

PROMPT ---->

```
class Solution {
    static List<Integer> firstNegInt(int arr[], int k) {
        // write code here
        List<Integer> result = new ArrayList<>();
        List<Integer> negativeNumIndexList = new ArrayList<>();

        for (int start = 0, end = 0; end < arr.length; end++) {

            if (arr[end] < 0) {
                negativeNumIndexList.add(end);
            }

            if ((end - start + 1) == k) {

                int firstNegativeNumIndex = (negativeNumIndexList.isEmpty()) ?
                    Integer.MIN_VALUE : negativeNumIndexList.get(0);

                if (firstNegativeNumIndex >= start && firstNegativeNumIndex <= end) {
                    result.add(arr[firstNegativeNumIndex]);
                } else {
                    result.add(0);
                }

                start++;
            }

            Iterator iterator = negativeNumIndexList.iterator();
            while (iterator.hasNext()) {
                Integer currentIndex = (int) iterator.next();
                if (currentIndex < start) {
                    iterator.remove();
                } else {
                    break;
                }
            }
        }

        return result;
    }
}

/* Above is the code for the problem 'First negative in every window of size k' on the GeeksForGeeks. It's passing all the testcases and submitted successfully multiple times. */
```

AI RESPONSE ---->

I've reviewed your code for the "First negative integer in every window of size k" problem. Your implementation is correct and follows an efficient approach using a

sliding window technique with a deque-like structure (implemented as an ArrayList) to track negative number indices.

Code Analysis:

Time Complexity: $O(n)$ - Each element is processed exactly once
Space Complexity: $O(k)$ - In the worst case, we might store up to k indices

Key Strengths:

1. Uses sliding window technique efficiently
2. Maintains negative number indices in sorted order
3. Handles edge cases properly (no negative numbers in window)
4. Efficiently removes indices that are no longer in the current window

Alternative Implementation (Using Deque):

Here's an alternative implementation using Java's `Deque` interface which might be slightly more efficient for removal operations:

```
```java
import java.util.*;

class Solution {
 static List<Integer> firstNegInt(int arr[], int k) {
 List<Integer> result = new ArrayList<>();
 Deque<Integer> negativeIndices = new LinkedList<>();

 for (int i = 0; i < arr.length; i++) {
 // Remove indices that are out of the current window
 if (!negativeIndices.isEmpty() && negativeIndices.peekFirst() < i - k + 1) {
 negativeIndices.pollFirst();
 }

 // Add current index if it's negative
 if (arr[i] < 0) {
 negativeIndices.addLast(i);
 }

 // Process window when it reaches size k
 if (i >= k - 1) {
 if (!negativeIndices.isEmpty()) {
 result.add(arr[negativeIndices.peekFirst()]);
 } else {
 result.add(0);
 }
 }
 }

 return result;
 }
}
```

```

```
## Test Cases:
```

```
Your code should handle these scenarios:
```

```
```java
// Test case 1: Basic case
int[] arr1 = {12, -1, -7, 8, -15, 30, 16, 28};
int k1 = 3;
// Expected: [-1, -1, -7, -15, -15, 0]
```

```
// Test case 2: No negative numbers
int[] arr2 = {1, 2, 3, 4, 5};
int k2 = 2;
// Expected: [0, 0, 0]
```

```
// Test case 3: All negative numbers
int[] arr3 = {-1, -2, -3, -4, -5};
int k3 = 3;
// Expected: [-1, -2, -3]
```

```
// Test case 4: Single element window
int[] arr4 = {-1, 2, -3};
int k4 = 1;
// Expected: [-1, 2, -3]
```
```

```
## Minor Improvement Suggestion:
```

```
You can use `!negativeNumIndexList.isEmpty()` instead of checking for `Integer.MIN_VALUE`:
```

```
```java
if (!negativeNumIndexList.isEmpty() && negativeNumIndexList.get(0) >= start) {
 result.add(arr[negativeNumIndexList.get(0)]);
} else {
 result.add(0);
}
```
```

Your current implementation is correct and efficient. The alternative using Deque might offer slightly better performance for large inputs due to more efficient removal operations, but both approaches have the same time complexity.

PROMPT ---->

What are the similar problems in LeetCode? List the relevant ones with Numbers and link

AI RESPONSE ---->

Of course! The problem "First negative integer in every window of size k" is a classic example of the **Sliding Window** technique. Here are the most relevant and similar problems on LeetCode, categorized by their approach.

1. Core Sliding Window Problems (Most Similar)

These problems use the exact same sliding window technique, often requiring you to track a specific element or property within each window.

| # | Title | Difficulty | Key Idea |
|---|-----------------------------|------------|--|
| --- | --- | --- | --- |
| **[239](https://leetcode.com/problems/sliding-window-maximum/)** | Sliding Window Maximum | Hard | **Most Direct Analog.** Instead of the first negative, find the maximum in each window. Typically solved with a **Monotonic Deque**. |
| **[480](https://leetcode.com/problems/sliding-window-median/)** | Sliding Window Median | Hard | Find the median in each window. Requires two heaps (a max-heap and a min-heap) and is significantly more complex. |
| **[643](https://leetcode.com/problems/maximum-average-subarray-i/)** | Maximum Average Subarray I | Easy | Find the window with the maximum average. Focuses on maintaining a running sum. |
| **[1425](https://leetcode.com/problems/constrained-subsequence-sum/)** | Constrained Subsequence Sum | Hard | A variation of sliding window maximum with a constraint on how far back you can look. Uses a Deque. |

2. Fixed-Length Sliding Window (Maintain Window Size `k`)

These problems involve processing every contiguous subarray (window) of a fixed length `k`.

| # | Title | Difficulty | Key Idea |
|---|---|------------|---|
| --- | --- | --- | --- |
| **[1456](https://leetcode.com/problems/maximum-number-of-vowels-in-a-substring-of-given-length/)** | Maximum Number of Vowels in a Substring of Given Length | Medium | Count vowels in each window of size `k` and find the maximum count. |
| **[1100](https://leetcode.com/problems/find-k-length-substrings-with-no-repeated-characters/)** | Find K-Length Substrings With No Repeated Characters | Medium | Check if all characters in each window of size `k` are unique. |
| **[2653](https://leetcode.com/problems/sliding-subarray-beauty/)** | Sliding Subarray Beauty | Medium | Find the x-th smallest element in each window (a variation of sliding window median). |

3. Variable-Length Sliding Window (Two Pointers)

These problems use a similar two-pointer technique to maintain a window, but the window size is not fixed. Instead, it grows and shrinks based on certain conditions (e.g., "longest substring without repeating characters"). The logic for managing the left pointer and the data structure (often a HashMap or HashSet) is very relevant.

| # | Title | Difficulty | Key Idea |
|--|--|------------|--|
| --- | --- | --- | --- |
| **[3](https://leetcode.com/problems/longest-substring-without-repeating-characters/)** | Longest Substring Without Repeating Characters | Medium | The foundational variable-length window problem. Uses a HashSet/Map to track characters in the current window. |
| **[76](https://leetcode.com/problems/minimum-window-substring/)** | Minimum Window Substring | Hard | Find the shortest window in `s` that contains all characters of `t`. A classic hard problem. |
| **[424](https://leetcode.com/problems/longest-repeating-character-replacement/)** | Longest Repeating Character Replacement | Medium | Find the longest window where you can replace `k` characters to make them all the same. |
| **[904](https://leetcode.com/problems/fruit-into-baskets/)** | Fruit Into Baskets | Medium | Find the longest window containing at most 2 distinct types of elements. |
| **[209](https://leetcode.com/problems/minimum-size-subarray-sum/)** | Minimum Size Subarray Sum | Medium | Find the shortest contiguous subarray with a sum >= `target`. |

4. Problems Using a Deque for Window Optimization

Your problem efficiently uses a list to act as a ****deque**** for storing indices. The following problems rely heavily on this exact pattern for optimal performance.

| # | Title | Difficulty | Key Idea |
|---|--------------------------------|------------|---|
| --- | --- | --- | --- |
| **[239](https://leetcode.com/problems/sliding-window-maximum/)** | Sliding Window Maximum | Hard | **The prime example.** Uses a deque to maintain *potential* maximums in the current window. |
| **[1425](https://leetcode.com/problems/constrained-subsequence-sum/)** | Constrained Subsequence Sum | Hard | Uses a deque to efficiently find the maximum value in a previous window to build the current solution. |
| **[1438](https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/)** | Longest Continuous Subarray... | Medium | Uses two deques (one for min, one for max) to track the current window's min and max and check the condition. |

Recommended Order of Practice:

1. ****Start with Fixed-Length:**** [643](<https://leetcode.com/problems/maximum-average-subarray-i/>), [1456](<https://leetcode.com/problems/maximum-number-of-vowels-in-a-substring-of-given-length/>) to get comfortable with the basic window mechanics.
2. ****Move to Variable-Length:**** [3](<https://leetcode.com/problems/longest-substring-without-repeating-characters/>), [209](<https://leetcode.com/problems/minimum-size-subarray-sum/>), [904](<https://leetcode.com/problems/fruit-into-baskets/>) to understand how the window expands and contracts.
3. ****Tackle the Deque Problems:**** [239](<https://leetcode.com/problems/sliding-window-maximum/>) is the most important one to master after solving the "first negative" problem. Then try [1438](<https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/>).
4. ****Challenge Yourself:**** Finally, attempt the hard problems like [76](<https://leetcode.com/problems/minimum-window-substring/>) and [480](<https://leetcode.com/problems/sliding-window-median/>).