23. Merge k Sorted Lists

You are given an array of `k` linked-lists `lists`, 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]
Explanation: The linked-lists are:
[
  1->4->5,
  1->3->4,
  2->6
]
merging them into one sorted list:
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 `lists[i].length` will not exceed `10^4`.

```
# Definition for singly-linked list.
# class ListNode:
#     def __init__(self, val=0, next=None):
#         self.val = val
#         self.next = next
import heapq
class Solution:
    def mergeKLists(self, lists: List[Optional[ListNode]]) -> Optional[ListNode]:
        heap = []
        i = 0
        for head in lists:
            while head:
                heapq.heappush(heap,(head.val,i,head))
                head = head.next
                i+=1

        temp = None
        head = None
        while heap:
            node = heapq.heappop(heap)[2]
            if not head:
                temp = node
                head = node
            else:
                temp.next =node
                temp = temp.next
        return head
```

215. Kth Largest Element in an Array

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.

Can you solve it without sorting?

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 <= 10^5`
- `-10^4 <= nums[i] <= 10^4`

```
import heapq
class Solution:
    def findKthLargest(self, nums: List[int], k: int) -> int:
        heap = []
        for i in nums:
            heapq.heappush(heap,-i)

        for i in range(k-1):
            heapq.heappop(heap)
        return -heapq.heappop(heap)
```

295. Find Median from Data Stream

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.

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

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^-5` of the actual answer will be accepted.

Example 1:

```
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 (i.e., (1 + 2) / 2)
medianFinder.addNum(3);    // arr[1, 2, 3]
medianFinder.findMedian(); // return 2.0
```

Constraints:

- -10^5 <= num <= 10^5
- There will be at least one element in the data structure before calling `findMedian`.
- At most `5 * 10^4` calls will be made to `addNum` and `findMedian`.

Follow up:

- If all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?
- If `99%` of all integer numbers from the stream are in the range `[0, 100]`, how would you optimize your solution?

```
import heapq

class MedianFinder:

    def __init__(self):
        self.small = []
        self.large = []

    def addNum(self, num: int) -> None:
        #which heap to push into
        if not self.small or num <= -self.small[0]:
            #push small (max-heap) as negative
            heapq.heappush(self.small, -num)
        else:
            #push into large (min-heap)
            heapq.heappush(self.large, num)
        #rebalance the heaps
        if len(self.small) > len(self.large) + 1:
            #move max small to large
            val = -heapq.heappop(self.small)
            heapq.heappush(self.large, val)
        elif len(self.large) > len(self.small) + 1:
            #move min large to small
            val = heapq.heappop(self.large)
            heapq.heappush(self.small, -val)

    def findMedian(self) -> float:
        if len(self.small) == len(self.large):
            if not self.small:  # Edge case: no elements
                return 0.0
            return (-self.small[0] + self.large[0]) / 2
        elif len(self.small) > len(self.large):
            #small more element
            return float(-self.small[0])
        else:
            #large element
            return float(self.large[0])
```

347. Top K Frequent Elements

Given an integer array `nums` and an integer `k`, return *the* `k` *most frequent elements*. You may return the answer in **any order**.

Example 1:

```
Input: nums = [1,1,1,2,2,3], k = 2
Output: [1,2]
```

Example 2:

```
Input: nums = [1], k = 1
Output: [1]
```

Constraints:

- `1 <= nums.length <= 10^5`
- `-10^4 <= nums[i] <= 10^4`
- `k` is in the range `[1, the number of unique elements in the array]`.
- It is **guaranteed** that the answer is **unique**.

Follow up: Your algorithm's time complexity must be better than `O(n log n)`, where n is the array's size.

```
from collections import Counter
import heapq
class Solution:
    def topKFrequent(self,nums, k):
        c = Counter(nums)
        return [x for x, _ in heapq.nlargest(k, c.items(), key=lambda x: x[1])]
```

---

355. Design Twitter

Design a simplified version of Twitter where users can post tweets, follow/unfollow another user, and is able to see the `10` most recent tweets in the user's news feed.

Implement the `Twitter` class:

- `Twitter()` Initializes your twitter object.
- `void postTweet(int userId, int tweetId)` Composes a new tweet with ID `tweetId` by the user `userId`. Each call to this function will be made with a unique `tweetId`.
- `List<Integer> getNewsFeed(int userId)` Retrieves the `10` most recent tweet IDs in the user's news feed. Each item in the news feed must be posted by users who the user followed or by the user

themself. Tweets must be **ordered from most recent to least recent**.

- `void follow(int followerId, int followeeId)` The user with ID `followerId` started following the user with ID `followeeId`.
- `void unfollow(int followerId, int followeeId)` The user with ID `followerId` started unfollowing the user with ID `followeeId`.

Example 1:

```
Input
["Twitter", "postTweet", "getNewsFeed", "follow", "postTweet", "getNewsFeed", "unfoll
[[], [1, 5], [1], [1, 2], [2, 6], [1], [1, 2], [1]]
Output
[null, null, [5], null, null, [6, 5], null, [5]]

Explanation
Twitter twitter = new Twitter();
twitter.postTweet(1, 5); // User 1 posts a new tweet (id = 5).
twitter.getNewsFeed(1);  // User 1's news feed should return a list with 1 tweet id -
twitter.follow(1, 2);    // User 1 follows user 2.
twitter.postTweet(2, 6); // User 2 posts a new tweet (id = 6).
twitter.getNewsFeed(1);  // User 1's news feed should return a list with 2 tweet ids
twitter.unfollow(1, 2);  // User 1 unfollows user 2.
twitter.getNewsFeed(1);  // User 1's news feed should return a list with 1 tweet id -
```

Constraints:

- `1 <= userId, followerId, followeeId <= 500`
- `0 <= tweetId <= 10^4`
- All the tweets have **unique** IDs.
- At most `3 * 10^4` calls will be made to `postTweet`, `getNewsFeed`, `follow`, and `unfollow`.
- A user cannot follow himself.

```
from collections import defaultdict, deque

class Twitter:

    def __init__(self):
        self.follows = defaultdict(set)
        self.feed = deque()

    def postTweet(self, userId: int, tweetId: int) -> None:
        self.feed.appendleft((userId, tweetId))

    def getNewsFeed(self, userId: int) -> List[int]:
        return [tweetId for user, tweetId in self.feed if userId == user or user in
self.follows[userId]][:10]

    def follow(self, followerId: int, followeeId: int) -> None:
        self.follows[followerId].add(followeeId)

    def unfollow(self, followerId: int, followeeId: int) -> None:
        self.follows[followerId].discard(followeeId)
```

---

621. Task Scheduler

You are given an array of CPU `tasks`, each labeled with a letter from A to Z, and a number `n`. Each CPU interval can be idle or allow the completion of one task. Tasks can be completed in any order, but there's a constraint: there has to be a gap of **at least** `n` intervals between two tasks with the same label.

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

Example 1:

> **Input:** tasks = ["A","A","A","B","B","B"], n = 2
>
> **Output:** 8
>
> **Explanation:** A possible sequence is: A -> B -> idle -> A -> B -> idle -> A -> B.
>
> After completing task A, you must wait two intervals before doing A again. The same applies to task B. In the 3rd interval, neither A nor B can be done, so you idle. By the 4th interval, you can do A again as 2 intervals have passed.

Example 2:

> **Input:** tasks = ["A","C","A","B","D","B"], n = 1
>
> **Output:** 6
>
> **Explanation:** A possible sequence is: A -> B -> C -> D -> A -> B.
>
> With a cooling interval of 1, you can repeat a task after just one other task.

Example 3:

> **Input:** tasks = ["A","A","A", "B","B","B"], n = 3
>
> **Output:** 10
>
> **Explanation:** A possible sequence is: A -> B -> idle -> idle -> A -> B -> idle -> idle -> A -> B.
>
> There are only two types of tasks, A and B, which need to be separated by 3 intervals. This leads to idling twice between repetitions of these tasks.

Constraints:

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

```
from collections import deque,defaultdict,Counter
class Solution:
    def leastInterval(self, tasks: List[str], n: int) -> int:
        cnt = Counter(tasks)
        maxHeap = [-num for num in cnt.values()]
        heapq.heapify(maxHeap)
        time = 0
        que = deque()
        while que or maxHeap:
            time+=1
            if maxHeap:
                count  = heapq.heappop(maxHeap)+1
                if count:que.append([count,time+n])
            if que and que[0][1] <= time:
                ele = que.popleft()[0]
                heapq.heappush(maxHeap,ele)
        return time
```

---

703. Kth Largest Element in a Stream

You are part of a university admissions office and need to keep track of the `kth` highest test score from applicants in real-time. This helps to determine cut-off marks for interviews and admissions dynamically as new applicants submit their scores.

You are tasked to implement a class which, for a given integer `k`, maintains a stream of test scores and continuously returns the `k` th highest test score **after** a new score has been submitted. More specifically, we are looking for the `k` th highest score in the sorted list of all scores.

Implement the `KthLargest` class:

- `KthLargest(int k, int[] nums)` Initializes the object with the integer `k` and the stream of test scores `nums`.
- `int add(int val)` Adds a new test score `val` to the stream and returns the element representing the k^{th} largest element in the pool of test scores so far.

Example 1:

Input:
["KthLargest", "add", "add", "add", "add", "add"]
[[3, [4, 5, 8, 2]], [3], [5], [10], [9], [4]]

Output: [null, 4, 5, 5, 8, 8]

Explanation:

KthLargest kthLargest = new KthLargest(3, [4, 5, 8, 2]);
kthLargest.add(3); // return 4
kthLargest.add(5); // return 5
kthLargest.add(10); // return 5
kthLargest.add(9); // return 8
kthLargest.add(4); // return 8

Example 2:

Input:
["KthLargest", "add", "add", "add", "add"]
[[4, [7, 7, 7, 7, 8, 3]], [2], [10], [9], [9]]

Output: [null, 7, 7, 7, 8]

Explanation:

KthLargest kthLargest = new KthLargest(4, [7, 7, 7, 7, 8, 3]);
kthLargest.add(2); // return 7

kthLargest.add(10); // return 7
kthLargest.add(9); // return 7
kthLargest.add(9); // return 8

Constraints:

- `0 <= nums.length <= 10^4`
- `1 <= k <= nums.length + 1`
- `-10^4 <= nums[i] <= 10^4`
- `-10^4 <= val <= 10^4`
- At most `10^4` calls will be made to `add`.

```
import heapq
class KthLargest:

    def __init__(self, k: int, nums: List[int]):
        self.size = k
        self.min_heap = nums
        heapq.heapify(self.min_heap)


    def add(self, val: int) -> int:
        heapq.heappush(self.min_heap,val)
        while len(self.min_heap) > self.size:
            heapq.heappop(self.min_heap)
        return self.min_heap[0]
```

846. Hand of Straights

Alice has some number of cards and she wants to rearrange the cards into groups so that each group is of size `groupSize`, and consists of `groupSize` consecutive cards.

Given an integer array `hand` where `hand[i]` is the value written on the `i^th` card and an integer `groupSize`, return `true` if she can rearrange the cards, or `false` otherwise.

Example 1:

```
Input: hand = [1,2,3,6,2,3,4,7,8], groupSize = 3
Output: true
Explanation: Alice's hand can be rearranged as [1,2,3],[2,3,4],[6,7,8]
```

Example 2:

```
Input: hand = [1,2,3,4,5], groupSize = 4
Output: false
Explanation: Alice's hand can not be rearranged into groups of 4.
```

Constraints:

- `1 <= hand.length <= 10^4`
- `0 <= hand[i] <= 10^9`
- `1 <= groupSize <= hand.length`

Note: This question is the same as 1296: https://leetcode.com/problems/divide-array-in-sets-of-k-consecutive-numbers/ (https://leetcode.com/problems/divide-array-in-sets-of-k-consecutive-numbers/)

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

class Solution:
    def isNStraightHand(self, hand: List[int], groupSize: int) -> bool:
        if len(hand) % groupSize != 0:
            return False

        count = Counter(hand)
        min_heap = list(count.keys())
        heapq.heapify(min_heap)

        while min_heap:
            first = min_heap[0]
            for i in range(groupSize):
                num = first + i
                if count[num] == 0:
                    return False
                count[num] -= 1
                if count[num] == 0:
                    if num != min_heap[0]:
                        return False  # out-of-order deletion is not allowed
                    heapq.heappop(min_heap)

        return True
```