the first greater element that appears to its right. If no greater element exists, return `-1`.

Input:

nums = {4,5,1,3,9,7,5}

Expected Output:

{5,9,3,9,-1,-1,-1}

Step-by-Step Walkthrough

We process elements one by one while maintaining a stack that stores **indices** of elements for which we haven't yet found an NGE.

Stack States at Each Step

Index	Element	Stack Before	Action Taken	Stack After
0	4	Empty	Push 0	`{0}`
1	5	`{0}`	5 > 4 → Set `res[0] = 5`, pop `0`, push `1`	`{1}`
2	1	`{1}`	Push 2	`{2,1}`
3	3	`{2,1}`	3 > 1 → Set `res[2] = 3`, pop `2`, push `3`	`{3,1}`
4	9	`{3,1}`	9 > 3 → Set `res[3] = 9`, pop `3`9 > 5 → Set `res[1] = 9`, pop `1`, push `4`	`{4}`
5	7	`{4}`	Push 5	`{5,4}`
6	5	`{5,4}`	Push 6	`{6,5,4}`

After the loop, elements `{6,5,4}` remain in the stack with no NGE, so we set their values to `-1`.

C++ Implementation

```
#include <iostream>
#include <vector>
#include <stack>

using namespace std;

void nextGreaterElement(vector<int>& nums) {
    vector<int> res(nums.size(), -1);
    stack<int> s;
    int n = nums.size();

    for (int i = 0; i < n; i++) {
```

```
        while (!s.empty() && nums[i] > nums[s.top()]) {
            res[s.top()] = nums[i];
            s.pop();
        }
        s.push(i);
    }

    for (int num : res) {
        cout << num << " ";
    }
    cout << endl;
}

int main() {
    vector<int> nums = {4, 5, 1, 3, 9, 7, 5};
    nextGreaterElement(nums);
    return 0;
}
```

Explanation of Code

1. **Initialize `res` with `-1`** because elements without an NGE should default to `-1`.
2. **Iterate through the array**:
    - If the stack is **not empty** and the current element is **greater** than the element at the top index in the stack, pop the stack and set `res[stack.top()] = nums[i]`.
    - Push the current index onto the stack.
3. **Process remaining elements in the stack**:
    - These elements do not have an NGE, so their result remains `-1`.

Time Complexity Analysis

- **Each element is pushed onto the stack once and popped at most once → O(n)**.

Conclusion

Monotonic stacks are a powerful tool for range-based queries in arrays, significantly reducing time complexity from **$O(n^2)$ to $O(n)$**. This approach can be extended to problems like:

- **Next Smaller Element**
- **Largest Rectangle in Histogram**
- **Trapping Rain Water**