Topic Introduction

Today, let's dive into a classic coding interview pattern: the **monotonic stack**. This is a powerful technique for efficiently solving "next greater element" problems and their variations.

A **monotonic stack** is a stack data structure that always maintains its elements in either increasing or decreasing order (monotonic means "one direction"). In these problems, we often care about finding, for each element in an array, the *next* element to the right (or left) that is larger or smaller.

When you use a monotonic stack:

- You process elements in order (usually left to right).
- You maintain a stack of indices or values that have not yet found their "answer."
- As you encounter a new element, you check if it resolves a pending query for elements in the stack.

Why is this useful in interviews?
These problems come up a lot, especially with arrays and intervals. A naive solution is often O(n^2), but a monotonic stack can reduce this to O(n) by ensuring each element is pushed and popped at most once. Mastering this pattern gives you an edge with all kinds of "next greater/smaller" queries.

Simple Example (Not one of today's problems):
Suppose you have the array `[2, 1, 2, 4, 3]` and you want to find, for each element, the next greater element to its right.

- For `2`, the next greater is `4`.
- For `1`, it's `2`.
- For `2`, it's `4`.
- For `4`, none (`-1`).
- For `3`, none (`-1`).

Using a stack, you can solve this in O(n) time.

Today's problems all use this technique:

- **Daily Temperatures** (LeetCode 739): For each day, how many days until a warmer temperature?
- **Next Greater Element I** (LeetCode 496): For each number in `nums1`, find its next greater in `nums2`.
- **Next Greater Element II** (LeetCode 503): Like the above, but the array is circular — wrap around at the end.

Why are these grouped together?
All three ask: "For this value, what's the *next* bigger value (and where)?" The monotonic stack is the tool that ties them together — you'll see the same pattern, with small twists for different rules (like wrapping around).

Let's explore each one!

Problem 1: Daily Temperatures

Problem statement (rephrased):
Daily Temperatures - LeetCode 739

You're given a list of daily temperatures. For each day, tell how many days you'd have to wait to get a warmer temperature. If there is no such future day, put `0` for that day.

Example:

Input: `[73, 74, 75, 71, 69, 72, 76, 73]`

Output: `[1, 1, 4, 2, 1, 1, 0, 0]`

- Day 0 (`73`): wait 1 day (for `74`)
- Day 1 (`74`): wait 1 day (for `75`)
- Day 2 (`75`): wait 4 days (for `76`)
- etc.

Thought process:

If you try to solve this by, for each day, scanning every future day, that's O(n^2) time. Can we do better?

Helpful tip: Try drawing the temperature array and, for each element, imagine scanning to the right to find the first warmer day.

Extra test case (try manually!):

Input: `[60, 62, 61, 65]`

Expected Output: `[1, 2, 1, 0]`

Brute-force approach:

For each day, look rightward until you find a warmer day.

- Time complexity: O(n^2)
- Not efficient for large inputs.

Optimal approach — monotonic stack:

Let's use a stack to track indices of days whose warmer day we haven't found yet.

- As we process each day, pop days from the stack if today is warmer — for each popped day, the answer is "today minus that day."
- Push today's index to the stack.

Step-by-step:

- Initialize an answer array with zeros.
- Initialize an empty stack (will store indices).
- For each day (index, temperature):
    - While stack not empty and current temperature > temperature at stack's top index:
        - Pop the top index.
        - The answer for that popped day is (current index - popped index).
    - Push current index to stack.
- At the end, days left in the stack had no warmer future days.

Python solution (with comments):

```python
def dailyTemperatures(temperatures):
    n = len(temperatures)
    answer = [0] * n  # Pre-fill with 0s
    stack = []  # Will store indices of unresolved days

    for current_day, current_temp in enumerate(temperatures):
        # Check if current day resolves any previous "colder" days
        while stack and temperatures[stack[-1]] < current_temp:
            prev_day = stack.pop()
            answer[prev_day] = current_day - prev_day  # Days to wait
        stack.append(current_day)  # Add this day to stack

    return answer
```

Time complexity: O(n)

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

Code breakdown

- `answer = [0] * n`: Prepares the output, defaulting to 0.
- `stack`: Holds indices of days waiting for a warmer day.
- For each day:
    - If the current temperature is warmer than the day at the top of the stack, it resolves that day's wait.
    - We keep popping until the stack is empty or the top is warmer or equal.
    - Push the current day so it can be resolved by a future day.

Trace on `[73, 74, 75, 71, 69, 72, 76, 73]`

Let's walk through:

- Day 0 (`73`): stack = [0]
- Day 1 (`74`): `74 > 73`, so pop 0, answer[0]=1; stack = []; push 1
- Day 2 (`75`): `75 > 74`, so pop 1, answer[1]=1; stack = []; push 2
- Day 3 (`71`): stack = [2,3]
- Day 4 (`69`): stack = [2,3,4]
- Day 5 (`72`): `72 > 69`, pop 4, answer[4]=1; `72 > 71`, pop 3, answer[3]=2; stack = [2,5]
- Day 6 (`76`): pop 5, answer[5]=1; pop 2, answer[2]=4; stack=[]; push 6
- Day 7 (`73`): stack=[6,7]

Try this test case yourself:

Input: `[65, 64, 63, 70, 62, 75]`

Expected output: ?

Take a moment to solve this on your own before jumping into the solution.

Problem 2: Next Greater Element I

Problem statement (rephrased):

Next Greater Element I - LeetCode 496

You're given two arrays, `nums1` and `nums2`, and each value in `nums1` also exists in `nums2`. For each value in `nums1`, find its "next greater" value in `nums2` (the first number to its right in `nums2` that's bigger). If there isn't one, return `-1` for that number.

Example:

`nums1 = [4,1,2]`, `nums2 = [1,3,4,2]`

For `4` in `nums1`, in `nums2` it appears at index 2, and no greater value to its right, so answer is `-1`.

For `1`, in `nums2` at index 0, next greater is `3`.

For `2`, in `nums2` at index 3, no greater to the right, so `-1`.

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

Extra test case:

`nums1 = [2,4]`, `nums2 = [1,2,3,4]`

Expected: `[3,-1]`

Brute-force approach:

For each value in `nums1`, find its index in `nums2`, then scan right to find a greater value.

- Time: O(m*n) (m = len(nums1), n = len(nums2))

Optimal approach — monotonic stack:

Let's precompute, for all numbers in `nums2`, their next greater value using a stack.

- Then, for each value in `nums1`, just look up the answer.

Step-by-step:

- Use a stack to process `nums2` from left to right.
- For each number, if it's greater than the stack's top, pop and record that the popped value's next greater is the current number.
- Store results in a hashmap: `next_greater[num] = x`.
- For each number in `nums1`, look up its next greater in the hashmap, or return `-1` if missing.

Pseudocode:

```
Create empty stack, empty hashmap next_greater
For each num in nums2:
    While stack not empty and num > stack[-1]:
        prev = stack.pop()
        next_greater[prev] = num
    stack.push(num)
For each num in nums1:
    result.append(next_greater.get(num, -1))
Return result
```

Trace on `nums2 = [1,3,4,2]`:

- `1`: stack empty, push 1
- `3`: 3 > 1, so next_greater[1] = 3; push 3
- `4`: 4 > 3, next_greater[3] = 4; push 4
- `2`: 2 < 4, push 2

Final map: `{1: 3, 3: 4}`

For `nums1 = [4,1,2]`

- 4: not in map => -1
- 1: 3
- 2: not in map => -1

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

Time complexity: O(n)
Space complexity: O(n) (n = length of nums2)

Try this test case:
`nums1 = [1,2,3]`, `nums2 = [3,2,1,4]`
What's the output?

Problem 3: Next Greater Element II

Problem statement (rephrased):
Next Greater Element II - LeetCode 503
Given a circular array `nums`, for each element, find the next greater element as if the array wraps around (after the end, continue from the start). If there is no next greater, put `-1`.

Example:
Input: `[1,2,1]`

- For index 0 (`1`): next greater is `2` (index 1)
- For index 1 (`2`): wrap around, no greater, so `-1`
- For index 2 (`1`): wrap to index 0, next greater is `2` (index 1)

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

Extra test case:
Input: `[3,8,4,1,2]`
Expected: `[8,-1,8,2,3]`

What's different?
Now, after reaching the end, you keep searching from the start. Each element can look forward up to 2n-1 steps.

Brute-force approach:
For each index, scan forward (wrapping around) until you find a greater element or return -1.

- Time: O(n^2)

Optimal approach — monotonic stack:
Process the array "twice" to simulate wrapping around.

Pseudocode:

```
Let n = length of nums
Initialize answer array of -1s, empty stack
```

```
For i from 0 to 2*n - 1:
    num = nums[i % n]
    While stack not empty and nums[stack[-1]] < num:
        prev = stack.pop()
        answer[prev] = num
    If i < n:
        stack.push(i)
Return answer
```

Explanation:

- Loop through the array twice (indexes 0..2n-1), using modulo to wrap around.
- Only push original indices (not their duplicates) to stack.
- For each value, if it's greater than the value at the top index of the stack, pop and set its answer.

Time complexity: O(n)

Space complexity: O(n)

Try this test case:

Input: `[5, 4, 3, 2, 1]`

Expected output: `[-1, 5, 5, 5, 5]`

Nudge:

Notice how you can use the same stack pattern, but you need to handle wrap-around by iterating twice the length. Can you think of another way to represent the circularity?

Summary and Next Steps

Today you tackled three problems united by the **monotonic stack** — a technique that unlocks O(n) solutions to "next greater" (or "next smaller") problems.

Key patterns to remember:

- Use a stack to track indices or values waiting for a "future answer."
- As you scan, resolve as many as you can with each new value.
- For circular problems, simulate wrap-around by looping twice.

Common interview traps:

- Forgetting to clear the stack or missing wrap-around logic.
- Accidentally pushing duplicate indices or using the wrong reference (value vs. index).
- Not pre-filling the answer array with default values.

Action list:

- Solve all 3 problems yourself, even if you've seen the code!
- Try implementing the second and third problems with variations (e.g., next smaller instead of greater).
- Explore other monotonic stack problems, like "Largest Rectangle in Histogram."
- Compare your solution to others — pay attention to handling of edge cases and code clarity.
- If you get stuck, sketch out the stack's state at each step; visualizing helps!

Keep practicing — every problem you solve with a monotonic stack adds another tool to your problem-solving arsenal!