**Documentation for Sliding Window Median**

1. **Problem Statement**

   You are given an array of integers, nums, and an integer k. There is a sliding window of size k moving from left to right of the array. At each step, the window moves right by one position.

   You must return an array of medians where medians[i] is the median of the i-th sliding window.

   - If k is odd, the median is the middle element.
   - If k is even, the median is the average of the two middle elements.

   Example:

   Input: nums = [1,3,-1,-3,5,3,6,7], k = 3
   Output: [1.00000, -1.00000, -1.00000, 3.00000, 5.00000, 6.00000]

2. **Intuition**

   To maintain the median efficiently as the window slides:

   - We need a way to insert and remove elements in log time.
   - We also need to quickly access the middle elements.

   This motivates the use of two heaps:

   - A max heap (small) for the smaller half of the window.
   - A min heap (large) for the larger half.

3. **Key Observations**

   - The max heap stores the negative values because Python only supports min heaps.
   - For a window of size k:
     - If k is odd, the median is the top of the small.
     - If k is even, it's the average of the tops of small and large.

- Elements exiting the window need to be delayed deleted, not immediately removed from the heap.
- We use a hash map (delayed) to keep track of elements that should be removed when they reach the top of a heap.

## 4. Approach

Steps:

- Use two heaps: small (max heap) and large (min heap).
- Maintain heap balance such that:
    - Small has the same size as large or one more.
- Insert new elements into one of the heaps.
- Remove elements that have exited the window using delayed deletion.
- Prune the top of heaps whenever invalid elements are found.
- After each valid window, compute the median.

## 5. Edge Cases

- nums with negative and positive values.
- Repeated values in nums.
- When k == 1, the median is simply the element itself.
- When k == len(nums), only one median is returned.
- Large values (up to $2^{31}-1$) must be handled efficiently.

## 6. Complexity Analysis

 Time Complexity:

- Each insertion/removal takes O(log k).
- Total operations = n (length of nums).
- So, overall time complexity: O(n log k).

💾 Space Complexity:

- Two heaps of size up to k: O(k)
- Delayed map for at most k items: O(k)
- Total space complexity: O(k)

7. **Alternative Approaches**

❌ Brute Force:

- Sort every window: O(k log k) per window.
- Total: O(nk log k) — too slow for large inputs.

✅ Balanced BST (like SortedList from bisect or SortedContainers):

- Insertion/removal in O(log k)
- Easier than heap-based with delayed deletions.
- Cleaner but requires external libraries or more complex code in pure Python.

8. **Test Cases**

assert Solution() medianSlidingWindow ([1,3,-1,-3,5,3,6,7], 3) == [1.0, -1.0, -1.0, 3.0, 5.0, 6.0]

assert Solution() medianSlidingWindow ([1,2,3,4,2,3,1,4,2], 3) == [2.0, 3.0, 3.0, 3.0, 2.0, 3.0, 2.0]

assert Solution().medianSlidingWindow([1], 1) == [1.0]

assert Solution().medianSlidingWindow([1, 2], 2) == [1.5]

9. **Final Thoughts**

- The two-heaps approach is efficient and optimal for this type of median tracking problem.
- It's a great example of using data structures cleverly to maintain invariants (like balance and order).
- The delayed deletion strategy may seem tricky at first but is necessary due to heap limitations in Python.