Apontamentos de Estruturas de Dados

Recursividade

Ricardo Santos| rjs@estg.ipp.pt

Escola Superior de Tecnologia e Gestão
Instituto Politécnico do Porto

Última versão: Outubro de 2022

Índice

9. Recursividade

9.1. Introdução

Um algoritmo é recursivo quando se chama a si mesmo para fazer parte do seu trabalho. Para que essa abordagem seja bem-sucedida, a "chamada para si mesmo" deve tratar um problema menor do que o originalmente tentado. De uma forma geral, um algoritmo recursivo deve ter duas partes: o caso base, que trata uma entrada simples que pode ser resolvida sem recorrer a uma chamada recursiva, e a parte recursiva que contém uma ou mais chamadas recursivas ao algoritmo onde os parâmetros estão em algum sentido "mais próximos" do caso base do que aqueles da chamada original. Podemos então dizer que uma definição recursiva é aquela que usa a palavra ou conceito a ser definido na própria definição.

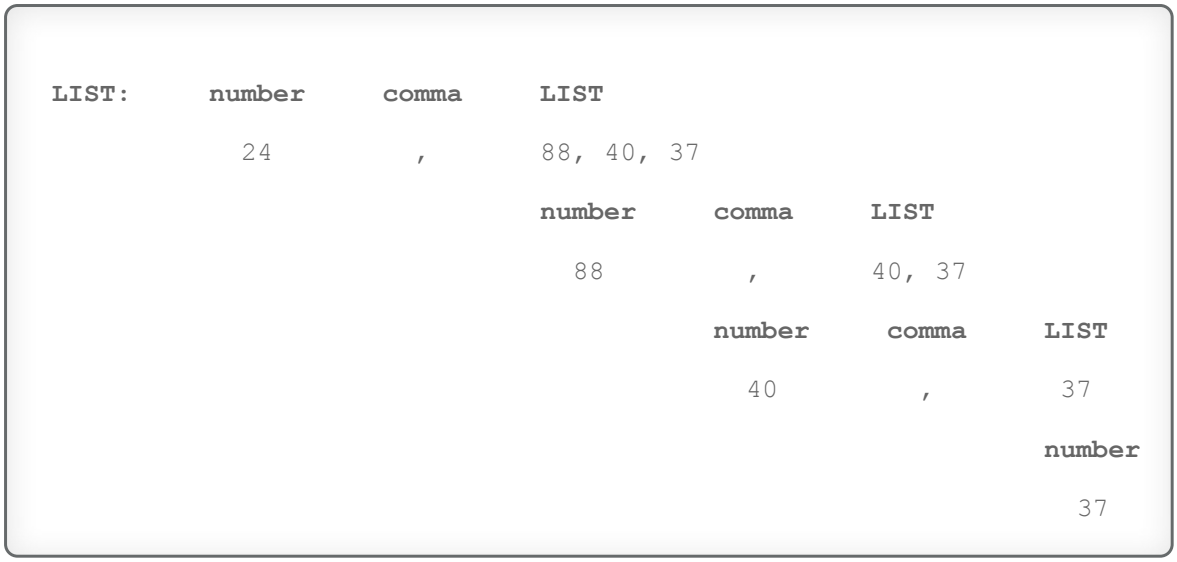
Em algumas situações, uma definição recursiva pode ser uma forma adequada para expressar um conceito, no entanto não é automaticamente sinónimo de eficiência. Antes de aplicarmos a recursividade à programação devemos primeiro praticar o pensamento recursivo. Considere a seguinte lista de números:

24, 88, 40, 37

A lista pode também ser definida recursivamente:

Uma LISTA é um: número
ou um: número vírgula LISTA

Ou seja, uma lista pode ser: 1) um número ou 2) um número seguido por uma vírgula seguida por uma lista. O conceito de lista é então utilizado para se definir. Com uma análise mais profunda através de uma traçagem da definição conseguimos perceber melhor o que acontece como demonstra a figura seguinte.



9.2. Programação Recursiva

Um método em Java pode chamar-se a si próprio se for definido dessa forma, e é denominado de método recursivo. Como já foi referido anteriormente o código de um método recursivo deve ser estruturado para lidar com ambos: caso base e o caso geral (ou recursivo). No entanto, deve-se ter muito cuidado com a definição de métodos desta forma já que cada chamada cria um ambiente de execução de novo, com novos parâmetros e novas variáveis locais. Como sempre, quando o método for concluído o controlo retorna para o método que o invocou (que pode ser uma outra instância de si mesmo).

Considere o problema de realizar a soma de todos os números entre **1** e **N**, inclusive

If N is 5, the sum is

1 + 2 + 3 + 4 + 5

Este problema pode ser definido através da seguinte forma recursiva:

A soma de 1 até N é N mais a soma de 1 até N-1

Podemos inclusivamente olhar para a formulação matemática do problema:

$$\sum_{i=1}^{N} i = N + \sum_{i=1}^{N-1} i = N + N-1 + \sum_{i=1}^{N-2} i$$

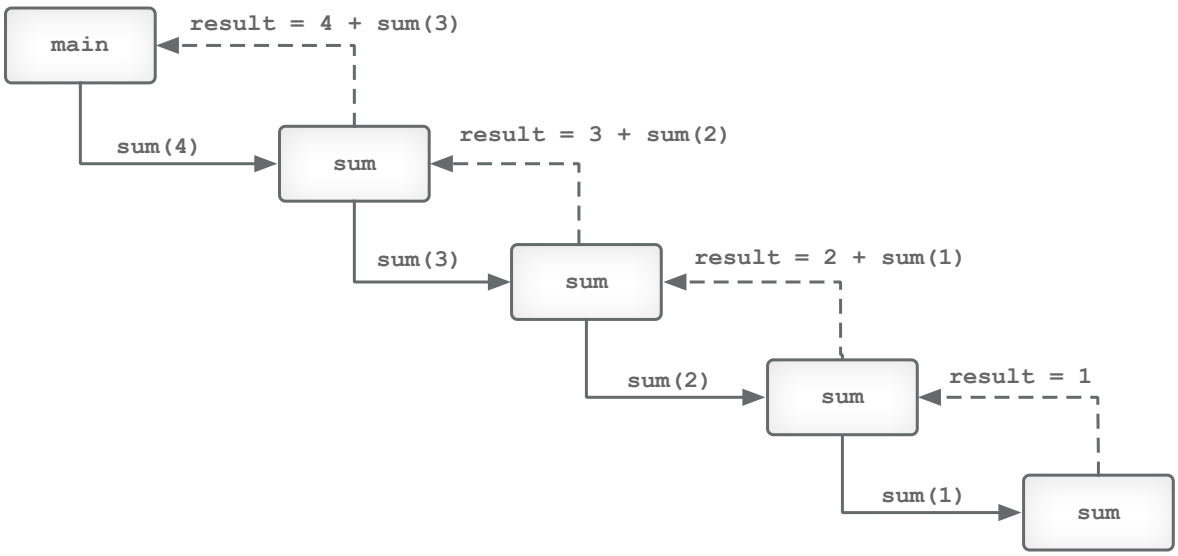
$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

$$= N + N-1 + N-2 + \ldots + 2 + 1$$

No entanto o nosso objetivo é conseguir conceber um algoritmo para o problema do cálculo do somatório através de uma abordagem recursiva. Uma possível implementação é apresentada de seguida.

```
1. public int sum (int num) {
2.     int result;
3.     if (num == 1)
4.        result = 1;
5.     else
6.        result = num + sum(num-1);
7.     return result;
8. }
```

Ao fazermos a taçagem do algoritmo para o valor 4, por exemplo `sum(4)` conseguimos perceber o que realmente acontece.



Com a traçagem apresentada na figura anteriormente apercebemo-nos que existem várias invocações do mesmo método em memória e só vai desbloquear quando chegarmos ao caso base, neste caso `sum(1)`. É quando chegamos a esse valor que é retornadado 1 e vamos finalizado todos os métodos em memória de trás para a frente até que chegamos ao método original que irá devolver o valor final a quem o invocou. As definições exigem muito cuidado, podem ser uma boa solução mas exigem cuidados já que a memória vai ser ocupada até chegarmos ao caso base. Se o caso base for mal definido o método não vai finalizar e vamos criar um problema para o nosso sistema operativo já que vai no limite ficar sem memória.

Só porque podemos usar a recursividade para resolver um problema não significa que devemos fazer. Por exemplo, normalmente não iríamos usar a recursividade para resolver a soma de 1 a N repare que a versão iterativa é mais fácil de entender.

```
1. public int sum (int num) {
2.     int sum = 0;
3.     for (int i = 1;i <= n; i++) {
4.         sum = sum + i;
5.     }
6.     return sum;
7. }
```

Podemos ainda usar uma função muito mais simples para o feito:

$$Sum = n*(n+1)/2$$

Quer isto dizer que devemos ser capazes de determinar quando a recursividade é a técnica mais correta de usar.

Cada solução recursiva tem então uma solução iterativa correspondente, como vimos anteriormente, a soma dos números entre 1 e N pode ser calculado através de um ciclo. Neste caso em particular a solução recursividade tem a sobrecarga de várias invocações do método no entanto, para alguns problemas as soluções recursivas são geralmente mais simples e elegantes do que as soluções iterativas. Um exemplo de solução recursiva menos eficiente que uma iterativa é o cálculo de uma sequência *Fibonacci*. A sequência de *Fibonacci* consiste numa série de números, tais que, definindo os seus dois primeiros números como sendo 0 e 1, os números seguintes são obtidos através da soma dos seus dois antecessores.

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

A solução recursiva é ineficiente já que recalcula várias vezes a solução para valores intermédios. Por exemplo para *fib(5) = 5*, temos *fib(4) + 2*fib(3) + 4*fib(2)*, de seguida, na figura 1, é apresentada a árvore de recursiva para *fib(5)*.

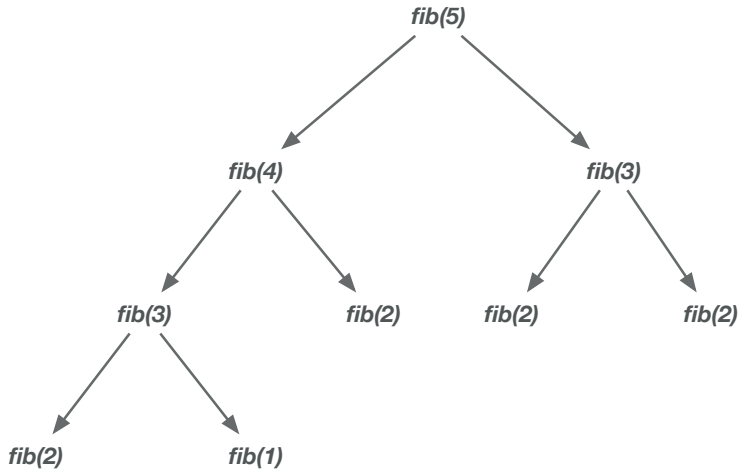
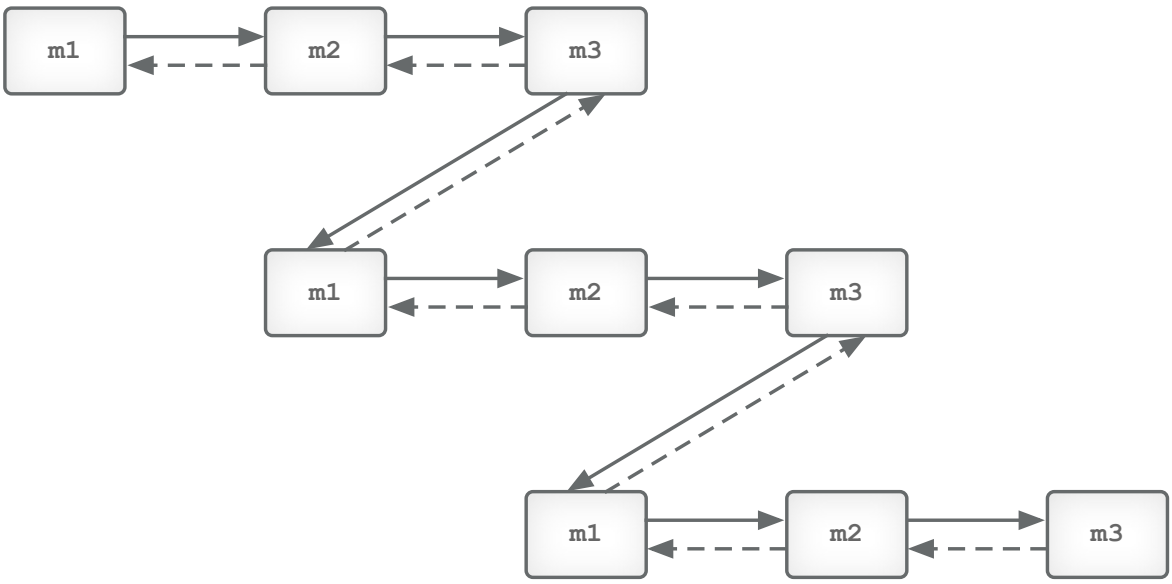


Figura 1 - Árvore recursiva

9.3. Recursividade Indireta

Até agora, todos os exemplos de recursividade tiveram uma coisa em comum: um único método envolvido que se invocou a si mesmo que é chamado de recursividade direta. Existe também a recursividade indireta onde vários métodos dependem uns dos outros.

Por exemplo, o método `m1` poderia invocar `m2`, que invoca o `m3`. Se o método `m3` sob qualquer circunstância invocar de novo `m1` é um caso de recursividade indireta. De seguida podemos ver a exemplificação do cenário descrito.



Normalmente, este tipo de abordagem acaba por complica o processo de análise e de *debugging*.

9.4. Problema das Torres de Hanói

As Torres de Hanói é um antigo quebra-cabeça que consiste em vários discos colocados em três pinos conforme apresentado na figura seguinte.

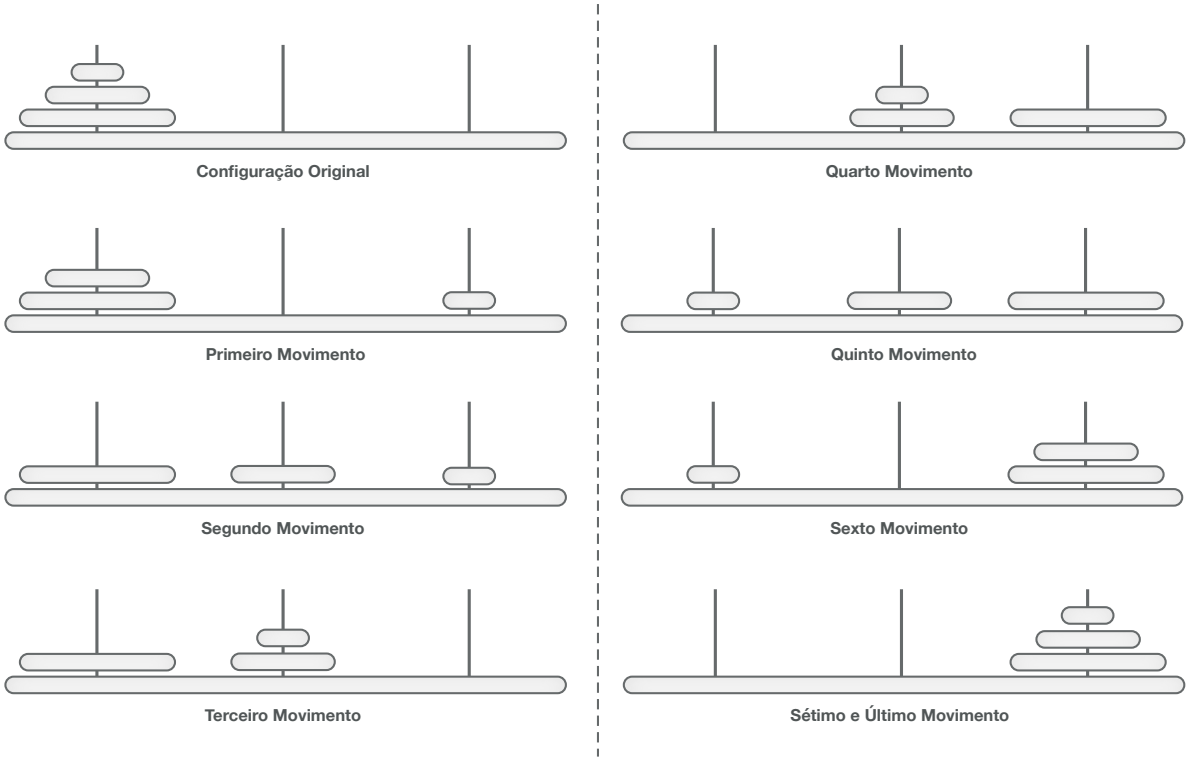
Todos os discos têm diâmetros diferentes e orifícios no meio para que se encaixem nos pinos. Todos os discos começam no pino 1. O objetivo do quebra-cabeça é transferir todos os discos do pino 1 para o pino 3. Apenas um disco pode ser movido de cada vez e nenhum disco pode ser colocado em cima de um disco menor que ele mesmo.

> *Existe um mito antigo de que em algum lugar na Índia, num templo remoto, monges trabalham dia e noite para transferir 64 discos de ouro de uma das três torres cravejadas de diamantes para outra. Quando terminarem, o mundo irá acabar. Qualquer preocupação que tenha ao ler isto irá-se dissipar quando perceber quanto tempo demora para resolver o quebra-cabeças para muito menos de 64 discos.*

O objetivo é então mover todos os discos de um pino para outro seguindo as seguintes regras:

- Apenas um disco pode ser movido num dado momento;
- Um disco não pode ser colocado em cima de um disco menor;
- Todos os discos devem estar em algum pino (excepto para o disco em trânsito).

De seguida podemos ver a solução para o quebra-cabeças das Torres de Hanói para três discos, atenção que esta é versão mais simples para o problema já que tem apenas três discos.



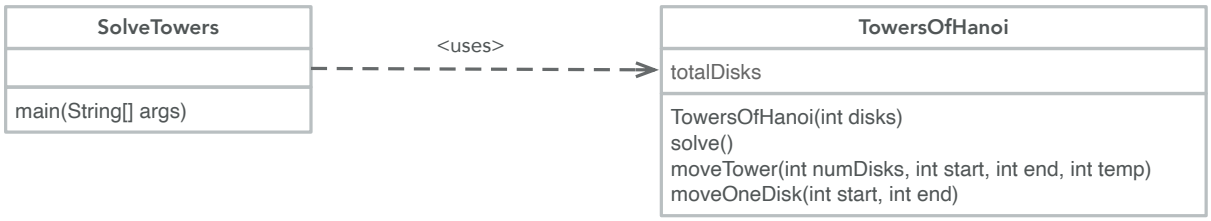
Existem várias abordagens para resolver este problema, no entanto normalmente usa-se uma solução recursiva já que a solução é bastante simples comparativamente a outra. Este costuma ser o primeiro problema que se procura resolver quando se procura aprender recursividade. Na nossa tentativa de especificar o problema de forma recursiva podemos dizer que para mover uma pilha de N discos a partir do pino original para o pino de destino é necessário:

- Mover os N-1 discos do pino original para o pino extra;
- Mover o maior disco do pino original para o pino de destino;
- Mover os N-1 discos do pino extra para o pino de destino.

O caso base ocorre quando uma "pilha" contém apenas um disco. Note que o número de movimentos aumenta exponencialmente à medida que aumenta o número de discos que à primeira vista passa despercebido. Como já foi referido a solução recursiva é simples e elegante de expressar (e programar). A solução iterativa para este problema que normalmente é sempre a nossa escolha acaba por ser muito mais complexa. De seguida é apesentada uma solução para o problema através de uma solução recursiva. A solução funciona para qualquer número de discos e é interessante analisar os resultados para uma

variedade de discos e estudar a implementação já que poderá replicar a abordagem noutros problemas semelhantes.

Solução para o problema




```
1. public class SolveTowers {
2.    /**
3.     * Creates a TowersOfHanoi puzzle and solves it.
4.     */
5.    public static void main(String[] args) {
6.       TowersOfHanoi towers = new TowersOfHanoi(4);
7.       towers.solve();
8.    }
9. }
```

```
1. public class TowersOfHanoi {
2.    private int totalDisks;
3.    /**
4.     * Sets up the puzzle with the specified number of disks.
5.     *
6.     * @param disks  the number of disks to start the
7.     * towers puzzle with
8.     */
9.    public TowersOfHanoi(int disks) {
10.      totalDisks = disks;
11.   }
12.   /**
13.    * Performs the initial call to moveTower to solve the puzzle.
```

```
14.    * Moves the disks from tower 1 to tower 3 using tower 2.
15.    */
16.   public void solve() {
17.      moveTower(totalDisks, 1, 3, 2);
18.   }
19.   /**
20.    * Moves the specified number of disks from one tower to another
21.    * by moving a subtower of n-1 disks out of the way, moving one
22.    * disk, then moving the subtower back. Base case of 1 disk.
23.    *
24.    * @param numDisks  the number of disks to move
25.    * @param start     the starting tower
26.    * @param end       the ending tower
27.    * @param temp      the temporary tower
28.    */
29.   private void moveTower(int numDisks, int start, int end, int temp) {
30.      if (numDisks == 1)
31.         moveOneDisk(start, end);
32.      else {
33.         moveTower(numDisks-1, start, temp, end);
34.         moveOneDisk(start, end);
35.         moveTower(numDisks-1, temp, end, start);
36.      }
37.   }
38.   /**
39.    * Prints instructions to move one disk from the specified
40.    * start tower to the specified end tower.
41.    *
42.    * @param start  the starting tower
43.    * @param end    the ending tower
44.    */
45.   private void moveOneDisk(int start, int end) {
46.      System.out.println("Move one disk from " + start +
47.                         " to " + end);
48.   }
49.}
```

9.5. Exercícios Propostos

Exercício 1

Quais as componentes que compõem uma definição recursiva?

Exercício 2

Deve usar sempre a recursividade para resolver os mais diversos problemas?

Exercício 3

Acrescente um método recursivo à implementação de lista duplamente ligada que devolva o conteúdo da lista com a ordem dos elementos invertida.

Exercício 4

Acrescente o método `void replace(T existingElement, T newElement)`, que funcione de forma recursiva, à sua implementação de lista ligada que seja capaz de substituir todas as ocorrências do argumento `existingElement` pelo argumento `newElement`.

Exercício 5

Acrescente um método recursivo à sua implementação de lista simplesmente ligada que seja capaz de inverter a ordem dos elementos.

Exercício 6

Escreva um algoritmo recursivo para avaliar expressões *postfix*.

Exercício 7

Escreva um algoritmo recursivo para avaliar expressões *infix*. Suponha que os operadores tenham precedência igual e sejam associativos à esquerda, de modo que, sem parênteses, as operações sejam avaliadas da esquerda para a direita. Os parênteses alteram a ordem de avaliação da maneira usual.