capítulo 3

Recursão

> *O teu êxito depende, muitas vezes, do êxito das pessoas que te rodeiam.*

Benjamin Franklin

O êxito, na maioria dos casos, é um processo recursivo.

OBJETIVOS DO CAPÍTULO

Ao final da leitura deste capítulo, o leitor deverá ser capaz de:

- conhecer e identificar objetos e processos recursivos;
- conhecer os prós e os contras da implementação recursiva e saber quando utilizar essa técnica, principalmente em termos de desempenho;
- conhecer e praticar a implementação dos principais algoritmos recursivos.

Para começar

Provavelmente, já deve ter acontecido isto com você antes: dormindo, você sonha que está dormindo e sonhando.

Em artes, esse fenômeno é conhecido como *efeito Droste* ou *miseenabyme*. Uma imagem que aparece dentro dela mesma, em um local semelhante ao da primeira imagem. Uma versão menor, que contém uma versão menor que a outra, que contém outra menor ainda, e assim por diante.

O efeito Droste tem esse nome por causa da imagem que ilustrava as caixas de cacau em pó Droste, uma das principais marcas holandesas: uma enfermeira carregando uma bandeja com uma xícara de chocolate quente e uma caixa do produto. A imagem foi feita em 1904 e mantida por décadas (Figura 3.1).

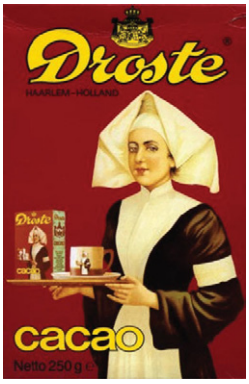

FIGURA 3.1: Caixa do cacau em pó Droste, gênese do "efeito Droste".

Esse efeito nos remete à ideia de repetição. Mas não qualquer repetição; a repetição de um objeto dentro dele mesmo, numa ideia de *looping* ou laço contínuo. A esse processo dá-se o nome de *recursão*, ou seja, um objeto é parcialmente definido em termos dele mesmo.

A recursão pode ser encontrada na matemática, nas ciências, em computação e no cotidiano. Experimente colocar um objeto entre dois espelhos e você terá uma ideia prática da recursão infinita (ou finita, até onde sua visão lhe permitir).

A recursão está relacionada à indução matemática. E, como sabemos, a matemática é a fonte de todo o formalismo necessário em computação.

Neste capítulo, aprenderemos sobre o processo de implementação de soluções computacionais utilizando a recursividade para resolver alguns tipos de problema. Veremos também que a recursividade, mesmo sendo a forma mais prática de implementar alguns tipos de resolução, não é a técnica mais eficiente. Preparado? Vamos lá!

Atenção

A implementação de um algoritmo recursivo em uma linguagem de programação, partindo de uma definição matemática também recursiva, é praticamente direta e imediata. Por essa razão, esse tipo de algoritmo recursivo possui código muito mais legível e compacto.

O cálculo da potência *p* de um dado número *n* é obtida pela multiplicação sucessiva desse número *n* por ele mesmo, *p* vezes. Supondo que $n = 2$ e quiséssemos calcular 2^5, teríamos: $2 \times 2 \times 2 \times 2 \times 2 = 32$. Ou seja, *5* vezes a multiplicação do número *2*.

Matematicamente, tomando como a base a potência do número *2*, poderíamos ter o seguinte:

$$2^n = \begin{cases} 1, & se\ n = 0 \\ 2 \cdot (2^{n-1}), & se\ n \geq 1 \end{cases}$$

Como você faria para implementar uma função em linguagem de programação para retornar o valor da *potência de 2*, dado como parâmetro um número inteiro, positivo, *n*?

Dica

Para responder a essa questão, pense que existe uma condição base (inicial ou final) que corresponde a potencia ser igual a zero, resultado no valor 1. Os demais correspondem a multiplicação sucessiva do número 2, *n* vezes.

E então? Conseguiu?

Possivelmente, um bom programador como você, que já passou pelo tópico Algoritmos, não teria dificuldade para desenvolver uma função que utilizasse um comando de repetição interno, como o apresentado no Código 3.1, a seguir:

Código 3.1

```
...

int pot2 (int n)                    // função potencia de 2, recebe um valor inteiro como parâmetro.
{
       int k, pot=1;
       if (n==0) return 1;          // resolve o caso base (ou situação de parada)
       else
          for (k=1; k<=n; k++) {
            pot = pot * 2;          // caso recursivo (para o n-ésimo valor)
          }
       return pot;
}
```

Mas, se pensarmos em termos da definição matemática, essa solução não se assemelha à definição inicial. Como isso poderia ser feito? Utilizando o conceito de *recursividade*. Ou seja, teríamos uma solução simples (ou caso base) quando o expoente fosse igual a zero. Os demais casos seriam recursivos, decrementando o valor do expoente. O Código 3.2 apresenta uma forma de resolução recursiva desse problema.

Código 3.2

```
...

int pot2 (int n)                    // função potencia de 2 recursiva, recebe um valor inteiro como parâmetro.
{
       if (n==0) return 1;          // resolve o caso base (ou situação de parada)
       else return 2 * pot2(n-1);   // caso recursivo (para o n-ésimo valor)
}
```

Pode ser que você ainda não tenha compreendido inteiramente o processo de recursão. Não se preocupe. É nisso que vamos nos aprofundar agora. Preparado? Então, vamos em frente!

Conhecendo a teoria para programar

Uma das mais elegantes (e, em muitos casos, complexas) técnicas da matemática é a recursividade. Ela pode ser caracterizada quando se define um objeto em termos dele mesmo.

O grande potencial da recursão está na possibilidade de poder definir elementos com base em versões mais simples desses mesmos elementos.

Em termos computacionais, trata-se de dividir um problema maior em problemas menores, em que a resolução é feita por uma mesma função (ou método), que é recorrentemente chamada. Muitos autores chamam essa técnica computacional de "dividir para conquistar", parafraseando o grande imperador francês Napoleão Bonaparte.

Para implementar uma função (ou método) recursiva, é necessário estabelecer pelo menos dois elementos:

- uma condição de parada ou terminação. Geralmente, essa condição estabelece uma solução trivial ou um evento que encerra a autochamada consecutiva;
- uma mudança de estado a cada chamada, ou seja, o estabelecimento de alguma diferença entre o estado inicial e o próximo estado da função (ou método). Isso pode ser feito, por exemplo, decrementando um parâmetro da função recursiva.

Um exemplo clássico que se utiliza para exemplificar a recursão computacional é o cálculo do fatorial de um número *n*.

Apenas para aquecer, você conseguiria calcular o fatorial de um número sem utilizar a técnica de recursão? Recordando: o fatorial de um número *n* é igual à multiplicação dos números inteiros de 1 até *n*, ou seja, 1 * 2 * 3 * 4 * ... * *n*. Dessa forma, se *n* for igual a 4, teríamos que o fatorial de 4 é igual a 24 ou 1 * 2 * 3 * 4 = 24.

Matematicamente, teríamos que o fatorial de um inteiro positivo *n*, denotado n!, é definido como o produto dos inteiros de 1 até n. Se n = 0, então *n!* é definido como 1 por convenção. De maneira mais formal, para qualquer inteiro n ≥ 0,

$$n! = \begin{cases} 1, & se\ n = 0 \\ n \cdot (n-1) \cdot (n-2) \cdot \ldots \cdot 3 \cdot 2 \cdot 1 & se\ n \geq 1 \end{cases}$$

Com isso em mente, tente fazer a implementação sem utilizar a técnica de recursão. Depois, consulte o Código 3.3, a seguir, que ilustra tal possibilidade.

Código 3.3

```
...

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

Conseguiu? É importante tentar!

Como você pôde ver, dependendo do valor de *n* (valor inteiro maior ou igual a zero), seu valor do fatorial é calculado e retornado pela função.

Como seria a implementação dessa mesma função, mas de forma recursiva? A primeira coisa a fazer é identificar o caso base ou situação de parada. No caso do cálculo do fatorial, a condição de parada é n = 0. Isso resulta em fatorial igual a 1. Os casos recursivos são sempre a multiplicação de um valor pelo próximo valor de *n*, com mudança de estado de seu valor decrementado de 1. No Código 3.4, propomos uma forma de implementar o cálculo fatorial de forma recursiva.

Código 3.4

```
...

int fatorial (int n)                    // função fatorial recursiva, recebe um valor inteiro como parâmetro.
{
       if (n==0) return 1;              // resolve o caso base (ou situação de parada)
       else return n*fatorial(n-1);     // caso recursivo (para o n-ésimo valor)
}
```

Para melhor o entendimento do funcionamento dessa função recursiva, vamos fazer um teste de mesa ou rastreamento das recursões, supondo a chamada da função fatorial, com um parâmetro inicial igual a 4. Assim, em algum lugar, como a função *main()*, existirá uma chamada como a seguinte:

```
...
x = fatorial(4);
...
```

Para que você possa acompanhar com detalhes o teste de mesa, vamos numerar as linhas internas da função fatorial(). Utilizaremos o Código 3.5, a seguir:

Código 3.5

```
  int fatorial (int n)                    // função fatorial recursiva, recebe um valor inteiro como parâmetro.
  {
L1     if (n==0)
L2         return 1;                      // resolve o caso base (ou situação de parada)
L3     else return n*fatorial(n-1);       // caso recursivo (para o n-ésimo valor)
  }
```

Para que a análise da execução (teste de mesa) fique mais clara, é conveniente tratar cada chamada da função fatorial() como uma nova instância da função.

Conceito

Uma instância de uma função corresponde ao processo de alocação de um novo espaço na memória (pilha) para comportar as necessidades de utilização de variáveis locais inerentes àquela função.

Na Tabela 3.1, a seguir, observe os valores passados pelas chamadas e retornos.

TABELA 3.1: Teste de mesa (análise da execução) da função fatorial() para o valor n = 4

Linha	Instância	Explicação	Valor *n*	Passagem de parâmetro	Retorno
L1	1	***n*** não é <= 1	4		
L3	1	Chama ***fatorial()***	4	3	
L1	2	***n*** não é <= 1	3		
L3	2	Chama ***fatorial()***	3	2	
L1	3	***n*** não é <= 1	2		
L3	3	Chama ***fatorial()***	2	1	
L1	4	***n*** é <= 1	1		
L2	4	Retorna 1	1		1
L3	3	Retorna 2*1	2		2
L3	2	Retorna 3*2	3		6
L3	1	Retorna 4*6	4		24

Esquematicamente, depois da primeira chamada da função ***fatorial()***, teríamos a seguinte sequência de chamadas, ilustrada na Figura 3.2.

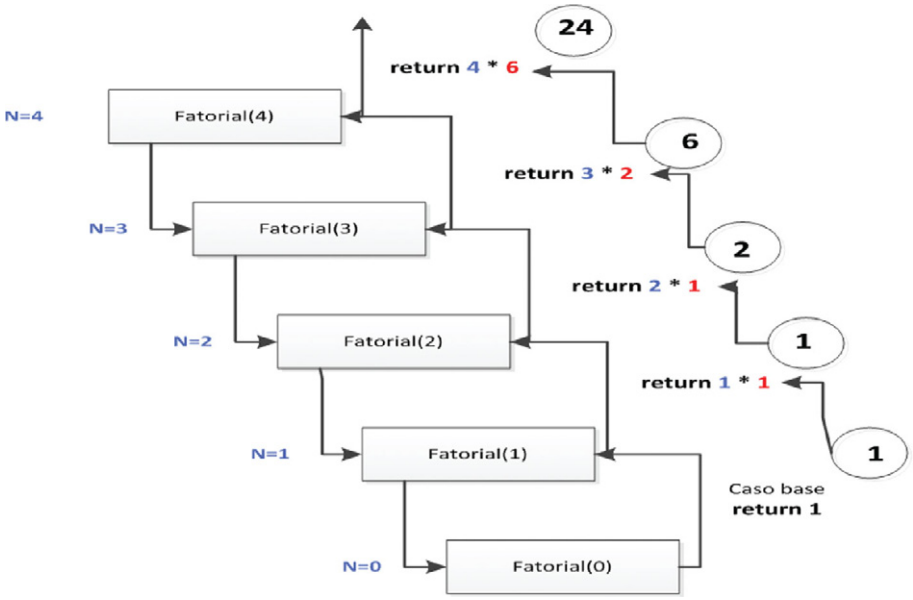

FIGURA 3.2: Sequência de chamadas da função fatorial (recursividade).

No Capítulo 2, mostramos como a memória é utilizada pelos processos de alocação dinâmica, pelas funções e suas variáveis. Fazendo uma breve recapitulação, no caso do cálculo do fatorial de um número *n*, observe que, para cada chamada da função fatorial, um novo espaço é alocado para empilhar essa nova função. Por isso, se o valor *n* for muito elevado, poderá ocorrer um estouro da pilha, resultando em erro de execução.

Diante disso, você pode, nesse momento, estar se perguntando: qual é a vantagem de utilizar uma função recursiva em comparação com uma não recursiva?

Embora a implementação recursiva, na maioria dos casos, seja mais simples que a versão iterativa, não existe nenhuma razão determinante para preferir a versão recursiva à iterativa. Em muitos casos, as versões recursivas consomem maior número de recursos (principalmente memória e processamento) e são muito mais difíceis de testar quando há muitas chamadas.

Entretanto, o que pode ser considerado positivo em sua utilização é a obtenção de códigos mais "enxutos" e mais fáceis de compreender, e, consequentemente, mais fáceis de implementar em linguagens de programação.

Atenção

A grande maioria dos algoritmos recursivos consome mais recursos computacionais. Por esse motivo, deve-se ter muita cautela ao utilizá-los.

Você deve utilizar a recursão quando:

1. o problema é naturalmente recursivo (clareza) e a versão recursiva do algoritmo não gera ineficiência evidente, se comparada com a versão iterativa;
2. o algoritmo se torna compacto, sem perda de clareza ou generalidade;
3. é possível prever que o número de chamadas (e, consequentemente, a alocação na pilha) não vão provocar interrupção no processo.

Você NÃO deve utilizar a recursão quando:

1. a solução recursiva causa ineficiência, se comparada com a versão iterativa;
2. existe uma única chamada do procedimento/função recursiva no fim ou no começo da rotina, com o procedimento/função podendo ser transformado numa iteração simples;
3. o uso de recursão acarreta número maior de cálculos que a versão iterativa;

4. a recursão é de cauda;
5. parâmetros consideravelmente grandes têm que ser passados por valor;
6. não é possível prever o número de chamadas que podem causar sobrecarga na pilha.

Papo técnico

Recursão de cauda é aquela que faz a chamada dos casos recursivos (ou valores subsequentes) no final da função, após testar ou apresentar o *n*-ésimo valor. Ou seja, não existe processamento a ser feito depois de encerrada a chamada recursiva. O cálculo do fatorial, a sequencia de Fibonacci, a soma de vetores são exemplos de recursão de cauda.

A grande vantagem da utilização de algoritmos recursivos é que, em certos casos, estes proporcionam uma forma mais simples e clara de se expressar em linguagem computacional (e, consequentemente, seu processamento). Contudo, isso não garante a melhor solução, principalmente em termos de tempo e consumo de memória.

É isso que podemos notar em outro exemplo clássico de algoritmos recursivos: o cálculo da sequência de Fibonacci. Quem leu o livro ou assistiu ao filme *Código Da Vinci*, história escrita por Dan Brown, deve se lembrar dessa sequência de números: 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, ... Essa sequência é composta por números naturais, e seus dois primeiros termos são 0 e 1. Cada termo subsequente corresponde à soma dos dois termos precedentes.

Matematicamente, teríamos o seguinte:

$$F(n)=\begin{cases} 0, & \text{se } n=0 \\ 1, & \text{se } n=1 \\ F(n-1)+F(n-2) & \text{se } n>1 \end{cases}$$

Note que a formulação matemática apresenta uma sequência definida recursivamente.

Portanto, pela formulação matemática, fica fácil convertê-la em uma implementação em linguagem computacional recursiva. O Código 3.6 apresenta essa função recursiva.

Código 3.6

```
...

int Fibonacci (int n)                          // função Fibonacci recursiva, recebe um valor inteiro como parâmetro.
{
      if ((n==0)||(n==1)) return n;                    // resolve os dois casos base (ou situação de parada)
      else return Fibonacci(n-1) + Fibonacci(n-2);     // caso recursivo (para o n-ésimo valor)
}
```

Para a resolução do caso n é preciso que os casos n-1 e n-2 sejam, ambos, resolvidos. Isso leva a uma quantidade de chamadas (a função Fibonacci) exponencial. Para entender isso melhor, suponha que você queira calcular, pelo método recursivo, a sequência dos 7 primeiros números da sequência. Considerando que o primeiro é 0, você passaria como parâmetro o valor de n = 6. A Figura 3.3 ilustra essa sequência de chamadas.

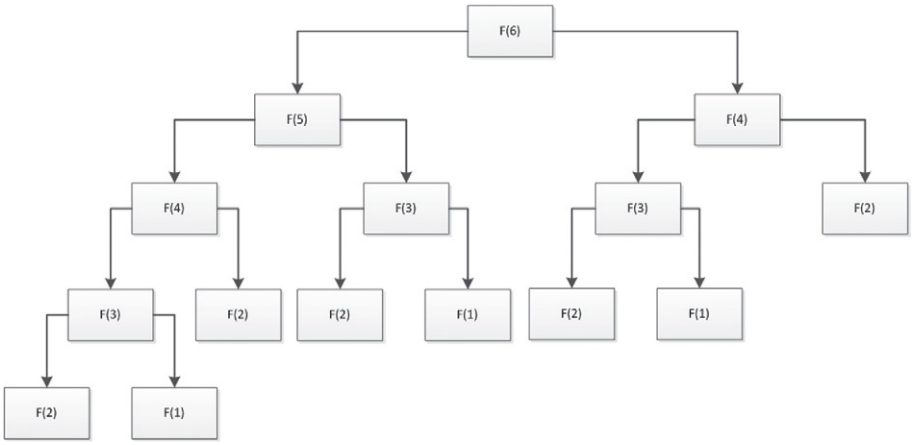

FIGURA 3.3: Sequência de chamada da função Fibonacci para n = 6.

Note que, antes de calcular Fibonacci(4), é preciso calcular Fibonacci(3) e Fibonacci(2), respectivamente, n-1 e n-2. Além disso, o cálculo não é reaproveitado. Por exemplo, Fibonacci(3), representado na Figura 3.2 por F(3), é calculado três vezes, e Fibonacci(2) – F(2) é calculado cinco vezes.

Em virtude dessa repetição de cálculos e chamadas exponenciais, a versão iterativa é muito mais eficiente. O Código 3.7 apresenta a versão iterativa do cálculo da sequência de Fibonacci.

Código 3.7

```
...

int Fibonacci (int n)                    // função Fibonacci iterativa, recebe um valor inteiro como parâmetro.
{
       int k, fant=0, fpos=1, f;
       if ((n==0)||(n==1)) return n;      // os dois casos base: F(0)=0 e F(1)=1.
       for (k=2; k<=n; k++) {
         f=fant+fpos;
         fant=fpos;
         fpos=f;
        }
       return f;       // retoma o valor do n número da sequência de Fibonacci
}
```

Tomando como base um computador desktop simples e comparando o tempo de execução da versão recursiva com a versão iterativa, teríamos algo semelhante ao apresentado na Tabela 3.2.

TABELA 3.2: Comparativo, em termos de tempo de execução, dos algoritmos recursivos e iterativos para cálculo da sequência de Fibonacci

n	20	30	50
Recursivo	1 s	2 min	21 dias
Iterativo	1/3 ms	1/2 ms	3/4 ms

Diferentemente desses dois exemplos (fatorial e Fibonacci), a solução recursiva para o problema ou quebra-cabeça conhecido como *Torre de Hanoi* apresenta, além de maior clareza e simplicidade, um desempenho equivalente ao da solução iterativa.

O problema da Torre de Hanoi foi publicado na forma de um jogo de tabuleiros em 1883, pelo matemático Edouard Lucas (com o codinome Professor N. Claus de Siam, e consistia em transferir, com o menor número possível de movimentos, a torre composta por *n* discos do pino *A* (origem) para o pino *C* (destino), utilizando o pino *B* como auxiliar. Apenas um disco poderia ser movido por vez. Além disso, um disco não poderia ser colocado sobre outro de menor diâmetro. A Figura 3.4 ilustra a situação inicial do problema.

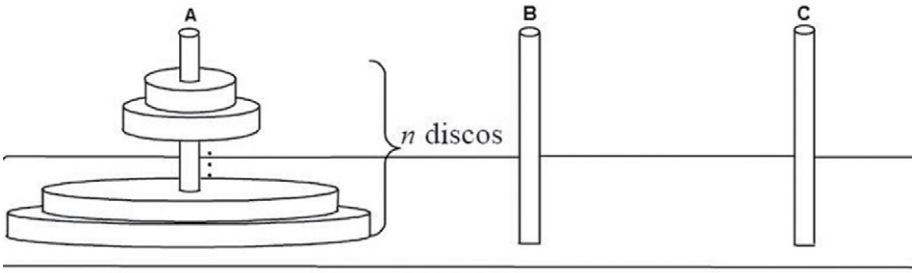

FIGURA 3.4: Situação inicial do problema da Torre de Hanoi.

A quantidade mínima de movimentos (*M*), dado um número *n* de discos, pode ser obtida pela seguinte equação matemática: $M = 2^n - 1$. Portanto, se tivermos 3 discos, o número mínimo de movimentos será 7. Se tivermos 5 discos, o número mínimo será 31, e assim por diante.

A ideia da utilização da recursão consiste em dividir um problema maior (*n* discos) em um problema menor (n - 1 discos) em cada uma das chamadas, até chegar ao problema mais simples: apenas um disco, que consiste em mover diretamente do pino *A* para o pino *C*.

Assim, se tivermos *n* discos, basta transferir os n - 1 discos de *A* para *B*, mover o maior disco de *A* para *C* e transferir os n - 1 discos de *B* para *C*. A cada passagem (transferir *x* discos), faríamos o decremento de um disco, até que restasse apenas um deles. A sequência de ilustrações na Figura 3.5, representa essa ideia.

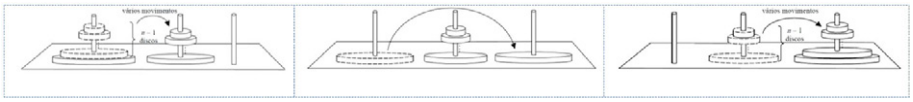

FIGURA 3.5: Ideia central para a resolução recursiva do problema da Torre de Hanoi.

O algoritmo recursivo que resolve esse problema é bem simples. Uma possível implementação é apresentada no Código 3.8, a seguir:

Código 3.8

```
...

void MoveTorre (int n, char Orig, char Dest, char Aux)  // função principal recursiva da Torre Hanoi
{
       if (n=1)
          MoveDisco(n, Orig, Dest);                    // caso base ou condição de parada (tem apenas um disco).
       else {
          MoveTorre(n-1, Orig, Aux, Dest);             // casos recursivos
          MoveDisco(n, Orig, Dest);
          MoveTorre(n-1, Aux, Dest, Orig);
       }
}

void MoveDisco (int disco, char Orig, char Dest)       // função que apenas mostra o movimento efetuado.
{
       printf("\nMovimento: Disco %i de %c --> %c", disco, Orig, Dest);
}
```

Seria bem interessante se você tentasse resolver esse mesmo problema sem utilizar a recursão. Você vai perceber que será extremamente difícil. E, como o problema da Torre de Hanoi, existem outros contextos nos quais a utilização de recursividade traz uma resolução mais simples e elegante, com a mesma eficiência.

Vamos programar

Para ampliar a abrangência do conteúdo apresentado, teremos, nas seções seguintes, implementações nas linguagens de programação JAVA e PHYTON.

Java

Código 3.1

```
public static int pot2 (int n)     // função potencia de 2, recebe um valor inteiro como parâmetro.
{
  int k, pot=1;
    if (n==0) return 1;            // resolve o caso base (ou situação de parada)
    else
     for (k=1; k<=n; k++) {
       pot = pot * 2;              // caso recursivo (para o n-ésimo valor)
    }
    return pot;
}
```

Código 3.2

```
public static int pot2 (int n)     // função potencia de 2 recursiva, recebe um valor inteiro como parâmetro.
{
       if (n==0) return 1;         // resolve o caso base (ou situação de parada)
       else return 2 * pot2(n-1); // caso recursivo (para o n-ésimo valor)
}
```

Código 3.3

```
public static int fatorial (int n)
{
      int fat=1;
       if (n==0) return fat;
       else
   while (n>0) {
      fat = fat * n;
      n--;
   }
      return fat;
}
```

Código 3.4

```
public static int fatorial (int n)     // função fatorial recursiva, recebe um valor inteiro como parâmetro.
{
       if (n==0) return 1;                  // resolve o caso base (ou situação de parada)
       else return n*fatorial(n-1);         // caso recursivo (para o n-ésimo valor)
}
```

Código 3.5

```
public static int fatorial (int n)      // função fatorial recursiva, recebe um valor inteiro como parâmetro.
{
L1     if (n==0)
L2          return 1;                       // resolve o caso base (ou situação de parada)
L3     else return n*fatorial(n-1);         // caso recursivo (para o n-ésimo valor)
}
```

Código 3.6

```
public static int Fibonacci (int n)  // função Fibonacci recursiva, recebe um valor inteiro como parâmetro.
{
       if ((n==0)||(n==1)) return n;              // resolve os dois casos base (ou situação de parada)
       else return Fibonacci(n-1) + Fibonacci(n-2);    // caso recursivo (para o n-ésimo valor)
}
```

Código 3.7

```
public static int Fibonacci (int n)  // função Fibonacci iterativa, recebe um valor inteiro como parâmetro.
{
       int k, fant=0, fpos=1, f;
       if ((n==0)||(n==1)) return n;       // os dois casos base: F(0)=0 e F(1)=1.
       for (k=2; k<=n; k++) {
         f=fant+fpos;aa
         fant=fpos;
         fpos=f;
        }
       return f;       // retorna o valor do n número da sequência de Fibonacci
}
```

Código 3.8

```
public static void MoveTorre (int n, char Orig, char Dest, char Aux)
// função principal recursiva da Torre Hanoi
{
       if (n==1)
          MoveDisco(n, Orig, Dest);              // caso base ou condição de parada (tem apenas um disco).
       else {
          MoveTorre(n-1, Orig, Aux, Dest);       // casos recursivos
          MoveDisco(n, Orig, Dest);
          MoveTorre(n-1, Aux, Dest, Orig);
       }
}

public static void MoveDisco (int disco, char Orig, char Dest)
// função que apenas mostra o movimento efetuado.
{
  System.out.println("\nMovimento: Disco " + disco + " de " + Orig + " --> " + Dest);
}
```

Python

Código 3.1

```
def pot2(n):
  k=1
  pot=1
  if n==0:
    return 1
  else:
    while k<=n:
      pot = pot * 2
      k=k+1
  return pot
```

Código 3.2

```
def pot2(n):
  if n==0:
    return 1
  else:
    return 2 * pot2(n-1)
```

Código 3.3

```
def fatorial(n):
  fat=1
  if n==0:
    return fat
  else:
    while n>0:
      fat = fat * n
      n-=1
  return fat
```

Código 3.4

```
def fatorial(n):
  if n==0:
    return 1
  else:
    return n*fatorial(n-1)
```

Código 3.5

```
def fatorial(n):
L1  if n==0:
L2    return 1
    else:
L3    return n*fatorial(n-1)
```

Código 3.6

```
def Fibonacci(n):
  if n==0 or n==1:
    return n
  else:
    return Fibonacci(n-1) + Fibonacci(n-2)
```

Código 3.7

```
def Fibonacci(n):
  k=2
  fant=0
  fpos=1
  f=0
  if n==0 or n==1:
    return n
  while k<=n:
    f=fant+fpos
    fant=fpos
    fpos=f
    k+=1
  return f
```

Código 3.8

```
def MoveTorre(n,Orig,Dest,Aux):
  if n==1:
    MoveDisco(n,Orig,Dest)
  else:
    MoveTorre(n-1, Orig, Aux, Dest)
    MoveDisco(n,Orig,Dest)
    MoveTorre(n-1,Aux,Dest,Orig)

def MoveDisco(disco,Orig,Dest):
  print "Movimento: Disco %d de %s --> %s" %(disco,Orig,Dest)
```

Para fixar

1. Faça uma função recursiva para calcular o máximo divisor comum (MDC) de dois números inteiros não negativos, n e m, usando o algoritmo de Euclides:

$$mdc(m,n) = \begin{cases} mdc(n,m) & se\ n > m, \\ m & se\ n = 0, \\ mdc(n,(m \bmod n)) & se\ n > 0. \end{cases}$$

Vamos lá, você consegue!

Papo técnico

Você pode medir o desempenho dos algoritmos utilizando funções existentes nas bibliotecas das principais linguagens de programação. Essas funções pegam a hora do sistema. Faça isso antes de chamar a função e imediatamente após seu retorno. A diferença entre os dois valores é o tempo gasto pelo algoritmo (geralmente em milissegundos).

Para saber mais

Você poderá encontrar mais informações sobre algoritmos recursivos em diversos livros, entre eles *Algoritmos: teoria e prática* (Cormen *et al.* 2012) e *Introdução à ciência da computação* (Mokarzel e Soma, 2008).

Navegar é preciso

Com uma linguagem bem-humorada, o site *Learnyou some Erlang* (http://learnyousomeerlang.com/recursion) trata o tema de recursão com uma apropriada profundidade. Além disso, existem vários objetos de aprendizagem para a demonstração das implementações de algoritmos recursivos. Um site que traz várias implementações de simulações de algoritmos recursivos, divididos em simples, intermediários e avançados, é o http://www.animatedrecursion.com/

Outra animação bem interessante é o da implementação do problema da Torre de Hanoi. Esse objeto de aprendizagem pode ser acessado em http://www.cut-the-knot.org/recurrence/hanoi.shtml

Exercícios

1. Faça uma função recursiva para calcular a potência de um número, considerando que o expoente é um número natural. Utilize o método das multiplicações sucessivas.

2. Faça uma função recursiva que converta um número na base 10 para a base 2.

3. Faça uma função recursiva que retorne o maior valor armazenado em um vetor *V*, contendo *n* números inteiros.

Glossário

Teste de mesa: metodologia de teste de um algoritmo sem a utilização de um computador. Para tanto, devemos identificar as variáveis envolvidas, criar uma tabela identificando as variáveis, as instruções, os valores atribuídos à variável em cada uma das linhas de código, valores de entrada e de retorno das funções etc. Também devemos percorrer o algoritmo, passo a passo, identificando todas as variações provocadas pelo mesmo.

Referências bibliográficas

CORMEN, T. H. et al. *Algoritmos: teoria e prática*. Rio de Janeiro: Elsevier, 2012.

MOKARZEL, F. C.; SOMA, N. Y. *Introdução à ciência da computação*. Rio de Janeiro: Elsevier, 2008.

SCHILDT, H. *C, completo e total*. 3. ed. São Paulo: Makron Books, 1996.

TENENBAUM, A. M. *Estruturas de dados usando C*. São Paulo: Makron Books, 1995.

O QUE VEM DEPOIS

Agora você já sabe como funcionam os algoritmos recursivos, como implementá-los, e também quando usar e não usar essa técnica. Chegou a hora de nos aprofundar nas técnicas de manipulação de estruturas de dados efetivamente. Para começar, vamos entender como podemos recuperar uma informação, estando ela armazenada em determinada ordem ou não. Preparado? Então, bons estudos!