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`x

**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.

## **Machine Coding Questions**

Machine coding questions are a type of technical assessment commonly used during software engineering interviews. They evaluate a candidate's ability to:

**1. Translate Requirements into Code:**

* The interviewer provides a problem statement or set of requirements for a program or functionality.
* You need to write functional code, often without the aid of high-level frameworks or libraries, to demonstrate your understanding and coding skills.

**2. Design and Develop Low-Level Logic:**

* The focus is on core programming concepts like data structures, algorithms, memory management, and control flow.
* You might need to implement data structures like linked lists or arrays, create functions to perform specific tasks, and handle edge cases efficiently.

**3. Understand System Behavior:**

* In some cases, the questions might involve working with simulated memory or manipulating bits to understand how code interacts with the underlying system.

**Here's a breakdown of the key aspects of machine coding questions:**

* **Emphasis on Fundamentals:** The goal is to assess your grasp of core programming principles, not your proficiency in a specific framework or language.
* **Conciseness and Efficiency:** While the code shouldn't be overly complex, it should be well-thought-out, efficient, and address the problem effectively.
* **Problem-Solving Skills:** You need to analyze the problem, break it down into smaller steps, and translate those steps into code.
* **Algorithmic Thinking:** Choosing the appropriate data structures and algorithms can significantly impact the solution's efficiency.

**Common Topics Covered in Machine Coding Questions:**

* String manipulation (e.g., reversing a string, finding substrings)
* Linked lists (e.g., implementing basic operations like insertion, deletion, and traversal)
* Arrays (e.g., searching, sorting, dynamic resizing)
* Tree data structures (e.g., binary search trees)
* Bit manipulation (e.g., setting or clearing specific bits)
* Sorting algorithms (e.g., bubble sort, merge sort, quick sort)
* Memory allocation and management (e.g., manual memory management in some languages)

**Tips for Preparing for Machine Coding Rounds:**

* **Practice with basic data structures and algorithms:** Brush up on your foundation and how to implement them efficiently.
* **Practice writing low-level code:** Solve problems without using high-level libraries or frameworks.
* **Improve your problem-solving skills:** Be able to analyze problems, break them down, and come up with solutions.
* **Learn about common coding interview patterns:** Familiarize yourself with frequently asked questions and their approaches.
* **Practice on online platforms:** There are websites offering practice problems for technical interviews.

## 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);

**map and sets takes O(1) for Crud operations**

**map store only key from 256 chars – when keys are chars the space complexity is O(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){}**

### Spiral matric implementation

ArrayList<Integer> sum = new ArrayList<Integer>();

         int index =c+r-1;

         int cEnd= 0;

         int rEnd = 0;

         int add =0;

*// diagonal sum -column*

            while (cEnd < c){

              int row=0;

              int col=cEnd;

              int value =c;

              while(iter < value){ *//witin matrix diagonal*

                add + = M.get(row).get(col);

                row++;

                col++;

                iter++;

            }

            cEnd++; *//increasing column*

             }

         while (j < rEnd){

         }

         while(i<index){

            for (int i=0; i<)

              add +=M.get(1).get(1);

             add+=  M.get(2)+M.get(2));

             }

         }

# knowledge of data structures

<https://www.hackerrank.com/domains/data-structures>

### Heap Data Structure

### *A Heap is a special****Tree-based Data Structure****in which the tree is a*[*complete binary tree*](https://www.geeksforgeeks.org/complete-binary-tree/)*.*

* **Heapify:** a process of creating a heap from an array. to balance the tree.
* **Insertion:** process to insert an element in existing heap time complexity O(log N).
* **Deletion:** deleting the top element of the heap or the highest priority element, and then organizing the heap and returning the element with time complexity O(log N).
* **Peek:** to check or find the first (or can say the top) element of the heap.

**Types of Data structure:**

1. **Max-Heap :** In max heap the key present at top node must be maximum/greatest among the keys. The same property must be recursively true for all sub-trees in that Binary Tree.
2. **Min-Heap :** in min heap the key present at top node must be minimum of all keys present in heap. The same property must be recursively true for all sub-trees in that Binary Tree.

### LinkedList

## **Problem solving approach**

1. Define the problem

* Clearly understand the problem and the nature of the data to be sorted. Consider the size of the dataset, the range of values, and whether the data is partially sorted or completely unsorted.

1. Understand Algorithm characteristics

* Different sorting algorithms have different time and space complexities. Understand the characteristics of common sorting algorithms such as Bubble Sort, Insertion Sort, Selection Sort, Merge Sort, Quick Sort, and Heap Sort.

1. Choose the right algorithm

* Select the sorting algorithm that best suits the specific requirements of the problem. Consider factors such as time complexity, space complexity, stability, and adaptability.

1. Implement the chosen algorithm

* Translate the chosen algorithm into code. Pay attention to details like array indexing, loop structures, and termination conditions. Ensure that the implementation handles edge cases and performs correctly on various input scenarios.

1. Optimize if Necessary:

* Analyze the performance of the implemented algorithm. If needed, consider optimization techniques such as tail recursion elimination, loop unrolling, or parallelization to improve efficiency. However, optimization should be based on actual performance measurements.

1. Test Extensively:

* Develop a comprehensive set of test cases to verify the correctness and efficiency of the sorting algorithm. Include cases with varying data sizes, sorted and reverse-sorted data, and data with duplicates.

1. Handle Edge Cases:

* Ensure that the sorting algorithm handles edge cases gracefully. For example, empty arrays, arrays with a single element, or datasets with identical elements.

1. Document and Analyze:

* Document the implemented algorithm, including its time and space complexity. Perform a thorough analysis of the algorithm's behavior under different scenarios. This documentation aids in understanding, maintaining, and sharing the sorting solution.

1. Consider Special Requirements:

* If the problem has specific constraints, such as limited memory or the need for stable sorting, consider these requirements during the selection and implementation of the sorting algorithm.

1. Iterate and Refine:

* Based on the testing and analysis, iterate on the implementation if needed. Refine the algorithm or choose a different one if the initial selection proves suboptimal for the problem at hand.

Problem Solving:  
Milestone 1: Understand the problem statement, with constraints and examples

Milestone 2: Walk through the different approaches to efficiently solving the problem

Milestone 3: write the instructions

Milestone 4 : code as per instructions

Milestone : TC and SC.

# knowledge of algorithms

## 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)

Here are two variations in two pointer approach, as a programmer must know the brute force approach before defining optimized approach.

The three variations are:

1. two pointers starts at both ends of the matrix, best of sum target2. starting – both pointers start at left side of array, best for merging of an array3. centre opposite – two pointers starts at ends of the arrays and the third pointer starts in middle of elements – best of triple targets.

## Optimized approach

Sort the data structure + loop ,O(nlogn)

## Newton Raphson Method  - to find sqrt(n)

public class SqrtCalculator {

public static double sqrt(int x) {

if (x == 0) {

return 0;

}

double x0 = x; // Initial estimate

while (true) {

double x1 = 0.5 \* (x0 + (x / x0)); // Calculate the next estimate

if (Math.abs(x1 - x0) < 1e-6) {

return x1; // Converged to a sufficiently accurate result

}

x0 = x1; // Update the current estimate

}

}

public class SqrtCalculator {

public static double sqrt(int x) {

if (x == 0) {

return 0;

}

double prev, curr = 1;

do {

prev = curr;

curr = (prev + x / prev) / 2;

} while (Math.abs(curr - prev) >= 1e-6); // Adjust the tolerance as needed

return curr;

}

}

## **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.*

    }

## **Bubble sort**

Bubble sort is used to sort the array of elements. The time complexity is O(n2) require one nested for loop, where each element is compared with its next element. In Ascending order below algorithm is used

* If the next element is less than the previous element, the next element will get swapped to prev index .
* Else the no swap.

Every element is with each n elements of an array

**int** n = array.length;

**for**(**int** i=0;i<n-1;i++) {

**for**(**int** j=0;j<n-i-1;j++) {

**if**(array[j+1] < array[j]) {

**int** tmp = array[j];

array[j] = array[j+1];

array[j+1] = tmp;

}

}

}

## **Selection sort**

The time complexity is o(n2) due to n\*n iterations, in selection sort the smallest element get selected and swapped to first index

* a third variable used to find the smallest number in exiting part of an array
* the nested loop is loops through the array to find the smallest, once loop ends the the third variable holds the index of smallest num
* swap the third variable index and i index.

int array [] = new int[]{10,3,1,6,2,4};

int mid =0;

for(int i=0;i<array.length;i++){

mid =i;

for(int j=i+1;j<array.length;j++){// smallest number

if(array[j]<array[mid]){

mid =j;

}

}

int tmp = array[i];

array[i] = array[mid];//smallest number is moved to initial index

array[mid]= tmp;

}

## **Insertion sort**

Best for partial sorted arrays, the time complexity O(n) and unsorted array worst O(n2)

* Start with second index of an array to loop though
* Store the current index element
* Declare and initialize the prev position variable i-1;
* Repeat the loop until j>=0 && arr[j]>key

If true assign the j to j+1

Decrease the j j-1

* Once repetition ends assign key to j+1;

The insertion sort is achieved by comparing previous elements with current element , if the current element (namely key) is less than previous element, the positions of all greater elements get shifted to right side, once loop ends the correct position of key gets find, then assigning key to that position , array get sorted.

Initial Array: {12, 11, 13, 5, 6}

Iteration 1:

key = 11

Compare 11 with 12 (element before it). Since 12 is greater than 11, move 12 to the right.

Array becomes {12, 12, 13, 5, 6}.

Now, assign the key to the correct position (j).

Array becomes {11, 12, 13, 5, 6}.

Iteration 2:

key = 13

Compare 13 with 12 (element before it). Since 12 is less than 13, stop the comparison.

Array remains {11, 12, 13, 5, 6}.

Iteration 3:

key = 5, j =2

Compare 5 with 13 (element before it). Since 13 is greater than 5, move 13 to the right.

Array becomes {11, 12, 13, 13, 6}.

Now, assign the key to the correct position (j).

Array becomes {11, 12, 5, 13, 6}.

Continue this process until 5 is in its correct position: {11, 5, 12, 13, 6}.

Iteration 4:

key = 6

Compare 6 with 13 (element before it). Since 13 is greater than 6, move 13 to the right.

Array becomes {11, 5, 12, 13, 13}.

Now, assign the key to the correct position (j).

Array becomes {11, 5, 12, 6, 13}.

Continue this process until 6 is in its correct position: {11, 5, 6, 12, 13}.

After these iterations, the array becomes sorted: {5, 6, 11, 12, 13}.

in this case j is the index , up to where elements are sorted

**for**(**int** i=1;i<arr.length;i++) {

**int** key = arr[i];

**int** j= i-1;

**while**(j>=0 && arr[j]>key) {

arr[j+1] = arr[j];

j = j-1;

}//while loop exits at sorted value

arr[j+1]= key;

}

**for**(**int** i:arr) {

System.***out***.println(i);

}

## **Merge sort**

Merge Sort is a divide-and-conquer algorithm that works by dividing the array into two halves, recursively sorting each half, and then merging the sorted halves. It is a stable and efficient sorting algorithm with a time complexity of O(n log n) in the worst, average, and best cases.

Merge sort achieved by dividing the array for all possible way by half, it is implemented in a mergeSort method while sorting the elements using merge method. Where is each half is sorted recursively.

Algorithm:  
need to implement two function one is for recursive call of merge and mergesort methods. Recursive

* Merge sort method > it have arguments array, left(0), and right(length-1).

Find the middle of the array by left+right/2 is equal to middle

Call mergesort method for left to middle and for middle+1 to right with arr

Call the Merge method for each divided array to sort and combine.

* Merge Method > where the divided parts of array sorted and merged

Create two arrays with size of left and right + middle for each half and copy the divide arrays elements to respective temporary array

And perform the sort logic for both temporary array assign to k index

If any elements left at any side of temporary array move them to k index

All about handling the array starting , ending and middle.

middle = starting + (ending-starting)/2

sub array from start to m => l to m

second array from m to end => m to end

And merging the sub array using the **two-pointer** approach from starting

Divide the array into possible sub arrays and sort the sub arrays

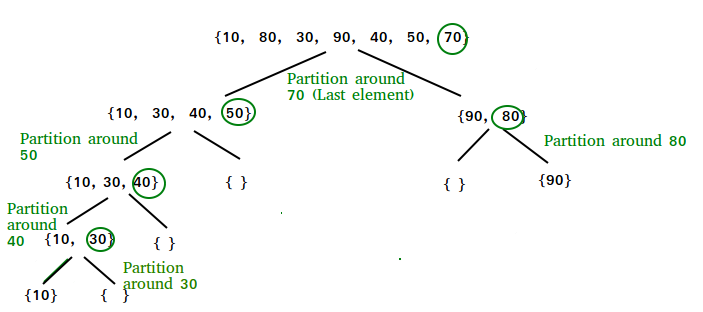
## **Quick sort**

Giving more priority to Pivot element. Radom pivot recrsive

as per context picking the pivot might be work, if that says array is partially sorted, I this case taking pivot as random can optimize the TC

Space complexity – O(1)

Time complexity – O(n logn) – worst-case best-case O(n2);



## **Heap sort**

Heap Sort is a comparison-based sorting algorithm that uses a binary heap data structure to build a max-heap or a min-heap. In the context of Heap Sort, we typically use a max-heap to sort elements in ascending order. Here's an overview of the Heap.

# knowledge of the application of data structures and algorithms

https://open.kattis.com/

### 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

Even palindrome – if the string contains even frequency of chars

Odd – if the middle char is single frequency.

permutation palindrome == even plindrome

*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;

# Print Matrix in spiral order

Time complexity is O(n)

The elements in matrix must be from 1 to input\*input in spiral order

1 2 3

8 9 4

7 6 5

Declare a array

Iterate for each side of the matrix.

From left index 0 to right <length

Maintain four variables for every side state ,

# Stack and Queue in Java Script:

Stack- LIFO in Java Script offers push and pop methods to add and remove the element from stack

Where Queue – FIFO offers push and shift methods to add and remove the element from stack

var evalRPN = function(tokens) {

const stack = [];

for (let token of tokens) {

if (!isNaN(token)) {

stack.push(parseInt(token));

} else {

const operand2 = stack.pop();

const operand1 = stack.pop();

switch (token) {

case "+":

stack.push(operand1 + operand2);

break;

case "-":

stack.push(operand1 - operand2);

break;

case "\*":

stack.push(operand1 \* operand2);

break;

case "/":

stack.push(parseInt(operand1 / operand2));

break;

}

}

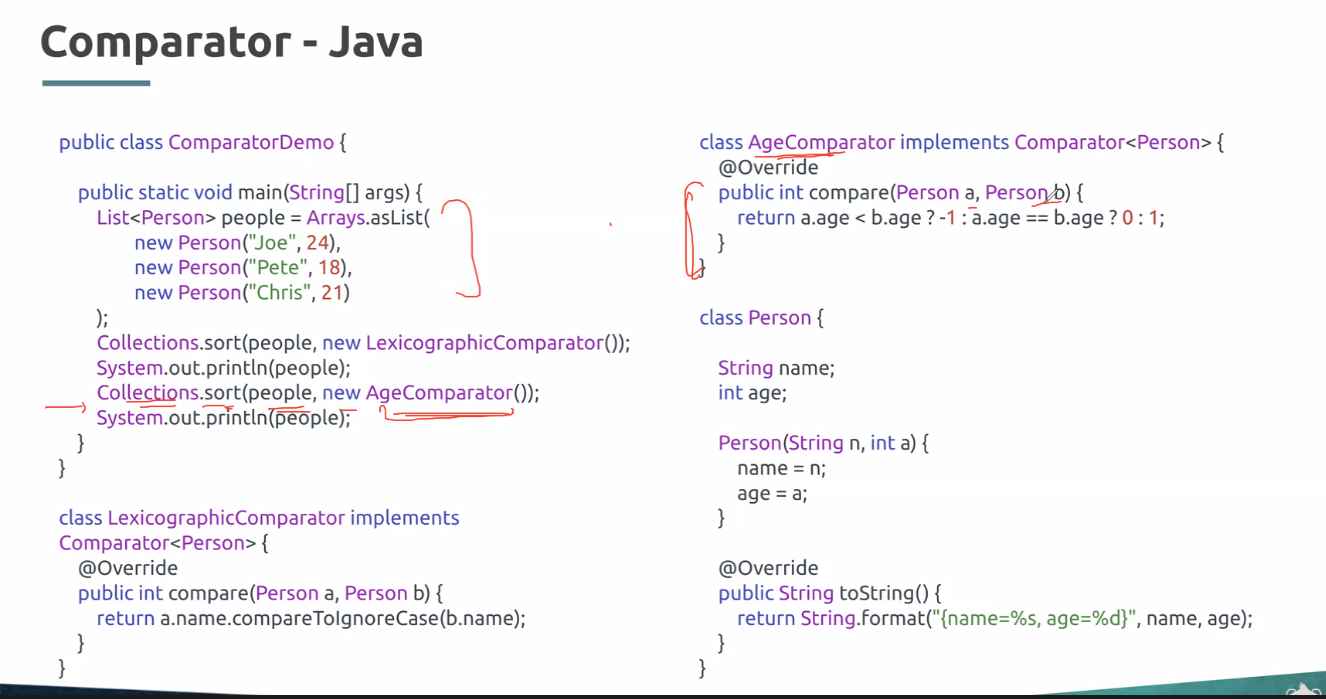
}

return stack.pop();

};

In Java queue offers add or offer to add element, remove to delete element and peek to view top value.

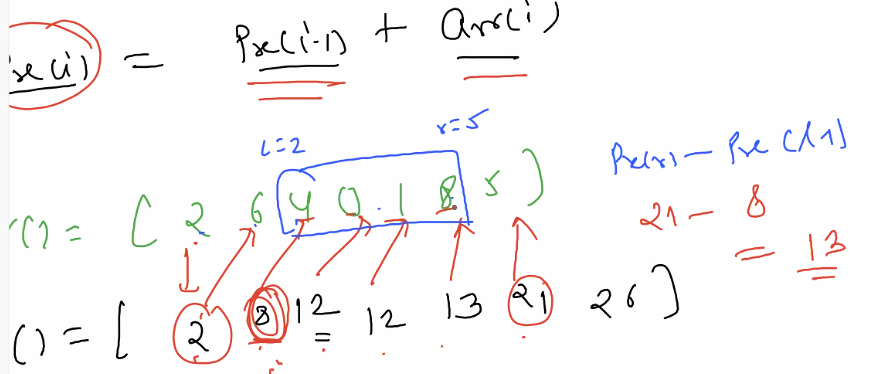
In Java Stack offers push to add element , pop to remove , peek to show top element, search



# Prefix & suffix of Sum in array

## Prefix

Left sum



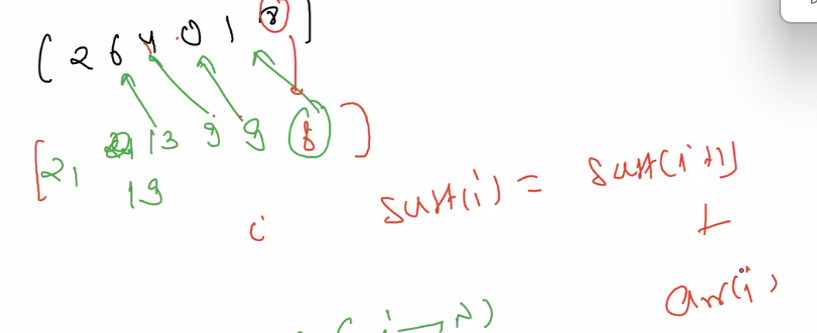
**prefix(i) = prefix(i-1) +arr(i)**, then the loop must be I=1 as prefix(0)=arr(0)

And to find sum L to R in original array , using the prefix sum **sum(L to R) = prefix(R) -prefix(L)**

## Suffix

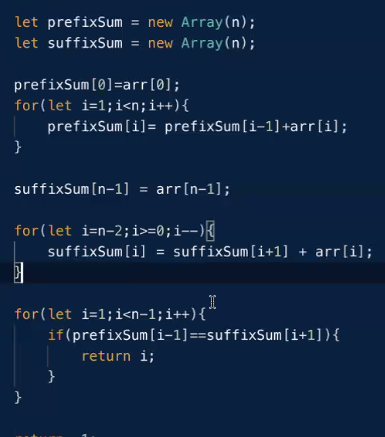
Right sum

**suffix(i) = suffix(i+1) +arr(i)**, then the loop must be I=n-2 as prefix(n-1)=arr(n-1)



Prefix and suffix Sum arrays best used to find the subarray asper requirements like maxSum or Sum =1 in given array with sliding window aproach

Usecase:

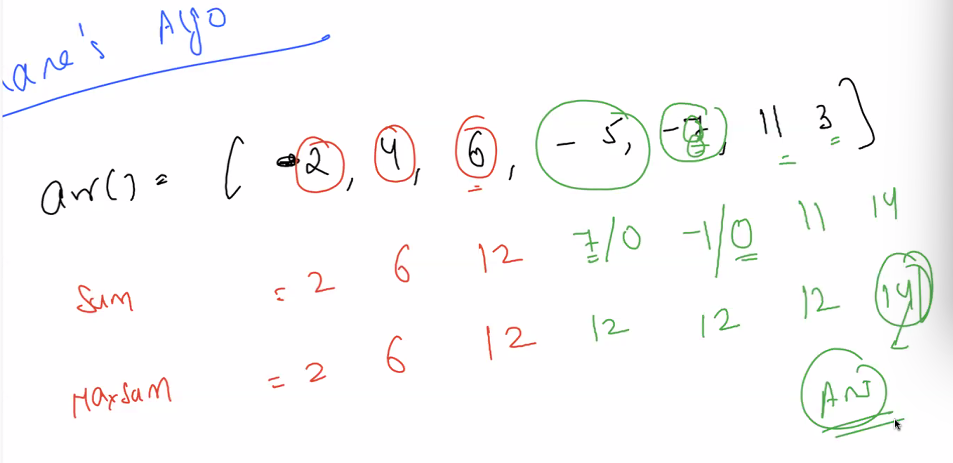


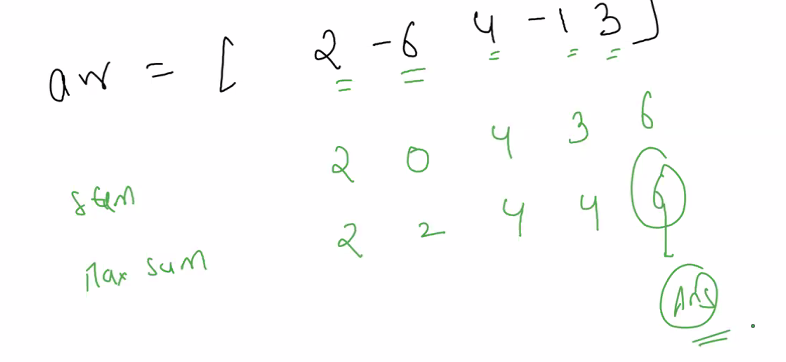
# Kadane's Alogorithm

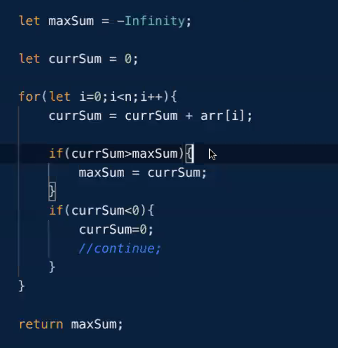
if sum<maxSum->skip for max sum

if sum getting negative <0 adding an element then skip the element. And reset the sum

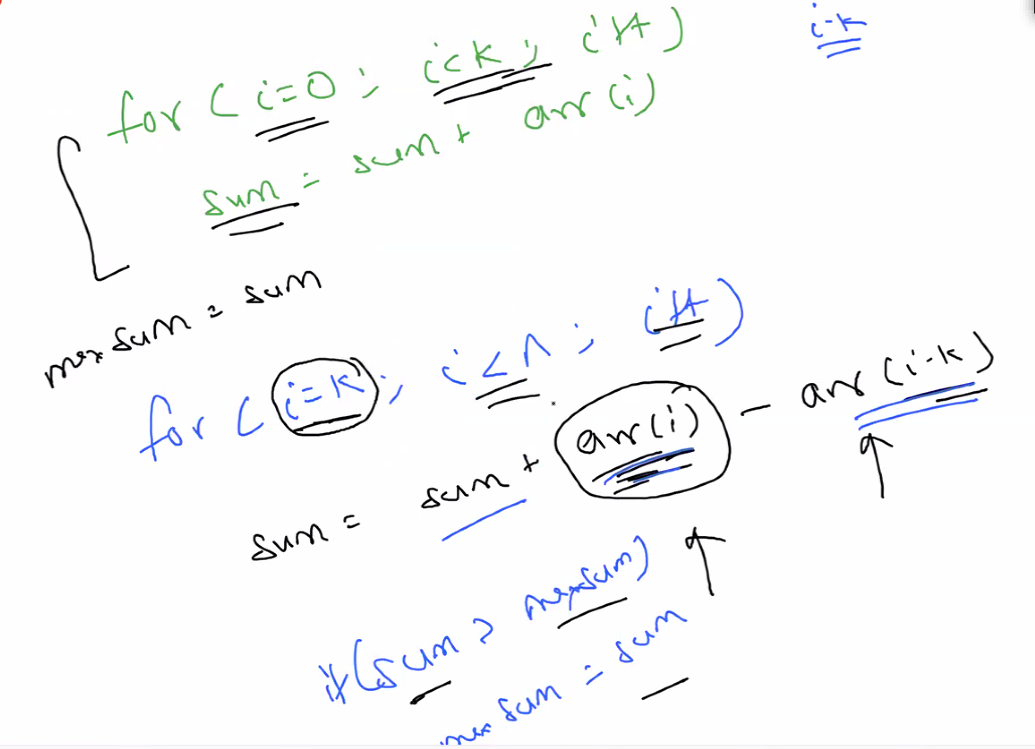
rather skipping when current sum falling below zero just reset to zero.



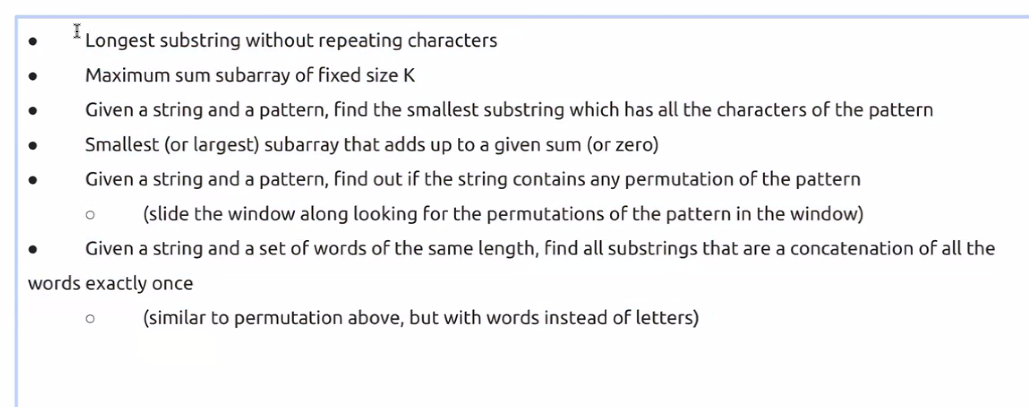


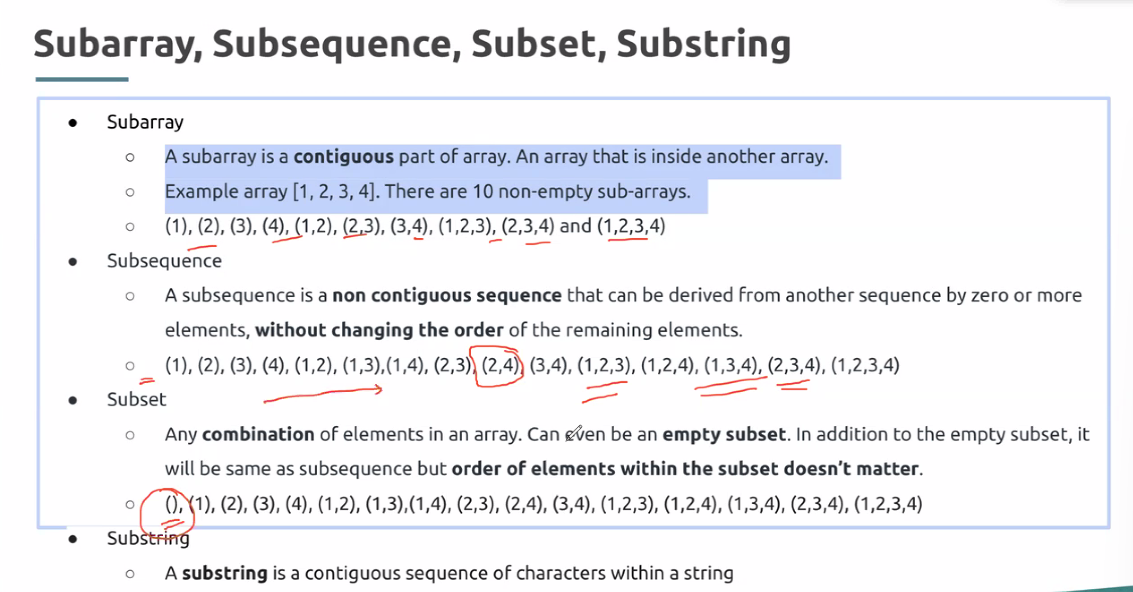


## Sliding window pattern



Used to find the max sum of an subarray in array





To find the length of two end point is > right -left+1

Find the max sum sub array in given array with size k of sub array

First line of input contains two integers N and K, where N is the size of the array and K is the subarray size.

Second line of input contains N integers denoting the array elements.

18 3

7 49 -73 58 -30 -72 44 -78 23 9 40 65 -92 42 87 3 27 -29

Output:

-17 i 2

34 i 3

-45 i 4

-44 i 5

-58 i 6

-106 i 7

-11 i 8

-46 i 9

72 i 10

114 i 11

13 i 12

15 i 13

37 i 14

132 i 15

117 i 16

1 i 17

132

possible sub arrays:

7 49 -73 = -17, loop 3

49 -73 58 = when sum sub array size id 3 => -17 + 58 -7 = 34

Current i = 3

i.e. => sum +=arr[i] ; sum = sum – arr[loop - i]

loop--;

-------------------------

-73 58 -30

-72 44 -78

44 -78 23

-78 23 9

23 9 40

9 40 65

40 65 -92

65 -92 42

92 42 87

42 87 3

87 3 27

3 27 -29

# Array Two Pointer

Overview

The two pointer method is an optimal approach used to improve approaches that require multiple nested loops, to single traversals using multiple pointers. It is used across linear data structures like arrays and linked lists.

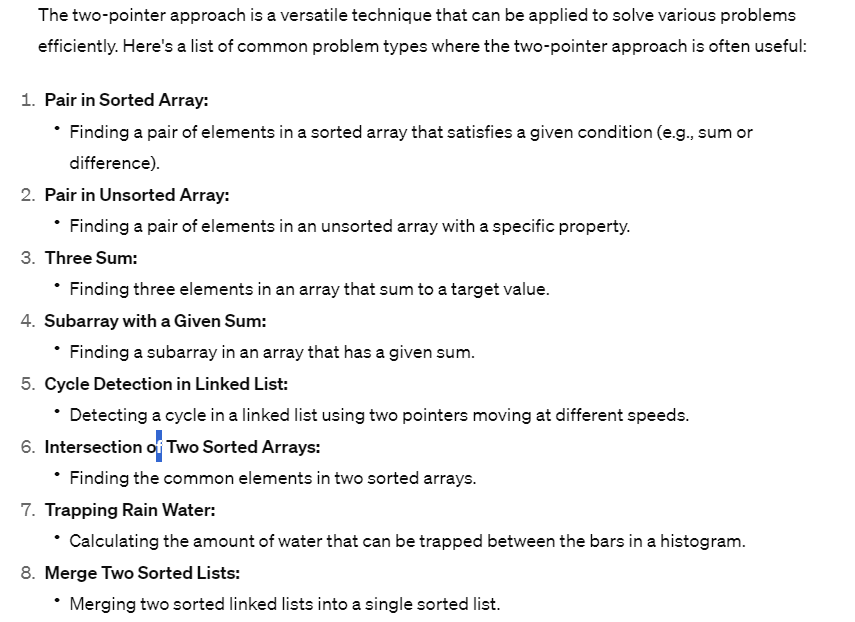
It involves using two variables, called pointers that traverse the array to find pairs/triplets of elements that satisfy a given condition, such as max sum, max product, and so on.

The idea is to position 2 pointers which can traverse at their own speed and in independent directions to solve the problem.

This approach would give you the right answer in linear time, since there's only one traversal, even though two pointers are used to do it.

How will you identify if a given problem can be solved using the two pointer approach : If the problem is along the lines of finding the max sum/product of a pair/triplet/quadruplet of elements, or a group of elements that sum to a given number, and so on, high chances are that it can be solved using the two pointer approach.

time complexity of this solution is O(n)





# Array Prefix Sum:

Overview

Prefix sum is a technique of storing cumulative sums of array elements in another array, called the prefix sum array.

prefixSum[i] will hold the sum of all elements from a[0] to a[i]. where a is the original array.

This technique is used in problems where you need to work with sums of specific ranges inside the array.

It helps in storing sum of all values, because otherwise, you'd need to calculate the sum repeatedly, thereby increasing the time complexity

Some standard problems solved using this approach include finding the maximum subarray sum, finding the longest span with same sum in two arrays, and so on.

If you come across a question that requires summation of array elements with some conditions, you should try to see if prefix sum can be applied

# Array Sorting

Overview

Sorting a data structure means to reorder elements based on some criteria, such as increasing order of their values.

Sorting arrays is a very frequently used programming pattern which is used to simplify a lot of other data structure question patterns. For instance, binary search works only on sorted arrays.

But it's not always integer arrays that need sorting. Often times, we might need to sort maps based on the keys, or strings based on the lexical ordering of the letters, or on their lengths.

There are some common sorting methods like Insertion sort, selection sort, merge sort, quick sort which are commonly asked in interviews. You should know how they work.

You don't need to write the sorting methods from scratch while solving problems. All languages also provide inbuilt sort methods that sort in O(n log n) time

These are the sorting methods that you should use when solving problems that require sorting before moving on to other parts, such as sorting for binary search.

Custom Sort:

Array.sort(callback function)

The callback function have access to values of array , can code as per our requirements to sort.

For strings sort can be use lexicographic sorting.

# Array Binary Search

Overview

Binary search is the method of repeatedly dividing a sorted array into two halves, and discarding one half each time, while looking for a value.

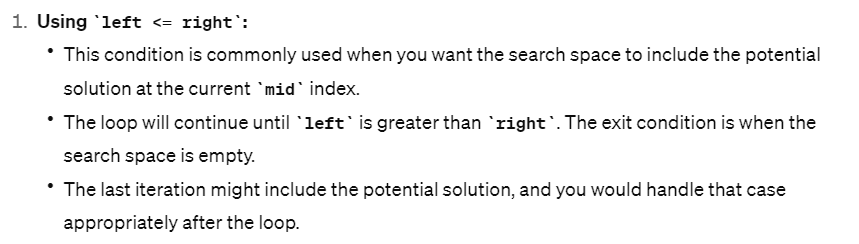
This is the most popular search technique used to find elements with a specific condition in sorted data structures, commonly arrays.

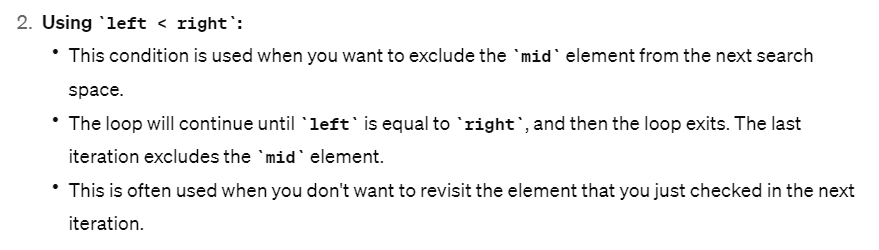
The technique works on the premise that given an element in a sorted array, the number can either be on the left of that element or the right of that element, and we can easily discard the half which we're sure will not have it.

This method finds the element in O(log N) (base 2) time, as opposed to normal linear search that takes O(N) time.

Apart from being an important problem in itself, binary search is commonly employed in continuation with.

When space must include mid index





Binary Search is a powerful algorithmic technique that is used to efficiently find or decide the position of a particular element in a sorted array or a sorted list. It has several variants based on the specific problem requirements. Here are some of the common types:

1. **Standard Binary Search:**
   * **Problem:** Find the index of a specific element in a sorted array.
   * **Approach:** Compare the target value with the middle element, and based on the result, narrow down the search to the left or right half of the array.
2. **Binary Search for Closest Element:**
   * **Problem:** Find the element in a sorted array that is closest to a given target.
   * **Approach:** Instead of terminating when the target is found, continue the search until **left** and **right** pointers meet. The result will be the element that is closest to the target.
3. **Binary Search for First Occurrence (Leftmost Binary Search):**
   * **Problem:** Find the leftmost occurrence of a target element in a sorted array.
   * **Approach:** If the middle element is equal to the target, continue the search in the left subarray. If not, adjust the pointers accordingly to move left or right.
4. **Binary Search for Last Occurrence (Rightmost Binary Search):**
   * **Problem:** Find the rightmost occurrence of a target element in a sorted array.
   * **Approach:** If the middle element is equal to the target, continue the search in the right subarray. If not, adjust the pointers accordingly to move left or right.
5. **Binary Search on Infinite Sorted Array:**
   * **Problem:** Find the position of a target element in an array of unknown size.
   * **Approach:** Use an exponential search to find a range where the target may exist, and then perform binary search within that range.
6. **Binary Search on Rotated Sorted Array (Modified Binary Search):**
   * **Problem:** Find an element in a rotated sorted array.
   * **Approach:** Divide the array into two halves. One of the halves will always be sorted. Check if the target is in the sorted half; if yes, continue the search in that half; otherwise, search in the other half.
7. **Count of Elements in a Sorted Array:**
   * **Problem:** Count the number of occurrences of a target element in a sorted array.
   * **Approach:** Use two binary searches to find the leftmost and rightmost occurrences of the target and calculate the count.

These variants demonstrate the flexibility and efficiency of binary search in solving a wide range of problems, particularly those involving sorted arrays or lists. Each variant adapts the basic binary search technique to suit the specific requirements of the problem at hand.

## Standard Binary Search:

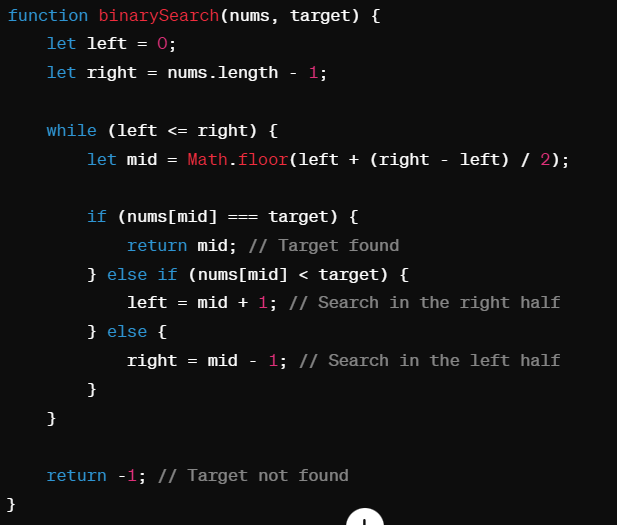
* This a process just handling the space between left and right side of an array , as how mid pointer is respective to target.

**Example:** Suppose you have the following sorted array: **[1, 3, 5, 7, 9, 11, 13, 15, 17]**.

**Target Element:** **7**

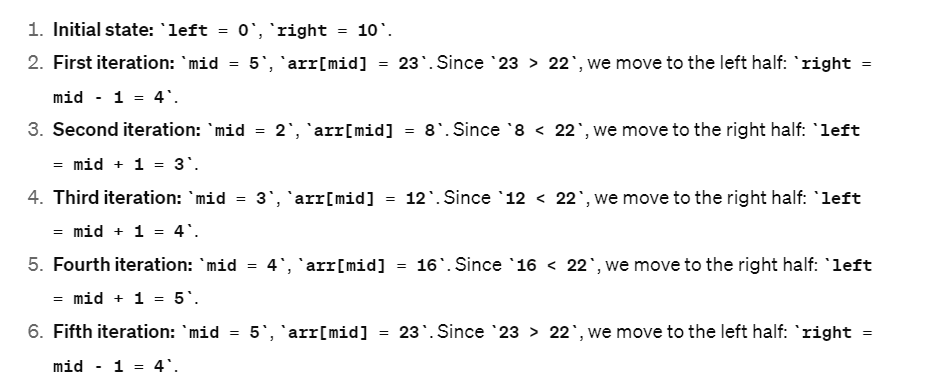
Binary search only works for sorted array. mid = Math.floor(left+(right-left)/2)

* Adjust the left and right as per the mid value to the target, if mid > target , move to the left side. If less than the target move to the right side of an array.



## Binary Search for Closest Element:

* The number which is just less than or greater to the target value
* Left, right, mid
* Rather returning an element , we just iterate through the whole array, then return right or left will be the result
* Move towards right until the target is less than the target
* Maintaining a closet variable , to track the mid-target <closet-target, closet =mid





## Binary Search for First Occurrence (Leftmost Binary Search):

* Store the index if mid == target , move left side. The store index will not update until next equal number finds.
* If the mid < target move right side2, else if mid >target move left side



## Binary Search for Last Occurrence (Rightmost Binary Search):

* Store the index if mid == target , move right side. The store index will not update until next equal number finds.
* If the mid < target move right side2, else if mid >target move left side



## Binary Search on Infinite Sorted Array:

When dealing with an infinite sorted array, a standard binary search is not feasible because we don't know the size of the array. However, we can use an approach called "exponential search" to find a range where the target might exist, and then perform binary search within that range. Here's an example in JavaScript:



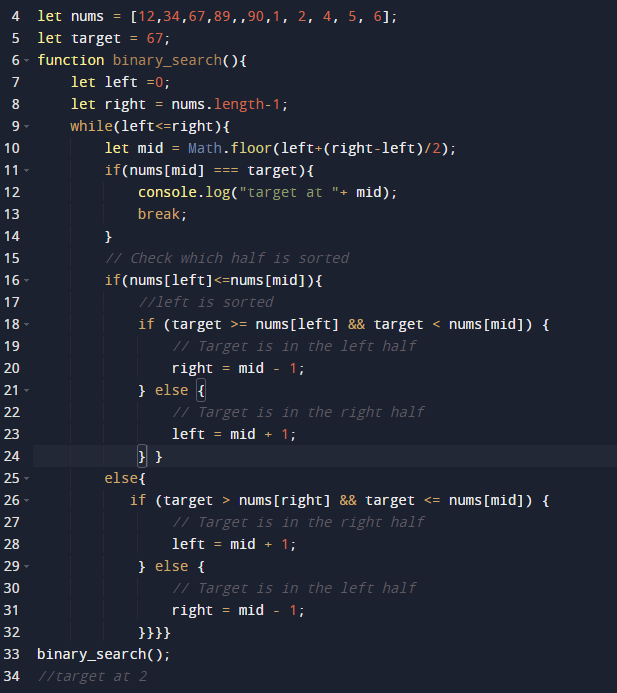
## Binary Search on Rotated Sorted Array (Modified Binary Search):

* As rotated array, left half array will be greater elements, and right side smaller elements.
* Find the pivot index, where the nums[mid]>nums[mid+1] store
* If(target< nums[n-1]) – the target must be on index pivot+1 to nums.length-1 – right half
* Else target> nums[n-1] – the target must be on index 0 to pivot – left half



Or follow below best suites for larger arrays

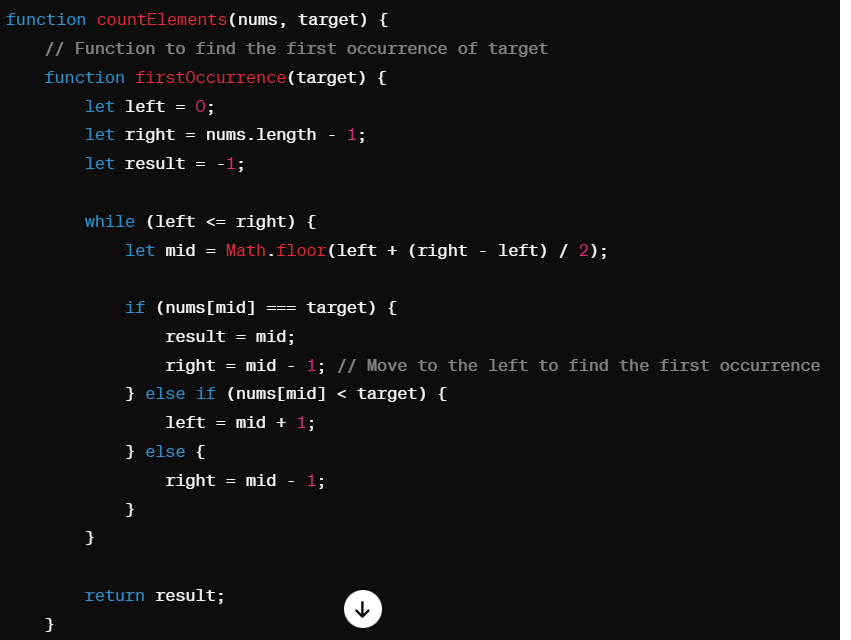
* As rotated array, left half array will be greater elements, and right side smaller elements.
* If(nums[left]<=nums[mid) – the target must be on left half – and check target < mid -> it wil left side else right half of left half
* Else nums[n-1]>target – the target must be on index – right half

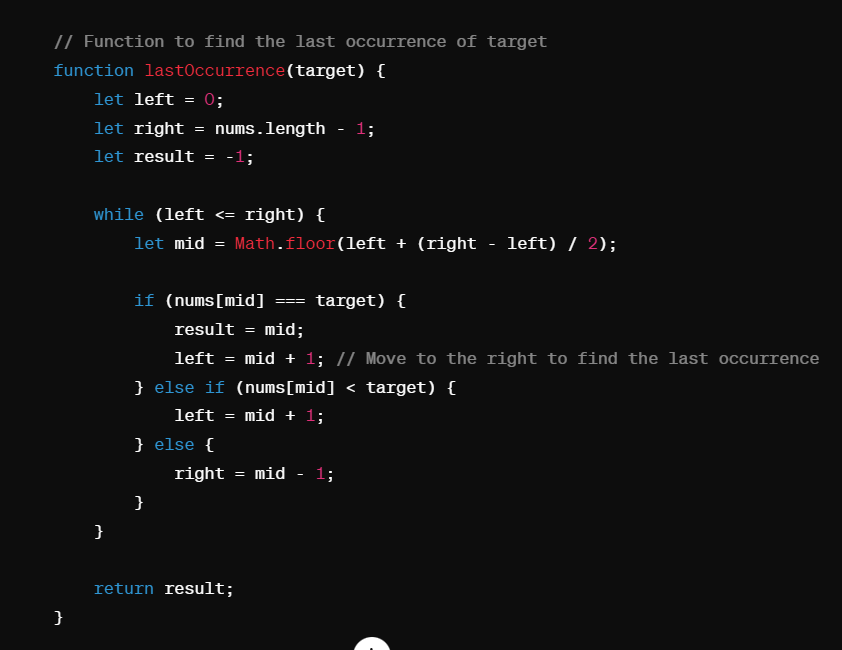


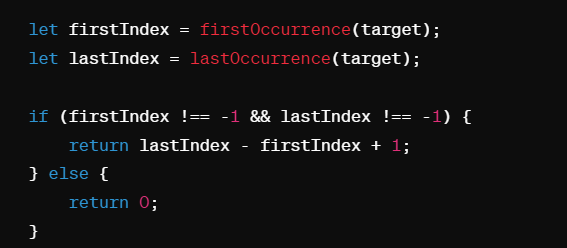
## Count of Elements in a Sorted Array:

* Combine both first occurrence and last occurrence binary search algorithms
* When mid === target,
* count left side occurrences 0 to <mid
* count right side occurrences mid to n-1
* combine both -Tc -> O(n)

OR TC -> O(log n)







## Modified Binary Search:

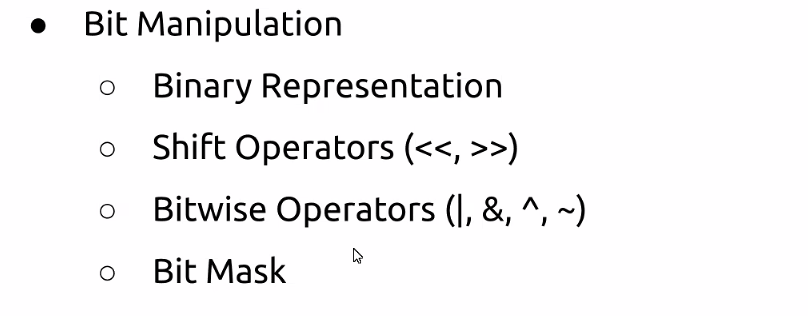
Finding an element in sorted and rotated element

Finding minimum speed required to read all piles of books

Finding the kth smallest element in sorted matrix

# Merge Interval Pattern

# Bit Manipulation



Decimal to Binary

(15)8 => ( )2

Divide and get reminder

Binary to Decimal

Sum all the digits it position power with 2^

## Binary operators

AND, OR, XOR , negation - > Multiply, Addition, toggle , flip respectively

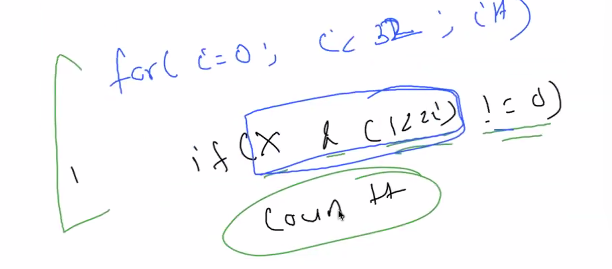
Left shift operator -> adding bits to number

1<<2 -> 100

Right Sift operator -> deleting the bits

5>>2 -> 1

Set and unset



DSA, CS fundamentals, projects, Resume. to Everyone

Gate smahers

Take u forward

CS -> DBMS and OOPS

<https://www.pramp.com/dashboard#/>

<https://topmate.io/>

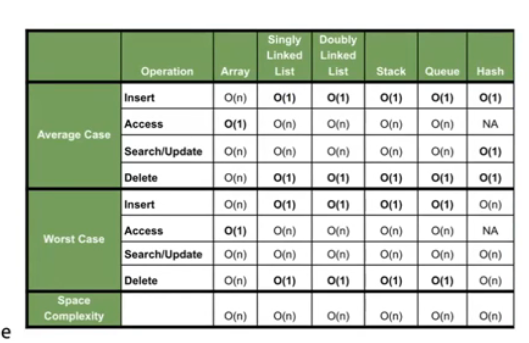
<https://mentro.tech/>

<https://wellfound.com/>

<https://www.instahyre.com/>

Kushdeep Mittal

# DSA -201



## **Stacks**

Stacks is linear data structure which has a working phenomenon of Last in first out (LIFO) store the any type of data types .

* **Stack as a Concept:** The concept of a stack is abstract and doesn't dictate data types. It focuses on the order of adding and removing elements.
* **Stack Implementation:** When you create a stack in a programming language, it's implemented using an underlying array or linked list. This underlying data structure is what actually stores the data.

**Methods:**

**1. push(item):**

* Functionality: Adds a new element (item) to the top of the stack.
* Time Complexity: Usually O(1) (constant time) in most implementations, as it involves adding to the beginning or end of an underlying data structure (array or linked list) with efficient access.

**2. pop():**

* Functionality: Removes and returns the element at the top of the stack.
* Time Complexity: Often O(1) (constant time) for similar reasons as push().
* Error Handling: In some implementations, if the stack is empty and pop() is called, an exception or error might be thrown indicating an "underflow" condition (trying to remove from an empty stack).

**3. peek():**

* Functionality: Returns the element at the top of the stack without removing it.
* Time Complexity: Generally O(1) (constant time), similar to push() and pop().

**4. isEmpty():**

* Functionality: Checks if the stack is empty and returns a boolean value (true if empty, false otherwise).
* Time Complexity: Usually O(1) (constant time) as it can efficiently determine the size of the underlying data structure.

**5. size():**

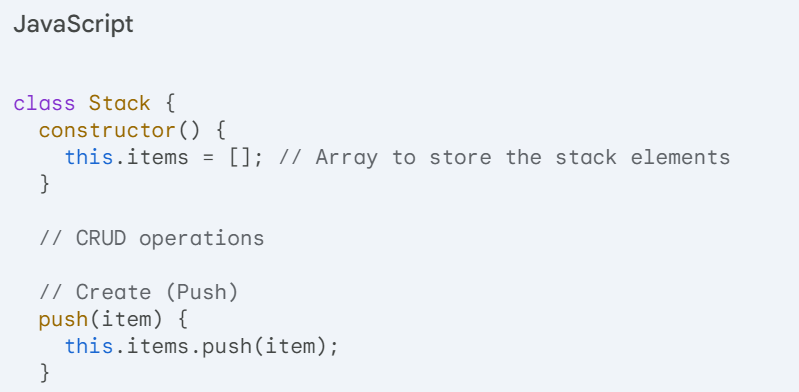
* Functionality: Returns the current number of elements in the stack.
* Time Complexity: Often O(1) (constant time), similar to isEmpty().

**Additional Methods (may vary depending on implementation):**

* search(item): Returns the position (index) of a specific element in the stack, if found. Time complexity can vary depending on the implementation (might involve iterating through the stack).
* clear(): Removes all elements from the stack. Time complexity can be O(n) (linear) in some cases if it involves iterating to remove elements one by one.

**UseCase –** undo/redo progress

JavaScript doesn't have a built-in Stack class, we can easily create a stack using an array and implement CRUD operations (Create, Read, Update, Delete) on it

****

## **Queue**

Standing in a line -FIFO top up of array / linked list

In Java Script Queue is implemented by list/array , so push and shift methods are used

**Queue:**

* **Concept:** A Queue follows the **First-In-First-Out (FIFO)** principle. It's like a line at a store, where the first person in line is the first one served.
* **Implementation:** Similar to stacks, queues are often implemented using arrays or linked lists. However, the access points differ.

**Similarities to Stacks:**

* **Data Type Flexibility:** Queues themselves don't have a specific data type. They can store various data types depending on the underlying implementation.
* **Custom Objects:** You can create queues that store custom objects defined in your program.

**Access Points:** A queue typically uses two pointers:

* **Front:** Points to the element at the front of the queue (the one to be removed next).
* **Rear (or Back):** Points to the element at the back of the queue (the most recently added one). Stacks typically only use one pointer (top) that points to the top element.

**Core Methods:**

* **enqueue(item): add or push**
  + **Functionality:** Adds a new element (item) to the **back** of the queue. This is sometimes called add or push (not to be confused with stack's push).
  + **Time Complexity:** Usually O(1) (constant time) in most implementations, as it involves adding to the end of an underlying data structure (array or linked list) with efficient access.
* **dequeue(): poll or remove**
  + **Functionality:** Removes and returns the element at the **front** of the queue (the one that was added first). This is sometimes called poll or remove.
  + **Time Complexity:** Often O(1) (constant time) for similar reasons as enqueue().
  + **Error Handling:** In some implementations, if the queue is empty and dequeue() is called, an exception or error might be thrown indicating an "underflow" condition (trying to remove from an empty queue).
* **peek():**
  + **Functionality:** Returns the element at the **front** of the queue **without removing it**.
  + **Time Complexity:** Generally O(1) (constant time), similar to enqueue() and dequeue().
* **isEmpty():**
  + **Functionality:** Checks if the queue is empty and returns a boolean value (true if empty, false otherwise).
  + **Time Complexity:** Usually O(1) (constant time) as it can efficiently determine the size of the underlying data structure.
* **size():**
  + **Functionality:** Returns the current number of elements in the queue.
  + **Time Complexity:** Often O(1) (constant time), similar to isEmpty().

**Additional Methods (may vary depending on implementation):**

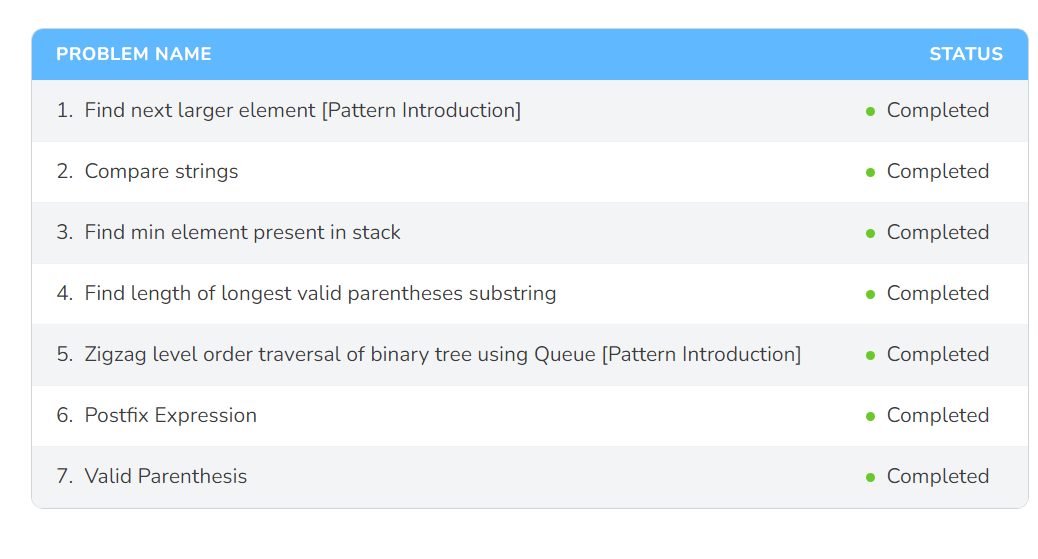
* **search(item):** Returns the position (index) of a specific element in the queue, if found. Time complexity can vary depending on the implementation (might involve iterating through the queue).
* **clear():** Removes all elements from the queue. Time complexity can be O(n) (linear) in some cases if it involves iterating to remove elements one by one.

**Queue Applications:**

Queues are used in various scenarios where the order of processing matters and elements need to be handled in a FIFO manner:

* **Task Scheduling:** Operating systems use queues to manage processes waiting for CPU time.
* **Event Handling:** Queues can buffer events (e.g., mouse clicks, network requests) for processing in a specific order.
* **Breadth-First Search (BFS) Algorithms:** BFS algorithms in graphs or trees often use queues to explore neighboring nodes in the order they are encountered.

Stack and Queue problems



## **Heaps**

A map, also known as a dictionary or associative array, is a fundamental data structure that stores data as a collection of **key-value pairs**. Each key acts as a unique identifier for its corresponding value. This allows for efficient retrieval of values based on their keys.

Here's a breakdown of key aspects of maps:

**Key Concepts:**

* **Key:** A unique identifier for a value within the map. Keys can be strings, numbers, objects, or any data type that can be used for comparison (hashable).
* **Value:** The actual data associated with a key. Values can be of any data type.
* **Key-Value Pair:** A combination of a key and its corresponding value.

**Operations:**

* **Insertion:** Add a new key-value pair to the map.
* **Deletion:** Remove a key-value pair from the map based on the key.
* **Lookup:** Retrieve the value associated with a specific key.
* **Size:** Determine the number of key-value pairs in the map.
* **Empty Check:** Verify if the map contains no key-value pairs.

**Time Complexity:**

* **Insertion:** Typically O(1) (constant time) in the average case for well-implemented hash tables (common map implementation).
* **Deletion:** O(1) (constant time) on average.
* **Lookup:** O(1) (constant time) on average.

**Applications:**

Maps are incredibly versatile and used in various scenarios:

* **Configuration Files:** Storing application settings where keys represent options and values hold the corresponding configurations.
* **User Data:** Managing user profiles where keys might be usernames and values could be user preferences, addresses, etc.
* **Caching:** Storing frequently accessed data for faster retrieval by key.
* **Symbol Tables:** Mapping variable names (keys) to their memory addresses (values) in compilers.
* **Graphs:** Representing relationships between entities where keys represent nodes and values store node data or connections to other nodes.

**Implementation:**

Maps are often implemented using **hash tables**. Hash tables use a hashing function to convert keys into unique indexes within an array. This allows for fast lookup of values based on their keys.

Here's a high-level overview of the hashing process:

1. **Hash Function:** The key is passed through a hash function, generating a hash value (typically an integer).
2. **Collision Handling:** If multiple keys map to the same hash value (collision), collision resolution techniques are used to store the key-value pair (e.g., separate chaining, linear probing).

**Core Methods:**

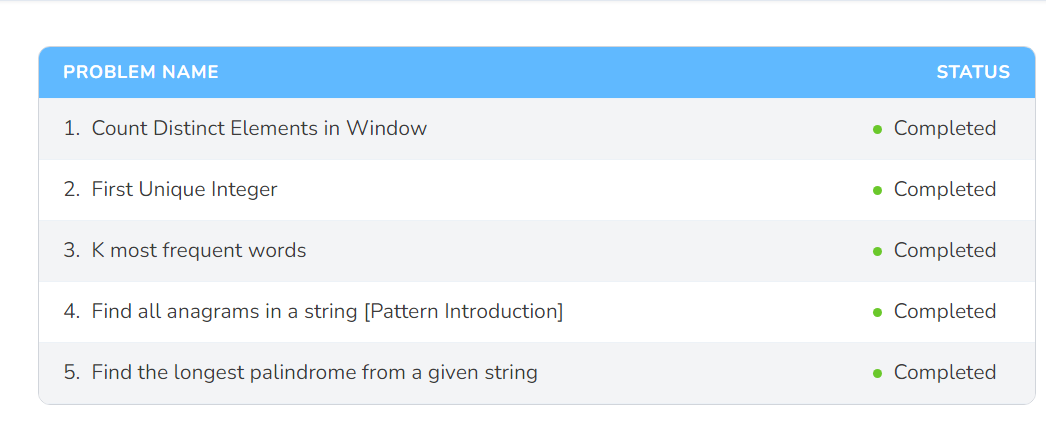
* **set(key, value):**
  + **Functionality:** Adds a new key-value pair to the map, or updates the value for an existing key.
  + **Parameters:**
    - key: The unique identifier for the data.
    - value: The data to be associated with the key.
  + **Time Complexity:** O(1) (constant time) on average for well-implemented hash tables (common map implementation).
* **get(key):**
  + **Functionality:** Retrieves the value associated with a specific key.
  + **Parameter:**
    - key: The key to be used for lookup.
  + **Return Value:**
    - The value associated with the key (if found), otherwise undefined, null, or a specific value indicating the key doesn't exist (depending on the language/implementation).
  + **Time Complexity:** O(1) (constant time) on average for successful lookups.
* **has(key):**
  + **Functionality:** Checks if the map contains a specific key.
  + **Parameter:**
    - key: The key to be searched for.
  + **Return Value:**
    - A boolean value (true if the key exists, false otherwise).
  + **Time Complexity:** O(1) (constant time) on average.
* **delete(key):**
  + **Functionality:** Removes the key-value pair associated with a specific key.
  + **Parameter:**
    - key: The key to be removed.
  + **Return Value:**
    - Often returns the value that was removed (if successful), or undefined, null, or a specific value indicating the key wasn't found (depending on the language/implementation).
  + **Time Complexity:** O(1) (constant time) on average.
* **size:**
  + **Functionality:** Returns the number of key-value pairs currently in the map.
  + **Return Value:**
    - An integer representing the number of elements in the map.
  + **Time Complexity:** O(1) (constant time) for efficient implementations.
* **clear():**
  + **Functionality:** Removes all key-value pairs from the map.
  + **Time Complexity:** O(n) (linear time) in the worst case, but can be faster (almost constant) in some implementations.

**Additional Methods (may vary depending on implementation):**

* **keys():** Returns an iterator or an array-like object containing all the keys in the map.
* **values():** Returns an iterator or an array-like object containing all the values in the map.
* **entries():** Returns an iterator or an array-like object containing all the key-value pairs in the map as an array of arrays (e.g., [[key1, value1], [key2, value2], ...]).
* **forEach(callback):** Executes a provided function for each key-value pair in the map.

**map and sets takes O(1) for Crud operations**

**map store only key from 256 chars – when keys are chars the space complexity is O(1)**

****

## **LinkedList**

LinkedList doesn’t have an library , must implement by array/list

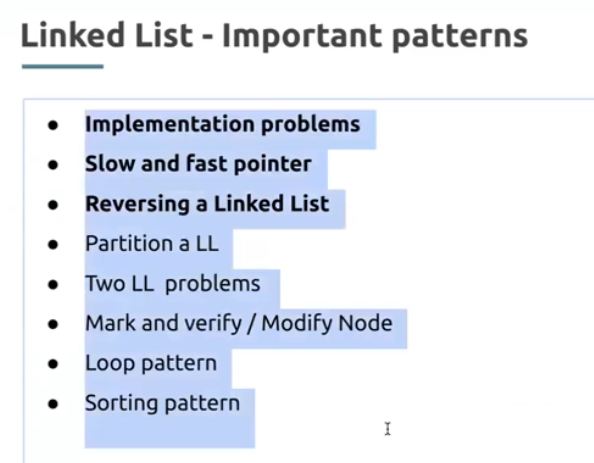
Array is continuous memory(array stored at one place in heap, is the place is not enough array won’t be created ) and inserting an element at first index time complexity is o(n).

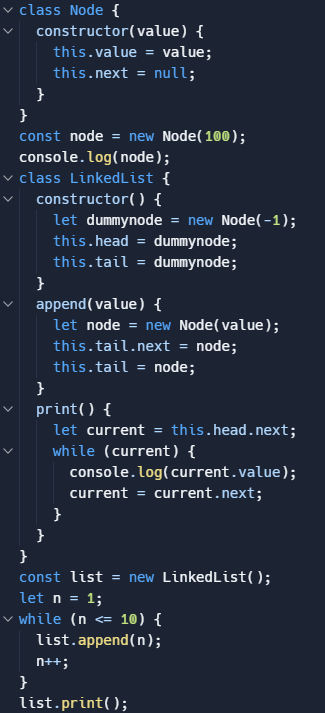
LinkedList is random memory occupation, each value linked with head and next available nodes (has next and value). Fast Insertion in LinkedList is O(1) has we need a node where to add the value and delete.

Search and update are slow due to the required sequential iteration

Node is an object of a class

while(hea.next != null){ head = head.next; }





Access are searching a value is slow in LinkedList, Delete and insert id fast

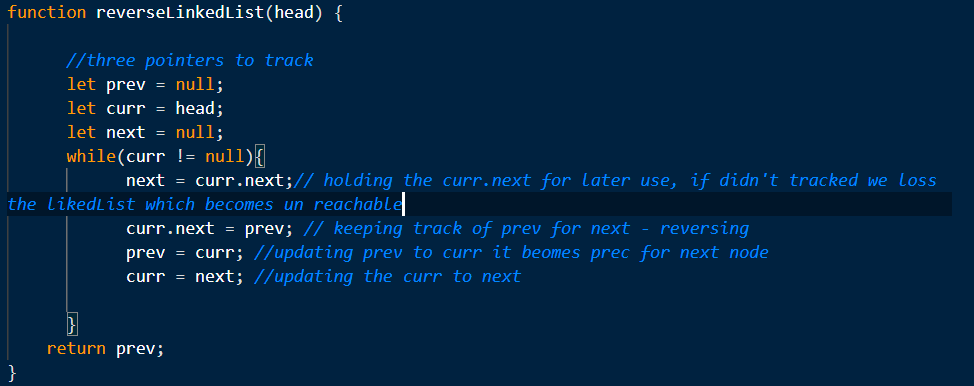
### Reverse a LinkedList

With additional data structure

* iterate though the linked list
* add node value to the new array
* reverse the array and create new LinkedList

without data structure

* change the current node pointer to prev node



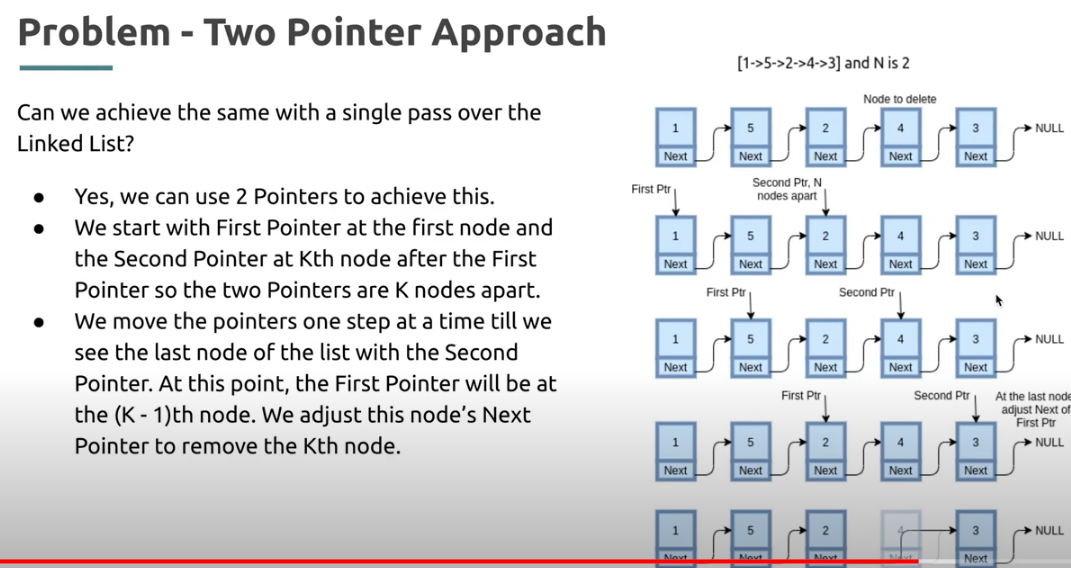
Dummy Node concept

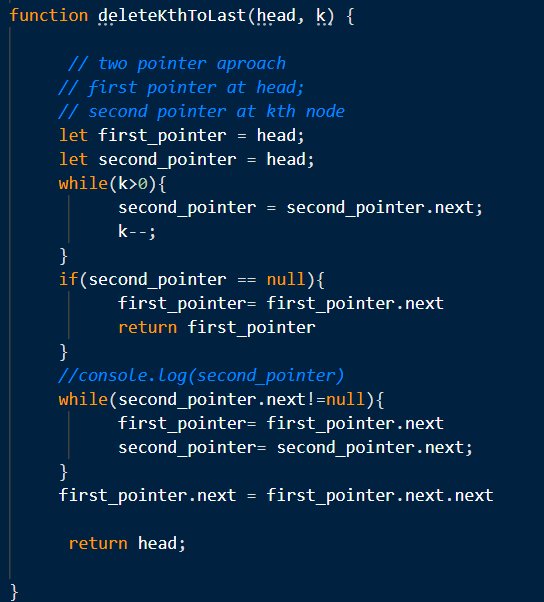
* Make the head and tail pointing to dummy node initial head
* Then only move the tail node tail.next = newNode; tail = node
* And start the Linked List from head.next , when retrieving

### Slow and Faster Pointer Pattern - to find mid node

### 

### Linked List Two pointer:

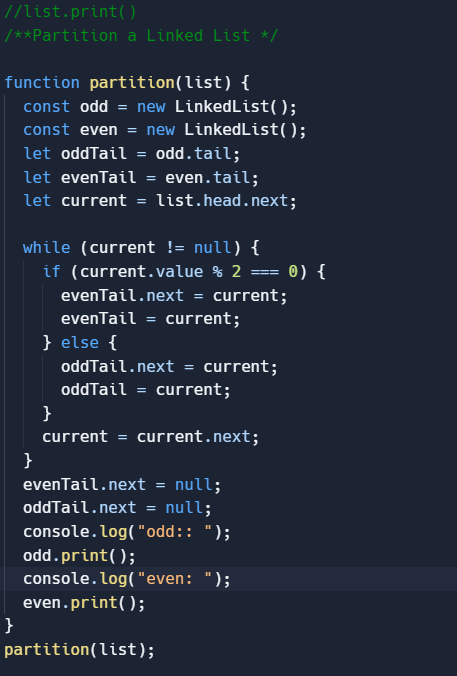




### Partition a Linked List: Break the linked list

Back track the tail from Linked List

* Create two Linked Lists for odd and even values
* Iterate through the list and separate the values and add to the lists as per their nature



### Merge two sorted Linked Lists ,Two LinkedList problems

* Follow the Merge two sorted arrays pattern with two pointer I and j

Take two sorted arrays, Create new array, Where loop through length of array, if(s[i]<b[j]) add a[i] to the new array i++ else add b[pointer] to the new array increment j++

Add left over elements to the new array (on while I,j less than length of arrays)

### Tortoise Hare Algorithm

Find the circle in LinkedList –

* Take two pointers for slow node which move slow.next and fast which moves fast.next.next
* Iterate up to no one is null
* Return at point where both points to same node

Find the circle in LinkedList (intersect) and remove link

The point where tortoise legs becomes equal, the distance from that point to intersect and the head node to intersect have same distance

* Take two pointers for slow node which move slow.next and fast which moves fast.next.next
* Iterate up to no one is null
* Stop at point where both points to same node
* Iterate through the head and iterate thought the stopped slow node
* The point where both are equal
* Return the node

### Merge Sort for Linked List

* Do the slow and fast pointer pattern to find break LL into half, where while loop continues upto fast!=null && fast.next != null
* Second half start from slow.next
* Recursively divide and conquer achieve sorting and merge two sorted Linked list



## **Binary Tree**

Non-linear

All nodes are connected , no cycle

At max two children’s for every node

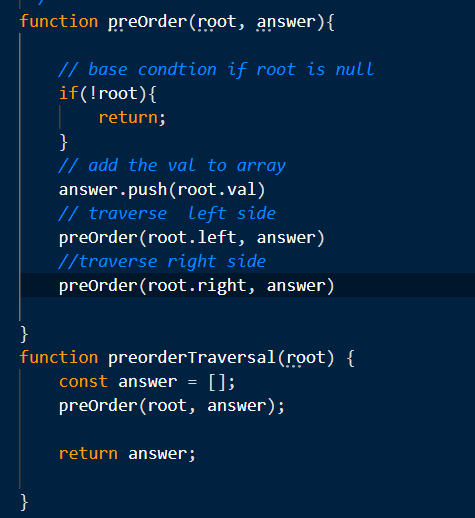
Level , Depth , and Height , sub tree

### Traversal of Binary Tree:

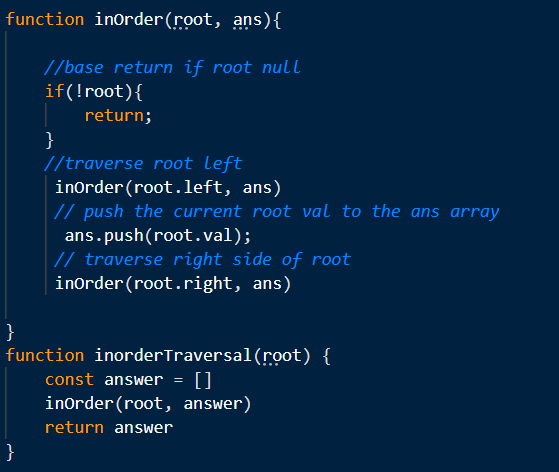
Depth First Search- DFS and Breadth first Search- BFS (level order - Queue)

DFS: (recursive / stack)

Pre Order – root , left, and right



In Order - left root right



Post Order – left right root

#### Inorder Successor in Binary Search Tree

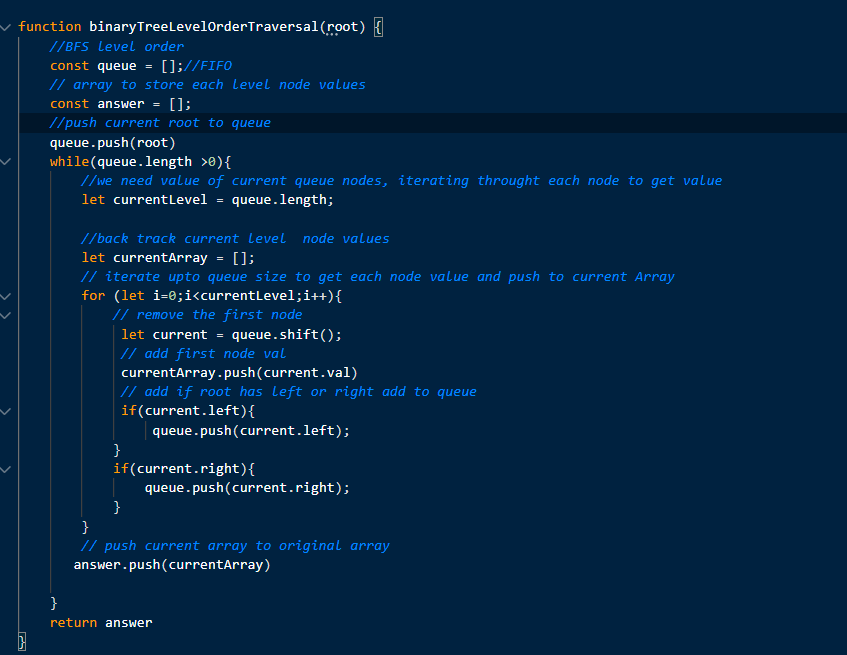
<https://www.geeksforgeeks.org/inorder-successor-in-binary-search-tree/>

### Balanced/UnBalanced Tree:

Difference between height of left and right should <2 for balanced tree

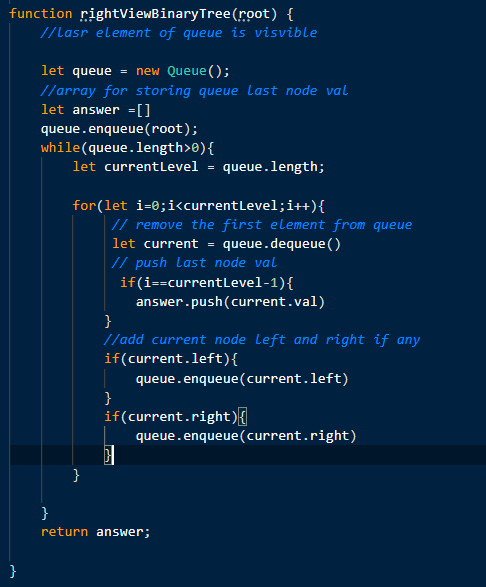
Diameter = height-1

### BFS Level Order Traversal – Top view left view , and right view

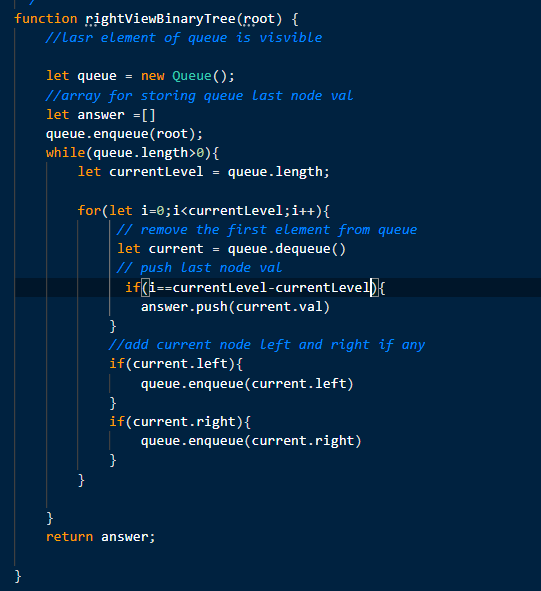


#### Level Order traversal

- Right view I == current level-1



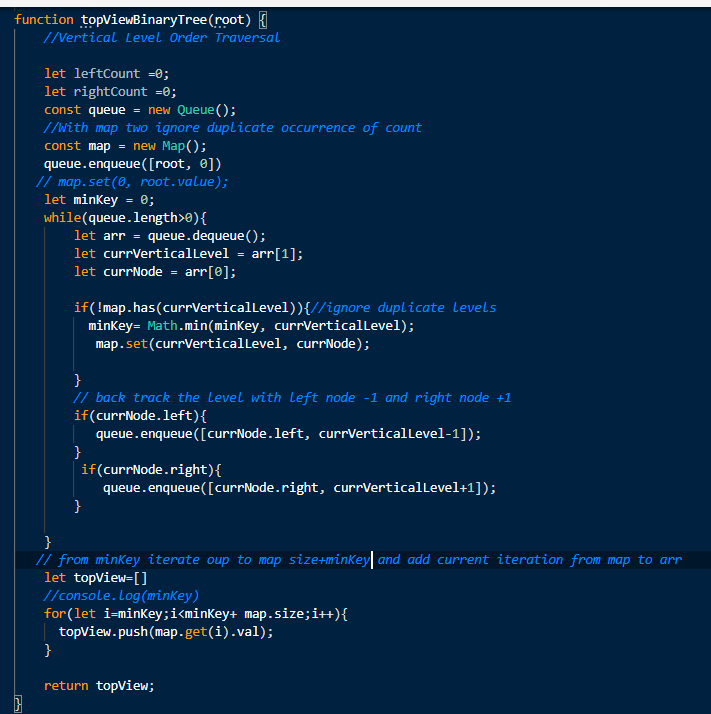
- Left view I ==0||currentLevel-currentLevel



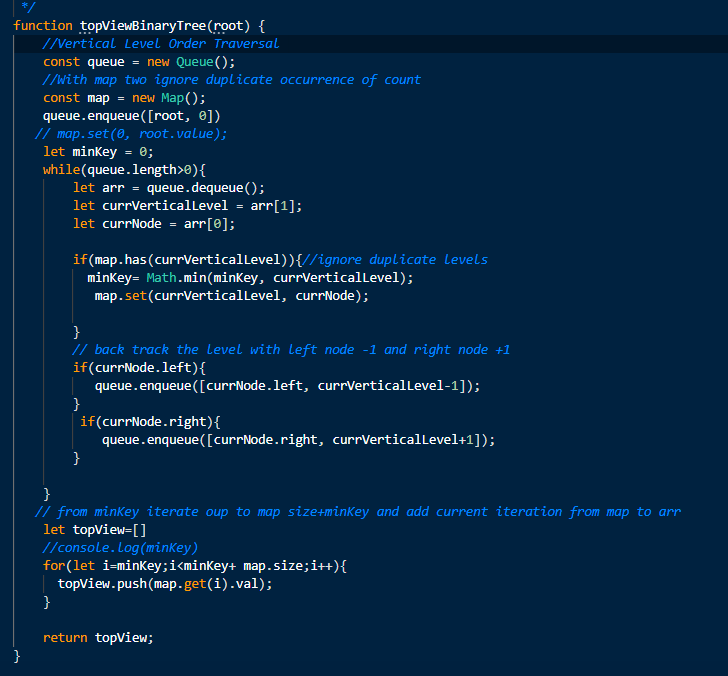
Vertical Level Order Traversal

- Top View =

With map two ignore duplicate occurrence of count



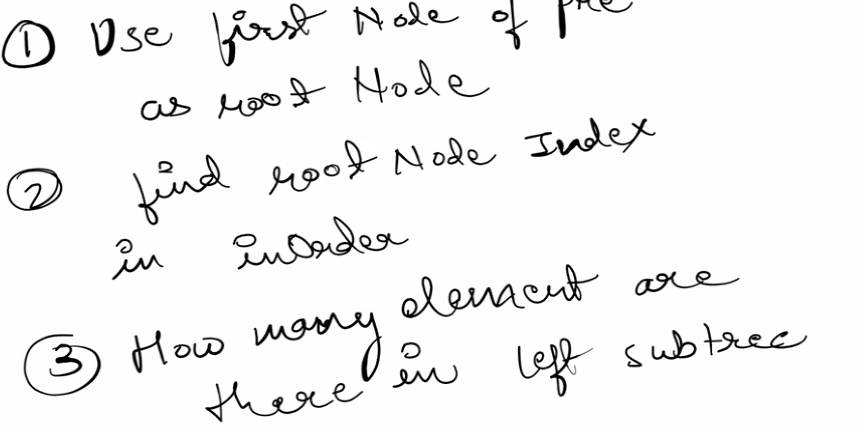
- Bottom View



### LCA – Least Common Ancestor

### Construct Binary tree from array

#### In order



Left sub tree – start of index and rootIndex-1

Right sub tree – rootIndex+1 to end of arrayx

#### Pre Order:

Size of left sub tree of in order – take elements after root (first index) the size of elements for left sub tree

Size of right subtree of in order – take elements after left completes (i.e firstIndex+1+size of left subtree to end)the size of elements for right sub tree



