

# MC-202

## Revisão de recursão

Rafael C. S. Schouery  
rafael@ic.unicamp.br

Universidade Estadual de Campinas

Atualizado em: 2024-09-10 09:23

# Recursão



# Recursão



A ideia é que um problema pode ser resolvido da seguinte maneira:

# Recursão



A ideia é que um problema pode ser resolvido da seguinte maneira:

- **Primeiro**, definimos as soluções para casos básicos

# Recursão



A ideia é que um problema pode ser resolvido da seguinte maneira:

- **Primeiro**, definimos as soluções para casos básicos
- **Em seguida**, tentamos reduzir o problema para instâncias menores do problema

# Recursão



A ideia é que um problema pode ser resolvido da seguinte maneira:

- **Primeiro**, definimos as soluções para casos básicos
- **Em seguida**, tentamos reduzir o problema para instâncias menores do problema
- **Finalmente**, combinamos o resultado das instâncias menores para obter um resultado do problema original

# Genericamente

Caso base:

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente



# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema
- chama a função recursivamente

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema
- chama a função recursivamente

```
1 int fat(int n) {
```

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema
- chama a função recursivamente

```
1 int fat(int n) {
```

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema
- chama a função recursivamente

```
1 int fat(int n) {  
2     if (n == 0) /* caso base */  
3         return 1;
```

# Genericamente

Caso base:

- resolve **instâncias pequenas** diretamente

Caso geral:

- reduz o problema para **instâncias menores** do mesmo problema
- chama a função recursivamente

```
1 int fat(int n) {
2     if (n == 0) /* caso base */
3         return 1;
4     else /* caso geral */
5         return n * fat(n - 1); /* instância menor */
6 }
```

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva



## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos têm uma definição recursiva

- Ex: fatorial, sequência de Fibonacci, palíndromos, etc...
- ou podem ser vistos do ponto de vista da recursão

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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...

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

**Ex:** Exponenciação

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

**Ex:** Exponenciação

Seja  $a$  é um número real e  $b$  é um número inteiro não-negativo

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

**Ex:** Exponenciação

Seja  $a$  é um número real e  $b$  é um número inteiro não-negativo

- Se  $b = 0$ , então  $a^b = 1$

## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

**Ex:** Exponenciação

Seja  $a$  é um número real e  $b$  é um número inteiro não-negativo

- Se  $b = 0$ , então  $a^b = 1$
- Se  $b > 0$ , então  $a^b = a \cdot a^{b-1}$



## Definições recursivas

Algumas operações matemáticas ou objetos matemáticos 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

**Ex:** Exponenciação

Seja  $a$  é um número real e  $b$  é um número inteiro não-negativo

- Se  $b = 0$ , então  $a^b = 1$
- Se  $b > 0$ , então  $a^b = a \cdot a^{b-1}$

```
1 double potencia(double a, int b) {
2     if (b == 0)
3         return 1;
4     else
5         return a * potencia(a, b - 1);
6 }
```

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  *$ap\alpha$*  onde

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  $ap\alpha$  onde
  - $\alpha$  é uma letra



# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  $ap\alpha$  onde
  - $\alpha$  é uma letra
  - $p$  é um palíndromo

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  *$ap\alpha$*  onde
  - *$\alpha$*  é uma letra
  - *$p$*  é um palíndromo

```
1 int eh_palindromo(char *palavra, int ini, int fim) {  
2     if (ini >= fim)  
3         return 1;
```

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  *$ap\alpha$*  onde
  - *$\alpha$*  é uma letra
  - *$p$*  é um palíndromo

```
1 int eh_palindromo(char *palavra, int ini, int fim) {  
2     if (ini >= fim)  
3         return 1;
```

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  *$ap\alpha$*  onde
  - *$\alpha$*  é uma letra
  - *$p$*  é um palíndromo

```
1 int eh_palindromo(char *palavra, int ini, int fim) {
2     if (ini >= fim)
3         return 1;
4     return palavra[ini] == palavra[fim] &&
5         eh_palindromo(palavra, ini + 1, fim - 1);
6 }
```

# Palíndromos

Uma palavra é um **palíndromo** se ela é igual ao seu reverso

- Ex: ana, ovo, osso, radar

Matematicamente, uma palavra é palíndromo se:

- ou tem zero letras (palavra vazia)
- ou tem uma letra
- ou é da forma  *$ap\alpha$*  onde
  - *$\alpha$*  é uma letra
  - *$p$*  é um palíndromo

```
1 int eh_palindromo(char *palavra, int ini, int fim) {
2     if (ini >= fim)
3         return 1;
4     return palavra[ini] == palavra[fim] &&
5         eh_palindromo(palavra, ini + 1, fim - 1);
6 }
7
8 eh_palindromo(palavra, 0, strlen(palavra) - 1);
```

# Busca Binária

Para buscar  $x$  no vetor ordenado **dados** entre as posições  $l$  e  $r$

# Busca Binária

Para buscar  $x$  no vetor ordenado **dados** entre as posições  $l$  e  $r$

Casos base:

# Busca Binária

Para buscar  $x$  no vetor ordenado  $dados$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor



# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

## Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

## Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {  
2     int m = (l + r)/2;
```



# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {  
2     int m = (l + r)/2;
```

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

## Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

## Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r)/2;
3     if (l > r)
4         return -1;
```

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

## Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

## Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r)/2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
```

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r)/2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
```

# Busca Binária

Para buscar  $x$  no vetor ordenado  $\text{dados}$  entre as posições  $l$  e  $r$

Casos base:

- Se o intervalo for vazio ( $l > r$ ),  $x$  não está no vetor
- Se  $\text{dados}[m] == x$ , onde  $m = (l + r)/2$ 
  - Devolvemos  $m$

Caso geral:

- Se  $\text{dados}[m] < x$ , então  $x$  só pode estar entre  $m + 1$  e  $r$ 
  - Devolvemos o resultado da chamada recursiva
- Se  $\text{dados}[m] > x$ , então  $x$  só pode estar entre  $l$  e  $m - 1$ 
  - Devolvemos o resultado da chamada recursiva

```
1 int busca_binaria(int *dados, int l, int r, int x) {
2     int m = (l + r)/2;
3     if (l > r)
4         return -1;
5     if (dados[m] == x)
6         return m;
7     else if (dados[m] < x)
8         return busca_binaria(dados, m + 1, r, x);
9     else
10        return busca_binaria(dados, l, m - 1, x);
11 }
```

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar



# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Isso sempre é possível? Quando for possível, sempre melhora a eficiência do algoritmo?

# Comparando recursão e algoritmos iterativos

Normalmente algoritmos recursivos são:

- mais simples de entender
- menores e mais fáceis de programar
- mais “elegantes”

Mas algumas vezes podem ser

- **muito** ineficientes (quando comparados a algoritmos iterativos para o mesmo problema)

Estratégia ideal:

1. encontrar algoritmo recursivo para o problema
2. reescrevê-lo como um algoritmo iterativo

Isso sempre é possível? Quando for possível, sempre melhora a eficiência do algoritmo?

- Veremos mais sobre isso no curso...



# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci:

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1,

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2,

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3,

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5,

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8,

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13,

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
```



# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {  
2     if (n == 1)  
3         return 1;
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {  
2     if (n == 1)  
3         return 1;  
4     else if (n == 2)  
5         return 1;
```

## Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2   if (n == 1)
3     return 1;
4   else if (n == 2)
5     return 1;
6   else
7     return fib_rec(n-2) +
8           fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2   int ant, atual, prox, i;
3   ant = atual = 1;
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = atual = 1;
4     for (i = 3; i <= n; i++) {
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = atual = 1;
4     for (i = 3; i <= n; i++) {
5         prox = ant + atual;
```



# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = atual = 1;
4     for (i = 3; i <= n; i++) {
5         prox = ant + atual;
6         ant = atual;
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = atual = 1;
4     for (i = 3; i <= n; i++) {
5         prox = ant + atual;
6         ant = atual;
7         atual = prox;
8     }
9 }
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2   if (n == 1)
3     return 1;
4   else if (n == 2)
5     return 1;
6   else
7     return fib_rec(n-2) +
8           fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2   int ant, atual, prox, i;
3   ant = atual = 1;
4   for (i = 3; i <= n; i++) {
5     prox = ant + atual;
6     ant = atual;
7     atual = prox;
8   }
9   return atual;
10 }
```

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

```
1 int fib_rec(int n) {
2     if (n == 1)
3         return 1;
4     else if (n == 2)
5         return 1;
6     else
7         return fib_rec(n-2) +
8             fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2     int ant, atual, prox, i;
3     ant = atual = 1;
4     for (i = 3; i <= n; i++) {
5         prox = ant + atual;
6         ant = atual;
7         atual = prox;
8     }
9     return atual;
10 }
```

Número de operações:

# Fibonacci: recursivo vs. iterativo

Sequência de Fibonacci: 1, 1, 2, 3, 5, 8, 13, ...

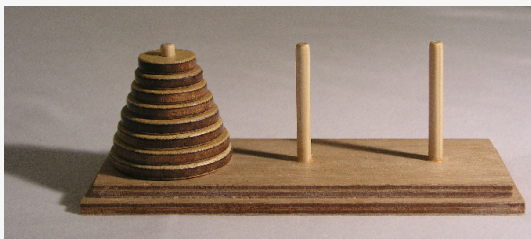
```
1 int fib_rec(int n) {
2   if (n == 1)
3     return 1;
4   else if (n == 2)
5     return 1;
6   else
7     return fib_rec(n-2) +
8           fib_rec(n-1);
9 }
```

```
1 int fib_iterativo(int n) {
2   int ant, atual, prox, i;
3   ant = atual = 1;
4   for (i = 3; i <= n; i++) {
5     prox = ant + atual;
6     ant = atual;
7     atual = prox;
8   }
9   return atual;
10 }
```

Número de operações:

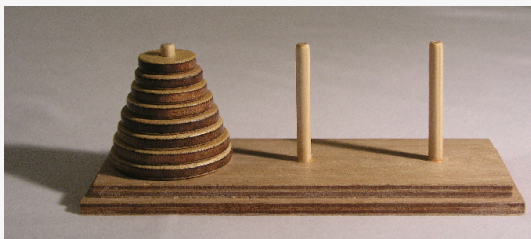
- iterativo:  $\approx n$
- recursivo:  $\approx \text{fib}(n)$  (aproximadamente  $1.6^n$ )

# Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

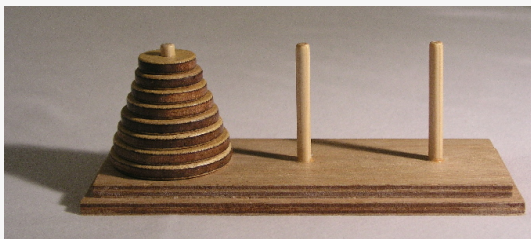
# Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

Objetivo:

# Torres de Hanói



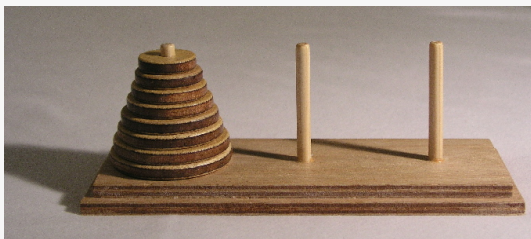
A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*



# Torres de Hanói



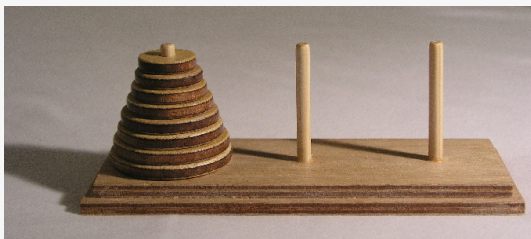
A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*

Regras:

# Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

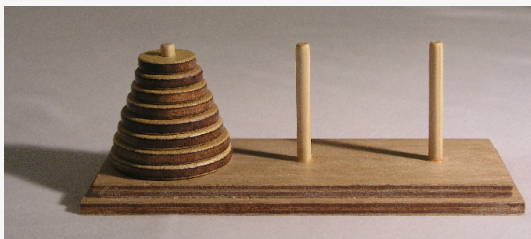
Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*

Regras:

- Apenas um disco pode ser movido de cada vez

# Torres de Hanói



A torre de Hanói é um brinquedo com três estacas *A*, *B* e *C* e discos de tamanhos diferentes

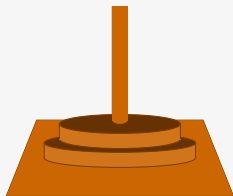
Objetivo:

- mover todos os discos da estaca *A* para a estaca *C*

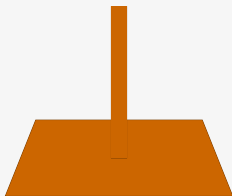
Regras:

- Apenas um disco pode ser movido de cada vez
- Um disco maior não pode ser colocado sobre um menor

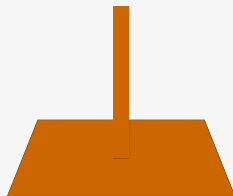
# Torres de Hanói recursivo



orig

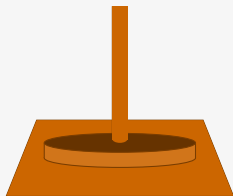


aux

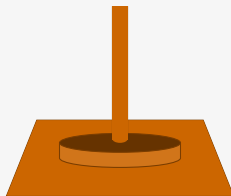


dest

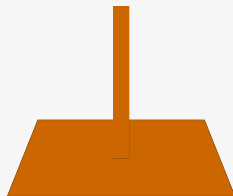
# Torres de Hanói recursivo



orig

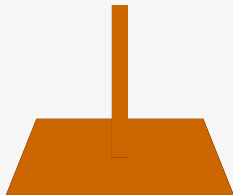


aux

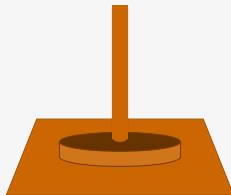


dest

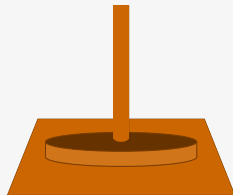
# Torres de Hanói recursivo



orig

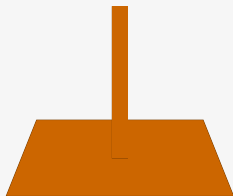


aux

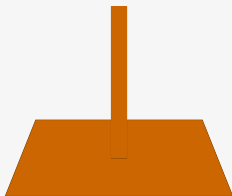


dest

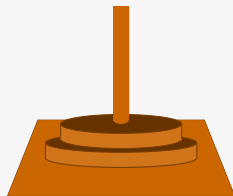
# Torres de Hanói recursivo



orig

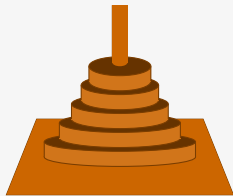


aux

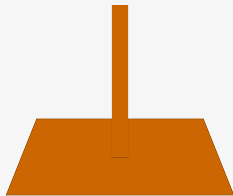


dest

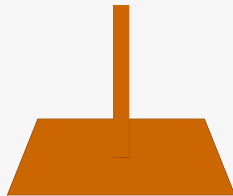
# Torres de Hanói recursivo



orig



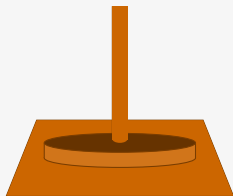
aux



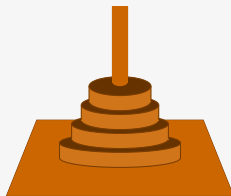
dest



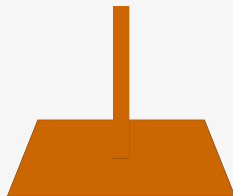
# Torres de Hanói recursivo



orig

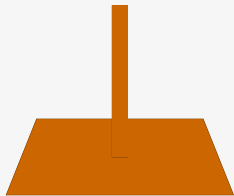


aux

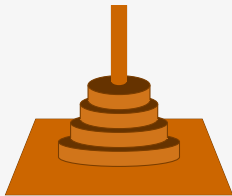


dest

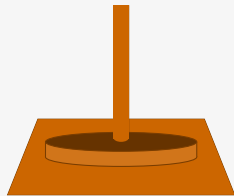
# Torres de Hanói recursivo



orig

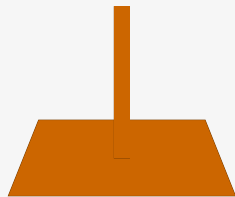


aux

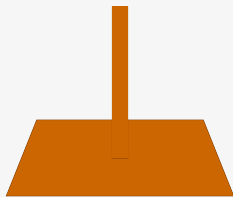


dest

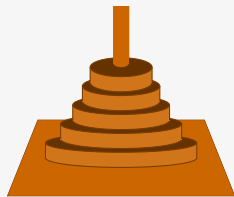
# Torres de Hanói recursivo



orig

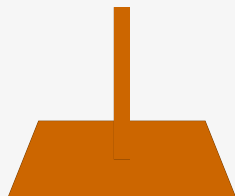


aux

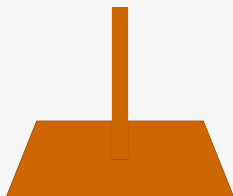


dest

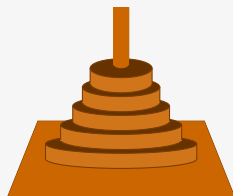
# Torres de Hanói recursivo



orig



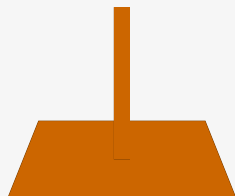
aux



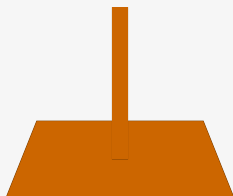
dest

```
1 void hanoi(int n, char orig, char dest, char aux) {  
2     /* caso base: n == 0 - não faz nada */  
3     if (n > 0) { /* caso geral */  
4         hanoi(n - 1, orig, aux, dest);  
5         printf("move de %c para %c\n", orig, dest);  
6         hanoi(n - 1, aux, dest, orig);  
7     }  
8 }
```

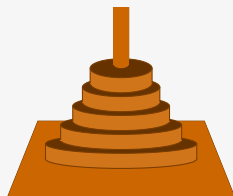
# Torres de Hanói recursivo



orig



aux



dest

```
1 void hanoi(int n, char orig, char dest, char aux) {
2     /* caso base: n == 0 - não faz nada */
3     if (n > 0) { /* caso geral */
4         hanoi(n - 1, orig, aux, dest);
5         printf("move de %c para %c\n", orig, dest);
6         hanoi(n - 1, aux, dest, orig);
7     }
8 }
```

Chamada da função: `hanoi(n, 'a', 'c', 'b');`

## Exercício — Calculando o Máximo

Escreva uma função recursiva que calcule o máximo de um vetor dado com  $n$  elementos

```
int maximo(int *v, int n)
```

# Solução

```
1 int maximo(int *v, int n) {
2     if (n == 1)
3         return v[0];
4     int m = maximo(v, n - 1);
5     return v[n - 1] > m ? v[n - 1] : m;
6 }
```

## Exercício — Coeficientes Binomiais

Escreva uma função recursiva que calcule, para  $n \geq 0$  e  $k \geq 0$

$$\binom{n}{k}$$

Relação de Stifel:

$$\binom{n-1}{k-1} + \binom{n-1}{k} = \binom{n}{k}$$



# Solução

```
1 int binomio(int n, int k) {  
2     if (k == 0 || k == n)  
3         return 1;  
4     return binomio(n - 1, k) + binomio(n - 1, k - 1);  
5 }
```

Dúvidas?