Topic Introduction

Welcome back to another PrepLetter! Today, we're focusing on a powerful technique that pops up across a surprising number of coding interview questions: **the monotonic stack**.

A **monotonic stack** is a stack that is either always increasing or always decreasing. You use it to efficiently solve problems where you need to find the "next greater" or "next smaller" element for each item in a sequence. The magic? It helps you avoid nested loops and brings down your time complexity to O(n) in many cases!

When is it useful?

- When you need to find the next (or previous) element that is greater or smaller than the current one.
- Common in array problems, especially those involving comparisons to neighbors, or "wait until a warmer day" types.

How does it work?

- As you scan through an array, you use the stack to keep track of candidates for the "next" element.
- When you find an element that is "greater", you pop items off the stack until the stack is empty or you find what you need.

Here's a quick, simple example:
Suppose you have `[2, 1, 2, 4, 3]` and want to find the next greater element for each number.

- For `2`, the next greater is `4`.
- For `1`, next is `2`.
- For `2`, next is `4`.
- For `4`, none, so `-1`.
- For `3`, none, so `-1`.

This is precisely the sort of problem the monotonic stack excels at. Let's see it in action with three classic problems, each building on the last.

Problem 1: Next Greater Element I

LeetCode 496 - Next Greater Element I

Problem Statement (in our words):
You are given two arrays, `nums1` and `nums2`. For each number in `nums1`, find its next greater number in `nums2`. The next greater number is the first number to the right of it in `nums2` that is larger. If it doesn't exist, use `-1`.

Example:
`nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`
For `4`: There is no greater number to its right in `nums2`, so `-1`.
For `1`: Next greater in `nums2` is `3`.
For `2`: No greater element to its right, so `-1`.

Output: `[ -1, 3, -1 ]`

Let's do another example. Try this one on paper:

- `nums1 = [2,4]`

- `nums2 = [1,2,3,4]`

What should the output be?

Take a moment to try solving this on your own before reading the solution.

Solution Approach: Monotonic Stack

We want to, for every element in `nums1`, quickly know the next greater element in `nums2`.
A brute-force solution would scan right for each element, but that's slow (O(n*m)).
Instead, we'll preprocess `nums2` with a stack to find the next greater for every number!

Step-by-step:

- Loop through `nums2` from left to right.
- Keep a stack of numbers for which we haven't yet found a next greater.
- For each number, if it's greater than the number on top of the stack, pop from stack and record that this is the "next greater" for that popped number.
- After going through `nums2`, we have a map from each number to its next greater element.
- For each number in `nums1`, look up its answer in the map.

Let’s see the code:

```
def nextGreaterElement(nums1, nums2):
    stack = []
    next_greater = {}  # Map from number to its next greater

    for num in nums2:
        while stack and num > stack[-1]:
            prev = stack.pop()
            next_greater[prev] = num
        stack.append(num)

    # For numbers with no next greater, they just aren't in the map
    return [next_greater.get(num, -1) for num in nums1]
```

Time Complexity: O(n + m), where n is length of nums2 and m is length of nums1.
Space Complexity: O(n) for the map and stack.

How does this work, step by step?

Let’s dry-run for `nums2 = [1,3,4,2]`:

- `1` goes onto stack.
- `3` is greater than `1`, so pop `1`, `next_greater[1]=3`, then push `3`.
- `4` is greater than `3`, so pop `3`, `next_greater[3]=4`, then push `4`.
- `2` is not greater than `4`, so push `2`.

After the loop, next_greater = `{1:3, 3:4}`.

Try running the code (or tracing it) with `nums2 = [1,2,3,4]` as your test case!

Alternate approach:
You could brute-force for each `nums1` element by scanning `nums2` to the right, but that's O(n*m). Try implementing that version for practice!

Now, let's level up. What if you want the next greater element for *every* number in an array, not just those from a subset? And what if you want to know *how far ahead* that greater element is? Enter...

Problem 2: Daily Temperatures

LeetCode 739 - Daily Temperatures

Problem Statement (in our words):
Given a list of daily temperatures, for each day, tell how many days you would have to wait until a warmer temperature. If there is no future day with a warmer temperature, put 0 for that day.

Example:
`temperatures = [73,74,75,71,69,72,76,73]`
Output: `[1, 1, 4, 2, 1, 1, 0, 0]`

Let's walk through index 0:

- Temp is 73. The next warmer day is 74 at index 1. Wait 1 day.
- At index 2 (75), the next warmer day is 76 at index 6. Wait 4 days.
- At the end (76, 73), no warmer day ahead, so 0.

Try this one for practice:
`temperatures = [70, 71, 69, 72, 68, 73]`
What should the output be?

Take a moment to try solving this on your own before reading the solution.

Solution Approach: Monotonic Stack

The challenge is to avoid, for each day, doing an O(n) search ahead for a warmer day. The monotonic stack helps us do it in O(n)!

Key difference from previous problem:
Now, we care about *how many days ahead* the next greater is, not just what the value is.

Step-by-step:

- Loop through the temperature list, keeping a stack of indices (not values).
- For each day, while the current temperature is higher than the temperature at the index on top of the stack:
  - Pop the index from the stack.
  - The difference between current index and the popped index is the answer for that day.

- Push current index onto the stack.

Why indices? Because we want to know "how many days" between them.

Here's the code:

```
def dailyTemperatures(temperatures):
    n = len(temperatures)
    answer = [0] * n
    stack = []  # Stores indices

    for i, temp in enumerate(temperatures):
        while stack and temp > temperatures[stack[-1]]:
            prev_day = stack.pop()
            answer[prev_day] = i - prev_day
        stack.append(i)
    return answer
```

Time Complexity: O(n), each index is pushed and popped at most once.
Space Complexity: O(n) for the stack and answer array.

Code walkthrough (using the example above):

Let's walk through `temperatures = [73,74,75,71,69,72,76,73]`:

- At index 0 (73), stack is empty, push 0.
- At index 1 (74), 74 > 73, pop 0, answer[0]=1-0=1.
- At index 2 (75), 75 > 74, pop 1, answer[1]=2-1=1.
- At index 6 (76), 76 > 72, pop 5, answer[5]=6-5=1, and so on.

Try tracing the code with your test case: `[70, 71, 69, 72, 68, 73]`.

Alternate approach:
A brute-force O(n^2) solution would scan ahead for each day, but that's not efficient. Can you implement that for practice?

Connection to previous problem:
Both problems use a monotonic stack to find the "next greater" element, but here we store indices to calculate distances.

Ready for a twist? What if the array is *circular*? Let's take on...

Problem 3: Next Greater Element II

LeetCode 503 - Next Greater Element II

Problem Statement (in our words):
Given a *circular* array, for each element, find the next greater element. The next greater element is the first element *after* it (wrapping around if necessary) that is larger. If it doesn't exist, put `-1`.

Example:

`nums = [1,2,1]`

- For `1` (index 0): next greater is `2`.
- For `2` (index 1): no greater ahead, wrap around, `1` is not greater, so `-1`.
- For `1` (index 2): next greater is `2` (wrapping to index 1).

Output: `[2, -1, 2]`

Try this one on your own:

- `nums = [3, 8, 4, 1, 2]`

Take a moment to try solving this on your own before reading the solution.

Solution Approach: Monotonic Stack

This problem is similar to the first, but with an important twist: the array is circular.

Key difference: We need to check elements *after* the end by "wrapping around".

How to simulate circularity?

Iterate through the array *twice* (i.e., from 0 to 2n - 1), always using modulo to access elements.

Step-by-step:

- Initialize answer array with -1.
- Loop through `2*n` indices:
  - For each index `i`, actual array index is `i % n`.
  - While stack is not empty and current element is greater than nums[stack[-1]]:
    - Pop from stack, set answer for that index.
  - If `i < n`, push `i` onto stack (we only want to push real indices, not the "second pass").

Let's see the code:

```
def nextGreaterElements(nums):
    n = len(nums)
    answer = [-1] * n
    stack = []  # Stores indices

    for i in range(2 * n):
        curr = nums[i % n]
        while stack and curr > nums[stack[-1]]:
            prev = stack.pop()
            answer[prev] = curr
        if i < n:
            stack.append(i)
    return answer
```

Time Complexity: O(n), each index is pushed and popped at most once.

Space Complexity: O(n) for the stack and answer array.

Code walkthrough:

• For `nums = [1,2,1]`, as we go through indices 0, 1, 2, 3, 4, 5 (modulo 3), we fill in the answer array just like in problem 1, but now wrapping around.

Try a dry run on your earlier example: `nums = [3, 8, 4, 1, 2]`.

Alternate approach:
You could duplicate the array and use the non-circular stack method, but this is more space and less elegant. Give it a try for practice!

Summary and Next Steps

You've just worked through three classic "next greater element" problems, each building on the idea of the **monotonic stack**:

- **Next Greater Element I:** Find next greater for specific elements.
- **Daily Temperatures:** Find how many steps until next greater, for all elements.
- **Next Greater Element II:** Handle circular arrays for next greater.

Key patterns and takeaways:

- Use a stack to keep track of indices or values waiting for their "next greater".
- Pop from the stack only when you find a greater element.
- For distances, store indices; for values, you can store actual values.
- For circular arrays, loop through twice and use modulo.

Common mistakes:

- Forgetting to use indices (when you need distances).
- Not handling circularity (missing the "wrap-around").
- Not initializing output arrays correctly (with 0 or -1 as needed).
- Pushing indices during the "second pass" in a circular array (be careful to only push real indices).

Action List:

- Solve all three problems yourself, from scratch, without peeking!
- Explore other problems that use monotonic stacks, like "Trapping Rain Water" or "Largest Rectangle in Histogram".
- Read other people's solutions to see different perspectives and edge case tricks.
- Try implementing the brute-force versions to appreciate the efficiency of stacks.
- If you get stuck, that's normal! Practice, revisit, and keep learning. Each time you work through these patterns, they'll make more sense.

That's it for today's PrepLetter. You've added a crucial interview tool to your toolbox. Keep at it, and remember: every step, even the tough ones, brings you closer to mastery. See you tomorrow!