# 2653. Sliding Subarray Beauty

Medium | Array | Hash Table | Sliding Window | Leetcode Link

## Problem Description

The problem presents a scenario where we are given an array `nums` consisting of `n` integers, and we are tasked with determining the "beauty" of every contiguous subarray that is of size `k`. Beauty in this context is defined as the value of the `x`th smallest integer in the subarray, but with a twist. If the `x`th smallest number is negative, that is the beauty. If not, or there are fewer than `x` negative integers in the subarray, then the beauty is `0`. The goal is to create an array that contains the beauty of each subarray starting from the beginning of the `nums` array, such that we end up with `n - k + 1` beauties.

It is important to understand that a subarray is considered a continuous part of the original array and cannot be empty.

In simpler terms, this problem is asking us to look at every possible continuous chunk of `k` elements in the array, count negative numbers within it and if there are at least `x` negative numbers, find the `x`th smallest one, otherwise record a beauty of `0`.

## Intuition

To solve this problem, a natural approach is to utilize a data structure that can keep numbers sorted as they are added or removed. This way, we can maintain the order of elements in each `k`-sized subarray in real time without having to sort the subarray on every iteration. This is crucial because sorting every subarray would lead to a high computational complexity, making it inefficient for large arrays.

The solution uses the `SortedList` data structure from the Python `sortedcontainers` module which helps maintain a sorted list efficiently. Here's how this works:

1. We initialize `SortedList` with the first `k` elements of the `nums` array. This automatically keeps these elements sorted.
2. We record the beauty of the initial subarray. The `x`th smallest element can be retrieved directly by index `x - 1` due to zero-based indexing. If this element is negative, it is the beauty; otherwise, we record `0`.
3. We then slide the window of size `k` across the array from start to finish. In each step:
   - We remove the leftmost (oldest) element of the subarray from the sorted list since it is no longer in the current window.
   - We add the new element (the one that comes into the window from the right) to the sorted list, keeping the list sorted.
   - We then compute and record the beauty of the new subarray in an analogous manner to step 2.
4. Once we have slid the window across the entire array, we will have computed the beauty of each subarray, and we return the collection of beauties recorded.

Using this method, we continuously maintain a sorted window of the current subarray, allowing us to efficiently answer queries about its `x`th smallest element and compute the beauty without re-sorting the entire subarray each time.

## Solution Approach

The implementation of the solution uses the `SortedList` data structure that provides the capability to manage a sorted sequence of numbers efficiently. In Python, `SortedList` handles changes to the sequence, such as adding or removing elements, while maintaining the sorted order. This capability is key to the solution because it allows us to efficiently manage the min-heap property that makes retrieval of the `x`th smallest number possible in constant time.

Let's walk through a detailed implementation strategy referencing the code provided:

1. We instantiate a `SortedList` with the first `k` elements of the `nums` array. This prepares our first subarray and maintains its elements in sorted order.

   ```python
   sl = SortedList(nums[:k])
   ```

2. We calculate the beauty of the first subarray immediately. If the `(x - 1)` index element is negative, this value is recorded as the beauty; otherwise, `0` is recorded. This becomes the first element of our answer list `ans`.

   ```python
   ans = [sl[x - 1] if sl[x - 1] < 0 else 0]
   ```

3. Next, we enter a loop that iterates over `nums` from the `k`th index to the end:

   ```python
   for i in range(k, len(nums)):
   ```

   - For each iteration, the leftmost (oldest) element of the previous subarray is removed from `SortedList`:
   ```python
   sl.remove(nums[i - k])
   ```
   - Simultaneously, the next element in the array (`nums[i]`) is added to `SortedList`, again maintaining sorting:
   ```python
   sl.add(nums[i])
   ```
   - We then evaluate the beauty of the current subarray: if the `(x - 1)` index element is negative, this value is recorded as the beauty; otherwise, `0` is recorded:
   ```python
   ans.append(sl[x - 1] if sl[x - 1] < 0 else 0)
   ```

4. Finally, after the loop completes, `ans` contains the beauty of each subarray. This result is then returned.

   ```python
   return ans
   ```

By using the `SortedList`, we avoid the need to sort each subarray individually, as it takes care of maintaining the sorted order upon insertions and deletions. This results in an efficient solution that can handle the sliding window computation in a way that is much faster than a naive solution that would sort every single subarray.

### Example Walkthrough

Let's use a small example to illustrate the solution approach described above. Suppose we are given an array `nums` with elements `[-1, 2, -3, 4, -5]` and we need to find the beauty of every subarray of size `k = 3`, where the beauty is defined as the `x = 2`nd smallest negative integer in the subarray.

1. Initialize a `SortedList` with the first `k` elements of the `nums` array:

   ```python
   sl = SortedList([-1, 2, -3])
   ```

   The sorted list would be `[-3, -1, 2]`.

2. Calculate the beauty of the first subarray:

   ```python
   ans = [-1 if sl[1] < 0 else 0]
   ```

   Since the element at index `1` (remembering zero-based indexing) is `-1`, which is negative, we record `-1` in `ans`.

   Now, `ans = [-1]`.

3. Slide the window and update the `SortedList`:

   For the next subarray `[2, -3, 4]`, remove the element `-1` from `sl` and add `4`:

   ```python
   sl.remove(-1)
   sl.add(4)
   ```

   Now `sl` becomes `[-3, 2, 4]`. The element at index `1` is now `2`, so the beauty of this subarray is `0`.

   Update `ans` to include the new beauty: `ans = [-1, 0]`.

4. Move the window to the next subarray `[4, -5]`:

   Remove the leftmost element `2` from `sl` and add `-5`:

   ```python
   sl.remove(2)
   sl.add(-5)
   ```

   The sorted list now is `[-5, -3, 4]`. The element at index `1` is `-3`, which is negative, so this subarray's beauty is `-3`.

   Update `ans` to include this new beauty: `ans = [-1, 0, -3]`.

After following these steps, we obtain the final result `ans = [-1, 0, -3]`, which contains the beauty of each contiguous subarray of size `k = 3`. This array tells us that the first subarray has a beauty value of `-1`, the second has `0` since there aren't enough negative numbers to consider, and the third subarray has a beauty value of `-3`.

Using the `SortedList`, we efficiently calculated the beauty of all subarrays without having to sort each one individually.

## Python Solution

```python
from sortedcontainers import SortedList
from typing import List  # importing typing module for List type hint


class Solution:
    def get_subarray_beauty(self, nums: List[int], k: int, x: int) -> List[int]:
        # Create a sorted list with the first 'k' elements of 'nums'
        sorted_list = SortedList(nums[:k])

        # Calculate the initial beauty and store it in the result list.
        # If the x-1 th element (0-indexed) of the sorted list is negative, store it; otherwise, store 0.
        result = [sorted_list[x - 1] if sorted_list[x - 1] < 0 else 0]

        # Iterate through the rest of 'nums' starting from index 'k'
        for i in range(k, len(nums)):
            # Remove the element that's no longer in the sliding window
            sorted_list.remove(nums[i - k])
            # Add the new element to the sorted list
            sorted_list.add(nums[i])

            # Calculate the beauty for the new subarray and append it to the result list
            result.append(sorted_list[x - 1] if sorted_list[x - 1] < 0 else 0)

        # Return the final list of beauties
        return result

```

## Java Solution

```java
class Solution {
    // Method to return the beauties of all subarrays with length k.
    public int[] getSubarrayBeauty(int[] nums, int k, int x) {
        int length = nums.length; // Total number of elements in nums

        // Frequency array with a size of 101, allowing for offset to handle negative numbers.
        // Offsetting values by 50 to allow for negative values in nums.
        int[] count = new int[101];
        // Initialize the count array with the first ‘k’ elements in nums.
        for (int i = 0; i < k; ++i) {
            count[nums[i] + 50]++;
        }

        // Result array to store beauty of each subarray of length k.
        int[] beautyValues = new int[length - k + 1];
        // Store the beauty of the first subarray.
        beautyValues[0] = calculateBeauty(count, x);
        // Sliding window approach to calculate beauty for remaining subarrays of length k.
        for (int end = k, start = 1; end < length; ++end) {
            // Include the next element in the window and update its count.
            count[nums[end] + 50]++;
            // Exclude the element that is now outside the window and update its count.
            count[nums[end - k] + 50]--;
            // Calculate beauty for the new subarray and store it.
            beautyValues[start++] = calculateBeauty(count, x);
        }

        return beautyValues;
    }

    // Helper method to calculate beauty using the frequency count array and value x.
    private int calculateBeauty(int[] count, int x) {
        int sum = 0;
        // Iterate over the count array to calculate cumulative sum.
        for (int i = 0; i < 50; ++i) {
            sum += count[i];
            // If the cumulative sum is at least x, return the value representing beauty.
            if (sum >= x) {
                return i - 50; // Offset by 50 to get the actual value.
            }
        }
        // If beauty couldn't be determined within loop range, return 0.
        return 0;
    }
}

```

## C++ Solution

```cpp
#include <vector>

class Solution {
public:
    // Defines a method to calculate the subarray beauties for all subarrays of size k.
    // A subarray's beauty is defined by the x-th smallest element in this subarray.
    // nums: an integer vector
    // k: the size of the subarrays
    // x: the position of the element that defines the beauty
    // Returns a vector with the beauty of each subarray
    vector<int> getSubarrayBeauty(vector<int>& nums, int k, int x) {
        int n = nums.size(); // total number of elements in the input array
        int counts[101] = {}; // an array to keep the count of elements (offset by 50 to handle negative numbers)

        // Initialize the counts for the first subarray
        for (int i = 0; i < k; ++i) {
            ++counts[nums[i] + 50];
        }

        // Create a vector to store the answer
        vector<int> beauties(n - k + 1);

        // Lambda function to find the x-th smallest number in the current subarray
        auto findXthSmallest = [&](int xth) {
            int sum = 0;
            for (int i = 0; i < 50; ++i) {
                sum += counts[i];
                if (sum >= xth) { // if the cumulative count hits xth, we found our xth smallest
                    return i - 50; // return the number considering the offset
                }
            }
            return 0; // placeholder, this case should not happen as sum should always hit xth before i reaches 50
        };

        // Calculate the beauty for the first subarray
        beauties[0] = findXthSmallest(x);

        // Calculate the beauty for remaining subarrays
        for (int i = k, j = 1; i < n; ++i, ++j) {
            ++counts[nums[i] + 50]; // include the new number in the count
            --counts[nums[i - k] + 50]; // exclude the oldest number from the count
            beauties[j] = findXthSmallest(x); // calculate the beauty for the current subarray
        }

        return beauties; // return the computed beauties
    }
};

```

## Typescript Solution

```typescript
function getSubarrayBeauty(nums: number[], windowSize: number, beautyRank: number): number[] {
    const numLength = nums.length;
    // Initialize a count array to keep track of number frequencies within a window
    const count = new Array(101).fill(0);

    // Populate the count for the initial window of size windowSize
    for (let i = 0; i < windowSize; ++i) {
        ++count[nums[i] + 50];
    }

    // Initialize an array to store the beauty of each subarray
    const beautyArray: number[] = new Array(numLength - windowSize + 1);

    // Helper function to calculate the beauty of the current window
    const calculateBeauty = (rank: number): number => {
        let sum = 0;
        for (let i = 0; i < 50; ++i) {
            sum += count[i];
            if (sum >= rank) {
                return i - 50;
            }
        }
        return 0; // Default return if rank is never reached
    };

    // Calculate the beauty for the first window
    beautyArray[0] = calculateBeauty(beautyRank);

    // Slide the window through the array and calculate the beauty of each subarray
    for (let i = windowSize, j = 1; i < numLength; ++i, ++j) {
        count[nums[i] + 50]++;
        count[nums[i - windowSize] + 50]--;
        beautyArray[j] = calculateBeauty(beautyRank);
    }

    // Return the array containing the beauty of each subarray
    return beautyArray;
}

```

## Time and Space Complexity

### Time Complexity

The time complexity of the code is determined by several operations: initializing the `SortedList`, adding elements to it, removing elements from it, and looking up an element at a specific index within it.

1. Initializing `SortedList` with `nums[:k]` takes O(k log(k)) time, as it needs to sort `k` elements.
2. The for loop runs (n - k) times, where `n` is the length of `nums`.
3. Inside the loop, `sl.remove(nums[i - k])` takes O(log(k)) time as `SortedList` maintains the elements in sorted order, so removal is a logarithmic operation.
4. `sl.add(nums[i])` also takes O(log(k)) time for the same reason – to maintain the sorted property of the list.
5. Accessing the `x-1`th smallest item with `sl[x - 1]` is an O(1) operation because `SortedList` allows for fast random access.

Combining these operations, the total time complexity for the loop is (n - k) * (2 * log(k)). So the overall time complexity of the code can be expressed as:

`O(k log(k) + (n - k) * log(k)) = O(n log(k))`

since `k log(k)` is insignificant compared to `(n - k) * log(k)` for large `n`.

### Space Complexity

The space complexity is determined by the extra space used for the `SortedList` and the `ans` list.

1. The `SortedList` at any time contains exactly `k` elements, giving a space complexity of O(k).
2. The `ans` list will contain `n - k + 1` elements (as we insert a new number for every element from `nums[k]` to `nums[n-1]`), so this also gives a space complexity of O(n).

Combining these, the dominant space complexity remains O(n) (as we typically consider the worst-case space usage to determine space complexity).

Therefore, the space complexity of the code is `O(n)`.