Solution: Find Maximum in Sliding Window

Let's solve the Find Maximum in Sliding Window problem using the Sliding Window pattern.

We'll cover the following

Statement

Given an integer list `nums`, find the maximum values in all the contiguous subarrays (windows) of size `w`.

> **Note:** If the window size is greater than the list size, we consider the entire list as a single window.

Constraints:

- $1 \leq$ `arr.length` $\leq 10^3$
- $-10^4 \leq$ `arr[i]` $\leq 10^4$
- $1 \leq$ `w`

Solution

So far, you've probably brainstormed some approaches on how to solve this problem. Let's explore some of these approaches and figure out which one to follow while considering time complexity and any implementation constraints.

Naive approach

The naive approach is to slide the window over the input list and find the maximum in each window separately. For this purpose, we create a list `currentWindow`. We iterate over the first window and add its elements to `currentWindow`. Then, we make a copy of `currentWindow` and sort the copy. The element stored at the first index of the copy will be the maximum in the first window, so we add it to the output list. Then, in each subsequent iteration, we append the new element to `currentWindow` and remove the first element, as the first element has dropped out of the window. We repeat this process to find the maximum in each window. Finally, when we have traversed the entire input list, we return the output list containing the maximums of all $n - w + 1$ windows, where w is the size of the window.

Let's look at the cost of this approach. As we only iterate over the input list once to find all the windows, the time complexity of this part will be $O(n)$, where n is the number of elements in the input list. Then, we make

a copy of the list `currentWindow` and sort it. The time complexity of sorting the copy of `currentWindow` will be $O(w \times \log w)$. Hence, the overall time complexity of this approach is $O(n(2w + w \log w)) = O(n(w \log w))$. The space complexity of this approach is $O(w)$ as we maintain a list for the current window.

Optimized approach using sliding window

Our first solution uses the sliding window technique to solve the problem. However, there is much room for improvement. In the following section, we will gradually optimize our naive approach to reduce the time complexity to $O(n)$.

> **Note**: In the following section, we will gradually build the solution. Alternatively, you can skip straight to just the code.

Step-by-step solution construction

Let's now try to build on the naive approach and optimize it. The first thing that comes to mind is to eliminate the step in which we make and sort an extra copy. We can use the core concept of the sliding window approach to our benefit.

Firstly, instead of adding values to `currentWindow`, we use their *indexes*. By doing this, we can easily check which index has fallen out of the current window and remove it.

Secondly, we process the elements of the first window as follows:

- Every time we add a new index to `currentWindow`, we clean it up, i.e., we iterate backward over `currentWindow`, starting from the end, and remove all the indexes whose corresponding values in the input list are smaller than or equal to the new element in the window. Let's understand how this step benefits us. Whenever a new element enters the window, all the previous elements smaller than the current element can no longer be the maximum in the current or any subsequent windows containing the new element. This is because all the subsequent windows holding the indexes of the removed elements will also include the new, *bigger* element. Since this new element is bigger than those elements, keeping those smaller elements in `currentWindow` is unnecessary.
- A key detail to note here is that we perform the clean-up step starting with the *second* element added to the first window. As a result, even in the first window, we will have excluded all elements smaller than the maximum of that window that occurs before it in the input list.
- Let's understand this using an initial frame of five elements, $[2, 4, 5, 3, 1]$. When the index of 4 is added to `currentWindow`, it causes the index of 2 to be deleted. The addition of the index of 5 causes the index of 4 to be deleted. However, the addition of the indexes of 3 and 1 does not trigger any deletions. At the end of this clean-up process, `currentWindow` contains the indexes, $[2, 3, 4]$, corresponding to the values, $[5, 3, 1]$.
- Once we have cleaned up the first window, the remaining indexes in `currentWindow` will be stored in the descending order of their values. Now, let's imagine the other possibility where the first frame actually contains $[2, 4, 5, 1, 3]$. Here, the addition of the index of 1 does not trigger any deletion, but 3 does cause the index of 1 to be deleted. `currentWindow` now holds the indexes $[2, 4]$, corresponding to the values $[5, 3]$, which are sorted in descending order. Since we've examined both possibilities (either the elements in `nums` after element 5 are in descending order, or they aren't), we can be sure that, after the clean-up process, the index of the maximum element in the first window will always be stored at the start of `currentWindow`.

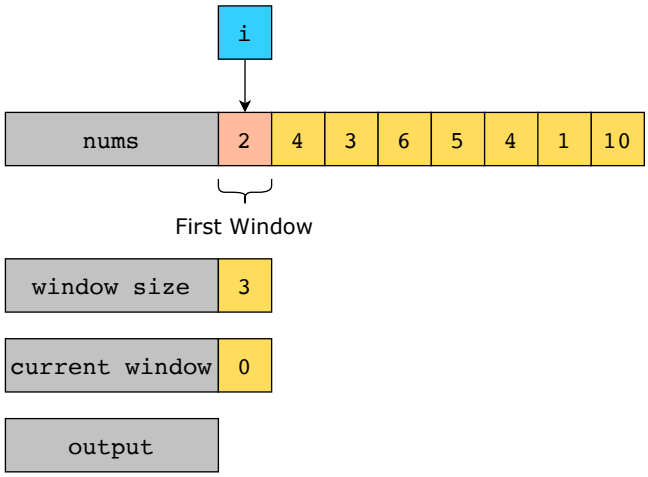
- We append the value corresponding to the index at the start of `currentWindow` to our output list.

Next, we iterate over the remaining input list. For each element, we perform the clean-up step as we did for the elements of the first window.

One difference when processing the second and all subsequent windows, compared to the processing of the first window, is an additional check that is carried out *before* we append the current element in `nums` to `currentWindow`. We check whether the first index in `currentWindow` is still a part of the current window. If not, we remove the first index from `currentWindow`.

Lastly, when the entire input list has been processed, one window at a time, we return the output list containing the maximum of each window.

Let's understand this algorithm with the help of an illustration:



We have defined two lists—**current window** to process the elements of a window and **output** to store the maximums of each window. We traverse the list, starting from the first element of **nums**. Currently, we are on the first element of **nums** whose value is 2. Since **current window** is empty, we simply append the index of 2, which is 0, to the list, and move on to the next element.

1 of 12

Now, let's analyze the algorithm in terms of complexity. We are still moving over the input list in $O(n)$ time and using the sliding window approach to maintain `currentWindow`. Inside the loop, we append the new element to `currentWindow`, which takes $O(1)$ time. The time complexity of the clean-up step is $O(1)$, which is explained in detail in the complexity analysis of the final solution. Lastly, we remove the index that no longer falls in the current window. Since this element is removed from the start of the list, the time complexity of removing this index will be $O(w)$. Therefore, the overall time complexity of this solution will be $O(n \times w)$. The space complexity of this solution is $O(w)$, since we maintain a list to store the indexes of significant elements from the current window.

Java

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

class SlidingWindowMaximum {
    // function to clean up the window
    public static ArrayList<Integer> cleanUp(int i, ArrayList<Integer> currentWindow, int[] nums) {
        // remove all the indexes from currentWindow whose corresponding values are smaller than or equal
```

```
        // remove all the indexes from currentWindow whose corresponding values are smaller than or equal
        while (currentWindow.size() != 0 && nums[i] >= nums[currentWindow.get(currentWindow.size() - 1)])
            System.out.println("\t\tAs nums[" + i + "] = " + nums[i] + " is greater than or equal to nums
            System.out.println("\t\tremove " + currentWindow.get(currentWindow.size() - 1) + " from curre
            currentWindow.remove(currentWindow.size() - 1);
        }

        // returning the altered currentWindow
        return currentWindow;
    }

    // function to find the maximum in all possible windows
    public static int[] findMaxSlidingWindow(int[] nums, int w) {
        // if the input array is empty, return an empty array
        if (nums.length == 0) {
            return new int[0];
        }

        // if window size is greater than the array size, set the window size to the array size
        if (w > nums.length) {
            w = nums.length;
        }
```

First optimization with current window as a list

In the first optimization, we eliminated the additional creation and sorting of the copy of `currentWindow`, reducing the time complexity by a significant factor. Let's see if we can further improve our solution.

At the moment, we are incurring two kinds of costs. The first cost is of iterating over the input array while sliding the window forward. No matter what approach we use, we cannot eliminate this cost. The second cost is of removing the first element from the list `currentWindow`. The list data structure will always impose this cost. Therefore, we need a data structure that removes elements in constant time both from the end and from the start.

The data structure that supports constant-time removals from both ends is a deque. A **deque** is a double-ended queue where the **Push()** and **Pop()** operations work in $O(1)$ time at both ends. Therefore, just by maintaining `currentWindow` as a deque instead of as a list, we can reduce the time complexity of the above solution to $O(n)$. This is the most optimized solution for solving this problem. The space complexity of the solution will still be $O(w)$—the maximum number of elements in the current window stored in the deque.

Java

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

class SlidingWindowMaximum {
    // function to clean up the window
    public static Deque<Integer> cleanUp(int i, Deque<Integer> currentWindow, int[] nums) {
        // remove all the indexes from currentWindow whose corresponding values are smaller than or equal
        while (currentWindow.size() != 0 && nums[i] >= nums[currentWindow.getLast()]) {
            System.out.println("\t\tAs nums[" + i + "] = " + nums[i] + " is greater than or equal to nums
            System.out.println("\t\tremove " + currentWindow.getLast() + " from currentWindow");
            currentWindow.removeLast();
        }

        // returning the altered currentWindow
        return currentWindow;
    }

    // function to find the maximum in all possible windows
    public static int[] findMaxSlidingWindow(int[] nums, int w) {
        // if the input array is empty, return an empty array
        if (nums.length == 0) {
            return new int[0];
        }

```

```
25          // if window size is greater than the array size, set the window size to the array size
26          if (w > nums.length) {
27              w = nums.length;
28          }
```

Second optimization with current window as a deque

Just the code

Here's the complete solution to this problem:

Java

```
1  import java.util.*;
2  import java.util.stream.*;
3
4  class SlidingWindowMaximum {
5      // function to clean up the window
6      public static Deque<Integer> cleanUp(int i, Deque<Integer> currentWindow, int[] nums) {
7          while (currentWindow.size() != 0 && nums[i] >= nums[currentWindow.getLast()]) {
8              currentWindow.removeLast();
9          }
10         return currentWindow;
11     }
12
13     // function to find the maximum in all possible windows
14     public static int[] findMaxSlidingWindow(int[] nums, int w) {
15         if (nums.length == 0) {
16             return new int[0];
17         }
18         if (w > nums.length) {
19             w = nums.length;
20         }
21         int [] output = new int[nums.length - w + 1];
22         Deque<Integer> currentWindow = new ArrayDeque<>();
23         for (int i = 0; i < w; i++) {
24             currentWindow = SlidingWindowMaximum.cleanUp(i, currentWindow, nums);
25             currentWindow.add(i);
26         }
27         output[0] = nums[currentWindow.getFirst()];
28         for (int i = w; i < nums.length; i++) {
```

Find Maximum in Sliding Window

Solution summary

To recap, the solution can be divided into the following parts:

1. First, we validate the inputs. If the input list is empty, we return an empty list, and if the window size is greater than the list length, we set the window to be the same size as the input list.
2. Then, we process the first w elements of the input list. We will use a deque to store the indexes of the candidate maximums of each window.
3. For each element, we perform the clean-up step, removing the indexes of the elements from the deque whose values are smaller than or equal to the value of the element we are about to add to the deque. Then, we append the index of the new element to the back of the deque.
4. After the first w elements have been processed, we append the element whose index is present at the front of the deque to the output list, since it is the maximum in the first window.
5. After finding the maximum in the first window, we iterate over the remaining input list. For each element, we repeat **Step 3** as we did for the first w elements.

6. Additionally, in each iteration, before appending the index of the current element to the deque, we check if the first index in the deque has fallen out of the current window. If so, we remove it from the deque.
7. Finally, we return the list containing the maximum elements of each window.

Time complexity

The input parameters of the function are a list of integers and an integer specifying the size of the window. In the discussion that follows, we will use n to represent the size of the list of integers, and w to represent the size of the window.

To get a clearer understanding of the time complexity of our solution, we need to consider the different ways in which the values in the input list change. The values in the list can be:

1. Strictly increasing
2. Strictly decreasing
3. Constant
4. Mixed, i.e, increasing, decreasing, constant, then decreasing again, then constant, then increasing, then constant and then decreasing

On the surface, we might expect our solution to take $O(n * w)$, but that would be no better than the naive solution. When we look closer at our solution, it comes down to figuring out how many times the clean-up loop actually runs. This is the loop that pops all elements from the deque that are smaller than the new element in the window.

Let's consider the first case when the values in the array are strictly increasing. The first time the window moves forward, the new element is larger than all the other elements in the deque. Therefore, we have to perform the `removeLast()` operation w times. Then, in all the subsequent steps, the `removeLast()` operation is performed only once, since the deque will only contain one element from this point onward. The number of subsequent steps is $n - w$. So, the complexity in this case is $O(w + n - w)$, that is, $O(n)$.

elements in the deque. Therefore, the `removeFirst()` operation is only performed once in every subsequent step to remove the element that does not fall in the window anymore. So, the time complexity in this case is $O(n - w + 1)$, that is, $O(n - w)$.

In the third case, the same behavior is repeated as in the second case, so the time complexity is $O(n - w)$ here, too.

Finally, in the fourth case, the time complexity for increasing values as well as decreasing and constant values will be the same as explained above. The only other situation is when the values increase after staying constant, or right after a sequence of decreasing numbers. In either case, we incur a cost of $O(w)$, since we clean up the deque using the `removeLast()` operation. If there is an increase in the value after every w elements, we pay the $O(w)$ cost to clean up the deque. This can only happen $\frac{n}{w}$ times. The cost of filling up the deque while the elements are constant or decreasing will be $O(w)$. So, the total cost with such data will be $O((w + w)(\frac{n}{w}))$, that is, $O(2w(\frac{n}{w}))$ or $O(n)$.

Hence, the time complexity of the solution, considering the worst of these four cases, is $O(n)$.

Space complexity

The space complexity of this solution is $O(w)$, where w is the window size.