LeetCode Problem Description-295(Easy Words me)

Problem:

- Ek data stream (integers ka sequence) aa raha hai, tumhe har waqt **median** nikalna hai.
- Median ka matlab:
  - Agar total elements odd → middle element.
  - Agar total elements even → do middle ka average.

Example:

```
arr = [1, 2] → median = (1 + 2)/2 = 1.5
arr = [1, 2, 3] → median = 2
```

Constraints:

- `10^5 <= num <= 10^5`
- Maximum 50,000 calls `addNum` / `findMedian`.

Why Two Heaps Approach?

Sorting har bar costly hota (`O(n log n)` per insertion).

Two heaps ka fayda:

- `left` (Max Heap): Chhota half store karega, **largest of smaller half** at top.
- `right` (Min Heap): Bada half store karega, **smallest of larger half** at top.
- Median:
  - Agar size equal → `(left.top() + right.top()) / 2.0`
  - Agar size unequal → jis heap ka size zyada hai uska `top`.

Code Explanation:

```
class MedianFinder {
public:
    priority_queue<int> left; // max heap (smaller half)
    priority_queue<int, vector<int>, greater<int>> right; // min heap (larger half)

    MedianFinder() {}

    void addNum(int num) {
        // Step 1: Decide where to insert
        if (left.empty() || left.top() > num) {
            left.push(num);
        } else {
            right.push(num);
        }

        // Step 2: Balance heaps (size diff ≤ 1)
        if (left.size() > right.size() + 1) {
            right.push(left.top());
            left.pop();
        }
        if (right.size() > left.size() + 1) {
            left.push(right.top());
            right.pop();
        }
    }

    double findMedian() {
        // Equal size → average of two middles
        if (left.size() == right.size()) {
            return (left.top() + right.top()) / 2.0;
        }
        // Size unequal → top of larger heap
        if (right.size() > left.size()) {
```

```
            return right.top();
        }
        return left.top();
    }
};
```

Dry Run Example:

Input:

```
["MedianFinder", "addNum", "addNum", "findMedian", "addNum", "findMedia
n"]
[[], [1], [2], [], [3], []]
```

Step-by-step:

1. `addNum(1)`

   left = [1], right = [] → median = 1 (not yet called)

2. `addNum(2)`

   left = [1], right = [2] → equal size → median = (1+2)/2 = 1.5

3. `findMedian()` → output 1.5

4. `addNum(3)`

   left = [1], right = [2,3] (balance → left=[2,1], right=[3])

   median = left.top() = 2

5. `findMedian()` → output 2.0

Complexity:

- `addNum()` → `O(log n)` (heap insert)
- `findMedian()` → `O(1)`

  Efficient for streaming data.
