Final Report: Monotonic Stack

Team: Pop Stars
(Rita He, **Jingjing Pan**, Ziye Huang, Melody Li, Jasmine Guo)

1 Introduction

The monotonic stack is a fundamental data structure widely used in algorithm design to solve problems involving comparisons across elements in sequences. In this project, we analyze its concept, utility, and implementation, with specific focus on solving relationships like "next greater" and "previous smaller" elements.

(1) **Our Question:** How does the monotonic stack optimize problems involving ordered relationships in arrays?

(2) **Personal Motivation:** As a team of five computer science students, we encounter these problems frequently in interviews and coursework. Our experiences with various competitive programming challenges and LeetCode exercises inspired us to explore efficient, real-world solutions.

(3) **Preview of Examples:** Examples include Nearest Smaller Height, Ocean View Buildings, Daily Temperatures, Browser History, and Network Traffic Analysis.

(4) **Report Details:**

- **Algorithm/Data Structure:** A monotonic stack maintains elements in either an increasing or decreasing order.
- **Problem Solved:** It quickly identifies the next element that is greater or smaller in a list, thereby reducing redundant comparisons.
- **Brief History:** Although stacks have been around for decades, the specific use of monotonic stacks became popular in the 2010s through competitive programming and technical interviews.
- **Overview of the Report:**
  Section 1: Monotonic Stack Introduction
  Section 2: In-depth analysis, including complexity and applications
  Section 3: Final reflections, what we learned, and future considerations

2 Analysis

2.1 Analysis of Monotonic Stack

(1) **Maintaining Order**

- **Increasing Order (Monotonically Increasing Stack):** Every new element added to the stack is greater than or equal to the element below it. If a new element is smaller, you pop elements until you can insert it without breaking the order.
- **Decreasing Order (Monotonically Decreasing Stack):** Every new element is smaller than or equal to the element immediately below it. If a new element is larger, you pop elements until the order can be maintained.

(2) **How It Works**

(a) **Push Operation:** When you add a new element, compare it with the element currently on the top of the stack.

- For an increasing monotonic stack: If the new element is less than the top, pop the top until finding an element that is less than or equal to the new element, then push the new element.
- For a decreasing monotonic stack: If the new element is greater than the top, pop the top until the condition is met, then push the new element.

(b) **Pop Operation:** Standard pop operations still apply, but they might occur during the push process if the monotonic order is disrupted.

(3) **Why Use It**

(a) **Efficiency in Solving Problems:** Monotonic stacks are particularly useful in solving problems where you need to quickly determine the next greater or smaller element, such as:

- Finding the next greater element for each element in an array.
- Calculating the largest rectangle in a histogram.

(b) **Reducing Redundant Comparisons:** By maintaining a strict order, many potential comparisons are avoided. Each element is processed only once for its insertion and possible removal, leading to efficient solutions often with linear time complexity.

(4) **Practical Example**

- Consider an array of numbers where you want to find the next larger number for each element.
- As you traverse the array, the monotonic stack keeps track of elements in a decreasing order.
- When a number larger than the element at the top of the stack is found, that element's next greater element is identified, and it is popped from the stack. This method ensures every number is processed efficiently without having to re-scan the array multiple times.

(5) **Complexity Analysis**

- **Time Complexity:** $O(n)$ – each element is pushed and popped only once.
- **Space Complexity:** $O(n)$ – in the worst case, all elements are stored in the stack.

2.2 Applications

(1) **Browser History**

- Monotonicity of access order allows clean back/forward operations.
- Two stacks (back and forward) maintain the correct page history efficiently.

(2) **Network Traffic Analysis**

- Detect sudden changes in network traffic.
- The stack maintains trends and helps mark deviation points as alerts.

2.3 Implementation

(1) **Language Used:** Python

(2) **Tools/Libraries:** The implementation relies primarily on Python's built-in data structures, with the `list` type used to simulate the stack.

(3) **Challenges:**

- **Handling Edge Cases:** Managing situations like an empty array or repeated elements required careful consideration. Preventing stack underflow (i.e., attempting to pop from an empty list) was critical.
- **Maintaining Monotonicity:** The core challenge lay in correctly enforcing the monotonic condition. Whether the use-case required an increasing or decreasing monotonic stack, implementing the pop and push operations while preserving the required order was non-trivial, especially when handling duplicate or borderline values.

(4) **Key Points of the Algorithm/Data Structure Implementation:**

(a) **Iteration Through the Array:** The algorithm traverses each element of the input list sequentially, using the stack to track elements that have not yet found a subsequent element that violates the desired order.

(b) **Maintaining the Stack:**

- **Push Operation:** When the current element fits the monotonic condition relative to the stack's top element, it is pushed onto the stack.
- **Pop Operation:** If the current element violates the monotonic order (for example, being greater in a decreasing stack), elements are popped from the stack until the condition is restored.

(5) **Pseudocode:**

```python
def monotonic_stack(nums, increasing=True):
    stack = []                      # stack to store indices or values
    result = [None] * len(nums)     # initialize result list

    for i in range(len(nums)):
        while stack is not empty and (
            (increasing and nums[stack[-1]] > nums[i]) or
            (not increasing and nums[stack[-1]] < nums[i])
        ):
            stack.pop()

        if stack is empty:
            result[i] = default_value  # e.g., -1 or None
        else:
            result[i] = stack[-1]      # store index or value as needed

        stack.append(i)

    return result
```

The monotonic stack implementation was successful in providing an efficient ($O(n)$) method to solve problems that require immediate access to previous or next greater/smaller elements. The choice of Python allowed rapid development and testing, while the use of clear, modular code ensured maintainability.

3 Conclusion

Our exploration shows that the monotonic stack is a powerful tool for comparison-based problems, transforming potential $O(n^2)$ solutions into efficient $O(n)$ approaches.

(1) Strengths:
The monotonic stack efficiently processes each element once by maintaining a controlled ordering, making it highly effective for problems like finding the next greater or smaller element.

(2) Weaknesses:

- **Not Intuitive to Beginners:** Its abstract nature and the logic behind adding or removing elements can be confusing for beginners.
- **Limited to Specific Scenarios:** It excels when data is ordered or when dealing with a single comparison condition. For unsorted data or multiple conditions, significant modifications may be required.
- **Edge Cases:** Handling duplicates or invalid comparisons demands extra care and possibly additional auxiliary data.

(3) Future Directions:

- **2D Extensions:** Adapting the concept to matrices could unlock new solutions in fields like computational geometry or image processing.
- **Hybrid Approaches:** Combining monotonic stacks with dynamic programming could more effectively handle problems with overlapping subproblems.

- **Generalization:** Developing customizable frameworks with user-defined comparators could broaden its application across diverse problem types.

(4) What We Learned:
The process reinforced the importance of breaking problems into manageable components and choosing the right data structure. These insights are valuable in academic settings, technical interviews, and real-world applications such as UI navigation and environmental modeling.