Heap Pattern in DSA for Problem Solving

Complete Interview-Ready Technical Handbook

TABLE OF CONTENTS

SECTION 1 – HEAP FOUNDATION

1.1 What is a Heap?

A **Heap** is a specialized **complete binary tree-based** data structure that satisfies the **heap property**:

- **Max Heap**: Parent node ≥ all its children
- **Min Heap**: Parent node ≤ all its children

Key Characteristics:

- **Complete Binary Tree**: All levels are fully filled except possibly the last, which is filled from left to right
- **Array Representation**: Can be efficiently stored in an array without pointers
- **Parent-Child Relationship**:
  - Parent index: `(i-1)/2`
  - Left child: `2*i + 1`
  - Right child: `2*i + 2`

1.2 Why is Heap Needed?

Problem Statement:
You need to repeatedly access the minimum/maximum element efficiently while also being able to insert and delete elements.

Solutions Comparison:

Approach	Access Min/Max	Insert	Delete Min/Max
Unsorted Array	O(n)	O(1)	O(n)

Approach	Access Min/Max	Insert	Delete Min/Max
Sorted Array	O(1)	O(n)	O(1)
BST	O(h)	O(h)	O(h)
Heap	**O(1)**	**O(log n)**	**O(log n)**

Why Heap Wins:

- ✅ Constant time access to min/max element
- ✅ Logarithmic time for insertions and deletions
- ✅ Space efficient (array-based implementation)
- ✅ Better practical performance than BST for priority operations
- ✅ No need to maintain full ordering (only partial ordering needed)

1.3 Real-World Applications

Operating Systems

- **Process Scheduling**: CPU assigns priority to processes using priority queues (heaps)
- **Memory Management**: Allocation of memory blocks based on size priority
- **Event-Driven Simulation**: Managing events by timestamp

Networking

- **Bandwidth Management**: Quality of Service (QoS) routing
- **Load Balancing**: Distributing requests to least-loaded servers
- **Packet Scheduling**: Managing network packet priorities

Streaming & Real-Time Systems

- **Media Streaming**: Buffer management for continuous playback
- **Stock Trading Systems**: Order book management (buy/sell orders)
- **Real-time Analytics**: Top-K trending items (Twitter trends, YouTube trending)

Data Processing

- **Dijkstra's Algorithm**: Shortest path finding
- **Huffman Encoding**: Data compression
- **Merge K Sorted Files**: External sorting in databases

E-commerce & Gaming

- **Product Recommendations**: Top-rated products
- **Leaderboards**: Top-K players
- **Dynamic Pricing**: Auction systems

1.4 Heap vs Sorted Array

Feature	Heap	Sorted Array
Structure	Complete binary tree	Linear contiguous memory

Feature	Heap	Sorted Array
Ordering	Partial (only parent-child)	Complete ordering
Access Min/Max	O(1)	O(1)
Search Element	O(n)	O(log n) via binary search
Insert	O(log n)	O(n) - need to maintain order
Delete Min/Max	O(log n)	O(1) for max, O(n) for min
Space	O(n)	O(n)
Use When	Frequent insertions/deletions with min/max access	Static data with frequent searches

💡 **Key Insight**: Heap maintains **just enough order** to efficiently access extremes without the overhead of full sorting.

1.5 Heap vs Balanced BST

Feature	Heap	Balanced BST
Structure	Complete binary tree	Balanced binary search tree
Property	Heap property (parent ≥/≤ children)	BST property (left < root < right)
Access Min	O(1)	O(log n)
Access Max	O(1) - in max heap	O(log n)
Search Element	O(n)	O(log n)
Insert	O(log n)	O(log n)
Delete	O(log n)	O(log n)
Sorted Traversal	O(n log n) - via repeated deletion	O(n) - inorder
Space Overhead	Array-based, no pointers	Pointers for each node
Cache Performance	Better (array locality)	Worse (pointer chasing)
Use When	Only need min/max access	Need range queries, sorted order

💡 **Key Insight**: BST maintains **complete ordering** allowing efficient range queries, while Heap maintains **partial ordering** optimized for priority access.

1.6 Time Complexity Table

Operation	Time Complexity	Space Complexity	Notes
Build Heap	O(n)	O(1) auxiliary	Bottom-up heapify
Insert (Push)	O(log n)	O(1)	Bubble up operation
Delete Min/Max (Pop)	O(log n)	O(1)	Bubble down operation
Peek Min/Max (Top)	O(1)	O(1)	Access root element

Operation	Time Complexity	Space Complexity	Notes
Search Element	O(n)	O(1)	No ordering for non-root
Decrease Key	O(log n)	O(1)	Bubble up after update
Increase Key	O(log n)	O(1)	Bubble down after update
Delete Arbitrary	O(log n)	O(1)	Requires element position
Merge Two Heaps	O(n + m)	O(n + m)	Build new heap
Heapify Array	O(n)	O(1) auxiliary	Surprising O(n) not O(n log n)
Heap Sort	O(n log n)	O(1)	In-place sorting

⚠️ **Important Notes:**

- **Why Build Heap is O(n)?** Bottom-up approach: most nodes are near leaves requiring fewer swaps
- **Amortized Analysis**: Some operations may have better amortized complexity in specific use cases
- **Space Complexity**: Heap typically uses O(n) for storage, O(1) auxiliary for operations

SECTION 2 – TYPES OF HEAPS

2.1 Min Heap

Definition: A complete binary tree where every parent node has a value **less than or equal to** its children.

Structure:

```
        10
       /  \
      15   20
     / \   / \
    25 30 35 40
```

Property:

- Root contains the **minimum** element
- For any node at index `i`: `heap[i] ≤ heap[2*i+1]` and `heap[i] ≤ heap[2*i+2]`

Array Representation:

```
[10, 15, 20, 25, 30, 35, 40]
```

Use Cases:

- Finding minimum element repeatedly
- Dijkstra's shortest path algorithm
- Prim's minimum spanning tree
- Huffman encoding (character frequency)
- Job scheduling (earliest deadline first)
- Event-driven simulation (next event time)

Operations:

- `getMin()` : O(1) - return root
- `insert()` : O(log n) - add at end, bubble up
- `extractMin()` : O(log n) - remove root, bubble down

2.2 Max Heap

Definition: A complete binary tree where every parent node has a value **greater than or equal to** its children.

Structure:

```
        40
       /  \
      35   30
     / \   / \
    25 20 15 10
```

Property:

- Root contains the **maximum** element
- For any node at index `i` : `heap[i] ≥ heap[2*i+1]` and `heap[i] ≥ heap[2*i+2]`

Array Representation:

```
[40, 35, 30, 25, 20, 15, 10]
```

Use Cases:

- Finding maximum element repeatedly
- Heap sort (descending order)
- Priority scheduling (highest priority first)
- Maximum flow algorithms
- Top-K smallest elements (counter-intuitive: use max heap!)
- Median finding (larger half in max heap)

2.3 Binary Heap

Definition: A heap implemented as a **complete binary tree** using an array.

Characteristics:

- Most common heap implementation
- Both min heap and max heap can be binary heaps
- **Complete**: All levels filled except possibly last (filled left to right)
- Array-based: No need for pointers

Index Formulas:

```
Parent(i) = (i - 1) / 2
LeftChild(i) = 2 * i + 1
RightChild(i) = 2 * i + 2
```

Visual Example:

```
Index:  0   1   2   3   4   5   6
Array: [3,  5,  8,  9, 10, 15, 20]

Tree:       3
          /   \
         5     8
        / \   / \
       9  10 15 20
```

Advantages:

- ✅ **Cache-friendly**: Contiguous memory improves cache performance
- ✅ **Space-efficient**: No pointer overhead (saves 16 bytes per node in 64-bit systems)
- ✅ **Simple implementation**: Easy to code and understand

Disadvantages:

- ❌ **Fixed type**: Either min or max, can't efficiently support both simultaneously
- ❌ **No efficient search**: Finding arbitrary element is O(n)
- ❌ **No efficient merge**: Merging two binary heaps is O(n)

Other Heap Variants:

- **Fibonacci Heap**: Better amortized complexity for decrease-key: O(1)
- **Binomial Heap**: Efficient merge: O(log n)
- **Pairing Heap**: Practical alternative to Fibonacci heap
- **D-ary Heap**: Each node has d children (not just 2)

2.4 Heap vs Priority Queue

Conceptual Relationship:

Aspect	Heap	Priority Queue
Type	Data Structure	Abstract Data Type (ADT)
Definition	Specific implementation (tree-based)	Interface/Concept
Analogy	ArrayList	List interface
Operations	heapify, bubble up, bubble down	push, pop, top
Implementation	One specific way to organize data	Can be implemented multiple ways

Priority Queue Implementations:

Implementation	Insert	Delete	Access	Use Case
Heap	O(log n)	O(log n)	O(1)	**Best general choice**
Unsorted Array	O(1)	O(n)	O(n)	Rare insertions
Sorted Array	O(n)	O(1)	O(1)	Rare insertions
BST	O(log n)	O(log n)	O(log n)	Need range queries too

Implementation	Insert	Delete	Access	Use Case
Fibonacci Heap	O(1)	O(log n)	O(1)	Theoretical algorithms

💡 **Key Distinction:**

```
Priority Queue = "What operations do I need?"
Heap = "How do I implement those operations?"
```

In Interviews:

- When someone says "use a priority queue," they typically mean "use a heap"
- In C++ STL: `priority_queue` is implemented using a binary heap
- In Java: `PriorityQueue` is implemented using a binary heap
- In Python: `heapq` module implements a binary min heap

SECTION 3 – HEAP DECLARATION IN C++

3.1 Basic priority_queue Syntax

Default Declaration (Max Heap):

```cpp
#include <queue>

// Max heap of integers (default)
priority_queue<int> maxHeap;

// Operations
maxHeap.push(10);        // Insert element
maxHeap.push(20);
maxHeap.push(5);

int top = maxHeap.top(); // Access maximum (20)
maxHeap.pop();           // Remove maximum

bool empty = maxHeap.empty(); // Check if empty
int size = maxHeap.size();    // Get size
```

Common Data Types:

```cpp
priority_queue<int> pq1;           // Integers
priority_queue<double> pq2;        // Doubles
priority_queue<char> pq3;          // Characters
priority_queue<string> pq4;        // Strings (lexicographic order)
```

3.2 Min Heap vs Max Heap

Max Heap (Default):

```
// Method 1: Default
priority_queue<int> maxHeap;

// Method 2: Explicit (same as default)
priority_queue<int, vector<int>, less<int>> maxHeap;
```

Min Heap:

```
// Method 1: Using greater<int>
priority_queue<int, vector<int>, greater<int>> minHeap;

// Method 2: Using negative values (hack for max heap)
priority_queue<int> maxHeapAsMin;
maxHeapAsMin.push(-10); // Insert negative
int val = -maxHeapAsMin.top(); // Negate to get original
```

Complete Syntax Breakdown:

```
priority_queue<Type, Container, Comparator>
               |         |           |
               |         |           +-- How to compare (less/greater)
               |         +-------------- Underlying container (vector)
               +------------------------ Data type
```

3.3 Custom Comparators

For Custom Objects:

```cpp
struct Node {
    int value;
    int priority;
};

// Method 1: Operator overloading
struct Node {
    int value;
    int priority;

    bool operator<(const Node& other) const {
        return priority < other.priority; // Max heap by priority
    }
};

priority_queue<Node> pq;

// Method 2: Comparator struct
struct CompareNode {
    bool operator()(const Node& a, const Node& b) {
        return a.priority < b.priority; // Max heap by priority
    }
};

priority_queue<Node, vector<Node>, CompareNode> pq;

// Method 3: Lambda function
auto cmp = [](const Node& a, const Node& b) {
    return a.priority < b.priority;
};

priority_queue<Node, vector<Node>, decltype(cmp)> pq(cmp);
```

⚠️ Important: Comparator Logic

```cpp
// For priority_queue, the comparator defines "LESS PRIORITY"
// If comp(a, b) returns true, b has HIGHER priority than a

// Max heap: use less<int> or a < b
// Returns true when a is smaller, so b (larger) has higher priority

// Min heap: use greater<int> or a > b
// Returns true when a is larger, so b (smaller) has higher priority
```

Multi-criteria Sorting:

```cpp
struct Task {
    int deadline;
    int profit;
};

// Sort by deadline (min), then by profit (max)
auto cmp = [](const Task& a, const Task& b) {
    if (a.deadline != b.deadline)
        return a.deadline > b.deadline; // Min deadline
    return a.profit < b.profit;         // Max profit if deadlines equal
};

priority_queue<Task, vector<Task>, decltype(cmp)> pq(cmp);
```

3.4 Pair and Vector Heaps

Pair Heaps:

```cpp
// Max heap of pairs (default: compare first, then second)
priority_queue<pair<int, int>> maxPairHeap;

maxPairHeap.push({5, 10});
maxPairHeap.push({5, 20});  // This has higher priority (same first, larger second)
maxPairHeap.push({10, 5});  // This is highest (larger first)

// Min heap of pairs
priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> minPairHeap;

// Custom pair comparator - sort by second element
auto cmp = [](pair<int,int> a, pair<int,int> b) {
    return a.second > b.second; // Min heap by second element
};
priority_queue<pair<int,int>,
               vector<pair<int,int>>,
               decltype(cmp)> customPairHeap(cmp);
```

Common Pair Patterns:

```cpp
// {value, index} - track original position
priority_queue<pair<int, int>> pq;
for (int i = 0; i < n; i++) {
    pq.push({arr[i], i});
}

// {frequency, element} - frequency-based problems
priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> minHeap;

// {distance, node} - Dijkstra's algorithm
priority_queue<pair<int, int>,
               vector<pair<int, int>>,
               greater<pair<int, int>>> dijkstraHeap;
```

Quick Reference Table:

Type	Max Heap	Min Heap
`int`	`priority_queue<int>`	`priority_queue<int, vector<int>, greater<int>>`
`pair<int,int>`	`priority_queue<pair<int,int>>`	`priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>>`
Custom `Node`	Override `operator<`	Use custom comparator with `>` logic

SECTION 4 – HEAP PATTERNS

Pattern 1: K-th Largest / Smallest Pattern

When to Use:

- Problem asks for "Kth largest" or "Kth smallest" element
- Need to find a single element at specific rank
- Stream of numbers where you need to track Kth element

How to Detect:

- 🔍 Keywords: "Kth largest", "Kth smallest", "Kth most frequent"
- 🔍 Constraint: K is significantly smaller than N
- 🔍 Need efficient solution better than full sorting O(n log n)

Approach:

Goal	Heap Type	Logic
Kth Largest	Min Heap of size K	Keep K largest elements; root is Kth largest
Kth Smallest	Max Heap of size K	Keep K smallest elements; root is Kth smallest

Why This Works:

```
Finding Kth Largest with Min Heap (K=3):
Array: [7, 10, 4, 3, 20, 15]

Process:
Step 1: [7]           size=1
Step 2: [7, 10]       size=2
Step 3: [7, 10, 4]    size=3 (heap full)
Step 4: [7, 10, 4] → min=4, new=3, 3<4, skip
Step 5: [7, 10, 4] → min=4, new=20, 20>4, pop(4), push(20) → [7, 10, 20]
Step 6: [7, 10, 20] → min=7, new=15, 15>7, pop(7), push(15) → [10, 15, 20]

Result: min of heap = 10 (3rd largest) ✓
```

💡 **Logic:**

- Min heap maintains K largest elements
- Smallest of these K elements is the Kth largest
- Any new element larger than heap top replaces the minimum

Pseudocode:

```
FUNCTION findKthLargest(array, k):
    CREATE minHeap of size k

    FOR each number in array:
        INSERT number into minHeap

        IF size of minHeap > k:
            REMOVE minimum from minHeap

    RETURN top of minHeap  // This is Kth largest

TIME: O(n log k)
SPACE: O(k)
```

Complexity:

- Time: O(n log k) - n insertions, heap size k
- Space: O(k) - heap stores k elements

Common Problems:

- Kth Largest Element in Array
- Kth Largest in Stream
- Kth Smallest Element in Sorted Matrix

Pattern 2: Top K Elements Pattern

When to Use:

- Need **all** K largest or K smallest elements
- Return list/array of K elements, not just one
- Order within K elements may or may not matter

How to Detect:

- 🔍 Keywords: "top K", "K largest elements", "K smallest elements", "K most frequent"
- 🔍 Output requires array/list of K items
- 🔍 K is much smaller than N (K << N)

Approach:

Goal	Heap Type	Size	Result
Top K Largest	Min Heap	K	All K largest elements
Top K Smallest	Max Heap	K	All K smallest elements

💡 **Difference from Pattern 1:**

- Pattern 1: Return **single** Kth element
- Pattern 2: Return **all K** elements

Pseudocode:

```
FUNCTION topKLargest(array, k):
    CREATE minHeap of size k

    // Build heap with K largest
    FOR each number in array:
        INSERT number into minHeap
        IF size of minHeap > k:
            REMOVE minimum

    // Extract all K elements
    CREATE result array
    WHILE minHeap is not empty:
        ADD top of minHeap to result
        REMOVE top from minHeap

    RETURN result

TIME: O(n log k + k log k) = O(n log k)
SPACE: O(k)
```

Complexity:

- Time: O(n log k + k log k) = O(n log k)
  - n log k for building heap
  - k log k for extraction (often ignored as k << n)
- Space: O(k)

Common Problems:

- Top K Frequent Elements
- Top K Frequent Words
- K Closest Points to Origin

Pattern 3: Merge K Sorted Lists / Arrays

When to Use:

- Multiple sorted sequences to merge into one
- K sorted arrays/lists/files
- Need to maintain sorted order in output

How to Detect:

- 🔍 Keywords: "merge K sorted", "K sorted arrays/lists"
- 🔍 Input: K sorted sequences
- 🔍 Output: Single sorted sequence

Approach:

- Use **Min Heap** to track smallest element from each list
- Store `{value, listIndex, elementIndex}` in heap
- Repeatedly extract minimum and add next element from that list

Why Min Heap?

- At any moment, answer is minimum among K candidates (one from each list)
- Heap efficiently finds minimum among K elements
- Better than repeatedly scanning K arrays: O(log k) vs O(k)

Pseudocode:

```
FUNCTION mergeKSorted(arrays):
    CREATE minHeap storing {value, arrayIndex, elementIndex}
    CREATE result array

    // Initialize heap with first element from each array
    FOR i = 0 to K-1:
        IF arrays[i] is not empty:
            INSERT {arrays[i][0], i, 0} into minHeap

    // Extract min and add next from same array
    WHILE minHeap is not empty:
        {value, arrIdx, elemIdx} = EXTRACT minimum from minHeap
        ADD value to result

        IF elemIdx + 1 < size of arrays[arrIdx]:
            nextValue = arrays[arrIdx][elemIdx + 1]
            INSERT {nextValue, arrIdx, elemIdx + 1} into minHeap

    RETURN result

TIME: O(N log K) where N = total elements, K = number of arrays
SPACE: O(K) for heap + O(N) for output
```

Complexity:

- Time: O(N log K) where N = total elements, K = number of arrays
  - Each of N elements: push/pop from heap of size K
- Space: O(K) for heap + O(N) for output

Common Problems:

- Merge K Sorted Lists
- Merge K Sorted Arrays

- Kth Smallest in K Sorted Arrays

Pattern 4: Closest / Nearest Pattern

When to Use:

- Find K elements closest to a target value
- Distance/similarity metric involved
- Need K nearest neighbors

How to Detect:

- 🔍 Keywords: "K closest", "K nearest", "closest to X"
- 🔍 Distance calculation: absolute difference, Euclidean, Manhattan
- 🔍 Typically K << N

Approach:

- Calculate distance for each element
- Use **Max Heap** of size K (stores K closest)
- Heap stores elements with largest distance at top
- If new element is closer, replace top

Why Max Heap for "Closest"?

```
Finding K=3 closest to target=5 in [1, 3, 4, 7, 8, 9]

Distances: [4, 2, 1, 2, 3, 4]

Use Max Heap of size 3:
Process (element, distance):
(1,4): [4]
(3,2): [4,2]
(4,1): [4,2,1] ← heap full
(7,2): 2<4 (closer), pop(4), push(2): [2,2,1]
(8,3): 3>2 (farther), skip
(9,4): 4>2, skip

Heap contains: elements with distance [2,2,1] → actual elements [3,7,4] ✓
```

💡 **Logic:**

- Max heap maintains K closest elements
- Element at top has largest distance among K
- Any new closer element replaces this furthest element

Pseudocode:

```
FUNCTION kClosest(points, target, k):
    CREATE maxHeap of size k storing {distance, point}

    FOR each point in points:
        distance = CALCULATE distance from point to target

        INSERT {distance, point} into maxHeap

        IF size of maxHeap > k:
            REMOVE maximum (furthest point)

    // Extract all K closest
    CREATE result array
    WHILE maxHeap is not empty:
        ADD top.point to result
        REMOVE top from maxHeap

    RETURN result

TIME: O(n log k)
SPACE: O(k)
```

Complexity:

- Time: O(n log k)
- Space: O(k)

Common Distance Metrics:

- **Euclidean**: `sqrt((x1-x2)² + (y1-y2)²)`
- **Manhattan**: `|x1-x2| + |y1-y2|`
- **Absolute Difference**: `|num - target|`

Common Problems:

- K Closest Points to Origin
- Find K Closest Elements
- K Nearest Neighbors

Pattern 5: Sliding Window + Heap

When to Use:

- Need max/min in every window of size K
- Window slides through array
- Brute force would be O(n*k) for each window

How to Detect:

- 🔍 Keywords: "sliding window", "maximum in window", "window of size K"
- 🔍 Need to track extremum as window moves
- 🔍 Window size is fixed

Approach:

- Use heap to track elements in current window

- Store `{value, index}` to know when element leaves window
- For each position, remove outdated elements and add new

Challenge:
Heap doesn't support efficient deletion of arbitrary elements.

Solution: Lazy Deletion

- Don't remove elements immediately
- When accessing top, check if it's still in window
- If not, pop and check next

Pseudocode:

```
FUNCTION maxSlidingWindow(array, k):
    CREATE maxHeap storing {value, index}
    CREATE result array

    FOR i = 0 to length of array - 1:
        INSERT {array[i], i} into maxHeap

        IF i >= k - 1:  // Window formed
            // Lazy deletion: remove outdated elements
            WHILE maxHeap is not empty AND top.index <= i - k:
                REMOVE top from maxHeap

            ADD top.value to result

    RETURN result

TIME: O(n log n) - worst case all elements in heap
SPACE: O(n) - heap can grow to n in worst case
```

Complexity:

- Time: O(n log n) - worst case all elements in heap
- Space: O(n) - heap can grow to n in worst case

⚠️ **Note:**
Better approach exists using **deque** in O(n) time, but heap approach is:

- Easier to understand and code
- More generalizable (works for K largest, not just max)
- Acceptable for interview if you mention deque optimization

Common Problems:

- Sliding Window Maximum
- Sliding Window Median
- Sum of K Largest in Each Window

Pattern 6: Two Heaps Pattern (Median Finding)

When to Use:

- Find median in data stream

- Need to divide data into two halves
- Maintain balance between lower and upper half

How to Detect:

- 🔍 Keywords: "median", "middle element", "running median"
- 🔍 Stream of numbers or sliding window median
- 🔍 Need constant/fast median access

Approach:

- Use **two heaps**:
  - **Max Heap** (left): stores smaller half
  - **Min Heap** (right): stores larger half
- Maintain property: `maxHeap.size() >= minHeap.size()`
- Median is either top of maxHeap or average of both tops

Structure:

```
Smaller half          |  Larger half
(Max Heap)            |  (Min Heap)
  5                   |     8
 / \                  |    / \
3   4                 |   9   10

Top of maxHeap = 5 (largest of small half)
Top of minHeap = 8 (smallest of large half)

If sizes equal: median = (5 + 8) / 2 = 6.5
If maxHeap has one more: median = 5
```

Invariants:

1. **Size**: `maxHeap.size() == minHeap.size()` OR `maxHeap.size() == minHeap.size() + 1`
2. **Order**: Every element in maxHeap ≤ every element in minHeap
3. **Median**:
   - If sizes equal: median = `(maxHeap.top() + minHeap.top()) / 2.0`
   - If maxHeap larger: median = `maxHeap.top()`

Pseudocode:

```
CLASS MedianFinder:
    maxHeap  // Stores smaller half (max heap)
    minHeap  // Stores larger half (min heap)

    FUNCTION addNum(num):
        // Add to maxHeap first
        INSERT num into maxHeap

        // Ensure order: max of maxHeap <= min of minHeap
        IF maxHeap is not empty AND minHeap is not empty:
            IF top of maxHeap > top of minHeap:
                MOVE top of maxHeap to minHeap

        // Rebalance sizes
        IF size of maxHeap > size of minHeap + 1:
            MOVE top of maxHeap to minHeap
        IF size of minHeap > size of maxHeap:
            MOVE top of minHeap to maxHeap

    FUNCTION findMedian():
        IF size of maxHeap == size of minHeap:
            RETURN (top of maxHeap + top of minHeap) / 2.0
        ELSE:
            RETURN top of maxHeap

TIME: Add O(log n), Find Median O(1)
SPACE: O(n)
```

Common Problems:

- Find Median from Data Stream
- Sliding Window Median
- IPO (Maximize Capital)

Pattern 7: Heap + HashMap Pattern

When to Use:

- Need to track frequency/count along with heap operations
- Problems involving "most frequent", "least frequent"
- Need to map elements to their heap positions or properties

How to Detect:

- 🔍 Keywords: "most frequent", "top K frequent", "least recently used"
- 🔍 Need to count occurrences
- 🔍 Combination of ordering and lookup

Approach:

- **HashMap** to store frequency/count
- **Heap** to order by frequency
- Combine for efficient frequent element finding

Pseudocode:

```
FUNCTION topKFrequent(array, k):
    // Step 1: Build frequency map
    CREATE hashMap for frequency counting
    FOR each element in array:
        INCREMENT hashMap[element]

    // Step 2: Min heap of size k (by frequency)
    CREATE minHeap of size k storing {frequency, element}

    // Step 3: Maintain k most frequent
    FOR each {element, frequency} in hashMap:
        INSERT {frequency, element} into minHeap

        IF size of minHeap > k:
            REMOVE minimum (least frequent)

    // Step 4: Extract result
    CREATE result array
    WHILE minHeap is not empty:
        ADD top.element to result
        REMOVE top from minHeap

    RETURN result

TIME: O(n + m log k) where n = array size, m = unique elements
SPACE: O(m) for map + O(k) for heap
```

Common Problems:

- Top K Frequent Elements
- Top K Frequent Words
- Reorganize String
- Task Scheduler

Pattern 8: Scheduling / Greedy with Heap

When to Use:

- Scheduling tasks/jobs with priorities
- Greedy algorithms where next choice depends on min/max
- Interval scheduling, meeting rooms
- Task execution with dependencies or cooldowns

How to Detect:

- 🔍 Keywords: "schedule", "minimum time", "maximum profit", "intervals"
- 🔍 Greedy choice at each step
- 🔍 Need to pick next best option repeatedly

Approach:

- Model problem as selecting optimal element at each step
- Use heap to efficiently get next optimal choice
- Update heap after each selection

Pseudocode (Meeting Rooms II):

```
FUNCTION minMeetingRooms(intervals):
    IF intervals is empty:
        RETURN 0

    // Sort by start time
    SORT intervals by start time

    // Min heap tracks end times of ongoing meetings
    CREATE minHeap for end times

    FOR each interval in intervals:
        // Remove meetings that ended
        IF minHeap is not empty AND top of minHeap <= interval.start:
            REMOVE top from minHeap

        // Add current meeting's end time
        INSERT interval.end into minHeap

    RETURN size of minHeap  // Max concurrent meetings

TIME: O(n log n) for sorting + heap operations
SPACE: O(n)
```

Common Problems:

- Meeting Rooms II
- Task Scheduler
- CPU Scheduling
- Minimum Platforms Required

SECTION 5 – GENERIC HEAP TEMPLATES

Template 1: Kth Element (Largest/Smallest)

```
// Find Kth Largest Element
FUNCTION findKthLargest(array, k):
    CREATE minHeap of size k

    FOR each number in array:
        INSERT number into minHeap
        IF size of minHeap > k:
            REMOVE minimum

    RETURN top of minHeap

TIME: O(n log k)
SPACE: O(k)

// Find Kth Smallest Element
FUNCTION findKthSmallest(array, k):
    CREATE maxHeap of size k

    FOR each number in array:
        INSERT number into maxHeap
        IF size of maxHeap > k:
            REMOVE maximum

    RETURN top of maxHeap

TIME: O(n log k)
SPACE: O(k)
```

Template 2: Top K Elements

```
// Top K Largest Elements
FUNCTION topKLargest(array, k):
    CREATE minHeap of size k

    FOR each number in array:
        INSERT number into minHeap
        IF size of minHeap > k:
            REMOVE minimum

    CREATE result array
    WHILE minHeap is not empty:
        ADD top to result
        REMOVE top

    RETURN result

TIME: O(n log k)
SPACE: O(k)

// Top K Frequent Elements
FUNCTION topKFrequent(array, k):
    CREATE frequencyMap
    FOR each element in array:
        INCREMENT frequencyMap[element]

    CREATE minHeap of size k storing {frequency, element}
    FOR each {element, frequency} in frequencyMap:
        INSERT {frequency, element} into minHeap
        IF size of minHeap > k:
            REMOVE minimum

    CREATE result array
    WHILE minHeap is not empty:
        ADD top.element to result
        REMOVE top

    RETURN result

TIME: O(n + m log k) where m = unique elements
SPACE: O(m)
```

Template 3: Two Heaps (Median)

```
CLASS MedianFinder:
    maxHeap  // Max heap for smaller half
    minHeap  // Min heap for larger half

    FUNCTION addNum(num):
        INSERT num into maxHeap

        // Balance order
        IF maxHeap not empty AND minHeap not empty:
            IF top of maxHeap > top of minHeap:
                MOVE top of maxHeap to minHeap

        // Balance size
        IF size of maxHeap > size of minHeap + 1:
            MOVE top of maxHeap to minHeap
        IF size of minHeap > size of maxHeap:
            MOVE top of minHeap to maxHeap

    FUNCTION findMedian():
        IF size of maxHeap == size of minHeap:
            RETURN (top of maxHeap + top of minHeap) / 2.0
        RETURN top of maxHeap

TIME: Add O(log n), Median O(1)
SPACE: O(n)
```

Template 4: Merge K Sorted

```
FUNCTION mergeKSorted(arrays):
    CREATE minHeap storing {value, arrayIndex, elementIndex}
    CREATE result array

    // Initialize with first elements
    FOR i = 0 to k-1:
        IF arrays[i] not empty:
            INSERT {arrays[i][0], i, 0} into minHeap

    // Extract and add next
    WHILE minHeap not empty:
        {value, arrIdx, elemIdx} = EXTRACT min from minHeap
        ADD value to result

        IF elemIdx + 1 < size of arrays[arrIdx]:
            nextValue = arrays[arrIdx][elemIdx + 1]
            INSERT {nextValue, arrIdx, elemIdx + 1} into minHeap

    RETURN result

TIME: O(N log K) where N = total elements
SPACE: O(K)
```

Template 5: Sliding Window Heap

```
FUNCTION maxSlidingWindow(array, k):
    CREATE maxHeap storing {value, index}
    CREATE result array

    FOR i = 0 to length-1:
        INSERT {array[i], i} into maxHeap

        IF i >= k - 1:
            // Lazy deletion
            WHILE maxHeap not empty AND top.index <= i - k:
                REMOVE top

            ADD top.value to result

    RETURN result

TIME: O(n log n)
SPACE: O(n)
```

SECTION 6 – PROBLEM BANK

📗 Easy Level (10 Problems)

#	Problem Name	Pattern	Why Heap Required
1	**Kth Largest Element in Array**	Kth Element	Efficient tracking of K largest, O(n log k) vs O(n log n) sorting
2	**Last Stone Weight**	Max Heap Operations	Repeatedly need largest elements to simulate process
3	**Relative Ranks**	Sorting Alternative	Assign ranks based on scores, heap provides ordered extraction
4	**Kth Largest in Stream**	Kth Element (Stream)	Maintain Kth largest as elements arrive dynamically
5	**Minimum Cost of Ropes**	Greedy + Heap	Always merge two smallest ropes, min heap optimal choice
6	**Sort Characters by Frequency**	Heap + HashMap	Need elements in frequency order efficiently
7	**Find K Pairs with Smallest Sums**	Merge K Sorted	Multiple sorted sequences, need K smallest combinations
8	**Height Checker**	Sorting Alternative	Compare with sorted version
9	**Third Maximum Number**	Kth Element	Special case of Kth largest with K=3
10	**Kth Missing Positive Number**	Kth Element Variant	Find Kth element in missing sequence

📘 Medium Level (15 Problems)

#	Problem Name	Pattern	Why Heap Required
11	**Top K Frequent Words**	Heap + HashMap + Custom Comparator	Frequency-based with lexicographic tiebreaker
12	**K Closest Points to Origin**	Closest Pattern	Distance-based selection, maintain K closest efficiently
13	**Meeting Rooms II**	Scheduling + Heap	Track overlapping intervals, min heap for end times
14	**Task Scheduler**	Scheduling + Greedy	Frequency-based scheduling with cooldown constraints
15	**Find Median from Data Stream**	Two Heaps	Dynamic median requires balanced partitioning
16	**Merge K Sorted Lists**	Merge K Sorted	Multiple sorted inputs, need global ordering
17	**Kth Smallest in Sorted Matrix**	Merge K Sorted	Each row sorted, conceptual K-way merge
18	**Reorganize String**	Greedy + Heap	Frequency-based placement with adjacency constraint
19	**Sliding Window Maximum**	Sliding Window + Heap	Max in each window, heap with lazy deletion
20	**Ugly Number II**	Heap + DP	Generate sequence using heap for ordering
21	**Super Ugly Number**	Heap + DP	Generalization with multiple prime factors
22	**Smallest Range Covering K Lists**	Merge K Sorted + Range	Track range while processing K sorted arrays
23	**K Closest Elements**	Closest Pattern	Distance to target, maintain K closest
24	**Process Tasks Using Servers**	Scheduling + Two Heaps	Available + busy servers management
25	**Single-Threaded CPU**	Scheduling + Heap	Process tasks by availability and processing time

📕 Hard Level (10 Problems)

#	Problem Name	Pattern	Why Heap Required
26	**Sliding Window Median**	Two Heaps + Sliding Window	Maintain median as window slides, requires balancing
27	**IPO (Maximum Capital)**	Greedy + Two Heaps	Available projects (min capital) + profitable (max profit)
28	**Find K-th Smallest Pair Distance**	Binary Search + Heap	Binary search with heap validation
29	**Trapping Rain Water II**	Heap + BFS	2D boundary processing, need minimum boundary height
30	**Employee Free Time**	Merge Intervals + Heap	Merge K sorted employee schedules to find gaps
31	**Maximum Performance of Team**	Sorting + Min Heap	Pick top K while satisfying minimum constraint
32	**The Skyline Problem**	Heap + Events	Track building heights, heap for max height at events

#	Problem Name	Pattern	Why Heap Required
33	**Minimum Cost to Hire K Workers**	Greedy + Heap	Optimize cost with quality constraints
34	**Find Minimum in Rotated Sorted Array II**	Binary Search + Heap	Handle duplicates in rotated array
35	**Smallest Range II**	Greedy + Heap	Minimize range with additive changes

🔥 Bonus: Heap with Other Data Structures

#	Problem Name	Combination	Why This Combo
36	**LRU Cache**	Heap/List + HashMap	Track least recently used with efficient removal
37	**LFU Cache**	Two Heaps + HashMap	Track least frequently used with tie-breaking
38	**Design Twitter**	Heap + HashMap	Merge K user feeds for news feed generation
39	**Exam Room**	Heap + TreeSet	Track available seats with maximum distance
40	**Stock Price Fluctuation**	Heap + HashMap	Track current, max, min prices with updates

SECTION 7 – HEAP TECHNIQUES

7.1 Lazy Deletion

Problem:
Heaps don't support efficient deletion of arbitrary elements. Standard deletion requires finding element O(n) then reheapifying O(log n).

Lazy Deletion Concept:

- Don't immediately remove elements from heap
- Mark them as "invalid" or track them separately
- Remove only when they reach the top

When to Use:

- ✅ Sliding window problems
- ✅ Elements leave the relevant set but are still in heap
- ✅ Cost of tracking invalidity < cost of deletion

Implementation Strategies:

Strategy 1: Index-based Validity (Sliding Window)

```
FOR i = 0 to n-1:
    INSERT {value[i], i} into maxHeap

    // Lazy deletion: remove elements outside window
    WHILE maxHeap not empty AND top.index <= i - k:
        REMOVE top from maxHeap

    IF i >= k - 1:
        result[i-k+1] = top.value
```

Strategy 2: HashSet for Deleted Elements

```
CREATE heap
CREATE deletedSet

FUNCTION remove(value):
    ADD value to deletedSet

FUNCTION getTop():
    WHILE heap not empty AND top in deletedSet:
        REMOVE top from heap
    RETURN top of heap
```

Complexity:

- Amortized O(log n) - each element pushed and popped once
- Worst case: heap grows larger temporarily

7.2 Using Heap with Map

Purpose:

Combine frequency/count tracking with priority ordering.

Pattern: Frequency-Based Problems

```
// Build frequency map
CREATE frequencyMap
FOR each element in array:
    INCREMENT frequencyMap[element]

// Use heap for top K
CREATE minHeap of size k storing {frequency, element}
FOR each {element, frequency} in frequencyMap:
    INSERT {frequency, element} into minHeap
    IF size > k:
        REMOVE minimum
```

Pattern: Weighted Elements (Dijkstra)

```
CREATE distanceMap
CREATE minHeap storing {distance, node}

INSERT {0, startNode} into minHeap
distanceMap[startNode] = 0

WHILE minHeap not empty:
    {dist, node} = EXTRACT min

    IF dist > distanceMap[node]:
        CONTINUE  // Lazy deletion

    FOR each {neighbor, weight} in graph[node]:
        newDist = dist + weight
        IF newDist < distanceMap[neighbor]:
            distanceMap[neighbor] = newDist
            INSERT {newDist, neighbor} into minHeap
```

7.3 Simulating Max Heap in Min Heap

Technique: Negate Values

```
// Max heap using min heap
CREATE minHeap

// Insert into "max heap"
minHeap.INSERT(-value)

// Get maximum
maxValue = -minHeap.TOP()

// Remove maximum
minHeap.REMOVE()
```

Why This Works:

- Min heap returns smallest value
- Negating makes largest value become smallest
- -∞ becomes +∞ in negated space

When to Use:

- ✅ Language doesn't support max heap natively (Python heapq)
- ✅ Simplify code when you need both min and max with similar logic

⚠️ **Caution:**

- Doesn't work for objects without negation
- Integer overflow: negating INT_MIN causes overflow
- Less readable than using correct heap type

7.4 When NOT to Use Heap

❌ Scenario 1: Full Sorting Required

- **Problem**: Need all elements in sorted order
- **Better Choice**: Sort O(n log n), don't use repeated heap extraction
- **Why**: Sorting has better cache locality and constants

❌ Scenario 2: Median of Fixed Array

- **Problem**: One-time median of static array
- **Better Choice**: Quickselect O(n) average
- **Why**: O(n log n) heap is unnecessary overhead

❌ Scenario 3: Deque Can Solve It

- **Problem**: Sliding window maximum/minimum
- **Better Choice**: Monotonic deque O(n)
- **Why**: Heap is O(n log n) vs deque's O(n)

❌ Scenario 4: Small K

- **Problem**: K=1 or K=2
- **Better Choice**: Linear scan
- **Why**: Overhead not justified

```
// Finding max: O(n) simpler than heap
maxElement = array[0]
FOR each element in array:
    IF element > maxElement:
        maxElement = element
```

❌ Scenario 5: Need Random Access

- **Problem**: Frequent access to middle elements
- **Better Choice**: Balanced BST or Array
- **Why**: Heap only efficient for top element

❌ Scenario 6: K is Large (K ≈ N)

- **Problem**: K close to N
- **Better Choice**: Sort or selection algorithm
- **Why**: O(n log n) vs O(n log k) difference minimal

7.5 Heap vs Sorting Decision Framework

✅ **Use Heap When:**

1. K << N (K significantly smaller than N)
2. Only need top/bottom K elements
3. Data arrives as stream (can't sort upfront)
4. Need repeated access to min/max
5. Intermediate states matter

✅ **Use Sorting When:**

1. K is large (K > N/2)
2. Need all elements in order
3. Need range queries or binary search
4. Data is static (one-time operation)
5. Simplicity matters

Decision Table:

Condition	Recommendation	Time Complexity
Find Kth, K << N, static	Heap or Quickselect	O(n log k) or O(n) avg
Find Kth, K ≈ N/2	Sort	O(n log n)
Find K largest, K << N	Heap	O(n log k)
Find K largest, K = N	Sort	O(n log n)
Stream + Kth element	Heap	O(log k) per element
Sliding window max/min	Deque	O(n)
Merge K sorted	Heap	O(N log k)
Single max/min	Linear scan	O(n)

SECTION 8 – COMMON MISTAKES

❌ Mistake 1: Wrong Heap Type for Kth Element

Wrong:

```
// Want Kth largest, using max heap
CREATE maxHeap
FOR each num in array:
    INSERT num into maxHeap
    IF size > k:
        REMOVE maximum  // WRONG!
RETURN top  // Returns K+1-th largest
```

✅ **Correct:**

```
// Kth largest needs MIN heap
CREATE minHeap of size k
FOR each num in array:
    INSERT num into minHeap
    IF size > k:
        REMOVE minimum
RETURN top  // Kth largest
```

💡 **Remember:** Min heap for K largest, Max heap for K smallest

❌ Mistake 2: Not Checking Empty Before top()

Wrong:

```
value = heap.TOP()  // Runtime error if empty!
heap.REMOVE()
```

✅ Correct:

```
IF heap not empty:
    value = heap.TOP()
    heap.REMOVE()
ELSE:
    Handle empty case
```

❌ Mistake 3: Wrong Size Comparison

Wrong:

```
IF heap.SIZE() >= k:  // Wrong condition
    heap.REMOVE()
```

✅ Correct:

```
IF heap.SIZE() > k:  // Maintain size exactly k
    heap.REMOVE()
```

❌ Mistake 4: Comparator Confusion

Wrong:

```
// Want min heap
comparator(a, b) RETURNS a < b  // Creates MAX heap!
```

✅ Correct:

```
// Min heap: comparator returns true when a has LESS priority
comparator(a, b) RETURNS a > b
```

💡 **Remember:** Comparator defines "less priority", not "less value"

❌ Mistake 5: Integer Overflow in Distance

Wrong:

```
distance = x * x + y * y  // Overflow if x, y large!
```

✅ **Correct:**

```
distance = (long)x * x + (long)y * y
// Or compare squared distances without calculating
```

❌ Mistake 6: Building Heap of Size N Instead of K

Wrong:

```
CREATE minHeap
FOR each num in array:
    INSERT num  // Heap grows to size N

FOR i = 1 to N - k:
    REMOVE minimum
// Time: O(N log N), Space: O(N)
```

✅ **Correct:**

```
CREATE minHeap
FOR each num in array:
    INSERT num
    IF size > k:
        REMOVE minimum
// Time: O(N log K), Space: O(K)
```

❌ Mistake 7: Forgetting Lazy Deletion Cleanup

Wrong:

```
FOR i = 0 to n-1:
    INSERT {value[i], i} into maxHeap
    IF i >= k - 1:
        result = top.value  // May be outdated!
```

✅ **Correct:**

```
FOR i = 0 to n-1:
    INSERT {value[i], i} into maxHeap

    // Clean outdated
    WHILE heap not empty AND top.index <= i - k:
        REMOVE top

    IF i >= k - 1:
        result = top.value
```

❌ Mistake 8: No Tiebreaker for Equal Elements

Wrong:

```
// What if frequencies equal?
comparator(a, b) RETURNS a.frequency > b.frequency
```

✅ **Correct:**

```
comparator(a, b):
    IF a.frequency != b.frequency:
        RETURN a.frequency > b.frequency
    RETURN a.value > b.value  // Tiebreaker
```

❌ Mistake 9: Assuming Heap is Sorted

Wrong Assumption:
Heap elements are in sorted order during construction.

✅ **Reality:**
Heap only guarantees parent-child relationship. Not globally sorted until full extraction.

❌ Mistake 10: Not Handling Edge Cases

Common Edge Cases:

- ✅ Empty input array
- ✅ K == 0 or K > N
- ✅ All elements equal
- ✅ K == 1 (single element)
- ✅ K == N (all elements)
- ✅ Negative numbers
- ✅ Duplicate elements
- ✅ Integer overflow

SECTION 9 – INTERVIEW VIVA

📌 Conceptual Questions (30)

Q1. What is a heap? How does it differ from BST?
A: Heap is a complete binary tree with heap property (parent ≥/≤ children). Unlike BST (left < root < right), heap only cares about parent-child, making it more relaxed and efficient for priority operations.

Q2. Why is heap stored as an array?
A: Complete binary tree allows implicit parent-child via indices: parent `(i-1)/2`, children `2i+1` and `2i+2`. Eliminates pointer overhead and improves cache locality.

Q3. Time complexity of building heap?
A: O(n), not O(n log n). Bottom-up heapify: most nodes near leaves need fewer swaps.

Q4. Can you search efficiently in heap?
A: No, O(n). Heap maintains partial ordering only. Use BST for O(log n) search.

Q5. How to implement max heap using min heap?
A: Negate values: `minHeap.push(-value)`, `max = -minHeap.top()`

Q6. O(n log k) vs O(n log n)?
A: When k << n, O(n log k) is much better. Example: k=10, n=1M: 3.3M vs 20M operations.

Q7. Explain two-heaps for median.
A: Max heap for smaller half, min heap for larger. Balance sizes. Median is top of larger or average of both tops. O(1) median, O(log n) insert.

Q8. When heap instead of sorting?
A: When K << N and only need top K. Heap: O(n log k), Sort: O(n log n). Also for streaming data.

Q9. What is lazy deletion?
A: Mark elements invalid, remove only when they reach top. Used in sliding window where tracking validity is cheaper than deletion.

Q10. Why min heap for K largest?
A: Min heap keeps K largest. Its minimum is Kth largest. New element > minimum replaces it.

Q11. Can heap have duplicates?
A: Yes. Heap property compares parent-child, not siblings.

Q12. Space complexity of heap sort?
A: O(1) auxiliary if in-place. Build heap, swap root with last, heapify.

Q13. How to handle custom objects?
A: Override operator< or provide custom comparator. Comparator defines "less priority".

Q14. Is heap sort stable?
A: No. Heapify can change relative order of equal elements.

Q15. Heap using linked list?
A: Inefficient. Lose O(1) parent/child access (becomes O(n)). Array is superior.

Q16. Inserting sorted data into heap?
A: Still O(log n) per insertion. Tree remains balanced but not optimal. Total O(n log n) vs O(n) bottom-up.

Q17. Merge two heaps efficiently?
A: No efficient merge for binary heaps - O(n+m). Binomial/Fibonacci heaps have O(log n) merge.

Q18. Heapify up vs down?
A:

- **Up**: After insertion, swap with parent until valid. Used in insert.
- **Down**: After deletion, swap with child until valid. Used in delete.

Q19. What is d-ary heap?
A: Each node has d children. Reduces height to log_d(n). Useful when extractions >> insertions.

Q20. Can heap sort in O(n)?
A: No. Building O(n) + extraction O(n log n) = O(n log n). Can't beat comparison lower bound.

Q21. Priority queue vs heap?
A: Priority queue is interface (operations), heap is implementation. Like List vs ArrayList.

Q22. Find kth smallest in max heap?
A: No efficient way. Better: quickselect O(n) or maintain min heap of size k.

Q23. Two heaps invariant?
A:

1. `|maxHeap.size - minHeap.size| ≤ 1`
2. All maxHeap ≤ all minHeap
3. Median from tops

Q24. Why heap for Dijkstra?
A: Only need min extraction, not searching. Heap better constants and cache vs BST.

Q25. Heap height with n elements?
A: $\lfloor \log_2(n) \rfloor$. Complete tree ensures balanced.

Q26. Decrease-key in C++ priority_queue?
A: No built-in. Must re-insert (lazy deletion) or use custom heap/set.

Q27. Handle overflow in distance?
A: Use long long or compare squared distances without sqrt.

Q28. Heap and complete binary tree?
A: Heap is always complete (all levels full except last). Enables array representation and O(log n) height.

Q29. Equal priorities in heap?
A: Unspecified order. Add tiebreaker if needed (timestamp, index).

Q30. Internal vs external sorting?
A: Internal: data in memory. External: data on disk, processed in chunks. Heaps used for k-way merge in external sort.

🔥 Tricky Edge Case Questions (10)

Q31. K > array size?
A: Validate input. Options: return all elements, return empty, or clamp K to array.size().

Q32. Handle negative numbers?
A: Heaps work fine. Watch for negation overflow (INT_MIN), distance calculations need abs().

Q33. All elements equal?
A: Valid. Any K are correct. Heap works but no benefit over linear scan.

Q34. K = 0?
A: Return empty array. Check edge case.

Q35. Median with 1 element?
A: Handle initialization: single element is median.

**