DSA EMG JS

Table of Contents

What is DSA

- Data structure is a specific way to of organizing, storing and accessing data

- Algorithm is a set of instructions that tells computer how to do something.
  - Algo = step-by-step solution of the problem

Setup

- use vsc code editor
- setup

```
sudo apt install code
mkdir tuts && cd tuts
touch DSA.js
code .
```

- install `coderunner` extension
- F1 > set keyboard extension > run code to be `ctrl+enter`
- sometimes refresh js code run ,in case wrong output glitch in vsc terminal
- each `eg` is a challenge to practice :0. Good Luck !!!

Need of DSA

```
// ## Need of DSA
// eg1: search person in arr
console.log("eg1");
const eg1empDb = ['bali','bhati','bhaskar','rijusmit'];
const eg1findEmp = (db,person) => {
    for(let i=0; i<db.length;i++){
        if(db[i]===person){console.log(`Found ${person} at ${i}th index!!!`)}
    }
};
eg1findEmp(eg1empDb,"bhaskar");
eg1findEmp(eg1empDb,"richa");

// now even if we find bhaskar early, still it would check === with rest of array
// now this eg1empDb is our data structure
// & eg1findEmp is our Algorithm

// bali-king@war-machine:~/BaliGit/KintsugiStack$ node "/home/bali-
king/BaliGit/KintsugiStack/DSA_EMG_JS/DSA.js"

// eg1
// Found bhaskar at 2th index!!!
// bali-king@war-machine:~/BaliGit/KintsugiStack$

// O(n)
```

- Efficient Problem Solving
- Algorithmic Awareness
- Stronger Coding Skills

- Interview Success
- Coding Confidence
- Efficiency Expert
- Improved Logical Thinking
- Future-Proof Your Skills
- Innovation Inspiration
- Lifelong Learning
- Critical Thinking Champion

> Leads to a Great Problem Solver

Big O Notation

- it's a Notation/Convention/Measurement
- tells "if the code is good code or bad code ?"
- helps in scaleability, handling large data, fastness, accuracy

> BigO helps us to understand `how long [Time Complexity]` an algorithm will take to run or `how much [Space Complexity]` memory it will need as the amount of data it handle grows.

- Big O shows how your cooking time changes as more guests arrive for dinner.

`O(n)`

- [no need to worry about maths ;0]
- Signifies that the execution time of the algorithm grows linearly in proportion to the size of the input data (n).

> As the number of items in the input data increases, the time it takes for the algorithm to run increases correspondingly.

> In eg1, Imagine you have a list of employee names. To find a specific person (like “bhaskar”), you scan through each name in the list one by one. Even if you find “bhaskar” early, you still continue checking the remaining names. If the list has only 4 employees, it’s quick. But if it grows to 4,000 employees, it’ll take much longer because the algorithm still compares each entry. Here, the empDb is the data structure (where the data is stored) and findEmp is the algorithm (the step-by-step process to search). This demonstrates linear time complexity in a real-world scenario.

- 1 `for/while/etc loop` = O(n)
- 2 adj. loop = O(n) + O(n) = 2O(n) = O(n) `remove constants like here 2 in final time complexity`
- X adj. loop = X*O(n) = O(n)
- `remove constants in final time complexity`
- if 2 nested loop = O(n*n) = O(n^2)
- always remove nonDominant TC, eg: `3O(n^3) + 1000(n) + O(n^4) = (n^3) + O(n) + O(n^4) = O(n^4)`, because we always measure most bottleneck part !!!

`O(1)`

- fastest

- O(1), aka `constant` time, signifies that the execution time of an algorithm remains `constant` regardless of the input size.

> When you withdraw cash from an ATM, it takes roughly the same amount of time to dispense the money whether you withdraw ₹500 or ₹50,000. The number of bills in your account doesn't affect the withdrawal speed — it's a constant time operation.

```
// eg2: ATM Machine
console.log("eg2");
const eg2cashDb = [5,10,20,50,100,500];
const eg2findCash = (db,index) => db[index];
console.log(eg2findCash(eg2cashDb,4));
// here we don't care how big is data ;)

// eg2
// 100

// O(1)
```

> Imagine you have a box filled with items, and you know exactly where each item is located. To get a specific item, you go directly to its location, taking the same amount of time irrespective of how many items are in the box.

`O(n^2)`

- Indicates that the algorithm's execution time grows quadratically with the size of the input data (represented by n).

> Imagine you have a box of items and want to compare each item with every other item to find specific pairs. As the number of items (n) increases, the number of comparisons (n^2) grows much faster.

- if 2 nested loop = O(n*n) = O(n^2)
- always remove nonDominant TC, eg: `3O(n^3) + 100O(n) + O(n^4) = (n^3) + O(n) + O(n^4) = O(n^4)`, because we always measure most bottleneck part !!!

```
// eg3: print pairs, [i,j] where i<j , no order matter
console.log("eg3");
const eg3arr = [0,1,2,3,4,5];
const eg3pairsPrint= (arr) => {
    for (let i =0; i<arr.length; i++){
        for (let j =i+1; j<arr.length; j++){
            console.log(` [ ${arr[i]}, ${arr[j]} ] `);

        }
        // O(n)
    }
    // O(n)

    // faltu, "NoContextJargonToProveAPoint"
    for (let i =0; i<arr.length; i++){ console.log(` [ ${arr[i]}, ${arr[i]} ] `);}
    // O(n)
};
eg3pairsPrint(eg3arr);
// nested loop

// eg3
//  [ 0, 1 ]
//  [ 0, 2 ]
//  [ 0, 3 ]
//  [ 0, 4 ]
//  [ 0, 5 ]
//  [ 1, 2 ]
//  [ 1, 3 ]
//  [ 1, 4 ]
//  [ 1, 5 ]
//  [ 2, 3 ]
//  [ 2, 4 ]
//  [ 2, 5 ]
//  [ 3, 4 ]
//  [ 3, 5 ]
//  [ 4, 5 ]
//  [ 0, 0 ]
//  [ 1, 1 ]
//  [ 2, 2 ]
//  [ 3, 3 ]
//  [ 4, 4 ]
//  [ 5, 5 ]

// O(n*n) + O(n) = O(n^2)

// If we combine all the "O" operations it becomes O(n^2 + n)
// O(n^2) is a Dominant term BOTTLENECK
// "n" is a Non-Dominant term
// So we remove the "non-dominant" term and we're only left with O(n^2)
// This way, we simplify our bigO

// O(n^2)
```

`O(log n)`

- O(log n) time complexity refers to an algorithm's runtime that grows logarithmically with the size of the input (represented by n).
- In simpler terms: as the input size increases, the time it takes for the algorithm to run increases slowly.
- eg : Divide and conquer

```
[1 2 3 4 | 5 6 7 8]         <- Step 1: Divide into two halves
[1 2 3 4]   [5 6 7 8]       <- Step 2: Work on each half separately
[1 2] [3 4]   [5 6] [7 8]   <- Step 3: Further divide
[1] [2] [3] [4] [5] [6] [7] [8] <- Step 4: Base case reached
```

```
(Then conquer: combine results)
[1 2] [3 4]   [5 6] [7 8]
[1 2 3 4]     [5 6 7 8]
[1 2 3 4 5 6 7 8]
```

> log2(8) = 3 = 3 steps of this algo

- in bs,trees etc., we will discuss more of it in depth

Arrays

Arrays DS

- A data structure array is an ordered collection of elements that can be accessed using a numerical index.
- primitive arrays

```
// eg4 : Array DS in JS
console.log('eg4')
const eg4stringArr = ["a","b","c"]; //string
const eg4numArr = [1,2,3,4]; //number
const eg4boolArr = [true,false,false,true]; //bool
const eg4mixedArr = ["a",2,true,undefined,null,{c:"c"},["d"]];
console.log(eg4stringArr,eg4numArr,eg4boolArr,eg4mixedArr);
// these are premitive arrays

// eg4
// [ 'a', 'b', 'c' ] [ 1, 2, 3, 4 ] [ true, false, false, true ] [ 'a', 2, true,
undefined, null, { c: 'c' }, [ 'd' ] ]
```

Custom Array

```


// eg5 : Custom Array
console.log('eg5');
class eg5CustomArray {
    constructor(){// initialize :)
        this.length = 0;
        this.data = {} ;
    }

    // custom operations
    // append
    push(element){
        // this.data+=element; // NO
        this.data[this.length]=element;
        this.length++;
        console.log(this.data);
    }

    // access
    get(index){
        // index
        return this.data[index];
    }

    // remove element from last
    pop(){
        const lastElement =this.get(this.length-1);
        delete this.data[this.length-1]; //IMP
        this.length--;
        return lastElement;
    }

    // remove element from first
    shift(){
        const firstElement = this.get(0);
        for ( let i=0; i<this.length && i+1 !== this.length; i++){
            this.data[i]=this.data[i+1];
        }
        this.pop();
        return firstElement;

    }

    // delete by index
    delete(index){
        const indexElement = this.data[index];
        for ( let i=index; i<this.length && i+1 !== this.length; i++){
            this.data[i]=this.data[i+1];
        }
        this.pop();
        return indexElement;
    }
}

const eg5MyNewArray = new eg5CustomArray();
console.log(eg5MyNewArray); //eg5CustomArray { length: 0, data: {} }
eg5MyNewArray.push(10);//{ '0': 10 }
eg5MyNewArray.push(200);//{ '0': 10, '1': 200 }
console.log(eg5MyNewArray.length);//2
console.log(eg5MyNewArray.get(1));//200
console.log(eg5MyNewArray.pop());//200
console.log(eg5MyNewArray); //eg5CustomArray { length: 1, data: { '0': 10 } }
eg5MyNewArray.push(20);//{ '0': 10, '1': 20 }
eg5MyNewArray.push(30);//{ '0': 10, '1': 20, '2': 30 }
eg5MyNewArray.push(40);//{ '0': 10, '1': 20, '2': 30, '3': 40 }
console.log(eg5MyNewArray);
// eg5CustomArray {
//   length: 4,
//   data: { '0': 10, '1': 20, '2': 30, '3': 40 }
// }
```

```
console.log(eg5MyNewArray.shift());//10
console.log(eg5MyNewArray);
// eg5CustomArray { length: 3, data: { '0': 20, '1': 30, '2': 40 } }
console.log(eg5MyNewArray.delete(1)      );//30
console.log(eg5MyNewArray);
// eg5CustomArray { length: 2, data: { '0': 20, '1': 40 } }

// eg5
// eg5CustomArray { length: 0, data: {} }
// { '0': 10 }
// { '0': 10, '1': 200 }
// 2
// 200
// 200
// eg5CustomArray { length: 1, data: { '0': 10 } }
// { '0': 10, '1': 20 }
// { '0': 10, '1': 20, '2': 30 }
// { '0': 10, '1': 20, '2': 30, '3': 40 }
// eg5CustomArray {
//   length: 4,
//   data: { '0': 10, '1': 20, '2': 30, '3': 40 }
// }
// 10
// eg5CustomArray { length: 3, data: { '0': 20, '1': 30, '2': 40 } }
// 30
// eg5CustomArray { length: 2, data: { '0': 20, '1': 40 } }
```

ASCII

- ASCII Table

```

Lowercase Letters (a-z):
a: 97
z: 122

Uppercase Letters (A-Z):
A: 65
Z: 90

Numbers (0-9):
0: 48
9: 57
```

- `String.fromCharCode(97)` method to find `String` <= ASCII no.
- `"a".charCodeAt(0)` method to find String => `ASCII no`.
- American Standard Code for Information Interchange. It is a character encoding standard used for representing text and control characters in computers and communication equipment.

Reverse String

- Hello => olleH
- Approach
  - convert string => array
  - reverse the array
  - convert array => string
- `eg6Array = eg6String.split('');` typecast str => `arr`
- `eg6Array.reverse();` reverse the original array
- `eg6String = eg6Array.join('');` typecast str`<= arr
- basically just this in short `eg6String.split("").reverse().join("")`

```
//eg6 : Reverse String
console.log('eg6');
//- Hello => olleH
//- convert string => array, reverse the array, convert array => string

const eg6ReverseString = (eg6String) => {
    let eg6Array = eg6String.split(''); // typecast str => `arr`
    eg6Array.reverse();
    return eg6Array.join(''); // typecast `str`<= arr
};

let eg6String = "Hello";//Hello
console.log(eg6String);
eg6String= eg6ReverseString(eg6String);
console.log(eg6String);//olleH

const eg6ReverseStringSmall = (eg6String) =>
eg6String.split("").reverse().join("");
eg6String= eg6ReverseStringSmall(eg6String);
console.log(eg6String);//Hello

eg6String= eg6String.split("").reverse().join("");
console.log(eg6String);//olleH

// eg6
// Hello
// olleH
// Hello
// olleH
```

Palindromes

- if the reverse string is equal to the original one then that word is palindrome
- eg: abba === abba [CORRECT]
- eg: cddc === cddc [CORRECT]
- eg: Hello !== olleH [INCORRECT]
- Approach
  - make stingCopy <= string

  - convert stingCopy => array
  - reverse the array
  - convert array => stingCopy
  - compare stingCopy === sting

```
//eg7 : Palindrome Checker
console.log("eg7")

const eg7PalindromeChecker = (str) => { return
str===str.split("").reverse().join("") };

// more short
const eg7PalindromeCheckerShort = str => str.split("").reverse().join("") === str
;

console.log(eg7PalindromeChecker("Hello"));//false
console.log(eg7PalindromeChecker("h"));//true
console.log(eg7PalindromeCheckerShort("cddc"));//true

// eg7
// false
// true
// true
```

Integer Reversal

- 1234 => 4321
- 5678 => 8765
- Approach
  - convert number => string
  - convert string => array
  - reverse the array
  - convert array => string
  - convert string => number
- typecast number => string by `String(num)` or `num.toString()` or `${num}`
- typecast number <= string by `parseInt(str)*Math.sign(str)`

```
//eg8 : Integer Reversal
console.log("eg8");
const eg8IntegerReverser = (Integer) =>
parseInt(Integer.toString().split("").reverse().join(""))*Math.sign(Integer);
console.log(eg8IntegerReverser(-1234));// -4321

// eg8
// -4321
```

Sentence Capitalization

- hello world => Hello World
- typical approach
    - convert string lowercase
    - convert string to array
    - capitalize each word
    - convert array to string
- ASCII Complex Approach
    - convert string to array
    - iterate each word
        - if any word's 1st letter is in range of a-z(97-112)
            - word = convert the first letter into Uppercase in range A-Z(65-90)+ rest of letters
                - convert firstLetter to ASCII number
                - -32 from it
                - convert this ASCII of A-Z range to String
                - add with rest of letters
    - convert array to string

```
// eg9:Sentence Capitalization
console.log("eg9");

//ASCII Complex way
const eg9SentenceCapitalizerASCII = (Sentence) =>{
    let eg9Array = Sentence.split(" ");
    for (let i =0; i<eg9Array.length;i++){ // never put any extraCondition at bw
because it will stop,not skip where extraCondition fails

        if (eg9Array[i][0].charCodeAt(0)>=97 && eg9Array[i][0].charCodeAt(0)<=122)
{

         eg9Array[i] = String.fromCharCode(eg9Array[i][0].charCodeAt(0) - 32) //
convert 1st letter
        +
        eg9Array[i].slice(1);// rest of stuff
        }

    }
    return eg9Array.join(" ");
}
console.log(eg9SentenceCapitalizerASCII("hello World i Am bALI"));// Hello World I
Am BALI

//typical approach
const eg9SentenceCapitalizer = (str) => str.toLowerCase().split("
").map((word)=>word[0].toUpperCase()+word.slice(1)).join(" ");
console.log(eg9SentenceCapitalizer("hello World i Am bALI"));//Hello World I Am
Bali
```

```
// eg9
// Hello World I Am BALI
// Hello World I Am Bali
```

FizzBuzz

1. Print numbers from 1 to `n`.
2. If the number is divisible by **3**, print `"Fizz"`.
3. If the number is divisible by **5**, print `"Buzz"`.
4. If the number is divisible by **both 3 and 5**, print `"FizzBuzz"`.
5. Otherwise, print the number.

```
// eg10: FizzBuzz
// new way of console log
console.log('eg10');
const eg10FizzBuzzNew = (n) => {for (let i =1; i<=n; i++) {
    console.log(
        i%3==0 ?
        (
            (i%5==0 ?
            ("FizzBuzz"):("Fizz")
        )
        )
        :
        (i%5==0 ?
            ("Buzz"):(i)
        )
    );
}};

eg10FizzBuzzNew(10);
// 1
// 2
// Fizz
// 4
// Buzz
// Fizz
// 7
// 8
// Fizz
// Buzz

//old normal way
const eg10FizzBuzzNormal = (n) => {
    for (let i =1; i<=n; i++)
    {
        if ( i%3==0 && i%5==0) console.log("FizzBuzz");
        else if ( i%3==0 ) console.log("Fizz");
        else if ( i%5==0 ) console.log("Buzz");
        else console.log(i);
    }
};

eg10FizzBuzzNormal(10);
// 1
// 2
// Fizz
// 4
// Buzz
// Fizz
// 7
// 8
// Fizz
// Buzz


// eg10
// 1
// 2
// Fizz
// 4
// Buzz
// Fizz
// 7
// 8
// Fizz
// Buzz
// 1
// 2
// Fizz
// 4
// Buzz
// Fizz
// 7
// 8
// Fizz
// Buzz
```

Max Profit

- Imagine you're buying and selling stocks throughout the year. Your job is to find the biggest profit you could make by buying low and selling high **only once**.
- Here's what you're given:
  - A list of stock prices for each day `[7, 1, 5, 3, 6, 4]`
- Here's what you need to find:
  - The difference between the cheapest price you could have bought the stock and the most expensive price you could have sold it later on.
- Approach
  - Input Array in func
  - Set `min_so_far = +∞ (or first element)`, `max_profit = 0`
  - iterate through Array elements , TodayPrice

        - if Difference of Min and TodayPrice exceeds max profit, make it new max profit
        - if Today Price < Min, make it new Min
    - return final max profit

```
// eg11: Max Profit
console.log("eg11");
// [7, 1, 5, 3, 6, 4]

const eg11MaxProfitCalOpt = (prices) => {
    let minPrice = prices[0]; // first day is cheapest day to buy for now
assumption at first place
    let maxProfit = 0;
    for (let i=1; i<prices.length; i++){
        const currentPrice = prices[i];
        minPrice = Math.min(minPrice,currentPrice); //update min price if the
lower price is found
        const potentialProfit = currentPrice - minPrice;
        maxProfit = Math.max(maxProfit, potentialProfit);
        console.log(maxProfit);
    }



};

const eg11MaxProfitCal = (array) => {
    let Min=Infinity, MaxProfit=0; //min_so_far = +∞ (or first element),
max_profit = 0
    for (let i =0; i<array.length; i++){
        let TodayPrice = array[i];

        if (TodayPrice-Min > MaxProfit){MaxProfit=TodayPrice-Min};// or MaxProfit
= Math.max(MaxProfit, TodayPrice-Min);
        if (TodayPrice<Min) {Min=TodayPrice;}// or Min = Math.min(Min,TodayPrice);


    }
    return MaxProfit;
};
// [7, 1, 5, 3, 6, 4]
console.log(eg11MaxProfitCal([3, 2, 6, 5, 0, 3])); //4
console.log(eg11MaxProfitCal([7, 1, 5, 3, 6, 4])); //5
console.log(eg11MaxProfitCalOpt([3, 2, 6, 5, 0, 3])); //4
console.log(eg11MaxProfitCalOpt([7, 1, 5, 3, 6, 4])); //5

// // Wrong Approach, Calculating just differences won't solve the problem, it
doesn't compare previous profit
// const eg11MaxProfitCal = (array) => {
//     let Min=Infinity, Max=0; //min_so_far = +∞ (or first element), max_profit =
0
//     for (let i =0; i<array.length; i++){
```

```
//          let TodayPrice = array[i];
//          if (TodayPrice<Min) {Min=TodayPrice;} //3 2 2 2 0 3
//          if (TodayPrice>Max) {Max=TodayPrice;} //3 0 6 6 0 3
//          if (Min===TodayPrice) {Max=0;}
//      }
//      return Max-Min;
// };
// console.log(eg11MaxProfitCal([3, 2, 6, 5, 0, 3])); //3

// eg11
// 4
// 5
// 4
// 5
```

Array Chunk

- Write a function that takes an array and a chunk size as input.
- The function should return a new array where the original array is split into chunks of the specified size.
- chunk([1, 2, 3, 4, 5, 6, 7, 8], 3) [[1, 2, 3], [4, 5, 6], [7, 8]]
- chunkArray([1, 2, 3, 4, 5], 2) // Output: [[1, 2], [3, 4]]
- approach1
  - Create an empty array to hold the chunks
  - Set a starting index to keep track of where we are in the original array
  - Loop through the original array as long as the index hasn't reached the end
  - Extract a chunk of the desired size from the original array
  - Add the extracted chunk to the `chunked` array
  - Move the index forward by the chunk size to get to the next chunk
  - Return the final array of chunks
- approach2
  - create empty chunkArray, final return
  - create empty subChunk , subset of chunkArray
  - create chunkCount=0, it's a counter of subChunk
  - create lastElement
  - iterate through array
    - each element push in subChunk
    - chunkCount increases
    - if chunkCount === chunk or element === lastElement
      - push subChunk into chunkArray
      - re-set subChunk into empty array
      - re-set chunkChunk into 0
  - basically, we made a sloting method
  - you can make it more modular
  - return final chunkArray

```
// eg12: Array Chunk
console.log("eg12");
```

```
const eg12ArrayChunker1 = (array,chunkSize) => {
    let index=0;
    let chunkArray=[];
    while (index<array.length){
        const chunk = array.slice(index,index+chunkSize);
        chunkArray.push(chunk);
        index+=chunkSize;
    }
    return chunkArray;
};

const eg12ArrayChunker2 = (array,chunk) => {

    let chunkArray = [];
    let subChunk = [];
    let chunkCount = 0;

    const allotChunk= (chunkArray,subChunk) => {chunkArray.push(subChunk);}

    const reset = () => {subChunk=[]; chunkCount=0};

    let lastElement = array[array.length-1];

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

        let element = array[i];

        subChunk.push(element);
        if (chunkCount===chunk || element===lastElement){
            allotChunk(chunkArray,subChunk);
            reset();

        }


    }
    return chunkArray;
};

console.log(eg12ArrayChunker1([1, 2, 3, 4, 5, 6, 7, 8], 3)); //[ [ 1, 2, 3 ], [ 4,
5, 6 ], [ 7, 8 ] ]
console.log(eg12ArrayChunker1([1, 2, 3, 4, 5], 2));//[ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]
console.log(eg12ArrayChunker2([1, 2, 3, 4, 5, 6, 7, 8], 3)); //[ [ 1, 2, 3 ], [ 4,
5, 6 ], [ 7, 8 ] ]
console.log(eg12ArrayChunker2([1, 2, 3, 4, 5], 2));//[ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]

// eg12
// [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8 ] ]
// [ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]
// [ [ 1, 2, 3 ], [ 4, 5, 6 ], [ 7, 8 ] ]
// [ [ 1, 2 ], [ 3, 4 ], [ 5 ] ]
```

Two Sum (Ugly Code)

- Imagine you have a list of numbers and a target number. Your job is to find two numbers in that list that add up to the target number.
- You also need to tell which positions (or indexes) those two numbers are at in the list.
- Example
  - if the list is [2, 7, 11, 15] and the target is 9, then the answer would be [0, 1] because 2 (at index 0) plus 7 (at index 1) equals 9.

```
// eg13: Two Sum ( Ugly Code )
// this is not better solution, there exists a better approach
console.log("eg13");

const eg13TwoSumUgly = (array, target) => {
    for( let i=0; i<array.length; i++){
        for( let j=0; j<array.length; j++){
            if (i!==j && (array[i]+array[j]===target)){
                return [i,j];
            }
        }
    }
};
const eg13TwoSumUgly2 = (array, target) => {
    for( let i=0; i<array.length; i++){
        for( let j=i+1; j<array.length; j++){
            if (array[i]+array[j]===target){
                return [i,j];
            }
        }
    }
    return []; //return empty if not exists
};
const eg13TwoSumLessUgly= (array, target) => {
    for( let i=0; i<array.length; i++){
        if ( array.includes(target-array[i]) ) { // array.includes()
            return [i,array.indexOf(target-array[i])];// array.indexOf()
        }
    }
};

console.log(eg13TwoSumUgly([2, 7, 11, 15],9));//[ 0, 1 ]
console.log(eg13TwoSumUgly2([2, 7, 11, 15],9));//[ 0, 1 ]
console.log(eg13TwoSumLessUgly([2, 7, 11, 15],9));//[ 0, 1 ]

// eg13
// [ 0, 1 ]
// [ 0, 1 ]
// [ 0, 1 ]
```

Linked List

- A linked list is a linear data structure where elements, called nodes, are not stored contiguously in memory. Instead, each node contains data and a reference, or link, to the next node in the sequence.
- A node is a basic building block of a linked list.
- node is object that contain 2 things
  - data(/value)
  - reference to address to another node in the sequence.

```
NODE

+-----------+
|   Data    |
|-----------|
| Reference | ->
+-----------+
```

Inside Linked List

```
LINKED LIST

+------+     +------+     +------+     +------+
| Data | --> | Data | --> | Data | --> | Data | --> NULL
| Ref  |     | Ref  |     | Ref  |     | Ref  |
+------+     +------+     +------+     +------+
  (Head)                                 (Tail)
```

- Head= first node
- Tail= last node, refering to null
- this example above is also singly linked list

Constructing Linked List

> Constructing each method gets easier if you under stand previous methods well !!!

> Don't Dare to Lazy shortening of Code when managing nodes(even in of if/else blocks), else it will leads to circular references, dangling pointers, and debugging nightmares.

> in JS, everything is object

> Don't try to play loop till index-1, as it will fail on index=0, if playing then make seperate case on above the loop

> If using counter & while loop, don't forget to inc./dec./change the counter inside while loop, sometimes forget b/w thinking various complex stuff.

1. Setup Node & Linked List

- Define a Node class that holds a value and a pointer to the next node (initially null).
- Define a LinkedList class that creates a new linked list with one node.
- The first node is assigned as both the head and the tail.
- The list length is set to 1.
- Use it& Enjoy

```
// 1. Setup Linked List

// First Node
class eg14Node{
    constructor(value){
        this.head=value;
        this.next=null;
    }
}

// First LL
//       Head
//        |
//        V
// +-------------------+
// |   Value:    1     |
// |   Next:   NULL    |
// +-------------------+
//        ^
//        |
//       Tail

class eg14LinkedList{
    constructor(value){
        this.head=new eg14Node(value);
        this.tail=this.head;
        this.length=1;
    }
}

// useage
const eg14MyNewLinkedList = new eg14LinkedList(1);
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node { head: 1, next: null },
//   tail: eg14Node { head: 1, next: null },
//   length: 1
// }
```

2. Push Method

- helps to add node at end of LL.

```
Before Push:
Head -> [1 | *] -> [2 | *] -> [3 | NULL]  Tail
```

- Approach
  - make new node
  - remove the next of last node of LL = null
  - set next of last node of LL = this new node
  - set tail = this new node
  - set next of this new node = null
- edge case
  - at case of null linkedlist, where head and tail refer to null
    - attach head to new node
    - attach tail to new node

```
After Push:
Head -> [1 | *] -> [2 | *] -> [3 | *] -> [4 | NULL]  Tail
```

```
// 2. Push Method
push(value){
    let newNode = new eg14Node(value);
    if(!this.head){
        this.head= newNode;
        this.tail= newNode;
        this.length++;

    }
    else{
        this.tail.next = newNode;
        this.tail = newNode;
        this.length++;
    }
    // In the empty list case, tail and newNode are the same. Setting tail.next = newNode creates a loop.
    // By adding the else, we only set tail.next when there’s already a node, preventing the [Circular *1] issue.

}
```

```
// useage
const eg14MyNewLinkedList = new eg14LinkedList(1);
console.log(eg14MyNewLinkedList);
```

```
// eg14LinkedList {
//   head: eg14Node { head: 1, next: null },
//   tail: eg14Node { head: 1, next: null },
//   length: 1
// }
eg14MyNewLinkedList.push(2);
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node { head: 1, next: eg14Node { head: 2, next: null } },
//   tail: eg14Node { head: 2, next: null },
//   length: 2
// }
```

3. Pop Method

- Analysis
  - attach tail to previous node
    - not simple
    - need to iterate to
    - find last element
    - find 2nd last element
    - attach the tail to 2nd last element
  - previous node's next refer to null
- Approach
  - make 2 vars, temp and var point to 1st element
  - then temp check if element is last(if element's next is null) , if not so it shift to next element
  - prev stays at 1st time
  - then when temp check 2nd element and if not last ,then it will shift to next element and also make prev shift too
  - then when temp reach to last element and that check pass, then we will attach tail to where prev attaches to (obv 2nd last element)
  - then next of the element refer to null
- edge case
  - if given linkedlist is empty before pop , return null/undefined
  - if linkedlist is empty after pop, set head and tail to null

```
// 3. Pop Method
pop(){
    if(!this.head){
        return undefined;
    }

    let temp = this.head;
    let prev=this.head;
    while(temp.next){ //not equals to null, it exists
        prev=temp;
        temp=prev.next;
    }
    this.tail = prev;
    this.tail.next=null;
    this.length--;

    if(this.length===0){
        this.head=null;
        this.tail=null;
    }
    return temp;
}
```

```
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node { head: 1, next: eg14Node { head: 2, next: null } },
//   tail: eg14Node { head: 2, next: null },
//   length: 2
// }
eg14MyNewLinkedList.pop();
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node { head: 1, next: null },
//   tail: eg14Node { head: 1, next: null },
//   length: 1
// }
eg14MyNewLinkedList.pop();
console.log(eg14MyNewLinkedList);
// eg14LinkedList { head: null, tail: null, length: 0 }
console.log(eg14MyNewLinkedList.pop());
// undefined
console.log(eg14MyNewLinkedList);
// eg14LinkedList { head: null, tail: null, length: 0 }
```

```
Before pops:
head → [1] → [2] → [3] → null
                      ↑
                     tail

After 1st pop:
head → [1] → [2] → null
                ↑
               tail

After 2nd pop:
head → [1] → null
         ↑
        tail

After 3rd pop:
```

```
head → null
tail → null
```

```
Start:
head → [1] → [2] → [3] → null
                    ↑
                    tail
prev → [1]
temp → [1]

Loop 1:
prev → [1]
temp → [2]

Loop 2:
prev → [2]
temp → [3]

Loop ends (temp.next = null):
prev → [2]
temp → [3]

After re-link:
tail = prev → [2]
tail.next = null

Final list:
head → [1] → [2] → null
              ↑
              tail
(returned [3])
```

4. Unshift Method

- it helps to add node at the beginning of LL
- Approach
    - make new node's next point to fist element
    - make head point to new node
- edge case
    - if LL is empty
        - attach head to new node
        - attach tail to new node

```
Before:
head → [A] → [B] → ... → tail
```

```
unshift(X)

After:
head → [X] → [A] → [B] → ... → tail
```

```
// 4. Unshift Method, Best Way
unshift(value){
    let newNode = new eg14Node(value);
    if (this.length===0){ // or if(!this.head)
        this.head=newNode;
        this.tail=newNode;
        // this.length++;
    }
    else{
        newNode.next=this.head;
        this.head=newNode;
        // this.length++;
    }

    // i wrote 2 lines below as they both are same in both the if and else block, 
so why not reduce the code :)
    this.length++;
    // this.head=newNode; // NO, Moving this.head = newNode and this.length++ 
outside the if/else seems DRY(Don’t Repeat Yourself), but in the empty-list case 
it can leave .next uninitialized or mislinked, risking a self-referencing node; 
keeping assignments inside the blocks ensures head and tail are set safely.
}
```

```
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node { head: 10, next: eg14Node { head: 20, next: [eg14Node] } },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }

// What the heck it this: next: [eg14Node]
// That next: [eg14Node] you’re seeing is not an error
// by using console.log(VAR_NAME)
// it’s just how Node.js (or Chrome DevTools) abbreviates deeply nested objects 
when printing them with console.log
// [eg14Node] means : "There’s another eg14Node object here, but I’m not expanding 
it."
// Node.js tries to keep console output short.
// console.dir(VAR_NAME, { depth: null }); forces Node.js to expand all nested 
objects
```

```
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }

eg14MyNewLinkedList.unshift(10000);
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10000,
//     next: eg14Node {
//       head: 10,
//       next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//     }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 4
// }
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node {
//     head: 10000,
//     next: eg14Node { head: 10, next: [eg14Node] }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 4
// }
eg14MyNewLinkedList.unshift(50000);
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 50000,
//     next: eg14Node {
//       head: 10000,
//       next: eg14Node {
//         head: 10,
//         next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//       }
//     }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 5
// }
console.log(eg14MyNewLinkedList);
// eg14LinkedList {
//   head: eg14Node {
//     head: 50000,
//     next: eg14Node { head: 10000, next: [eg14Node] }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 5
// }
eg14MyNewLinkedList.pop();
eg14MyNewLinkedList.pop();
eg14MyNewLinkedList.pop();
eg14MyNewLinkedList.pop();
eg14MyNewLinkedList.pop();
console.log(eg14MyNewLinkedList);
// eg14LinkedList { head: null, tail: null, length: 0 }
eg14MyNewLinkedList.unshift(200000);
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node { head: 200000, next: null },
//   tail: eg14Node { head: 200000, next: null },
//   length: 1
// }
```

```
Case 1: Empty list

Before:
head → null
tail → null

unshift(200000)

After:
head → [200000] → null
tail → [200000]
```

```
Case 2: Non-empty list

Before:
head → [10] → [20] → [30] → null
tail → [30]

unshift(10000)

Step:
newNode → [10000]
newNode.next = head

After:
head → [10000] → [10] → [20] → [30] → null
tail → [30]
```

5. Shift Method

- it just's remove 1st element of LL
- Approach
  - let temp var
  - set temp attach 1st element(through head)
  - set head attach 1st element's next(i.e. 2nd element)
  - set 1st element(through temp) attach to null
- edge cases
  - if null LL
    - return undefined
  - if 1 element
    - set head to null
      - this would be done automatic , as at attaching temp.next ;head is actually attaching to null
    - set tail to null

```
// 5. Shift Method
shift(){
    if (!this.head){return undefined;}
    let temp = this.head;
    this.head = temp.next; // or this.head.next
    temp.next = null;
    this.length--;
    if(this.length===0){this.tail=null;
        // this.head=null; // this would be done automatic , as at attaching 
temp.next ;head is actually attaching to null
    }
    return temp;
}
```

```
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node { head: 200000, next: eg14Node { head: 10, next: null } },
//   tail: eg14Node { head: 10, next: null },
//   length: 2
// }
eg14MyNewLinkedList.shift();
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node { head: 10, next: null },
//   tail: eg14Node { head: 10, next: null },
//   length: 1
// }
eg14MyNewLinkedList.shift();
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList { head: null, tail: null, length: 0 }
```

```
Case 1: More than 1 element

Before:
 head → A → B → ... → null
 tail → last
 length = n

Step 1: temp = head (A)
Step 2: head = temp.next (B)
Step 3: temp.next = null (A detached)
Step 4: length--
Step 5: tail unchanged

After:
 head → B → ... → null
 tail → last
 length = n-1
```

```
Case 2: Only 1 element

Before:
 head → A → null
 tail → A
 length = 1

Step 1: temp = head (A)
Step 2: head = temp.next → null
Step 3: temp.next = null (A detached)
Step 4: length-- → 0
Step 5: tail = null

After:
 head → null
 tail → null
 length = 0
```

```
Case 3: Empty list

Before:
 head → null
 tail → null
 length = 0

Step 1: if !head → return undefined

After:
 head → null
```

```
  tail → null
  length = 0
```

6. Get First Method

- it just return 1st element of LL
- most easiest

```
// 6. Get First Method
getFirst(){return this.head;}
```

```
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList { head: null, tail: null, length: 0 }
console.log(eg14MyNewLinkedList.getFirst());
// null

// just filling LL
eg14MyNewLinkedList.push(10);
console.log(eg14MyNewLinkedList.getFirst());
// eg14Node { head: 10, next: null }

eg14MyNewLinkedList.push(20);
console.log(eg14MyNewLinkedList.getFirst());
// eg14Node { head: 10, next: eg14Node { head: 20, next: null } }

eg14MyNewLinkedList.push(110);
console.log(eg14MyNewLinkedList.getFirst());
// eg14Node {
//   head: 10,
//   next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
// }
```

7. Get Last Method

- Return Last Element
- Approach 1
  - just extract what tail's refer to, Power of JavaScript OOPs
- Approach 2
  - while loop to check if element's next is null, thus last element

```
// 7. Get Last Method
getLast(){return this.tail;}
getLast2(){
    if(!this.head) return null;
    let temp = this.head;
    while(temp){
        if(!temp.next){
            return temp
        }
        temp=temp.next;
    }
}
```

```
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }
console.log(eg14MyNewLinkedList.getLast());
// eg14Node { head: 110, next: null }
console.log(eg14MyNewLinkedList.getLast2());
// eg14Node { head: 110, next: null }
```

```
Linked List: Head → 10 → 20 → 110 → null

  Approach 1 (tail ref)        Approach 2 (while loop)
+-------------------+         +------------------------+
| getLast()         |         | getLast2()             |
| returns tail ↓    |         | traverse from head ↓ |
| 110 → null        |         | 10 → 20 → 110 → null |
+-------------------+         +------------------------+
```

8. Get Method. Get Element By Index

- return element of that index
- Approach
  - if index is out of limit, return undefined
  - else, if index is valid
    - let temp refference to 1st element(this.head)
    - counter=0;
    - for/while loop till counter = index
      - temp jump forward
      - counter++

```
// 8. Get Method. Get Element By Index
get(index){
    if (index >= this.length) return undefined;
    else {
        let temp = this.head;
        for(let i=0; i<index; i++){
            temp=temp.next;
        }
        return temp;
    }

}
```

```
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }
console.log(eg14MyNewLinkedList.get(0));
// eg14Node {
//   head: 10,
//   next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
// }
console.log(eg14MyNewLinkedList.get(1));
// eg14Node { head: 110, next: null }

console.log(eg14MyNewLinkedList.get(2));
// eg14Node { head: 110, next: null }

console.log(eg14MyNewLinkedList.get(3));
// undefined
```

```
Linked List: Head → 10 → 20 → 110 → null

Index Access:

 get(0) → 10 → 20 → 110 → null
 get(1) → 20 → 110 → null
 get(2) → 110 → null
 get(3) → undefined
```

9. Set Method

- if given
  - just index, return the node at that index at ll

    - both index& new value, first replace the value of that node at that index at ll to the new value at arguement, then return the new node
- Approach
    - this approach is independent
    - index is given as parameter, so
    - if index out of limit, return undefined
    - if index is 0
        - make temp
        - set temp refer to 1st element(this.head)
        - if value given as parameter
            - make new node with value
            - set head of ll = new node
            - set new node's next = temp's next
            - set temp's next = null, flush it
            - return new node
        - if value doesn't given as parameter
            - return temp
    - else, index is valid
        - make temp, prev attach to 1st element of ll
        - then traverse the loop till just index-1,
        - now temp and prev attach to element at index-1
        - so, just jump temp to next element 1 time
        - if value doesn't given as parameter
            - return temp, as we are expected to not modify
        - if value is given as parameter
            - then make a new node(value)
            - then attach previous node's next to new node
            - then attach new node's next to temp's next
            - then attach temp's next to null, i.e. flush that node of index, replace it with new node
            - if index is last index
                - then set tail = new node
            - return the new node
- Wrong Approach
    - this approach is dependent to get() method
    - make temp var attach to that node using get(index)
    - if temp exist valid
        - temp's node's value = new value
        - return true
    - else
        - return false
    - problem is that here we are not thinking about the head,tail,prev's next, OOPS

```
// 9. Set Method
set(index,value){
    if (index >= this.length) return undefined;
    if (index===0){
        let temp=this.head;
        if(value){
            let newNode = new eg14Node(value);
            this.head=newNode;
            newNode.next = temp.next;
            temp.next=null
            return newNode;
        }
        else{
            return temp;
        }


    }
    else {
        let temp = this.head;
        let prev = this.head;
        for(let i=0; i<index-1; i++){// we did to make prev lagging behind temp by
1 time
            temp=temp.next;
            prev=prev.next;
        }
        temp=temp.next;// now temp is just after prev
        if(value){
            let newNode = new eg14Node(value);
            prev.next=newNode;
            newNode.next=temp.next;
            temp.next=null;
            if(newNode.next===null){// if element is last element
                this.tail= newNode
            }
            return newNode;
        }else{
            return temp;
        }

    }

}
    // Wrong approach
    // set1(index,value){
    //     let temp = this.get(index);
    //     if (temp){temp.value=value;return true;}
    //     else {return false;}
    // }
```

```


console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
```

```
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 20, next: eg14Node { head: 110, next: null } }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }

console.log(eg14MyNewLinkedList.set(1));
// eg14Node { head: 20, next: eg14Node { head: 110, next: null } }

console.log(eg14MyNewLinkedList.set(1,100));
// eg14Node { head: 100, next: eg14Node { head: 110, next: null } }

console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 100, next: eg14Node { head: 110, next: null } }
//   },
//   tail: eg14Node { head: 110, next: null },
//   length: 3
// }

console.log(eg14MyNewLinkedList.set(2,10000));
// eg14Node { head: 10000, next: null }

console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 10,
//     next: eg14Node { head: 100, next: eg14Node { head: 10000, next: null } }
//   },
//   tail: eg14Node { head: 10000, next: null },
//   length: 3
// }


console.log(eg14MyNewLinkedList.set(0,1));
// eg14Node {
//   head: 1,
//   next: eg14Node { head: 100, next: eg14Node { head: 10000, next: null } }
// }
console.dir(eg14MyNewLinkedList, { depth: null });
// eg14LinkedList {
//   head: eg14Node {
//     head: 1,
//     next: eg14Node { head: 100, next: eg14Node { head: 10000, next: null } }
//   },
//   tail: eg14Node { head: 10000, next: null },
//   length: 3
// }
```

10. Insert Method

- allow to insert node to anywhere in ll
- now it's open ended, to insert after the index or before the index

10.1. InsertFront Method

-

10.2. InsertBack Method