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

```
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__":
```

```python
    # 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:

```python
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 Summary

Given k linked lists, each sorted in ascending order, merge them into one sorted list.

Pattern

- **Heap / Priority Queue** (k-way merge)
- At each step, pick the smallest head from k lists

Solution with Inline Comments

```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

def mergeKLists(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
    # Create a dummy head to simplify list construction
    dummy = ListNode(0)
    current = dummy

    # Min-heap to store (value, node) pairs
    heap = []

    # Initialize heap with the first node of each non-empty list
    for lst in lists:
        if lst:
            heapq.heappush(heap, (lst.val, lst))

    # While there are nodes in the heap
    while heap:
        # Pop the smallest element
        val, node = heapq.heappop(heap)

        # Link it to the result list
        current.next = node
        current = current.next

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

    # Return the merged list (skip dummy)
    return dummy.next


# ---- Official LeetCode Example ----
```

```python
if __name__ == "__main__":
    # Example Input: lists = [[1,4,5],[1,3,4],[2,6]]
    # Build linked lists
    l1 = ListNode(1, ListNode(4, ListNode(5)))
    l2 = ListNode(1, ListNode(3, ListNode(4)))
    l3 = ListNode(2, ListNode(6))

    lists = [l1, l2, l3]

    # Call function
    merged = mergeKLists(lists)

    # Print Output: [1,1,2,3,4,4,5,6]
    result = []
    while merged:
        result.append(merged.val)
        merged = merged.next
    print("Output:", result)  # Output: [1, 1, 2, 3, 4, 4, 5, 6]
```

Let's walk through your **LeetCode "Merge k Sorted Lists"** code step by step, explaining how it works and why each part is important.

Problem Overview

You are given `k` sorted linked lists and need to merge them into one sorted linked list.

Example Input:

```python
lists = [
    [1, 4, 5],
    [1, 3, 4],
    [2, 6]
]
```

Expected Output:

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

Code Walkthrough

1. Define the ListNode Class

```
class ListNode:
    def __init__(self, val=0, next=None):
        self.val = val
        self.next = next
```

- This defines a node in a singly-linked list.
- Each node has a `val` (value) and a `next` pointer to the next node.
- If `next` is `None`, it's the end of the list.

  Example: `ListNode(1, ListNode(4, ListNode(5)))` creates:
  `1 → 4 → 5 → None`

---

2. Main Function: `mergeKLists`

```
def mergeKLists(lists: List[Optional[ListNode]]) -> Optional[ListNode]:
```

- Takes a list of `k` linked lists (`lists`) as input.
- Returns a single merged sorted linked list.

---

3. Create Dummy Head

```
dummy = ListNode(0)
current = dummy
```

- A **dummy head** is used to simplify the logic of building the result list.
- `dummy` acts as a placeholder so we don't have to handle edge cases like empty result or setting the first node specially.
- `current` points to the current end of the result list — we'll append new nodes here.

> Why? Because we can always return `dummy.next`, which skips the dummy node.

---

4. Initialize Min-Heap

```
heap = []
```

- We'll use a **min-heap** to efficiently get the smallest value among the current heads of all non-empty lists.

> Heap stores tuples: `(value, node)`
> Python's `heapq` compares the first element of the tuple, then second if needed.

---

5. Push First Nodes into Heap

```
for lst in lists:
    if lst:
        heapq.heappush(heap, (lst.val, lst))
```

- Loop over each list in `lists`.
- If the list is not empty (`lst` exists), push its **first node** (`lst`) into the heap with its value.
- So after this loop:
  - Heap contains: `(1, l1_head)`, `(1, l2_head)`, `(2, l3_head)`
  - The smallest values are at the top.

Important: We store the **node**, not just the value, so we can later access `node.next`.

---

6. Main Loop: While Heap Is Not Empty

```
while heap:
    val, node = heapq.heappop(heap)
```

- Extract the smallest element from the heap (based on `val`).
- Now `node` is the next node to add to our result list.

---

7. Link Node to Result List

```
current.next = node
current = current.next
```

- Attach the selected node to the end of the result list.
- Move `current` forward to point to this new node.

After this, `current` tracks where the next node should be added.

---

8. Add Next Node from Same List (if exists)

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

- If the current node has a `next`, push that next node into the heap.
- This ensures we keep track of the next available node from the same list.

This maintains the invariant: heap always holds the **current head** of each non-empty list.

---

9. Return Merged List

```
return dummy.next
```

- Return the real start of the merged list (skip the dummy node).

Step-by-Step Execution (Example)

Given:

```
l1 = 1 → 4 → 5
l2 = 1 → 3 → 4
l3 = 2 → 6
```

Initial State:

- `heap = [(1, l1), (1, l2), (2, l3)]`
- `dummy → ?`, `current → dummy`

Iteration 1:

- Pop `(1, l1)` → add `l1` to result.
- `l1.next = 4`, so push `(4, l1.next)` into heap.
- Now heap: `[(1, l2), (2, l3), (4, l1.next)]`
- Result: `dummy → 1`

Iteration 2:

- Pop `(1, l2)` → add `l2` to result.
- `l2.next = 3`, push `(3, l2.next)` into heap.
- Heap: `[(2, l3), (3, l2.next), (4, l1.next)]`
- Result: `dummy → 1 → 1`

Iteration 3:

- Pop `(2, l3)` → add `l3` to result.
- `l3.next = 6`, push `(6, l3.next)` into heap.
- Heap: `[(3, l2.next), (4, l1.next), (6, l3.next)]`
- Result: `... → 2`

Iteration 4:

- Pop `(3, l2.next)` → add it.
- `l2.next.next = 4`, push `(4, l2.next.next)`
- Heap: `[(4, l1.next), (4, l2.next.next), (6, l3.next)]`
- Result: `... → 3`

Iteration 5:

- Pop `(4, l1.next)` → add it.
- `l1.next.next = 5`, push `(5, l1.next.next)`
- Heap: `[(4, l2.next.next), (5, l1.next.next), (6, l3.next)]`
- Result: `... → 4`

Iteration 6:

- Pop `(4, l2.next.next)` → add it.
- No more next, so nothing pushed.
- Heap: `[(5, l1.next.next), (6, l3.next)]`
- Result: `... → 4`

Iteration 7:

- Pop `(5, l1.next.next)` → add it.
- No next, so no push.
- Heap: `[(6, l3.next)]`
- Result: `... → 5`

Iteration 8:

- Pop `(6, l3.next)` → add it.
- Done!
- Heap is empty.

Final Result:

```
1 → 1 → 2 → 3 → 4 → 4 → 5 → 6
```

Correct!

Key Concepts & Optimizations

Feature	Purpose
Min-Heap	Efficiently find the smallest current head across `k` lists in O(log k) time
Dummy Node	Avoid special handling for first node
Heap Stores Nodes	Allows us to access `.next` and continue traversal
Time Complexity	O(N log k), where N = total number of nodes, k = number of lists
Space Complexity	O(k) for the heap

Final Output

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

This matches the expected result perfectly.

Summary

Your implementation is **correct, efficient, and well-structured**! It uses the standard optimal approach for merging k sorted linked lists:

> Use a min-heap to always pick the smallest available node
> Advance only one list at a time
> Build result incrementally using a dummy head

Great job!

Let me know if you'd like a version with comments, visualization, or comparison to other approaches (like divide-and-conquer).

Complexity

- **Time**: O(N log k), where N = total nodes, k = number of lists
- **Space**: O(k) — heap holds at most k nodes

---

4. Task Scheduler

The **"Task Scheduler"** problem on LeetCode (Problem #621) is a popular algorithm challenge that tests greedy thinking and efficient use of data structures. Below is a detailed editorial-style explanation of the optimal solution.

Problem: Task Scheduler (LeetCode #621)

Problem Statement:

You are given a character array `tasks` representing the tasks a CPU needs to do, where each letter represents a different task. Tasks could be done in any order, but each task must be done in an interval of **at least `n` units** (where `n` is the cooling interval). Return the **least number of units of time** the CPU will take to finish all the tasks.

Example:

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

Key Observations

1. The only constraint is that **the same task cannot be scheduled within `n` units of time**.
2. We can insert **idle cycles** or **other tasks** between two same tasks to satisfy the cooldown.
3. The goal is to **minimize total time**, so we want to **minimize idle time**.

Strategy: Greedy with Math Insight

We can solve this using a **greedy approach**:

1. **Count frequency** of each task.
2. Find the task with **maximum frequency**, say `maxFreq`.
3. There will be `(maxFreq - 1)` full **blocks** of size `(n + 1)` between the most frequent task.
   - For example, if `maxFreq = 3` and `n = 2`, we have:

     ```
     A _ _ A _ _ A
     ```

     $\rightarrow$ 2 gaps of size 2 $\rightarrow$ total block size = 3 per segment $\rightarrow$ 2 full segments of length (n+1) = 3

4. Then, we **fill other tasks** into these gaps.
5. Finally, **add the remaining tasks** that occur with the same max frequency.

Formula

Let: - `maxFreq` = maximum frequency among tasks - `countMaxFreq` = number of tasks that have frequency = `maxFreq`

Then, the **minimum time** required is:

```
result = max(
    (maxFreq - 1) * (n + 1) + countMaxFreq,
    total_tasks
)
```

Why this formula?

- `(maxFreq - 1) * (n + 1)` → number of slots taken by the most frequent task and its required cooldown.
- `+ countMaxFreq` → the last occurrence of each task with max frequency comes after the last gap.
- But if there are **so many different tasks** that they fill all idle slots, the total time is just `len(tasks)`.

So we take the **maximum** of the two to ensure we don't underestimate.

Example Walkthrough

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

- Frequencies: A → 3, B → 3
- `maxFreq = 3`, `countMaxFreq = 2` (both A and B appear 3 times)
- Total tasks = 6

Compute:

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

Compare with total tasks: `max(8, 6) = 8`

Output: `8`

Code (Python)

```
from collections import Counter

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        freq = Counter(tasks)
        maxFreq = max(freq.values())
        countMaxFreq = sum(1 for f in freq.values() if f == maxFreq)

        result = (maxFreq - 1) * (n + 1) + countMaxFreq
        return max(result, len(tasks))
```

Absolutely! Let's walk through the **mathematical solution** to the **Task Scheduler (LeetCode #621)** with a **detailed, step-by-step example** to build deep intuition.

Problem Recap

Goal: Find the **minimum time** to execute all tasks, given that: - Each task is a letter (e.g., `'A'`, `'B'`). - The same task must wait at least `n` units between executions. - You can interleave other tasks or insert **idle** slots.

Mathematical Formula (Recap)

The answer is:

```
result = max(
    (maxFreq - 1) * (n + 1) + countMaxFreq,
    total_tasks
)
```

Where: - `maxFreq` = highest frequency of any task - `countMaxFreq` = number of tasks that have frequency = `maxFreq` - `total_tasks` = length of the `tasks` array

---

Example 1: Classic Case

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

Step 1: Count Frequencies

Task	Count
A	3
B	3

So: - `maxFreq = 3` → A and B both appear 3 times - `countMaxFreq = 2` → two tasks (A and B) have max frequency - `total_tasks = 6`

Step 2: Apply the Formula

We compute the **minimum required time due to the most frequent tasks**:

```
(maxFreq - 1) * (n + 1) + countMaxFreq
= (3 - 1) * (2 + 1) + 2
= 2 * 3 + 2
= 6 + 2
= 8
```

Now compare with total number of tasks:

```
max(8, 6) = 8
```

Answer = 8

Visual Schedule

We schedule the most frequent tasks first (A and B), spaced at least `n=2` apart.

We start with A:

```
A _ _ A _ _ A
```

Each `A` is 3 units apart → satisfies `n=2` cooldown.

Now fill in `B` into the gaps:

```
A B _ A B _ A B
```

We used both B's in the gaps. Final B at the end.

No idle time left? Actually, we had to insert idle initially, but B filled most slots.

But wait — we still have:

```
Time:  1   2   3   4   5   6   7   8
      [A] [B] [idle] [A] [B] [idle] [A] [B]
```

Wait — actually, we can do better:

```
1: A
2: B
3: idle
4: A
5: B
6: idle
7: A
8: B
```

Yes → 8 units.

We **can't do better** because A needs to run 3 times with 2 gaps of at least 2 → forces structure.

So **8 is optimal**.

Example 2: Many Unique Tasks (No Idle Needed)

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

- Frequencies: A:2, B:2, C:2
- `maxFreq = 2`
- `countMaxFreq = 3` (A, B, C all appear 2 times)
- `total_tasks = 6`

Apply formula:

```
(2 - 1) * (2 + 1) + 3 = 1 * 3 + 3 = 6
max(6, 6) = 6
```

Answer = 6

Schedule:

We can do:

```
A B C A B C
```

Each task appears twice, separated by 2 other tasks → satisfies `n=2`.

No idle needed! The **gaps are fully filled** by other tasks.

So total time = 6.

This shows: when other tasks can **fill the gaps**, we don't need idle time.

Example 3: One Very Frequent Task

Input:
`tasks = ["A","A","A","A","A","A","B","C","D","E"]`
`n = 2`

- A:6, B:1, C:1, D:1, E:1
- `maxFreq = 6` (A)
- `countMaxFreq = 1` (only A has frequency 6)
- `total_tasks = 10`

Compute:

```
(6 - 1) * (2 + 1) + 1 = 5 * 3 + 1 = 15 + 1 = 16
max(16, 10) = 16
```

Answer = 16

Why 16?

We must schedule A like this:

```
A _ _ A _ _ A _ _ A _ _ A _ _ A
```

- 5 gaps between 6 A's
- Each gap is 3 units long (`n+1 = 3`)
- Total so far: `5*3 + 1` (last A) → but wait, structure is:

```
[A _ _] [A _ _] [A _ _] [A _ _] [A _ _] [A]
```

→ 5 full blocks of size 3 → 15, plus last A → but it's already included.

Actually, total length = `(maxFreq - 1) * (n + 1) + 1` = `5*3 + 1 = 16`

Now, we can fill B, C, D, E into the gaps.

Each gap has 2 empty slots → 5 gaps × 2 = 10 idle spots.

We only have 5 other tasks → we can place them, but still have idle time.

So minimum time is **16**, even though only 10 tasks.

We **can't compress** A's schedule.

Example 4: n = 0 (No Cooldown)

Input:
`tasks = ["A","A","B","B"]`, `n = 0`

- `maxFreq = 2`, `countMaxFreq = 2`, `total_tasks = 4`

Formula:

```
(2 - 1)*(0 + 1) + 2 = 1*1 + 2 = 3
max(3, 4) = 4
```

Answer = 4

Schedule: `A B A B` → no idle needed, `n=0` means no wait.

So total time = 4 → matches.

This shows why we take `max(..., total_tasks)` — you can't finish faster than the number of tasks.

Example 5: All Same Task

Input: `tasks = ["A","A","A"]`, `n = 1`

- `maxFreq = 3`, `countMaxFreq = 1`, `total_tasks = 3`

```
(3 - 1)*(1 + 1) + 1 = 2*2 + 1 = 5
max(5, 3) = 5
```

Answer = 5

Schedule:

`A _ A _ A`

Time: 1 (A), 2 (idle), 3 (A), 4 (idle), 5 (A)

→ 5 units.

Perfect.

Term	Meaning

Key Insight Summary

Term	Meaning
`(maxFreq - 1)`	Number of **full gaps** between the most frequent task
`(n + 1)`	Each gap + task needs at least `n+1` time (1 for task, n for cooldown)
`+ countMaxFreq`	The **last occurrence** of each max-frequency task goes at the end
`max(..., total_tasks)`	Can't take less time than total number of tasks

When Does Each Term Win?

Case	Which term wins?
Few unique tasks, high `n`	`(maxFreq-1)*(n+1)+...` → idle-heavy
Many unique tasks	`total_tasks` → no idle, fully packed
`n = 0`	Always `total_tasks`
One dominant task	Formula term wins

Final Thoughts

The **mathematical solution** works because:

> The **most frequent task** dictates the **minimum skeleton** of the schedule.
> Other tasks either **fill the gaps** or extend the time.
> If they fill everything, total tasks wins. Otherwise, the structure wins.

You don't need to simulate — just compute the **tightest possible schedule** around the bottleneck task.

Let me know if you'd like a **diagram version** or a **whiteboard-style explanation**!

Complexity Analysis

- **Time:** O(m), where `m = len(tasks)` — for counting frequencies.
- **Space:** O(1) — since there are at most 26 unique tasks (uppercase letters), the `Counter` uses constant space.

Why Greedy Works?

We always want to **schedule the most frequent tasks first** to avoid having them pile up at the end, which would require more idle time. By structuring around the most frequent task, we create a framework that minimizes idle slots.

Even if we interleave other tasks optimally, the **bottleneck** is the most frequent task and its cooldown requirement.

Edge Cases

- `n = 0`: No cooldown → answer is `len(tasks)`
- All tasks are the same: `["A","A"], n=1` → `A idle A` → 3
- Many distinct tasks: If `n=1`, tasks = [A,B,C,D,E], then no idle needed → answer = 5

Conclusion

The key insight is **not simulating** the schedule, but using **mathematical reasoning** to compute the minimum required time.

> **Answer = max((maxFreq - 1) * (n + 1) + countMaxFreq , total_tasks)**

This elegant solution runs in linear time and is optimal.

Let me know if you'd like a **simulation-based approach** (using heap or queue) as an alternative method!

---

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)
```

```python
    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

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

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

  ```python
  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

```
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`

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

1. **num = 3**
   - len(heap) $= 0 < 2 \rightarrow$ push 3
   - heap = [3]
2. **num = 2**
   - len(heap) $= 1 < 2 \rightarrow$ push 2
   - heap = [2, 3] (heap property: min at front)
3. **num = 1**
   - len(heap) $= 2 \rightarrow$ not less than k
   - Is $1 >$ heap[0]? $\rightarrow 1 > 2$? No $\rightarrow$ 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')
```

```python
    # 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)]` → `[10, 12, 18]`

Window: `{10, 12, 18}` → 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)]` → `[12, 15, 18]`

Window: `{12, 15, 18}` → 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]
```

Iteration	Popped From	New Max	Current Window	Range	Best Range

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**.