they said Recursion is tough!

classmate
Date
Page

Recursion Theory

describing an action in term of itself.

Why → task that are composed of similar subtask.

Shorter code (may be easier)

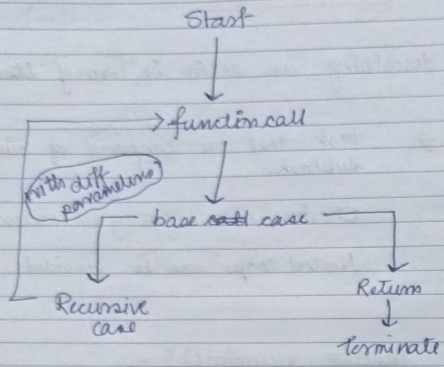
Nested loops can be avoided.

eg:

```
function recursion(){
  // base case
  if (anybasecase()){
    return
  }

  // recursive case.
  recursion()
}
```

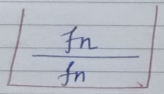
→



Imp

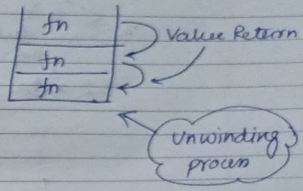
→ Memory in Recursion

- A Recursive function call itself
  - the memory for a called function is allocated on the top of memory allocated for called function.



- Each function call gets a different copy of local variables.
- When base case is reached, child function returns value to the function from which it was called and then the process continues.

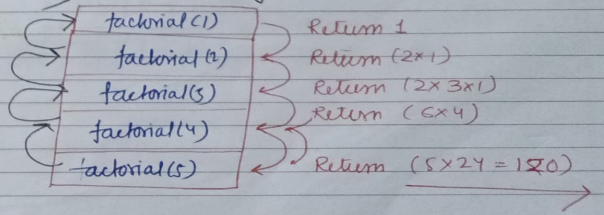
Unwinding

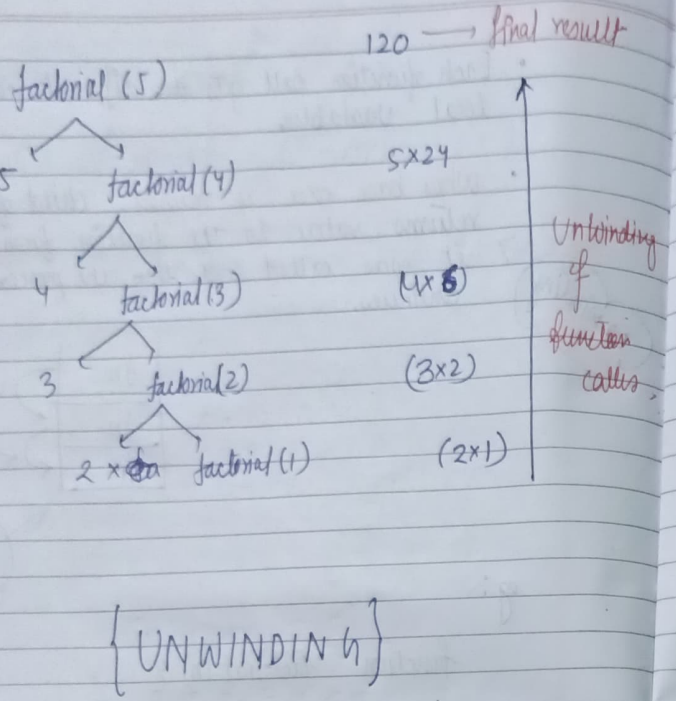


eg:

```
function factorial (n) {
    if (n === 0) {
        return 1
    }
    return n * factorial (n-1)
}
console.log (factorial (5))
```

base case





{UNWINDING}

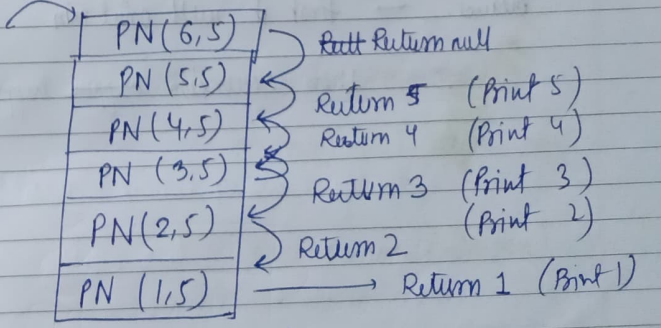
→ Direct and Indirect recursion →

- Direct Recursion
  - When function call itself

eg:

```
function PrintNumber(min, max) {
    if (min > max) {
        return null
    }
    PrintNumber (min+1, max)
    console.log (min)
}
```

PrintNumber (1,5) → 5, 4, 3, 2, 1



↳ 5, 4, 3, 2, 1

another —

```
function PrintNumber(min, max) {
    if (min > max) {
        return null
    }

    console.log(min)
    PrintNumber(min+1, max)
}

PrintNumber(1, 5)
```

↳ 1, 2, 3, 4, 5

because → first it will print min

Then recursion ~~happen~~ happen

→ Indirect Recursion

↳ logic extracted

↳ logic calling printNumber again and again

↓

```
function PrintNumber(min, max) {
    if (min > max) {
        return ~~null~~
    }
    console.log(min)
    min = min + 1
    logic(min, max)
}

P. function logic(min, max) {
    if (min > max) return null

    PrintNumber(min, max)
}

PrintN(1,5)
```

Problem statement →

Q. Reverse a string with recursive

→ // Iterative approach

// Hello.

```
function reverseString (str) {
    let reverse = "";
    let length = ~~length - 1~~ string.length - 1
    while (length >= 0) {
        reverse += reverse + string [length]
        length = length - 1
    }
    return reverse
}
console.log (reverseString ("Hello"))
                └→ olleH.
```

// Recursive method →

```
function recursiveReverseString (string) {
    console.log (' current string : ${~~str~~string}')
    // base case
    if (string === "") {
        return string
    }
    let reverseString = recursiveReverseString
        (string.substring [0, string.length - 1])
    let result = string [string.length - 1] + reverseString.
    console.log (' Reversing last char: ${string [str.length-1]}
        added in ~~front~~ of ${reverseString}
        to
        form a result : ${result}')
    return result
}
```

...t Vowels in string →

```
...unction  isVowel (character) {
    let  lowerChar = character.toLowerCase()
    let  vowels = "aeiou"

    vowels.~~char~~
      vowels.indexOf (lowerchar)
    // vowels.indexOf (lowerchar)        → true
    if (vowels.indexOf (lowerchar) !== -1) {
            return true
    } else {
      return false
    }
}

function  vowelsCount (string) {
    let  count = 0
    for (let i = 0  i < string.length, i++) {
      count if (isVowel (string [i])) ) {
            count += 1
      }
    }
    return count
    console.
}
console.log (countVowels ("Animesh")) → 3
```

→ Recursive method →

```
function  recursiveCountVowels (string, stringLength) {

    if (stringLength == 1) {

        return Number (isVowel (string[0]))
    }

    let ~~reto~~ result = recursiveCountVowel (string,
                                  stringLength - 1)
        + isVowel (string [stringLength - 1])
```

```
return result
}
```

⇒ Rewrite

```
function isVowel (char) {
    let lowerchar = char.toLowerCase()
    let vowels = "aeiou"

    if (vowels.indexOf(lowerchar)) {
        return true
    } else {
        return false
    }
}
```

```
function recursiveCountVowel (String, StringLength) {
    if (stringLength == 1) {
        return Number (isVowel (string[0]))
    }
```

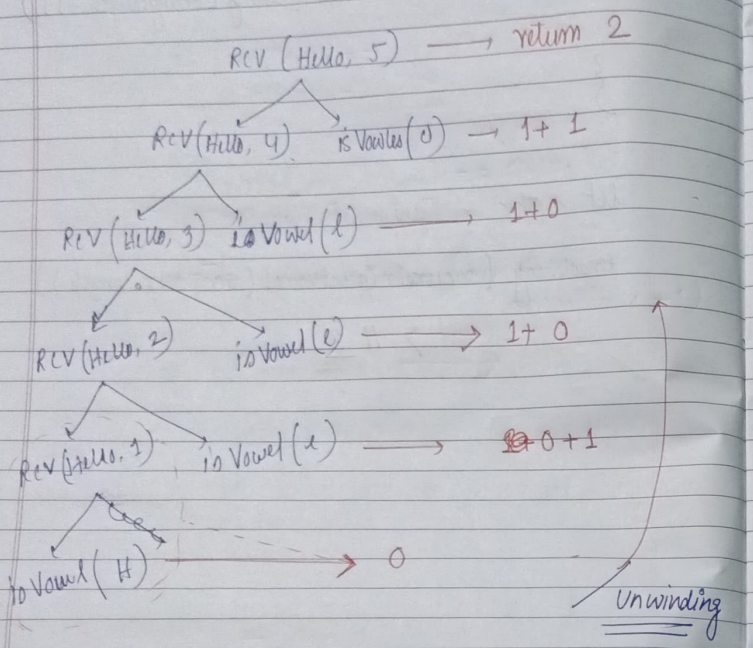
```
return recursiveCountVowel (string, stringLength - 1)
        + isVowel (string [stringLength - 1])
}

let str = "animesh"
console.log (recursiveCountVowel (str, str.length))
```

↳ // 3

Visualize Count Vowels

Recursive Count Vowels – RCV.



⇒ first occurance →

```
// arr = [3, 2, 1, 8, 1, 7]
// findMe = 1
// curr Index = 0
// output = 2
// if not found = -1.

function firstOccurance (arr, findMe, currentIndex) {
    while (currentIndex < arr.length) {
        if (arr[currentIndex] == findMe) {
            return currentIndex
        }
        currentIndex += 1
    }
    return -1
}
```

go out of if

loop stop

Iterative Approach

⇒ Recursive

```
function recursiveFirstOcc(arr, FindMe, currentIndex) {
    if (arr.length == currentIndex) {
        return -1
    }
    if (arr[currentIndex] == FindMe) {
        return 1
    }
    return recursiveFirstOcc(arr, FindMe,
                             currentIndex + 1)
}
```

⇒ Fibonacci

0, 1
0, 1, 1 (0+1 = 2nd digit)
0, 1, 1, 2
0, 1, 1, 2, 3
0, 1, 1, 2, 3, (5) → fib(5)
0, 1, 1, 2, 3, 5, 8

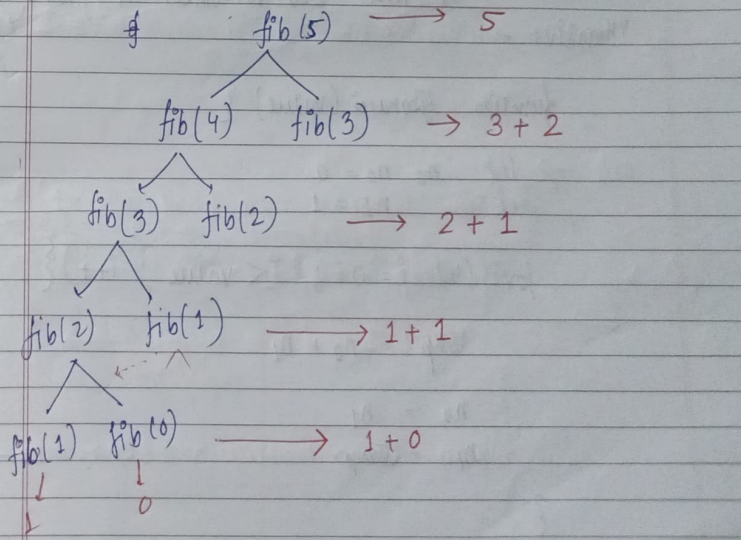
here $n_0 = 0$, $n_1 = 1$

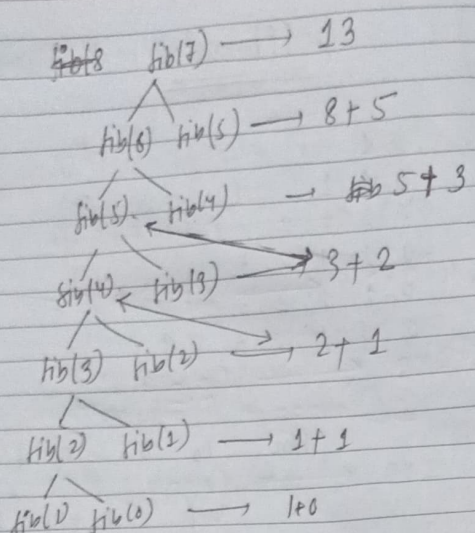
temp = $n_0 + n_1 = 1$

$n_0 = n_1$ and $n_1 = temp$

so

$F_n = F_{n-1} + F_{n-2}$





code →

// find fibo ~~ro~~number at given index (Value)

Iterative -

```
function fibonacci (value) {

    let n0 = 0
    let n1 = 1

    for (let i = 0 ; i < value ; i++) {

        temp = n0 + n1

        n0 = n1
        n1 = temp

    }
```

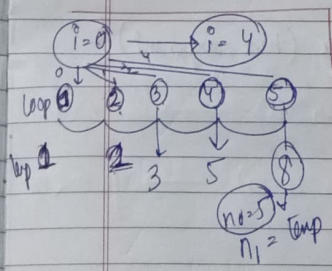
Value <= 1 → 1
→ 0

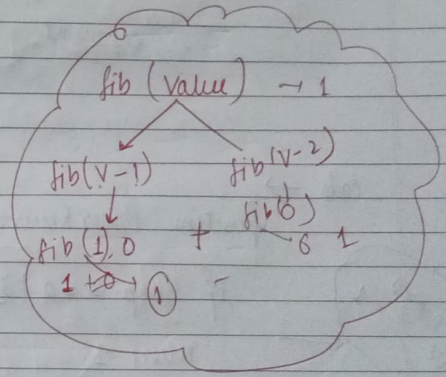
```
function recursive fibonacci (value) {

    if (value <= 1) {
        return value
    }

    return ( recursive fibo (value - 1)
           + recursive fibo (value - 2) )

}
```





⇒ Power of Number

$2^4 = ?$

P(2,4) ⟶ 16

2 × P(2,3) ⟶ 2 × 2 × 2 × 2 = 2 × 8

2 × P(2,2) ⟶ 2 × 2 × 2 = 2 × 4

2 × P(2,1) ⟶ 2 × 2

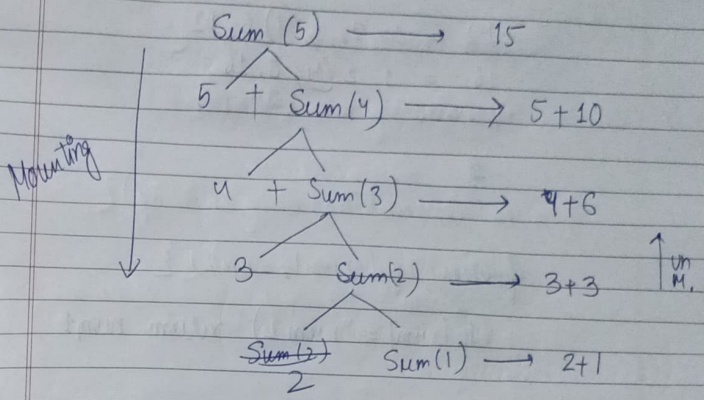
2 × P(2,0) ⟶ 2 × 1

exponent = 0 ⟶ 1 $\quad n^0 = 1$

code →

```
function PowerRecursive (base, exponent) {
    if (exp == 0) {
        return 1
    }
    return base x PowerRecursive (base, expo - 1)
}
```

⇒ Sum of N number (1 to N)



code →

```
function getSum (value) {
    if (value == 1) {
        return value
    }
    return value + getSum (value - 1)
}
```

highest common

⇒ Greatest common division

12 = 1, 2, 3, 4, 6, 12
16 = 1, 2, 4, 8, 16

4

function gcd (num1, num2) {
  if (num1 = num2) return num1
  if (num1 > num2) {
    return gcd (num1 - num2, num2)
  } else {
    return gcd (num1, num2 - num1)
  }
}

$$gcd = \begin{cases} m \longrightarrow m = n \\ gcd(m-n, n) \longrightarrow \text{if } m > n \\ gcd(m, n-m) \longrightarrow \text{if } m < n \end{cases}$$

12, 16

gcd (12, 16)
12 < 16
gcd (12, 4)
gcd (8, 4)
gcd (4, 4) ⟶ 4 = 4 ⟶ (4)

another → 42, 56

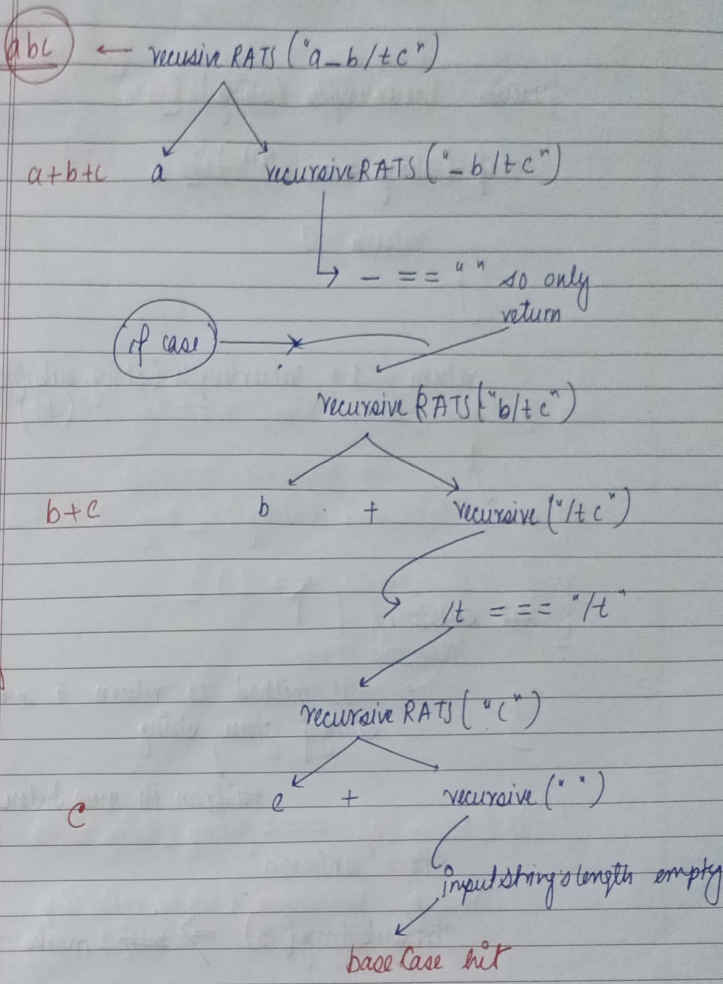
gcd (42, 56)
gcd (42, 14)
gcd (28, 14)
gcd (14, 14) ⟶ (14)

Q Remove all tabs and spaces from a string. Includes character \t and " " from

Sol" → Recursive –

```
function recursiveRTAS (string inputstring) {
    if (inputString length === 0) {
        return ""
    }
    let firstchar = inputstring[0]
    let restString = inputstring.slice(1)
    if (firstchar === " " || firstchar === "\t") {
        return recursiveRATS (restString)
    }
    return firstchar + recursiveRATS (restString)
}
```

recursive RATS ("a_b/tc")



unwinding

// ~~the~~ dot length implement

```
function recurLength (string) {
    if (string === "") {
        return 0
    }
    return 1 + recurLength (string.substring (1))
}
```

[~~st~~ substring ( )

↳ It method to return a new string of given string from its new index

Str = 'animesh'

Str.substring (3) ⇒ ~~mesh~~ mesh

System — substring (startIndex, lastIndex)

→ if (2) → lastIndex ↳ last of original string

~~recurse ("animesh")~~

(3) recurLength ("abc")

1+1+1 ← 1 + recurLength ("bc")

1+1 ← 1 + RL ("c")

1 + RL ("")

↳ 0

base case

Unwinding

• ~~string~~ last index

substring (firstIndex, lastIndex)

↳ result ⇒ firstIndex but not the last index.

'animesh'.substring (3, 6)

↳ mes

// check Palindrome

Reads same from backwards

madam ↔ madam

code →

```
function Palindrome (string) {
  if (string.length <= 1) {
    return true
  }
  if (String[0] === String[String.length - 1]) {
    return Palindrome (string.substring
                       (1, string.length - 1)
  } else {
    return false
  }
}
```

// check key occurance in array

[1, 2, 4, 2, 2, 4, 2, 6]

key = 2 (how many times 2 occure)

// iterative

```
function countkeyIA (arr, key) {
  const count = 0
  for (let i = 0; i < arr.length ; i++) {
    if (array[i] = key) {
      count++
    }
  }
  return count
}
```

```
function recursiveCountKey (arr, key, index = 0) {

    if (index >= arr.length) {
        return 0
    }

    return (array[index] === key ? 1 : 0)
        + recursiveCountKey
            (arr, key, index + 1)
```

// Balance Paranthesses problem

( ) { } [ ]

```
function balanceParantheses (arr, startIndex = 0, currentIndex = 0) {

    if (startIndex == arr.length) {
        return currentIndex = 0
    }

    if (currentIndex < 0) {
        return false
    }

    if (arr [startIndex] = "(") {
        balanceParantheses (arr, startIndex + 1,
                            currentIndex + 1)
    }
```