# PCC104 - Projeto e Análise de Algoritmos

Marco Antonio M. Carvalho

Departamento de Computação
Instituto de Ciências Exatas e Biológicas
Universidade Federal de Ouro Preto

# Conteúdo

## Fonte

Este material é baseado nos livros

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, and C. Stein. *Introduction to Algorithms*. The MIT Press, 3rd edition, 2009.
- S. Halim. *Competitive Programming*. 3rd Edition, 2013.
- Ian Parberry and William Gasarch. *Problems on Algorithms*. Second Edition, 2002.
- Ian Parberry *Lecture Notes on Algorithm Analysis and Complexity Theory*. Fourth Edition, 2001.

## Licença

## Definição

Um algoritmo pode ser composto por funções, que, por sua vez, podem invocar outras funções.

Quando uma função invoca a si própria, a denominamos **função recursiva**.

É um conceito poderoso, pois define sucintamente conjuntos infinitos de instruções finitas.

A idéia é aproveitar a solução de um ou mais subproblemas com estrutura semelhante para resolver o problema original.

# Algoritmos Recursivos

## Exemplo

Um exemplo clássico de algoritmo recursivo é o cálculo de fatorial:

$0! = 1! = 1$
$n! = n \times (n-1)!$

```
long fatorial(int n)
{
    if(n<=1)
        return 1;
    return n * fatorial(n-1);
}
```

Como é a execução passo a passo para fatorial(4)?

## Projeto

Um algoritmo recursivo é composto, em sua forma mais simples, de uma **condição de parada** e de um **passo recursivo**.

## Passo Recursivo

Realiza as chamadas recursivas e processa os diferentes valores de retorno, quando adequado.

A idéia é associar um parâmetro $n$ e realizar o passo recursivo sobre $n - 1$ (ou outra fração de $n$).

## Condição de Parada

Garante que a recursividade é finita, geralmente, definida sobre um **caso base**.

Por exemplo, a condição $n \leq 1$ do exemplo anterior garante a parada com $n$ positivo (considerando o decremento unitário).

# Algoritmos Recursivos

## Análise de Complexidade do Fatorial Recursivo

Seja $T(n)$ a complexidade de tempo do fatorial recursivo:

- Caso Base: $T(1) = a$;
- Passo Recursivo: $T(n) = T(n-1) + b, n > 1$.

## Calculando...

- $T(2) = T(1) + b = a + b$
- $T(3) = T(2) + b = a + 2b$
- $T(4) = T(3) + b = a + 3b$
- Generalizando, temos que a **forma fechada** para esta recorrência é $T(n) = a + (n-1)b$.
- Logo, sendo $a$ e $b$ constantes, $T(n) = O(n)$.

## Análise de Complexidade do Fatorial Recursivo

Seja $T(n)$ a complexidade de tempo do fatorial recursivo:

- Caso Base: $T(1) = a$;
- Passo Recursivo: $T(n) = T(n-1) + b, n > 1$.

## Calculando...

- $T(2) = T(1) + b = a + b$
- $T(3) = T(2) + b = a + 2b$
- $T(4) = T(3) + b = a + 3b$
- Generalizando, temos que a **forma fechada** para esta recorrência é $T(n) = a + (n-1)b$.
- Logo, sendo $a$ e $b$ constantes, $T(n) = O(n)$.

# Algoritmos Recursivos

## Observações

Todo algoritmo recursivo possui uma versão iterativa equivalente, bastando utilizar uma pilha explícita.

Se um problema é definido em termos recursivos, a implementação via algoritmos recursivos é facilitada.

Entretanto, isto não quer dizer que esta será a melhor solução.

É necessário estar atento ao **fator de ramificação** da recursão, ou seja, quantas chamadas recursivas serão feitas por vez.

Erros de implementação fatalmente geram loop infinito ou estouro de pilha.

# Recursivo vs Iterativo

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

Complexidade $O(\phi^n)$, com $\phi = (1 + \sqrt{5})/2 = 1,61803$

```
int fibIt(int n)
{
      int i=1; fib=1; anterior=0;
      while (i < n) {
            temp = fib;
            fib = fib + anterior;
            anterior = temp;
            i++;
      }
      return fib;
}
```

Complexidade $O(n)$

# Recursivo vs Iterativo

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

Complexidade $O(\phi^n)$, com $\phi = (1+\sqrt{5})/2 = 1,61803$

```
int fibIt(int n)
{
    int i=1; fib=1; anterior=0;
    while (i < n) {
        temp = fib;
        fib = fib + anterior;
        anterior = temp;
        i++;
    }
    return fib;
}
```

Complexidade $O(n)$

## Torres de Hanói

Na versão clássica, é necessário mover $n$ discos de diâmetros diferentes, um de cada vez, de uma torre de origem para a torre de destino, usando ainda uma torre auxiliar.
Não é permitido posicionar um disco maior sobre outro menor.

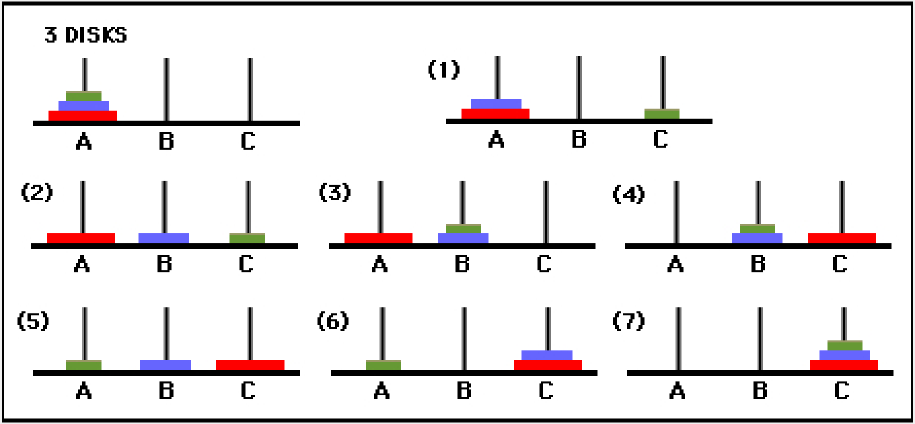
## Método de Solução

Seja $T(n)$ o número mínimo de movimentos necessários para mover todos os $n$ discos para a torre de destino de acordo com o enunciado do problema.

Por inspeção, temos que:

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 3;$
- $T(3) = 7.$

## Método de Solução $n = 3$, destino = C



## Método de Solução Generalizado

Para $n$ discos:

- Mova os $n$-1 discos do topo da origem para auxiliar ($T(n-1)$ movimentos);
- Mova o maior disco da origem para o destino (1 movimento);
- Mova os $n$-1 discos de auxiliar para o destino ($T(n-1)$ movimentos).

## *Insight*

Divida o problema original sucessivamente em subproblemas de tamanho $n$-1 e resolva recursivamente para tamanhos de $n$ crescentes.

# Algoritmos Recursivos

```
void hanoi(int n, origem, destino, aux)
{
    if(n==1)
        cout«“Mova de ”«origem«“ para ”«destino«endl;
    hanoi(n-1, origem, aux, destino);
    cout«“Mova de ”«origem«“ para ”«destino«endl;
    hanoi(n-1, aux, destino, origem);
}
```

## Complexidade de Tempo

- Caso Base: $T(1) = 1$;
- Passo Recursivo: $T(n) = 2T(n-1) + 1, n > 1$.

# Algoritmos Recursivos

```
void hanoi(int n, origem, destino, aux)
{
    if(n==1)
        cout«“Mova de ”«origem«“ para ”«destino«endl;
    hanoi(n-1, origem, aux, destino);
    cout«“Mova de ”«origem«“ para ”«destino«endl;
    hanoi(n-1, aux, destino, origem);
}
```

## Complexidade de Tempo

- Caso Base: $T(1) = 1$;
- Passo Recursivo: $T(n) = 2T(n-1) + 1, n > 1$.

## Calculando $T(n) = 2T(n-1) + 1, n \geq 1$

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 2 + 1 = 3;$
- $T(3) = 6 + 1 = 7;$
- $T(4) = 14 + 1 = 15.$

Generalizando, temos a forma fechada $T(n) = 2^n - 1$, ou seja, $T(n) = O(2^n)$.

### Calculando $T(n) = 2T(n-1) + 1, n \geq 1$

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 2 + 1 = 3;$
- $T(3) = 6 + 1 = 7;$
- $T(4) = 14 + 1 = 15.$

Generalizando, temos a forma fechada $T(n) = 2^n - 1$, ou seja, $T(n) = O(2^n)$.

## Calculando $T(n) = 2T(n-1) + 1, n \geq 1$

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 2 + 1 = 3;$
- $T(3) = 6 + 1 = 7;$
- $T(4) = 14 + 1 = 15.$

Generalizando, temos a forma fechada $T(n) = 2^n - 1$, ou seja, $T(n) = O(2^n)$.

## Calculando $T(n) = 2T(n-1) + 1, n \geq 1$

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 2 + 1 = 3;$
- $T(3) = 6 + 1 = 7;$
- $T(4) = 14 + 1 = 15.$

Generalizando, temos a forma fechada $T(n) = 2^n - 1$, ou seja, $T(n) = O(2^n)$.

## Calculando $T(n) = 2T(n-1) + 1, n \geq 1$

- $T(0) = 0;$
- $T(1) = 1;$
- $T(2) = 2 + 1 = 3;$
- $T(3) = 6 + 1 = 7;$
- $T(4) = 14 + 1 = 15.$

Generalizando, temos a forma fechada $T(n) = 2^n - 1$, ou seja, $T(n) = O(2^n)$.

## Provando a forma fechada por Indução Matemática

**Caso base**: Temos que $T(0) = 2^0 - 1 = 0$ e $T(1) = 2^1 - 1 = 1$, conforme descrito na recorrência.

**Indução**: Faremos a indução em $n$. Supomos que a forma fechada seja válida para todos os valores até $n-1$, ou seja, $T(n-1) = 2^{n-1} - 1$.

Provaremos que a forma fechada também é válida para $T(n)$:

$$\begin{aligned} T(n) &= 2T(n-1) + 1 \\ &= 2(2^{n-1} - 1) + 1 \\ &= 2^n - 2 + 1 \\ &= 2^n - 1 \end{aligned}$$

$\therefore$ a forma fechada também é válida para $n$.

## Encontrando Formas Fechadas

Podemos encontrar a forma fechada para o valor de $T(n)$ normalmente em três etapas:

1. Analisar os pequenos casos de $T(n)$, o que pode nos fornecer *insights*;
2. Encontrar e provar uma recorrência para o valor de $T(n)$;
3. Encontrar e provar uma forma fechada para a recorrência.

## Definição

Uma recorrência é uma equação ou desigualdade que descreve uma função em termos de seu próprio valor em entradas menores.

## Aplicação e Resolução

A complexidade de algoritmos recursivos pode ser frequentemente descrita através de recorrências.

Geralmente, recorremos ao **Teorema Mestre** para resolver estas recorrências.

Em casos em que o Teorema Mestre não se aplica, a recorrência deve ser resolvida de outras maneiras.

Resolver uma recorrência significa eliminar as referências que ela faz a si mesma.

Três dos métodos mais comuns para resolução de recorrências são o **método de substituição**, o **método de árvore de recursão** (ou **expansão**) e o **teorema mestre**.

## "Truques" de Resolução

Existem alguns "truques" para a resolução de recorrências:

- Procurar por algum padrão ao expandir uma recorrência, como alguma recorrência básica.
- Realizar manipulações algébricas, como troca de variáveis ou divisão da recorrência, que favoreçam a resolução.

Para tanto, é necessário ter conhecimento algébrico, de recorrências básicas e uma dose de "maldade".

## Alguns Somatórios Úteis

$$\sum_{i=1}^{n} i = \frac{n(n+1)}{2} \quad (1)$$

$$\sum_{i=0}^{k} 2^i = 2^{k+1} - 1 \quad (2)$$

$$\sum_{i=0}^{k} \frac{1}{2^i} = 2 - \frac{1}{2^k} \quad (3)$$

$$\sum_{i=0}^{k} a^i = \frac{a^{k+1} - 1}{a - 1} \quad (a \neq 1) \quad (4)$$

$$\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6} \quad (5)$$

## Exemplo 1

$T(1) = 0$
$T(n) = T\left(\frac{n}{2}\right) + 1$ (para $n \geq 2$)

Supondo que $n = 2^k \Rightarrow k = \log n$, expandimos:

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= (T(2^{k-2}) + 1) + 1 \\
&= (T(2^{k-3}) + 1) + 1 + 1 \\
&\vdots \\
&= ((T(2) + 1) + 1 + \ldots + 1 \\
&= ((T(1) + 1) + 1 + \ldots + 1 \\
&= 0 + 1 + \ldots + 1 \\
&= k
\end{aligned}
$$

$T(n) = log\ n$
$T(n) = O(log\ n)$

### Exemplo 1

$T(1) = 0$
$T(n) = T\left(\frac{n}{2}\right) + 1$ (para $n \geq 2$)

Supondo que $n = 2^k \Rightarrow k = \log n$, expandimos:

$$
\begin{aligned}
T(2^k) &= T(2^{k-1}) + 1 \\
&= (T(2^{k-2}) + 1) + 1 \\
&= (T(2^{k-3}) + 1) + 1 + 1 \\
&\vdots \\
&= ((T(2) + 1) + 1 + \ldots + 1 \\
&= ((T(1) + 1) + 1 + \ldots + 1 \\
&= 0 + 1 + \ldots + 1 \\
&= k
\end{aligned}
$$

$T(n) = log\ n$
$T(n) = O(log\ n)$

# Recorrências Básicas

## Exemplo 2 – Parte 1

$T(1) = 1$
$T(n) = 2T\left(\frac{n}{2}\right) + n$ (para $n > 1$)

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n && = 2^2T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n && = 2^3T\left(\frac{n}{2^3}\right) + 3n \\
&= 2^3\left(2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3}\right) + 3n && = 2^4T\left(\frac{n}{2^4}\right) + 4n \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + kn
\end{aligned}
$$

## Exemplo 2 – Parte 1

$T(1) = 1$
$T(n) = 2T\left(\frac{n}{2}\right) + n$ (para $n > 1$)

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n && = 2^2T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n && = 2^3T\left(\frac{n}{2^3}\right) + 3n \\
&= 2^3\left(2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3}\right) + 3n && = 2^4T\left(\frac{n}{2^4}\right) + 4n \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + kn
\end{aligned}
$$

## Exemplo 2 – Parte 1

$T(1) = 1$
$T(n) = 2T\left(\frac{n}{2}\right) + n$ (para $n > 1$)

$$
\begin{aligned}
T(n) &= 2T\left(\frac{n}{2}\right) + n \\
&= 2\left(2T\left(\frac{n}{2^2}\right) + \frac{n}{2}\right) + n && = 2^2T\left(\frac{n}{2^2}\right) + 2n \\
&= 2^2\left(2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\right) + 2n && = 2^3T\left(\frac{n}{2^3}\right) + 3n \\
&= 2^3\left(2T\left(\frac{n}{2^4}\right) + \frac{n}{2^3}\right) + 3n && = 2^4T\left(\frac{n}{2^4}\right) + 4n \\
&\vdots \\
&= 2^kT\left(\frac{n}{2^k}\right) + kn
\end{aligned}
$$

## Exemplo 2 – Parte 2

$$= 2^k T\left(\frac{n}{2^k}\right) + kn$$

Supondo que $n = 2^k \Rightarrow k$ = log$n$, temos:

$$= nT(1) + n\ log\ n$$
$$= O(n\ log\ n)$$

## Exercício

1. Proponha um algoritmo recursivo para determinar se um número é primo ou não.
2. Proponha um algoritmo iterativo para o mesmo fim.
3. Qual dos dois algoritmos possui menor complexidade de tempo?

# Dúvidas?