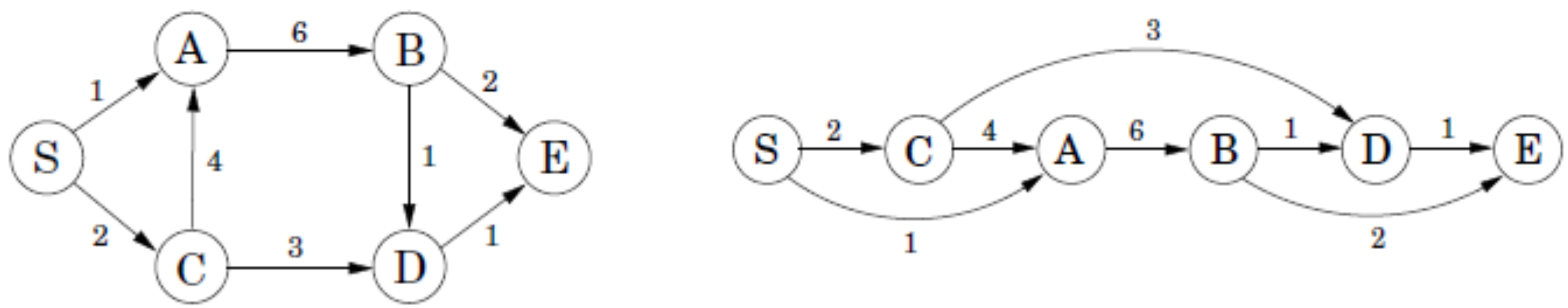
Dynamic programming

Gou Guanglei(苟光磊)
ggl@cqut.edu.cn

Shortest paths in dags

- A dag and its linearization

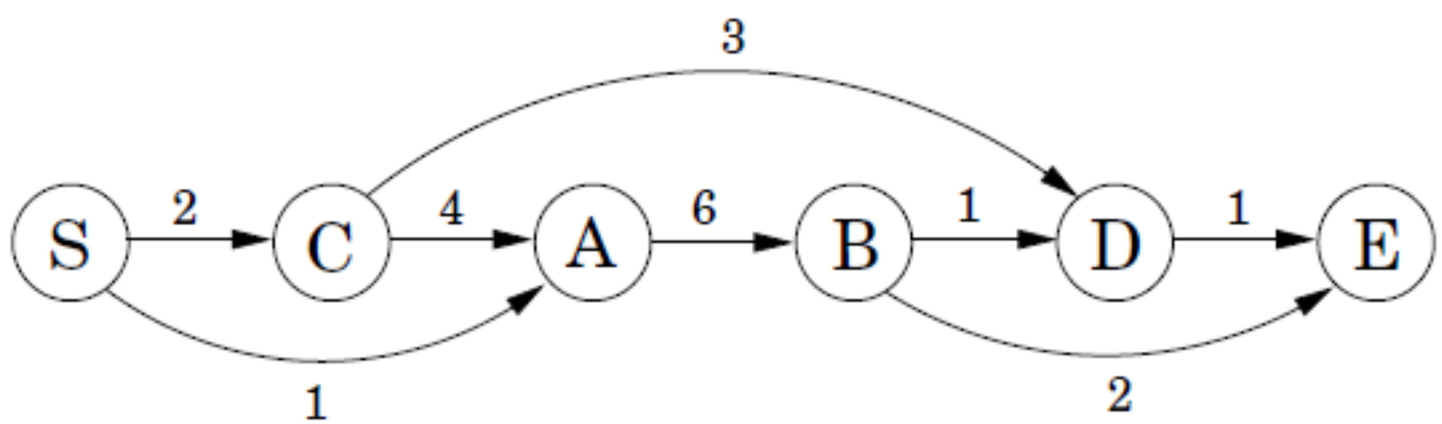


- To compute the distance from S to D, only need to consider distance to C and to B (because B and C are two predecessors to D).
- Dist (D) = min { dist(B) + 1 , dist(C) + 3 }
- If we compute dist values in the left-to-right order, we can make sure that when we get to a node v, we **already have all the information** we need to compute dist(v).

Shortest paths in dags

```
initialize all dist(·) values to ∞
dist(s) = 0
for each v ∈ V\{s}, in linearized order:
    dist(v) = min_{(u,v)∈E}{dist(u) + l(u,v)}
```

- The algorithm solves a collection of subproblems, $\{\text{dist}(u) : u \in V\}$.
- Starting with dist(s), then solve "larger" subproblems.



Dynamic programming

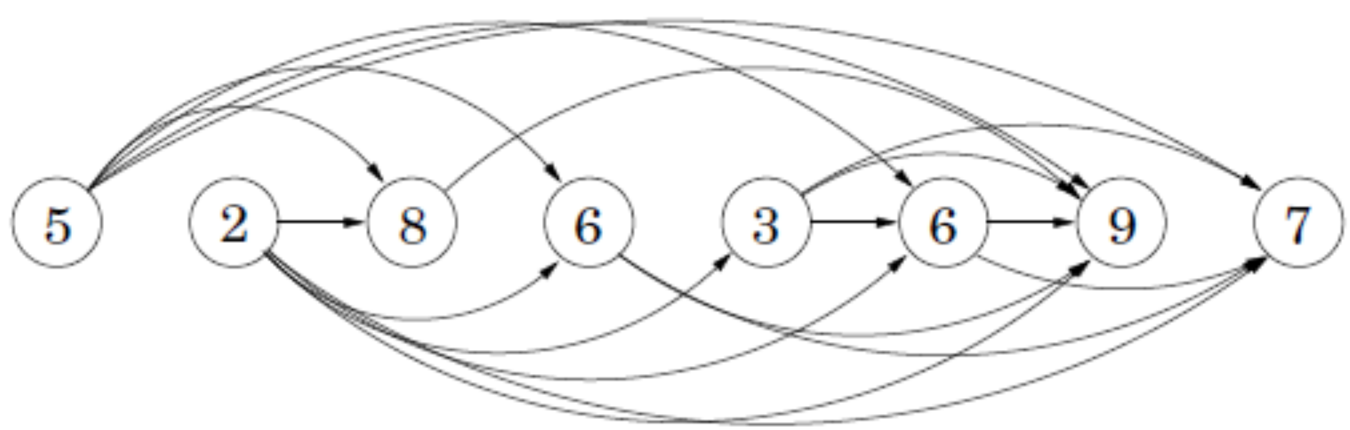
- a very powerful algorithmic paradigm
- a problem is solved by identifying a collection of ***subproblems*** and tackling them one by one
  - smallest first
  - using the answers to small problems to solve larger ones,
  - until the original problem is solved.

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

Longest increasing subsequences

- Input : a sequence of numbers *a1, ..., an*
- A *subsequence* is any subset of these numbers taken in order, of the form *ai1, ai2 ,..., aik where* $1 \le i1 < i2 < \ldots < ik \le n$
- Goal : to find the increasing subsequence of greatest length.
- E.g.) the longest increasing subsequence of 5, 2, 8, 6, 3, 6, 9, 7 : 2, 3, 6, 9



- Find *the longest path* in the dag!

Longest increasing subsequences

- $L(j)$: the length of the longest path – the longest increasing subsequence – ending at j
- Algorithm

$$
\begin{array}{l}
\texttt{for } j = 1, 2, \ldots, n: \\
\quad L(j) = 1 + \max\{L(i) : (i,j) \in E\} \\
\texttt{return } \max_j L(j)
\end{array}
$$

- *Dynamic programming* : In order to solve our original problem, we have defined a collection of *subproblems* { $L(j)$: $1 \leq j \leq n$ } with the following key property :
  - (*) There is an *ordering* on the subproblems, and a *relation* that shows how to solve a subproblem given the answers to "smaller" subproblems (subproblems that appear earlier in the ordering).

Longest increasing subsequences

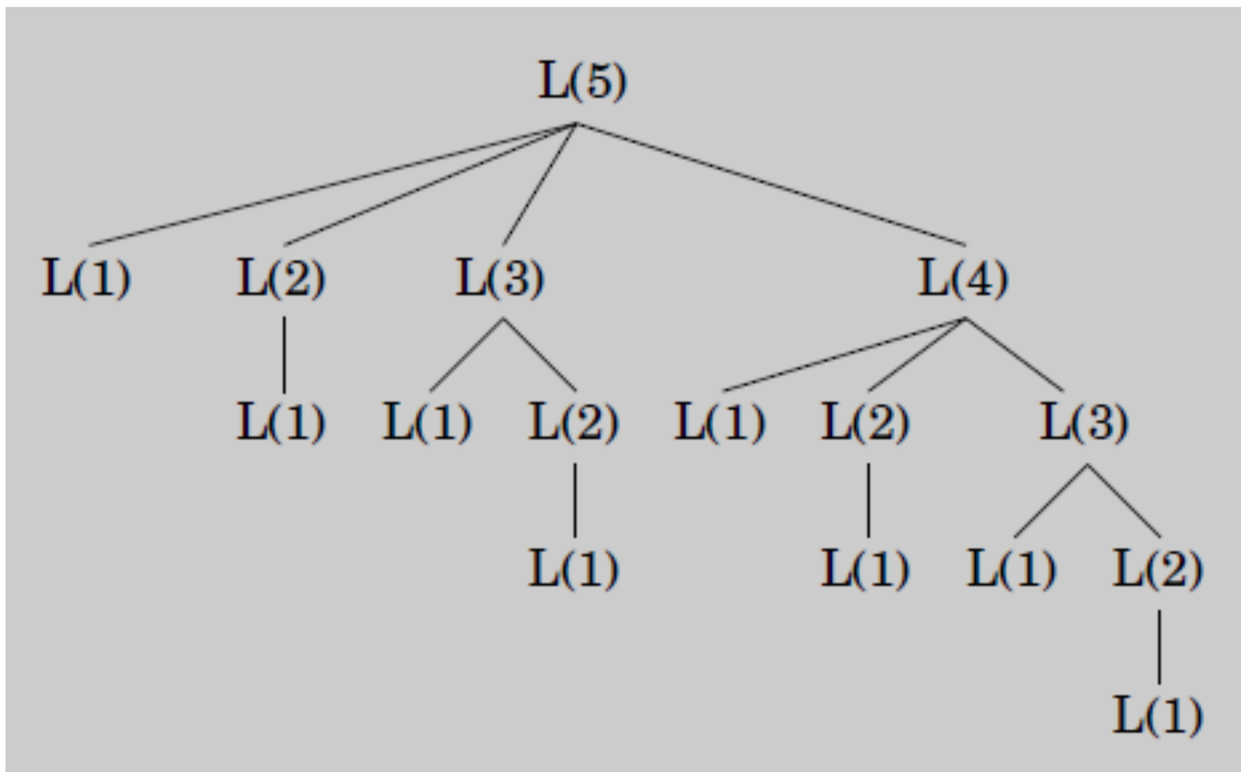
- Each subproblem is solved using the relation :
  - $L(j) = 1 + \max\{L(i) : (i, j) \in E\}$

- How long does this step take?
  - To compute $L(j)$: O(in-degree(j)).
  - Total : $O(|E|) \rightarrow O(n^2)$.

- L values only tells us the *length* of the optimal subsequence.How to construct the subsequence?
  - While computing $L(j)$, record prev(j), the previous node on the longest path to j.

Recursive vs. dynamic programming

- The formula for $L(j)$ suggests an alternative, *recursive* algorithm.
- Suppose that the numbers are sorted. Then, $L(j) = 1 + \max\{ L(1), L(2), \ldots, L(j-1)\}$.
- The following figure unravels the recursion for $L(5)$:



- The tree for $L(n)$ has *exponential* size. Many *repeated* nodes!
- Only *small* number of *distinct* subproblems -> DP solve them in the right order.

Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

Longest Common Subsequence

- How similar are these two species?

DNA:
AGCCCTAAGGGCTACCTAGCTT

DNA:
GACAGCCTACAAGCGTTAGCTTG

- Pretty similar, their DNA has a long common subsequence:

AGCCTAAGCTTAGCTT

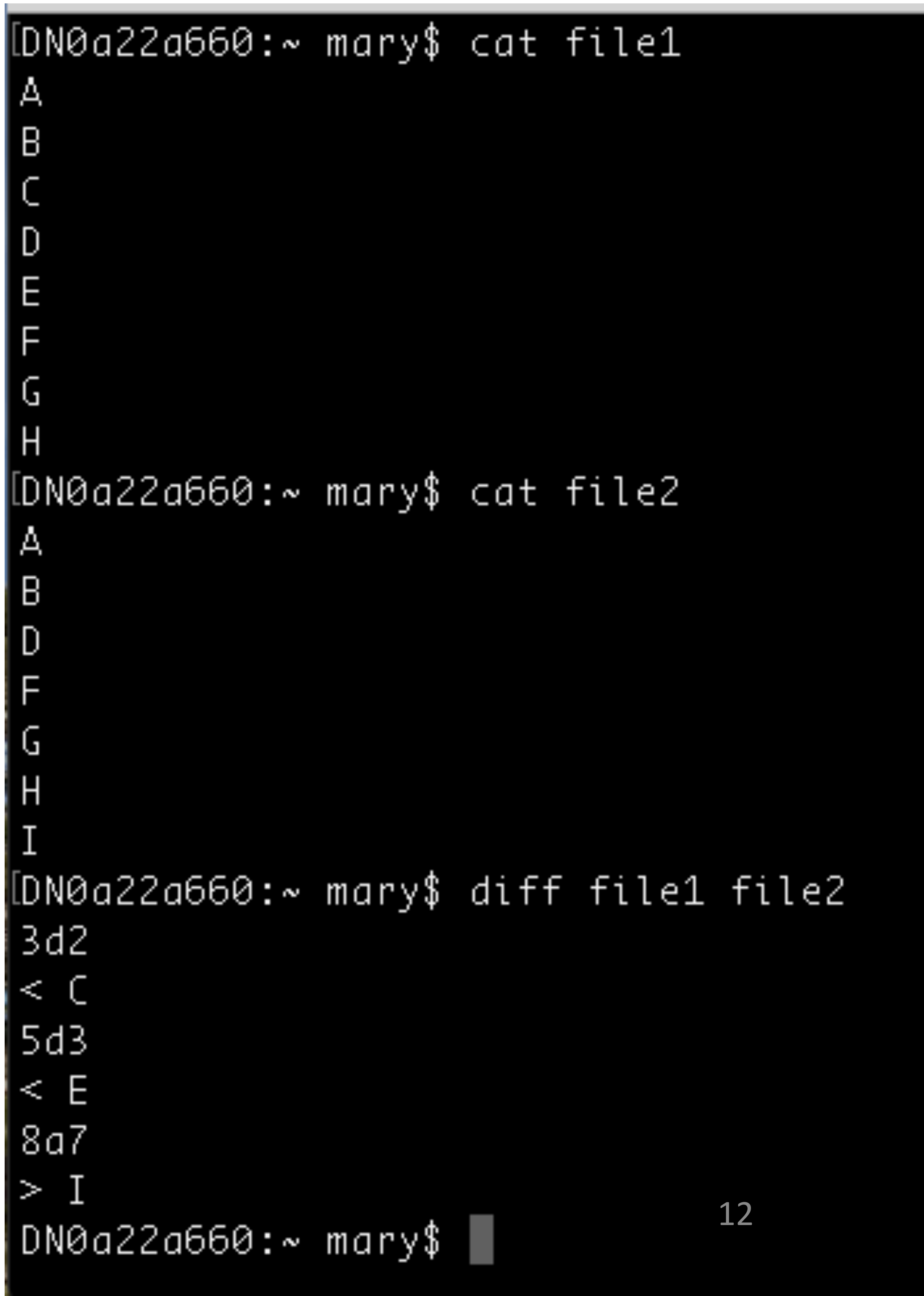
Longest Common Subsequence

- Subsequence:
  - BDFH is a **subsequence** of ABCDEFGH
- If X and Y are sequences, a **common subsequence** is a sequence which is a subsequence of both.
  - BDFH is a **common subsequence** of ABCDEFGH and of ABDFGHI
- A **longest common subsequence**…
  - …is a common subsequence that is longest.
  - The **longest common subsequence** of ABCDEFGH and ABDFGHI is ABDFGH.

We sometimes want to find these

- Applications in bioinformatics

- The unix command `diff`
- Merging in version control
  - svn, git, etc…

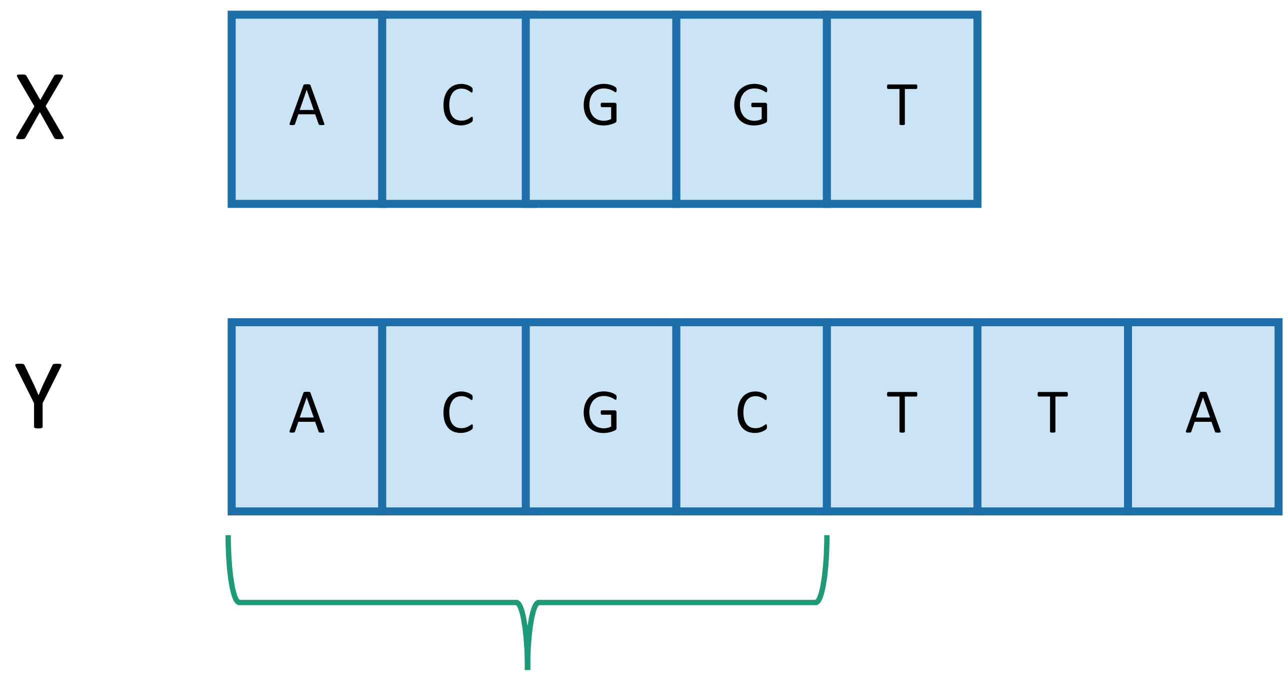


Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Step 1: Optimal substructure

Prefixes:



Notation: denote this prefix **ACGC** by Y_4

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let $C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Examples: $C[2,3] = 2$
$C[4,4] = 3$

Optimal substructure ctd.

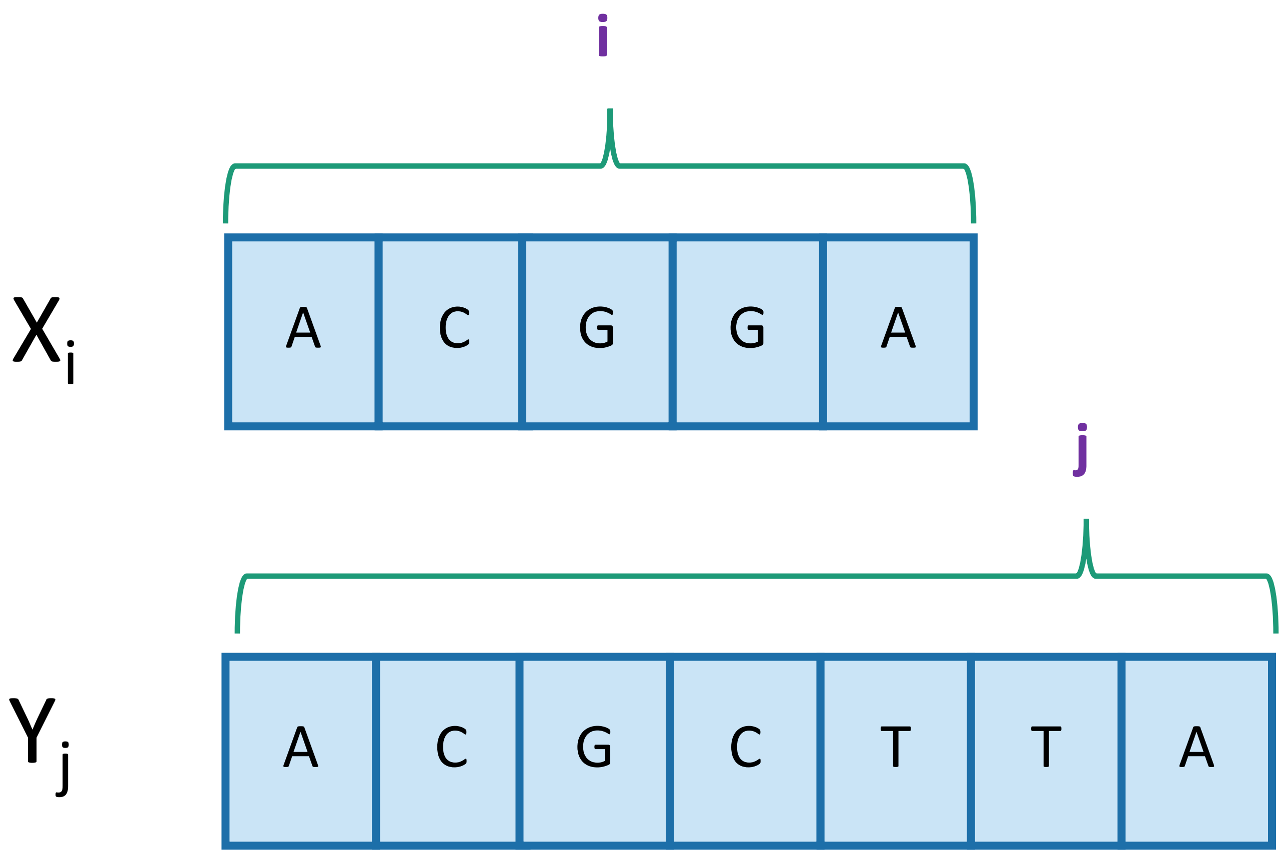
- Subproblem:
  - finding LCS's of prefixes of X and Y.
- Why is this a good choice?
  - As we will see, there's some relationship between LCS's of prefixes and LCS's of the whole things.
  - These subproblems overlap a lot.

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Goal

- Write C[i,j] in terms of the solutions to smaller sub-problems

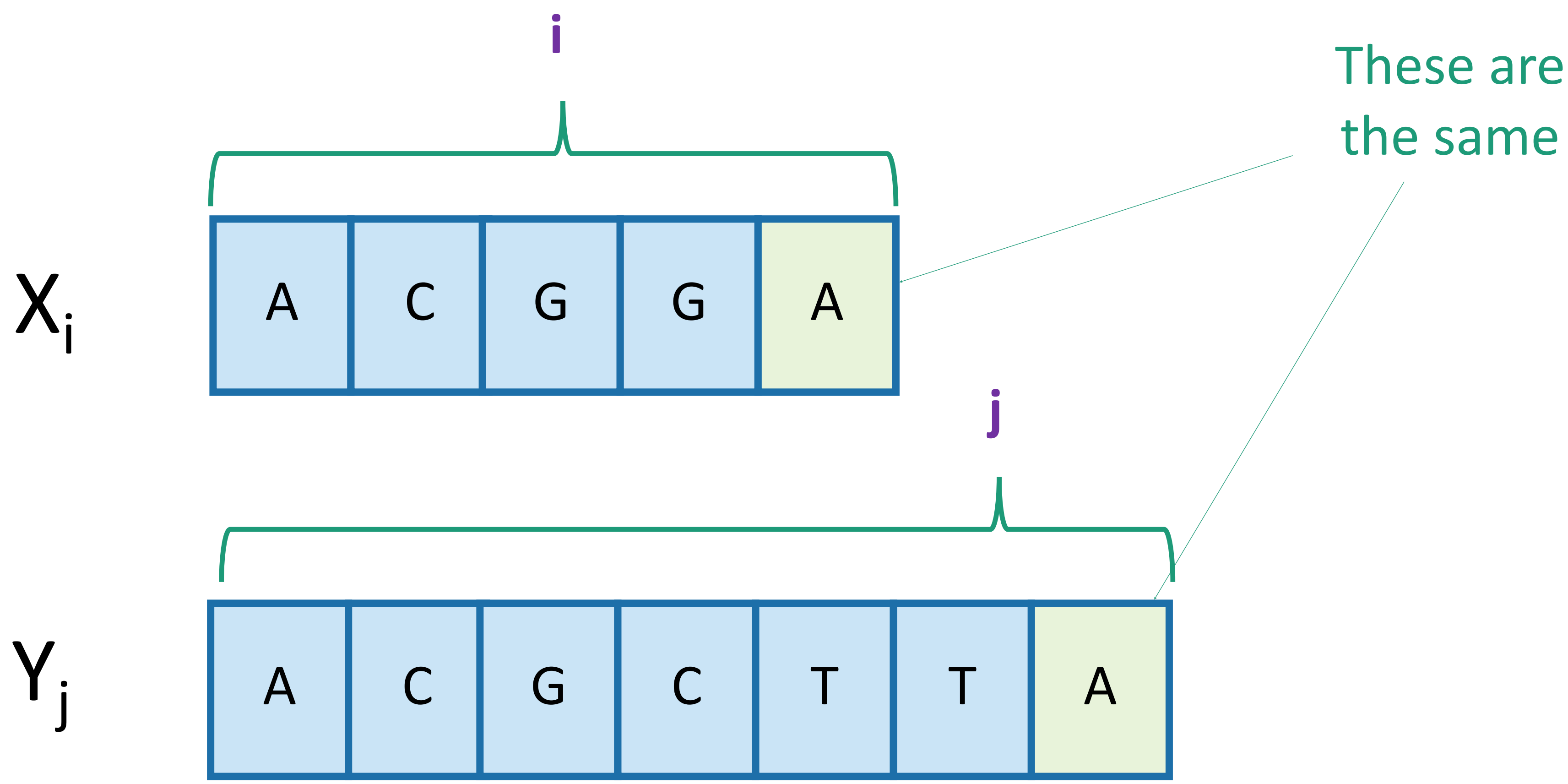


$C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$

Two cases

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let C[i,j] = length_of_LCS(X_i, Y_j)

Case 1: X[i] = Y[j]

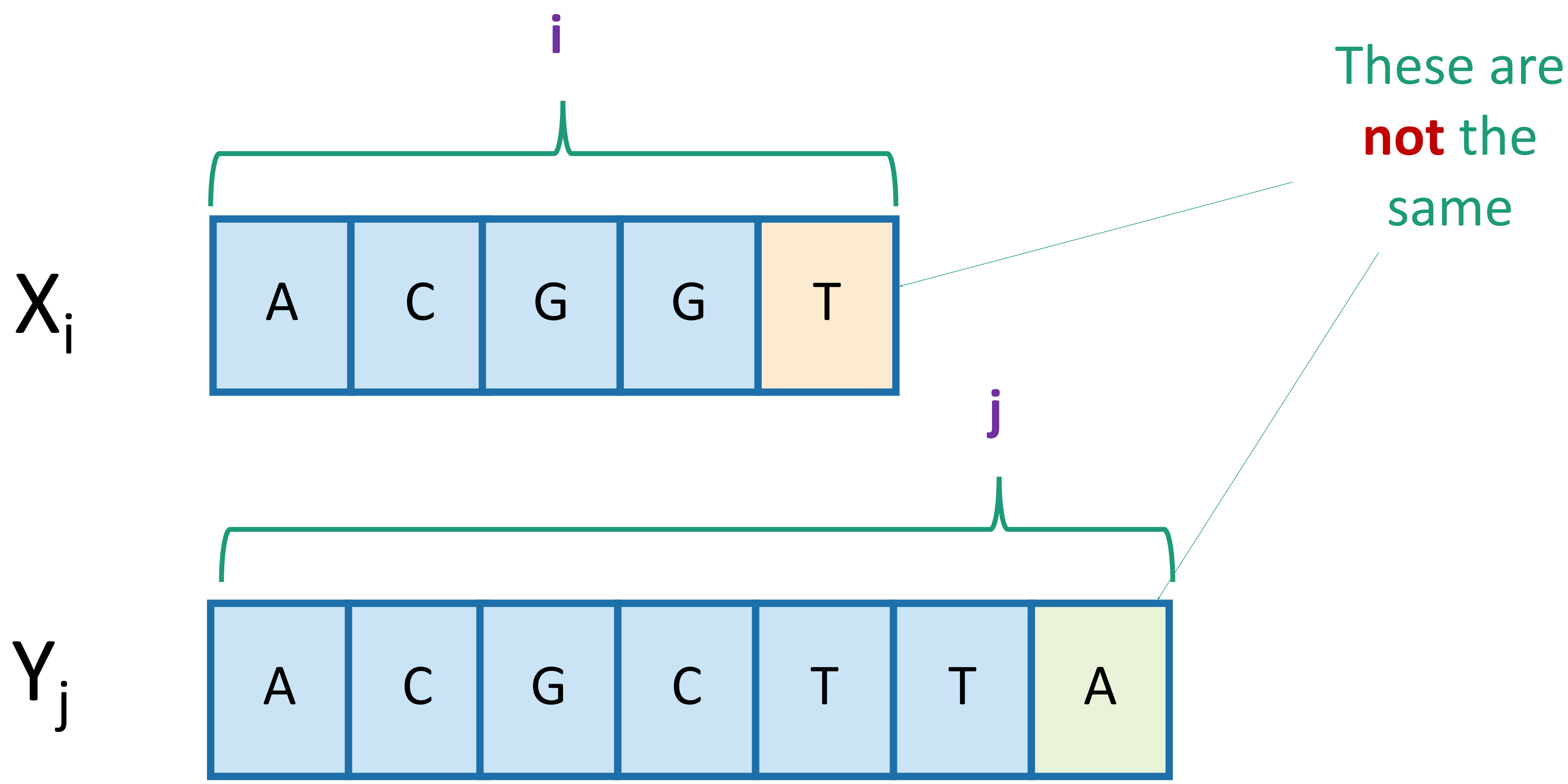


- Then C[i,j] = 1 + C[i-1,j-1].
  - because LCS(X_i,Y_j) = LCS(X_{i-1},Y_{j-1}) followed by $\boxed{A}$

Two cases

Case 2: X[i] != Y[j]

- Our sub-problems will be finding LCS's of prefixes to X and Y.
- Let $C[i,j] = \text{length\_of\_LCS}(X_i, Y_j)$



- Then $C[i,j] = \max\{ C[i-1,j], C[i,j-1] \}$.
  - either $LCS(X_i,Y_j) = LCS(X_{i-1},Y_j)$ and $\boxed{T}$ is not involved,
  - or $LCS(X_i,Y_j) = LCS(X_i,Y_{j-1})$ and $\boxed{A}$ is not involved,
  - (maybe both are not involved, that's covered by the "or").

Recursive formulation
of the optimal solution

X_0

Y_j	A	C	G	C	T	T	A

Case 0

- $$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Case 1

X_i	A	C	G	G	A		
Y_j	A	C	G	C	T	T	A

Case 2

X_i	A	C	G	G	T		
Y_j	A	C	G	C	T	T	A

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

LCS DP

- **LCS**(X, Y):
  - C[i,0] = C[0,j] = 0 for all i = 0,…,m, j=0,…n.
  - **For** i = 1,…,m and j = 1,…,n:
    - **If** X[i] = Y[j]:
      - C[i,j] = C[i-1,j-1] + 1
    - **Else:**
      - C[i,j] = max{ C[i,j-1], C[i-1,j] }
  - Return C[m,n]

Running time: O(nm)

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j-1],\ C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X	A	C	G	G	A

Y	A	C	T	G

Y

		A	C	T	G
	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

X

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X	A	C	G	G	A
Y	A	C	T	G	

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

So the LCM of X and Y has length 3.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

Example

X					
	A	C	G	G	A

Y				
	A	C	T	G

		Y			
		A	C	T	G
X	0	0	0	0	0
A	0				
C	0				
G	0				
G	0				
A	0				

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X	A	C	G	G	A
Y	A	C	T	G	

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X	A	C	G	G	A
Y	A	C	T	G	

		Y: A	C	T	G
	0	0	0	0	0
X: A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X	A	C	G	G	A

Y	A	C	T	G

Y

X		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.

That 3 must have come from the 3 above it.

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i, j-1], C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X: | A | C | G | G | A |

Y: | A | C | T | G |

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

This 3 came from that 2 – we found a match!

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X: A C G G A

Y: A C T G

		Y: A	C	T	G
X	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

That 2 may as well have come from this other 2.

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{ C[i,j-1],\ C[i-1,j] \} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Example

X: A C G G A

Y: A C T G

X \ Y		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{\, C[i, j-1],\ C[i-1, j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X	A	C	G	G	A
Y	A	C	T	G	

Y

		A	C	T	G
	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

X

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

C	G

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1, j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i, j > 0 \\ \max\{ C[i, j-1],\ C[i-1, j] \} & \text{if } X[i] \neq Y[j] \text{ and } i, j > 0 \end{cases}$$

Example

X	A	C	G	G	A

Y	A	C	T	G

		Y: A	C	T	G
X	0	0	0	0	0
A	0	1	1	1	1
C	0	1	2	2	2
G	0	1	2	2	3
G	0	1	2	2	3
A	0	1	2	2	3

- Once we've filled this in, we can work backwards.
- A diagonal jump means that we found an element of the LCS!

A	C	G

This is the LCS!

$$C[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ C[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \text{ and } i,j > 0 \\ \max\{\, C[i,j-1],\ C[i-1,j] \,\} & \text{if } X[i] \neq Y[j] \text{ and } i,j > 0 \end{cases}$$

Finding an LCS

- See CLRS for pseudocode
- Takes time O(mn) to fill the table
- Takes time O(n + m) on top of that to recover the LCS
  - We walk up and left in an n-by-m array
  - We can only do that for n + m steps.
- Altogether, we can find LCS(X,Y) in time O(mn).

Recipe for applying Dynamic Programming

- **Step 1:** Identify optimal substructure.
- **Step 2:** Find a recursive formulation for the length of the longest common subsequence.
- **Step 3:** Use dynamic programming to find the length of the longest common subsequence.
- **Step 4:** If needed, keep track of some additional info so that the algorithm from Step 3 can find the actual LCS.
- **Step 5:** If needed, code this up like a reasonable person.

<u>Edit distance</u>

- Given two strings, how can we measure how close they are?
- Ex) SNOWY, SUNNY : 2 possible alignments

```
S  —  N  O  W  Y          —  S  N  O  W  —  Y
S  U  N  N  —  Y          S  U  N  —  —  N  Y
     Cost: 3                    Cost: 5
```

- \- : gap (we may place any number of gaps in either string)
- Cost : the number of columns in which the letters differ
- *Edit distance* of two strings : the cost of their best possible alignment = minimum number of *edits* – insertions, deletions, and substitutions of characters – needed to transform the first string into the second

Dynamic programming

- *What are the subproblems*?
  - (*) There is an ordering on the subproblems, and a relation that shows how to solve a subproblem given the answers to "smaller" subproblems (subproblems that appear earlier in the ordering).
- Input : $x[1..n]$, $y[1..m]$
- Consider prefixes : $x[1..i]$, $y[1..j]$ -> call this subproblem $E(i, j)$
- Subproblem $E(7, 5)$

$\boxed{\text{E X P O N E N}}$ T I A L

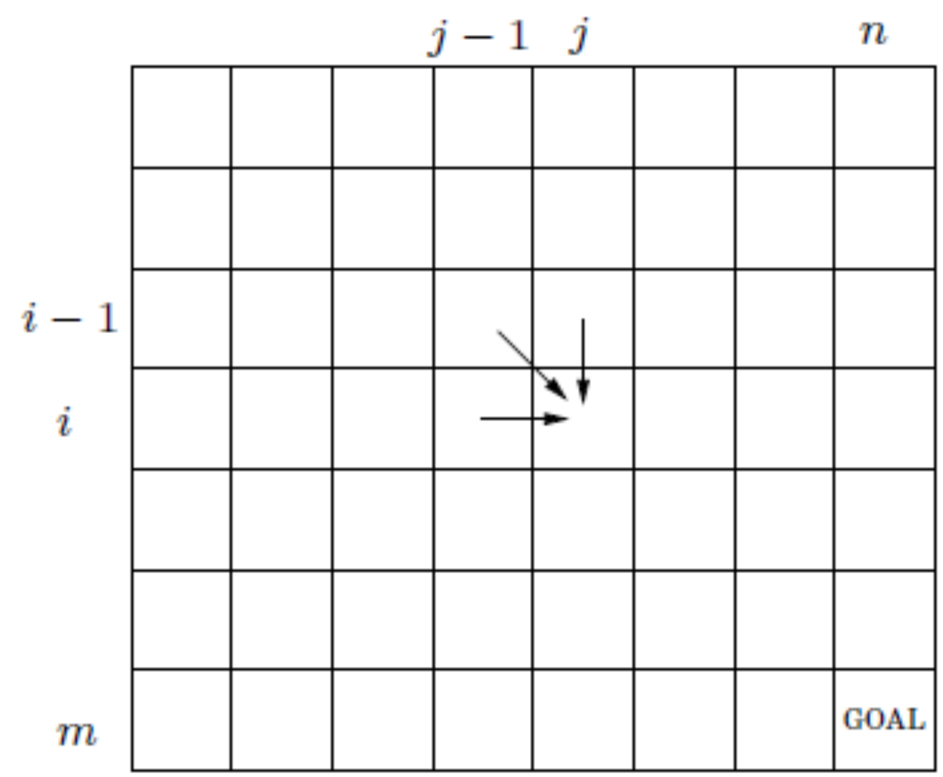
$\boxed{\text{P O L Y N}}$ O M I A L

- Goal : $E(m, n)$

- Express $E(i, j)$ in terms of smaller subproblems!
- The rightmost column of the best alignment can be one of the following :

$$\begin{array}{c} x[i] \\ - \end{array} \qquad \textbf{or} \qquad \begin{array}{c} - \\ y[j] \end{array} \qquad \textbf{or} \qquad \begin{array}{c} x[i] \\ y[j] \end{array}$$

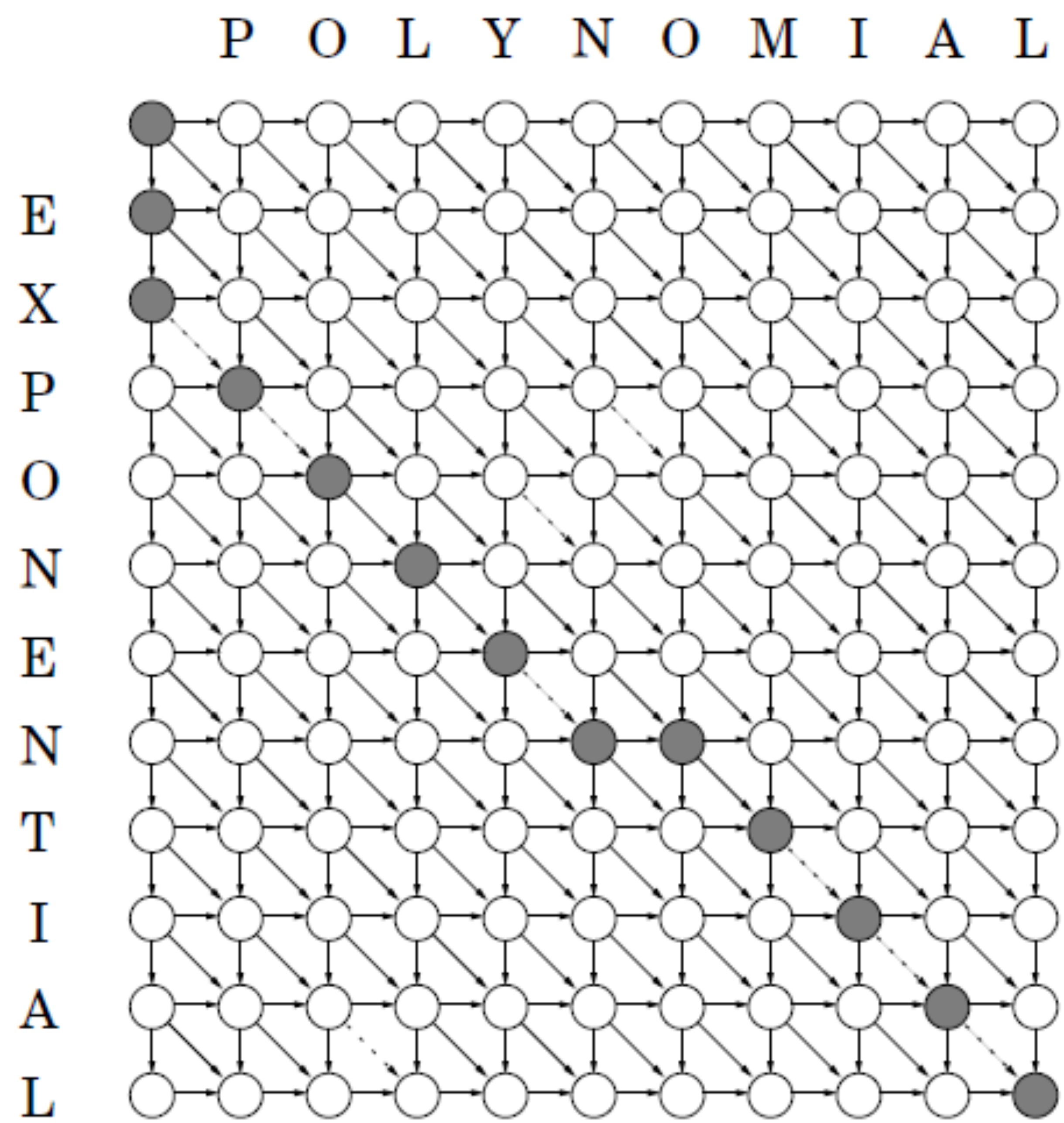
- **$E(i, j)$ = min {1 + $E(i$-1, $j)$, 1+ $E(i, j$-1), diff(i, j) + $E(i$-1, j-1) }**
- where diff(i, j) = 0 if $x[i] = y[j]$ and 1 otherwise.
- **Base cases** : i=0 or j=0
- The answers to all the subproblems $E(i, j)$ form a two-dimensional table.



```
for i = 0, 1, 2, ..., m:
    E(i, 0) = i
for j = 1, 2, ..., n:
    E(0, j) = j
for i = 1, 2, ..., m:
    for j = 1, 2, ..., n:
        E(i, j) = min{E(i − 1, j) + 1, E(i, j − 1) + 1, E(i − 1, j − 1) + diff(i, j)}
return E(m, n)
```

		P	O	L	Y	N	O	M	I	A	L
	0	1	2	3	4	5	6	7	8	9	10
E	1	1	2	3	4	5	6	7	8	9	10
X	2	2	2	3	4	5	6	7	8	9	10
P	3	2	3	3	4	5	6	7	8	9	10
O	4	3	2	3	4	5	5	6	7	8	9
N	5	4	3	3	4	4	5	6	7	8	9
E	6	5	4	4	4	5	5	6	7	8	9
N	7	6	5	5	5	4	5	6	7	8	9
T	8	7	6	6	6	5	5	6	7	8	9
I	9	8	7	7	7	6	6	6	6	7	8
A	10	9	8	8	8	7	7	7	7	6	7
L	11	10	9	8	9	8	8	8	8	7	6

Underlying dag



Common subproblems

- Finding the right subproblem takes creativity and experimentation.
- Standard choices

i. The input is $x_1, x_2, \ldots, x_n$ and a subproblem is $x_1, x_2, \ldots, x_i$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is therefore linear.

ii. The input is $x_1, \ldots, x_n$, and $y_1, \ldots, y_m$. A subproblem is $x_1, \ \ldots, x_i$ and $y_1, \ldots, y_j$.

$$\boxed{x_1 \quad x_2 \quad x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

$$\boxed{y_1 \quad y_2 \quad y_3 \quad y_4 \quad y_5} \quad y_6 \quad y_7 \quad y_8$$

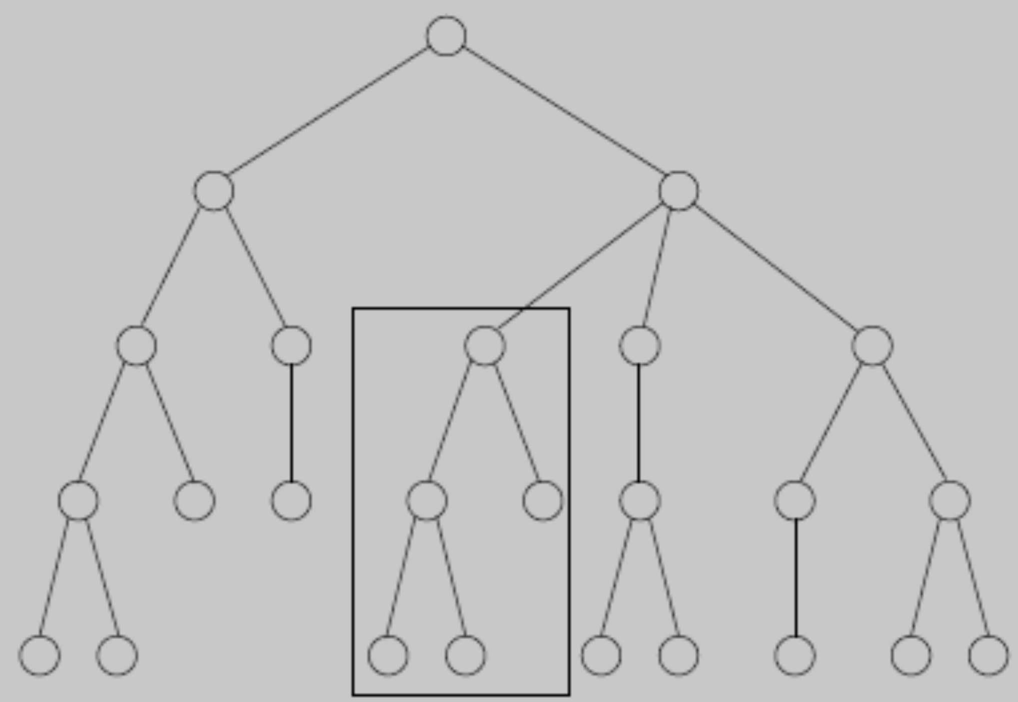
The number of subproblems is $O(mn)$.

iii. The input is $x_1, \ldots, x_n$ and a subproblem is $x_i, x_{i+1}, \ldots, x_j$.

$$x_1 \quad x_2 \quad \boxed{x_3 \quad x_4 \quad x_5 \quad x_6} \quad x_7 \quad x_8 \quad x_9 \quad x_{10}$$

The number of subproblems is $O(n^2)$.

iv. The input is a rooted tree. A subproblem is a rooted subtree.



Knapsack

- Given a knapsack of capacity W, n items of weight $w1, \ldots, wn$ and value $v1, \ldots, vn$, choose the most valuable combination of items.
- E.g.) W=10

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

- Two versions :
  - 1) allow unlimited quantities :
    pick item 1 and two of item 4 (total \$48)
  - 2) allow only 1 of each item :
    pick items 1 and 3 (total \$46).

Knapsack with repetitions

- What are the subproblems?
- Define $K(w)$ = maximum value achievable with a knapsack of capacity w.
- If the optimal solution to $K(w)$ includes item i, then removing it leaves an optimal solution to $K(w\text{-}wi)$.
- We don't know which i, so try all possibilities.

$$K(w) = \max_{i:w_i \leq w} \{K(w - w_i) + v_i\}$$

Algorithm

$K(0) = 0$
$\texttt{for } w = 1 \texttt{ to } W\texttt{:}$
$\quad K(w) = \max\{K(w - w_i) + v_i : w_i \leq w\}$
$\texttt{return } K(W)$

Analysis

- This algorithm fills in a one-dimensional table of length W+1, in left-to-right order.
- Each entry can take up to O(n) time to compute.
- The overall running time = O(nW).

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

ω	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
K(ω)	0	0	9	14	18	23	30	32	39	44	48

K(ω) = max{k(3-3)+v3,k(3-2)+v2}={0+14,0+9} =14

K(ω) = max{k(4-3)+v3,k(4-2)+v2}={0+14,9+9} =18

K(ω) = max{k(5-1)+v4,k(5-2)+v3,k(5-3)+v2,k(5-4)+v1}

={0+16,9+14,14+9,18+0} =18

Knapsack without repetition

- Need to refine the subproblem to carry **additional** information about the items being used by adding a **second parameter**, $0 \leq j \leq n$.
- $\boldsymbol{K(w, j)}$ **= maximum value achievable** using a knapsack of capacity w and items $1,\ldots,j$.
- Goal : $K(W, n)$.
- Express $K(w, j)$ in terms of smaller subproblems considering whether item j is needed or not.
- $\boldsymbol{K(w, j) = \max \{ K(w\text{-}wj, j\text{-}1) + vj , K(w, j\text{-}1) \}}$
- (The first case is invoked only if $wj \leq w$)

Algorithm

```
Initialize all K(0,j) = 0 and all K(w,0) = 0
for j = 1 to n:
   for w = 1 to W:
      if w_j > w:   K(w,j) = K(w,j-1)
      else:   K(w,j) = max{K(w,j-1), K(w-w_j, j-1) + v_j}
return K(W,n)
```

Analysis

- The algorithm fills out a 2-dimensional table, with W+1 rows and n+1 columns.
- Each table entry takes constant time.
- Running time : O(nW).

Item	Weight	Value
1	6	\$30
2	3	\$14
3	4	\$16
4	2	\$9

ω	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
j=0	0	0	0	0	0	0	0	0	0	0	0
j=1	0	0	0	0	0	0	30	30	30	30	30
j=2	0	0	0	14	14	14	30	30	30	44	44
j=3	0	0	0	14	16	16	30	30	30	44	46
j=4	0	0	9	14	16	23	30	30	39	44	46

K(ω,j) = max{k(w,j-1),k(w-wj,j-1)+vj}={30,30+14} ={44}

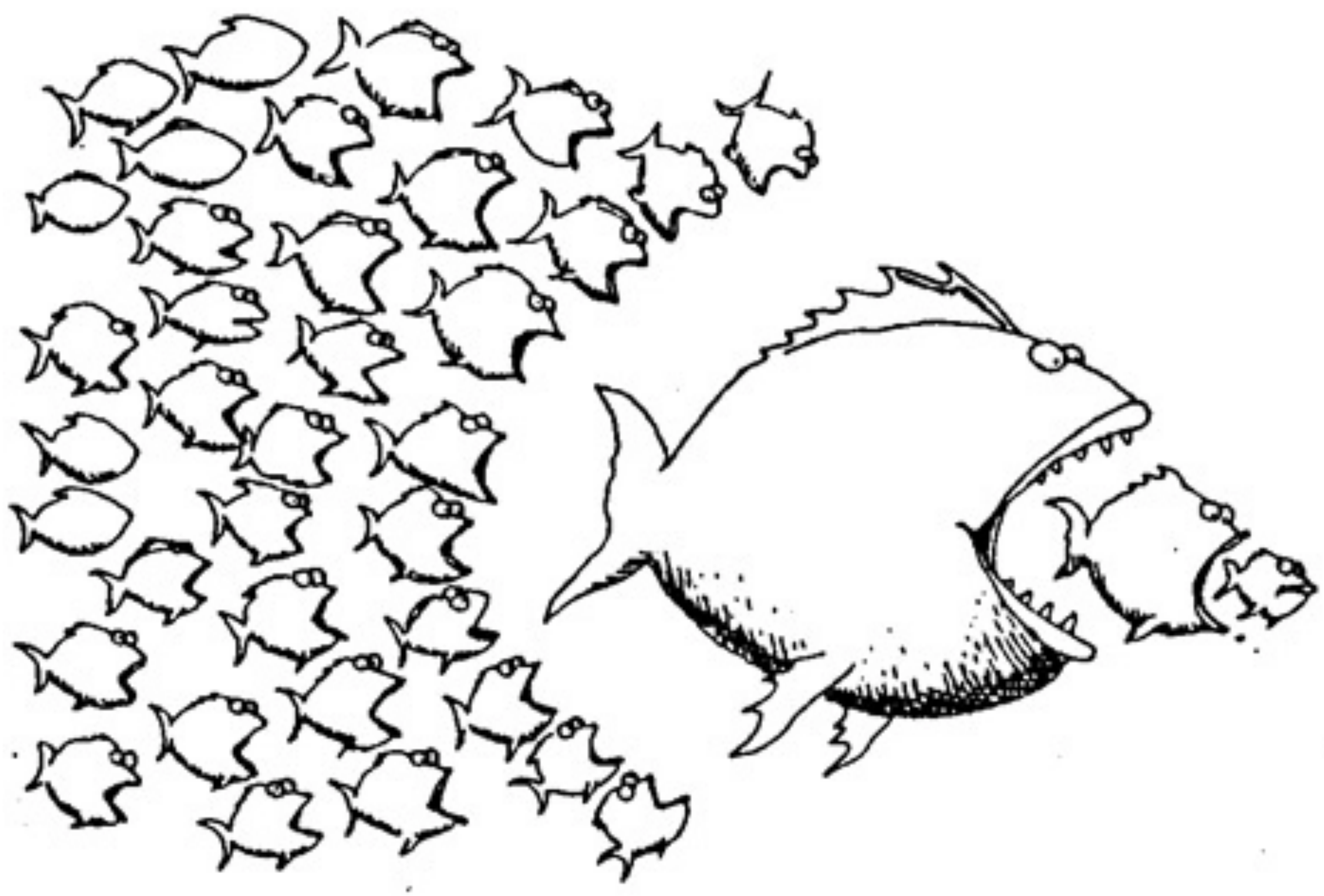
K(ω,j) = max{k(w,j-1),k(w-wj,j-1)+vj}={44,30+16} ={46}

Memoization

After computing a solution to a subproblem, **store it in a table**.
Subsequent calls check the table to **avoid redoing work**.

Two ways to think about and/or implement DP algorithms

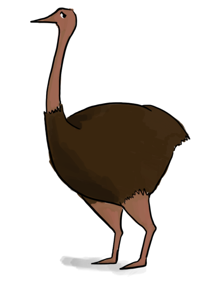
- Top down
- Bottom up



This picture isn't hugely relevant but I like it.

Candidate alg.

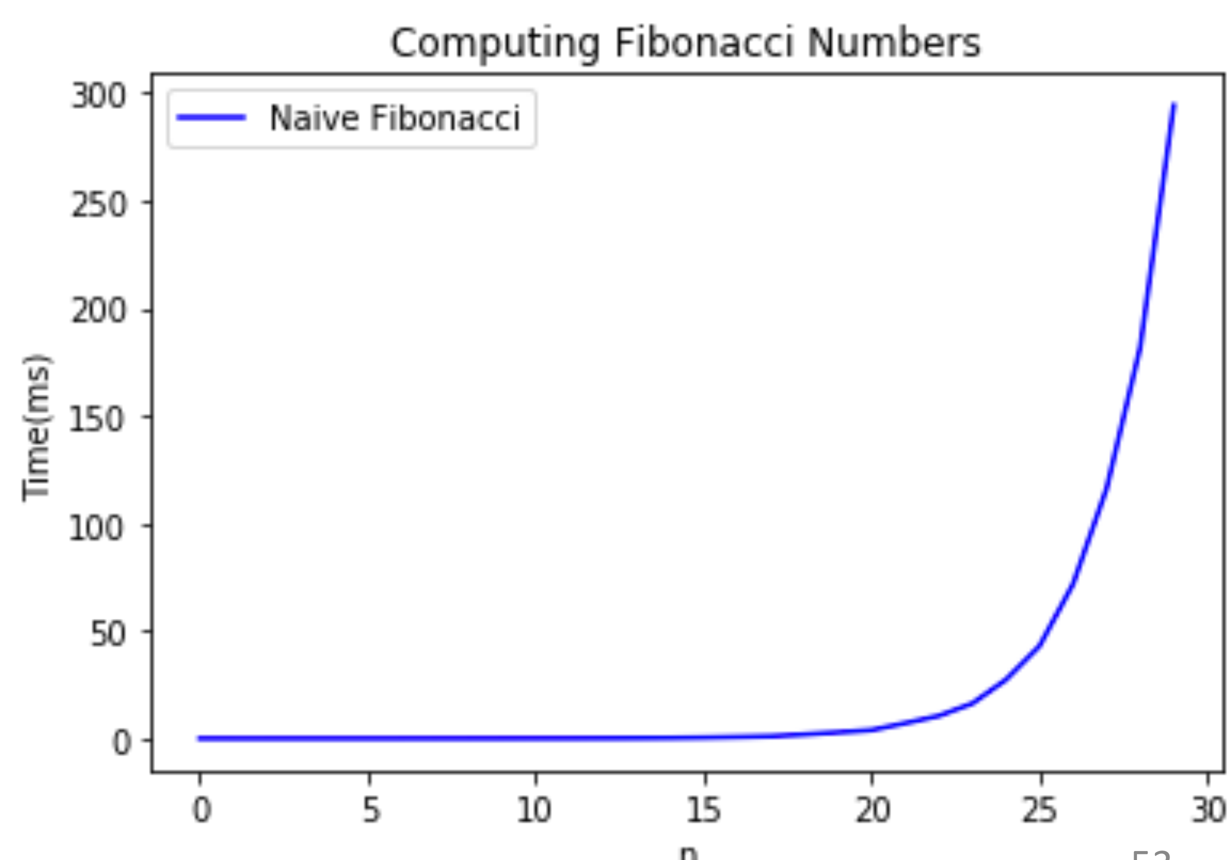
See CLRS Problem 4-4 for a walkthrough of how fast the Fibonacci numbers grow!



```
• def Fibonacci(n):
    • if n == 0 or n == 1:
        • return 1
    • return Fibonacci(n-1) + Fibonacci(n-2)
```

Running time?

- T(n) = T(n-1) + T(n-2) + O(1)
- T(n) $\geq$ T(n-1) + T(n-2) for n ≥ 2
- So T(n) grows *at least* as fast as the Fibonacci numbers themselves…
- Fun fact, that's like ϕ^n where $\phi = \frac{1+\sqrt{5}}{2}$ is the golden ratio.
- aka, **EXPONENTIALLY QUICKLY** ☹

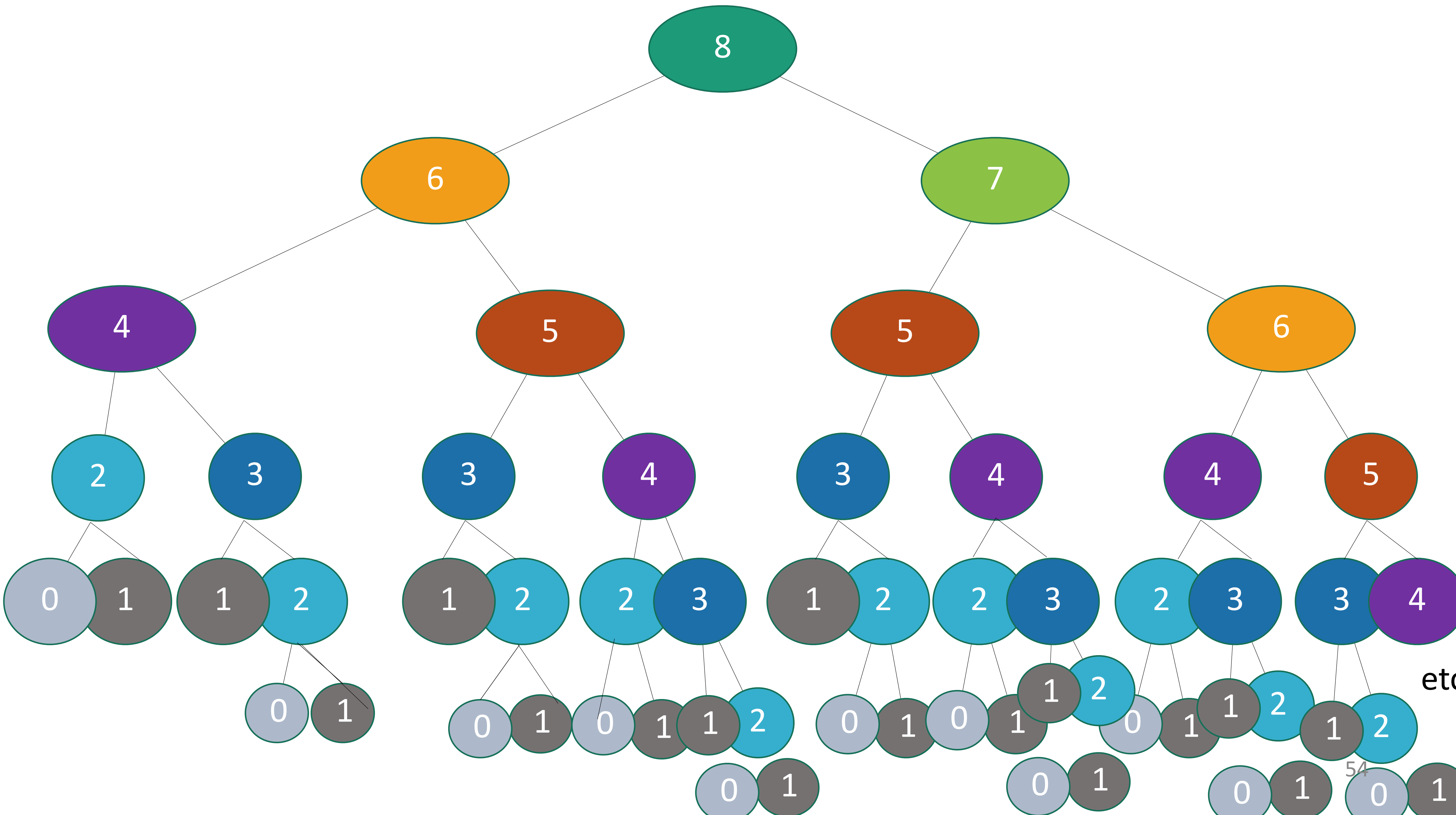


See IPython notebook for lecture 12

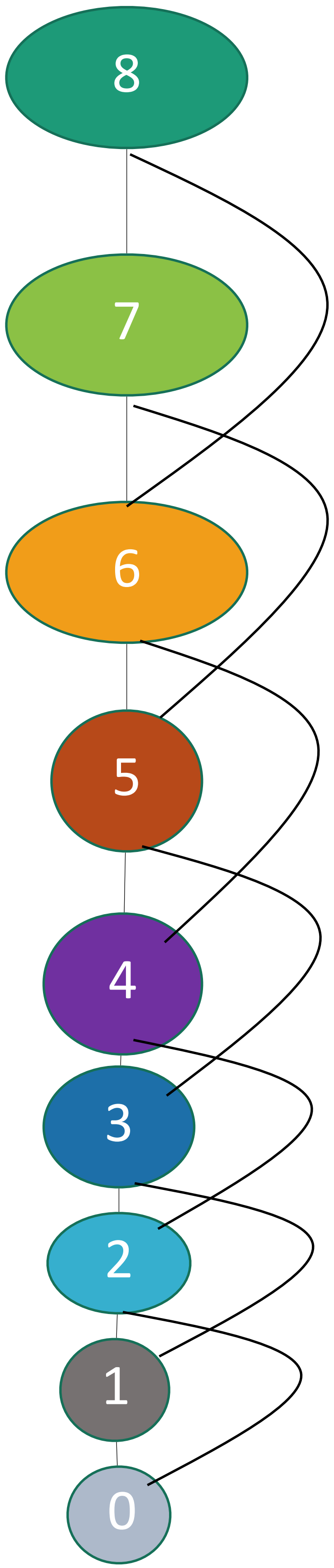
What's going on?

Consider Fib(8)

That's a lot of repeated computation!

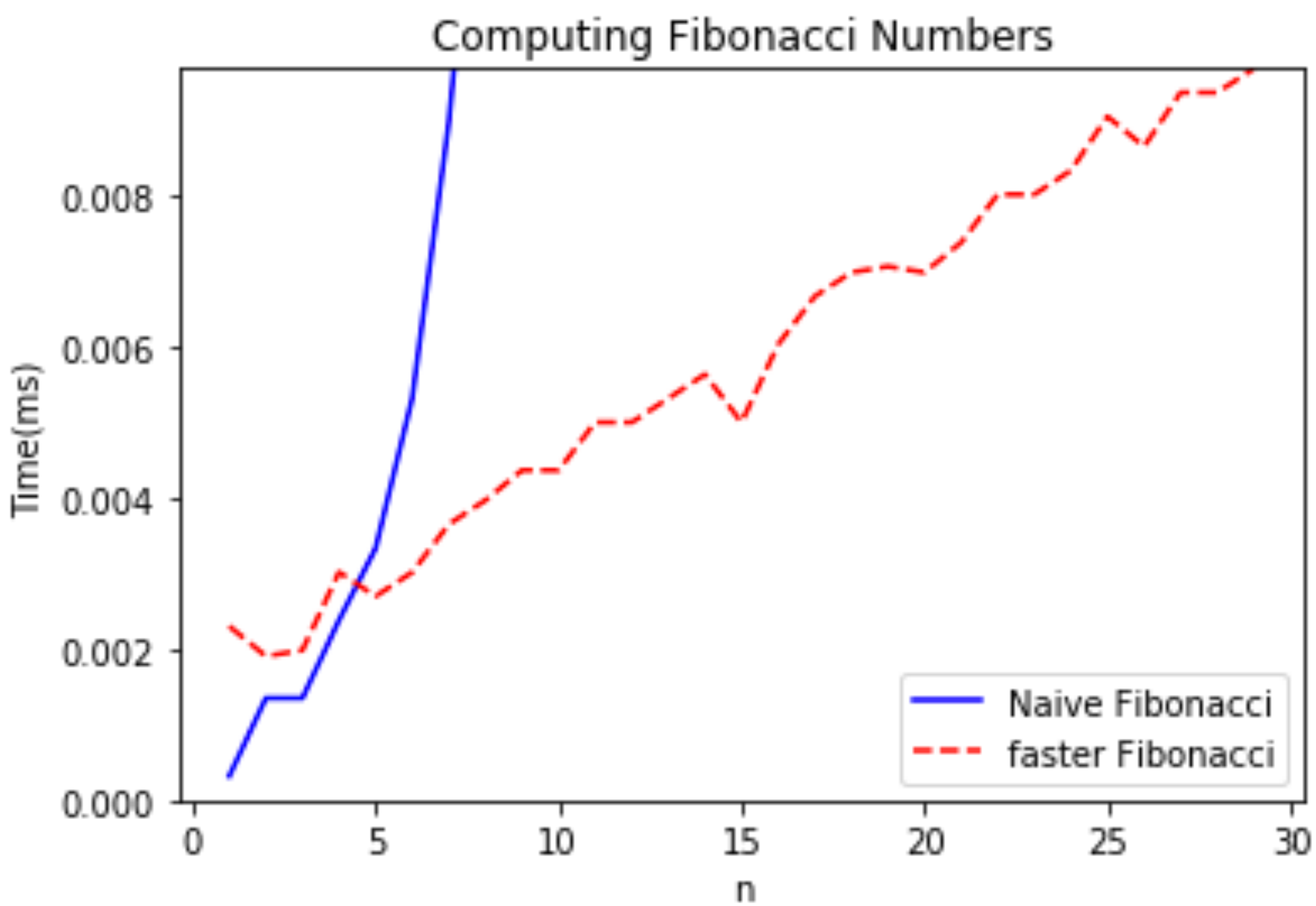


Maybe this would be better:




```
def fasterFibonacci(n):
  • F = [1, 1, None, None, …, None ]
    • \\ F has length n + 1
  • for i = 2, …, n:
    • F[i] = F[i-1] + F[i-2]
  • return F[n]
```

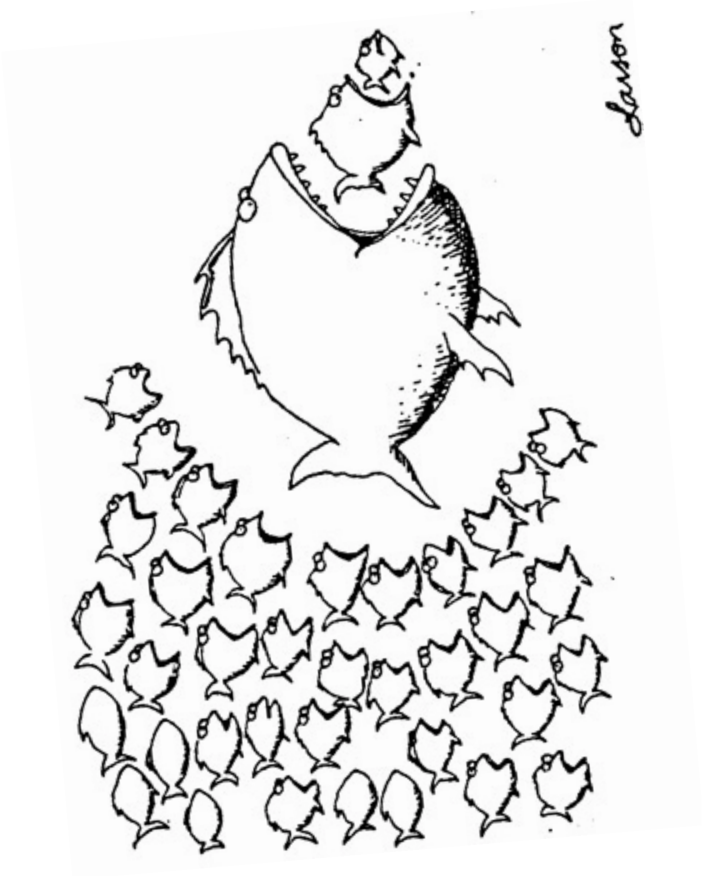
Much better running time!



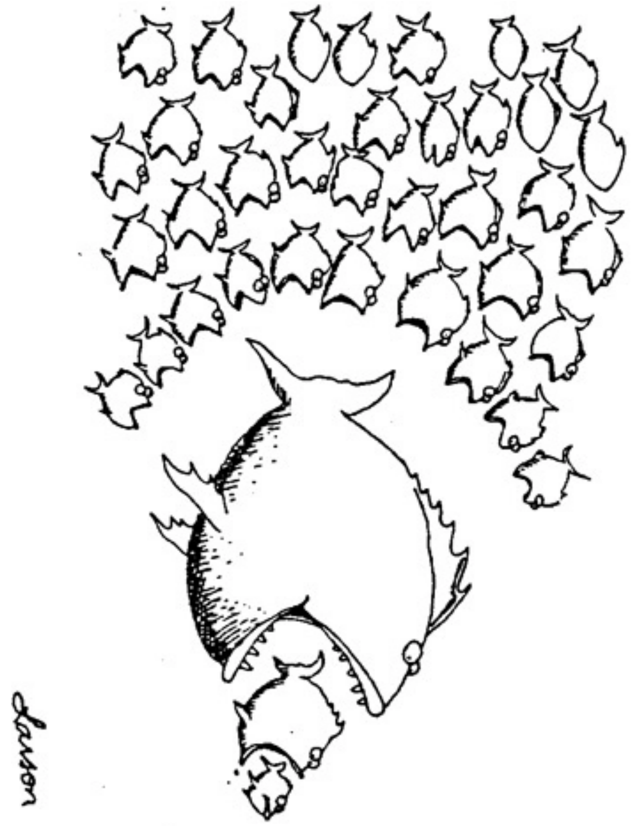
Bottom up approach

what we just saw.

- For Fibonacci:
- Solve the small problems first
  - fill in F[0],F[1]
- Then bigger problems
  - fill in F[2]
- …
- Then bigger problems
  - fill in F[n-1]
- Then finally solve the real problem.
  - fill in F[n]



Top down approach

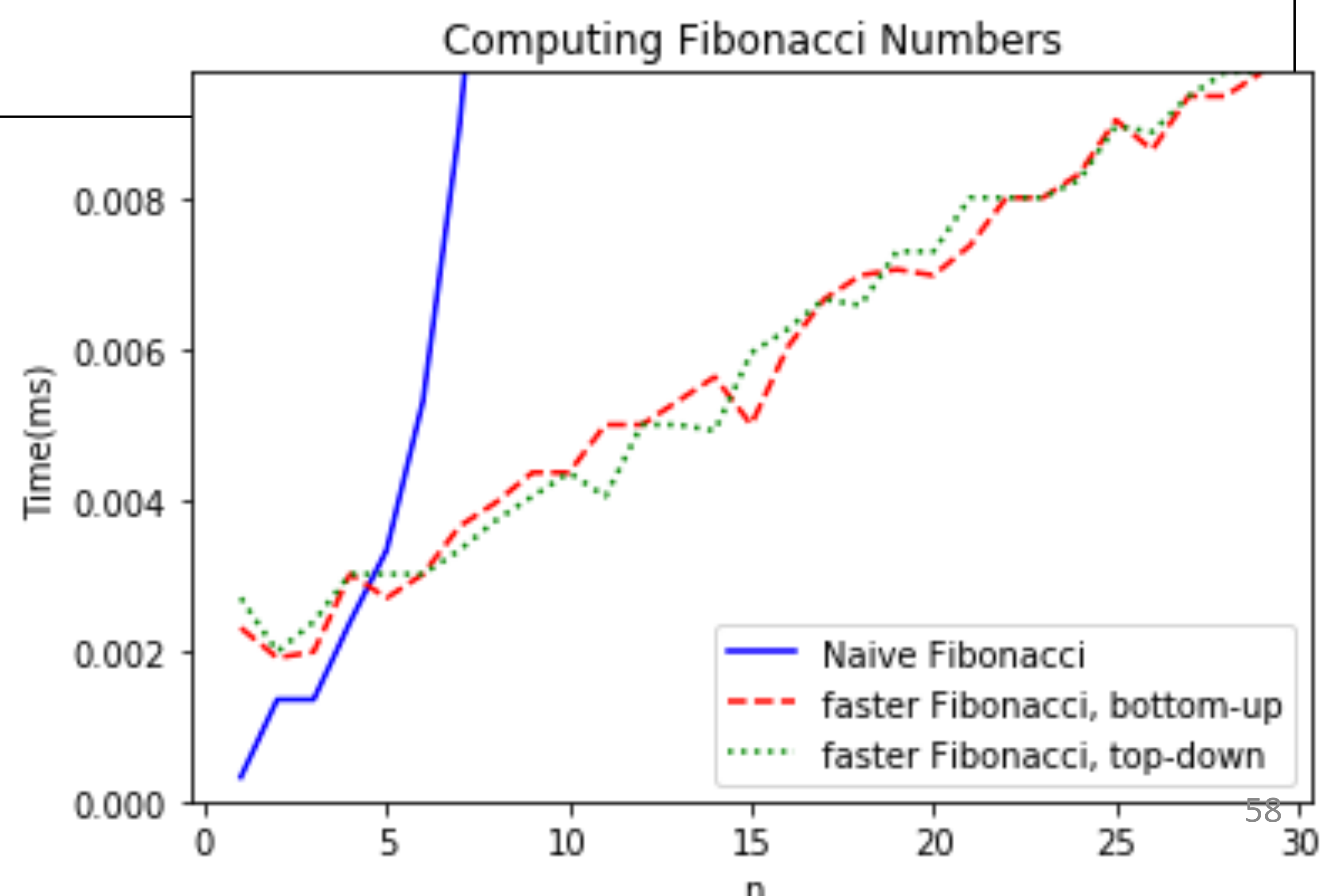


- Think of it like a recursive algorithm.
- To solve the big problem:
  - Recurse to solve smaller problems
    - Those recurse to solve smaller problems
      - etc..

- The difference from divide and conquer:
  - Keep track of what small problems you've already solved to prevent re-solving the same problem twice.
  - Aka, "**memoization**"

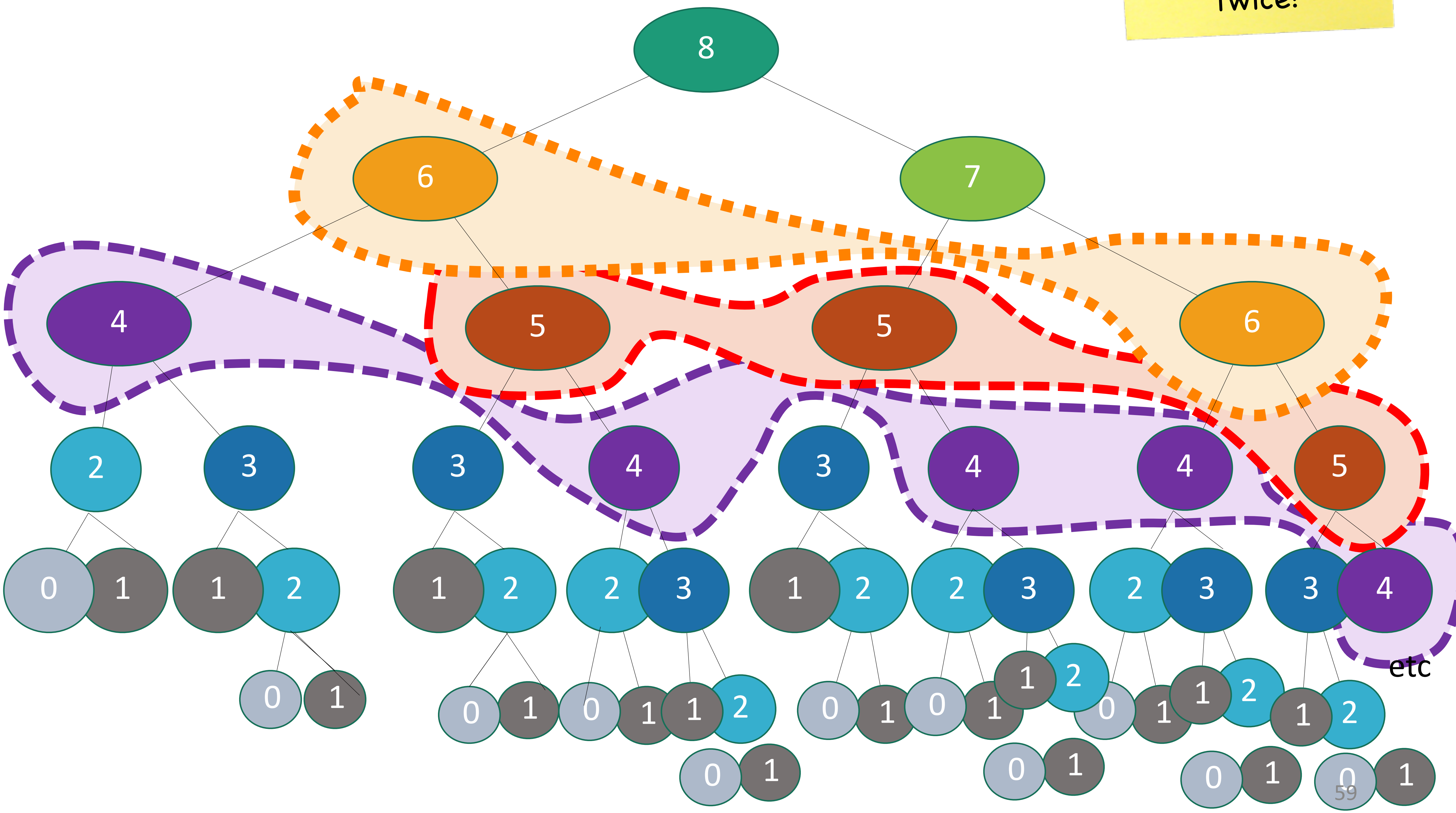
Example of top-down Fibonacci

```
• define a global list F = [1,1,None, None, …, None]
• def Fibonacci(n):
    • if F[n] != None:
        • return F[n]
    • else:
        • F[n] = Fibonacci(n-1) + Fibonacci(n-2)
    • return F[n]
```



Memo-ization visualization

Collapse repeated nodes and don't do the same work twice!

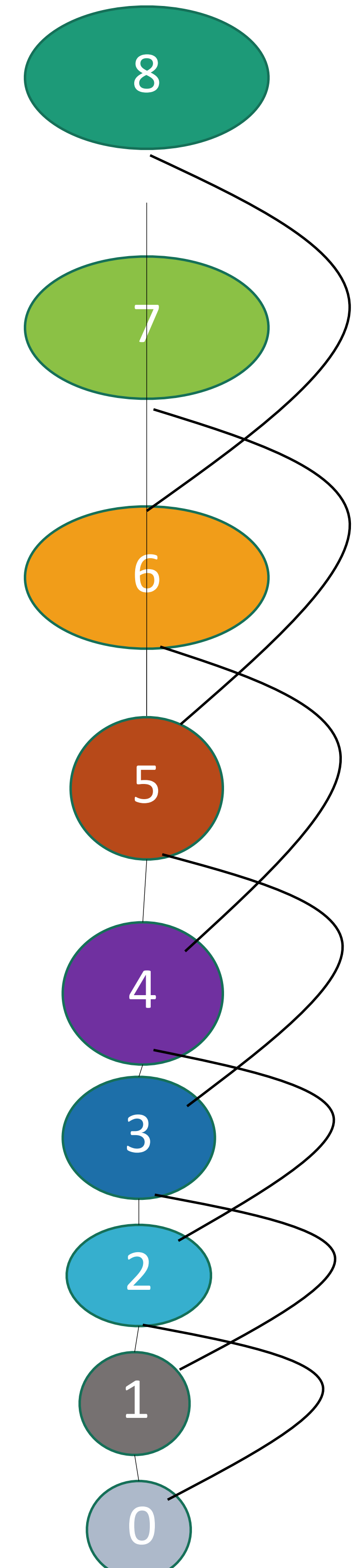


Memo-ization Visualization

ctd

Collapse repeated nodes and don't do the same work twice!

But otherwise treat it like the same old recursive algorithm.



- define a global list F = [1,1,None, None, …, None]
- **def** Fibonacci(n):
  - **if** F[n] != None:
    - **return** F[n]
  - **else:**
    - F[n] = Fibonacci(n-1) + Fibonacci(n-2)
  - **return** F[n]

What have we learned?

- ***Dynamic programming:***
  - Paradigm in algorithm design.
  - Uses **optimal substructure**
  - Uses **overlapping subproblems**
  - Can be implemented **bottom-up** or **top-down**.
  - It's a fancy name for a pretty common-sense idea: