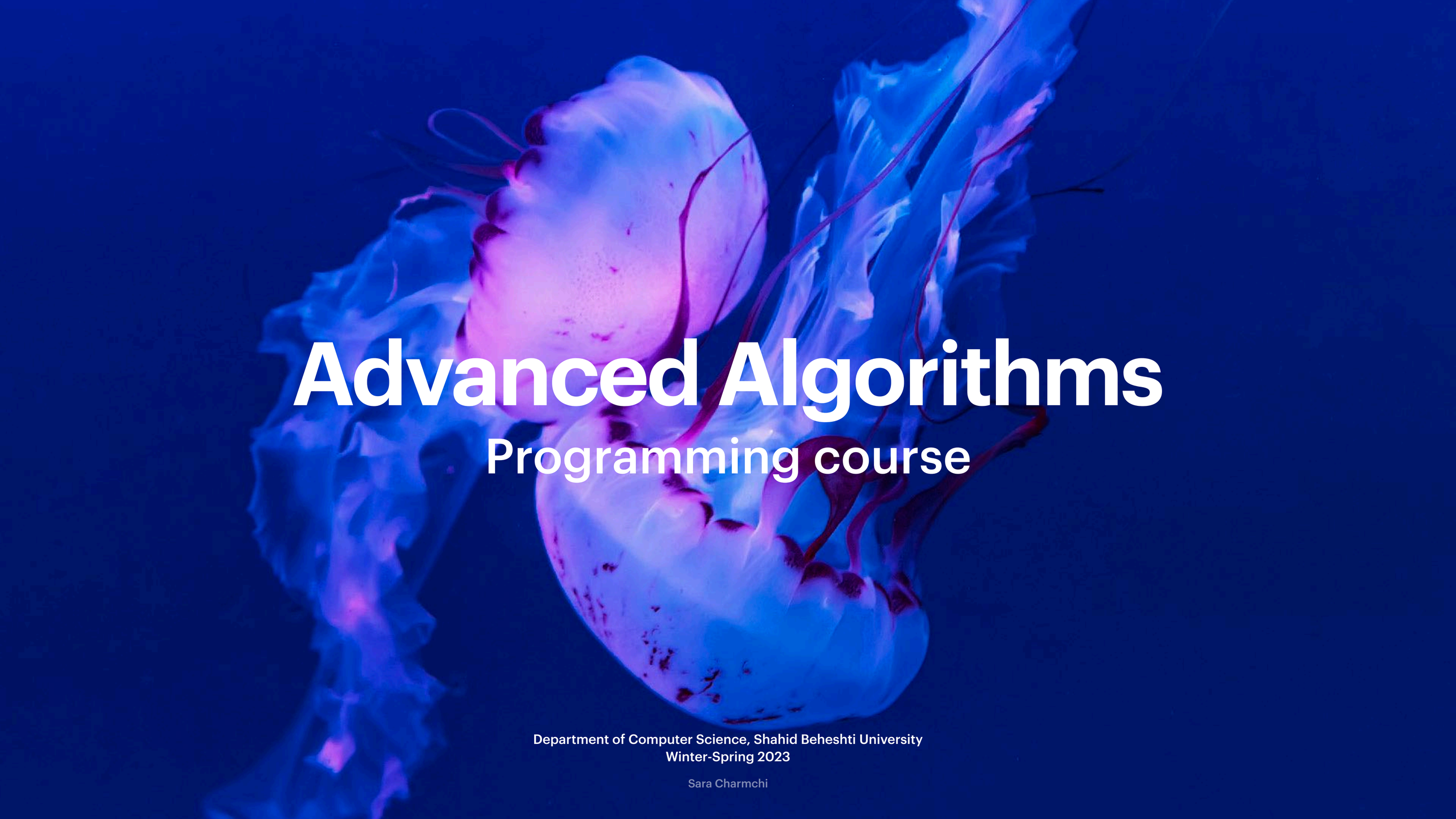


Advanced Algorithms

Programming course

Department of Computer Science, Shahid Beheshti University
Winter-Spring 2023

Sara Charmchi

Strategies for Algorithm Design

Dynamic Programming

Dynamic Programming
Definition

- General, powerful algorithm design technique
- Optimization problems : max/min result

DP≈ “Careful Exhaustive Search”

DP≈ Subproblems + re-use

re-use the solutions of subproblems

main problem —> split main problem subproblems —>slove subproblems

Dynamic Programming

Definition; Fibonacci numbers

0,1,1,2,3,5,8,13,...

$$\begin{cases} F_1 = F_2 = 1 \\ F_n = F_{n-1} + F_{n-2} \end{cases}$$

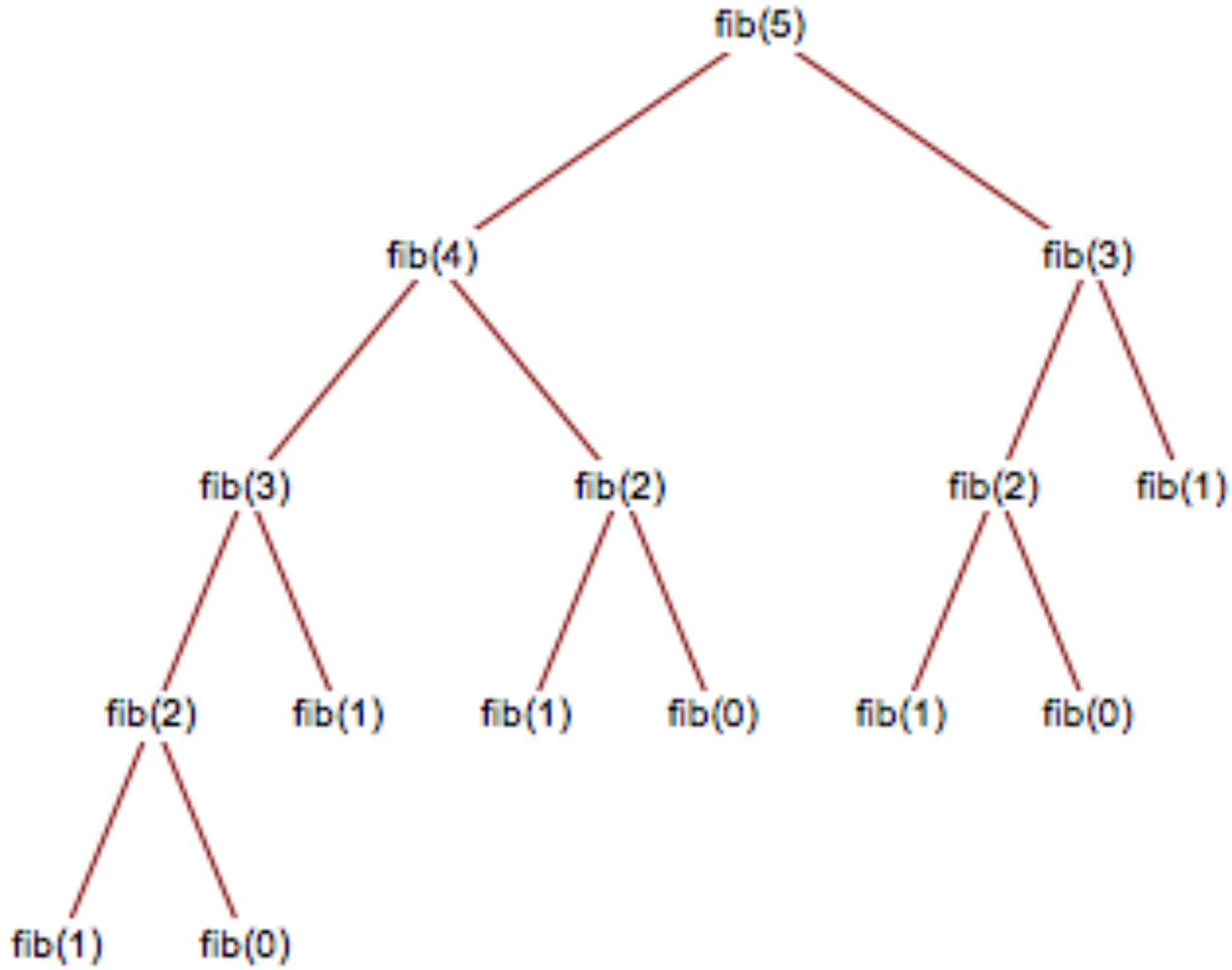
- Goal : compute F_n

Dynamic Programming

Fibonacci numbers; Naive Approach

```
fib(n):
    if n≤1 : f=n
    else : f = fib(n-1) + fib(n-2)
 return f
```

compute fib(n=5) :recursion tree



Dynamic Programming

Fibonacci numbers; Naive Approach

```
fib(n):
    if n≤1 : f=n
 else : f = fib(n−1) + fib(n−2)
 return f
```

Time complexity ? exponential time

consider running time as recurrence :

$T(n) = T(n-1) + T(n-2) + \Theta(1)$

$T(n) \geq 2T(n-2)$ ⟶ $T(n) = O(2^{n/2})$

Dynamic Programming

Fibonacci numbers; DP : Memoization (Top-down)

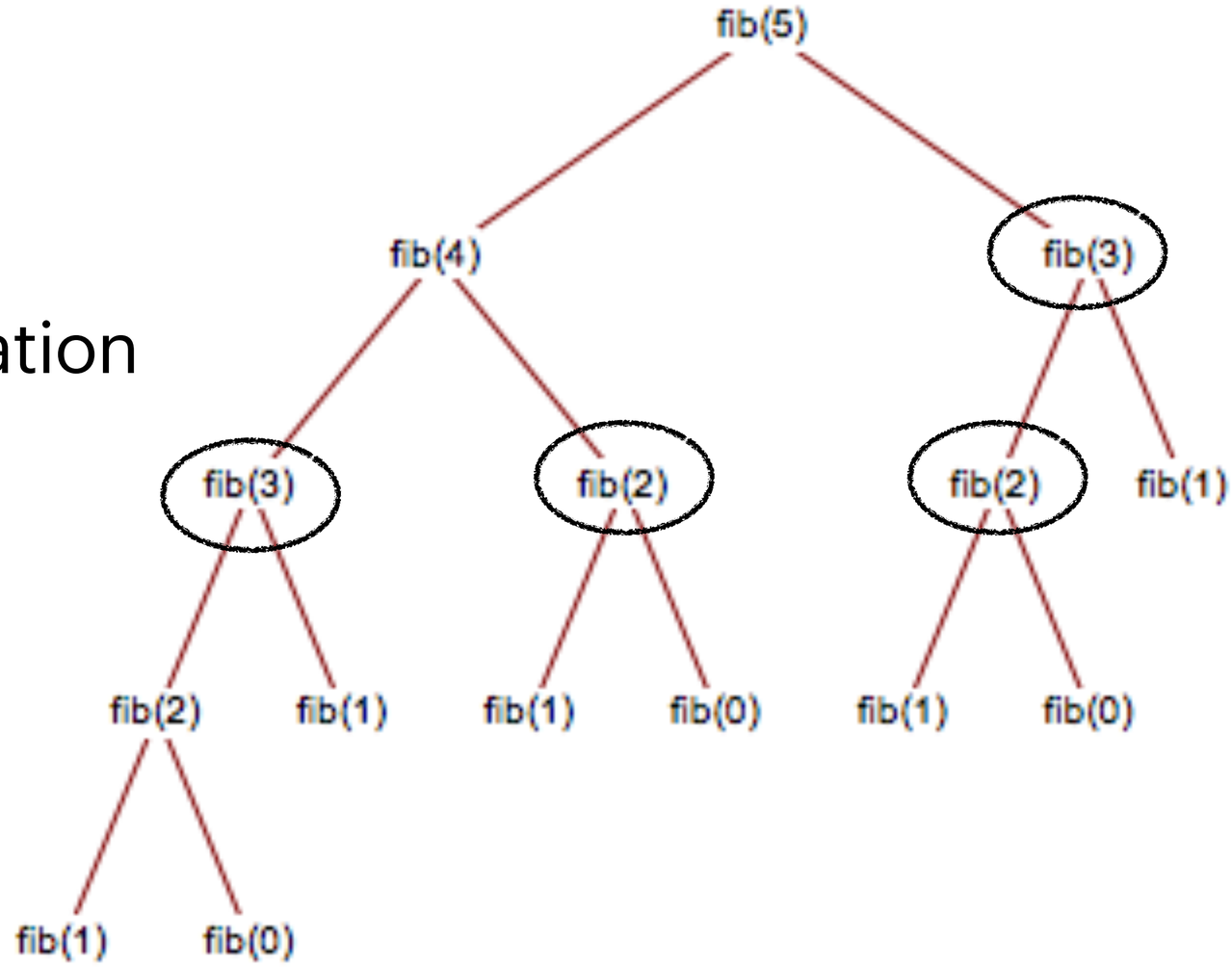
Any better idea?

store the computed result
somewhere, to avoid re-computation

Fib(n=5) ; take one global array

initially fill with -1 :

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5



Dynamic Programming

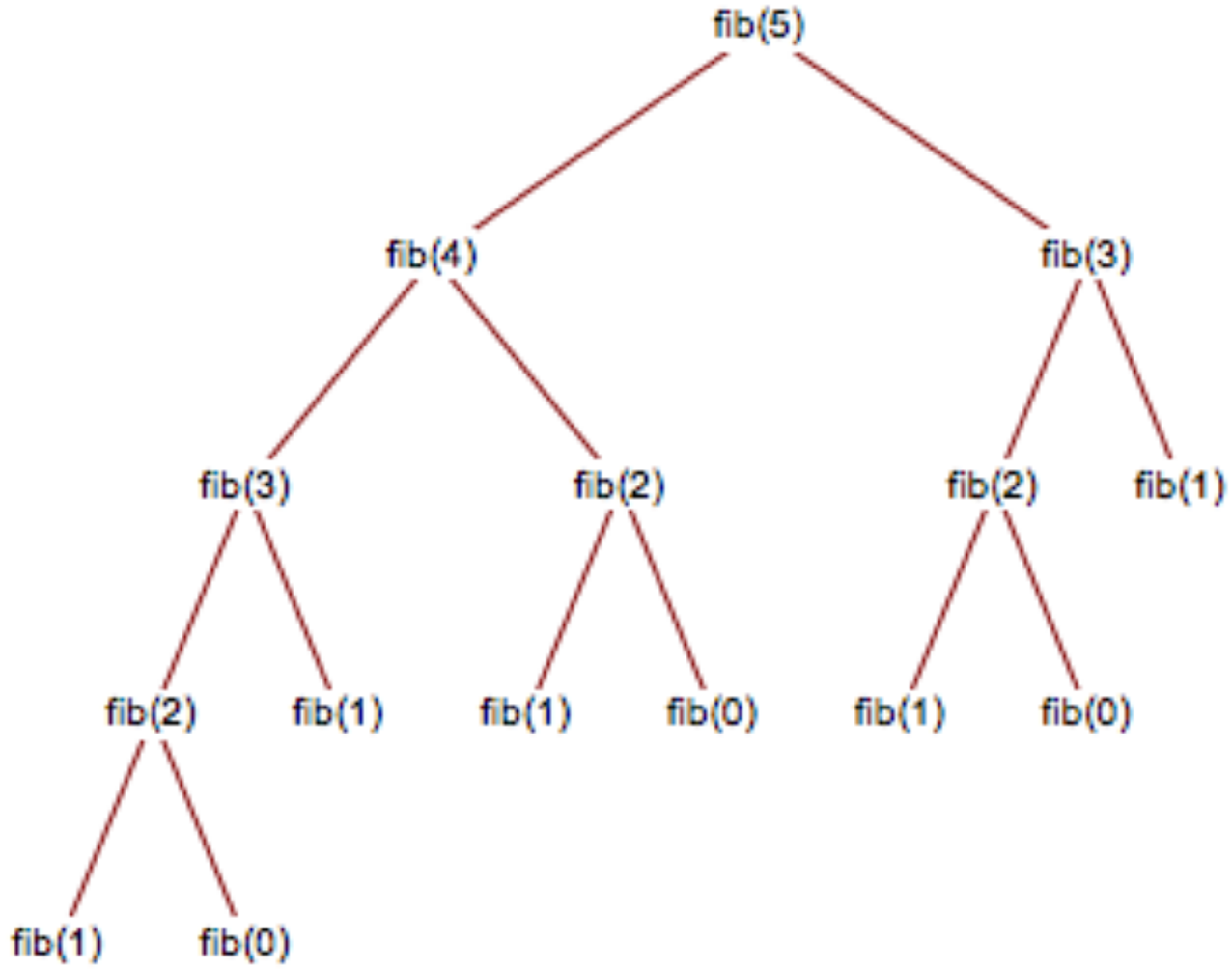
Fibonacci numbers; DP : Memoization

```
fib(n):
    if n≤1 : f=n
 else : f = fib(n-1) + fib(n-2)
 return f
```

Fib(n=5) ; take one global array

Call Fib(n=5) —> follow the tree

-1	-1	-1	-1	-1	-1
0	1	2	3	4	5



Dynamic Programming

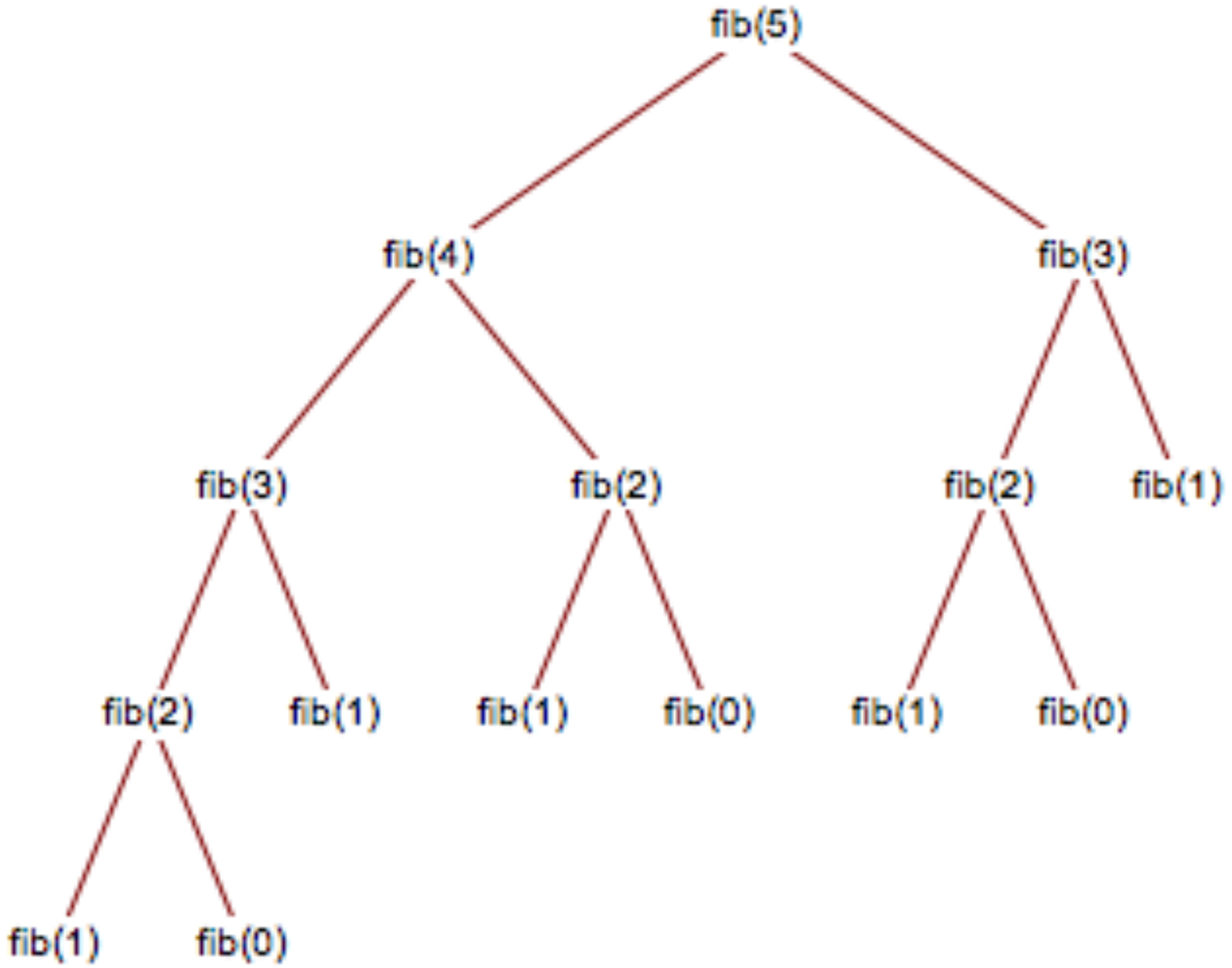
Fibonacci numbers; DP : Memoization

```
fib(n):
    if n≤2 : f=1
 else : f = fib(n-1) + fib(n-2)
 return f
```

Fib(n=5) ; take one global array

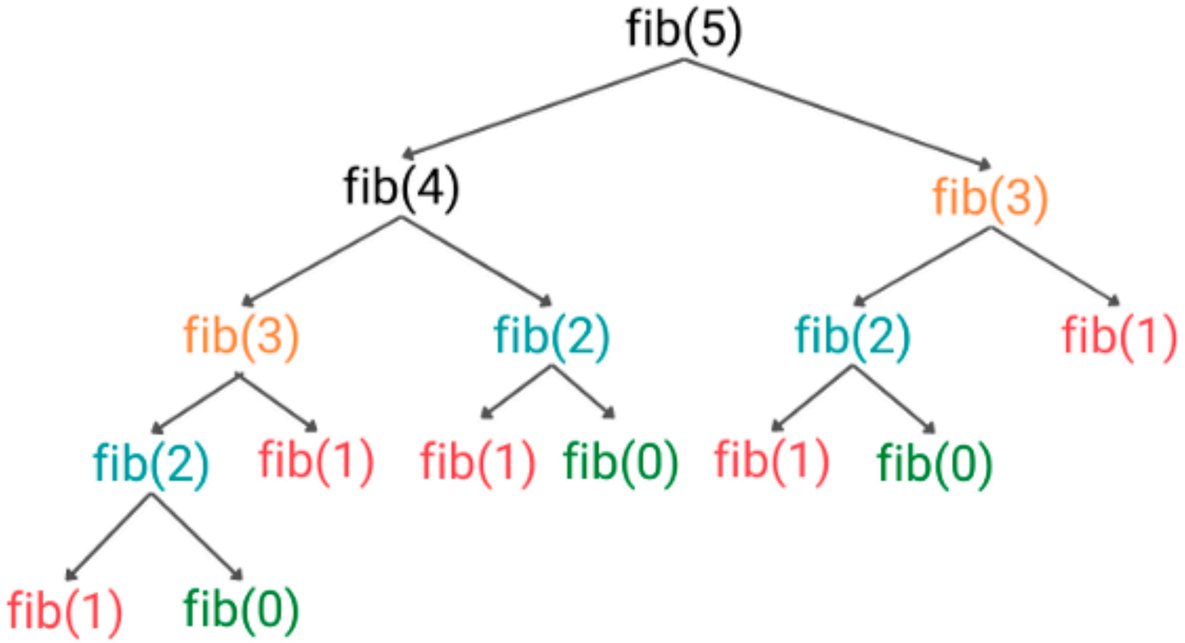
Call Fib(n=5) —> follow the tree

0	1	1	2	3	5
0	1	2	3	4	5

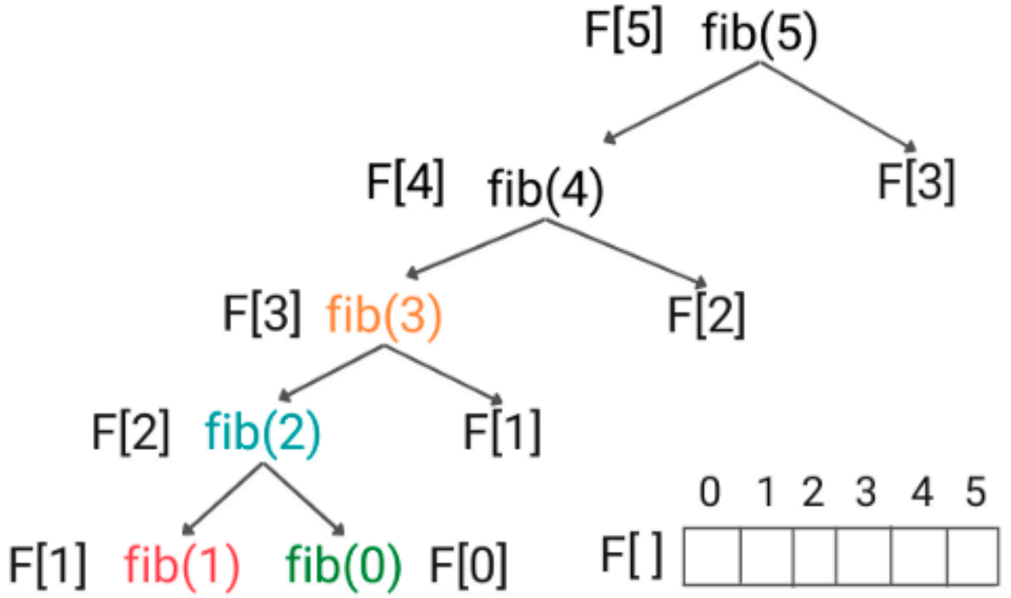


Dynamic Programming

Fibonacci numbers; DP : Memoization



Calculating 5th Fibonacci recursively.



Calculating 5th Fibonacci recursively but storing sub-problems solution in extra memory.

Dynamic Programming

Fibonacci numbers; DP : Memoization

```
memo = {}
fib(n):
    if n in memo : return memo [n]
 if n≤2 : f=1
 else : f = fib(n-1) + fib(n-2)
 memo[n] = f

 return f
```

f(k) only recurses the first time it's called, $\forall k$

two versions : memoized calls, non-memoized calls

Dynamic Programming

Fibonacci numbers; DP : Memoization

- Memoized calls cost constant time : $\Theta(1)$
- Number of non-memoized calls is n

  fib(1),fib(2),fib(3),...,fib(n)
- non recursive work per call is constant

Total time complexity : linear : $\Theta(n)$

```
memo = {}
fib(n):
    if n in memo : return memo [n]
  if n≤2: f=1
  else : f = fib(n-1) + fib(n-2)
  memo[n] = f

  return f
```

Naive Approach : Exponential ⟶ Dynamic Programming : Linear

Dynamic Programming

DP : Memoization

- Memoize (remember) & reuse solutions to subproblems that help solve the problem

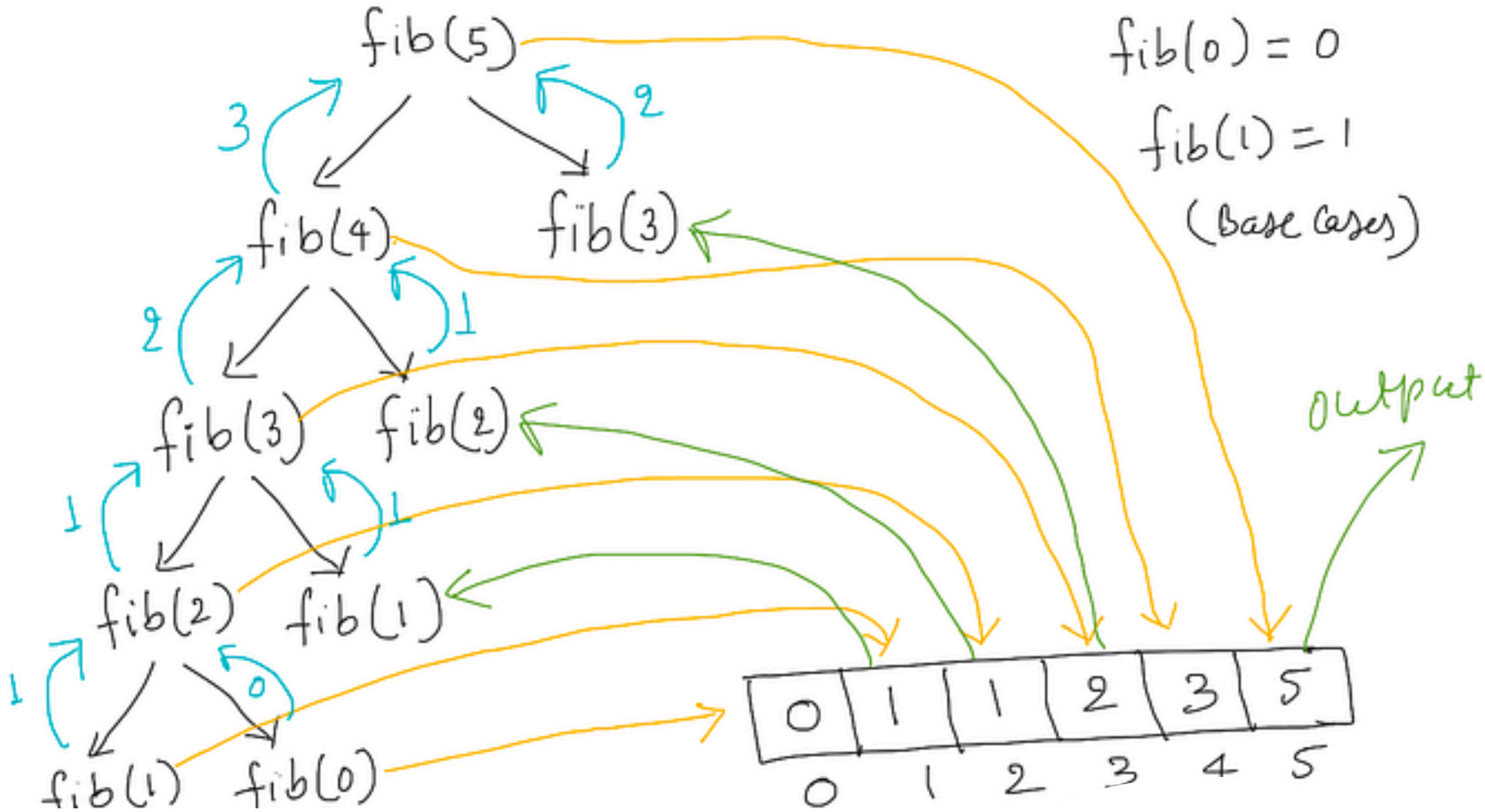
DP≈ recursion + memoization

- for any DP algorithm the time complexity would be :

time = number of subproblems * time/subproblem

Dynamic Programming

Fibonacci numbers; DP : Tabulation (Bottom-up)

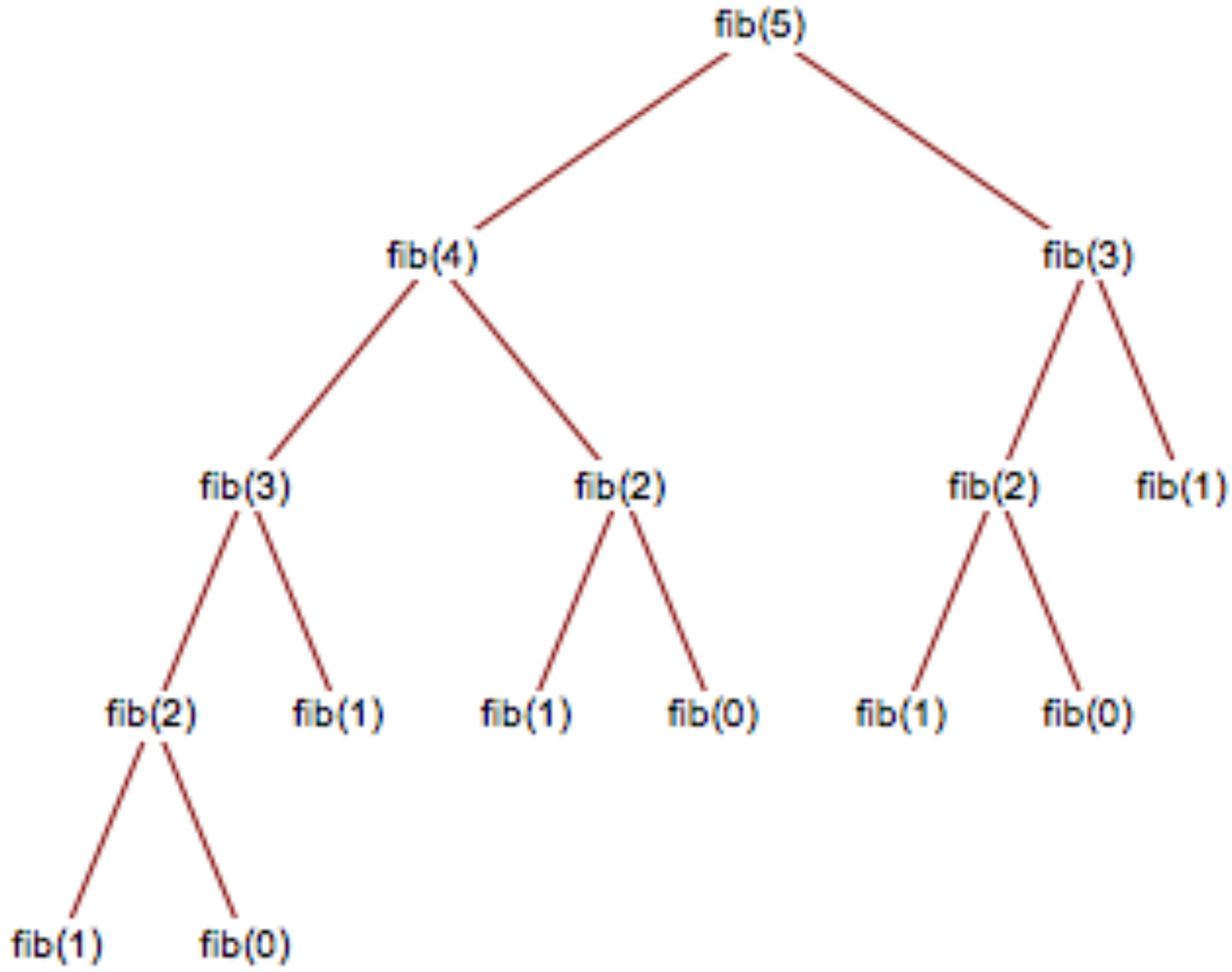


Dynamic Programming

Fibonacci numbers; DP : Tabulation (Bottom-up)

```
fib = {}
for k in range(1,n+1):
    if k≤2 f=1
    else f=fib[k-1]+fib[k-2]
    fib[k]=f
return fib[n]
```

0	1	1	2	3	5
0	1	2	3	4	5



Dynamic Programming

Fibonacci numbers; DP : Tabulation (Bottom-up)

```
fib = {}
for k in range(1,n+1):
    if k≤2 f=1
    else f=fib[k-1]+fib[k-2]
    fib[k]=f
return fib[n]
```

same method, different perspective

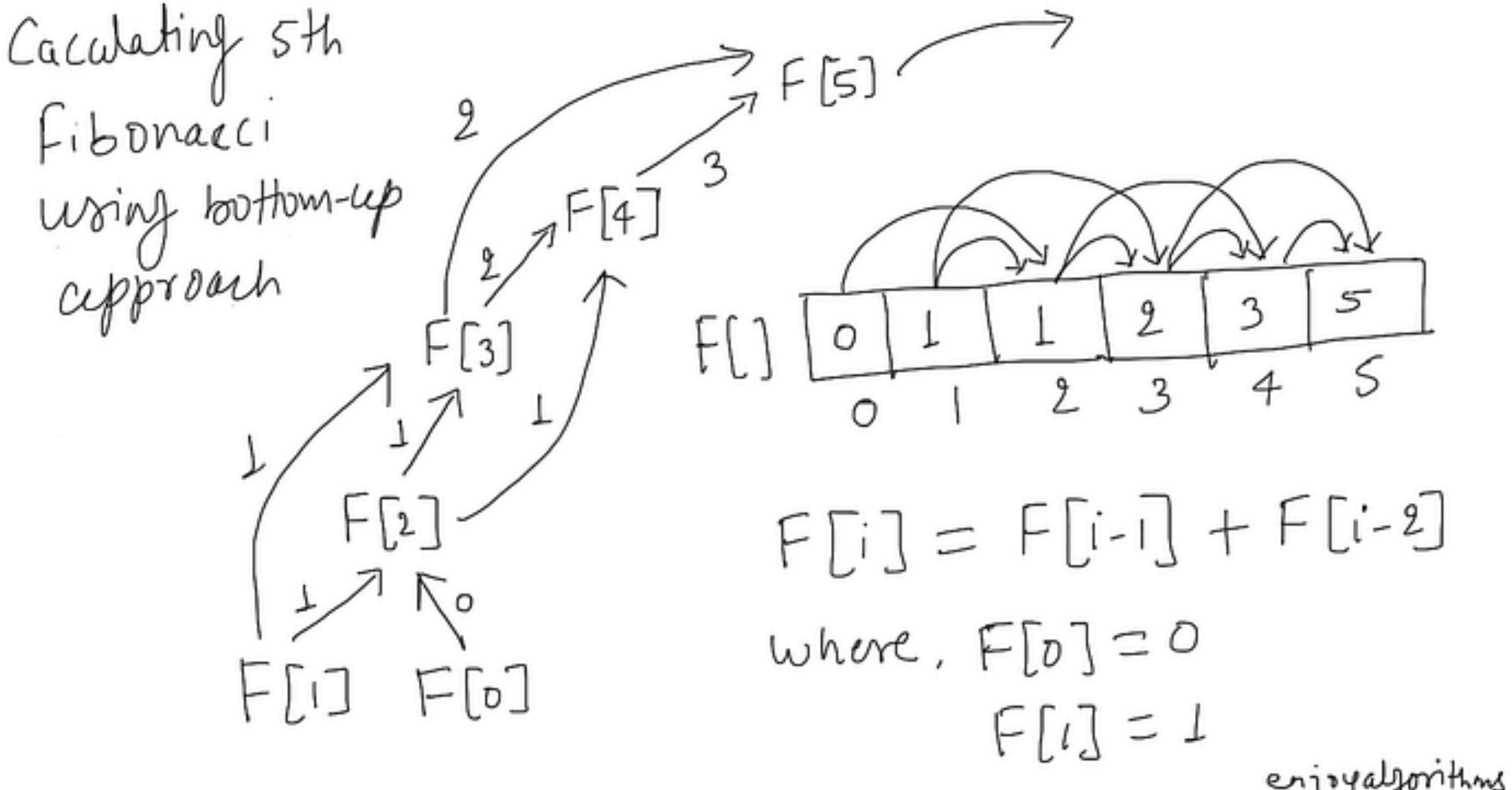
main difference : use loop instead of recursion

which one is better in practice?

Tabulation ⟶ not recursive calls , just look in a table. ⟶ save space

Dynamic Programming

DP : Tabulation



Dynamic Programming
DP : Tabulation

- Topological sort of subproblem dependency **directed acyclic graph(DAG)** :
  - vertices (the subproblems) and edges between the subproblems (dependencies).
  - The edges have a direction, since it matters which subproblem for every connected pair is the dependent one.
  - The graph has no cycles, meaning there must not be a way to start at one subproblem, then end up at that same subproblem just by following the arrows. Otherwise, we would end up in a situation where, to calculate one subproblem, we'd need to first calculate itself!

Rod-Cutting Problem
Definition

- you are in a company that buys long steel rods and cuts them into shorter rods, and then sells them. cutting is free
- they hired you to find the best way to cut up the rods

- the company charges p_i $ for a length of rod i. Rods are always integral in length

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

- Rod-cutting problem ; Given a rod of length n inches and a table of pieces p_i (i is in the range ($1,2,\ldots,n$), determine the maximum revenue r_n obtained by cutting the rod and selling the pieces. if no cutting gives the best price, we don't cut at all.

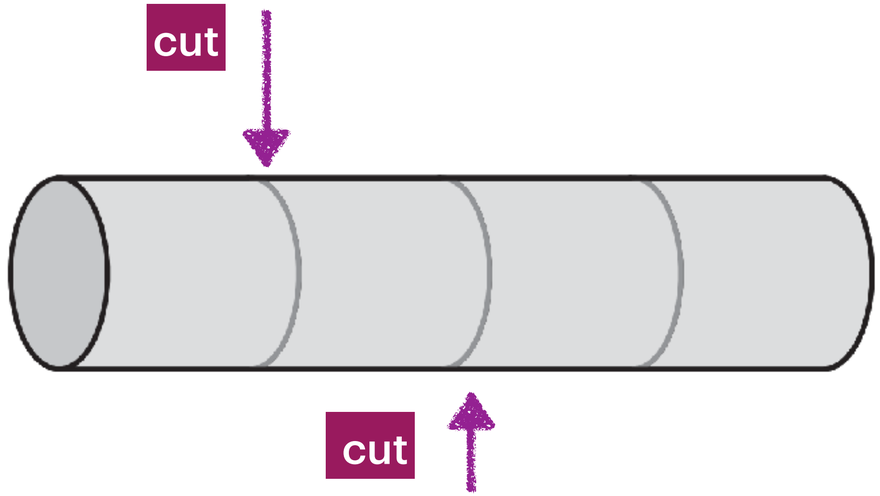
Rod-Cutting Problem
Definition

- you are given a rod of 4 inches and a price table

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

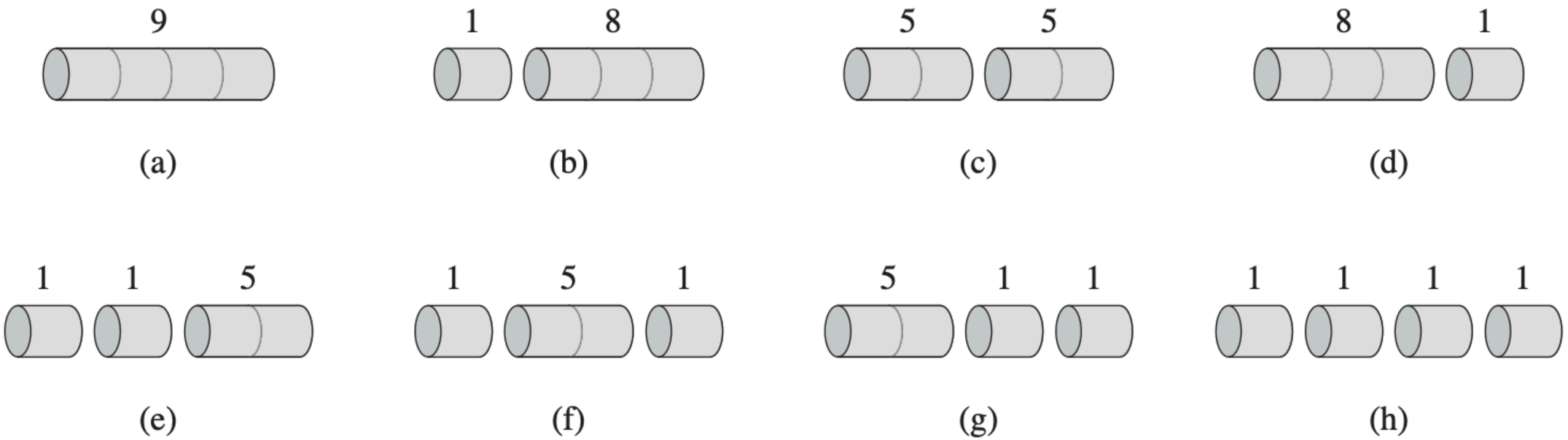
- should you cut the rod at all ?

- no cut : l=4 → 9
- 1 cut :1,3 : l= 1+3 → 1+8=9
- 1 cuts :2,2 l=2+2 → 5+5=10 ✓



Rod-Cutting Problem
Definition

- what are all possible cuts for length=4 ?



- we cut up a rod of length n in 2^{n-1} different ways

Rod-Cutting Problem

Definition

Length(n)	Price(p_n)	Cut/Don't Cut	Revenue(r_n)
1	1	N	1
2	5	N	5
3	8	N	8
4	9	2,2	10
5	10	2,3	13

$$
\begin{aligned}
r_1 &= 1 \quad \text{from solution } 1 = 1 \quad \text{(no cuts)}, \\
r_2 &= 5 \quad \text{from solution } 2 = 2 \quad \text{(no cuts)}, \\
r_3 &= 8 \quad \text{from solution } 3 = 3 \quad \text{(no cuts)}, \\
r_4 &= 10 \quad \text{from solution } 4 = 2 + 2, \\
r_5 &= 13 \quad \text{from solution } 5 = 2 + 3, \\
r_6 &= 17 \quad \text{from solution } 6 = 6 \quad \text{(no cuts)}, \\
r_7 &= 18 \quad \text{from solution } 7 = 1 + 6 \text{ or } 7 = 2 + 2 + 3, \\
r_8 &= 22 \quad \text{from solution } 8 = 2 + 6, \\
r_9 &= 25 \quad \text{from solution } 9 = 3 + 6, \\
r_{10} &= 30 \quad \text{from solution } 10 = 10 \quad \text{(no cuts)}.
\end{aligned}
$$

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

Rod-Cutting Problem
Formal Definition

- The optimal revenue from a rod of length n , r_n :

$$r_n = \max(p_n, r_1 + r_{n-1}, r_2 + r_{n-2}, \ldots, r_{n-1} + r_1) \ .$$

p_n ⟶ making no cut at all, $r_1 + r_{n-1}$ ⟶ a cut so that we have rods of length *1* and *n-1*

- In other words :

$$r_n = \max_{1<i<n} (p_i + r_{n-i}) \ .$$

p_i ⟶ piece of rod length *i*, r_{n-1} ⟶ revenue from rod of length *n-1*

- Is this a recursive approach?

Rod-Cutting Problem

Recursive top-down Implementation

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

Rod-Cutting Problem

Recursive top-down Implementation

```
price = [-100000,1, 5, 8, 9, 10, 17, 17, 20]
# Returns the best obtainable price for a rod of length n
# and price[] as prices of different pieces
def cutRod(price, n):
    if(n <= 0):
        return 0
    max_val = -100000

    # Recursively cut the rod in different pieces
    # and compare different configurations
    for i in range(1, n+1):
        max_val = max(max_val, price[i] +
                      cutRod(price, n - i))
    return max_val
```

Rod-Cutting Problem

Recursive top-down Implementation

- what is one limitation of the solution?

  gives the optimal revenue, but no answer to where to cut the rod!
- why is the recursive program so inefficient(slow) ?

Rod-Cutting Problem

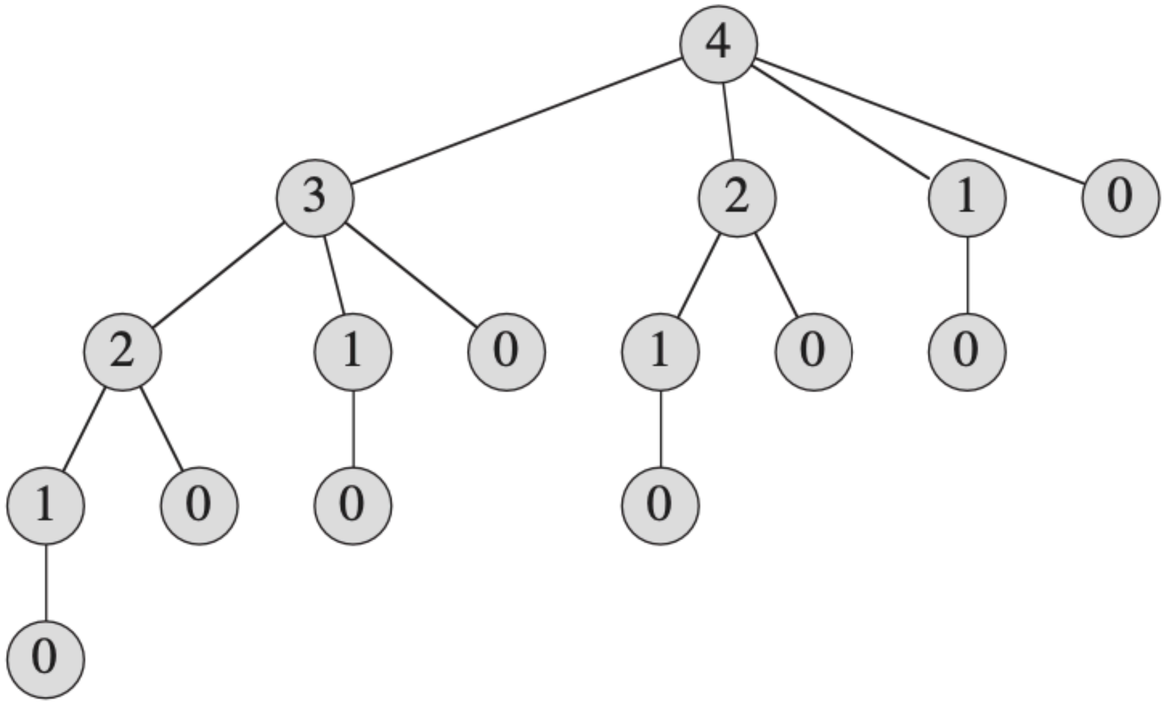
Analyze Recursive solution : recursion tree

- the rod-cut algorithm calls itself recursively over and over again with the same parameter values; it solves the same problems repeatedly

- the running time of the algorithm is exponential in n :

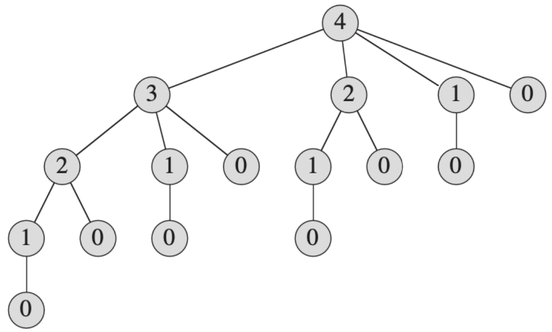
$$T(n) = O(2^n)$$

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) .$$



Rod-Cutting Problem
Dynamic Programming

- We arrange for each subproblem to be solved only **once**, saving it's solution
- During the computation if we need the solution to a subproblem again, we simply look it up rather than recomputing it.
- Two Dynamic Programming approaches:
    - Top-Down Dynamic Programming approach with memoization
    - Bottom-Up Dynamic Programming approach



Rod-Cutting Problem

Dynamic Programming : Memoization

```
MEMOIZED-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  for i = 0 to n
3      r[i] = -∞
4  return MEMOIZED-CUT-ROD-AUX(p, n, r)
```

initialize new auxilary array with -inf

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

if the solution is already in the table

r : presolved table of solutions;an array that maintains the best known solution for subproblem

memoized version of previous procedure

once found the solution, store it in the table

Rod-Cutting Problem

Dynamic Programming (Memoization) vs Recursive

```
CUT-ROD(p, n)
1  if n == 0
2      return 0
3  q = -∞
4  for i = 1 to n
5      q = max(q, p[i] + CUT-ROD(p, n - i))
6  return q
```

each time you increase n by 1, your program's running time would approximately double

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = -∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n - i, r))
8  r[n] = q
9  return q
```

check whether it is already in the table

update the table

Rod-Cutting Problem

Dynamic Programming : Tabulation

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

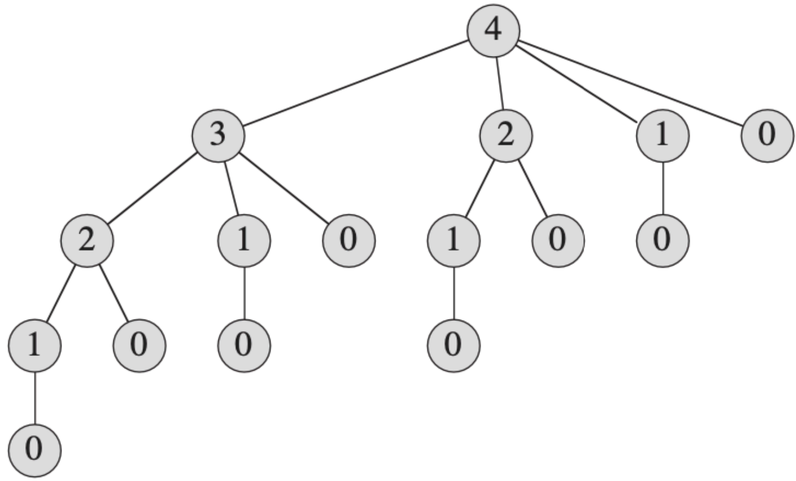
creates a new array

initialize r[0]

calculate the revenue for the rod of all the length :n

looks for all the possible ways to cut the rod

no recursive call



- a problem of size i is “smaller” than a subproblem of size j if i < j
- Thus, the procedure solves subproblems of sizes j=0,1,…,n, in that order

Rod-Cutting Problem

Dynamic Programming (Memoization) vs Recursive

- the bottom-up and top-down versions have same asymptotic running time : $\Theta(n^2)$, because of the double for loops
- the bottom-up approach usually outperforms the top-down approach by a small constant factor

```
MEMOIZED-CUT-ROD-AUX(p, n, r)
1  if r[n] ≥ 0
2      return r[n]
3  if n == 0
4      q = 0
5  else q = −∞
6      for i = 1 to n
7          q = max(q, p[i] + MEMOIZED-CUT-ROD-AUX(p, n − i, r))
8  r[n] = q
9  return q
```

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = −∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j − i])
7      r[j] = q
8  return r[n]
```

Rod-Cutting Problem

Returning actual solution

- For each rod of size j, we would like to compute not only the maximum revenue r_j, but also s_j, the optimal size of the first piece to cut off.

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30

i	0	1	2	3	4	5	6	7	8	9	10
$r[i]$	0	1	5	8	10	13	17	18	22	25	30
$s[i]$	0	1	2	3	2	2	6	1	2	3	10

Rod-Cutting Problem

Returning actual solution

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
 1  let r[0..n] and s[0..n] be new arrays
 2  r[0] = 0
 3  for j = 1 to n
 4      q = -∞
 5      for i = 1 to j
 6          if q < p[i] + r[j - i]
 7              q = p[i] + r[j - i]
 8              s[j] = i
 9      r[j] = q
10  return r and s
```

list of indexes to make the optimal cut

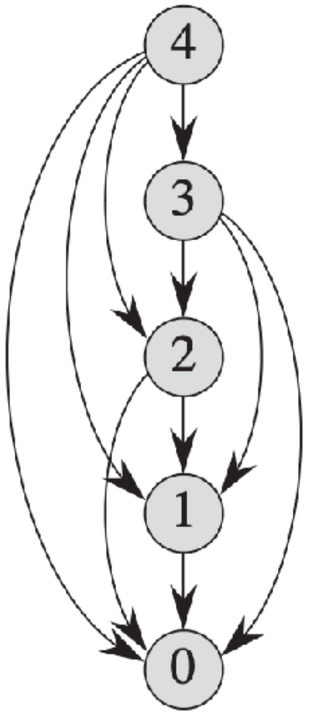
max —> if

```
PRINT-CUT-ROD-SOLUTION(p, n)
1  (r, s) = EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
2  while n > 0
3      print s[n]
4      n = n - s[n]
```

Rod-Cutting Problem

Subproblem Graphs

- When applying dynamic programming to a problem we should understand the set of subproblems involved.
- How do the subproblems depend on each other?
- A subproblem graph for the problem embodies this information
- It is like a ‘reduced’ or ‘collapsed’ version of the recursion tree



- subproblem graph for n=4 for the rod -cutting problem
- the vertex labels give the size of the corresponding subproblems
- a directed edge x->y indicates that we need a solution to subproblem y when solving subproblem x

Elements of Dynamic Programming

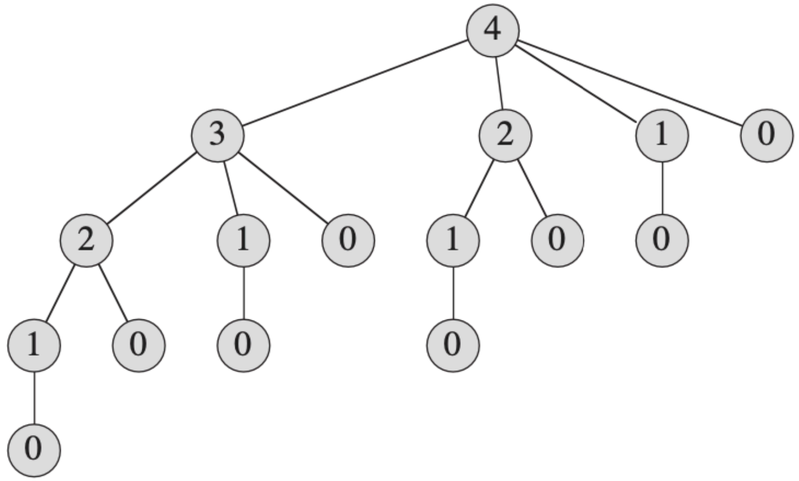
Two key ingredients that an optimization problem must have in order for dynamic programming to apply :

- **Optimal Substructure**

  a problem exhibits Optimal Substructure if an optimal solution to the problem containns within it optimal solutions to subproblems

- **Overlapping subproblems**

  the space of subproblems must be ‘small’ in the sense that the recursive algorithm for the problem solves the same subproblems over and over (not generating new subproblems)



Elements of Dynamic Programming

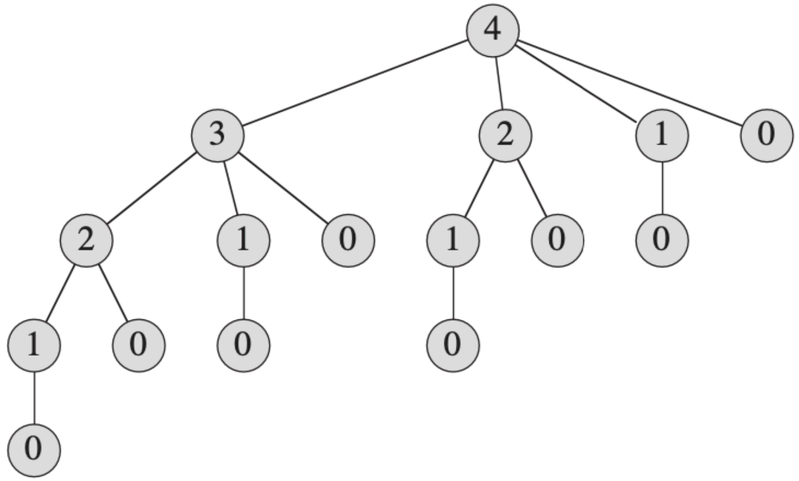
Two key ingredients that an optimization problem must have in order for dynamic programming to apply :

- **Optimal Substructure**

  a problem exhibits Optimal Substructure if an optimal solution to the problem containns within it optimal solutions to subproblems

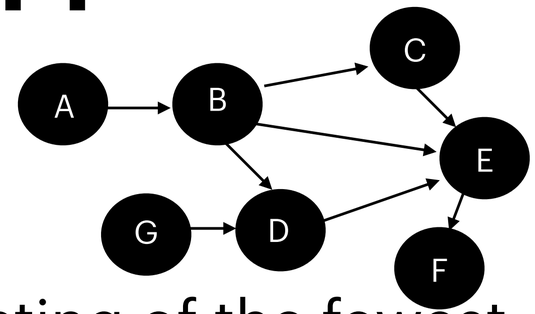
- **Overlapping subproblems**

  the space of subproblems must be ‘small’ in the sense that the recursive algorithm for the problem solves the same subproblems over and over (not generating new subproblems)



# of unique subproblems << # of total subproblems

When is Dynamic Programming not applicable?



Given a directed graph G= (V,E) and vertices u,v ∈ V

✓ **(a)Unweighted shortest path** : Find a path from u to v consisting of the fewest edges. Such a path must be simple (no cycles).

Any path p from u to v must contain w (say). Then we can decompose p into p1 and p2 so that p1 is the path from u to w and p2 from w to v

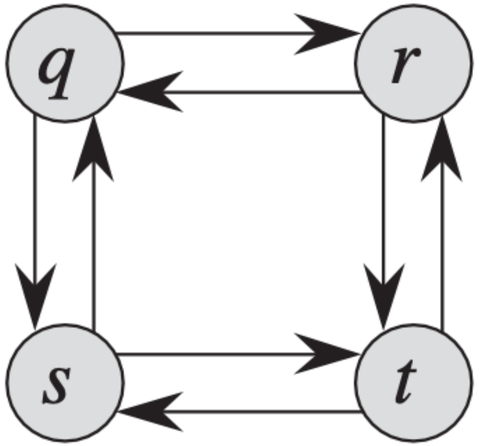
Can we argue that p1 is the shortest path from u to w, and p2 from w to v?

✗ **(a)Unweighted longest path** : Find a simple path from u to v consisting of the most edges

Unweighted longest path

Say we have a path p as the longest path from u to v. we have intermediate vertex w that lies in the path such that path p is composed of p1 and p2.

Does this imply that p1 is the longest path from u to w and p2 is the longest path from w to v?

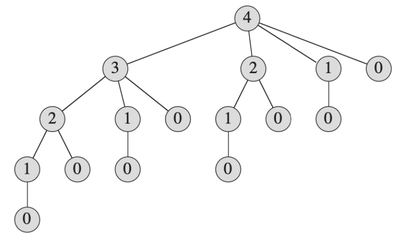


consider path q—>r—>t → the longest path from q to t

Is q—>r the longest path from q to r?

Is r—>t the longest path from r to t?

Steps for developing a dynamic programming algorithm



1. Characterize the structure of n optimal solution
2. Recursively define the value of an optimal solution

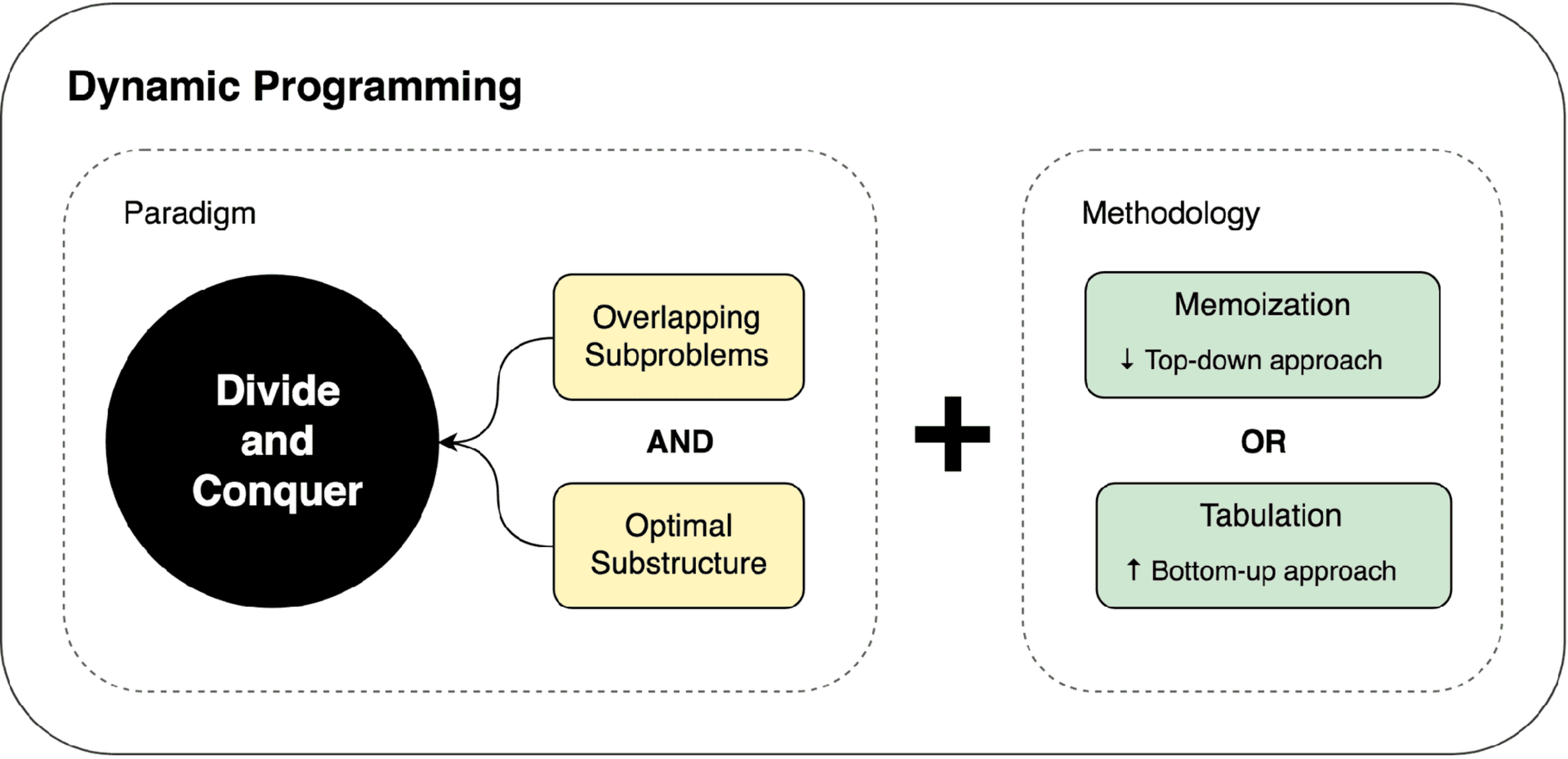
$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}) \ .$$

3. Compute the value of an optimal solution, typically in a bottom-up fashion
4. Construct an optimal solution. (where to make a cut in the rod-cutting problem

```
BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] be a new array
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          q = max(q, p[i] + r[j - i])
7      r[j] = q
8  return r[n]
```

```
EXTENDED-BOTTOM-UP-CUT-ROD(p, n)
1  let r[0..n] and s[0..n] be new arrays
2  r[0] = 0
3  for j = 1 to n
4      q = -∞
5      for i = 1 to j
6          if q < p[i] + r[j - i]
7              q = p[i] + r[j - i]
8              s[j] = i
9      r[j] = q
10 return r and s
```

What is the difference between Divide and Conquer Algorithms and Dynamic Programming Algorithms?


Dynamic Programming
Paradigm
Divide and Conquer
Overlapping Subproblems
AND
Optimal Substructure
Methodology
Memoization
↓ Top-down approach
OR
Tabulation
↑ Bottom-up approach

Longest Common Subsequence

Finding DNA similarity

- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine.
- we can express a strand of DNA as a string over the finite set :{A; C; G; T}
- compare two strands of DNA to determine how “similar” the two strands are : some measure of how closely related the two organisms are.

S_1 =ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

how to measure similarity? find a third strand S_3 in which the bases in S_3 appear in each of S_1 and S_2; these bases must appear in the same order, but not necessarily consecutively.

S_3 =GTCGTCGGAAGCCGGCCGAA.

Longest Common Subsequence

What is a subsequence

Given a sequence $X = <x_1, x_2, ..., x_m>$

A sequence $Z = <z_1, z_2, ..., z_k>$ is a subsequence of X if there exists a strictly increasing sequence $i_1, i_2, ..., i_k$ of indices of X such that for all j=1, 2, .., k we have $x_{ij}=z_j$

Example: $X= <A, B, C, B, D, A, B>$ then $Z =<B, C, D, B>$ is a subsequence of X with corresponding index sequence $<2, 3, 5, 7>$

Longest Common Subsequence

What is a longest common subsequence

Z is a **common subsequence** of X and Y if it is sequence of both X and Y.

Example: X= $<$A, B, C, B, D, A, B$>$ then Y =$<$B, D, C, A, B, A$>$ then the sequence $<$B, C, A$>$ is a common subsequence of X and Y.

$<$B, C, A$>$ is not the longest common subsequence.

$<$B, C, A, B$>$, $<$B, D, A, B$>$ and $<$B, C, B, A$>$ are the three longest common subsequences.

The LCS problem: Given two sequences X = $<x_1, x_2, \ldots, x_m>$ and Y = $<y_1, y_2, \ldots, y_n>$, find the maximum length common subsequence of X and Y

Longest Common Subsequence

The brute-force approach

1. Enumerate all subsequences of X and Y

   takes 2^m time, because X has that many subsequences

2. Then for each subsequence in X check if it matches each subsequence in Y

3. Find the longest matching subsequence

Exponential time

Longest Common Subsequence

Optimal Substructure of LCS problem

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

   X= <A, B, C, B, D, A, A> then Y =<B, D, C, A, B, A> , Z =<B, C, B, A>

2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.

   X= <A, B, C, B, D, A> then Y =<B, D, C, A, B> , Z =<B, C, B>

3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

   X= <A, B, C, B, D, A, B> then Y =<B, D, C, A, B, A> , Z =<B, D, A, B>

Longest Common Subsequence

Optimal Substructure of LCS problem

Theorem 15.1 (Optimal substructure of an LCS)
Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of X and Y.

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Sketch of proofs:

(1) can be proven by contradiction: if the last characters of X and Y are not included in Z, then a longer LCS can be constructed by adding this character to Z, a contradiction.

(2) and (3) have symmetric proofs: Suppose there exists a subsequence W of X_{m-1} and Y (or of X and Y_{n-1}) with length > k. Then W is a common subsequence of X and Y, contradicting Z being an LCS.

Therefore, an LCS of two sequences contains as prefix an LCS of prefixes of the sequences. We can now use this fact construct a recursive formula for the value of an LCS.

Longest Common Subsequence

A recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1],c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

S_1 =ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

Longest Common Subsequence

A recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

x_i

S_1 =ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

y_j

Longest Common Subsequence

A recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

x_i

S_1 =ACCGGTCGAGTGCGCGGAAGCCGGCCG

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

y_j

Longest Common Subsequence

A recursive solution

$$c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0\,, \\ c[i-1,j-1]+1 & \text{if } i,j > 0 \text{ and } x_i = y_j\,, \\ \max(c[i,j-1], c[i-1,j]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j\,. \end{cases}$$

x_i

S_1 =ACCGGTCGAGTGCGCGGAAGCCGGCCG

S_2=GTCGTTCGGAATGCCGTTGCTCTGTAAA

y_j

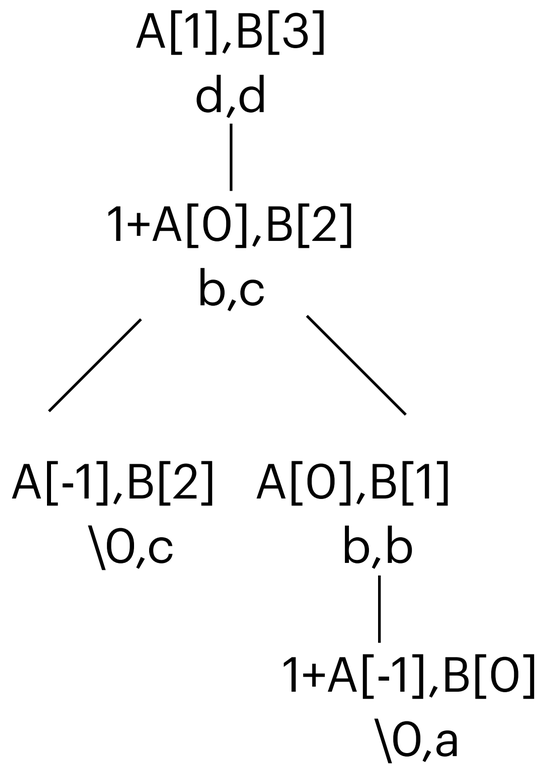
A

b	d
0	1

B

a	b	c	d
0	1	2	3

```
int lcs(A,B,i,j):
{
    if (i==-1 & j==-1):
      return 0;
  else if (A[i]==B[j]):
      return 1+lcs(A,B,i-1,j-1);
  else
      return max(lcs(A,B,i-1,j),lcs(A,B,i,j-1));
}
```



A

b	d
0	1

B

a	b	c	d
0	1	2	3

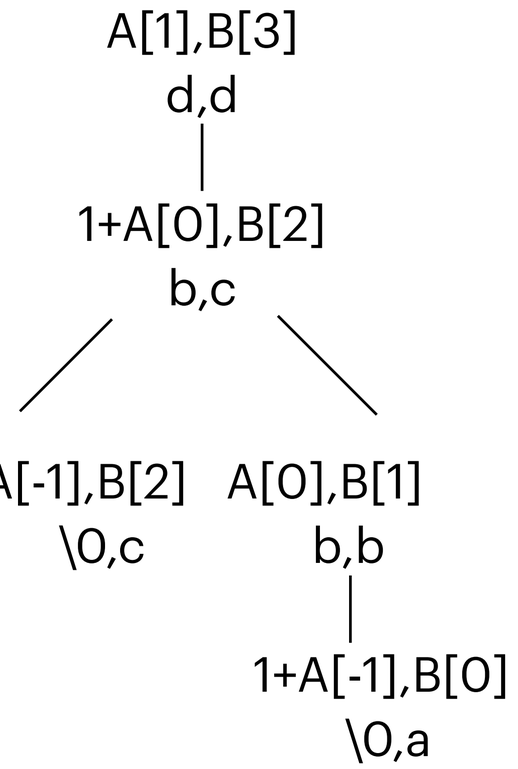
```
int lcs(A,B,i,j):
{
    if (i==-1 & j==-1):
      return 0;
  else if (A[i]==B[j]):
      return 1+lcs(A,B,i-1,j-1);
  else
      return max(lcs(A,B,i-1,j),lcs(A,B,i,j-1));
}
```

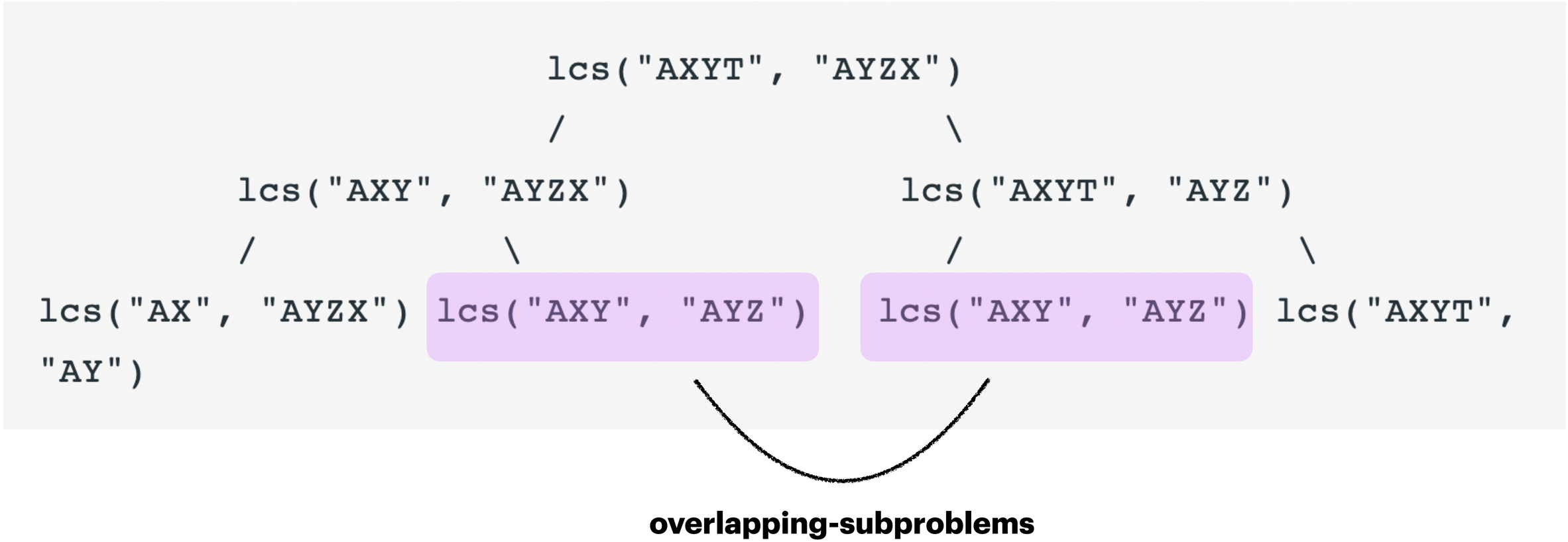
2

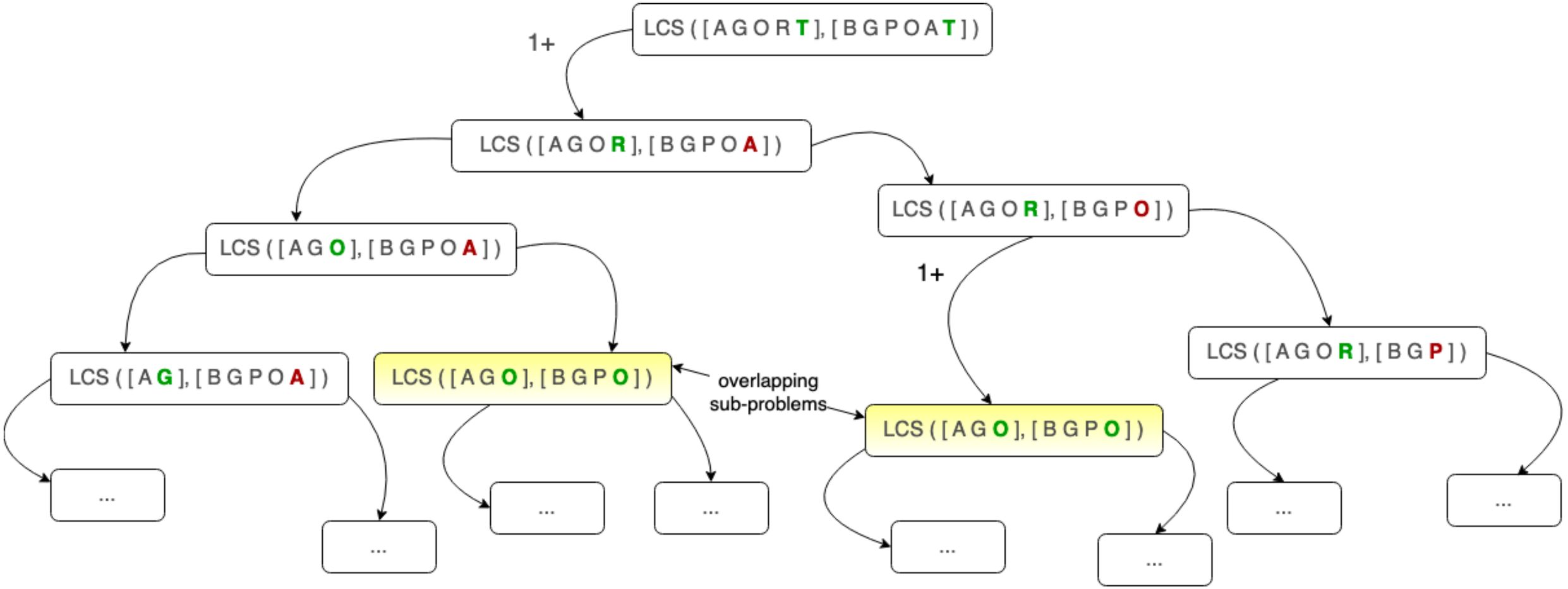
1+1=2

MAX(0,1)=1

1+0=1




lcs("AXYT", "AYZX")
lcs("AXY", "AYZX")
lcs("AXYT", "AYZ")
lcs("AX", "AYZX")
lcs("AXY", "AYZ")
lcs("AXY", "AYZ")
lcs("AXYT", "AY")
overlapping-subproblems



Recursion tree for finding the longest common sub-sequence

Longest Common Subsequence

Dynamic Programming

- Procedure LCS-LENGTH takes two sequences :

$$X = \langle x_1, x_2, \ldots, x_m \rangle \qquad Y = \langle y_1, y_2, \ldots, y_n \rangle$$

- Returns : b and c tables; c[m][n] contains the length of LCS of length X and Y
  - ❖store c[i,j] values in a table c[0...m][0...n]

    compute entries in row-major order (i.e. fill in the first row of c from left to right, then second row, and so on.)
  - ❖Also maintain table b[0...m][0...n] to help constuct an optimal solution

    b[i][j] points to the table entry corresponding to optimal subproblem chosen when computing c[i][j]

Time Complexity : T(n) = O (m * n)

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Longest Common Subsequence

Dynamic Programming

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

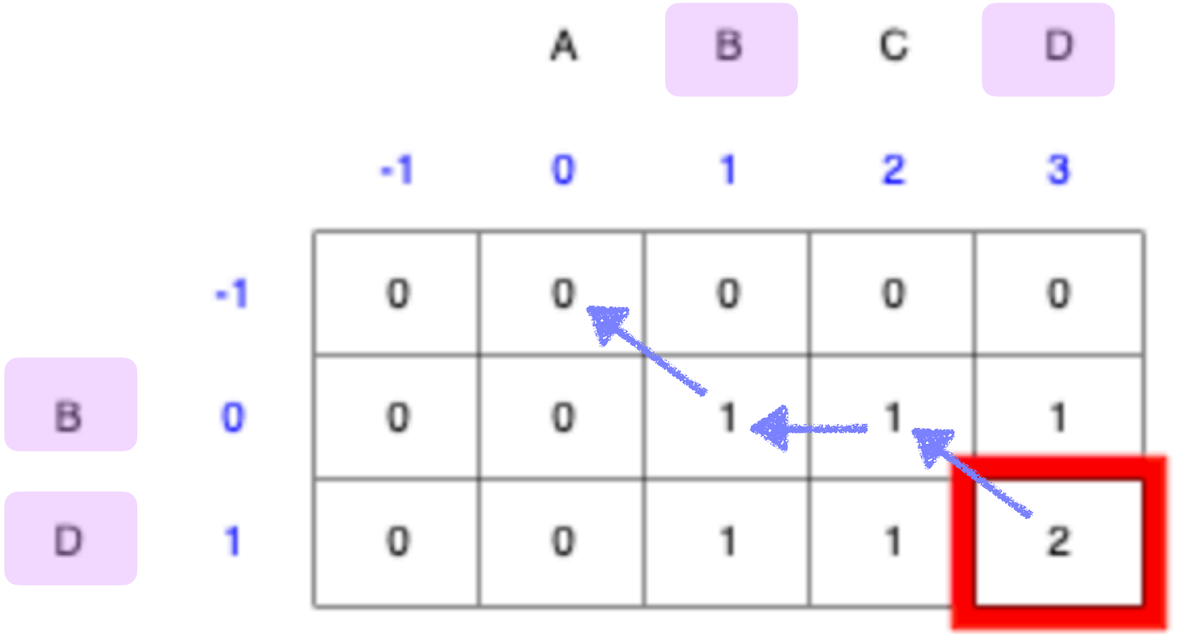
Longest Common Subsequence

Dynamic Programming; Constructing LCS

- Begin at b[m][n] and trace through the table by following the arrows.
- whenever we encounter a "↖" in the entry b[i][j] it implies that xi=yi is an element of the LCS
- we encounter the elements of LCS in reverse order

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

		j	0	1	2	3	4	5	6
i			y_j	B	D	C	A	B	A
0	x_i		0	0	0	0	0	0	0
1	A		0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B		0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C		0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B		0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D		0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A		0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B		0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4


A
B
C
D
-1
0
1
2
3
B
D
-1
0
1
0 0 0 0 0
0 0 1 1 1
0 0 1 1 2