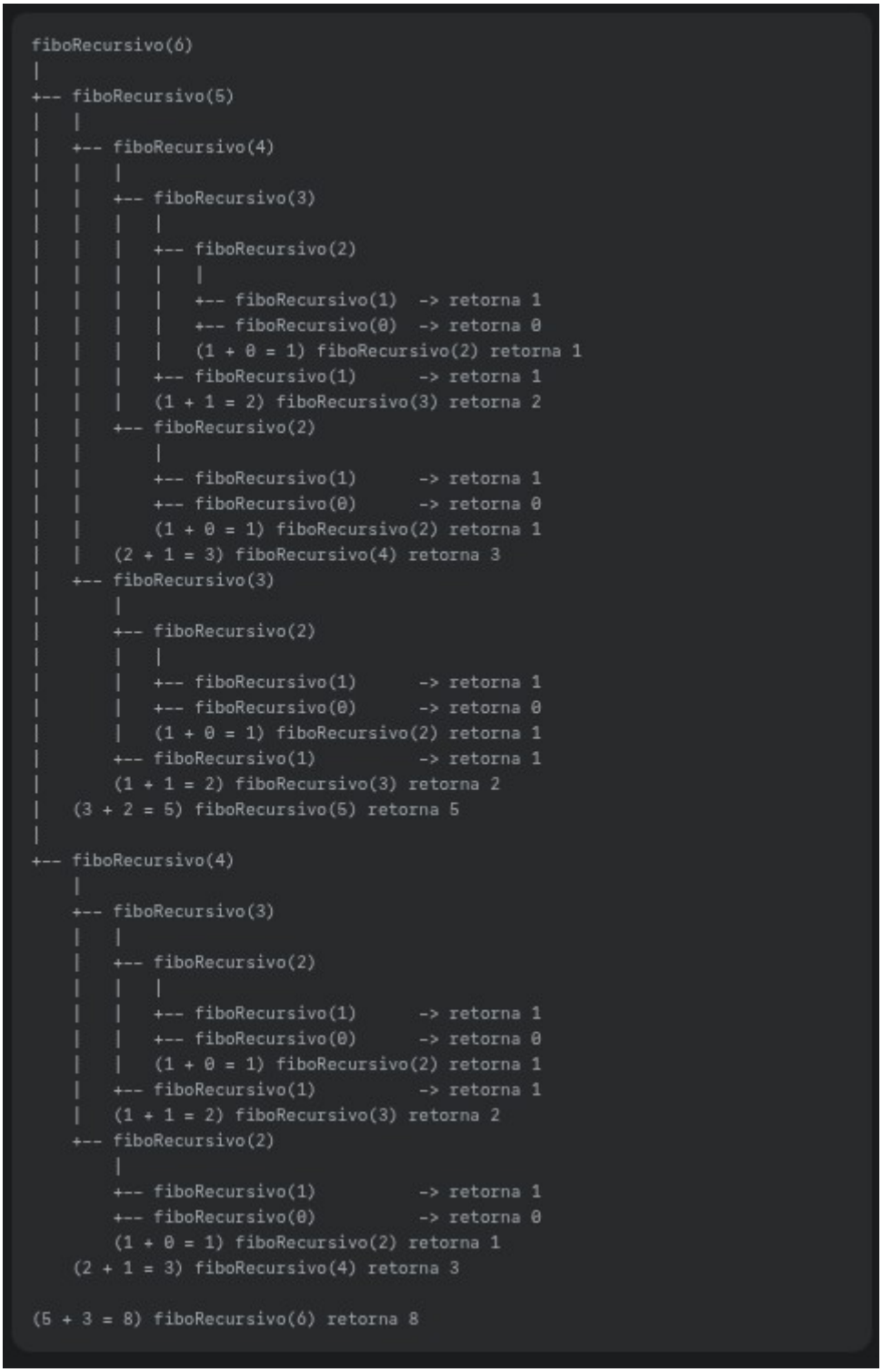
Fibonacci Recursivo – Exemplo de uma recursão

```
function fiboRecursivo(n) {
  if (n == 0) {
    return 0;
  } else if (n == 1) {
    return 1;
  } else {
    return fiboRecursivo(n - 1) + fiboRecursivo(n - 2);
  }
}

console.log(fiboRecursivo(10)); // forneço o index e ele dá o valor nesse index
```



Vamos traçar a execução principal de forma mais linear, focando nos retornos:

1. **fiboRecursivo(6) é chamado.**
    - Não é 0 nem 1. Então, calcula fiboRecursivo(5) + fiboRecursivo(4).

2. **fiboRecursivo(5) é chamado (o primeiro termo da soma).**
    - Não é 0 nem 1. Calcula fiboRecursivo(4) + fiboRecursivo(3).
    - **fiboRecursivo(4)** (para fibo(5)):
        - Calcula fiboRecursivo(3) + fiboRecursivo(2).
        - **fiboRecursivo(3)** (para fibo(4)):
            - Calcula fiboRecursivo(2) + fiboRecursivo(1).
            - **fiboRecursivo(2)** (para fibo(3)):
                - Calcula fiboRecursivo(1) + fiboRecursivo(0).
                - fiboRecursivo(1) retorna 1.
                - fiboRecursivo(0) retorna 0.
                - fiboRecursivo(2) retorna 1 + 0 = 1.
            - fiboRecursivo(1) retorna 1.
            - fiboRecursivo(3) retorna 1 + 1 = 2.
        - **fiboRecursivo(2)** (para fibo(4)): (Já sabemos que retorna 1, mas seria recalculado)
            - Retorna 1.
        - fiboRecursivo(4) retorna 2 + 1 = 3.
    - **fiboRecursivo(3)** (para fibo(5)): (Já sabemos que retorna 2, mas seria recalculado)
        - Retorna 2.
    - fiboRecursivo(5) retorna 3 + 2 = 5.
3. **fiboRecursivo(4) é chamado (o segundo termo da soma para fibo(6)).**
    - (Já sabemos que retorna 3, mas seria recalculado independentemente).
    - Não é 0 nem 1. Calcula fiboRecursivo(3) + fiboRecursivo(2).
    - **fiboRecursivo(3)** (para fibo(4)):
        - Retorna 2 (após suas próprias chamadas recursivas a fibo(2) e fibo(1)).
    - **fiboRecursivo(2)** (para fibo(4)):
        - Retorna 1 (após suas próprias chamadas recursivas a fibo(1) e fibo(0)).
    - fiboRecursivo(4) retorna 2 + 1 = 3.
4. **Finalmente, fiboRecursivo(6) pode calcular seu resultado.**
    - Recebeu 5 de fiboRecursivo(5).
    - Recebeu 3 de fiboRecursivo(4).
    - fiboRecursivo(6) retorna 5 + 3 = 8.

Portanto, console.log(fiboRecursivo(6)) imprimirá 8.

Como a Recursão Funciona nos Bastidores (Pilha de Chamadas)

Cada vez que uma função é chamada em JavaScript (e em muitas outras linguagens), um novo "quadro" (frame) é adicionado à "pilha de chamadas" (call stack). Esse quadro contém informações sobre a chamada da função, como seus parâmetros e variáveis locais.

Quando uma função recursiva se chama, um novo quadro é empilhado sobre o anterior. Quando uma função retorna, seu quadro é removido do topo da pilha, e o controle volta para a função que a chamou (que está no quadro abaixo).

No caso de fiboRecursivo(6), a pilha cresce à medida que as chamadas vão se aprofundando até atingir um caso base. Por exemplo, para calcular fiboRecursivo(6) → fiboRecursivo(5) → fiboRecursivo(4) → fiboRecursivo(3) → fiboRecursivo(2) → fiboRecursivo(1), a pilha teria 6 quadros. Quando fiboRecursivo(1) retorna 1, seu quadro é removido, e o valor 1 é usado pela chamada de fiboRecursivo(2). Esse processo de empilhar e desempilhar continua até que a chamada original (fiboRecursivo(6)) obtenha seu resultado.

Considerações sobre a Eficiência

Esta implementação recursiva de Fibonacci é muito didática para entender o conceito de recursão, mas é **ineficiente** para valores de n maiores. Observe na árvore de chamadas que muitos cálculos são repetidos. Por exemplo, fiboRecursivo(4) é calculado duas vezes, fiboRecursivo(3) é calculado três vezes, fiboRecursivo(2) é calculado cinco vezes, e assim por diante. Essa redundância leva a um crescimento exponencial no número de operações, tornando-o lento para números de Fibonacci maiores (por exemplo, fiboRecursivo(40) já demoraria bastante).

Em ciência de dados e desenvolvimento de software, quando enfrentamos problemas de desempenho com recursão devido a subproblemas sobrepostos, frequentemente usamos técnicas como:

- **Memoização (Programação Dinâmica):** Armazenamos os resultados de chamadas de função já calculadas (por exemplo, em um objeto ou array) e, antes de recalcular, verificamos se o resultado já existe.
- **Abordagem Iterativa:** Usar um loop (como for ou while) para calcular os números de Fibonacci sequencialmente, armazenando apenas os dois últimos valores necessários. Esta é geralmente a forma mais eficiente.

Conclusão

O algoritmo fiboRecursivo(n) demonstra elegantemente como a definição matemática da sequência de Fibonacci pode ser traduzida diretamente em código usando recursão. Ele se baseia em:

1. **Casos base** (n=0 e n=1) que fornecem resultados diretos e param a recursão.
2. Um **passo recursivo** que quebra o problema em subproblemas menores (fibo(n-1) e fibo(n-2)) até que os casos base sejam alcançados.

O fluxo de execução envolve múltiplas chamadas de função que são gerenciadas pela pilha de chamadas, com os resultados sendo propagados de volta à medida que as chamadas recursivas retornam. Embora seja conceitualmente simples, é importante estar ciente de suas implicações de desempenho para problemas maiores.

Espero que esta explicação didática tenha sido útil para entender o funcionamento do algoritmo!