Documentation

The sliding window maximum problem requires us to find the maximum value in each subarray or "window" of length (k) as it slides over the array. In this context, a sliding window of size (k) moves one position to the right at a time, and for each new position, we want to know the highest number within that range. The brute-force solution would involve examining each window independently and finding the maximum value, but this approach becomes inefficient for large arrays since it requires repeatedly traversing subarrays. For a more optimal solution, we can use a deque (double-ended queue) to keep track of indices of potential maximum values within each window, maintaining the highest element's index at the front of the deque.

To solve this problem efficiently, the deque will help maintain a list of useful indices in a way that avoids unnecessary computations and keeps track of the maximum for the current window. As we slide the window from left to right, the deque will remove indices of elements that are no longer within the window's range. Additionally, before adding a new element, we compare it to the elements at the back of the deque. If it’s greater, we remove all elements smaller than it from the back since they will not contribute to future maximums. This is because a larger, more recent element will remain in the window longer, making smaller elements irrelevant.

The steps for this algorithm involve iterating through each element of the array and managing the deque's contents according to the sliding window position. Specifically, at each step, we remove indices from the front of the deque if they are outside the bounds of the current window. We also ensure that the elements in the deque are ordered by their values in descending order. This means that the front of the deque will always store the index of the largest element within the window, enabling us to retrieve the maximum for each window in constant time.

Once we’ve processed enough elements to reach the size (k), we can start collecting the maximums from the front of the deque. This ensures that as each window slides to the right, we capture the maximum efficiently without redundant comparisons. Since we only add the first element of the deque (the maximum) to the results once the window is full, our algorithm ensures the first (k - 1) elements are processed without adding incomplete window maximums.

In terms of complexity, this approach is optimal as it achieves (O(n)) time complexity, where (n) is the number of elements in the input list. Each element is processed only once, as it is added and removed from the deque at most one time. The space complexity is (O(k)), which is used to store indices in the deque that could be as large as (k). This efficient approach is particularly useful for handling very large arrays where a brute-force solution would be too slow. By leveraging the properties of the deque, we reduce the time required to find maximums for each sliding window while avoiding the need for repetitive subarray scans.