**Assignment Heaps**

Q1. Given an integer array, find the kth largest element using priority queue.

Input 1 : arr[] = {1,2,3,5,2,6,9} k = 3

Output 1: 5

Input 2 : arr[] = {1,7,8,5,2,6,9} k = 6

Output 2: 2

import java.util.PriorityQueue;

public class KthLargestElement {

public static int findKthLargest(int[] arr, int k) {

// Min-heap with a size of k

PriorityQueue<Integer> pq = new PriorityQueue<>(k);

// Add elements to the priority queue

for (int num : arr) {

pq.offer(num);

if (pq.size() > k) {

pq.poll(); // Remove the smallest element if size exceeds k

}

}

// The root of the priority queue is the kth largest element

return pq.peek();

}

public static void main(String[] args) {

int[] arr1 = {1, 2, 3, 5, 2, 6, 9};

int k1 = 3;

System.out.println("Output 1: " + findKthLargest(arr1, k1)); // Output: 5

int[] arr2 = {1, 7, 8, 5, 2, 6, 9};

int k2 = 6;

System.out.println("Output 2: " + findKthLargest(arr2, k2)); // Output: 2

}

}

Q2. Given n ropes of different lengths, connect them into a single rope with minimum cost. Assume that the

cost to connect two ropes is the same as the sum of their lengths.

For example,

Input: [5, 4, 2, 8]

Output: The minimum cost is 36

[5, 4, 2, 8] –> First, connect ropes of lengths 4 and 2 that will cost 6.

[5, 6, 8] –> Next, connect ropes of lengths 5 and 6 that will cost 11.

[11, 8] –> Finally, connect the remaining two ropes that will cost 19.

Therefore, the total cost for connecting all ropes is 6 + 11 + 19 = 36.

import java.util.PriorityQueue;

public class ConnectRopesWithMinCost {

public static int minCost(int[] ropes) {

// Create a priority queue (min-heap) to store all rope lengths

PriorityQueue<Integer> pq = new PriorityQueue<>();

// Add all rope lengths to the priority queue

for (int rope : ropes) {

pq.offer(rope);

}

int totalCost = 0;

// While there is more than one rope in the heap

while (pq.size() > 1) {

// Extract the two smallest ropes

int first = pq.poll();

int second = pq.poll();

// Connect the ropes and calculate the cost

int cost = first + second;

totalCost += cost;

// Insert the new rope back into the heap

pq.offer(cost);

}

return totalCost;

}

public static void main(String[] args) {

int[] ropes = {5, 4, 2, 8};

int result = minCost(ropes);

System.out.println("The minimum cost is " + result); // Output: 36

}

}

Q3. Given an array of string ‘words’ and an integer k, return the k most frequent strings. Return the answer

sorted by the frequency from highest to lowest. Sort the words with the same frequency by their

lexicographical order.

Example 1:

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

Output: ["i","love"]

Explanation: "i" and "love" are the two most frequent words.

Note that "i" comes before "love" due to a lower alphabetical order.

Example 2:

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

Output: ["the","is","sunny","day"]

Explanation: "the", "is", "sunny" and "day" are the four most frequent words, with the number of occurrences

being 4, 3, 2 and 1 respectively.

import java.util.\*;

public class TopKFrequentWords {

public static List<String> topKFrequent(String[] words, int k) {

// Step 1: Count the frequency of each word

HashMap<String, Integer> frequencyMap = new HashMap<>();

for (String word : words) {

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

}

// Step 2: Use a priority queue to store the k most frequent words

PriorityQueue<String> pq = new PriorityQueue<>(

(word1, word2) -> {

int freq1 = frequencyMap.get(word1);

int freq2 = frequencyMap.get(word2);

if (freq1 == freq2) {

return word2.compareTo(word1); // Lexicographical order (reverse)

} else {

return freq1 - freq2; // Frequency order (min-heap)

}

}

);

// Step 3: Add words to the priority queue

for (String word : frequencyMap.keySet()) {

pq.offer(word);

if (pq.size() > k) {

pq.poll(); // Remove the word with the lowest frequency (or highest lexicographical order)

}

}

// Step 4: Extract the words from the priority queue and reverse the order

List<String> result = new ArrayList<>();

while (!pq.isEmpty()) {

result.add(pq.poll());

}

Collections.reverse(result); // Reverse to get correct order

return result;

}

public static void main(String[] args) {

String[] words1 = {"i", "love", "leetcode", "i", "love", "coding"};

int k1 = 2;

System.out.println("Output 1: " + topKFrequent(words1, k1)); // Output: ["i", "love"]

String[] words2 = {"the", "day", "is", "sunny", "the", "the", "the", "sunny", "is", "is"};

int k2 = 4;

System.out.println("Output 2: " + topKFrequent(words2, k2)); // Output: ["the", "is", "sunny", "day"]

}

}

Q4. You are given an array of integer stones where stones[i] is the weight of the ith stone. We are playing a game with the stones. On each turn, we choose the heaviest two stones and smash them together. Supposethe heaviest two stones have weights x and y with x <= y. The result of this smash is:

If x == y, both stones are destroyed, and

If x != y, the stone of weight x is destroyed, and the stone of weight y has new weight y - x.

At the end of the game, there is at most one stone left.

Return the weight of the last remaining stone. If there are no stones left, return 0.

Example 1:

Input: stones = [2,7,4,1,8,1]

Output: 1

Explanation:

We combine 7 and 8 to get 1 so the array converts to [2,4,1,1,1] then,

we combine 2 and 4 to get 2 so the array converts to [2,1,1,1] then,

we combine 2 and 1 to get 1 so the array converts to [1,1,1] then,

we combine 1 and 1 to get 0 so the array converts to [1] then that's the value of the last stone.

Example 2:

Input: stones = [1]

Output: 1

import java.util.PriorityQueue;

public class LastStoneWeight {

public static int lastStoneWeight(int[] stones) {

// Max-heap using PriorityQueue with reverse order

PriorityQueue<Integer> maxHeap = new PriorityQueue<>((a, b) -> b - a);

// Add all stones to the max-heap

for (int stone : stones) {

maxHeap.offer(stone);

}

// Process the stones

while (maxHeap.size() > 1) {

int stone1 = maxHeap.poll(); // Largest stone

int stone2 = maxHeap.poll(); // Second largest stone

if (stone1 != stone2) {

maxHeap.offer(stone1 - stone2); // Insert the difference back into the heap

}

}

// Return the last stone's weight if there is one, otherwise return 0

return maxHeap.isEmpty() ? 0 : maxHeap.poll();

}

public static void main(String[] args) {

int[] stones1 = {2, 7, 4, 1, 8, 1};

System.out.println("Output 1: " + lastStoneWeight(stones1)); // Output: 1

int[] stones2 = {1};

System.out.println("Output 2: " + lastStoneWeight(stones2)); // Output: 1

}

}