Introduction to Algorithms

Meng-Tsung Tsai

10/08/2019

Announcements

Programming Assignment 1 is due by Oct 9, 23:59. at https://oj.nctu.me

Written Assignment 1 is due by Oct 15, 10:20. You need hand in your writeup in class. No late submissions because the solution will be announced right after the deadline. at https://e3new.nctu.edu.tw

Programming Assignment 2 is due by Oct 29, 23:59. at https://oj.nctu.me

We will not normalize the points that you receive from assignments. 100 points is a perfect score, and extra points are considered as a bonus.

Caution: it is very difficult to solve all problems in an assignment.

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 $n \geq 1$.

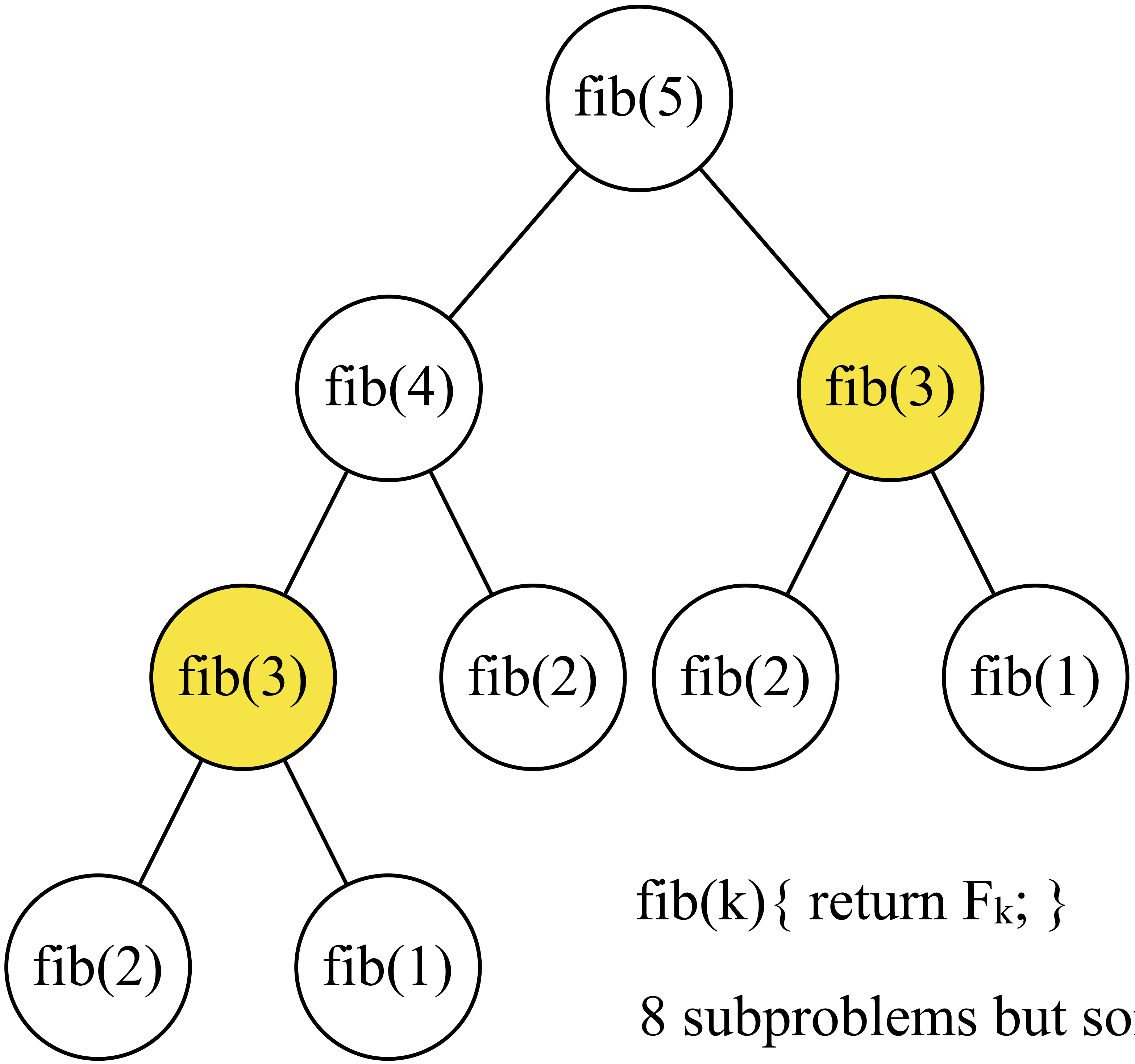
Input: an integer k.

Output: F_k.

--- Example ---

$F_3 = F_2 + F_1 = 2$.
$F_5 = F_4 + F_3 = (F_3 + F_2) + F_3 = (F_2 + F_1) + F_2 + (F_2 + F_1) = 5$.

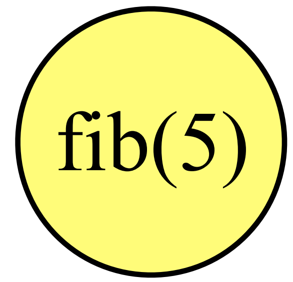
Divide and Conquer + Memoization



fib(k){ return F_k; }

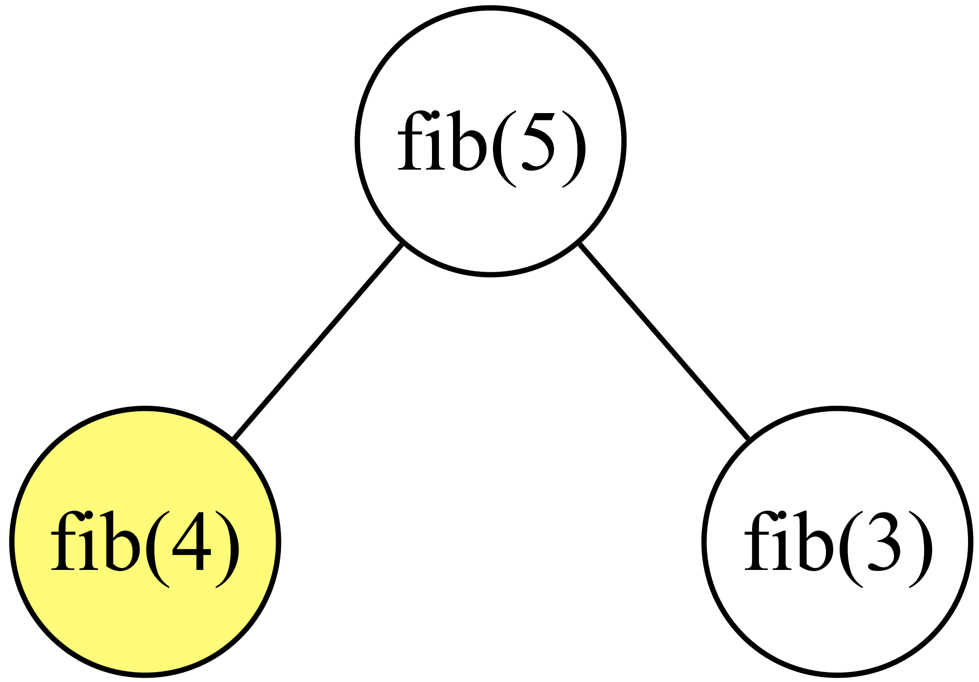
8 subproblems but some are duplicates.

fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1

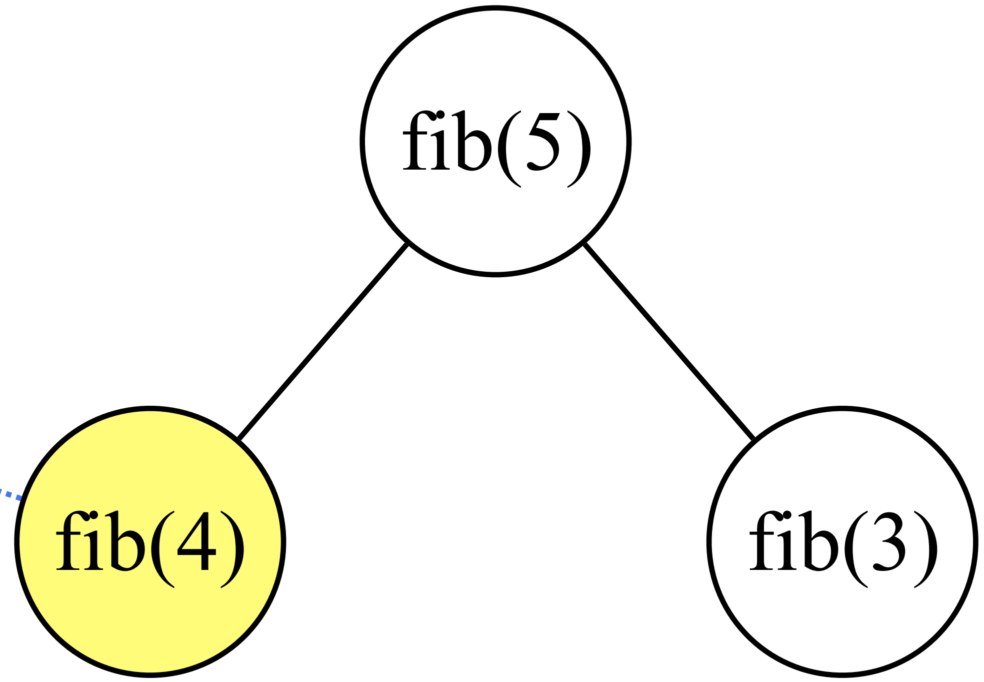


fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1

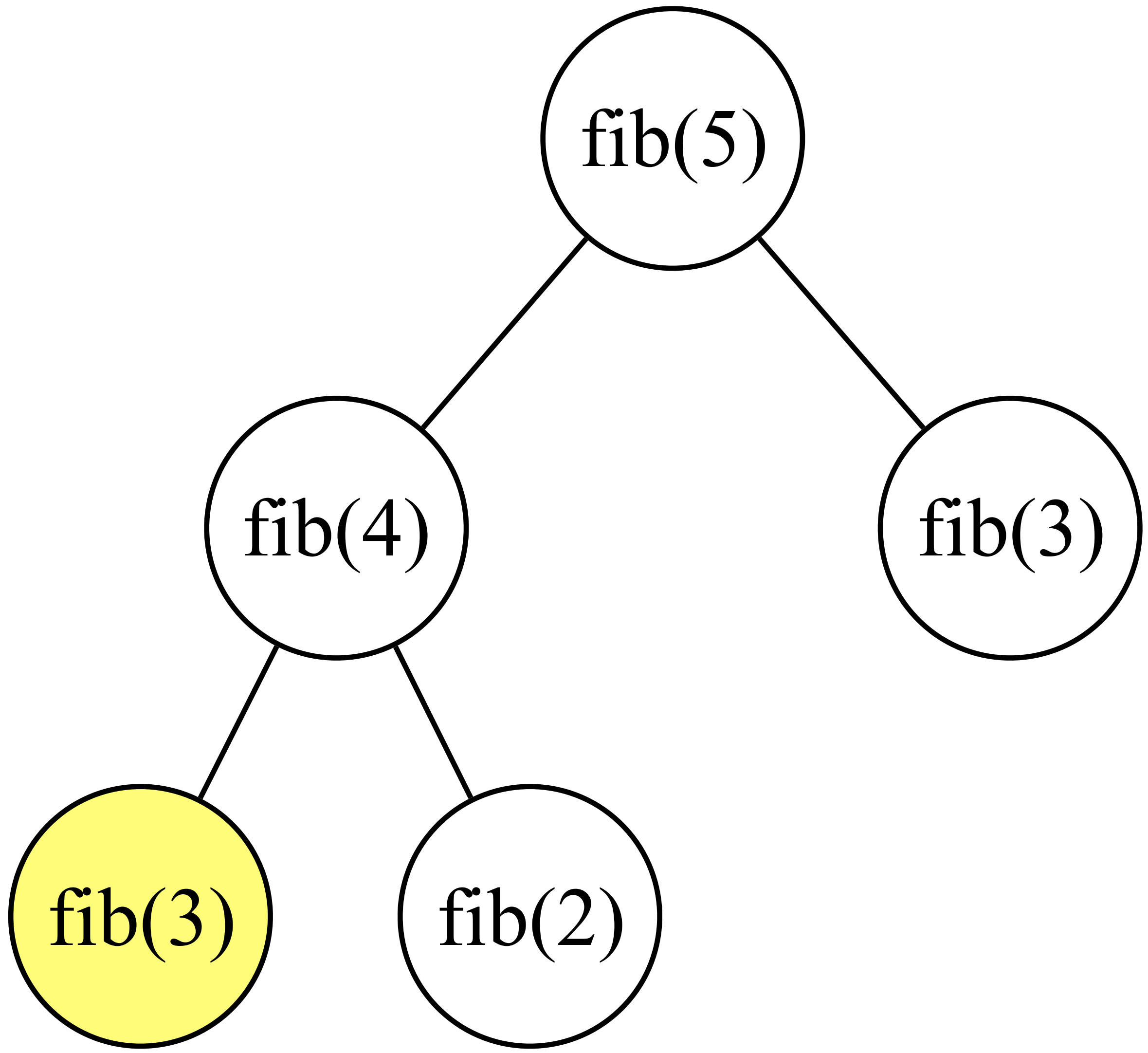
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



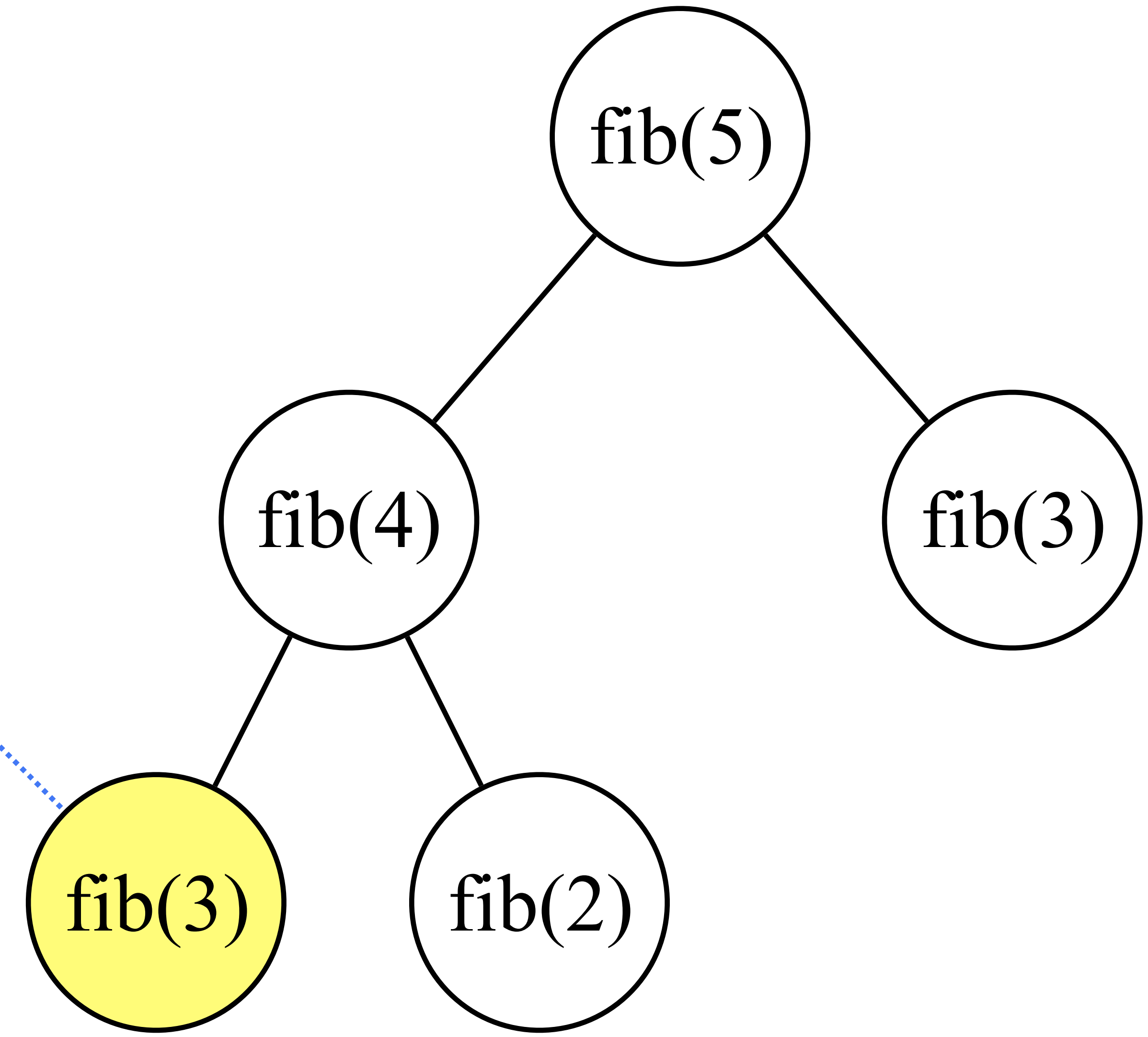
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



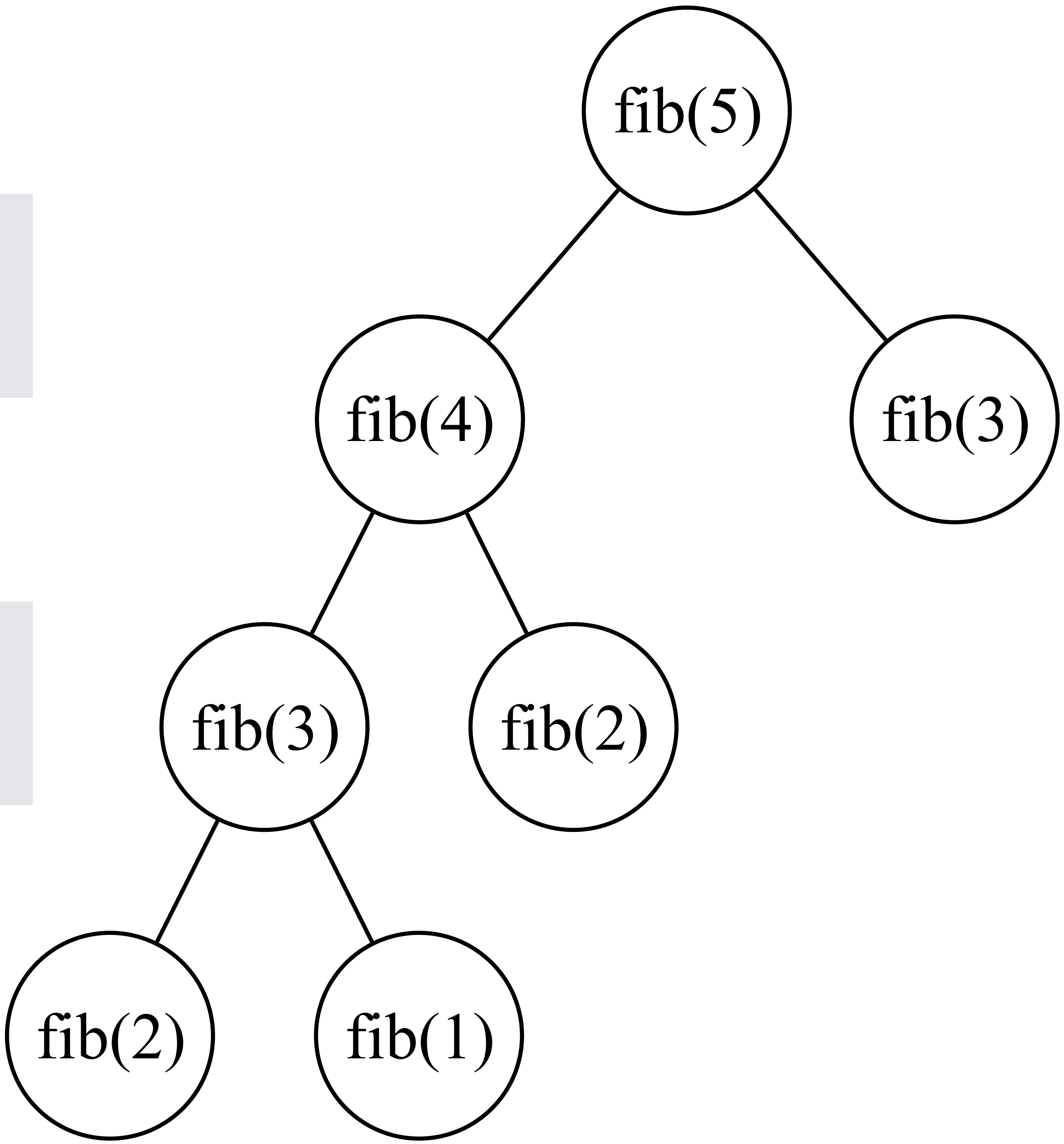
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



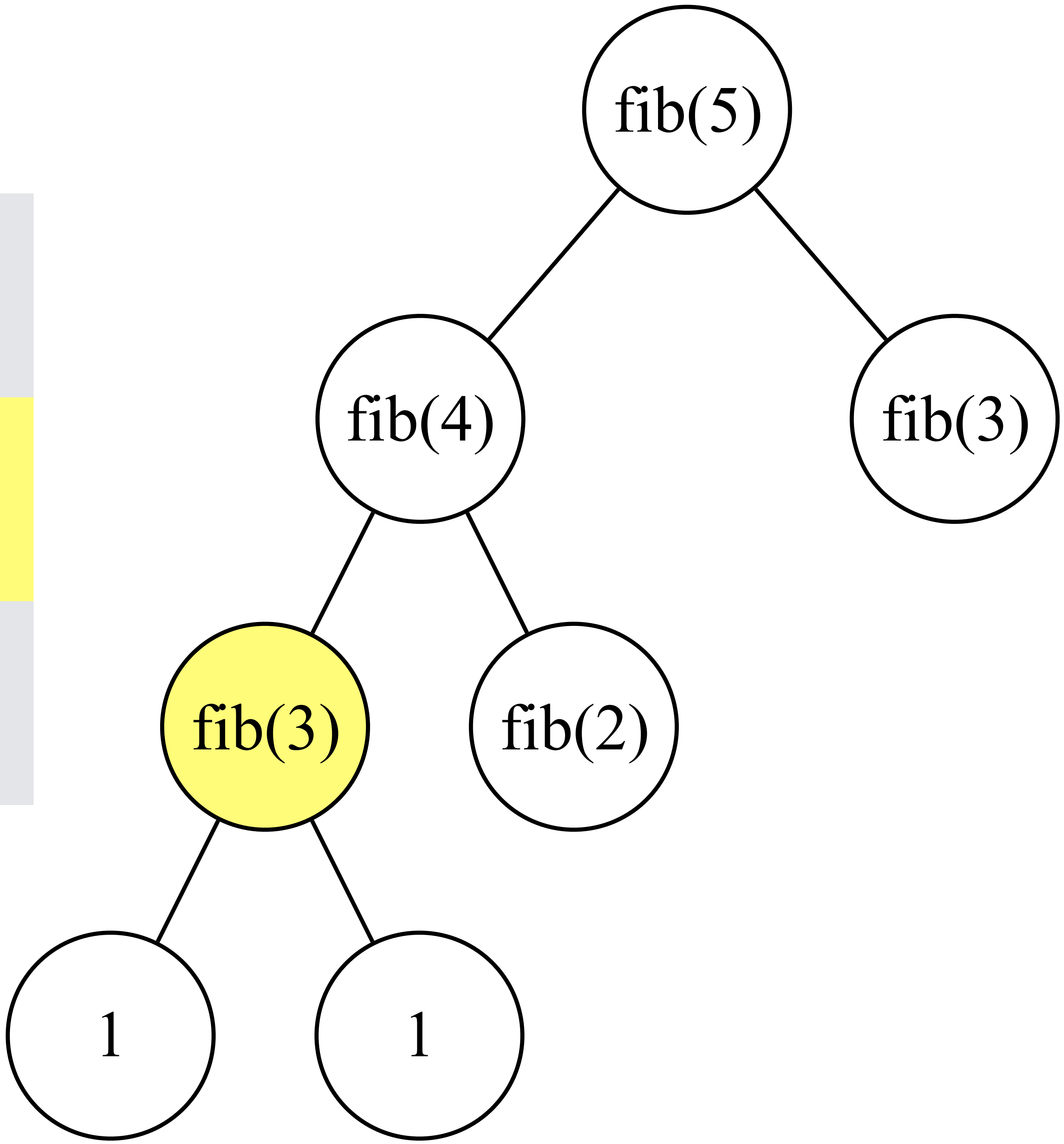
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



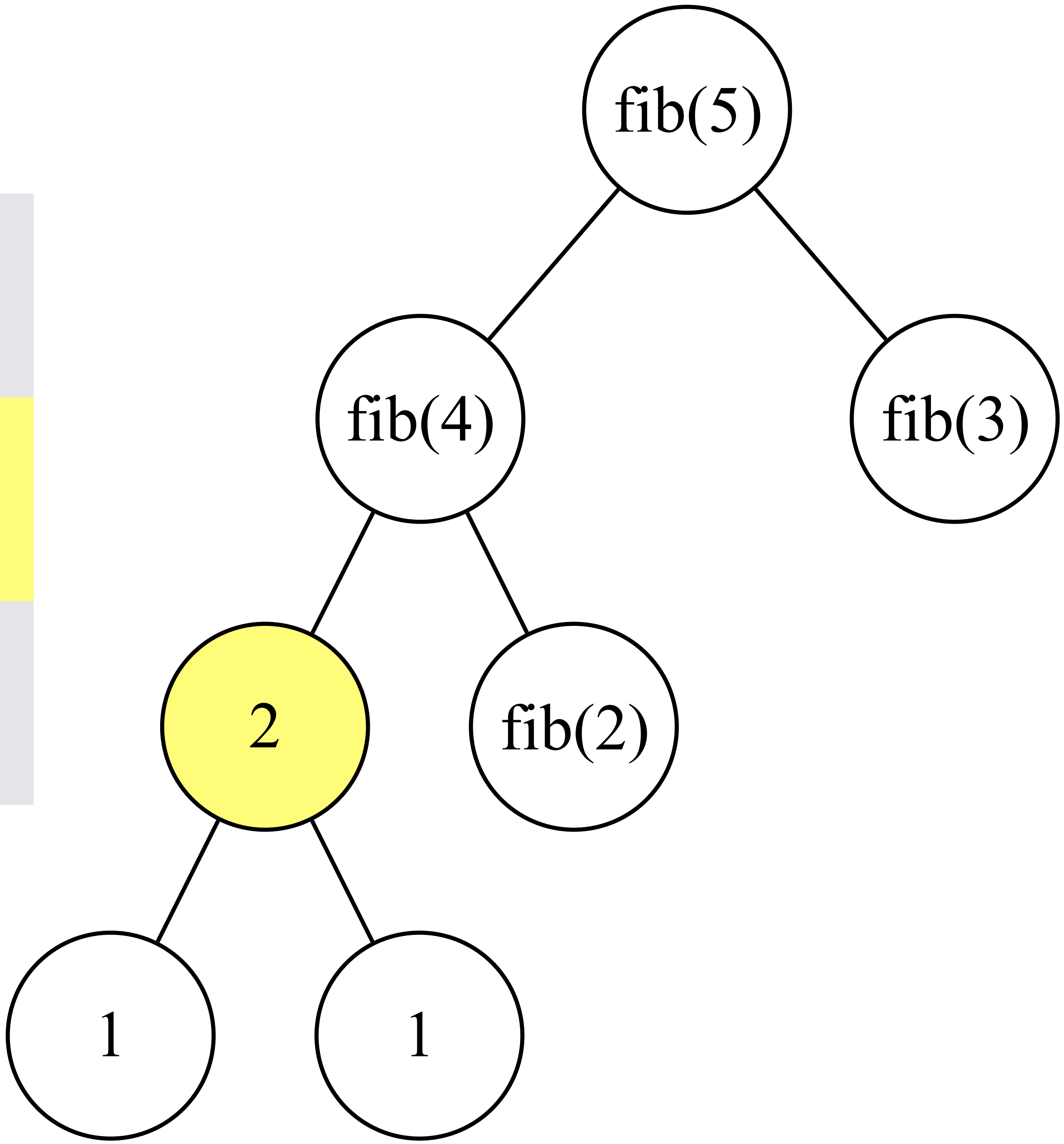
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



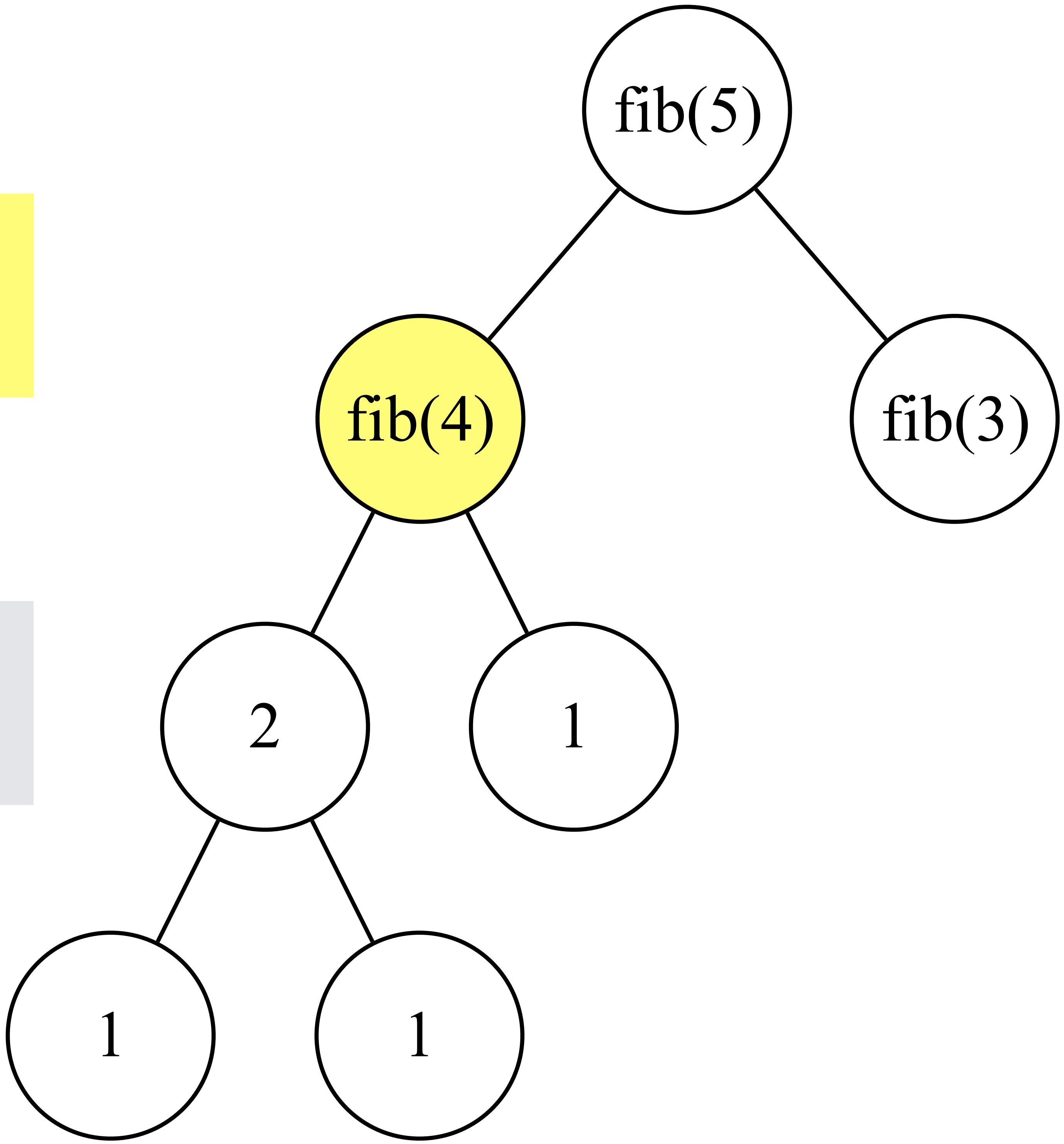
fib(5)	?
fib(4)	?
fib(3)	?
fib(2)	1
fib(1)	1



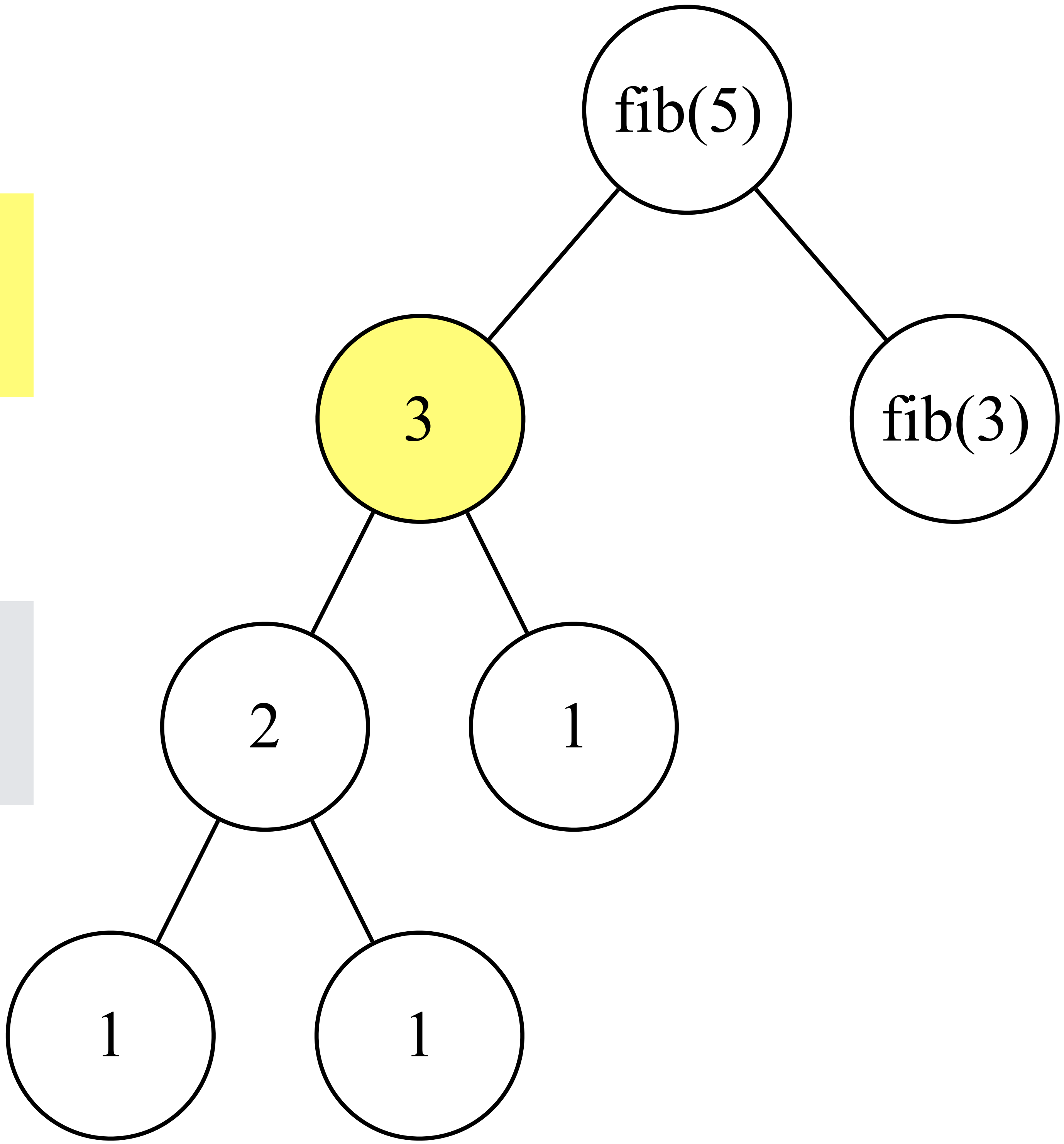
fib(5)	?
fib(4)	?
fib(3)	2
fib(2)	1
fib(1)	1



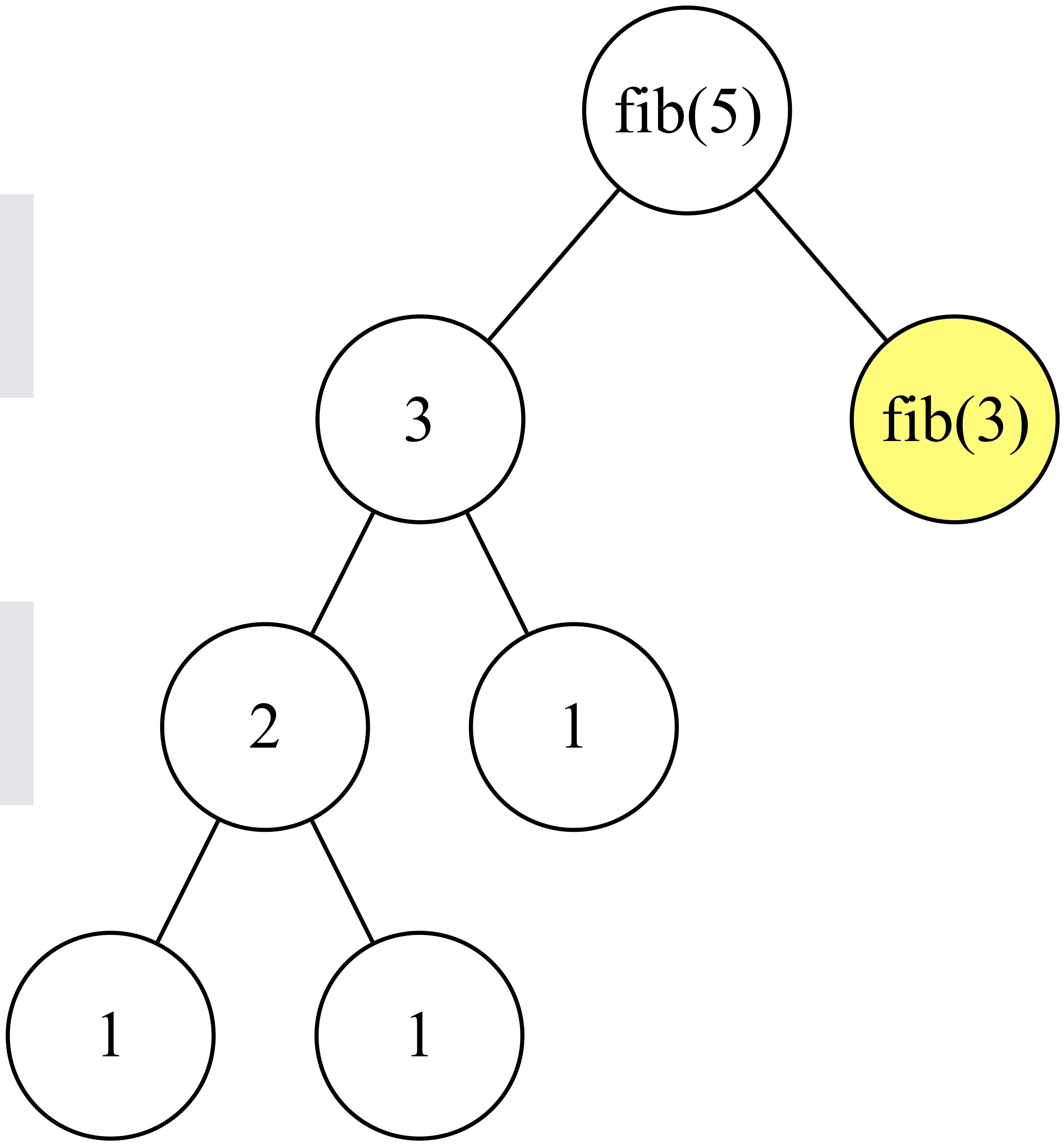
fib(5)	?
fib(4)	?
fib(3)	2
fib(2)	1
fib(1)	1



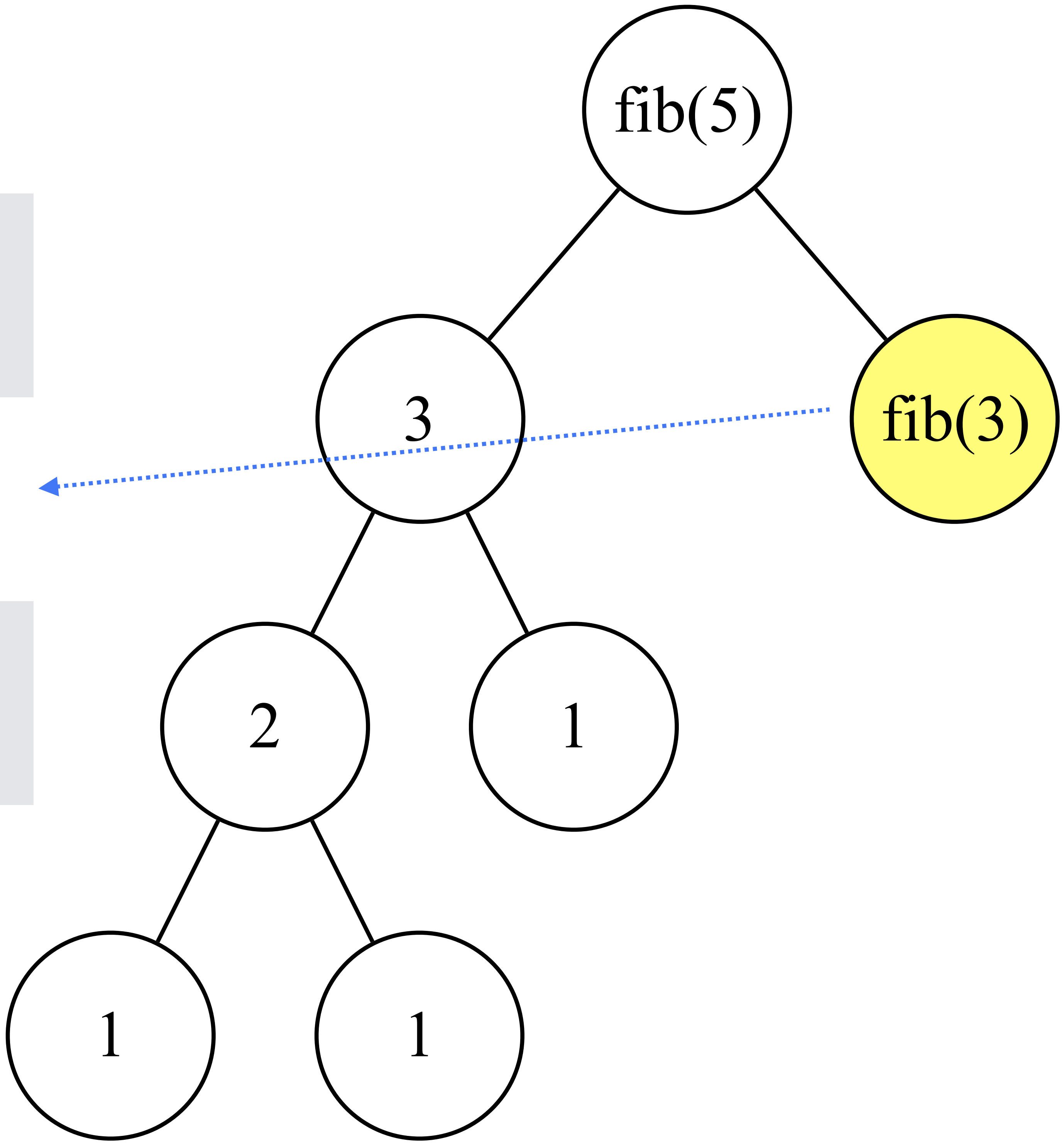
fib(5)	?
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1



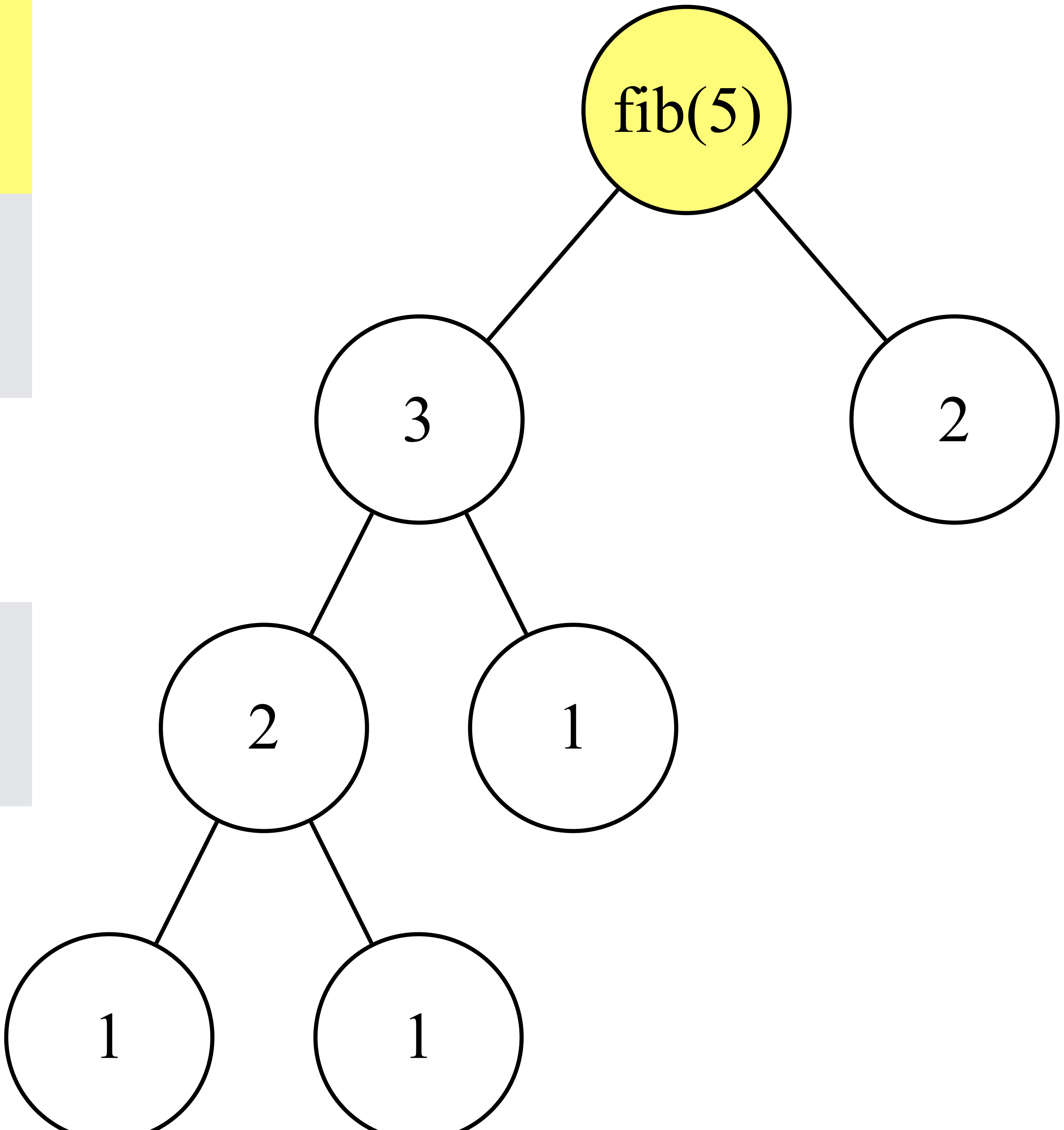
fib(5)	?
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1



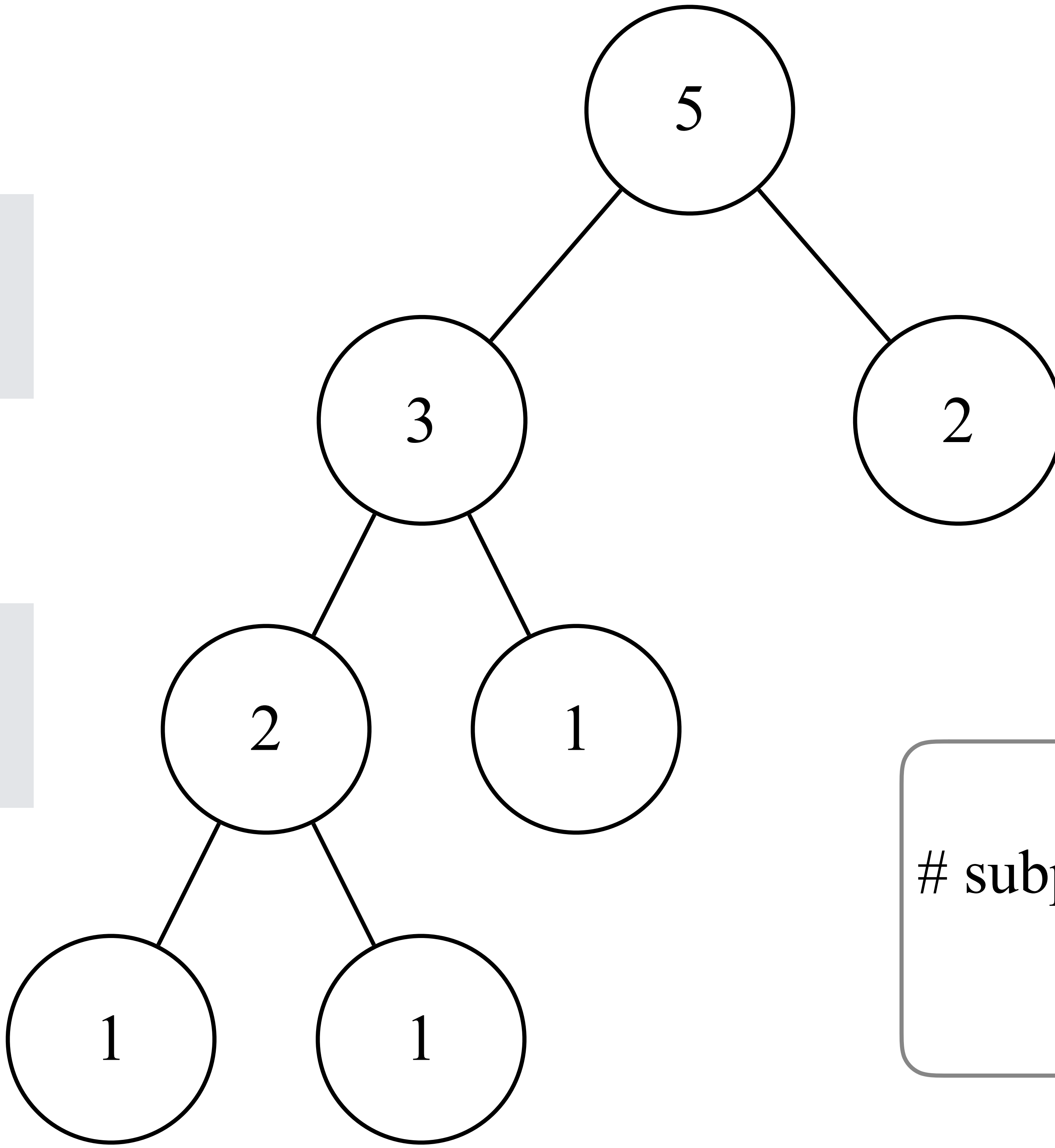
fib(5)	?
fib(4)	3
fib(3)	2
fib(2)	1
fib(1)	1



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.

Exercise

Prove that it needs $2^{\Omega(k)}$ time to compute F_k by the divide and conquer approach, and $O(k)$ time by dynamic programming.

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

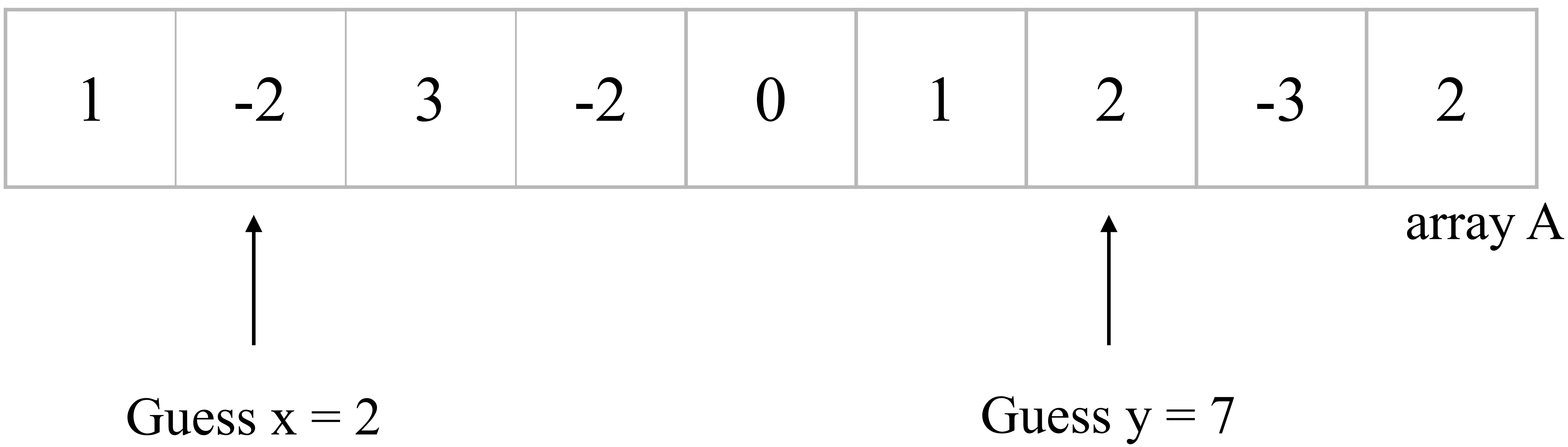
$a, b \in \{1, 2, \ldots, n\}$, $a \leq b$.

Example.

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

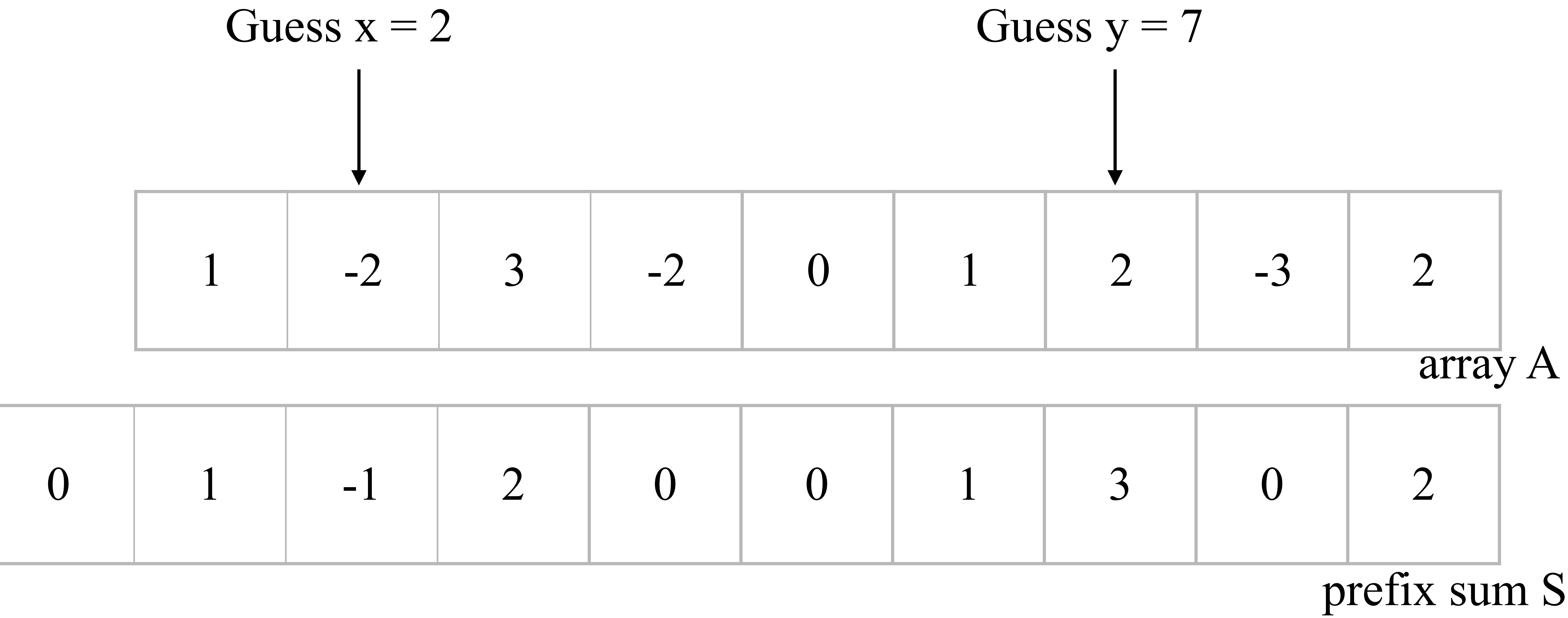
array A

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



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

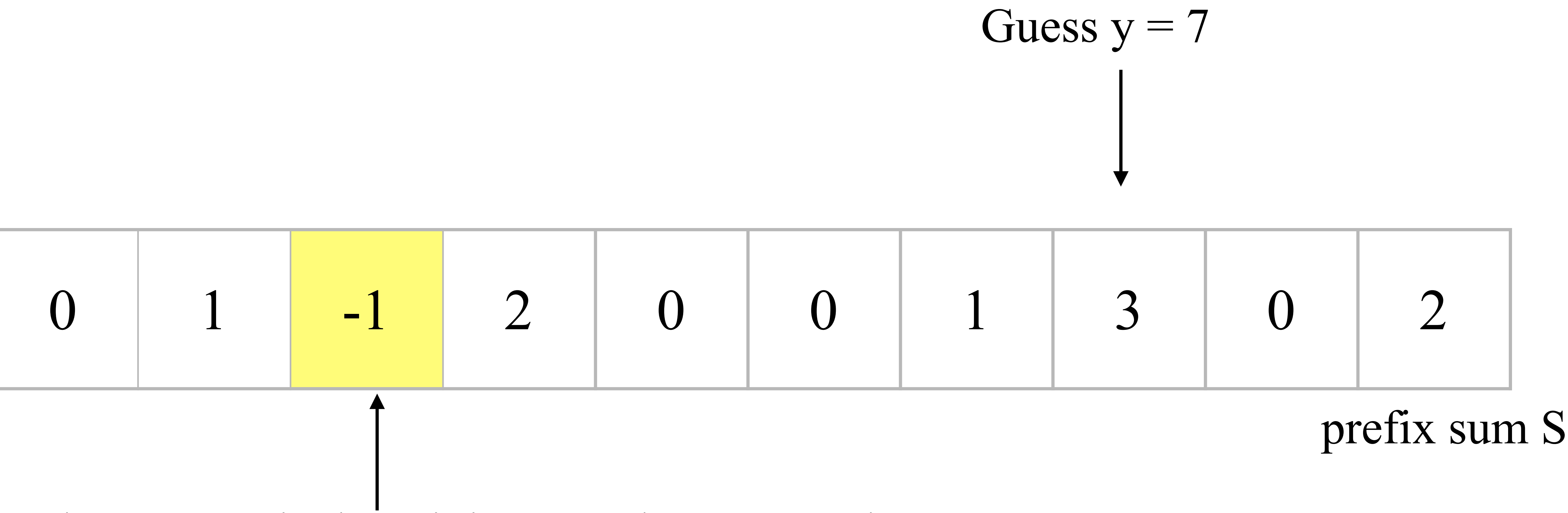
0	1	-1	2	0	0	1	3	0	2

prefix sum S

When S[x-1] is the minimum value among those preceding 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



When S[x-1] is the minimum value among those preceding 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.

Exercise

Input: an n by n matrix A of real numbers

Output: a contiguous submatrix with max sum

<u>Example</u>.

1	-2	3	-2
-2	1	2	0
3	0	3	-1
-2	-1	0	4

Is it possible to solve it in $O(n_3)$ time?

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 $i = 1, 2, \ldots, n$ 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.

<u>Example</u>.

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, …, 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

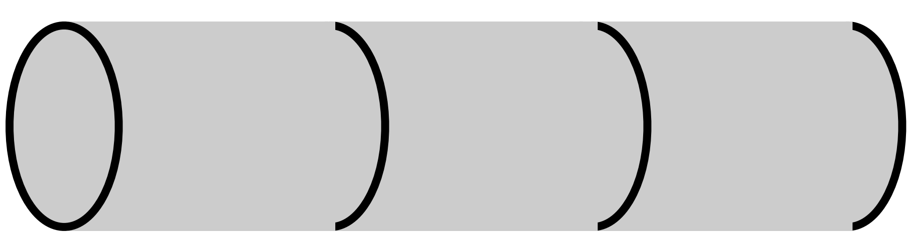
Revenue is 6.

Finding a way to cut a rod to maximize the revenue

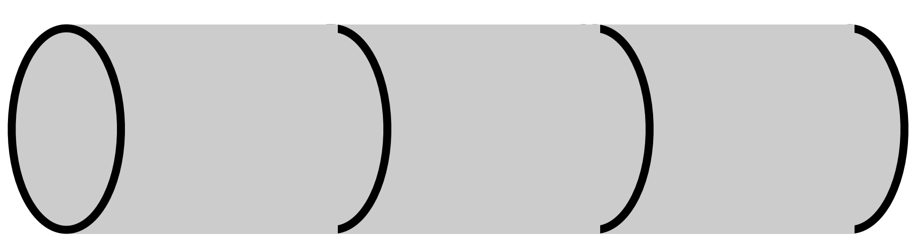
Input: given a rod of n inches and a table of prices p_i for i = 1, 2, …, n where p_i denotes the price of a rod of i inches.

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

Example.



inches	1	2	3
price	2	5	6



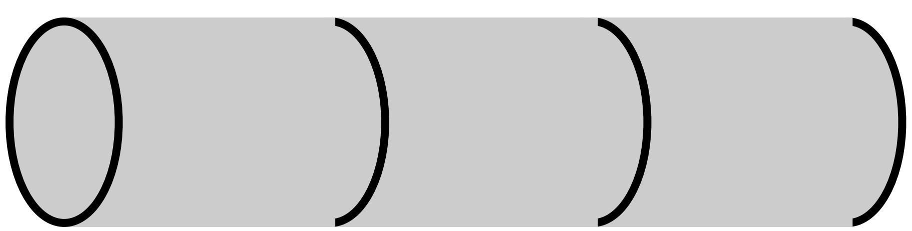
Revenue is 6.

Finding a way to cut a rod to maximize the revenue

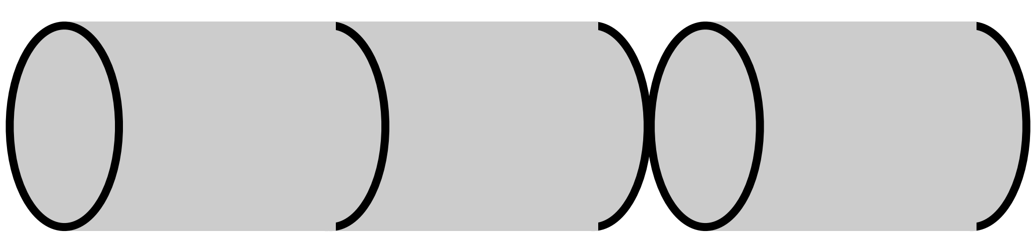
Input: given a rod of n inches and a table of prices p_i for $i = 1, 2, \ldots, n$ where p_i denotes the price of a rod of i inches.

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 = p_n; // no cut
    // guess that the length of first cut is i inches
    foreach i ∈ {1, 2, …, n-1}
      if(p_i + rod_cutting(n-i) > max_revenue){
          max_revenue = p_i + 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 = p_n; // no cut
    // guess that the length of first cut is i inches
    foreach i ∈ {1, 2, …, n-1}
      if(p_i + rod_cutting(n-i,cached_solution[]) > max_revenue){
          max_revenue = p_i + 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

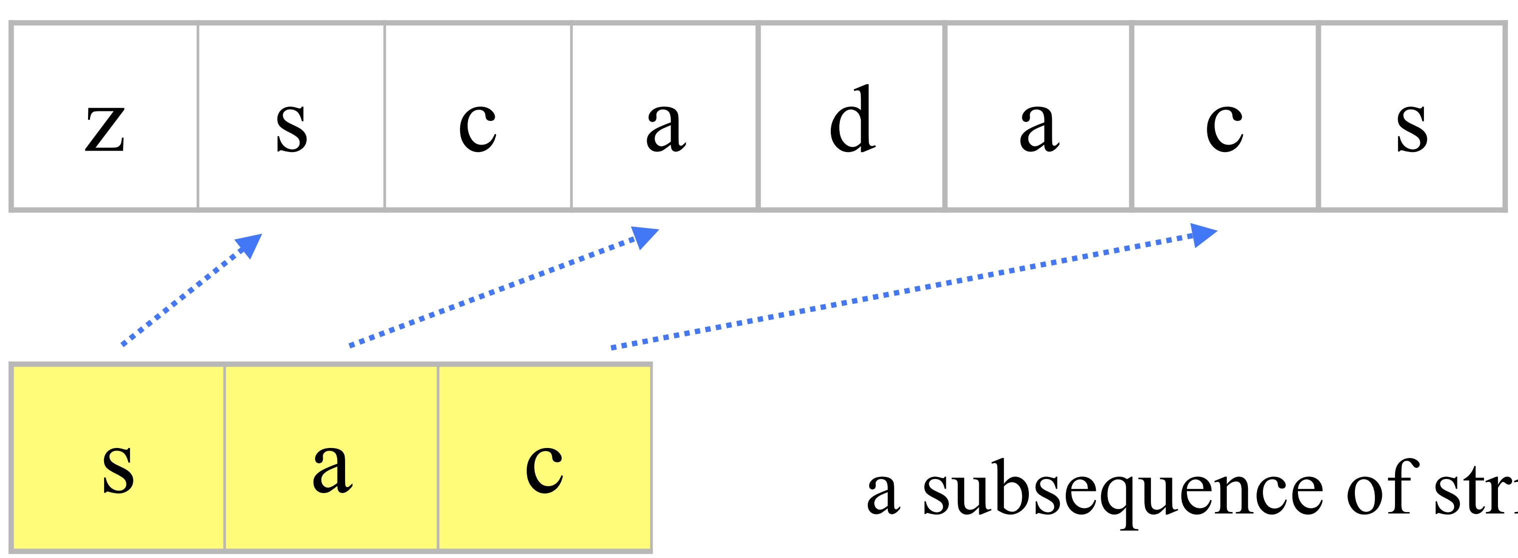
string A

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.



string A

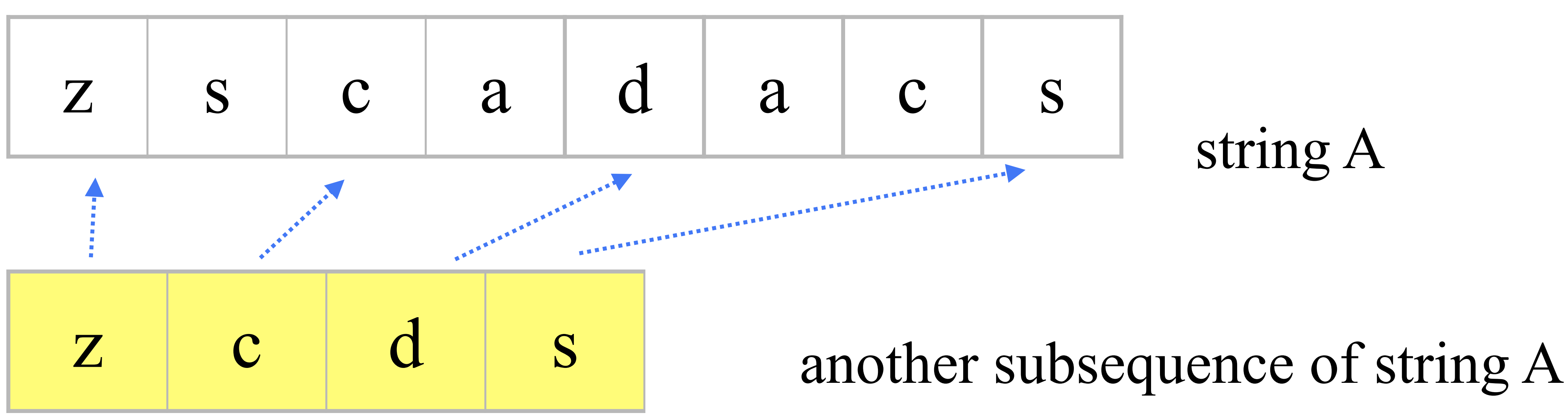
a subsequence of string A (any subset)

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.

<u>Example</u>.



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;
}
```

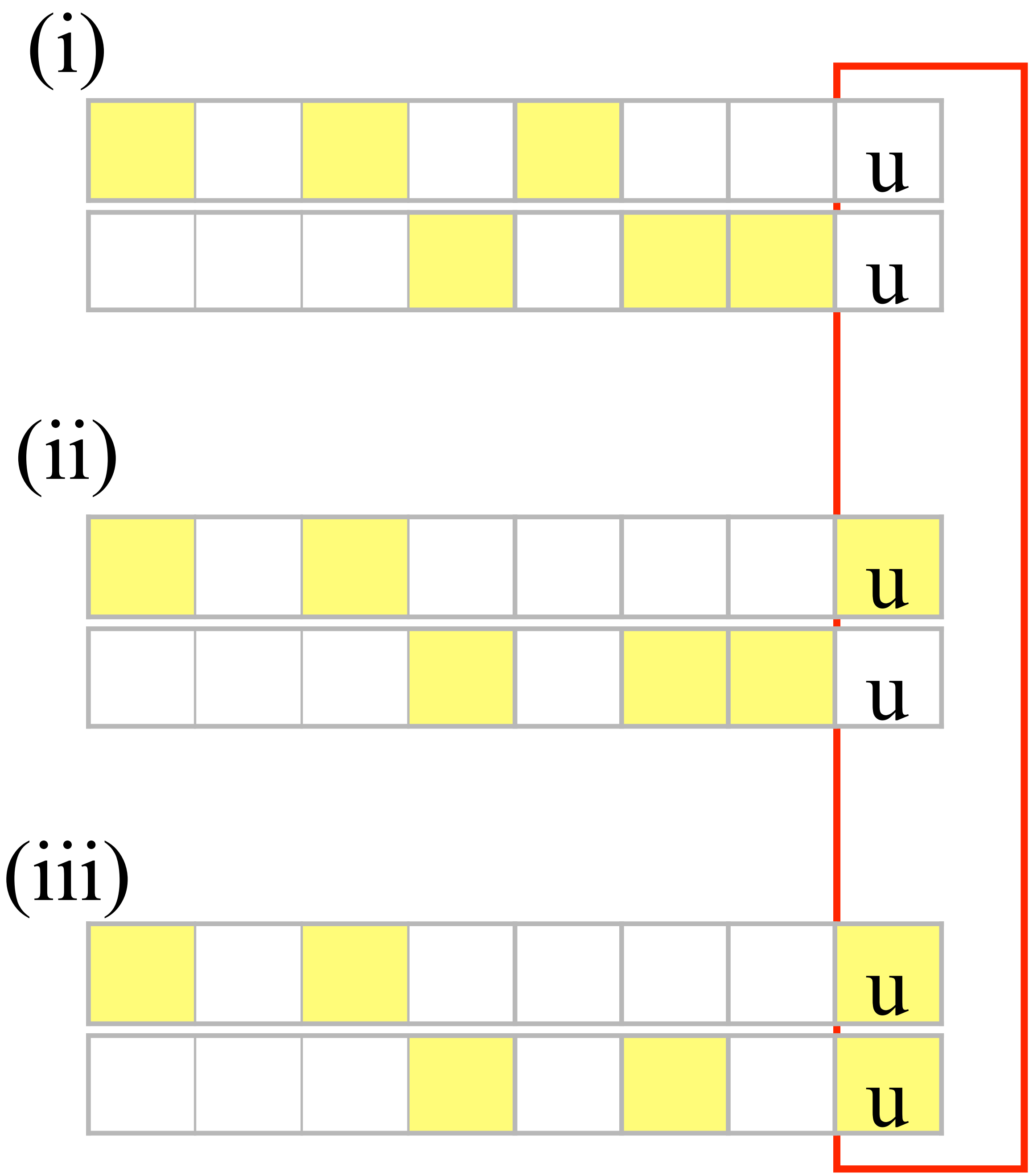
Divide and Conquer

```
// 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 LCS(n, m) = 1+LCS(n-1, m-1)
if A[n] = B[m]?

Correctness

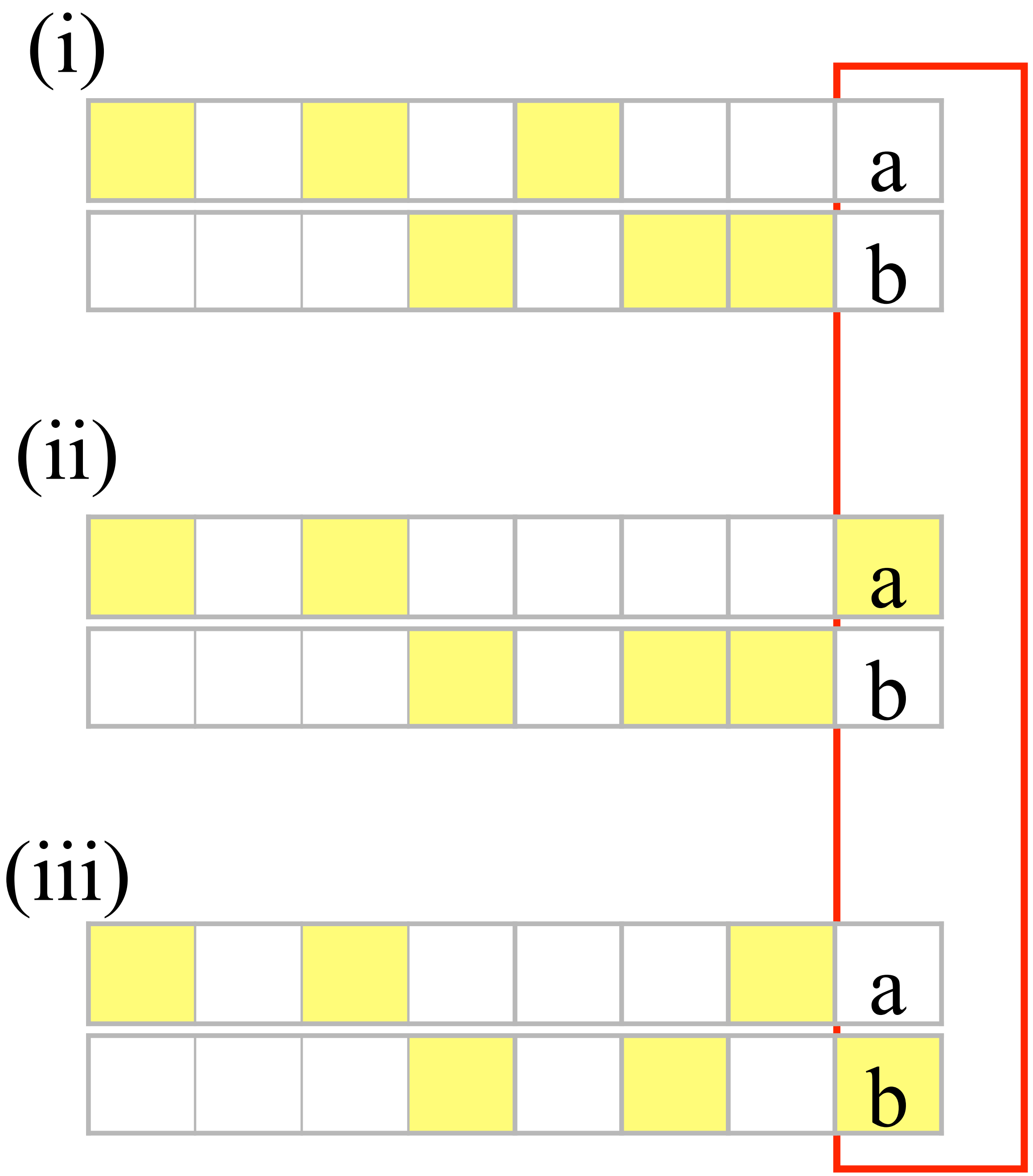
Why does LCS(n, m) = 1+LCS(n-1, m-1)
if A[n] = B[m]?



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

Correctness

Why does LCS(n, m) = max(LCS(n-1, m), LCS(n, m-1)) otherwise?



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 iterativelly fill the table

a	b	c	d

string A

0	0	0	0	0
0				
0				
0				
0				

a
a
d
z

string B

cached_sol[][] for LCS(*,*)

Alternatively, we could iterativelly fill the table

a	b	c	d

string A

a
a
d
z

string B

0	0	0	0	0
0	1	1	1	1
0				
0				
0				

cached_sol[][] for LCS(*,*)

Alternatively, we could iterativelly fill the table

a	b	c	d

string A

0	0	0	0	0
0	1	1	1	1
0	1	1	1	1
0				
0				

cached_sol[][] for LCS(*,*)

a
a
d
z

string B

Alternatively, we could iterativelly fill the table

string B		a	b	c	d
	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				

string A

string B

cached_sol[][] for LCS(*,*)

Alternatively, we could iterativelly fill the table

a	b	c	d	string A

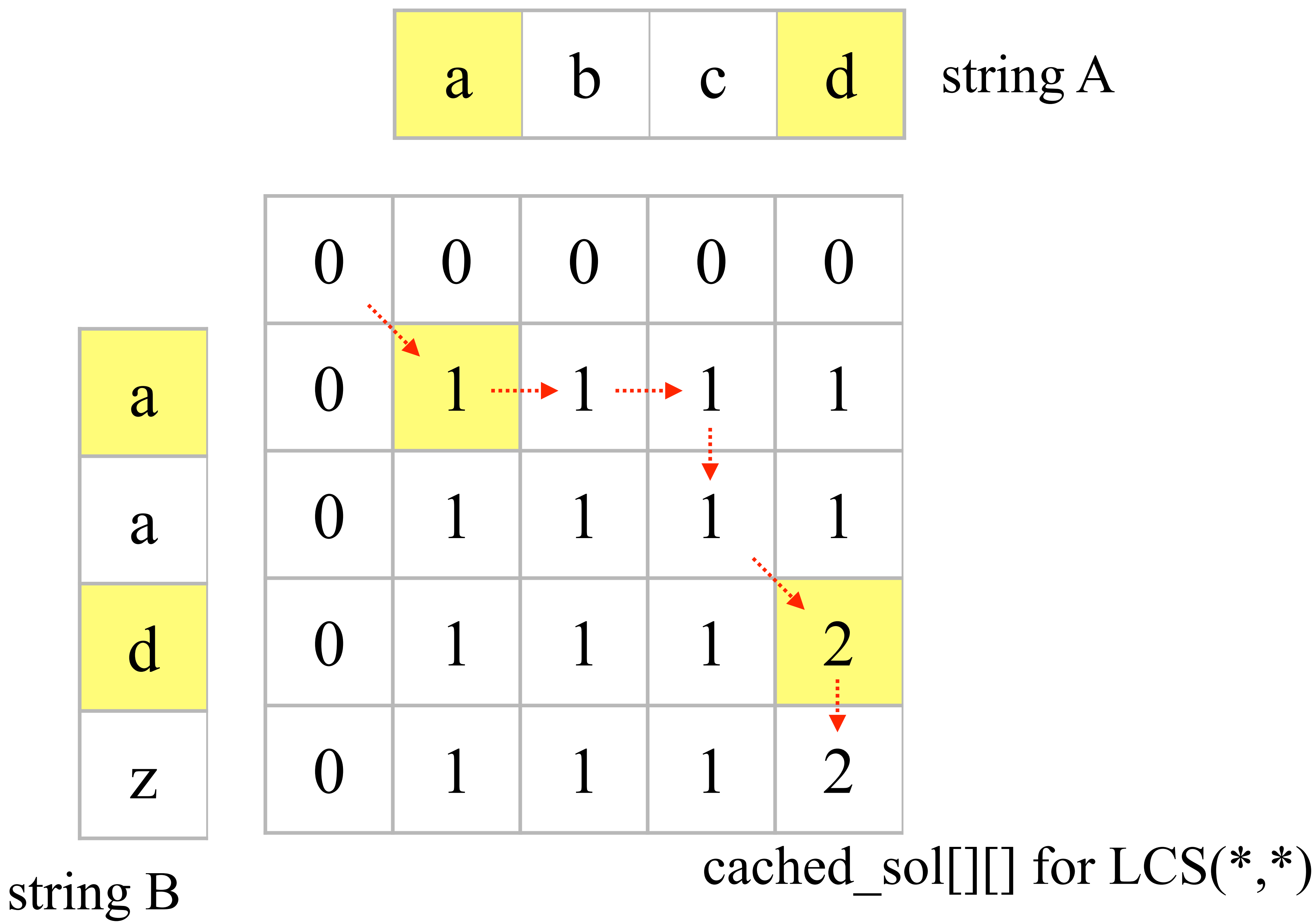
	0
a	0
a	0
d	0
z	0

string B

0	0	0	0	0
0	1	1	1	1
0	1	1	1	1
0	1	1	1	2
0	1	1	1	2

cached_sol[][] for LCS(*,*)

Finding a LCS rather than only its length



Reducing the working space

a	b	c	d

string A

string B					
	0	0	0	0	0
a	0	1	1	1	1
a	0	1	1	1	1
d	0				
z	0				

cached_sol[][] for LCS(*,*)

string B

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

Reducing the working space

string B \ string A		a	b	c	d
	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				

string A

string B

cached_sol[][] for LCS(*,*)

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

Reducing the working space

a	b	c	d	string A

string B					
	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				

cached_sol[][] for LCS(*,*)

Only two rows is needed at every time step.

Exercise

Can we find the LCS (rather than only outputting the length of the LCS) of A and B if the working space that you can afford is only 2 rows?

(Hint. If you have the i-th row,
can you recover the (i-1)-th row?)

Time and Space Complexities for finding a LCS

Time: $O(nm)$, Space: $O(n+m)$.

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

Exercise

Finding the longest **palindrome** subsequence of an array of characters.

(Hint. Reduce to LCS.)

Example.

a	b	c	z	d	d	c	a

a	b	c	z	d	d	c	a