



Entendendo a Recursão I

Ao desenvolver um programa ou algoritmo é comum repetir um pedaço do código algumas vezes. Para isso, na maioria dos casos, utilizamos as estruturas de repetição. Porém, os códigos podem ficar muito complexos utilizando essas estruturas. Por isso, temos a possibilidade de substituir as estruturas de repetição pela estrutura recursiva ou Recursão.

A recursão é bastante poderosa pois permite que uma função ou um procedimento possa ser definido dentro dele mesmo. Uma vez definidos, você pode executá-los várias vezes. Isso acontece, pois, a chamada do módulo ou da função acontece dentro deles mesmos.


DEFINIÇÃO DA RECURSÃO

A recursão faz a chamada de uma função ou procedimento como parte de sua própria definição. A chave do funcionamento dela é garantir que existe uma condição que faça eles pararem.

Neste caso, acontece a execução de uma ação que não é recursiva pela mesma função ou procedimento, ou seja, na ação da parada, não se chama a função ou o procedimento recursivo novamente.

De forma direta ou indireta, há controle de fluxo do algoritmo, de forma que uma função ou procedimento chama a si mesmo(a).

Se uma função ou procedimento faz a própria chamada, a execução também chama a si mesma e assim por diante, de forma contínua e sem fim. Esse acontecimento denomina-se recursão infinita, que é parecida com

os erros quando se usa a estrutura de repetição, que deixam o programa rodar infinitamente. 

Para que você não tenha recursão infinita em seus algoritmos, ela precisa ser criada de forma que você garanta que, em algum momento do programa, seja finalizada sem se chamar novamente.

Exemplo para o cálculo do fatorial de um número:

se $n = 0$, então o $\text{fat}(n)$ é 1

caso contrário, $\text{fat}(n) = n * \text{fat}(n-1)$

REGRAS DA RECURSÃO

Ao desenvolver uma recursão, você precisa garantir a aplicação das três regras a seguir:

1. Saber quando parar.
2. Decidir como fazer a próxima ação.
3. Quebrar uma jornada recursiva em um passo mais uma jornada recursiva menor.

A primeira regra é a checagem para garantir que a recursão chegou ao fim, antes de tomar a decisão de ir para a outra jornada recursiva.

Na função do fatorial, se $n = 0$, então a recursão é encerrada, devolvendo o valor 1 de $\text{fat}(0)$.

A segunda regra é uma forma de garantir que o problema seja quebrado em partes menores e mais fáceis de resolver.

Na função fatorial, se $n > 0$, então a recursão vai devolver $n * \text{fat}(n-1)$, a chamada recursiva se chama, porém, passando como parâmetro um número menor.

A terceira regra permite que a recursão se chame novamente, com um problema menor. Veja como fica:

$\text{fat}(n)$

$n * \text{fat}(n-1)$

$n * (n-1) * \text{fat}(n-2)$

...

$n * (n-1) * \dots * \text{fat}(2)$

$n * (n-1) * \dots * 2 * \text{fat}(1)$

$n * (n-1) * \dots * 2 * 1 * \text{fat}(0)$

$n * (n-1) * \dots * 2 * 1 * 1$

RECURSÃO COM CAUDA E SEM CAUDA

A recursão com cauda acontece quando, durante a execução da recursão, ele deixa um rastro na memória, o que chamamos de cauda. Normalmente, acontece quando a sua recursão é uma função e retorna um valor.

Exemplo para o cálculo do fatorial na recursão com cauda.

inteiro fat (inteiro n)

início

Declarar f inteiro;



se (n = 0) então

retornar 1; // regra 1

senão

retornar n * fat(n-1); // regra 2 e 3

fimse;

fim_módulo;

Vamos ver um exemplo da execução do algoritmo do fatorial escrito em recursão com cauda e veja o rastro que ele deixa quando está executando.

fat (5)

5 * fat(4)

5 * 4 * fat(3)

5 * 4 * 3 * fat(2)

5 * 4 * 3 * 2 * fat(1)

5 * 4 * 3 * 2 * 1 * fat(0)

5 * 4 * 3 * 2 * 1 * 1

5 * 4 * 3 * 2 * 1

5 * 4 * 3 * 2

5 * 4 * 6

$$5 * 24$$



120 = resultado do fatorial de 5.

A recursão sem cauda acontece quando, durante a execução da recursão, não deixa um rastro na memória. Ou quando sua recursão é um procedimento e não retorna um valor.

Exemplo para o cálculo do fatorial na recursão sem cauda.

fat (inteiro n, inteiro f)

início

se (n > 1) // regra 1

então

fat (n-1, f * n); regra 2 e 3

senão

escrever ("O fatorial é " + f);

fimse;

fim_módulo;

Vamos ver um exemplo da execução do algoritmo do fatorial escrito em recursão sem cauda e veja não fica rastro durante sua execução.

fat (5, 1)

fat (4, 5)

fat (3, 20)

fat (2, 60)



fat (1, 120)

O fatorial é 120

APLICANDO RECURSÕES

Vamos exemplificar a aplicação de recursão com duas funções bastante conhecidas. Uma delas é o cálculo do fatorial de um número e o outro é o cálculo do enésimo termo da série de Fibonacci.

Sabemos que o fatorial de um número é o produto da multiplicação deste pelo seu antecessor e, assim, sucessivamente, até chegar no número 1, com exceção do 0, cujo fatorial é 1.

Escrevendo a definição da função fatorial na forma de função matemática, podemos representá-la como segue, considerando que n é um número natural:

se $n = 0$, então, o $\text{fatorial}(n)$ é 1

caso contrário, $\text{fatorial}(n) = n * \text{fatorial}(n-1)$, para $n > 0$

Sabemos que o enésimo termo da série de Fibonacci é definido pela soma dos seus dois antecessores consecutivos, com exceção do Fibonacci do primeiro e do segundo termo que é igual a 1.

Escrevendo a definição da função Fibonacci na forma de função matemática, podemos representá-la como segue, considerando que n é um número natural:

se $n = 1$, então, $\text{fibonacci}(n) = 1$

se $n = 2$, então, $\text{fibonacci}(n) = 1$



caso contrário, $\text{fibonacci}(n) = \text{fibonacci}(n-1) + \text{fibonacci}(n-2)$, para $n > 2$

Quando desenvolvemos algoritmos recursivos, conseguimos listar algumas vantagens que podem ser observadas:

têm menos linhas de comandos de programação

são muito mais fáceis de entender e compreender

são mais legíveis e claros

os seus programas são mais fáceis de serem implementados

Quando desenvolvemos algoritmos recursivos, conseguimos listar algumas desvantagens que podem ser observadas:

utilização de muita memória para armazenar os dados que ficam na recursão

utilização de pilhas de memórias com alocação e liberação destas memórias

são mais lentos quando comparados aos seus programas iterativos

menos eficientes do que os programas iterativos

dificuldade para verificar a depuração durante o desenvolvimento

Quando desenvolvemos algoritmos recursivos, conseguimos listar algumas aplicações que podem ser observadas:

um algoritmo recursivo nem sempre é o melhor algoritmo para resolver o problema proposto

algoritmos recursivos são aplicados nas manipulações de estruturas de dados de árvores



nos compiladores para os analisadores léxicos

problemas que consideram tentativa e erro

Vamos exemplificar construindo um algoritmo recursivo para o seguinte problema: queremos calcular o fatorial de um número inteiro. Podemos dividir o problema em dois casos especiais:

- se $n = 0$ então, o valor de $n!$ é 1
- caso contrário, $n! = 1 * 2 * 3 * \dots * (n-2) * (n-1) = (n-1)! * n$

Podemos calcular o valor de $n!$ calculando o valor de $(n-1)!$ e então multiplicando esse valor pelo valor de n .

Matematicamente, essa definição pode ser escrita da seguinte forma:

$\text{fat}(n) = 1$ se $n = 0$

$\text{fat}(n) = n * \text{fat}(n-1)$, caso contrário.

Observe que a própria função fat encarrega-se de se “chamar”, ou seja, foi definida em termos dela mesma.

fatP (n inteiro, x inteiro, f inteiro)

início

se ($x = 0$ ou $x = 1$) então

escrever("O fatorial de " + n + " é " + f); // regra 1

senão

// chamada da função recursiva



fatP(n, x-1, fx); // regra 2 e 3

fimse;

fim_módulo;

Algoritmo FatorialP

início

nro inteiro;

escrever ("Digite um valor que você deseja saber o fatorial");

ler(nro);

se (nro < 0)

então

escrever ("Valor inválido para cálculo de fatorial, o valor precisa ser maior ou igual a zero");

//sair do programa

senão

fatP(nro, nro, 1); // chamada da função recursiva

fimse;

fim_módulo;

O algoritmo completo para a resolução deste problema, em pseudocódigo, é o seguinte:

// módulo função recursivo



numérico_inteiro fat (numérico inteiro n)

início

Declarar

f numérico_inteiro;

se ($n = 0$)

então

retornar 1; // regra 1

senão

$f \leftarrow n * \text{fat}(n-1)$; // regra 2 e 3

retornar *f*;

fimse;

fim_módulo;

// módulo principal

Algoritmo Fatorial

início

Declarar

f, *nro* numérico_inteiro;

escrever("Digite um valor que você deseja saber o fatorial");

ler(nro);



se (nro < 0)

então

escrever ("Valor inválido para cálculo de fatorial, o valor precisa ser maior ou igual a zero");

<sair do programa>;

senão

f ← fat(nro);

escrever("O fatorial de ", nro, " é ", f);

fimse;

fim_algoritmo.

Realizando um teste de mesa para o fatorial de 5, temos:

fat (5)

*5 * fat(4)*

*5 * 4 * fat(3)*

*5 * 4 * 3 * fat(2)*

*5 * 4 * 3 * 2 * fat(1)*

*5 * 4 * 3 * 2 * 1 * fat(0)*

*5 * 4 * 3 * 2 * 1 * 1*

$$5 * 4 * 3 * 2 * 1$$



$$5 * 4 * 3 * 2$$

$$5 * 4 * 6$$

$$5 * 24$$

120 = resultado do fatorial de 5.


Passo a passo, suponha que queremos saber o fatorial de 3. Primeiro, o algoritmo verifica se o valor digitado é menor que zero, como não é, o algoritmo faz a chamada da função fat passando como parâmetro o valor inteiro 3. (chamada 1)

Dentro da função fat, o algoritmo verifica se o valor passado como parâmetro é igual a 0, como não é, o algoritmo retorna o valor do cálculo da multiplicação de 3 pela chamada da função fat passando como parâmetro o valor inteiro 2. (chamada 2)

Dentro da função fat, o algoritmo verifica se o valor passado como parâmetro é igual a 0, como não é, o algoritmo retorna o valor do cálculo da multiplicação de 2 pela chamada da função fat passando como parâmetro o valor inteiro 1. (chamada 3)

Dentro da função fat, o algoritmo verifica se o valor passado como parâmetro é igual a 0, como não é, o algoritmo retorna o valor do cálculo da multiplicação de 1 pela chamada da função fat passando como parâmetro o valor inteiro 0. (chamada 4)

Dentro da função fat, o algoritmo verifica se o valor passado como parâmetro é igual a 0, como é, o algoritmo retorna o valor 1 para a chamada 4.

A chamada 4 recebe o retorno da função fat, multiplica esse valor por 1 e retorna o cálculo da multiplicação, ou seja, 1, para a chamada 3. 

A chamada 3 recebe o retorno da função fat, multiplica esse valor por 2 e retorna o cálculo da multiplicação, ou seja, 2, para a chamada 2.

A chamada 2 recebe o retorno da função fat, multiplica esse valor por 3 e retorna o cálculo da multiplicação, ou seja, 6, para a chamada 1.

A chamada 1 recebe o retorno da função fat e atribui esse valor para a variável f.

O algoritmo, então, envia a seguinte mensagem para o usuário:

O fatorial de 3 é 6

Atividade extra

Indicação de leitura: Fundamentos da Programação de Computadores algoritmos, pascal, C/C++ e Java, da Ana Fernanda Gomes Ascencio e Edilene Aparecida Veneruchi de Campos, capítulo 1.

Referência Bibliográfica

PUGA, S.; RISSETTI, G. **Lógica de Programação e Estruturas de Dados, com aplicações em Java**. São Paulo. Editora Pearson. 3ª. Edição. 2016.

FORBELLONE, A.L.V.; EBERSPACHER, H.F. **Lógica de Programação: a construção de algoritmos e estruturas de dados**. 3ª Edição. São Paulo. Prentice Hall. 2005.

Atividade Prática 04 – Entendendo recursão I

Título da Prática: Recursão de soma de números inteiros



Objetivos: Entender como utilizar o netbeans para desenvolver programas em Java para somar números inteiros

Materiais, Métodos e Ferramentas: Computador, netbeans, Java.

Atividade Prática

Ao desenvolver um programa ou algoritmo é comum repetir um pedaço do código algumas vezes. Para isso, na maioria dos casos, utilizamos as estruturas de repetição. Porém, os códigos podem ficar muito complexos utilizando essas estruturas. Por isso, temos a possibilidade de substituir as estruturas de repetição pela estrutura recursiva ou Recursão.

A recursão é bastante poderosa pois permite que uma função ou um procedimento possa ser definido dentro dele mesmo. Uma vez definidos, você pode executá-los várias vezes. Isso acontece, pois, a chamada do módulo ou da função acontece dentro deles mesmos.

O Algoritmo de recursão para calcular a soma de números inteiros pode ser escrito como segue.

```
numérico_inteiro soma1(numérico_inteiro nro , numérico_inteiro soma)
```

```
início
```

```
Declarar
```

```
    s numérico_inteiro;
```

```
se (nro = 100)
```

```
    então
```

s ← nro + soma;



senão

s ← soma1(nro+1,nro+soma);

fimse;

retornar s;

fim_módulo;

Algoritmo Soma100

início

Declarar

soma numérico_inteiro;

soma ← soma1(1,0);

escrever("a soma dos cem primeiros números é " , soma);

fim_algoritmo.

Desenvolva o programa em Java deste algoritmo no NetBeans.

Gabarito Atividade Prática

```
import javax.swing.*;
```

```
class Somar100
```



```
{  
  
public static int Soma1 (int nro, int soma)
```

```
{
```

```
    int s;
```

```
    if (nro == 100){
```

```
        s = nro + soma;
```

```
    }
```

```
    else{
```

```
        s = Soma1(nro+1, nro+soma);
```

```
    }
```

```
    return s;
```

```
}
```

```
public static void main (String arg [])
```

```
{
```

```
    int soma;
```

```
    soma = Soma1(1, 0);
```

```
    JOptionPane.showMessageDialog(null,"A soma dos cem primeiros  
números é: "+soma);
```




```
System.exit(0);
```

```
}
```

```
}
```

[Ir para exercício](#)