1. arr = [1, 3, 7, 9, 12, 10, 8, 16, 18, 22, 27]
   Create a buildHeap method that returns a minheap.
   heapify(arr, n, i):
   // Write your own code
   buildHeap(arr, n):
   //Write your own code
   heapify(arr, n, i)

```
// const arr = [1, 3, 7, 9, 12, 10, 8, 16, 18, 22, 27];
const arr = [75, 25, 35, 45, 90, 80, 60, 20, 30]


class MinHeap {
    constructor() {
        this.heap = [];
    }

    heapify(i) {
        const l = 2 * i + 1;
        const r = 2 * i + 2;
        const n = this.heap.length;
        // if no child then node
        if (l >= n || r >= n)
            return;

        // if node is smaller than both child nodes then skip
        if (this.heap[l] > this.heap[i] && this.heap[r] > this.heap[i])
return;

        // if node is greater than child node, then swap with it
        if (this.heap[l] < this.heap[i] && this.heap[l] < this.heap[r]) {
            this.swap(l, i);
            this.heapify(l)
        } else {
            this.swap(r, i);
            this.heapify(r)
        }
    }

    swap(j, i) {
        let temp = this.heap[j];
        this.heap[j] = this.heap[i];
        this.heap[i] = temp;
    }

    buildHeap(arr) {
        this.heap = arr;
        const startIndex = Math.floor(this.heap.length / 2) - 1;
        for (let i = startIndex; i > -1; i--) {
            this.heapify(i);
        }
    }
}

console.log(arr);
const obj = new MinHeap();
obj.buildHeap(arr)
console.log(obj.heap)

// Time Complexity : O(n) Building heap is highest time cosuming in the code
```

2. Given an array of strings words and an integer k, return the k most frequent words.
   Your output should be in lexicographical order.
   Words = [‘priya’, ‘bhatia’, ‘akshay’, ‘arpit’, ‘priya’, ‘arpit’]
   K = 3
   Output = [‘arpit’, ‘akshay’, ‘priya’]

```
import { PriorityQueue } from "datastructures-js";

function kFreq(words, k) {
    const wordsMap = new Map();

    // counting freq of each word and storing it in map
    words.forEach((eachword) => {
        if (wordsMap.has(eachword)) {
            wordsMap.set(eachword, wordsMap.get(eachword) + 1);
        } else {
            wordsMap.set(eachword, 1);
        }
    });

    // converting map to arrays eg - { priya : 2, arpit : 1}  will be covert
to - [[priya, 2], [arpit, 1]]
    // this converion has been done for the ease of comparision
    const mapEntries = [...wordsMap.entries()];

    // storing mapEntries array in minHeap : word with lowest freq will
appeared at the top
    const minHeap = PriorityQueue.fromArray(mapEntries, (a, b) => a[1] -
b[1]);

    // pop all words from the heap until the size of minheap is greater than k
    while (minHeap.size() > k) {
        minHeap.pop();
    }

    const res = [];
    // clear the last k elements in minHeap which will have the max frequency and store it ini res arr
    while (!minHeap.isEmpty()) {
        res.push(minHeap.pop()[0]);
    }
    console.log(res)
}

const words = ["priya", "bhatia", "akshay", "arpit", "priya", "arpit"];
const k = 3;
kFreq(words, k);

// Time Complexity : O(nLogn)
// heap sort is most time consuming in the code,
// heap contains k elements so it will be kLogn but for worst case consider O(nlogn)
```

3. Find the k closest points to the origin.
   Points = [[1, 3], [-2, 2]]
   K = 1
   Output = [-2,2]

```
import { PriorityQueue } from "datastructures-js";

function kClosestPoints(points, k) {
    // adding dist value as third elem of each point eg - [[-2, -2]] converted to [[-2, -2, 8]]
    const pointsDist = points.map((point) => {
        const dist = Math.pow(point[0], 2) + Math.pow(point[1], 2);
        point.push(dist);
        return point;
    })

    // maxHeap sorted on the basis of the distance value
    const maxHeap = PriorityQueue.fromArray(pointsDist, (a, b) => b[2] - a[2]);

    // keep the bottom k values in maxHeap and removing all top elements
    while (maxHeap.size() > k) {
        maxHeap.pop();
    }

    // storing the remaining points in res arr
    const res = [];
    while (!maxHeap.isEmpty()) {
        const point = maxHeap.pop().splice(0, 2);
        res.push(point);
    }

    console.log(res)
}

const points = [[1, 3], [-2, 2]];
const k = 1;
kClosestPoints(points, k)

// Time Complexity : O(nLogn)
// heap sort is most time consuming in the code,
// heap contains k elements so it will be kLogn but for worst case consider O(nlogn)
```