```
public class 9_ Heaps{
// Kth largest element in an array
class Solution {
    // Function to return k largest elements from an array.
    public static ArrayList<Integer> kLargest(int arr[], int n, int k) {
        // code here
        PriorityQueue<Integer> pq = new PriorityQueue<>();
        for (int i = 0; i < n; i++) {
            pq.add(arr[i]);
            if (pq.size() > k) {
                pq.remove();
            }
        }
        ArrayList<Integer> ans = new ArrayList<>();
        while (pq.size() > 0) {
            ans.add(0, pq.remove());
        }
        return ans;
    }
}

// Kth smallest element in an array
class Solution {
    public static int kthSmallest(int[] arr, int l, int r, int k) {
        // Your code here
        PriorityQueue<Integer> pqmax = new PriorityQueue<>(Collections.reverseOrder())
;
        for (int i = l; i <= r; i++) {
            pqmax.add(arr[i]);
            if (pqmax.size() > k) {
                pqmax.remove();
            }
        }
        return pqmax.peek();
    }
}

// Sort K sorted array
// Merge M sorted Lists
class Solution {
    class NodeComparator implements Comparator<ListNode> {

        public int compare(ListNode k1, ListNode k2) {
            if (k1.val > k2.val)
                return 1;
            else if (k1.val < k2.val)
                return -1;
            return 0;
        }
    }

    class Solution {
        public ListNode mergeKLists(ListNode[] lists) {
            PriorityQueue<ListNode> pq = new PriorityQueue<>(new NodeComparator());
            int n = lists.length;
            ListNode ans_head = new ListNode(0);
            ListNode ans_curr = ans_head;
            for (int i = 0; i < n; i++) {
                if (lists[i] != null)
                    pq.add(lists[i]);
            }
            while (!pq.isEmpty()) {
                ListNode curr = pq.poll();
                ans_curr.next = curr;
                ans_curr = ans_curr.next;
                if (curr.next != null) {
                    pq.add(curr.next);
                }

            }
            return ans_head.next;
        }
    }
}

// Replace each array element by its c…
// Task Scheduler
// Hands of Straights
class Solution {
    public boolean isNStraightHand(int[] hand, int W) {
        if (hand.length % W != 0) {
            return false;
        }
        PriorityQueue<Integer> heap = new PriorityQueue<>();
        for (int h : hand) {
            heap.offer(h);
        }
        while (!heap.isEmpty()) {
            int val = heap.peek();
            for (int i = val; i < val + W; i++) {
                if (!heap.remove(i)) {
                    return false;
                }
            }
        }
        return true;
    }
}
```

```
// Design twitter
// Connect `n` ropes with minimal cost
class Solution {
    // Function to return the minimum cost of connecting the ropes.
    long minCost(long arr[], int n) {
        // your code here
        PriorityQueue<Long> pq = new PriorityQueue<>();
        for (int i = 0; i < n; i++) {
            pq.add(arr[i]);
        }
        long sum = 0;
        while (pq.size() >= 2) {
            long first = pq.remove();
            long second = pq.remove();
            long newrope = first + second;
            sum += newrope;
            pq.add(newrope);
        }
        return sum;
    }
}

// Kth largest element in a stream of numbers
class KthLargest {
    PriorityQueue<Integer> p = new PriorityQueue<>();
    int ks;

    public KthLargest(int k, int[] nums) {
        this.ks = k;
        for (int i = 0; i < nums.length; i++) {
            p.add(nums[i]);
            if (p.size() > k) {
                p.remove();
            }
        }
    }

    public int add(int val) {
        p.offer(val);
        if (p.size() > ks) {
            p.remove();
        }
        return p.peek();
    }
}

// Maximum Sum Combination
// Find Median from Data Stream
// K most frequent elements
```

```
class Solution {
    class Pair {
        int freq;
        int value;

        Pair(int X, int Y) {
            this.freq = X;
            this.value = Y;
        }
    }

    public int[] topKFrequent(int[] nums, int k) {
        PriorityQueue<Pair> pq = new PriorityQueue<>((x, y) -> x.freq - y.freq);
        HashMap<Integer, Integer> mp = new HashMap<>();
        int n = nums.length;
        for (int i = 0; i < n; i++) {
            mp.put(nums[i], mp.getOrDefault(nums[i], 0) + 1);
        }
        for (Integer it : mp.keySet()) {
            int F = mp.get(it);
            int V = it;
            pq.add(new Pair(F, V));
            if (pq.size() > k) {
                pq.remove();
            }
        }
        int ans[] = new int[k];
        int i = 0;
        while (pq.size() > 0) {
            Pair p = pq.remove();
            ans[i++] = p.value;
        }
        return ans;

    }
}
}
```