Microservices: it is a controller for distributed systems, where a architect develops a application in order handle the different servers.

***a service-oriented application component that is tightly scoped, strongly encapsulated, loosely coupled, independently deployable and independently scalable***

Microservices architectures, or simply Microservices is an SDLC approach based on which larger applications are built as a collection of small functional modules. These functional modules are independently deployable, scalable, target specific business goals, and communicate with each other over standard protocols like HTTP request/response with resource APIs and lightweight asynchronous messaging.

***The project I have worked is great example for microservice, it is large scale data storage application built on small functional modules (like creating pools, volumes , hosts, and volume groups ) , each module is scalable , independently deployable, functional, target specific goals and communicate with each other with HTTP.***

## ***Time complexity:***

Constant Time Complexity: O(1) – when only one statement executes in program

**Linear Time Complexity: O(n) – when programs iterates through loop n times -** more efficient and faster

**Quadratic Time Complexity: O(n²) -** Nested **For Loops** run on quadratic time – less efficient, and their performance can degrade quickly as the input size grows

**Exponential Time Complexity: O(2^n) - Brute-Force algorithms – loop iterate growth rate increases.**

**Logarithmic Time Complexity: O(log n) -** inversely proportional to the input “n” or “Divide and Conquer” – **Two pointer algorithms**

The optimal time complexity of Two pointer approach typically lies in O(n) as linear search, In most of the cases like searching and sorting (merge sort or binary search) the time complexity falls in between O(log n) or O(n log n).

For two pointer approach the time complexity depends on underlying developed logic too.

## Data Structure – Java Script

Array, Objects, Linked Lists, Queues, Hash Table, Trees, Graphs

typeOf(variable\_ref); > returns the data type of variable

parseInt(variable\_ref); > returns the NUMBER data type ;

Floor>Math.floor(1.2) > 1

Floor means closet integer lesser than the given number.

Ceil> Math.ceil(3.5) > 4

Ceil means closet integer greater than the given number.

### Maps:

Based on the concept of key and value pair.

Properties:

* Keyes can’t be duplicated, has to be unique.
* INSERT is not possible if the key already presents.
* UPDATE value for a key
* DELETE based on key
* Position doesn’t matter

Obj[“key”] =17;

Object.keys(obj);

Let map = new Map();

map.set(‘a’,1);

### Objects:

class Car {

constructor(name, year) {

this.name = name;

this.year = year;

}

}

const myCar1 = new Car("Ford", 2014);

const myCar2 = new Car("Audi", 2019);

<script>

**const person = {**

**firstName : "John",**

**lastName : "Doe",**

**age : 50,**

**eyeColor : "blue"**

**};**

document.getElementById("demo").innerHTML = person.firstName + " " + person.lastName;

</script>

### Arrays:

Homogeneous Data

Constant size, go for it when the known quantity of items.

Accessed and stored data using index.

Int array[] = new int[]; || var array[] = new Array();

Length is property of Array class.

Push – unshift -adding an element pop-shift -to delete an element.

### Comparator interface

Multiple sorting based on object properties

Comparator provides **compare() method** to sort elements.

**Collections.sort(List, Comparator) or Arrys.sort(array,comparator)**

**More like object comparision**

new Comparator<Integer> (){

                public int compare(Integer a, Integer b){

                    return a-b;

                    }}

### Comparable Interface

Comparable provides **compareTo() method** to sort elements.

We can sort the list elements of Comparable type by **Collections.sort(List)** method.

More like object specific comparesion

## **Data Structure**

**STRINGS, ARRAYS ,LIST, MAP, SET**

Maps – doesn’t allows the duplicate keys, to get sorted map use Treemap – no null treemap

Set - doesn’t allows the duplicate values, , to get sorted set use Treeset- no null in treemap

### **Set** -JAVA

Interface Set interface extends Collection interface. In a set, no duplicates are allowed. Every element in a set must be unique.

* HashSet does not guarantee any order
* LinkedHashSet maintain insertion order
* TreeSet maintain sorting order
* HashSet and LinkedHashSet allows only one null
* TreeSet does not allow null
* All three are not thread-safe

**JAVA- put(), remove,clear(),containsKey(),containsValue(), get()**

**for(int i: arra){}**

## **Brute Force:**

Two loops, quadratic TC > O(n\*2)

 int result=1;

        int count=0;

        for(int i=1;i<n;i++){*//7 4 10 9 6 1 8 2 5 3*

            for(int j=i-1;j>=0;j--){

                if(nums[i]!=nums[j]){

                    count++;

                }

            }

            if(count == i){

                result ++;

            }

            count =0;

        }

        return result;

## Two pointer approach

Two pointers, one loop , logarithmic TC > O(log N)

## Optimized approach

Sort the data structure + loop ,O(nlogn)

# **Binary Search:**

In binary search:

* There is one primary pointer (usually called "mid" or "pivot") that divides the search space into two halves.
* It repeatedly adjusts this primary pointer based on a comparison between the target value and the value at the midpoint.
* The search space is divided into two halves, and one of them is eliminated at each iteration.

You can efficiently find the target value in a sorted array of distinct integers with a binary search algorithm, which has a time complexity of O(log N) logarithmic time complexity for sorted arrays. Here's a Java example of how to do this:

int left = 0;

        int right = nums.length - 1;

        while (left <= right) {

            int mid = left + (right - left) / 2;

            if (nums[mid] == target) {

                return mid; *// Target found, return its index.*

            }

            if (nums[mid] < target) {

                left = mid + 1; *// Target is on the right side.*

            } else {

                right = mid - 1; *// Target is on the left side.*

            }

        }

        return -1; *// Target not found in the array.*

    }

### Example: HashMap and ArrayList program

TO find intersection elements of two arrays

 static List<Integer> intersectionOfTwoArraysBasic(int n, List<Integer>a, int m, List<Integer>b){

    List<Integer> result = new ArrayList<>();

*// Time complexity - O(n+m)*

*//use a for each loop and Hashmap(duplicate keys not allowed)*

*//check the value of a[i] not a key in map*

*//if yes put new key in map, else update the exiting key with old+1;*

    Map<Integer,Integer> freq = new HashMap<Integer,Integer>();

    int count =0;

    for(int i: a){

        if(!freq.containsKey(i)){

            freq.put(i,1);

        }

        else{

        freq.put(i,freq.get(i)+1);

        }

    }

*//iterating through the b[] using for each loop*

*// check the element of b being as key in map or not*

*// if yes check the accurance count >0*

*//if yes decrement the its value and add to the result list*

    for(int i: b){

        if(freq.containsKey(i)){

            if(freq.get(i)>0){

            result.add(i);

            freq.put(i,freq.get(i)-1);

            }

        }

    }

*//sort the sort list and return*

        Collections.sort(result);

        return result;

    }

### Priority Queue:

*PriorityQueue<E> pq = new PriorityQueue<E>();*

*Add()- to add, peek() – to retrive, poll() – to remove and retrieve , contains() - tocchek*

### Recursion

Implementing the function call inside same function.

**In recursion the result depends on base return. And initialized value**

function factorial(n) {

    var result = n;

   if(n==0){

       return 1;

   }

   return result = factorial(n-1)\*result;

//two find nth Fibonacci number

function fib(n){ //O(n) – if stack space consider or O(1) - if stack space not considered   
if(n==0||n==1)

return n;

return fib(n-1)+fib(n-2);

}

//Tribonacci series

function fib(n){ //O(n) – if stack space consider or O(1) - if stack space not considered   
if(n==0)

return n;

else if(n== 1|| n==2){  
return 1;

return fib(n-1)+fib(n-2)+fib(n-3);

}

With out recursive function fib series: O(n)  
var tmp=0

var i=1;

while(i<5){

console.log(tmp);

tmp = i+tmp;

i= tmp-i;

}

Prime which having only two perfect divisible 1 and it self.

HCF & GCD >

Euclid rule > the GCD or HCF of a two number is value of smallest number subtracted from greatest number.

GCD(34, 8) > (26, 8) > (18, 8) > (10, 8)> (2, 8)>(2, 6)>(2, 4)>(2,2) > 2 : TC=O(min(a,b))

Using recursion function

function gcd(a,b){

if(a==0)

return b;

if(b==0)

return a;

return gcd(b, a%b);  
}//TC = O(logmin(a,b)))

CoPrime > the sum two numbers which divisor is only 1.

### Prime

Math.sqrt(n) is best way to loop upto this point. To reduce the number repeating iterations. 1 is not prime so can exempt this. So which having exactly one divisor from 2 to Math.sqrt(n).

 public static boolean isPrime(int n){

       int count = 1;

         for(int i=2;i<=(int)Math.sqrt(n);i++){

            if(n%i ==0){

              count++;

            }

         }

         if(count == 1){

             return true;

         }

         return false;

   }

### Modules pros

Modules helps code lie between the given constraints or between size of an array.

static int[] leftRotation(int[] a, int d) {

        int n = a.length;

        int[] ans = new int[n];

        int j=0;

        for (int i = d; i <n; i++) {

*//int new\_pos = i - d; (i - d + N) % N*

            ans[j] = a[i];

            j++;

        }

        for(int i=0; i<d;i++){

            ans[j] = a[i];

            j++;

        }

        return ans;

    }

Rather than using two for loops using mod can easily get index

 static int[] leftRotation(int[] a, int d) {

        int n = a.length;

        int[] ans = new int[n];

        for (int i = 0; i < n; i++) {

            int new\_pos = (i - d + n) % n;

            ans[new\_pos] = a[i];

        }

        return ans;

    }

### Incrementing array element by 1:

static int[] incrementNumber(int n, int digits[]){

*//int n = digits.length;*

    int carry = 1;

    for (int i = n - 1; i >= 0; i--) {*//9 1 1*

        int sum = digits[i] + carry;*//9  1 2*

        digits[i] = sum % 10;*//9 1 2*

        carry = sum / 10;*//0  0 0*

    }

    if (carry == 1) {

        int[] result = new int[n + 1];

        result[0] = 1;

        return result;

    } else {

        return digits;

    }

    }

Here the approach is space and time optimized because rather than doing 10th position

multiplication and adding, we just using carry to forward to all elements if any.

### Palindrome - String

*to check a string is permutation of palindrome , the string must contains even number of times charcters or only one time, if any charcters is odd number(>1) times then the string is not permutaion of palindrome*

Anagrams – the string which is having same chars but order can be different.

If an array sorted go for binary search which uses two pointer approach.

If an array is not sorted go for linear search;