

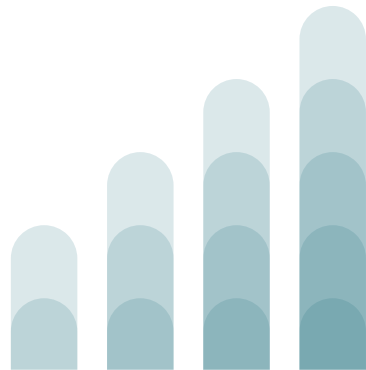
INSTITUTO FEDERAL DE
EDUCAÇÃO, CIÊNCIA E TECNOLOGIA
SÃO PAULO
Campus Campos do Jordão

ESTRUTURA DE DADOS

Recursividade

Introdução e Conceitos, Exemplos e Prática.

Professor Mestre Igor de Moraes Sampaio
igor.sampaio@ifsp.edu.br





O que é Recursividade?

- Recursividade é uma técnica de programação em que uma função chama a si mesma para resolver um problema.
- Em vez de resolver um problema de uma vez, a função **divide** o problema em partes menores (mais simples) e **resolve** essas partes recursivamente, até chegar a um caso base.
- Ela é bastante usada para simplificar a resolução de problemas que demandam muitos passos.



Estrutura de uma Função Recursiva

Uma função recursiva geralmente possui duas partes essenciais:

- Caso base (caso de parada):
 - É o momento em que a função não se chama mais, evitando um loop infinito.
 - Serve como condição de parada.
- Chamada recursiva:
 - É quando a função se chama a si mesma com um valor mais simples (reduzido) do problema.




Divisão e Conquista

- Divisão e Conquista é um paradigma de resolução de problemas que divide um problema grande em subproblemas menores e mais simples, resolve esses subproblemas (geralmente de forma recursiva) e, em seguida, combina as soluções para resolver o problema original.
- Passos do Algoritmo de Divisão e Conquista
 - Dividir: Quebrar o problema em subproblemas menores do mesmo tipo.
 - Conquistar: Resolver os subproblemas recursivamente.
 - Combinar: Juntar as soluções dos subproblemas para formar a solução final.

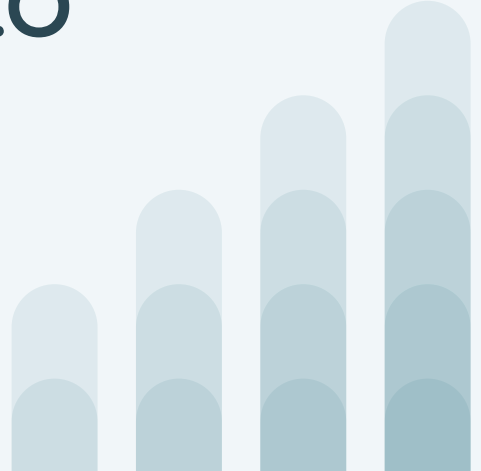


Exemplo: Cálculo do Fatorial

- O fatorial é um número natural inteiro positivo, sendo representado por $n!$, na matemática.
- O cálculo do fatorial de um número é obtido pela multiplicação desse número por todos os seus antecessores até chegar ao número 1.
- Assim: $3! = 3 * 2 * 1 = 6$
 $4! = 4 * 3 * 2 * 1 = 24$
 $5! = 5 * 4 * 3 * 2 * 1 = 120$
- Além disso, por definição: $1! = 1$, $0! = 1$ e $n! = n * (n-1)!$



Pensando de Forma Recursiva para Resolver o Cálculo do Fatorial





Exemplo: Cálculo do Fatorial

- O fatorial é um número pode ser definido de forma recursiva:
- Caso base (caso de parada):
 - $n! = 1$, se $n = 0$
- Chamada recursiva:
 - $n! = n * (n-1)!$

Exemplo em C++ de forma recursiva

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

- Em geral, uma função definida recursivamente pode ser também definida de uma forma iterativa (através de estruturas de repetição).

Comparando Iteração e Recursão

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

Característica	Iteração	Recursão
Uso de Memória	Usa menos memória	Usa pilha de chamadas
Simplicidade	Pode ser mais complexo	Pode ser mais intuitivo
Performance	Mais eficiente em alguns casos	Pode ser menos eficiente



Como Funciona a Pilha de Chamadas?

- Cada vez que uma função recursiva se chama, uma nova instância da função é empilhada na memória (chamada de "call stack").
- Quando o caso base é atingido, as chamadas começam a ser resolvidas de trás pra frente, desempilhando uma a uma até voltar para a primeira.
- Isso consome **memória**, então se não tiver um caso base ou ele for mal definido, isso pode causar **estouro de pilha** (stack overflow).



Desafio





Desafio

Crie um programa em C++ para calcular a soma dos números inteiros entre 1 e n , onde n é fornecido pelo usuário como entrada, com n maior que zero.

Crie duas funções soma (uma recursiva e a outra não recursiva) que recebe como parâmetro de entrada o número lido.



Recursividade na Matemática

- Algumas operações matemáticas ou objetos matemáticas têm uma definição recursiva
 - Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão:
 - multiplicação, divisão, exponenciação, etc...
- Isso nos permite projetar algoritmos para lidar com essas operações/objetos.



Sequência de Fibonacci

Em 1202, um matemático italiano conhecido como Fibonacci propôs o seguinte problema:

- Um coelho macho e uma fêmea nascem no início do ano.
Assumindo que:
 - Depois de dois meses de idade, o casal de coelhos produz um par misto (um macho e uma fêmea), e então outro par misto de coelhos a cada mês.
 - Não ocorrem mortes durante o ano.
- Quantos coelhos teremos no fim do ano?



Sequência de Fibonacci

Considere f_n o número de pares de coelhos no começo no mês n . Assim, temos que:

$$f_0 = 1, f_1 = 1, f_2 = 2, f_3 = 3, f_4 = 5, f_5 = 8 \dots$$

Assim, podemos observar que $f_n = f_{n-1} + f_{n-2}$. Isto significa que todos os termos da sequência (a partir de $n = 2$) são obtidos pela soma dos dois termos anteriores.

A sequência gerada a partir dessa construção, foi nomeada Sequência de Fibonacci, em homenagem ao matemático Leonardo Fibonacci, também conhecido como Leonardo de Pisa.



Sequência de Fibonacci

- Ao final de um ano (12 meses) teremos $f_{12} = 144$ casais de coelhos.
- Um problema é obter o valor f_n quando n é um número muito grande. Nesse caso a expressão não é muito viável, uma vez que precisamos calcular todos os termos anteriores a f_n .
- Como resolver essa situação?
- Definição da sequência de Fibonacci é: $f_n = f_{n-1} + f_{n-2}$
- Com os casos base: $f_0 = 0, f_1 = 1$



Desafio





Desafio

Elabore um programa recursivo, sendo dado o valor de n . No caso, equivale a descobrir o n - ésimo elemento da Sequência de Fibonacci.

Caso tenha interesse, procure saber sobre a ocorrência da Sequência de Fibonacci na natureza.

```
#include <iostream>
using namespace std;

int fibonacci(int n) {
    if (n == 0 || n == 1)
        return n;
    return fibonacci(n - 1) + fibonacci(n - 2);
}

int main() {
    int num;
    cout << "Digite um número: ";
    cin >> num;
    cout << "Fibonacci de " << num << " é " << fibonacci(num) << endl;
    return 0;
}
```



Algoritmo recursivo para sequência de fibonacci.

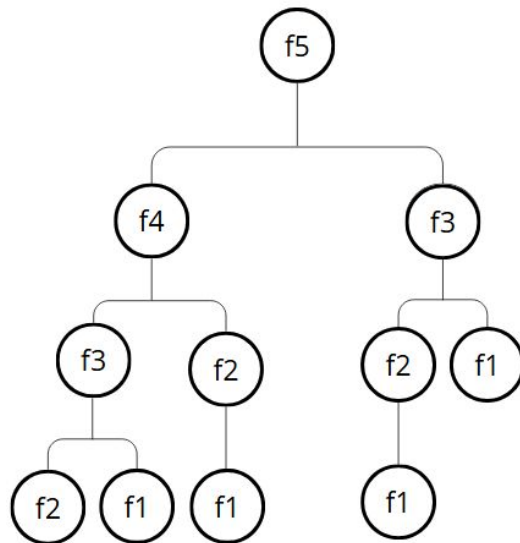


Sequência de Fibonacci

- Testar diferentes valores de entrada.
- O que acontece com fibonacci(50)?
- O problema de desempenho na recursão.
 - Fibonacci sem otimização gera muitas chamadas desnecessárias.
 - Solução: Memorização usando um dicionário.

Chamadas Redundantes na Fibonacci

O problema da recursão ingênua na Fibonacci ocorre porque a mesma subproblema é recalculado várias vezes, aumentando exponencialmente o número de chamadas. O seguinte diagrama mostra como $f(5)$ é calculado:



```
#include <iostream>
using namespace std;


Long Long memo[100] = {0}; // Array para armazenar os valores já calculados

Long Long fibonacci(int n) {
    if (n <= 1) return n; // Casos base
    if (memo[n] != 0) return memo[n]; // Se já foi calculado, retorna o valor armazenado
    return memo[n] = fibonacci(n - 1) + fibonacci(n - 2); // Calcula e armazena
}


int main() {
    cout << fibonacci(50) << endl; // Muito mais rápido!
    return 0;
}
```



Demonstração do conceito de memorização.



Para Aprofundar Seus
Conhecimentos Sobre
Recursividade



Tipos de Recursão

1 - Recursão Direta: Quando uma função chama a si mesma diretamente.

```
void exemplo() {  
    exemplo(); // chamada direta  
}
```

2 - Recursão Indireta: Quando uma função A chama B, que chama A.

```
void B(); // declaração antecipada  
  
void A() {  
    B(); // A chama B  
}  
  
void B() {  
    A(); // B chama A  
}
```




Tipos de Recursão

3 - Recursividade Linear: A função faz apenas uma chamada recursiva por vez.

```
void linear(int n) {  
    if (n == 0)  
        return;  
    cout << n << endl;  
    linear(n - 1); // chamada única  
}
```

4 - Recursividade Múltipla: A função faz mais de uma chamada recursiva.

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



Tipos de Recursão

5 - Recursividade de Cauda (Tail Recursion): A última operação da função é a chamada recursiva.

```
void contagem(int n) {  
    if (n == 0)  
        return;  
    cout << n << endl;  
    contagem(n - 1); // chamada é a última coisa que acontece  
}
```

Observação: C++ pode otimizar chamadas recursivas de cauda (tail call optimization) em alguns compiladores, mas não é garantido por padrão. Para garantir eficiência, geralmente se prefere transformações iterativas nesses casos.



Otimização de Recursão

- Recursão com Memoization (Programação Dinâmica): Evita chamadas repetidas armazenando valores já calculados.
- Tail Recursion (Recursão de Cauda): O compilador pode otimizar a recursão para evitar o empilhamento de chamadas.
- Conversão de Recursão para Iteração: Em alguns casos, um algoritmo recursivo pode ser reescrito iterativamente para melhorar a eficiência.



Problemas Clássicos Resolvidos com Recursão

- Torres de Hanói (Divide e Conquista).
- Problema do Caixeiro Viajante (Backtracking e Programação Dinâmica).
- Subconjuntos e Permutações (Backtracking).
- Solução de Sudoku (Backtracking).
- Resolver Labirintos usando DFS recursivo.
- Cálculo de fatorial.
- Sequência de Fibonacci.
- Percorrer estruturas de árvore.
- Busca binária.
- Problemas combinatórios (permutações, subconjuntos).
- etc.



Estratégias Avançadas

- Recursão Pura vs. Híbrida: Combinar recursão com iteração para otimizar desempenho.
- Técnica de Branch & Bound: Reduzir o espaço de busca em problemas combinatórios.
- Uso de técnicas probabilísticas: Como Monte Carlo e Algoritmos Genéticos que usam recursão.



Conclusão

- Vantagens da Recursividade
 - Código mais simples e elegante, facilitando a leitura.
 - Boa para problemas complexos, como árvores e grafos.
 - Divide naturalmente o problema.
 - Evita código repetitivo, tornando a solução mais limpa.



Conclusão

- Desvantagens da Recursividade
 - Alto consumo de memória, podendo causar Stack Overflow.
 - Menos eficiente em alguns casos, como Fibonacci sem otimização.
 - Difícil de depurar, devido às chamadas aninhadas.
 - Versões iterativas podem ser melhores, como no cálculo de fatorial.



Conclusão

- Quando usar?
 - Para problemas naturalmente recursivos (árvores, grafos, divisão e conquista).
 - Evitar se a recursão for muito profunda ou houver alternativas iterativas mais eficientes.



Desafio





Teste de Conhecimento

- Resolver os problemas, 1029, 2166, 1166 usando recursividade.
- Se quiser um desafio maior tente o problema 1033, recursividade não irá conseguir resolver o problema sozinho.

Beecrowd