CURSO SUPERIOR DE ADS

Recursividade

Prof. Fernando Marlon Soares Figueiredo

Disciplina: Programação Estruturada

SECTRAS

ADS

O Que É Recursividade?

A recursividade é uma técnica de programação na qual uma **função é capaz de chamar a si mesma** durante sua execução. Essa abordagem é utilizada para resolver problemas que podem ser divididos em versões menores do próprio problema original.

Em vez de utilizar estruturas de repetição, como os laços **for** e **while**, a recursão resolve um problema realizando sucessivas chamadas da mesma função até que uma condição específica determine o encerramento do processo.

A ideia central da recursividade baseia-se em um princípio muito simples: um problema complexo pode ser dividido em problemas menores e semelhantes, que são resolvidos utilizando a mesma lógica.

Esse processo continua até que seja atingida uma condição de parada, momento em que as chamadas recursivas deixam de ocorrer e os resultados começam a retornar.

O Que É Recursividade?

Características da Recursividade

- Uma função chama a si mesma durante a execução;
- O problema é reduzido progressivamente;
- Existe uma condição responsável por interromper as chamadas;
- As chamadas são armazenadas na pilha de execução (stack).

Exemplo intuitivo

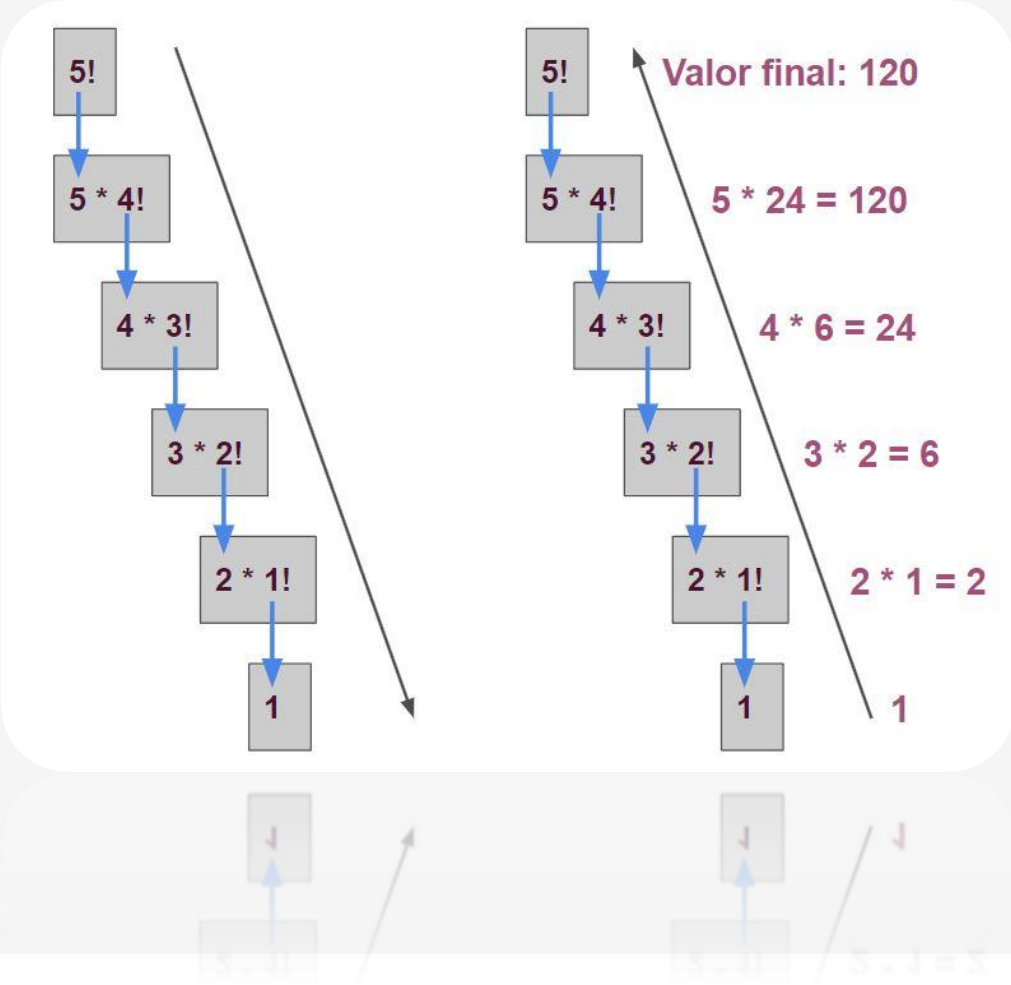
Imagine uma pessoa descendo uma escada com 100 degraus.

Para chegar ao final, ela executa sempre a mesma tarefa: descer um degrau.

Após cada passo, o problema torna-se menor, pois resta um degrau a menos para percorrer.

A recursividade segue exatamente essa lógica: resolver uma pequena parte do problema e repetir o processo até atingir o objetivo final.

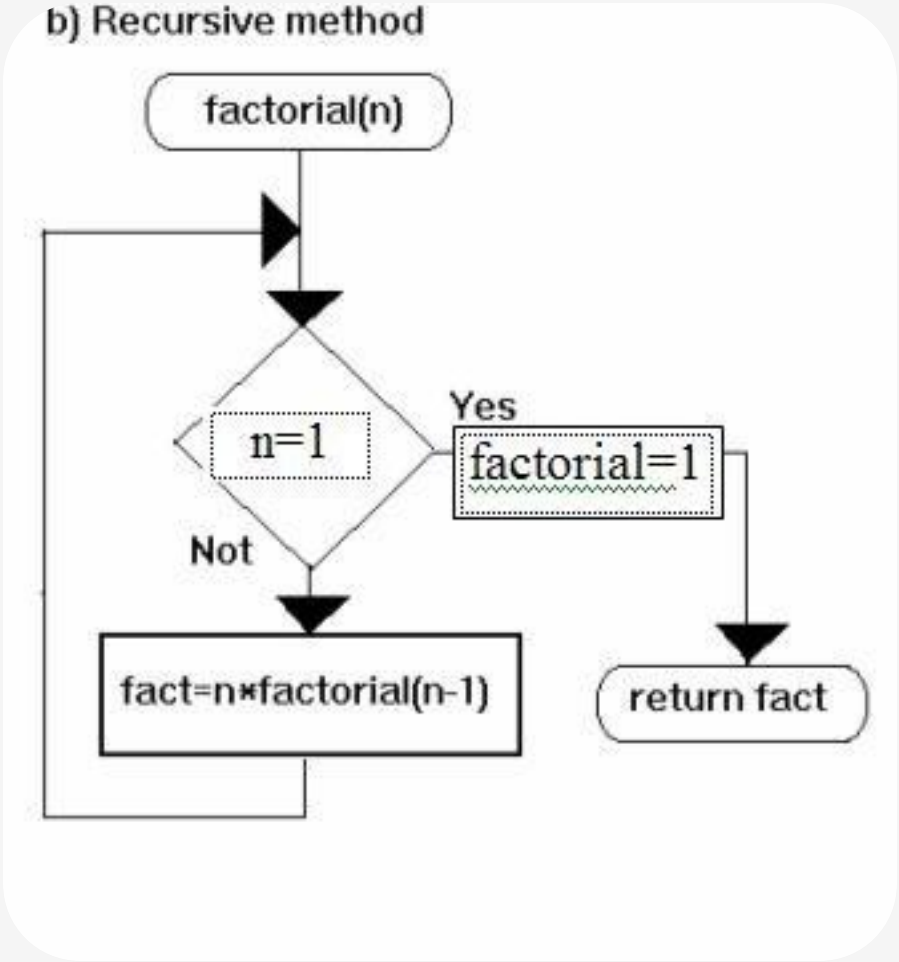
A recursão é amplamente utilizada em algoritmos de busca, ordenação, processamento de árvores, grafos e diversas estruturas hierárquicas presentes na computação.



Estrutura De Uma Função Recursiva

Para que uma função recursiva funcione corretamente, ela deve possuir uma estrutura bem definida. Sem essa estrutura, a função pode entrar em um ciclo infinito de chamadas, consumindo recursos da memória até causar falhas na execução do programa.

Independentemente da complexidade do problema, praticamente toda função recursiva é composta por dois elementos essenciais: o **caso base** e o **caso recursivo**.



Estrutura De Uma Função Recursiva

Caso Base (Condição de Parada)

O caso base é a condição responsável por interromper as chamadas recursivas.

Quando essa condição é atingida, a função deixa de chamar a si mesma e começa a retornar os resultados acumulados.

Sem um caso base, a função continuará sendo executada indefinidamente.

Caso Recursivo

O caso recursivo é a parte da função responsável por reduzir o problema e realizar uma nova chamada da própria função.

A cada chamada, o problema deve se aproximar do caso base.

teste.c

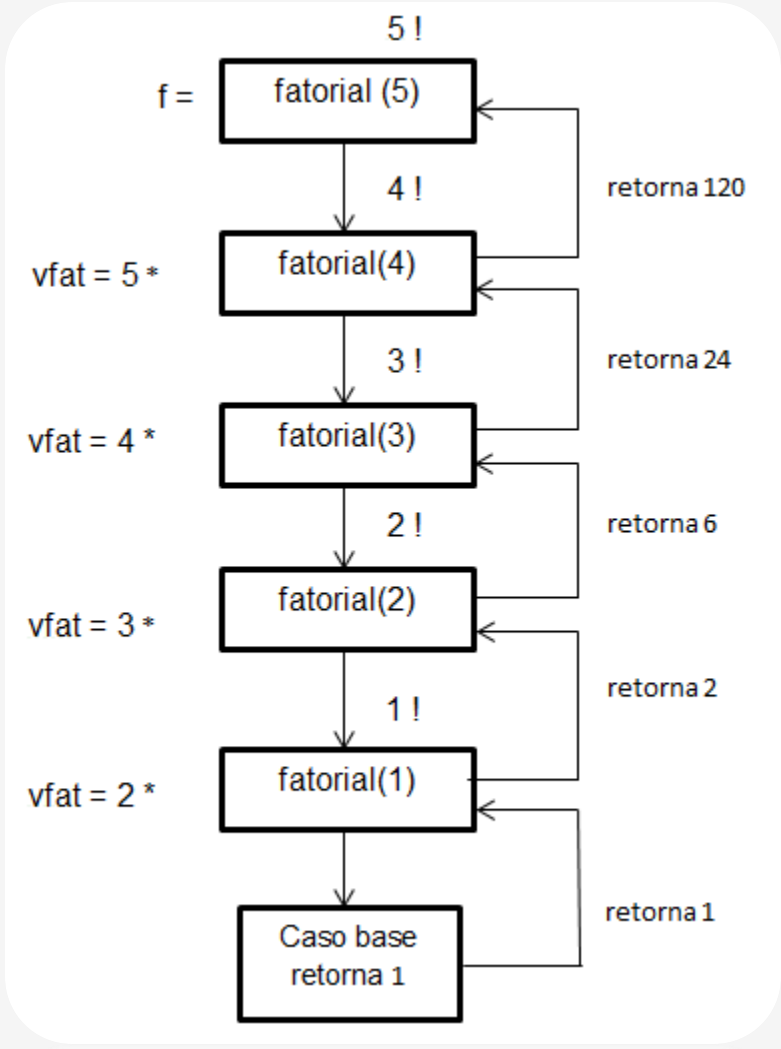
```c
tipo funcao(parametro) {

    // Caso Base
    if (condicao_de_parada) {
        return valor;
    }

    // Caso Recursivo
    return funcao(proximo_estado);
}
```

Como A Recursividade Funciona?

Quando uma função recursiva é executada, cada nova chamada da função gera uma nova instância na memória do programa. Essas chamadas ficam armazenadas temporariamente até que a condição de parada seja atingida.

Somente após alcançar o caso base é que as funções começam a retornar seus resultados, encerrando as chamadas em ordem inversa à que foram criadas.

Esse comportamento faz com que a recursividade seja diferente das estruturas de repetição tradicionais, como **for** e **while**.



Como A Recursividade Funciona?

O ciclo de execução da recursão

1. Receber o problema inicial

A função é chamada com um determinado valor ou conjunto de dados.

2. Verificar o caso base

A função verifica se já atingiu a condição de parada.

Se a resposta for positiva, a execução é encerrada.

3. Reduzir o problema

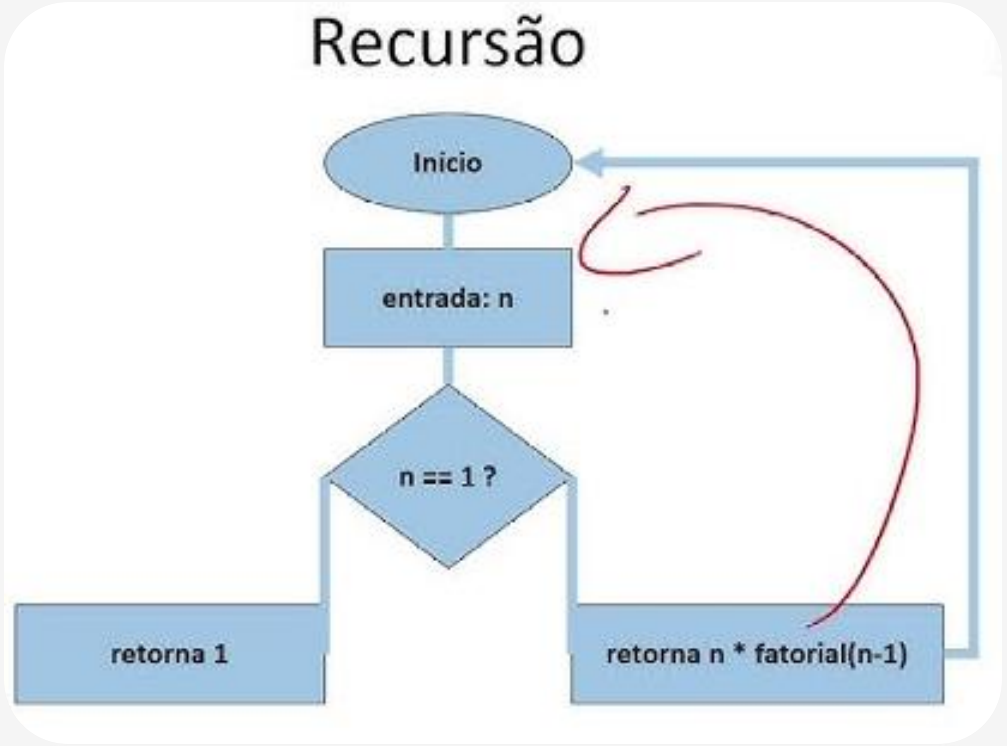
Caso o caso base ainda não tenha sido atingido, o problema é reduzido para uma versão menor.

4. Realizar nova chamada

A função chama novamente a si mesma utilizando o novo valor reduzido.

5. Retornar os resultados

Quando o caso base é alcançado, as chamadas começam a retornar até chegar à primeira função executada.



A Pilha De Execução (Stack)

Quando uma função recursiva chama a si mesma, o computador precisa armazenar informações sobre cada chamada realizada para que possa retornar corretamente após a execução.

Para isso, é utilizada uma estrutura de memória chamada **pilha de execução** (call stack).

A pilha funciona seguindo o princípio **LIFO (Last In, First Out)**, ou seja:

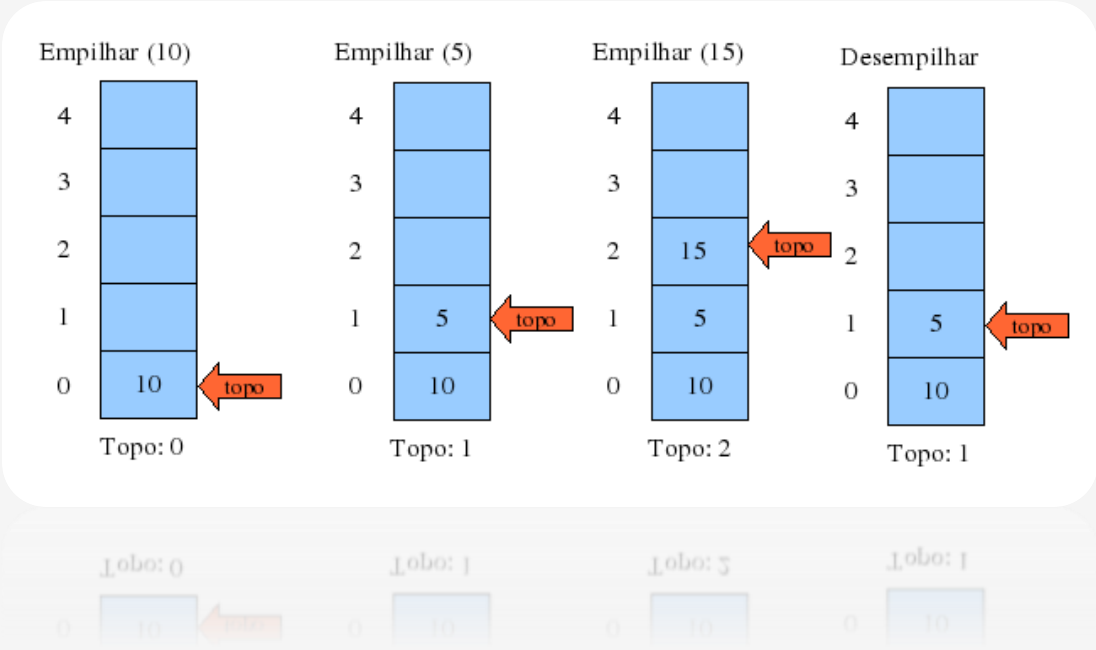
O último elemento que entra é o primeiro a sair.

O que é armazenado na pilha?

Cada vez que uma função é chamada, o sistema armazena informações importantes, como:

- Valor dos parâmetros recebidos
- Variáveis locais da função
- Endereço de retorno da execução
- Estado atual da chamada

Essas informações permanecem armazenadas até que a função seja concluída.



Exemplo: Contagem Regressiva

Lógica do problema

A função deve:

1. Exibir o valor atual;
2. Reduzir esse valor em uma unidade;
3. Chamar novamente a si mesma;
4. Encerrar quando o valor chegar a zero.

A contagem regressiva demonstra claramente os dois pilares da recursividade:

1. Existe uma condição de parada;
2. O problema é reduzido a cada chamada.

Sem a condição **n == 0**, a função continuaria chamando a si mesma indefinidamente, causando um erro de execução e consumindo toda a pilha de memória disponível.

teste.c

```c
#include <stdio.h>

void contagem(int n) {

    if (n == 0) {
        printf("Fim!\n");
        return;
    }

    printf("%d\n", n);

    contagem(n - 1);
}

int main() {

    contagem(5);

    return 0;
}
```

Exemplo: Fatorial

O cálculo do fatorial é um dos exemplos mais clássicos utilizados para introduzir a recursão. Isso acontece porque sua própria definição matemática já é naturalmente recursiva.

O fatorial de um número inteiro positivo é obtido multiplicando esse número por todos os seus antecessores até chegar ao valor 1.

Definição Matemática

O fatorial de um número n é representado por:

$$n! = n \times (n-1)!$$

com a condição especial:

$$0! = 1$$

teste.c

```c
#include <stdio.h>

int fatorial(int n) {

    if (n == 0) {
        return 1;
    }

    return n * fatorial(n - 1);
}

int main() {

    printf("%d", fatorial(5));

    return 0;
}
```

Exemplo: Série De Fibonacci

A sequência de Fibonacci é um dos exemplos mais conhecidos de aplicação da recursividade. Nessa sequência, cada termo é obtido pela soma dos dois termos anteriores.

Embora seja um excelente exemplo para demonstrar chamadas recursivas múltiplas, ele também evidencia algumas limitações da recursão quando utilizada sem otimizações.

A sequência de Fibonacci demonstra que a recursividade pode produzir soluções extremamente elegantes e próximas da definição matemática do problema.

Entretanto, nem toda solução recursiva é eficiente. Em alguns casos, a quantidade de chamadas cresce rapidamente, aumentando o tempo de execução e o consumo de memória.

Por esse motivo, é importante analisar não apenas a correção do algoritmo, mas também seu desempenho.

teste.c

```c
#include <stdio.h>

int fibonacci(int n) {

    if (n == 0)
        return 0;

    if (n == 1)
        return 1;

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

int main() {

    printf("%d", fibonacci(6));

    return 0;
}
```

Busca Binária E Recursão

A busca binária é um dos algoritmos mais eficientes para localizar elementos em uma coleção de dados ordenada. Sua principal característica é reduzir o espaço de busca pela metade a cada etapa, tornando a procura muito mais rápida do que uma busca sequencial.

Por utilizar repetidamente a mesma estratégia sobre subconjuntos menores dos dados, a busca binária é um excelente exemplo de aplicação da recursividade.

Por que a busca binária é eficiente?
Diferentemente da busca sequencial, que pode percorrer todos os elementos do vetor, a busca binária elimina metade das possibilidades a cada etapa.

Exemplo:

1.000.000 elementos

Busca Sequencial:

até 1.000.000 comparações

Busca Binária:

aproximadamente 20 comparações

teste.c

```c
#include <stdio.h>

int buscaBinaria(int vetor[], int inicio,
                 int fim, int valor) {

    if (inicio > fim)
        return -1;

    int meio = (inicio + fim) / 2;

    if (vetor[meio] == valor)
        return meio;

    if (valor < vetor[meio])
        return buscaBinaria(
            vetor, inicio, meio - 1, valor
        );

    return buscaBinaria(
        vetor, meio + 1, fim, valor
    );
}

int main() {

    int v[] = {10,20,30,40,50,60,70};

    int posicao =
        buscaBinaria(v, 0, 6, 50);

    printf("%d", posicao);

    return 0;
}
```

Recursividade Em Árvores

Uma das aplicações mais importantes da recursividade ocorre no processamento de árvores. Isso acontece porque uma árvore é uma estrutura hierárquica composta por nós que podem possuir outros nós conectados a eles.

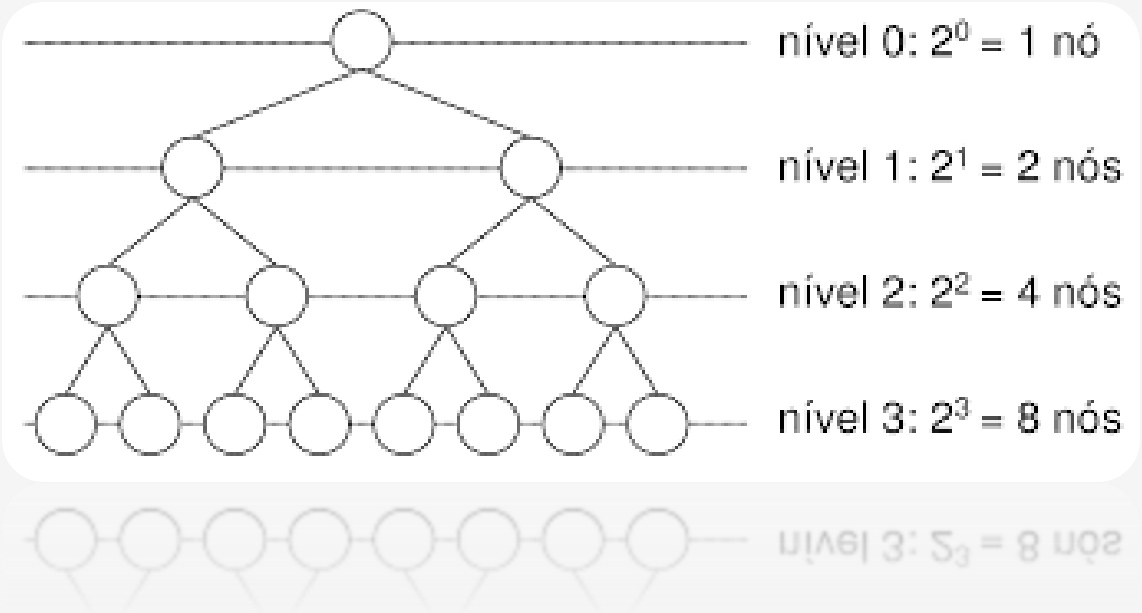
O aspecto mais interessante é que cada ramo de uma árvore pode ser visto como uma árvore menor, possuindo exatamente a mesma estrutura da árvore original.

Por esse motivo, a recursão torna-se uma solução extremamente natural para percorrer e processar esse tipo de estrutura.

Onde árvores são utilizadas?

Estruturas em árvore aparecem em diversas áreas da computação:

- Sistemas de arquivos e pastas;
- Bancos de dados;
- Compiladores;
- Motores de busca;
- Inteligência Artificial;
- Estruturas de navegação de sites.



Vantagens Da Recursividade

A recursividade é uma técnica poderosa que permite resolver determinados problemas de forma mais simples e elegante. Embora nem sempre seja a solução mais eficiente em termos de desempenho, ela pode tornar o código mais fácil de entender e mais próximo da lógica natural do problema.

Em diversas situações, a recursão oferece uma solução mais intuitiva do que abordagens baseadas em estruturas de repetição.

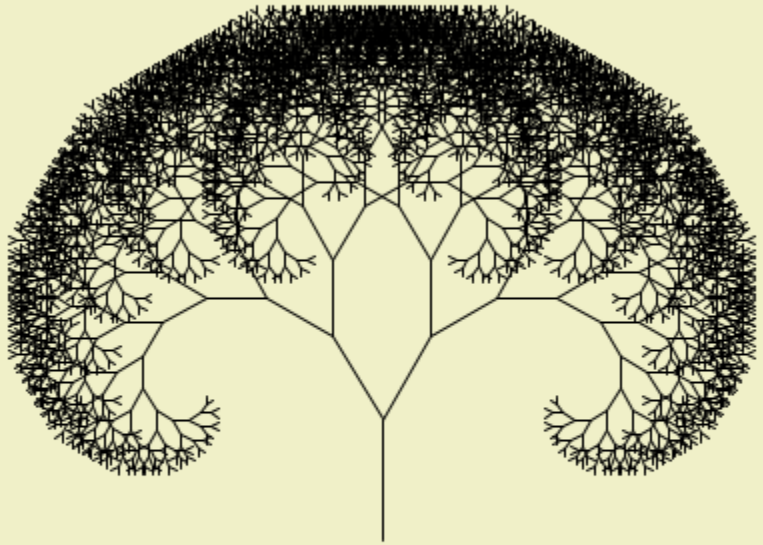
Código mais simples e elegante

Muitos problemas possuem uma definição naturalmente recursiva.

Nesses casos, a implementação utilizando recursão tende a ser menor e mais fácil de compreender.

Por exemplo:

- Fatorial
- Fibonacci
- Busca Binária
- Árvores



Vantagens Da Recursividade

Facilidade para resolver problemas hierárquicos

Problemas organizados em níveis ou ramificações costumam ser resolvidos naturalmente através da recursão.

Exemplos:

- Estruturas de diretórios
- Árvores binárias
- Grafos
- Menus hierárquicos
- Estruturas XML e JSON

Menor complexidade lógica

Em algumas situações, a solução iterativa exige:

- Variáveis auxiliares
- Pilhas manuais
- Controle adicional de estados

A recursão elimina parte dessa complexidade, tornando o algoritmo mais próximo da forma como o problema é pensado.

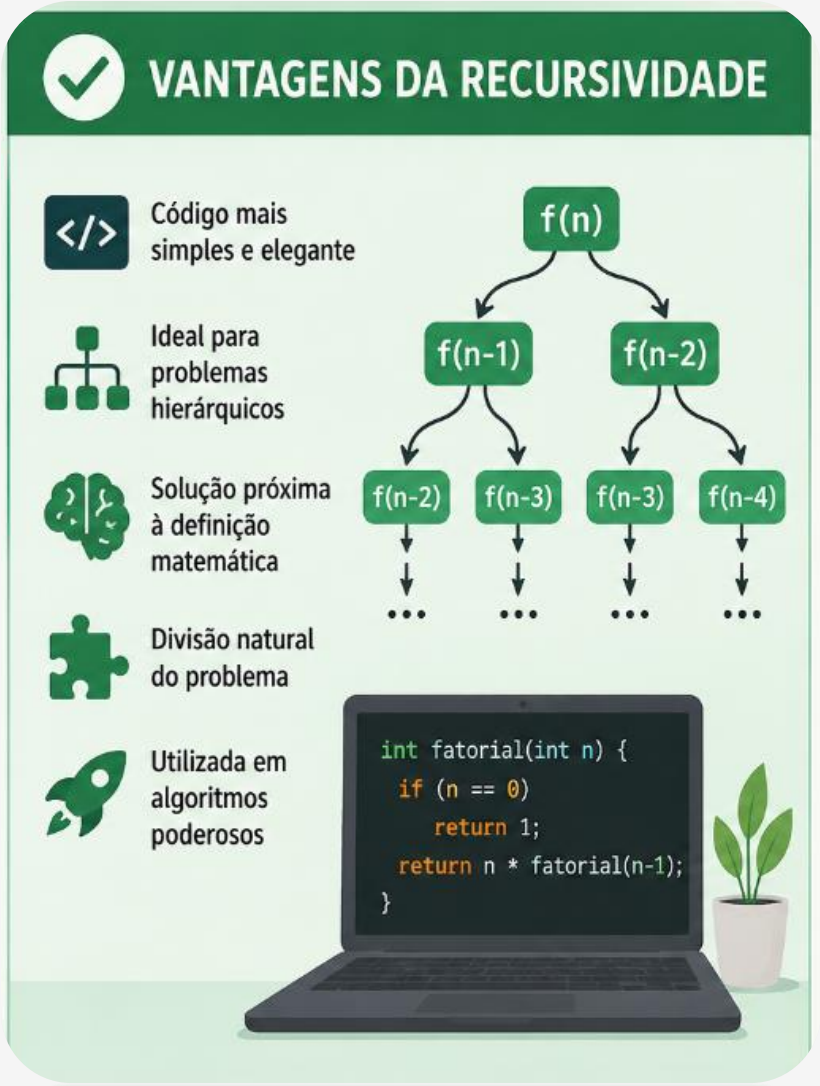
Divisão natural do problema

A recursividade segue uma estratégia conhecida como:

Dividir para Conquistar (Divide and Conquer)

O problema é dividido em partes menores, que são resolvidas individualmente até chegar à solução completa.

Essa estratégia é utilizada em diversos algoritmos importantes da computação.



Limitações E Desvantagens Da Recursividade

Maior consumo de memória

Cada chamada recursiva gera uma nova entrada na pilha de execução (stack).

Isso significa que, quanto maior a profundidade da recursão, maior será a quantidade de memória utilizada pelo programa.

Possibilidade de Stack Overflow

Se a função recursiva continuar realizando chamadas sem atingir o caso base, a pilha de execução poderá atingir seu limite de armazenamento.

Desempenho inferior em alguns casos

Existem situações em que a recursão realiza cálculos repetidos desnecessariamente.

O exemplo clássico é o Fibonacci recursivo.

Maior dificuldade de depuração

Para iniciantes, acompanhar o fluxo de chamadas e retornos pode ser mais difícil do que analisar um laço de repetição.

Erros relacionados a:

- Condições de parada;
- Parâmetros incorretos;
- Chamadas infinitas;

costumam ser mais difíceis de identificar.

Nem todo problema precisa de recursão

Em muitos casos, um simples laço resolve o problema de forma mais eficiente.