Capítulo 2

Técnicas de programação recursiva

Você já deve ter visto, à venda em supermercados ou lojas de utilidades domésticas, um conjunto de travessas de diferentes tamanhos, mas com a mesma estrutura, encaixadas uma dentro da outra. Ou já deve ter notado que, quando se posiciona um espelho em frente a outro, a imagem refletida se torna infinita e menor a cada reflexo, não passando de um pixel ao final. Esse processo que ocorre em ambos os exemplos é chamado de recursão. A recursão está presente em diversas situações do dia a dia, principalmente em princípios matemáticos e, por consequência, no mundo dos algoritmos computacionais.

Algoritmos recursivos são programas com chamadas hierárquicas nas quais um método chama a ele próprio em sua implementação. A

recursão pode ocorrer de maneira direta ou indireta, sendo a segunda executada por um método intermediário (DEITEL; DEITEL, 2016).

Ao longo deste capítulo, serão apresentados os conceitos principais de um algoritmo e como ele pode ser utilizado para deixar o código mais elegante e de fácil compreensão. Também abordaremos o que deve ser evitado na definição de um algoritmo recursivo, sem que ele perca alguns de seus benefícios.

1 Definição da recursão

O primeiro ponto relevante ao se trabalhar com recursão é saber que ela somente pode resolver um caso simplificado do problema em si, que é chamado de caso básico. Sempre que o método recursivo encontrar um caso básico do problema, ele consegue retornar um resultado definitivo. Como consequência, os demais casos, os casos recursivos, obtêm sua resolução. Assim, toda vez que um problema complexo é apresentado, este deve ser dividido em pequenos subproblemas, mais simples, para que o problema original seja resolvido. Os subproblemas devem manter a mesma estrutura do original, sendo apenas um pouco mais simples ou menores, em termos de processamento (DEITEL; DEITEL, 2016).

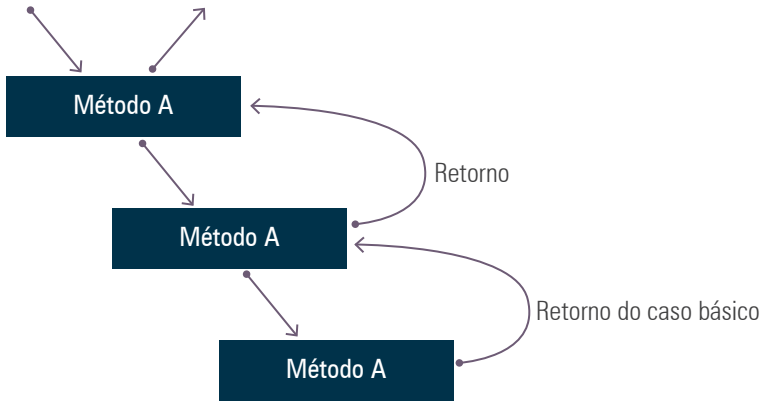
IMPORTANTE

Ao analisar problemas complexos e algoritmos com muitos laços aninhados, estes podem ter uma solução recursiva mais simples do que a solução inicial (GOODRICH; TAMASSIA, 2013).

O passo recursivo geralmente possui uma instrução do tipo *return*, cuja execução se encerra retornando o valor do resultado do processamento, o que torna possível a combinação entre os subproblemas ao obter o resultado do caso básico (WIRTH, 1989; DEITEL; DEITEL, 2016).

Esse passo é executado sempre que um método realiza uma chamada a ele mesmo na sua implementação. Assim como um algoritmo iterativo utiliza laços de repetições, a recursão também deve possuir um critério de parada, que determinará o fim de sua execução. Para que isso aconteça, cada chamada recursiva deve ser realizada com uma versão mais simples que o problema original. A versão mais simples, feita em cada passo da execução, levará o algoritmo a atingir o caso básico. Sendo assim, uma solução é encontrada e retornada para o resultado final (DEITEL; DEITEL, 2016). Veja na figura 1 a ilustração de uma recursão simples.

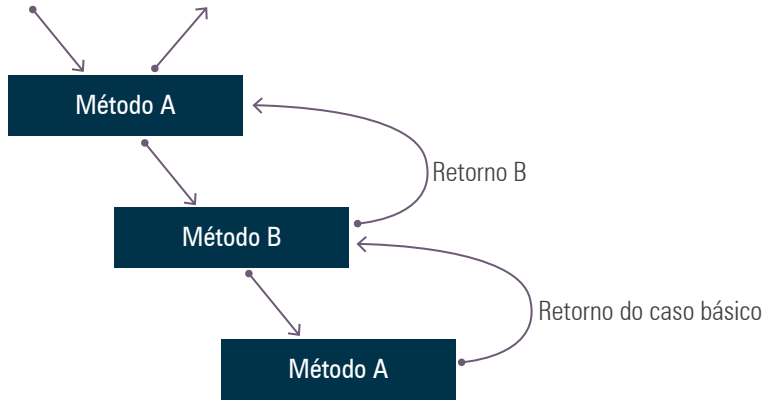
Figura 1 – Exemplo de recursão direta ou simples



Além de o método realizar uma chamada a si próprio, a recursão direta, pode ocorrer em um algoritmo o que se chama de recursão indireta. Esse tipo de recursão é feita através de um método intermediário. O funcionamento ocorre da seguinte maneira: o método A (recursivo) realiza a chamada a um método B (intermediário). Na implementação do método B, existe uma nova chamada recursiva ao método A. Ao observar esse fluxo, nota-se que, mesmo com a nova chamada ao método A através de B, a primeira chamada de A continua em aberto e aguardando o resultado retornado de B. Nesse momento, ocorre o processo recursivo, porém com um segundo método intermediando. Por isso, o

nome "recursão indireta" (GOODRICH; TAMASSIA, 2013; DEITEL; DEITEL, 2016). Veja a ilustração de uma recursão indireta na figura 2.

Figura 2 – Exemplo de recursão indireta



Em problemas complexos, a aplicação de recursão na solução traz o benefício de deixar o algoritmo legível e ainda a possibilidade de se manter uma solução eficiente em termos de processamento (GOODRICH; TAMASSIA, 2013).

Um exemplo de aplicação da recursão em computação é o sistema de arquivos existente em qualquer sistema operacional. Este é iniciado por um diretório ou pasta principal. Nele são armazenados arquivos e novos diretórios (recursão), os quais também possuem outros arquivos e diretórios. Esse processo ocorre até que o caso básico seja obtido. Na questão do sistema de arquivos, o caso básico é um diretório que possui em sua raiz apenas arquivos armazenados (GOODRICH; TAMASSIA, 2013; DEITEL; DEITEL, 2016).

2 Traga a matemática à vida

Neste tópico, apresentaremos alguns conceitos matemáticos que possuem definições recursivas, bem como suas respectivas implementações,

utilizando a linguagem de programação Java. Em alguns momentos, dicas para melhorar o desempenho do algoritmo nessa linguagem serão demonstradas.

2.1 Fatorial: o clássico da recursão

O problema do cálculo de um número fatorial é bem discutido em qualquer curso da área de exatas. A função fatorial é definida formalmente assim: dado um número inteiro e positivo chamado *n*, seu fatorial é obtido pelo produto de todos os números inteiros entre *n* e 1 (TENENBAUM; LANGSAM; AUGENSTEIN, 2004). Em termos matemáticos, a definição é:

$$n! = \prod_{k=1}^{n} k \ \forall n \in \mathbb{N}$$

Em que a função fatorial é definida por um número seguido de um ponto de exclamação. Seu resultado é dado pelo produto da variável *k* iniciando em 1 e percorrendo até *n*. Isso acontece para todo *n* pertencente ao conjunto dos números naturais. Veja o exemplo de 5! = 5 · 4 · 3 · 2 · 1 = 120. Perceba que o 5! pode ser definido de maneira recursiva, na qual 5! = 5 · 4!. A regra da função fatorial é então definida da seguinte maneira:

$$n! \begin{cases} 1 & se\ n = 0 \\ n \cdot (n-1)! & se\ n > 0 \end{cases}$$

O caso básico da função recursiva fatorial ocorre quando o valor de *n* é igual a 0. Nesse caso, a função fatorial retorna 1 por ser um valor indiferente à operação de multiplicação. Para todos os outros casos, em passos recursivos, o retorno é *n* multiplicado pelo valor do fatorial de *n* – 1. Veja a implementação da classe FuncaoFatorial:

```
1  public class FuncaoFatorial {
2      // função recursiva
3      public static int fatorial(int n) {
4             if(n <= 1) return 1; // caso básico
5             return n * fatorial(n - 1); // passo
recursivo
6      }
7
8      public static void main(String[] args) {
9             // cálculo do fatorial de 0 a 5
10            for(int n = 0; n <= 5; n++)
11                   System.out.printf("%d! =>
%d\n", n, fatorial(n));
12     }
13 }
```

A classe implementada apresenta dois métodos. O primeiro, definido entre as linhas 3 e 6, é o método recursivo para o cálculo do fatorial de um número *n* qualquer. Na linha 4, é determinado o caso básico que verifica se *n* é menor ou igual a 1, retornando assim o próprio valor 1 (o menor valor de um fatorial). Na linha 5, é retornado o valor de *n* multiplicado pelo resultado da chamada recursiva para o fatorial de *n* – 1. O método *main*, definido entre as linhas 8 e 12, apenas executa a chamada para o cálculo dos fatoriais de 0 até 5. O resultado é apresentado abaixo:

```
0! => 1
1! => 1
2! => 2
3! => 6
4! => 24
5! => 120
```

Perceba que foi utilizado nas declarações de métodos e variáveis o tipo *int*. Isso pode gerar um problema de memória no cálculo fatorial. A partir do 12!, a variável do tipo *int* não consegue mais armazenar valores, ocorrendo um estouro de memória. A solução paliativa é efetuar a troca

para variáveis do tipo *long*, mas perceba que, desse modo, só será possível efetuar o cálculo até o 21!. Após isso, o estouro de memória ocorre novamente. No Java, existem duas classes que podem auxiliar em cálculos de problemas que possuem a precisão arbitrária e não podem ser feitos através dos tipos primitivos. São elas: BigInteger e BigDecimal, ambas encontradas no pacote java.math (DEITEL; DEITEL, 2016).

PARA SABER MAIS

Para compreender melhor os objetos BigInteger e BigDecimal utilizados nas implementações, acesse os links a seguir:

- https://docs.oracle.com/javase/8/docs/api/java/math/class-use/BigInteger.html (acesso em: 20 abr. 2020).
- https://docs.oracle.com/javase/8/docs/api/java/math/class-use/BigDecimal.html (acesso em: 20 abr. 2020).

Agora, acompanhe a reimplementação do algoritmo recursivo fatorial utilizando BigInteger como tipo:

```
1  import java.math.BigInteger;
2
3  public class FuncaoFatorial {
4      // função recursiva
5      public static BigInteger fatorial(BigInteger n) {
6             if(n.compareTo(BigInteger.ONE) <= 0) // caso
básico
7                    return BigInteger.ONE;
8             return n.multiply( // passo recursivo
9                           fatorial(n.
subtract(BigInteger.ONE)));
10     }
11
```

```
12     public static void main(String[] args) {
13            // cálculo do fatorial de 0 a 5
14            for(int n = 0; n <= 5; n++)
15                   System.out.printf("%d! => %d\n", n,
16                                fatorial(BigInteger.
valueOf(n)));
17     }
18 }
```

Algumas particularidades sobre o código refatorado: (a) por não se tratar de um tipo primitivo, o uso dos operadores comuns não é possível nesse caso. (b) Na linha 6, o método *compareTo* retorna –1 se *n* for menor que 1, e nesse caso foi utilizada a constante que representa o valor 1 para o objeto BigInteger. Também retorna 0 se forem iguais e 1 se for maior. (c) Nas linhas 8 e 9, utilizam-se os métodos *multiply* e *subtract* para calcular o fatorial de *n*.

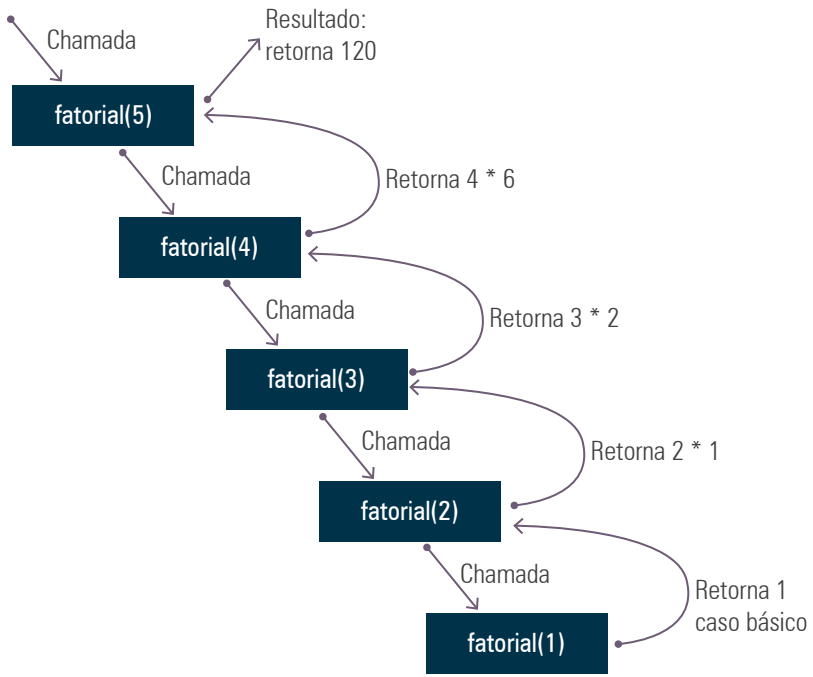
IMPORTANTE

Um erro muito comum ao trabalhar com algoritmos recursivos é omitir o caso básico. Esse erro pode levar a um problema chamado de recursão infinita. É um erro parecido com o do *loop* infinito em laços de repetição. Contudo, a recursão infinita pode gerar um estouro de memória de maneira muito mais rápida que um laço de repetição (DEITEL; DEITEL, 2016).

Entender um algoritmo recursivo muitas vezes não é uma tarefa trivial e conseguir visualizar todos os caminhos tomados pelo algoritmo pode ser algo bem complicado. Uma ferramenta que pode auxiliar nessa tarefa é chamada de rastreamento recursivo (GOODRICH; TAMASSIA, 2013). Basicamente, essa ferramenta desenha um quadro para cada chamada recursiva, conectando-as através de uma seta reta que representa cada chamada. No final, uma seta curva é retornada ao quadro

acima para representar o resultado do valor obtido naquele passo. A figura 3 apresenta o rastreamento recursivo para o fatorial do número 5:

Figura 3 – Rastreamento recursivo da função fatorial para o número 5



Na figura 3, ocorre uma chamada (seta direcional) de um método qualquer para a função fatorial, passando o valor 5 como parâmetro (retângulo representando o método). A função fatorial realiza uma nova chamada para ela mesma passando como parâmetro agora o valor 4, representando a subtração do número informado inicialmente em uma unidade. Esse processo se repete até que a função fatorial execute uma chamada com o parâmetro de valor 1. Nesse momento, pela implementação realizada, atinge-se o caso básico. Isso significa que as chamadas recursivas foram finalizadas. Então, retornam os resultados de cada método para o método anterior que realizou a chamada. Inicia-se o retorno do próprio valor 1; o retorno é o resultado da operação de multiplicação de 1 por 2, representando o fatorial de 2. Isso se repete até chegar

ao primeiro nível de chamada, o fatorial de 5. O resultado esperado é 120, passado ao programa principal ou ao método que realizou a primeira chamada para a função fatorial. Apesar de ser um modelo de entrada em algoritmos recursivos, o fatorial não é a única aplicação para essa técnica. Algumas outras aplicações serão abordadas na sequência.

2.2 A beleza de Fibonacci

A série de Fibonacci é uma sequência de números muito utilizada em diversas áreas, envolvendo principalmente arte e computação. Acredita-se que ela seja a medida para a beleza perfeita. Já na natureza, a série de Fibonacci é utilizada para descrever o comportamento de uma espiral. Ela se inicia com 0 e 1, sendo cada número subsequente o cálculo da soma entre os dois anteriores. A sequência converge para um valor constante de 1,618, aproximadamente. Essa constante é chamada de média áurea ou relação áurea (DEITEL; DEITEL, 2016).

Sua definição formal é:

$$fibonacci(n)\begin{cases} 0 & se\ n = 0 \\ 1 & se\ n = 1 \\ fibonacci(n-1) + fibonacci(n-2) & para\ todo\ n > 1 \end{cases}$$

Perceba que existem dois casos básicos no cálculo do número de Fibonacci. Os casos básicos são exatamente o resultado de *fibonacci*(1) e *fibonacci*(0), que retornam o mesmo valor de *n* recebido, 1 e 0, respectivamente. Acompanhe a implementação a seguir que utiliza a chamada do método de Fibonacci para o cálculo dos representantes de número 0 até 10.

```
1. import java.math.BigInteger;
2.
3. public class NumeroFibonacci {
4.
```

```
5.     // Constante de apoio
6.     private static final BigInteger TWO = BigInteger.
valueOf(2);
7.
8.     // Declaracao do método de fibonacci recursivo
10.    public static BigInteger fibonacci(BigInteger n) {
11.           // Casos básicos
12.           if (n.equals(BigInteger.ZERO) ||
13.                        n.equals(BigInteger.ONE)) {
14.                  return n;
15.           }
16.           // Chamada recursiva de Fibonacci
17.           return fibonacci(n.subtract(BigInteger.ONE))
18.                        .add(fibonacci(n.
subtract(TWO)));
19.    }
20.
21.    public static void main(String[] args) {
22.           for(int i = 0; i <= 10; i++)
23.                  System.out.printf("Fibonacci(%d) =>
%d\n", i,
24.                                fibonacci(BigInteger.
valueOf(i)));
25.    }
26.}
```

A implementação do cálculo do valor de Fibonacci utiliza o objeto do tipo BigInteger para evitar o problema de armazenamento na memória do computador, como o cálculo do fatorial anteriormente apresentado. Analisando alguns pontos importantes na implementação, nota-se que nas linhas 17 e 18 o método de Fibonacci é invocado recursivamente duas vezes. Como curiosidade, a chamada da linha 24 não é recursiva e é a chamada que origina o processo de recursão. O resultado apresentado pela implementação é:

```
Fibonacci(0) => 0
Fibonacci(1) => 1
Fibonacci(2) => 1
Fibonacci(3) => 2
Fibonacci(4) => 3
```

```
Fibonacci(5) => 5
Fibonacci(6) => 8
Fibonacci(7) => 13
Fibonacci(8) => 21
Fibonacci(9) => 34
Fibonacci(10) => 55
```

Ao invocar duas vezes o método, é necessário que haja um cuidado redobrado sobre o consumo de memória gerado por um método recursivo. Para exemplificar melhor essa situação, observe que, ao solicitar o cálculo do valor de Fibonacci para 30, são efetuadas 2.692.537 chamadas. O valor para 32 é calculado com 7.049.155 chamadas (DEITEL; DEITEL, 2016). Perceba na diferença entre os dois números que a quantidade de chamadas praticamente triplicou ao longo do processo. Isso significa que a quantidade de chamadas é $nk > 2k/2$, em outras palavras, o número de chamadas é exponencial em relação a k (GOODRICH; TAMASSIA, 2013).

Com todas essas informações é possível dizer que a implementação de Fibonacci através de recursão não deve ser considerada. Um método recursivo ocupa mais memória que um método iterativo, uma vez que para cada chamada do método são criadas novas instâncias de todas as informações presentes nele. Isso pode gerar um estouro de memória mais rápido do que um simples laço de repetição infinito. Nos tópicos seguintes, serão discutidos os tipos de recursão mais comuns, neles apresentamos uma forma de solucionar o problema de memória discutido no algoritmo recursivo de Fibonacci.

3 Tipos de recursão

Existem basicamente três tipos de recursão. São eles: linear, binária e múltipla. Ao longo do capítulo, alguns exemplos de cada um dos tipos serão apresentados para auxiliar na compreensão e principalmente para explicar como utilizar cada um deles na solução de problemas.

3.1 Recursão linear

Pode-se afirmar que um problema é resolvido por uma recursão linear quando um método possui apenas uma chamada recursiva em sua execução (GOODRICH; TAMASSIA, 2013). Um exemplo já apresentado aqui é a função fatorial. Um outro problema resolvido a partir de uma recursão linear é o somatório de um vetor. Esse é um problema bem simples que poderia ser resolvido de maneira iterativa, mas, como processo didático para a compreensão da recursão linear, ele será apresentado como um algoritmo recursivo.

A entrada do problema será um vetor *v* e um inteiro *n*, sendo $n \geq 1$. A implementação é apresentada a seguir:

```
1.  public class Somatorio {
2.      // Recursão linear para função do somatório
3.      public static int somatorio(int[] v, int n) {
4.              // Caso básico
5.              if (n == 1) return v[0];
6.              // Passo recursivo
8.              return somatorio(v, --n) + v[n];
9.      }
10.     // Aplicacao da função
11.     public static void main(String[] args) {
12.             int[] v = {1, 3, 5, 7, 9, 11, 13};
13.             System.out.printf("O valor do somatório é
%d\n",
14.                             somatorio(v, v.length));
15.     }
16. }
```

No algoritmo, são trabalhados valores do tipo inteiro, pois não existe a necessidade de se armazenar números muito grandes nesse problema. Em um caso mais complexo, valores do tipo *long* já seriam suficientes para se trabalhar com ele. Da linha 3 até a linha 9, é implementada a função de somatório recursiva. Ela recebe como parâmetro os valores de *v*,

que é um vetor de inteiros, e *n*, que representa o tamanho do vetor. A linha 5 apresenta a definição do caso base, que é exatamente quando resta apenas um elemento no vetor. Nesse momento, este elemento deve ser retornado. O passo recursivo encontra-se implementado na linha 8 do código apresentado. Nele, é implementada a chamada recursiva do método somatório, passando como parâmetro o vetor e *n* subtraído em 1 para representar o vetor diminuindo de tamanho e o próprio *n*.

PARA PENSAR

Ao desenvolver um código, é importante pensar em como o compilador irá lê-lo. Essa leitura sempre se inicia da esquerda para a direita. Em Java, utilizar o atalho de incremento (++) ou de decremento (--) tem efeitos distintos em relação ao posicionamento da variável associada. Caso a variável venha depois dessa instrução, a operação é realizada primeiro. Após a operação, o programa irá consultar o valor armazenado na variável, como no caso da linha 8 do algoritmo somatório utilizando recursão.

Temos uma consulta de valor primeiro à variável e em seguida o efeito da operação, caso o atalho de incremento ou decremento esteja posicionado depois da variável associada.

Em relação à recursão linear, ainda podemos apresentar o conceito de recursão final. A recursão final ocorre quando o último passo de um algoritmo recursivo é a chamada recursiva (GOODRICH; TAMASSIA, 2013). A somatória e o fatorial, apresentados anteriormente, não atendem a essa característica, pois ambos possuem uma operação matemática antes de realizar o retorno do método.

Uma das operações que podem ser exemplificadas como recursão final é a inversão de um vetor. Esse algoritmo deve receber um vetor, sua posição inicial e sua posição final. Após sua chamada, ele começa o processo de trocar os valores presentes nas posições inicial e final, e o processo se repete até que a posição inicial seja maior ou igual à posição final do vetor.

Confira a implementação do método *inverter* da classe Vetor:

```
1.  import java.util.Arrays;
2.
3.  public class Vetor {
4.      // Método auxilar para troca de posição
5.      public static void troca(int[] v, int posA, int
posB) {
6.              int aux = v[posA];
7.              v[posA] = v[posB];
8.              v[posB] = aux;
9.      }
10.
11.     public static void inverter(int[] v, int inicio,
int fim) {
12.             if(inicio >= fim) return; //caso básico
13.             troca(v, inicio, fim);
14.             inverter(v, ++inicio, --fim); //passo
recursivo
15.     }
16.
17.     public static void main(String[] args) {
18.             int[] v = {1, 3, 5, 7, 9, 11, 13};
19.             inverter(v, 0, v.length - 1);
20.             System.out.println(Arrays.toString(v));
21.     }
22. }
```

O algoritmo recursivo está implementado entre as linhas 11 e 15. O ponto importante ocorre exatamente na linha 14, antes de o bloco ser concluído. Nesse ponto, nada além da chamada recursiva é executado. Essa é a principal característica para a definição de uma recursão final.

3.2 Recursão binária

A recursão é chamada de binária quando o algoritmo recursivo possui duas chamadas recursivas dentro de si (GOODRICH; TAMASSIA, 2013).

A essa altura, já sabemos que o cálculo do valor de Fibonacci, apresentado anteriormente, é um algoritmo de recursão binária. Notamos também que este não é um algoritmo eficiente, devido à quantidade de chamadas recursivas ao longo de seu processamento. Mas existem algoritmos de recursão binária que continuam eficientes durante todo o processo. Um exemplo simples é a somatória binária. Essa somatória altera um pouco a original, que apresentamos durante o tópico de recursão linear.

A primeira alteração são os parâmetros recebidos pelo método. Na versão linear, apenas o vetor e seu fim eram informados. Para a versão binária, é necessário informar, além do vetor, sua posição inicial e seu tamanho. O algoritmo irá dividir o vetor ao meio, até que este tenha o tamanho de um elemento apenas. Esse é o caso básico. No retorno dos passos recursivos, é feita a soma de cada um desses elementos para chegar ao resultado final do problema.

Veja a implementação da somatória binária a seguir:

```
1.  public class Somatorio {
2.      // Recursão binária para função do somatório
3.      public static int somatorio(int[] v, int i,int n) {
4.              // Caso básico
5.              if (n == 1) return v[i];
6.              // Passo recursivo
7.              return somatorio(v, i, (n + 1)/2)
8.                              + somatorio(v, i + (n + 1)/2,
n/2);
9.      }
10.     // Aplicação da função
11.     public static void main(String[] args) {
12.             int[] v = {1, 3, 5, 7, 9, 11, 13};
13.             System.out.printf("O valor do somatório é
%d\n",
14.                             somatorio(v, 0, v.length));
15.     }
16. }
```

A função recursiva é definida entre as linhas 3 e 9. Na linha 5, é verificado o caso básico, em que, se *n* (tamanho do vetor) for igual a 1, retorna-se o valor armazenado na posição *i* do vetor *v*. As linhas 7 e 8 estão com as chamadas recursivas da função. Como são duas as chamadas recursivas, trata-se de uma recursão binária.

Na primeira chamada, são passados o valor de *i* (início do vetor) e a posição referente ao meio do vetor. O valor obtido deve ser arredondado para cima, atingindo assim o teto do valor decimal. A operação matemática pra realizar esse arredondamento é (número + (divisor – 1))/divisor; como o divisor é 2, um atalho foi adotado ao somar 1 diretamente em *n* para obter o resultado desejado. Na chamada recursiva apresentada na linha 8, o meio arredondado para cima é informado como o início do vetor, e o meio com arredondamento para baixo define o tamanho do vetor. Assim, os vetores são divididos até que reste apenas um elemento e este seja retornado somando com o elemento da chamada ao lado.

3.3 Recursão múltipla

A recursão múltipla ocorre quando um método necessita realizar várias chamadas dentro dele próprio (GOODRICH; TAMASSIA, 2013). Quando um método de recursão binária é apresentado, seu espaço de memória pode facilmente chegar a um problema exponencial, assim como ocorre em Fibonacci. Uma recursão múltipla é bem mais do que o dobro de chamadas recursivas, então é sempre bom analisar o problema para tentar diminuir sua complexidade ao final.

Esse tipo de recursão é muito utilizado em soluções de jogos conhecidos como *puzzles*, que possibilitam a análise de diversos caminhos que podem ser seguidos para se encontrar a solução.

4 Iteração ou recursão? Eis a questão

Quando se analisa um problema, tanto de maneira iterativa como por meio da recursão, deve-se ter em mente que ambos possuem sua base em uma instrução de controle. A instrução de controle pode ser chamada de critério de parada, no caso dos laços de repetição como *for*, *while* e *do-while*. Já para os casos recursivos, as instruções de controle são conhecidas como caso básico.

A instrução de controle serve para auxiliar o término do procedimento, assim é possível evitar que entrem no chamado *loop* infinito ou em recursão infinita. O objetivo de cada uma é um pouco diferente quando pensamos em sua construção. A versão iterativa utiliza uma variável de controle para verificar quando o algoritmo chegou ao fim, por exemplo, um contador de passos executados. A versão recursiva sempre divide o problema em versões mais simples e menores para que possa ser resolvido de maneira mais fácil.

Quando a solução é iterativa, ela geralmente consome menos memória que uma versão recursiva. Contudo, muitas vezes ela pode ser uma solução muito complexa de ser implementada e, às vezes, até mesmo de ser encontrada. A solução recursiva pode compensar o problema de memória utilizando uma infraestrutura mais avançada como um *cluster* de computadores. Assim, ela pode processar cada chamada recursiva em computadores diferentes, melhorando o desempenho do processamento. Encontrar uma solução recursiva é mais fácil e muitas vezes mais elegante que a iterativa.

Então qual deve ser escolhida? Não existe uma resposta exata a esta pergunta, mas sim uma recomendação de boa prática. Se a solução iterativa for tão clara quanto a recursiva, opte sempre por ela. A recursão foi desenvolvida para simplificar problemas maiores que não são trivialmente solucionados por um algoritmo iterativo.

5 Exercícios de fixação

Para praticar, segue uma lista de exercícios:

1. Implemente o fatorial de maneira iterativa e compare com a implementação recursiva. Utilize o gerenciador de processos para comparar o consumo de memória e o processamento.
2. Implemente o cálculo de Fibonacci de maneira iterativa e compare com a implementação recursiva. Utilize o gerenciador de processos para comparar o consumo de memória e o processamento.
3. A implementação recursiva de Fibonacci mais facilmente encontrada é a binária. Porém, o problema de Fibonacci é um problema linear. Dessa maneira, pode-se deduzir que sua implementação pode ser realizada de maneira recursiva linear. Faça a implementação.
4. Implemente a busca sequencial de maneira recursiva, se possível.
5. Implemente a busca binária de maneira recursiva, se possível.
6. Resolva de maneira recursiva um produtório, a partir de valores armazenados em um vetor.
7. Resolva o produtório como uma recursão binária.

Considerações finais

Neste capítulo, foram apresentados os conceitos de recursão, suas vantagens e desvantagens. Problemas matemáticos foram apresentados para demonstrar como questões do dia a dia já utilizam a teoria de recursividade para serem resolvidas. Questões importantes relacionadas ao consumo de memória, entre outros fatores, foram

apresentadas para auxiliar a tomada de decisão sobre como e quando utilizar um algoritmo recursivo.

Alguns exemplos foram implementados para apresentar os tipos de recursão e quais são os pontos de atenção para cada um deles. Por fim, uma comparação com algoritmos iterativos foi apresentada para auxiliar a tomada de decisão para o uso de algoritmos em projetos complexos. É importante compreender o problema a ser resolvido para se adotar a melhor estratégia para realizar sua implementação.

Referências

DEITEL, Paul J.; DEITEL, Harvey M. **Java**: como programar. 10. ed. São Paulo: Pearson Education, 2016.

GOODRICH, Michael T.; TAMASSIA, Roberto. **Estruturas de dados & algoritmos em Java**. Porto Alegre: Bookman Editora, 2013.

TENENBAUM, Aaron M.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. **Estruturas de dados usando C**. São Paulo: MAKRON Books, 2004.

WIRTH, Niklaus. **Algoritmos e estruturas de dados**. Rio de Janeiro: Prentice Hall do Brasil, 1989.