Agenda :

1. **Master Theorem**
2. **JavaScript Rescursion**

We have learned about the Algorithms and Data Structure basics in the last session, now in this session we are going to learn about Master Theore and Recursion in Javascript.

Master Theorem

The master method is a formula for solving recurrence relations of the form:

```
T(n) = aT(n/b) + f(n),
where,
n = size of input
a = number of subproblems in the recursion
n/b = size of each subproblem. All subproblems are assumed
     to have the same size.
f(n) = cost of the work done outside the recursive call,
      which includes the cost of dividing the problem and
      cost of merging the solutions

Here, a ≥ 1 and b > 1 are constants, and f(n) is an asymptotically positive function.
```

An asymptotically positive function means that for a sufficiently large value of n, we have `f(n) > 0` .

If `a ≥ 1` and `b > 1` are constants and `f(n)` is an asymptotically positive function, then the time complexity of a recursive relation is given by

```
T(n) = aT(n/b) + f(n)

where, T(n) has the following asymptotic bounds:

    1. If f(n) = O(n^(log_b a-ϵ)), then T(n) = θ(n^(log_b a)).

    2. If f(n) = θ(n^(log_b a)), then T(n) = θ(n^(log_b a) * log n).

    3. If f(n) = Ω(n^(log_b a+ϵ)), then T(n) = θ(f(n)).

ϵ > 0 is a constant.
```

Each of the above conditions can be interpreted as:

1. If the cost of solving the sub-problems at each level increases by a certain factor, the value of `f(n)` will become polynomially smaller than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of the last level ie. $n^{\log_b a}$
2. If the cost of solving the sub-problem at each level is nearly equal, then the value of `f(n)` will be $n^{\log_b a}$. Thus, the time complexity will be `f(n)` times the total number of levels ie. $n^{\log_b a} * \log n$
3. If the cost of solving the subproblems at each level decreases by a certain factor, the value of `f(n)` will become polynomially larger than $n^{\log_b a}$. Thus, the time complexity is oppressed by the cost of `f(n)`.

Solved Example of Master Theorem

```
T(n) = 3T(n/2) + n2
Here,
a = 3
n/b = n/2
f(n) = n^2

log_b a = log_2 3 ≈ 1.58 < 2

ie. f(n) < n^(log_b a+ϵ) , where, ϵ is a constant.

Case 3 implies here.

Thus, T(n) = f(n) = θ(n^2)
```

Master Theorem Limitations

The master theorem cannot be used if:

- `T(n)` is not monotone. eg. `T(n) = sin n`
- `f(n)` is not a polynomial. eg. `f(n) = 2^n`
- `a` is not a constant. eg. `a = 2n`
- `a < 1`

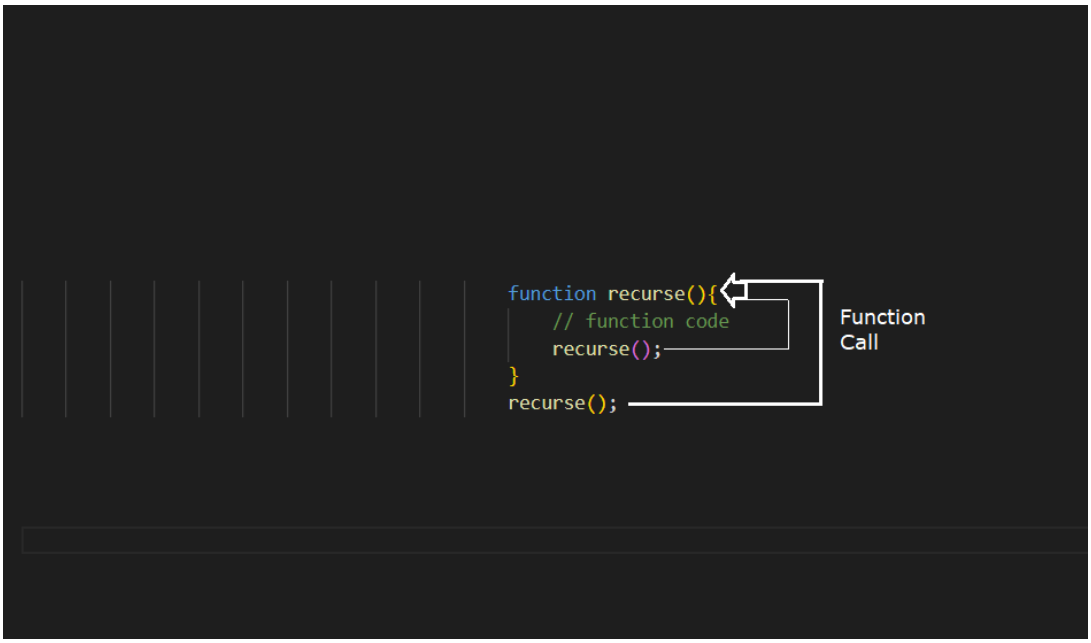
JavaScript Recursion

Recursion is a process of calling itself. A function that calls itself is called a recursive function.

The syntax for recursive function is:

```
function recurse() {
    // function code
    recurse();
    // function code
}

recurse();
```

Here, the recurse() function is a recursive function. It is calling itself inside the function.



A recursive function must have a condition to stop calling itself. Otherwise, the function is called indefinitely.

Once the condition is met, the function stops calling itself. This is called a base condition.

To prevent infinite recursion, you can use if...else statement (or similar approach) where one branch makes the recursive call, and the other doesn't.

So, it generally looks like this.

A simple example of a recursive function would be to count down the value to 1.

Example 1: Print Numbers

```
// program to count down numbers to 1
function countDown(number) {

    // display the number
    console.log(number);

    // decrease the number value
    const newNumber = number - 1;

    // base case
    if (newNumber > 0) {
        countDown(newNumber);
    }
}

countDown(4);
```

Output

```
4
3
2
1
```

In the above program, the user passes a number as an argument when calling a function.

In each iteration, the number value is decreased by **1** and function `countDown()` is called until the number is positive. Here, `newNumber > 0` is th base condition.

This recursive call can be explained in the following steps:

```
countDown(4) prints 4 and calls countDown(3)
```

```
countDown(3) prints 3 and calls countDown(2)
countDown(2) prints 2 and calls countDown(1)
countDown(1) prints 1 and calls countDown(0)
```

When the number reaches **0**, the base condition is met, and the function is not called anymore.

Example 2: Find Factorial

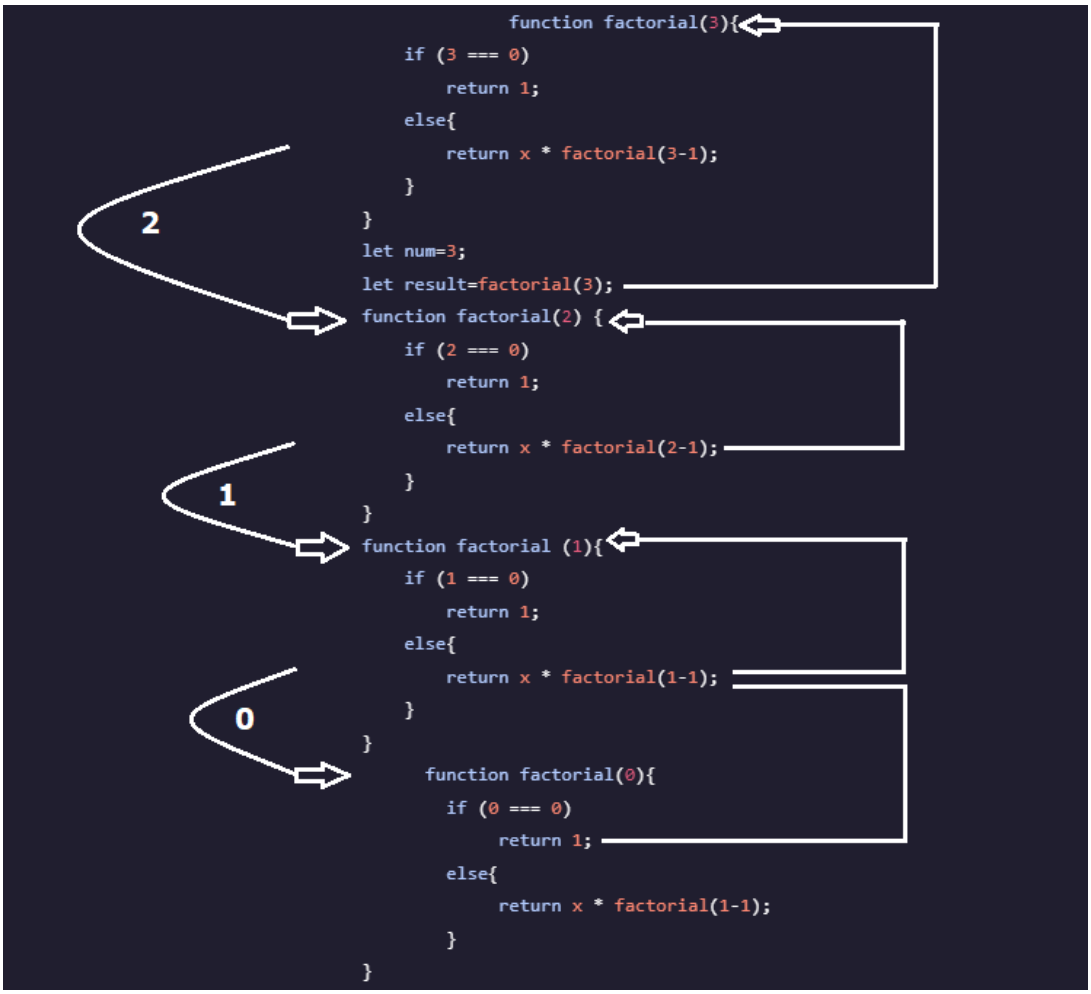
```
// program to find the factorial of a number
function factorial(x) {

    // if number is 0
    if (x === 0) {
        return 1;
    }

    // if number is positive
    else {
        return x * factorial(x - 1);
    }
}

const num = 3;

// calling factorial() if num is non-negative
if (num > 0) {
    let result = factorial(num);
    console.log(`The factorial of ${num} is ${result}`);
}
```

Output

```
The factorial of 3 is 6
```

When you call function `factorial()` with a positive integer, it will recursively call itself by decreasing the number.

This process continues until the number becomes **1**. Then when the number reaches **0**, **1** is returned.



This recursive call can be explained in the following steps:

```
factorial(3) returns 3 * factorial(2)
factorial(2) returns 3 * 2 * factorial(1)
factorial(1) returns 3 * 2 * 1 * factorial(0)
factorial(0) returns 3 * 2 * 1 * 1
```

Calculate the sum of n natural numbers example

Suppose you need to calculate the sum of natural numbers from 1 to n using the recursion technique. To do that, you need to defir the `sum()` recursively as follows:

```
sum(n) = n + sum(n-1)
sum(n-1) = n - 1 + sum(n-2)
...
sum(1) = 1
```

The following illustrates the sum()recursive function:

```
function sum(n) {
  if (n <= 1) {
    return n;
  }
  return n + sum(n - 1);
}
```

Scenario Example

```
function growBeanstalk(years) {
   if (years <= 0) {
     return 0;
     }
         else if (years <= 2) {
         //console.log(1);
         return 1;
     }
      console.log("recursion of " + years + ":")console.log(years-1);console.log(years-2)
          return growBeanstalk(years - 1) +
            growBeanstalk(years - 2);
   }
   growBeanstalk(5);
```

- Run this two times, the first time with the first console.log commented out (//) and the second console.log uncommented. Only when years<=2 will the value 1 be returned.

```
The results are:
```

```
recursion of 5: 4 3 recursion of 4: 3 2 recursion of 3: 2 1 recursion of 3: 2 1 ==> 5.
```

- The second time run with the first console.log uncommented and add the comments (//) to the second console.log. This will show us every time the else if statement is evaluated when years<= 2 and the value 1 is returned. This is the step that produces the desired sum. The results are: 1 1 1 1 1 ==> 5 Play around with the number used as the argument to see how the pattern works.

JavaScript Program to Display Fibonacci Sequence Using Recursion

Example: Q - Print the first n numbers in Fibonacci series

```
function fibonacci(n){
 if(n < 2) return n
 return fibonacci(n-1) + fibonacci(n-2)
}

const num = 7;

for(let i = 0; i<num;  ++i){
 console.log(fibonacci(i))
}
```

Output

```
0
1
1
2
3
5
8
```

Summary

- A recursive function is a function that calls itself until it doesn't
- A recursive function always has a condition that stops the function from calling itself.

Interview Questions

- **What is Recursion?**

Recursion is a method of program design where you break apart a problem into smaller repeatable subtasks. The program will complete each subtas later combined to achieve a solution.

- **What is the difference between Recursion and Iteration?**

Recursive solutions use self-calling methods and run until their base case is reached. Iterative solutions do not call themselves and instead are repeate until a certain number of loops are reached or until a condition is met (`i==10` , for example).

Iterative solutions have the upper hand in memory usage and speed (usually).

These two benefits are actually derived from the same quality; while recursive methods add a new call to the call stack with each recurrence, iterati methods add only one call for the whole loop! This means less methods are stored and called, meaning the program uses less memory and usual creates a faster run-time.

- **What is Base Case?**

The base case (or base condition) is the state where the program's solution has been reached. An achievable base case is essential to avoid an infini loop. Recursive methods are built with two paths: the method first checks if the base state has been reached, if yes, the method ends and returns th current data, if not the method instead goes the other path and executes the recursive case, altering the input and calling the method again.