Table of Contents

1. Introduction

The Monotonic Stack algorithm is a technique used to solve problems that involve finding the next greater or smaller element in an array. It utilizes a stack data structure to efficiently track the elements that satisfy certain monotonicity conditions, either increasing or decreasing.

2. Algorithm

Here's the general idea and steps of the Monotonic Stack algorithm:

1. Initialize an empty stack to store the indices of the elements from the input array.
2. Iterate through the array from left to right.
3. For each element `current` encountered:
    - While the stack is not empty and the element at the top of the stack is less than `current`, pop the stack and process the popped element as the next greater (or smaller) element for the corresponding index.
    - Push the index of `current` onto the stack.
4. After the iteration finishes, if there are any remaining elements in the stack, process them accordingly (e.g., there is no next greater (or smaller) element for those indices).

The Monotonic Stack algorithm can be adapted to solve a variety of problems, such as finding the next greater element, next smaller element, or computing the span of elements.

Here's a Python implementation of the Monotonic Stack algorithm to find the next greater element for each element in an array:

```
def nextGreaterElements(nums):
    n = len(nums)
    result = [-1] * n
    stack = []
    for i in range(2 * n):
        while stack and nums[stack[-1]] < nums[i % n]:
            index = stack.pop()
            result[index] = nums[i % n]
        stack.append(i % n)

    return result
```

In this example, we consider the array to be circular, so we iterate twice the length of the array (`2 * n`). The modulo operation (`i % n`) allows us to wrap around and access the original array elements.

Note: the specific implementation of the Monotonic Stack algorithm may vary depending on the problem you are trying to solve. The above implementation demonstrates the basic concepts and can be adapted as needed.

3. Key Points

Here are a few key points to understand about the Monotonic Stack algorithm:

1. Monotonicity: The algorithm exploits the concept of monotonicity, which refers to the property of either increasing or decreasing elements. Depending on the problem, you can use either an increasing monotonic stack or a decreasing monotonic stack.
2. Stack Data Structure: The algorithm uses a stack to keep track of indices or elements that satisfy the monotonicity condition. The top of the stack always represents the most recent element that hasn't found the next greater or smaller element yet.
3. Iterative Approach: The algorithm usually utilizes a loop that iterates through the array or input elements from left to right. The loop performs comparisons and stack operations to determine the next greater or smaller element for each element.
4. Pop and Process: When a new element is encountered, the algorithm checks if it breaks the monotonicity condition with the elements on top of the stack. If it does, it pops the stack and processes the popped element accordingly (e.g., assigning it as the next greater or smaller element for the corresponding index).
5. Efficient Solution: The Monotonic Stack algorithm typically provides an efficient solution to problems that involve finding next greater or smaller elements. It avoids redundant comparisons by keeping track of only relevant elements in the stack.
6. Adaptability: The algorithm can be adapted to different variations of the problem, such as finding next greater/smaller elements, computing spans, or solving related scenarios. The

specific implementation may vary depending on the problem requirements.

4. Variations

There are two variations known as the Monotonic Increasing Stack and the Monotonic Decreasing Stack algorithms. These variations are used when the problem requires finding the next greater or smaller element in a monotonic increasing or decreasing sequence, respectively. Let's explore each algorithm in more detail:

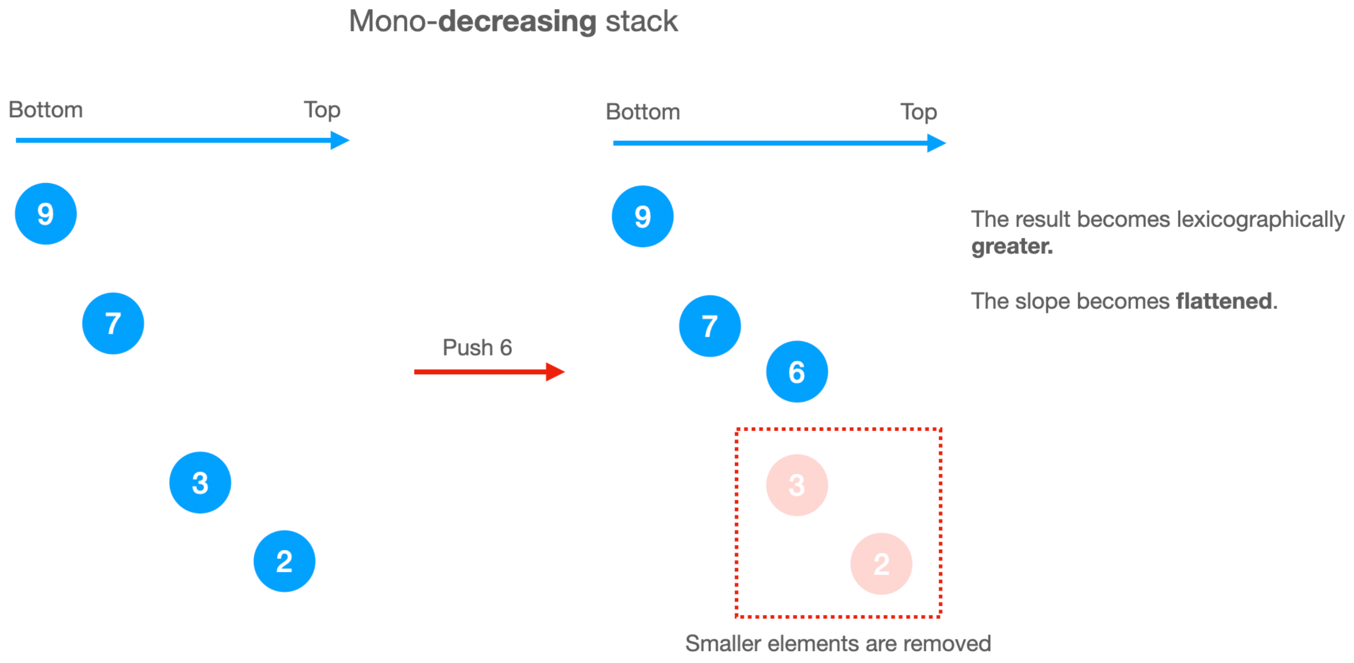
1. Monotonic Increasing Stack Algorithm:
    - This algorithm is used when we need to find the next greater element in a monotonic increasing sequence.
    - The idea is similar to the Monotonic Stack algorithm, but with a different monotonicity condition.
    - Initialize an empty stack and iterate through the array (or elements) from left to right.
    - For each element `current` encountered:
        - While the stack is not empty and the element at the top of the stack is less than `current`, pop the stack and process the popped element as the next greater element for the corresponding index.
        - Push the `current` element or its index onto the stack.
    - After the iteration finishes, if there are any remaining elements in the stack, process them accordingly (e.g., there is no next greater element for those indices).
2. Monotonic Decreasing Stack Algorithm:
    - This algorithm is used when we need to find the next smaller element in a monotonic decreasing sequence.
    - The approach is similar to the Monotonic Increasing Stack algorithm, but with a different monotonicity condition.
    - Initialize an empty stack and iterate through the array (or elements) from left to right.
    - For each element `current` encountered:
        - While the stack is not empty and the element at the top of the stack is greater than `current`, pop the stack and process the popped element as the next smaller element for the corresponding index.
        - Push the `current` element or its index onto the stack.
    - After the iteration finishes, if there are any remaining elements in the stack, process them accordingly (e.g., there is no next smaller element for those indices).

Both the Monotonic Increasing Stack and Monotonic Decreasing Stack algorithms follow a similar structure to the original Monotonic Stack algorithm, but with different monotonicity conditions. The choice between these variations depends on the problem requirements and whether the sequence is expected to be increasing or decreasing.

These variations of the Monotonic Stack algorithm are particularly useful when solving problems like finding the next greater element in an increasing sequence or the next smaller element in a decreasing sequence efficiently. By leveraging the monotonicity property and the stack data structure, these algorithms provide an optimized approach to such problems.

Increasing or decreasing?

If we need to pop smaller elements from the stack before pushing a new element, the stack is decreasing from bottom to top. Otherwise, it's increasing from bottom to top


Mono-**decreasing** stack
Bottom
Top
9
7
3
2
Push 6
Bottom
Top
9
7
6
3
2
Smaller elements are removed
The result becomes lexicographically **greater.**
The slope becomes **flattened**.