1. Recursion vs Iteration

Important Links

- Educative Link - Recursion versus Iteration

General Idea behind Iteration & Recursion

- The concept of *Recursion* & *Iteration* is to execute a set of instruction repeatedly.
- Both *Iteration* & *Recursion* depend on a condition which determines whether to stop the execution or repeat it

Differences between Iteration & Recursion

RECURSION	ITERATION
Recursion refers to a situation where a function calls itself again and again until some *base condition* is reached.	Iteration refers to a situation where some statements are executed again and again using loops until some condition is satisfied.
Recursion is always called on a function, therefore it is called a **process**.	Iterative code is applied on variables and is a set of instructions that are executed repeatedly.
Recursive code terminates due to the *base condition* being satisfied.	Iterative code either runs for a particular number of times or until a specified condition is met.
Recursive code is slower than iterative code as it not only runs the program but also has to manage stack memory.	Iterative code has a relatively faster runtime.
Recursive function uses Stack	Iterative function does not use Stack

Steps for Converting Iterative Code to Recursive

The following are some of the MOST important points of observation when it comes to effectively writing a recursive function / algorithm -

- Identify the *Most simplest task* required to complete / exit the loop and mark it as the **Base Case**.
- There can be more than 1 ***base cases***
- **Local variables** of a Iterative version (usually) turn into **Parameters** in a Recursive version
- If the Local Variable is used as parameter - the Recursive function should either include
    - Passing the same function **as a value** to the Parameter (ie., Variable assignment) & then return the parameter
    - Having a **return keyword** to the same function
- If somewhere in your recursive function there is a `(n-2)`, then your base case condition should be as follows

```
if (n <= 1) return x; // the <= is very important as n-2 can lead to -ve number scenario
```

- There are ALWAYS **2** return statements in a recursive function
    - One for base case
    - One for recursion
    - Sometimes you can assign to a variable the recursion, but then you need to return that variable

> *NOTE: Your base case & recursive case **BOTH** must have a **RETURN** keyword. Recursive function most likely ALWAYS **calls the recursive function, with a diminishing condition***

i. Find the Square of a Number Recursively

Implement a function that takes a number `testVariable` and returns the square of the number. Try using the following mathematical identity to solve this problem -

$$(n-1)^2 = n^2 - 2*n + 1$$

Solution

To Solve the above problem recursively, we need to tweak the formula

$$(n-1)^2 = n2 - 2*n + 1$$

$$(n-1)^2 + 2n - 1 = n^2$$

OR

$$recursiveFn() = recursiveFn(n-1) + 2 * n - 1$$

```
function findSquare(testVariable) {
  if (testVariable == 0) return 0;
  return findSquare(testVariable-1) + 2*testVariable -1;
}
```

ii. Search the First Occurrence of a Number

Implement a function that takes an array `arr`, a `testVariable` containing the number to search and `currentIndex` containing the starting index as parameters and outputs the index of the first occurrence of `testVariable` in `arr`. If `testVariable` is not found in `arr` return `-1`.We want to search for a `targetNumber` from index `currentIndex` to the end of the array.

Solution

```
const firstIndex = (arr, testVariable, currentIndex) => { //
returns the first occurrence of testVariable
  // Base case1
  if (arr.length == currentIndex) {
    return -1;
  }

  // Base case2
  if (arr[currentIndex] == testVariable) {
    return currentIndex;
  }

  // Recursive case
  return firstIndex(arr, testVariable, currentIndex + 1);
};

// Test Solution
var arr = [9, 8, 1, 8, 1, 7];
var testVariable = 1;
var currentIndex = 0;
console.log(firstIndex(arr, testVariable, currentIndex));
```

iii. Write an Algorithm to find the nth Fibonacci Number

Implement a function that takes a variable `testVariable` and finds the number that is placed at that index in the *Fibonacci sequence*.

What is the Fibonacci Sequence?

The Fibonacci sequence is one of the most famous sequences in mathematics. Each number in the sequence is the sum of the two numbers that precede it.

So, the sequence goes:

$ 0, 1, 1, 2, 3, 5, 8, 13, 21, 34 ... $

Solution

```
function fibonacci(testVariable) {
  // Base case
  if (testVariable <= 1) {
    return testVariable;
  }

  return(fibonacci(testVariable - 1) + fibonacci(testVariable -
2));
}

// Driver Code
var testVariable = 7;
console.log(fibonacci(testVariable));
```

2. Recursion with Numbers

i. The Power of a Number

Write a function, recursively to return - *The Power of a Number*

The **power** (or **exponent**) `a`, of a number `x`, represents the number of times `x` needs to be multiplied by itself. `x` to the power `a` is the product of `x` with itself `a` times!

It's written as a small number to the right and above the ***base*** number.For example, if the base number is `2` and the ***exponent*** value is `3`, we represent this as follows:

$$2^3 = 8$$

- 3 = exponent
- 2 = base

- 8 = result

Approach

Let's break this problem down -

- The minimum value of the power of any number is when the ***exponent*** reaches `0`
- When the ***exponent*** reaches 0, the return value is `1`
- So, that's our ***base*** case -

```
if (exponent == 0) return 1;
```

- Now, we must strive for the ***exponent*** to reach this minimum value, while returning the logic
- Let's say our recursive function is called ***recursiveFn( )***
- It will take 2 values - ***base*** & ***exponent***

```
recursiveFn(base, exponent) { .... }
```

- Then our recursive case to reach the minimum value is to multiply the ***base*** with `recursiveFn(base, exponent-1)`

Solution

The above approach can then be coded as follows

```
const recursiveFn = (base, exponent) => {
  if (exponent == 0) return 1;
  else return base * recursiveFn(base, exponent-1);
};
```

ii. Find the Greatest Common Divisor

Given 2 numbers - m, n, find the Greatest common divisor among them

For example, take 2 numbers - 42 & 56

- 42 can be completely divided by 1,2,3,6,7, **14**, 21 & 42
- 56 can be completely divided by 1,2,4,7,8, **14**, 28 & 56
- So the Greatest Common Divisor of 42 & 56 is **14**

Approach

The following is the mathematical simplification of finding the **Greatest Common Divisor - GCD**

$$gcd(m,n) = \begin{vmatrix} m & if(m == n) \\ gcd(m-n,n) & if(m > n) \\ gcd(m,n-m) & if(m < n) \end{vmatrix}$$

Solution

So, using that above equation we can apply the recursive function and write the solution as follows

```
// greatest common divisor
const gcd = (testVar1, testVar2) => {
  // base case
  if (testVar1 == testVar2) return testVar1;
  // recursive case #1
  if (testVar1 > testVar2) return gcd(testVar2 - testVar1,
testVar1);
  else if (testVar2 < testVar1) return gcd(testVar1, testVar2 -
testVar1);
};

// main script
const var1 = 6;
const var2 = 9;
console.log(`Greatest common divisor of ${var1} & ${var2} is: `,
gcd(var, var2));
```

iii. Pascal's Triangle

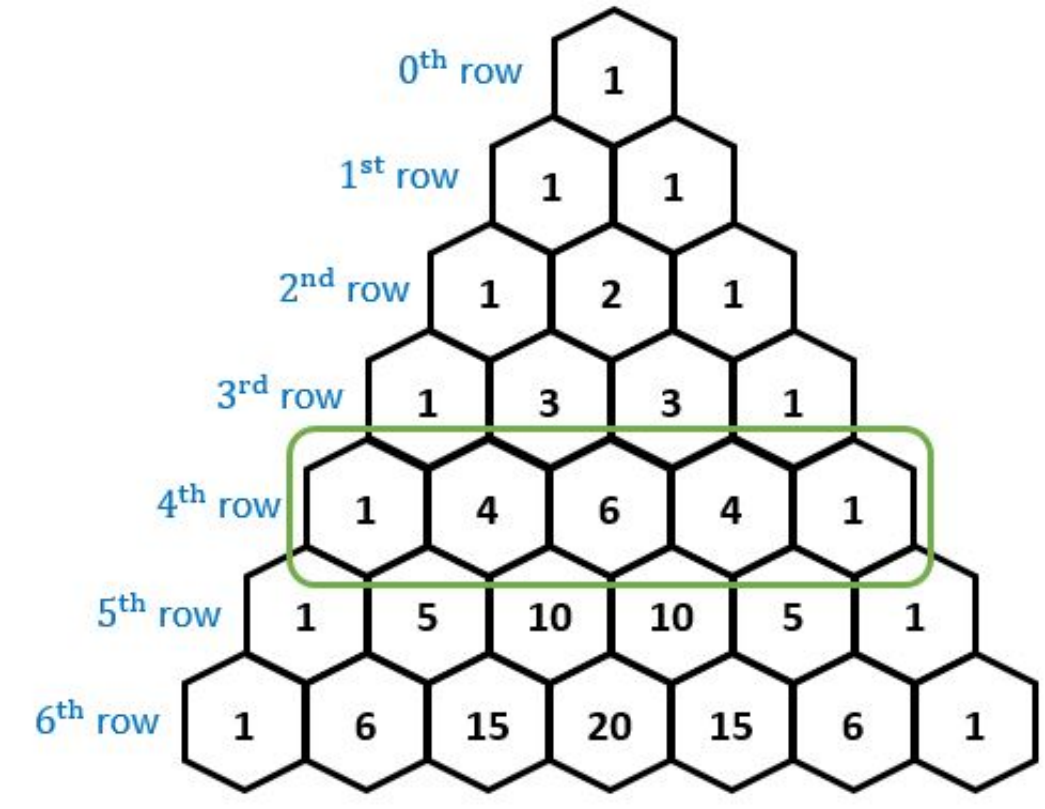
Given a number, return a list containing the values of the Pascal's Triangle of that size.

What is Pascal's triangle ?

Pascal's triangle is a triangular array constructed by summing adjacent elements in preceding rows.

Pascal's triangle contains the values of the binomial coefficient. It is named after the 17th century French mathematician, *Blaise **Pascal** (1623 - 1662)*.

Pascal's triangle is a triangular array of the binomial coefficients.



The entry in the **nth** row and **kth** column of Pascal's triangle is denoted as:

$$\left|\begin{matrix} n \\ k \end{matrix}\right|$$

For example, the unique nonzero entry in the topmost row is

$$\left|\begin{matrix} 0 \\ 0 \end{matrix}\right| = 1$$

with this notation, any element can be written as

$$\left|\begin{matrix} n \\ k \end{matrix}\right| = \left|\begin{matrix} n-1 \\ k-1 \end{matrix}\right| + \left|\begin{matrix} n-1 \\ k \end{matrix}\right|$$

Approach

Let's look at the minimum condition required. This will be our base case

- The minimum condition is if the input = 1
- if input = 1, we just return [1]. This will be the base case for the recursion

- Now let's see how to strive to this condition

Solution

The above approach can be written in javascript as follows:

```
const pascalsTriangle = (input) => {
    // base case
    if (input == 1) return [1];
    // recursive case
    const previousLine = pascalsTriangle(input-1);
    let currentLine = [1];
    for(let i=0; i<previousLine.length-1; i++) {
        currentLine.push(previousLine[i] + previousLine[i+1]);
    }
    currentLine.push(1);
    return currentLine;
}

// Test method
const row = 5;
console.log(`Pascals Triangle for the row ${row} is: `,
pascalsTriangle(row));
// [1,4,6,4,1]
```