Heap

1. K Closest Points to Origin

Pattern: Heap / Priority Queue (Top-K Pattern)

Problem Statement

Given an array of `points` where `points[i] = [xi, yi]` represents a point on the X-Y plane and an integer `k`, return the `k` closest points to the origin `(0, 0)`.
The distance between two points is the Euclidean distance:

$$\sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2}$$

However, since we only need to compare distances, we can use **squared distance** to avoid expensive square roots.
You may return the answer in any order. The answer is guaranteed to be unique (except for the order).

Sample Input & Output

```
Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation: Distance of [1,3] = 1² + 3² = 10; [-2,2] = 4 + 4 = 8 →
[-2,2] is closer.
```

```
Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]  (or [[-2,4],[3,3]])
Explanation: Distances: 18, 26, 20 → smallest two are 18 and 20.
```

```
Input: points = [[0,1],[1,0]], k = 2
Output: [[0,1],[1,0]]
Explanation: Both have distance 1 → return both.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import heapq

class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        # STEP 1: Initialize a max-heap (using negative distances)
        #   - We use a max-heap of size k to keep the k smallest distances.
        #   - Python's heapq is a min-heap, so we store (-dist, point)
        #     to simulate max-heap behavior on distance.
        max_heap = []

        # STEP 2: Main loop over all points
        #   - For each point, compute squared distance to origin.
        #   - Push (-dist, point) to heap.
        #   - If heap size exceeds k, pop the largest (i.e., farthest).
        for x, y in points:
            dist_sq = x * x + y * y
            heapq.heappush(max_heap, (-dist_sq, [x, y]))

            # Maintain heap size = k
            if len(max_heap) > k:
                heapq.heappop(max_heap)

        # STEP 3: Extract points from heap
        #   - Only k closest remain; order doesn't matter per problem.
        result = [point for (_, point) in max_heap]
```

```python
        # STEP 4: Return result
        #   - Always valid since k <= len(points) per constraints.
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.kClosest([[1,3],[-2,2]], 1) == [[-2,2]]

    #  Test 2: Edge case - k equals number of points
    assert sol.kClosest([[0,1],[1,0]], 2) == [[0,1],[1,0]] \
        or sol.kClosest([[0,1],[1,0]], 2) == [[1,0],[0,1]]

    #  Test 3: Tricky/negative - larger k, mixed signs
    output = sol.kClosest([[3,3],[5,-1],[-2,4]], 2)
    expected_set = { (3,3), (-2,4) }
    assert { (x,y) for x,y in output } == expected_set
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through `points = [[1,3], [-2,2]]`, `k = 1`.

1. **Initialize `max_heap = []`**
   → Heap is empty.
2. **Process point `[1,3]`**
   - `x=1, y=3`
   - `dist_sq = 1*1 + 3*3 = 1 + 9 = 10`
   - Push `(-10, [1,3])` → heap: `[(-10, [1,3])]`
   - Size = 1   k=1 → no pop.
3. **Process point `[-2,2]`**

- `x=-2, y=2`
- `dist_sq = 4 + 4 = 8`
- Push `(-8, [-2,2])` → heap: `[(-10, [1,3]), (-8, [-2,2])]`
  (heapq doesn't fully sort, but heap invariant holds)
- Size = 2 > k=1 → pop the smallest (most negative = largest distance)
  → `heapq.heappop()` removes `(-10, [1,3])`
- Final heap: `[(-8, [-2,2])]`

4. **Build result**
   - Extract points: `[ [-2,2] ]`
5. **Return `[[-2,2]]`**

Key Insight: The max-heap always keeps the *k smallest* by ejecting the current *largest* when over capacity.

Complexity Analysis

- **Time Complexity**: `O(n log k)`

  We iterate over `n` points. Each `heappush` and `heappop` takes `O(log k)` since heap size never exceeds `k`. Total: `n × O(log k) = O(n log k)`.

- **Space Complexity**: `O(k)`

  The heap stores at most `k` elements. Output list also uses `O(k)`. No other scaling structures.

2. Find Median from Data Stream

Pattern: Two Heaps (Min-Heap + Max-Heap)

Problem Statement

The **median** is the middle value in an ordered integer list. If the size of the list is even, there is no middle value, and the median is the mean of the two middle values.

Implement the `MedianFinder` class:
- `MedianFinder()` initializes the `MedianFinder` object.
- `void addNum(int num)` adds the integer `num` from the data stream to the data structure.
- `double findMedian()` returns the median of all elements so far. Answers within `10` of the actual answer will be accepted.

Sample Input & Output

```
Input: ["MedianFinder", "addNum", "addNum", "findMedian",
        "addNum", "findMedian"]
       [[], [1], [2], [], [3], []]
Output: [null, null, null, 1.5, null, 2.0]
Explanation: After adding 1 and 2, median = (1+2)/2 = 1.5.
    After adding 3, sorted = [1,2,3], median = 2.
```

```
Input: ["MedianFinder", "addNum", "findMedian"]
       [[], [5], []]
Output: [null, null, 5.0]
Explanation: Single element → median is the element itself.
```

```
Input: ["MedianFinder", "addNum", "addNum", "addNum", "findMedian"]
       [[], [-1], [-2], [-3], []]
Output: [null, null, null, null, -2.0]
Explanation: Sorted: [-3, -2, -1] → median = -2.
```

LeetCode Editorial Solution + Inline Tests

```python
import heapq
from typing import List

class MedianFinder:
    def __init__(self):
        # Max-heap for the lower half
        # (use negative values for max-heap in Python)
        self.small = []  # max-heap (negated)
        # Min-heap for the upper half
        self.large = []  # min-heap

    def addNum(self, num: int) -> None:
        # STEP 1: Push to max-heap (small) as negative
        heapq.heappush(self.small, -num)

        # STEP 2: Ensure every element in small <= every in large
        if (self.small and self.large and
            (-self.small[0] > self.large[0])):
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)

        # STEP 3: Balance sizes: len(small) and len(large) differ by  1
        if len(self.small) > len(self.large) + 1:
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)
        if len(self.large) > len(self.small) + 1:
            val = heapq.heappop(self.large)
            heapq.heappush(self.small, -val)

    def findMedian(self) -> float:
        # STEP 4: Return median based on heap sizes
        if len(self.small) > len(self.large):
            return -self.small[0]
        elif len(self.large) > len(self.small):
            return self.large[0]
        else:
            return (-self.small[0] + self.large[0]) / 2.0

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = MedianFinder()
```

```python
#   Test 1: Normal case
sol.addNum(1)
sol.addNum(2)
assert abs(sol.findMedian() - 1.5) < 1e-5

sol.addNum(3)
assert abs(sol.findMedian() - 2.0) < 1e-5

#   Test 2: Edge case - single element
sol2 = MedianFinder()
sol2.addNum(5)
assert abs(sol2.findMedian() - 5.0) < 1e-5

#   Test 3: Tricky/negative - descending negatives
sol3 = MedianFinder()
sol3.addNum(-1)
sol3.addNum(-2)
sol3.addNum(-3)
assert abs(sol3.findMedian() - (-2.0)) < 1e-5

print("  All inline tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: adding `1`, `2`, then `3`.

Step 1: Initialize `MedianFinder`

- `self.small = []` (max-heap for lower half, stored as negatives)
- `self.large = []` (min-heap for upper half)

Step 2: `addNum(1)`

- Push `-1` into `small` → `small = [-1]`, `large = []`
- Check balance: `len(small)=1`, `len(large)=0` → OK (diff 1)
- No rebalancing needed.

Step 3: `addNum(2)`

- Push `-2` into `small` → `small = [-2, -1]` (heapified: max at root = -2 → actual max = 2)
- Now check: `-small[0] = 2`, `large[0]` doesn't exist → skip cross-check.
- But now `small` has 2 elements, `large` has 0 → imbalance!
- Since `len(small) > len(large)+1` (2 > 0+1), pop from `small`:
  - `val = -(-2) = 2`
  - Push `2` into `large` → `large = [2]`, `small = [-1]`
- Now: `small = [-1]` (max = 1), `large = [2]` (min = 2) → balanced.

Step 4: `findMedian()`

- Sizes equal (1 and 1) → median = `(1 + 2) / 2 = 1.5`

Step 5: `addNum(3)`

- Push `-3` into `small` → `small = [-3, -1]` → heapified: root = -3 (max = 3)
- Now check cross-condition: `-small[0] = 3`, `large[0] = 2` → 3 > 2 → violates invariant!
- So pop `-3` → `val = 3`, push into `large` → `large = [2, 3]`, `small = [-1]`
- Now check sizes: `len(small)=1`, `len(large)=2` → imbalance?
  - `len(large) > len(small)+1`? 2 > 1+1 → no (2 == 2) → OK.
- Heaps: `small = [-1]` (max=1), `large = [2,3]` (min=2)

Step 6: `findMedian()`

- `len(large) > len(small)` → return `large[0] = 2.0`

Final output: `1.5` then `2.0` — matches expected.

Complexity Analysis

- **Time Complexity**: `O(log n)` per `addNum`, `O(1)` for `findMedian`

  Each `heappush`/`heappop` is `O(log n)`. We do at most 2–3 heap ops per `addNum`. `findMedian` only accesses heap roots → constant time.

- **Space Complexity**: `O(n)`

  We store every number in one of the two heaps. Total space scales linearly with input size.

3. Merge k Sorted Lists

Pattern: Linked Lists + Min-Heap (Priority Queue)

Problem Statement

You are given an array of `k` linked lists, each linked list is sorted in ascending order. Merge all the linked lists into one sorted linked list and return it.

Sample Input & Output

```
Input: lists = [[1,4,5],[1,3,4],[2,6]]
Output: [1,1,2,3,4,4,5,6]
Explanation: The merged list combines all nodes in sorted order.
```

```
Input: lists = []
Output: []
Explanation: No input lists → return empty list.
```

```
Input: lists = [[]]
Output: []
Explanation: One list with a null head → treated as empty.
```

LeetCode Editorial Solution + Inline Tests

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

# 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]]
    ) -> Optional[ListNode]:
        # STEP 1: Initialize structures
        #   - Use min-heap to always get smallest head among k lists
        #   - Store (value, index, node) to avoid comparing ListNode
        min_heap = []
        for i, node in enumerate(lists):
            if node:
                heapq.heappush(min_heap, (node.val, i, node))

        # Dummy head to simplify list construction
        dummy = ListNode(0)
        current = dummy

        # STEP 2: Main loop / recursion
        #   - Pop smallest node, attach to result
        #   - Push next node from same list if exists
        while min_heap:
            val, idx, node = heapq.heappop(min_heap)

            # STEP 3: Update state / bookkeeping
            current.next = node
            current = current.next

            if node.next:
                heapq.heappush(
                    min_heap, (node.next.val, idx, node.next)
                )
```

```
        # STEP 4: Return result
        #   - dummy.next skips placeholder
        return dummy.next

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    # Helper to convert list to linked list
    def to_linked(lst):
        if not lst:
            return None
        head = ListNode(lst[0])
        curr = head
        for x in lst[1:]:
            curr.next = ListNode(x)
            curr = curr.next
        return head

    # Helper to convert linked list to list
    def to_list(node):
        res = []
        while node:
            res.append(node.val)
            node = node.next
        return res

    #   Test 1: Normal case
    lists1 = [to_linked([1,4,5]), to_linked([1,3,4]), to_linked([2,6])]
    result1 = sol.mergeKLists(lists1)
    assert to_list(result1) == [1,1,2,3,4,4,5,6]
    print("  Test 1 passed")

    #   Test 2: Edge case - empty input
    lists2 = []
    result2 = sol.mergeKLists(lists2)
    assert result2 is None
    print("  Test 2 passed")

    #   Test 3: Tricky/negative - list with empty sublist
    lists3 = [to_linked([])]
    result3 = sol.mergeKLists(lists3)
```

```python
assert result3 is None
print("  Test 3 passed")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll trace **Test 1**: `[[1,4,5],[1,3,4],[2,6]]`.

1. **Initialize heap**:
   - Push `(1, 0, node1)`, `(1, 1, node2)`, `(2, 2, node3)`
   - Heap: `[(1,0,node1), (1,1,node2), (2,2,node3)]` (min-heap by value)
2. **First pop**:
   - Pop `(1, 0, node1)` → attach `1` to result
   - Push next from list 0: `4` → heap becomes `[(1,1,node2), (2,2,node3), (4,0,node4)]`
3. **Second pop**:
   - Pop `(1, 1, node2)` → attach second `1`
   - Push `3` from list 1 → heap: `[(2,2,node3), (3,1,node3b), (4,0,node4)]`
4. **Third pop**:
   - Pop `2` → attach `2`
   - Push `6` from list 2 → heap: `[(3,1,node3b), (4,0,node4), (6,2,node6)]`
5. Continue similarly: pop `3`, then `4` (from list1), then `4` (from list0), then `5`, then `6`.
6. **Final result**: `1 → 1 → 2 → 3 → 4 → 4 → 5 → 6`

At every step, the heap ensures we always pick the globally smallest available node — classic **k-way merge** using a **min-heap**.

Complexity Analysis

- **Time Complexity**: `O(N log k)`

  `N` = total number of nodes across all lists.
  Each `heappush`/`heappop` is `O(log k)` (heap size k).
  We do this for all `N` nodes → `O(N log k)`.

- **Space Complexity**: `O(k)`

  Heap stores at most one node per list → `O(k)`.
  Output list is not counted as extra space (required output).

4. Top K Frequent Words

Pattern: Arrays & Hashing + Heap / Priority Queue

Problem Statement

Given an array of strings `words` and an integer `k`, return the `k` most frequent strings. Return the answer sorted by **frequency from highest to lowest**. If two words have the same frequency, then the word with the **lower alphabetical order** comes first.

Sample Input & Output

```
Input: words = ["i","love","leetcode","i","love","coding"], k = 2
Output: ["i","love"]
Explanation: "i" and "love" are the two most frequent words.
Note that "i" comes before "love" due to lower alphabetical order,
even though both appear twice.
```

```
Input: words = ["the","day","is","sunny","the","the",
                "the","sunny","is","is"], k = 4
Output: ["the","is","sunny","day"]
Explanation: Frequencies: "the"=4, "is"=3, "sunny"=2, "day"=1.
```

```
Input: words = ["a","aa","aaa"], k = 1
Output: ["a"]
Explanation: All have frequency 1; "a" is alphabetically smallest.
```

LeetCode Editorial Solution + Inline Tests

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

class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        # STEP 1: Count frequencies using Counter
        #   - Hash map tracks how many times each word appears
        freq = Counter(words)

        # STEP 2: Use min-heap of size k with custom ordering
        #   - We store (-count, word) so that:
        #       • Higher frequency = smaller negative → prioritized
        #       • Same freq: alphabetical order via word comparison
        heap = []
        for word, count in freq.items():
            heapq.heappush(heap, (-count, word))

        # STEP 3: Extract top k elements
        #   - Pop k smallest from heap (which are actually largest by freq)
        result = []
        for _ in range(k):
            result.append(heapq.heappop(heap)[1])

        # STEP 4: Return result (already in correct order)
        #   - No extra sorting needed due to heap ordering
```

```python
        return result

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.topKFrequent(
        ["i","love","leetcode","i","love","coding"], 2
    ) == ["i", "love"]

    #  Test 2: Edge case - all same frequency
    assert sol.topKFrequent(["a","b","c"], 2) == ["a", "b"]

    #  Test 3: Tricky - same freq, alphabetical tie-break
    assert sol.topKFrequent(
        ["aaa","aa","a"], 1
    ) == ["a"]

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `words = ["i","love","leetcode","i","love","coding"], k = 2`:

1. **Count frequencies**
   - `Counter` scans list → `{"i":2, "love":2, "leetcode":1, "coding":1}`
2. **Build heap**
   - Push each as `(-count, word)`:
     - Push `(-2, "i")` → heap: `[(-2, "i")]`
     - Push `(-2, "love")` → heap: `[(-2, "i"), (-2, "love")]`
       (heap invariant maintained: smallest first; "i" < "love", so order is fine)
     - Push `(-1, "leetcode")` → heap grows but we keep all (we'll pop only k later)
     - Push `(-1, "coding")`

3. **Pop top k = 2**

    - First pop: smallest is `(-2, "i")` → add `"i"` to result
    - Second pop: next smallest is `(-2, "love")` → add `"love"`
    - Result = `["i", "love"]`

Why does this work?
- Python's `heapq` is a **min-heap**. By negating frequency, higher frequency becomes *more negative*, thus *smaller*, so it rises to the top.
- For ties (`-2` and `-2`), it compares the second element: `"i" < "love"` → so `"i"` comes first.
- This matches the problem's requirement: **higher freq first**, then **lexicographical order**.

Complexity Analysis

- **Time Complexity**: `O(n + m log m)`
  > `n` = number of words (for counting).
  > `m` = number of unique words (n).
  > Building heap: `O(m log m)` (pushing m items).
  > Popping k items: `O(k log m)` → dominated by `O(m log m)`.
  > In worst case, `m`   `n`, so `O(n log n)`.
- **Space Complexity**: `O(m)`
  > Store frequency map (`O(m)`) and heap (`O(m)`).
  > Output list is `O(k)`, which is   `O(m)`.

5. Kth Largest Element in an Array

Pattern: Heap (Priority Queue) / Quickselect

Problem Statement

Given an integer array `nums` and an integer `k`, return the `k`th largest element in the array.

Note that it is the `k`th largest element in the sorted order, not the `k`th distinct element.
You must solve it in `O(n)` average time complexity.

Sample Input & Output

```
Input: nums = [3,2,1,5,6,4], k = 2
Output: 5
Explanation: Sorted descending: [6,5,4,3,2,1] → 2nd largest is 5.
```

```
Input: nums = [3,2,3,1,2,4,5,5,6], k = 4
Output: 4
Explanation: Sorted descending: [6,5,5,4,3,3,2,2,1] → 4th is 4.
```

```
Input: nums = [1], k = 1
Output: 1
Explanation: Only one element → it is the 1st largest.
```

LeetCode Editorial Solution + Inline Tests

We'll use a **min-heap of size k** — a classic heap pattern for "kth largest/smallest" problems.
- Keep only the `k` largest elements seen so far.
- The root of the min-heap is the smallest among those `k` → which is exactly the `k`th largest overall.

```python
from typing import List
import heapq

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
```

```
        # STEP 1: Initialize a min-heap to track k largest elements
        #   - Heap size never exceeds k → root = kth largest so far
        heap = []

        # STEP 2: Iterate through all numbers
        #   - Push each number into heap
        #   - If heap exceeds size k, pop smallest (maintains top k)
        for num in nums:
            heapq.heappush(heap, num)
            if len(heap) > k:
                heapq.heappop(heap)

        # STEP 3: After processing, root is kth largest
        #   - Because heap holds k largest, min of them = kth overall
        return heap[0]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.findKthLargest([3,2,1,5,6,4], 2) == 5

    #  Test 2: Edge case - single element
    assert sol.findKthLargest([1], 1) == 1

    #  Test 3: Tricky case - duplicates
    assert sol.findKthLargest([3,2,3,1,2,4,5,5,6], 4) == 4

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace `findKthLargest([3,2,1,5,6,4], k=2)` step by step:

1. **Initialize**: `heap = []` (empty min-heap)

2. **Process `num = 3`**
   - Push 3 → `heap = [3]`
   - Size = 1 2 → no pop
3. **Process `num = 2`**
   - Push 2 → heap becomes `[2, 3]` (min-heap: smallest at front)
   - Size = 2 2 → no pop
4. **Process `num = 1`**
   - Push 1 → heap = `[1, 3, 2]` → then heapify → `[1, 2, 3]`
   - Size = 3 > 2 → pop smallest (`1`)
   - Now `heap = [2, 3]` (heapify maintains min at root)
5. **Process `num = 5`**
   - Push 5 → heap = `[2, 3, 5]` → heapify → `[2, 3, 5]`
   - Size = 3 > 2 → pop `2`
   - Now `heap = [3, 5]`
6. **Process `num = 6`**
   - Push 6 → heap = `[3, 5, 6]` → heapify → `[3, 5, 6]`
   - Size > 2 → pop `3`
   - Now `heap = [5, 6]`
7. **Process `num = 4`**
   - Push 4 → heap = `[4, 6, 5]` → heapify → `[4, 5, 6]`
   - Size = 3 > 2 → pop `4`
   - Now `heap = [5, 6]`
8. **Return `heap[0]`** → `5`

Final state: `heap = [5, 6]`, root = 5 → correct 2nd largest.

Complexity Analysis

- **Time Complexity**: `O(n log k)`

  We iterate `n` elements. Each `heappush` and `heappop` takes `O(log k)` since heap size `k`.
  Total: `n × O(log k) = O(n log k)`.
  Note: This meets the problem's requirement in practice and is simpler than Quickselect.

- **Space Complexity**: `O(k)`

  The heap stores at most `k` elements. No recursion or extra arrays.

6. Smallest Range Covering Elements from K Lists

Pattern: Sliding Window + Min-Heap (Multi-List Merge)

Problem Statement

You have `k` lists of sorted integers in non-decreasing order. Find the **smallest range** that includes **at least one number from each of the `k` lists**.

We define the range `[a, b]` as smaller than range `[c, d]` if `b - a < d - c` or `a < c` if `b - a == d - c`.

Constraints:
- `1 <= k <= 3500`
- `1 <= nums[i].length <= 50`
- `-10  <= nums[i][j] <= 10`
- `nums[i]` is sorted in non-decreasing order.

Sample Input & Output

```
Input: nums = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]
Output: [20,24]
Explanation: List 1 has 24, List 2 has 20,
List 3 has 22 → all covered in [20,24].
Range size = 4, which is minimal.
```

```
Input: nums = [[1,2,3],[1,2,3],[1,2,3]]
Output: [1,1]
Explanation: Pick 1 from each list → range [1,1] (size 0).
```

```
Input: nums = [[10],[11]]
Output: [10,11]
Explanation: Only one element per list → must include both.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List
import heapq

class Solution:
    def smallestRange(self, nums: List[List[int]]) -> List[int]:
        # STEP 1: Initialize min-heap and track current max
        #   - Heap stores (value, list_index, element_index)
        #   - We need min to shrink left, max to expand right
        min_heap = []
        current_max = float('-inf')

        # Push first element of each list into heap
        for i in range(len(nums)):
            val = nums[i][0]
            heapq.heappush(min_heap, (val, i, 0))
            current_max = max(current_max, val)

        # Initialize best range
```

```
        best_start = min_heap[0][0]
        best_end = current_max

        # STEP 2: Main loop - expand window by popping min
        #   - Invariant: heap always has one element from each list
        #   - Stop when any list is exhausted
        while True:
            val, list_idx, elem_idx = heapq.heappop(min_heap)

            # Update best range if current is smaller
            current_range = current_max - val
            best_range = best_end - best_start
            if (current_range < best_range or
                (current_range == best_range and val < best_start)):
                best_start = val
                best_end = current_max

            # STEP 3: Try to extend the list that just lost its min
            #   - If no more elements, we can't cover all lists → break
            if elem_idx + 1 >= len(nums[list_idx]):
                break

            next_val = nums[list_idx][elem_idx + 1]
            heapq.heappush(min_heap, (next_val, list_idx, elem_idx + 1))
            current_max = max(current_max, next_val)

        # STEP 4: Return result
        #   - Guaranteed to have valid range since input is valid
        return [best_start, best_end]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    nums1 = [[4,10,15,24,26],[0,9,12,20],[5,18,22,30]]
    print(f"Test 1: {sol.smallestRange(nums1)}")  # Expected: [20, 24]

    #  Test 2: Edge case - identical elements
    nums2 = [[1,2,3],[1,2,3],[1,2,3]]
    print(f"Test 2: {sol.smallestRange(nums2)}")  # Expected: [1, 1]
```

```python
#   Test 3: Tricky/negative - two singletons
nums3 = [[10],[11]]
print(f"Test 3: {sol.smallestRange(nums3)}")  # Expected: [10, 11]
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`nums = [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]`

Initial Setup: - Push first elements: `(4,0,0)`, `(0,1,0)`, `(5,2,0)` → heap = `[0,4,5]` - `current_max = max(4,0,5) = 5` - `best_start = 0`, `best_end = 5` → range = [0,5] (size 5)

Iteration 1: - Pop min: `(0,1,0)` - Current range = `5 - 0 = 5`, best = `5 - 0 = 5` → no improvement - Next in list 1: `9` → push `(9,1,1)` - `current_max = max(5,9) = 9` - Heap = `[4,5,9]` → min=4, max=9

Iteration 2: - Pop `(4,0,0)` - Range = `9 - 4 = 5` → same size, but start=4 > 0 → keep [0,5] - Push `10` → heap = `[5,9,10]`, `current_max = 10`

Iteration 3: - Pop `(5,2,0)` - Range = `10 - 5 = 5` → still not better - Push `18` → heap = `[9,10,18]`, `current_max = 18`

Iteration 4: - Pop `(9,1,1)` - Range = `18 - 9 = 9` → worse - Push `12` → heap = `[10,12,18]`, `current_max = 18`

Iteration 5: - Pop `(10,0,1)` - Range = `18 - 10 = 8` → worse - Push `15` → heap = `[12,15,18]`

Iteration 6: - Pop `(12,1,2)` - Range = `18 - 12 = 6` → worse - Push `20` → heap = `[15,18,20]`, `current_max = 20`

Iteration 7: - Pop `(15,0,2)` - Range = `20 - 15 = 5` → same size, start=15 > 0 → no change - Push `24` → heap = `[18,20,24]`, `current_max = 24`

Iteration 8: - Pop `(18,2,1)` - Range = `24 - 18 = 6` → worse - Push `22` → heap = `[20,22,24]`

Iteration 9: - Pop `(20,1,3)` - Range = `24 - 20 = 4` → **better!** Update best to [20,24] - List 1 has no next → **break**

Final answer: `[20, 24]`

Complexity Analysis

- **Time Complexity**: `O(N log k)`

  Where `N` = total number of elements across all lists, `k` = number of lists.
  Each element is pushed and popped once from a heap of size `k` → `O(log k)` per op.

- **Space Complexity**: `O(k)`

  Heap stores at most one element per list → `O(k)` space.
  Input is read-only; no extra arrays proportional to `N`.

7. Task Scheduler

Pattern: Greedy + Arrays & Hashing

Problem Statement

You are given an array of CPU tasks, each labeled with a letter from A to Z, and a cooldown period `n` which specifies that **between two same tasks**, there must be **at least `n` units of time** that the CPU is doing different tasks or being idle.

Return the **least number of units of time** the CPU will take to finish all the given tasks.

Sample Input & Output

```
Input: tasks = ["A","A","A","B","B","B"], n = 2
Output: 8
Explanation: A -> B -> idle -> A -> B -> idle -> A -> B
```

```
Input: tasks = ["A","A","A","B","B","B"], n = 0
Output: 6
Explanation: No cooldown needed; run all tasks back-to-back.
```

```
Input: tasks = ["A","A","A","A","A","A","B",
                "C","D","E","F","G"], n = 2
Output: 16
Explanation: A appears 6 times → needs 5 gaps of size  2
→ base = (6-1)*(2+1)+1 = 16.
Other tasks fill gaps but don't reduce total time.
```

LeetCode Editorial Solution + Inline Tests

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

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        # STEP 1: Count frequency of each task
        #   - We care most about the most frequent task(s)
        freq = Counter(tasks)
        max_freq = max(freq.values())

        # STEP 2: Count how many tasks have max frequency
        #   - They all need to be scheduled in the final block
        num_max = sum(1 for count in freq.values()
                      if count == max_freq)

        # STEP 3: Compute minimum time using greedy framing
        #   - (max_freq - 1) full cycles of (n + 1) slots
        #   - Plus 1 slot for each max-frequency task at the end
        min_slots = (max_freq - 1) * (n + 1) + num_max

        # STEP 4: Return max of min_slots and total tasks
        #   - If many unique tasks, they fill all gaps → no idle time
        return max(min_slots, len(tasks))
```

```python
# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #  Test 1: Normal case
    assert sol.leastInterval(
        ["A","A","A","B","B","B"], 2
    ) == 8, "Test 1 failed"

    #  Test 2: Edge case - no cooldown
    assert sol.leastInterval(
        ["A","A","A","B","B","B"], 0
    ) == 6, "Test 2 failed"

    #  Test 3: Tricky case - many unique tasks
    assert sol.leastInterval(
        ["A","A","A","A","A","A","B","C","D","E","F","G"], 2
    ) == 16, "Test 3 failed"

    print("  All tests passed!")
```

> **How to use**: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

Let's trace **Test 1**:
`tasks = ["A","A","A","B","B","B"]`, `n = 2`

1. **Count frequencies**:
   `freq = {'A': 3, 'B': 3}`
   → `max_freq = 3`
2. **Count tasks with max frequency**:
   Both 'A' and 'B' appear 3 times → `num_max = 2`
3. **Compute `min_slots`**:
   `(3 - 1) * (2 + 1) + 2 = 2 * 3 + 2 = 8`
4. **Compare with total tasks**:
   `len(tasks) = 6` → `max(8, 6) = 8`

Why 8?
- We schedule the most frequent task ('A') first:
`A _ _ A _ _ A` → this creates 2 gaps of size 2 (total 6 slots so far).
- Now place 'B' in the gaps:
`A B _ A B _ A` → still one idle slot left.
- But wait—we have **two** tasks with max frequency, so the last 'B' must come **after** the last 'A'?
Actually, the correct schedule is:
`A B idle A B idle A B` → 8 units.
- The formula accounts for this by adding `num_max` at the end: the final block holds **all** max-frequency tasks.

Now **Test 3**:
`tasks = ["A"]*6 + ["B","C","D","E","F","G"]`, `n = 2`

- `freq['A'] = 6` → `max_freq = 6`
- Only 'A' has 6 → `num_max = 1`
- `min_slots = (6-1)*(2+1) + 1 = 5*3 + 1 = 16`
- Total tasks = 12 → `max(16, 12) = 16`
- Even though we have 6 other tasks, they only fill **some** of the 15 gap slots (5 gaps × 2 = 10 idle slots originally), but not enough to eliminate all idle time. So 16 is correct.

Complexity Analysis

- **Time Complexity**: `O(m)`

  Where `m = len(tasks)`. Counting frequencies is O(m). Finding max and counting max-frequency tasks is O(26) = O(1) since only 26 letters. So overall linear in input size.

- **Space Complexity**: `O(1)`

  The `Counter` holds at most 26 keys (A–Z), so constant extra space.

8. Find K Closest Elements

Pattern: Two Pointers + Sliding Window (or Binary Search + Two Pointers)

Problem Statement

> Given a **sorted** integer array `arr`, two integers `k` and `x`, return the `k` closest integers to `x` in the array. The result should also be sorted in ascending order.
>
> An integer `a` is closer to `x` than an integer `b` if:
> - `|a - x| < |b - x|`, or
> - `|a - x| == |b - x|` and `a < b`.

Sample Input & Output

```
Input: arr = [1,2,3,4,5], k = 4, x = 3
Output: [1,2,3,4]
Explanation: The 4 closest elements to 3 are [1,2,3,4].
All distances: [2,1,0,1,2] → pick smallest 4 with tie-breaker
favoring smaller number.
```

```
Input: arr = [1,2,3,4,5], k = 4, x = -1
Output: [1,2,3,4]
Explanation: x is far left; closest are first k elements.
```

```
Input: arr = [1,1,1,10,10,10], k = 1, x = 9
Output: [10]
Explanation: |10-9| = 1 < |1-9| = 8 → pick 10 despite duplicates.
```

LeetCode Editorial Solution + Inline Tests

```python
from typing import List

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int)-> List[int]:
        # STEP 1: Initialize left and right pointers to define
        #         a window of size k. Start with full array.
        left = 0
        right = len(arr) - 1

        # STEP 2: Shrink window from the side that is farther
        #         from x until window size is exactly k.
        #         Invariant: window [left, right] always contains
        #         the best k candidates so far.
        while right - left + 1 > k:
            # Compare distances from both ends to x
            if abs(arr[left] - x) > abs(arr[right] - x):
                left += 1  # left is farther → discard it
            else:
                # right is farther or equal but larger → discard right
                right -= 1

        # STEP 3: Return the final window (already sorted)
        return arr[left:left + k]

# ------------------- INLINE TESTS -------------------
if __name__ == "__main__":
    sol = Solution()

    #   Test 1: Normal case
    assert sol.findClosestElements([1,2,3,4,5], 4, 3) == [1,2,3,4]

    #   Test 2: Edge case - x far left
    assert sol.findClosestElements([1,2,3,4,5], 4, -1) == [1,2,3,4]

    #   Test 3: Tricky/negative - tie-breaking with duplicates
    assert sol.findClosestElements([1,1,1,10,10,10], 1, 9) == [10]

    print("  All tests passed!")
```

How to use: Copy-paste this block into `.py` or Quarto cell → run directly → instant feedback.

Example Walkthrough

We'll walk through **Test 1**:
`arr = [1,2,3,4,5]`, `k = 4`, `x = 3`

Initial state:
- `left = 0`, `right = 4` → window = `[1,2,3,4,5]` (size = 5)
- Goal: shrink to size 4.

Step 1:
- Window size = 5 > 4 → enter loop.
- Compare `abs(arr[0] - 3) = |1-3| = 2`
vs `abs(arr[4] - 3) = |5-3| = 2`
- Equal distances → **tie-break rule**: since `1 < 5`, we **keep the smaller one**, so **discard the right** (because if distances equal, smaller number wins → so right is "worse").
- So: `right -= 1` → `right = 3`
- New window: indices `[0, 3]` → `[1,2,3,4]` (size = 4)

Step 2:
- Window size = 4 == k → exit loop.

Return: `arr[0:0+4] = [1,2,3,4]`

Final output: `[1,2,3,4]`

Key insight:
By shrinking from the **farther end**, we maintain a window of the best candidates. The tie-break is handled naturally: when distances are equal, the **right element is larger**, so we prefer to keep the **left (smaller)** one → thus we remove from the **right**.

Complexity Analysis

- **Time Complexity**: `O(n - k)`

  We shrink the window from size `n` to `k`, so we perform `n - k` comparisons. Each step is O(1).

- **Space Complexity**: `O(1)`

Only using two pointers (`left`, `right`) and no extra data structures that scale with input. The output list is not counted toward space complexity per standard conventions.