

The background is dark teal with white geometric lines forming a circuit-like pattern. There are several yellow circles, teal circles, a teal triangle, a teal 'x', and a cluster of small teal and blue squares. A solid pink circle is on the left.

# Algoritmos e Estruturas de Dados II

**Lucila Bento**  
lucila.bento [at] ime.uerj.br

# Divisão e Conquista

Precisamos resolver um programa com uma entrada grande

DIVISÃO

Para facilitar a resolução do problema, quebramos a entrada em pedaços menores

CONQUISTA

Cada pedaço da entrada é então tratado separadamente

Ao final, os resultados parciais são combinados para gerar o resultado final procurado

# Divisão e Conquista - Estratégia

A instância dada do problema é dividida em duas ou mais instâncias menores.



Cada instância menor é resolvida usando o próprio algoritmo que está sendo definido.



As soluções das instâncias menores são combinadas para produzir uma solução da instância original.

# Divisão e Conquista

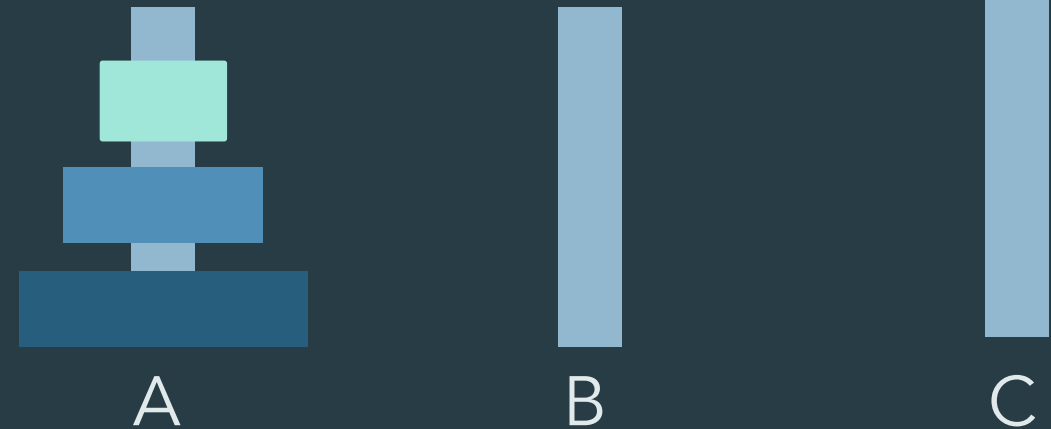
O problema da Torre de Hanói



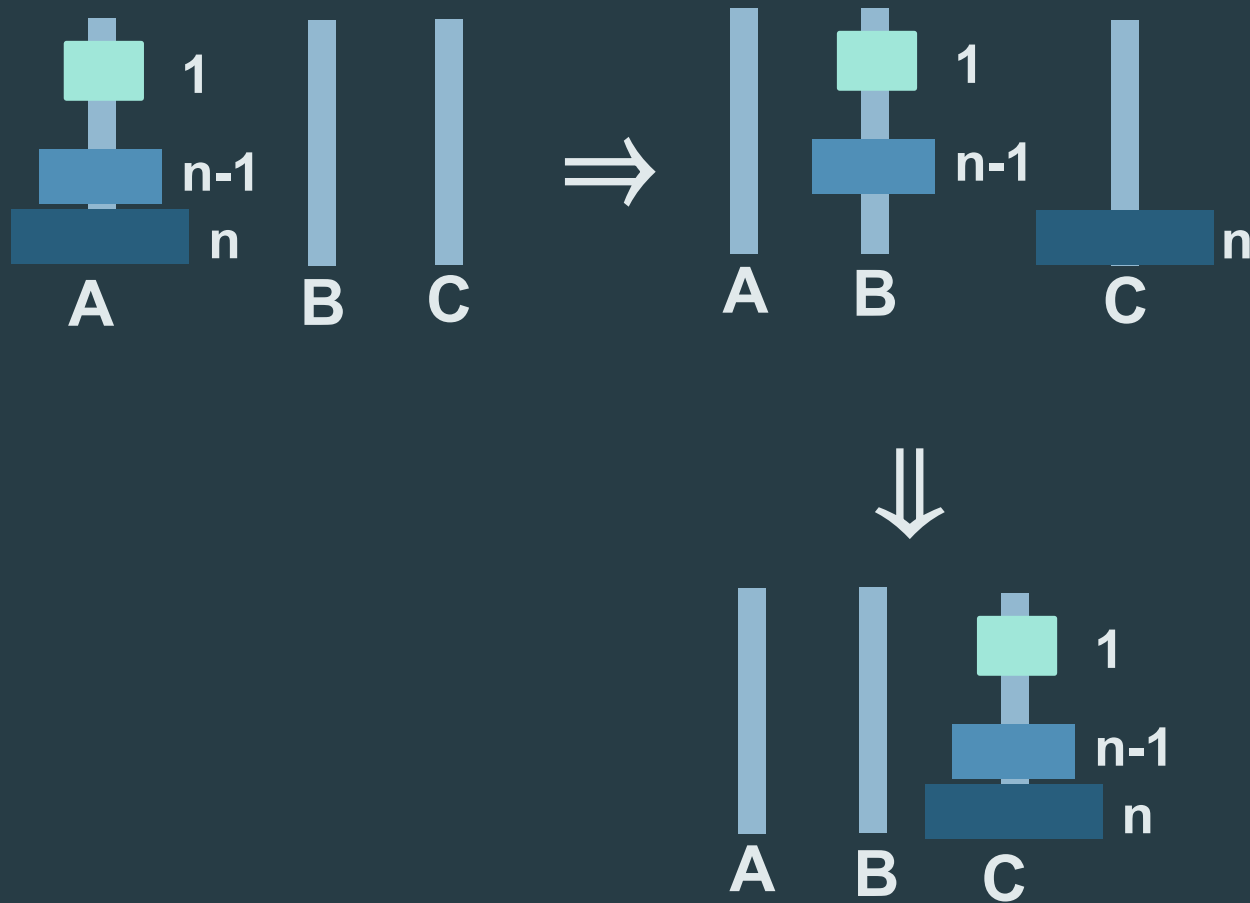
# Torre de Hanoi

## Problema:

Levar os  $n$  discos do pino A para o pino C, usando B como auxílio, sem nunca colocar um disco sobre um disco menor e movendo um único disco por vez.



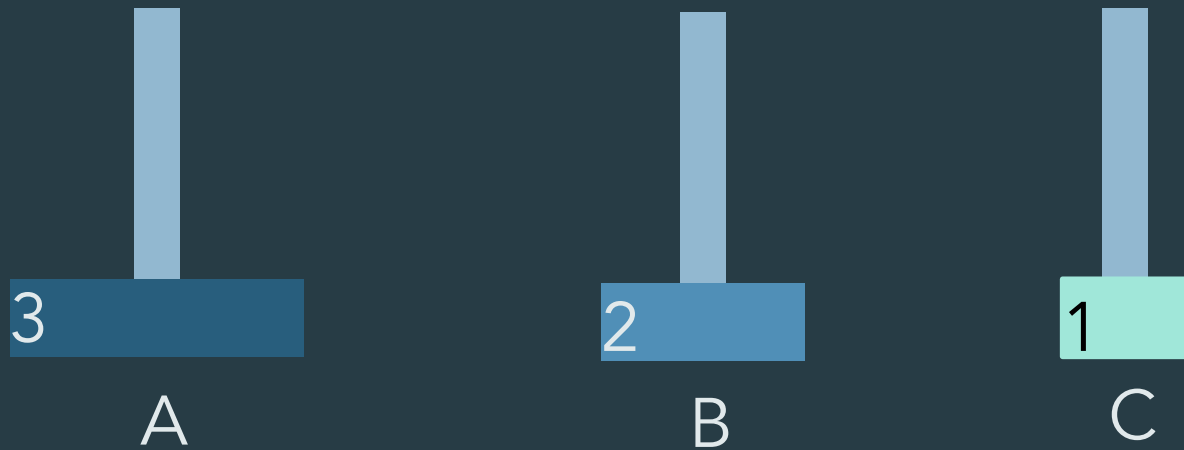
# Torre de Hanoi com Divisão e Conquista



Quando o problema for grande:

- a) levar  $n-1$  discos do pino A para a B,
- b) mover o último disco do pino A para a C e
- c) mover os  $n-1$  discos do pino B para a C.

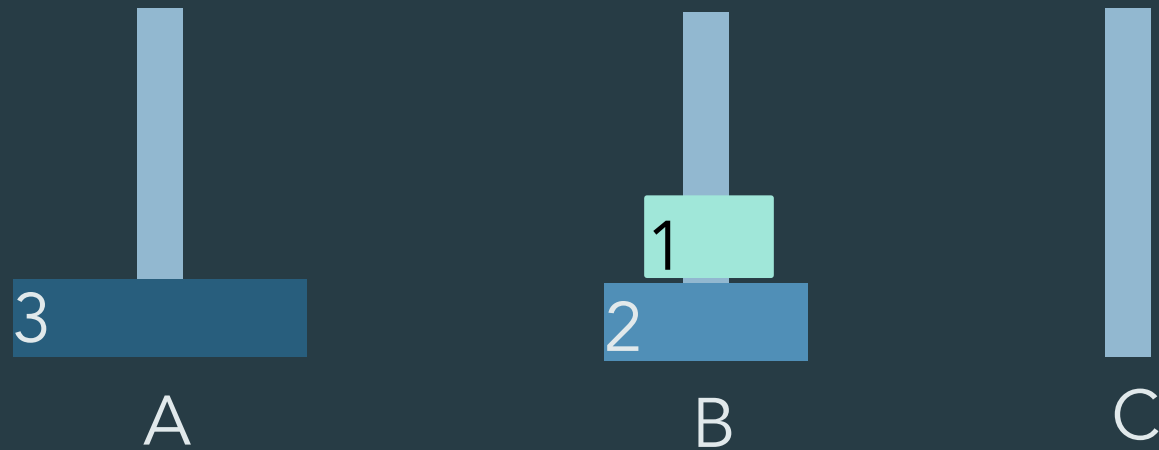
# Torre de Hanoi - Exemplo



Mover 1 de A para C

Mover 2 de A para B

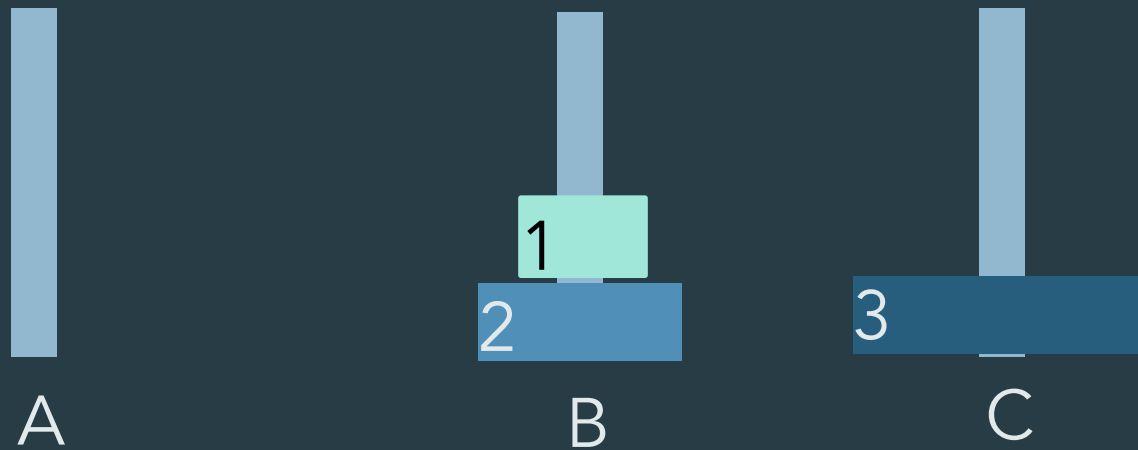
# Torre de Hanoi - Exemplo



Mover 1 de C para B

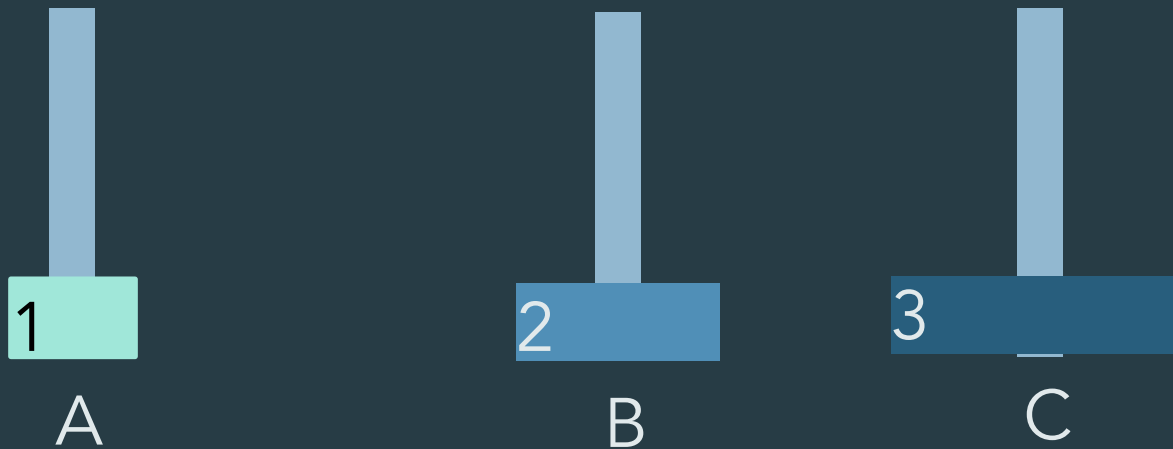


# Torre de Hanoi - Exemplo



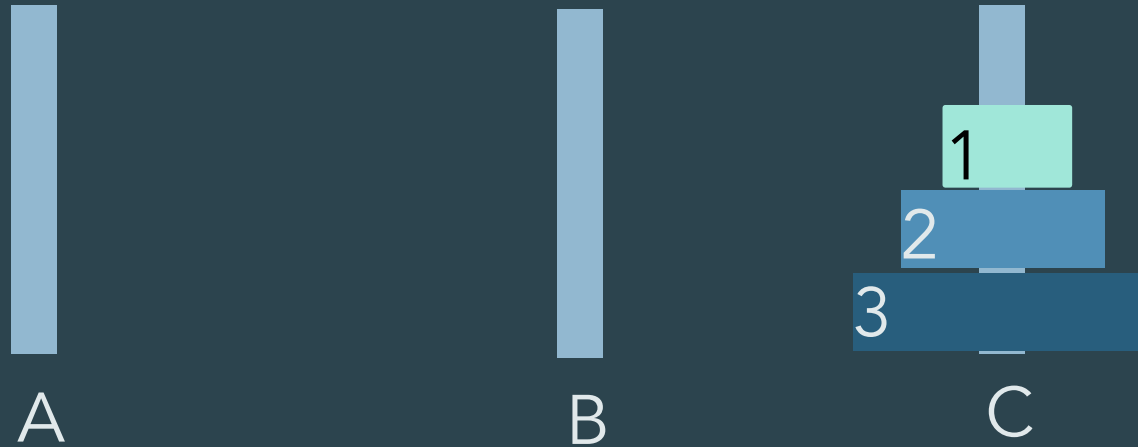
Mover 3 de A para C

# Torre de Hanoi - Exemplo



Mover 1 de B para A

# Torre de Hanoi - Exemplo

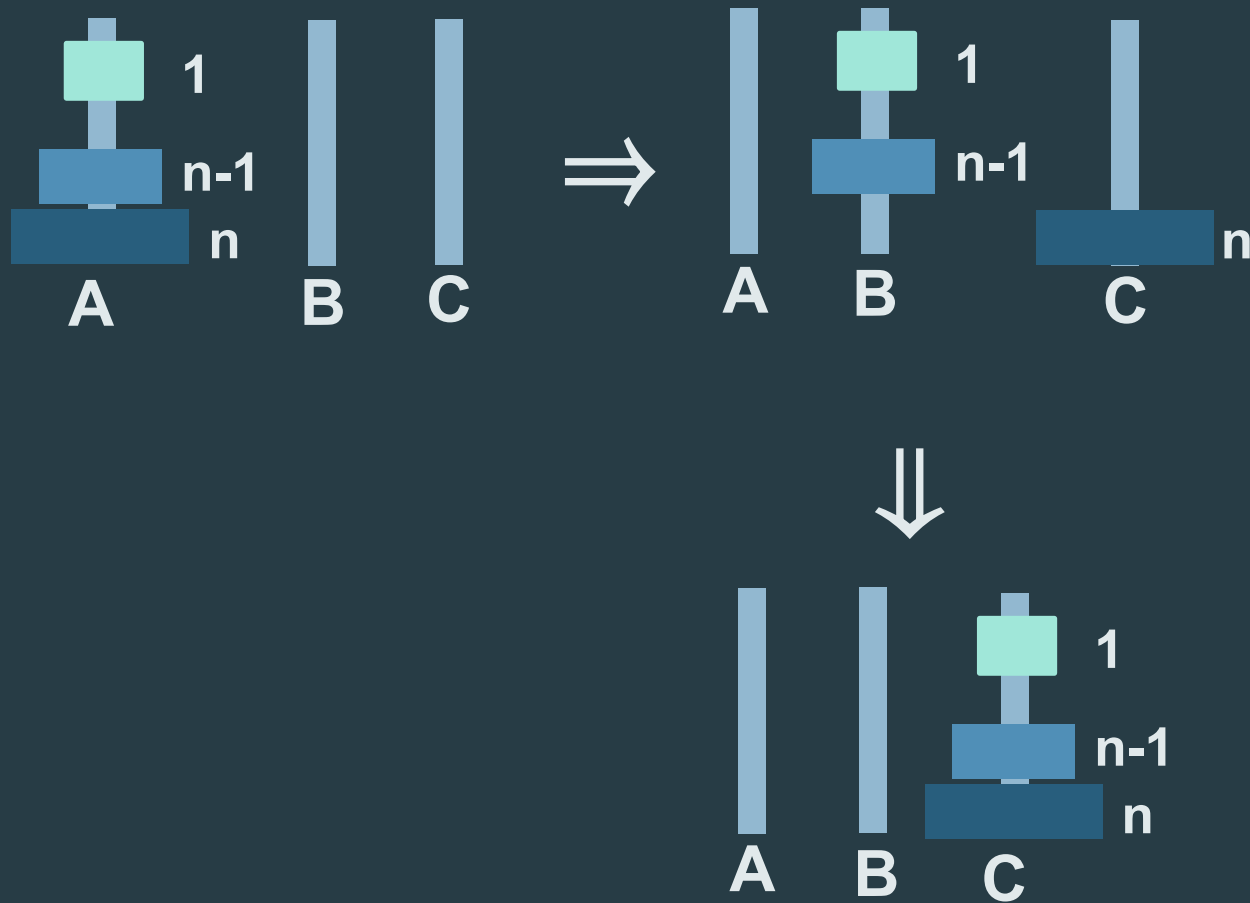


Mover 2 de B para C

Mover 1 de A para C

Problema resolvido com 7  
movimentos!

# Torre de Hanoi com Divisão e Conquista

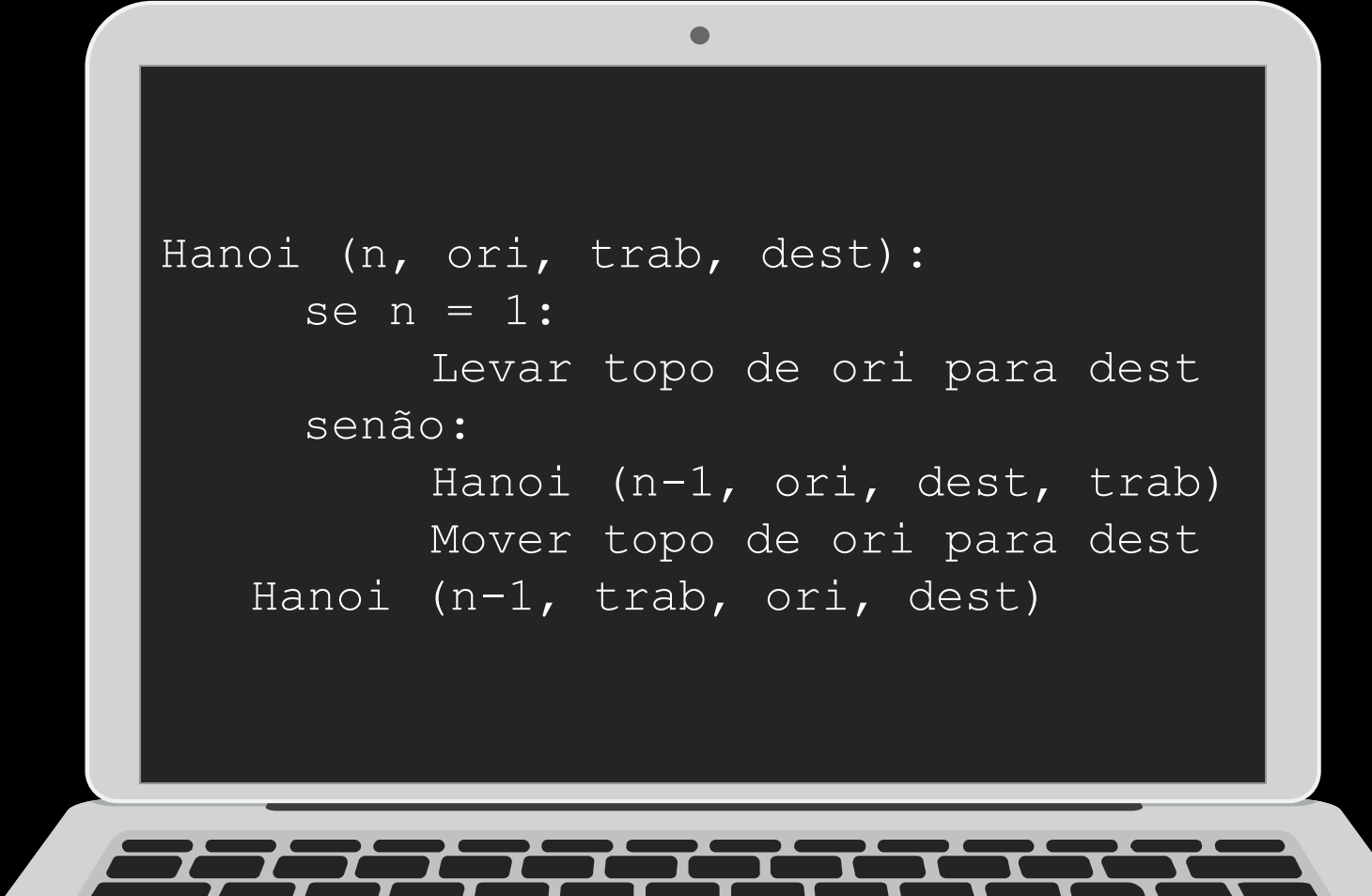


Quando o problema for grande:

- a) levar  $n-1$  discos do pino A para a B,
- b) mover o último disco do pino A para a C e
- c) mover os  $n-1$  discos do pino B para a C.

# TORRE DE HANOI - Formulação I

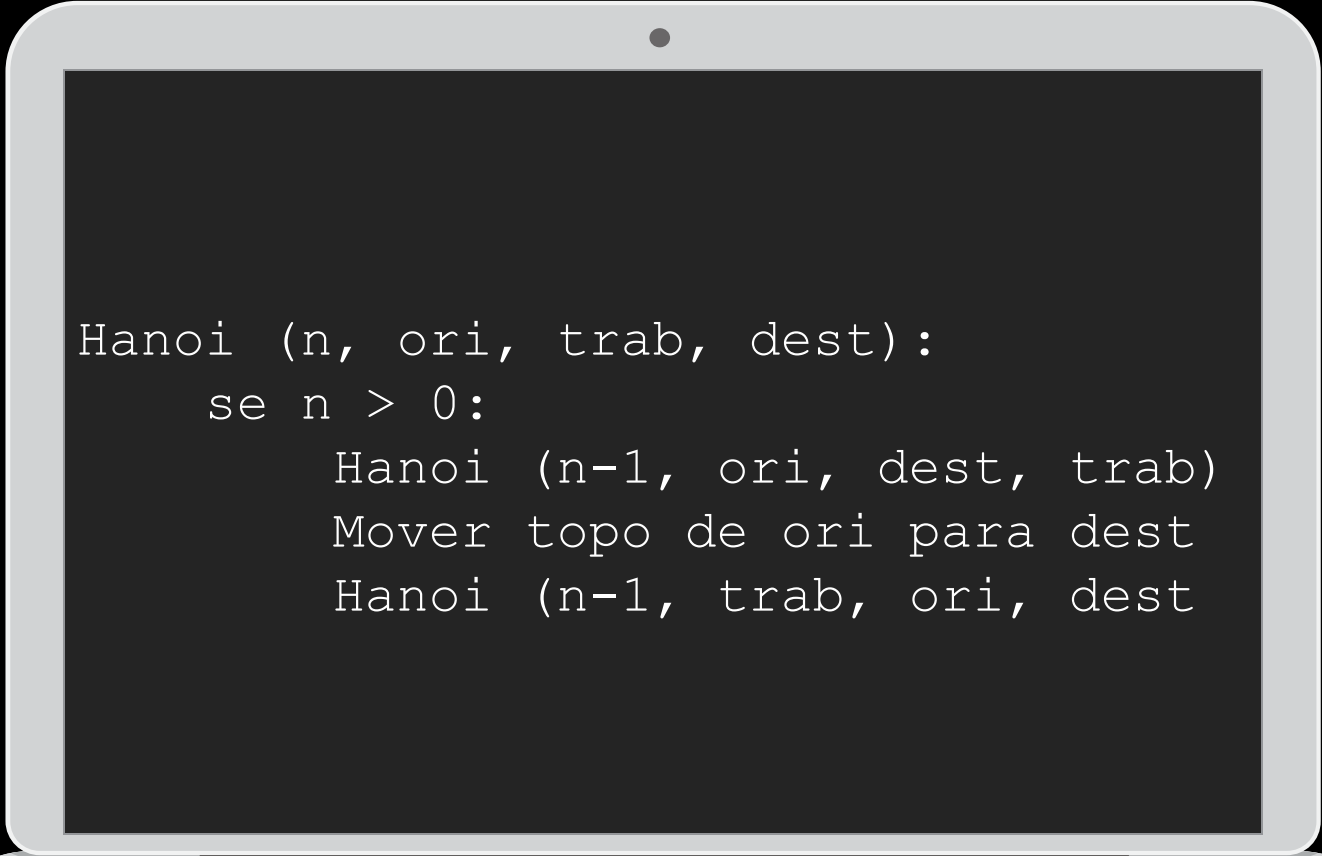
O problema pequeno é quando  $n = 1$ . Neste caso basta levar o disco de A para C.



```
Hanoi (n, ori, trab, dest):  
    se n = 1:  
        Levar topo de ori para dest  
    senão:  
        Hanoi (n-1, ori, dest, trab)  
        Mover topo de ori para dest  
        Hanoi (n-1, trab, ori, dest)
```

# TORRE DE HANOI - Formulação 2

O problema  
pequeno é  
quando  $n = 0$ .  
Neste caso nada  
precisa ser  
feito.



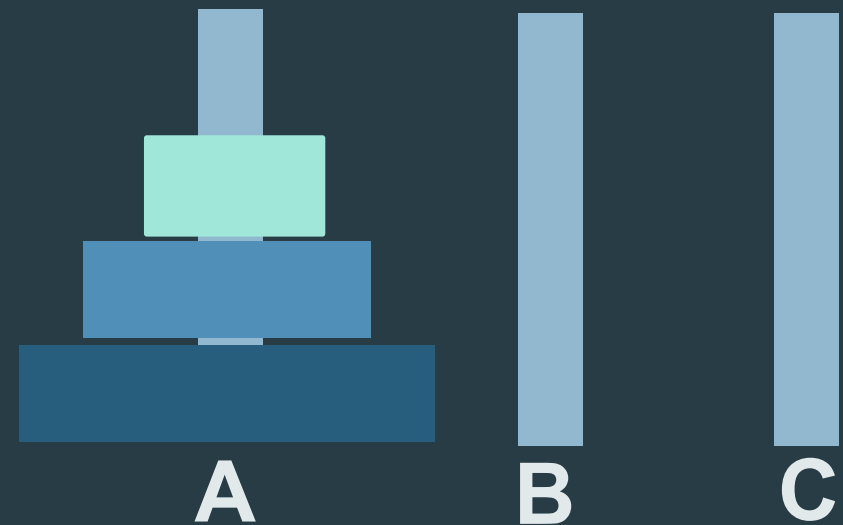
```
Hanoi (n, ori, trab, dest):  
    se n > 0:  
        Hanoi (n-1, ori, dest, trab)  
        Mover topo de ori para dest  
        Hanoi (n-1, trab, ori, dest
```

# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

- (1) se  $n > 0$ :
- (2)     Hanoi (n-1, ori, dest, trab)
- (3)     Mover topo de ori para dest
- (4)     Hanoi (n-1, trab, ori, dest);

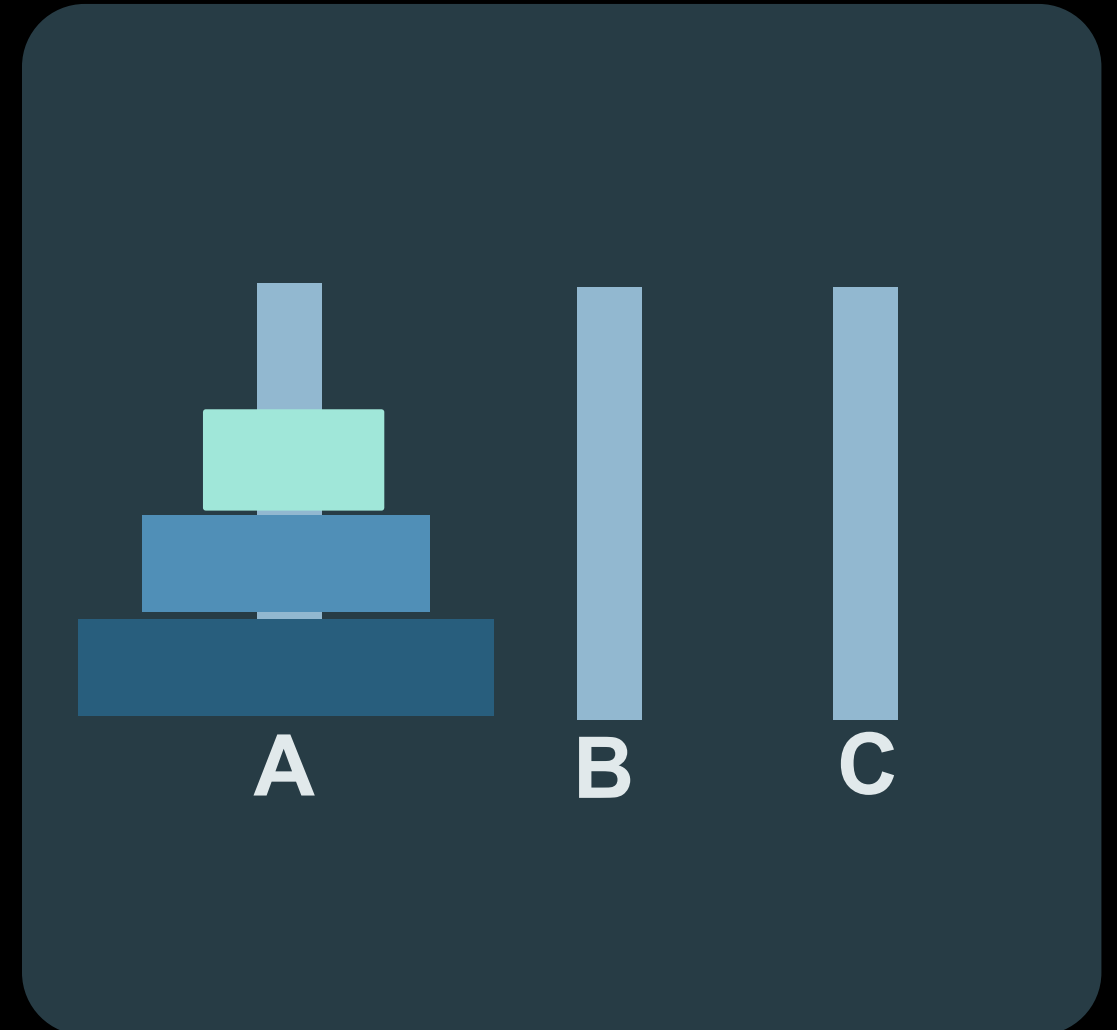
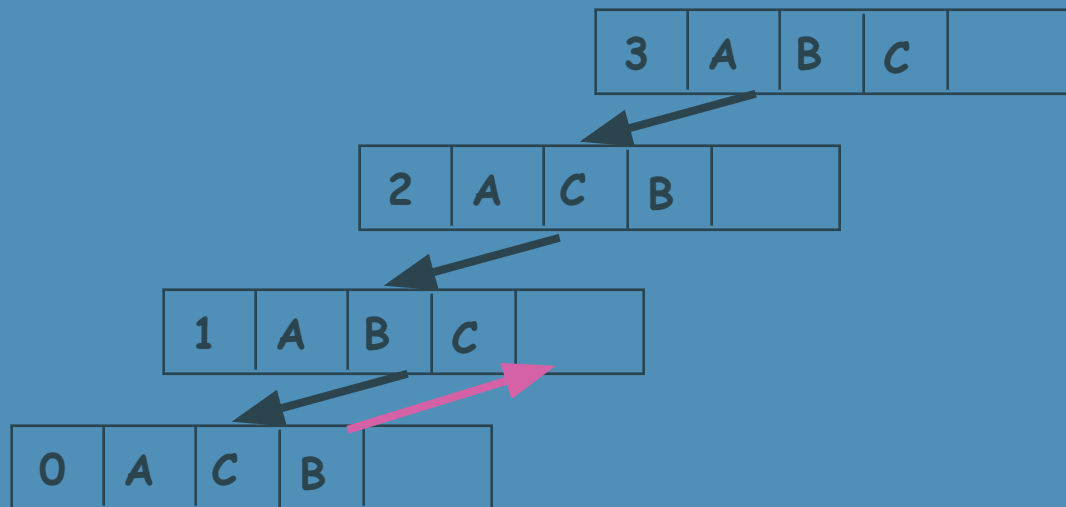
Hanoi (3, A, B, C)



# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

- (1) se  $n > 0$ :
- (2)     Hanoi (n-1, ori, dest, trab)
- (3)     Mover topo de ori para dest
- (4)     Hanoi (n-1, trab, ori, dest);

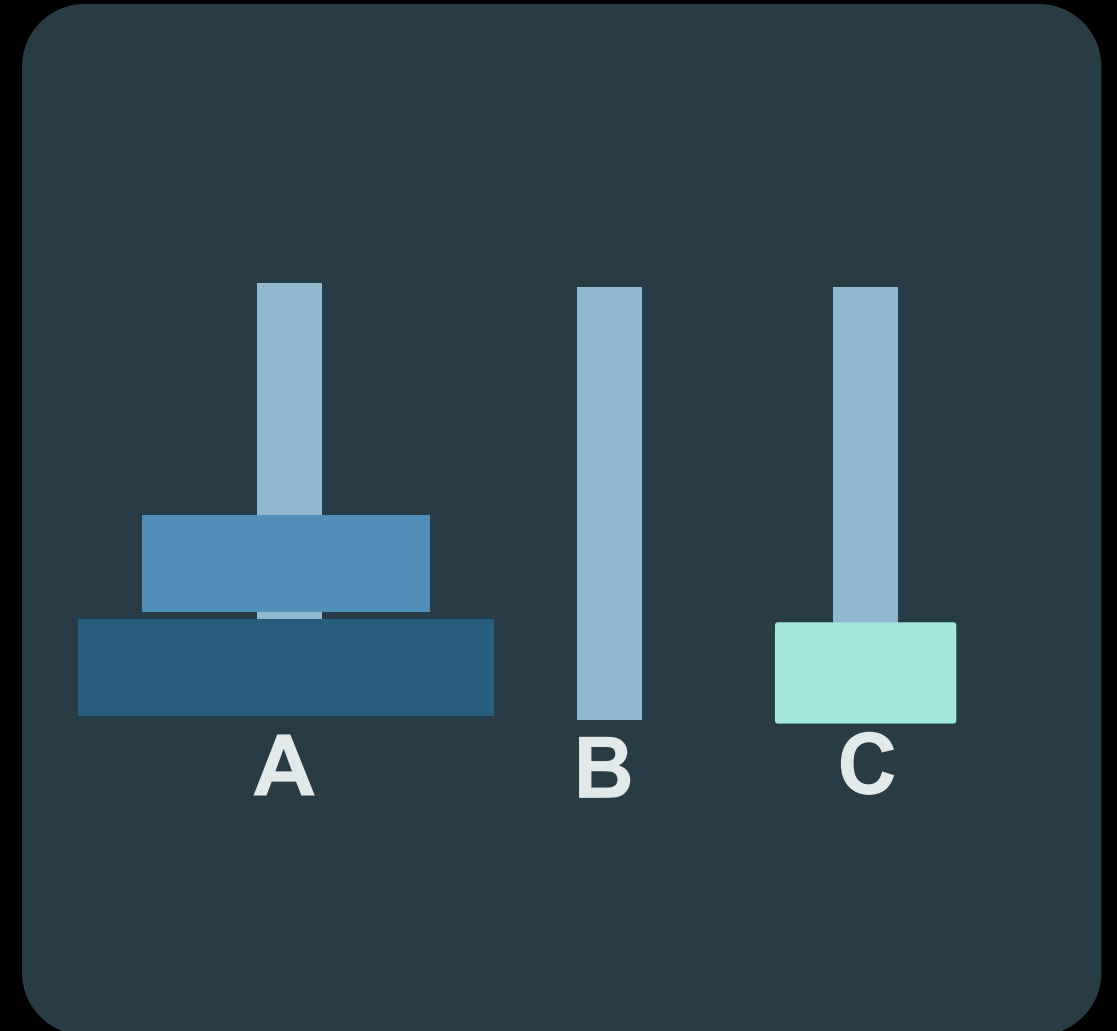
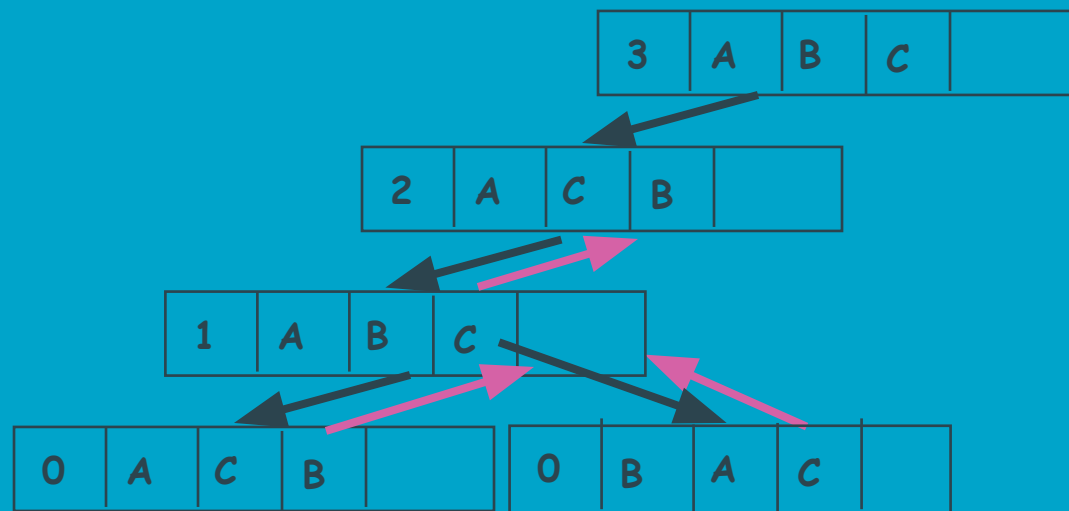




# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

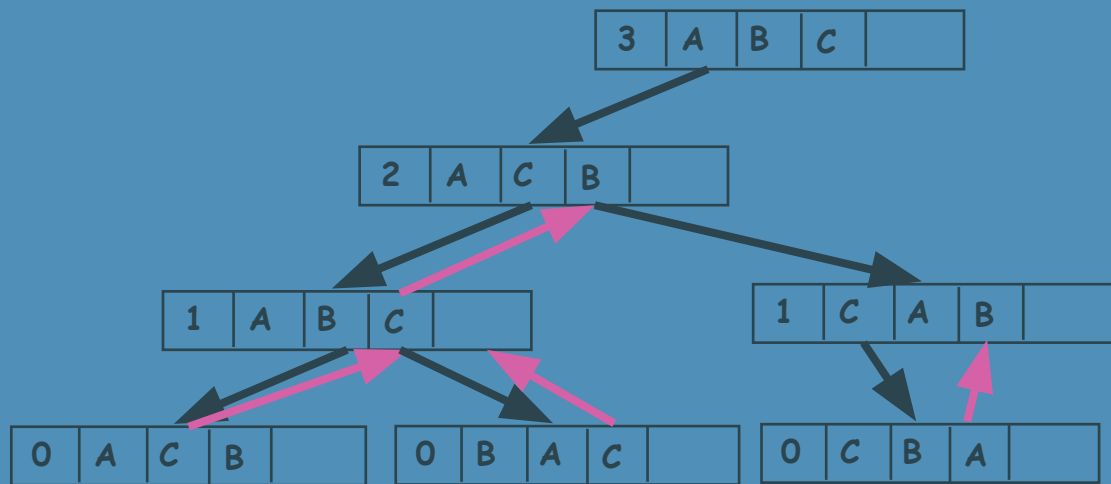
- (1) se  $n > 0$ :
- (2)     Hanoi (n-1, ori, dest, trab)
- (3)     Mover topo de ori para dest
- (4)     Hanoi (n-1, trab, ori, dest);



# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

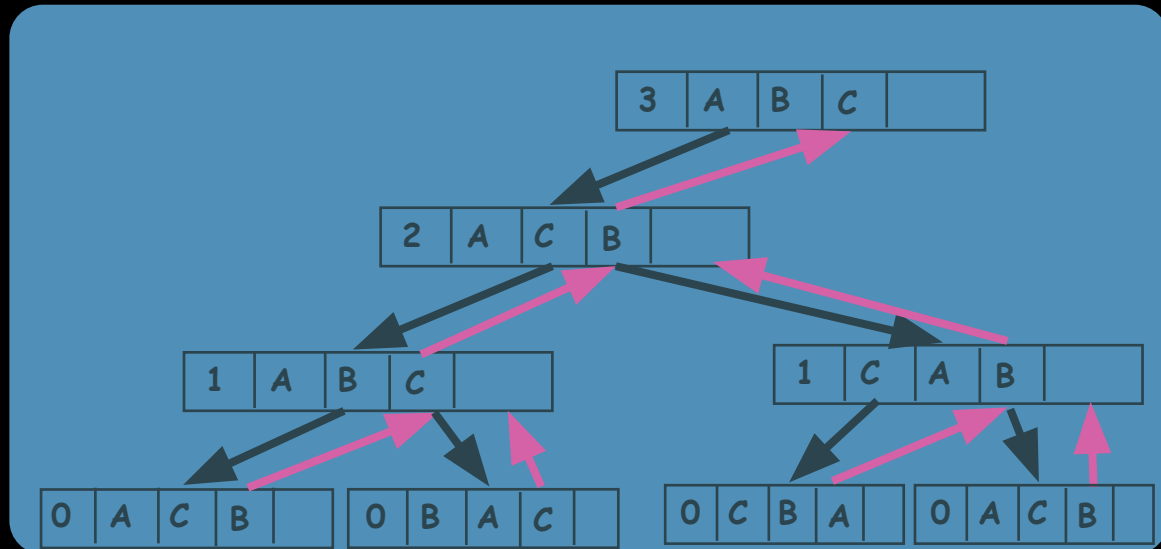
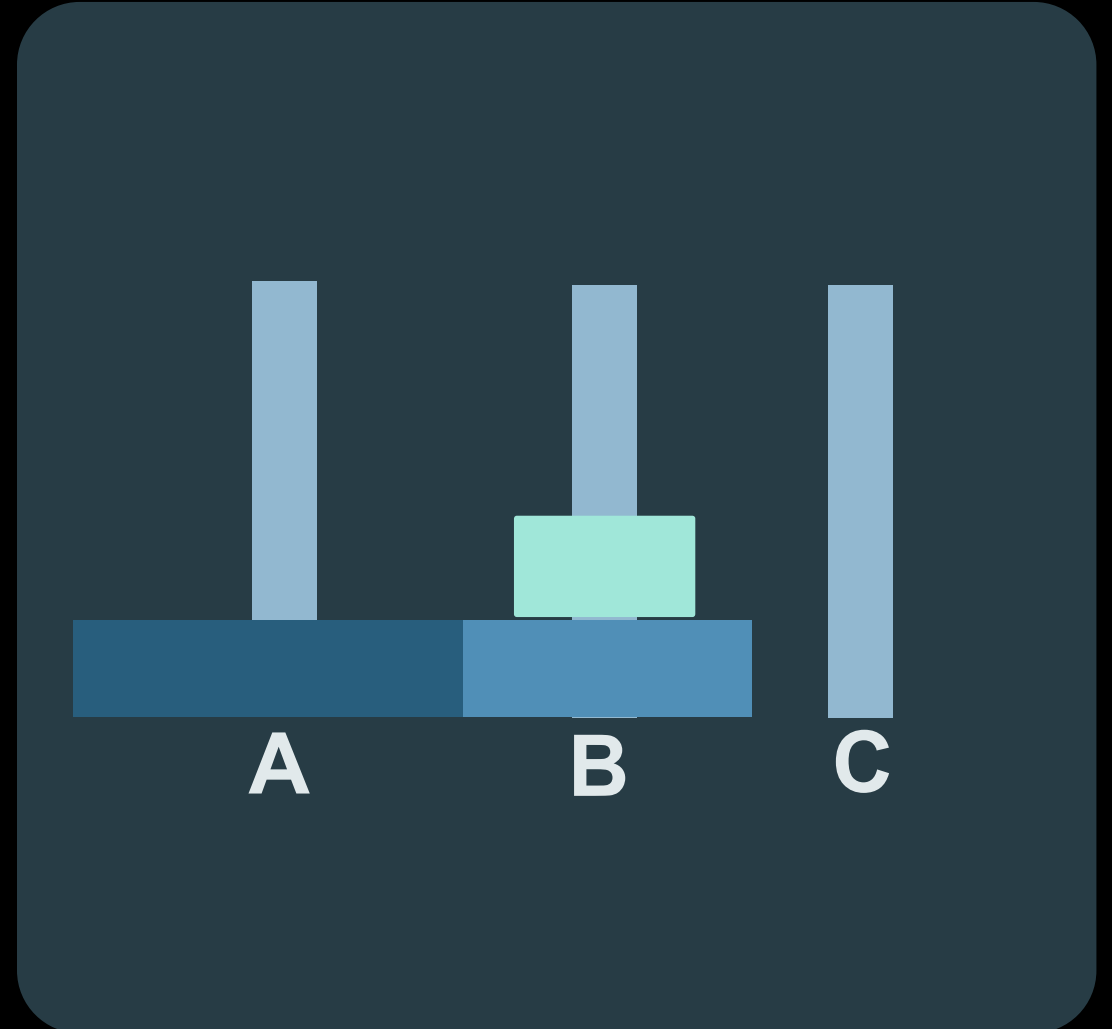
- (1) se  $n > 0$ :
- (2)     Hanoi (n-1, ori, dest, trab)
- (3)     Mover topo de ori para dest
- (4)     Hanoi (n-1, trab, ori, dest);



# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

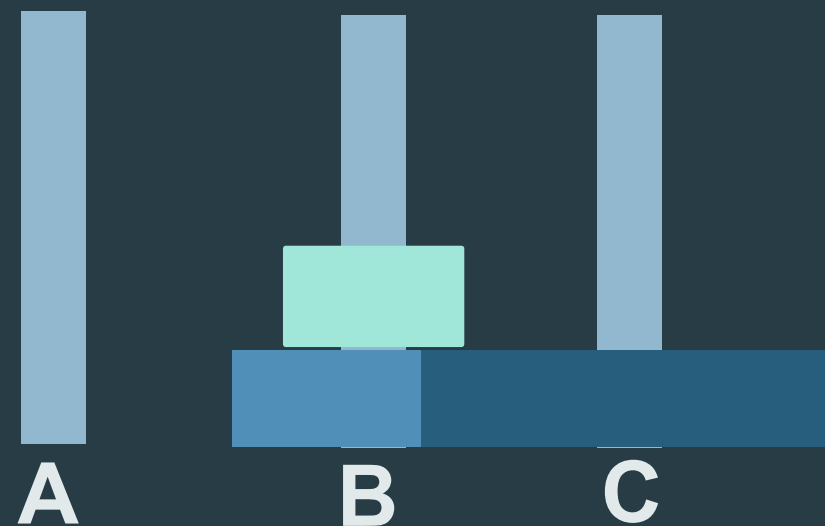
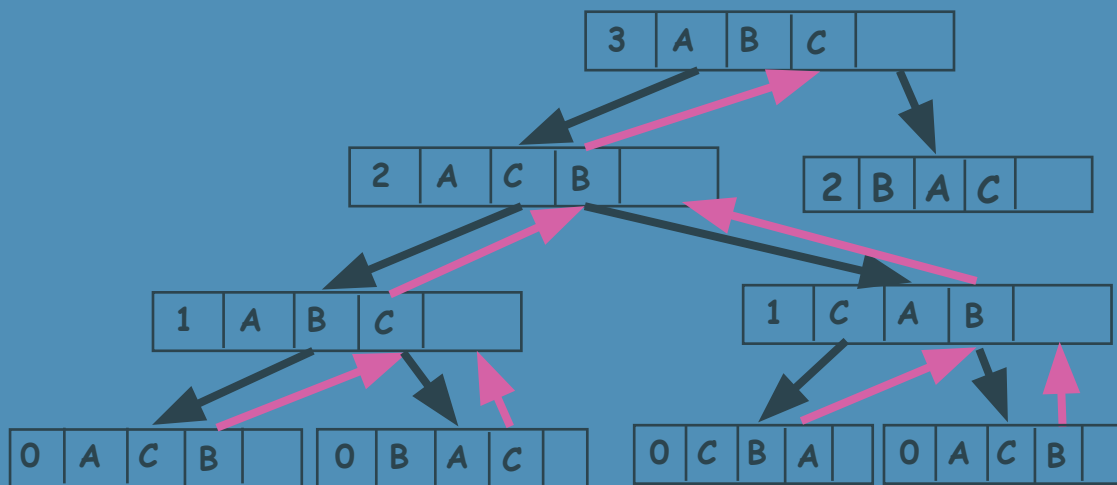
- (1) se  $n > 0$ :
- (2)     Hanoi (n-1, ori, dest, trab)
- (3)     Mover topo de ori para dest
- (4)     Hanoi (n-1, trab, ori, dest);



# TORRE DE HANOI - Árvore de Recursão

Hanoi (n, ori, trab, dest):

- ```
(1) se  $n > 0$ :
(2)   Hanoi ( $n-1$ , ori, dest, trab)
(3)   Mover topo de ori para dest
(4)   Hanoi ( $n-1$ , trab, ori, dest);
```



# TORRE DE HANOI - Recorrência

$T(n)$  = número de movimentos para mover  $n$  discos

$$T(n) = 2 * T(n-1) + 1$$

$$T(0) = 0 \quad \text{ou} \quad T(1) = 1$$

Solução da recorrência:

$$T(n) = 2^n - 1$$

Exercício  
recomendado:

Provar, por indução,  
que a fórmula ao  
lado é verdadeira.

# TORRE DE HANOI - Recorrência

$T(n)$  = número de movimentos para mover  $n$  discos

$$T(n) = 2^n - 1$$

Então o algoritmo é ineficiente?

Resposta:

Não. Prova-se que qualquer solução exige um número exponencial de passos. Então o problema é que é "ruim", não o algoritmo.

$$2^{64} - 1 = 18.446.744.073.709.551.615$$

# Divisão e Conquista

Exponenciação Rápida  
(Eficiência)



# Exponenciação com expoente inteiro

Problema: Dados dois inteiros  $a$  e  $n$ , calcular  $a^n$ .

O problema pode ser resolvido com uma abordagem ingênua  $O(n)$ , executando  $n$  multiplicações:

```
int Exp(int a, int n):  
    p ← 1  
    para i ← 1..n incl.:  
        p ← p*a  
    retornar p
```

Podemos fazer melhor?

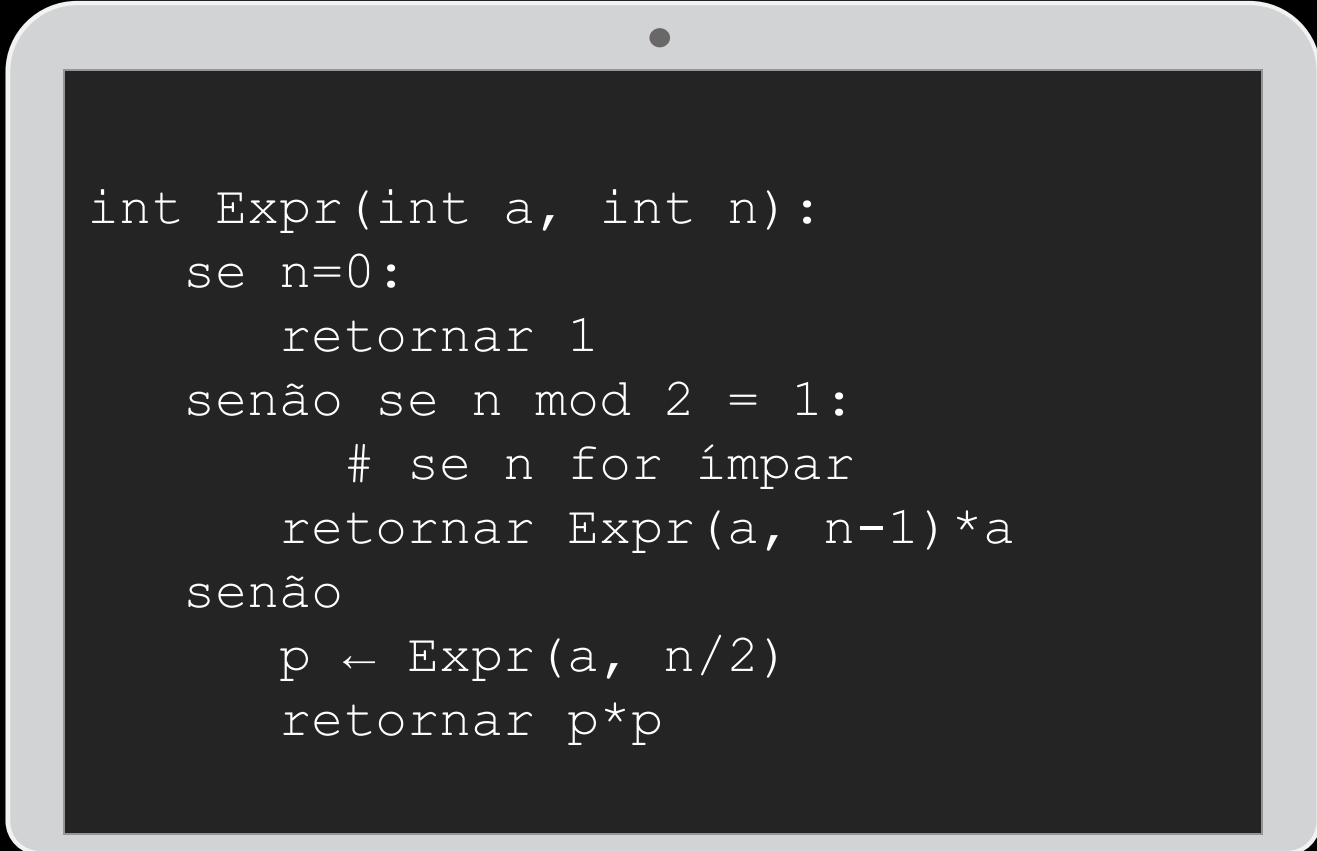


# Exponenciação com expoente inteiro

Algoritmo  $O(\log_2 n)$ :

**Idéia:** se tivermos calculado  $p = a^{n/2}$ , com mais uma multiplicação conseguimos calcular  $p' = a^n$ .

Basta fazer  
 $p' = p * p$ .



```
int Expr(int a, int n):  
    se n=0:  
        retornar 1  
    senão se n mod 2 = 1:  
        # se n for ímpar  
        retornar Expr(a, n-1)*a  
    senão  
        p ← Expr(a, n/2)  
        retornar p*p
```

# Divisão e Conquista

Busca Binária





## Problema

Dado um subarray de inteiros diferentes ordenados em ordem crescente e um inteiro , retornar "Sim" se aparece em "Não", caso contrário.

# Busca binária

```
def busca_binaria(A, p, r, z):  
    if r >= p:  
        meio = (r + p) // 2  
        if A[meio] == z: # Se z está no meio  
            return meio  
        elif A[meio] > z:  
            # Se z < meio, z pode estar a esq  
            return busca_binaria(A, p, meio - 1, z)  
        else: #Caso contrário, z está à direita  
            return busca_binaria(A, meio + 1, r, z)  
  
    else: #Elemento não está em A  
        return -1
```

**Complexidade:**  
 $O(\log n)$

# Divisão e Conquista

MergeSort



# Recapitulando...

- **Divida o problema em vários subproblemas**
  - Subproblemas semelhantes de tamanho menor
- **Conquiste os subproblemas**
  - Resolva os subproblemas recursivamente
  - Tamanho do subproblema pequeno o suficiente  $\Rightarrow$  resolva os problemas de maneira direta
- **Combine as soluções dos subproblemas**
  - Obtenha a solução para o problema original

# Abordagem do MergeSort

Para ordenar um array  $A[p \dots r]$ :

- **Divisão**
  - Divide a sequência de  $n$  elementos a ser ordenada em duas subsequências de  $n/2$  elementos cada
- **Conquista**
  - Ordena as subsequências recursivamente usando o MergeSort
  - Quando o tamanho das sequências é 1 não há mais nada a fazer
- **Combinação**
  - Mesclar as duas subsequências ordenadas

# MergeSort

```
MergeSort(A, p, r)
```

```
  if p < r  —————→ Verifica o caso base
```

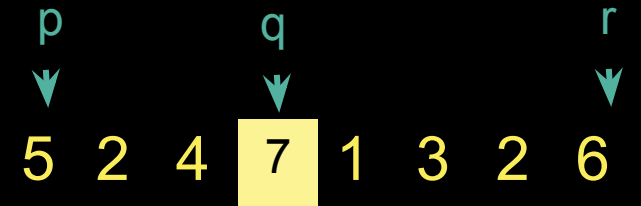
```
    then  $q \leftarrow \lfloor (p + r) / 2 \rfloor$  —————→ Divisão
```

```
      MERGE-SORT(A, p, q) —————→ Conquista
```

```
      MERGE-SORT(A, q + 1, r) —————→ Conquista
```

```
      MERGE(A, p, q, r) —————→ Combinação
```

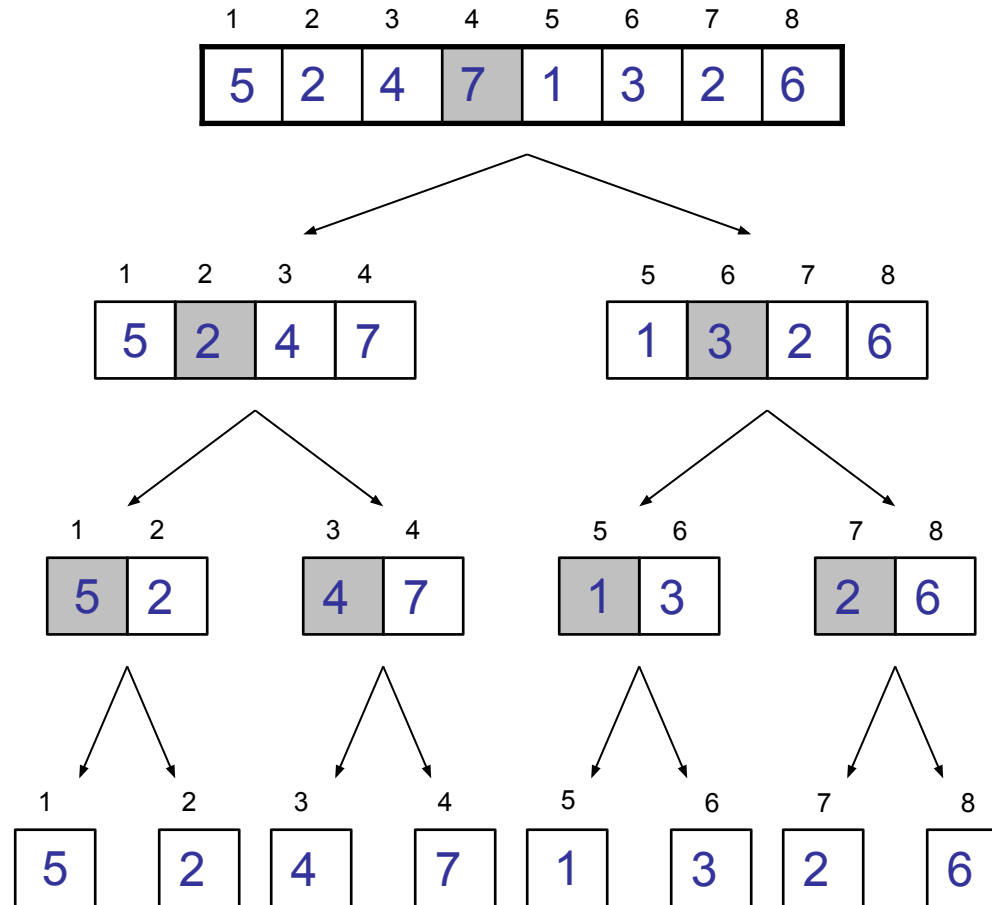
Chamada inicial: MERGE-SORT(A, 1, n)





# Exemplo: n Potência de 2

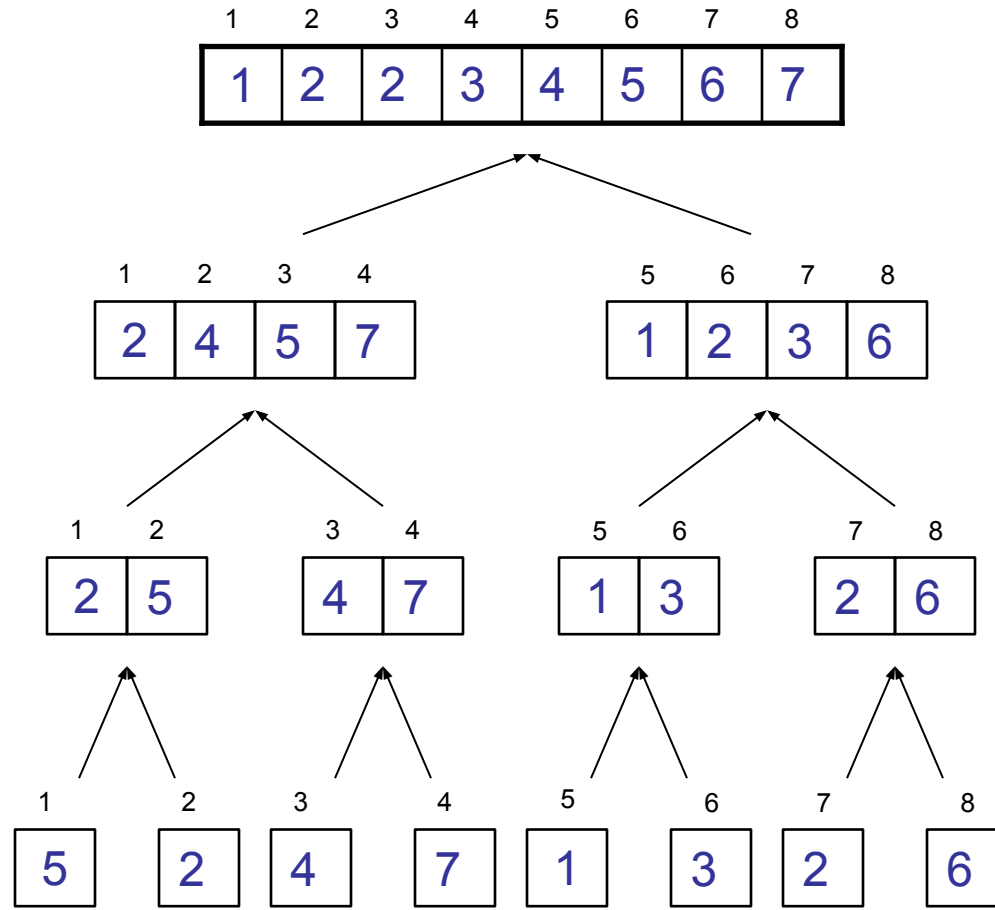
Divisão



$q = 4$

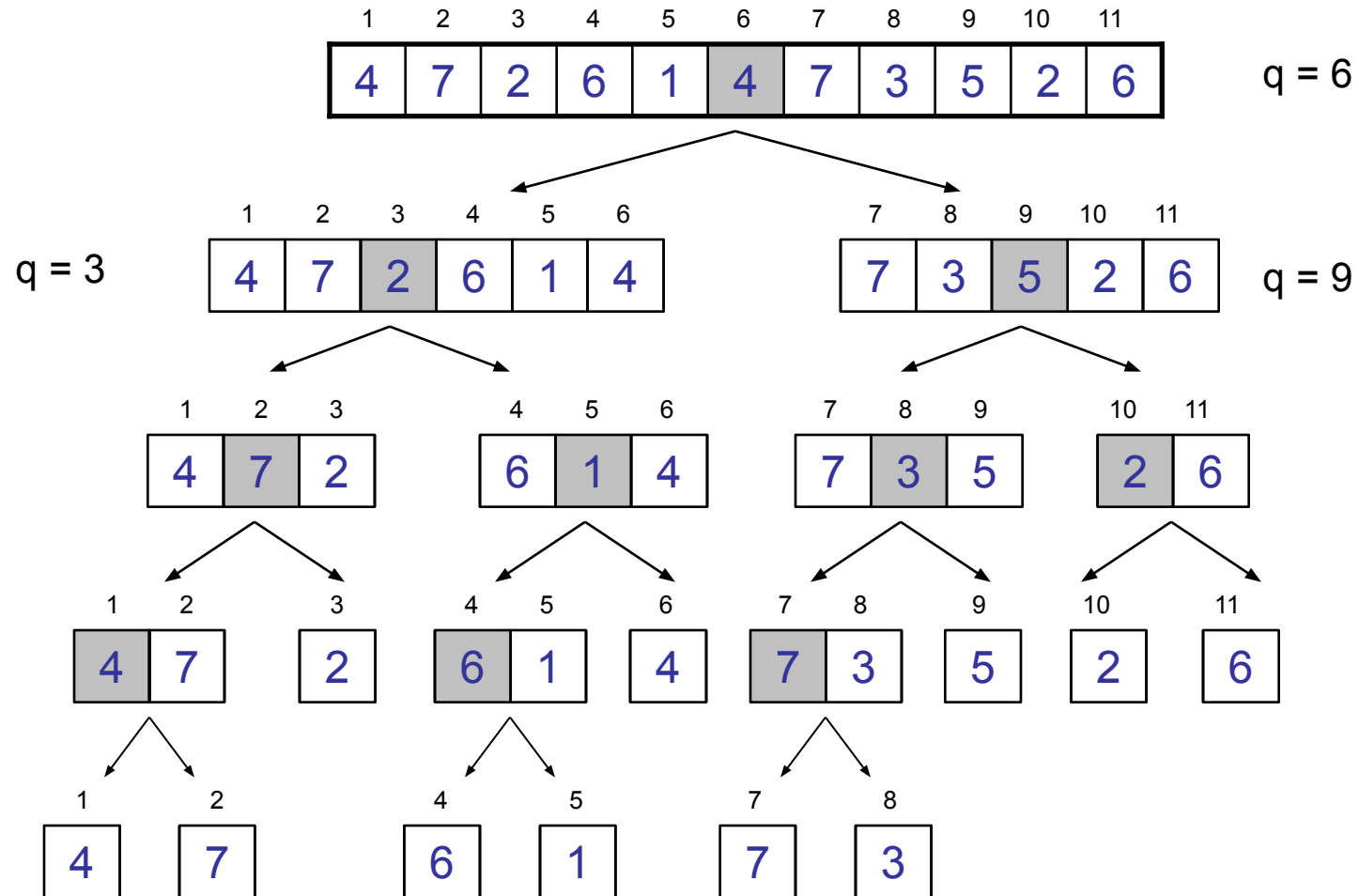
# Exemplo: n Potência de 2

Conquista e  
Combinação



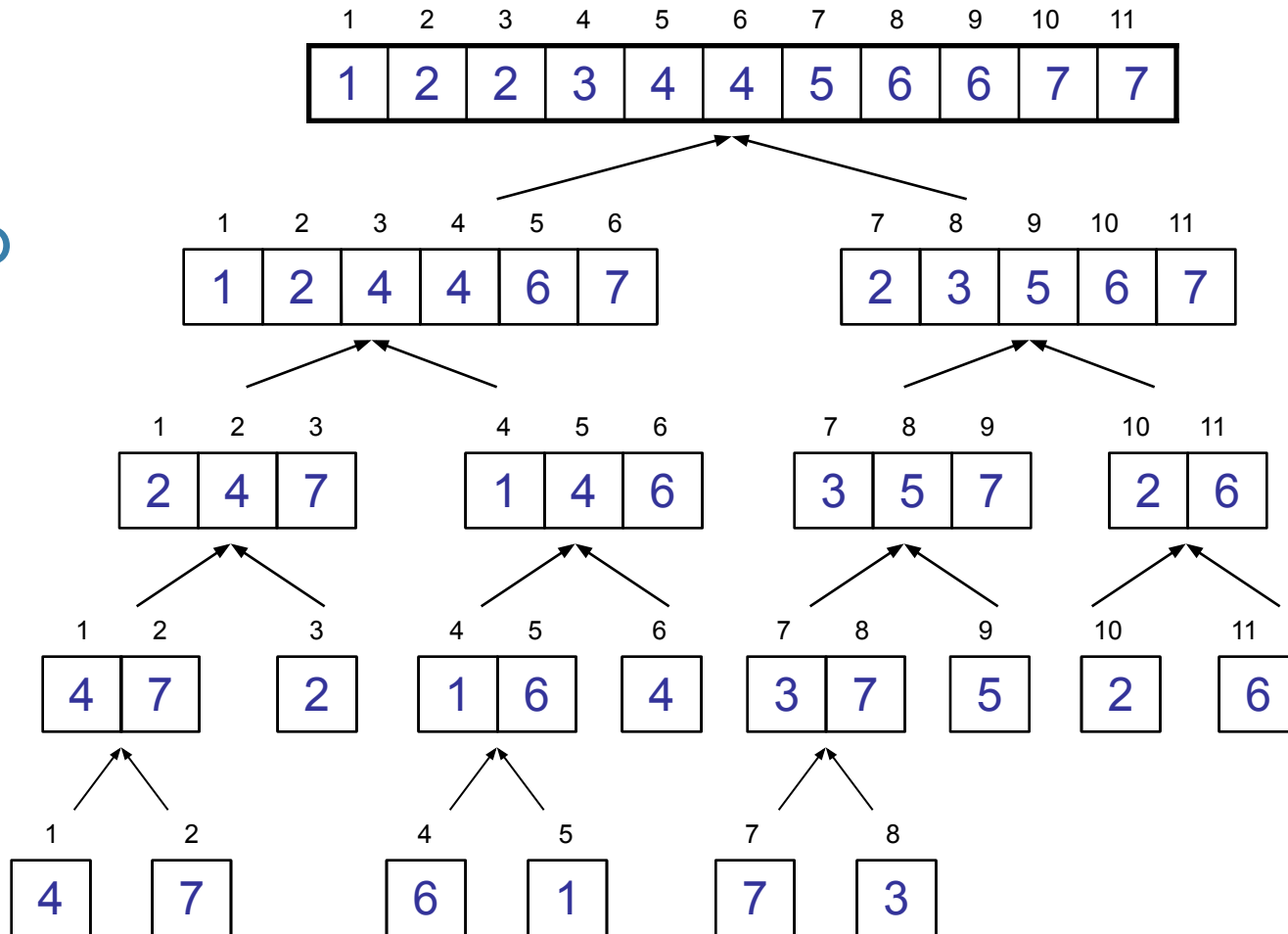
# Exemplo: $n$ não é potência de 2

## Divisão

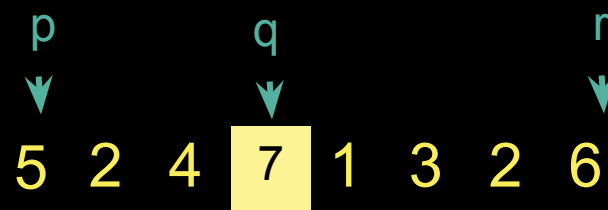


# Exemplo: $n$ não é potência de 2

Conquista e  
Combinação

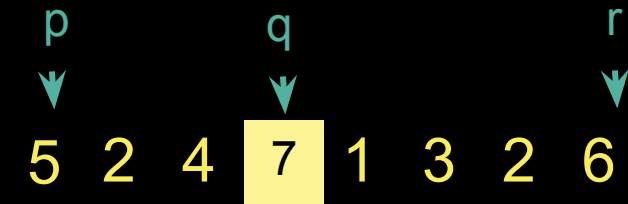


# Merge (combinação)

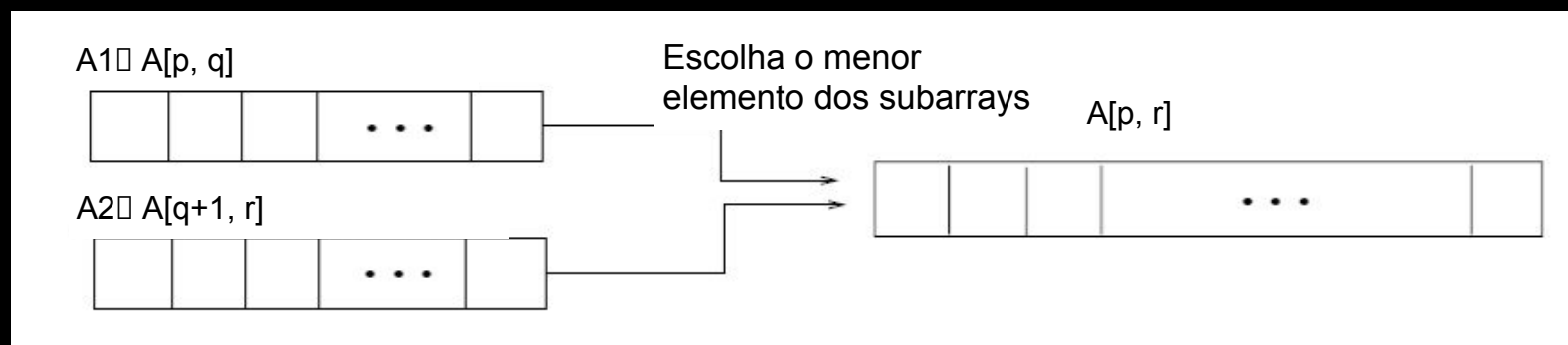


- **Entrada:** Array  $A$  e índices  $p, q, r$  tal que  $p \leq q < r$ 
  - Subarrays  $A[p \dots q]$  e  $A[q + 1 \dots r]$  são ordenados
- **Saída:** Um único subarray ordenado  $A[p \dots r]$

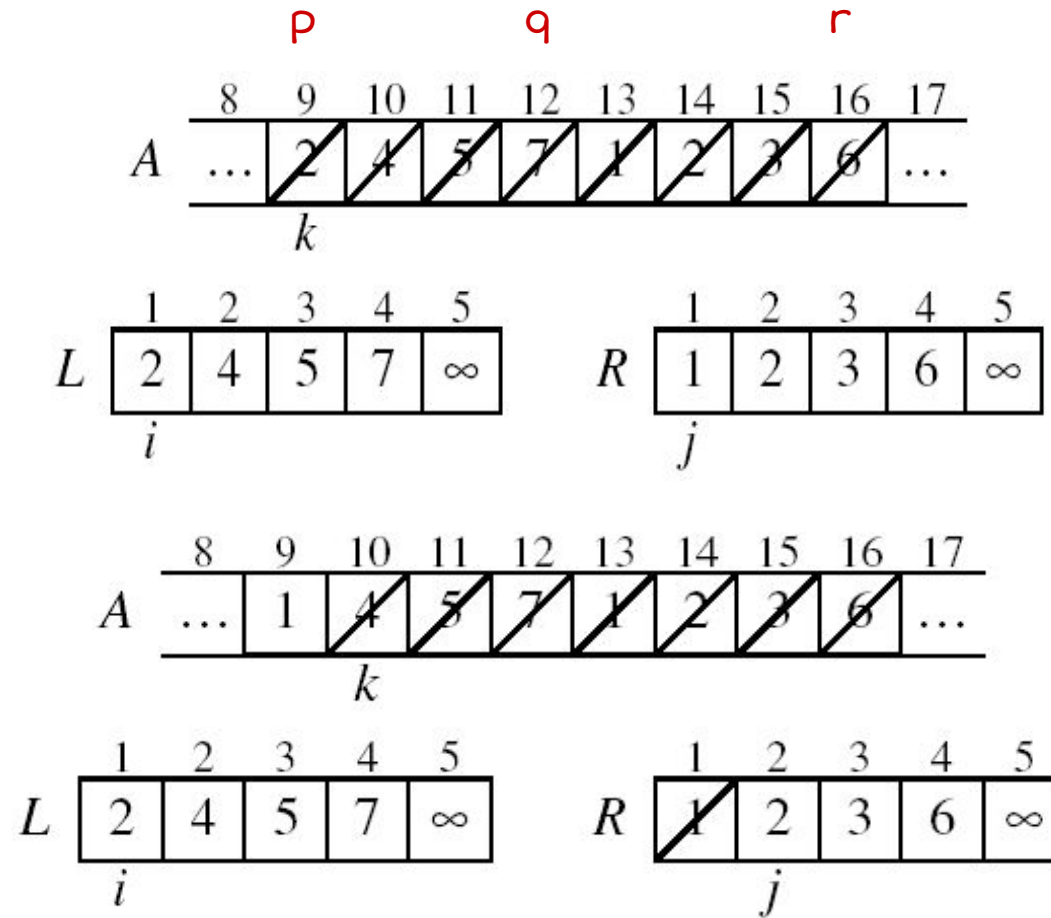
# Merge



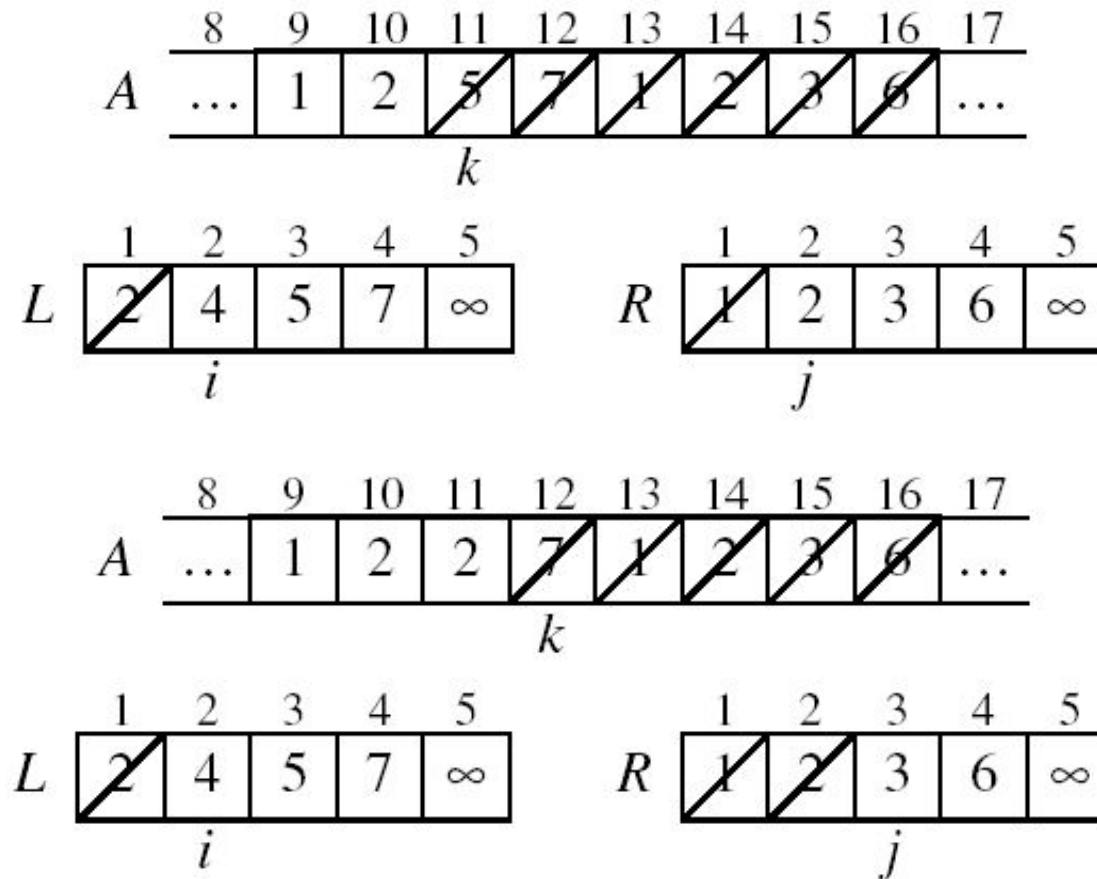
- Ideia para combinar os dois subarrays:
  - Temos duas pilhas de cartas ordenadas
    - Escolha a menor das duas cartas do topo
    - Remova-a e coloque-a na pilha de saída
    - Repita o processo até que uma pilha esteja vazia
  - Pegue a pilha restante e coloque-a virada para baixo na pilha de saída



# Exemplo: MERGE(A, 9, 12, 16)

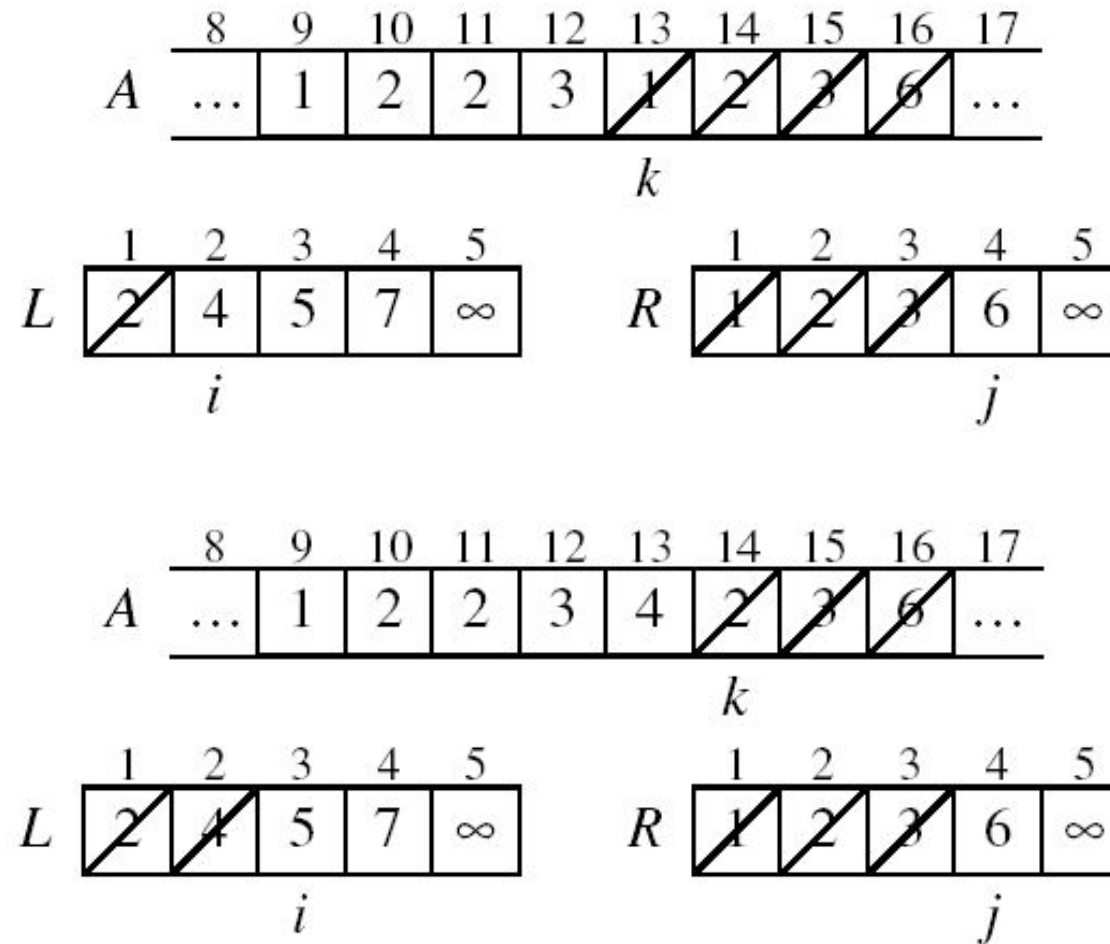


# Exemplo: MERGE(A, 9, 12, 16)

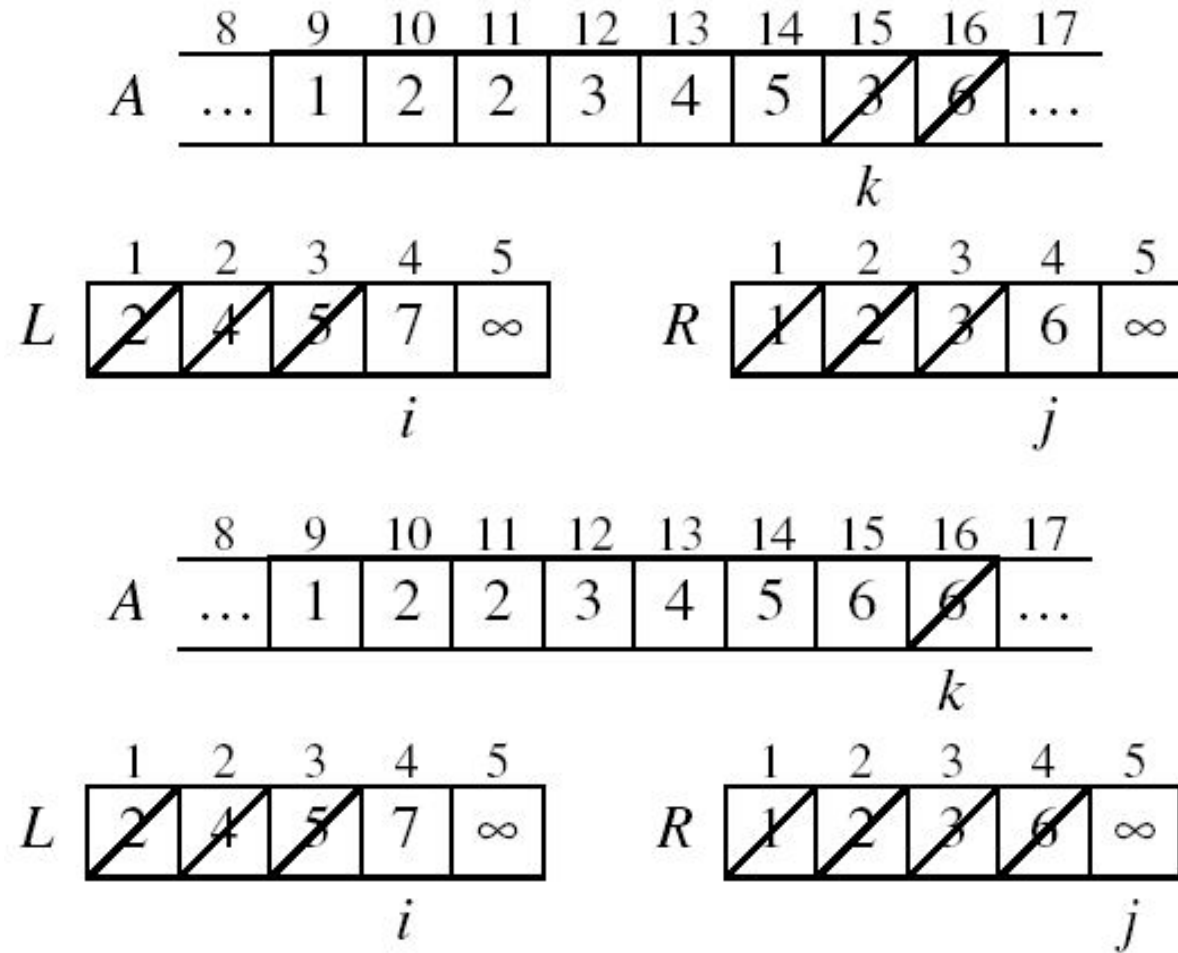




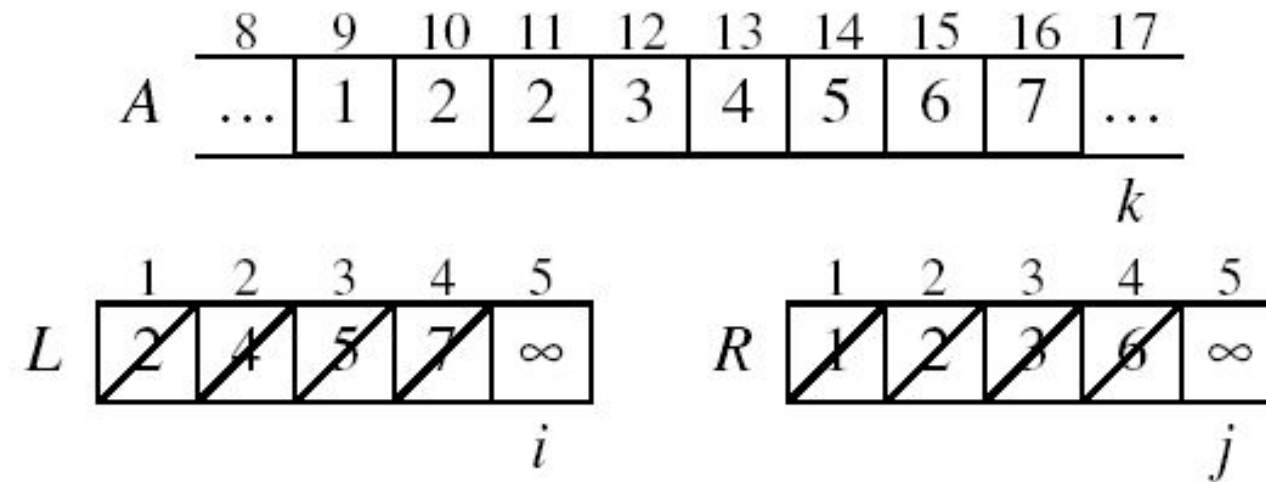
# Exemplo (cont.)



# Exemplo (cont.)



# Exemplo (cont.)



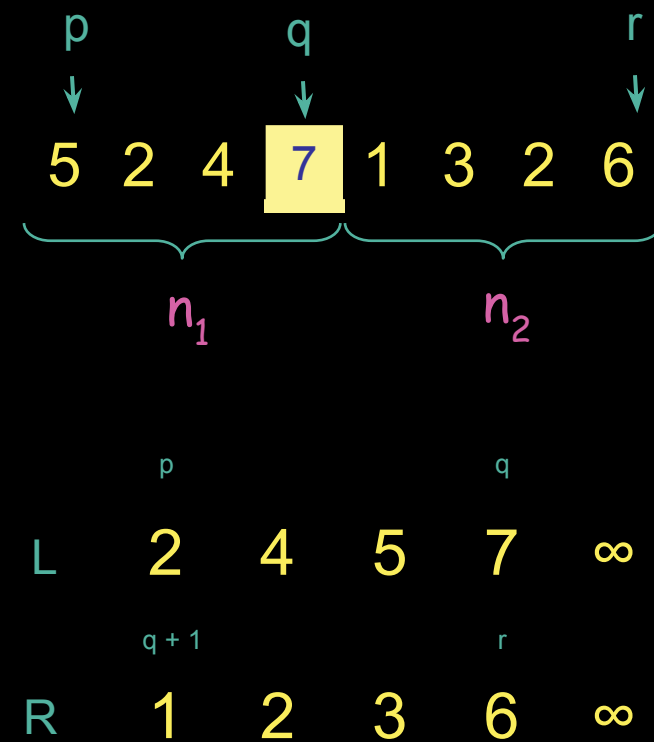
Pronto!

# Merge

```

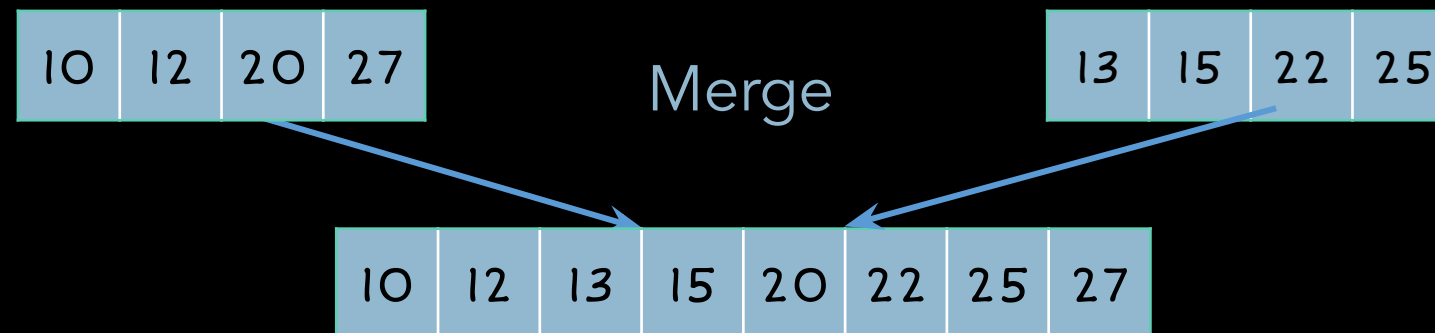
MERGE(A, p, q, r)
1.  Compute  $n_1$  e  $n_2$ 
2.  Copie o primeiro elemento de  $n_1$  em  $L[1 \dots n_1 + 1]$  e o
    próximo elemento de  $n_2$  em  $R[1 \dots n_2 + 1]$ 
3.   $L[n_1 + 1] \leftarrow \infty$ ;       $R[n_2 + 1] \leftarrow \infty$ 
4.   $i \leftarrow 1$ ;       $j \leftarrow 1$ 
5.  para  $k \leftarrow p$  até  $r$  faça
6.      se  $L[i] \leq R[j]$  então
7.           $A[k] \leftarrow L[i]$ 
8.           $i \leftarrow i + 1$ 
9.      senão
10.          $A[k] \leftarrow R[j]$ 
11.          $j \leftarrow j + 1$ 

```



# Tempo de execução do Merge (assuma o último loop)

- Inicialização (copiando em arrays temporários):
  - $\Theta(n_1 + n_2) = \Theta(n)$
- Adicionando os elementos ao array final:
  - $n$  iterações, cada uma levando tempo constante  $\Rightarrow \Theta(n)$
- Tempo total para realizar o merge:
  - $\Theta(n)$





E agora?  
Como analisar a  
complexidade do  
MergeSort?

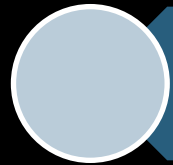
# Relembrando os passos para análise da complexidade de algoritmos

Identificar  
operações  
primitivas

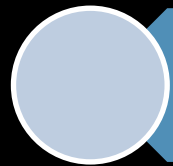
Identificar a  
quantidade de  
vezes que cada  
uma dessas  
primitivas é  
executada

Somar essas  
execuções

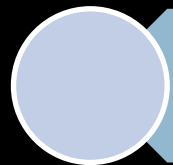
# Quais são as operações primitivas?



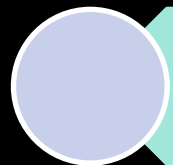
Avaliação de expressões booleanas



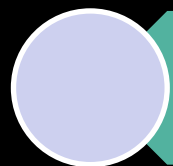
Operações matemáticas



Retorno de métodos



Atribuição



Acesso à variáveis e posições arbitrárias de um array



# Análise de algoritmos recursivos

Ao seguir os passos anteriores chegamos a uma função que descreve o tempo de execução do algoritmo.

Estamos interessados na ordem de crescimento dessa função, mais do que nos seus termos detalhados.

Deste modo, podemos aplicar as seguintes diretrizes para identificar a classe de complexidade das funções:

- Eliminar constantes;
- Eliminar expoentes de menor magnitude.

Exemplo:

$$f(n) = 70n + 32n + 231$$

$$f(n) = \Theta(n)$$

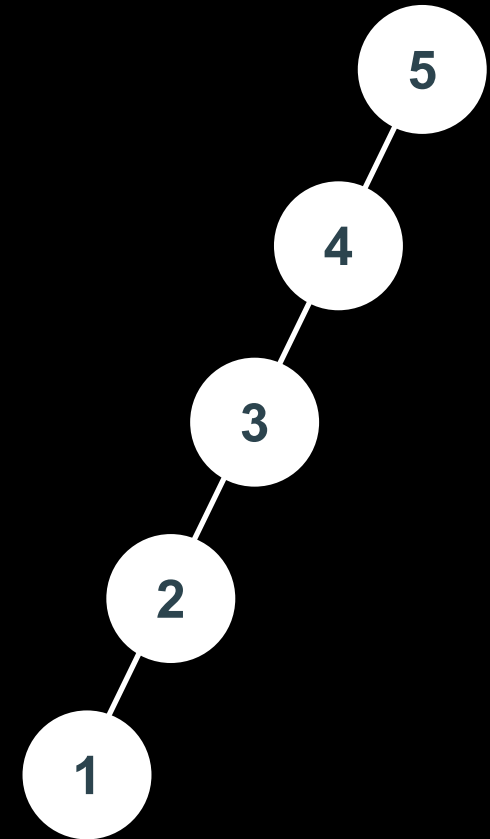
# Análise de algoritmos recursivos - FAT

Número de chamadas recursivas:  $n$

Complexidade do procedimento em  
cada chamada:  $O(1)$

**Complexidade do FAT recursivo:  $O(n)$**

```
inteiro FAT(n);  
  se  $n < 2$ :  
    retornar 1  
  senão:  
    retornar  $n * \text{FAT}(n-1)$ 
```



# Análise de algoritmos recursivos - FIB

Número de chamadas recursivas:

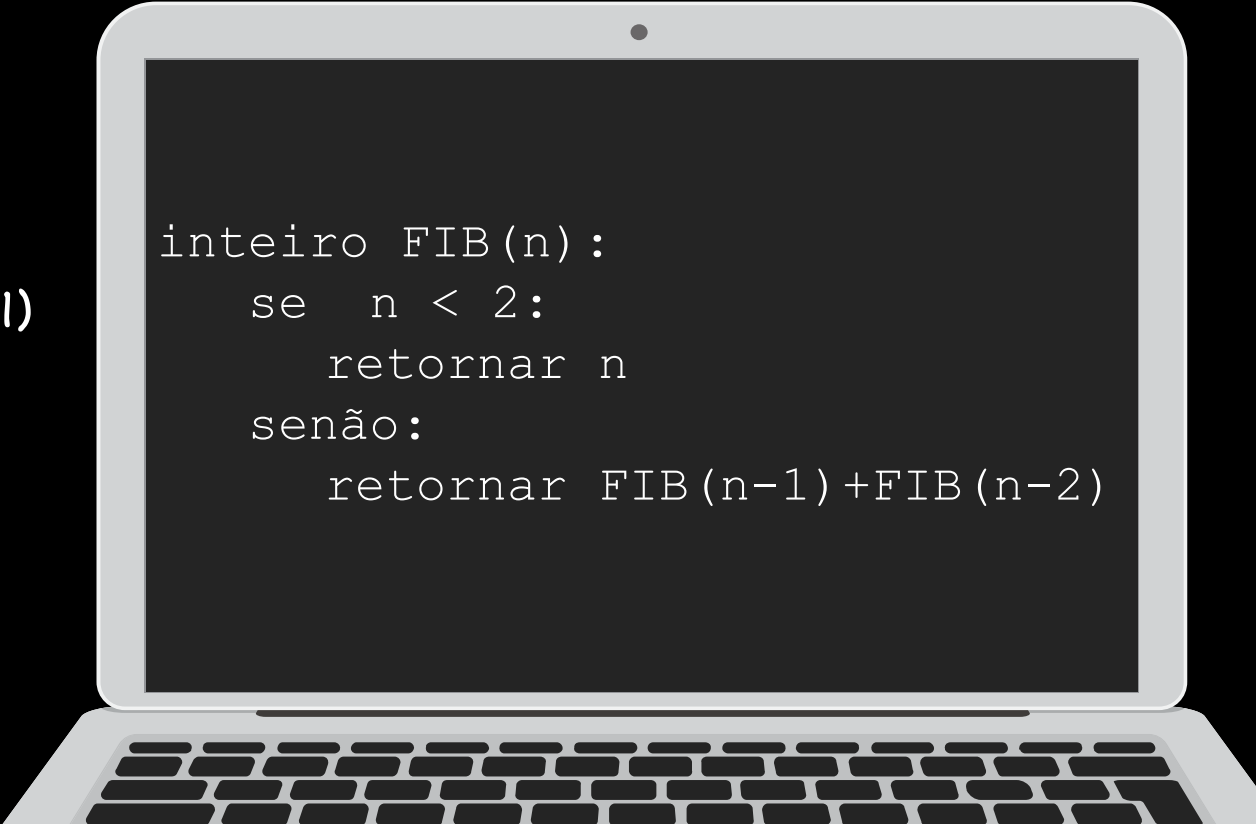
$T(n)$

$T(0) = 1; T(1) = 1;$

$T(n) = 1 + T(n-1) + T(n-2);$

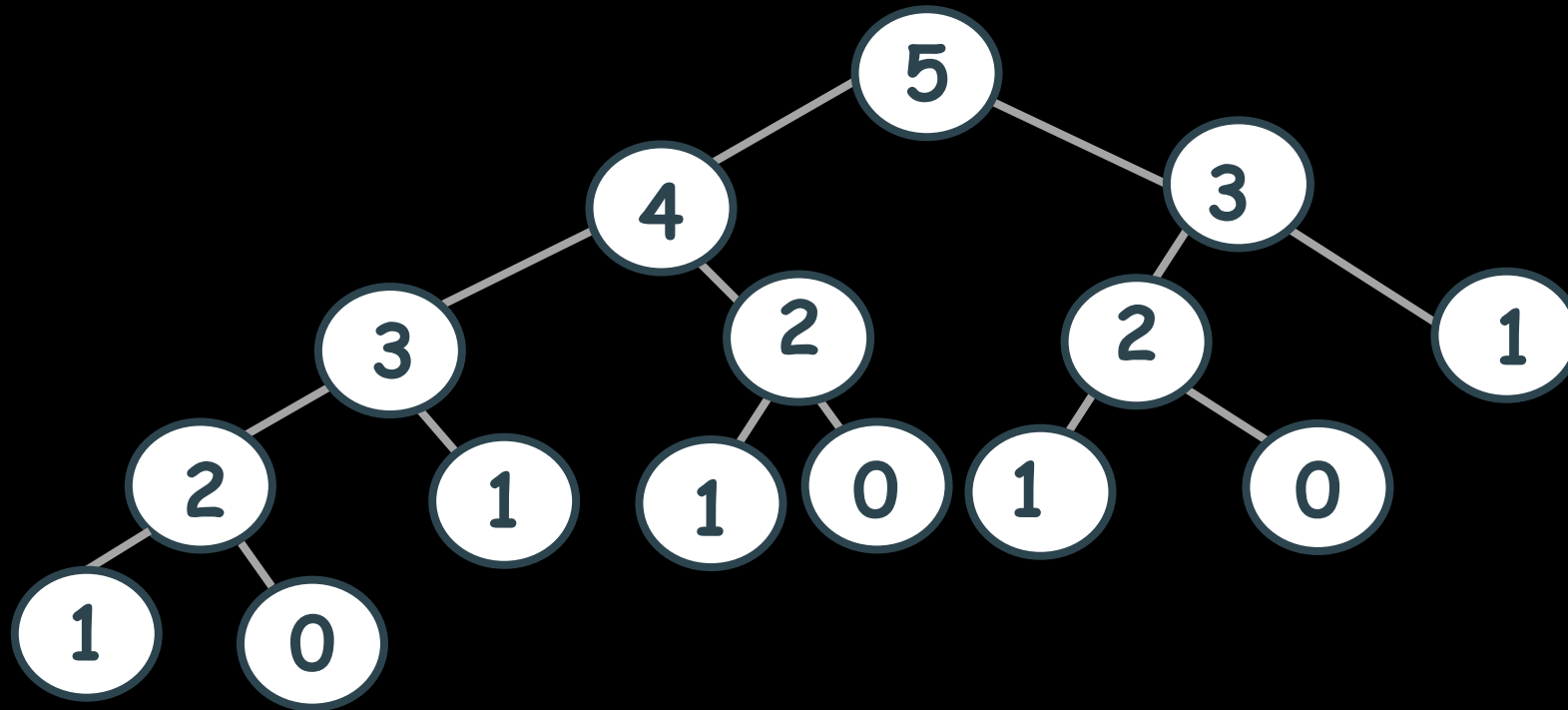
Complexidade do procedimento:  $O(1)$

Complexidade do FIB recursivo: ?

A laptop is shown from a front-three-quarter view, with its screen displaying a recursive function for calculating Fibonacci numbers. The code is written in Portuguese and uses a simple conditional structure with 'se' (if) and 'senão' (else) keywords. The function is named 'FIB' and takes a parameter 'n'. The base cases are 'se n < 2: retornar n'. The recursive case is 'senão: retornar FIB(n-1) + FIB(n-2)'. The laptop has a silver-colored bezel and a black keyboard.

```
inteiro FIB(n):  
    se n < 2:  
        retornar n  
    senão:  
        retornar FIB(n-1) + FIB(n-2)
```

# Análise de algoritmos recursivos - FIB(5)

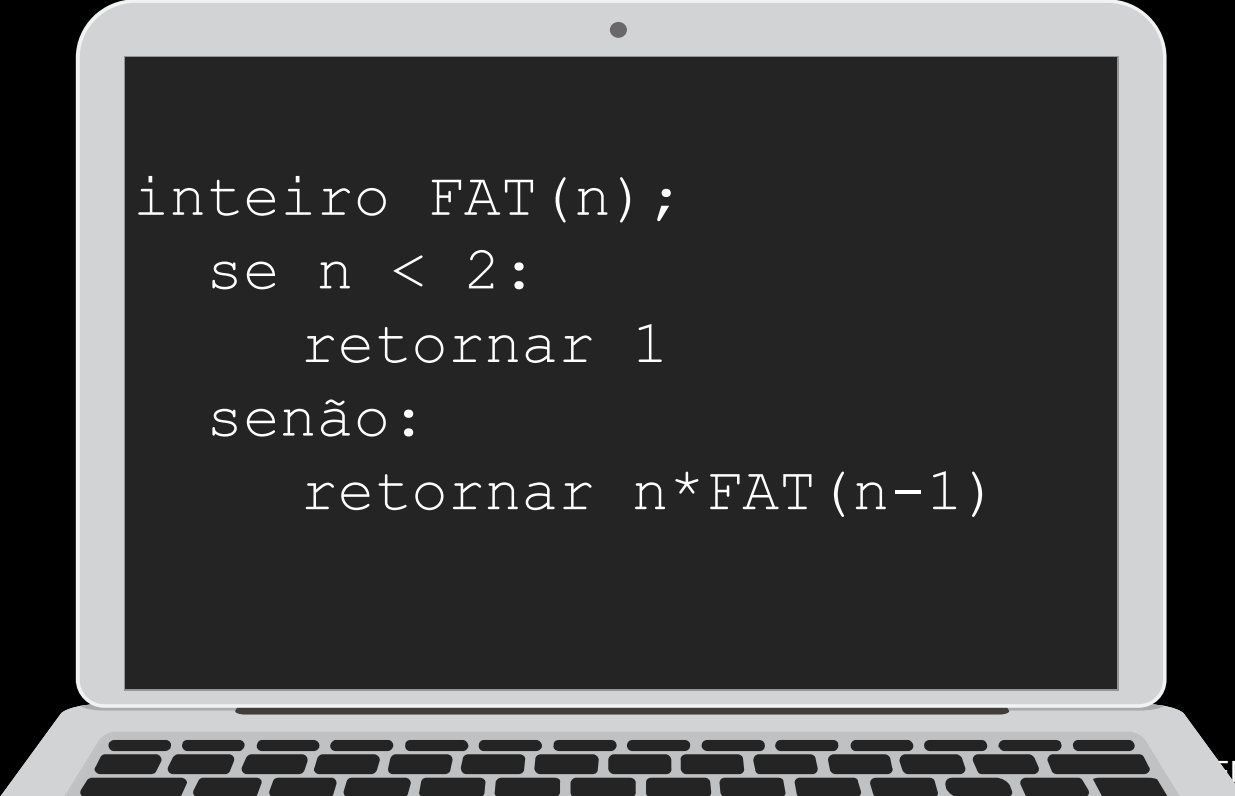


**Pergunta:** Afinal qual o problema da recursão ?

**Resposta:** A mesma chamada é feita inúmeras vezes (milhares, para valores médios de  $n$ ), para realizar o mesmo cálculo.

# Qual é o problema?

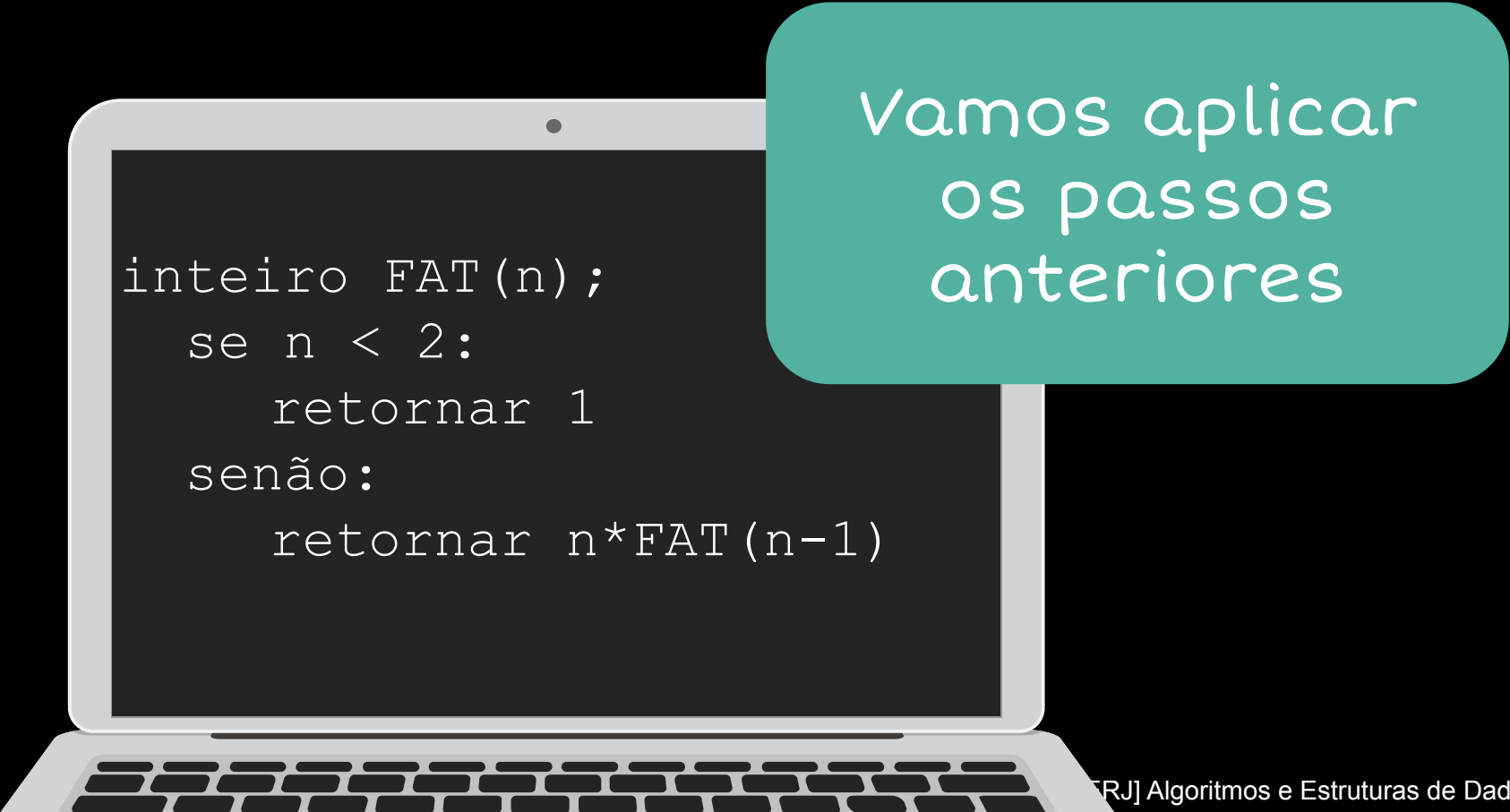
Em algoritmos recursivos a aplicação dos passos anteriores não é direta, pois um algoritmo recursivo é definido em termos dele mesmo.

A stylized illustration of a laptop with a light gray frame and a dark gray screen. The screen displays a recursive function in a light gray monospace font. The function is for calculating the nth Fibonacci number, with a base case for n < 2 and a recursive case for n >= 2. The laptop is shown from a slightly elevated front perspective.

```
inteiro FAT(n);  
  se n < 2:  
    retornar 1  
  senão:  
    retornar n * FAT(n-1)
```

# Qual é o problema?

Em algoritmos recursivos a aplicação dos passos anteriores não é direta, pois um algoritmo recursivo é definido em termos dele mesmo.

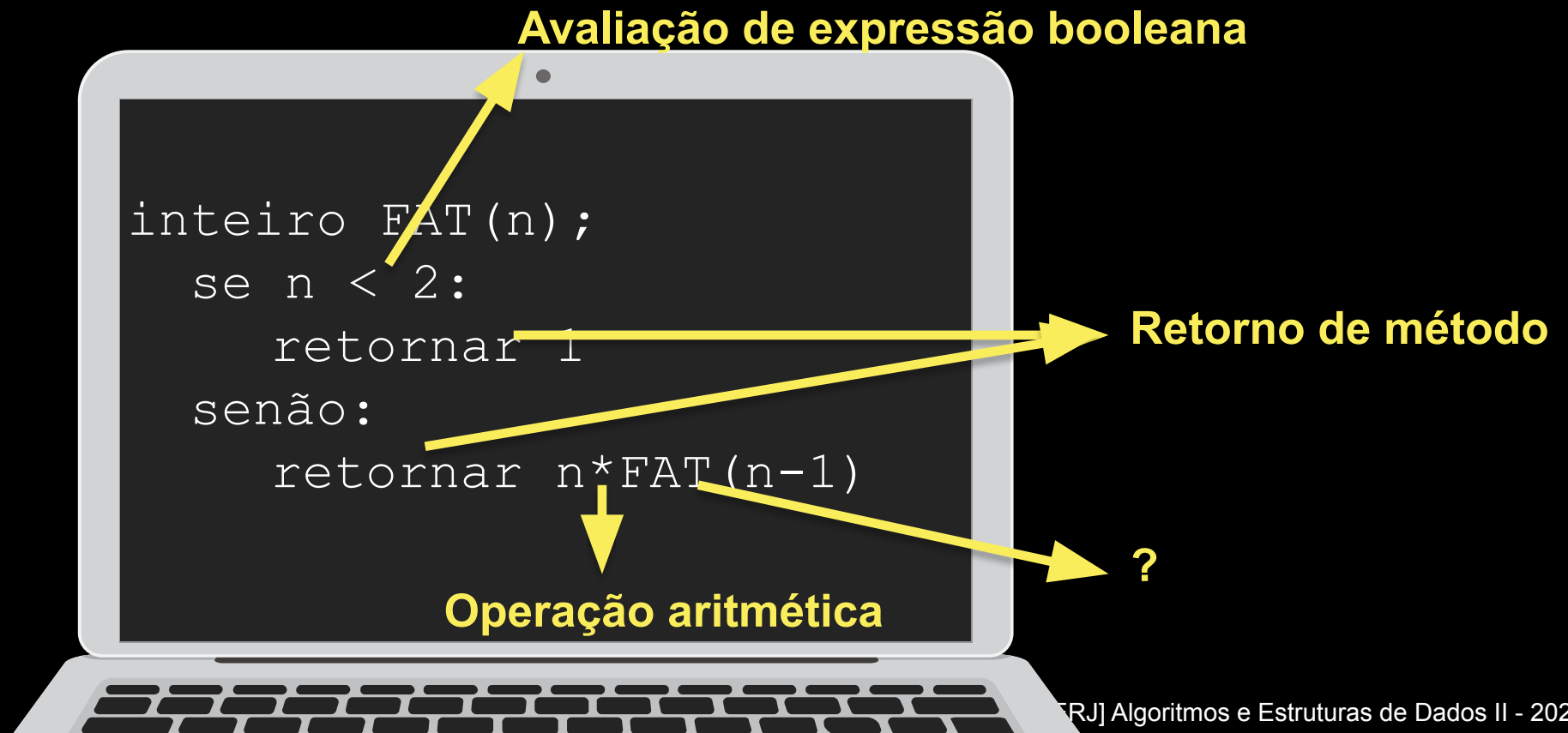
A laptop is shown with a dark screen displaying a recursive function in a light-colored monospace font. To the right of the laptop, a teal-colored rounded rectangle contains text in white. The laptop's keyboard is visible at the bottom.

```
inteiro F(n);  
  se n < 2:  
    retornar 1  
  senão:  
    retornar n * F(n-1)
```

Vamos aplicar  
os passos  
anteriores

# Qual é o problema?

Em algoritmos recursivos a aplicação dos passos anteriores não é direta, pois um algoritmo recursivo é definido em termos dele mesmo.



# Relação de recorrência

Relação de recorrência é uma equação ou inequação que descreve uma função em termos dela mesma considerando entradas menores.

A função que descreve o tempo de execução de um algoritmo recursivo é dada por sua relação de recorrência.

A relação de recorrência que descreve o algoritmo de cálculo do fatorial: simplificando temos:

Ou seja, o custo de calcular  $\text{fatorial}(n)$  é o custo de calcular  $\text{fatorial}(n-1)$  somado às primitivas que são executadas a cada passo da recursão que, nesse caso, representam 1.



# Desafio

“ Resolver a relação de recorrência correspondente ao algoritmo para determinar o seu tempo de execução. ”

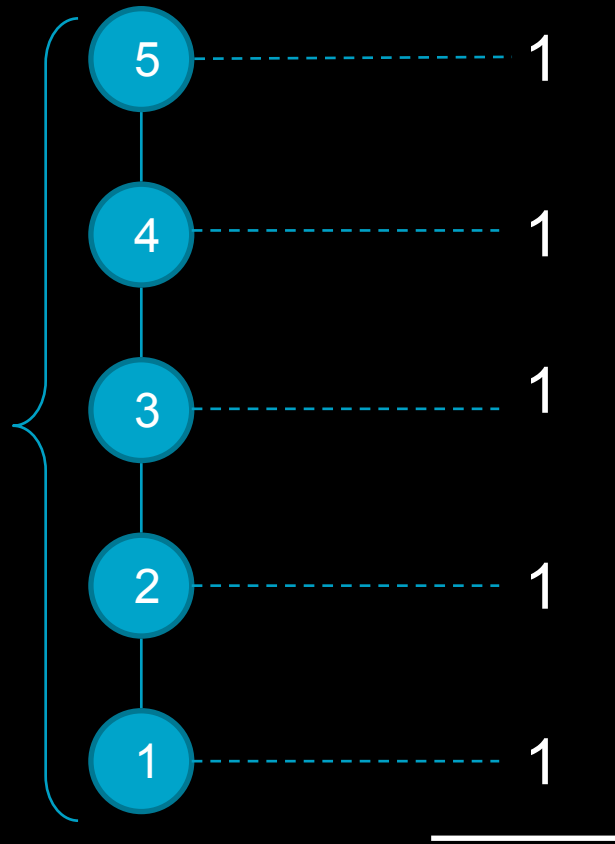
# Árvore de recursão

Método iterativo

A ideia para resolver uma relação de recorrência é simular a sua execução através de uma árvore, onde os nós representam a entrada e as arestas representam a chamada recursiva.

# Árvore de recursão

Exemplo: Fatorial

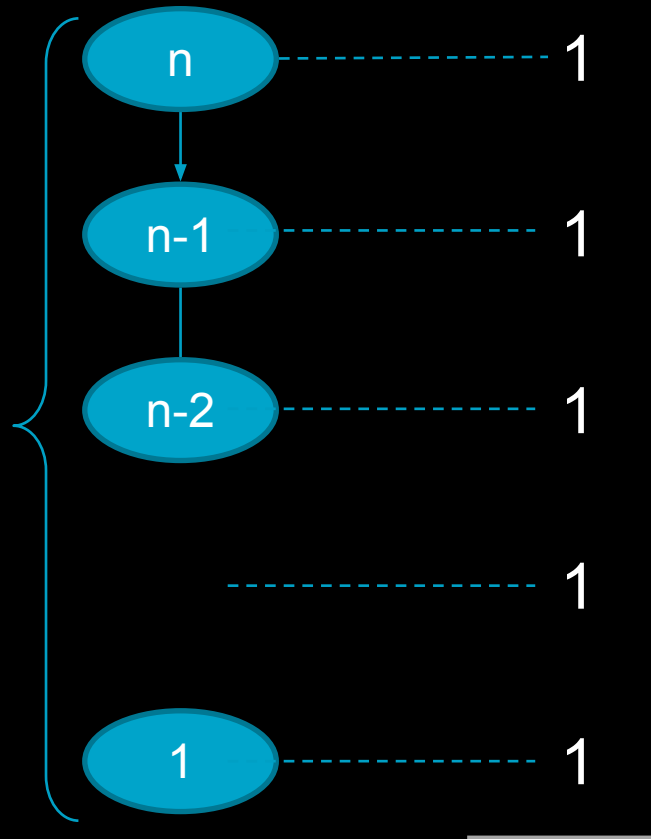


Custo total = 5

O custo total é a soma dos custos de cada nível, ou seja, a soma dos custos de cada passo da recursão

# Árvore de recursão

Exemplo: Fatorial



$$\text{Custo total} = 1 \cdot (n-1) + 1$$

Queremos definir o tempo de execução do algoritmo em função de uma entrada de tamanho  $n$ .

- Para calcular a função que define o tempo de execução desse algoritmo, precisamos somar os custos de cada nível
- Somaremos o valor 1 uma quantidade de vezes representada por  $h+1$ , onde  $h$  é a altura da árvore e o  $+1$  é o custo da última execução (if  $n < 2$ )

$h$  é o comprimento do maior caminho da raiz a uma das folhas

# Passos para análise de algoritmos recursivos

Estabelecer  
a relação  
de  
recorrência

Expandir a  
árvore de  
execução  
baseado na  
relação de  
recorrência

Determinar  
a altura  $h$   
máxima da  
árvore

Somar o  
custo de  
cada nível  
de execução

Somar o  
custo total  
(soma do  
custo de  
todos os  
níveis)

# Voltando ao Mergesort

```
MergeSort(A, p, r)

  if p < r
    then q ← ⌊(p + r) / 2⌋
         MERGE-SORT(A, p, q)
         MERGE-SORT(A, q + 1, r)
         MERGE(A, p, q, r)
```

Chamada inicial: MERGE-SORT(A, 1, n)

# Voltando ao Mergesort

Primeira etapa: Identificar a relação de recorrência

```
MergeSort(A, p, r)

  if p < r
    then q ← ⌊(p + r) / 2⌋
         MERGE-SORT(A, p, q)
         MERGE-SORT(A, q + 1, r)
         MERGE(A, p, q, r)
```

Chamada inicial: MERGE-SORT(A, 1, n)

# Voltando ao Mergesort

Primeira etapa: Identificar a relação de recorrência

```
MergeSort(A, p, r)
```

```
  if p < r
```

```
    then q ← ⌊(p + r) / 2⌋
```

```
      MERGE-SORT(A, p, q)
```

```
      MERGE-SORT(A, q + 1, r)
```

```
      MERGE(A, p, q, r)
```

→  $\theta(1)$

→  $\theta(1)$

→  $T(n/2)$

→  $T(n/2)$

→  $\theta(n)$

Chamada inicial: MERGE-SORT(A, 1, n)

$$T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{2}\right) + n$$

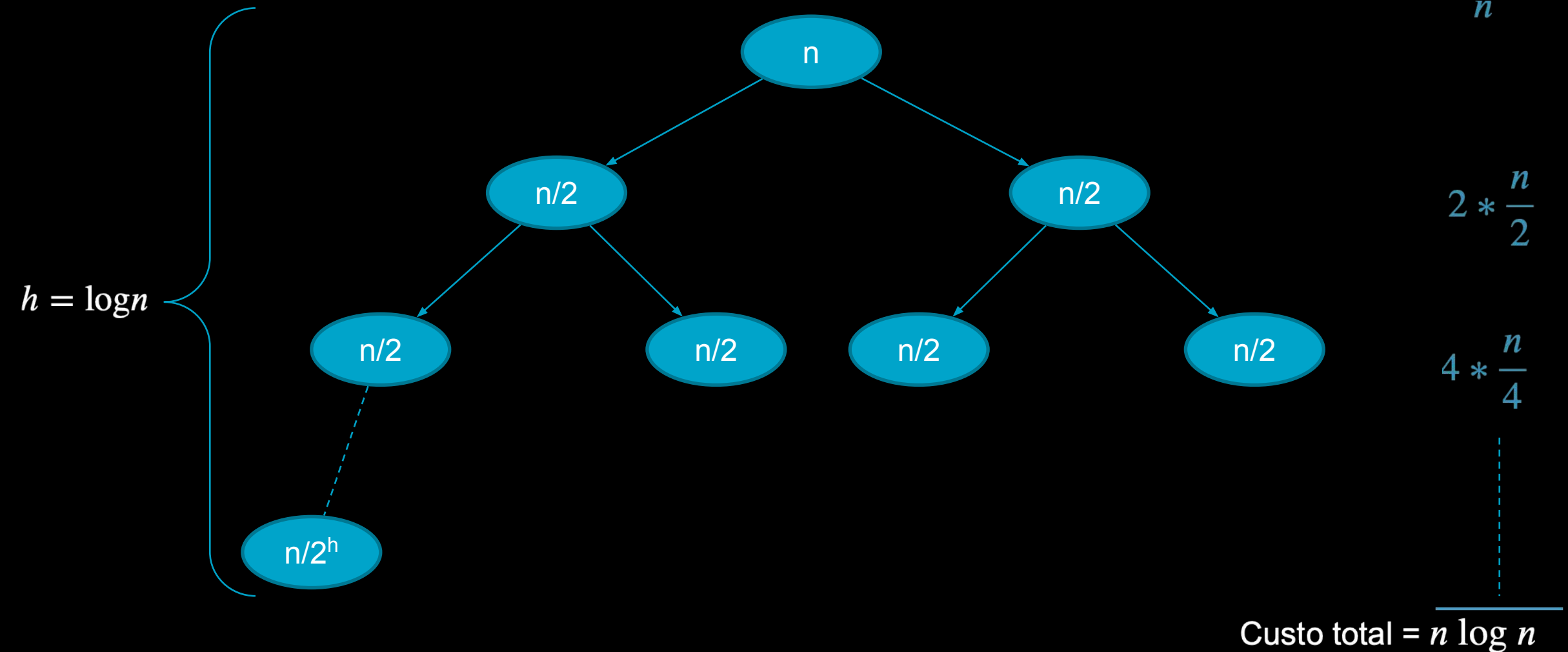
simplicando

$$T(n) = 2T\left(\frac{n}{2}\right) + n$$



# Exemplo: Mergesort

Árvore de recursão





E se determinar  
a complexidade  
com a árvore de  
recursão for  
muito  
trabalhoso?

# Teorema Mestre

Permite identificar a classe de complexidade de um algoritmo aplicando apenas algumas operações matemáticas e comparando ordem de complexidade de funções.



# Como funciona?

O primeiro ponto a ser observado é que a relação de recorrência precisa ter determinadas propriedades.

Vamos analisar essas propriedades:

$$T(n) = a * T\left(\frac{n}{b}\right) + f(n)$$

Sendo  $a \geq 1$ ,  $b > 1$  e  $f(n)$  não negativa.

$a$  representa o número de chamadas recursivas (quantidade de subproblemas

$b$  representa em quanto a entrada é diminuída a cada chamada recursiva

$f(n)$  representa o custo parcial de cada etapa da recursão

# Teorema Mestre

É uma maneira direta de resolver relações de recorrência.

O Teorema Mestre estabelece que:

(1) Se  $f(n) < n^{\log_b a}$ , então  $T(n) = \theta(n^{\log_b a})$ .

(2) Se  $f(n) = n^{\log_b a}$ , então  $T(n) = \theta(f(n) * \log n)$ .

(3) Se  $f(n) > n^{\log_b a}$ , então  $T(n) = \theta(f(n))$ .

Desse modo, se a relação de recorrência obedecer às restrições  $a \geq 1$ ,  $b > 1$  e  $f(n)$  não negativa, basta aplicarmos o teorema.

# Teorema Mestre

Exemplo

$$T(n) = 9T\left(\frac{n}{2}\right) + 1000 * n^2$$

$$a = 9;$$

$$b = 2;$$

$$f(n) = 1000 * n^2.$$

Comparando  $1000 * n^2$  com  $n^{\log_b a}$ , temos que  $1000 * n^2 < n^3$ . Portanto, aplicando o caso 1 do Teorema Mestre, podemos afirmar que  $T(n) = \Theta(n^{\log_b a})$  e, portanto,

$$T(n) = \theta(n^3)$$

# Teorema Mestre

Exemplo

$$T(n) = 2T\left(\frac{n}{2}\right) + 10 * n$$

$$a = 2;$$

$$b = 2;$$

$$f(n) = 10 * n .$$

Comparando  $10 * n$  com  $n^{\log_b a}$  temos que  $10 * n = n$ , pois comparamos a ordem de grandeza das funções e, quando fazemos isso, as constantes não importam. Portanto, aplicando o caso 2 do Teorema Mestre, podemos afirmar que

$$T(n) = \theta(n \log_2 n)$$

# Teorema Mestre

Exemplo

$$T(n) = 2T\left(\frac{n}{2}\right) + n^2$$

$$a = 2;$$

$$b = 2;$$

$$f(n) = n^2.$$

Comparando  $n^2$  com  $n^{\log_b a}$  temos que  $n^2 > n$ . Portanto, aplicando o caso 3 do Teorema Mestre, podemos afirmar que  $T(n) = \theta(f(n))$  e, portanto,

$$T(n) = \theta(n^2)$$



# Resumo da análise de algoritmos de divisão e conquista

- A recorrência é baseada nos três passos do paradigma:
  - $T(n)$  – tempo de execução em um problema de tamanho  $n$
  - **Divide** o problema em  $a$  subproblemas, cada um de tamanho  $n/b$ : leva  $D(n)$
  - **Conquista** (resolve) os subproblemas  $aT(n/b)$
  - **Combina** as soluções  $C(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq c \\ aT(n/b) + D(n) + C(n) & \text{caso contrário} \end{cases}$$

# Tempo de Execução do MergeSort

- **Divisão:**
  - computa  $q$  como a média de  $p$  e  $r$ :  $D(n) = \Theta(1)$
- **Conquista:**
  - Recursivamente resolve 2 subproblemas, cada um com tamanho  $n/2 \Rightarrow 2T(n/2)$
- **Combinação:**
  - MERGE em um subarray de  $n$  elementos leva tempo  $\Theta(n) \Rightarrow C(n) = \Theta(n)$

$$T(n) = \begin{cases} \Theta(1) & \text{se } n = 1 \\ 2T(n/2) + \Theta(n) & \text{se } n > 1 \end{cases}$$

# MergeSort - Solução da Recorrência

$$T(n) = \begin{cases} c & \text{se } n = 1 \\ 2T(n/2) + cn & \text{se } n > 1 \end{cases}$$

Usando o Teorema Mestre

Compara  $n$  com  $f(n) = cn$

Caso 2:  $T(n) = \Theta(n \log n)$

# Divisão e Conquista

## Quicksort

(Eficiência no caso médio)



# Quicksort

Utiliza a estratégia de divisão e conquista.

É muito rápido em média, mesmo sendo lento no pior caso

O passo da divisão é parte crítica do algoritmo, nele o vetor  $A[p..r]$  é particionado devolvendo um índice  $q$  tal que  $A[p..q-1] \leq A[q] < A[q+1..r]$ .

O elemento  $x = A[q]$  é chamado de pivô.

# Quicksort

A idéia recursiva básica é a seguinte:

a) Se o vetor tiver 0 ou 1 elementos ("problema infantil"), nada deve ser feito.

b) Senão, fazer uma partição no vetor, através de trocas, baseada em um pivô  $q$ , tal que os elementos da partição esquerda sejam  $\leq q$ , e os da direita,  $> q$ .



c) Depois ordenar, de forma independente, as partições obtidas. Todo o vetor estará, então, ordenado.

Este é um importante método de ordenação, inventado por Hoare, em 1962.

## Mergesort

1. Ordenar separadamente cada metade do vetor.
2. Fazer merge das duas metades ordenadas.



## Quicksort

1. Fazer uma partição no vetor com base em um pivô qualquer.
2. Ordenar separadamente cada uma das duas partições.

Nos dois casos, o problema “infantil” é ordenar um vetor com 0 ou 1 elemento, quando nada deve ser feito.

# Quicksort: Escolha do pivô

- A escolha do pivô é importante para o bom desempenho do algoritmo. Opções:
  - Primeiro elemento do vetor;
  - Último elemento do vetor;
  - Elemento do meio do vetor;
  - Escolha aleatória de um elemento do vetor.



# Quicksort: Particione

Invariantes:

no começo de cada  
iteração do loop para,

$A[p \dots i] \leq x$ ,

$A[i+1 \dots j-1] > x$ ,

$A[r] = x$

Consumo de tempo:

$(n)$  onde  $n := r - p$ .

```
Particione(A, p, r)
x ← A[r] %x é o "pivô"
i ← p-1
para j ← p até r - 1 faça
    se A[j] ≤ x então
        i ← i + 1
        A[i] ↔ A[j]
A[i+1] ↔ A[r]
retorne i + 1
```

# Quicksort: Particione

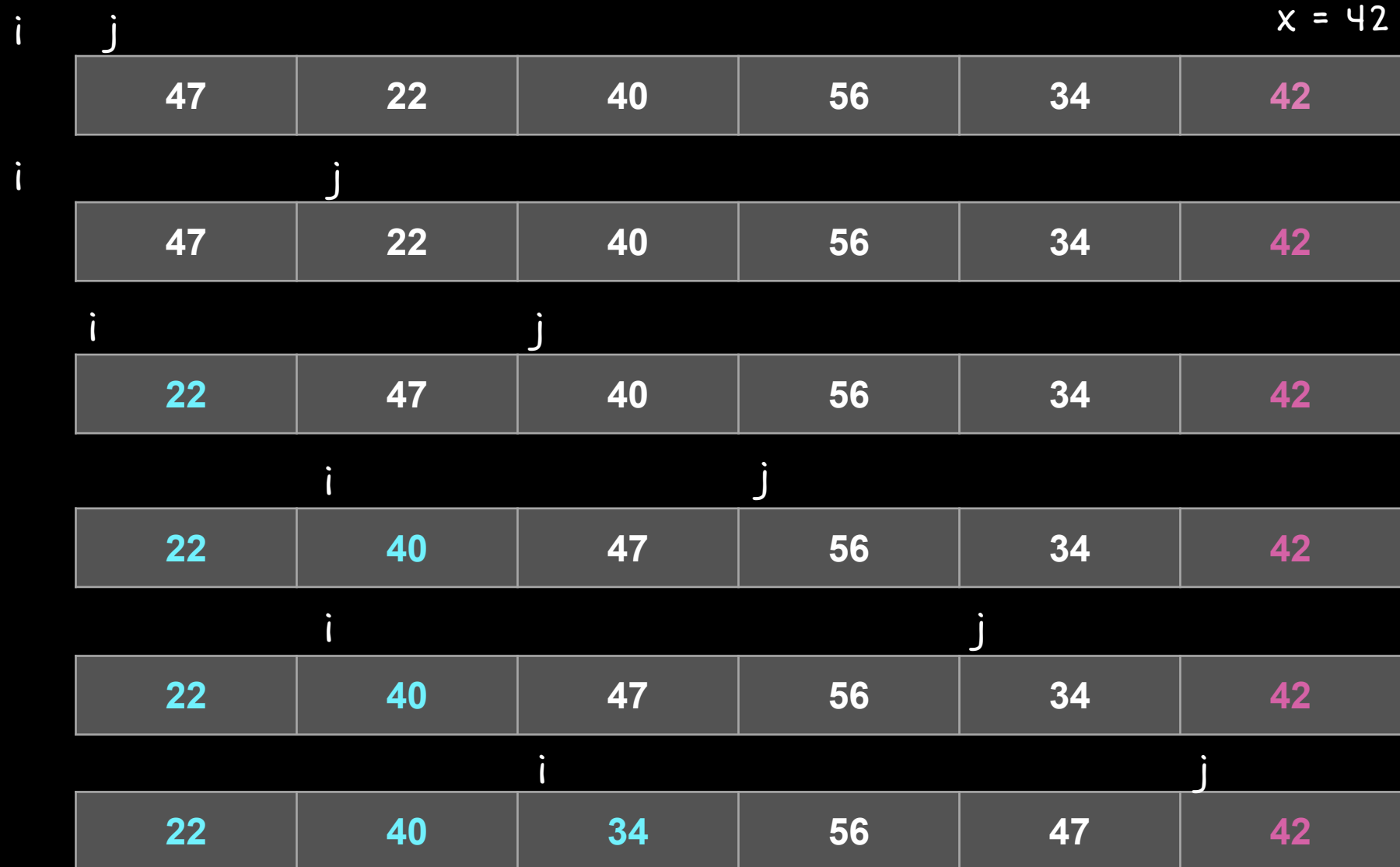
O vetor  $A[p \dots r]$  é particionado devolvendo um índice  $q$  tal que  $A[p \dots q-1] \leq A[q] < A[q+1 \dots r]$ .

O pivô será o último elemento do vetor

|   |    |    |    |    |    |    |
|---|----|----|----|----|----|----|
| p |    |    |    |    |    | r  |
|   | 47 | 22 | 40 | 56 | 34 | 42 |

|   |    |    |    |    |    |    |
|---|----|----|----|----|----|----|
| p |    |    | q  |    |    | r  |
|   | 47 | 22 | 40 | 42 | 34 | 56 |

# Quicksort: Particione



# Quicksort: Particione

|    |    |    |    |    |    |  |
|----|----|----|----|----|----|--|
|    |    | i  |    |    | j  |  |
| 22 | 40 | 34 | 56 | 47 | 42 |  |

Fazendo  $A[i+1] \leftrightarrow A[r]$

|    |    |    |    |    |    |  |
|----|----|----|----|----|----|--|
| 22 | 40 | 34 | 42 | 47 | 56 |  |
|----|----|----|----|----|----|--|

# Quicksort

```
QuickSort(A, p, r)
se p < r então
    q ← Particione(A, p, r)
    QuickSort(A, p, q - 1)
    QuickSort(A, q + 1, r)
```

```
Quicksort(A, 1, n)
```

# Complexidade do Quicksort

QuickSort(A,p,r)

se  $p < r$  então

$q \leftarrow \text{Particione}(A,p,r)$

    QuickSort(A,p,q - 1)

    QuickSort(A,q + 1,r)

$\Theta(1)$

$\Theta(n)$

$T(k)$

$T(n-k-1)$

$= T(k) + T(n-k-1) + \Theta(n + 1)$

Melhor caso (vetor ordenado):

$T(n) = 2T(n/2) + n + 2 = O(n \log n)$  (= Mergesort)

# Pior caso da Partição

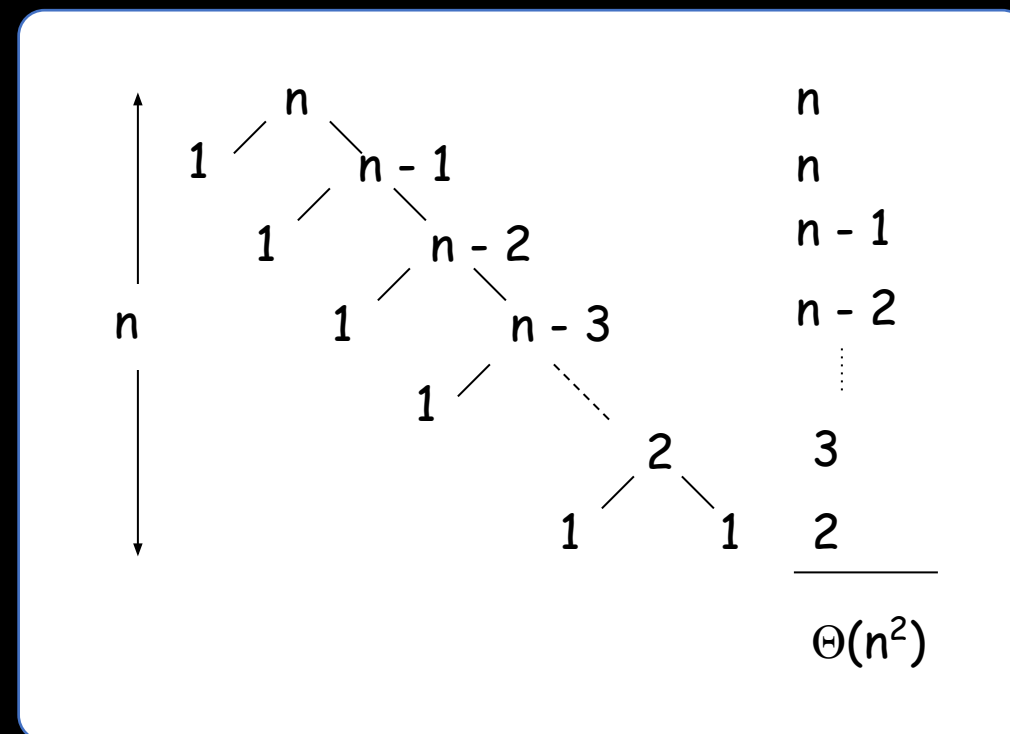
- Quando acontece?
  - Uma região tem 1 elemento e a outra tem  $n - 1$  elementos
  - Maior desbalanceamento possível
- Recorrência:  $q=1$

$$T(n) = T(1) + T(n - 1) + n,$$

$$T(1) = \Theta(1)$$

$$T(n) = T(n - 1) + n$$

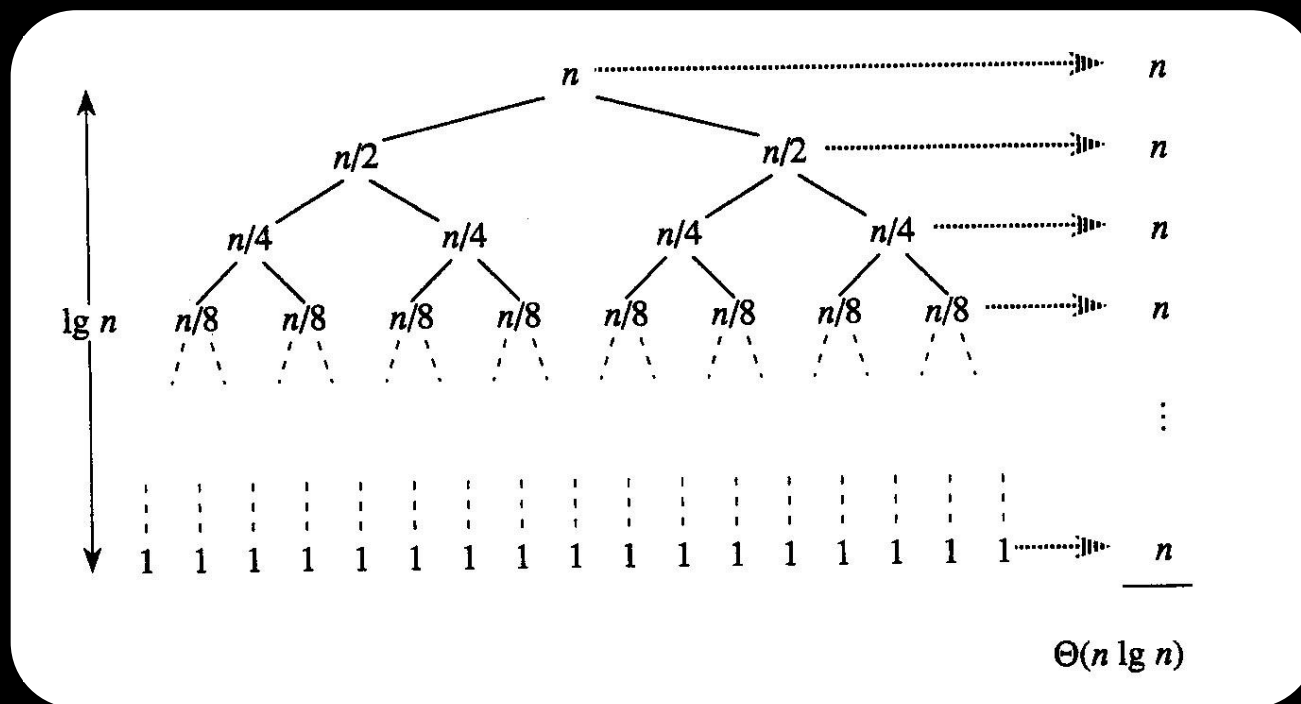
$$= n + \left( \sum_{k=1}^n k \right) - 1 = \theta(n) + \theta(n^2) = \theta(n^2)$$



Quando o pior caso acontece?

# Melhor caso da Partição

- Quando acontece?
  - O particionamento produz duas regiões de tamanho  $n/2$
- Recorrência:  $q=n/2$   
 $T(n) = 2T(n/2) + \Theta(n)$   
 $T(n) = \Theta(n \lg n)$   
(Pelo Teorema Mestre)





# Análise da Ordenação pelo Quicksort

## Complexidade:

Pior caso:  $O(n^2)$

Melhor caso = caso médio:  $O(n \log n)$

## Estabilidade (manutenção da ordem relativa de chaves iguais):

Algoritmo não estável

## Memória adicional:

Pilha para recursão

## Usos especiais:

Melhor algoritmo de ordenação em geral

# Divisão e Conquista

Cálculo de Combinações  
(Oportunidade)



# Cálculo de Combinações

## Problema:

Às vezes tem-se problemas numéricos em certos cálculos, como no cálculo de combinações, se usar a fórmula padrão.

$$\text{Comb}(n,p) = n!/(p!(n-p)!)$$

Por exemplo, o cálculo de  $\text{Comb}(50, 3)$  pode levar a um estouro numérico, se a função  $\text{Comb}()$  usar inteiros com 32 ou 64 bits.

# Cálculo de Combinações

A versão recursiva para o cálculo de  $\text{Comb}(n, p)$  pode evitar o problema.

Versão 1. Reescreve-se a função como uma recorrência:

$$\text{Comb}(n, p) = n, \text{ se } p = 1$$

$$\text{Comb}(n, p) = \text{Comb}(n-1, p-1) * n/p, \text{ se } p > 1,$$

$$\begin{aligned} \text{pois } \text{Comb}(n, p) &= n! / (p!(n-p)!) \\ &= (n-1)!n / ((p-1)!p (n-1-(p-1))!) \\ &= [(n-1)! / ((p-1)!(n-1-(p-1))!)] * n/p \\ &= \text{Comb}(n-1, p-1) * n/p \end{aligned}$$

# Cálculo de Combinações

Define-se a recorrência:

$\text{Comb}(n, p) = n$ , se  $p = 1$

$\text{Comb}(n, p) = \text{Comb}(n-1, p-1) * n/p$ .

```
Comb(n, p):  
    se p = 1  
        retornar n  
    senão:  
        retornar Comb(n-1, p-1) * n/p
```

Exemplo:  $\text{Comb}(50, 3)$ .

|    |   |                                           |
|----|---|-------------------------------------------|
| 50 | 3 | <b><math>1176 * 50 / 3 = 19600</math></b> |
| 49 | 2 | <b><math>48 * 49 / 2 = 1176</math></b>    |
| 48 | 1 | <b>48</b>                                 |

# Divisão e Conquista

## Torneio

(Simetria e Adaptação)



# Divisão e Conquista

Máximo e Mínimo

(Etapa intermediária)



# Máximo e Mínimo

**Problema:** Dado um conjunto de números  $S = \{s_1, s_2, \dots, s_n\}$ , determinar simultaneamente o menor e o maior elemento do conjunto.

Solução ingênua

Encontrar o mínimo e o máximo, separadamente.

Minimo:

```
min ← 1
para i ← 2..n
    se  $V[i] < V[\text{min}]$ 
        min ← i
```

Maximo:

```
max ← 1
para i ← 2..n
    se  $V[i] > V[\text{max}]$ 
        max ← i
```

Faz  $2(n-1)$  comparações.



# Máximo e Mínimo

**Problema:** Dado um conjunto de números  $S = \{s_1, s_2, \dots, s_n\}$ , determinar simultaneamente o menor e o maior elemento do conjunto.

É possível encontrar os elementos procurados fazendo menos que  $2(n-1)$  comparações?

R: Sim, usando divisão e conquista.

Se  $n = 1$ , o elemento  $s_1$  é simultaneamente o menor e o maior elemento, sem precisar comparar;

Senão se  $n = 2$ , com uma comparação se descobre qual dos dois elementos é menor e qual o maior;

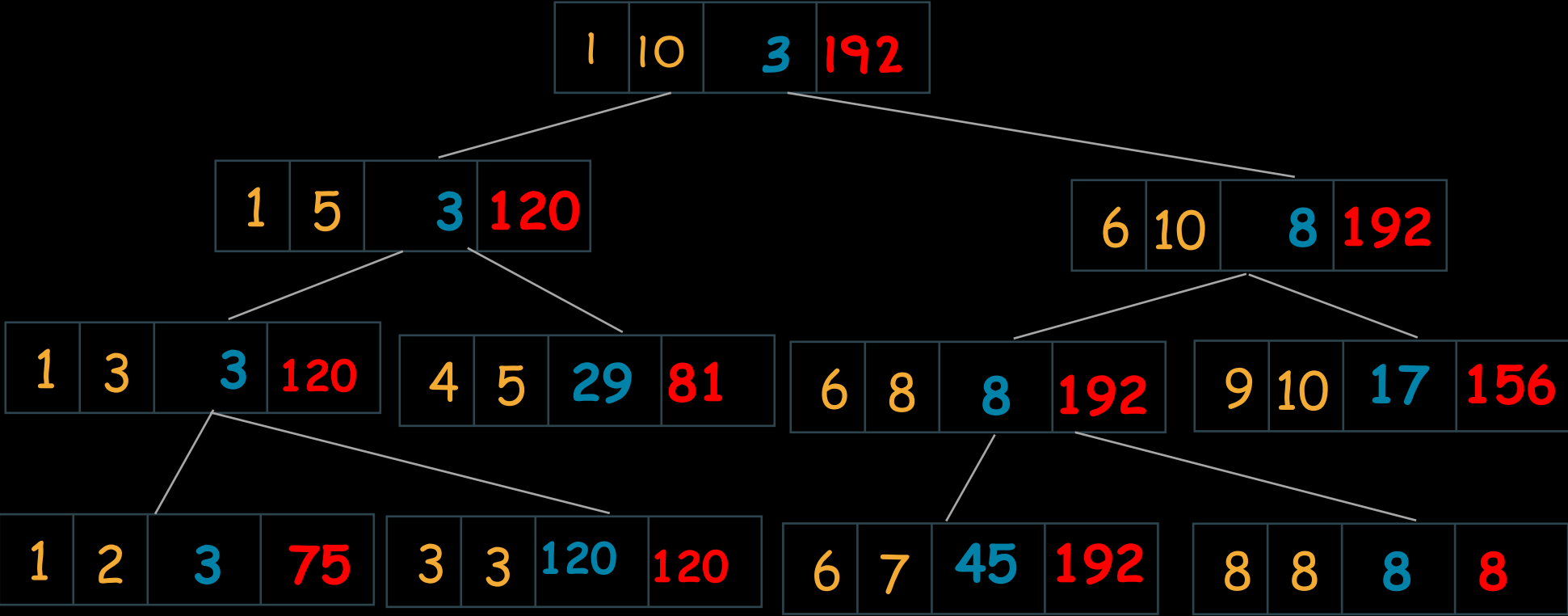
Senão, encontrar  $(a_1, b_1)$ , mínimo e máximo da primeira metade,  $(a_2, b_2)$ , mínimo e máximo da segunda metade e retornar  $(\min(a_1, a_2), \max(b_1, b_2))$ .

# Máximo e Mínimo - Enfoque Recursivo

```
Maxmin(int e, int d):  
    se e = d:  
        retornar (s[e], s[e])  
    senão  
        se d = e+1:  
            se s[e] < s[d]:  
                retornar (s[e], s[d])  
            senão:  
                retornar (s[d], s[e])  
        senão:  
            m ← ⌊(e+d)/2⌋  
            (a1, b1) ← MaxMin(e, m)  
            (a2, b2) ← MaxMin(m+1, d)  
            retornar (min(a1, a2), max (b1, b2))
```

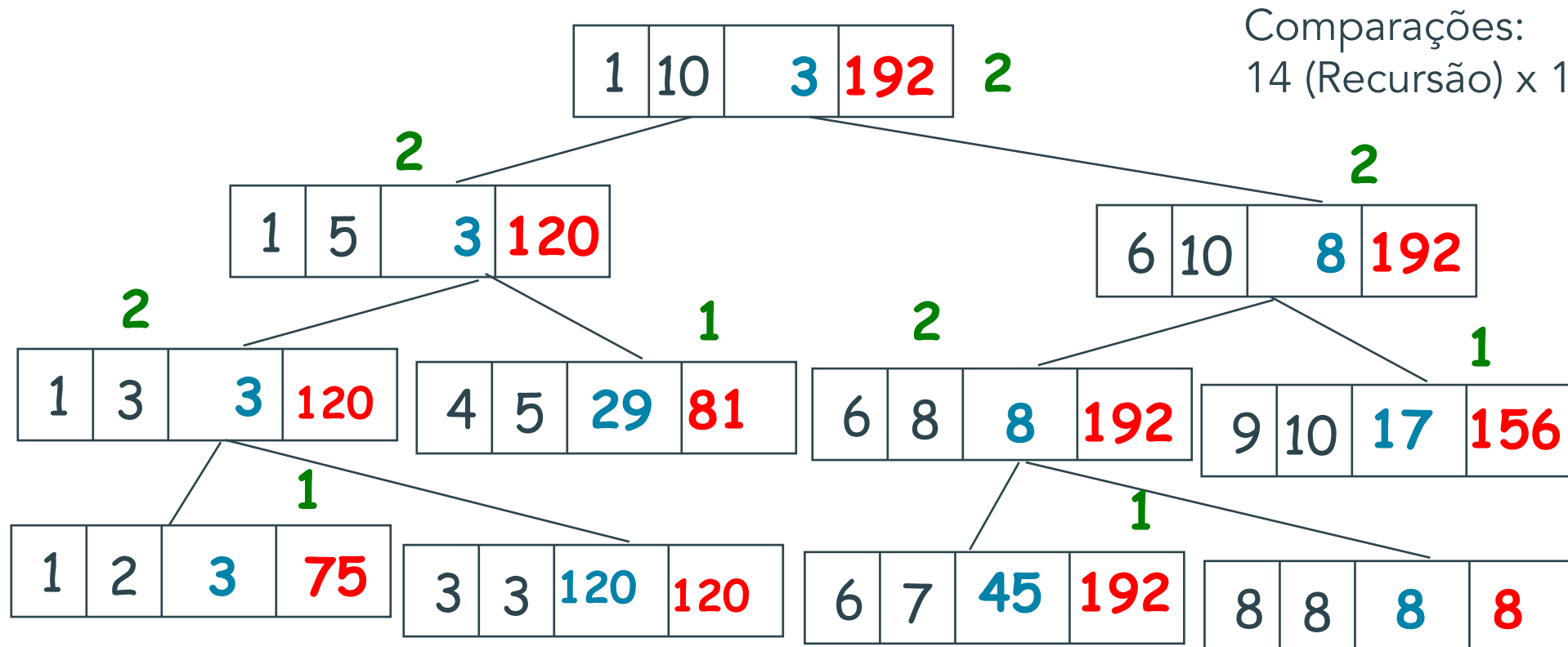
# Máximo e Mínimo - Exemplo

|    |   |     |    |    |     |    |   |     |    |
|----|---|-----|----|----|-----|----|---|-----|----|
| 1  | 2 | 3   | 4  | 5  | 6   | 7  | 8 | 9   | 10 |
| 75 | 3 | 120 | 81 | 29 | 192 | 45 | 8 | 156 | 17 |



# Máximo e Mínimo - Comparações do exemplo

| 1  | 2 | 3   | 4  | 5  | 6   | 7  | 8 | 9   | 10 |
|----|---|-----|----|----|-----|----|---|-----|----|
| 75 | 3 | 120 | 81 | 29 | 192 | 45 | 8 | 156 | 17 |



Comparações:  
14 (Recursão) x 18 (2 Loops)

# Máximo e Mínimo

É possível encontrar os números procurados fazendo menos que  $2(n-1)$  comparações?

Sim, prova-se que o mínimo de comparações possível é  $3n/2-2$ , que é o resultado com divisão e conquista.

# Máximo e Mínimo - Enfoque Não Recursivo

O processo de divisão e conquista permite observar que a comparação inicial feita nas folhas entre dois elementos classifica cada elemento como ou candidato a mínimo ou máximo.

Pode-se, então, criar o seguinte algoritmo iterativo:

- a) inicialmente varre-se o vetor, comparando os elementos 2 a 2, classificando um deles como candidato a mínimo e o outro como candidato a máximo.
- b) Depois obtém-se mínimo e máximo de forma independente, em cada grupo de candidatos.

A divisão e conquista é, então, uma etapa intermediária para a obtenção de um algoritmo iterativo eficiente.

# Máximo e Mínimo - Enfoque Não Recursivo

```
Maxmin(int n):  
    vmin ← V[n];  vmax ← V[n];  
    para i ← 1.. $\lfloor n/2 \rfloor$  incl.:  
        se  $V[2i-1] < V[2i]$ :  
            vmin ← min(V[2i-1], vmin)  
            vmax ← max(V[2i], vmax)  
        senão:  
            vmin ← min(V[2i], vmin)  
            vmax ← max(V[2i-1], vmax)  
  
    retornar (vmin, vmax)
```

**Número de  
comparações:  
s:**

O algoritmo  
faz  $3\lfloor n/2 \rfloor$   
comparações,  
praticamente  
o valor ótimo.

# Divisão e Conquista

Técnicas Complementares  
(Memorização)





# MEMORIZAÇÃO - Fibonacci

$\text{Fib}(n) = i$ , se  $i \leq 1$

$\text{Fib}(n) = \text{Fib}(n-1) + \text{Fib}(n-2)$ , se  $n > 1$

```
inteiro Fib(n):  
  se n < 2:  
    retornar n  
  senão  
    se T[n] = -1:  
      T[n] ← Fib(n-1) + Fib(n-2)  
    retornar T[n]
```

```
T[*] ← -1  
Fib(n)
```

Solução iterativa:

```
Fib():  
  T[0] ← 0;  
  T[1] ← 1;  
  para i ← 2..n incl.:  
    T[i] ← T[i-1] + T[i-2]
```

# MEMORIZAÇÃO

Técnica para fugir da complexidade exponencial, pelo tabelamento de soluções intermediárias.

Exemplo: Moedas

Dados  $m$  tipos de moedas e seus valores  $V[1]..V[m]$ , determinar quantas maneiras distintas existem para um troco de valor  $n$ .

No Brasil,  $m = 6$  e  $V = [1, 5, 10, 25, 50, 100]$ .

Temos 4 maneiras distintas de dar um troco de 11 centavos:

$1+1+1...+1$      $1+1+1+1+1+1+5$

$1+5+5$

$1+10$

# MEMORIZAÇÃO - Moedas

$T(p, n)$  = número de trocos distintos para  $n$ , usando as moedas de tipos 1 a  $p$ .

Recorrência para calcular  $T(p, n)$ :

$T(p, n) = 0$ , se  $n < 0$ , ou  $p=0$

$T(p, 0) = 1$

$T(p, n) = T(p, n - V[p]) + T(p-1, n)$ , se  $n > 0$

Explicação da recorrência:

$T(p, n) = 0$ , se  $n < 0$ , ou  $p=0$ : não tem como dar um troco para valor negativo ou com tipo de moeda inexistente.

$T(p, 0) = 1$ : a única maneira aqui é não dar troco algum.

$T(p, n) = T(p, n - V[p]) + T(p-1, n)$ , se  $n > 0$ : contamos dois casos: ou usa a moeda do tipo  $p$  ou não.

# MEMORIZAÇÃO - Moedas

$T(p, n)$  = número de trocos distintos para  $n$ , usando as moedas de tipos 1 a  $p$ .

$T(p, n) = 0$ , se  $n < 0$ , ou  $p=0$

$T(p, 0) = 1$

$T(p, n) = T(p, n - V[p]) + T(p-1, n)$ , se  $n > 0$

Temos 6 maneiras de dar um troco de 16 centavos:

1+1+1...+1

1+...+1+5

1+...+1+5+5

1+5+5+5

1+...+1+10

1+5+10

$$T(3, 16) = T(3, 16-10) + T(2, 16) = 6$$

$T(3, 16-10)$  = todas as maneiras de dar um troco de 16 que usam uma moeda de 10 = 2

$T(2, 16)$  = todas as maneiras de dar um troco de 16 que só usam moedas inferiores a 10 = 4

$$\begin{aligned} T(6, 16) &= \text{todas as maneiras de dar um troco de 16 centavos} = \\ &T(6, 16-100) + T(5, 16-50) + T(4, 16-25) + T(3, 16-10) + T(2, 16) = \\ &0 + 0 + 0 + 2 + 4 = 6 \end{aligned}$$

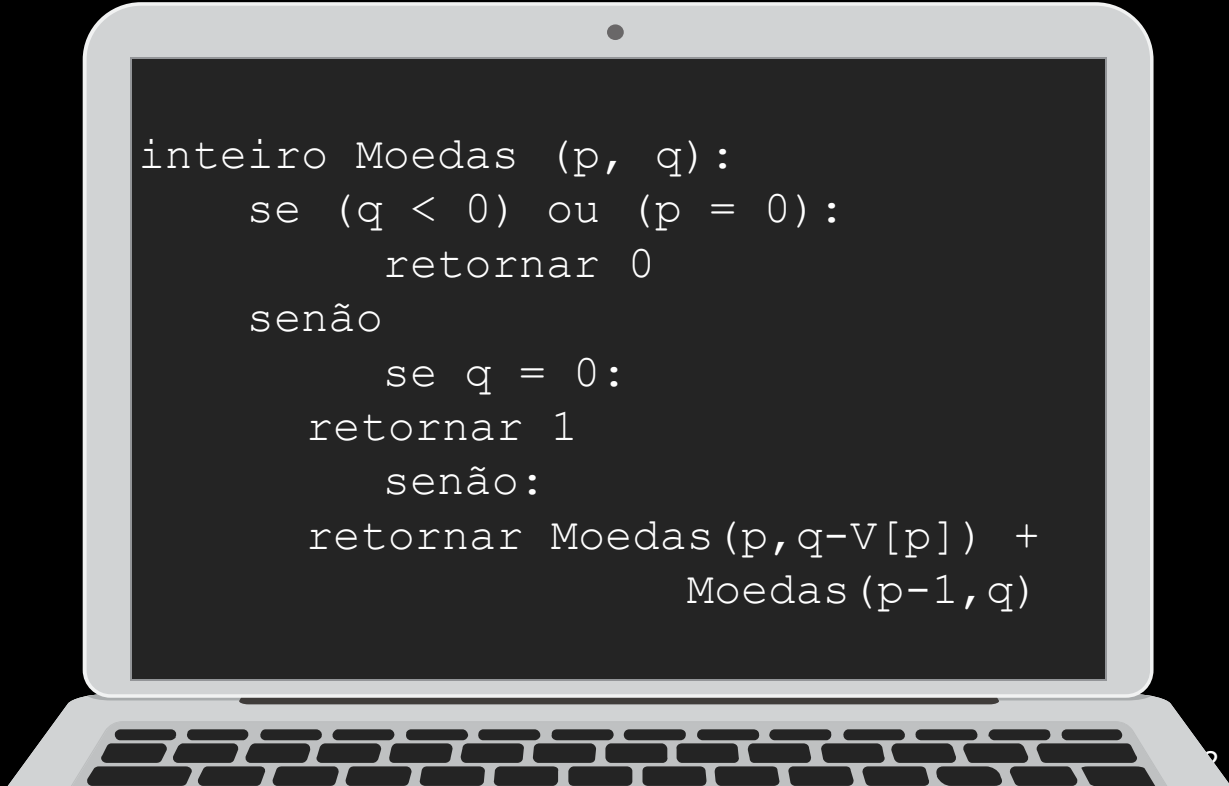
# MEMORIZAÇÃO - Moedas

$T(p, n)$  = número de trocos distintos para  $n$ , usando as moedas de tipos 1 a  $p$ .

$T(p, n) = 0$ , se  $n < 0$ , ou  $p=0$

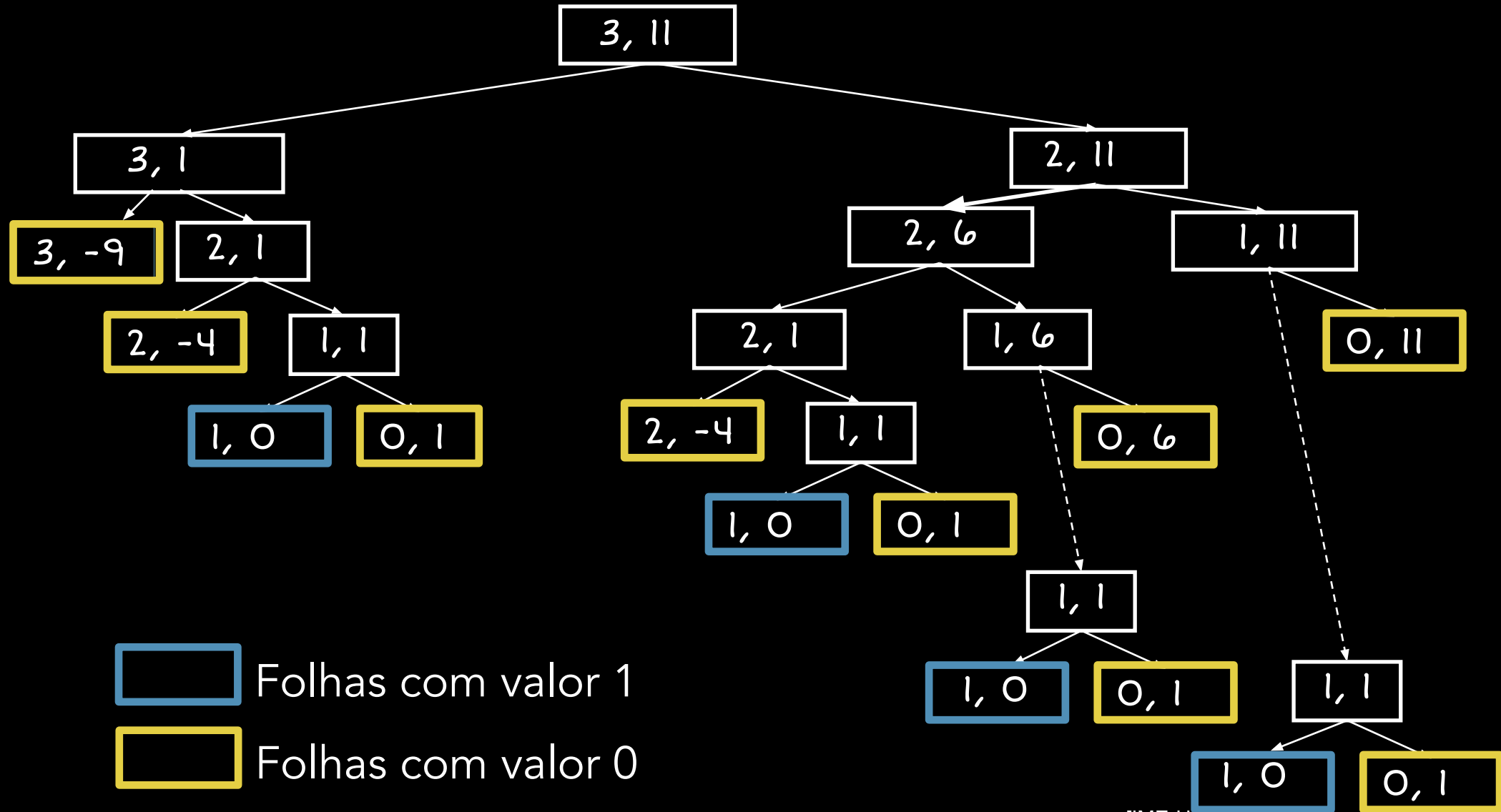
$T(p, 0) = 1$ ;

$T(p, n) = T(p, n - V[p]) + T(p-1, n)$



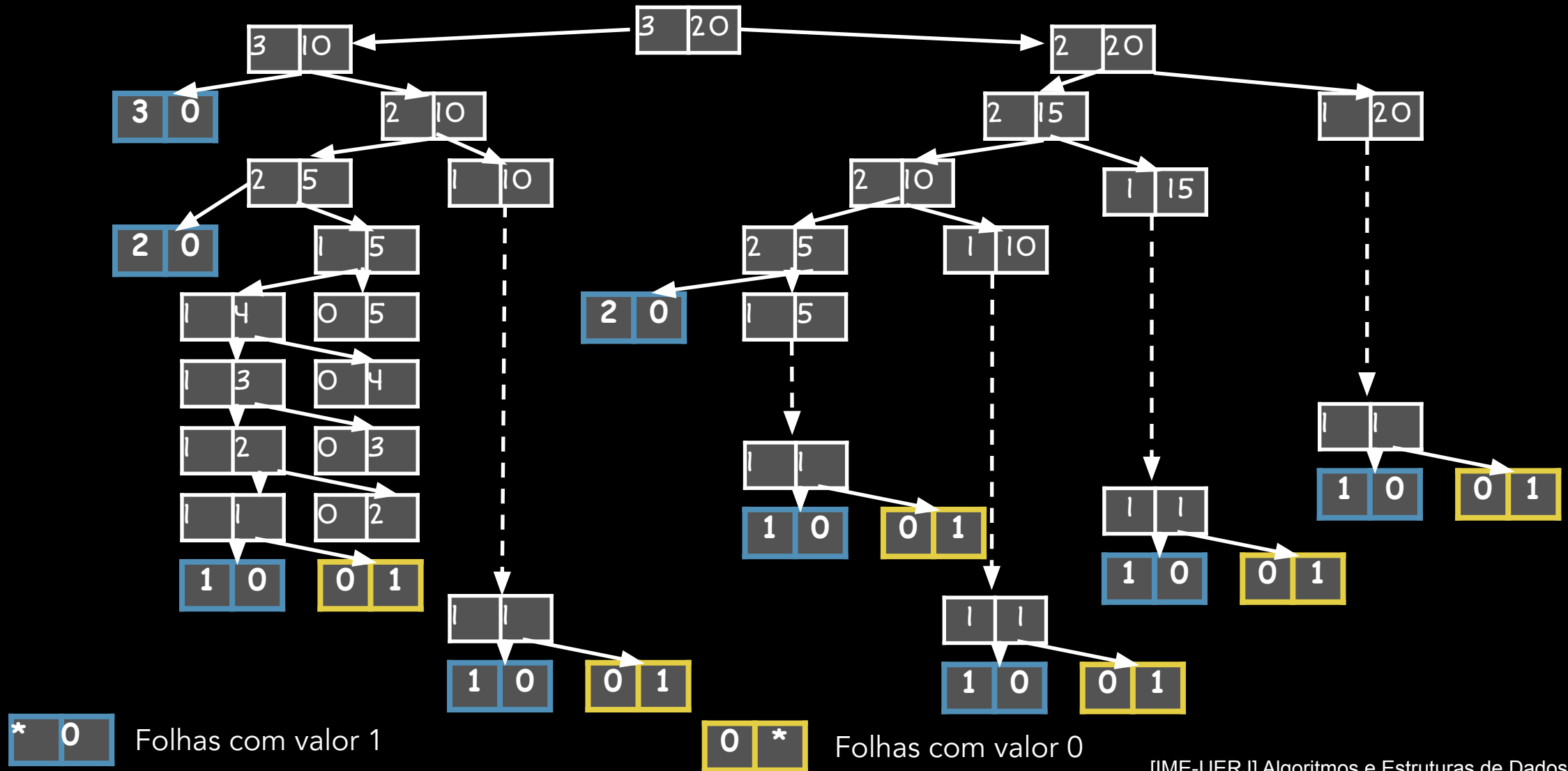
```
inteiro Moedas (p, q):  
    se (q < 0) ou (p = 0):  
        retornar 0  
    senão  
        se q = 0:  
            retornar 1  
        senão:  
            retornar Moedas (p, q-V[p]) +  
                      Moedas (p-1, q)
```

# MEMORIZAÇÃO - Moedas T(3,11)

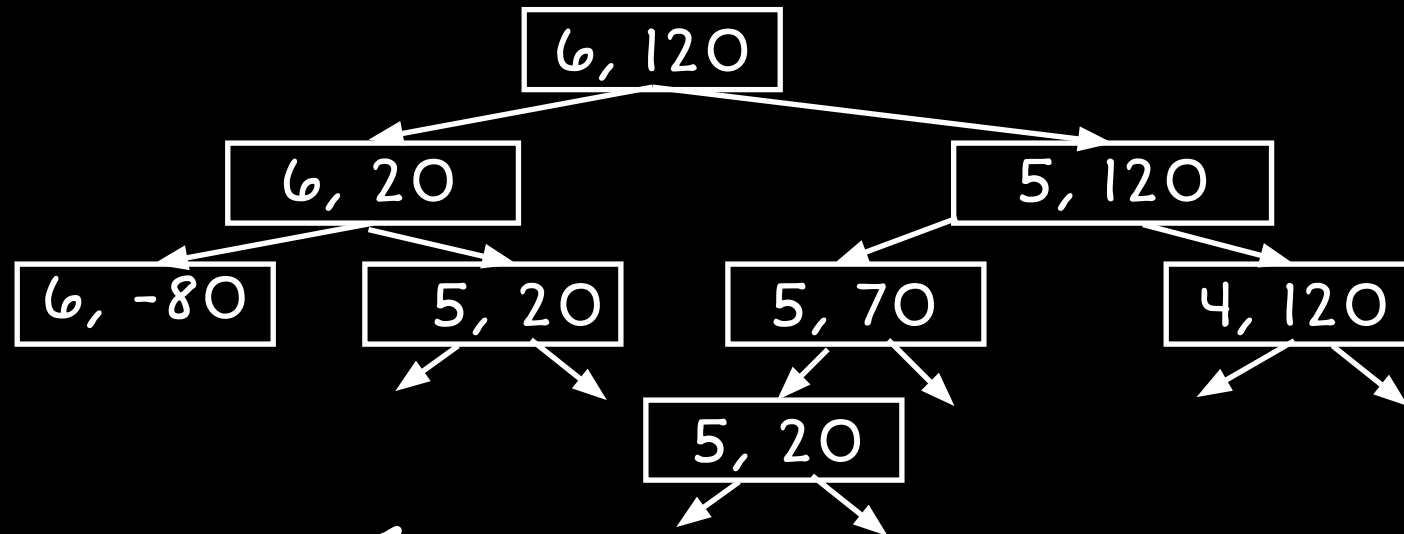


# SEM MEMORIZAÇÃO - Moedas

## T(3,20)



# MEMORIZAÇÃO - Moedas $n = 120$



Árvore de recursão  
exponencial!!!

## O que fazer?

Usar memorização que consiste em guardar resultados de sub-problemas resolvidos. Para o presente exemplo pode-se usar uma matriz  $6 \times n$ .



# MEMORIZAÇÃO - Moedas

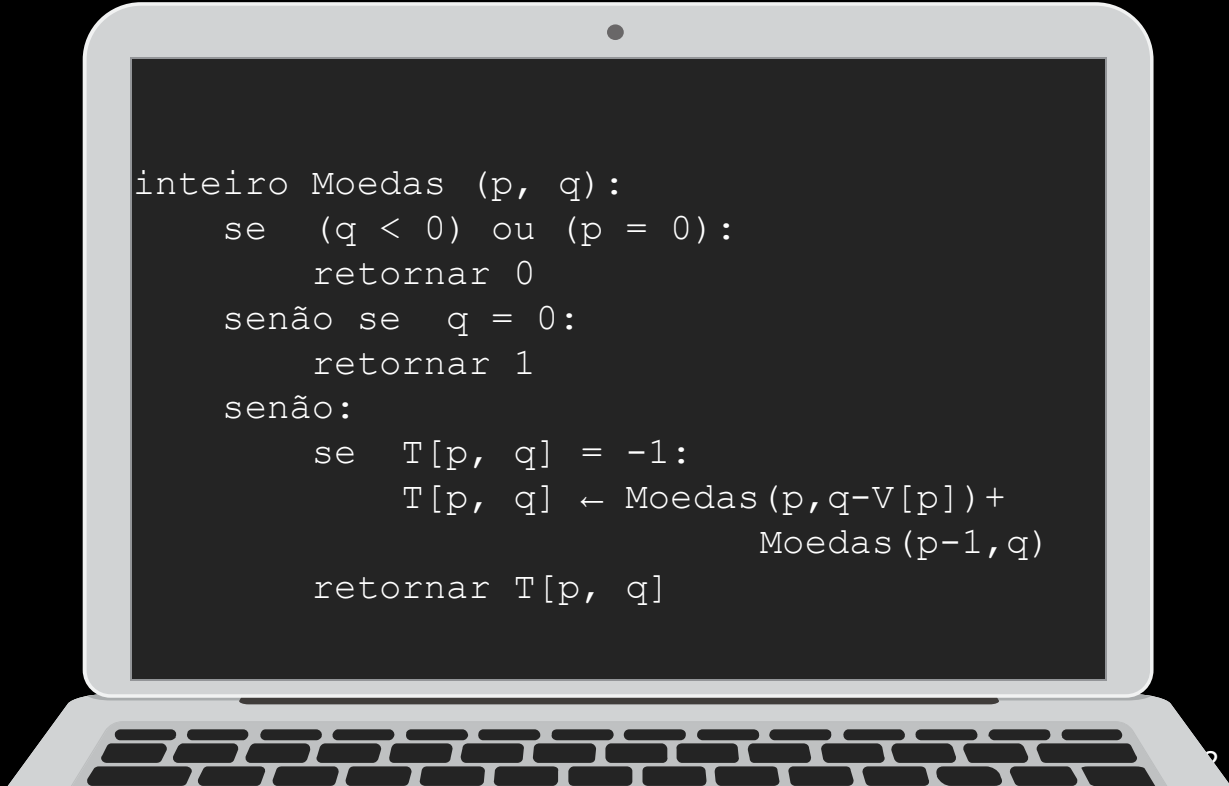
$T(p, n) = 0$ , se  $n < 0$  ou  $p = 0$ ;

$T(p, 0) = 1$ ;

$T(p, n) = T(p, n - V[p]) + T(p-1, n)$ , se  $n > 0$

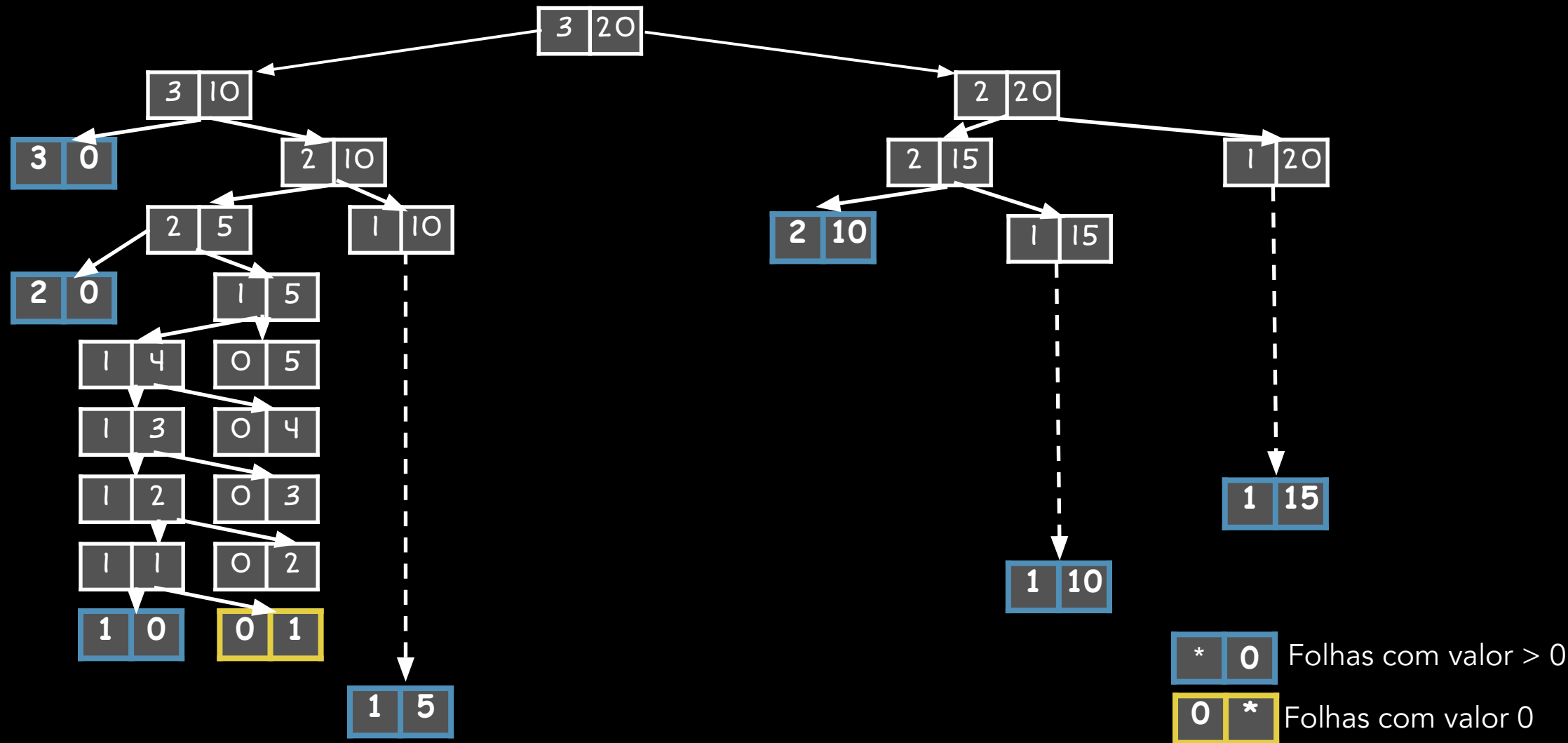
$T[*, *] \leftarrow -1$

Moedas (m, n)



```
inteiro Moedas (p, q):  
    se (q < 0) ou (p = 0):  
        retornar 0  
    senão se q = 0:  
        retornar 1  
    senão:  
        se T[p, q] = -1:  
            T[p, q] ← Moedas (p, q-V[p]) +  
                        Moedas (p-1, q)  
        retornar T[p, q]
```

# SEM MEMORIZAÇÃO - Moedas T(3,20)



# MEMORIZAÇÃO - Moedas

(Sub-problemas resolvidos)

$V = \{1, 5, 10, 25, 50, 100\}, n = 20$

|   | 0 | 1  | 2  | 3  | 4  | 5  | 6  | 7  | 8  | 9  | 10 | 11 | 12 | 13 | 14 | 15 | 16 | 17 | 18 | 19 | 20 |
|---|---|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|----|
| 1 | 1 | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  | 1  |
| 2 | 1 | -1 | -1 | -1 | -1 | 2  | -1 | -1 | -1 | -1 | 3  | -1 | -1 | -1 | -1 | 4  | -1 | -1 | -1 | -1 | 5  |
| 3 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 4  | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 9  |
| 4 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 9  |
| 5 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 9  |
| 6 | 1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | -1 | 9  |

$$T[6, 20] \leftarrow T[6, -80] + T[5, 20];$$

$$T[5, 20] \leftarrow T[5, -30] + T[4, 20];$$

$$T[4, 20] \leftarrow T[5, -5] + T[3, 20];$$

$$T[3, 20] \leftarrow T[3, 10] + T[2, 20];$$

$$T[3, 10] \leftarrow T[3, 0] + T[2, 10];$$

$$T[2, 10] \leftarrow T[2, 5] + T[1, 10];$$

$$T[2, 5] \leftarrow T[2, 0] + T[1, 5];$$

$$T[1, 5] \leftarrow T[1, 4] + T[0, 5];$$

...

# Dúvidas?

lucila.bento [at] ime.uerj.br