Recursion and tail call optimization

Absolutely! Let's dive deep into **Recursion** and **Tail Call Optimization (TCO)** — two powerful concepts in JavaScript.

🔁 Recursion — What Is It?

Recursion is a technique where a function **calls itself** to solve a smaller version of a problem until it reaches a **base case**.

🧠 Basic Structure of a Recursive Function

```
function recurse(args) {
  if (baseCondition) {
    return result;
  } else {
    return recurse(smallerArgs);
  }
}
```

✅ Example: Factorial of a Number

```
function factorial(n) {
  if (n === 0 || n === 1) return 1;
  return n * factorial(n - 1);
}

console.log(factorial(5)); // Output: 120
```

- **Base case:** `n === 0 || n === 1`
- **Recursive case:** `n * factorial(n - 1)`

🔄 Another Example: Sum of Array Elements

```
function sum(arr) {
  if (arr.length === 0) return 0;
  return arr[0] + sum(arr.slice(1));
}

console.log(sum([1, 2, 3, 4])); // Output: 10
```

⚠️ Pitfalls of Recursion

- Recursive functions can **consume a lot of memory** due to **call stack** growth.
- Deep recursion can cause **stack overflow** errors.

Example:

```
function recurseForever() {
  recurseForever();
}

recurseForever(); // ❌ RangeError: Maximum call stack size exceeded
```

🚀 Tail Call Optimization (TCO)

🧾 Definition:

Tail Call Optimization is an optimization technique where the **JavaScript engine reuses the current stack frame** for a recursive function **if the recursive call is the last thing the function does**.

📌 What is a Tail Call?

A **tail call** happens when a function returns the result of a **direct function call**, with **no additional computation** afterward.

✅ Tail Call Example:

```
function factorial(n, acc = 1) {
  if (n === 0) return acc;
  return factorial(n - 1, n * acc); // Tail call
}
```

Here, the recursive call to `factorial` is the **last statement**, so it's eligible for optimization.

❌ Non-Tail Call Example:

```
function factorial(n) {
  if (n === 0) return 1;
  return n * factorial(n - 1); // Not a tail call (multiplication happens after)
}
```

Here, `n * factorial(n - 1)` means the multiplication is deferred, so it's **not** a tail call.

🛑 Caveat: Is TCO Supported in JavaScript?

Unfortunately, **Tail Call Optimization is not reliably supported** in most JavaScript engines (including Chrome's V8 and Node.js), despite being part of the **ES6 specification**.

✅ Safari (WebKit) has some support, but **you should not rely on TCO** for production unless you're using a compiler like Babel or targeting environments that support it.

🧠 Iterative vs Recursive: Factorial

❌ Recursive (without TCO):

```
function factorial(n) {
  if (n === 0) return 1;
```

```
  return n * factorial(n - 1);
}
```

✅ Iterative (safe alternative):

```
function factorial(n) {
  let result = 1;
  for (let i = 2; i <= n; i++) {
    result *= i;
  }
  return result;
}
```

📦 When to Use Recursion

- Tree/graph traversal (DOM, JSON, file systems)
- Divide and conquer algorithms (merge sort, quicksort)
- Problems that can be broken into similar subproblems

📝 Summary

Concept	Explanation
Recursion	Function calls itself to solve a problem in smaller steps
Base Case	Stops the recursion
Tail Call	When the recursive call is the **last thing** executed
Tail Call Optimization	Optimization to avoid growing the call stack (not widely supported in JS)
Risk	Deep recursion may cause **stack overflow** without TCO

Would you like to explore an example of recursion with the DOM or move on to **Generator Functions and** `yield` next?