

Dynamic Programming

What is dynamic programming?

Memoization: store the return value of an expensive function call and return the stored value when the same input of the function call appears.

Dynamic Programming: divide and conquer + memoization.

Use divide and conquer to generate a lot of subproblems (function calls).

Use memoization to reuse the result of computed subproblems.

Fibonacci Numbers

Output F_k

$F_1 = 1, F_2 = 1, F_{n+2} = F_{n+1} + F_n$ for every $n \geq 1$.

Input: an integer k .

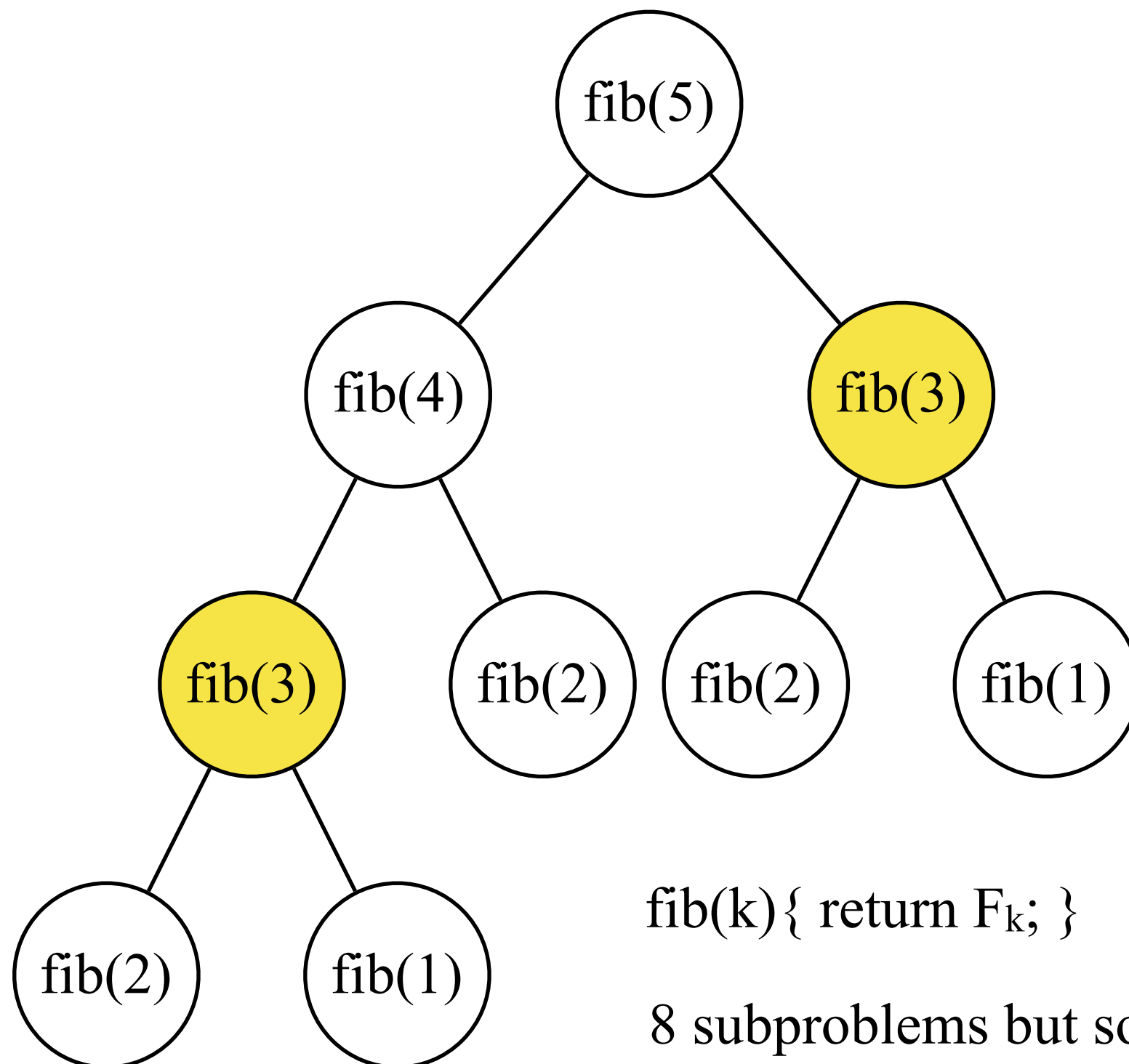
Output: F_k .

--- Example ---

$$F_3 = F_2 + F_1 = 2.$$

$$F_5 = F_4 + F_3 = (F_3 + F_2) + F_3 = (F_2 + F_1) + F_2 + (F_2 + F_1) = 5.$$

Divide and Conquer + Memoization



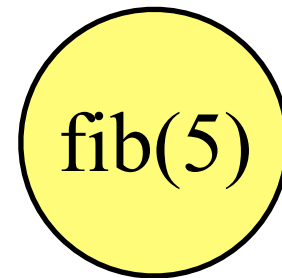
fib(5) ?

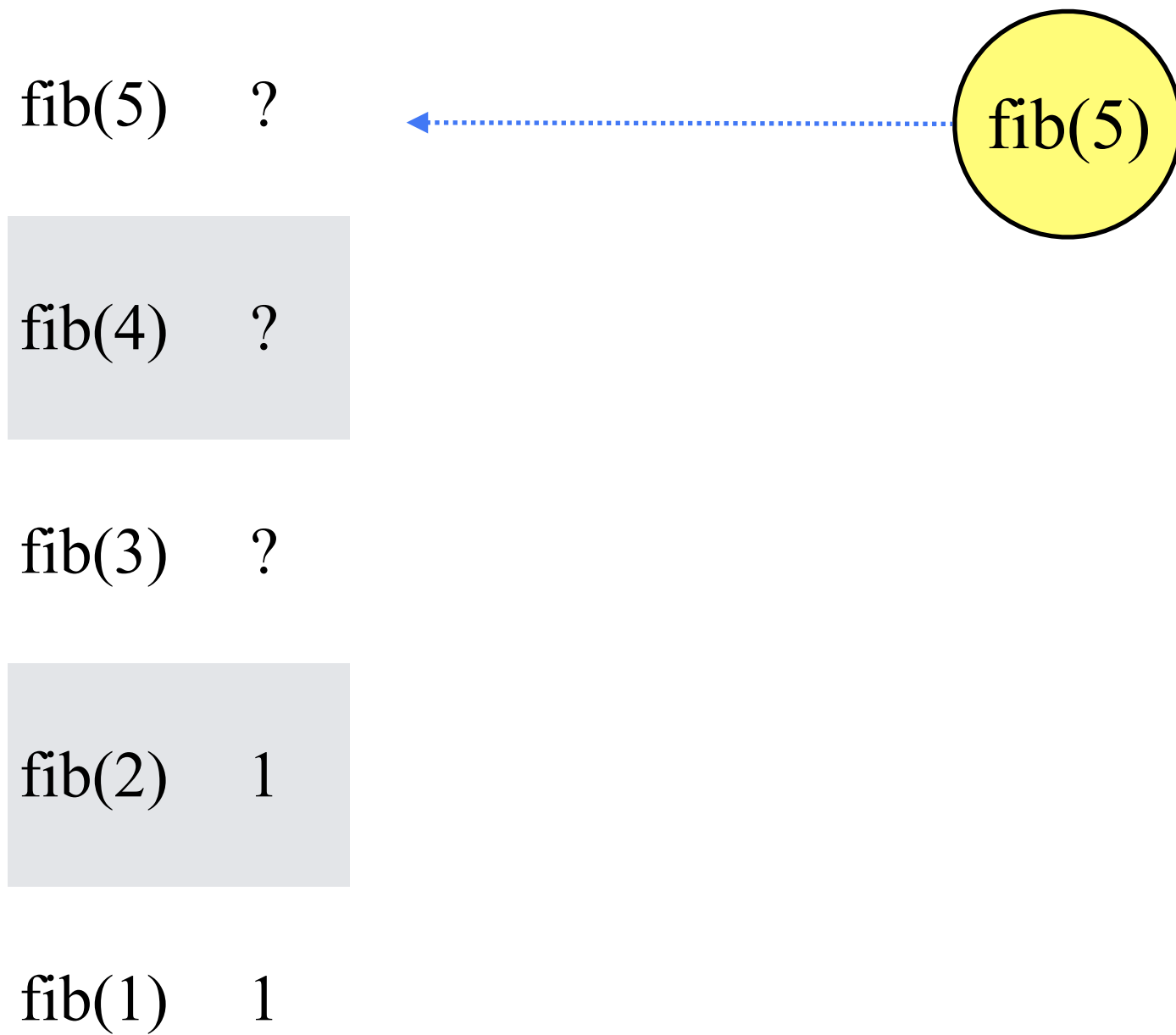
fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1





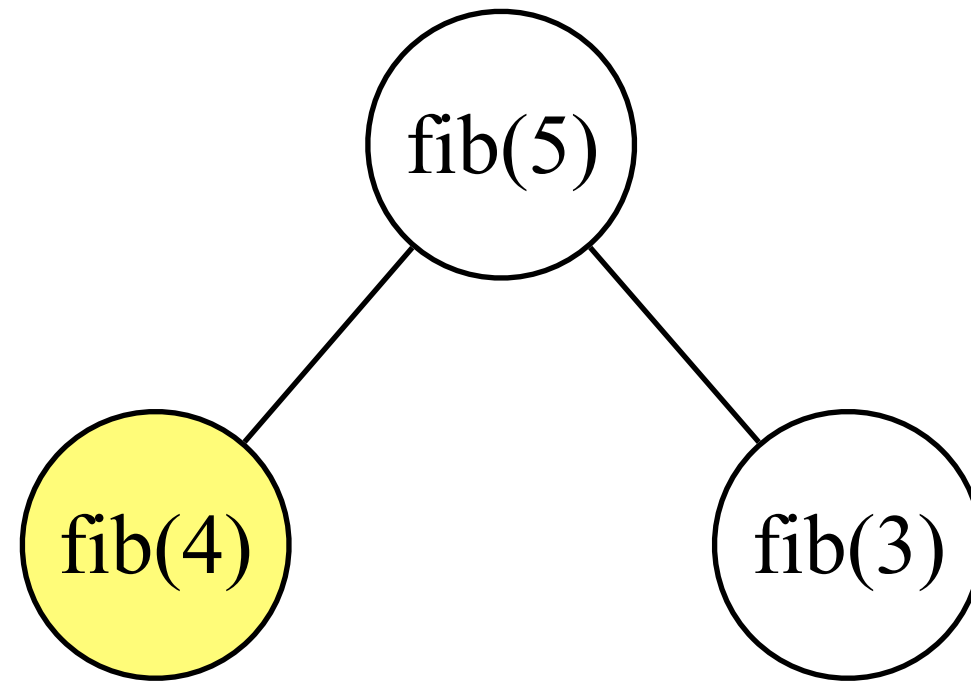
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



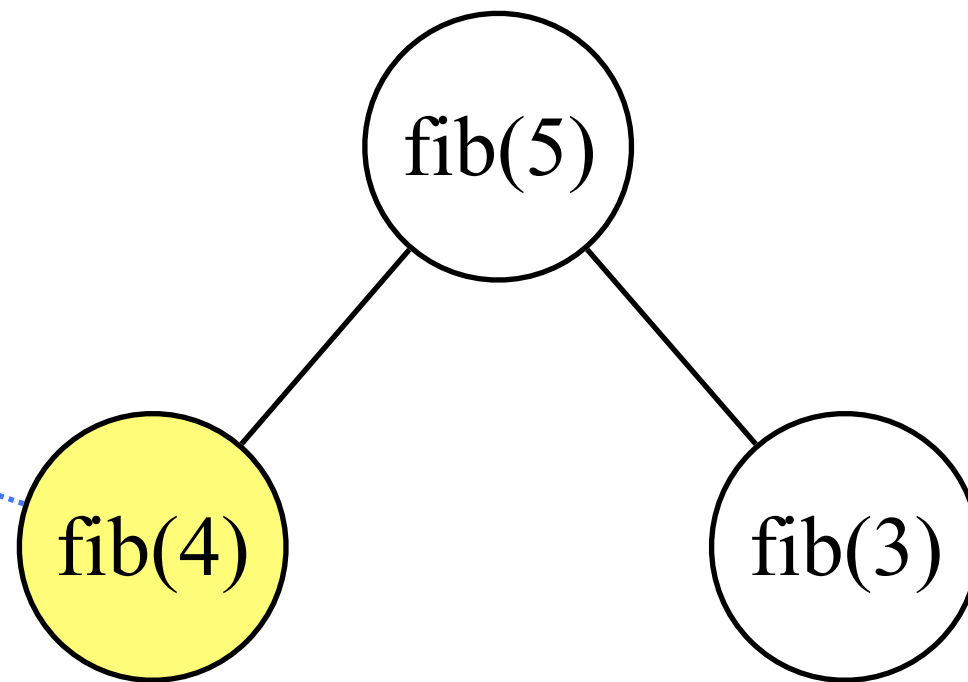
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



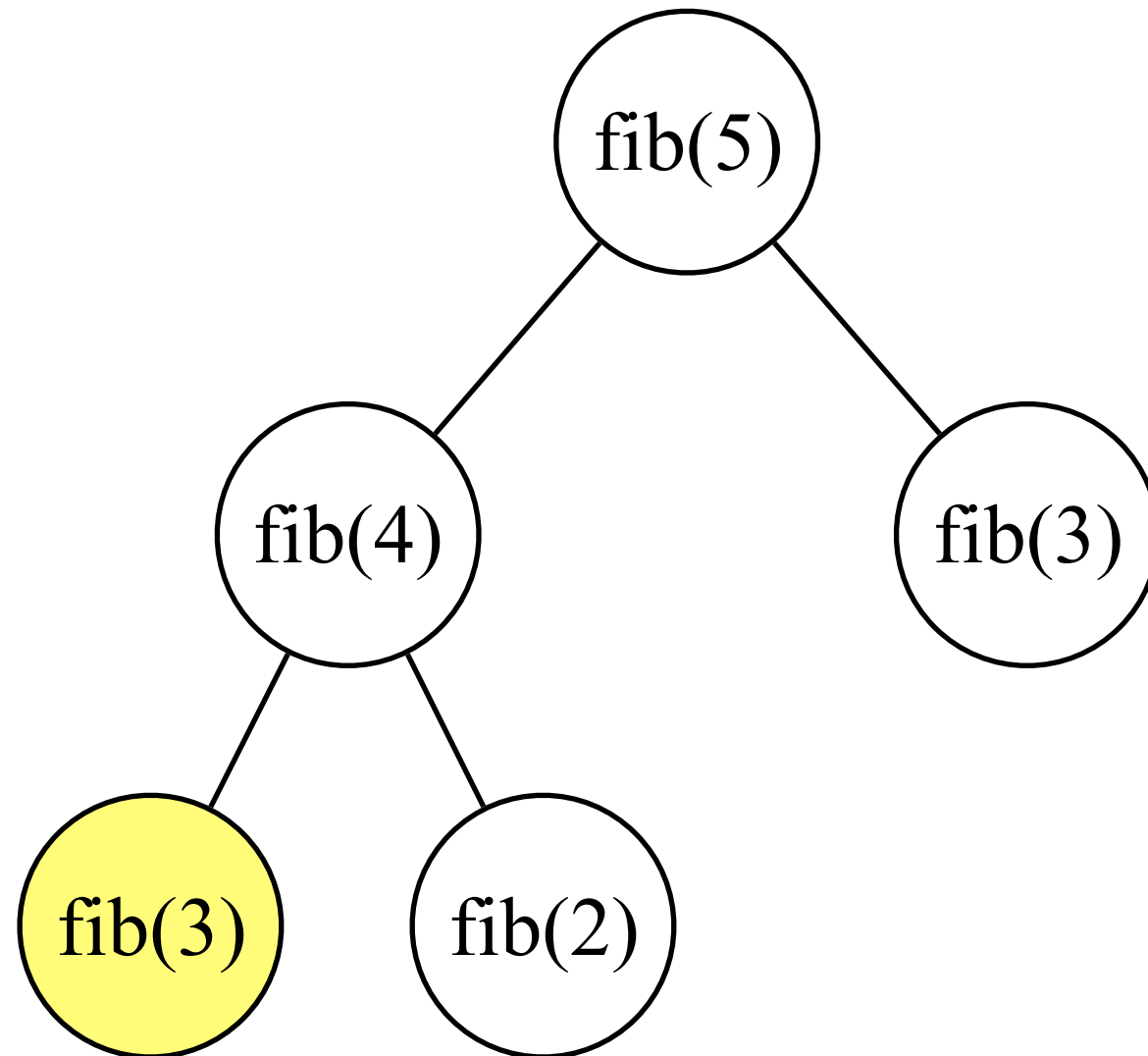
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



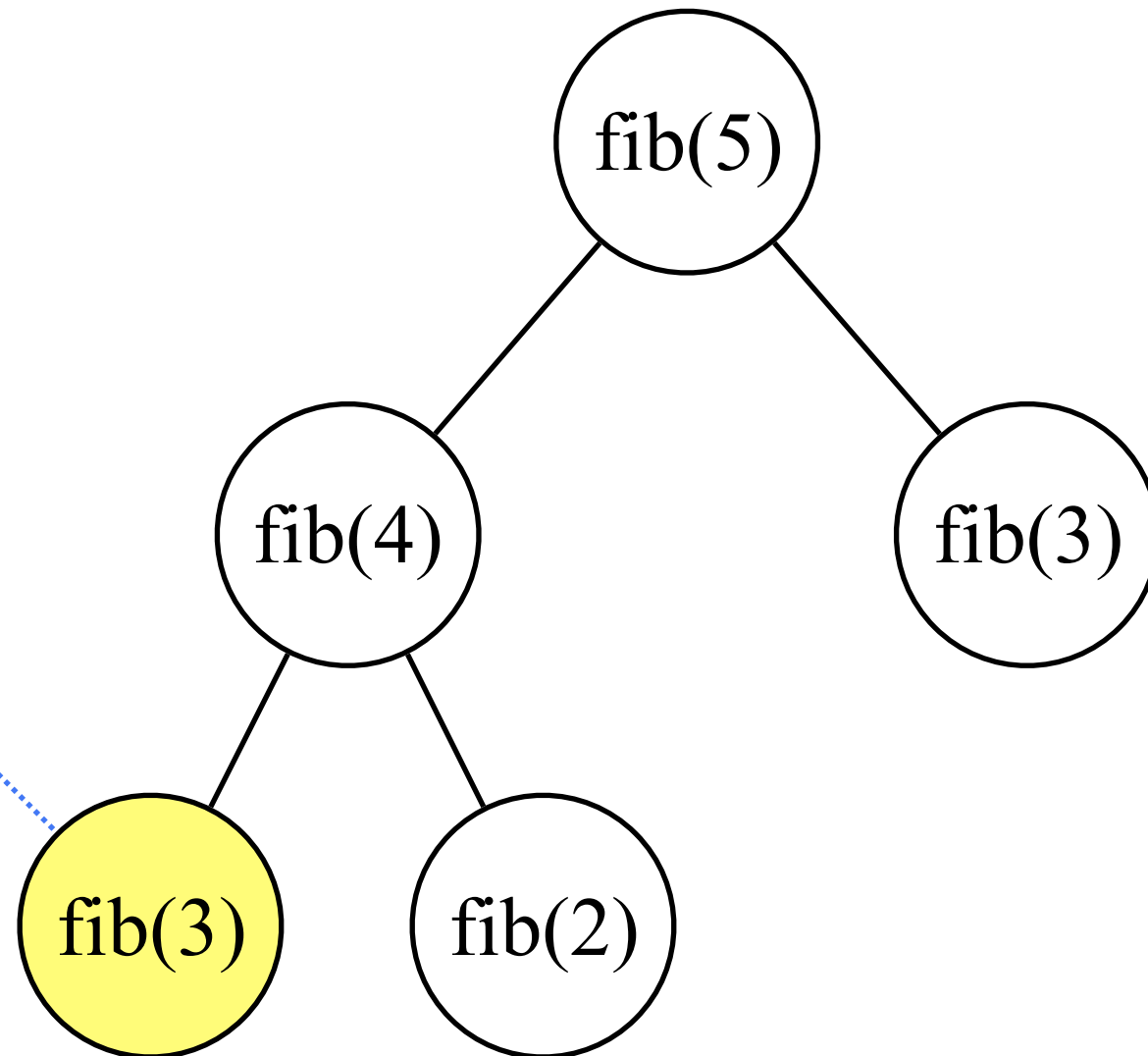
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



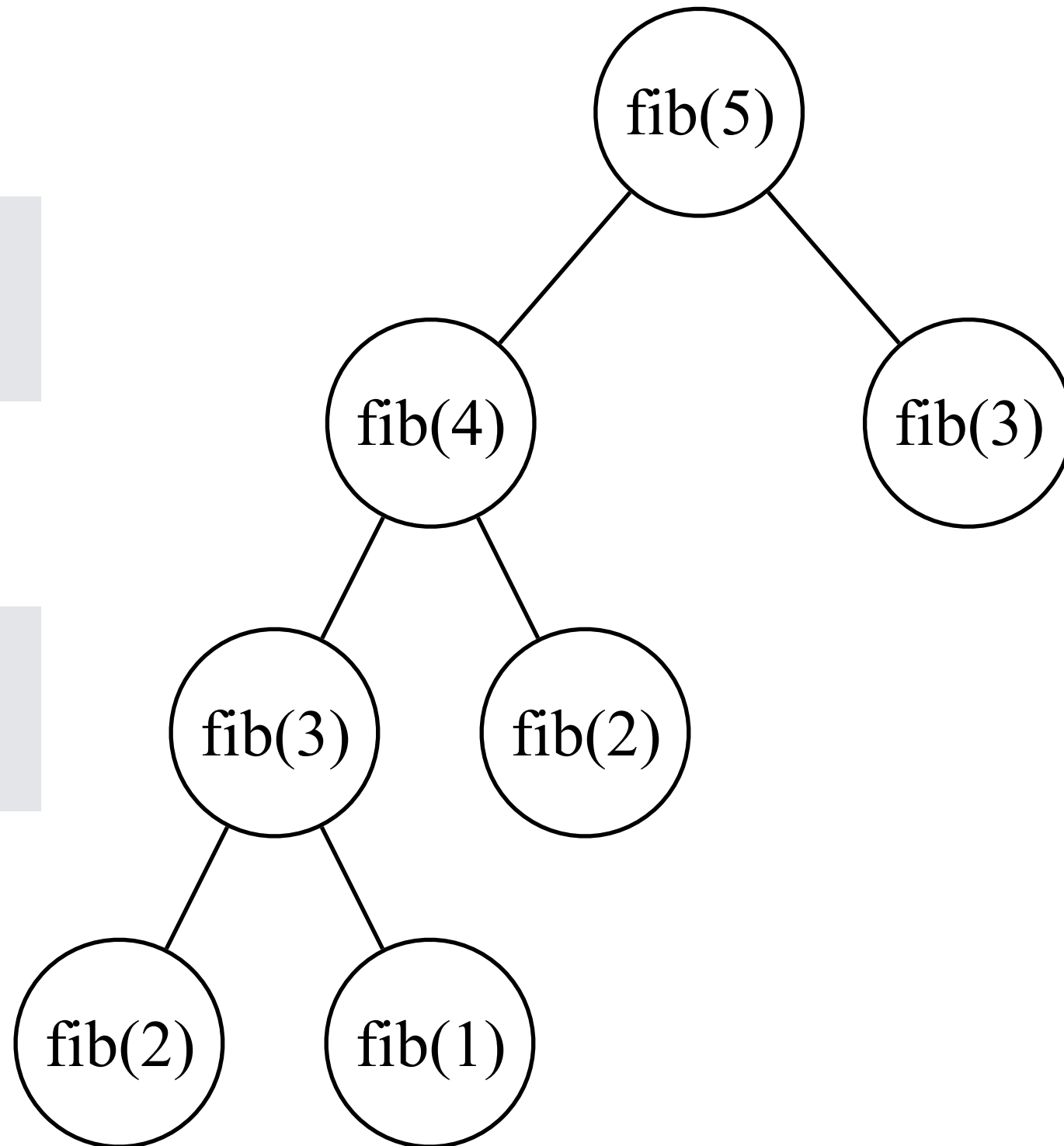
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



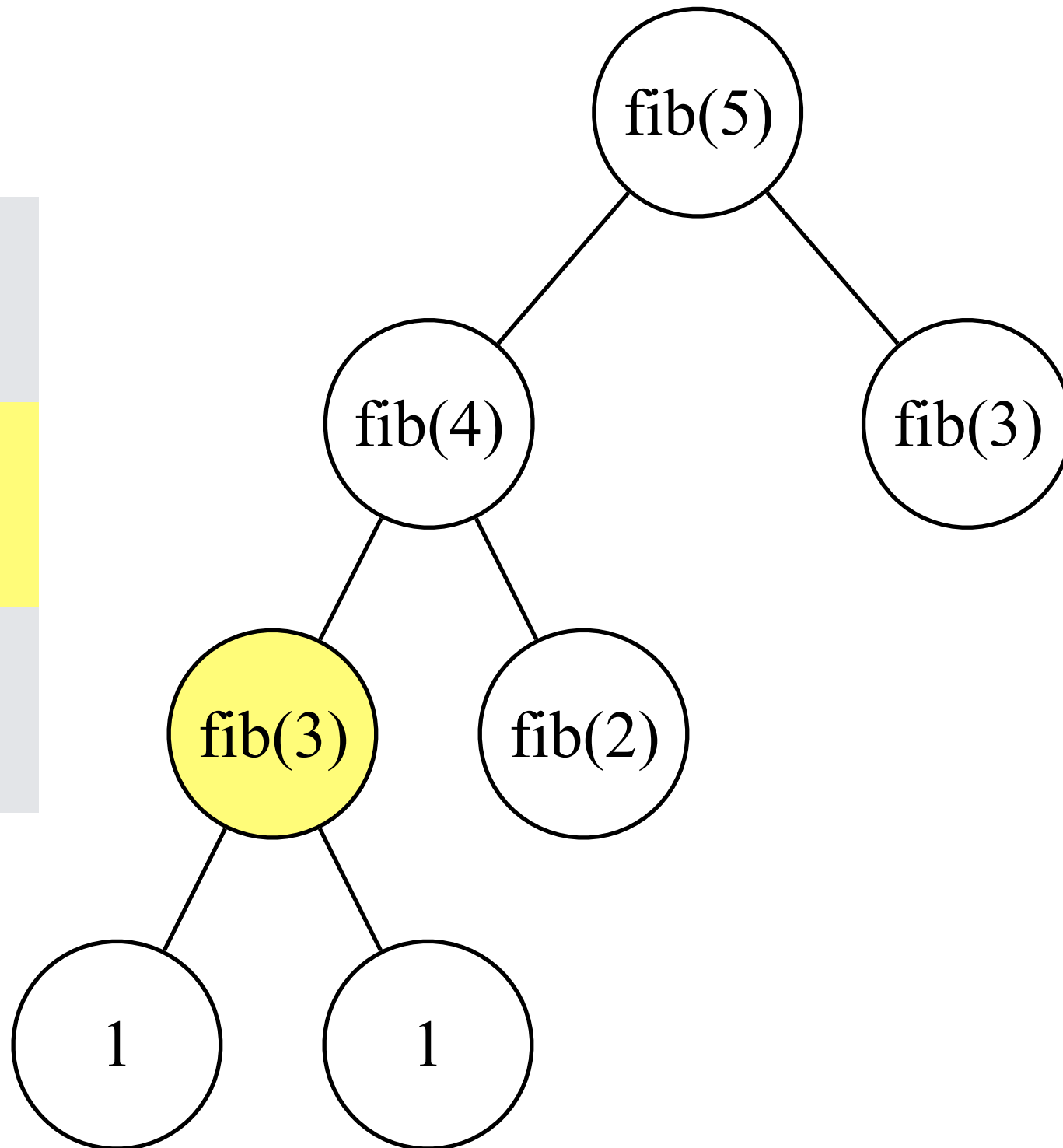
fib(5) ?

fib(4) ?

fib(3) ?

fib(2) 1

fib(1) 1



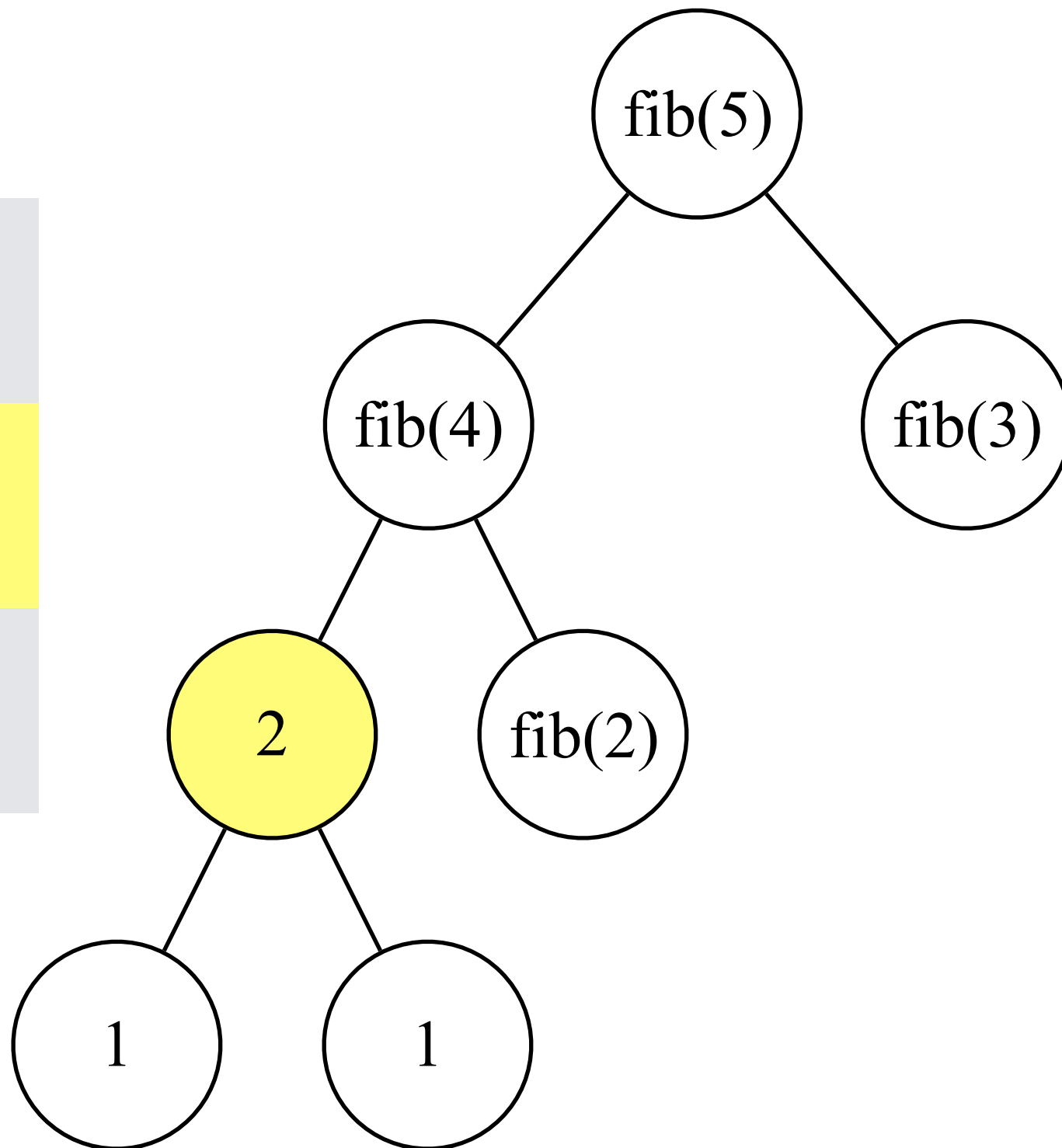
fib(5) ?

fib(4) ?

fib(3) 2

fib(2) 1

fib(1) 1



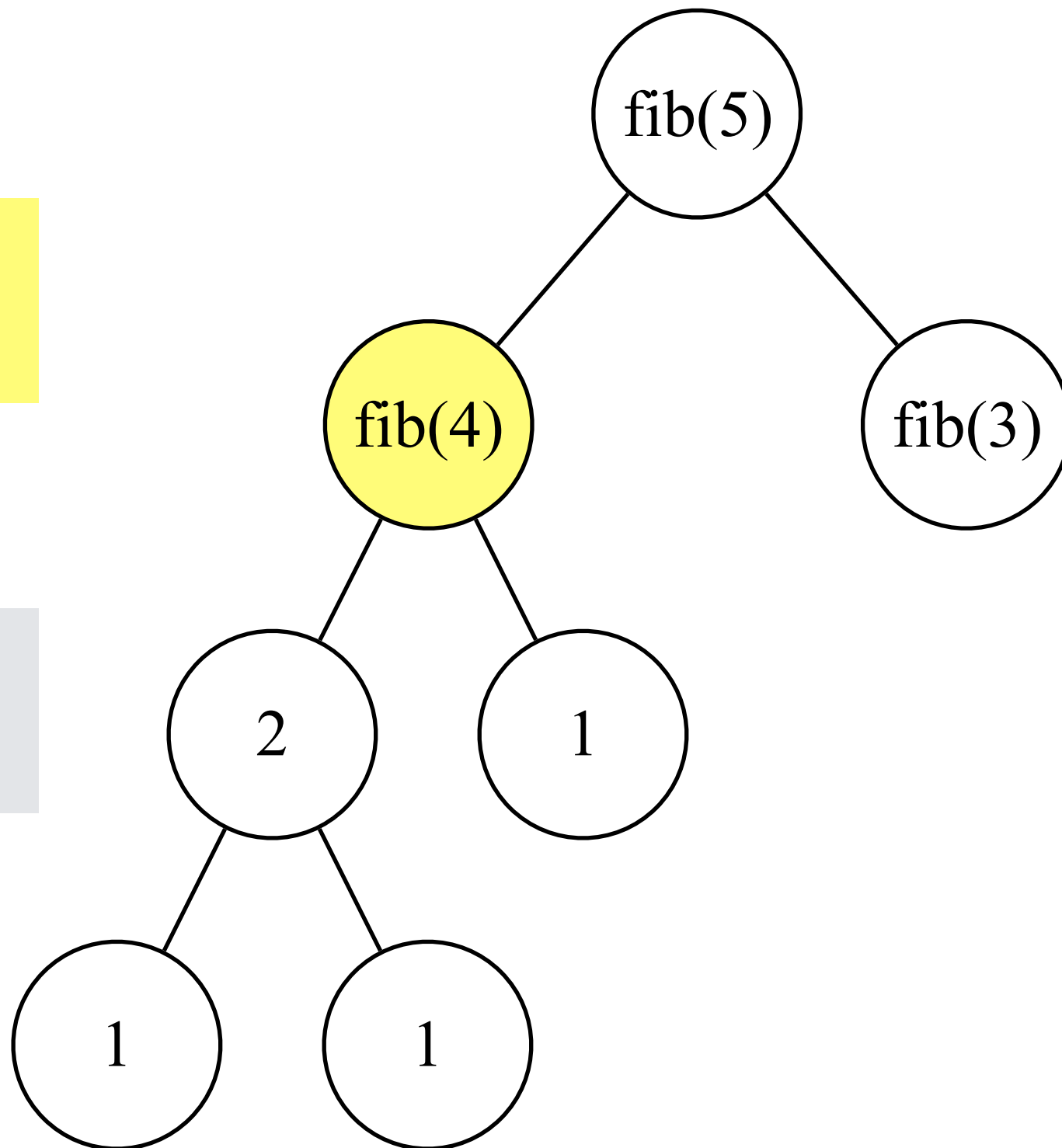
fib(5) ?

fib(4) ?

fib(3) 2

fib(2) 1

fib(1) 1



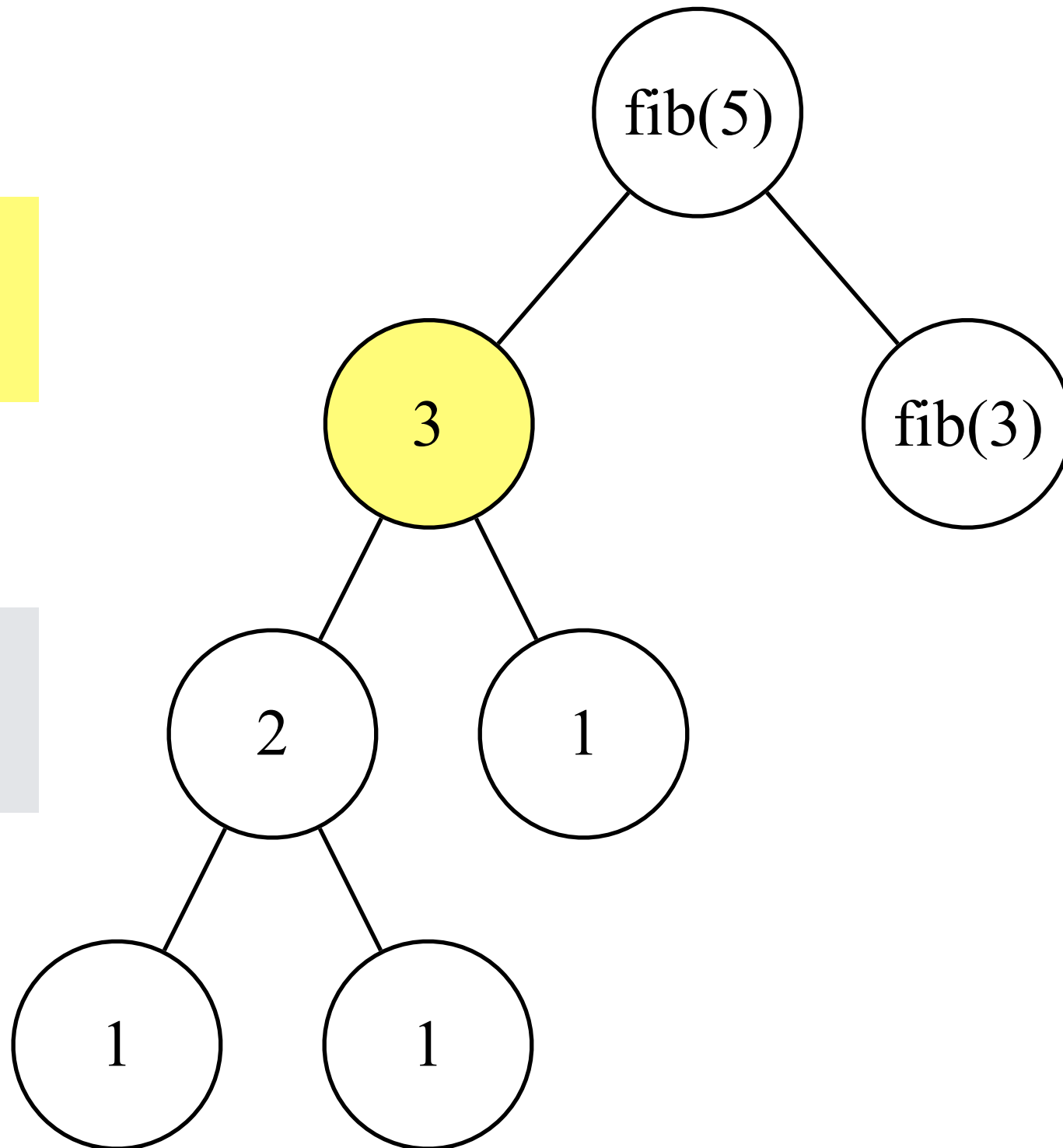
fib(5) ?

fib(4) 3

fib(3) 2

fib(2) 1

fib(1) 1



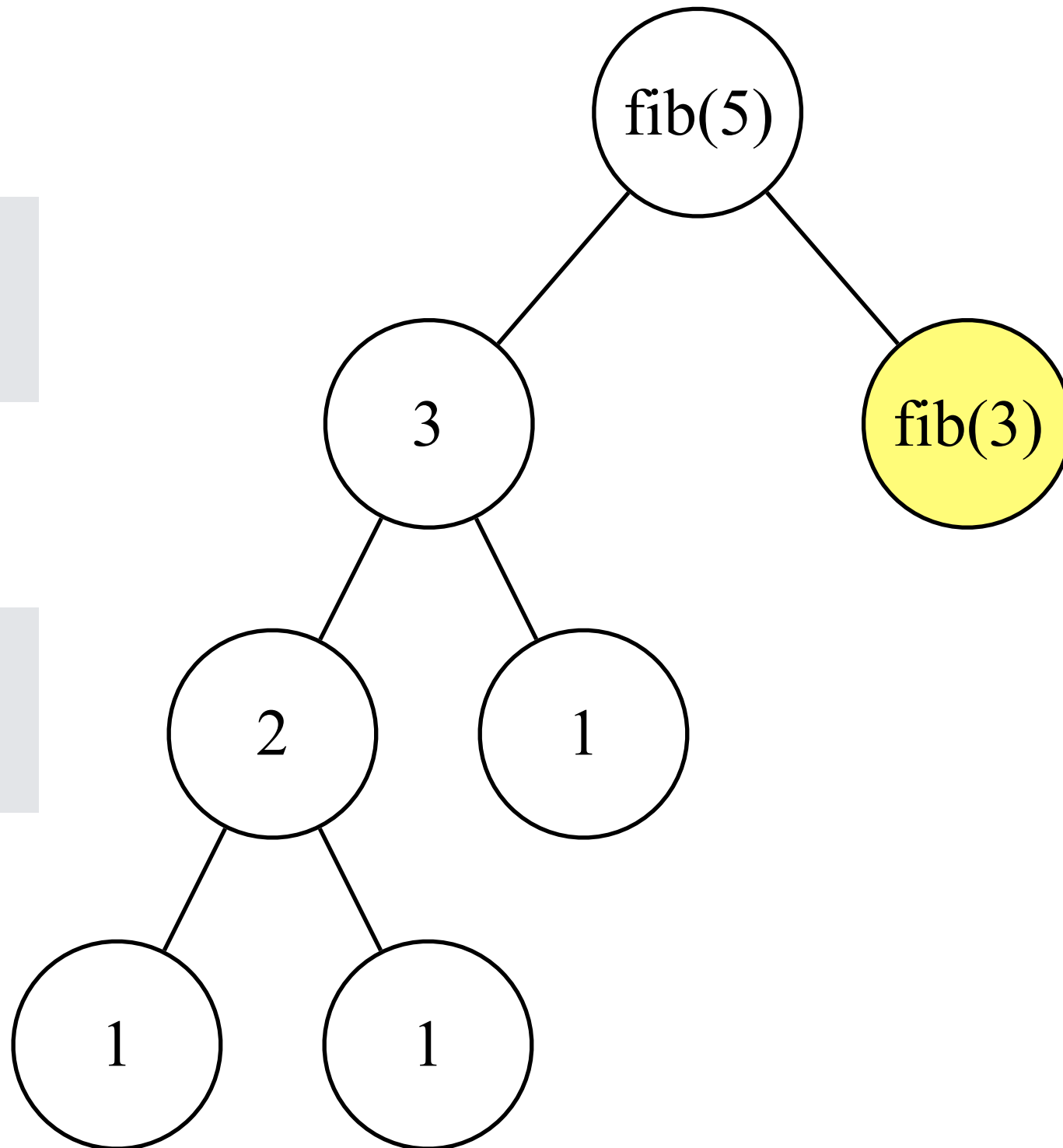
fib(5) ?

fib(4) 3

fib(3) 2

fib(2) 1

fib(1) 1



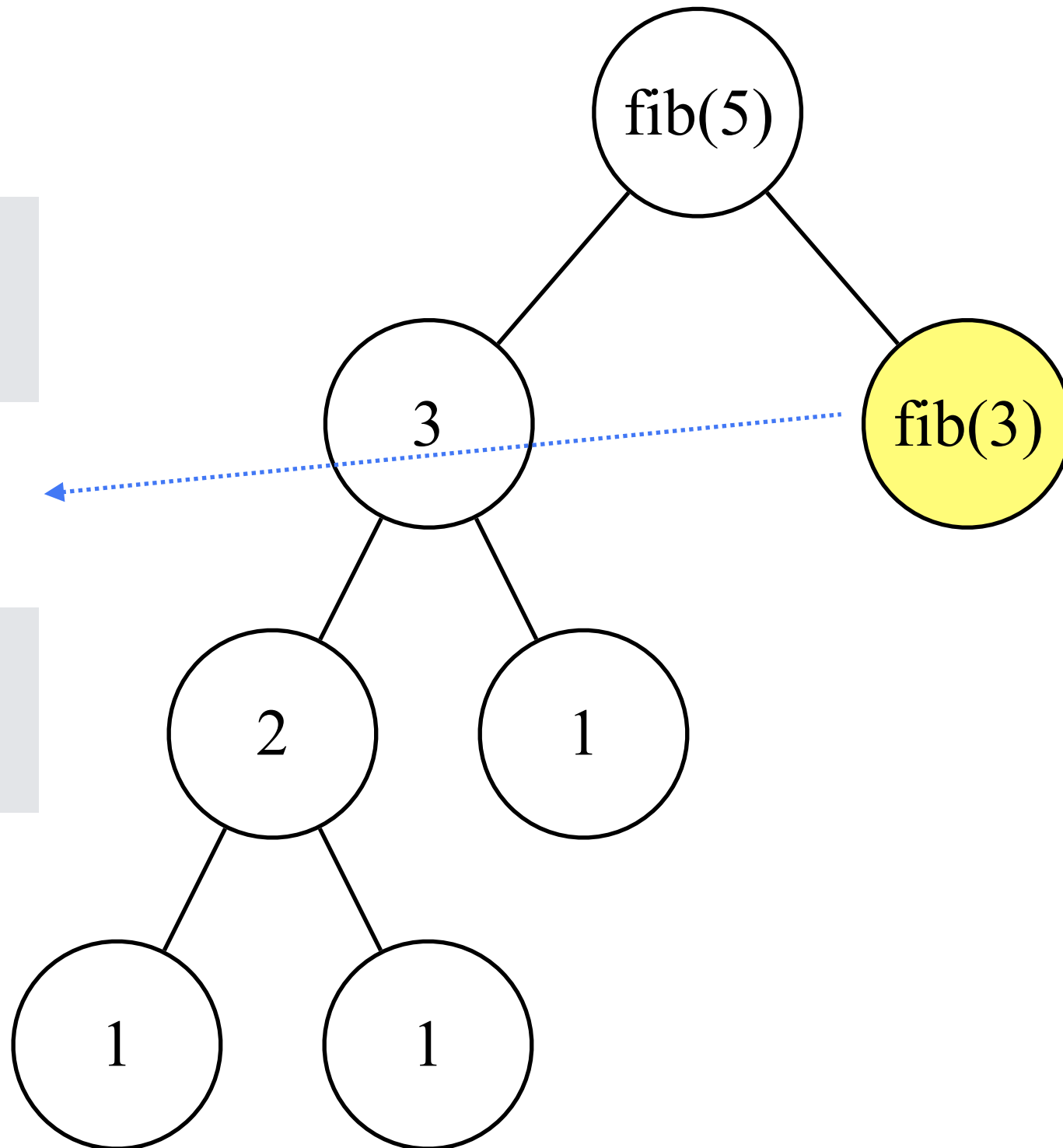
fib(5) ?

fib(4) 3

fib(3) 2

fib(2) 1

fib(1) 1



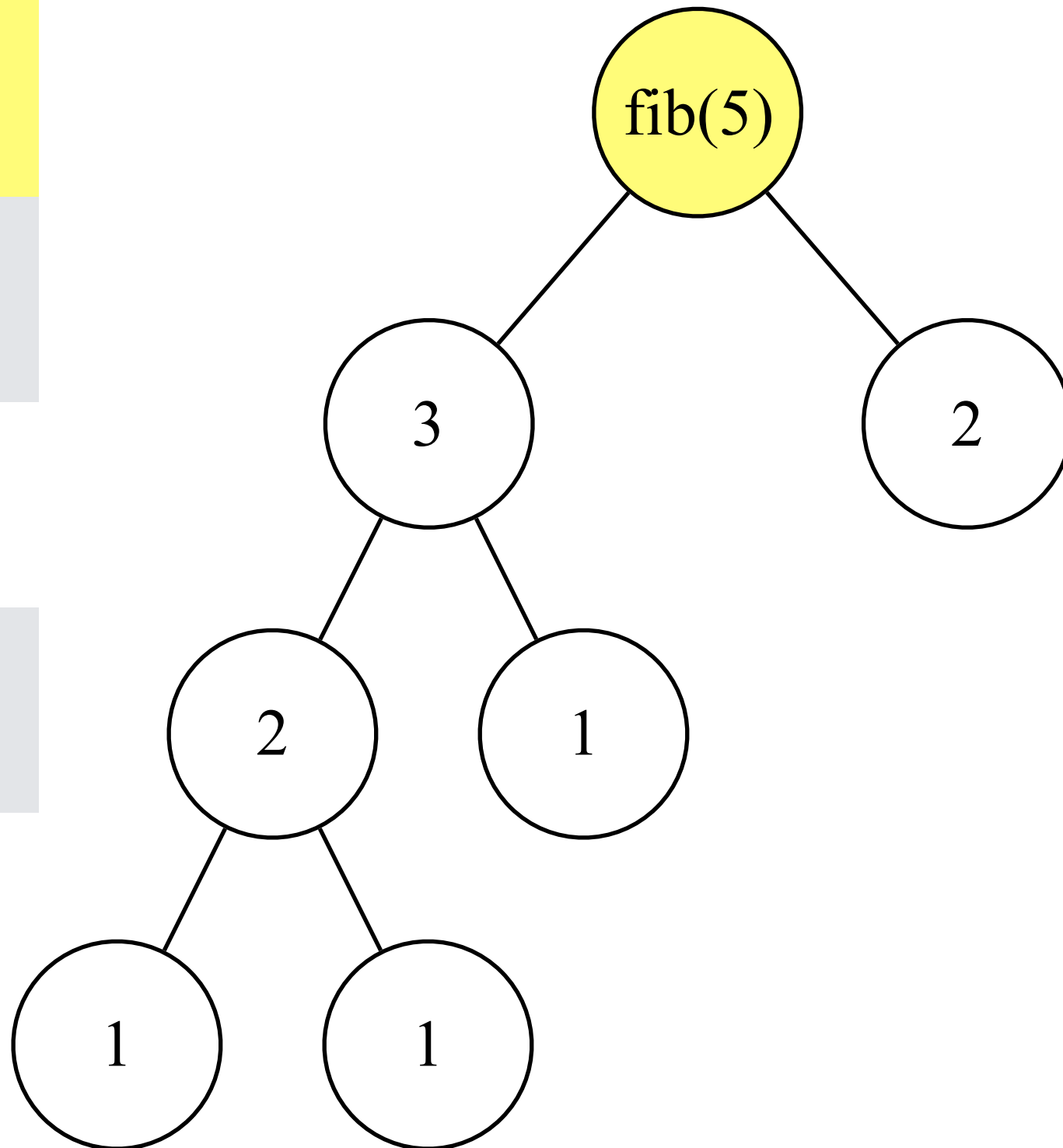
fib(5)	?
--------	---

fib(4)	3
--------	---

fib(3)	2
--------	---

fib(2)	1
--------	---

fib(1)	1
--------	---



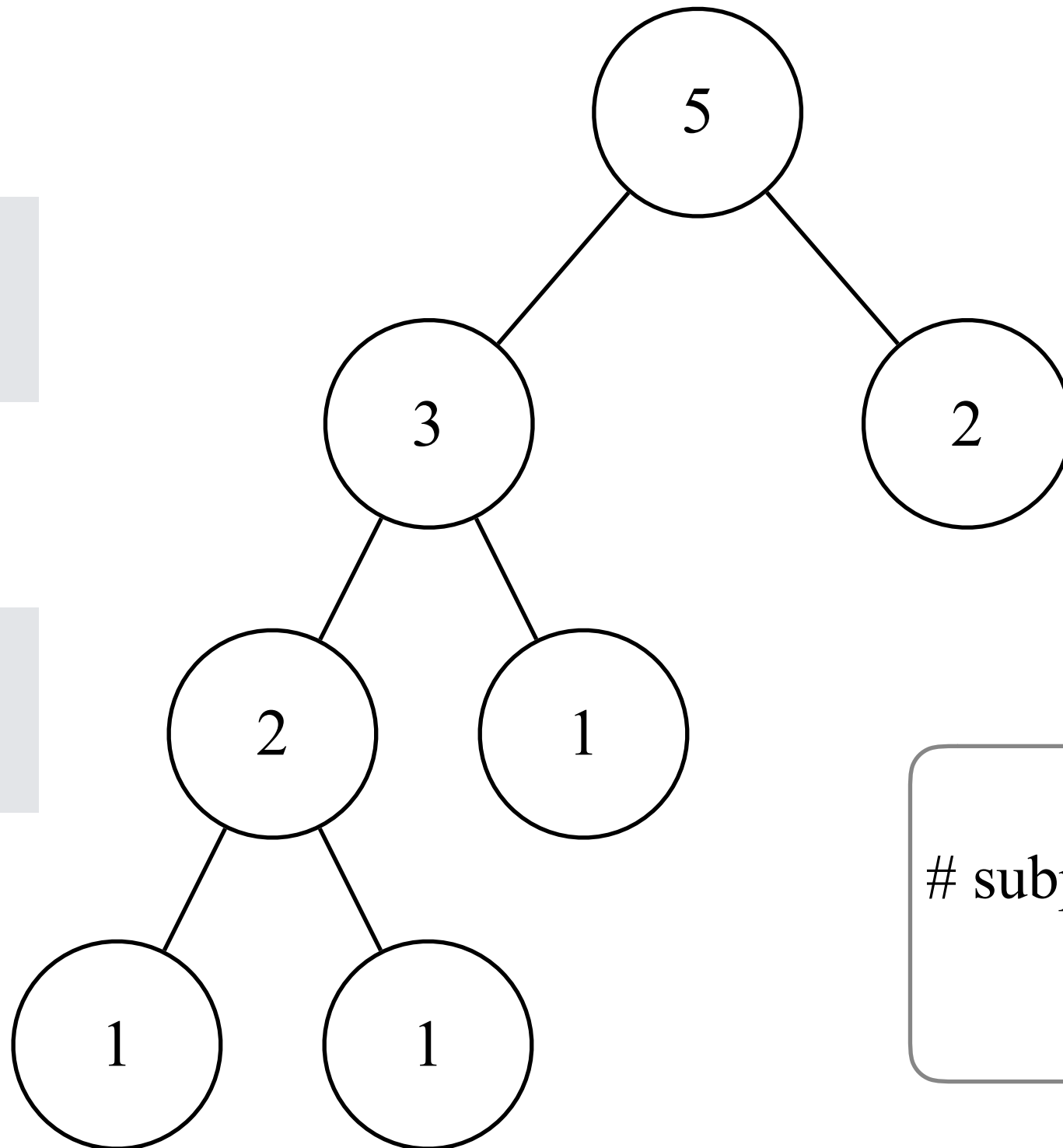
fib(5) 5

fib(4) 3

fib(3) 2

fib(2) 1

fib(1) 1



subproblems is reduced
from 8 to 6.

Maximum Subarray Problem

Find a contiguous subarray with max sum

Input: an array $A[1..n]$ of real numbers.

Output: x and y so that $\sum_{x \leq i \leq y} A[i] \geq \sum_{a \leq i \leq b} A[i]$ for any

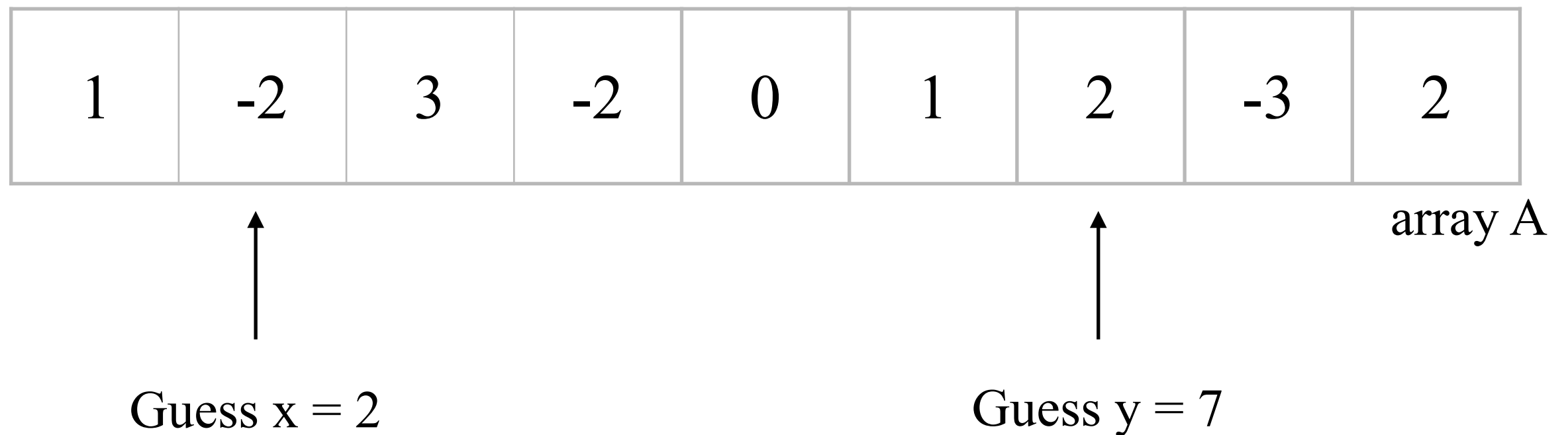
$a, b \in \{1, 2, \dots, n\}, a \leq b$.

Example.

1	-2	3	-2	0	1	2	-3	2
---	----	---	----	---	---	---	----	---

array A

A naive approach



Try $O(n^2)$ guesses. Each guess needs $O(n)$ time to calculate the sum. In total, the algorithm runs in $O(n^3)$ time.

A better approach

Guess $x = 2$

Guess $y = 7$



1	-2	3	-2	0	1	2	-3	2
---	----	---	----	---	---	---	----	---

array A

0	1	-1	2	0	0	1	3	0	2
---	---	----	---	---	---	---	---	---	---

prefix sum S

Still try $O(n^2)$ guesses, but each guess needs $O(1)$ time to calculate the sum, given the array of prefix sums.

In total, the algorithm runs in $O(n^2)$ time.

An optimal approach

Guess $y = 7$



0	1	-1	2	0	0	1	3	0	2
---	---	----	---	---	---	---	---	---	---

prefix sum S

Guess y only. There are $O(n)$ guesses.
To maximize $S[y] - S[x-1]$ for a fixed y , what is $x-1$?

An optimal approach

Guess $y = 7$



0	1	-1	2	0	0	1	3	0	2
---	---	----	---	---	---	---	---	---	---

prefix sum S



When $S[x-1]$ is the minimum value among those
preceeding $S[y]$,
 $S[y] - S[x-1]$ is maximized.

Guess y only. There are $O(n)$ guesses.
To maximize $S[y] - S[x-1]$ for a fixed y , what is $x-1$?

An optimal approach

Guess $y = 7$



0	1	-1	2	0	0	1	3	0	2
---	---	----	---	---	---	---	---	---	---

prefix sum S



When $S[x-1]$ is the minimum value among those
preceeding $S[y]$,
 $S[y]-S[x-1]$ is maximized.

It takes $O(n)$ time to identify the corresponding x 's for all y 's. (Why?)

In total, the running time is $O(n)$.

This algorithm is optimal because any algorithm to solve this problem
needs to read all entries in the array.

The Rod-cutting Problem

Finding a way to cut a rod to maximize the revenue

Input: given a rod of n inches and a table of prices p_i for $i = 1, 2, \dots, n$ where p_i denotes the price of a rod of i inches.

Output: the maximum revenue obtainable by cutting up the rod and selling the pieces.

Example.



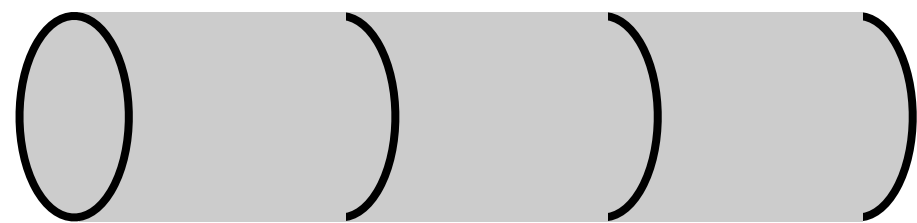
inches	1	2	3
price	2	5	6

Finding a way to cut a rod to maximize the revenue

Input: given a rod of n inches and a table of prices p_i for $i = 1, 2, \dots, n$ where p_i denotes the price of a rod of i inches.

Output: the maximum revenue obtainable by cutting up the rod and selling the pieces.

Example.



inches	1	2	3
price	2	5	6



Revenue is 6.

Finding a way to cut a rod to maximize the revenue

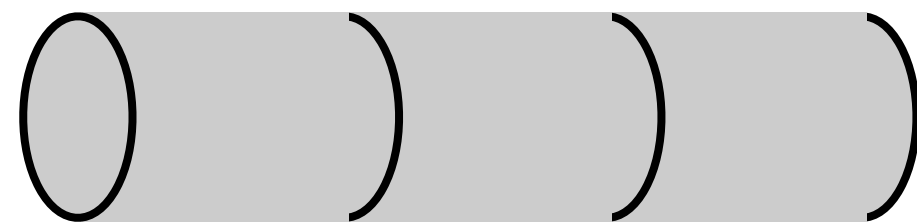
Input: given a rod of n inches and a table of prices p_i for $i = 1, 2, \dots, n$ where p_i denotes the price of a rod of i inches.

Output: the maximum revenue obtainable by cutting up the rod and selling the pieces.

Example.



inches	1	2	3
price	2	5	6



Revenue is 6.

Finding a way to cut a rod to maximize the revenue

Input: given a rod of n inches and a table of prices p_i for $i = 1, 2, \dots, n$ where p_i denotes the price of a rod of i inches.

Output: the maximum revenue obtainable by cutting up the rod and selling the pieces.

Example.



inches	1	2	3
price	2	5	6



Revenue is 7,
the maximum.

Divide and Conquer

```
rod_cutting(n){  
    int max_revenue = pn; // no cut  
    // guess that the length of first cut is i inches  
    foreach i ∈ {1, 2, ..., n-1}  
        if(pi + rod_cutting(n-i) > max_revenue){  
            max_revenue = pi + rod_cutting(n-i);  
        }  
    return max_revenue;  
}
```

rod_cutting(n) will invoke $2^{\Omega(n)}$ subproblems, but only $O(n)$ distinct ones.

Divide and Conquer + Memoization

```
rod_cutting(n, cached_solution[]){  
  
    if(cached_solution[n] ≥ 0)//cached_solution[] was -1 initially  
        return cached_solution[n];  
  
    int max_revenue = pn; // no cut  
    // guess that the length of first cut is i inches  
    foreach i ∈ {1, 2, ..., n-1}  
        if(pi + rod_cutting(n-i,cached_solution[]) > max_revenue){  
            max_revenue = pi + rod_cutting(n-i, cached_solution[]);  
        }  
    cached_solution[n] = max_revenue;  
    return cached_solution[n];  
}
```

$O(n)$ subproblems. Each needs $O(n)$ time to compute. In total, $O(n^2)$ time.

Longest Common Subsequence

Finding a longest common subsequence of two strings

Input: a string A of n characters and a string B of m characters.

Output: a common subsequence of A and B whose length is the longest.

Example.

z	s	c	a	d	a	c	s
---	---	---	---	---	---	---	---

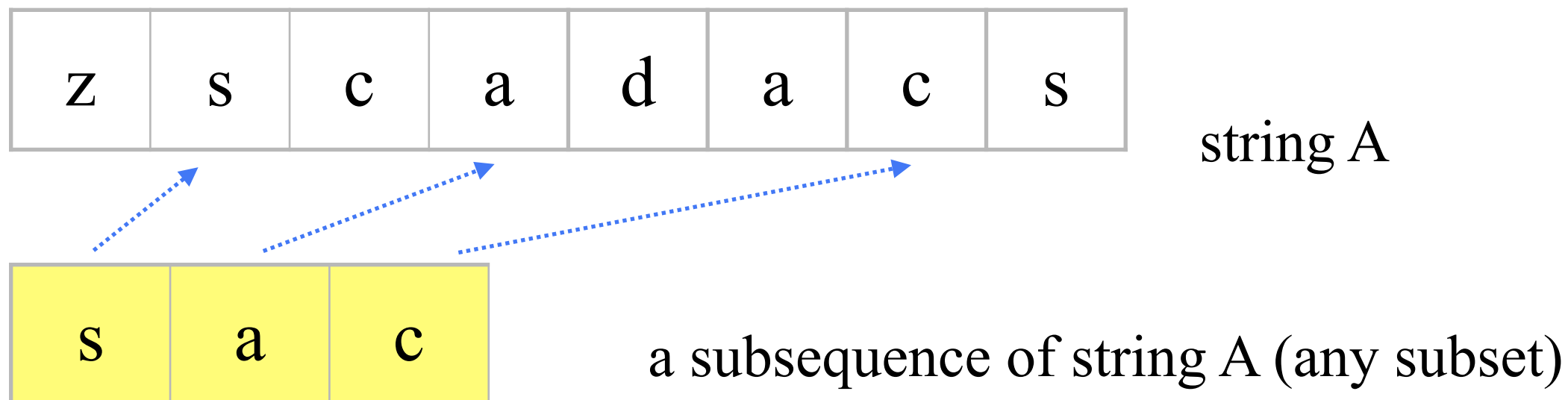
string A

Finding a longest common subsequence of two strings

Input: a string A of n characters and a string B of m characters.

Output: a common subsequence of A and B whose length is the longest.

Example.

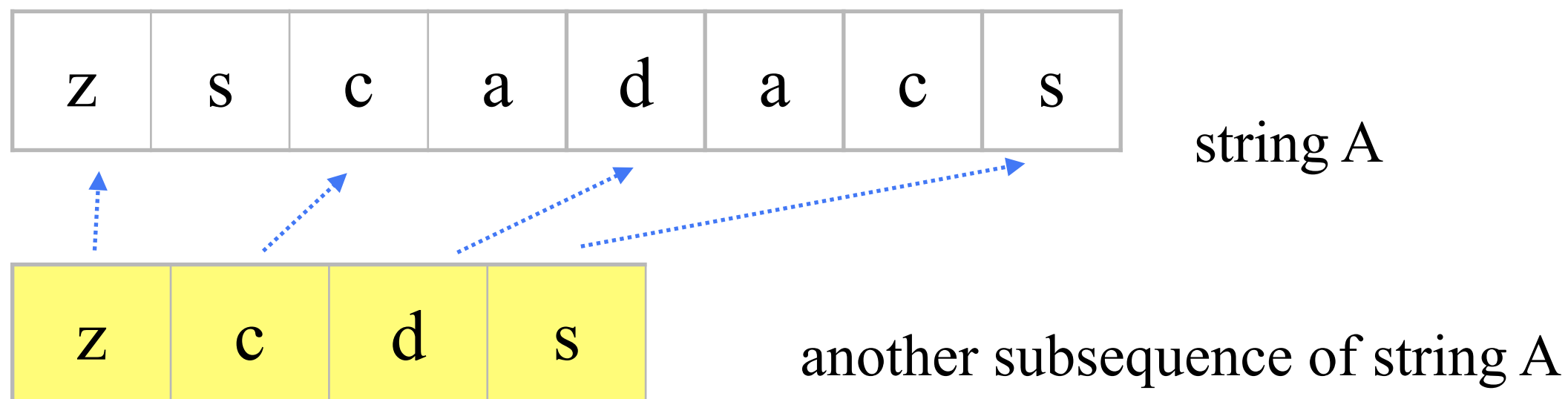


Finding a longest common subsequence of two strings

Input: a string A of n characters and a string B of m characters.

Output: a common subsequence of A and B whose length is the longest.

Example.



Finding a longest common subsequence of two strings

Input: a string A of n characters and a string B of m characters.

Output: a common subsequence of A and B whose length is the longest.

Example.

z	s	c	a	d	a	c	s
---	---	---	---	---	---	---	---

string A

String A is also a subsequence of itself.

Finding a longest common subsequence of two strings

Input: a string A of n characters and a string B of m characters.

Output: a common subsequence of A and B whose length is the longest.

Example.

A common subsequence.

z	s	c	a	d	a	c	s
---	---	---	---	---	---	---	---

string A

a	a	d	z	b	c	d	a
---	---	---	---	---	---	---	---

string B

Divide and Conquer

```
LCS(n, m){ // return the length of the LCS of string A and B
    if(n == 0 or m == 0) return 0;

    int max_length = 0;

    if(A[n] == B[m]) // if A[n] and B[m] is a part of LCS
        max_length = 1 + LCS(n-1, m-1);
    else // otherwise one of A[n], B[m] is not a part of LCS
        max_length = max(LCS(n-1, m), LCS(n, m-1));

    return max_length;
}
```

Divide and Conquer

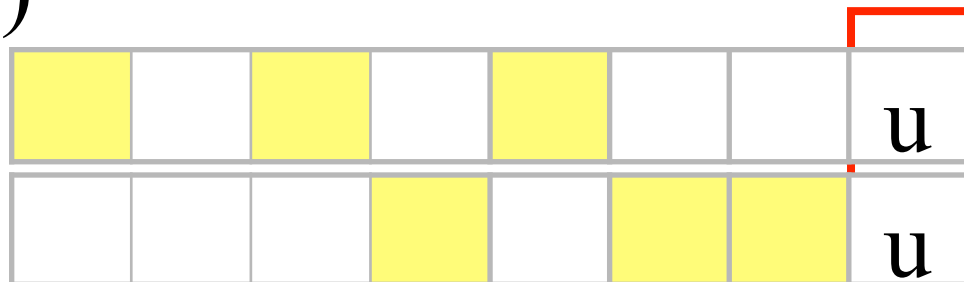
```
// if A[n] and B[m] is a part of LCS  
if(A[n] == B[m])  
    max_length = 1 + LCS(n-1, m-1);
```

Why does $\text{LCS}(n, m) = 1 + \text{LCS}(n-1, m-1)$
if $A[n] = B[m]$?

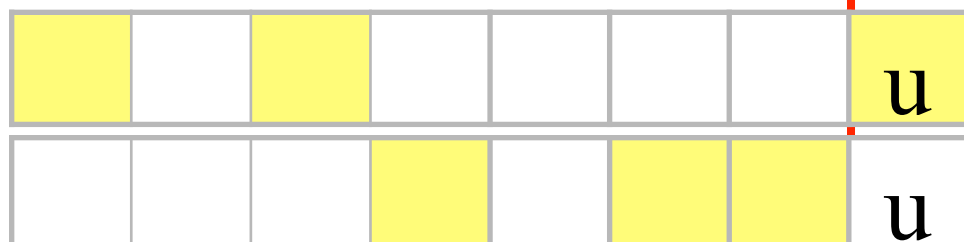
Correctness

Why does $\text{LCS}(n, m) = 1 + \text{LCS}(n-1, m-1)$
if $A[n] = B[m]$?

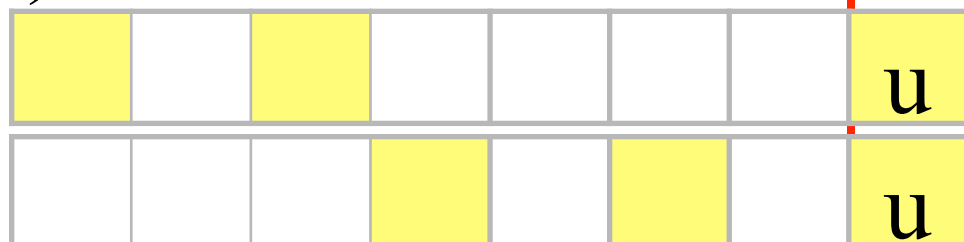
(i)



(ii)



(iii)

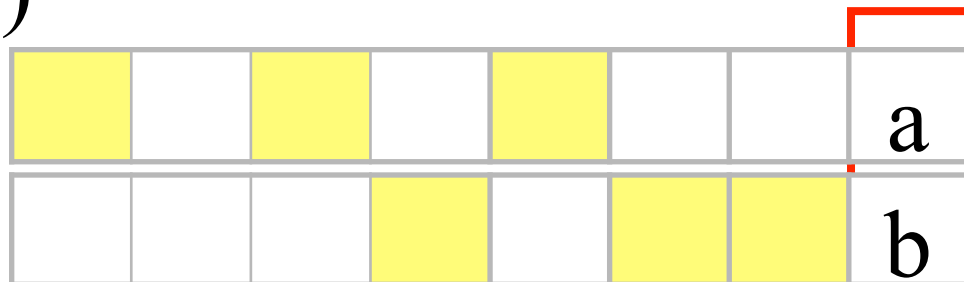


Every CS of $A[1..n]$ and $B[1..m]$ has
length at most $\text{LCS}(n, m) - 1$.

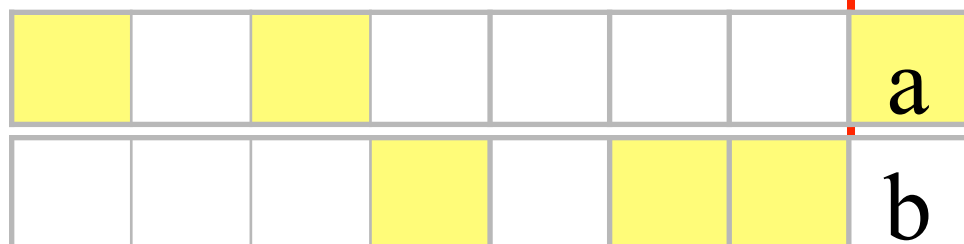
Correctness

Why does $\text{LCS}(n, m) = \max(\text{LCS}(n-1, m), \text{LCS}(n, m-1))$ otherwise?

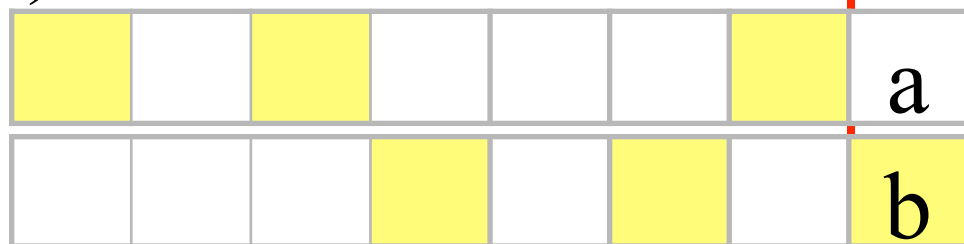
(i)



(ii)



(iii)



Divide and Conquer + Memoization

```
LCS(n, m, sol[][]){//return the length of the LCS of A and B
    if(n == 0 or m == 0) return 0;
    if(sol[n][m] ≥ 0) return sol[n][m]; // sol[][] was -1 initially

    int max_length = 0;

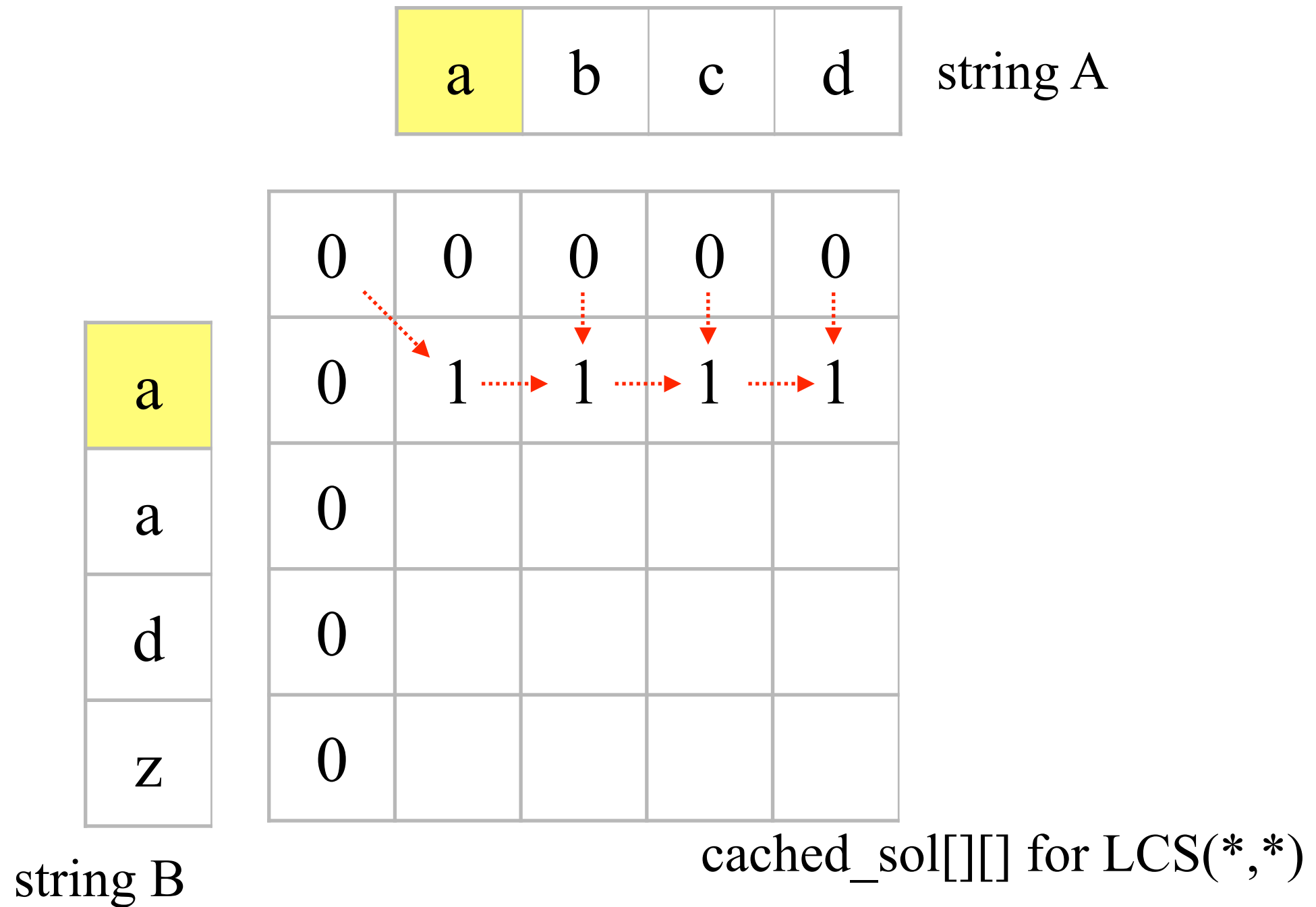
    if(A[n] == B[m]) // if A[n] and B[m] is a part of LCS
        max_length = 1 + LCS(n-1, m-1);
    else // otherwise one of A[n], B[m] is not a part of LCS
        max_length = max(LCS(n-1, m), LCS(n, m-1));

    return s[n][m] = max_length; // memoize the solution
}
```

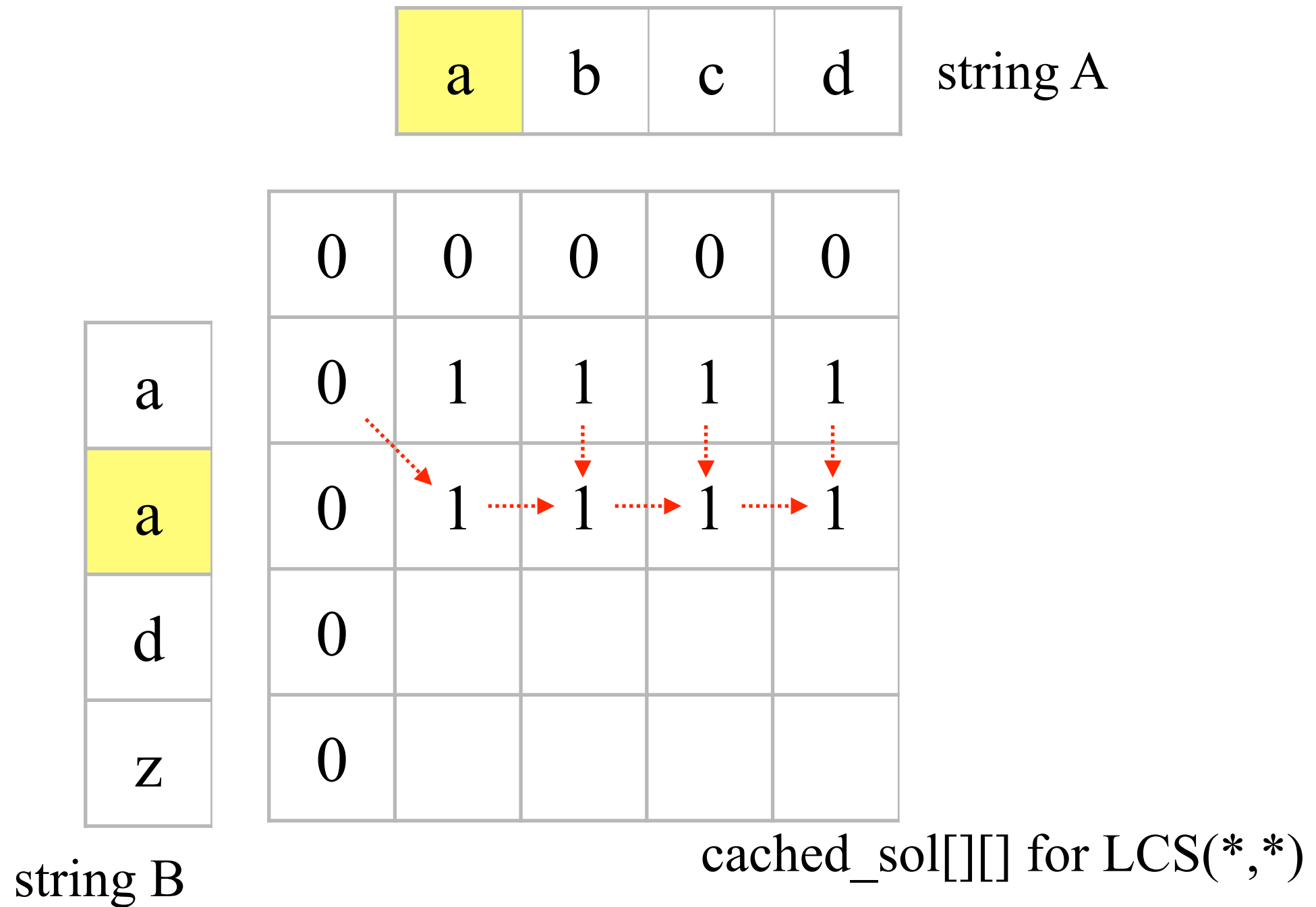
Alternatively, we could iteratively fill the table

		a	b	c	d	string A
		0	0	0	0	0
	a	0				
	a	0				
	d	0				
	z	0				
string B		cached_sol[][] for LCS(*,*)				

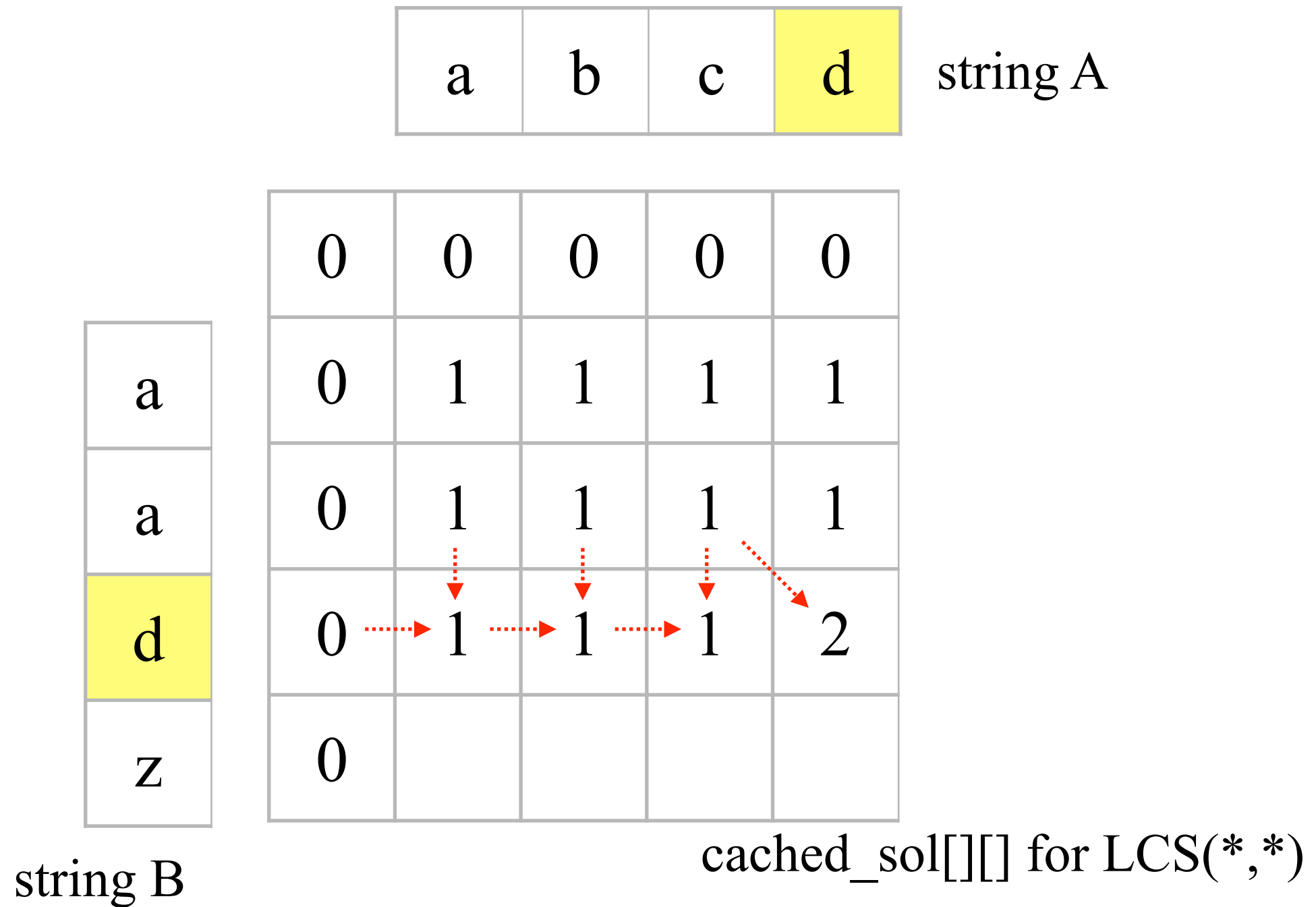
Alternatively, we could iteratively fill the table



Alternatively, we could iteratively fill the table



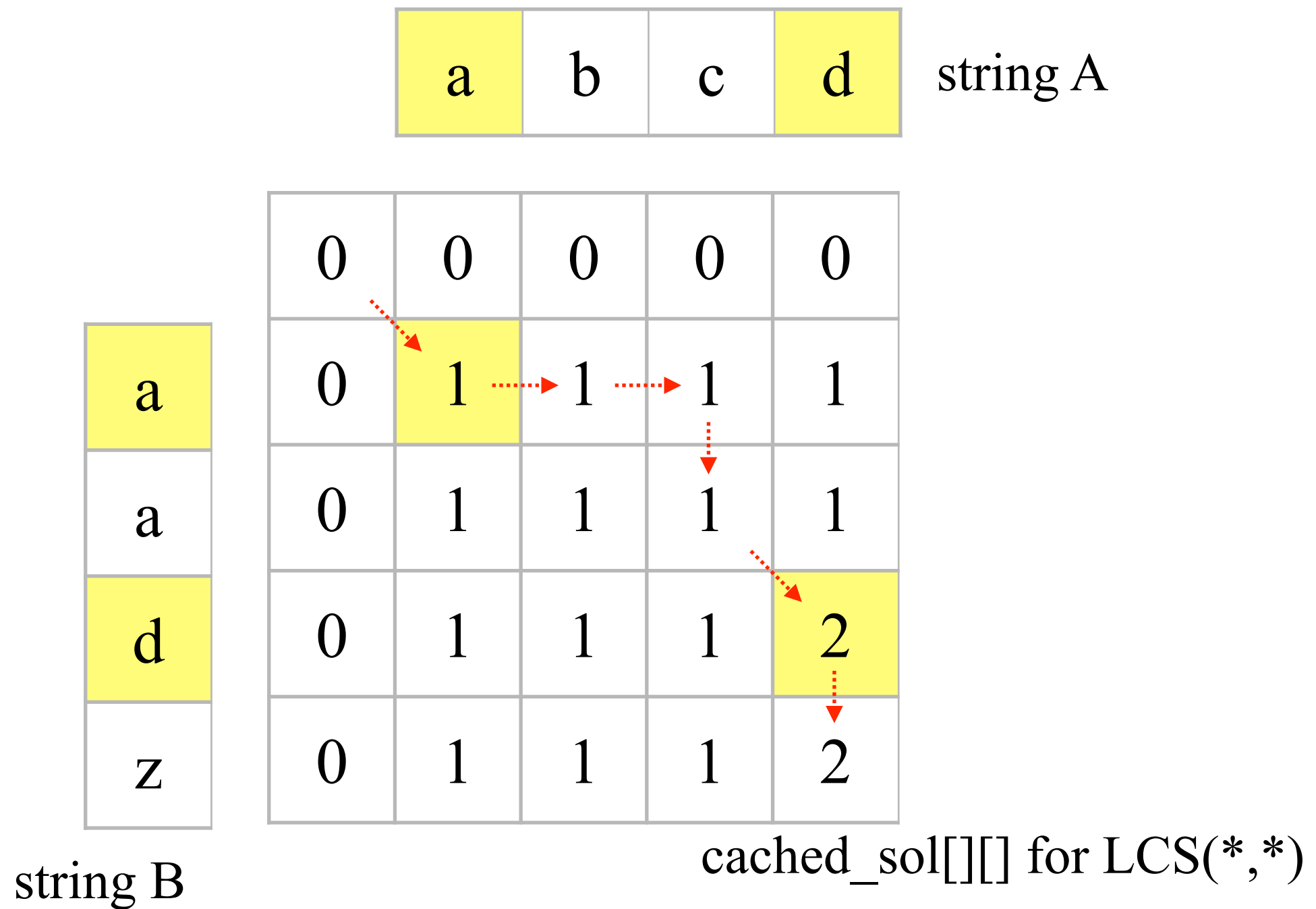
Alternatively, we could iteratively fill the table



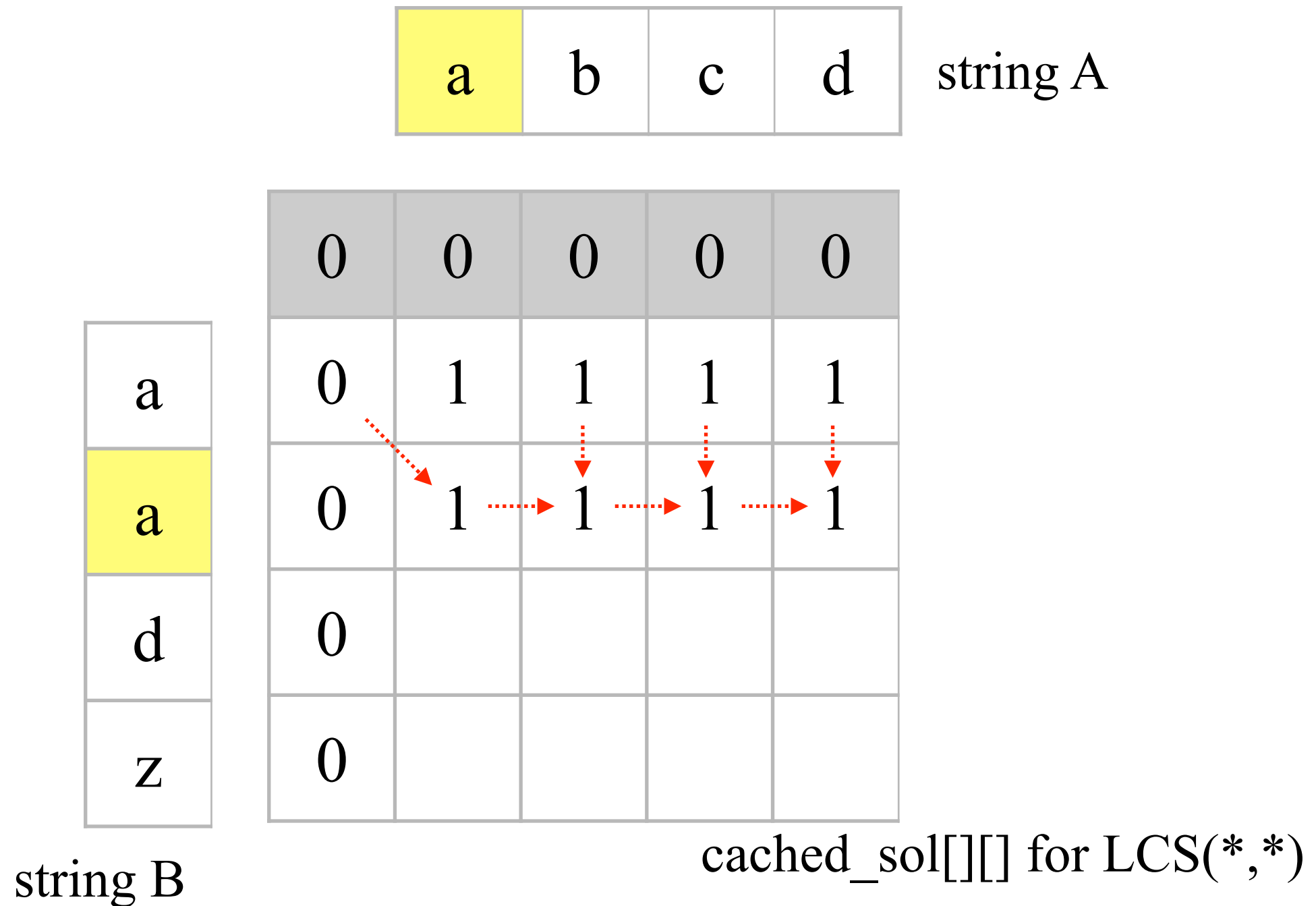
Alternatively, we could iteratively fill the table

		a	b	c	d	string A
		0	0	0	0	0
	a	0	1	1	1	1
	a	0	1	1	1	1
	d	0	1	1	1	2
	z	0	1	1	1	2
string B						cached_sol[][] for LCS(*,*)

Finding a LCS rather than only its length

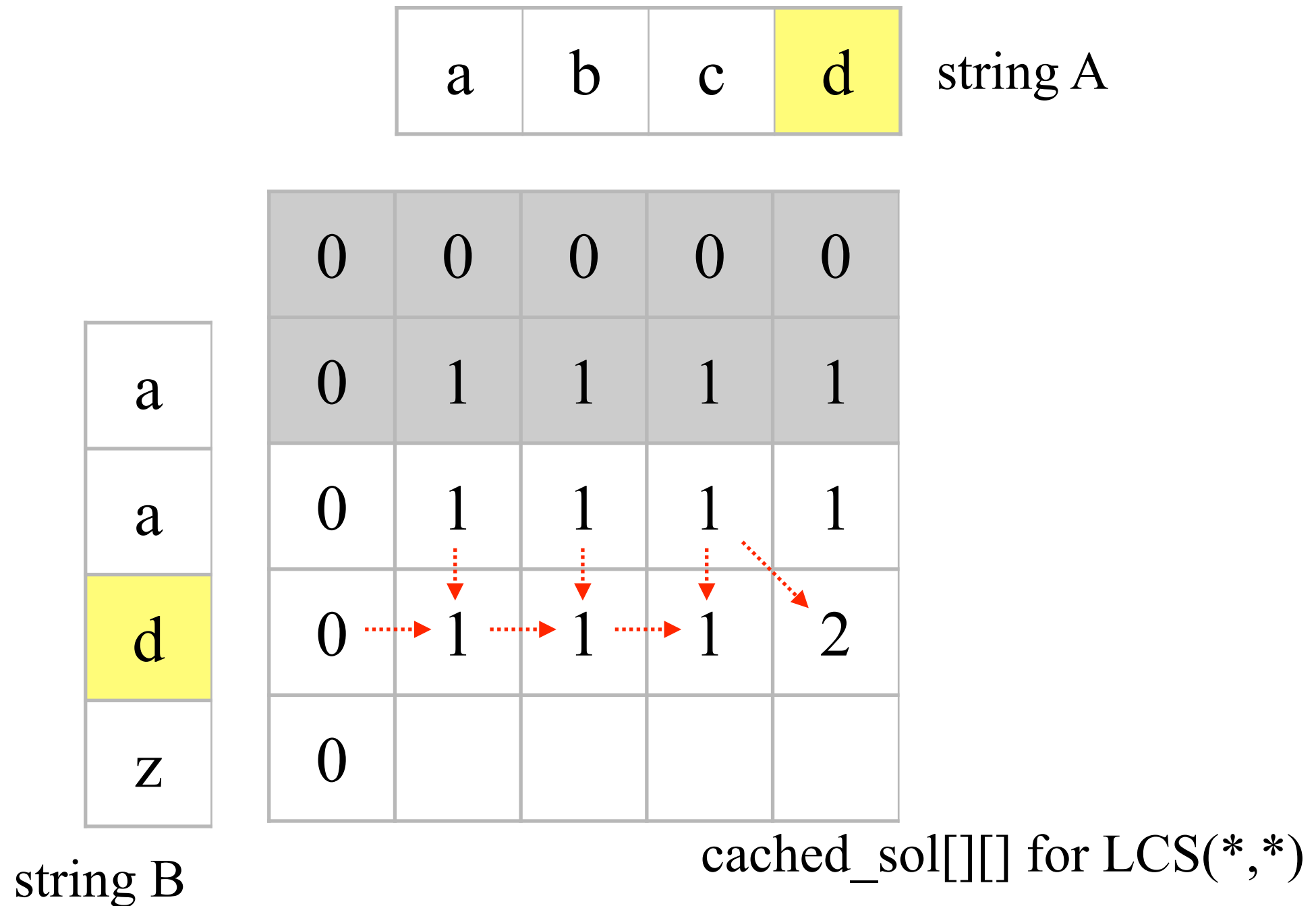


Reducing the working space



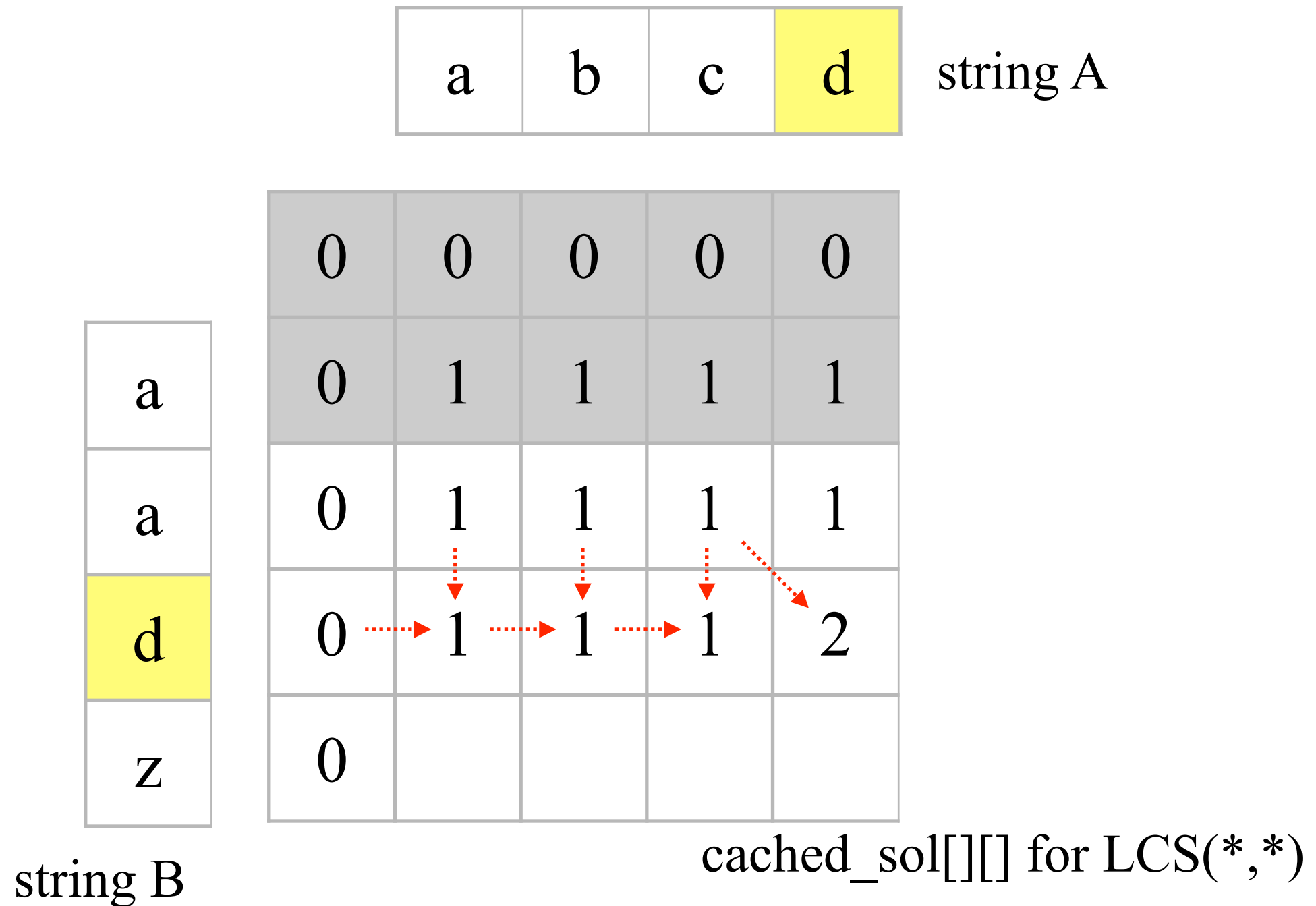
While filling the entries on the 3rd row, the rows above the 2nd row is no longer needed.

Reducing the working space



While filling the entries on the 4th row, the rows above the 3rd row is no longer needed.

Reducing the working space



Only two rows is needed at every time step.

Exercise

Finding the longest monotonic increasing subsequence of an array of real numbers.

(Hint. Reduce to LCS.)

Example.

12	4	13	9	10	1	2	15
----	---	----	---	----	---	---	----