<https://leetcode.com/problems/smallest-range-covering-elements-from-k-lists/description/>

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] is smaller than 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:

List 1: [4, 10, 15, 24,26], 24 is in range [20,24].

List 2: [0, 9, 12, 20], 20 is in range [20,24].

List 3: [5, 18, 22, 30], 22 is in range [20,24].

**Example 2:**

Input: nums = [[1,2,3],[1,2,3],[1,2,3]]

Output: [1,1]

**Constraints:**

* nums.length == k
* 1 <= k <= 3500
* 1 <= nums[i].length <= 50
* -105 <= nums[i][j] <= 105
* nums[i] is sorted in **non-decreasing** order.

**Attempt 1: 2023-11-24**

**Solution 1: Not fixed length Sliding Window (120 min)**

class Solution {

public int[] smallestRange(List<List<Integer>> nums) {

int k = nums.size();

// Merge all k original groups into one list

List<int[]> list = new ArrayList<>();

// Element on list contains both value and belong to which

// original group information [val, group id]

for(int i = 0; i < k; i++) {

List<Integer> tmp = nums.get(i);

for(int num : tmp) {

list.add(new int[]{num, i});

}

}

// Sort the list with element value

Collections.sort(list, (a, b) -> a[0] - b[0]);

// Sliding window start index

int i = 0;

// How many original group covered in sliding window

int count = 0;

// Original group id frequency in sliding window tracking

Map<Integer, Integer> freq = new HashMap<>();

int[] result = new int[2];

int diff = Integer.MAX\_VALUE;

for(int j = 0; j < list.size(); j++) {

int group\_id = list.get(j)[1];

int prev\_group\_id\_freq = freq.getOrDefault(group\_id, 0);

// The group id not in sliding window yet, treat as new member

if(prev\_group\_id\_freq == 0) {

count++;

}

freq.put(group\_id, prev\_group\_id\_freq + 1);

// When reach target as covering all k original groups id

// into sliding window, we try to shrink the window and

// check if able to get smaller range

while(count == k) {

if(list.get(j)[0] - list.get(i)[0] < diff) {

diff = list.get(j)[0] - list.get(i)[0];

result[0] = list.get(i)[0];

result[1] = list.get(j)[0];

}

int shrink\_group\_id = list.get(i)[1];

int shrink\_group\_id\_freq = freq.get(shrink\_group\_id) - 1;

if(shrink\_group\_id\_freq == 0) {

count--;

}

freq.put(shrink\_group\_id, shrink\_group\_id\_freq);

i++;

}

}

return result;

}

}

Time Complexity: O(N\*logN + N), the time complexity of Collections.sort() is O(N\*logN)

Space Complexity: O(N)

**Refer to**

<https://grandyang.com/leetcode/632/>

这道题给了我们一些数组，都是排好序的，让求一个最小的范围，使得这个范围内至少会包括每个数组中的一个数字。虽然每个数组都是有序的，但是考虑到他们之间的数字差距可能很大，所以最好还是合并成一个数组统一处理比较好，但是合并成一个大数组还需要保留其原属数组的序号，所以大数组中存pair对，同时保存数字和原数组的序号。然后重新按照数字大小进行排序，这样问题实际上就转换成了求一个最小窗口，使其能够同时包括所有数组中的至少一个数字。这不就变成了那道 [Minimum Window Substring](http://www.cnblogs.com/grandyang/p/4340948.html)。所以说啊，这些题目都是换汤不换药的，总能变成我们见过的类型。这里用两个指针 left 和 right 来确定滑动窗口的范围，还要用一个 HashMap 来建立每个数组与其数组中数字出现的个数之间的映射，变量 cnt 表示当前窗口中的数字覆盖了几个数组，diff 为窗口的大小，让 right 向右滑动，然后判断如果 right 指向的数字所在数组没有被覆盖到，cnt 自增1，然后 HashMap 中对应的数组出现次数自增1，然后循环判断如果 cnt 此时为k(数组的个数)且 left 不大于 right，那么用当前窗口的范围来更新结果，然后此时想缩小窗口，通过将 left 向右移，移动之前需要减小 HashMap 中的映射值，因为去除了数字，如果此时映射值为0了，说明有个数组无法覆盖到了，cnt 就要自减1。这样遍历后就能得到最小的范围了，参见代码如下：

class Solution {

public:

vector<int> smallestRange(vector<vector<int>>& nums) {

vector<int> res;

vector<pair<int, int>> v;

unordered\_map<int, int> m;

for (int i = 0; i < nums.size(); ++i) {

for (int num : nums[i]) {

v.push\_back({num, i});

}

}

sort(v.begin(), v.end());

int left = 0, n = v.size(), k = nums.size(), cnt = 0, diff = INT\_MAX;

for (int right = 0; right < n; ++right) {

if (m[v[right].second] == 0) ++cnt;

++m[v[right].second];

while (cnt == k && left <= right) {

if (diff > v[right].first - v[left].first) {

diff = v[right].first - v[left].first;

res = {v[left].first, v[right].first};

}

if (--m[v[left].second] == 0) --cnt;

++left;

}

}

return res;

}

};

**Refer to**

<https://leetcode.com/problems/smallest-range-covering-elements-from-k-lists/solutions/624185/c-sliding-window-easy-solution/>

**Firstly we merge k groups to one group, each number record it's group id,**

**e.g:**

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

**after merged we got：[(0, 1), (4, 0), (5, 2), (9, 1), (10, 0), (12, 1), (15, 0), (18, 2), (20, 1), (22, 2), (24, 0), (26, 0), (30, 2)]**

**and see only group, it's[1, 0, 2, 1, 0, 1, 0, 2, 1, 2, 0, 0, 2]**

**we can slide window by group id when current groups satisfies condition and record min range.**

[1 0 2] 2 1 0 1 0 2 1 2 0 0 2 [0, 5]

1 [0 2 1] 1 0 1 0 2 1 2 0 0 2 [0, 5]

1 0 [2 1 0] 0 1 0 2 1 2 0 0 2 [0, 5]

1 0 [2 1 0 1] 1 0 2 1 2 0 0 2 [0, 5]

1 0 [2 1 0 1 0] 0 2 1 2 0 0 2 [0, 5]

1 0 2 1 0 [1 0 2] 2 1 2 0 0 2 [0, 5]

1 0 2 1 0 1 [0 2 1] 1 2 0 0 2 [0, 5]

1 0 2 1 0 1 [0 2 1 2] 2 0 0 2 [0, 5]

1 0 2 1 0 1 0 2 [1 2 0] 0 0 2 [20, 24]

1 0 2 1 0 1 0 2 [1 2 0 0] 0 2 [20, 24]

1 0 2 1 0 1 0 2 [1 2 0 0 2] 2 [20, 24]

**C++:**

class Solution {

public:

vector<int> smallestRange(vector<vector<int>>& nums) {

vector<pair<int, int>> ordered; // (number, group)

for (size\_t k = 0; k < nums.size(); ++k)

for (auto n: nums[k]) ordered.push\_back({n, k});

sort(ordered.begin(), ordered.end());

int i = 0, k = 0;

vector<int> ans;

unordered\_map<int, int> count;

for (size\_t j = 0; j < ordered.size(); ++j) {

if (! count[ordered[j].second]++) ++k;

if (k == nums.size()) {

while (count[ordered[i].second] > 1) --count[ordered[i++].second]; // minialize range

if (ans.empty() || ans[1] - ans[0] > ordered[j].first - ordered[i].first) {

ans = vector<int>{ordered[i].first, ordered[j].first};

}

}

}

return ans;

}

};

**Java:**

Java version based on OP's code. Time Complexity is O(nlogn + n). It can be improved to O(nlogk) by using a Priority Queue to create merged nums instead of sorting it.

public int[] smallestRange(List<List<Integer>> kLists) {

List<int[]> mergedNums = new ArrayList<>();

for (int k=0; k < kLists.size(); k++) {

for (int i : kLists.get(k)) {

mergedNums.add(new int[]{i, k});

}

}

Collections.sort(mergedNums, (a,b) -> Integer.compare(a[0],b[0]));

int[] ans = null;

// i is window start

// k is num of lists covered in current window

int i=0, k=0;

Map<Integer, Integer> listMemberCount = new HashMap<>();

for (int j=0; j<mergedNums.size(); j++) {

int windowEndList = mergedNums.get(j)[1];

int prevCount = listMemberCount.getOrDefault(windowEndList, 0);

listMemberCount.put(windowEndList, prevCount + 1);

if (prevCount==0) {

k++;

}

if (k==kLists.size()) {

while (listMemberCount.get(mergedNums.get(i)[1]) > 1) {

int windowStartList = mergedNums.get(i++)[1];

listMemberCount.put(windowStartList, listMemberCount.get(windowStartList)-1);

}

if (ans == null || ans[1]-ans[0] > mergedNums.get(j)[0]-mergedNums.get(i)[0]) {

ans = new int[] {mergedNums.get(i)[0], mergedNums.get(j)[0]};

}

}

}

return ans;

}

**Solution 2: Priority Queue (30 min)**

class Solution {

public int[] smallestRange(List<List<Integer>> nums) {

int k = nums.size();

int cur\_max = Integer.MIN\_VALUE;

// Element on PriorityQueue will store value, its list group id

// and its index in group: {value, group id, index in group}

PriorityQueue<int[]> minPQ = new PriorityQueue<>((a, b) -> a[0] - b[0]);

// Start with all first number on each group

for(int i = 0; i < k; i++) {

int val = nums.get(i).get(0);

minPQ.offer(new int[]{val, i, 0});

cur\_max = Math.max(cur\_max, val);

}

int[] result = new int[2];

int diff = Integer.MAX\_VALUE;

// The condition is not minPQ.isEmpty() because at any moment, each

// sorted list group will only have one number stored on minPQ,

// when pull out one number will add the next number from the same

// list group, like how L23/P15.1.Merge k Sorted Lists works, if

// not able to find next number means that list group exhausted, we

// cannot satisfy "includes at least one number from each of the k

// lists" anymore, that's the end of loop

while(minPQ.size() == k) {

int[] e = minPQ.poll();

int val = e[0];

int group\_id = e[1];

int val\_group\_idx = e[2];

if(cur\_max - val < diff) {

diff = cur\_max - val;

result[0] = val;

result[1] = cur\_max;

}

// Add the next number into minPQ from the same list group

if(val\_group\_idx + 1 < nums.get(group\_id).size()) {

val\_group\_idx++;

int next\_val = nums.get(group\_id).get(val\_group\_idx);

minPQ.offer(new int[]{next\_val, group\_id, val\_group\_idx});

if(cur\_max < next\_val) {

cur\_max = next\_val;

}

}

}

return result;

}

}

Time complexity : O(N∗log(M)). Heapification of M elements requires O(log(M)) time. This step could be done for all the elements of the given lists in the worst case. Here, N refers to the total number of elements in all the lists. M refers to the total number of lists.

Space complexity : O(M). next array of size M is used. A Min-Heap with M elements is also used.

**Refer to**

<https://grandyang.com/leetcode/632/>

这道题还有一种使用 priority\_queue 来做的，优先队列默认情况是最大堆，但是这道题我们需要使用最小堆，重新写一下 comparator 就行了。解题的主要思路很上面的解法很相似，只是具体的数据结构的使用上略有不同，这 curMax 表示当前遇到的最大数字，用一个 idx 数组表示每个 list 中遍历到的位置，然后优先队列里面放一个pair，是数字和其所属list组成的对儿。遍历所有的list，将每个 list 的首元素和该 list 序号组成 pair 放入队列中，然后 idx 数组中每个位置都赋值为1，因为0的位置已经放入队列了，所以指针向后移一个位置，还要更新当前最大值 curMax。此时 queue 中是每个 list 各有一个数字，由于是最小堆，所以最小的数字就在队首，再加上最大值 curMax，就可以初始化结果 res 了。然后进行循环，注意这里循环的条件不是队列不为空，而是当某个 list 的数字遍历完了就结束循环，因为范围要 cover 每个 list 至少一个数字。所以 while 循环条件即是队首数字所在的 list 的遍历位置小于该 list 的总个数，在循环中，取出队首数字所在的 list 序号t，然后将该 list 中下一个位置的数字和该 list 序号t组成 pair，加入队列中，然后用这个数字更新 curMax，同时 idx 中t对应的位置也自增1。现在来更新结果 res，如果结果 res 中两数之差大于 curMax 和队首数字之差，则更新结果 res，参见代码如下：

class Solution {

public:

vector<int> smallestRange(vector<vector<int>>& nums) {

int curMax = INT\_MIN, n = nums.size();

vector<int> idx(n, 0);

auto cmp = [](pair<int, int>& a, pair<int, int>& b) {return a.first > b.first;};

priority\_queue<pair<int, int>, vector<pair<int, int>>, decltype(cmp) > q(cmp);

for (int i = 0; i < n; ++i) {

q.push({nums[i][0], i});

idx[i] = 1;

curMax = max(curMax, nums[i][0]);

}

vector<int> res{q.top().first, curMax};

while (idx[q.top().second] < nums[q.top().second].size()) {

int t = q.top().second; q.pop();

q.push({nums[t][idx[t]], t});

curMax = max(curMax, nums[t][idx[t]]);

++idx[t];

if (res[1] - res[0] > curMax - q.top().first) {

res = {q.top().first, curMax};

}

}

return res;

}

};

**Refer to**

<https://leetcode.com/problems/smallest-range-covering-elements-from-k-lists/solutions/104893/java-code-using-priorityqueue-similar-to-merge-k-array/>

Image you are merging k sorted array using a heap. Then every time you pop the smallest element out and add the next element of that array to the heap. By keep doing this, you will have the smallest range.

public int[] smallestRange(int[][] nums) {

PriorityQueue<Element> pq = new PriorityQueue<Element>(new Comparator<Element>() {

public int compare(Element a, Element b) {

return a.val - b.val;

}

});

int min = Integer.MAX\_VALUE, max = Integer.MIN\_VALUE;

for (int i = 0; i < nums.length; i++) {

Element e = new Element(i, 0, nums[i][0]);

pq.offer(e);

max = Math.max(max, nums[i][0]);

}

int range = Integer.MAX\_VALUE;

int start = -1, end = -1;

while (pq.size() == nums.length) {

Element curr = pq.poll();

if (max - curr.val < range) {

range = max - curr.val;

start = curr.val;

end = max;

}

if (curr.idx + 1 < nums[curr.row].length) {

curr.idx = curr.idx + 1;

curr.val = nums[curr.row][curr.idx];

pq.offer(curr);

if (curr.val > max) {

max = curr.val;

}

}

}

return new int[] { start, end };

}

class Element {

int val;

int idx;

int row;

public Element(int r, int i, int v) {

val = v;

idx = i;

row = r;

}

}

**Complexity Analysis:**

Time complexity : O(N∗log(M)). Heapification of M elements requires O(log(M)) time. This step could be done for all the elements of the given lists in the worst case. Here, N refers to the total number of elements in all the lists. M refers to the total number of lists.

Space complexity : O(M). next array of size M is used. A Min-Heap with M elements is also used.