Heap

1. K Closest Points to Origin (Medium)

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 on the X-Y plane is the Euclidean distance (not squared): **[text{Distance} = sqrt{(xi - 0)^2 + (yi - 0)^2}]**

You may return the answer in **any order**. The answer is **guaranteed** to be unique (i.e., there will not be multiple points at the same distance).

Sample Input and Output:

Example 1:

```
Input: points = [[1,3],[-2,2]], k = 1
Output: [[-2,2]]
Explanation:
The distance of the point (1, 3) from the origin is sqrt(10).
The distance of the point (-2, 2) from the origin is sqrt(8).
Since sqrt(8) < sqrt(10), the closest point is [-2, 2].
```

Example 2:

```
Input: points = [[3,3],[5,-1],[-2,4]], k = 2
Output: [[3,3],[-2,4]]
Explanation:
The distance of the points from the origin are:
(3, 3) -> sqrt(18)
(5, -1) -> sqrt(26)
```

```
(-2, 4) -> sqrt(20)
The two closest points are (3, 3) and (-2, 4).
```

Notes:

- The output points can be in any order, as long as they are among the `k` closest.

```python
# Importing the required module
from typing import List
import heapq

class Solution:
    def kClosest(self, points: List[List[int]], k: int) -> List[List[int]]:
        # Initialize an empty list to use as a max heap
        heap = []

        # Iterate over each point (x, y) in the list of points
        for (x, y) in points:
            # Calculate the negative of the squared Euclidean
            # distance from the origin
            # This is because the `heapq` module in Python
            # only supports a min-heap,
            # so we store the negative distance to simulate a max-heap
            dist = -(x*x + y*y)

            # If the heap contains fewer than k elements,
            # simply add the current point
            if len(heap) < k:
                heapq.heappush(heap, (dist, x, y))
            # Otherwise, if the heap already contains k elements,
            # add the current point and remove the farthest point from
            # the origin if the current one is closer
            else:
                # `heappushpop` pushes a new element and pops the
                # largest one (in terms of absolute value since distances
                # are negative here) in a single operation
                heapq.heappushpop(heap, (dist, x, y))

        # Extract the (x, y) coordinates from the heap and return them as a list
        return [(x, y) for (dist, x, y) in heap]
```

```python
# Create an instance of the Solution class
solution = Solution()

# Define a sample list of points and the value of k
points = [[1, 3], [-2, 2], [5, 8], [0, 1]]
k = 2

# Call the kClosest method and print the result
result = solution.kClosest(points, k)
print("The k closest points are:", result)
```

```
The k closest points are: [(-2, 2), (0, 1)]
```

2. Find K Closest Elements (Medium)

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

Note that it is the **kth largest element in sorted order**, not the kth distinct element.

Example 1:

Input:
```
nums = [3,2,1,5,6,4]
k = 2
```

Output:
```
5
```

Example 2:

Input:
```
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

Output:
```
4
```

Constraints:

- `1 <= k <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`

```
from typing import List

class Solution:
    def findClosestElements(self, arr: List[int], k: int, x: int) -> List[int]:
        # Initialize pointers for the binary search space
        left, right = 0, len(arr) - k

        # Perform binary search to find the starting index of k closest elements
        while left < right:
            # Calculate the mid-point of the current search space
            mid = left + (right - left) // 2

            # Compare the distances from x for elements at mid and mid+k
            if x - arr[mid] > arr[mid + k] - x:
                # If element at mid is further from x, move the left pointer to mid + 1
                left = mid + 1
            else:
                # If element at mid+k is further (or perfectly balanced), move right to mid
                right = mid

        # Return the subarray of k elements starting from left
        return arr[left:left + k]

# Test case
solution = Solution()

# Example test case: arr has both elements closer and further from x, k elements need to be c
arr = [1, 2, 3, 4, 5]
k = 4
x = 3
print(solution.findClosestElements(arr, k, x))  # Expected output: [1, 2, 3, 4]

# Additional test case: x is not present in the array, choosing elements closest to x
arr = [1, 3, 5, 7, 9]
k = 3
x = 4
print(solution.findClosestElements(arr, k, x))  # Expected output: [3, 5, 7]
```

```
# Explanation of additional test case:
# Possible windows of k=3 elements: [1, 3, 5], [3, 5, 7], and [5, 7, 9]
# Distances from x=4:
# - [1, 3, 5] => [3, 1, 1]
# - [3, 5, 7] => [1, 1, 3]
# - [5, 7, 9] => [1, 3, 5]
# Among these, the second subarray [3, 5, 7] is the closest to x.
```

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

3. Kth Largest Element in an Array (Medium)

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**.

Constraints:

- `1 <= k <= nums.length <= 10^4`
- `-10^4 <= nums[i] <= 10^4`

Example 1:

Input:

```
nums = [3,2,1,5,6,4]
k = 2
```

Output:

```
5
```

Example 2:

Input:

```
nums = [3,2,3,1,2,4,5,5,6]
k = 4
```

Output:

```
4
```

Explanation:

- In the first example, after sorting `nums` in descending order, we get `[6, 5, 4, 3, 2, 1]`. The 2nd largest element is `5`.
- In the second example, after sorting `nums` in descending order, we get `[6, 5, 5, 4, 3, 3, 2, 2, 1]`. The 4th largest element is `4`.

Follow-up:

Can you solve the problem without fully sorting the array? (e.g., using a heap or quickselect algorithm for better efficiency).

```python
import heapq
from typing import List

class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        # Initialize an empty min-heap
        min_heap = []

        # Iterate over each number in the given list
        for num in nums:
            # Push the current number into the min-heap
            heapq.heappush(min_heap, num)
            # If the size of the min-heap exceeds k, remove the smallest element
            if len(min_heap) > k:
                heapq.heappop(min_heap)

        # Return the top of the min-heap, which is the k-th largest element
```

```
        return min_heap[0]

# Test case
solution = Solution()
nums = [3, 2, 3, 1, 2, 4, 5, 5, 6]
k = 4
result = solution.findKthLargest(nums, k)
print(f'The {k}th largest element in the array is: {result}')  # Output should be 4
```

```
The 4th largest element in the array is: 4
```

4. Top K Frequent Words (Medium)

Given an array of strings `words` and an integer `k`, return the **k most frequent strings**.
The answer should be sorted by **frequency** from highest to lowest.
If multiple words have the same frequency, **sort them lexicographically** (in ascending order).

Example 1:

Input:

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

Output:

```
["i", "love"]
```

Explanation:
- “i” and “love” both appear 2 times.
- “leetcode” and “coding” appear 1 time.
- Lexicographical order resolves ties.

Example 2:

Input:

```
words = ["the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"], k = 4
```

Output:

```
["the", "is", "sunny", "day"]
```

Explanation:
- “the” appears 4 times.
- “is” appears 3 times.
- “sunny” appears 2 times.
- “day” appears 1 time.
- Words are sorted by frequency, and ties are resolved lexicographically.

Constraints:

- `1 <= words.length <= 500`
- `1 <= words[i].length <= 10`
- `words[i]` consists of lowercase English letters.
- `1 <= k <= number of unique words`

Follow-up:

- Can you solve it in `O(N log k)` time complexity, where `N` is the length of the array?

```python
from typing import List
from collections import Counter
from heapq import heappush, heappop

class Solution:
    def topKFrequent(self, words: List[str], k: int) -> List[str]:
        # Check if the list of words is empty
        if not words:
            return []

        # Step 1: Count frequency of each word using a Counter
        # Counter creates a dictionary with words as keys and their frequencies as values
```

```
        count = Counter(words)

        # Step 2: Use a heap to keep track of the top k frequent elements
        # Python's heap is implemented as a min-heap, to simulate a max-heap, we use negative
        heap = []
        for word, freq in count.items():
            # Push each word with its negative frequency onto the heap
            # This allows us to use the smallest frequency on the heap as the highest freque
            heappush(heap, (-freq, word))

        # Step 3: Extract the top k elements from the heap
        # We do this by popping from the heap 'k' times
        result = [heappop(heap)[1] for _ in range(k)]
        return result

# Test case to illustrate how the function works
solution = Solution()
words = ["i", "love", "leetcode", "i", "love", "coding", "coding", "coding"]
k = 2
print(solution.topKFrequent(words, k))  # Expected output: ['coding', 'i']

# Explanation of test case:
# The frequencies are: "i" -> 2, "love" -> 2, "leetcode" -> 1, "coding" -> 3
# The top 2 frequent words are "coding" (3 times) and "i" (2 times).
```

```
['coding', 'i']
```

5. Find Median from Data Stream (Hard)

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 average of the two middle values.

- For example:
  - `[2, 3, 4]` → the median is `3`.
  - `[2, 3]` → the median is `(2 + 3) / 2 = 2.5`.

Design a data structure that efficiently supports the following operations:

1. `addNum(int num)`: Add an integer `num` to the data stream.
2. `findMedian()`: Return the median of all elements so far.

You should implement the `MedianFinder` class:

- **MedianFinder()** initializes the data structure.
- **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^-5` of the actual answer will be accepted.

Example:

Input:

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

Output:

```
[null, null, null, 1.5, null, 2.0]
```

Explanation:

```
MedianFinder medianFinder = new MedianFinder();
medianFinder.addNum(1);    // arr = [1]
medianFinder.addNum(2);    // arr = [1, 2]
medianFinder.findMedian(); // return 1.5 (average of [1, 2])
medianFinder.addNum(3);    // arr = [1, 2, 3]
medianFinder.findMedian(); // return 2.0 (middle element)
```

```
import heapq

class MedianFinder:
    def __init__(self):
        # Two heaps: one max heap (lower) and one min heap (upper)
        self.lower = []  # Max heap using negative values
        self.upper = []  # Min heap

    def addNum(self, num: int) -> None:
        # Add to min heap first and then to max heap
        heapq.heappush(self.upper, num)
        heapq.heappush(self.lower, -heapq.heappop(self.upper))  # Move the smallest element

        # Balance the heaps if necessary
```

```
        if len(self.lower) > len(self.upper):
            heapq.heappush(self.upper, -heapq.heappop(self.lower))

    def findMedian(self) -> float:
        if len(self.lower) == len(self.upper):
            # If even, median is the average of two middle elements
            return (self.upper[0] - self.lower[0]) / 2.0
        else:
            # If odd, median is the middle element
            return float(self.upper[0])

# Test case
def test_median_finder():
    mf = MedianFinder()
    numbers = [1, 3, 2, 4]

    for num in numbers:
        mf.addNum(num)
        print(f"Added {num}, current median: {mf.findMedian()}")

# Run the test case
test_median_finder()
```

```
Added 1, current median: 1.0
Added 3, current median: 2.0
Added 2, current median: 2.0
Added 4, current median: 2.5
```

6. Merge k Sorted Lists (Hard)

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

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

Example 1:

Input:

```
lists = [
    1 -> 4 -> 5,
```

```
    1 -> 3 -> 4,
    2 -> 6
]
```

Output:

```
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6
```

Example 2:

Input:

```
lists = []
```

Output:

```
[]
```

Example 3:

Input:

```
lists = [[]]
```

Output:

```
[]
```

Constraints:

- `k == lists.length`
- `0 <= k <= 10^4`
- `0 <= lists[i].length <= 500`
- `-10^4 <= lists[i][j] <= 10^4`
- `lists[i]` is sorted in **ascending order**.
- The sum of all `lists[i].length` will not exceed `10^4`.

```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]]) -> Optional[ListNode]:
        """
        This function merges k sorted linked lists and returns it as one sorted list.

        :param lists: A list of ListNode objects, where each ListNode is the head of a sorte
        :return: A ListNode that is the head of the merged sorted linked list.
        """
        # Create a dummy head for the merged list
        dummy = ListNode()
        # This will point to the last node in the merged linked list, starts at dummy
        current = dummy
        # Min-heap to efficiently get the node with the smallest value
        heap = []

        # Edge case: if the input list is empty, return None
        if not lists:
            return None

        # Initialize the heap with the first node of each list (if not empty)
        for i, node in enumerate(lists):
            if node:
                # Push tuple (node value, list index, node itself) into the heap
                heapq.heappush(heap, (node.val, i, node))

        # While the heap is not empty, process nodes
        while heap:
            # Pop the smallest item from heap (value, index and node)
            val, i, node = heapq.heappop(heap)
            # Append the extracted node to the resulting merged list
            current.next = node
            # Move the current pointer
            current = current.next
```

```
            # If there is a next node in the list, push it into the heap
            if node.next:
                heapq.heappush(heap, (node.next.val, i, node.next))

        # Return the head of the merged linked list, which is next of dummy
        return dummy.next

# Helper function to create linked list from list of values
def create_linked_list(arr):
    dummy = ListNode()
    current = dummy
    for val in arr:
        current.next = ListNode(val)
        current = current.next
    return dummy.next

# Helper function to print linked list
def print_linked_list(node):
    while node:
        print(node.val, end=' -> ')
        node = node.next
    print('None')

# Test case
list1 = create_linked_list([1, 4, 5])
list2 = create_linked_list([1, 3, 4])
list3 = create_linked_list([2, 6])

lists = [list1, list2, list3]

# Create a Solution object and use it to merge the lists
solution = Solution()
merged_list = solution.mergeKLists(lists)

# Print the merged linked list
print_linked_list(merged_list)  # Expected output: 1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6 -> No
```

```
1 -> 1 -> 2 -> 3 -> 4 -> 4 -> 5 -> 6 -> None
```

7. Task Scheduler (Medium)

You are given an array of CPU tasks represented by characters `tasks`, where each task is a single uppercase English letter. Tasks must be executed **in the given order**, but they may require a **cooldown interval** of `n` units between two **same tasks**.

That is, there must be at least `n` units of time between two occurrences of the same task.

Return the **minimum number of intervals** required to complete all the tasks.

Constraints:

- `1 <= tasks.length <= 10^4`
- `tasks[i]` is an uppercase English letter.
- `0 <= n <= 100`

Example 1:

Input:

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

Output:

```
8
```

Explanation:

One possible sequence of task execution is:

```
A -> B -> idle -> A -> B -> idle -> A -> B
```

- A cooldown interval of `2` is required between the same tasks `A` and `A` or `B` and `B`.
- Total intervals = `8`.

Example 2:

Input:

```
tasks = ["A", "C", "A", "B", "D", "A"]
n = 1
```

Output:

```
6
```

Explanation:

The tasks can be executed as:

```
A -> C -> A -> B -> D -> A
```

- The cooldown interval of `1` is respected.

Example 3:

Input:

```
tasks = ["A", "A", "A", "B", "B", "C"]
n = 3
```

Output:

```
7
```

Explanation:

The tasks can be executed as:

```
A -> B -> C -> A -> idle -> idle -> A
```

- Cooldown interval = `3` units.
- Total intervals = `7`.

```python
from collections import Counter
from heapq import heappop, heappush, heapify
from typing import List

class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        # Corner case: If there are no tasks, return 0
        if not tasks:
            return 0

        # Count the frequency of each task
        task_count = Counter(tasks)

        # Create a max heap based on the negative count (since Python's heapq is a min-heap)
        max_heap = [-count for count in task_count.values()]
        heapify(max_heap)

        # Initialize total time counter
        total_time = 0

        while max_heap:
            # List to keep track of tasks completed in each cycle of size 'n+1'
            temp = []
            # Simulate running tasks for 'n+1' time slots
            idle_time = n + 1

            # Process the tasks in the current cycle
            while idle_time > 0 and max_heap:
                # Task executed, increment total time
                total_time += 1
                # Reduce idle slots
                idle_time -= 1

                # Pop the most frequent task
                count = -heappop(max_heap)

                # If there are remaining instances of this task, add back reduced by one
                if count > 1:
                    temp.append(count - 1)

            # Push the remaining tasks back into the heap
            for count in temp:
```

```
                heappush(max_heap, -count)

            # If there are still tasks left in the heap, the CPU has to be idle for the rema
            if max_heap:
                total_time += idle_time

        return total_time

# Test case
solution = Solution()
tasks = ['A', 'A', 'A', 'B', 'B', 'B']  # Example tasks
n = 2  # Cooling period

print(solution.leastInterval(tasks, n))  # Expected output: 8
```

```
8
```

8. Smallest Range Covering Elements from K Lists (Hard)

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

The range `[a, b]` is smaller than the range `[c, d]` if: - `b - a < d - c` **or** - `a < c` if `b - a == d - c`.

Example 1:

Input:

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

Output:

```
[20, 24]
```

Explanation: - The smallest range is `[20, 24]`, which includes 20 (from list 2), 24 (from list 1), and 22 (from list 3). - It satisfies the condition of having at least one number from each list.

Example 2:

Input:

```
nums = [
    [1, 2, 3],
    [1, 2, 3],
    [1, 2, 3]
]
```

Output:

```
[1, 1]
```

Explanation: - The smallest range is `[1, 1]`, as all lists contain the number `1`.

Constraints:

- `nums.length == k`
- `1 <= k <= 3500`
- `1 <= nums[i].length <= 50`
- `-10^5 <= nums[i][j] <= 10^5`
- `nums[i]` is sorted in **non-decreasing** order.

```python
import heapq

def smallestRange(nums):
    # Initialize the heap
    min_heap = []
    max_value = float('-inf')

    # Insert the first element of each list in the min_heap
    for i in range(len(nums)):
        heapq.heappush(min_heap, (nums[i][0], i, 0))
        max_value = max(max_value, nums[i][0])
```

```
    # To store the result range
    smallest_range = [float('-inf'), float('inf')]

    while min_heap:
        min_value, list_index, element_index = heapq.heappop(min_heap)

        # Update the range if it's smaller
        if max_value - min_value < smallest_range[1] - smallest_range[0]:
            smallest_range = [min_value, max_value]

        # Check if the current list is exhausted
        if element_index + 1 == len(nums[list_index]):
            break

        # Insert the next element from the current list into the heap
        next_value = nums[list_index][element_index + 1]
        heapq.heappush(min_heap, (next_value, list_index, element_index + 1))

        # Update the max_value
        max_value = max(max_value, next_value)

    return smallest_range

# Example Usage:
nums = [[4,10,15,24,26], [0,9,12,20], [5,18,22,30]]
print(smallestRange(nums))  # Output: [20, 24]
```

```
[20, 24]
```