**Recursion**

A function that refers to itself inside of the function.

function foo() {

    foo();

}

foo();

Recursion is good for tasks that are repetitive.

**Recursive functions have 2 paths:**

1. The recursive function
2. The base case

**Recursion Rules:**

1. Identify the base case
2. Identify the recursive case
3. Get closer and closer and return when needed. Usually, you have 2 returns meaning we must return the base case and the recursive case.

Diagram

Description automatically generated

**Base Case**

Condition to meet to exit out of recursive function.

let counter = 0;

function foo() {

    if (counter > 3) { // this is our base case

        return 'done';

    }

    counter++

    foo(); // there is no return here so the output will be undefined

}

foo();

This function will return undefined.

let counter = 0;

function foo() {

    if (counter > 3) { // this is our base case

        return 'done';

    }

    counter++

    return foo(); // this will return ‘done’

}

foo();

This function will return ‘done’ because of the return foo();

**Note\*** with recursion, you must return the recursive function so that the value bubbles up to the top otherwise you will get undefined.

O(2^n) – exponential time – Recursive Function

// O(2^n) - exponential time.  Size increases exponentially.

function findFactorialRecursive(number) {

    if (number === 2) {

        return 2;

    }

    return number \* findFactorialRecursive(number - 1)

}

findFactorialRecursive(5);

O(n) – Iterative Function

// O(n)

function findFactorialIterative(number) {

    let answer = 1;

    for (let i = 2; i <= number; i++) {

        answer = answer \* i;

    }

    return answer;

}

findFactorialIterative(5);

**Advantages of Recursion:**

* DRY
* Readability
* Great for tree data structures and traversal

**Disadvantages of Recursion:**

* Large Stack
* Avoid too many recursive calls if memory is expensive
* Extra memory footprint although it keeps the code DRY

**Tail Call Optimization**

* Allows recursion to be called without using the call stack.

**When to use recursion?**

* Every time you are using a tree or converting something into a tree.
* A problem can be divided into several subproblems that are smaller instances of the same problem.
* Each instance of the subproblem is identical in nature.
* The solutions of each subproblem can be combined to solve the problem at hand.
* Divide and Conquer algorithms

**Recursion used in:**

* Merge Sort
* Quick Sort
* Tree Traversal
* Graph Traversal