# Recursividade



# Introdução

A recursividade é um conceito fundamental na ciência da computação e na programação, representando uma técnica poderosa para resolver problemas complexos de forma elegante e concisa. Em essência, a recursividade ocorre quando uma função chama a si mesma para resolver versões menores do mesmo problema, até atingir um caso simples que pode ser resolvido diretamente. Este princípio está profundamente enraizado tanto na matemática quanto na computação, e seu domínio é essencial para qualquer programador que busque desenvolver soluções eficientes e elegantes.

Neste material, exploraremos o conceito de recursividade, seus fundamentos teóricos, implementações práticas, vantagens, limitações e aplicações. Analisaremos como a recursividade se compara com abordagens iterativas e discutiremos as considerações importantes ao implementar soluções recursivas. Embora os exemplos sejam principalmente em linguagem C, os conceitos são aplicáveis a qualquer linguagem de programação que suporte funções recursivas.

# Conceito de Chamada Recursiva

Uma função recursiva é aquela que chama a si mesma durante sua execução. Essa auto-referência permite que problemas complexos sejam decompostos em instâncias menores e mais simples do mesmo problema, seguindo o paradigma "dividir para conquistar". Cada chamada recursiva trabalha com uma versão reduzida

do problema original, aproximando-se gradualmente de um caso base que pode ser resolvido diretamente, sem recursão adicional.

## A estrutura básica de uma função recursiva inclui:

1. Caso base (ou condição de parada): Define quando a recursão deve terminar, retornando um resultado direto sem novas chamadas recursivas.
2. Caso recursivo: Define como o problema maior é decomposto em problemas menores, incluindo uma ou mais chamadas à própria função.

Exemplo simples em C - cálculo do fatorial:

```
int fatorial(int n) {
    // Caso base
    if (n == 0 || n == 1) {
        return 1;
    }
    // Caso recursivo
    else {
        return n * fatorial(n - 1);
    }
}
```

Neste exemplo, o fatorial de n (n!) é calculado como n multiplicado pelo fatorial de (n-1). O caso base ocorre quando n é 0 ou 1, onde o fatorial é definido como 1.

## Casos Base e Recursão Infinita

O caso base é o elemento mais crítico de qualquer função recursiva, pois garante que a recursão eventualmente terminará. Sem um caso base adequado, ou se o caso base nunca for alcançado, a função entrará em recursão infinita, consumindo cada vez mais memória até causar um estouro de pilha (stack overflow). Exemplo de recursão infinita (problemática):

```
int recursao_infinita(int n) {
    // Sem caso base ou caso base inalcançável
    return n + recursao_infinita(n - 1);
}
```

Esta função nunca terminará, pois não há condição que interrompa as chamadas recursivas.

### Para evitar a recursão infinita, é essencial:

1. Definir claramente o caso base.
2. Garantir que cada chamada recursiva se aproxime do caso base.
3. Verificar se o caso base será sempre alcançado eventualmente.

Exemplo corrigido:

```
int soma_ate_zero(int n) {
    // Caso base
    if (n <= 0) {
        return 0;
    }
    // Caso recursivo
    else {
        return n + soma_ate_zero(n - 1);
    }
}
```

Neste exemplo, a função soma todos os números inteiros de n até 1. O caso base (n <= 0) garante que a recursão terminará.

# Tipos de Recursão

Existem diferentes formas de implementar a recursão, cada uma com características e aplicações específicas:
1. Recursão Direta: Quando uma função chama diretamente a si mesma.
2. Recursão Indireta: Quando uma função A chama uma função B, que por sua vez chama a função A, formando um ciclo.
3. Recursão de Cauda: Quando a chamada recursiva é a última operação executada pela função, não havendo mais operações pendentes após o retorno da chamada recursiva.
A recursão de cauda é particularmente importante porque muitos compiladores podem otimizá-la, transformando-a em um loop iterativo, evitando o overhead de múltiplas chamadas de função.
4. Recursão Múltipla: Quando uma função faz múltiplas chamadas recursivas a si mesma.

```
int fibonacci(int n) {
    if (n <= 1) {
        return n;
    }
    else {
        return fibonacci(n - 1) + fibonacci(n - 2);
    }
}
```

Neste exemplo da sequência de Fibonacci, cada chamada recursiva gera duas novas chamadas, resultando em uma árvore de recursão.

# Pilha de Execução e Recursão

Para entender completamente a recursão, é essencial compreender como as chamadas de função são gerenciadas pelo sistema através da pilha de execução (call stack). Quando uma função é chamada, o sistema cria um novo registro de ativação (activation record) na pilha, contendo:

1. Parâmetros da função
2. Variáveis locais
3. Endereço de retorno (para onde o programa deve voltar após a função terminar)
4. Outros dados de controle

Cada chamada recursiva adiciona um novo registro de ativação à pilha. Quando uma função atinge seu caso base e retorna, seu registro de ativação é removido da pilha, e a execução continua a partir do ponto onde a chamada foi feita.

Exemplo de rastreamento da pilha para o cálculo de fatorial(4):

```
Chamada: fatorial(4)
  Verifica: 4 != 0 e 4 != 1
  Calcula: 4 * fatorial(3)
    Chamada: fatorial(3)
      Verifica: 3 != 0 e 3 != 1
      Calcula: 3 * fatorial(2)
        Chamada: fatorial(2)
          Verifica: 2 != 0 e 2 != 1
          Calcula: 2 * fatorial(1)
            Chamada: fatorial(1)
              Verifica: 1 == 1
              Retorna: 1
          Retorna: 2 * 1 = 2
      Retorna: 3 * 2 = 6
  Retorna: 4 * 6 = 24
Resultado final: 24
```

Este rastreamento mostra como a pilha cresce com cada chamada recursiva e depois diminui à medida que as funções retornam.

# Comparação com Soluções Iterativas

Muitos problemas que podem ser resolvidos recursivamente também podem ser abordados de forma iterativa, usando estruturas de repetição como for ou while. A escolha entre uma abordagem recursiva ou iterativa depende de vários fatores:

Vantagens da recursão:

1. Código mais limpo e elegante para certos problemas
2. Solução mais natural para problemas inerentemente recursivos
3. Facilita a implementação de algoritmos "dividir para conquistar"

Vantagens da iteração:

1. Geralmente mais eficiente em termos de memória
2. Evita o risco de estouro de pilha
3. Frequentemente mais rápida em termos de tempo de execução

Exemplo comparativo - Fatorial:

Versão recursiva:

```
int fatorial_recursivo(int n) {
    if (n == 0 || n == 1) {
        return 1;
    }
    else {
        return n * fatorial_recursivo(n - 1);
    }
}
```

Versão iterativa:

```
int fatorial_iterativo(int n) {
    int resultado = 1;
    for (int i = 2; i <= n; i++) {
        resultado *= i;
    }
    return resultado;
}
```

Exemplo comparativo - Fibonacci:

Versão recursiva (ineficiente):

```
int fibonacci_recursivo(int n) {
    if (n <= 1) {
        return n;
    }
    else {
                        return fibonacci_recursivo(n - 1) +
fibonacci_recursivo(n - 2);
    }
}
```

Versão iterativa:

```
int fibonacci_iterativo(int n) {
    if (n <= 1) {
        return n;
    }

    int a = 0, b = 1, c;
    for (int i = 2; i <= n; i++) {
        c = a + b;
        a = b;
        b = c;
    }
    return b;
}
```

A versão recursiva de Fibonacci é notoriamente ineficiente devido à duplicação de cálculos. Para n = 5, fibonacci_recursivo(5) calcula fibonacci_recursivo(3) duas vezes e fibonacci_recursivo(2) três vezes.

## Aplicações Clássicas da Recursão

A recursão é particularmente útil em diversos problemas clássicos da ciência da computação:

1. Cálculo de Fatorial
2. Sequência de Fibonacci
3. Torre de Hanói
4. Algoritmos de Busca Recursivos
5. Algoritmos de Ordenação Recursivos (Merge Sort, Quick Sort)
6. Percurso em Estruturas de Dados Recursivas
7. Problemas de Backtracking (Problema das N-Rainhas)

# Otimização de Recursão

Embora a recursão ofereça soluções elegantes, ela pode ser ineficiente em certos casos. Existem técnicas para otimizar funções recursivas:

1. Memoização: Armazenar resultados de chamadas anteriores para evitar recálculos.
    - Exemplo - Fibonacci com memoização:
2. Recursão de Cauda: Reescrever a função para que a chamada recursiva seja a última operação.
    - Exemplo - Fatorial com recursão de cauda:

Muitos compiladores modernos otimizam automaticamente a recursão de cauda, transformando-a em um loop iterativo.

3. Programação Dinâmica: Combinar memoização com uma abordagem bottom-up.

Exemplo - Fibonacci com programação dinâmica:

```
int fibonacci_dp(int n) {
    int dp[n+1];
    dp[0] = 0;
    dp[1] = 1;

    for (int i = 2; i <= n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }

    return dp[n];
}
```

# Limitações e Cuidados com Recursão

Apesar de suas vantagens, a recursão apresenta algumas limitações importantes:

1. Estouro de Pilha (Stack Overflow): Cada chamada recursiva consome espaço na pilha. Se a profundidade da recursão for muito grande, pode ocorrer um estouro de pilha.
2. Overhead de Chamadas de Função: Cada chamada recursiva envolve o custo de criar um novo registro de ativação, salvar o contexto atual e restaurá-lo posteriormente.
3. Duplicação de Cálculos: Em recursões múltiplas sem memoização, o mesmo subproblema pode ser resolvido repetidamente.

Para mitigar esses problemas:

1. Limite a profundidade da recursão.
2. Use técnicas de otimização como memoização e recursão de cauda.
3. Considere reescrever funções recursivas críticas usando abordagens iterativas.
4. Aumente o tamanho da pilha quando necessário (em ambientes que permitem isso).

# Recursão e Estruturas de Dados Recursivas

Algumas estruturas de dados são inerentemente recursivas em sua definição, tornando a recursão a abordagem natural para manipulá-las:
1. Listas Ligadas
2. Árvores
3. Grafos

# Recursão em Diferentes Paradigmas de Programação

A recursão é utilizada em diversos paradigmas de programação, com algumas diferenças importantes:
1. Programação Funcional: A recursão é um conceito central, frequentemente preferida sobre loops iterativos. Linguagens como Haskell e Scheme incentivam o uso de recursão e otimizam automaticamente a recursão de cauda.
2. Programação Orientada a Objetos: A recursão é usada principalmente para percorrer estruturas de dados hierárquicas, como árvores de componentes em interfaces gráficas.
3. Programação Imperativa: A recursão é uma ferramenta disponível, mas frequentemente substituída por abordagens iterativas por razões de desempenho.

# Boas Práticas no Uso de Recursão

Para utilizar a recursão de forma eficaz e segura:
1. Identifique claramente o caso base e garanta que ele seja alcançável.
2. Certifique-se de que cada chamada recursiva se aproxime do caso base.
3. Analise a complexidade de tempo e espaço da solução recursiva.
4. Use memoização para evitar recálculos em recursões múltiplas.
5. Considere a recursão de cauda para funções que podem ser otimizadas.
6. Teste com casos extremos para verificar limites de profundidade.
7. Documente o funcionamento da recursão para facilitar a manutenção.
8. Compare com soluções iterativas para avaliar trade-offs de desempenho.

# Conclusão

A recursão é uma técnica poderosa e elegante para resolver problemas complexos, especialmente aqueles que podem ser naturalmente decompostos em subproblemas menores e similares. Ela oferece uma abordagem intuitiva para muitos algoritmos fundamentais na ciência da computação, como ordenação, busca e percurso em estruturas de dados hierárquicas.
No entanto, a recursão não é uma solução universal. Ela vem com custos em termos de uso de memória e potencial overhead de desempenho. Em muitos casos, soluções iterativas podem ser mais eficientes, especialmente para problemas com grande volume de dados ou restrições de recursos.
O programador eficiente deve entender tanto as vantagens quanto as limitações da recursão, sabendo quando aplicá-la e como otimizá-la quando necessário. Com o conhecimento adequado das técnicas de

otimização, como memoização, recursão de cauda e programação dinâmica, é possível aproveitar a elegância da recursão sem sacrificar significativamente o desempenho.
Dominar a recursão não apenas amplia o repertório de técnicas de programação disponíveis, mas também desenvolve uma forma de pensar que facilita a abordagem de problemas complexos através da decomposição em partes mais simples - uma habilidade valiosa que transcende a programação e se aplica à resolução de problemas em geral.

# Referências


- 📖 Livros e Apostilas
  - CORMEN, T. H.; LEISERSON, C. E.; RIVEST, R. L.; STEIN, C. Algoritmos: Teoria e Prática. Elsevier.
  - SEDGEWICK, R.; WAYNE, K. Algorithms. Addison-Wesley.
  - KNUTH, D. E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. Addison-Wesley.
  - KERNIGHAN, B. W.; RITCHIE, D. M. C: A Linguagem de Programação Padrão ANSI. Campus.
  - DEITEL, H. M.; DEITEL, P. J. C: Como Programar. Pearson.
- 🌐 Recursos Online
  - IC-Unicamp - Recursão - https://ic.unicamp.br/~mc102/aulas/aula12.pdf
  - USP - Recursão - https://www.ime.usp.br/~pf/algoritmos/aulas/recu.html
  - MIT OpenCourseWare - Recursion - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-videos/lecture-6-recursion-and-dictionaries/
- 📽️ Vídeos e Cursos
  - Curso em Vídeo - Recursividade - https://www.youtube.com/watch?v=p7Wka25AcvE
  - Programação Descomplicada - Recursão - https://programacaodescomplicada.wordpress.com/2012/09/19/aula-51-recursao/
  - MIT OpenCourseWare - Recursion - https://ocw.mit.edu/courses/electrical-engineering-and-computer-science/6-0001-introduction-to-computer-science-and-programming-in-python-fall-2016/lecture-videos/lecture-6-recursion-and-dictionaries/

# RECURSIVIDADE

- DIVISÃO E CONQUISTA
  - RESOLVE PROBS. COMPLEXOS
  - VERSÕES MENORES RESOLVE PROB. ATÉ ATINGIR O CASO BASE
- ESTRUTURA:
  - CASO BASE
  - CASO RECURSIVO
  - CASO BASE: QUANDO A FUNC. CB TERMINA
  - CASO REC.: COMO O PROB. MAIOR É DECOMP. EM PROBS. MENORES
- SOLUÇÕES EFICIENTES ELEGANTES
- TIPOS DE RECURSÃO
  - DIRETA: $F_A \rightarrow F_A$
  - INDIRETA: $F_A \rightarrow F_B$
  - CAUDA: FUNÇÃO É A ÚLTIMA OP. EXECUTADA PELA FUNÇÃO
    - COMPILADORES OTIMIZAM ESSES CASOS
  - MÚLTIPLAS: $F_A \rightarrow F_{A-1} + F_{A-2}$
- REC. INFINITA
  - CB NUNCA ALCANÇADO/INADEQUADO
  - EVITAR:
    - DEFINIR CB BEM
    - CR SE APROXIMA CB
    - CB ALCANÇÁVEL
- ESTRUTURA DADOS
  - LISTAS LIGADAS, ÁRVORES, GRAFOS
  - USADA P/ PERCORRER E.D. HIERÁRQUICAS
- PILHA DE EXECUÇÃO
  - GERENCIAMENTO DAS CHAMADAS REC.
  - CRIA REG. DE ATIVAÇÃO
    - PARAM. FUNÇÃO
    - VARIÁVEIS LOCAIS
    - END. RETORNO
    - DADOS CONTROLE
- LIMITAÇÕES
  - ESTOURO DE PILHA
  - DUPLICAÇÃO DE CÁLCULOS
  - OVERHEAD DE CHAMADAS
- SOLUÇÕES
  - LIMITAR PROFUNDIDADE
  - OTIMIZAÇÕES (MEM., REC. DE CAUDA)
  - AUMENTAR O TAM. PILHA
  - USAR TÉCNICA ITERATIVA EM RECURSÕES CRÍTICAS
  - GARANTIR CB ALCANÇÁVEL
  - CB CLARO
  - CHAMADAS APROXIMAM DO CB
  - TESTE CASOS EXTREMOS P/ VERIFICAR OS LIMITES DE PROFUNDIDADE
  - ANÁLISE DE COMPLEXIDADE

Ex: FIBONACCI

- CÓDIGO LIMPO E ELEGANTE
- FACILITA IMPLEMENTAÇÃO DE ALG. DE DC

* RECURSIVO:

```
FIB(N){
    SE(N<=1){
        RETORNA N;}
    SENÃO{
        RETORNA FIB(N-1) + FIB(N-2);}
}
```

- EFICIENTE (MEM., EXEC.)
- NÃO TEM ESTOURO DE PILHA

* ITERATIVO

```
FIB(N){
    SE(N<=1){
        RETORNA N;}
    A=0;
    B=1;
    C=0;
    PARA(i=2; i<=N; i++){FAÇA{
        C=A+B;
        A=B;
        B=C;
    }
    RETORNA B;
}
```