1. Given an integer array nums of length n and an integer target, find three integers in nums
   such that the sum is closest to the target.[Amazon]
   You need to return the sum of three integers.
   For example:arr = [-1, 2, 1, -4], target = 1
   Output: 2
   Explanation: [-1+2+1] = 2 (The sum that is closest to the target is 2)

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

function targetSum(arr, i, sum, target, minHeap, k) {

    // if k = 0 (i.e. sum = addition of all 3 elements) push the diff in
minHeap, this heap will have lowest diff on the top
    if (k === 0 && i < arr.length + 1) {
        const diff = Math.abs(target - sum);

        const diffs = [];   // diffs[] contains [sum, diff]
        diffs.push(sum);
        diffs.push(diff)

        minHeap.push(diffs);   // pushing diffs into minHeap, heap will sort
elements with minimum diff on the top

        return;
    }

    // if i exceeds the lenght of arr then return
    if (i > arr.length){
        return;
    }

    // including the element in sum
    targetSum(arr, i + 1, sum + arr[i], target, minHeap, k - 1);
    // excluding the element in sum
    targetSum(arr, i + 1, sum, target, minHeap, k);
    return;
}

const minHeap = new PriorityQueue((a, b) => a[1] - b[1]);
const arr = [-1, 2, 1, -4];
const target = 1;
const k = 3; // no. of elements to be considered for sum

targetSum(arr, 0, 0, target, minHeap, k);
const sum = minHeap.pop()[0];  // sum which is closest to target
console.log(sum)
```

2. Given three points, check whether they lie on a straight (collinear) or not. [Google]
   For example:
   Input- [(1,1), (1,4), (1,5)]
   Output- Yes

```
function collinear(points) {
    // slope of first and second points
    const slope1 = points[1][1] - points[0][1] / points[1][0] - points[0][0];
    // slope of second and third points
    const slope2 = points[2][1] - points[1][1] / points[2][0] - points[1][0];

    let isSlopeEqual = slope1 === slope2;

    let isXEqual = false;
    let isYEqual = false;

    // also check if each of coordinate (X or Y) is equal for all points
    for (let i = 0; i < 2; i++) {
        if (points[i][0] === points[i + 1][0]) {
            isXEqual = true;
        } else {
            isXEqual = false;
        }
        if (points[i][1] === points[i + 1][1]) {
            isYEqual = true;
        } else {
            isYEqual = false;
        }
    }

    // if any of this three condition is true it is a COLLINEAR
    if (isSlopeEqual || isXEqual || isYEqual) {
        console.log("Yes")
    } else {
        console.log("No")
    }
}

// const points = [[1, 1], [1, 6], [0, 9]];
const points = [[1, 1], [1, 4], [1, 5]];
collinear(points)

// Time Complexity : O(n)
// iterating through each element of array for calculating slope between 2
points

// Space Complexity : O(1)
```

3. An e-commerce site tracks the purchases made each day. The product that is purchased
   the most one day is the featured product for the following day. If there is a tie for the product
   purchased most frequently, those product names are ordered alphabetically ascending and
   the last name in the list is chosen.[Amazon]
   ['yellowShirt', 'redHat', 'blackShirt', 'bluePants', 'redHat','pinkHat', 'blackShirt', 'yellowShirt',
   'greenPants', 'greenPants', 'greenPants']
   'yellowShirt' - 2
   'redHat' - 2
   'blackShirt' - 2
   'bluePants' - 1
   'greenPants' - 3
   'pinkHat' - 1
   Output – greenPants

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

function mostFreqProducts(products) {
    // creating map for mainting frequency of each products
    const productsMap = new Map();
    products.forEach((product) => {
        if (productsMap.has(product)) {
            productsMap.set(product, productsMap.get(product) + 1);
        } else {
            productsMap.set(product, 1);
        }
    });

    // converting map to arrays eg - { redHat : 2, greenPants : 1}  will be
covert to - [[redHat, 2], [greenPants, 1]]
    // this converion has been done for the ease of comparision
    const productsEntries = [...productsMap.entries()];
    // console.log(productsEntries)

    // storing productsEntries array in MaxHeap : product with highest freq
will appeared at the top
    const maxHeap = PriorityQueue.fromArray(productsEntries, (a, b) => b[1] -
a[1])

    // get the top 2 products of maxHeap using pop
    const firstProduct = maxHeap.pop();
```

```
    const secondProduct = maxHeap.pop();

    let res = [];
    // compare if the top 2 products has the equal freq then store them in
res[] arr and sort it
    if (firstProduct[1] === secondProduct[1]) {
        // push first and second product (which we pop earlier) into the res[]
arr
        res.push(firstProduct[0])
        res.push(secondProduct[0])

        // push the remaing products into the res[] arr, if there is any
product with same freq as firstProduct
        while (maxHeap.front()[1] === firstProduct[1]) {
            res.push(maxHeap.pop()[0])
        }
    } else {
        // if top 2 products are not equal then return the firstProduct
        console.log(firstProduct)
    }
    // sort for lexicography
    console.log(res.sort())
}

const products = ['yellowShirt', 'redHat', 'cap', 'cap', 'blackShirt',
'bluePants', 'redHat', 'pinkHat', 'pinkHat', 'blackShirt', 'yellowShirt',
    'greenPants', 'greenPants', 'greenPants'];

mostFreqProducts(products)


// Time Complexity : O(nLogn)
// array sort is the highest time consuming in the code
```

4. An almost sorted array is given to us and the task is to sort that array completely. Then,
   which sorting algorithm would you prefer and why?[Salesforce]

4. An almost sorted array is given to us and the task is to sort that array completely. Then, which sorting algorithm would you prefer and why? [Salesforce]

ANSWER

First of all, `What is nearly sorted array or almost sorted array ?`

> Given an array of n elements, where each element is at most k away from its target position is called nearly sorted.

> *For example* :- let us consider k is 2, an element at index 7 in the sorted array, can be at indexes 5, 6, 7, 8, 9 in the given array.

Eg:

1. {6, 5, 3, 2, 8, 10, 9}, k = 3
2. {10, 9, 8, 7, 4, 70, 60, 50}, k = 4

Efficient Sorting algorithm : *INSERTION SORT*

Reason :

> Because what Insertion Sort does is it compares the key element with all left elements and placed it into its correct position.

> So in nearly sorted array every elements correct position will be at k distance far. So, for each element there will be only K comparisions not N comparisions.