Recursividade

Prof. Thiago Felski Pereira, MSc.

Definições

- Repetição pode ser obtida de 2 maneiras:
  - Laços (for, while, etc)
  - Chamada recursiva de métodos (recursão)

- Funções recursivas são funções que chamam a si mesmas para compor a solução de um problema.

- Uma função é dita recursiva quando dentro do seu código existe uma chamada para si mesma.

Recursão Direta

- Quando uma função chama a si mesma diretamente

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

Recursão Indireta

- **Recursão indireta** ocorre quando uma função chama outra, e essa, por sua vez chama a primeira

- **Exemplo**: verificar se um número é par ou ímpar sem utilizar o operador de %
  - Um número par é qualquer número divisível por 2
  - Um número ímpar é qualquer número não divisível por 2
  - Definição recursiva de um número par
    - 0 é par. N é par se N-1 é ímpar.
    - 1 é ímpar. N é impar se N-1 é par.

Recursão Indireta

```
bool par (int N){
    if (N == 0){
        return true;
    }
    else if (N == 1){
        return false;
    }
    return impar(N-1);
}
```

```
bool impar (int N){
    if (N == 0){
        return false;
    }
    else if (N == 1){
        return true;
    }
    return par(N-1);
}
```

Tipos de Recursão

- Linear
  - Faz somente uma chamada recursiva (para si mesma)
  - Exemplo: `return (num * fatorial(num-1));`

- Binária
  - Existem duas chamadas recursivas para cada caso não básico
  - Exemplo: `return (fibonacci(posicao-1) + fibonacci(posicao-2));`

- Múltiplas chamadas:
  - Quando faz mais de duas chamadas recursivas
  - Exemplo: `return (x(N-1) + x(N-2) + x(N-3));`

Vantagens e Desvantagens

- **Vantagens da Recursão**
  - Redução do tamanho do código fonte
  - Maior clareza do algoritmo para problemas de definição naturalmente recursiva

- **Desvantagens da Recursão**
  - Baixo desempenho na execução devido ao tempo para gerenciamento das chamadas
  - Dificuldade de depuração dos subprogramas recursivos, principalmente se a recursão for muito profunda

- **Resumindo**
  - Na maioria das vezes a codificação na forma recursiva é mais simples (reduzida), mas a forma interativa tende a ser mais eficiente

Definições

- **Todo algoritmo recursivo pode ser implementado de forma iterativa e vice-versa.**
  - **Isso quer dizer que você pode implementar um algoritmo similar sem utilizar recursividade que executa a mesma tarefa**

Como criar uma função recursiva

- A solução de qualquer problema recursivo pode ser dividida em passos menores

1. Definir uma **Regra Geral** que seja válida para todos os casos
2. Definir um **Ponto de Parada**
3. Verificar se o **Ponto de Parada** é atingido, ou seja, o algoritmo não entrará em um loop infinito

Construindo uma função recursiva

- Um dos exemplos mais utilizados na literatura para explicação de recursividade é o fatorial
- Vamos, então, entender esse problema
  - **5!** = 5 * 4 * 3 * 2 * 1 = 120
  - **4!** = 4 * 3 * 2 * 1 = 24
  - **3!** = 3 * 2 * 1 = 120
  - **2!** = 2 * 1 = 120
  - **1!** = 1 = 1
  - **0!** = 1

  - **N! = 1, se N <= 0**
  - **N! = 1 * 2 * 3 * ... * N se N > 0**

Regra Geral

- A **Regra Geral** reduz a resolução do problema por meio da chamada recursiva de casos menores, que são resolvidos pela chamada de casos ainda menores da própria função, e assim seguindo até atingir o Ponto de Parada.
- No problema do fatorial
  - **5!** = 5 * 4 * 3 * 2 * 1 = 120
- Dividindo agora o problema em partes menores
  - **5!**
  - (5 * 4!) Observe que o 5! foi dividido em 2 partes
  - 5 * (4 * 3!) Observe que o 4! foi dividido em 2 partes
  - 5 * 4 * (3 * 2!) Observe que o 3! foi dividido em 2 partes
  - 5 * 4 * 3 * (2 * 1!) Observe que o 2! foi dividido em 2 partes
- Assim sendo, temos como **Regra Geral**
  - **N! = N * (N-1)!, para N > 0**

Ponto de Parada

- O Ponto de Parada é o ponto onde a função será encerrada, e normalmente é o limite inferior ou o limite superior da Regra Geral

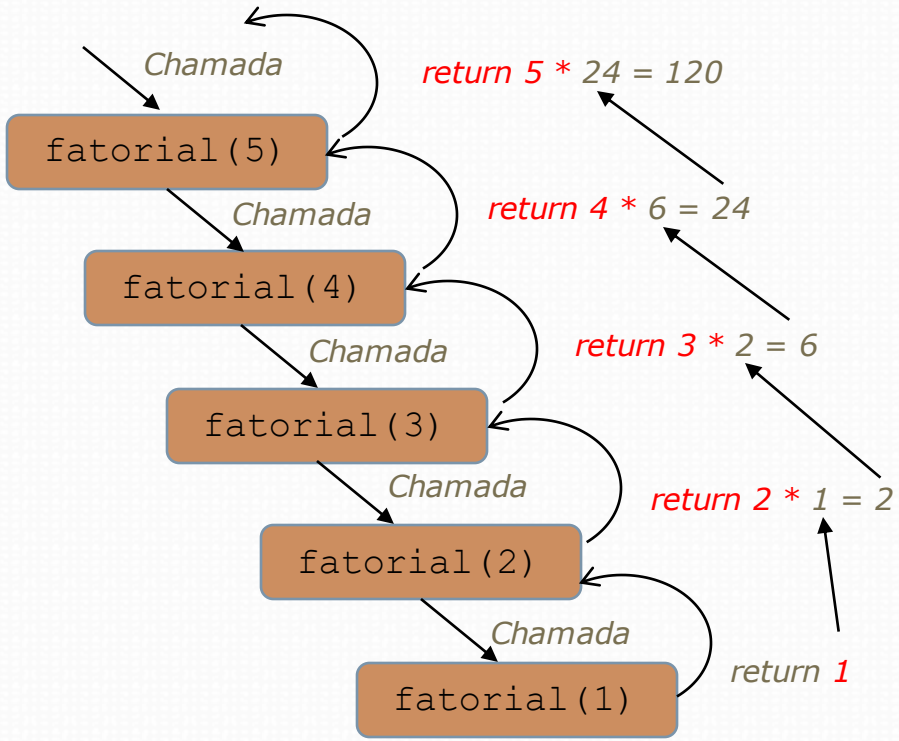
- Observe que todo fatorial encerra em 1
  - **5!** = 5 * 4 * 3 * 2 * 1
  - **4!** = 4 * 3 * 2 * 1
  - **3!** = 3 * 2 * 1
  - **2!** = 2 * 1
  - **1!** = 1
  - **0!** = 1

- Assim sendo, temos como **Ponto de Parada**
  - **N = 1, para N <= 1**

Teste de mesa: fatorial recursivo

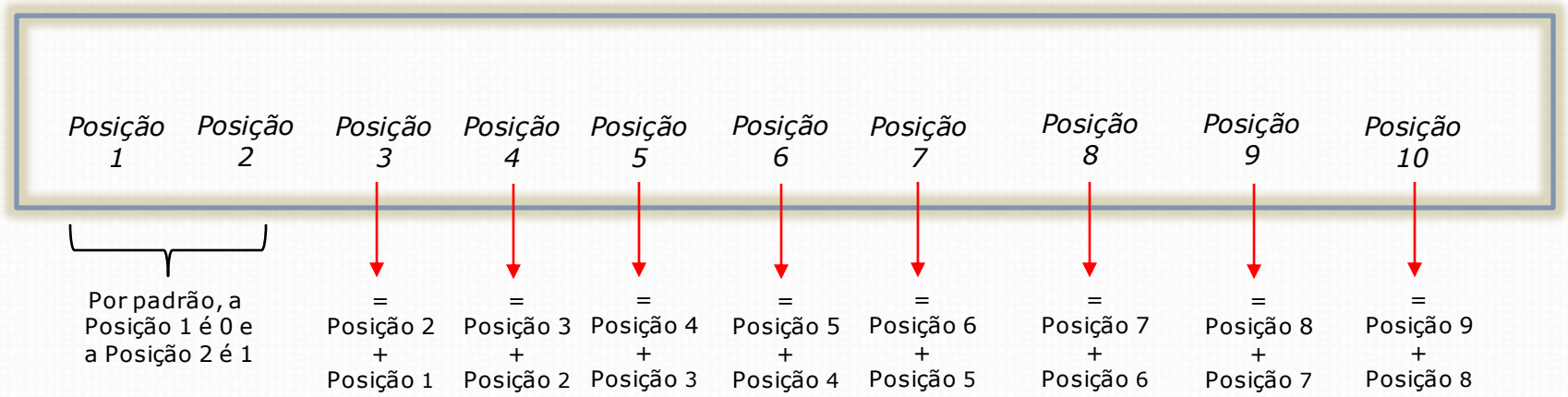
```
int fatorial (int num) {
    if (num <= 1){
        return 1;
    }
    return (num * fatorial(num-1));
}
```

Considere que o usuário digitou num = 5



Fibonacci Recursivo

- Um dos exemplos mais utilizados para explicação de Recursividade é a Série de Fibonacci. Considerando que se quer saber o elemento de uma posição N na série



- Por exemplo, se o usuário digitar que quer saber o elemento da posição 6, a resposta deverá ser 8.

Fibonacci Recursivo

- Exercício
  - Para entender melhor o processo, busque implementar a função Fibonacci **não recursiva** primeiro

Fibonacci Recursivo

- `1, 1, 2, 3, 5, 8, 13, 21, 34, …`

- Analisando, pode-se verificar que o próximo elemento é gerado somando-se a posição anterior com a antepenúltima posição. Por exemplo, a Posição 4 é adquirida somando-se a Posição 3 e a Posição 2.

- **Regra Geral**
  - Fibo(pos) = 1, se posição <= 2
  - Fibo(pos) = Fibo(pos-1) + Fibo(pos-2), se posição > 2
  -

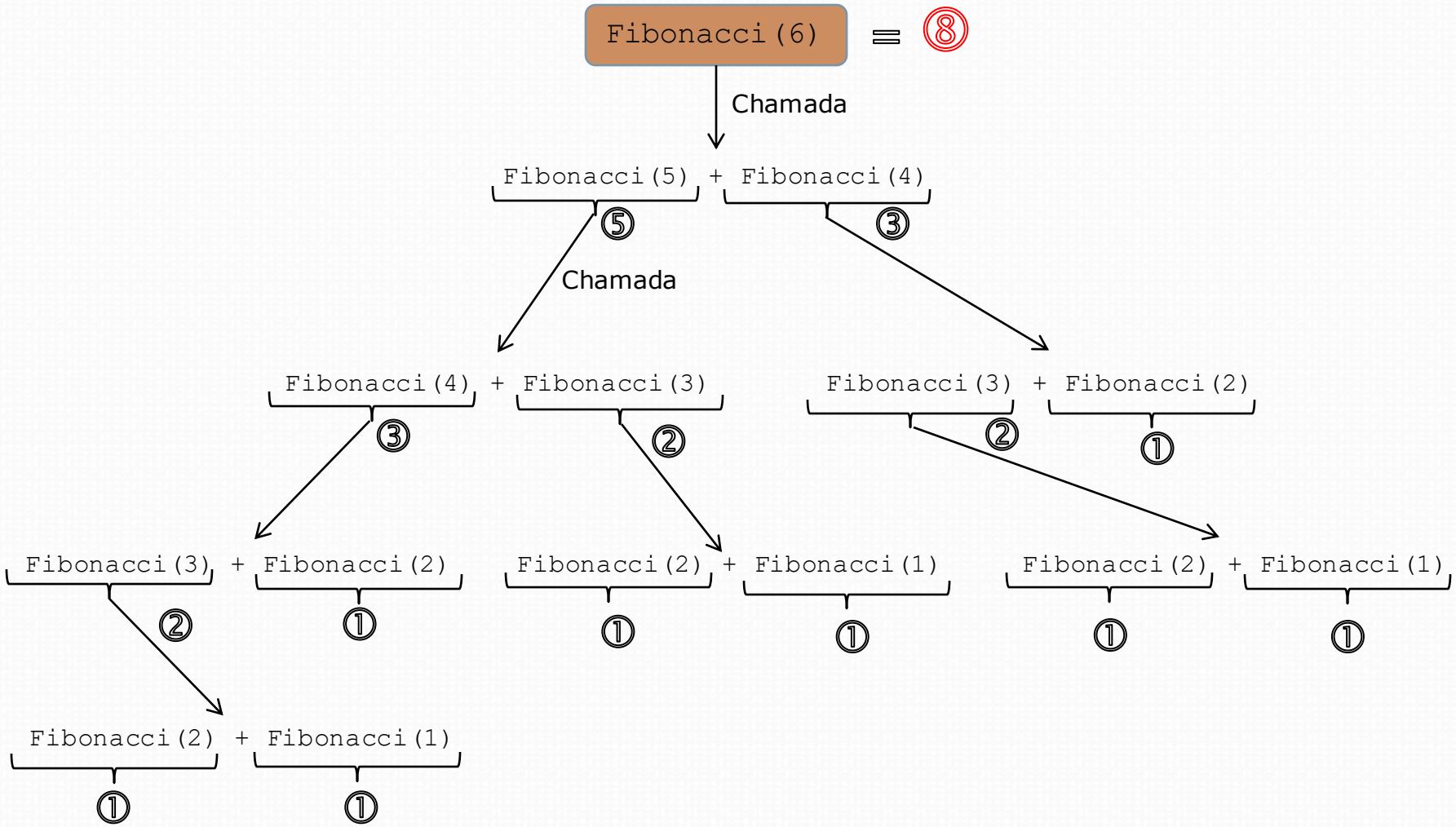
Fibonacci Recursivo

- Analisando, a série para a esquerda depende do valor digitado pelo usuário, mas o início é sempre o mesmo (1). Como o usuário entra com a posição final, o Ponto de Parada será as duas primeiras posições iniciais, pois ambas retornam 1.

- Então, o **Ponto de Parada** seria:
  - Fibo(pos) = 1, para N <= 2

- Agora que conhecemos bem o Fibonacci, será que consegue fazer a versão recursiva dele?

Teste de Mesa: Fibonacci Recursivo



Obrigado pela atenção

contato: Felski@univali.br