Heap

Pattern: Heap / Priority Queue (Min-Heap / Max-Heap)

How to Recognize

- You're asked to find **top K elements**, **kth smallest/largest**, or **median**.
- There's a need to maintain a **running order** or **priority** among elements.
- The problem involves **frequent insertions and deletions** of extremes (min/max).
- Often paired with **sorting**, **frequency counting**, or **streaming data**.

Step-by-Step Thinking Process (Template)

1. **Identify what you want to track**: e.g., k largest, k closest, top frequent.
2. **Choose the right heap type**:
    - Min-heap: keep smallest k elements → pop when size > k
    - Max-heap: keep largest k elements → use negative values in Python (min-heap trick)
3. **Use a heap of size K** to maintain only relevant candidates.
4. **Pop or push based on comparison logic**.
5. **Extract result** after processing all inputs (e.g., return root or sort remaining).

Common Pitfalls & Edge Cases

- Forgetting that Python `heapq` is a **min-heap only** → use negative values for max-heap.
- Not limiting heap size → leads to O(n log n) instead of O(k log n).
- Incorrectly handling ties (e.g., in "Top K Frequent Words", tie-breaking by lexicographic order).
- Empty input → handle early return.

---

1. K Closest Points to Origin

Problem Summary

Given an array of points in 2D space, return the `k` closest points to the origin `(0, 0)` based on Euclidean distance.

Pattern

- **Heap / Priority Queue** (max-heap of size k)
- Alternative: **Sorting** (but less efficient for large datasets)

Solution with Inline Comments

```python
import heapq
from typing import List, Tuple

def kClosest(points: List[List[int]], k: int) -> List[List[int]]:
    # Use a max-heap to store the k closest points
    # We store (-distance, point) so that the farthest
    # (largest distance) is at top
    # Negative distance ensures we simulate max-heap behavior using min-heap
    heap = []

    for x, y in points:
        # Calculate squared distance (avoid sqrt for speed & precision)
        dist = x*x + y*y

        # If heap has fewer than k elements, add current point
        if len(heap) < k:
            heapq.heappush(heap, (-dist, [x, y]))
        # Else,if current point is closer than the farthest in heap,replace it
        elif dist < -heap[0][0]:  # -heap[0][0] is the max distance in heap
            heapq.heappop(heap)
            heapq.heappush(heap, (-dist, [x, y]))

    # Extract points from heap (they are in no particular order)
    return [point for _, point in heap]
```

```python
# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: points = [[1,3],[-2,2]], k = 1
    points = [[1, 3], [-2, 2]]
    k = 1

    # Call function
    result = kClosest(points, k)

    # Expected Output: [[-2,2]]
    # Because distance of (1,3): 1+9=10; (-2,2): 4+4=8 → (-2,2) is closer
    print("Output:", result)  # Output: [[-2, 2]]
```

Example Walkthrough

- Input: `points = [[1,3],[-2,2]], k = 1`
- Process (1,3): dist $= 1^2 + 3^2 = 10 \rightarrow$ heap = [(-10, [1,3])]
- Process (-2,2): dist $= 4 + 4 = 8 \rightarrow 8 < 10 \rightarrow$ pop (-10,...), push (-8, [-2,2])
- Final heap: [(-8, [-2,2])] $\rightarrow$ return `[[ -2, 2 ]]`

Complexity

- **Time**: O(n log k) — each insertion/removal takes O(log k), done n times
- **Space**: O(k) — heap stores at most k elements

---

2. Find Median from Data Stream

Problem Summary

Design a data structure that supports adding integers and finding the median of all added numbers dynamically.

Pattern

- **Two Heaps**: Max-heap for left half, Min-heap for right half
- Balance sizes: difference 1
- Median = top of larger heap or average of both

Solution with Inline Comments

```python
import heapq

class MedianFinder:
    def __init__(self):
        # Max-heap for smaller half (store negative values)
        self.small = []  # represents left half (max-heap via negatives)
        # Min-heap for larger half
        self.large = []  # represents right half (min-heap)

    def addNum(self, num: int) -> None:
        # Push to small (max-heap) first
        heapq.heappush(self.small, -num)

        # Ensure every number in small <= every number in large
        # If top of small > top of large, swap
        if self.small and self.large and (-self.small[0]) > self.large[0]:
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)

        # Balance the heaps: difference should be at most 1
        if len(self.small) > len(self.large) + 1:
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)
        elif len(self.large) > len(self.small) + 1:
            val = heapq.heappop(self.large)
            heapq.heappush(self.small, -val)

    def findMedian(self) -> float:
        # If heaps are same size, median is average
        if len(self.small) == len(self.large):
            return (-self.small[0] + self.large[0]) / 2.0
        # Else, median is top of larger heap
        elif len(self.small) > len(self.large):
            return -self.small[0]
        else:
            return self.large[0]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Usage:
    mf = MedianFinder()
    mf.addNum(1)
    mf.addNum(2)
    print("Median after [1,2]:", mf.findMedian())  # Output: 1.5

    mf.addNum(3)
    print("Median after [1,2,3]:", mf.findMedian())  # Output: 2.0
```

Example Walkthrough

We'll go through this sequence:

```
mf = MedianFinder()
mf.addNum(1)
mf.addNum(2)
print(mf.findMedian())  # 1.5
mf.addNum(3)
print(mf.findMedian())  # 2.0
```

Step 1: `addNum(1)`

- Push `-1` into `small` → `small = [-1]`, `large = []`
- No need to compare since `large` is empty.
- Size check:
  - `len(small) = 1`, `len(large) = 0` → difference is 1 → acceptable.

Final state: - `small = [-1]` (i.e., contains 1) - `large = []`

Step 2: `addNum(2)`

- Push `-2` into `small` → `small = [-2, -1]` (min-heap of negatives → top is -2 → actual value is 2)
- Now check: is `top(small)` > `top(large)`?
  - But `large` is still empty → skip comparison.
- Balance sizes:

  - `len(small) = 2`, `len(large) = 0` → difference is 2 (>1), so move one element.
  - Pop from `small`: `val = -heapq.heappop(self.small)` → pop `-2`, so `val = 2`
  - Push 2 into `large`: `large = [2]`
  - Now `small = [-1]`, `large = [2]`

Final state: - `small = [-1]` → {1} - `large = [2]` → {2}

Now both heaps differ in size by only 1 → good.

Step 3: `findMedian()` → after adding [1,2]

- `len(small) == 1`, `len(large) == 1` → equal sizes
- Median = `(-self.small[0] + self.large[0]) / 2.0`
  - `-self.small[0] = -(-1) = 1`
  - `self.large[0] = 2`
  - Median = `(1 + 2) / 2 = 1.5`

Output: `1.5` → Correct

Step 4: `addNum(3)`

- Push `-3` into `small` → `small = [-3, -1]` → top is `-3` → value is 3
- Check: is `top(small)` > `top(large)`?
  - `-self.small[0] = 3`, `self.large[0] = 2`
  - Is `3 > 2`? Yes → need to fix!
- So:
  - Pop from `small`: `val = -heapq.heappop(self.small)` → pop `-3  val = 3`
  - Push 3 into `large`: now `large = [2, 3]` → min-heap: `[2, 3]`
  - Now `small = [-1]`, `large = [2, 3]`
- Recheck size balance:
  - `len(small) = 1`, `len(large) = 2` → difference is 1 → acceptable

Final state: - `small = [-1]` → {1} - `large = [2, 3]` → {2, 3}

Step 5: `findMedian()` → after [1,2,3]

- `len(small) = 1`, `len(large) = 2` → not equal
- Since `large` has more elements → median is `large[0] = 2`

Output: `2.0` → Correct

Summary of States

Operation	`small` (max-heap)	`large` (min-heap)	Median
addNum(1)	[-1]	[]	—
addNum(2)	[-1]	[2]	—
findMedian()	[-1]	[2]	1.5
addNum(3)	[-1]	[2, 3]	—
findMedian()	[-1]	[2, 3]	2.0

Complexity

- **addNum**: O(log n) — heap operations
- **findMedian**: O(1)
- **Space**: O(n)

---

3. Merge k Sorted Lists

Problem Statement

You are given an array of `k` linked lists, each of which is sorted in ascending order.

Merge all the lists into one sorted linked list and return it.

> **Constraints:** - `k == lists.length` - `0 <= k <= 10^4` - `0 <= lists[i].length <= 500` - `-10^4 <= lists[i][j] <= 10^4` - All lists are sorted in non-decreasing order.

Example:

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
```

Solution: Min-Heap Approach (Optimal)

```python
import heapq
from typing import List, Optional

# Definition for singly-linked list.
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next

class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]):
        # Step 1: Create a min-heap to store (value, index, node)
        heap = []

        # Step 2: Push the head of each non-empty list into the heap
        for i, lst in enumerate(lists):
            if lst:  # Skip empty lists
                heapq.heappush(heap, (lst.val, i, lst))

        # Step 3: Dummy node to simplify result construction
        dummy = ListNode(0)
        current = dummy

        # Step 4: While heap is not empty, extract smallest and add to result
        while heap:
            # Pop the smallest element: (val, index, node)
            val, idx, node = heapq.heappop(heap)

            # Add this node to the result list
            current.next = node
            current = current.next

            # If the node has a next, push it back into the heap
            if node.next:
                heapq.heappush(heap, (node.next.val, idx, node.next))

        # Step 5: Return the merged list (skip dummy head)
        return dummy.next
```

Code Walkthrough with Example

Input:

```
lists = [
    [1 → 4 → 5],    # list 0
    [1 → 3 → 4],    # list 1
    [2 → 6]         # list 2
]
```

We'll simulate every step.

Initial Setup

- `heap = []`
- `dummy = ListNode(0)`
- `current = dummy`

Step 1: Push Heads into Heap

Loop over `lists`:

`i`	`lst`	`lst.val`	Pushed?	Tuple
0	`nodeA` (1)	1	Yes	`(1, 0, nodeA)`

i	lst	lst.val	Pushed?	Tuple
1	nodeD (1)	1	Yes	(1, 1, nodeD)
2	nodeG (2)	2	Yes	(2, 2, nodeG)

Now:

```
heap = [(1, 0, nodeA), (1, 1, nodeD), (2, 2, nodeG)]
```

Heap maintains min-heap property: smallest value first. Ties broken by index.

Iteration 1: Pop `(1, 0, nodeA)`

- `val = 1`, `idx = 0`, `node = nodeA`
- `current.next = nodeA` → result: `[1]`
- `current = nodeA`
- `nodeA.next = nodeB` → push `(4, 0, nodeB)`

Heap now:

```
[(1, 1, nodeD), (2, 2, nodeG), (4, 0, nodeB)]
```

Iteration 2: Pop `(1, 1, nodeD)`

- Add `nodeD` → result: `[1, 1]`
- Push `nodeE` (value 3) → `(3, 1, nodeE)`

Heap:

```
[(2, 2, nodeG), (3, 1, nodeE), (4, 0, nodeB)]
```

Iteration 3: Pop `(2, 2, nodeG)`

- Add `nodeG` → result: `[1, 1, 2]`
- Push `nodeH` (value 6) → `(6, 2, nodeH)`

Heap:

```
[(3, 1, nodeE), (4, 0, nodeB), (6, 2, nodeH)]
```

Iteration 4: Pop `(3, 1, nodeE)`

- Add `nodeE` → result: `[1, 1, 2, 3]`
- Push `nodeF` (value 4) → `(4, 1, nodeF)`

Heap:

```
[(4, 0, nodeB), (4, 1, nodeF), (6, 2, nodeH)]
```

> Two 4s → pick `(4, 0, nodeB)` because `0 < 1`

Iteration 5: Pop `(4, 0, nodeB)`

- Add `nodeB` → result: `[1, 1, 2, 3, 4]`
- Push `nodeC` (value 5) → `(5, 0, nodeC)`

Heap:

```
[(4, 1, nodeF), (5, 0, nodeC), (6, 2, nodeH)]
```

Iteration 6: Pop `(4, 1, nodeF)`

- Add `nodeF` → result: `[1, 1, 2, 3, 4, 4]`
- No `next` → don't push

Heap:

```
[(5, 0, nodeC), (6, 2, nodeH)]
```

Iteration 7: Pop `(5, 0, nodeC)`

- Add `nodeC` → result: `[1, 1, 2, 3, 4, 4, 5]`
- No `next` → stop

Heap:

```
[(6, 2, nodeH)]
```

Iteration 8: Pop `(6, 2, nodeH)`

- Add `nodeH` → result: `[1, 1, 2, 3, 4, 4, 5, 6]`
- Done!

Final Result

Return `dummy.next` → the full merged list:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

Output: `[1,1,2,3,4,4,5,6]`

Time & Space Complexity

Metric	Complexity	Explanation
Time	`O(n log k)`	- `n` = total number of nodes- Each node is pushed and popped once → `n` operations- Each heap op takes `O(log k)`- Total: `O(n log k)`
Space	`O(k)`	- Heap stores at most `k` elements (one per list)- `dummy` and pointers: `O(1)`- Output list not counted

Efficient even for large `k` (e.g., 1000 lists)

Key Takeaways (Revision Notes)

Concept	Why It Matters
Min-Heap	Always gives smallest current head in `O(log k)`
Tuple `(val, idx, node)`	Prevents errors when values are equal; tiebreaker via `idx`
Dummy Node	Avoids edge cases when building result list
One Pass Per Node	Each node inserted and removed exactly once
Heap Size k	Memory efficient — only keeps front nodes

Pro Tips for Interviews

1. **Always use `(val, idx, node)`** — never compare `ListNode` directly!
2. **Handle empty lists** (`if lst`) — avoid `None` errors.
3. **Use `heapq`** — Python's built-in min-heap is perfect.
4. **Draw the heap evolution** on paper during interviews.
5. **Explain the logic clearly**: "We always pick the smallest available head."

---

4. Task Scheduler

Problem Statement

You are given a characters array `tasks` representing tasks labeled from `'A'` to `'Z'`. Each task takes **one unit of time** to complete. You are also given an integer `n`, which represents the **cooldown period** between two identical tasks.

You can only perform one task at a time. After performing a task, you must wait **exactly `n` units** before performing the same task again.

Return the **minimum number of time units** needed to complete all tasks.

> **Constraints:** - `1 <= tasks.length <= 10` - `tasks[i]` is an uppercase English letter. - `0 <= n <= 100`

Sample Input & Output

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8

Explanation:
One valid schedule is:
Time:   0   1   2   3   4   5   6   7
       [A,  B,  _,  A,  B,  _,  A,  B]
```

> All `A`s are separated by 2 units → valid
> All `B`s are separated by 2 units → valid
> Total time = 8 units

Solution Code

```python
from collections import Counter
from typing import List

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        """
        Find the minimum time to schedule all tasks with cooldown n.

        :param tasks: List of task labels (e.g., ['A','A','B','B'])
        :param n: Cooldown period between identical tasks
        :return: Minimum time units required

        Intuition:
        - The most frequent task(s) determine the minimum time.
        - We need to place max_freq tasks with
        - at least n other tasks/idle in between.
        - If there are enough other tasks to fill gaps → no idle time needed.
        """

        # Step 1: Count frequency of each task
        task_count = Counter(tasks)

        # Step 2: Find the maximum frequency
        max_freq = max(task_count.values())

        # Step 3: Count how many tasks have the maximum frequency
        count_max_freq = sum(
            1 for freq in task_count.values() if freq == max_freq
            )

        # Step 4: Calculate minimum time using formula
        # (max_freq - 1) full cycles of length (n + 1) each
        # Plus the final instances of the most frequent tasks
        min_time = (max_freq - 1) * (n + 1) + count_max_freq

        # Step 5: Return the maximum between min_time and total tasks
        # Why? If there are enough other tasks to fill all slots,
        # we don't need idle time → answer is just len(tasks)
        return max(min_time, len(tasks))
```

Step-by-Step Walkthrough (Example)

Input:

```
tasks = ["A", "A", "A", "B", "B", "B"]
n = 2
```

Step 1: Count Frequencies

```
task_count = {'A': 3, 'B': 3}
```

Step 2: Max Frequency

```
max_freq = 3
```

Step 3: Count Tasks with Max Frequency

```
count_max_freq = 2  # Both A and B appear 3 times
```

Step 4: Apply Formula

```
min_time = (3 - 1) * (2 + 1) + 2
         = 2 * 3 + 2
         = 6 + 2
         = 8
```

Step 5: Compare with Total Tasks

```
len(tasks) = 6
return max(8, 6) = 8
```

Answer: 8

Valid Schedule Visualization

Time	Task
0	A
1	B
2	_
3	A
4	B
5	_
6	A
7	B

Final output: `8` time units

Check cooldowns: - `A` at 0 → next at 3 → gap = 3 3 → valid - `B` at 1 → next at 4 → gap = 3 3 → valid - `A` at 3 → next at 6 → gap = 3 3 → valid - `B` at 4 → next at 7 → gap = 3 3 → valid

All constraints satisfied!

Time Complexity

- **O(m)** where `m = len(tasks)`
  - One pass to build `Counter`: O(m)
  - Finding `max()` and counting frequencies: O(26) O(1)
- So overall: **O(m)**

Space Complexity

- **O(1)** because:
  - The `Counter` stores at most 26 keys (letters A–Z)
  - No additional data structures grow with input size
- Thus, space used is constant regardless of input size

Key Takeaways

- The bottleneck is the **most frequent task**.
- Use the formula: `(max_freq - 1) * (n + 1) + count_max_freq`
- Only add idle time if needed — otherwise, use `len(tasks)`

---

5. Top K Frequent Words

Problem :

We want to find the `k` most frequent words in a list, with ties broken by **lexicographical (dictionary) order**.

```
words = ["the","day","is","sunny","the","the","the","sunny","is","is"]
k = 4
print(topKFrequent(words, k))
# Output: ["the", "is", "sunny", "day"]
```

Solution (Python):

```python
import heapq
from collections import Counter

def topKFrequent(words, k):
    # Step 1: Count frequency of each word
    count = Counter(words)

    # Step 2: Create a min-heap (or use negative frequency for max behavior)
    heap = [(-freq, word) for word, freq in count.items()]
    heapq.heapify(heap)

    # Step 3: Pop the top k elements
    return [heapq.heappop(heap)[1] for _ in range(k)]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: words = ["i","love","leetcode","i","love","coding"],k = 2
    words = ["i", "love", "leetcode", "i", "love", "coding"]
    k = 2

    # Call function
    result = topKFrequent(words, k)

    # Expected Output: ["i","love"]
    # i:2, love:2, coding:1 → top 2 → i and
    # love (tie broken by lex order: i < love)
    print("Output:", result)  # Output: ['i', 'love']
```

Step-by-Step Breakdown

Step 1: Count Frequencies

```
count = Counter(words)
```

- `Counter` is a subclass of `dict` that counts occurrences.
- Example:

  ```
  words = ["i","love","leetcode","i","love","coding"]
  count = Counter(words)
  # count = {'i': 2, 'love': 2, 'leetcode': 1, 'coding': 1}
  ```

Step 2: Build a List for the Heap

```
heap = [(-freq, word) for word, freq in count.items()]
```

- We create a list of tuples: `(-frequency, word)`
- We use **negative frequency** because:
  - Python's `heapq` is a **min-heap** by default.
  - To simulate a **max-heap** for frequency, we negate the frequency.
  - So higher actual frequency becomes more negative → smaller in min-heap → comes out first.

Example:

From our `count`, this gives:

```
[(-2, 'i'), (-2, 'love'), (-1, 'leetcode'), (-1, 'coding')]
```

Now, when we heapify, the **smallest tuple** (by first element, then second) will be at the top.

But here's the **key insight**:
When two frequencies are the same (e.g., `-2`), Python compares the **second element**, which is the word.

So: - `(-2, 'i')` vs `(-2, 'love')` → `'i' < 'love'` lexicographically → `(-2, 'i')` is smaller.
- But we want **higher frequency first**, and **lexicographically smaller word first** in the result.

Wait — doesn't that mean `'i'` should come before `'love'`? Yes.

But in the heap, since `(-2, 'i')` is smaller than `(-2, 'love')`, it will be **popped first** — which is exactly what we want.

So the tuple `(-freq, word)` naturally gives us: - Higher frequency first (because of `-freq`)
- Lexicographically smaller word first in case of tie

This is why the tuple ordering works perfectly.

Step 3: Heapify the List

```
heapq.heapify(heap)
```

- Converts the list into a **min-heap** in-place.
- The smallest element (i.e., highest frequency, then lexicographically smallest) is at the top.

After heapify, the internal structure maintains heap property: - `heap[0]` is always the smallest (i.e., the "best" candidate).

But note: The entire list is **not sorted** — just heap-ordered.

Step 4: Extract Top k Elements

```
return [heapq.heappop(heap)[1] for _ in range(k)]
```

- We pop `k` times.
- Each `heappop()` removes and returns the smallest (i.e., most frequent, or lexicographically smaller) element.
- We take `[1]` → the **word** part of the tuple `(-freq, word)`.

Each pop takes **O(log n)** time, so k pops → **O(k log n)**.

Example Walkthrough

Let's run through:

```
words = ["i","love","leetcode","i","love","coding"]
k = 2
```

Step 1: Count

```
count = {'i': 2, 'love': 2, 'leetcode': 1, 'coding': 1}
```

Step 2: Build heap list

```
heap = [(-2, 'i'), (-2, 'love'), (-1, 'leetcode'), (-1, 'coding')]
```

Step 3: heapify

After `heapify`, the heap is reordered so that: - `(-2, 'i')` is at the top (smallest), because `'i' < 'love'` lexicographically.

So the heap order ensures: 1. `(-2, 'i')` 2. `(-2, 'love')` 3. `(-1, ...)` etc.

Step 4: Pop k=2 times

- **1st pop**: `(-2, 'i')` → append `'i'`
- **2nd pop**: `(-2, 'love')` → append `'love'`

Result: `["i", "love"]`

Why Doesn't Lex Order Mess It Up?

Suppose we had:

```
words = ["love", "i", "i", "love"]  # same frequencies
```

Then: - `(-2, 'i')` and `(-2, 'love')` - `'i' < 'love'` → so `(-2, 'i')` is smaller → popped first → correct.

So **the natural tuple comparison handles the tie-break correctly**.

Time & Space Complexity

Aspect	Complexity
Time	O(n + k log n) - O(n) for counting - O(n) for heapify - O(k log n) for popping k times
Space	O(n) for counter and heap

Final Notes

- The **tuple `(-freq, word)`** is the magic key.
- Python's **lexicographic comparison of strings in tuples** makes tie-breaking automatic.
- `heapq` only supports min-heap, so we **negate frequency** to simulate max behavior.

Let me know if you'd like to see the **bucket sort** version (O(n) time) too!

6. Find K Closest Elements

Problem Summary

Given a sorted array and integer `k`, return the `k` closest elements to a target value `x`. Return them in ascending order.

Pattern

- **Binary Search on Answer** (find left boundary of result window)
- **Two Pointers** (after finding start, expand outward)
- Or: **Sliding Window** on sorted array

Solution with Inline Comments

```python
from typing import List

def findClosestElements(arr: List[int], k: int, x: int) -> List[int]:
    # Use binary search to find the leftmost starting index of k elements
    left, right = 0, len(arr) - k  # right is len-k because we need k elements

    while left < right:
        mid = (left + right) // 2

        # Compare the distances from mid and mid+k to x
        # If arr[mid] is farther than arr[mid+k],
        # then mid cannot be the left bound
        # Because we'd get better elements by moving right
        if x - arr[mid] > arr[mid + k] - x:
            left = mid + 1
        else:
            right = mid

    # Now left is the starting index of the k closest elements
    return arr[left:left + k]


# ---- Official LeetCode Example ----
if __name__ == "__main__":
    # Example Input: arr = [1,2,3,4,5], k = 4, x = 3
    arr = [1, 2, 3, 4, 5]
    k = 4
    x = 3

    # Call function
    result = findClosestElements(arr, k, x)

    # Expected Output: [1,2,3,4]
    # Distances: |1-3|=2, |2-3|=1, |3-3|=0, |4-3|=1, |5-3|=2
    # Closest 4: 2,3,4,2 → but 1,2,3,4 are closer than 5
    print("Output:", result)  # Output: [1, 2, 3, 4]
```

Example Walkthrough

Example Input

```
arr = [1, 2, 3, 4, 5]
k = 4
x = 3
```

We want the **4 closest elements** to `3`.

Step-by-Step Walkthrough

Step 1: Initial Setup

```
left = 0
right = len(arr) - k = 5 - 4 = 1
```

So our binary search range is `[0, 1)` → only possible values for `left` are 0 or 1.

We are trying to find the **starting index** of a subarray of length `k=4` that contains the closest elements to `x=3`.

Possible windows: - Start at 0 → `[1,2,3,4]` - Start at 1 → `[2,3,4,5]`

We'll use binary search to pick the best one.

Binary Search Loop

Iteration 1:

```
left = 0, right = 1
mid = (0 + 1) // 2 = 0
```

Now compare: - `x - arr[mid]` → distance from `x` to left end of window - `arr[mid + k] - x` → distance from `x` to right end of window

Why this comparison?
Because we're comparing two overlapping windows: - One starting at `mid = 0`: `[1,2,3,4]` - One starting at `mid + 1 = 1`: `[2,3,4,5]`

We decide which one is better by comparing the **outer edges**: `arr[mid]` vs `arr[mid + k]`.

> If `arr[mid + k]` is closer to `x`, then we should move the window right → discard current `mid`.

Let's compute:

```
x - arr[mid] = 3 - arr[0] = 3 - 1 = 2
arr[mid + k] - x = arr[0 + 4] - 3 = arr[4] - 3 = 5 - 3 = 2
```

So:

```
if 2 > 2 → False
```

So we go to `else`:

```
right = mid = 0
```

Now `left = 0`, `right = 0` → loop ends.

Final Result

```
return arr[left : left + k] = arr[0:4] = [1, 2, 3, 4]
```

Why [1,2,3,4] and not [2,3,4,5]?

Let's compute distances to `x = 3`:

Element	Distance
1	
2	
3	
4	
5	

Top 4 smallest distances: all except one of the 2s.

But both `1` and `5` are equally distant from `3`. Since `1 < 5`, we prefer `1`. So we pick `[1,2,3,4]`.

This matches our result.

Key Insight of the Algorithm

Instead of comparing individual elements, we compare **candidate windows** of size `k`.

At each `mid`, we consider: - Window starting at `mid`: includes `arr[mid]` to `arr[mid + k - 1]` - The next window would start at `mid + 1`

To decide whether to move right, we compare: - `x - arr[mid]` → how far the **leftmost element** of current window is from `x` - `arr[mid + k] - x` → how far the **next element after the window** is from `x`

If the next element (`arr[mid+k]`) is **closer** than the current leftmost (`arr[mid]`), we should shift the window right.

Hence:

```
if x - arr[mid] > arr[mid + k] - x:
    left = mid + 1   # shift window right
else:
    right = mid      # keep current left or go left
```

Complexity

- **Time**: O(log(n - k)) — binary search over `n - k` positions
- **Space**: O(1) — only indices used

---

7. Kth Largest Element in an Array

Problem

Given an array `nums` and integer `k`, find the **kth largest element**.

> Example:
> `nums = [3,2,1,5,6,4]`, `k = 2` → return `5` (since 5 is the 2nd largest)

Why Use a Min-Heap?

We want the **kth largest**, so we only need to keep track of the **top k largest elements**.

Code

```python
import heapq

class Solution:
    def findKthLargest(self, nums: list[int], k: int) -> int:
        # Min-heap to store the k largest elements
        heap = []

        for num in nums:
            if len(heap) < k:
                # If we have space, add the number
                heapq.heappush(heap, num)
            elif num > heap[0]:
                # If current number is bigger than the smallest in heap,
                # replace the smallest with this one
                heapq.heapreplace(heap, num)
        # The root of the min-heap is the kth largest
        return heap[0]
```

Step-by-Step Walkthrough with `nums = [3,2,1,5,6,4]`, `k = 2`

```
heap = []  # min-heap
```

1. **num = 3**
   - len(heap) = 0 < 2 → push 3
   - heap = [3]
2. **num = 2**
   - len(heap) = 1 < 2 → push 2
   - heap = [2, 3] (heap property: min at front)
3. **num = 1**
   - len(heap) = 2 → not less than k
   - Is 1 > heap[0]? → 1 > 2? No → skip
4. **num = 5**
   - len(heap) = 2 → check if 5 > 2 → Yes
   - Replace: `heapreplace(heap, 5)` → removes 2, adds 5
   - heap = [3, 5] → now min is 3
5. **num = 6**
   - 6 > 3 → Yes
   - `heapreplace(heap, 6)` → removes 3, adds 6
   - heap = [5, 6] → min is 5
6. **num = 4**
   - 4 > 5? No → skip

Final heap: `[5, 6]` → `heap[0] = 5` → return **5**

Time & Space Complexity

Metric	Complexity	Explanation
Time	O(n log k)	For each of n elements: heap operation takes O(log k)
Space	O(k)	Heap stores at most k elements

Efficient when **k is small** compared to n (e.g., k = 10, n = 10000)

Pro Tips

- Use `heapq.heapreplace()` instead of `heappop()` + `heappush()` for efficiency.
- Always compare `num > heap[0]` — not `>=`, because duplicates are allowed.
- This method works even if there are duplicate values.

  Example: `nums = [1,1,1,2,2]`, `k = 3` → 3rd largest is `1` → correct.

---

8. Smallest Range Covering Elements from K Lists

Problem Statement:

You are given `k` sorted integer arrays. You need to find the **smallest range** that includes **at least one number from each array**.

The range is defined as `[start, end]`, and its **size** is `end - start`.

Return the **smallest such range**. If multiple ranges have the same size, return any one of them.

Example:

```
Input: nums = [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]
Output: [20,24]
```

Explanation: The range `[20,24]` covers: - 20 from the second list, - 24 from the first list, - 22 from the third list.

All lists are covered, and it's the smallest possible range.

Key Insight:

We want to minimize the difference (`end - start`) while ensuring that **each of the k lists contributes at least one element** in the range.

A greedy + heap approach works well here.

Python Implementation:

```python
import heapq
from typing import List

class Solution:
    def smallestRange(self, nums: List[List[int]]) -> List[int]:
        # Min-heap to store (value, list_index, index_in_list)
        heap = []
        max_val = float('-inf')

        # Initialize: add the first element from each list
        for i in range(len(nums)):
            heapq.heappush(heap, (nums[i][0], i, 0))
            max_val = max(max_val, nums[i][0])

        # Initialize result range
        best_start, best_end = float('-inf'), float('inf')

        while heap:
            min_val, list_idx, idx_in_list = heapq.heappop(heap)

            # Update the best range if current range is smaller
            if max_val - min_val < best_end - best_start:
                best_start, best_end = min_val, max_val

            # Move to next element in the same list
            if idx_in_list + 1 < len(nums[list_idx]):
                next_val = nums[list_idx][idx_in_list + 1]
                heapq.heappush(heap, (next_val, list_idx, idx_in_list + 1))
                max_val = max(max_val, next_val)
            else:
                # One list is exhausted; we can't form a valid range anymore
                break

        return [best_start, best_end]
```

Complexity Analysis:

- **Time Complexity:**
  O(N log k), where N is the total number of elements across all lists, and k is the number of lists.
  Each element is pushed and popped once from the heap (log k per operation).
- **Space Complexity:**
  O(k) for the heap (stores one element per list at a time).

Why This Works:

- We always maintain one element from each list (initially), then replace the smallest one with the next in its list.
- By doing this, we ensure we never skip a potentially better range.
- The heap ensures we always process the smallest current element, which helps shrink the range.

Example walkthrough

We'll use this example:

```
nums = [
    [4, 10, 15, 24, 26],   # List 0
    [0, 9, 12, 20],        # List 1
    [5, 18, 22, 30]         # List 2
]
```

Line-by-Line Walkthrough (With Visuals & Tracing)

Let's now go **step-by-step**, updating variables at every stage.

Step 1: Initialize `heap` and `max_val`

```
heap = []
max_val = float('-inf')  # -∞
```

Now loop over each list (`i = 0, 1, 2`):

i = 0: List 0 → element = `4`

- Push `(4, 0, 0)` into heap
- `max_val = max(-∞, 4) = 4`

Heap: `[(4, 0, 0)]`

i = 1: List 1 → element = `0`

- Push `(0, 1, 0)` into heap
- `max_val = max(4, 0) = 4`

Heap: `[(0, 1, 0), (4, 0, 0)]` → min-heap sorted: `[0, 4]`

i = 2: List 2 → element = `5`

- Push `(5, 2, 0)` into heap
- `max_val = max(4, 5) = 5`

Heap: `[(0, 1, 0), (4, 0, 0), (5, 2, 0)]` → sorted by value

After initialization: - `heap = [(0, 1, 0), (4, 0, 0), (5, 2, 0)]` - `max_val = 5` - `best_start = -∞`, `best_end = ∞`

> This window: `{0 (list1), 4 (list0), 5 (list2)}` → covers all lists!

Step 2: Set `best_start`, `best_end`

```
best_start, best_end = float('-inf'), float('inf')
```

So far, no valid range → we'll update it when we find a better one.

Step 3: Start the `while heap:` Loop

We process the heap until it's empty or a list runs out.

Let's trace each iteration.

Iteration 1: Pop `(0, 1, 0)`

```
min_val, list_idx, idx_in_list = heapq.heappop(heap)
# → min_val = 0, list_idx = 1, idx_in_list = 0
```

Now check:

```
if max_val - min_val < best_end - best_start:
    # 5 - 0 = 5 < ∞ - (-∞) → True
    best_start, best_end = 0, 5
```

Update best range: `[0, 5]` (size = 5)

Now try to advance list 1:

```
if idx_in_list + 1 < len(nums[1]):  # 0+1=1 < 4 → True
    next_val = nums[1][1] = 9
    heapq.heappush(heap, (9, 1, 1))
    max_val = max(5, 9) = 9
```

New heap: `[(4, 0, 0), (5, 2, 0), (9, 1, 1)]`
→ Sorted: `[4, 5, 9]`

Now window: `{4, 5, 9}` → min=4, max=9 → range=5

Iteration 2: Pop `(4, 0, 0)`

```
min_val = 4, list_idx = 0, idx_in_list = 0
```

Check:

```
if 9 - 4 = 5 < 5 - 0 = 5? → No (5 < 5 is False)
```

No update.

Advance list 0:

```
if 0+1=1 < 5 → True
next_val = nums[0][1] = 10
push (10, 0, 1)
max_val = max(9, 10) = 10
```

Heap: `[(5, 2, 0), (9, 1, 1), (10, 0, 1)]` → sorted: `[5, 9, 10]`

Window: `{5, 9, 10}` → range = 5

Iteration 3: Pop `(5, 2, 0)`

```
min_val = 5, list_idx = 2, idx_in_list = 0
```

Check:

```
10 - 5 = 5 < 5 → False → no update
```

Advance list 2:

```
1 < 4 → True
next_val = nums[2][1] = 18
push (18, 2, 1)
max_val = max(10, 18) = 18
```

Heap: `[(9, 1, 1), (10, 0, 1), (18, 2, 1)]` → `[9, 10, 18]`

Window: `{9, 10, 18}` → range = 9

Iteration 4: Pop `(9, 1, 1)`

```
min_val = 9, list_idx = 1, idx_in_list = 1
```

Check:

```
18 - 9 = 9 < 5? → No → skip
```

Advance list 1:

```
1+1=2 < 4 → True
next_val = nums[1][2] = 12
push (12, 1, 2)
max_val = max(18, 12) = 18
```

Heap: `[(10, 0, 1), (12, 1, 2), (18, 2, 1)]` $\rightarrow$ `[10, 12, 18]`

Window: `{10, 12, 18}` $\rightarrow$ range $= 8$

Iteration 5: Pop `(10, 0, 1)`

```
min_val = 10, list_idx = 0, idx_in_list = 1
```

Check:

```
18 - 10 = 8 < 5? → No
```

Advance list 0:

```
1+1=2 < 5 → True
next_val = nums[0][2] = 15
push (15, 0, 2)
max_val = max(18, 15) = 18
```

Heap: `[(12, 1, 2), (15, 0, 2), (18, 2, 1)]` $\rightarrow$ `[12, 15, 18]`

Window: `{12, 15, 18}` $\rightarrow$ range $= 6$

Iteration 6: Pop `(12, 1, 2)`

```
min_val = 12, list_idx = 1, idx_in_list = 2
```

Check:

```
18 - 12 = 6 < 5? → No
```

Advance list 1:

```
2+1=3 < 4 → True
next_val = nums[1][3] = 20
push (20, 1, 3)
max_val = max(18, 20) = 20
```

Heap: `[(15, 0, 2), (18, 2, 1), (20, 1, 3)]` → `[15, 18, 20]`

Window: `{15, 18, 20}` → range = 5 → same as before → no update

Iteration 7: Pop `(15, 0, 2)`

```
min_val = 15, list_idx = 0, idx_in_list = 2
```

Check:

```
20 - 15 = 5 < 5? → No
```

Advance list 0:

```
2+1=3 < 5 → True
next_val = nums[0][3] = 24
push (24, 0, 3)
max_val = max(20, 24) = 24
```

Heap: `[(18, 2, 1), (20, 1, 3), (24, 0, 3)]` → `[18, 20, 24]`

Window: `{18, 20, 24}` → range = 6

Iteration 8: Pop `(18, 2, 1)`

```
min_val = 18, list_idx = 2, idx_in_list = 1
```

Check:

```
24 - 18 = 6 < 5? → No
```

Advance list 2:

```
1+1=2 < 4 → True
next_val = nums[2][2] = 22
push (22, 2, 2)
max_val = max(24, 22) = 24
```

Heap: `[(20, 1, 3), (22, 2, 2), (24, 0, 3)]` → `[20, 22, 24]`

Now check:

```
24 - 20 = 4 < 5? → YES!
```

Update best range: `best_start = 20`, `best_end = 24`

We found a better range: `[20, 24]` (size = 4)

Iteration 9: Pop `(20, 1, 3)`

```
min_val = 20, list_idx = 1, idx_in_list = 3
```

Check:

```
24 - 20 = 4 < 4? → No (4 == 4)
```

Now try to advance list 1:

```
3+1=4 < 4? → False → list 1 is exhausted!
break
```

Loop ends.

Final Output

```
return [best_start, best_end]  # → [20, 24]
```

Summary Table: Key Variables Over Time

Iteration	Popped From	New Max	Current Window	Range	Best Range
1	List 1 (0)	9	{4,5,9}	5	[0,5]
2	List 0 (4)	10	{5,9,10}	5	[0,5]
3	List 2 (5)	18	{9,10,18}	9	[0,5]
4	List 1 (9)	18	{10,12,18}	8	[0,5]
5	List 0 (10)	18	{12,15,18}	6	[0,5]
6	List 1 (12)	20	{15,18,20}	5	[0,5]
7	List 0 (15)	24	{18,20,24}	6	[0,5]
8	List 2 (18)	24	{20,22,24}	4	[20,24]
9	List 1 (20)	24	List 1 done → break		

Why This Works: Algorithm Logic

Concept	Explanation
Min-Heap	Always picks the smallest current element → helps shrink the left side of the range.
Track `max_val`	Ensures we know how wide the current window is.
Replace with next in same list	Keeps one element per list, explores new combinations.
Break when list exhausted	Can't form a full window anymore → stop.
Greedy but optimal	Because arrays are sorted, advancing the smallest guarantees we don't miss the global minimum.

Final Answer

```
[20, 24]
```

Pro Tips for Understanding

- Think of the heap as a **“priority queue”** of “front runners” — always the smallest.
- The `max_val` is like the **tallest person in the group** — we care about the span between shortest and tallest.
- Every time we move the shortest forward, we’re trying to **tighten the group**.