Topic Introduction

Today, let's dive into a fascinating family of problems: **Order Statistics in Data Streams using Heaps**.

Imagine you are getting a stream of numbers, one by one, and you need to answer queries like "What's the median so far?", "What's the kth largest element now?", or "What's the median in the last k numbers?". These scenarios are common in systems that process real-time data, like stock tickers, live sensors, or user analytics.

The key concept uniting all these problems is the **heap** (or priority queue). A heap is a special tree-based structure that allows you to quickly get, add, or remove the largest or smallest element. There are two main kinds:

- **Min-heap:** Smallest element is always on top.
- **Max-heap:** Largest element is always on top.

Why heaps? They let you efficiently maintain order statistics (like the median or kth largest) as new items arrive, especially when sorting the whole stream every time would be too slow.

How does a heap work? Imagine a binary tree where each parent node is less than (min-heap) or greater than (max-heap) its children. With Python's `heapq`, you get a min-heap by default, but you can simulate a max-heap by pushing negative values.

When are heaps useful in interviews?

- Whenever you need the "top k" or "middle" value in a changing list.
- When streaming data arrives and you can't (or don't want to) sort everything each time.
- Common in real-time analytics, simulations, and scheduling.

Quick example (not one of our focus problems):
Suppose you want to always know the smallest 3 numbers out of a stream. You can use a max-heap of size 3:

- As numbers arrive, push them into the heap.
- If the heap grows bigger than 3, pop the largest.
- Now the heap always contains the 3 smallest elements seen so far.

Why Are These Problems Grouped Together?

Today’s trio all deal with **finding order statistics (like median or kth largest) in a stream of data, using heaps**:

- **Find Median from Data Stream:** Maintain the median as numbers arrive.
- **Sliding Window Median:** Maintain the median of the last k numbers as a sliding window moves.
- **Kth Largest Element in a Stream:** Always return the kth largest number seen so far.

They all use heaps to keep track of the middle or kth element efficiently, but each adds its own twist. Let’s tackle them in an order that builds your understanding step by step.

Problem 1: Find Median from Data Stream

Problem Statement (Rephrased):
Leetcode 295. Find Median from Data Stream

Design a class that supports two operations:

- `addNum(num)`: Add a number from a data stream to your data structure.
- `findMedian()`: Return the median of all elements so far.

The median is the middle number if the count is odd, or the average of the two middle numbers if the count is even.

Example:

Suppose the numbers arrive as: [1, 2, 3]

- Add 1; median is 1.
- Add 2; median is (1+2)/2 = 1.5.
- Add 3; median is 2.

Let's walk through a harder example:

Add: 5, 3, 8, 9

Medians after each insertion:

- [5] => 5
- [3, 5] => (3+5)/2 = 4
- [3, 5, 8] => 5
- [3, 5, 8, 9] => (5+8)/2 = 6.5

Challenge for you:

Add: 7, 1, 4, 10

What are the medians after each insertion?

Brute-force approach:

Keep all numbers in a list, sort it whenever `findMedian()` is called, and return the middle(s).

- Time for `addNum`: O(1)
- Time for `findMedian`: O(n log n) (because of sorting)

Optimal approach: Two Heaps!

The Pattern:

- Use a **max-heap** for the smaller half of numbers.
- Use a **min-heap** for the larger half.
- The heaps are balanced so that their sizes differ by at most 1.
- Median is either the top of one heap (if odd total) or the average of the tops (if even).

Step-by-step logic:

- When a new number comes:
  - If it's less than or equal to the max of the smaller half, put it in max-heap (as negative, since Python's heapq is min-heap).
  - Else, put it in min-heap.
- Balance the heaps if their sizes differ by more than 1.
- To find median:
  - If heaps are the same size, median is average of the tops.
  - Else, the heap with more elements has the median on top.

Here's a clean Python solution:

```python
import heapq

class MedianFinder:
    def __init__(self):
        # Max-heap (invert numbers to use Python's min-heap)
        self.small = []
        # Min-heap
        self.large = []

    def addNum(self, num):
        # Add to max-heap (invert num for max-heap)
        heapq.heappush(self.small, -num)
        # Move the largest in small to large to maintain order
        heapq.heappush(self.large, -heapq.heappop(self.small))
        # If large has more, move back to small
        if len(self.large) > len(self.small):
            heapq.heappush(self.small, -heapq.heappop(self.large))

    def findMedian(self):
        # If equal size, average the two middles
        if len(self.small) == len(self.large):
            return (-self.small[0] + self.large[0]) / 2
        # Else, small has one extra
        return -self.small[0]
```

Time Complexity:

- `addNum`: O(log n) (for heap insertions)
- `findMedian`: O(1) (just look at heap tops)

Space Complexity:

- O(n) for storing all numbers in heaps.

Let’s explain each part:

- `self.small` is the max-heap (as negatives).
- `self.large` is the min-heap.
- When adding, always push to `small` (as negative), then move the largest to `large` to keep order.
- If `large` ever has more elements, move one back to `small` to balance.
- `findMedian` just checks heap sizes and returns accordingly.

Trace with a test case:

Add 2, 1, 5, 7, 2, 0, 5

- After each add, the heaps look like:
    - [2] | []   => median: 2
    - [1] | [2]   => median: (1+2)/2 = 1.5
    - [2,1] | [5]   => median: 2

- [2,1] | [5,7] => median: (2+5)/2 = 3.5
- [2,2,1] | [5,7] => median: 2
- [2,1,0] | [2,5,7] => median: (2+2)/2 = 2
- [2,2,1,0] | [5,5,7] => median: 2

Try this test case yourself:

Add: 6, 10, 2, 6

What are the medians after each insertion?

Take a moment to solve this on your own before jumping into the solution!

Problem 2: Sliding Window Median

Problem Statement (Rephrased):

Leetcode 480. Sliding Window Median

Given an array of numbers and a window size k, as the window slides from left to right, return the median of each window.

Example:

Input: nums = [1,3,-1,-3,5,3,6,7], k = 3

Output: [1, -1, -1, 3, 5, 6]

Explanation:

- Window [1,3,-1] -> median = 1
- Window [3,-1,-3] -> median = -1
- Window [-1,-3,5] -> median = -1
- Window [-3,5,3] -> median = 3
- Window [5,3,6] -> median = 5
- Window [3,6,7] -> median = 6

Try this test case:

nums = [2, 4, 6, 8, 10], k = 2

What should the output be?

How is this similar and different from Problem 1?

- Similar: Need to maintain the median as numbers enter.
- Difference: Also need to efficiently remove numbers as they leave the window.

Brute-force approach:

For every window, copy and sort the window, then get the median.

- Time: O(n k log k) (sorting for each window)

Optimal approach:

Still use two heaps (like before). But now, when the window slides, we must also remove the outgoing number from the relevant heap.

Challenge:

Heaps don't support O(log n) removal of arbitrary elements. So we use a "delayed removal" approach:

- Keep a 'delayed' map for numbers that should be removed, but are not at the heap top yet.

- When popping from the heap, skip any numbers that are marked for removal.

Step-by-step pseudocode:

```
Initialize two heaps: small (max-heap), large (min-heap)
For i in 0 to n-1:
    Add nums[i] to proper heap (same as before)
    Balance heaps
    If window has k elements:
        Append median to result
        Mark nums[i-k+1] for removal in delayed map
        If nums[i-k+1] <= -small[0]:  # It was in small
            Decrease size of small heap
        Else:
            Decrease size of large heap
        While small and top of small is marked for removal:
            Pop from small
        While large and top of large is marked for removal:
            Pop from large
```

Time Complexity:

- Each add/remove: O(log k)
- Total: O(n log k)

Space Complexity:

- O(k) for heaps and delayed map

Example dry-run:

nums = [1, 3, -1, -3, 5, 3, 6, 7], k = 3

First window: [1, 3, -1]

- Add 1: small=[-1], large=[]
- Add 3: small=[-1], large=[3]
- Add -1: small=[-1,-1], large=[3]
- Median: 1

When sliding, remove 1, add -3, and so on. Remember to "lazy delete".

Test case to try:

nums = [4, 2, 12, 3, 7], k = 3

What are the medians at each step?

Want a challenge?

Try implementing the delayed removal yourself!

Problem 3: Kth Largest Element in a Stream

Problem Statement (Rephrased):

Leetcode 703. Kth Largest Element in a Stream

Design a class initialized with an integer k and an integer array nums. Each time you call `add(val)`, it adds val to the stream and returns the kth largest element seen so far.

Example:

k = 3, nums = [4, 5, 8, 2]

After add(3): [4, 5, 8, 2, 3] -> 4 (third largest)

After add(5): [4, 5, 8, 2, 3, 5] -> 5

After add(10): [4, 5, 8, 2, 3, 5, 10] -> 5

After add(9): ...

After add(4): ...

How is this similar/different?

- Similar: Streaming numbers, need to maintain an order statistic.
- Difference: Always the kth largest, not the median. Only need to track k largest elements.

Brute-force approach:

Store all numbers, sort every time:

- `add`: O(1), `get kth`: O(n log n)

Optimal approach:

- Use a min-heap of size k.
- Always keep the k largest elements in the heap.
- When adding a new number:
  - If heap size < k, push val.
  - Else, push val and pop the smallest.
- kth largest is always at heap[0].

Pseudocode:

```
Initialize min-heap with first k elements of nums (or all, if < k)
For each add(val):
    If heap size < k:
        Add val to heap
    Else if val > heap[0]:
        Push val and pop smallest
    Return heap[0]
```

Try this test case:

k = 2, nums = [1, 2, 3]

add(4): ?

add(0): ?

add(5): ?

Time Complexity:

- Each add: O(log k)
- Space: O(k)

Can you spot the pattern?

This is a classic "maintain a running top-k" problem.

Stretch:

How would you solve this if you needed the kth *smallest* instead?

Summary and Next Steps

Let’s recap:

- All three problems deal with **order statistics in a stream** using heaps.
- **Two-heaps** (max/min) let you quickly find the median as numbers arrive (and even as they leave, with extra care).
- A **single min-heap** of size k is perfect for tracking the kth largest element in a stream.
- The main trick: heaps let you maintain the relevant middle or kth element without sorting everything every time.

Common mistakes and traps:

- Not balancing the two heaps for the median.
- Forgetting to "lazy-delete" outgoing elements in the sliding window median.
- Using a max-heap for kth largest instead of a min-heap of size k.

Key patterns to remember:

- **Median:** Two balanced heaps
- **Kth largest:** One min-heap of size k
- **Sliding window:** Same as median, but handle removal carefully (lazy deletion)

Action List:

- Solve all three problems on your own, even the one with code provided.
- For Problem 2 and 3, try a different approach (e.g., multiset/binary search tree for window median).
- Explore related problems, like "Top K Frequent Elements" or "Minimum Sliding Window".
- Compare your solutions with others online, especially for edge-case handling.
- Dry-run with pen and paper for tricky cases.
- Don’t worry if you get stuck — keep practicing and the patterns will become second nature!

Happy coding, and see you at the next PrepLetter!