/\*  
https://leetcode.com/problems/minimum-cost-to-connect-sticks (premium)  
Given n ropes of different lengths, we need to connect these ropes into one rope. We can connect only 2 ropes at a time. The cost required to connect 2 ropes is equal to sum of their lengths. The length of this connected rope is also equal to the sum of their lengths. This process is repeated until n ropes are connected into a single rope. Find the min possible cost required to connect all ropes.  
  
Example 1:  
  
Input: ropes = [8, 4, 6, 12]  
Output: 58  
Explanation: The optimal way to connect ropes is as follows  
1. Connect the ropes of length 4 and 6 (cost is 10). Ropes after connecting: [8, 10, 12]  
2. Connect the ropes of length 8 and 10 (cost is 18). Ropes after connecting: [18, 12]  
3. Connect the ropes of length 18 and 12 (cost is 30).  
Total cost to connect the ropes is 10 + 18 + 30 = 58  
Example 2:  
  
Input: ropes = [20, 4, 8, 2]  
Output: 54  
Example 3:  
  
Input: ropes = [1, 2, 5, 10, 35, 89]  
Output: 224  
Example 4:  
  
Input: ropes = [2, 2, 3, 3]  
Output: 20  
Solution  
Java heap solution: https://leetcode.com/playground/unLWYVJF  
Time complexity: O(nlogn).  
Space complexity: O(n).  
\*/  
  
import java.util.PriorityQueue;  
import java.util.Queue;  
  
public class MinCosttoConnectRopes {  
  
 public int minCosttoConnectRopes(int[] ropes){  
 int result =0;  
 if(ropes == null || ropes.length == 0)  
 return result;  
  
 Queue<Integer> queue = new PriorityQueue<Integer>();  
  
 for(int i=0;i<ropes.length;i++){  
 queue.add(ropes[i]);  
 }  
  
 while(queue.size() > 1){  
 int tmp = queue.poll() + queue.poll();  
 queue.add(tmp);  
 result += tmp;  
 }  
 return result;  
 }  
}

// Java program to connect n ropes with minimum cost

// A class for Min Heap

class MinHeap {

    int[] harr; // Array of elements in heap

    int heap\_size; // Current number of elements in min heap

    int capacity; // maximum possible size of min heap

    // Constructor: Builds a heap from

    // a given array a[] of given size

    public MinHeap(int a[], int size)

    {

        heap\_size = size;

        capacity = size;

        harr = a;

        int i = (heap\_size - 1) / 2;

        while (i >= 0) {

            MinHeapify(i);

            i--;

        }

    }

    // A recursive method to heapify a subtree

    // with the root at given index

    // This method assumes that the subtrees

    // are already heapified

    void MinHeapify(int i)

    {

        int l = left(i);

        int r = right(i);

        int smallest = i;

        if (l < heap\_size && harr[l] < harr[i])

            smallest = l;

        if (r < heap\_size && harr[r] < harr[smallest])

            smallest = r;

        if (smallest != i) {

            swap(i, smallest);

            MinHeapify(smallest);

        }

    }

int parent(int i) { return (i - 1) / 2; }

    // to get index of left child of node at index i

    int left(int i) { return (2 \* i + 1); }

    // to get index of right child of node at index i

int right(int i) { return (2 \* i + 2); }

    // Method to remove minimum element (or root) from min heap

    int extractMin()

    {

        if (heap\_size <= 0)

            return Integer.MAX\_VALUE;

        if (heap\_size == 1) {

            heap\_size--;

            return harr[0];

        }

        // Store the minimum value, and remove it from heap

        int root = harr[0];

        harr[0] = harr[heap\_size - 1];

        heap\_size--;

        MinHeapify(0);

        return root;

    }

    // Inserts a new key 'k'

    void insertKey(int k)

    {

        if (heap\_size == capacity) {

            System.out.println("Overflow: Could not insertKey");

            return;

        }

        // First insert the new key at the end

        heap\_size++;

        int i = heap\_size - 1;

        harr[i] = k;

        // Fix the min heap property if it is violated

        while (i != 0 && harr[parent(i)] > harr[i]) {

            swap(i, parent(i));

            i = parent(i);

        }

    }

    // A utility function to check

    // if size of heap is 1 or not

    boolean isSizeOne()

    {

        return (heap\_size == 1);

    }

    // A utility function to swap two elements

    void swap(int x, int y)

    {

        int temp = harr[x];

        harr[x] = harr[y];

        harr[y] = temp;

    }

    // The main function that returns the minimum cost to connect n ropes of lengths stored in len[0..n-1]

    static int minCost(int len[], int n)

    {

        int cost = 0; // Initialize result

        // Create a min heap of capacity equal

        // to n and put all ropes in it

        MinHeap minHeap = new MinHeap(len, n);

        // Iterate while size of heap doesn't become 1

        while (!minHeap.isSizeOne()) {

            // Extract two minimum length ropes from min heap

            int min = minHeap.extractMin();

            int sec\_min = minHeap.extractMin();

            cost += (min + sec\_min); // Update total cost

            // Insert a new rope in min heap with length equal to sum

            // of two extracted minimum lengths

            minHeap.insertKey(min + sec\_min);

        }

        // Finally return total minimum cost for connecting all ropes

        return cost;

    }

    // Driver code

    public static void main(String args[])

    {

        int len[] = { 4, 3, 2, 6 };

        int size = len.length;

        System.out.println("Total cost for connecting ropes is " + minCost(len, size));

    }

};