

```
class Solution {
    public ArrayList<Integer> maxOfSubarrays(int[] arr, int k) {
        // code here

        ArrayList<Integer> result = new ArrayList<>();

        Comparator<Integer> sortByDesc = (numA, numB) -> numB - numA;
        Queue<Integer> maxHeap = new PriorityQueue<>(sortByDesc);

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

            maxHeap.add(arr[end]);

            if ((end - start + 1) == k) {
                result.add(maxHeap.peek());
                if (maxHeap.peek() == arr[start]) {
                    maxHeap.remove();
                } else {
                    Iterator itr = maxHeap.iterator();
                    while (itr.hasNext()) {
                        if (arr[start] == (int) itr.next()) {
                            itr.remove();
                            break;
                        }
                    }
                }
                start++;
            }
        }
    }
}
```

```

    }

    return result;
}
}

/* Above is the code for https://www.geeksforgeeks.org/problems/maximum-of-all-
subarrays-of-size-k3101/1 */

/* All test cases are passing except last testcase showing time limit exceeded. Fix the
issues by keeping the code as same as possible and variable name the same. */

```

The issue with your code is that it uses a PriorityQueue (max heap) and manually iterates to remove elements, which leads to $O(n*k)$ time complexity in the worst case. The last test case is timing out because of this inefficiency.

To fix this while keeping the code as similar as possible, we can use a Deque to maintain the indices of elements in the current window in descending order. This approach runs in $O(n)$ time.

However, since you asked to keep the variable names and structure as similar as possible, I'll modify your solution to use a Deque instead of a PriorityQueue while maintaining the same logic flow.

Here's the optimized solution:

```

java

import java.util.*;

class Solution {
    public ArrayList<Integer> maxOfSubarrays(int[] arr, int k) {

```

```

    ArrayList<Integer> result = new ArrayList<>();
    Deque<Integer> deque = new LinkedList<>(); // To store indices of elements in descending order

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

        // Remove elements from the rear that are smaller than the current element
        while (!deque.isEmpty() && arr[deque.peekLast()] < arr[end]) {
            deque.pollLast();
        }

        deque.addLast(end); // Add current index to the rear

        // When the window size is reached, add the maximum to the result
        if (end >= k - 1) {
            result.add(arr[deque.peekFirst()]);
        }
    }

    return result;
}

```

This solution uses a Deque to efficiently track the maximum element in each sliding window. The key ideas are:

1. The front of the Deque always holds the index of the maximum element for the current window.

2. We remove indices that are out of the current window from the front.
3. We remove indices from the rear whose corresponding elements are smaller than the current element, as they cannot be the maximum in any future window containing the current element.

This approach runs in $O(n)$ time and $O(k)$ space, which should pass all test cases including the large ones.

Note: I've kept the variable names `result` and `k` as in your original code, and maintained the same overall structure of iterating with `end` from 0 to $n-1$. The logic inside the loop is different but more efficient.

Write a dry run for the provided solution

Dry Run for the Solution using Deque

Let's take a small example to demonstrate how the solution works. Consider:

- `arr = [1, 3, -1, -3, 5, 3, 6, 7]`
- `k = 3`

We'll go through each step:

Initialization:

- `result = []`
- `deque = []` (empty deque)

Iteration 1: `end = 0` (element = 1)

- Check if deque is empty? Yes -> skip removing from front.

- Check if deque is empty? Yes -> skip removing from rear.
- Add current index to deque: `deque = [0]`
- Check if window size reached? (`end >= 2`) No -> skip adding to result.

Iteration 2: end = 1 (element = 3)

- Check front: `deque.peekFirst() = 0`, which is not equal to `end-k = 1-3 = -2` -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[0] = 1` with current element 3. Since `1 < 3`, remove last: `deque = []`
- Now deque is empty, so add current index: `deque = [1]`
- Check window size? `end=1 < 2` -> skip.

Iteration 3: end = 2 (element = -1)

- Check front: `deque.peekFirst() = 1`, which is not equal to `end-k = 2-3 = -1` -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[1] = 3` with current element -1. Since `3 > -1`, don't remove.
- Add current index to deque: `deque = [1, 2]`
- Window size reached (`end=2 >= 2`): `result.add(arr[deque.peekFirst()]) = arr[1] = 3` -> `result = [3]`

Iteration 4: end = 3 (element = -3)

- Check front: `deque.peekFirst() = 1`, which equals `end-k = 3-3 = 0`? No (`1 ≠ 0`) -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[2] = -1` with current element -3. Since `-1 > -3`, don't remove.
- Add current index to deque: `deque = [1, 2, 3]`

- Window size reached: `result.add(arr[deque.peekFirst()]) = arr[1] = 3` -> `result = [3, 3]`

Iteration 5: end = 4 (element = 5)

- Check front: `deque.peekFirst() = 1`, which equals `end-k = 4-3 = 1`? Yes -> remove front:
`deque = [2, 3]`
- Now remove from rear: Compare `arr[deque.peekLast()] = arr[3] = -3` with current element 5. Since `-3 < 5`, remove last: `deque = [2]`
- Compare `arr[deque.peekLast()] = arr[2] = -1` with 5. Since `-1 < 5`, remove last: `deque = []`
- Add current index: `deque = [4]`
- Window size reached: `result.add(arr[deque.peekFirst()]) = arr[4] = 5` -> `result = [3, 3, 5]`

Iteration 6: end = 5 (element = 3)

- Check front: `deque.peekFirst() = 4`, which equals `end-k = 5-3 = 2`? No -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[4] = 5` with current element 3. Since `5 > 3`, don't remove.
- Add current index: `deque = [4, 5]`
- Window size reached: `result.add(arr[deque.peekFirst()]) = arr[4] = 5` -> `result = [3, 3, 5, 5]`

Iteration 7: end = 6 (element = 6)

- Check front: `deque.peekFirst() = 4`, which equals `end-k = 6-3 = 3`? No -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[5] = 3` with current element 6. Since `3 < 6`, remove last: `deque = [4]`
- Compare `arr[deque.peekLast()] = arr[4] = 5` with 6. Since `5 < 6`, remove last: `deque = []`

- Add current index: `deque = [6]`
- Window size reached: `result.add(arr[deque.peekFirst()]) = arr[6] = 6 -> result = [3, 3, 5, 5, 6]`

Iteration 8: end = 7 (element = 7)

- Check front: `deque.peekFirst() = 6`, which equals `end-k = 7-3 = 4`? No -> skip.
- Remove from rear: Compare `arr[deque.peekLast()] = arr[6] = 6` with current element 7. Since `6 < 7`, remove last: `deque = []`
- Add current index: `deque = [7]`
- Window size reached: `result.add(arr[deque.peekFirst()]) = arr[7] = 7 -> result = [3, 3, 5, 5, 6, 7]`

Final Result: `[3, 3, 5, 5, 6, 7]`

This matches the expected output for the given input. The deque efficiently maintains the indices of potential maximum elements in descending order, ensuring that we always have the maximum of the current window at the front.

But, was the solution provided by me was logically correct?

Let's analyze the **original solution** that used a **PriorityQueue (max heap)**:

java

```
class Solution {
    public ArrayList<Integer> maxOfSubarrays(int[] arr, int k) {
        ArrayList<Integer> result = new ArrayList<>();
        Comparator<Integer> sortByDesc = (numA, numB) -> numB - numA;
        Queue<Integer> maxHeap = new PriorityQueue<>(sortByDesc);
```

```
for (int start = 0, end = 0; end < arr.length; end++) {
    maxHeap.add(arr[end]);

    if ((end - start + 1) == k) {
        result.add(maxHeap.peak());
        if (maxHeap.peak() == arr[start]) {
            maxHeap.remove();
        } else {
            Iterator itr = maxHeap.iterator();
            while (itr.hasNext()) {
                if (arr[start] == (int) itr.next()) {
                    itr.remove();
                    break;
                }
            }
        }
        start++;
    }
}
return result;
}
```

✓ Logical Correctness

The logic is **correct** in terms of functionality. It does:

1. Add each new element to the max-heap.
2. When the window size `k` is reached:

- The maximum (peek of the heap) is added to the result.
- The leftmost element (at `arr[start]`) is removed from the heap.
- If the maximum is the element being removed, we remove it directly (which is efficient).
- Otherwise, we iterate through the heap to find and remove the element (which is inefficient).

✗ Performance Issue

The problem is **performance**:

- The `PriorityQueue.remove(Object)` method is $O(n)$ in the worst case because it must search for the element to remove.
- In the worst case (when the element to remove is not the maximum), we are iterating through the entire heap to remove it (which is $O(k)$ per operation).
- Since we do this for each window ($n - k + 1$ windows), the worst-case time complexity becomes $O(n * k)$, which is too slow for large inputs.

Why the Last Test Case Times Out

The last test case likely has a large n and k , causing the $O(n*k)$ solution to exceed the time limit.

Comparison with the Optimized Solution

The optimized solution using a **Deque**:

- Runs in $O(n)$ time because each element is pushed and popped at most once.

- Does not require expensive searches or removals from the middle of a data structure.

Conclusion

Your original solution was **logically correct** but **inefficient** for large inputs. The Deque-based solution is the standard efficient approach for the sliding window maximum problem.

If you want to keep the heap-based approach for its simplicity, you can use a **lazy removal** technique with a heap. However, that would require additional space and is still not as efficient as the Deque method. But for the purpose of this problem, the Deque solution is preferred.

Also provide the similar problems in LeetCode with Number, Name of the problems and URL in a tabular format.

Here are some similar sliding window problems from LeetCode:

Number	Problem Name	URL
239	Sliding Window Maximum	https://leetcode.com/problems/sliding-window-maximum/
480	Sliding Window Median	https://leetcode.com/problems/sliding-window-median/
643	Maximum Average Subarray I	https://leetcode.com/problems/maximum-average-subarray-i/

Number	Problem Name	URL
1425	Constrained Subsequence Sum	https://leetcode.com/problems/constrained-subsequence-sum/
1438	Longest Continuous Subarray With Absolute Diff Less Than or Equal to Limit	https://leetcode.com/problems/longest-continuous-subarray-with-absolute-diff-less-than-or-equal-to-limit/
1493	Longest Subarray of 1's After Deleting One Element	https://leetcode.com/problems/longest-subarray-of-1s-after-deleting-one-element/
1696	Jump Game VI	https://leetcode.com/problems/jump-game-vi/
2090	K Radius Subarray Averages	https://leetcode.com/problems/k-radius-subarray-averages/
2134	Minimum Swaps to Group All 1's Together II	https://leetcode.com/problems/minimum-swaps-to-group-all-1s-together-ii/
2762	Continuous Subarrays	https://leetcode.com/problems/continuous-subarrays/

These problems all involve sliding window techniques, with variations including:

- Finding maximum/minimum values in windows
- Calculating medians or averages
- Maintaining constraints on window values