Heaps

1. Connect N Ropes

```java
import java.util.PriorityQueue;

public class ConnectNropes {
    public static void main(String[] args) {
        int ropes[] = {2, 3, 3, 4, 6};
        PriorityQueue <Integer> pq = new PriorityQueue<>();
        for(int i = 0; i < ropes.length; i++){
            pq.add(ropes[i]);
        }
        int cost = 0;
        while (pq.size() > 1) {
            int min = pq.remove();
            int min2 = pq.remove();
            cost += min + min2;
            pq.add(min + min2);
        }
        System.out.println("Total Cost Connecting N Ropes: " + cost);
    }
}
```

2. Heap Sort

```java
public class HeapSort {
    public static void heapify(int arr[], int i, int size){
        int left = 2 * i + 1;
        int right = 2 * i + 2;
        int maxIdx = i;

        if (left < size && arr[left] > arr[maxIdx]) {
            maxIdx = left;
        }
        if (right < size && arr[right] > arr[maxIdx]) {
            maxIdx = right;
        }
        if (maxIdx != i) {
            // Swap
            int temp = arr[i];
            arr[i] = arr[maxIdx];
            arr[maxIdx] = temp;
            // Recursively heapify the affected subtree
            heapify(arr, maxIdx, size);
        }
    }

    public static void heapSort(int arr[]) {
        // Build max heap
        int n = arr.length;
        for (int i = n / 2 - 1; i >= 0; i--) {
            heapify(arr, i, n);
        }
        // Extract elements from the heap one by one
        for (int i = n - 1; i > 0; i--) {
            // Move current root to the end
            int temp = arr[0];
            arr[0] = arr[i];
            arr[i] = temp;
            // Heapify the reduced heap
            heapify(arr, 0, i);
        }
    }

    public static void main(String[] args) {
        int arr[] = {1, 2, 4, 5, 3};
        heapSort(arr);
        // Print sorted array
        for (int i = 0; i < arr.length; i++) {
            System.out.print(arr[i] + " ");
        }
        System.out.println();
    }
}
```

3. Kth Largest Element In A Stream

```java
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

public class KthLargestElementInAStream {
    private PriorityQueue<Integer> minHeap;
    private int k;

    public KthLargestElementInAStream(int k) {
        this.k = k;
        this.minHeap = new PriorityQueue<>();
    }

    public List<Integer> getAllKthNumbers(int[] arr) {
        List<Integer> result = new ArrayList<>();
        for (int val : arr) {
            if (minHeap.size() < k) {
                minHeap.add(val);
            } else {
                if (val > minHeap.peek()) {
                    minHeap.poll();
                    minHeap.add(val);
                }
            }
            if (minHeap.size() >= k) {
                result.add(minHeap.peek());
            } else {
                result.add(-1);
            }
        }
        return result;
    }

    public static void main(String[] args) {
        int k = 3;
        int[] arr = {1, 2, 3, 4, 5, 6};
        KthLargestElementInAStream stream = new KthLargestElementInAStream(k);
        List<Integer> res = stream.getAllKthNumbers(arr);
        for (int x : res) {
            System.out.print(x + " ");
        }
    }
}
```

4. Minimum Heap

```java
import java.util.ArrayList;

public class MinHeap {
    static class Heap{
        ArrayList <Integer> arr = new ArrayList<>();
        public void add(int data) {     // O(log n)
            // add at last index
            arr.add(data);
            int x = arr.size() - 1; // x is child index
            int par = (x-1)/2; // parent index
            while (arr.get(x) < arr.get(par)) {
                // swapping
                int temp = arr.get(x);
                arr.set(x, arr.get(par));
                arr.set(par, temp);
                x = par;
                par = (x-1)/2;
            }
        }
        public int peek(){
            return arr.get(0);  // Minimum element
        }
        private void heapify(int i){
            int left = 2*i+1;
            int right = 2*i+2;
            int minIdx = i;
            if (left < arr.size() && arr.get(minIdx) > arr.get(left)) {
                minIdx = left;
            }
            if (right < arr.size() && arr.get(minIdx) > arr.get(right)) {
                minIdx = right;
            }
            if (minIdx != i) {
                // Swapping
                int temp = arr.get(i);
                arr.set(i, arr.get(minIdx));
                arr.set(minIdx, temp);
                heapify(minIdx);
            }
        }
        public int remove(){
            int data = arr.get(0);
            // S1 - swap first & last
            int temp = arr.get(0);
            arr.set(0, arr.get(arr.size()-1));
            arr.set(arr.size()-1, temp);
            // S2 - delete last
            arr.remove(arr.size()-1);
            // S3 - Fix heap - Heapify
            heapify(0);
            return data;
        }
        public boolean isEmpty() {
            return arr.size() == 0;
        }
    }
    public static void main(String[] args) {
        Heap pq = new Heap();
        pq.add(3);
        pq.add(4);
        pq.add(1);
        pq.add(5);
        System.out.print("Priority Queue: ");
        while (!pq.isEmpty()) {
            System.out.print(pq.peek() + " ");
            pq.remove();
        }
    }
}
```

5. Minimum Operations to Halve Array Sum

```java
import java.util.ArrayList;
import java.util.List;
import java.util.PriorityQueue;

public class MinimumOperationstoHalveArraySum {
    static int minops(ArrayList<Integer> nums) {
        // Calculate the total sum of the array
        double sum = 0;
        for (int num : nums) {
            sum += num;
        }

        // Use a max-heap (simulated using negative values in a priority queue)
        PriorityQueue<Double> pq = new PriorityQueue<>((a, b) -> Double.compare(b, a));
        for (int num : nums) {
            pq.add((double) num);
        }

        double temp = sum;
        int cnt = 0;

        // Halve the largest element until the sum is reduced to half
        while (temp > sum / 2) {
            double x = pq.poll(); // Get the largest element
            temp -= x / 2;
            pq.add(x / 2); // Add half of it back to the heap
            cnt++;
        }

        return cnt;
    }

    public static void main(String[] args) {
        ArrayList<Integer> nums = new ArrayList<>(List.of(10, 20, 7));
        int count = minops(nums);
        System.out.println("Minimum operations to halve the array sum: " + count);
    }
}
```

6. Minimum Time Require to Fill N Slots

```java
import java.util.LinkedList;
import java.util.Queue;

public class MinimumTimeRequireToFillNslots {
    public static void minTime(int arr[], int N, int K) {
        Queue<Integer> q = new LinkedList<>();
        boolean[] vis = new boolean[N + 1];
        int time = 0;
        // Add initially filled slots to the queue and mark them as visited
        for (int i = 0; i < K; i++) {
            q.add(arr[i]);
            vis[arr[i]] = true;
        }
        // Perform BFS to calculate the time required to fill all slots
        while (!q.isEmpty()) {
            int size = q.size(); // Fixed size for the current level
            for (int i = 0; i < size; i++) {
                int curr = q.poll();
                if (curr - 1 >= 1 && !vis[curr - 1]) {
                    vis[curr - 1] = true;
                    q.add(curr - 1);
                }
                if (curr + 1 <= N && !vis[curr + 1]) {
                    vis[curr + 1] = true;
                    q.add(curr + 1);
                }
            }
            time++;
        }

        // Subtract 1 as the last increment occurs after all slots are filled
        System.out.println(time - 1);
    }

    public static void main(String[] args) {
        int N = 6;
        int arr[] = {2, 5};
        int K = arr.length;
        minTime(arr, N, K);
    }
}
```

7. Nearby Cars

```java
import java.util.PriorityQueue;

public class NearbyCars {
    static class Point implements Comparable <Point>{
        int x;
        int y;
        int distSq;
        int idx;
        public Point(int x, int y, int distSq, int idx){
            this.x = x;
            this.y = y;
            this.distSq = distSq;
            this.idx = idx;
        }
        @Override
        public int compareTo(Point p2){
            return this.distSq - p2.distSq;
        }
    }
    public static void main(String[] args) {
        int pts[][] = {{3, 3}, {5, -1}, {-2, 4}};
        int k = 2;
        PriorityQueue <Point> pq = new PriorityQueue<>();
        for(int i = 0; i < pts.length; i++){
            int distSq = pts[i][0] * pts[i][0] + pts[i][1] * pts[i][1];
            pq.add(new Point(pts[i][0], pts[i][1], distSq, i));
        }
        //nearest K cars
        for(int i = 0; i < k; i++){
            System.out.println("C" + pq.remove().idx);
        }
    }
}
```

8. Path with Minimum Effort

```java
import java.util.PriorityQueue;

public class PathWithMinEffort {
    // A utility class to represent the coordinate of each cell
    static class Cell implements Comparable<Cell> {
        int x, y, cost;

        Cell(int x, int y, int cost) {
            this.x = x;
            this.y = y;
            this.cost = cost;
        }

        @Override
        public int compareTo(Cell other) {
            return this.cost - other.cost;
        }
    }
    public static int minEffortPath(int grid[][]) {
        int rows = grid.length;
        int cols = grid[0].length;

        // Directions for moving up, down, left, right
        int[] rowDir = {-1, 1, 0, 0};
        int[] colDir = {0, 0, -1, 1};

        // Priority queue for Dijkstra's algorithm
        PriorityQueue<Cell> pq = new PriorityQueue<>();
        pq.offer(new Cell(0, 0, grid[0][0]));

        // Array to track minimum cost to reach each cell
        int[][] minCost = new int[rows][cols];
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                minCost[i][j] = Integer.MAX_VALUE;
            }
        }
        minCost[0][0] = grid[0][0];

        while (!pq.isEmpty()) {
            Cell current = pq.poll();

            // If we reach the bottom-right cell, return the cost
            if (current.x == rows - 1 && current.y == cols - 1) {
                return current.cost;
            }

            // Traverse all possible directions
            for (int i = 0; i < 4; i++) {
                int newX = current.x + rowDir[i];
                int newY = current.y + colDir[i];

                // Check for valid bounds
                if (newX >= 0 && newX < rows && newY >= 0 && newY < cols) {
                    int newCost = current.cost + grid[newX][newY];
                    // If a better cost is found, update minCost and add to queue
                    if (newCost < minCost[newX][newY]) {
                        minCost[newX][newY] = newCost;
                        pq.offer(new Cell(newX, newY, newCost));
                    }
                }
            }
        }

        // Return the minimum cost to reach the bottom-right cell
        return minCost[rows - 1][cols - 1];
    }

    public static void main(String[] args) {
        int[][] grid = {
            {1, 3, 1},
            {1, 5, 1},
            {4, 2, 1}
        };
        int result = minEffortPath(grid);
        System.out.println("The minimum cost to reach the bottom-right cell is: " + result);
    }
}
```

9. Priority Queue for Objects

```
import java.util.PriorityQueue;

public class PQforObjects {
    static class Student implements Comparable <Student> {
        String name;
        int rank;
        public Student(String name, int rank){
            this.name = name;
            this.rank = rank;
        }
        @Override
        public int compareTo(Student s2){
            return this.rank - s2.rank;
        }
    }
    public static void main(String[] args) {
        PriorityQueue <Student> pq = new PriorityQueue<>();
        pq.add(new Student("A", 4));
        pq.add(new Student("B", 5));
        pq.add(new Student("C", 2));
        pq.add(new Student("D", 12));
        while (!pq.isEmpty()) {
            System.out.println(pq.peek().name + "->" + pq.peek().rank);
            pq.remove();
        }
    }
}
```

10. Priority Queue in JCF

```
import java.util.PriorityQueue;

public class PriorityQueueInJCF {
    public static void main(String[] args) {
        PriorityQueue <Integer> pq = new PriorityQueue<>();
        pq.add(3);  // add operation - O(log n)
        pq.add(4);
        pq.add(1);
        pq.add(7);
        System.out.print("Priority Queue: ");
        while (!pq.isEmpty()) {
            System.out.print(pq.peek() + " ");  // peek operation - O(1)
            pq.remove(); // remove or pop operation - O(log n)
        }
    }
}
```

11. Sliding Window Maximum

```java
import java.util.PriorityQueue;

public class SlidingWindowMaximum {
    static class Pair implements Comparable <Pair>{
        int val;
        int idx;
        public Pair(int val, int idx){
            this.val = val;
            this.idx = idx;
        }
        @Override
        public int compareTo(Pair p2){
            //for ascending
            /* return this.val - p2.val; */
            //for descending
            return p2.val - this.val;
        }
    }
    public static void main(String[] args) {
        int arr[] = {1, 3, -1, -3, 5, 3, 6, 7};
        int k = 3;  //Window size
        int res[] = new int[arr.length - k + 1];    //n-k+1
        PriorityQueue <Pair> pq = new PriorityQueue<>();
        // 1st Window
        for(int i=0; i<k; i++){
            pq.add(new Pair(arr[i], i));
        }
        res[0] = pq.peek().val;
        for(int i=k; i<arr.length; i++){
            while (pq.size() > 0 && pq.peek().idx <= (i-k)) {
                pq.remove();
            }
            pq.add(new Pair(arr[i], i));
            res[i-k+1] = pq.peek().val;
        }
        // print result
        for(int i=0; i<res.length; i++){
            System.out.print(res[i] + " ");
        }
        System.out.println();
    }
}
```

12. Weakest Soldier

```java
import java.util.PriorityQueue;

public class WeakestSoldier {
    static class Row implements Comparable <Row>{
        int soldiers;
        int idx;
        public Row(int soldiers, int idx){
            this.soldiers = soldiers;
            this.idx = idx;
        }
        @Override
        public int compareTo(Row r2){
            if (this.soldiers == r2.soldiers) {
                return this.idx - r2.idx;
            } else {
                return this.soldiers - r2.soldiers;
            }
        }
    }
    public static void main(String[] args) {
        int arr[][] = {{1, 0, 0, 0},
                       {1, 1, 1, 1},
                       {1, 0, 0, 0},
                       {1, 0, 0, 0}};
        int k=2;
        PriorityQueue <Row> pq = new PriorityQueue<>();
        for(int i=0; i < arr.length; i++){
            int count = 0;
            for(int j = 0; j < arr[0].length; j++){
                count += arr[i][j] == 1 ? 1 : 0;
            }
            pq.add(new Row(count, i));
        }
        for (int i = 0; i < k; i++) {
            System.out.println("R" + pq.remove().idx);
        }
    }
}
```