ESTRUTURAS DE DADOS

2024/2025

Aula 07

- Pensamento Recursivo
- Programação Recursiva
- Recursividade Indirecta
- Torres de Hanói
- Análise dos Algoritmos Recursivos

Ricardo Santos | rjs@estg.ipp.pt

ESCOLA SUPERIOR DE TECNOLOGIA E GESTÃO

Pensamento Recursivo

- A recursividade é uma técnica de programação em que um método se pode chamar a ele próprio para resolver um dado problema
- 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
- Antes de aplicar a recursividade à programação, é melhor praticar o pensamento recursivo

Definições Recursivas

- Considere a seguinte lista de números:

```
24, 88, 40, 37
```

- Essa lista pode ser definida recursivamente:

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

- Ou seja, uma lista pode ser um **número** ou um **número seguido por uma vírgula seguida por uma lista**

- O conceito de lista é utilizada para se definir

Realizar o *trace* da definição recursiva da Lista

```
LIST:   number   comma    LIST
          24       ,      88, 40, 37
                        number   comma    LIST
                          88       ,      40, 37
                                        number   comma      LIST
                                          40       ,        37
                                                          number
                                                            37
```

Recursão Infinita

- Todas as definições recursivas devem ter uma parte não-recursiva
- Se não tiverem, não existe nenhuma forma de terminar o caminho recursivo
- Uma definição sem uma parte não-recursiva faz uma recursão infinita
- Este problema é semelhante a um ciclo infinito
- A parte não-recursiva é frequentemente chamada de caso base

Programação Recursiva

- Um método em *Java* pode chamar-se: se for definido dessa forma, e é chamado de método recursivo
- O código de um método recursivo, deve ser estruturado para lidar com ambos: caso base e o caso recursivo
- 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 da seguinte forma recursiva:

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

A soma dos números 1 até N, definidos recursivamente

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

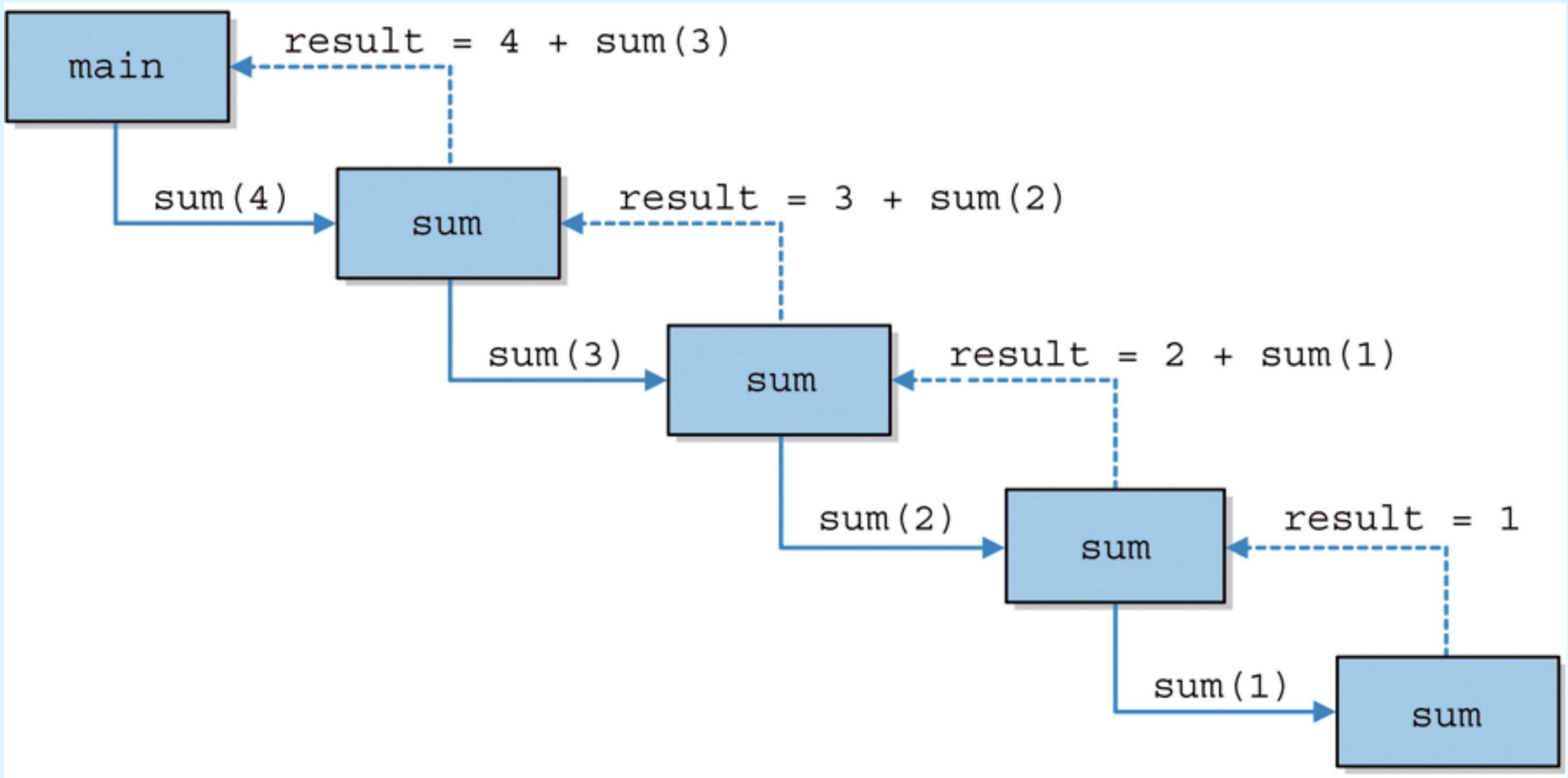
$$= N + N-1 + N-2 + \sum_{i=1}^{N-3} i$$

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

Programação Recursiva

```
public int sum (int num)
{
    int result;
    if (num == 1)
        result = 1;
    else
        result = num + sum(num-1);
    return result;
}
```

Chamadas Recursivas ao método `sum`



Recursividade versus Iteração

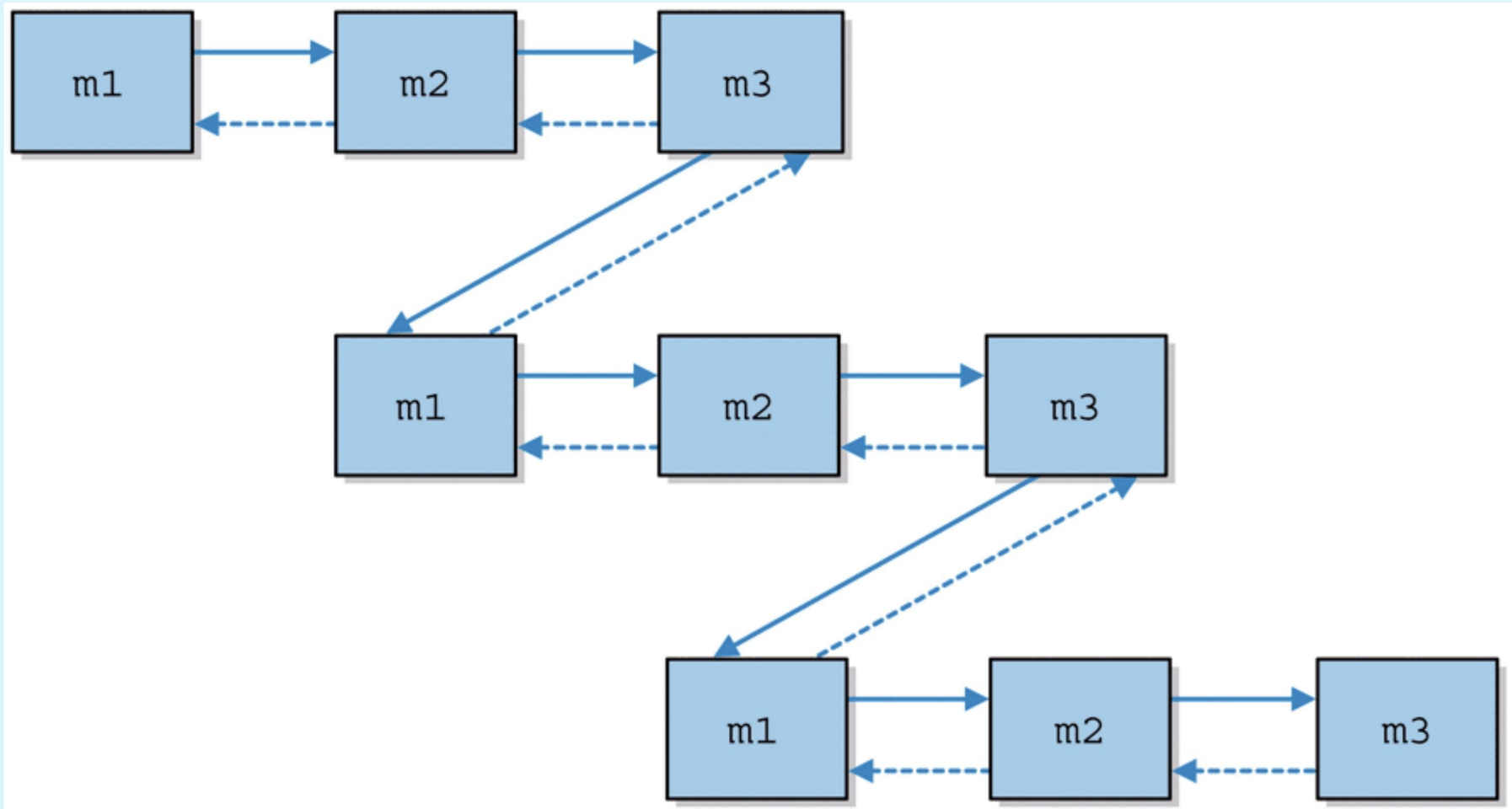
- Só porque podemos usar a recursividade para resolver um problema, não significa que devemos
- Por exemplo, normalmente não iríamos usar a recursividade para resolver a soma de **1** a **N**
- A versão iterativa é mais fácil de entender (na verdade existe uma fórmula que é superior tanto à versão recursiva como iterativa, neste caso)
- Devemos ser capazes de determinar quando a recursividade é a técnica mais correcta de usar

- Cada solução recursiva tem uma solução iterativa correspondente
- Por exemplo, a soma dos números entre **1** e **N** pode ser calculado com um ciclo
- A 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

Recursividade Indirecta

- Um método a invocar-se a si próprio é considerado recursão directa
- Mas, um método pode invocar outro método, que chama outro, etc, até que o método original é chamado novamente
- Por exemplo, o método `m1` poderia invocar `m2`, que invoca o `m3`, que chama `m1` novamente
- Isso é chamado de recursão indirecta
- Muitas vezes é mais difícil de analisar e realizar o *debugging*

Recursividade Indirecta



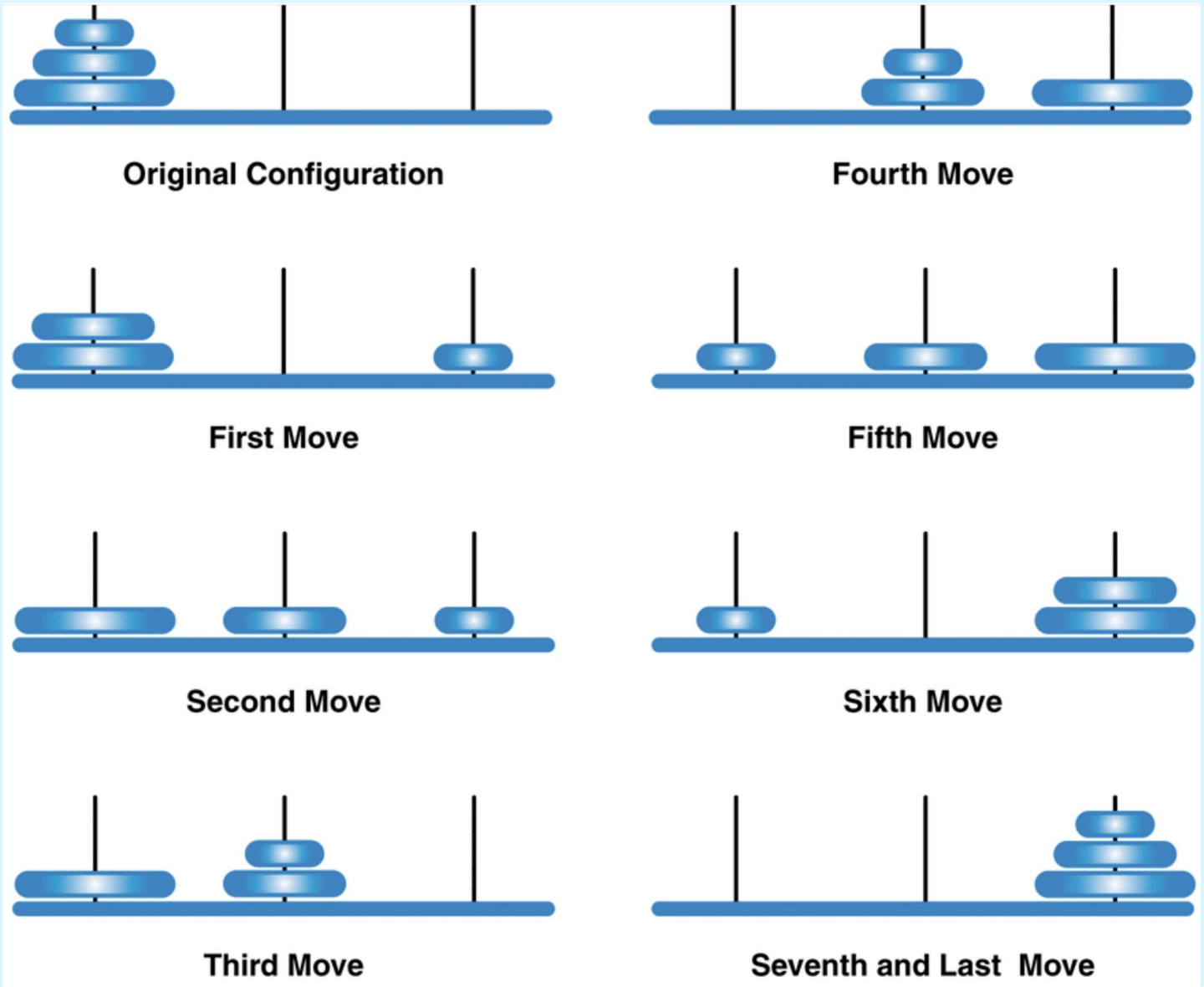
As Torres de Hanói

- As Torres de Hanói é um quebra-cabeça composto por três pinos verticais e vários discos que deslizam sobre os pinos
- Os discos são de tamanhos variados, inicialmente colocados num pino com o maior disco na parte inferior e os discos cada vez menores no topo

- O objectivo é mover todos os discos de um pino para outro seguindo estas 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 em trânsito)

Quebra-cabeça
Torres de Hanói

Solução para o quebra-cabeça das Torres de Hanói de três discos



Torres de Hanói

- Para mover uma pilha de **N** discos a partir do pino original para o pino de destino:
  - 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
- A solução recursiva é simples e elegante de expressar (e programar)
- Uma solução iterativa para este problema é muito mais complexa

Resolução!

```
public class SolveTowers {

   /**
    * Creates a TowersOfHanoi puzzle and solves it.
    */
   public static void main (String[] args)
   {
      TowersOfHanoi towers = new TowersOfHanoi (4);
      towers.solve();
   }
}
```

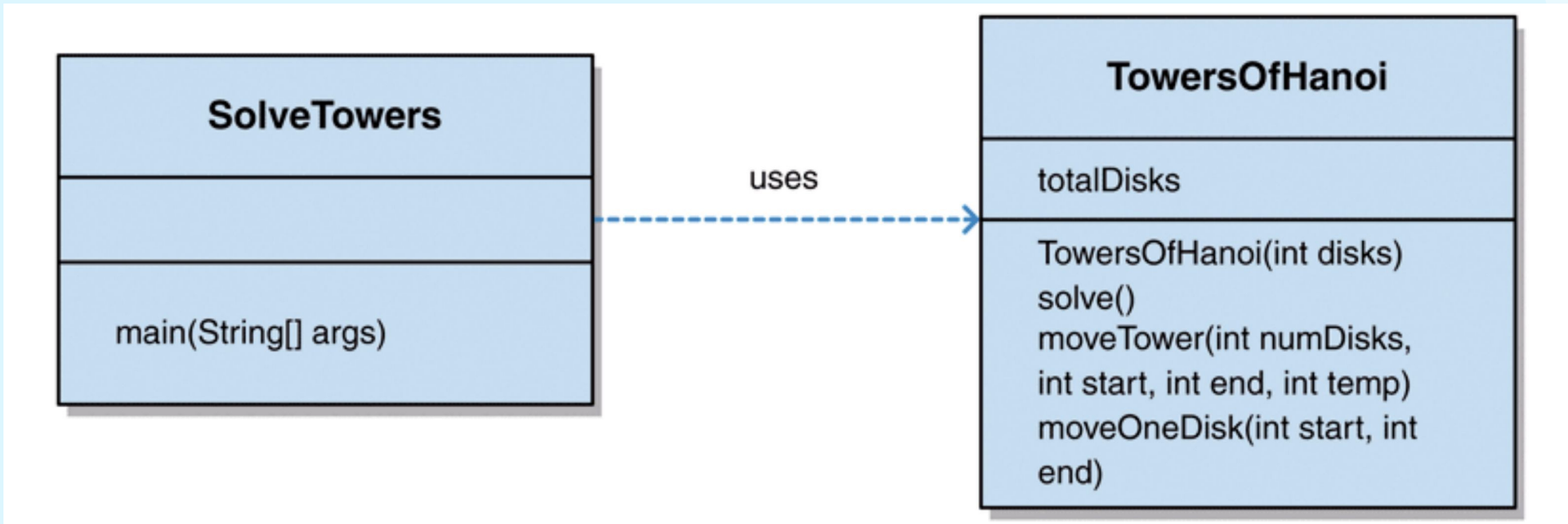
```
public class TowersOfHanoi {

   private int totalDisks;

   /**
    * Sets up the puzzle with the specified number of disks.
    *
    * @param disks  the number of disks to start the
    * towers puzzle with
    */
   public TowersOfHanoi (int disks)
   {
      totalDisks = disks;
   }

   /**
    * Performs the initial call to moveTower to solve the puzzle.
    * Moves the disks from tower 1 to tower 3 using tower 2.
    */
   public void solve ()
   {
      moveTower (totalDisks, 1, 3, 2);
   }
```

```
/**
 * Moves the specified number of disks from one tower to another
 * by moving a subtower of n-1 disks out of the way, moving one
 * disk, then moving the subtower back. Base case of 1 disk.
 *
 * @param numDisks  the number of disks to move
 * @param start     the starting tower
 * @param end       the ending tower
 * @param temp      the temporary tower
 */
private void moveTower (int numDisks, int start,
   int end, int temp) {
   if (numDisks == 1)
      moveOneDisk (start, end);
   else
   {
      moveTower (numDisks-1, start, temp, end);
      moveOneDisk (start, end);
      moveTower (numDisks-1, temp, end, start);
   }
}
```

```
/**
 * Prints instructions to move one disk from the specified
 * start tower to the specified end tower.
 *
 * @param start  the starting tower
 * @param end    the ending tower
 */
private void moveOneDisk (int start, int end)
{
   System.out.println ("Move one disk from " + start +
                       " to " + end);
}
}
```

Descrição UML das Classes

`SolveTowers` e `TowersofHanoi`



Análise aos Algoritmos Recursivos

- Ao analizar um ciclo, vamos determinar a ordem do corpo do ciclo e multiplicar pelo número de vezes que o ciclo é executado
- A análise recursiva é similar
- Iremos determinar a ordem do corpo do método e multiplicá-lo pela ordem da recursividade (o número de vezes que a definição recursiva é seguida)

- Para as Torres de Hanói, o tamanho do problema é o número de discos e a operação de interesse é a de mover um disco
- Excepto para o caso base, cada chamada recursiva resulta em chamar-se mais duas vezes
- Para resolver um problema de **N** discos, fazemos $\mathbf{2^{N-1}}$ movimentos do disco
- Portanto, o algoritmo é $\mathbf{O(2^n)}$, que é denominado de complexidade exponencial