**Assignment Priority Queue**

Q1. Given a string s, rearrange the characters of s so that any two adjacent characters are not the same.

Return any possible rearrangement of s or return "" if not possible.

Example 1:

Input: s = "aab"

Output: "aba"

Example 2:

Input: s = "aaab"

Output: ""

import java.util.HashMap;

import java.util.PriorityQueue;

import java.util.Map;

public class RearrangeString {

public static String reorganizeString(String s) {

// Step 1: Count frequencies of characters

Map<Character, Integer> frequencyMap = new HashMap<>();

for (char c : s.toCharArray()) {

frequencyMap.put(c, frequencyMap.getOrDefault(c, 0) + 1);

}

// Step 2: Add characters to max-heap based on frequencies

PriorityQueue<Map.Entry<Character, Integer>> maxHeap = new PriorityQueue<>(

(a, b) -> b.getValue() - a.getValue()

);

maxHeap.addAll(frequencyMap.entrySet());

// Step 3: Build the result string

StringBuilder result = new StringBuilder();

Map.Entry<Character, Integer> prev = null;

while (!maxHeap.isEmpty()) {

Map.Entry<Character, Integer> current = maxHeap.poll();

result.append(current.getKey());

// Decrease the frequency and add the previous entry back to the heap

if (prev != null && prev.getValue() > 0) {

maxHeap.offer(prev);

}

// Update current entry as the previous one and decrease its count

current.setValue(current.getValue() - 1);

prev = current;

}

// If the result string length is equal to the original string length, return it

if (result.length() == s.length()) {

return result.toString();

}

// Otherwise, it's not possible to rearrange the string

return "";

}

public static void main(String[] args) {

String s1 = "aab";

System.out.println("Output 1: " + reorganizeString(s1)); // Output: "aba"

String s2 = "aaab";

System.out.println("Output 2: " + reorganizeString(s2)); // Output: ""

}

}

Q2. You are given two integer arrays nums1 and nums2 sorted in ascending order and an integer k.

Define a pair (u, v) which consists of one element from the first array and one element from the second

array.

Return the k pairs (u1, v1), (u2, v2), ..., (uk, vk) with the smallest sums.

Example 1:

Input: nums1 = [1,7,11], nums2 = [2,4,6], k = 3

Output: [[1,2],[1,4],[1,6]]

Explanation: The first 3 pairs are returned from the sequence: [1,2],[1,4],[1,6],[7,2],[7,4],[11,2],[7,6],[11,4],[11,6]

Example 2:

Input: nums1 = [1,1,2], nums2 = [1,2,3], k = 2

Output: [[1,1],[1,1]]

Explanation: The first 2 pairs are returned from the sequence: [1,1],[1,1],[1,2],[2,1],[1,2],[2,2],[1,3],[1,3],[2,3]

Example 3:

Input: nums1 = [1,2], nums2 = [3], k = 3

Output: [[1,3],[2,3]]

Explanation: All possible pairs are returned from the sequence: [1,3],[2,3]

import java.util.\*;

public class KSmallestPairs {

public static List<int[]> kSmallestPairs(int[] nums1, int[] nums2, int k) {

// Result list to store the k smallest pairs

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

if (nums1 == null || nums2 == null || nums1.length == 0 || nums2.length == 0 || k <= 0) {

return result;

}

// Min-heap to store pairs based on their sum

PriorityQueue<int[]> minHeap = new PriorityQueue<>(

(a, b) -> (nums1[a[0]] + nums2[a[1]]) - (nums1[b[0]] + nums2[b[1]])

);

// Initialize the heap with pairs from nums1 with the first element of nums2

for (int i = 0; i < Math.min(nums1.length, k); i++) {

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

}

// Extract the smallest pairs and add new pairs into the heap

while (k > 0 && !minHeap.isEmpty()) {

int[] currentPair = minHeap.poll();

int i = currentPair[0];

int j = currentPair[1];

result.add(new int[] { nums1[i], nums2[j] });

if (j + 1 < nums2.length) {

minHeap.offer(new int[] { i, j + 1 });

}

k--;

}

return result;

}

public static void main(String[] args) {

int[] nums1 = {1, 7, 11};

int[] nums2 = {2, 4, 6};

int k = 3;

List<int[]> result1 = kSmallestPairs(nums1, nums2, k);

System.out.print("Output 1: ");

for (int[] pair : result1) {

System.out.print(Arrays.toString(pair) + " ");

}

System.out.println();

int[] nums1\_2 = {1, 1, 2};

int[] nums2\_2 = {1, 2, 3};

int k2 = 2;

List<int[]> result2 = kSmallestPairs(nums1\_2, nums2\_2, k2);

System.out.print("Output 2: ");

for (int[] pair : result2) {

System.out.print(Arrays.toString(pair) + " ");

}

System.out.println();

int[] nums1\_3 = {1, 2};

int[] nums2\_3 = {3};

int k3 = 3;

List<int[]> result3 = kSmallestPairs(nums1\_3, nums2\_3, k3);

System.out.print("Output 3: ");

for (int[] pair : result3) {

System.out.print(Arrays.toString(pair) + " ");

}

}

}

Q3. You are playing a solitaire game with three piles of stones of sizes a​, b,​ and c​ respectively. Each turn you

choose two different non-empty piles, take one stone from each, and add 1 point to your score. The game

stops when there are fewer than two non-empty piles (meaning there are no more available moves).

Given three integers a​, b,​ and c​, return the maximum score you can get.

Example 1:

Input: a = 2, b = 4, c = 6

Output: 6

Explanation: The starting state is (2, 4, 6). One optimal set of moves is:

- Take from 1st and 3rd piles, state is now (1, 4, 5)

- Take from 1st and 3rd piles, state is now (0, 4, 4)

- Take from 2nd and 3rd piles, state is now (0, 3, 3)

- Take from 2nd and 3rd piles, state is now (0, 2, 2)

Take from 2nd and 3rd piles, state is now (0, 1, 1)

- Take from 2nd and 3rd piles, state is now (0, 0, 0)

There are fewer than two non-empty piles, so the game ends. Total: 6 points.

Example 2:

Input: a = 4, b = 4, c = 6

Output: 7

Explanation: The starting state is (4, 4, 6). One optimal set of moves is:

- Take from 1st and 2nd piles, state is now (3, 3, 6)

- Take from 1st and 3rd piles, state is now (2, 3, 5)

- Take from 1st and 3rd piles, state is now (1, 3, 4)

- Take from 1st and 3rd piles, state is now (0, 3, 3)

- Take from 2nd and 3rd piles, state is now (0, 2, 2)

- Take from 2nd and 3rd piles, state is now (0, 1, 1)

- Take from 2nd and 3rd piles, state is now (0, 0, 0)

There are fewer than two non-empty piles, so the game ends. Total: 7 points.

import java.util.Arrays;

public class MaxScoreFromPiles {

public static int maximumScore(int a, int b, int c) {

// Sort the piles so that a <= b <= c

int[] piles = {a, b, c};

Arrays.sort(piles);

// While the two largest piles are non-empty, keep taking stones

while (piles[1] > 0 && piles[2] > 0) {

// Take one stone from the two largest piles

piles[1]--;

piles[2]--;

// Increase the score by 1 for this move

piles[0]++;

// Re-sort to maintain the order

Arrays.sort(piles);

}

return piles[0];

}

public static void main(String[] args) {

int a1 = 2, b1 = 4, c1 = 6;

System.out.println("Output 1: " + maximumScore(a1, b1, c1)); // Output: 6

int a2 = 4, b2 = 4, c2 = 6;

System.out.println("Output 2: " + maximumScore(a2, b2, c2)); // Output: 7

}

}

Q4. You are given an m x n matrix mat that has its rows sorted in non-decreasing order and an integer k.

You are allowed to choose exactly one element from each row to form an array.

Return the kth smallest array sum among all possible arrays.

Example 1:

Input: mat = [[1,3,11],[2,4,6]], k = 5

Output: 7

Explanation: Choosing one element from each row, the first k smallest sum are:

[1,2], [1,4], [3,2], [3,4], [1,6]. Where the 5th sum is 7.

Example 2:

Input: mat = [[1,3,11],[2,4,6]], k = 9

Output: 17

Example 3:

Input: mat = [[1,10,10],[1,4,5],[2,3,6]], k = 7

Output: 9

Explanation: Choosing one element from each row, the first k smallest sum are:

[1,1,2], [1,1,3], [1,4,2], [1,4,3], [1,1,6], [1,5,2], [1,5,3]. Where the 7th sum is 9.

import java.util.\*;

public class KthSmallestArraySum {

public static int kthSmallest(int[][] mat, int k) {

// Min-heap to store the sum along with the index combination

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

Set<String> visited = new HashSet<>();

int m = mat.length, n = mat[0].length;

int[] initial = new int[m];

int initialSum = 0;

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

initialSum += mat[i][0];

}

minHeap.offer(new int[] {initialSum, 0, 0, 0}); // initial sum and index combination

visited.add(Arrays.toString(new int[m])); // Mark the initial state as visited

while (--k > 0) { // Process k-1 smallest sums

int[] current = minHeap.poll();

int sum = current[0];

int[] indices = Arrays.copyOfRange(current, 1, m + 1);

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

if (indices[i] + 1 < n) {

int[] nextIndices = Arrays.copyOf(indices, m);

nextIndices[i]++;

int nextSum = sum - mat[i][indices[i]] + mat[i][nextIndices[i]];

if (visited.add(Arrays.toString(nextIndices))) {

minHeap.offer(combine(nextSum, nextIndices));

}

}

}

}

return minHeap.poll()[0];

}

private static int[] combine(int sum, int[] indices) {

int[] combined = new int[indices.length + 1];

combined[0] = sum;

System.arraycopy(indices, 0, combined, 1, indices.length);

return combined;

}

public static void main(String[] args) {

int[][] mat1 = {{1, 3, 11}, {2, 4, 6}};

int k1 = 5;

System.out.println("Output 1: " + kthSmallest(mat1, k1)); // Output: 7

int[][] mat2 = {{1, 3, 11}, {2, 4, 6}};

int k2 = 9;

System.out.println("Output 2: " + kthSmallest(mat2, k2)); // Output: 17

int[][] mat3 = {{1, 10, 10}, {1, 4, 5}, {2, 3, 6}};

int k3 = 7;

System.out.println("Output 3: " + kthSmallest(mat3, k3)); // Output: 9

}

}

Q5. Given that integers are read from a data stream. Find the median of elements read so for in an efficient

way. For simplicity assume, there are no duplicates. For example, let us consider the streams 5, 15, 1, 3 ...

After reading 1st element of stream - 5 -> median - 5

After reading 2nd element of stream - 5, 15 -> median - 10

After reading 3rd element of stream - 5, 15, 1 -> median - 5

After reading the 4th element of stream - 5, 15, 1, 3 -> median - 4, so on.

import java.util.Collections;

import java.util.PriorityQueue;

public class MedianFinder {

// Max-Heap for the left half

private PriorityQueue<Integer> maxHeap;

// Min-Heap for the right half

private PriorityQueue<Integer> minHeap;

/\*\* Initialize your data structure here. \*/

public MedianFinder() {

maxHeap = new PriorityQueue<>(Collections.reverseOrder());

minHeap = new PriorityQueue<>();

}

/\*\* Adds a number into the data structure. \*/

public void addNum(int num) {

if (maxHeap.isEmpty() || num <= maxHeap.peek()) {

maxHeap.offer(num);

} else {

minHeap.offer(num);

}

// Balance the heaps if necessary

if (maxHeap.size() > minHeap.size() + 1) {

minHeap.offer(maxHeap.poll());

} else if (minHeap.size() > maxHeap.size()) {

maxHeap.offer(minHeap.poll());

}

}

/\*\* Returns the median of current data stream \*/

public double findMedian() {

if (maxHeap.size() == minHeap.size()) {

return (maxHeap.peek() + minHeap.peek()) / 2.0;

} else {

return maxHeap.peek();

}

}

public static void main(String[] args) {

MedianFinder medianFinder = new MedianFinder();

medianFinder.addNum(5);

System.out.println("Median: " + medianFinder.findMedian()); // Output: 5.0

medianFinder.addNum(15);

System.out.println("Median: " + medianFinder.findMedian()); // Output: 10.0

medianFinder.addNum(1);

System.out.println("Median: " + medianFinder.findMedian()); // Output: 5.0

medianFinder.addNum(3);

System.out.println("Median: " + medianFinder.findMedian()); // Output: 4.0

}

}