

Longest Common Subsequence

Algoritmo LCS



Equipe: Eduardo Braga, Henrique Franca, Isabela Medeiros, Júlia Vilela, Rafael Viana

O que é LCS?

Longest Common Subsequence
(Maior Subsequência Comum)

O objetivo desse algoritmo é encontrar a sequência mais longa **comum a duas strings**, mas os caracteres nessa sequência não precisam estar próximos uns dos outros. Eles só precisam aparecer na **mesma ordem**.



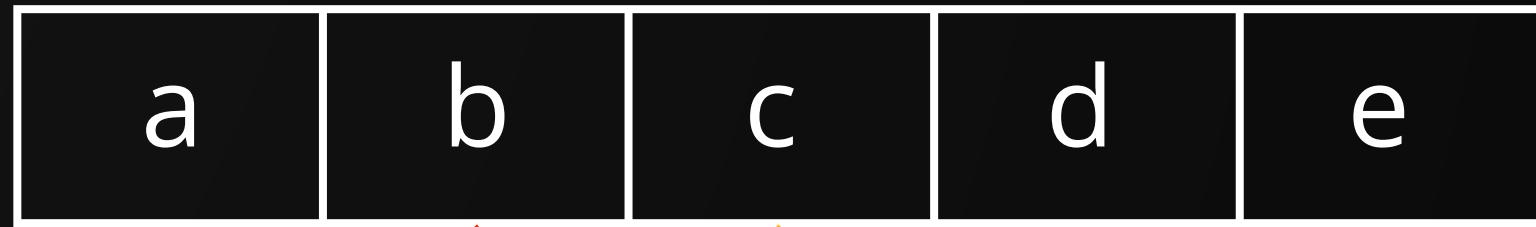
String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

String 1



String 2



Aparecem em **ordem diferente**.
Não pode!

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

► ac

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

► ac

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

► ac

► c

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

► ac

► c

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

- ▶ ac
- ▶ c
- ▶ b

String 1

a	b	c	d	e
---	---	---	---	---

String 2

a	c	b
---	---	---

- ▶ ac (tamanho 2) → maior subsequência comum
- ▶ c (tamanho 1)
- ▶ b (tamanho 1)

Recursivo Simples

Algoritmo LCS (A, B, i, j)

Entrada: duas strings A e B a serem comparadas e seus índices i e j.

Saída: o tamanho de uma subsequência C comum à A e B.

```
se A[i] == '\0' ou B[j] == '\0' então
    retorne 0
se A[i] == B[j] então
    retorne 1 + LCS(A, B, i+1, j+1)
senão
    retorne max(LCS(A, B, i+1, j), LCS(A, B, i, j+1))
```

A[0], B[0]
b, a

A

b	d	\0
0	1	2

B

a	b	c	d	\0
0	1	2	3	4

(LCS(A, B, i+1, j), LCS(A, B, i, j+1))

A[0], B[0]
b, a

A[1], B[0]
d, a

A[0], B[1]
b, b

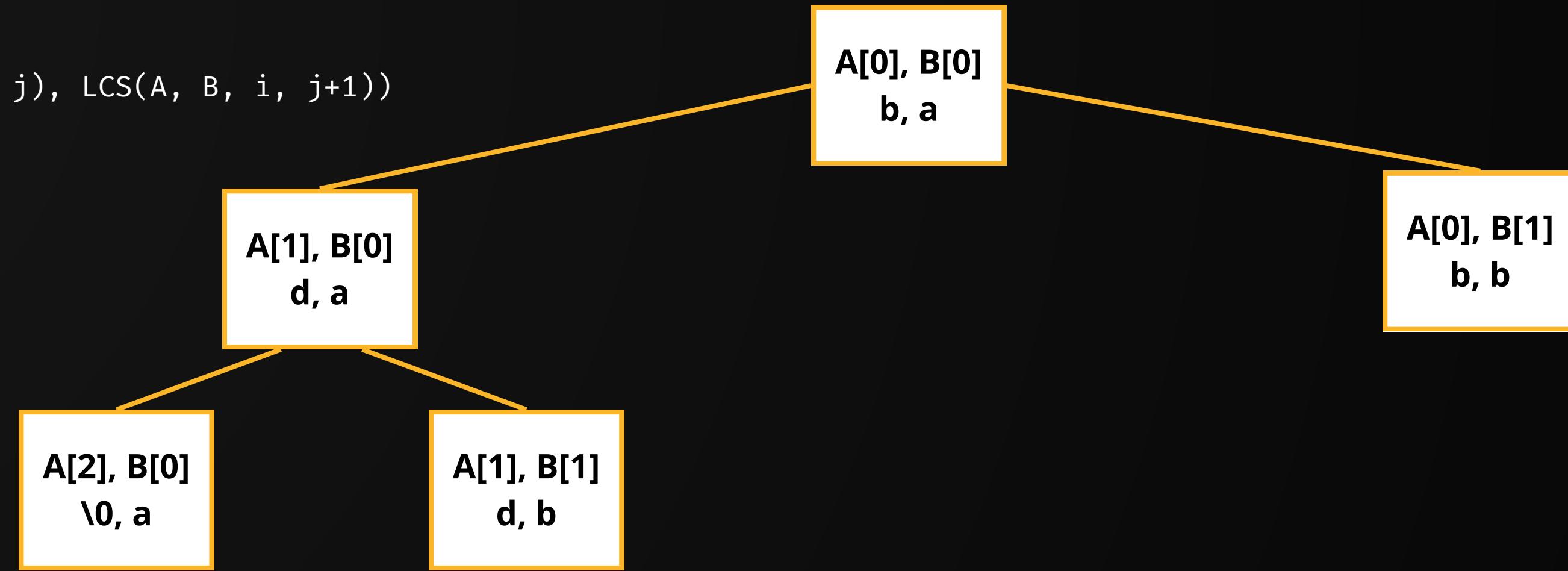
A

b	d	\0
0	1	2

B

a	b	c	d	\0
0	1	2	3	4

(LCS(A, B, i+1, j), LCS(A, B, i, j+1))



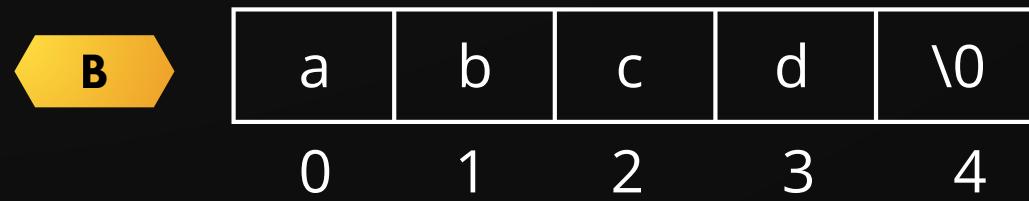
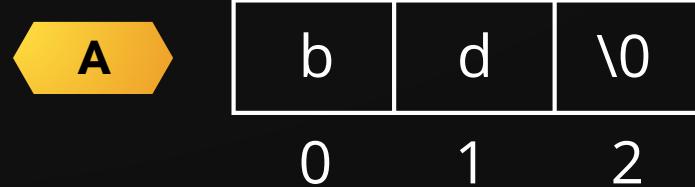
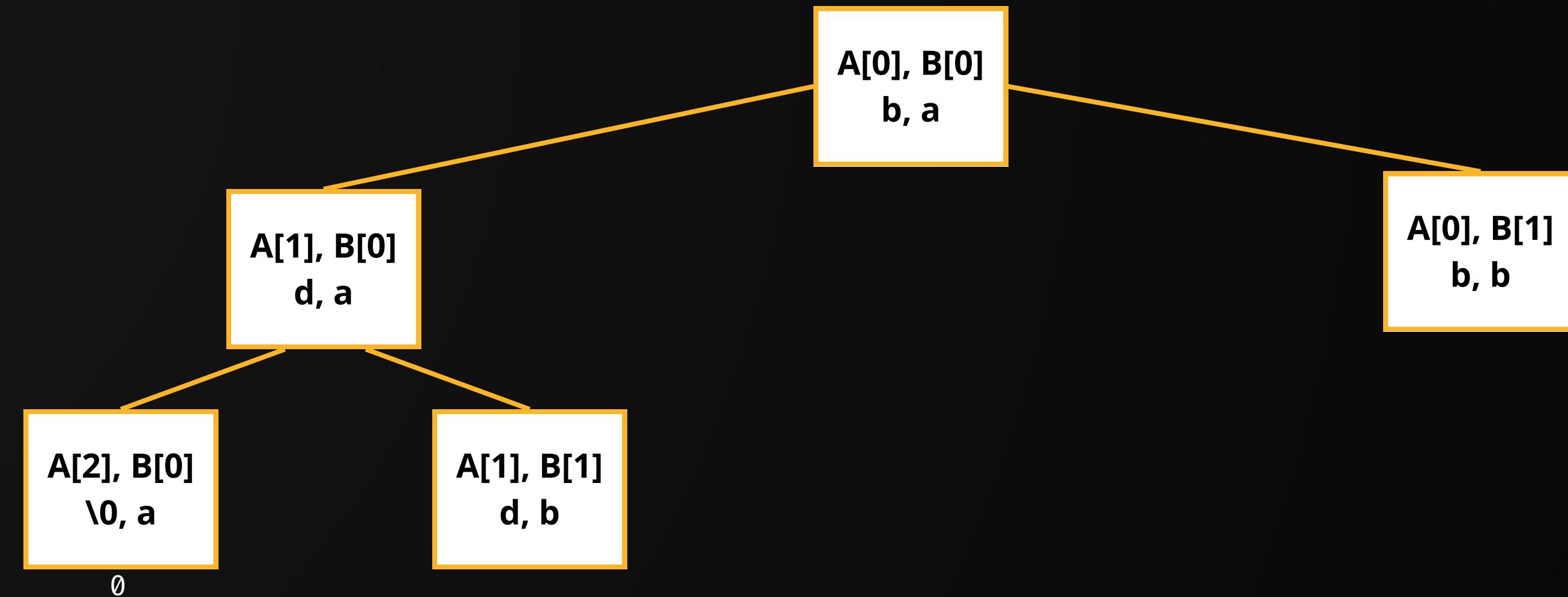
A

b	d	\0
0	1	2

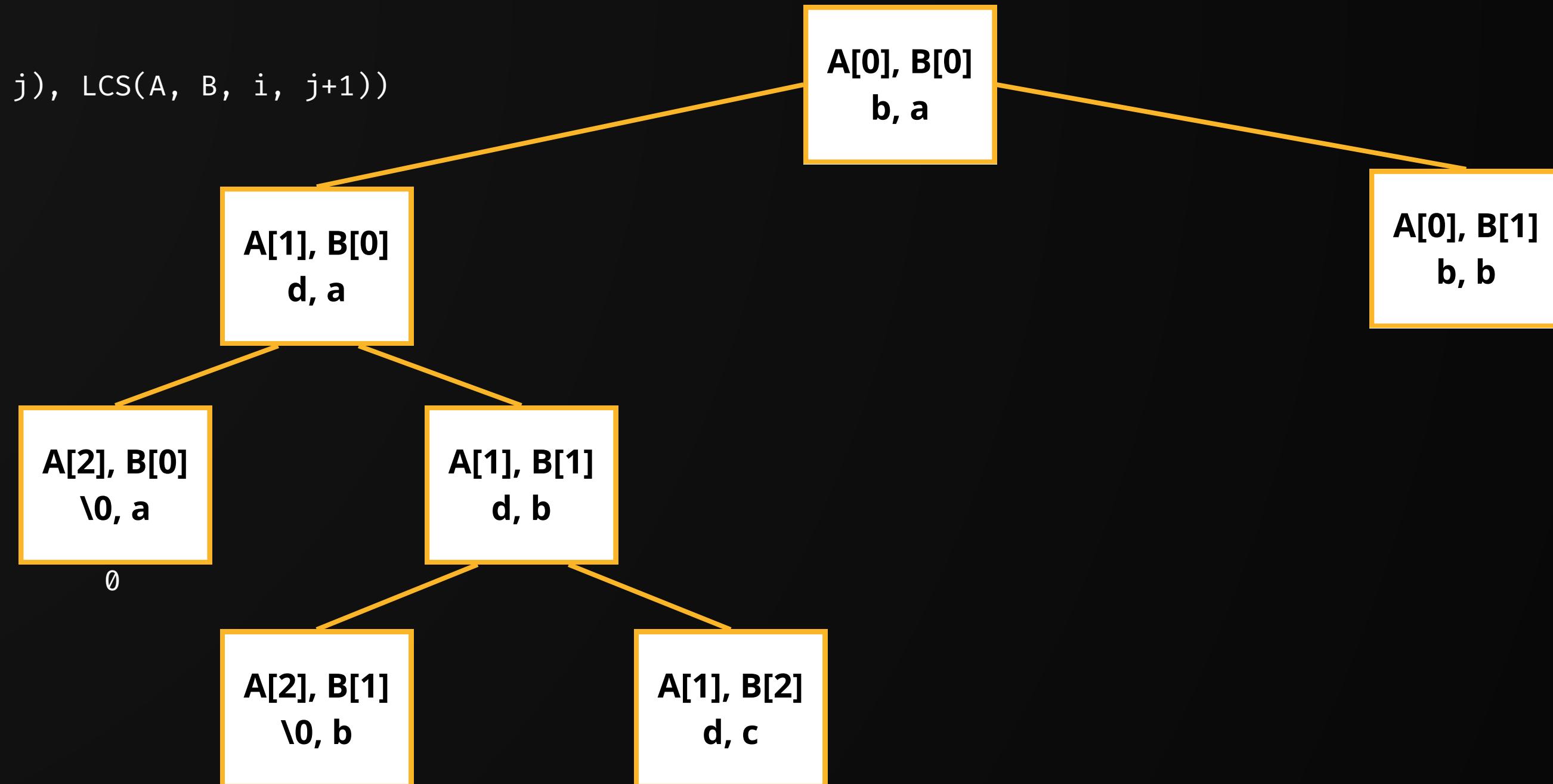
B

a	b	c	d	\0
0	1	2	3	4

retorne 0



(LCS(A, B, i+1, j), LCS(A, B, i, j+1))



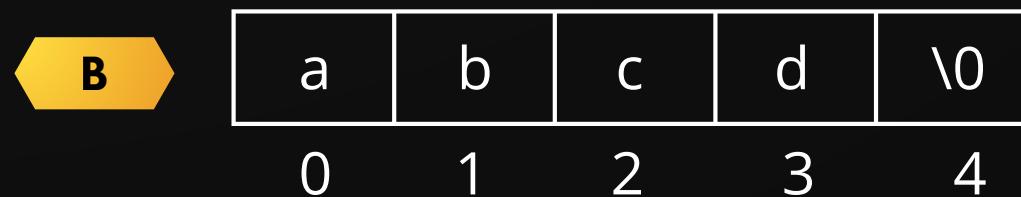
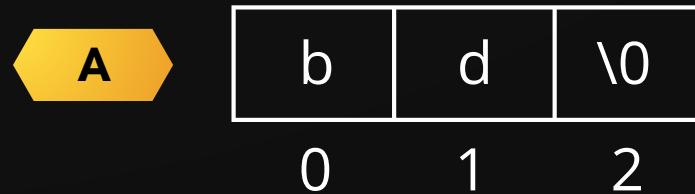
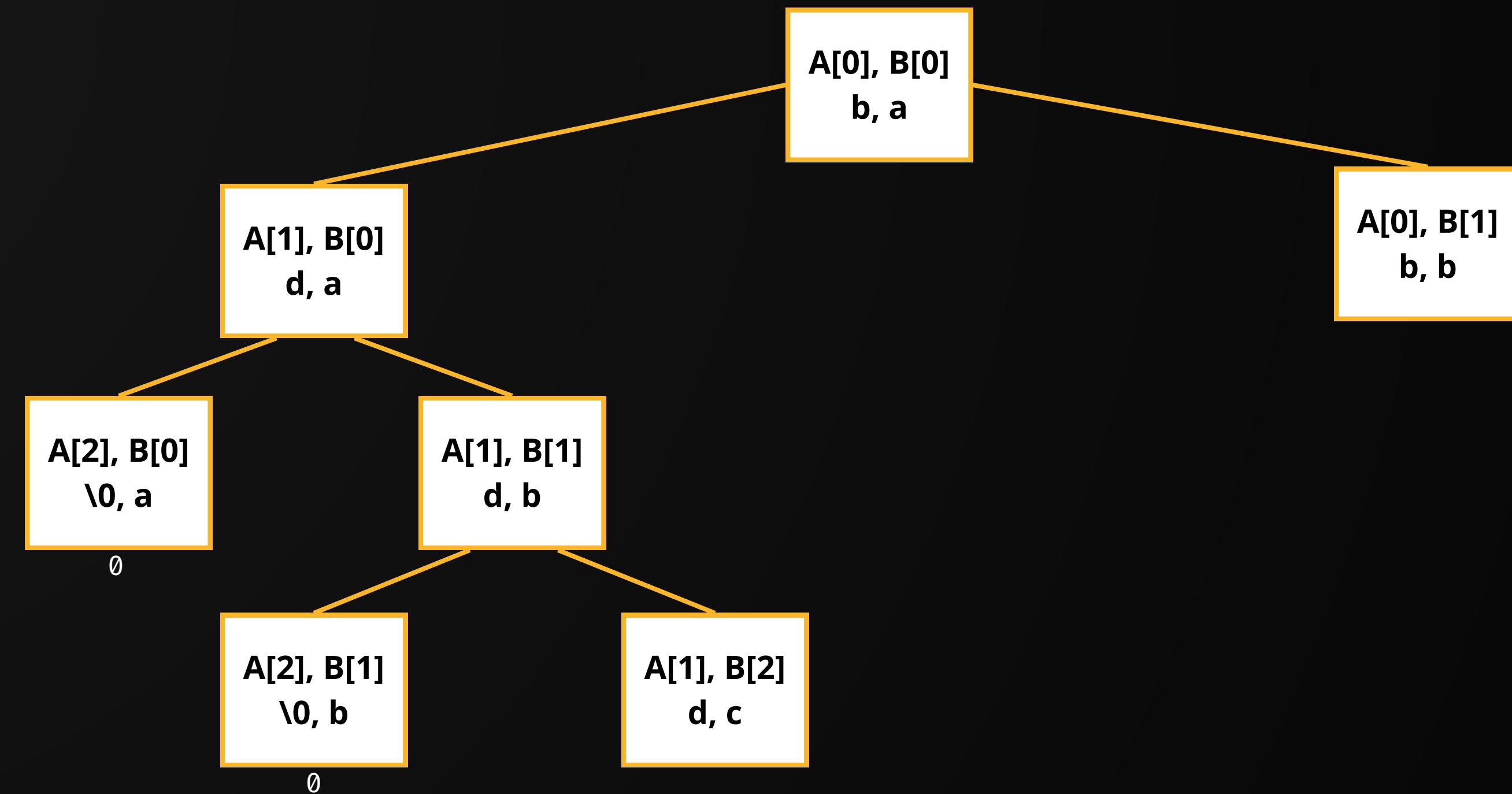
A

b	d	\0
0	1	2

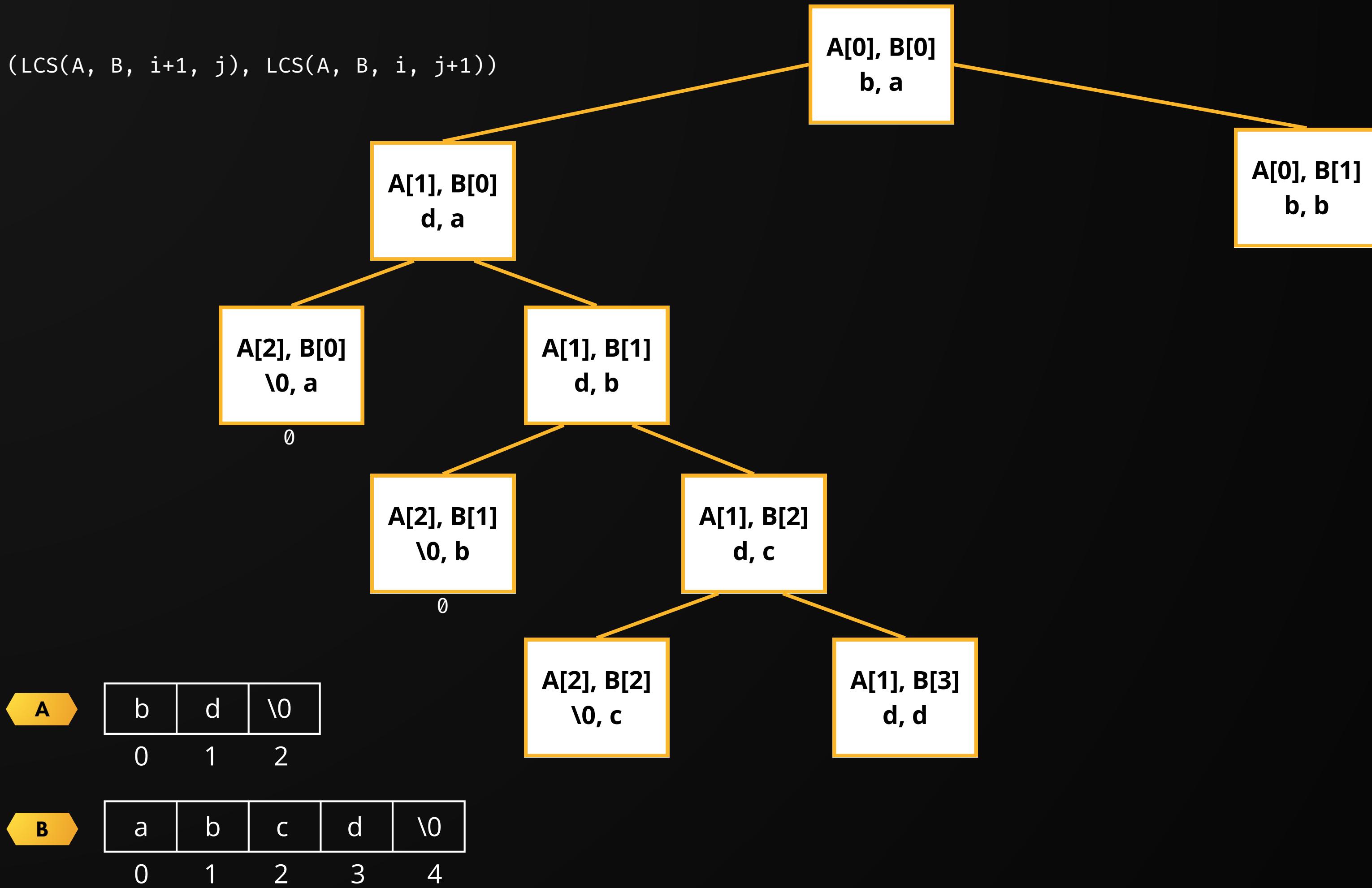
B

a	b	c	d	\0
0	1	2	3	4

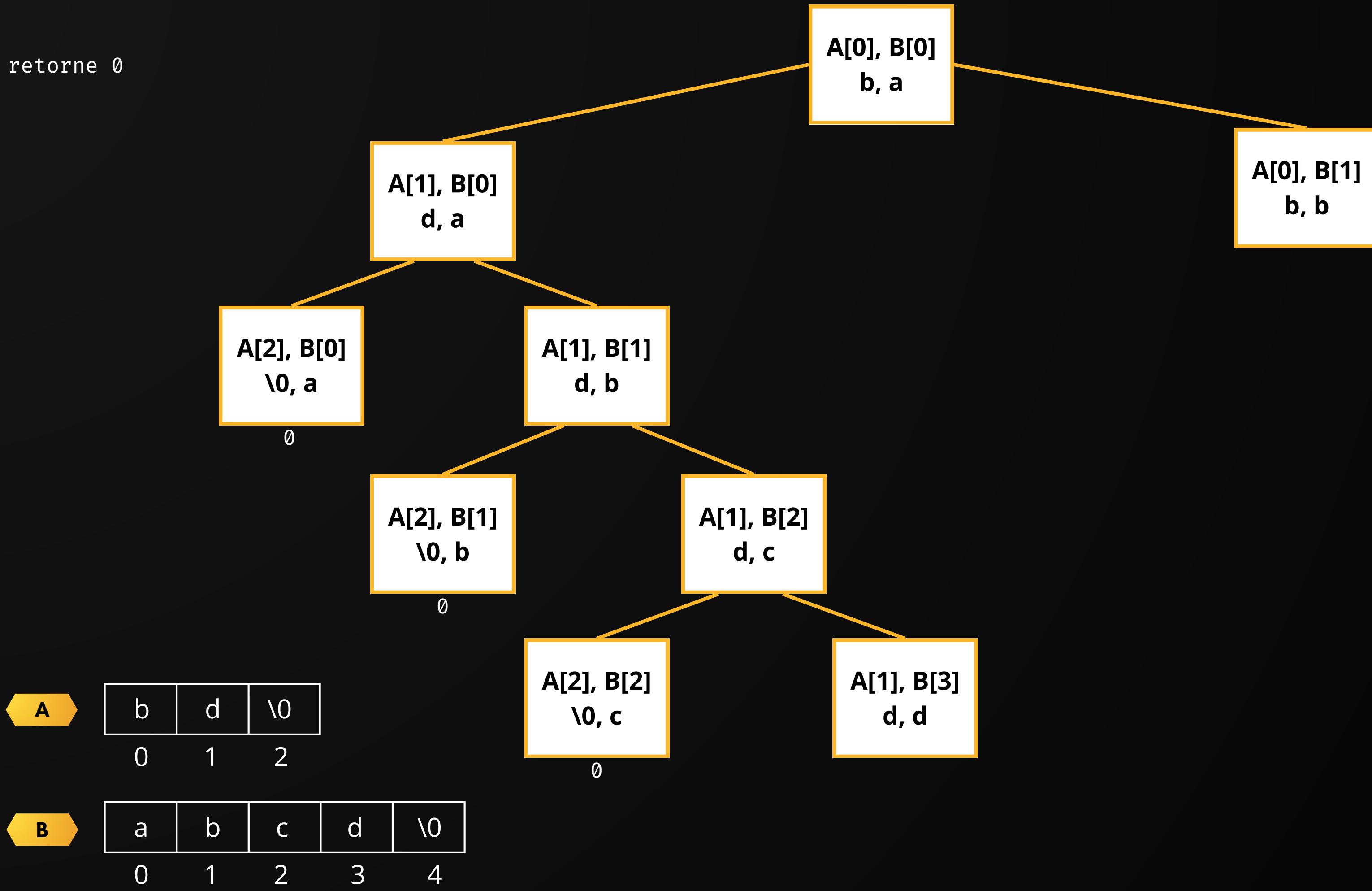
retorne 0



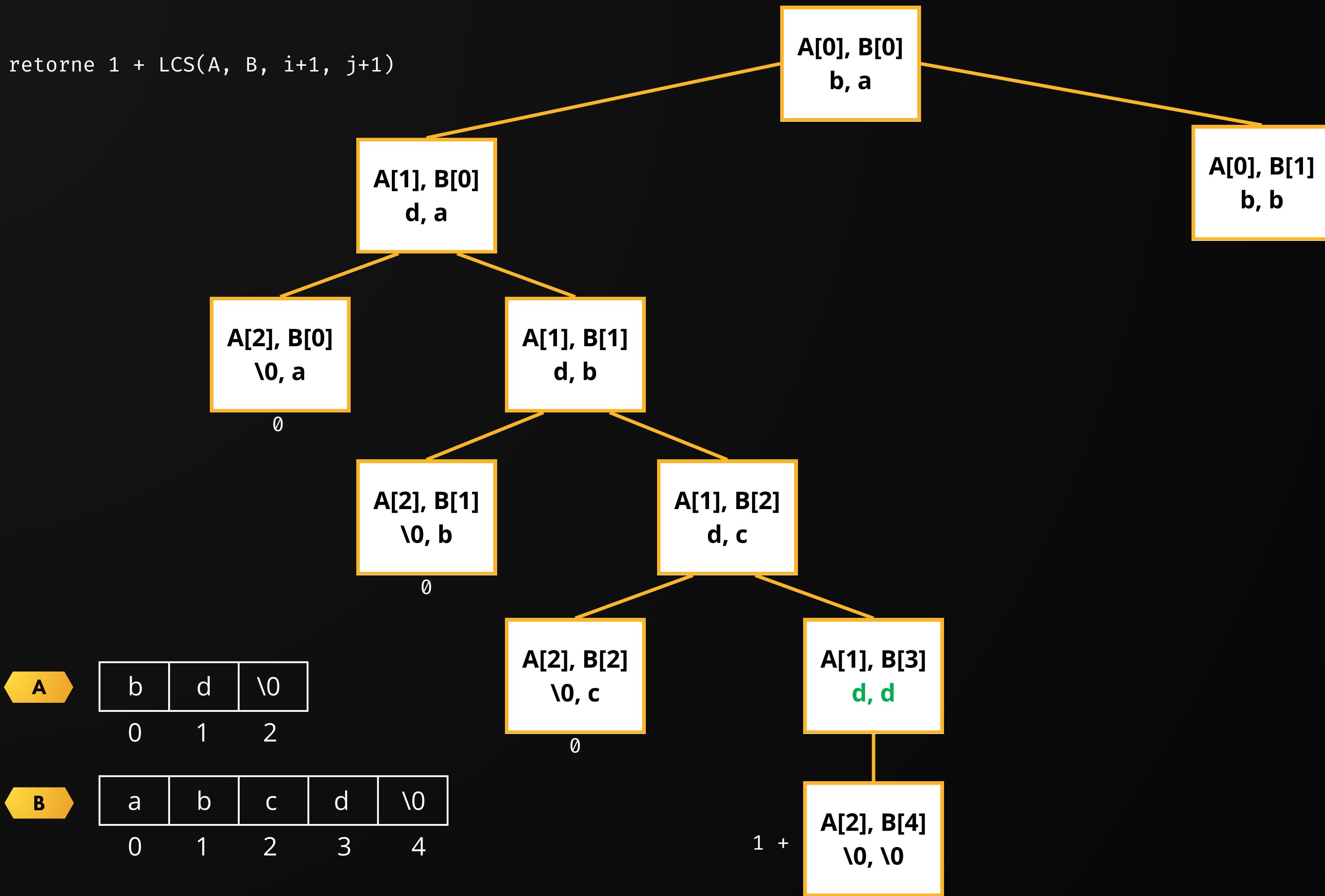
(LCS(A, B, i+1, j), LCS(A, B, i, j+1))



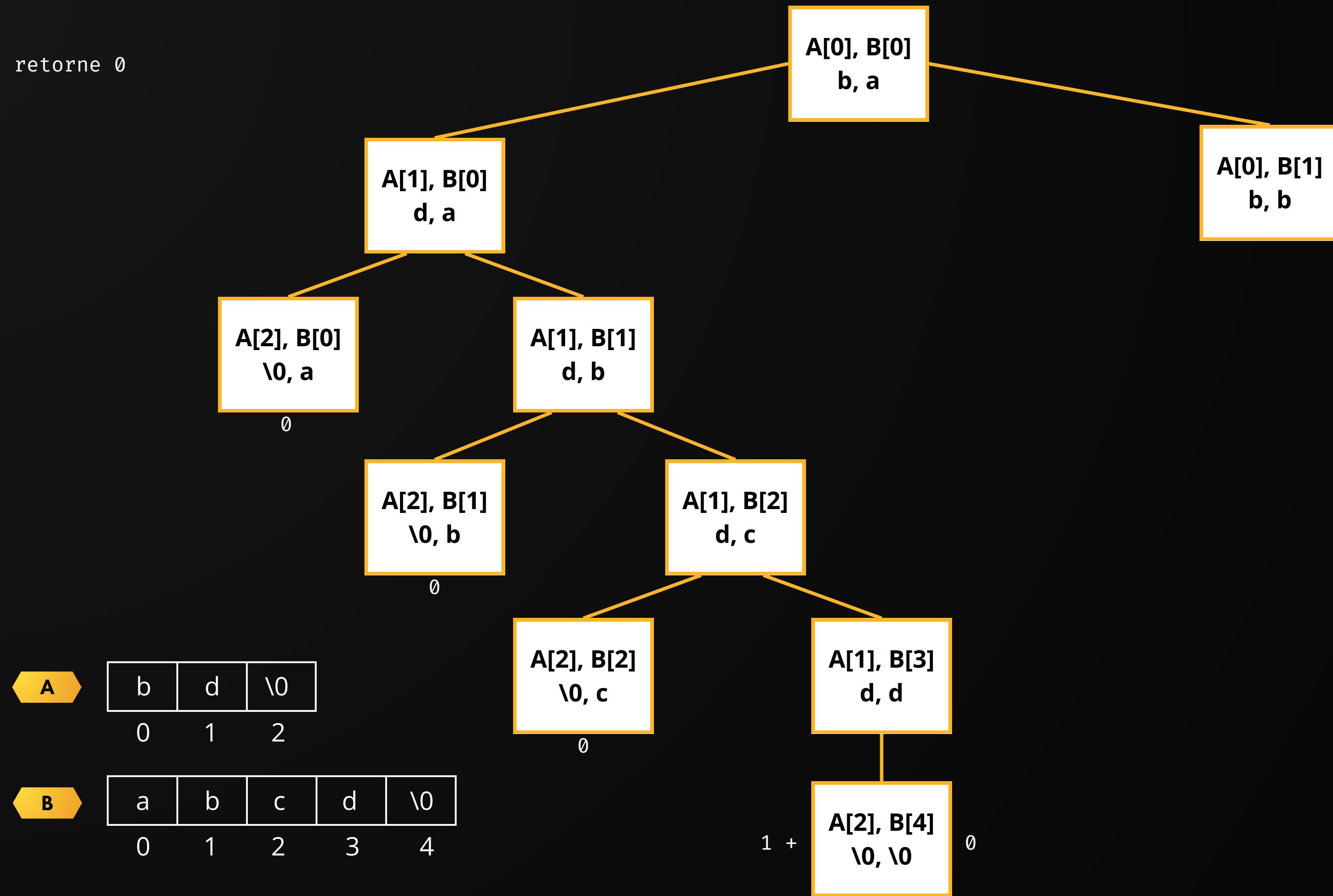
retorne 0

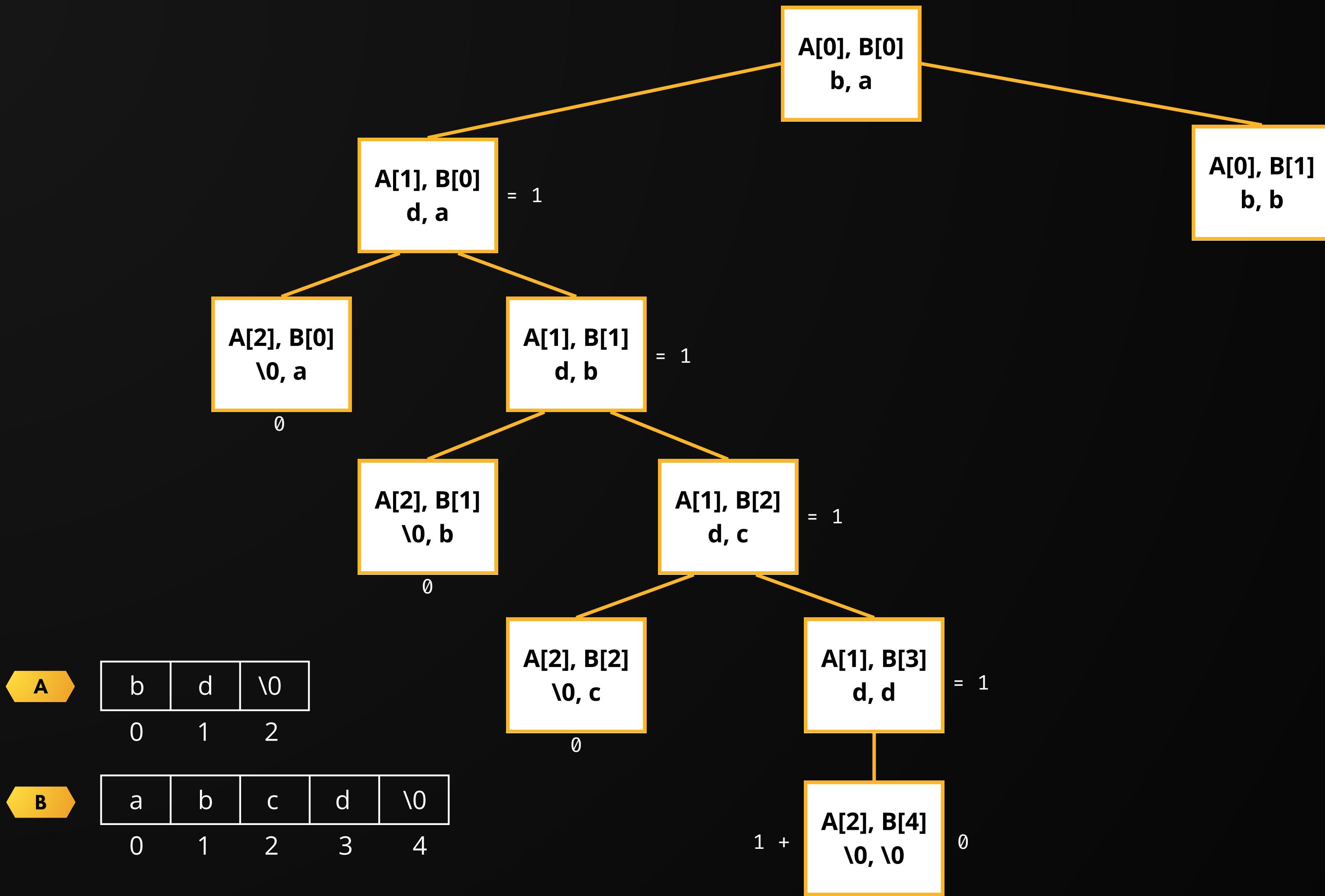


```
retorne 1 + LCS(A, B, i+1, j+1)
```

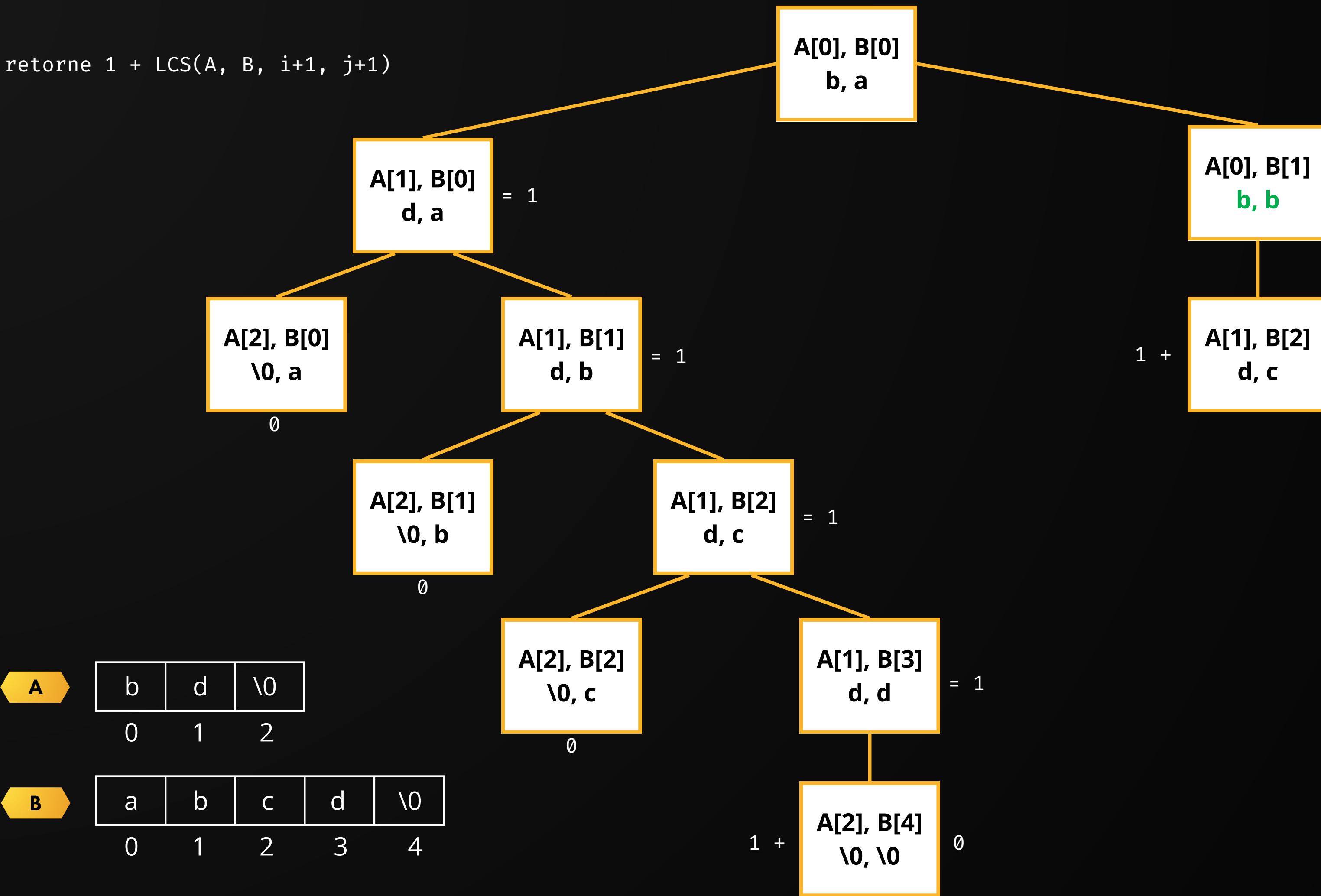


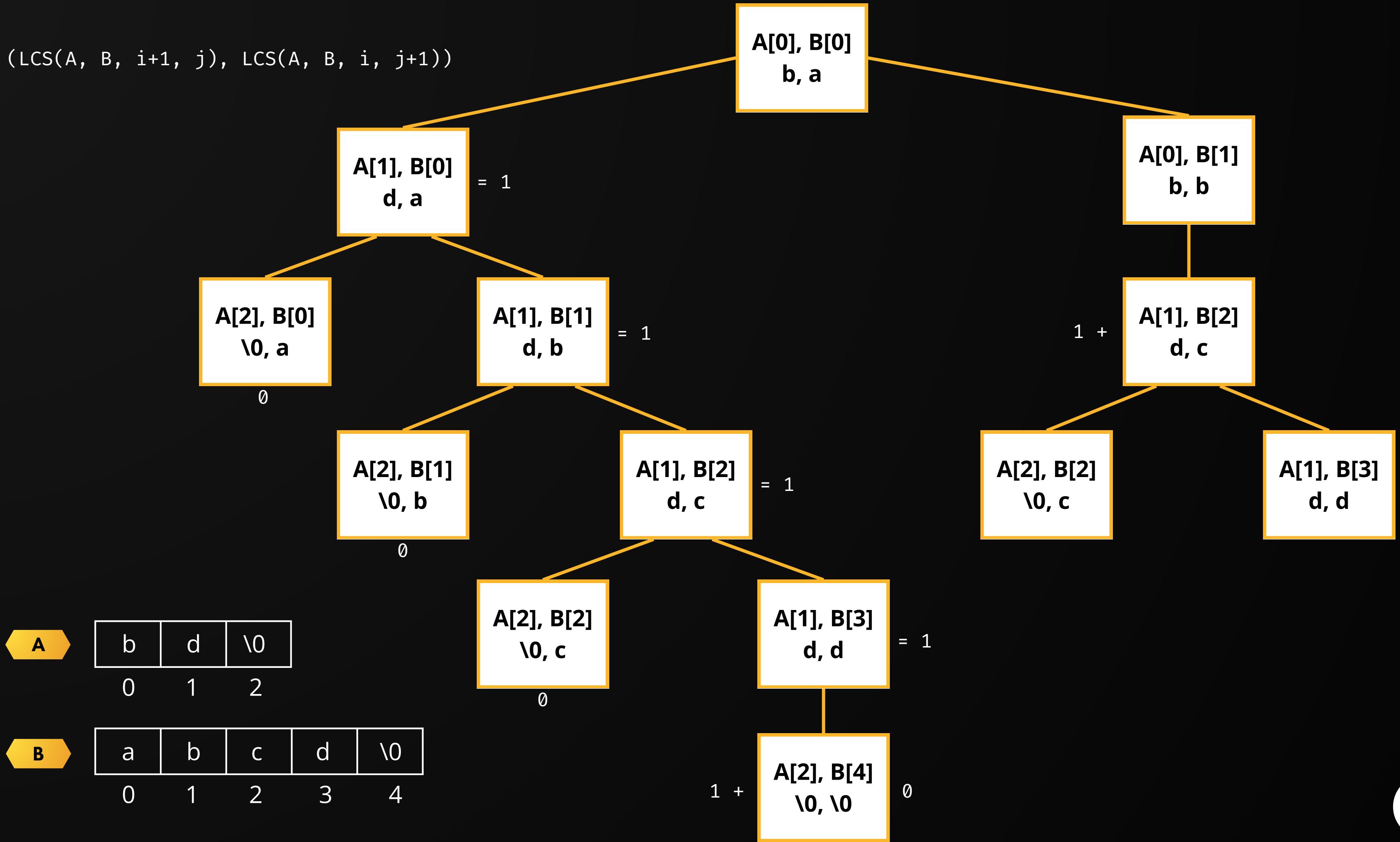
retorne 0



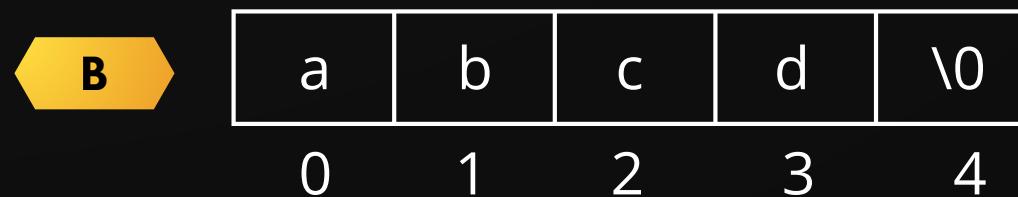
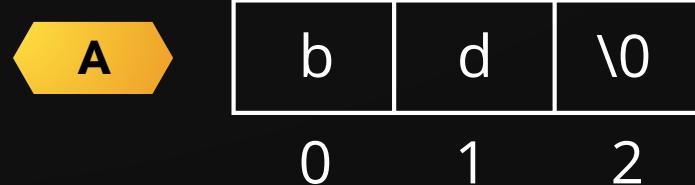
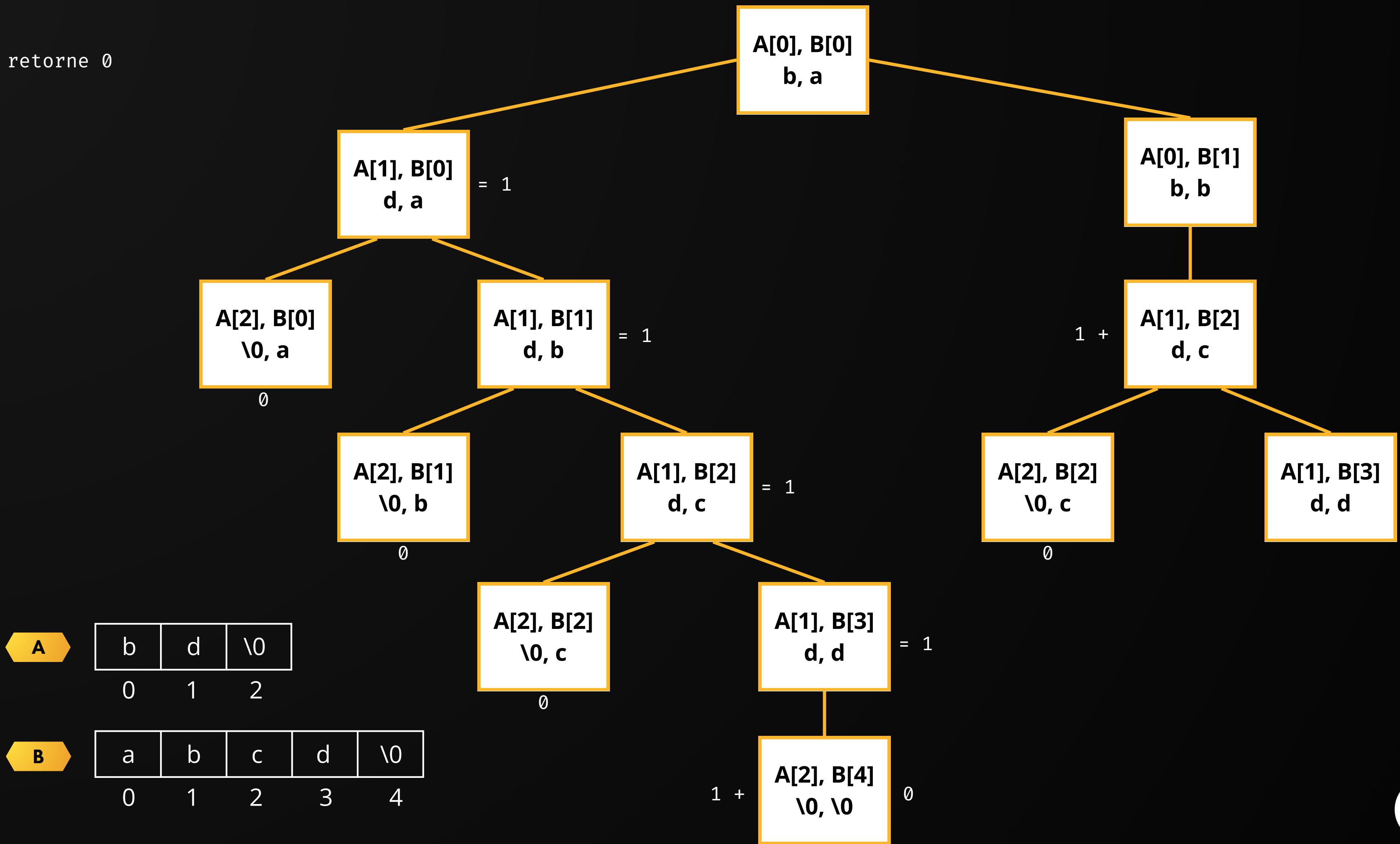


```
retorne 1 + LCS(A, B, i+1, j+1)
```

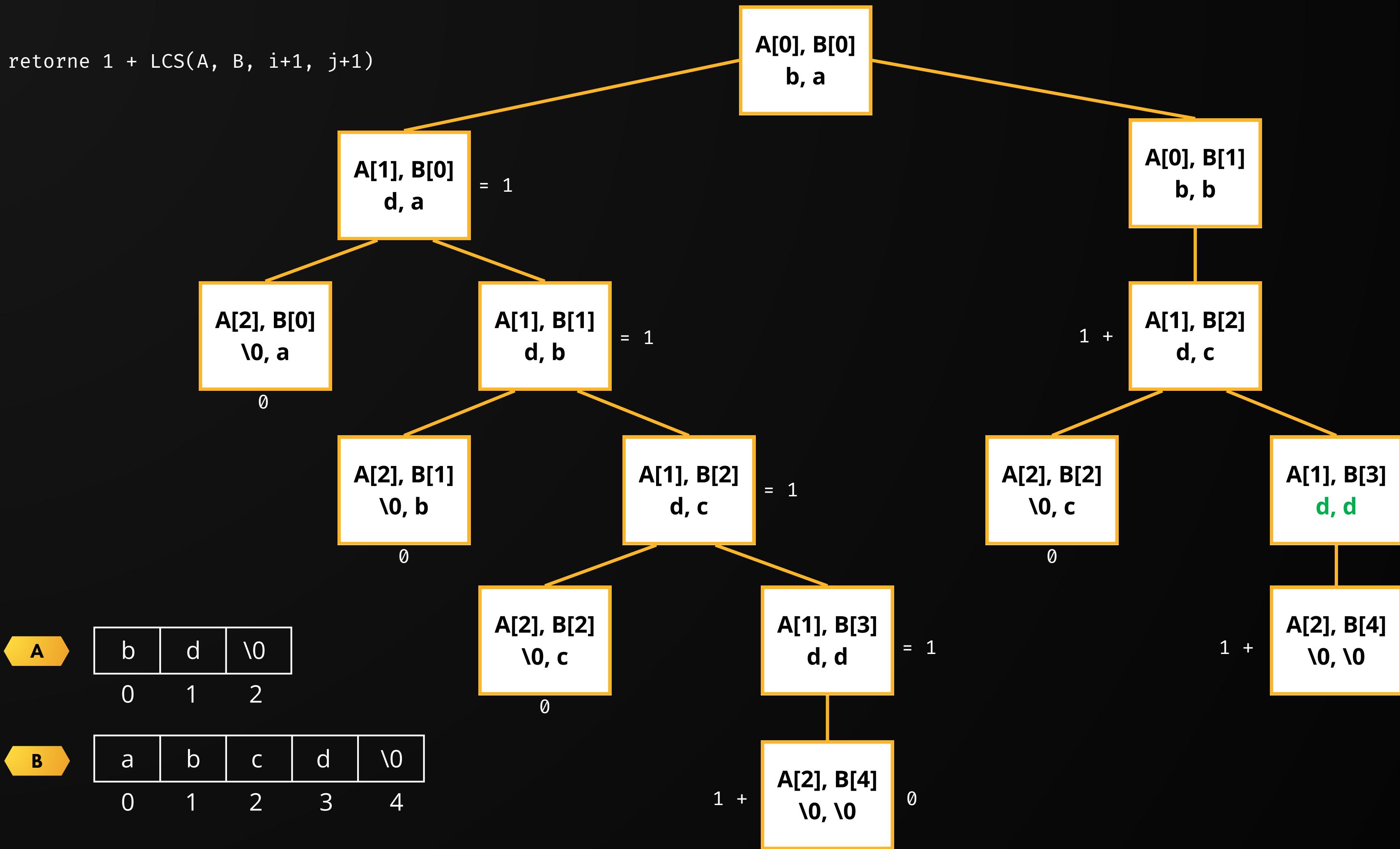




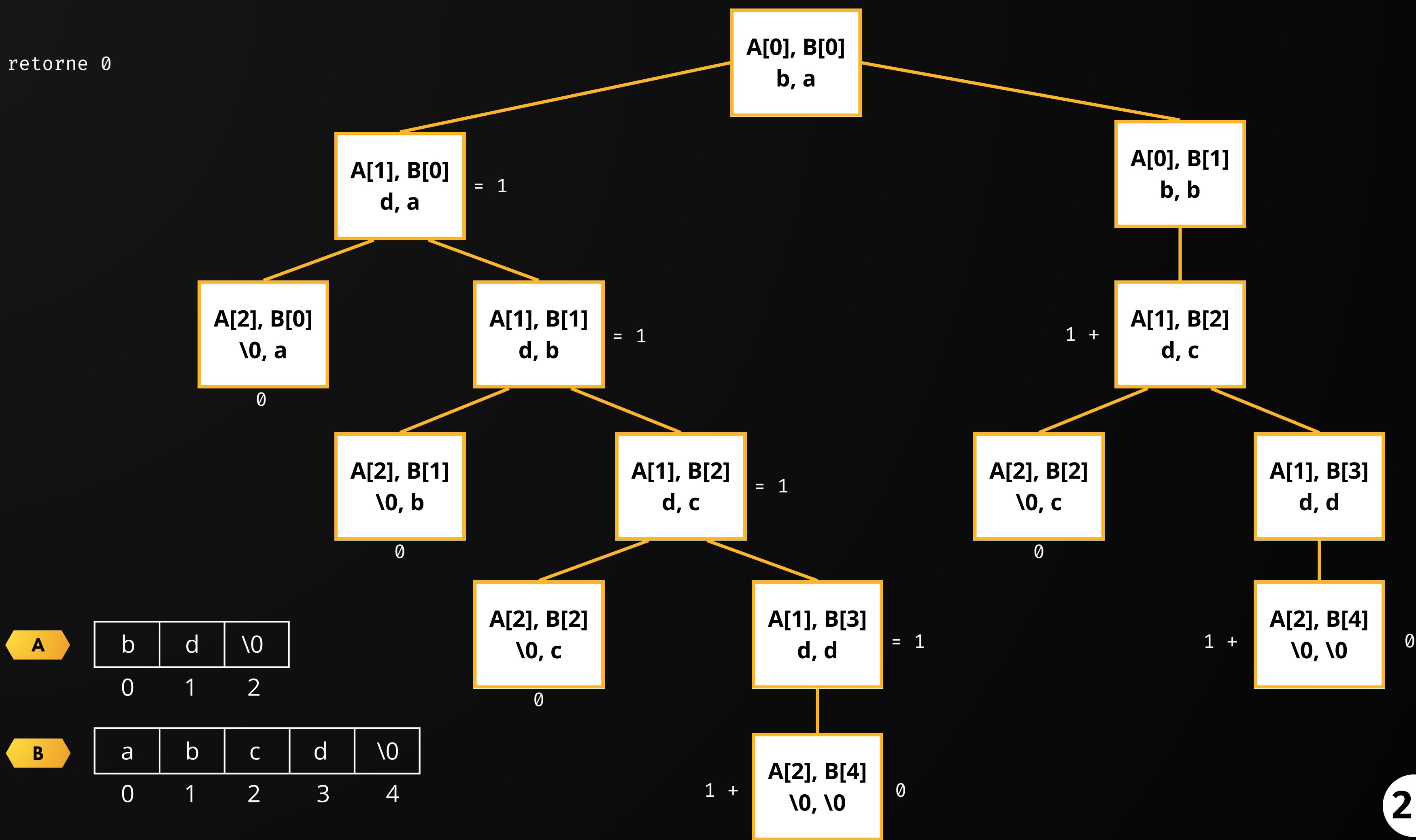
retorne 0

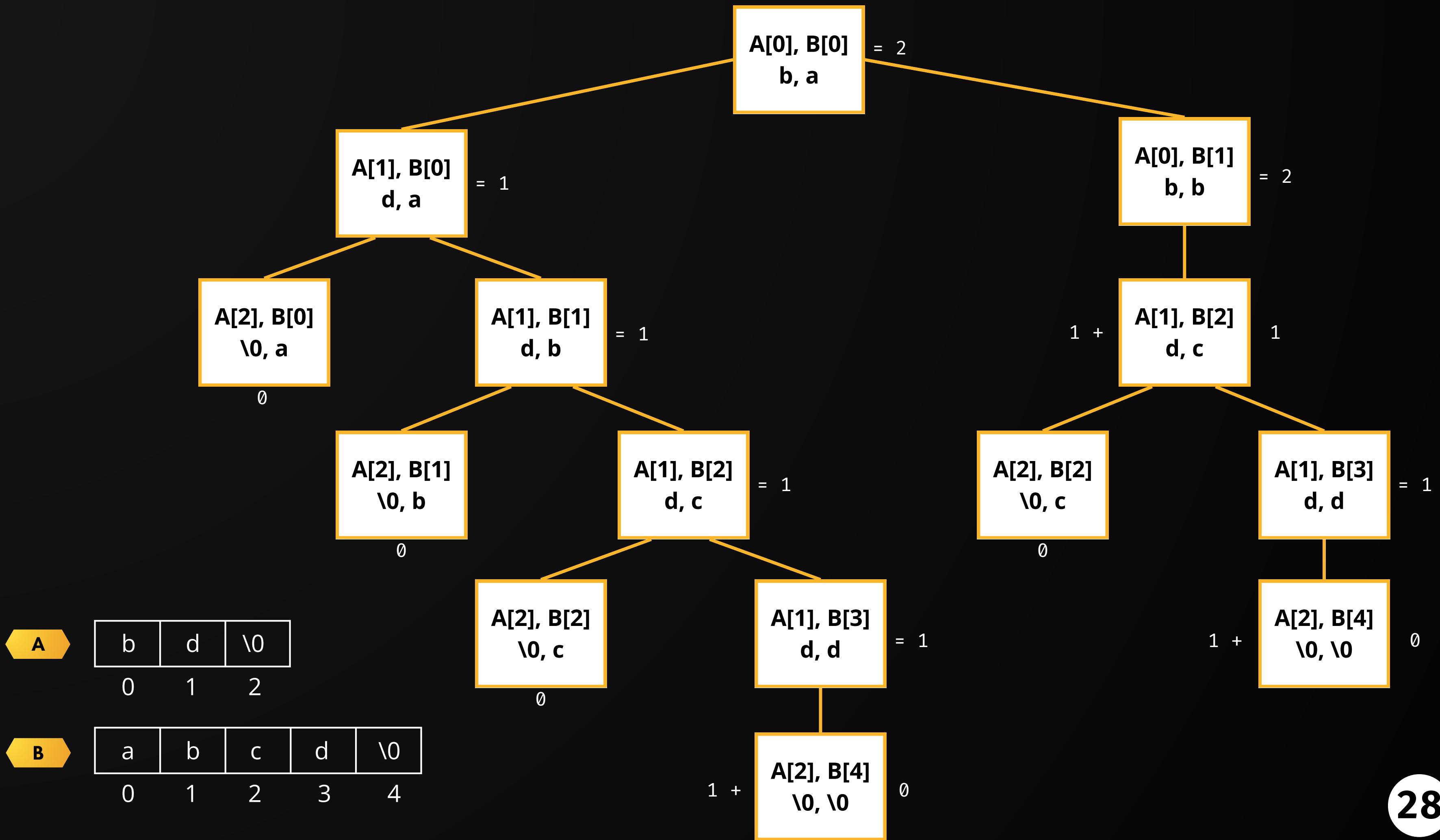


```
retorne 1 + LCS(A, B, i+1, j+1)
```



retorne 0





Qual é a complexidade desse algoritmo?

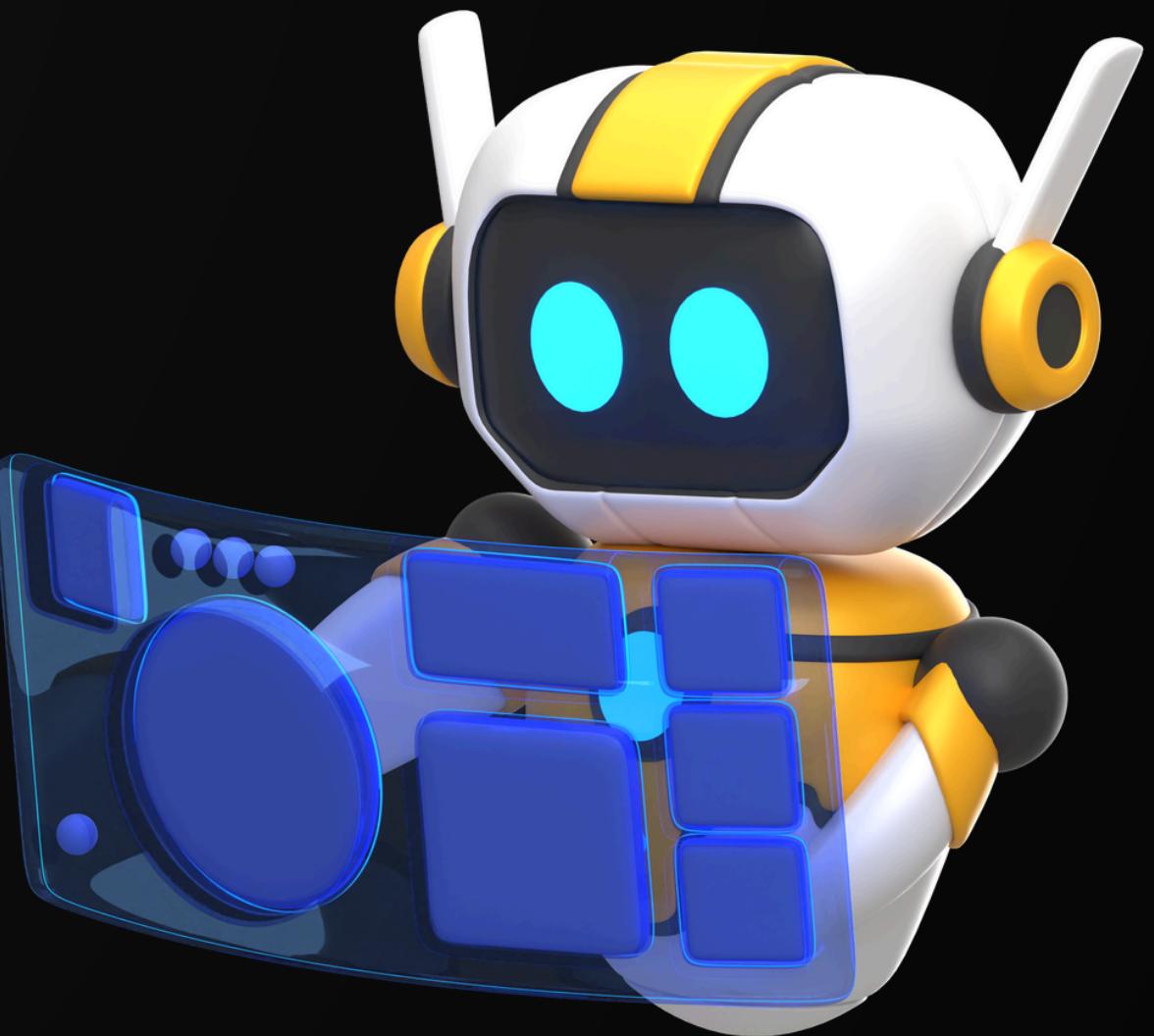


Equação de Recorrência:

$$T(m, n) = \begin{cases} 0, & \text{se } m = 0 \text{ ou } n = 0 \\ 1 + T(m - 1, n - 1), & \text{se } A[m] == B[n] \\ \max(T(m - 1, n), T(m, n - 1)), & \text{senão} \end{cases}$$



**Consideramos o pior caso:
Strings totalmente diferentes!**



Complexidade LCS

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

- pior caso: se $m == n$

Então, podemos simplificar a equação para:

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



Cada chamada gera 2 subproblemas de tamanho $n - 1$

Complexidade LCS

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

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

$$T(n - 2) = 2^3T(n - 3) + 2^2c$$

⋮
⋮
⋮

$$T(n - (k - 1)) = 2^k T(n - k) + 2^{k-1}c$$

Complexidade LCS

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

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

$$T(n - 2) = 2^3T(n - 3) + 2^2c$$

⋮
⋮
⋮

$$T(n - (k - 1)) = 2^k T(n - k) + 2^{k-1}c$$

$$T(n) = 2^k \cdot T(0) + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1})$$

$$T(n) = 2^k \cdot 0 + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1})$$



Complexidade LCS

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

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

$$T(n - 2) = 2^3 T(n - 3) + 2^2 c$$

⋮
⋮
⋮

$$T(n - (k - 1)) = 2^k T(n - k) + 2^{k-1} c$$

$$T(n) = 2^k \cdot T(0) + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1})$$

$$T(n) = 2^k \cdot 0 + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1}) \rightarrow \text{Progressão Geométrica: } S_n = 1 \cdot \frac{2^n - 1}{2 - 1} = 2^n - 1$$

$$T(n - k) = T(0)$$

$$n - k = 0$$

$$n = k$$



- Primeiro termo $a = 1$
- Razão $r = 2$
- Número de termos: n

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Complexidade LCS

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

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

$$T(n - 2) = 2^3 T(n - 3) + 2^2 c$$

⋮
⋮
⋮

$$T(n - (k - 1)) = 2^k T(n - k) + 2^{k-1} c$$

$$T(n) = 2^k \cdot T(0) + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1})$$

$$T(n) = 2^k \cdot 0 + c(2^0 + 2^1 + 2^2 + \dots + 2^{k-1}) \rightarrow \text{Progressão Geométrica: } S_n = 1 \cdot \frac{2^n - 1}{2 - 1} = 2^n - 1$$

$$\begin{aligned} T(n - k) &= T(0) \\ n - k &= 0 \\ n &= k \end{aligned}$$

$$T(n) = c(2^n - 1) \Leftrightarrow T(n) \in O(2^n)$$

- Primeiro termo $a = 1$
- Razão $r = 2$
- Número de termos: n

$$\sum_{i=0}^{n-1} 2^i = 2^n - 1$$

Limitações

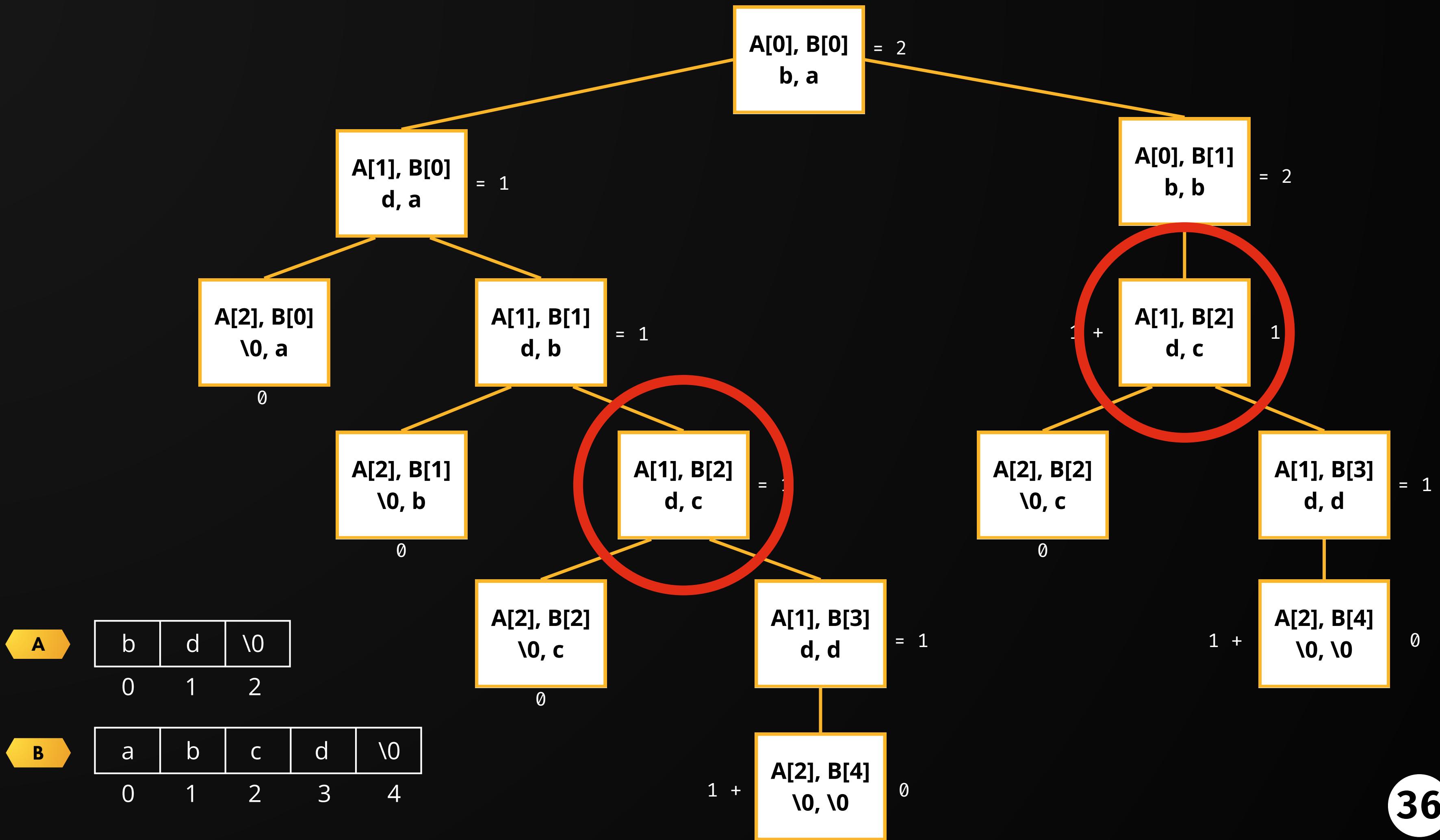


Complexidade Exponencial

Recalcula os mesmos subproblemas

Pode causar stack overflow

Inviável para aplicações reais



Melhorias



Memoization



Programação Dinâmica

Memoization

Algoritmo LCS (A, B, i, j)

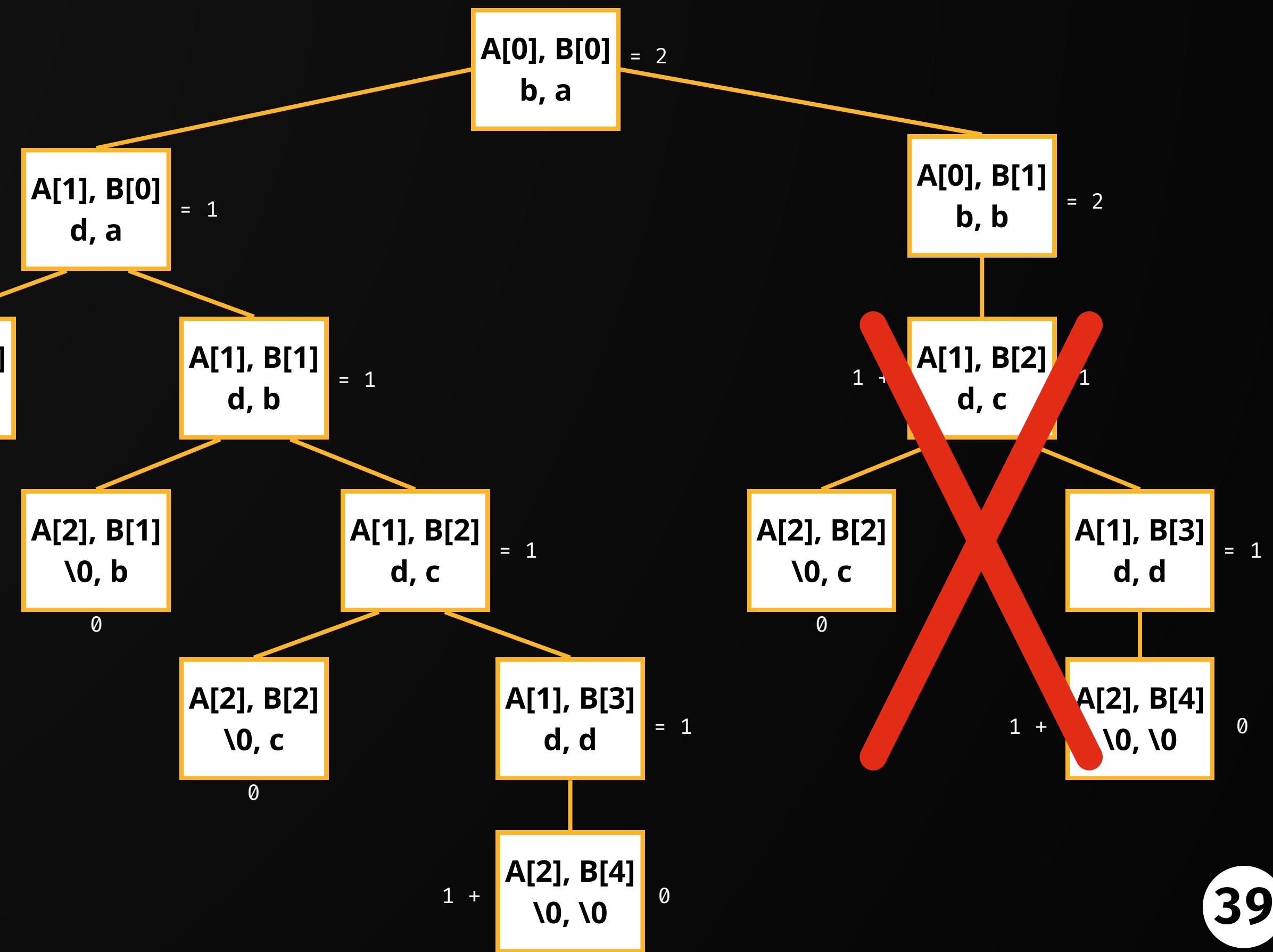
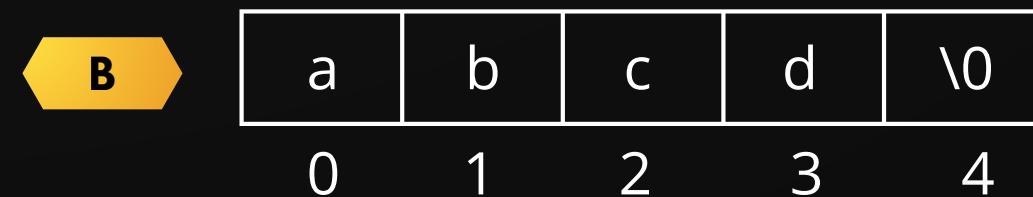
Entrada: duas strings A e B a serem comparadas e seus índices i e j.

Saída: o tamanho de uma subsequência C comum à A e B.

```
se A[i] == '\0' ou B[j] == '\0' então
    retorne 0
se memo[i][j] != -1 então
    retorne memo[i][j]
se A[i] == B[j] então
    retorne memo[i][j] ← 1 + LCS(A, B, i+1, j+1, memo)
senão
    retorne memo[i][j] ← max(LCS(A, B, i+1, j, memo), LCS(A, B, i, j+1, memo))
```

a	b	c	d	\0
0	1	2	3	4

b	0	2	2		
d	1	1	1	1	
\0	2	0	0	0	0



LCS com Memoization

Vantagens

- **Salva resultados** já resolvidos
- Muito mais **rápido**
- Reduz a complexidade para **$O(m*n)$** no pior caso
- Se torna um algoritmo viável para strings maiores

Desvantagens

- Por ser recursivo, continua tendo risco de **stack overflow**
- **Consome memória** para guardar os resultados
- Ainda pode ser um problema se os tamanhos das strings forem grandes demais

Dynamic Programming

Algoritmo LCS (A, B)

Entrada: duas strings A e B a serem comparadas.

Saída: o tamanho de uma subsequência C comum à A e B.

$m \leftarrow$ comprimento de A

$n \leftarrow$ comprimento de B

criar matriz $dp[0..m][0..n]$ preenchida com zeros

para $i \leftarrow 1$ até m faça

 para $j \leftarrow 1$ até n faça

 se $A[i-1] == B[j-1]$ então

$dp[i][j] \leftarrow 1 + dp[i-1][j-1]$

 senão

$dp[i][j] \leftarrow$ máximo entre $dp[i-1][j]$ e $dp[i][j-1]$

retorne $dp[m][n]$

A

b	d
0	1

B

a	b	c	d
0	1	2	3

$m \leftarrow$ comprimento de A
 $n \leftarrow$ comprimento de B

```
para i  $\leftarrow 1$  até  $m$  faça
    para j  $\leftarrow 1$  até  $n$  faça
        se A[i-1] == B[j-1] então
            dp[i][j]  $\leftarrow 1 + dp[i-1][j-1]$ 
        senão
            dp[i][j]  $\leftarrow \max(dp[i-1][j], dp[i][j-1])$ 
    retorno dp[m][n]
```

	a	b	c	d
0	0	0	0	0
b	0			
1				
d	0			
2				

A

b	d
1	2

B

a	b	c	d
1	2	3	4

	a	b	c	d
0	0	0	0	0
b	0	0	1	1
1	0	0	1	1
d	0	0	1	2
2	0	0	1	2

Reconstrução da LCS

A

b	d
0	1

B

a	b	c	d
0	1	2	3

```
i < m  
j < n  
LCS < string vazia
```

```
enquanto i > 0 e j > 0 faça  
    se A[i-1] == B[j-1] então  
        adicionar A[i-1] no início de LCS  
        i < i - 1  
        j < j - 1  
    senão se dp[i-1][j] > dp[i][j-1] então  
        i < i - 1  
    senão  
        j < j - 1
```

retorne dp[m][n], LCS

	a	b	c	d
0	0	0	0	0
b 1	0	0	1	1
d 2	0	0	1	1

Tempo de Execução

LCS com DP

Vantagens

- **Reutiliza** resultados
- Por ser **iterativo**, evita stack overflow
- Mais eficiente: $O(m*n)$
- Com um código adicional, também é possível **reconstruir a substring LCS**

Desvantagens

- Mesmo no melhor caso, ele ainda preenche toda a matriz
- **Consome memória** para guardar os resultados

Aplicações na Vida Real



Bioinformática

Sequências de DNA, RNA e proteínas

- O LCS é uma **base** para algoritmos de comparação de cadeias biológicas
- Detectar regiões semelhantes entre sequências genéticas
- Inferir **relações evolutivas** e árvores filogenéticas
- Análise de **mutações**: ajuda a identificar inserções, deleções e substituições

Controle de Versões

Git, diff, merge...

- ▶ Determinação de linhas adicionadas, removidas ou mantidas entre versões
- ▶ Detecção de conflitos e união de modificações de diferentes usuários
- ▶ Histórico de alterações que foram mantidas durante o tempo
- ▶ Minimizar conflitos durante integração contínua

Comparação de textos

Detecção de plágio

- Detecção de **cópias** literais ou quase literais (longas subsequências)
- Cálculo de **similaridade textual** baseado na proporção de subsequências comuns
- Identificação de **reuso** código-fonte (plágio em programação)
- Análise de autorial textual: apoio à detecção de **manipulação** de conteúdo



Obrigado!

Referências Bibliográficas

- Alinhamento de Sequências, OnlineBioinfo Bioinformática
- Longest Common Subsequence - Recursion and DP, Abdul Bari
- Um estudo avançado do problema da Maior Subsequência Comum, UFBA
- Longest Common Subsequence, GeeksForGeeks
- Subsequência Comum Mais Longa, WsCube Tech
- Maior subsequência comum, Wikipedia