Monotonic

Monotonic: (of a function or quantity) varying in such a way that it either never decreases or never increases.

A monotonic stack or queue is one whose elements are always sorted (either by ascending or descending order). Monotonic stacks and queues maintain their sorted property by removing elements that would violate the property before adding new elements. For example, let's say you had a monotonically increasing stack, currently `stack = [1, 5, 8, 15, 23]`. You want to push `14` onto the stack. To maintain the sorted property, we need to first pop the `15` and `23` before pushing the `14` - after the push operation, we have `stack = [1, 5, 8, 14]`.

Here is some pseudocode for a monotonic increasing stack:

```
Given an integer array nums

stack = []
for num in nums:
    while stack.length > 0 AND stack.top >= num:
        stack.pop()
    // Between the above and below lines, do some logic depending on the
problem
    stack.push(num)
```

Before we push a `num` onto the stack, we first check if the monotonic property would be violated, and pop elements until it won't be.

Despite the nested loop, the time complexity is $O(n)$ where n is the length of the array because collectively can't push and pop more elements that are in the array.

Monotonic stacks and queues are useful in problems that for each element, involves finding the "next" element based on some criteria, for example, the next greater element. They're also good when you have a dynamic window of elements and you want to maintain knowledge of the maximum or minimum elements as the window changes. Sometimes a monotonic stack or queue is only one part of the entire algorithm.

Example 1: 739 - Daily Temperatures

Given an array of integers `temperatures` that represents the daily temperatures, return an array `answer` such that `answer[i]` is the number of days you have to wait after the ith day to get a warmer temperature. If there is no future day that is warmer, have `answer[i] = 0` instead.

The brute force approach would be to iterate over the input, and for each temperature, iterate through the rest of the array until we find a warmer temperature. Let's say we had `temperatures = [34, 33, 32, 31, 30, 50]`. The first 5 days all share the same "answer" day, the 6th day. Can we leverage this fact to improve from an $O(n^2)$ time complexity?

The second element `33` is not warmer than the first element `34`. The third element `32` is not warmer than the second element `33`. This property is transitive, and it implies that the third element is not warmer than the first element (32 <= 33 <= 34). This means that there's no point in worrying about the first element until we have found a warmer temperature than the second element because any temperature that isn't warmer than the second element is also not warmer than the first element.

This logic of handling elements in backward order reminds us of a stack. We can push the temperatures onto a stack, and pop them off once we find a warmer temperature. Let's look at another example, `temperatures = [40, 35, 32, 37, 50]`. Once we get to the 4th element, we have `stack = [40, 35, 32]`. Now, we see that `37 > 32` and `37 > 35`, so we can pop both of them off the stack. This leaves us with `stack = [40, 37]` after pushing the `37`. At the `50`, we can pop both elements off the stack because `50` is a greater than both of them.

Because the stack is monotonically decreasing, we are guaranteed to pop elements only when we find the first warmer temperature.

The problem wants the distance between elements, so we can store the indices instead of the actual temperatures.

Essentially, the stack holds temperatures that we have not yet found a warmer temperature for. Because we are forcing it to be monotonically decreasing, the temperature at the top of the stack will always be the coldest one.

```
class Solution:
    def dailyTemperatures(self, temperatures: List[int]) -> List[int]:
        stack = []
        answer = [0] * len(temperatures)

        for i in range(len(temperatures)):
            while stack and temperatures[stack[-1]] < temperatures[i]:
                j = stack.pop()
                answer[j] = i - j
            stack.append(i)

        return answer
```

Here, monotonically non-increasing would be a more accurate term than monotonically decreasing because we could have elements in the stack with equal temperatures in this problem.

Example 2: 239 - Sliding Window Maximum

Given an integer array `nums` and an integer `k`, there is a sliding window of size `k` that moves from the very left to the very right. For each window, find the maximum element in the window.

For example, given `nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3`, return `[3, 3, 5, 5, 6, 7]`. The first window is [1, 3, -1] and the last window is [3, 6, 7].

It's easy to know what the maximum number if for a given window. You can just record it when you build it. The difficult part is, when the maximum number leaves the window, how do you know what is the second largest? When that number leaves, what's next?

We are concerned about the largest elements. We want to store the elements in a way that when the maximum element is removed, we know the second maximum, and when that element is removed, we know the third maximum, and so forth. This should also be updated in new elements being added. Let's say the window was `[5, 3, 7, 1]`. Then the order of max elements would be `[7, 5, 3, 1]`. What happens if we add a `6`? We no longer care about the `5`, `3`, or `1`. Because the `6` came after them, it won't be removed before them, and since it is larger, there is no chance that the `5`, `3`, or `1` will ever be a maximum.

In general, when we see a number, we no longer care about any numbers in the window smaller than it, because they have no chance of ever being the maximum. Thus, a monotonically decreasing data structure is what we should aim for (we remove elements smaller than new ones -> the newest element is the smallest element). Should we use a stack or queue? The problem describes a sliding window, which adds elements on the right and removes them from the left - this suggests a queue since operations are happening at opposite ends. However, the logic of removing elements smaller than the current number needs to happen from the right since we are monotonically decreasing. Therefore, we will be adding and removing from the right, and removing from the left as the window tightens - this means we should use a **double-ended queue (deque)** for efficient $O(1)$ operations.

Because we need to keep the window of size `k`, we should store indices instead of the actual numbers in our deque. This way, we know when the max element is beyond our window.

To reiterate, if we keep a monotonic decreasing data structure that contains only elements in the current window, then the first element will always be the maximum element of the window. If the window slides past the element, we can just remove it from the data structure, and the second largest element slides over to become the new largest element.

By storing indices, we can easily check when the window slides past the element at the first position, which means we need to remove it.

The tricky part is maintaining the data structure so that it is always in decreasing order and only contains elements in the current window. When we see an element `curr`, **any element less than curr in the current window** is now useless. The window will slide past those elements before `curr`, so any future window that contains them will also contain `curr`, and since `curr` is larger than them, there is no chance they will ever be part of the answer.

Lastly, we only add to the answer once we have processed at least `k` elements. The first element in our monotonic data structure will always be the answer for the current window.

```python
from collections import deque

class Solution:
    def maxSlidingWindow(self, nums: List[int], k: int) -> List[int]:
        ans = []
        queue = deque()
        for i in range(len(nums)):
            # maintain monotonic decreasing.
            # all elements in the deque smaller than the current one
            # have no chance of being the maximum, so get rid of them
            while queue and nums[i] > nums[queue[-1]]:
                queue.pop()

            queue.append(i)

            # queue[0] is the index of the maximum element.
            # if queue[0] + k == i, then it is outside the window
            if queue[0] + k == i:
                queue.popleft()

            # only add to the answer once our window has reached size k
            if i >= k - 1:
                ans.append(nums[queue[0]])

        return ans
```

The time complexity is $O(n)$! The space complexity is $O(k)$, since the deque can't grow beyond that size.

To summarize:

- We use a monotonic decreasing deque, which implies that the first element is the maximum.
- Once the maximum element is too far to stay in the window we remove it from the deque, and the next greatest element moves to position 0.
- To maintain the decreasing order, we remove elements from the deque that are smaller than the elements being added.

Example 3: 1438 - Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit

Given an array of integers `nums` and an integer `limit`, return the size of the longest subarray such that the absolute difference between any two elements of this subarray is less than or equal to the limit.

The "maximum absolute difference between any two elements" is the maximum element minus the minimum element. We want the longest subarray such that `max - min <= limit`. We can use a sliding window for this.

From the previous example we learned how to keep the maximum element in a sliding window. We need to do that again here but also keep the minimum element.

Use two deques - one monotonic increasing and one monotonic decreasing. The monotonic increasing one has the minimum element in the window at the first index. The monotonic decreasing one has the maximum element in the window at the first index. Add as much as you can without breaking the constraint and then when you break it remove from the left side of the window until it's fixed.

```
from collections import deque

class Solution:
    def longestSubarray(self, nums: List[int], limit: int) -> int:
        increasing = deque()
        decreasing = deque()
        left = ans = 0

        for right in range(len(nums)):
            # maintain the monotonic deques
            while increasing and increasing[-1] > nums[right]:
                increasing.pop()
            while decreasing and decreasing[-1] < nums[right]:
                decreasing.pop()

            increasing.append(nums[right])
            decreasing.append(nums[right])

            # maintain window property
            while decreasing[0] - increasing[0] > limit:
                if nums[left] == decreasing[0]:
                    decreasing.popleft()
                if nums[left] == increasing[0]:
                    increasing.popleft()
                left += 1

            ans = max(ans, right - left + 1)

        return ans
```

With efficient queues, this algorithm has a time and space complexity of $O(n)$ as each for loop iteration is amortized $O(1)$ and the deques can grow to size n, where n is the size of `nums`.

SEE PROBLEMS 496, 901