Algoritmos Recursivos: Parte 1

Algoritmos e Programação 2

Prof. Dr. Anderson Bessa da Costa
Universidade Federal de Mato Grosso do Sul

Função Fatorial

- $n!$ ($n > 0$) é o produto de todos inteiros entre n e 1
  - Por exemplo, $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$
  - O fatorial de 0 é definido como 1

Função Fatorial: Definição

```
n! = 1 se n == 0
n! = n * (n - 1) * (n - 2) * … * 1 se n > 0
```

- Os três pontos são abreviatura para todos números entre $n - 3$ e 2 multiplicados

Função Fatorial: Listagem

- Para evitar abreviatura, precisaríamos listar uma fórmula para $n!$ para cada valor de $n!$:

```
0! = 1
1! = 1
2! = 2 * 1
3! = 3 * 2 * 1
4! = 4 * 3 * 2 * 1
…
```

- Evidentemente, não esperamos listar uma fórmula para o fatorial de cada inteiro ..

Função Fatorial: Algoritmo

- Para evitar qualquer abreviatura e conjunto infinito de definições, podemos representar um **algoritmo** que aceite um inteiro n e retorne o valor de $n!$

Algoritmo Fatorial Iterativo

```
int fatorial(int n) {
    int x, prod;

    prod = 1;
    for (x = n; x > 0; x--){
        prod = prod * x;
    }
    return prod;
}
```

- Esse algoritmo é chamado **iterativo**
    - Requer a repetição de um processo até que determinada condição seja satisfeita

Função Fatorial: Notação Matemática

- Examinaremos a definição de $n!$ que lista uma fórmula separada para cada valor de n

```
0! = 1
1! = 1 * 0!
2! = 2 * 1!
3! = 3 * 2!
4! = 4 * 3!
…
```

ou, empregando a notação matemática usada anteriormente.

$$n! = \begin{cases} 1 & \text{se } n = 0 \text{ (âncora ou caso base)} \\ n \cdot (n-1)! & \text{se } n > 0 \text{ (passo de indução)} \end{cases}$$

Função Fatorial: Definição Recursiva

- Definimos a função fatorial em termos de si mesma
- Parece uma definição circular .. mas não é
- Tal definição, que define um objeto em termos de um caso mais simples de si mesmo, é chamado **definição recursiva**

Exemplo 4!

- Examinaremos como a definição recursiva da função fatorial pode ser usada para avaliar $4!$

```
1. 4! = 4 * 3!
2.          3! = 3 * 2!
3.                   2! = 2 * 1!
4.                            1! = 1 * 0!
5.                                     0! = 1
```

- Cada caso é reduzido a um caso mais simples até chegarmos ao caso de $0!$, que é definido imediatamente como 1

Exemplo 4! (cont.)

- Na linha 5, temos um valor que é definido diretamente
- Podemos, voltar da linha 5 até linha 1, retornando valor calculado em uma linha para avaliar o resultado da linha anterior

```
5’ 0! = 1
4’ 1! = 1 * 0! = 1 * 1 = 1
3’ 2! = 2 * 1! = 2 * 1 = 2
2’ 3! = 3 * 2! = 3 * 2 = 6
1’ 4! = 4 * 3! = 4 * 6 = 24
```

Algoritmo Fatorial

```c
int fatorial(int n) {
    int x, y, fact;

    if (n == 0) {
        fact = 1;
    }
    else {
        x = n - 1;
        // ache o valor de x!. Chame-o de y
        fact = n * y;
    }
    return fact;
}
```

Algoritmo Fatorial Recursivo

```
int fatorial(int n) {
    int x, y, fact;

    if (n == 0) {
        fact = 1;
    }
    else {
        x = n - 1;
        // ache o valor de x!. Chame-o de y
        y = fatorial(x);
        fact = n * y;
    }
    return fact;
}
```

Fatorial(4)

```
fatorial(4)
```

```
int fatorial(int n=0) {
  int fact, x, y;

  if (n == 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x);
    fact = n * y;
  }
  return fact;
}
```

Fatorial(4): Retorno

```
fatorial(4)
```

```
int fatorial(int n=0) {
  int fact, x, y;

  if (n == 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x);
    fact = n * y;
  }
  return fact=1;
}
```

Fatorial(4): Retorno

```
fatorial(4)
```

```
int fatorial(int n=1) {
  int fact, x, y;

  if (n = 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x)=1;
    fact = n * y;
  }
  return fact=1;
}
```

Fatorial(4): Retorno

```
fatorial(4)
```

```
int fatorial(int n=2) {
  int fact, x, y;

  if (n = 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x)=1;
    fact = n * y;
  }
  return fact=2;
}
```

Fatorial(4): Retorno

fatorial(4)

```
int fatorial(int n=3) {
  int fact, x, y;

  if (n = 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x)=2;
    fact = n * y;
  }
  return fact=6;
}
```

Fatorial(4): Retorno

```
fatorial(4)
```

```
int fatorial(int n=4) {
  int fact, x, y;

  if (n = 0) {
    fact = 1;
  }
  else {
    x = n - 1;
    // ache o valor de x!. Chame-o de y
    y = fatorial(x)=6;
    fact = n * y;
  }
  return fact=24;
}
```

Pilha Durante a Execução

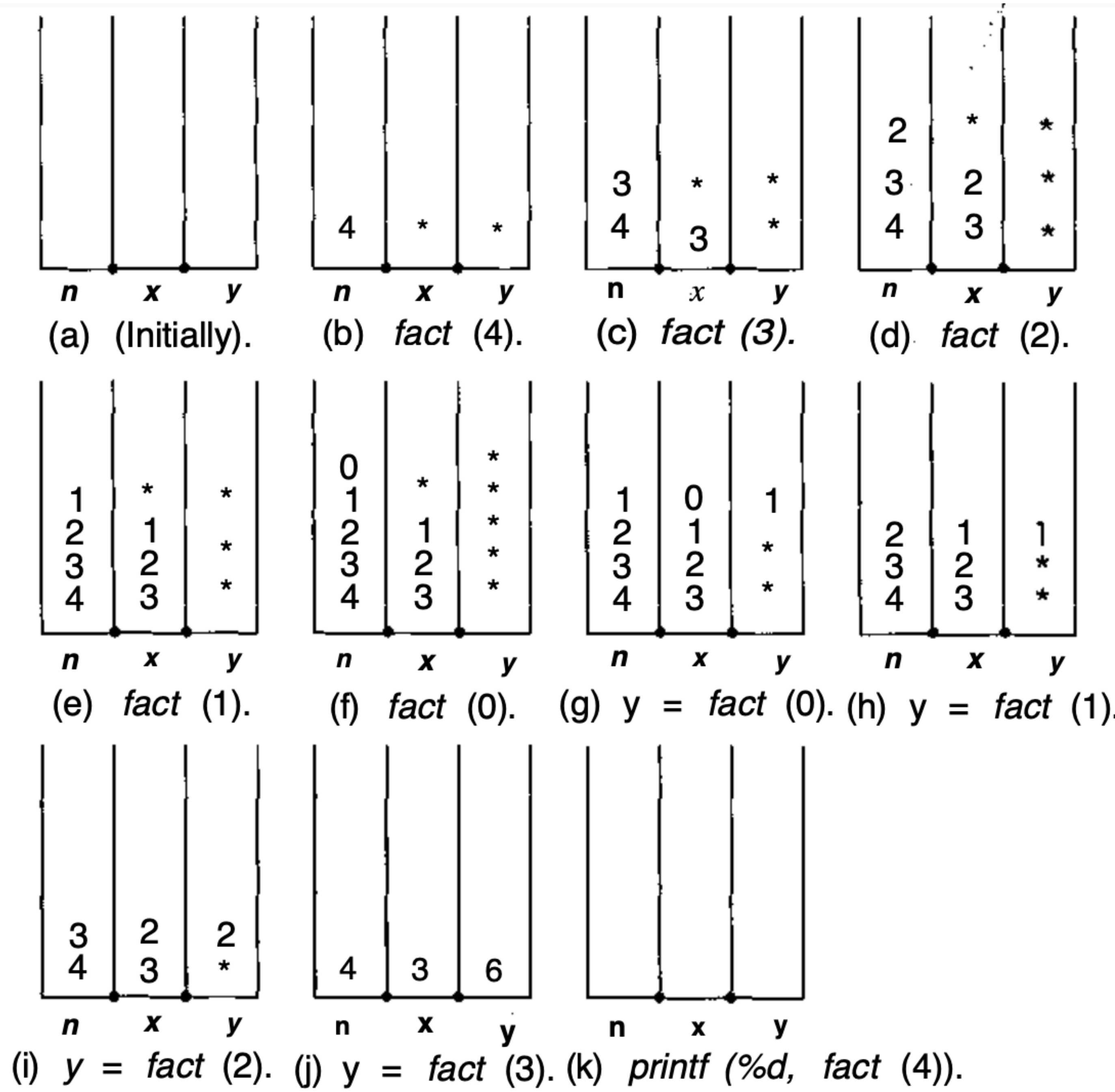


Figura 3.2.1 A pilha, em vários instantes, durante a execução. (Um asterisco indica um valor não-inicializado.)

Multiplicação de Números Naturais

Multiplicação de Números Naturais

- O produto $a \cdot b$, em que a e b são inteiros positivos, pode ser definido como a somado a si mesmo b vezes. Essa é uma definição iterativa
- Uma definição recursiva equivalente é

```
a * b = a se b == 1
a * b = a * (b - 1) + a se b > 1
```

$$prod(a,b) = \begin{cases} a & \text{se } b = 1 \\ prod(a, b-1) + a & \text{se } b > 1 \end{cases}$$

Definições Recursivas

- Observe um padrão:
  - Um **caso simples** é estabelecido explicitamente
  - Os outros casos são definidos aplicando uma operação sobre o resultado da **avaliação de um caso mais simples**

Sequência de Fibonacci

Sequência de Fibonacci

- A sequência de Fibonacci é a sequência de inteiros:

  0, 1, 1, 2, 3, 5, 8, 13, 21, 34 …

- Elemento é a soma dos dois anteriores
  - Por exemplo, 0 + 1 = 1, 1 + 1 = 2, 1 + 2 = 3, 2 + 3 = 5, …

Definição Recursiva Fibonacci

- Se permitirmos que $fib(0) = 0$ e $fib(1) = 1$, e assim por diante, então poderemos definir a sequência de Fibonacci por meio da seguinte definição recursiva:

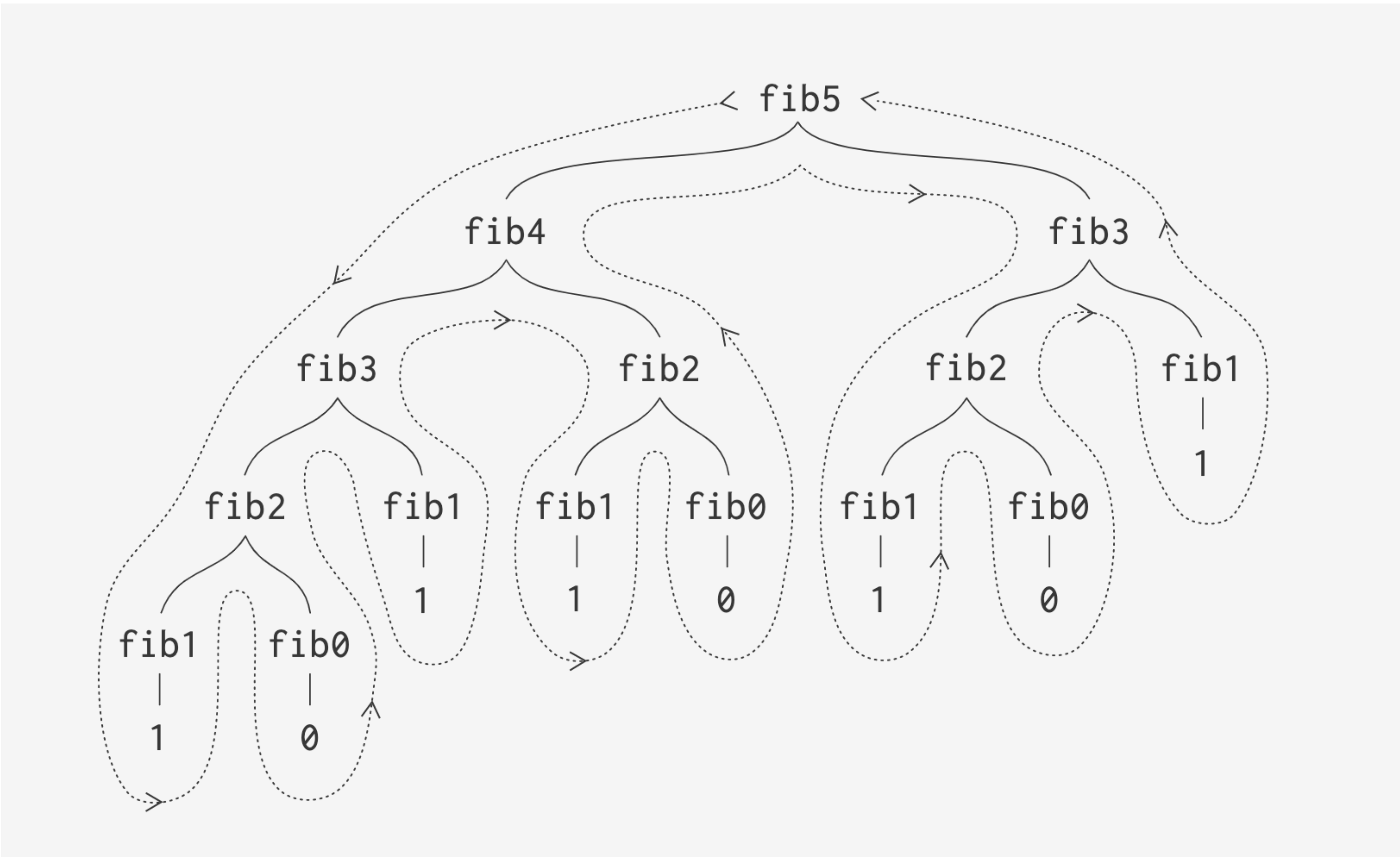
$$fib(n) = \begin{cases} 0 & \text{se } n = 0 \\ 1 & \text{se } n = 1 \\ fib(n-2) + fib(n-1) & \text{se } n \geq 2 \end{cases}$$

Fibonacci(5)

- Para calcular $fib(5)$, por exemplo, podemos aplicar a definição recursivamente para obter:

  fib(5) = fib(3) + fib(4) = … (quadro)

Árvore Chamadas Recurs. Fibonacci



Redundância Computacional

- Definição recursiva dos números de Fibonacci refere-se a si mesma duas vezes
  - Muita redundância computacional
- Método iterativo $fib(n)$ é muito mais eficiente

Propriedades Algoritmos Recursivos

- Não gere sequência **infinita** de chamadas a si mesmo
- Para, no mínimo, um argumento ou grupo de argumentos, a função recursiva f deve ser definida de modo a não envolver f
- Deve existir uma “saída” da sequência de chamadas recursiva

Propriedades: Exemplos

- Nos exemplos, as partes não-recursivas das definições foram:
  - fatorial: $0! = 1$
  - multiplicação: $a \cdot 1 = a$
  - sequência de fibonacci: $fib(0) = 0; fib(1) = 1$

Escrevendo Programas Recursivos

- Desenvolver solução recursiva para problema cujo algoritmo não é fornecido é uma tarefa difícil
    - Definições e algoritmos originais precisam ser desenvolvidos
- Ao enfrentar a tarefa de resolver um problema, não há por que procurar uma solução recursiva pois a maioria dos problemas pode ser solucionada de maneira não-recursivos

Solução Recursiva: Passos

1. Identificar um **caso “trivial”** (caso base) para o qual uma solução não-recursiva é imediatamente obtida;
2. Encontrar um método para solucionar um **caso “complexo"** em termos de um caso “mais simples”;

Solução Recursiva Fatorial

1. O caso trivial é $0!$, que é definido como 1
2. No caso da função fatorial, temos esta definição, uma vez que: $n! = n \cdot (n-1)!$

Definição Recursiva Fatorial

$$fatorial(n) = \begin{cases} 1 & \text{se } n = 0 \text{ (caso base)} \\ n \cdot fatorial(n-1) & \text{se } n > 0 \text{ (passo de indução)} \end{cases}$$

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

Solução Recursiva Fibonacci

1. No caso da função de Fibonacci, dois casos triviais foram definidos $fib(0) = 0$ e $fib(1) = 1$
2. Um caso complexo, $fib(n)$, é, portanto, reduzido a dois casos mais simples:
$fib(n) = fib(n-1) + fib(n-2)$

Obs.: Por causa da definição de $fib(n)$ como $fib(n-1) + fib(n-2)$, são necessários dois casos triviais diretamente definidos

Eficiência da Recursividade

- Em geral, versão não-recursiva é mais eficiente, em termos de tempo e espaço, do que versão recursiva
  - Existe um custo para entrar e sair de um bloco
- Contudo, existem situações onde uma solução recursiva é o método mais natural e lógico de solucionar um problema

Referências

- TENENBAUM, Aaron M; ANGENTEIN, Moshe; LANGSAM, Yedidyah. Estruturas de dados usando C. Sao Paulo, SP: Pearson, 1995. 884p.
- FEOFILOFF, Paulo. Algoritmos em linguagem C. Rio de Janeiro: Elsevier, 2009. 208 p.