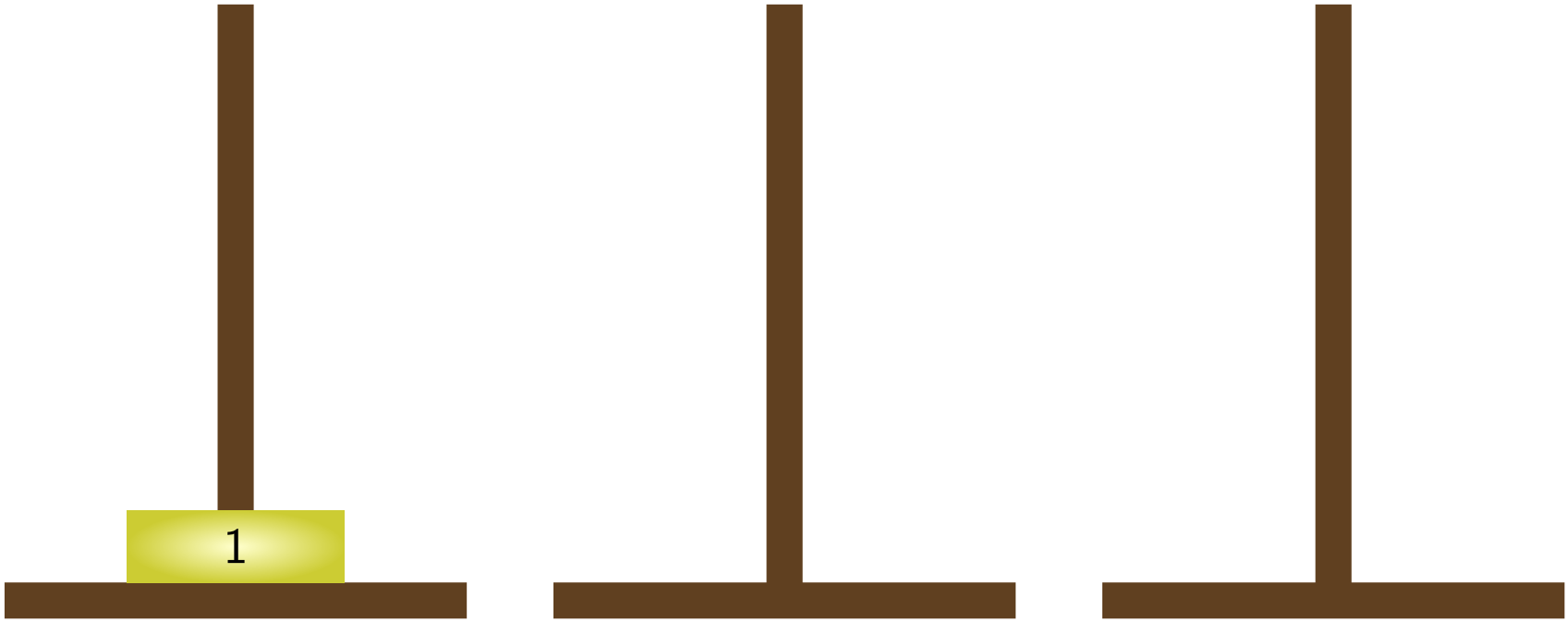
30. Recursividade 2

DIM0320

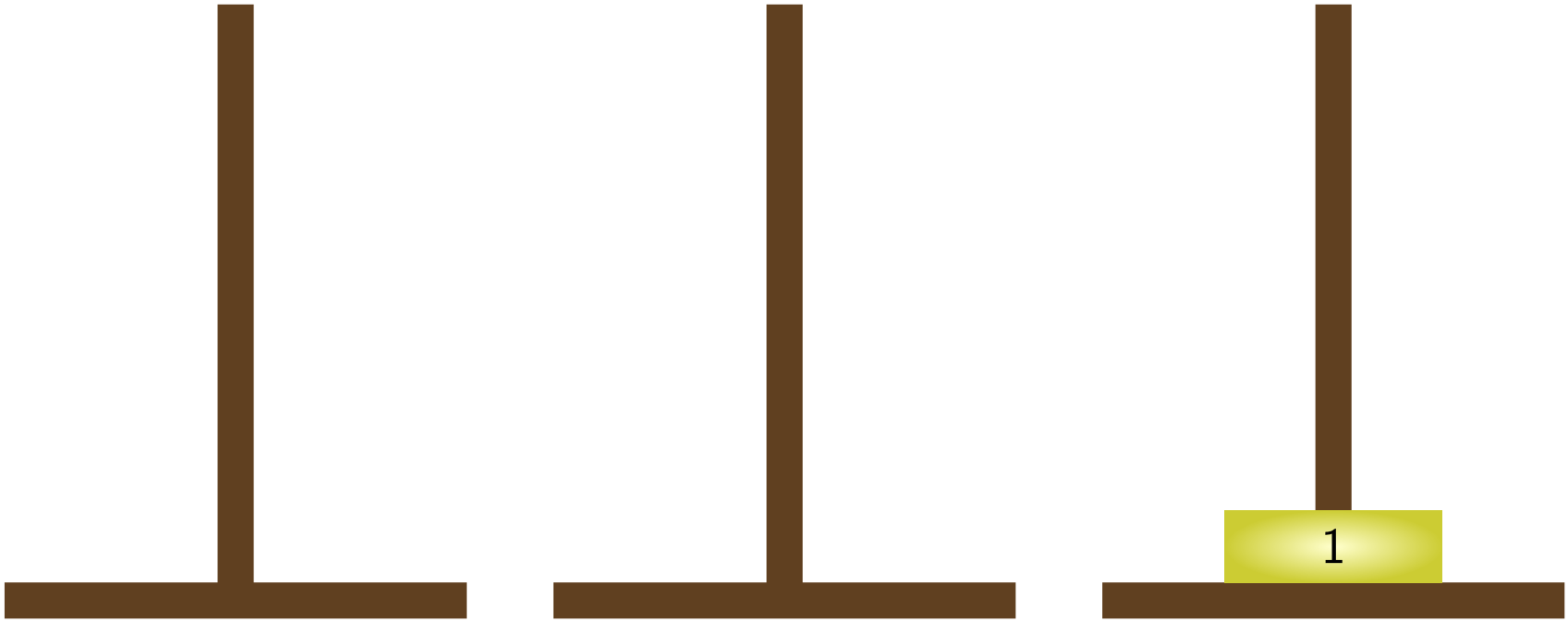
2015.1

Sumário

Torre de Hanoi – 1 disco

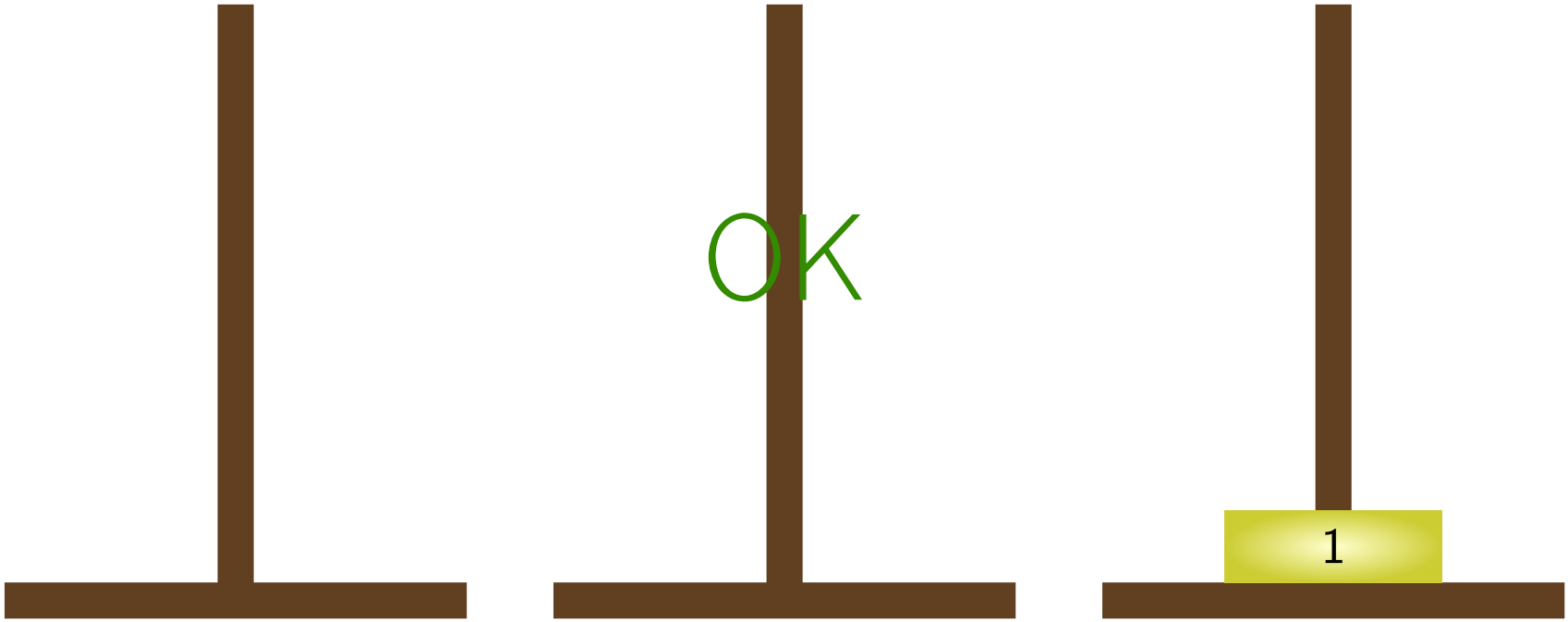


Torre de Hanoi – 1 disco

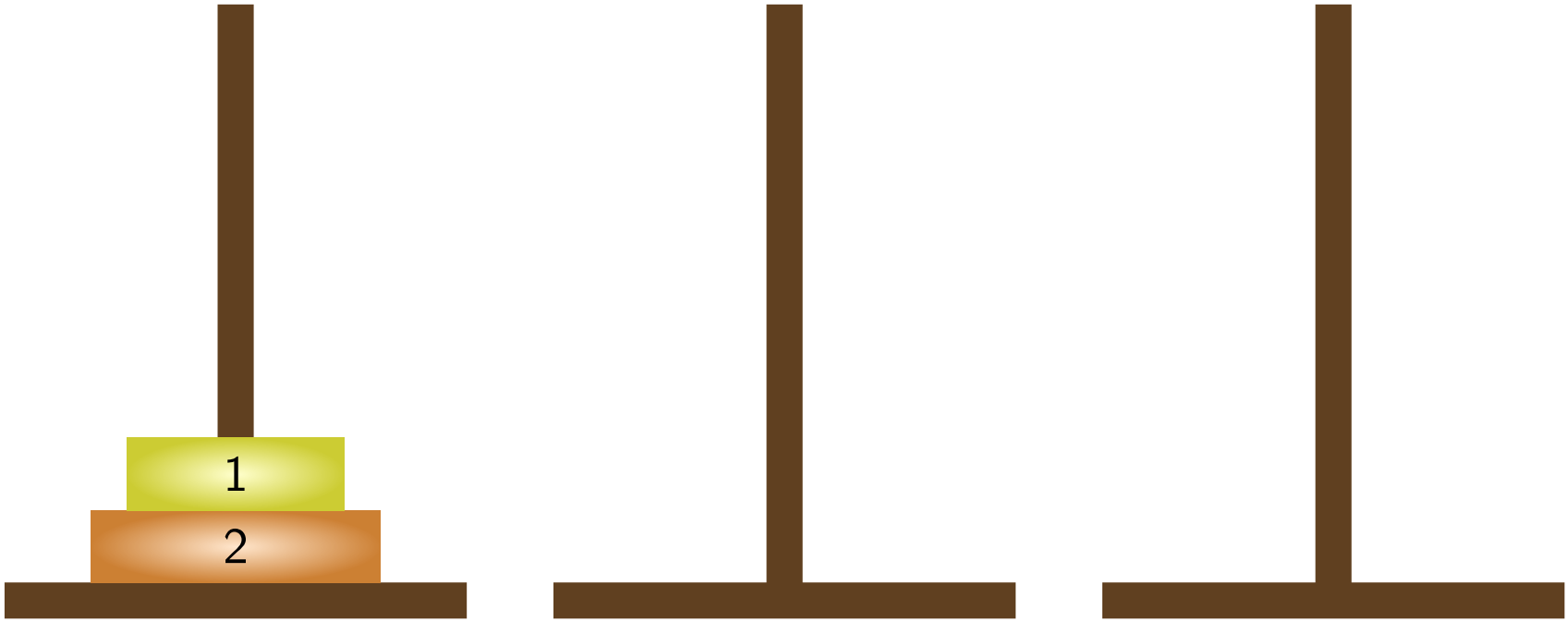


Mudei disco do pino 1 ao pino 3.

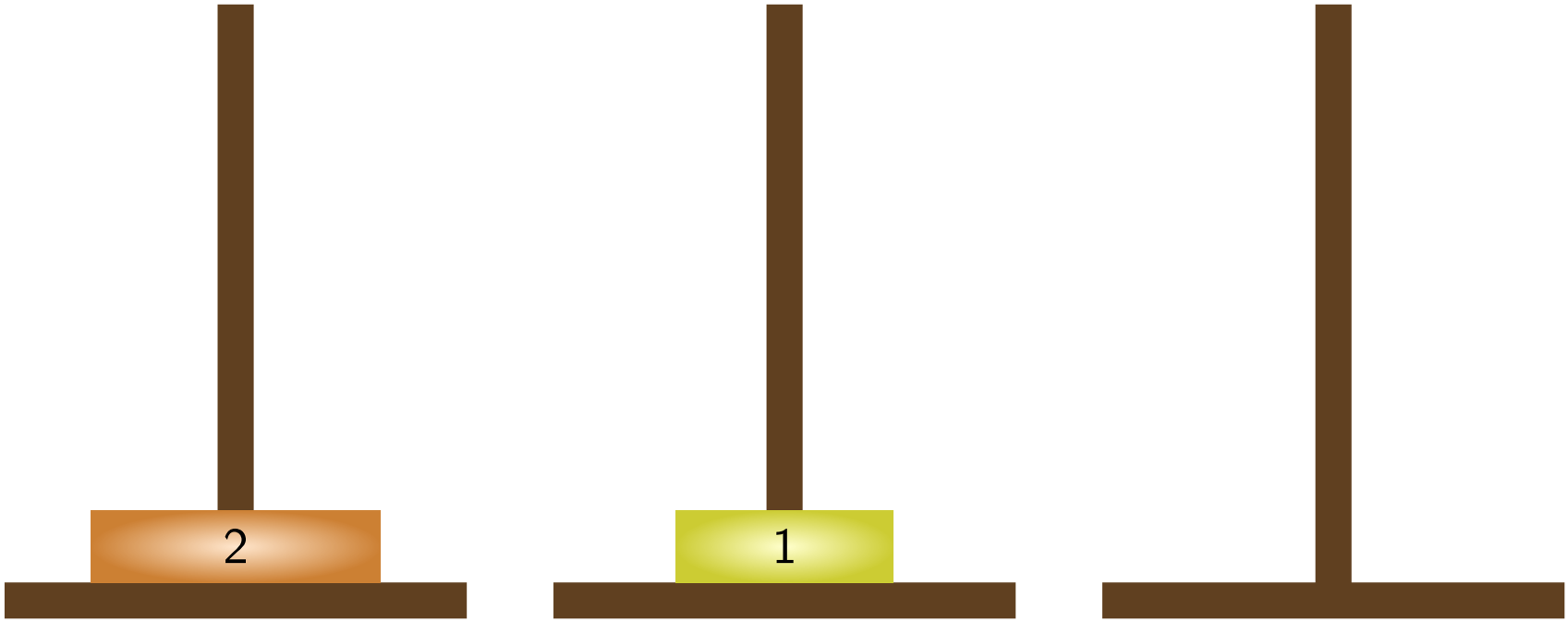
Torre de Hanoi – 1 disco



Torre de Hanoi – 2 discos

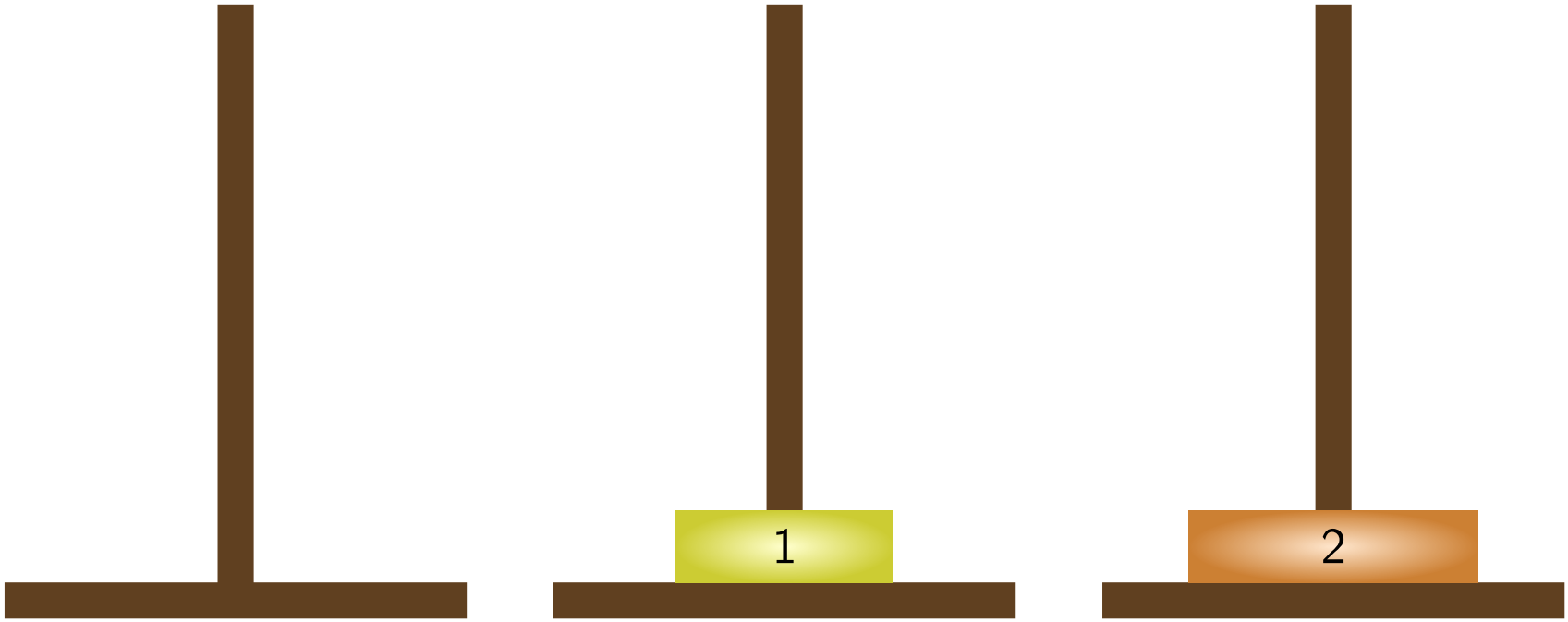


Torre de Hanoi – 2 discos



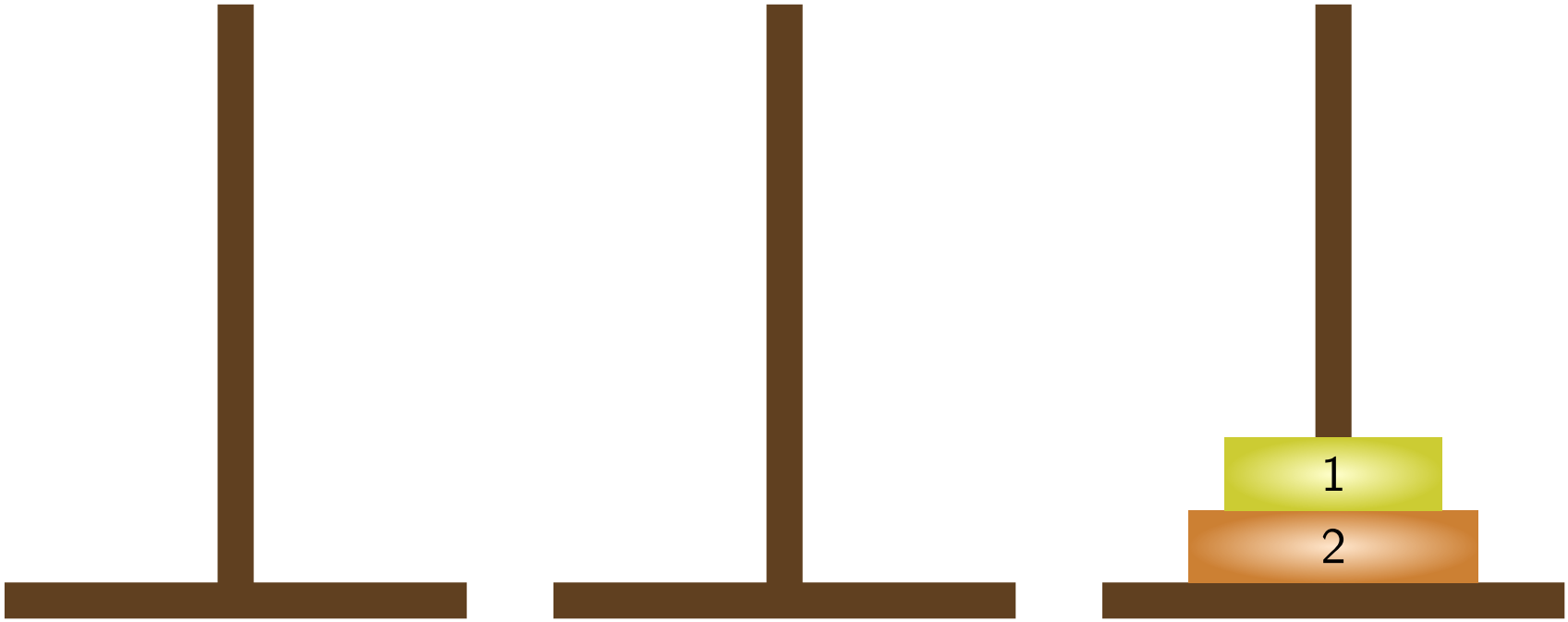
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 2 discos



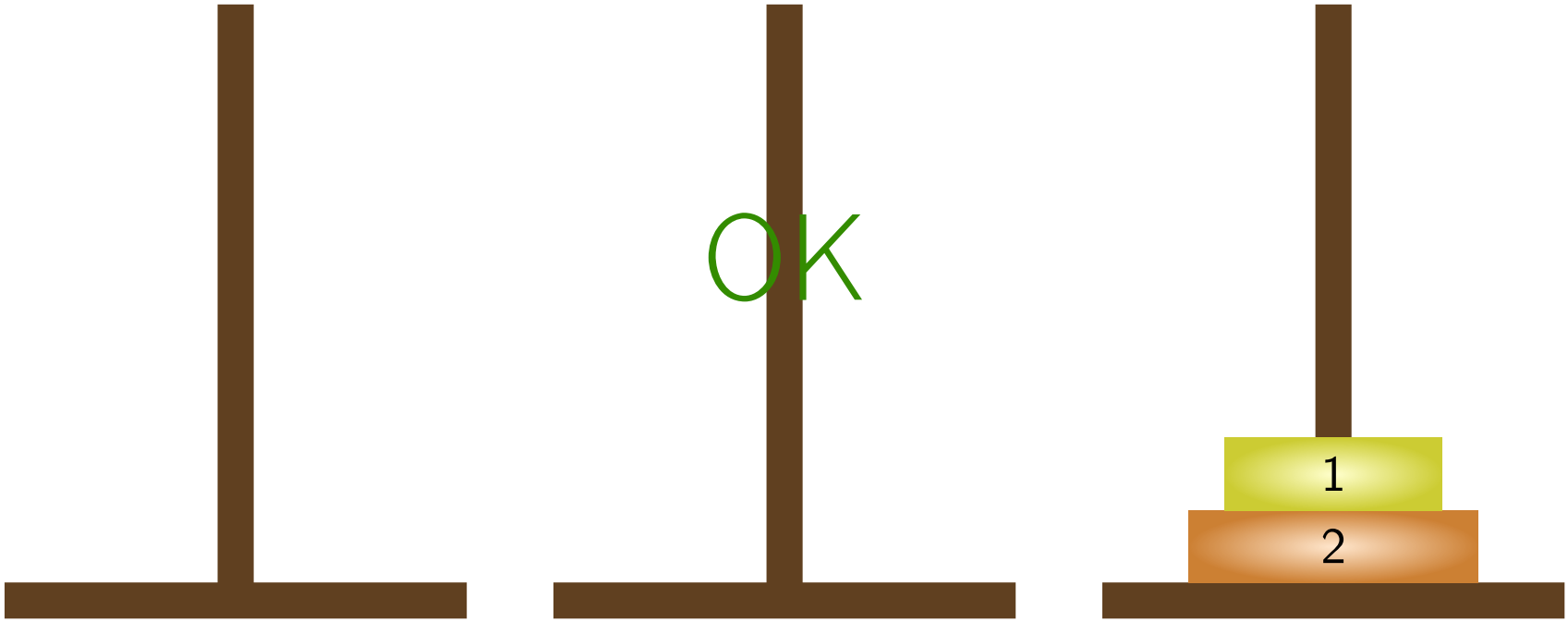
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 2 discos

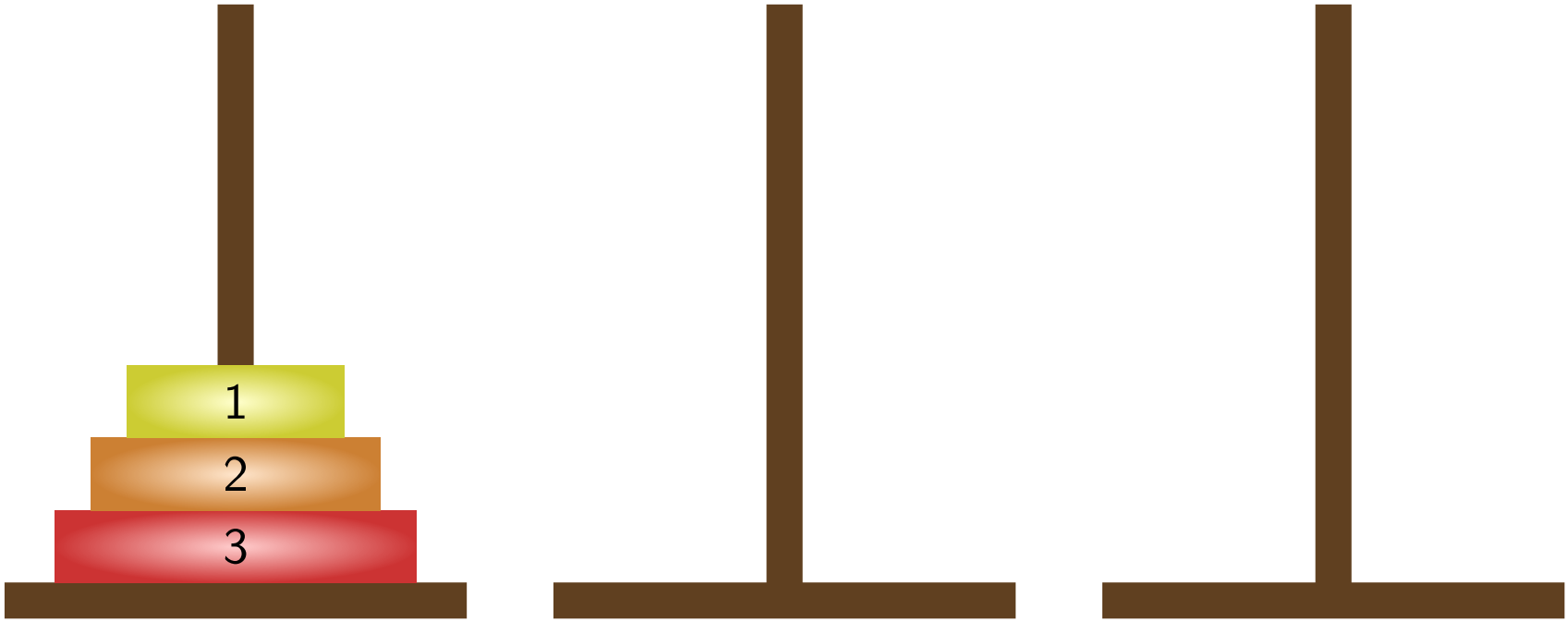


Mudei disco do pino 2 ao pino 3.

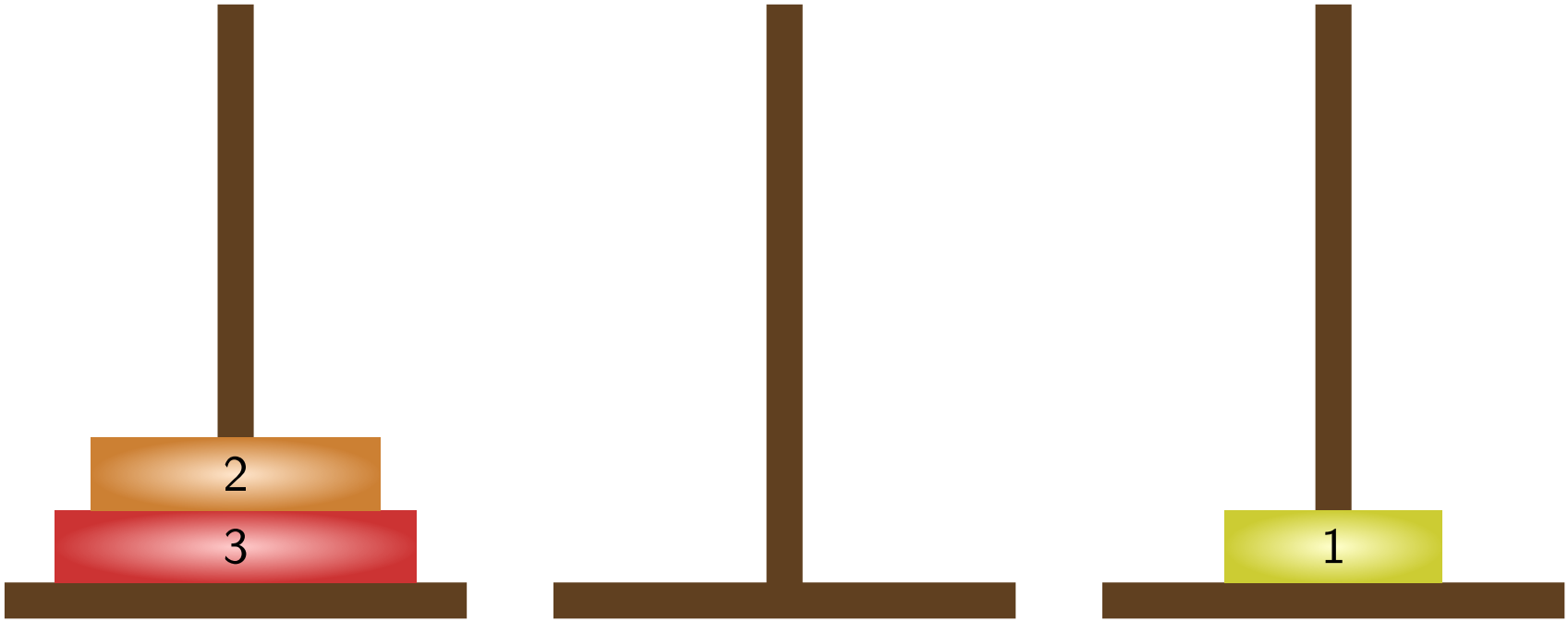
Torre de Hanoi – 2 discos



Torre de Hanoi – 3 discos

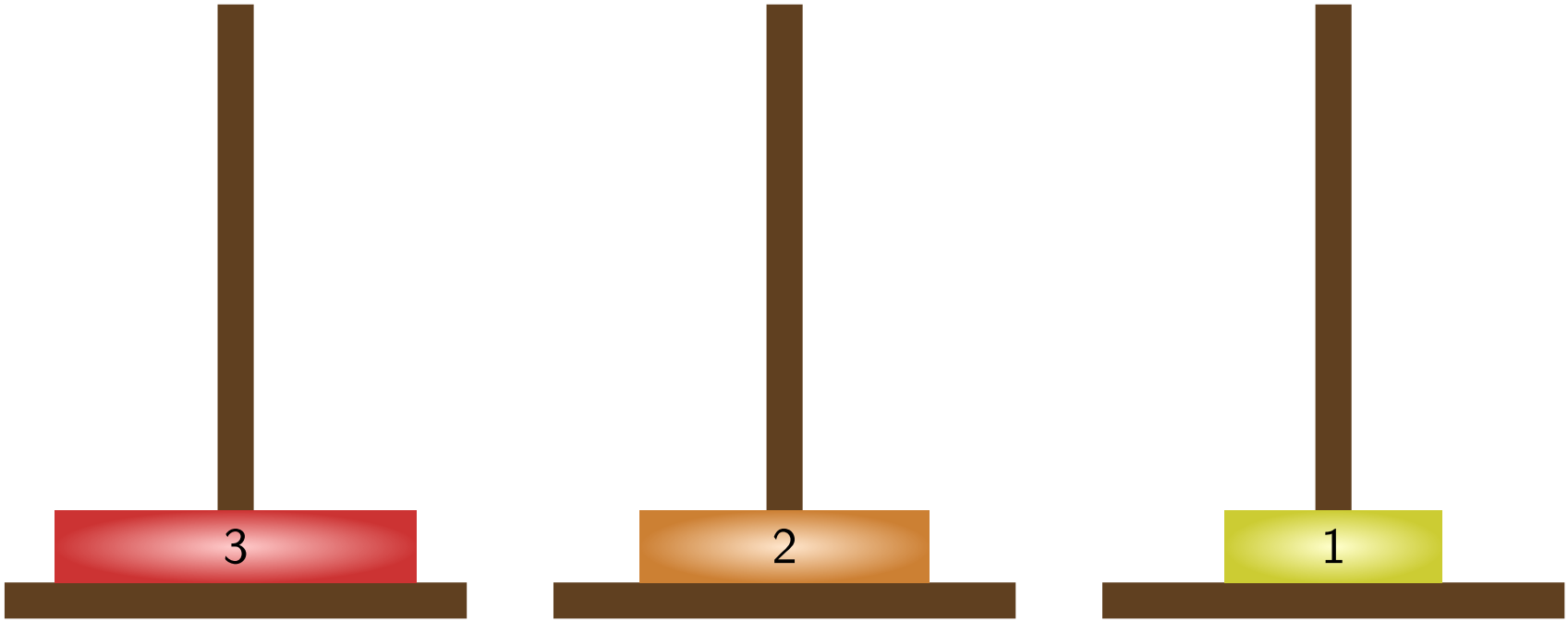


Torre de Hanoi – 3 discos



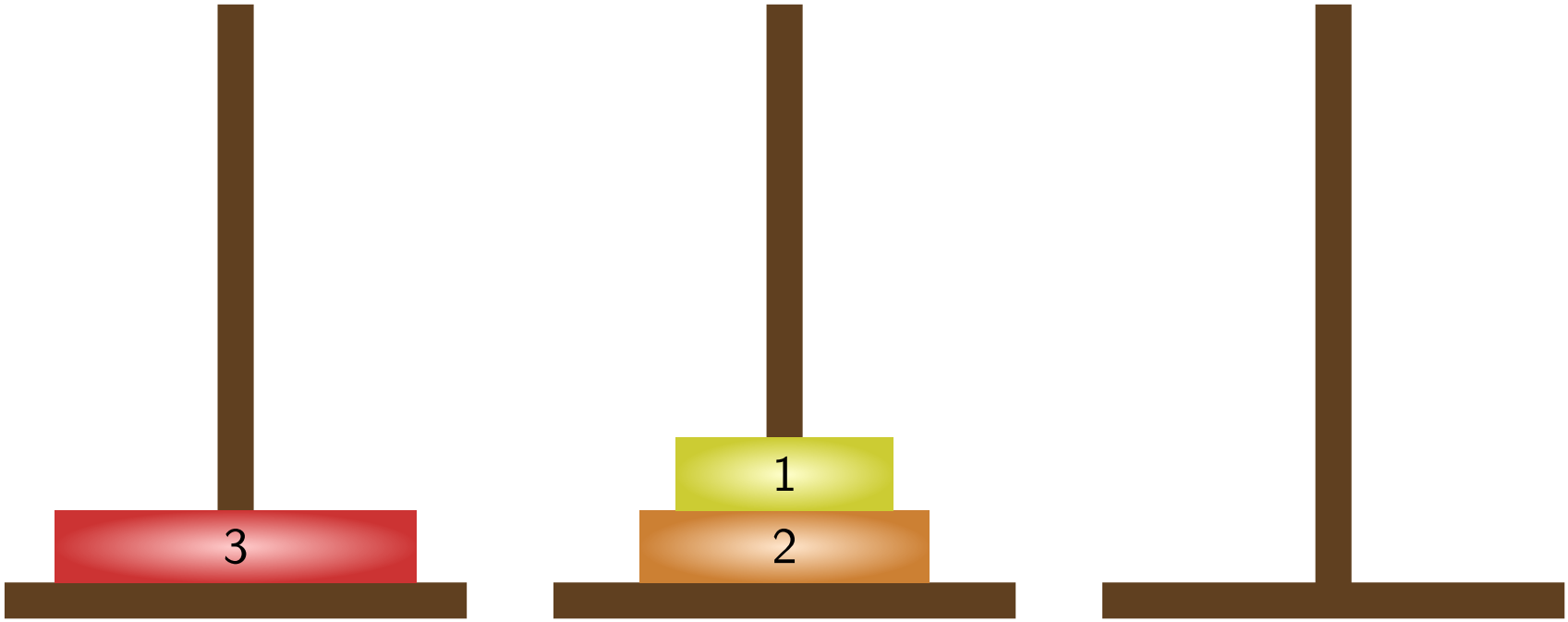
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 3 discos



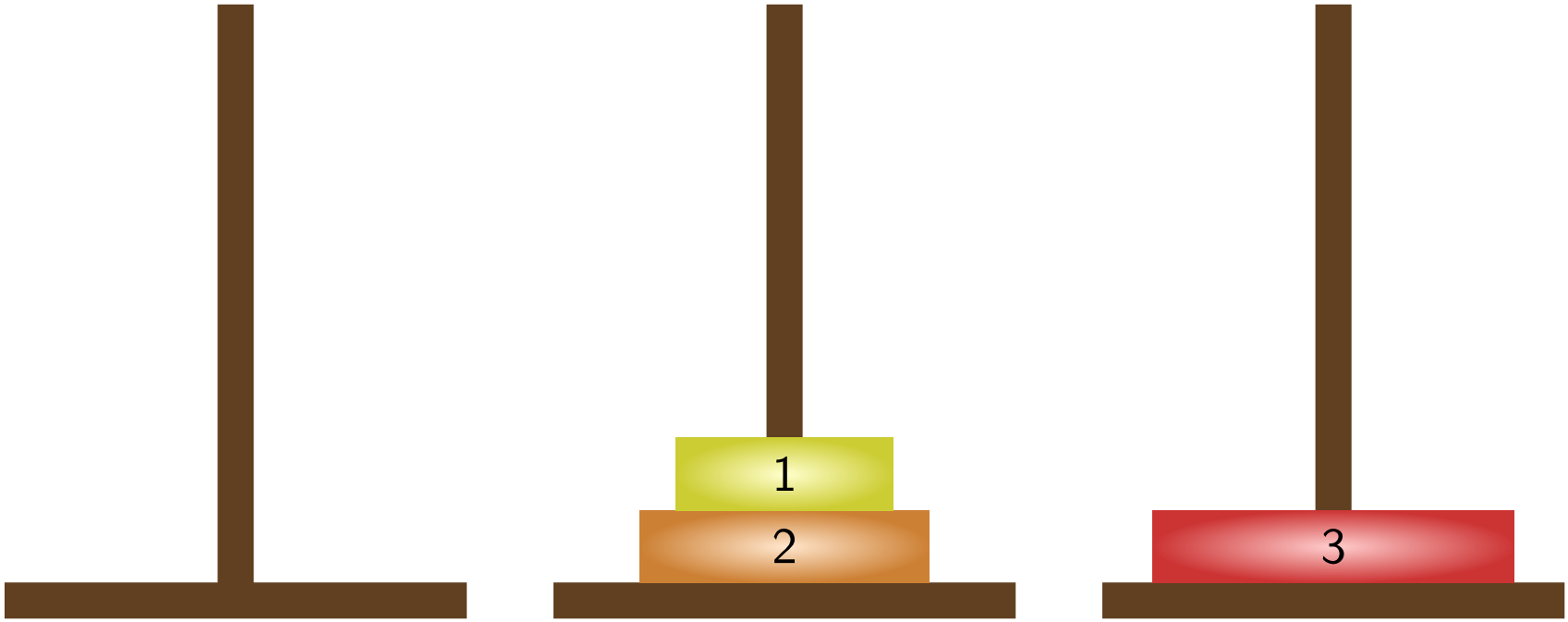
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 3 discos



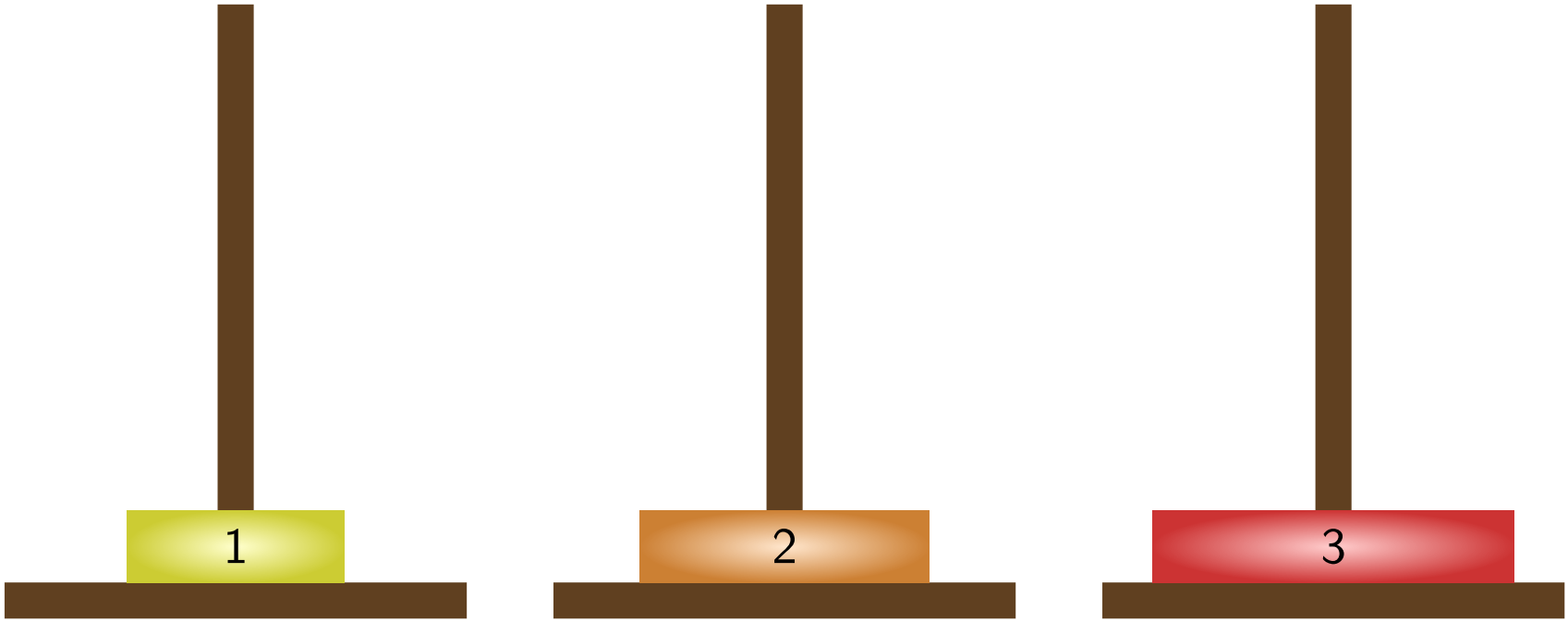
Mudei disco do pino 3 ao pino 2.

Torre de Hanoi – 3 discos



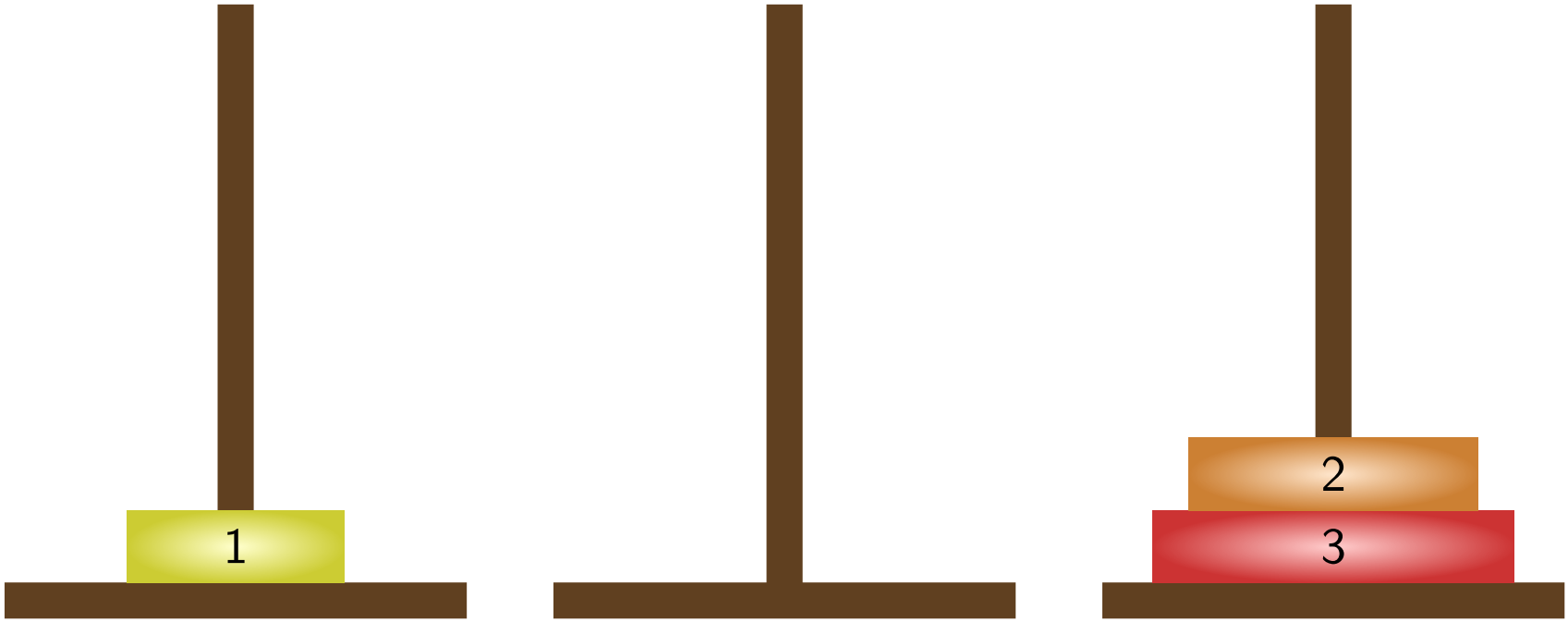
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 3 discos



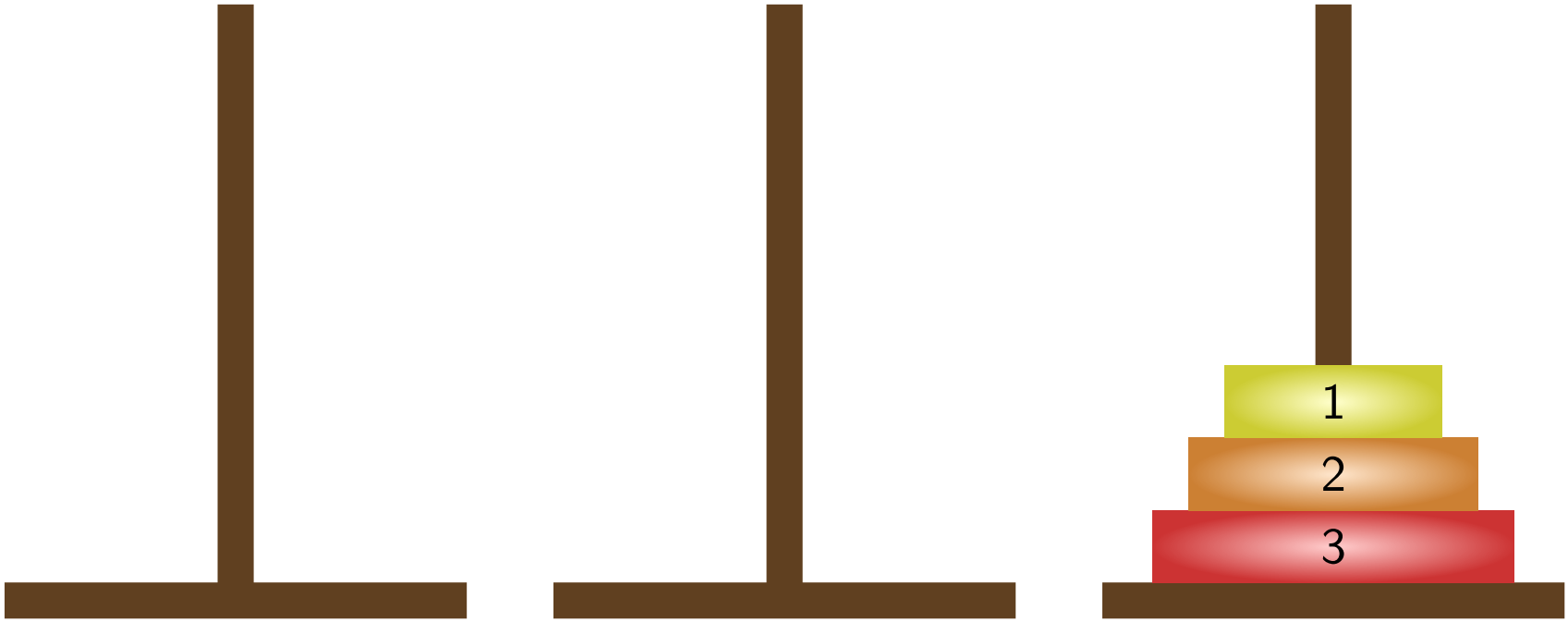
Mudei disco do pino 2 ao pino 1.

Torre de Hanoi – 3 discos



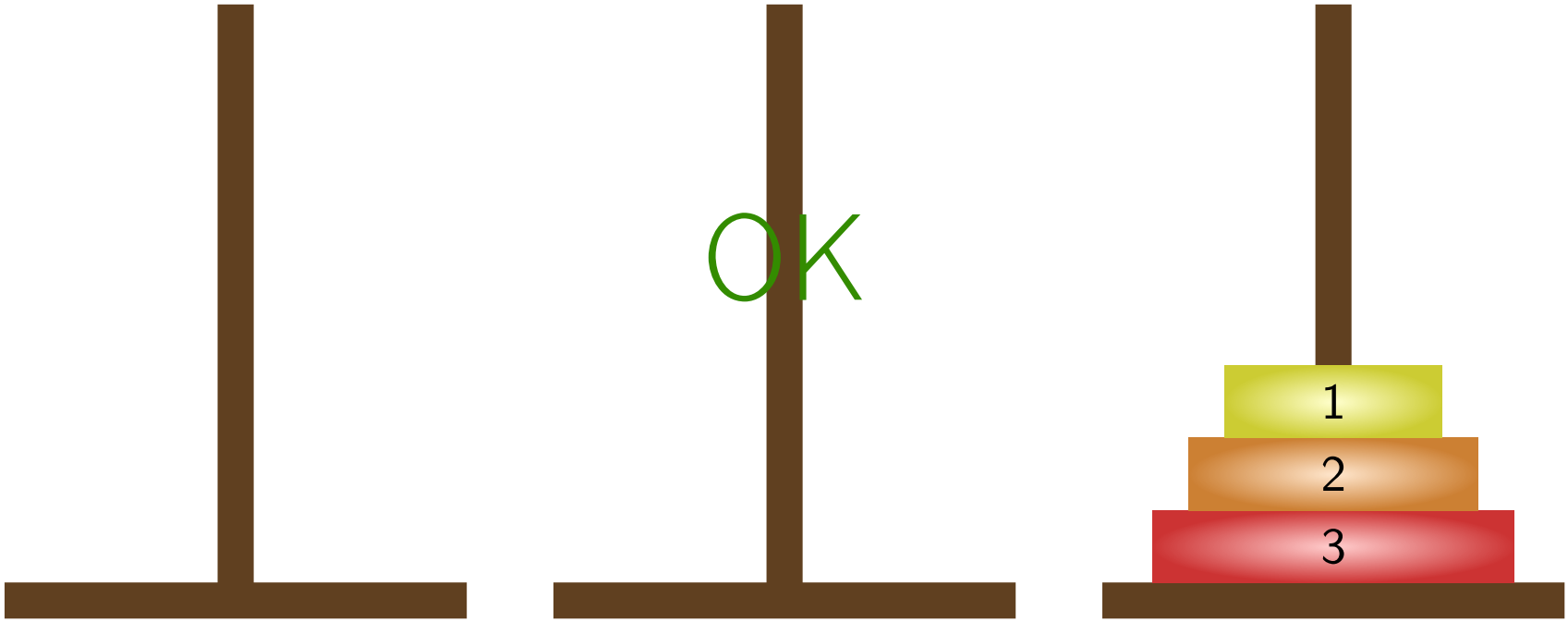
Mudei disco do pino 2 ao pino 3.

Torre de Hanoi – 3 discos

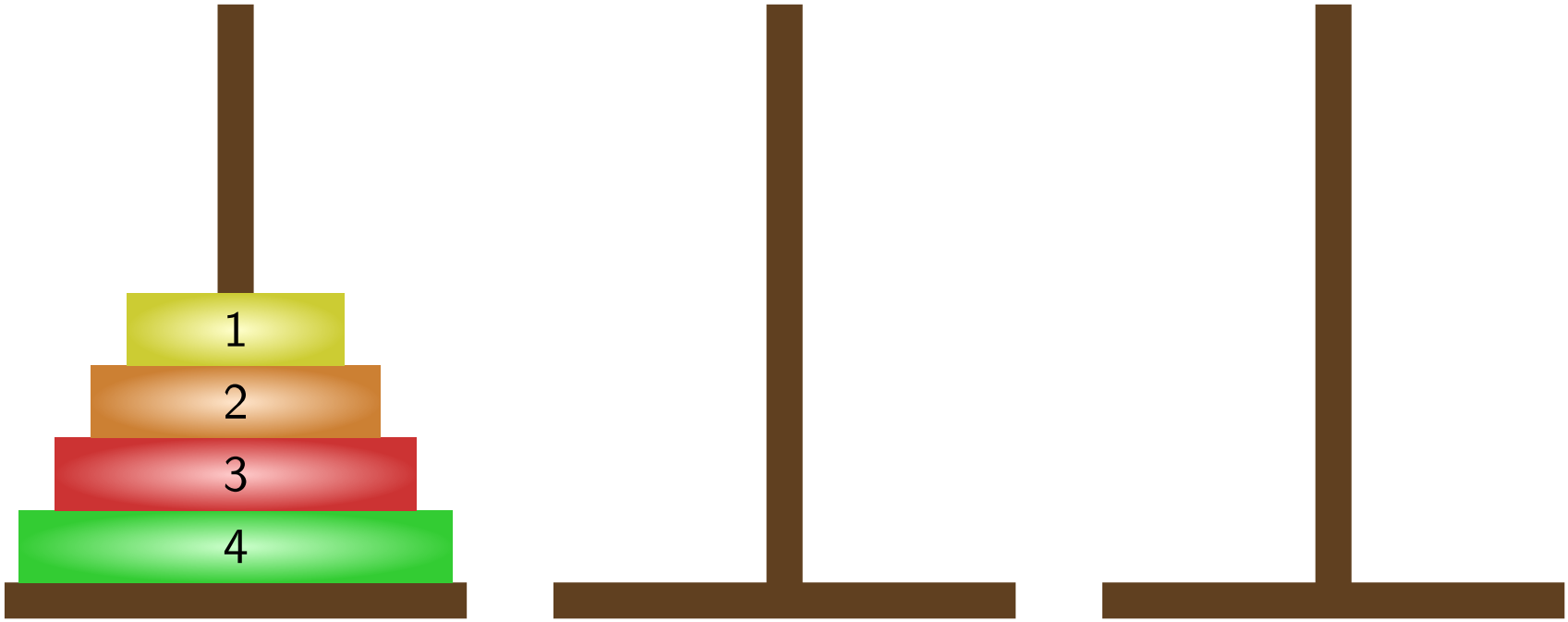


Mudei disco do pino 1 ao pino 3.

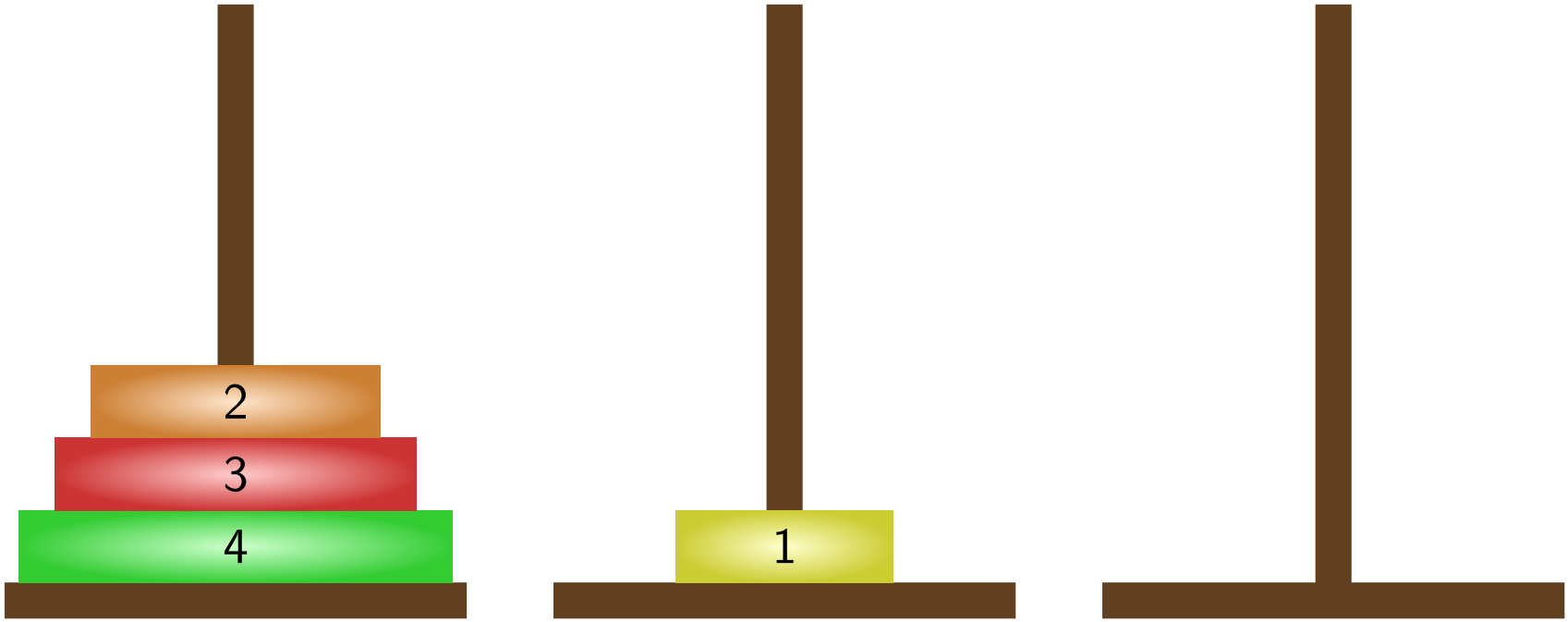
Torre de Hanoi – 3 discos



Torre de Hanoi – 4 discos

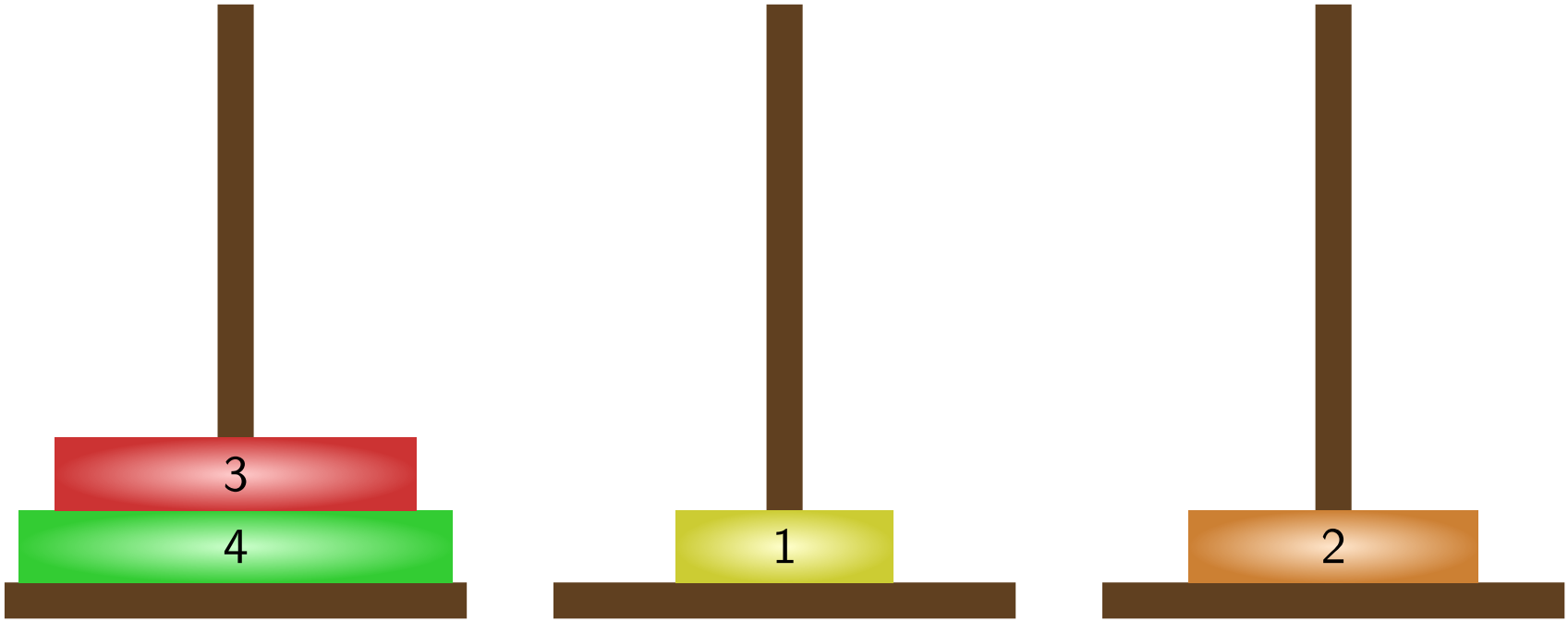


Torre de Hanoi – 4 discos



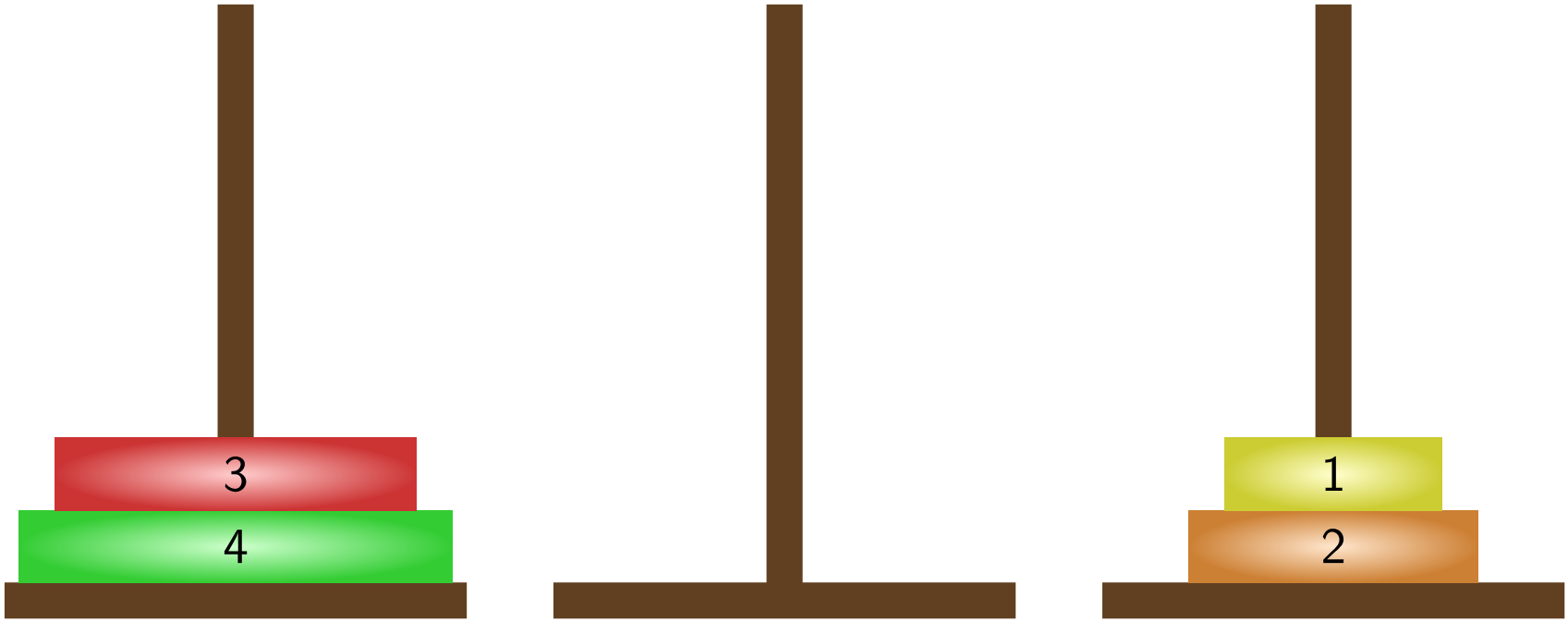
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 4 discos



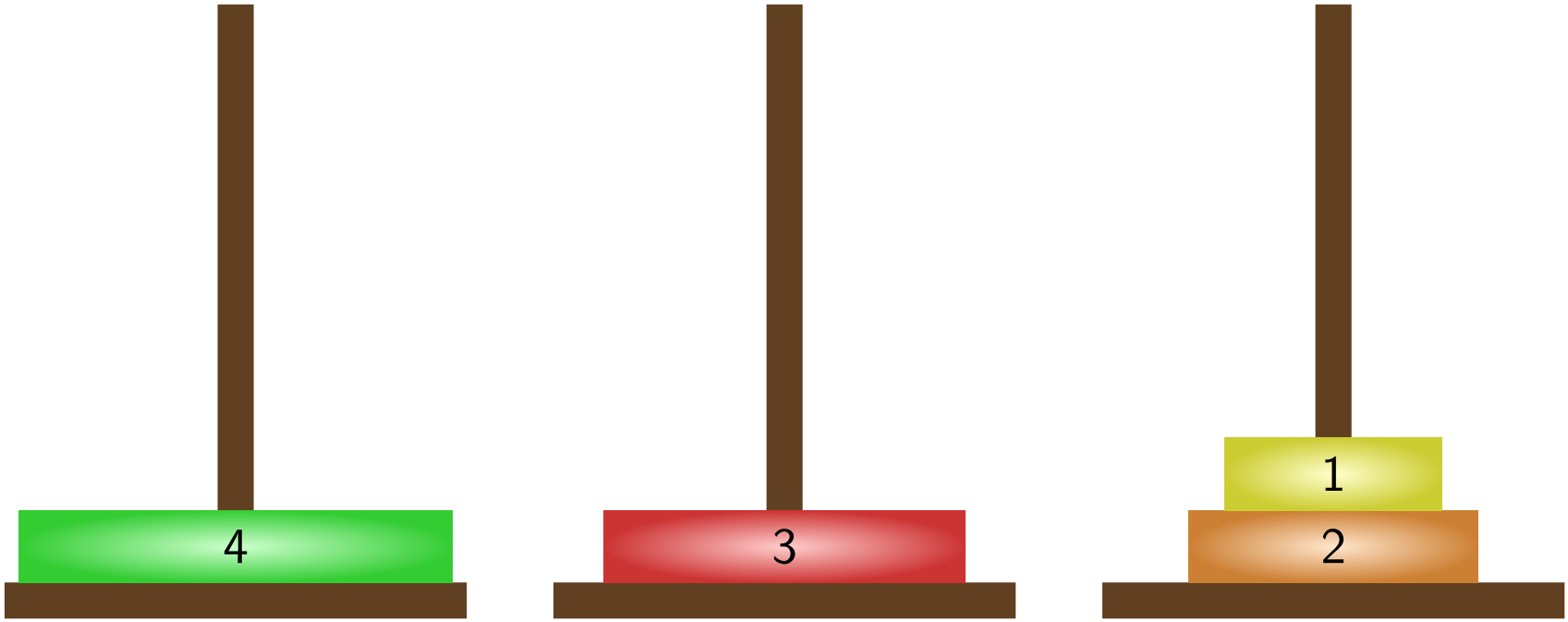
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 4 discos



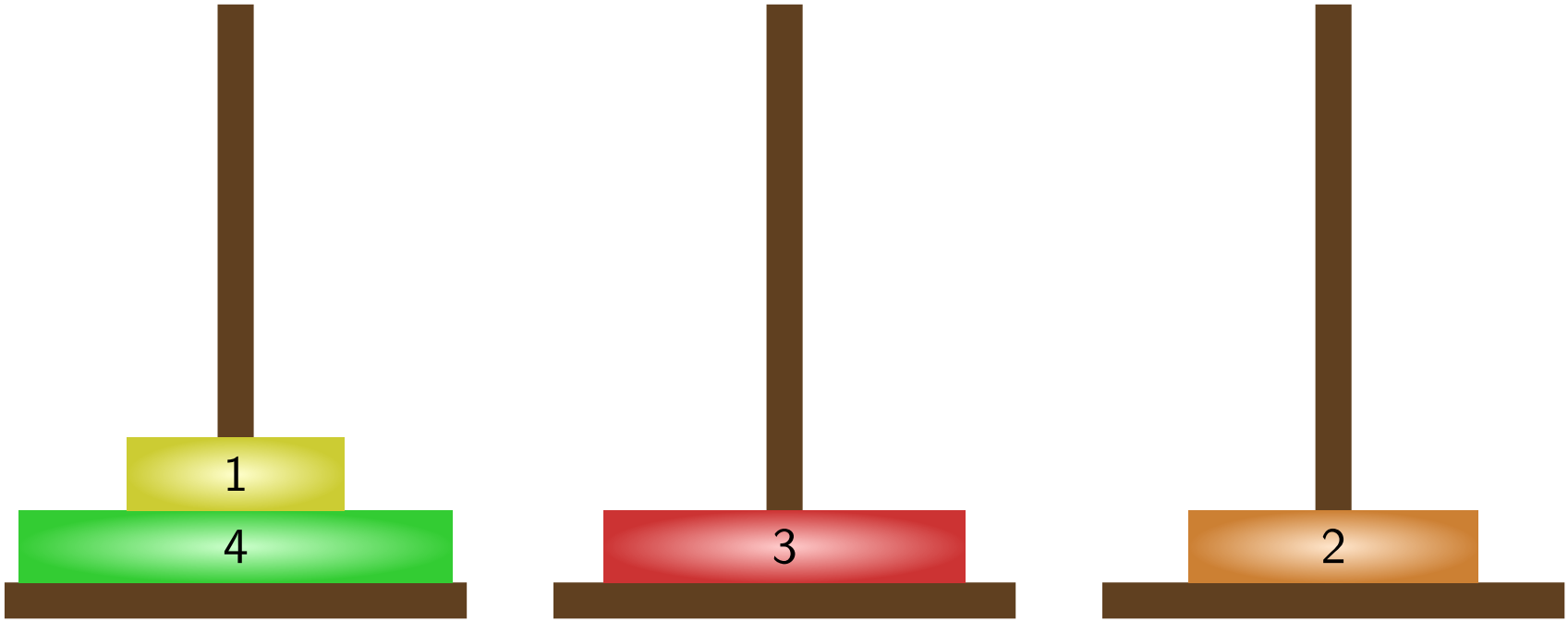
Mudei disco do pino 2 ao pino 3.

Torre de Hanoi – 4 discos



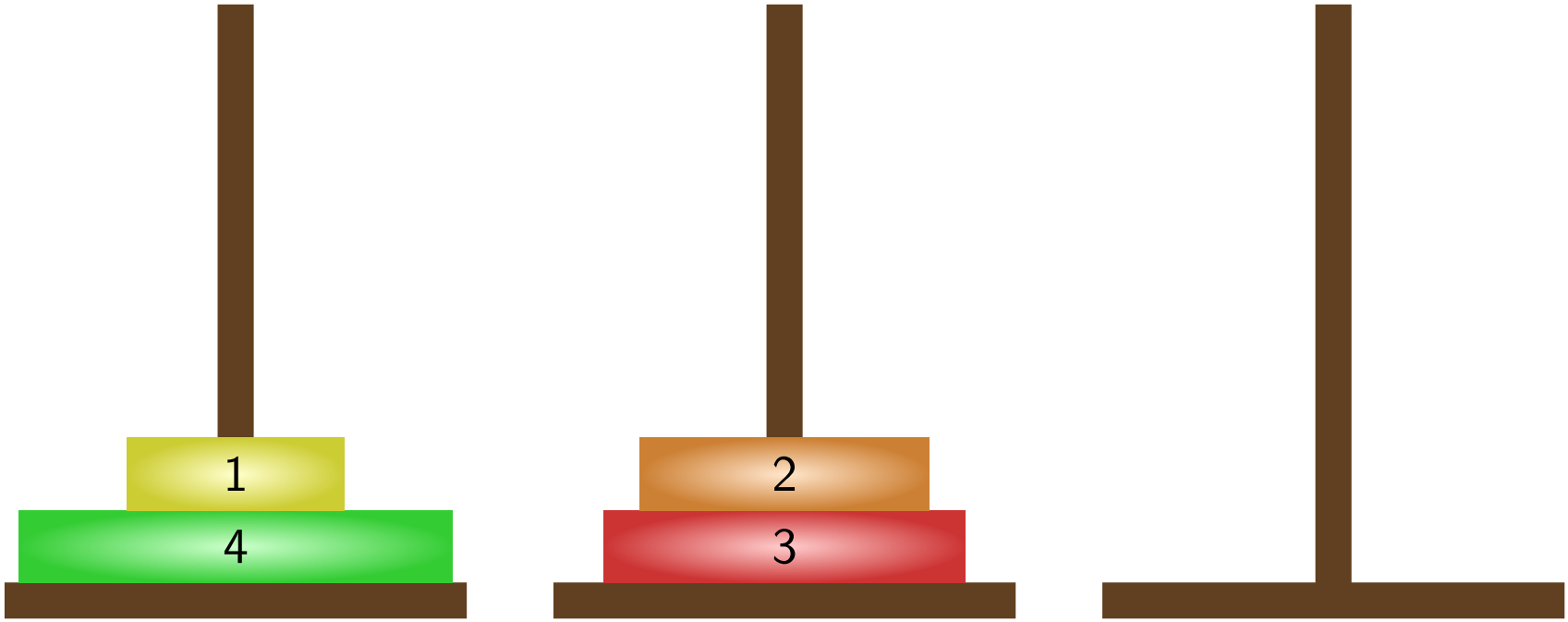
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 4 discos



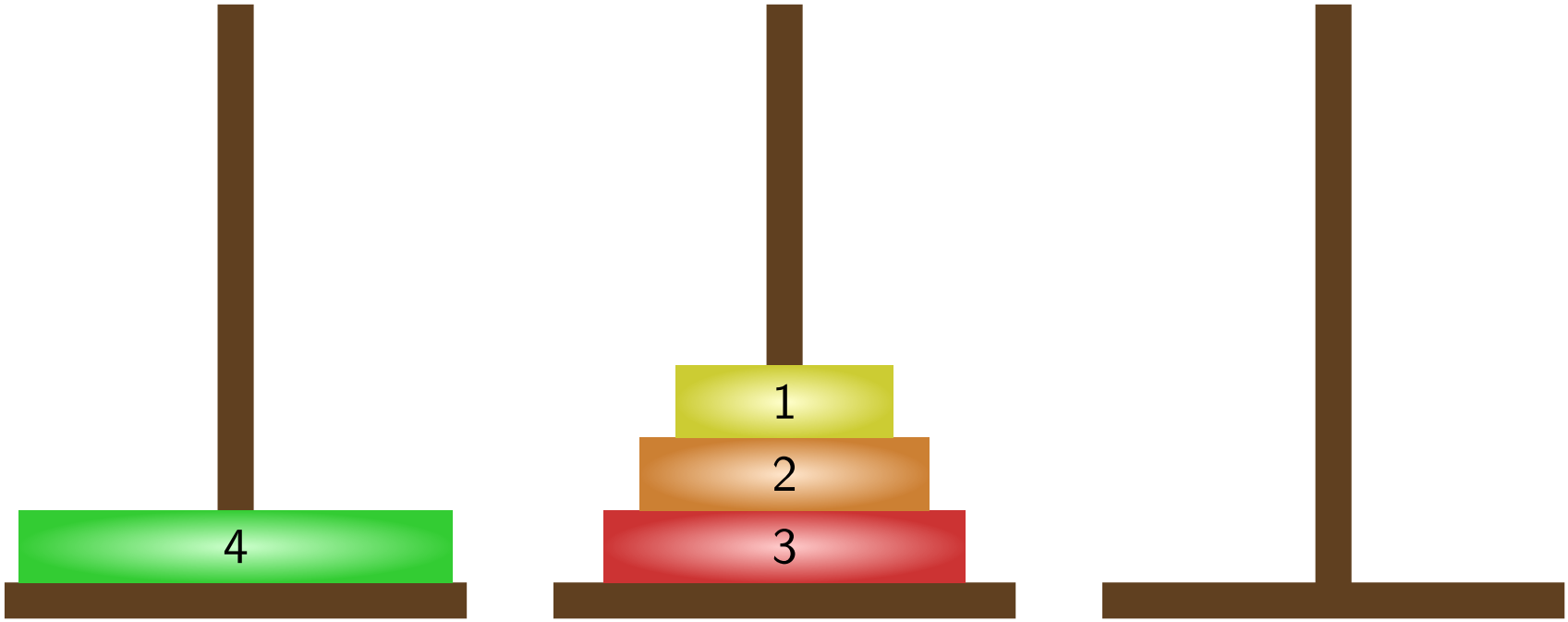
Mudei disco do pino 3 ao pino 1.

Torre de Hanoi – 4 discos



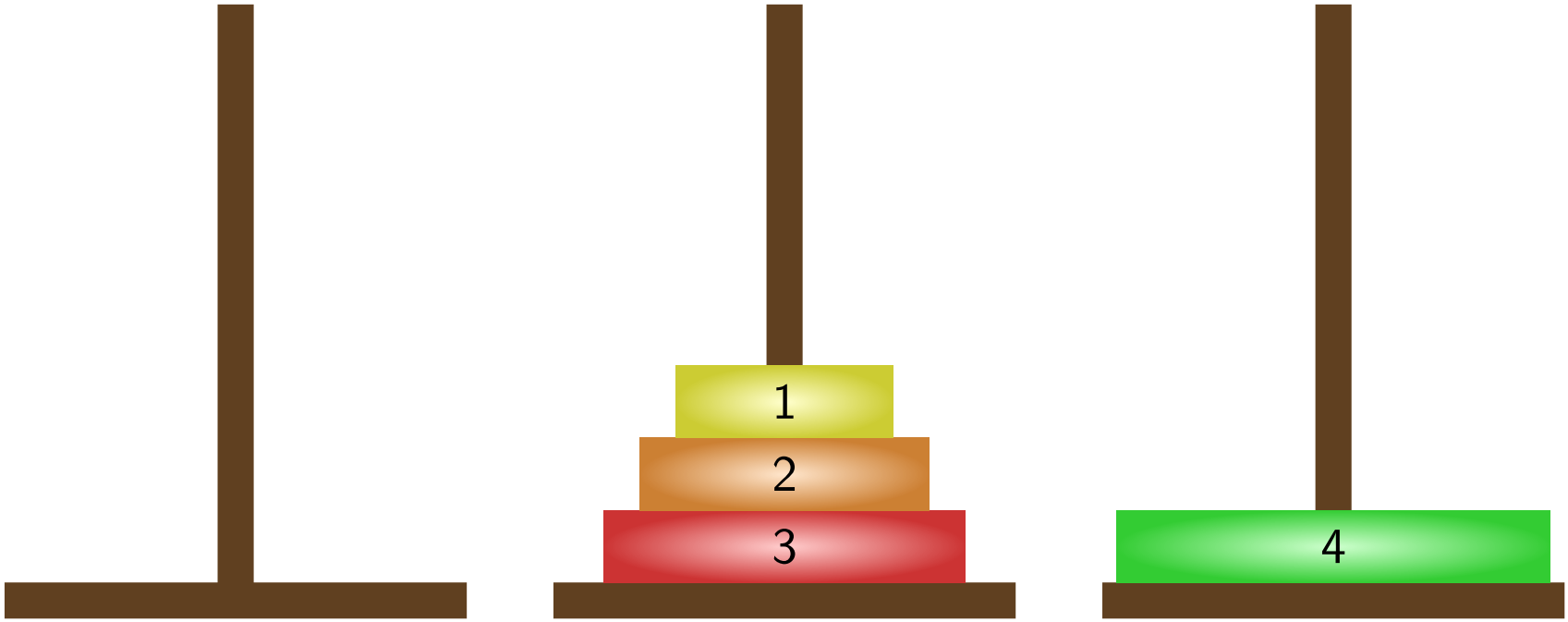
Mudei disco do pino 3 ao pino 2.

Torre de Hanoi – 4 discos



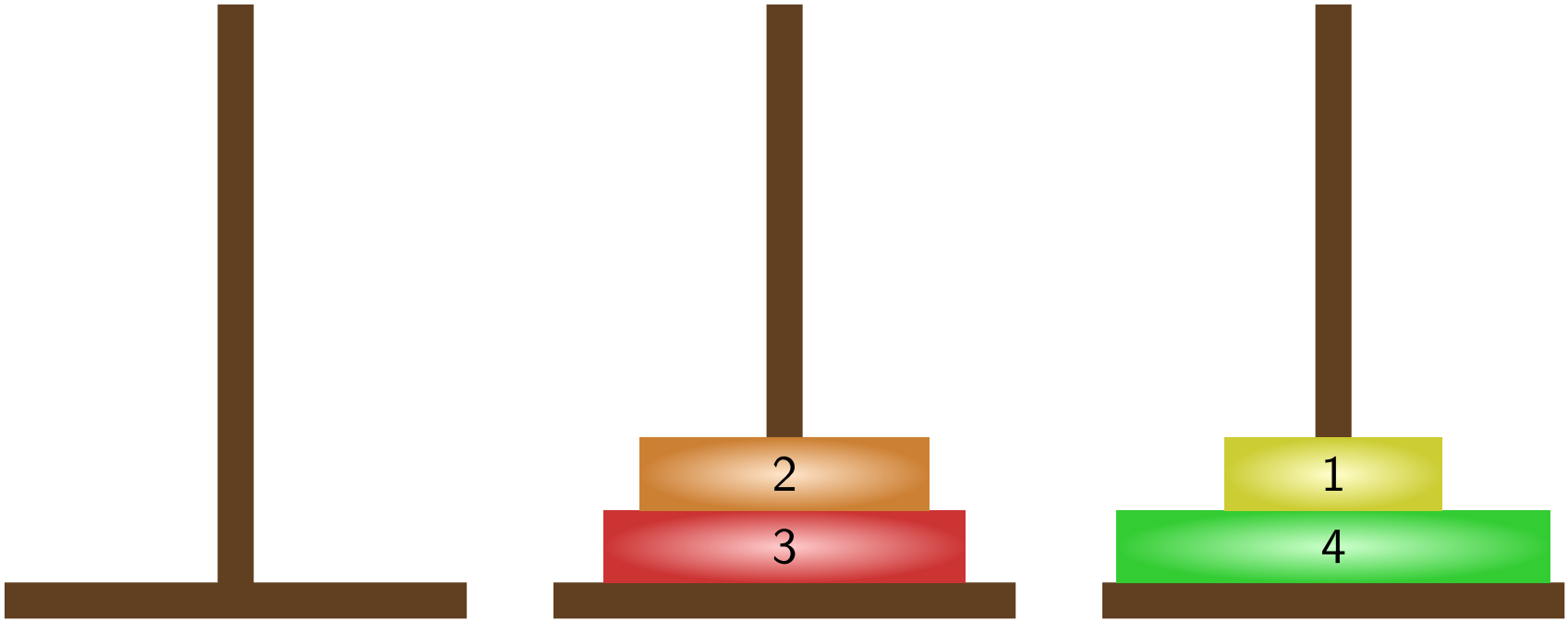
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 4 discos



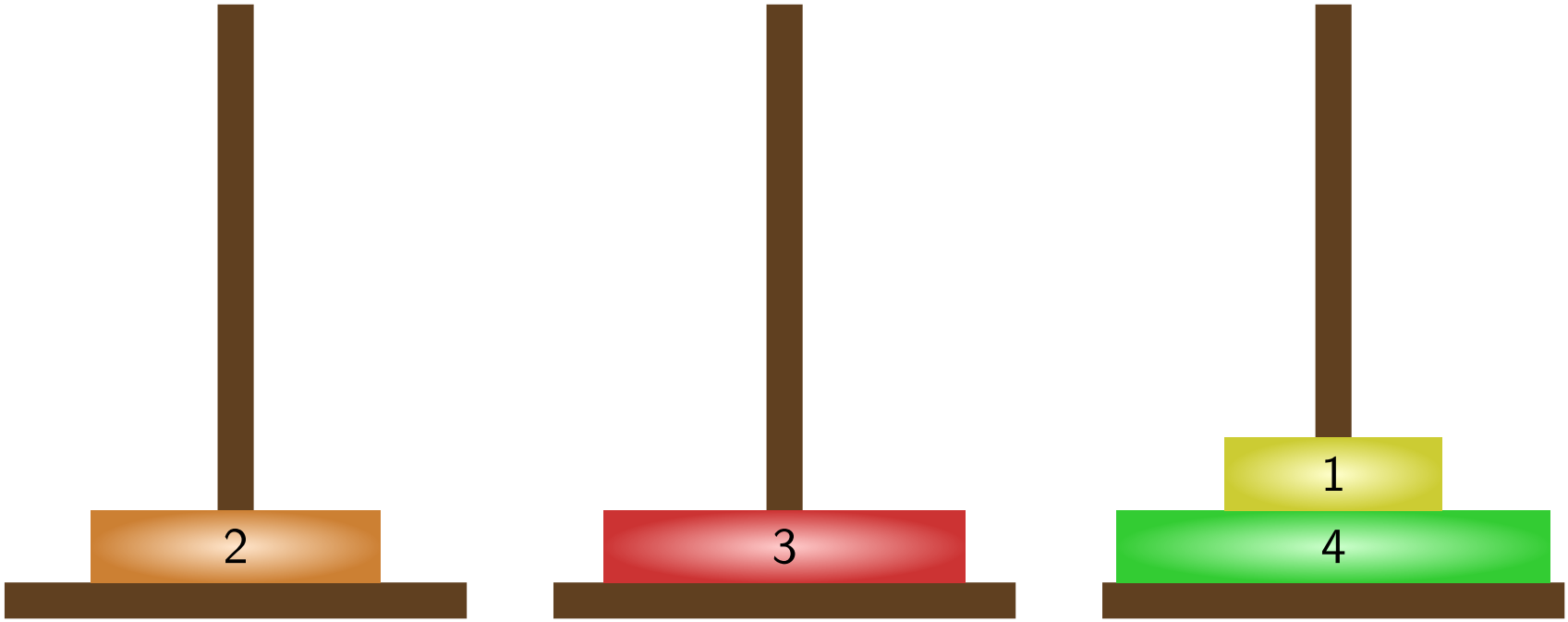
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 4 discos



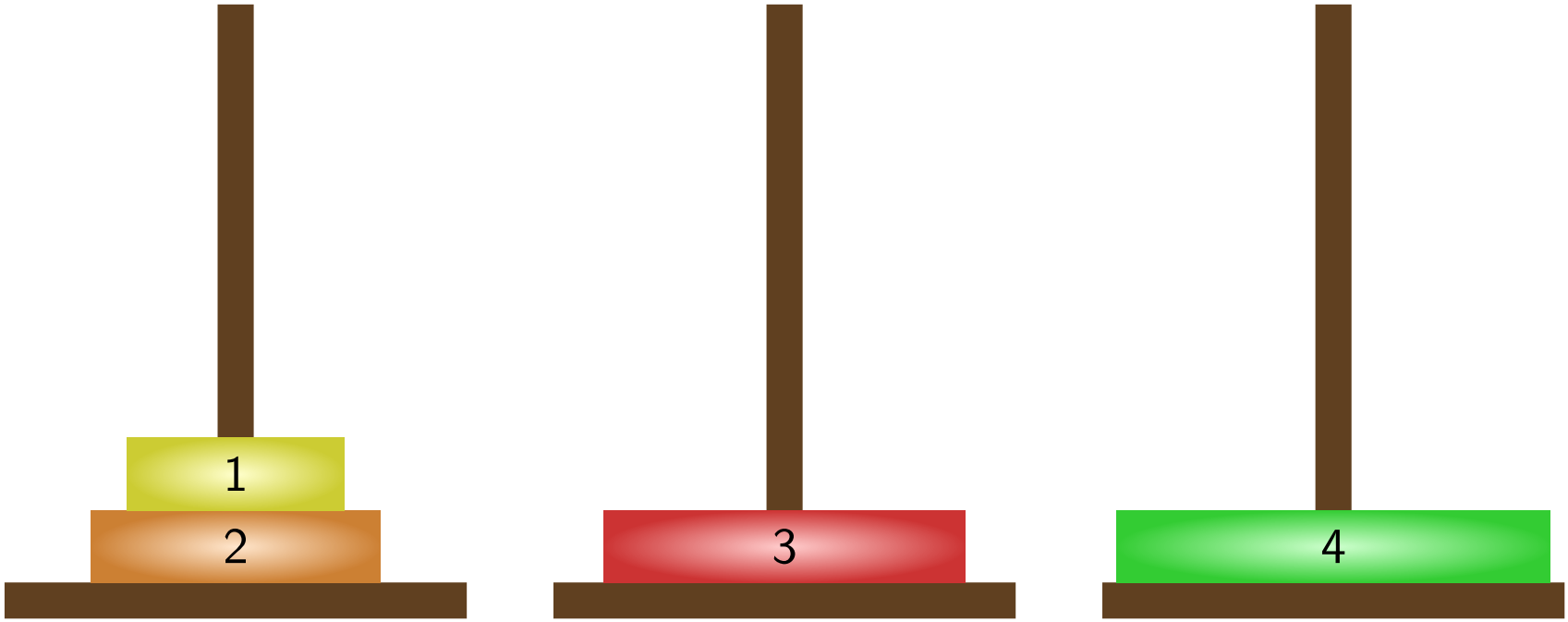
Mudei disco do pino 2 ao pino 3.

Torre de Hanoi – 4 discos



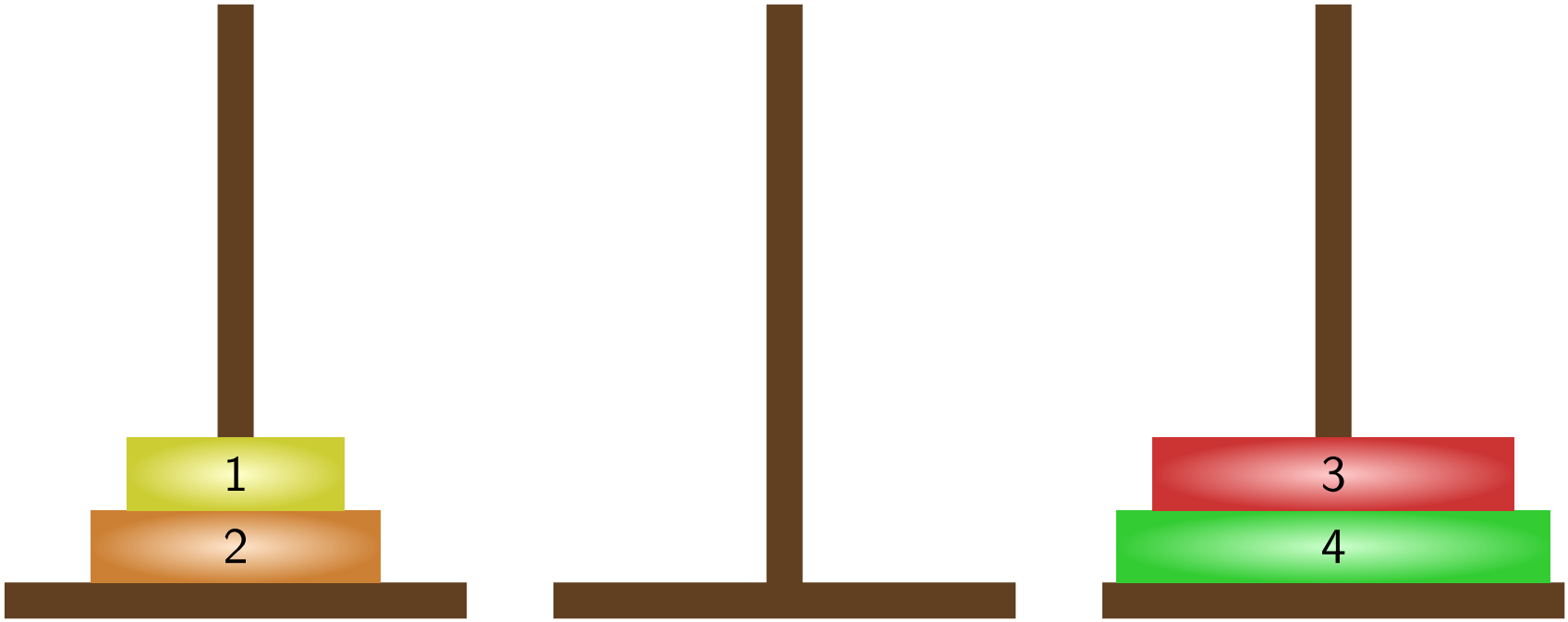
Mudei disco do pino 2 ao pino 1.

Torre de Hanoi – 4 discos



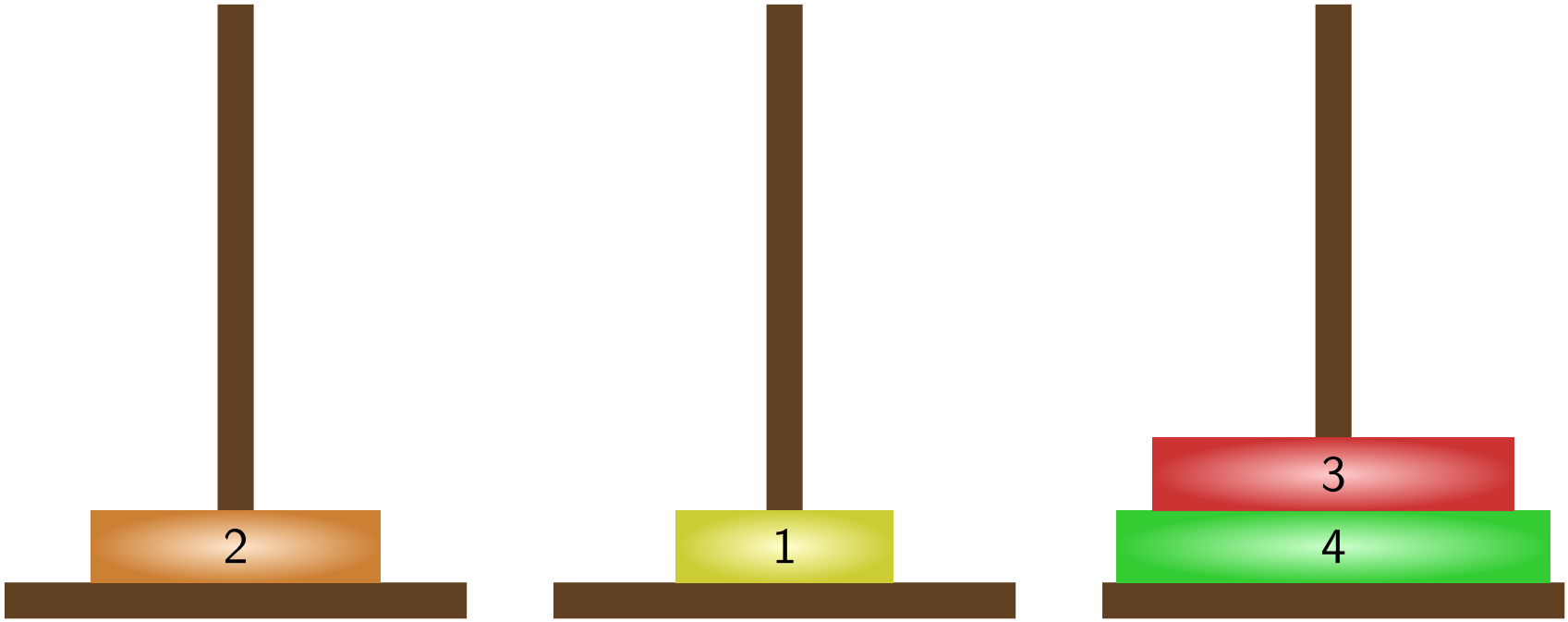
Mudei disco do pino 3 ao pino 1.

Torre de Hanoi – 4 discos



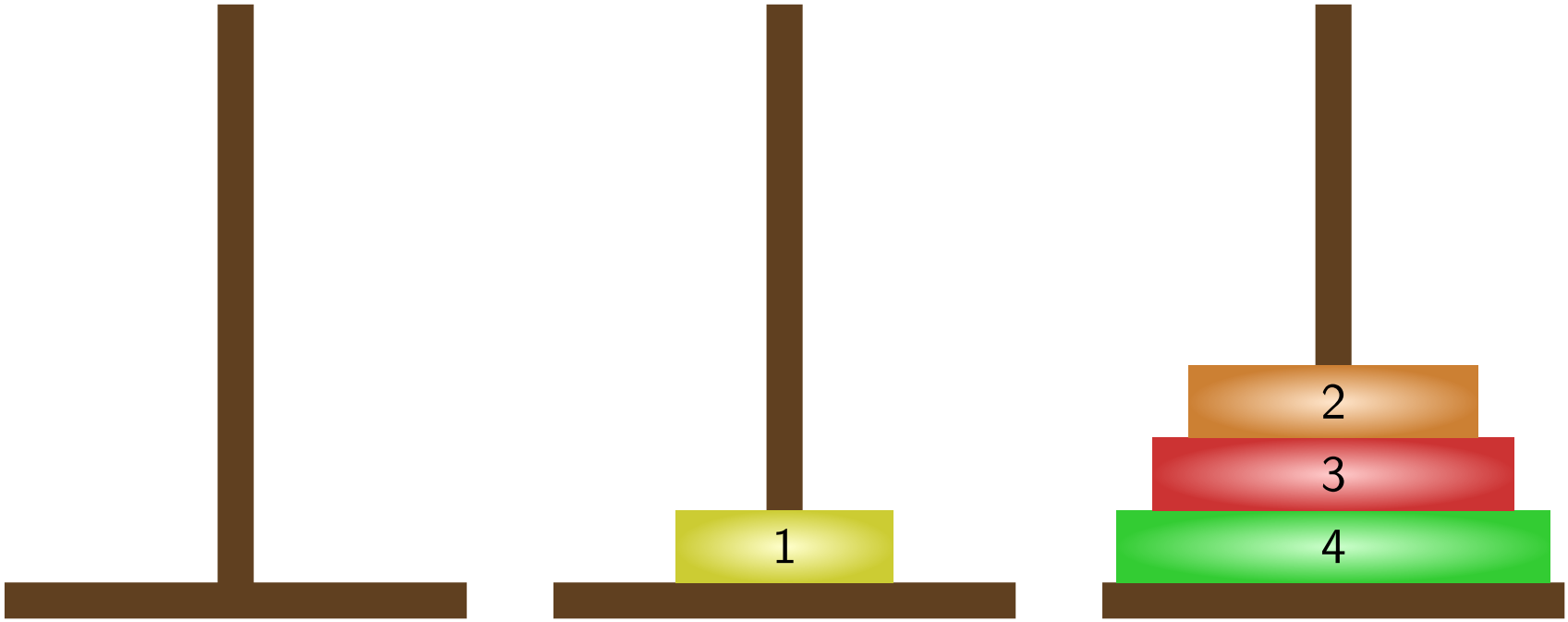
Mudei disco do pino 2 ao pino 3.

Torre de Hanoi – 4 discos



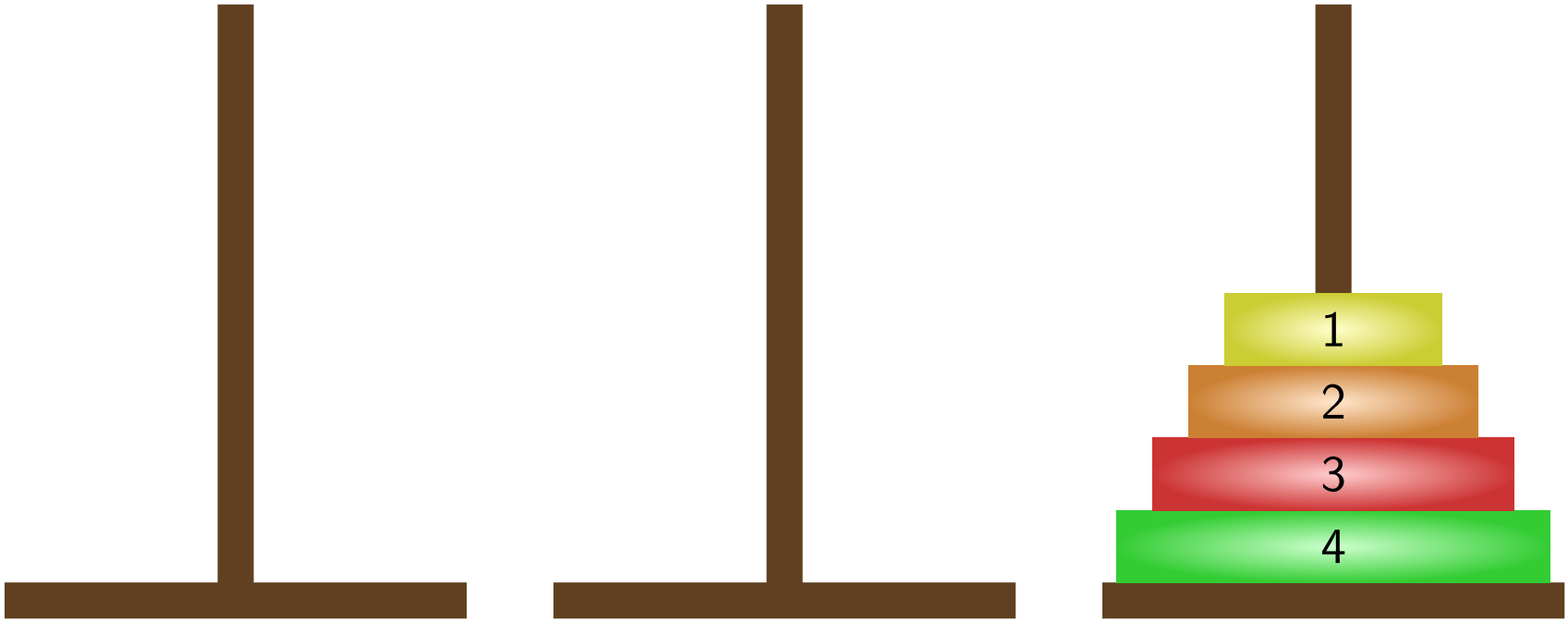
Mudei disco do pino 1 ao pino 2.

Torre de Hanoi – 4 discos



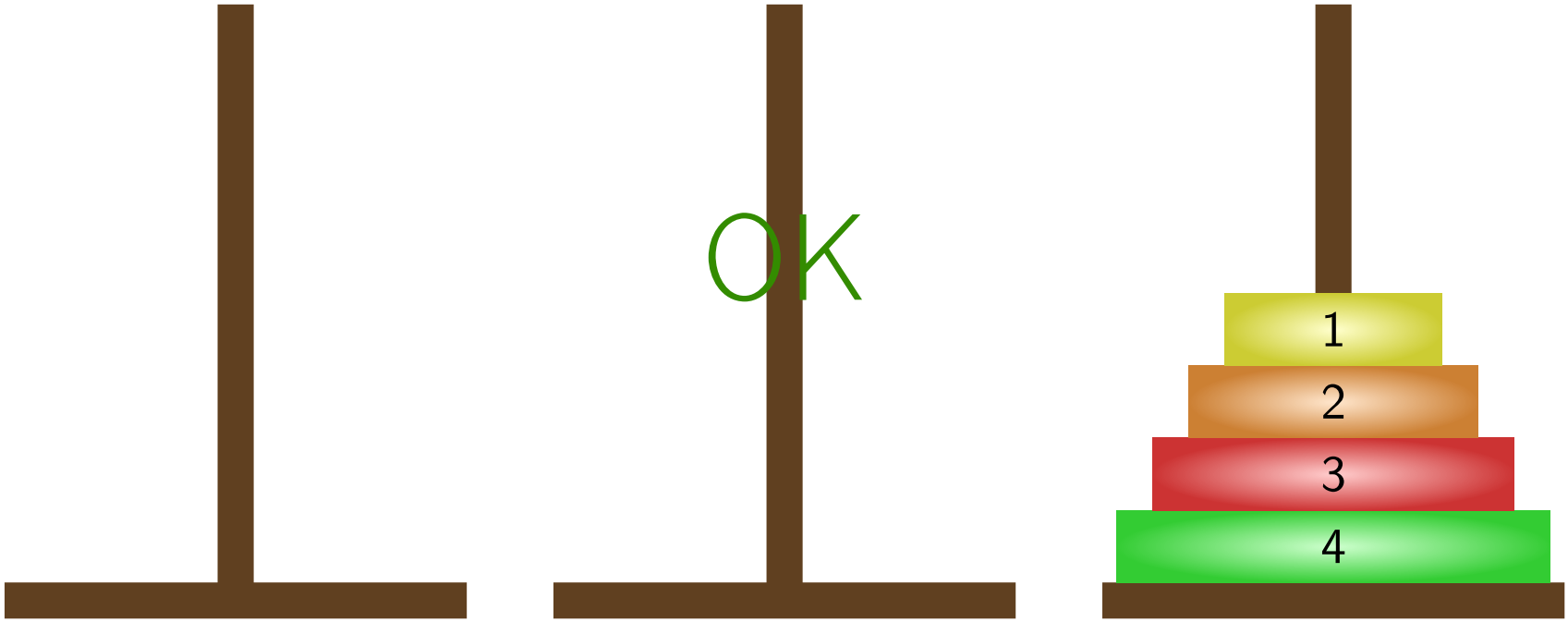
Mudei disco do pino 1 ao pino 3.

Torre de Hanoi – 4 discos

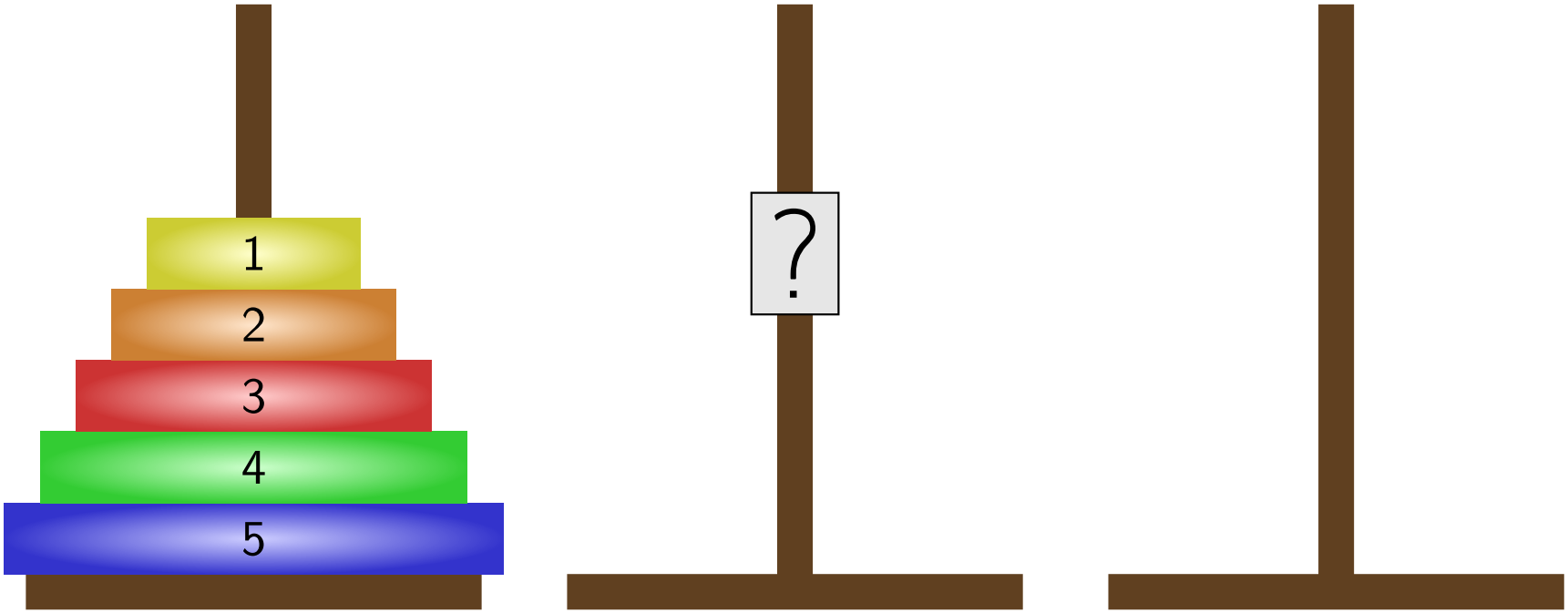


Mudei disco do pino 2 ao pino 3.

Torre de Hanoi – 4 discos



Tower of Hanoi – 5 Disc



Implementação

```
algoritmo "Hanoi"
var  n, contador : inteiro

procedimento movaTorre(n : inteiro; Orig, Dest, Aux : caractere)
inicio
  se n = 1 entao
    movaDisco(Orig, Dest)
  senao
    movaTorre(n - 1, Orig, Aux, Dest)
    movaDisco(Orig, Dest)
    movaTorre(n - 1, Aux, Dest, Orig)
  fimse
fimprocedimento

procedimento movaDisco(Orig, Dest : caractere)
inicio
  escreval ("Movimento: ", Orig, " -> ", Dest)
  contador <- contador + 1
fimprocedimento

inicio
  escreval ("Torre de Hanoi")
  escreva ("Informe a quantidade de discos a mover: ")
  leia (n)
  contador <- 0
  movaTorre(n, "A", "C", "B")
  escreva ("Número de movimentos: ", contador)
fimalgoritmo
```

O problema da ordenação

Definição

O **problema de ordenação** tem a definição seguinte:

Entrada Uma sequência de n números $(a_1, a_2, \ldots, a_n)$

Saída Uma permutação (re-ordenação) $(a'_1, a'_2, \ldots, a'_n)$ da sequência de entrada tal que $a'_1 \leq a'_2 \leq \ldots \leq a'_n$

Observação

- Em Portugol, sequência = **vetor**

Especificação

- O *quicksort* adota uma estrategia tipo **divisão e conquista**
- Rearranjar os valores de modo que as chaves menores precedam as maiores
- Ordenar separadamente os dois subproblemas: as chaves menores de um lado, as maiores do outo

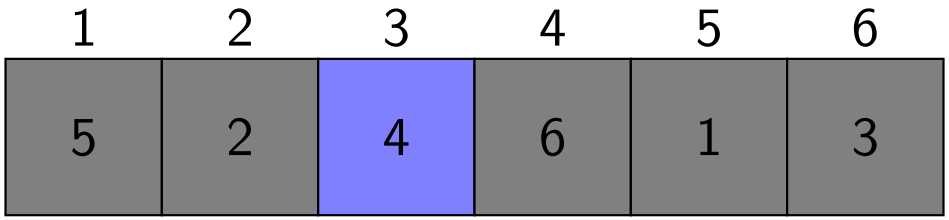
C.A.R. Hoare(1934–)

- Prêmio Turing 1980

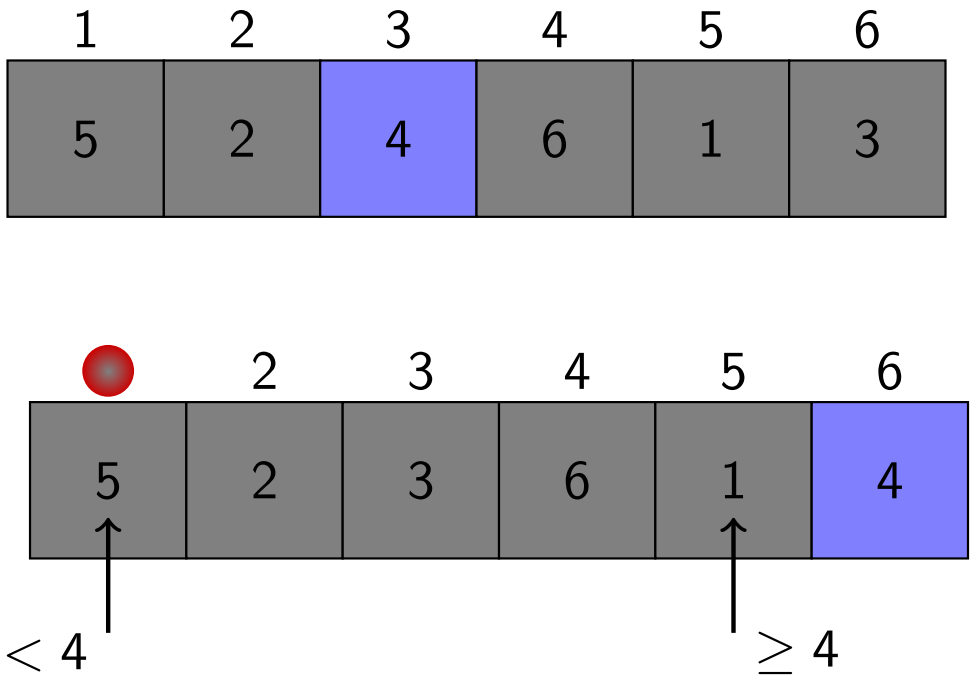
Passos

1. Escolher um elemento do vetor: o **pivô**
2. Rearranjar o vetor de forma que:
   1. Todos os elementos anteriores ao pivô sejam menores do que ele, e
   2. Os elementos posteriores sejam maiores
   3. Ao fim do processo, o pivô esta na posição final certa, com dois subvetores não ordenados.
3. **Recursivamente**, ordenar os subvetores.

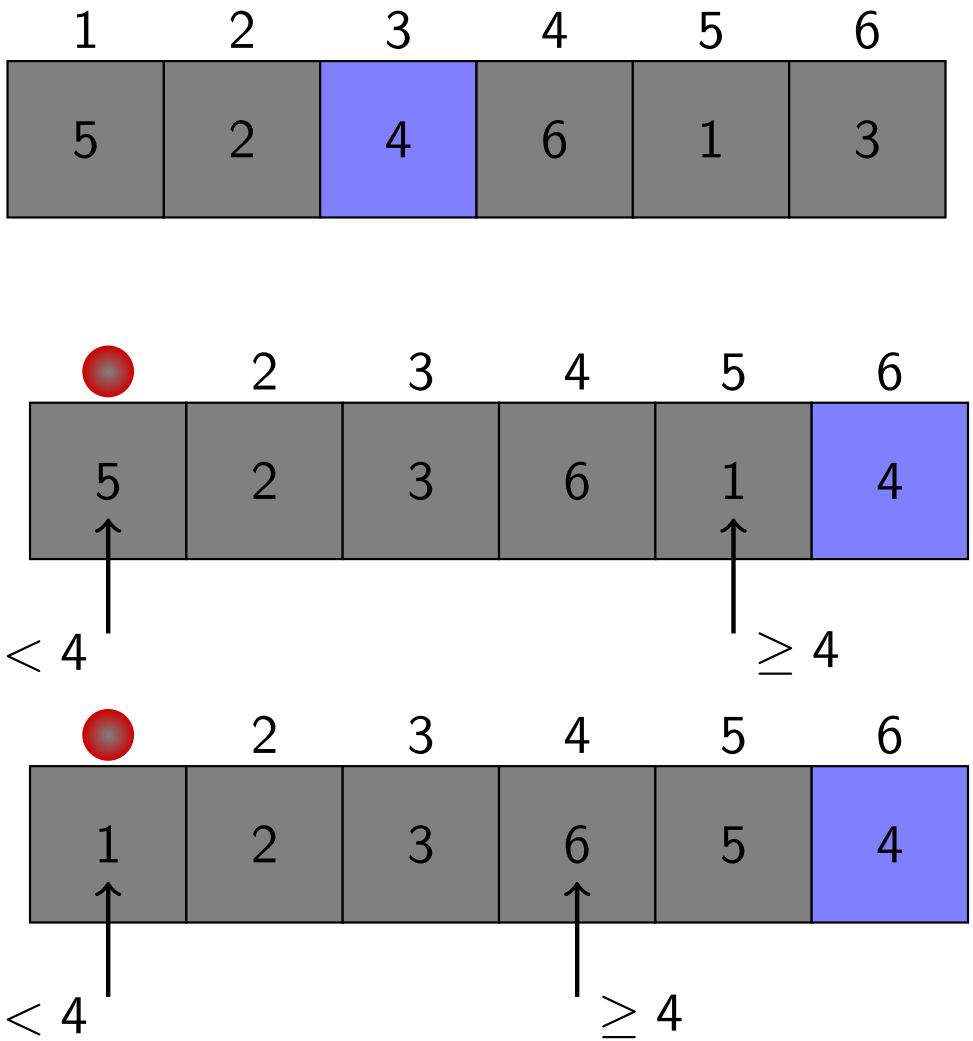
Execução



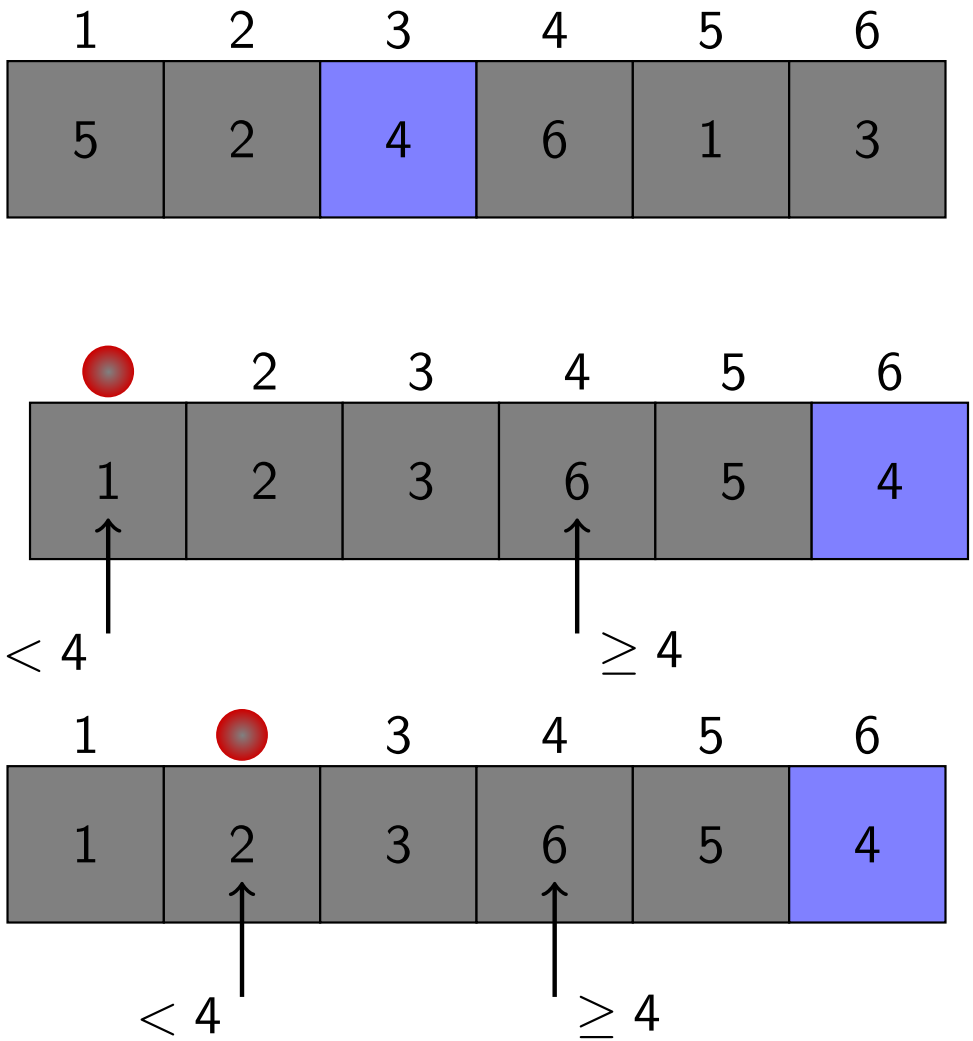
Execução



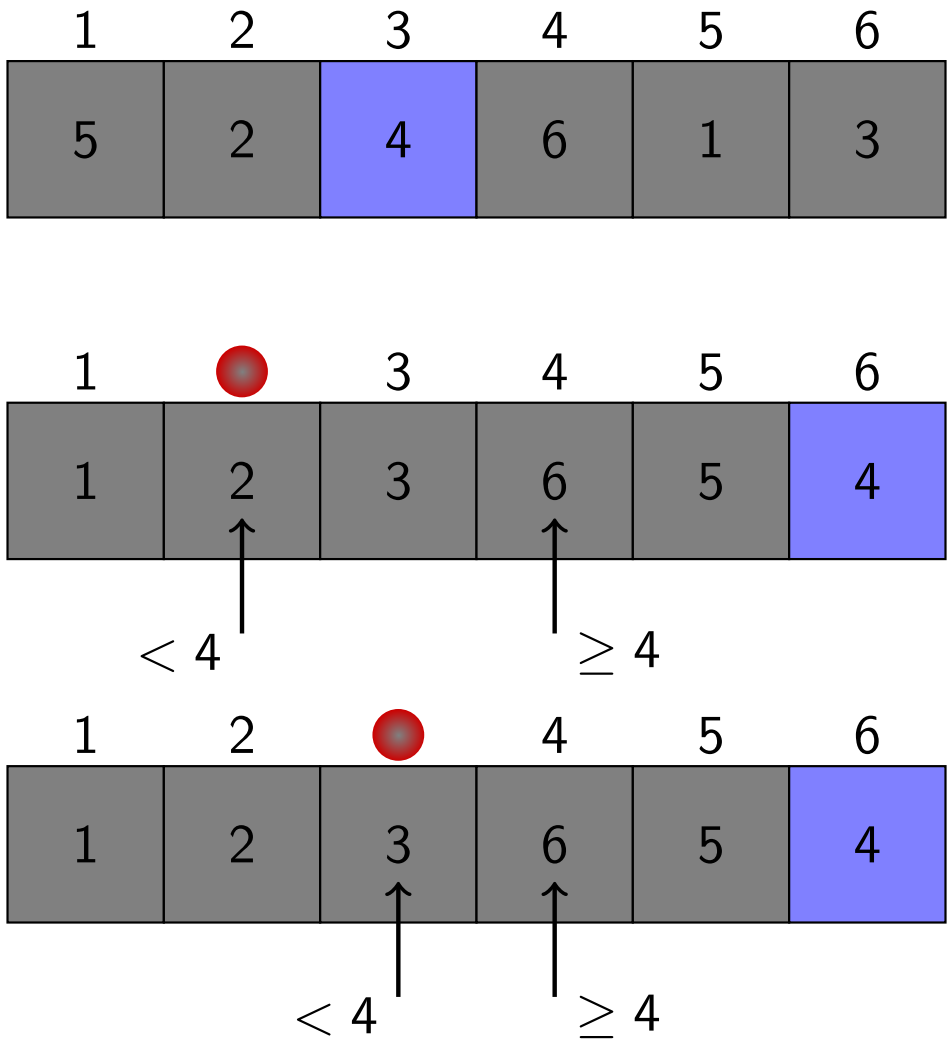
Execução



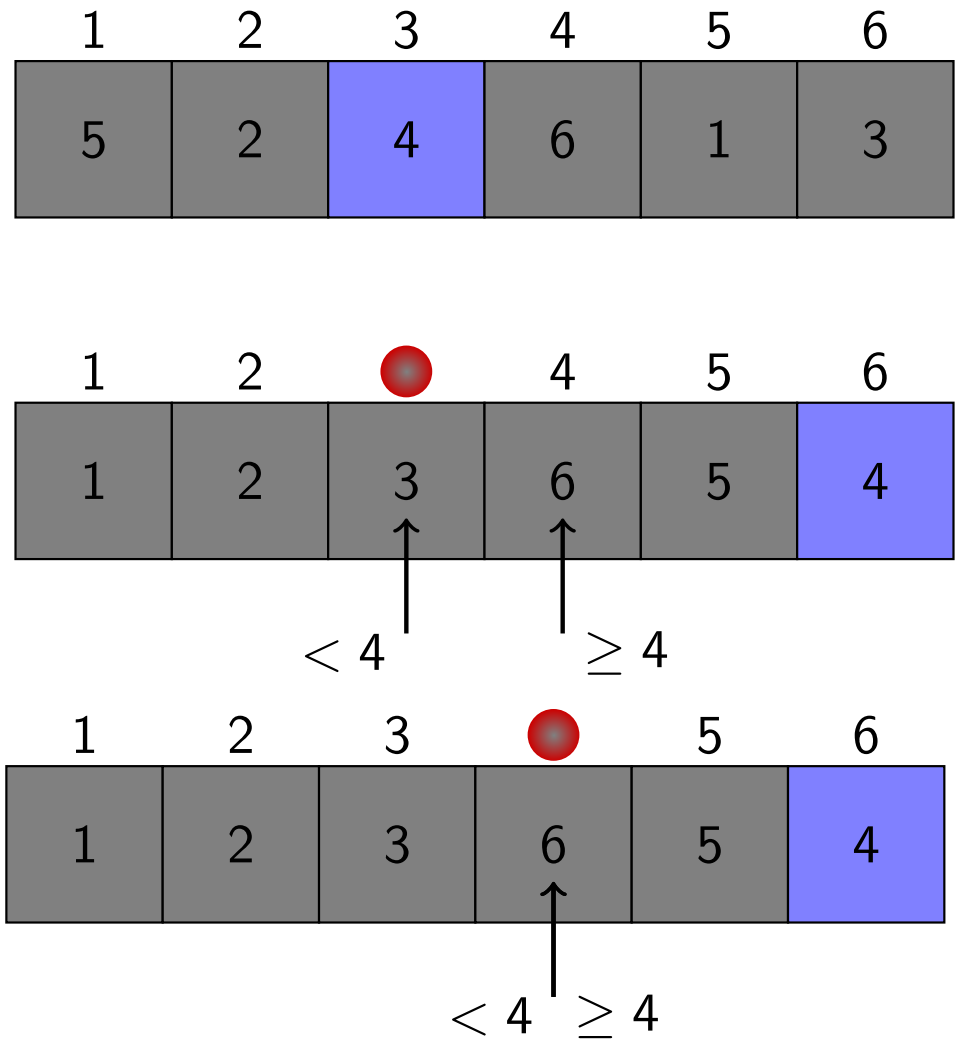
Execução



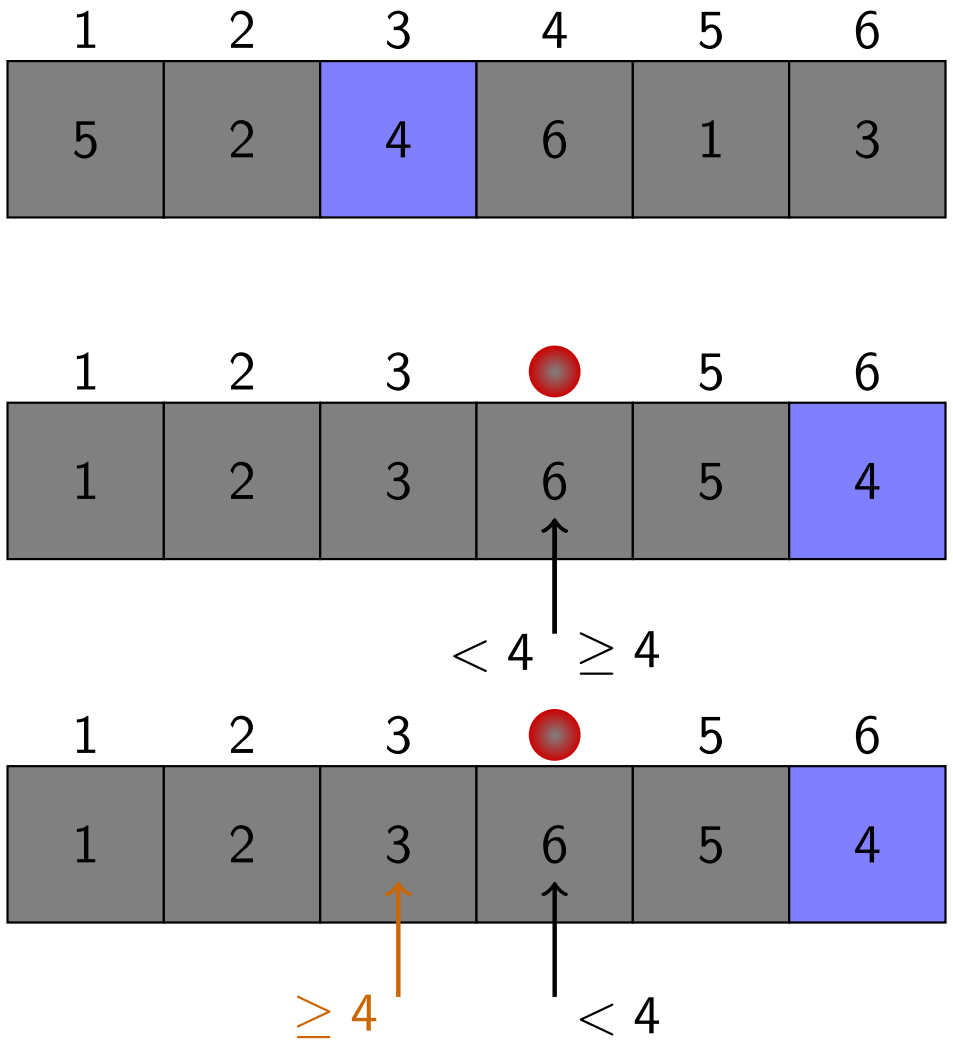
Execução



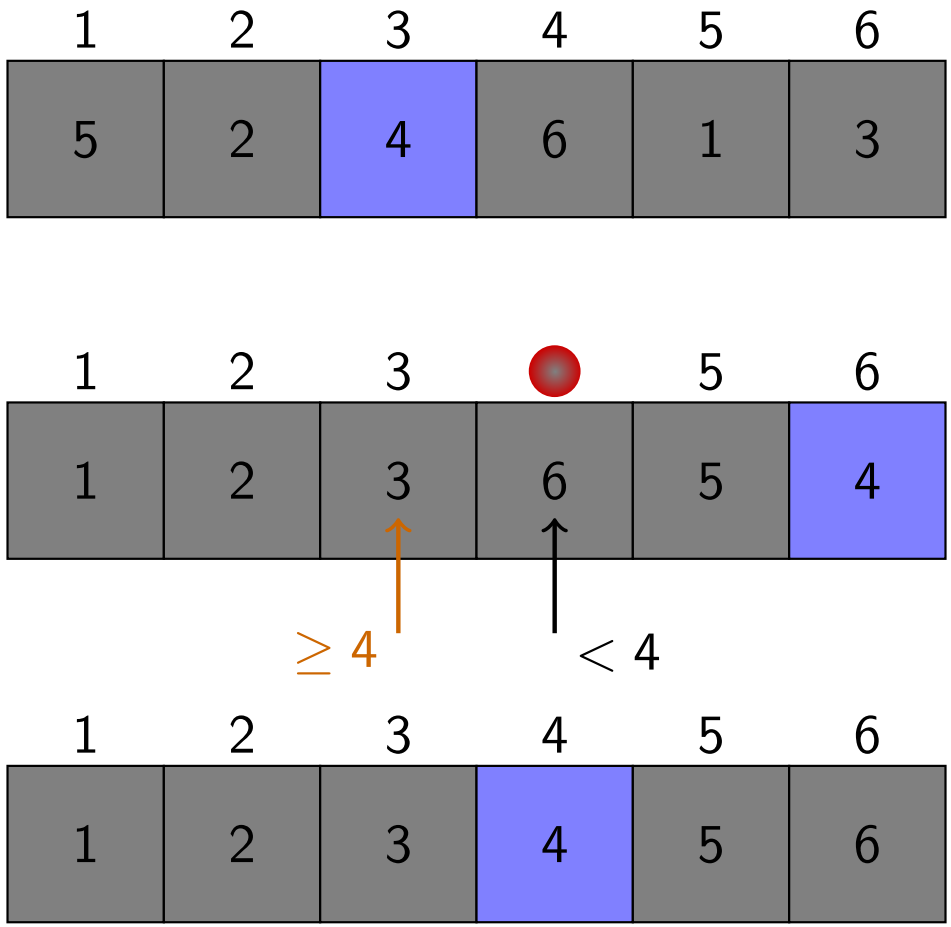
Execução



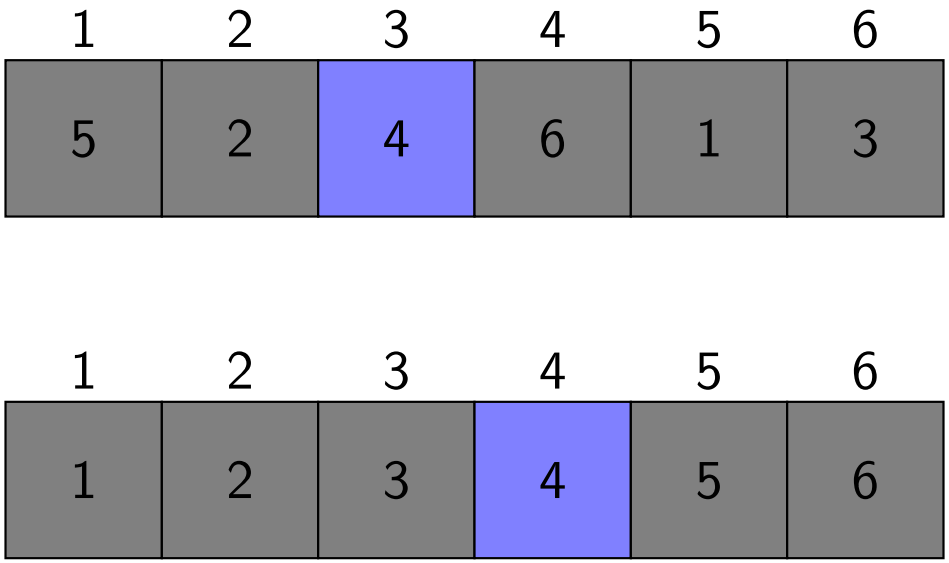
Execução



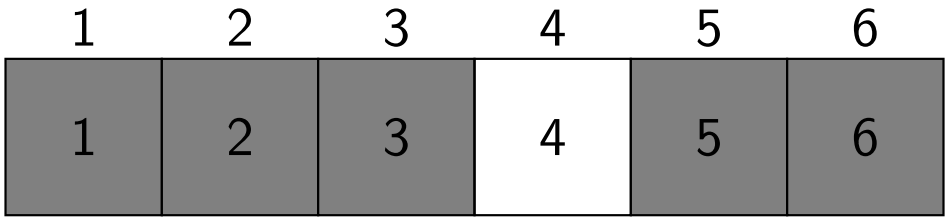
Execução



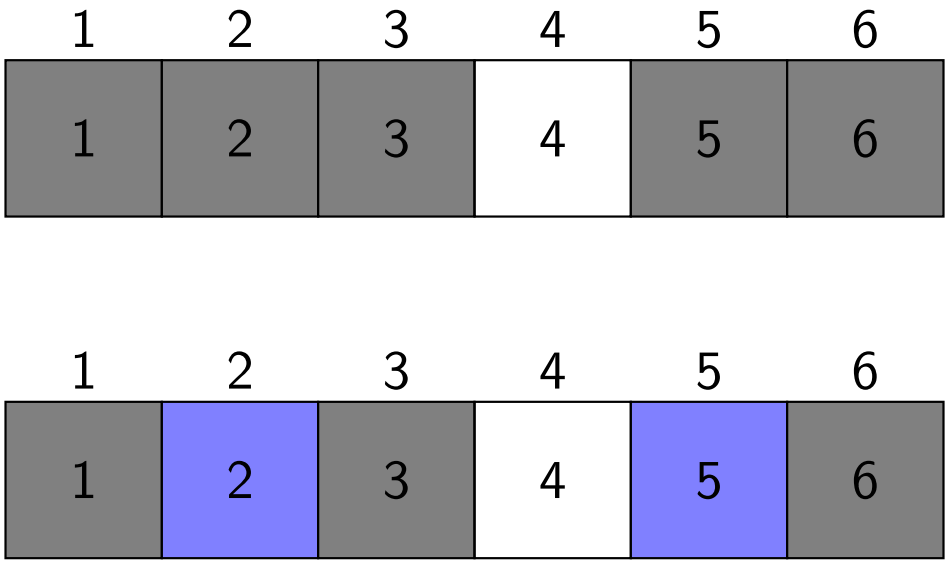
Execução



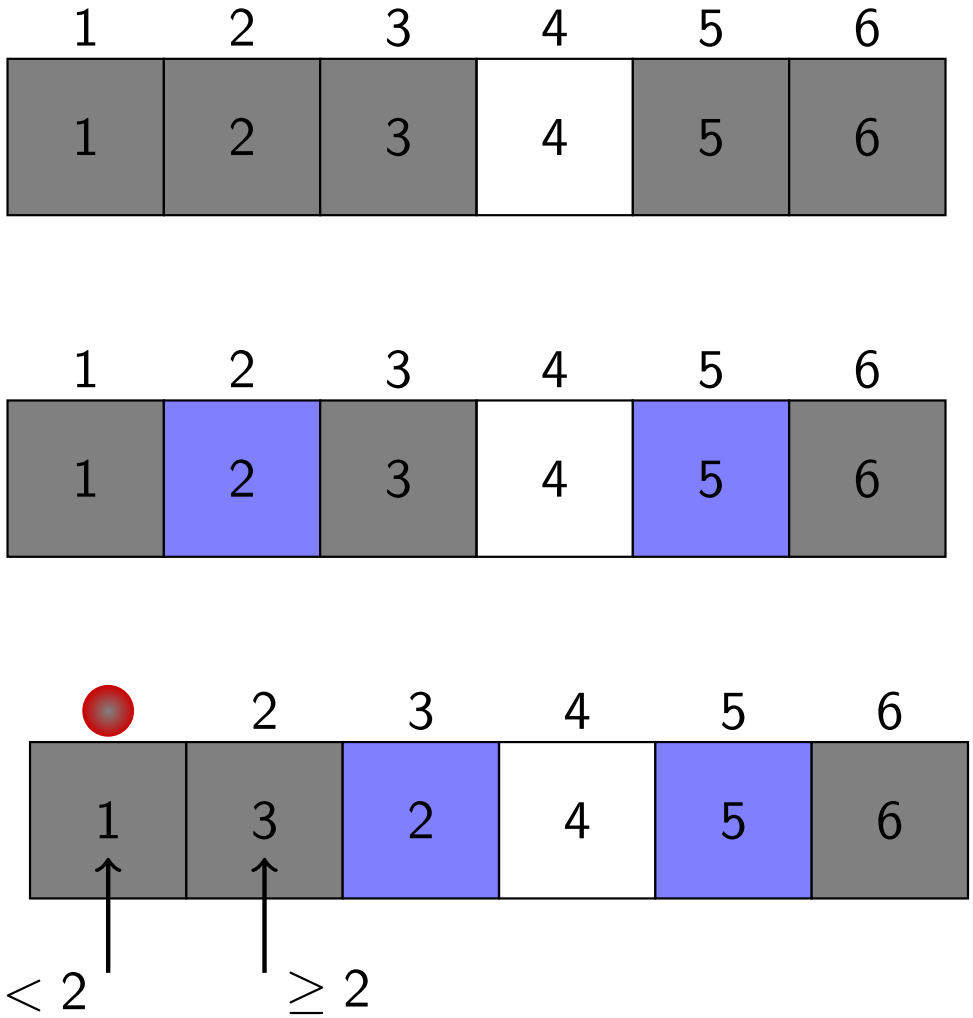
Execução recursiva



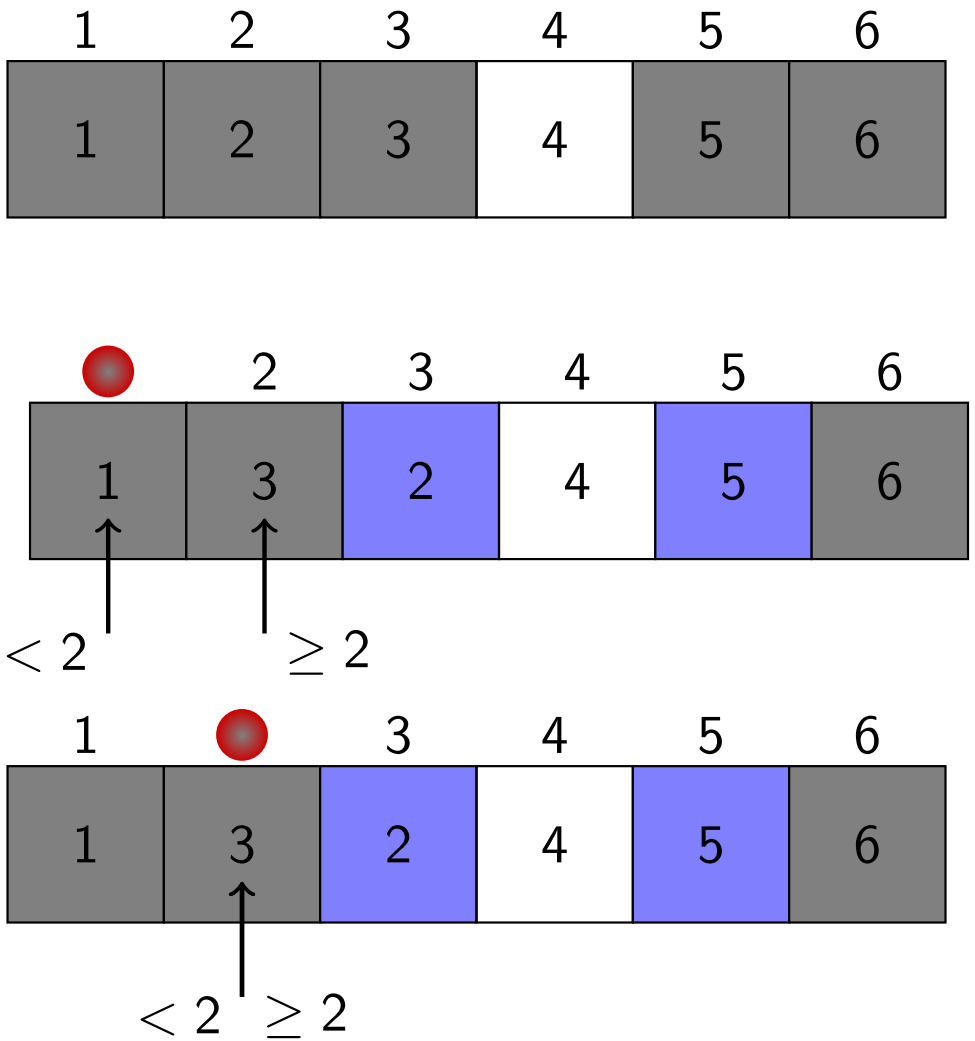
Execução recursiva



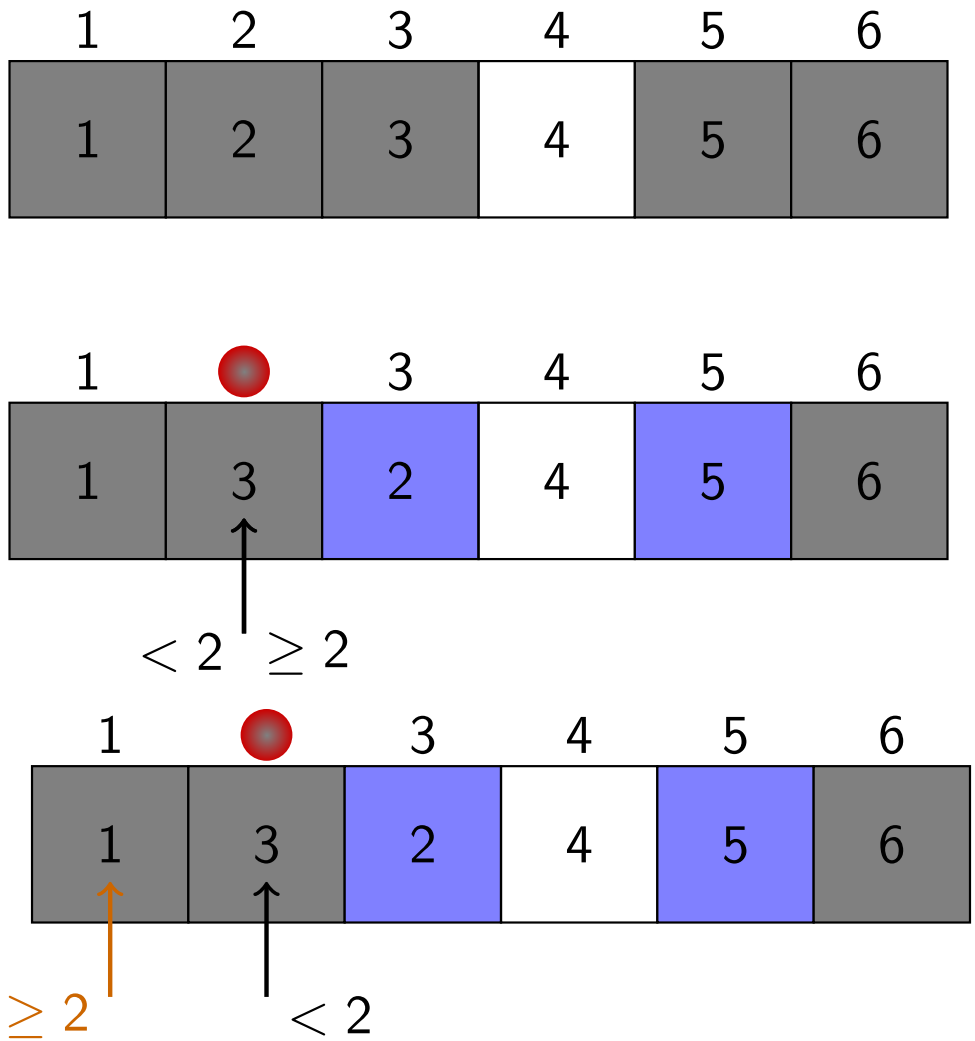
Execução recursiva



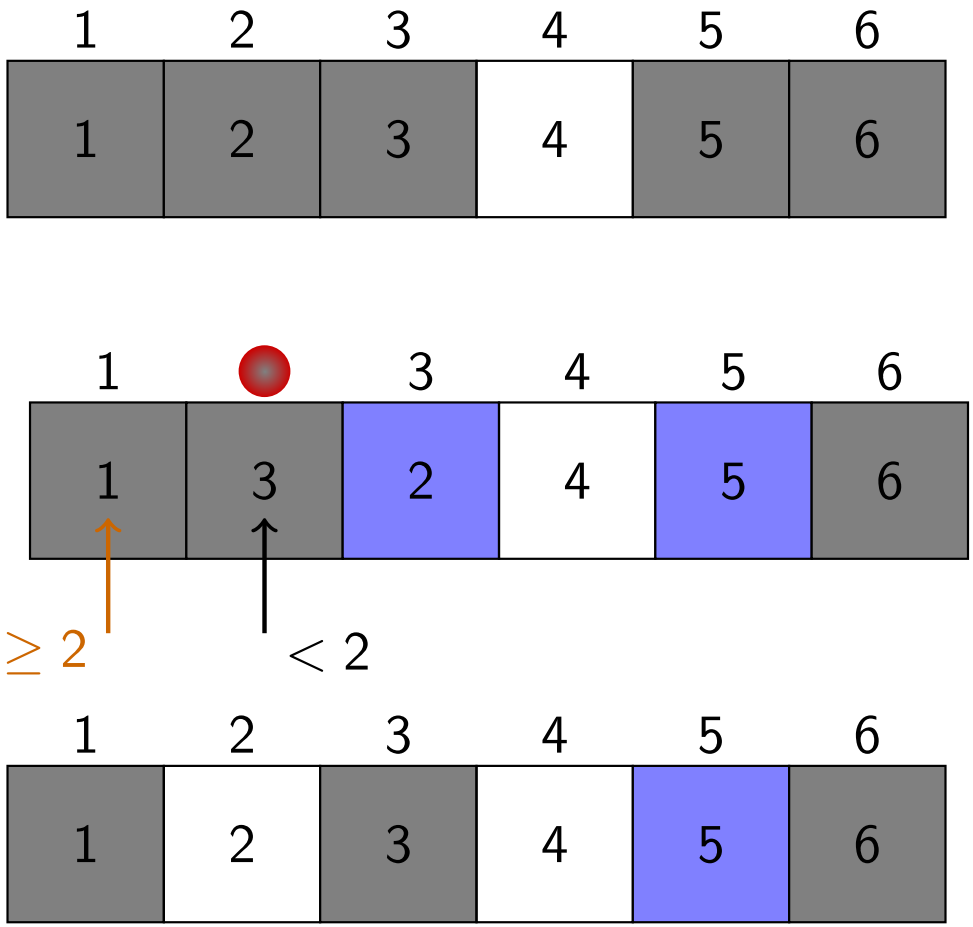
Execução recursiva



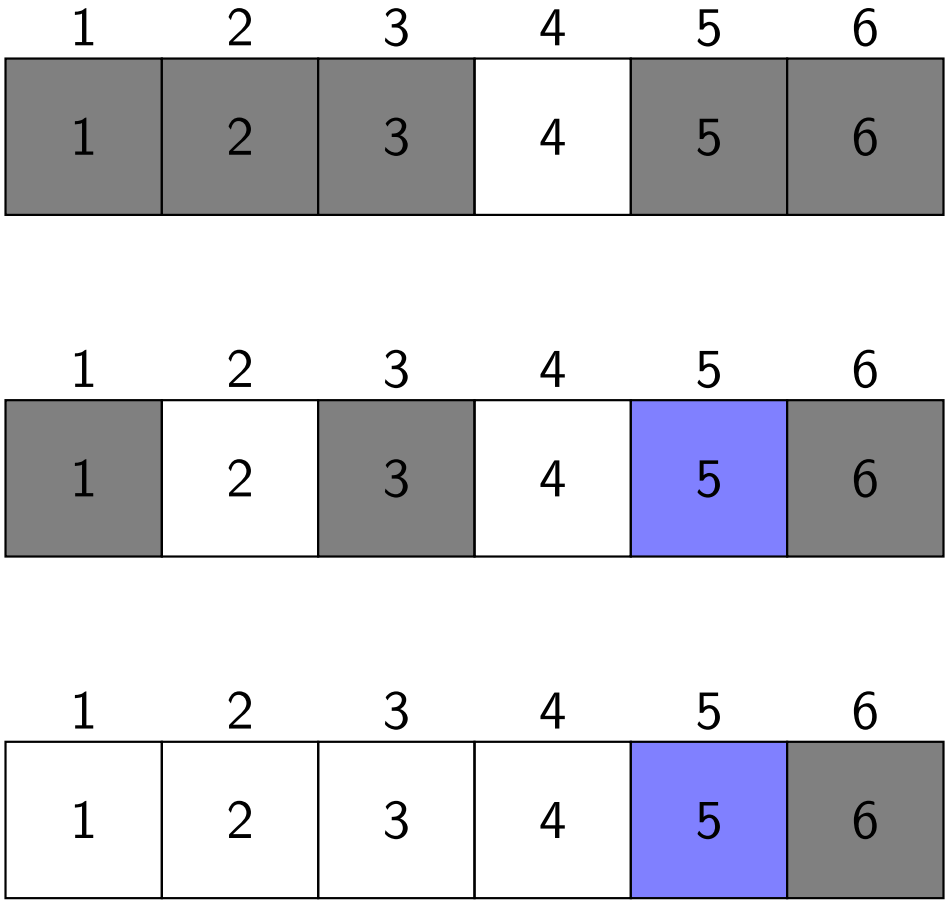
Execução recursiva



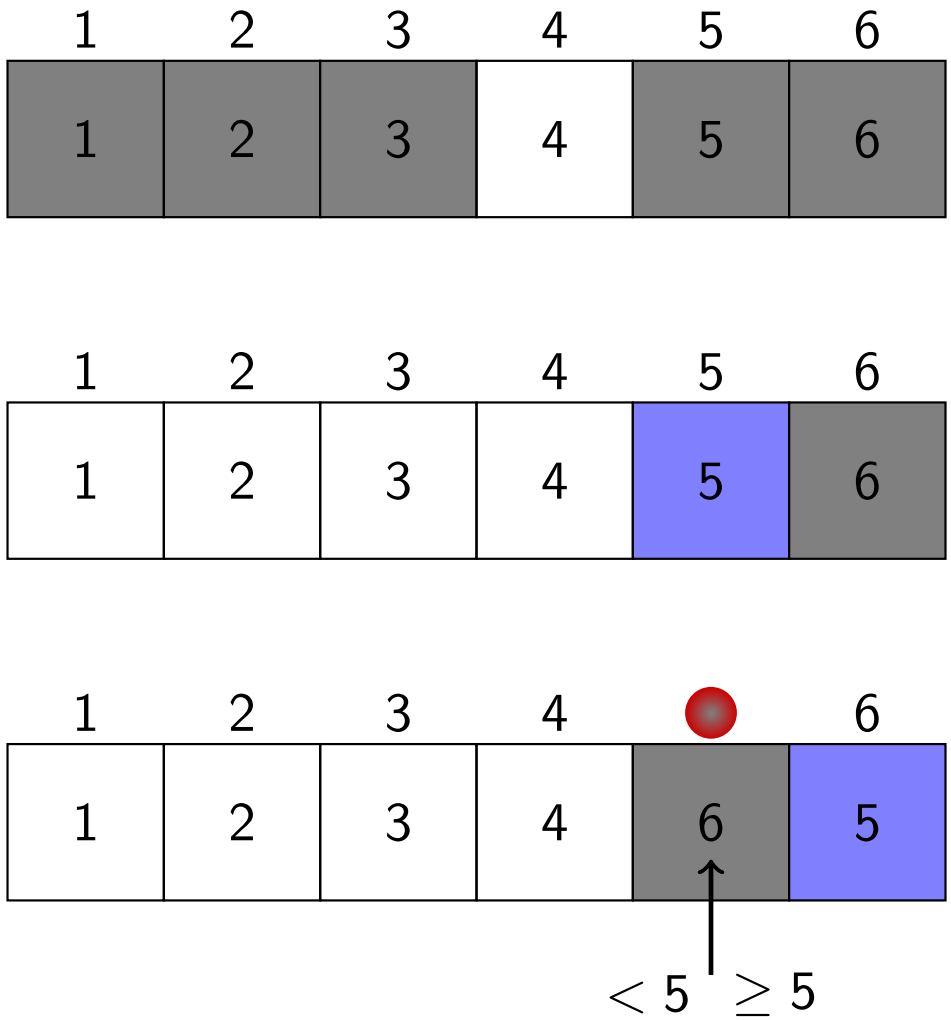
Execução recursiva



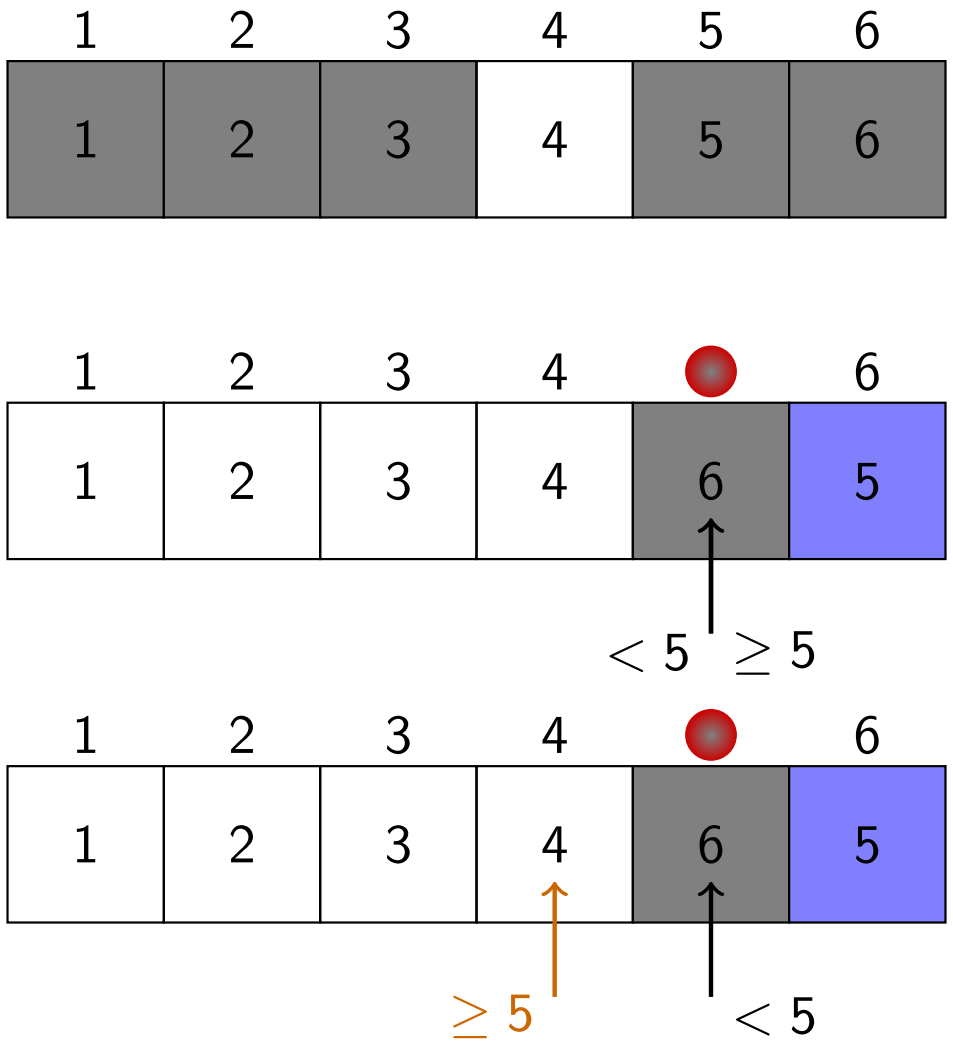
Execução recursiva



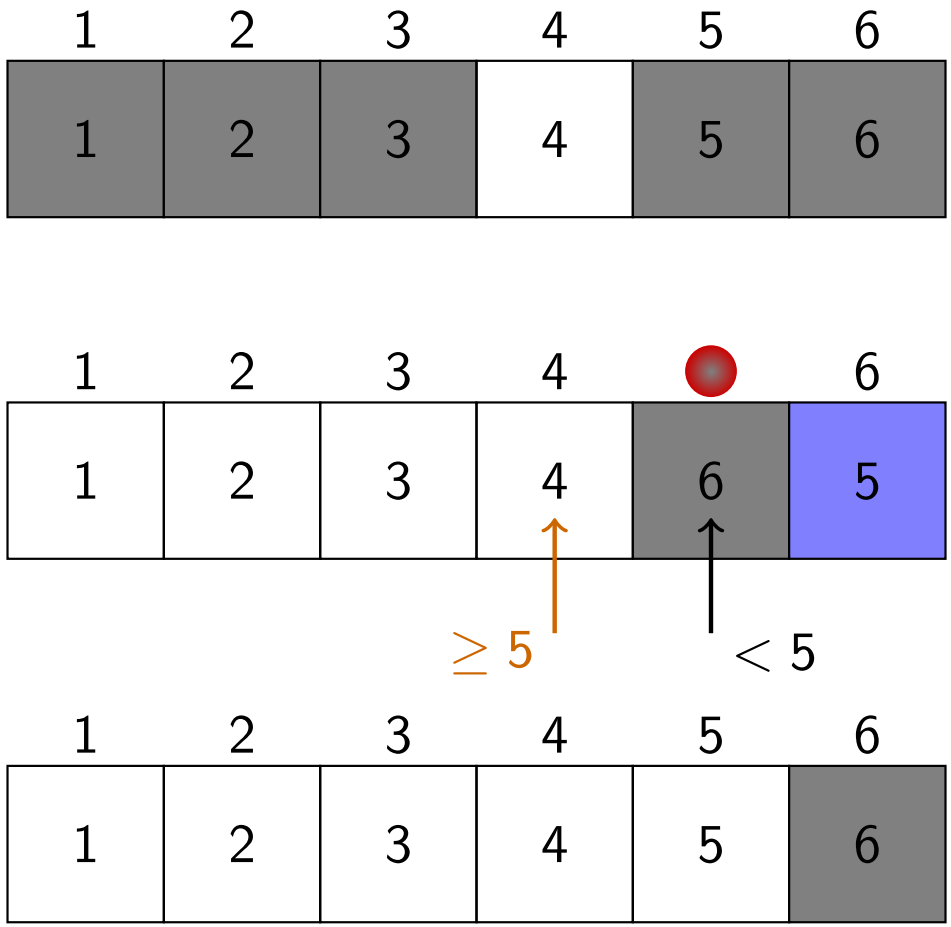
Execução recursiva



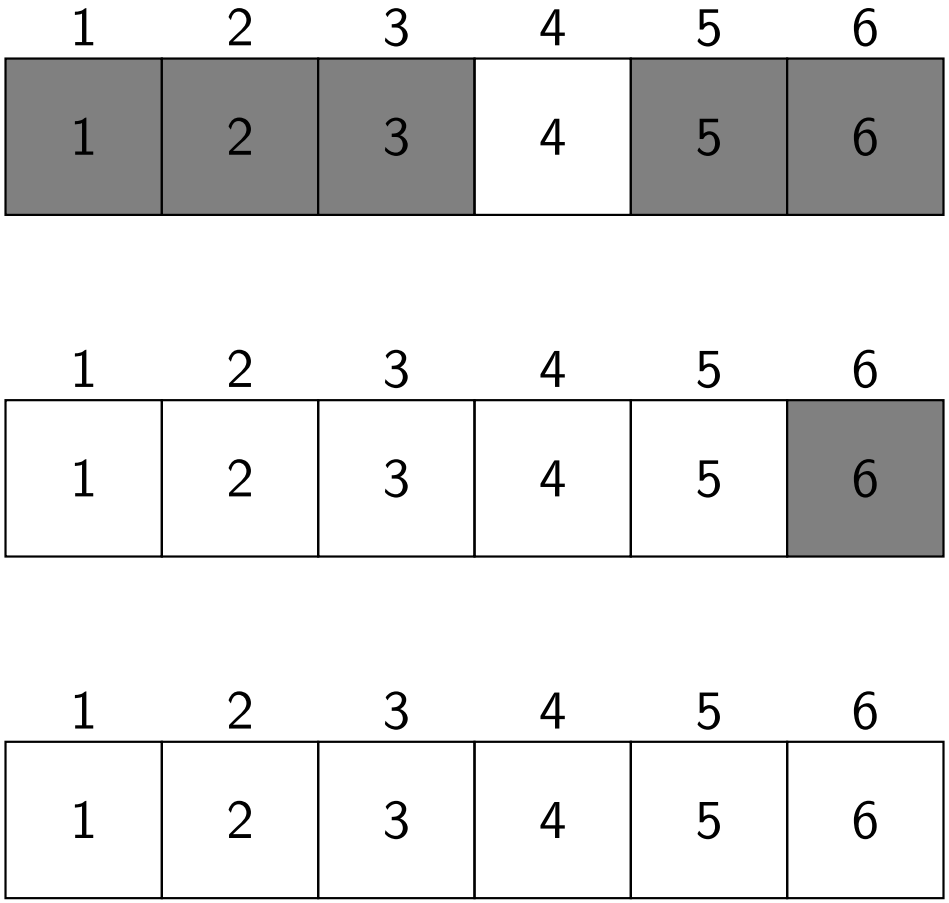
Execução recursiva



Execução recursiva



Execução recursiva



Implementação: escolha do pivô

```
algoritmo "quicksort"
var a : vetor [1..8] de inteiro
    i : inteiro
    c : real

funcao calcula_pivo(first_idx, last_idx: inteiro) : inteiro
inicio
       retorne ((first_idx + last_idx) \ 2)
fimfuncao
```

- Poderiamos também pegar o último indício
- O objetivo é tentar **em média** dividir o vetor por 2 (ter a metade de menores, ter a outra de maiores)

Implementação: o procedimento qsort

```
procedimento qsort(first_idx, last_idx: inteiro)
var lt_idx, pividx: inteiro
    tmp : inteiro
    j : inteiro
inicio
        pividx <- calcula_pivo(first_idx, last_idx)
        se first_idx < last_idx entao
          tmp <- a[last_idx]
          a[last_idx] <- a[pividx]
          a[pividx] <- tmp
          lt_idx <- last_idx - 1
          j <- first_idx
          enquanto j <= lt_idx faca
               se a[j] > a[last_idx] entao
                  tmp <- a[j]
                  a[j] <- a[lt_idx]
                  a[lt_idx] <- tmp
                  lt_idx <- lt_idx - 1
               senao
                  j <- j + 1
               fimse
          fimenquanto
          tmp <- a[lt_idx + 1]
          a[lt_idx + 1] <- a[last_idx]
          a[last_idx] <- tmp
          qsort(first_idx, lt_idx)
          qsort(lt_idx + 2, last_idx)
        fimse
fimprocedimento
```

Corretude

Pré-condiçãob

- $\texttt{first\_idx} \leq \texttt{last\_idx}$

Pós-condições

- $\forall k, \texttt{first\_idx} \leq k \leq \text{pivô} \Rightarrow a[k] < a[\text{pivô}]$
- $\forall k, \text{pivô} \leq k \leq \texttt{last\_idx} \Rightarrow a[\text{pivô}] \leq a[k]$

Especificação

Ordenação por fusão (1945)

- O algoritmo de ordenação por fusão (*mergesort*) adota uma estrategia tipo **divisão e conquista**
- Ordenar separadamente as duas metades do vetor
- A partir de duas metades ordenadas, fusionar e obter um vetor ordenado

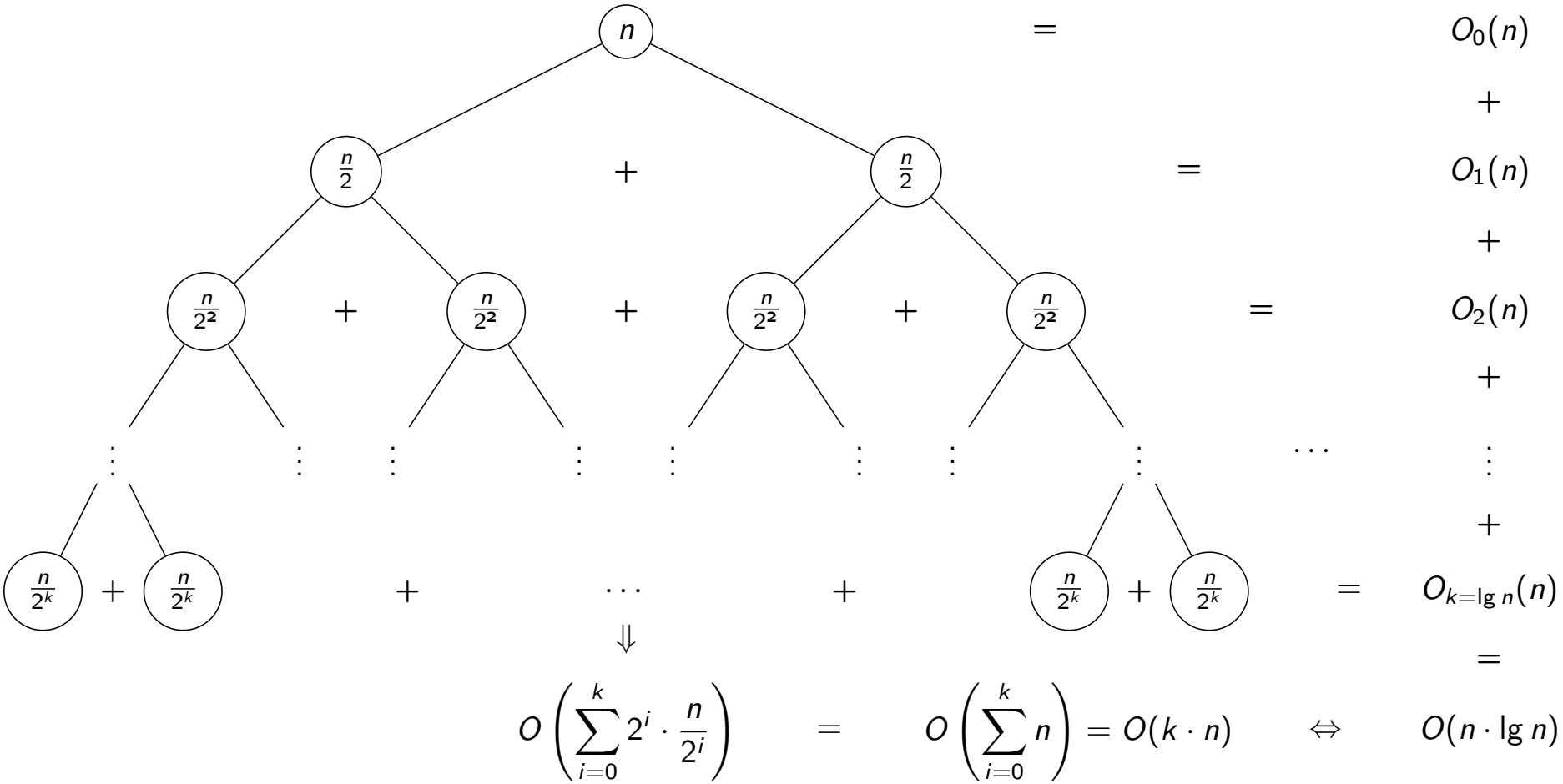
Passos

1. Dividir o vetor em 2
2. **Recursivamente**, ordenar os 2 subvetores.
3. Fusionar os 2 subvetores **ordenados**

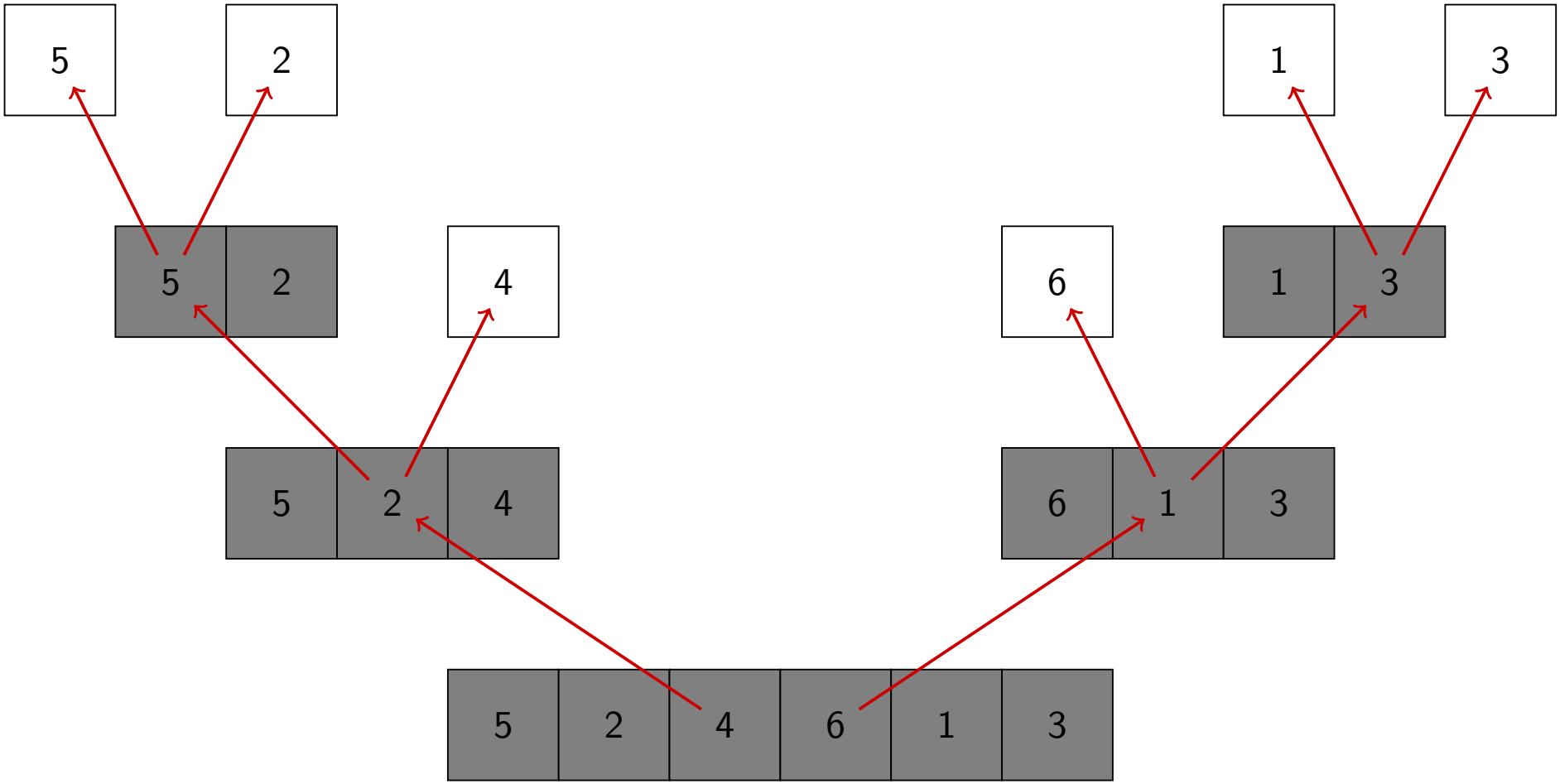
John von Neumann (1903–1957)

- Criador da arquitura lógica dos computadores modernos
- Um dos maiores cientistas do século XX.

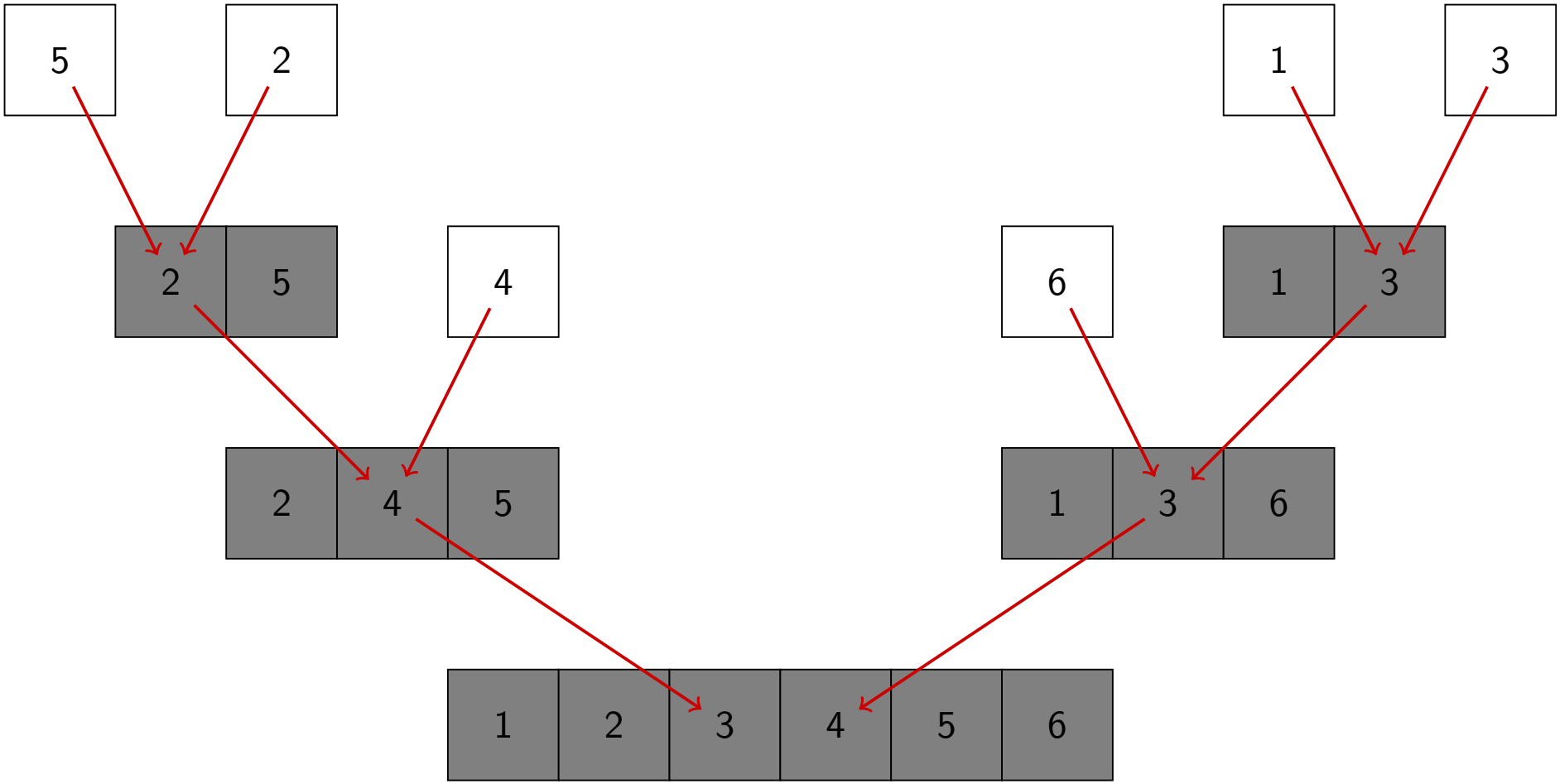
Visualização



Execução: divisão



Execução: fusão



Fusão

```
i <- 1 // v1[i]
j <- 1 // v2[j]
para k de 1 ate m + n faca
  se (i <= n) e (j <= m) entao
    se v1[i] <= v2[j] entao
       v3[k] <- v1[i]
       i <- i + 1
    senao
       v3[k] <- v2[j]
       j <- j + 1
    fimse
  senao // i > n ou j > m
    se i > n entao
      v3[k] <- v2[j]
      j <- j + 1
    senao // j > m
      v3[k] <- v1[i]
      i <- i + 1
    fimse
   fimse
```

Implementação : dados iniciais

```
algoritmo "mergesort"
var a, t : vetor [1..8] de inteiro
    i : inteiro

funcao calcula_media(first_idx, last_idx: inteiro) : inteiro
inicio
       retorne ((first_idx + last_idx + 1) \ 2)
fimfuncao

procedimento copytotemp(first_idx, last_idx: inteiro)
inicio
        para i de first_idx ate last_idx faca
             t[i] <- a[i]
        fimpara
fimprocedimento
```

Implementação : função de ordenação

```
procedimento msort (first_idx, last_idx: inteiro)
var j, k: inteiro
    m: inteiro
    cond: logico
inicio
  se first_idx < last_idx entao
     m <- calcula_media(first_idx, last_idx)
     // sort the two subpart of the array
     msort(m, last_idx)
     msort(first_idx, m - 1)
     // save in temporary array
     copytotemp(first_idx, last_idx)
     // merge the two copies of ordered sub-arrays
     // into the original array
     // counter to the first array
     j <- first_idx
    // counter for the second array
     k <- m
     para i de first_idx ate last_idx faca
          se j >= m entao
             a[i] <- t[k]
             k <- k + 1
          senao
               se k > last_idx entao
               a[i] <- t[j]
                j <- j + 1
               senao
                    se t[k] < t[j] entao
                    a[i] <- t[k]
                    k <- k + 1
                    senao
                         a[i] <- t[j]
                         j <- j + 1
                    fimse
               fimse
          fimse
     fimpara
  fimse
fimprocedimento
```

Corretude

Pré-condição

- $\texttt{first\_idx} \leq \texttt{last\_idx}$

Pós-condições

- $\forall k, \texttt{first\_idx} \leq k < \texttt{last\_idx} \Rightarrow a[k] \leq a[k+1]$

Resumo

1. Torre de Hanoi
2. Quicksort
3. Ordenação por fusão
4. Exercícios

Perguntas ?

http://dimap.ufrn.br/~richard/dim0320

Leitura/interpretação

Assunto

Dada a implementação:

```
funcao f(n: inteiro): inteiro
inicio
        se n < 4 entao retorne (3 * n)
        senao retorne (2 * f(n - 4) + 5)
        fimse
fimfuncao
```

- Escreva as árvores de chamada de função para `f(3)` e `f(7)`.
- Quais são os valores obtidos?

Elemento máximo de um vetor

Assunto

Implemente uma função **recursiva** `max` que retorne o maior valor de um vetor de n números inteiros.

Combinações

Assunto

Considere a função $comb(n, k)$, que representa o número de grupos distintos com k pessoas que podem ser formados a partir de n pessoas. Por exemplo, $comb(4, 3) = 4$, pois com 4 pessoas (A, B, C, D), é possível formar 4 diferentes grupos: ABC, ABD, ACD e BCD. Sabe-se que:

$$comb(n, k) = \begin{cases} n & k = 1 \\ 1 & k = n \\ comb(n-1, k-1) + comb(n-1, k) & 1 < k < n \end{cases}$$

- Implemente em portugol uma função recursiva para comb (n, k) e mostre o diagrama de execução para chamada $comb(5, 3)$.
- Sabendo-se ainda $comb(n, k) = n!/(k!(n-k)!)$, implemente uma função não recursiva de $comb(n, k)$.