Type Coercion

1. Memory Implicit Conversion: JavaScript automatically converts one data type to another when required.
2. String Conversion: Numbers and booleans can be implicitly converted to strings.
3. Number Conversion: Strings and booleans can be implicitly converted to numbers.
4. Boolean Conversion: Various data types can be converted to booleans (e.g., 0, "", null, undefined are false).

```
// String Conversion
let num = 10;
let str = "The number is " + num;
console.log(str); // Output: "The number is 10"

// Number Conversion
let strNum = "15";
let result = strNum - 5;
console.log(result); // Output: 10

// Boolean Conversion
let boolValue = true;
let numberValue = 1;
 // Output: true
console.log(boolValue == numberValue);
//true is converted to 1 before comparison
```

Type Coercion

1. Memory Addition (+)
   - Number + Number: Adds the numbers.
   - String + String: Concatenates the strings.
   - String + Number: Converts the number to a string.
   - Boolean + String: Converts the boolean to a string.
2. Subtraction (-), Multiplication (*), Division (/), Modulus (%)
   - Number - Number: Subtracts the numbers.
   - String - Number: Converts the string to a number.
   - Boolean - Number: Converts the boolean to a number (true to 1, false to 0).
3. Equality (==)
   - Number == String: Converts the string to a number.
   - Boolean == Number: Converts the boolean to a number.
   - Boolean == String: Converts the boolean to a number and the string to a number.
4. Strict Equality (===)
   - No type coercion is performed. The types and values must be identical.

```
// Addition (+)
console.log(1 + 2);          // 3 (Number + Number)
console.log('Hi' + ' Sam'); // "Hi Sam" (String + String)
console.log(5 + '5');        // "55" (Number + String)
console.log(true + 'yes');  // "trueyes" (Boolean + String)

// Subtraction (-)
console.log(10 - 5); // 5 (Number - Number)
console.log('10' - 5); // 5 (String - Number, "10" converted to 10)
console.log(10 - '5'); // 5 (Number - String, "5" converted to 5)
console.log(true - 1); // 0 (Boolean - Number, true converted to 1)

// Multiplication (*)
console.log(2 * 3); // 6 (Number * Number)
console.log('2' * 3); // 6 (String * Number, "2" converted to 2)
console.log(2 * '3'); // 6 (Number * String, "3" converted to 3)
console.log(false * 10); // 0 (Boolean * Number, false converted to 0)

// Equality (==)
console.log(5 == '5');        // true (String converted to Number)
console.log(true == 1);       // true (Boolean converted to Number)
console.log(false == 0);      // true (Boolean converted to Number)

// Strict Equality (===)
console.log(5 === '5');       // false (Different types)
console.log(true === 1);      // false (Different types)
```

== VS ===

```javascript
let num = 5;
let str = "5";

// true, because "5" == "5" after type coercion
console.log(num == str);
// false, because 5 (number) !== "5" (string)
console.log(num === str);
```

```javascript
let booleanValue = true;
let numValue = 1;

// true, because true is converted to 1
console.log(booleanValue == numValue);
// false, because true (boolean) !== 1 (number)
console.log(booleanValue === numValue);
```

1. ==: Performs type coercion, converting values to a common type before comparing.
2. ===: Does not perform type coercion, compares both value and type.
3. ==: Loosely checks for equality, can lead to unexpected results.
4. ===: Strictly checks for equality, more predictable and recommended.

Memory Allocation

```
// number type
let age = 25;
// string type
let name = 'John';
```

```
// initially a number
let data = 42;
// reassign to a string
data = 'Hello World';
```

1. age Variable:
   - Memory Allocation: Stack
   - Variable Memory: Holds the reference (address) to the actual data.
   - Value Memory: Typically, numbers occupy 8 bytes in memory.
2. name Variable:
   - Memory Allocation: Stack
   - Variable Memory: Holds the reference (address) to the actual data.
   - Value Memory: Strings occupy memory based on their length. Each character typically uses 2 bytes (UTF-16 encoding).

1. Initial State (data as Number):
   - Memory Allocation: Stack
   - Variable Memory: Holds the reference to the number.
   - Value Memory: 8 bytes for the number.
2. After Reassignment (data as String):
   - Memory Allocation: Stack (reference), Heap (actual string data)
   - Variable Memory: Holds the reference to the string.
   - Value Memory: Length of "Hello World" (11 characters) x 2 bytes = 22 bytes on the heap.

Garbage Collection

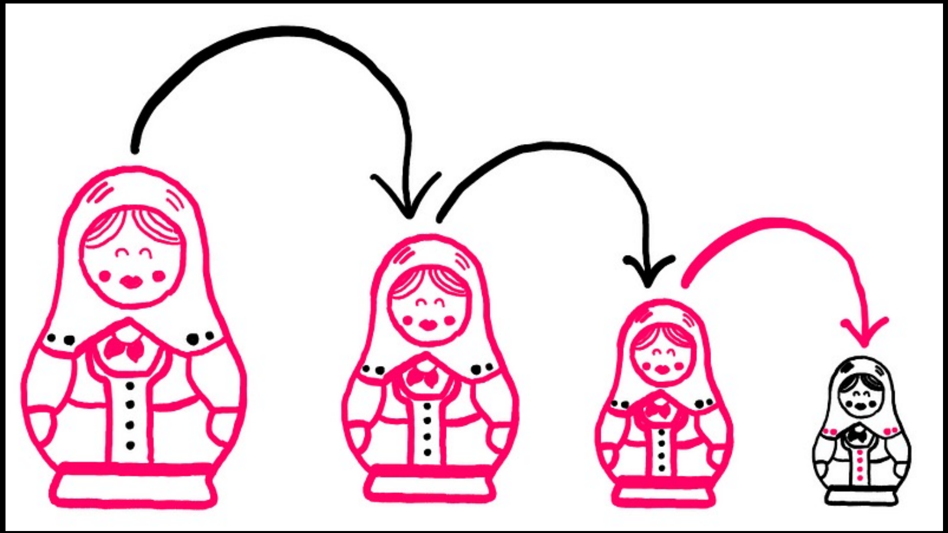
1. Automatic memory management using garbage collection.
2. Reachability determines if an object is collected.
3. Performance optimization to reduce pauses.
4. Memory leaks can still occur due to lingering references.
5. Best practices include minimizing global variables and clearing unused references.

Common Algorithms:

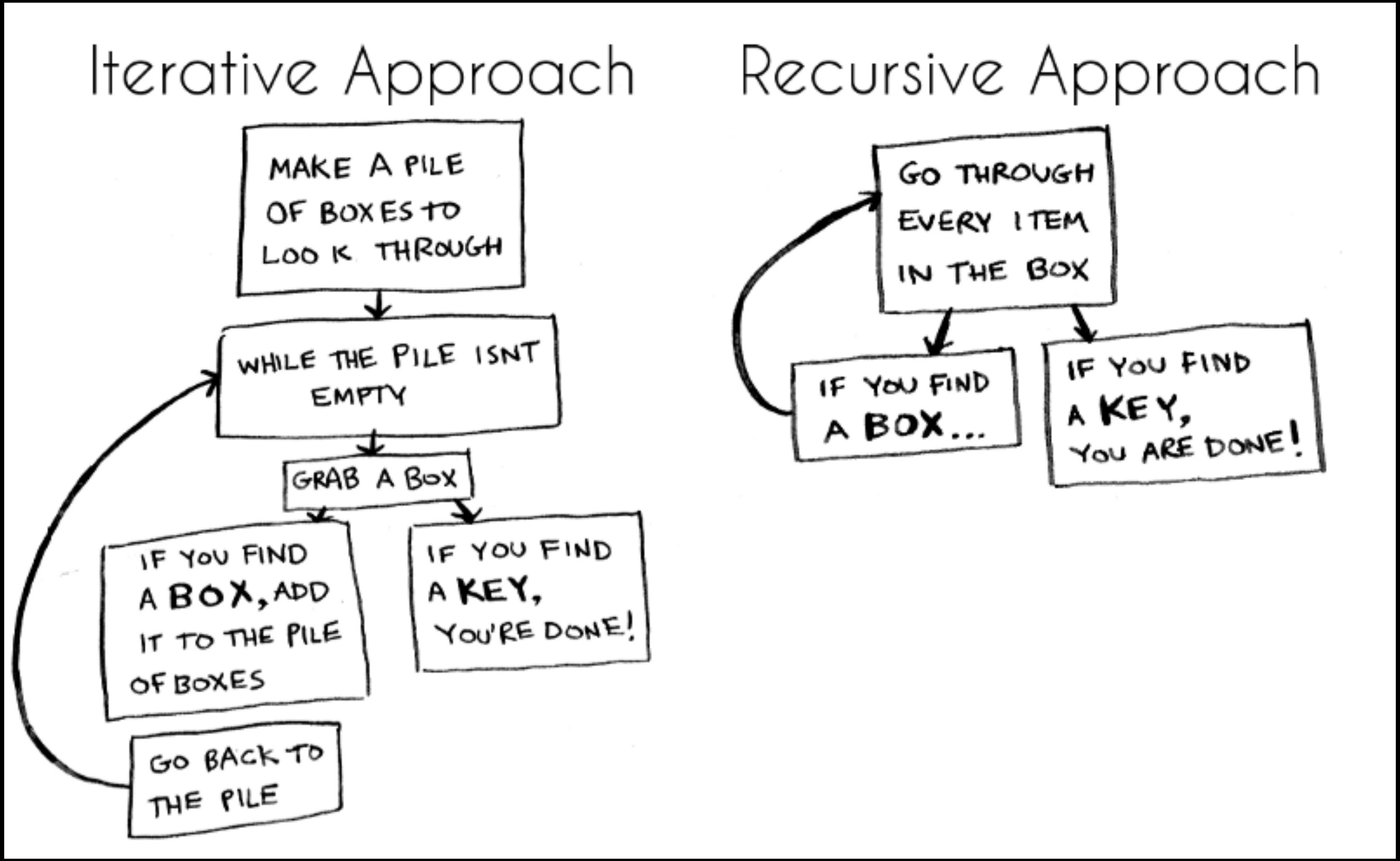
- Mark-and-Sweep: The most common garbage collection algorithm. It marks all reachable objects and then sweeps or collects all unmarked objects, freeing up their memory.
- Reference Counting: Keeps track of the number of references to each object. An object is collected when its reference count drops to zero. However, it cannot handle circular references.

Recursion

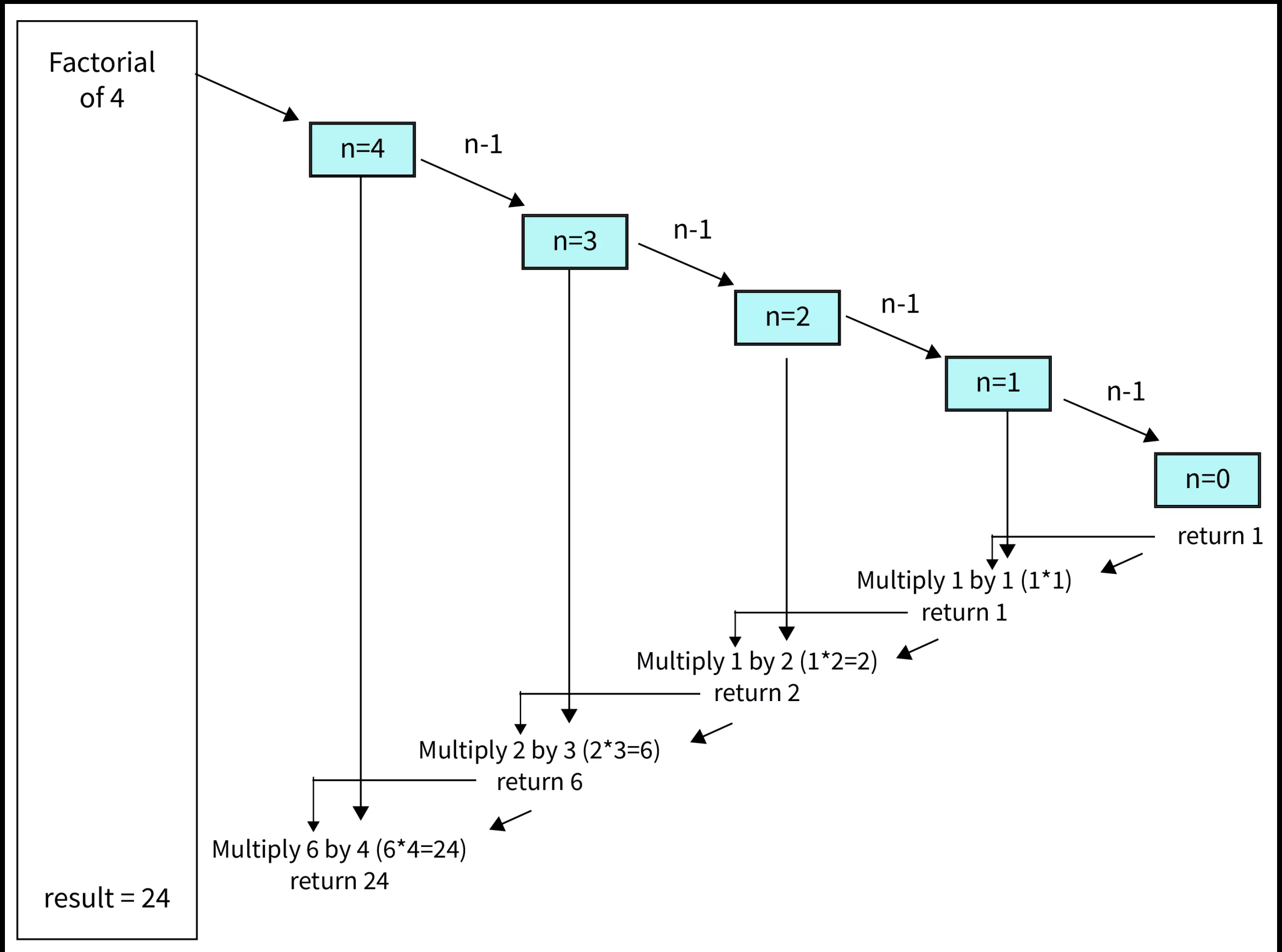
1. Self-Calling: A function that calls itself to solve sub-problems.
2. Base Case: Essential to stop recursion and prevent infinite loops.
3. Recursive Case: The condition under which the function keeps calling itself.
4. Stack Usage: Consumes stack space with each call, risk of overflow.
5. Applications: Ideal for divisible tasks, tree/graph traversals, sorting algorithms.
6. Memory Intensive: More overhead than iterative solutions.
7. Clarity: Often simplifies complex problems.



Iterative vs Recursive



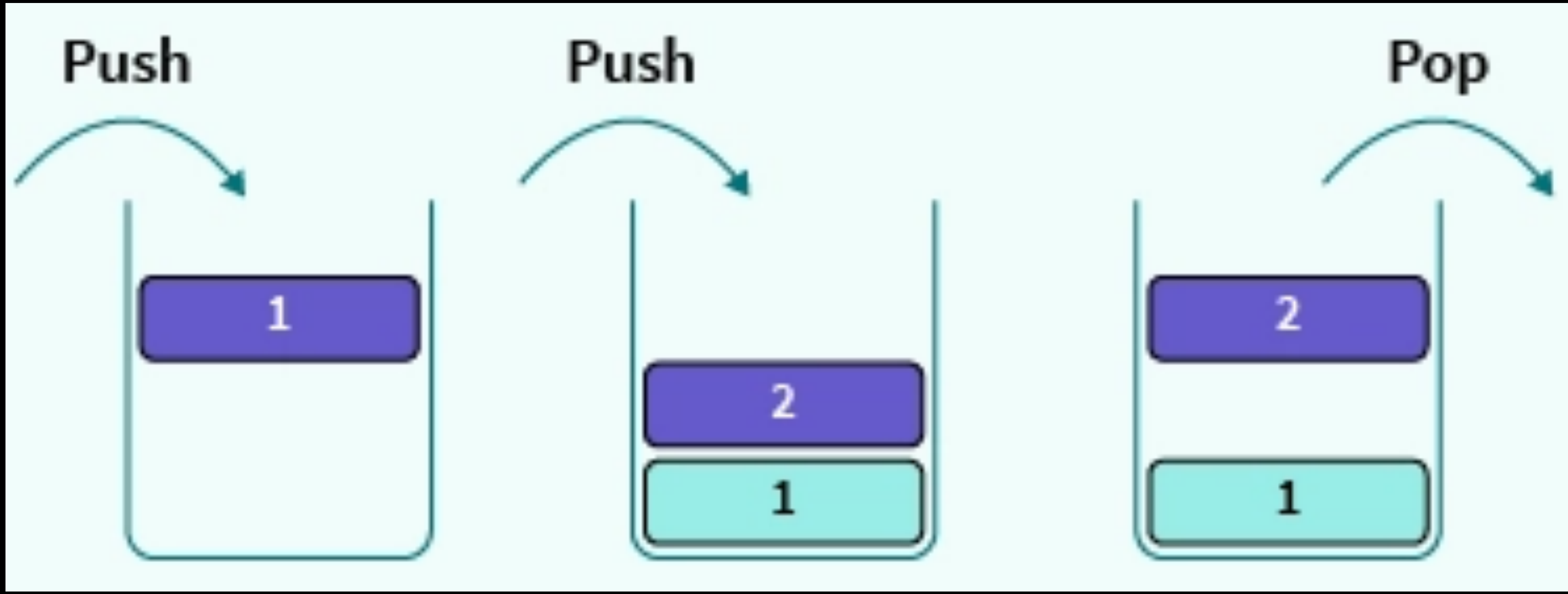
Factorial Using Recursion

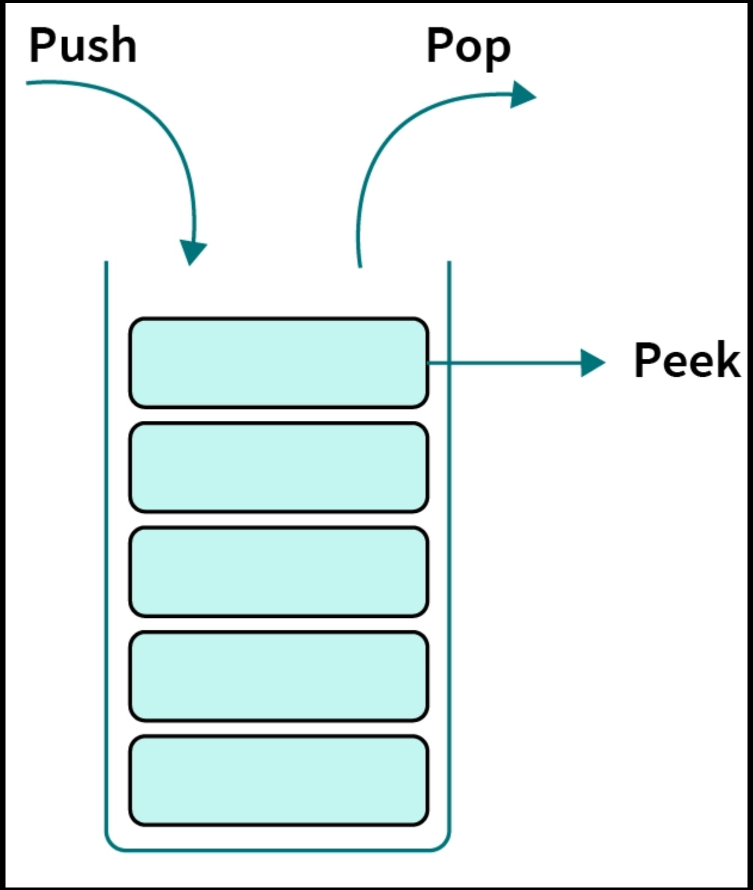



```javascript
function factorial(n) {
  // Base case: factorial of 0 is 1
  if (n === 0) {
    return 1;
  }
  // Recursive case: n! = n * (n-1)!
  return n * factorial(n - 1);
}

// Example usage:
console.log(factorial(5)); // Output: 120
console.log(factorial(0)); // Output: 1
console.log(factorial(3)); // Output: 6
```

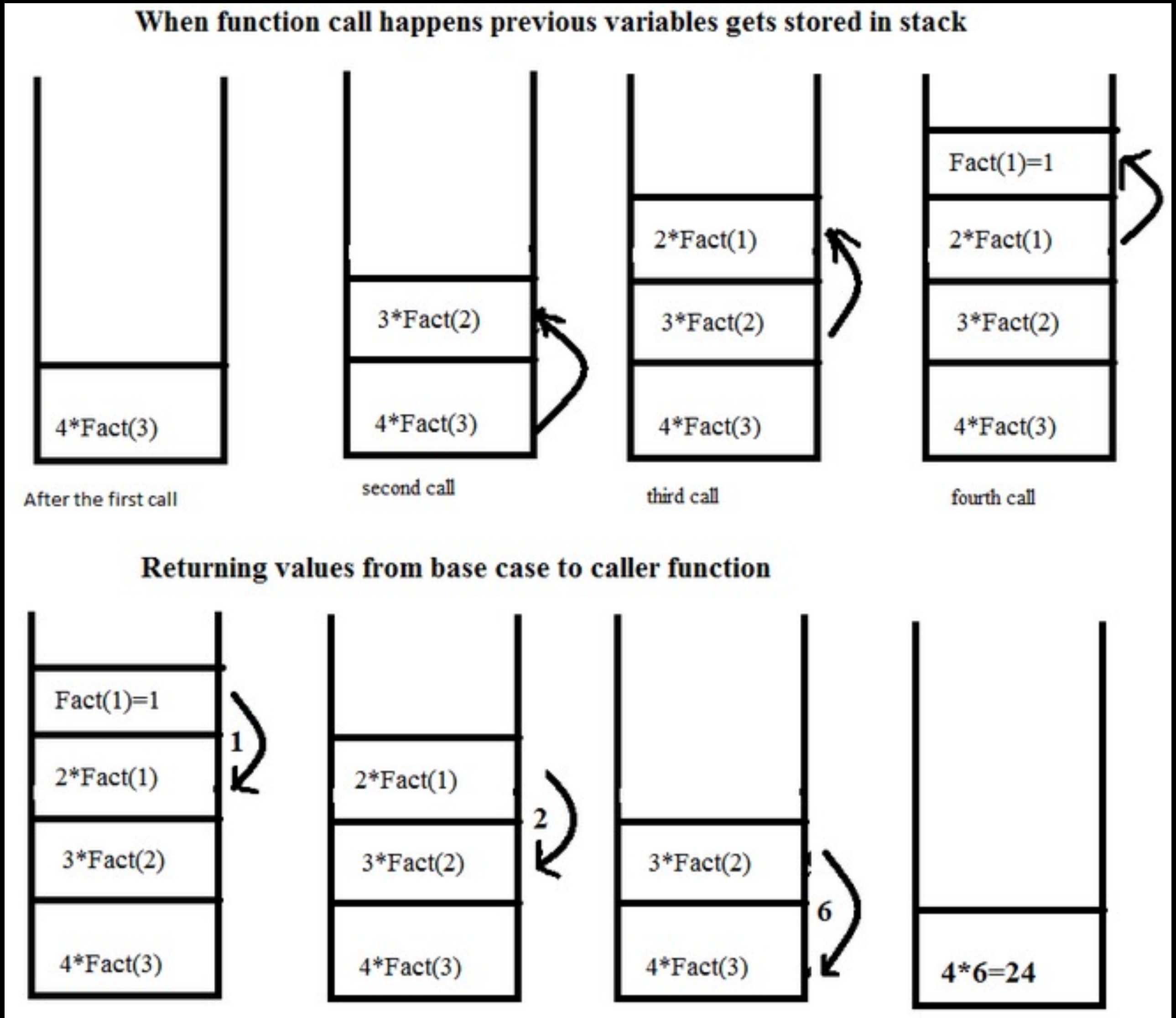
What is Stack





1. LIFO: Last-In, First-Out operation; the last element added is the first to be removed.
2. Operations: Mainly push (add an item) and pop (remove an item).
3. Top Element: Only the top element is accessible at any time, not the middle or bottom ones.
4. Overflow and Underflow: Overflow when full, underflow when empty during operations.

Recursion Stack



1. Memory Allocation: Recursive calls add frames to the call stack for variables and return points.
2. Growth with Depth: Deeper recursion equals more stack space.
3. Base Case: Essential to limit recursion depth and prevent stack overflow.
4. Stack Overflow Risk: Excessive recursion depth can crash the program.
5. Tail Recursion Optimization: Can reduce stack usage for certain recursive patterns.
6. Efficiency: Iterative solutions may be more stack-efficient than recursion for some problems.

Practice Exercise

Recursion

1. Write a recursive function to print all numbers from 1 to N.
2. Create a function to calculate Fibonacci element using recursion.
3. Write a recursive function to find the sum of digits of a given positive integer n.
4. Write a recursive function to calculate x raised to the power of n (i.e., x^n).
5. Write a recursive function to find the greatest common divisor (GCD) of two numbers a and b.
6. Write a recursive function to count the number of times a specific character appears in a string.
7. Write a recursive function to check if a given string is a palindrome (reads the same forwards and backwards).