CS 341: ALGORITHMS

Trevor Brown
trevor.brown@uwaterloo.ca
DC 2338, Office hour M3-4pm
THIS TIME

- Dynamic programming (DP) algorithms
  - Coin changing
  - Longest common subsequence (partially covered)
- Memoization VS dynamic programming (didn't cover)
There is a denomination with unit value!

In 0-1 knapsack, we only considered two subproblems in our recurrence: taking an item, or not.

Here we can do more than use a coin denomination or not.

What do you think?
Let $N[i, t]$ denote the optimal solution to the subproblem consisting of the first $i$ coin denominations $d_1, \ldots, d_i$ and target sum $t$.

<table>
<thead>
<tr>
<th>Exploring: some sensible base case(s)?</th>
</tr>
</thead>
<tbody>
<tr>
<td>General case:</td>
</tr>
<tr>
<td>What are the different ways we could use coin denomination $d_i$?</td>
</tr>
<tr>
<td>What subproblems / solutions should we use?</td>
</tr>
</tbody>
</table>

Also $N[i, 0] = 0$ for all $i$.

Final recurrence relation
**FILLING THE ARRAY**

\( N[1 \ldots n, 1 \ldots T] \):

\[
N[i, t] = \begin{cases} 
\min\{j + N[i - 1, t - jd_i] : 0 \leq j \leq |t/d_i|\} & \text{if } i \geq 2 \\
\; t & \text{if } i = 1. \text{ OR } t = 0
\end{cases}
\]

- **i-axis** (coin type)
- **t-axis** (target sum remaining)

No data dependencies on any other array cells.

(recall: \( N[i, t] \) uses coin types 1..i)
FILLING THE ARRAY $N[1 \ldots n, 1 \ldots T]$:

No data dependencies on any other array cells.

**$i$-axis** (coin type)

(recall: $N[i, t]$ uses coin types $1..i$)

$N[i, t] = \begin{cases} \min\{j + N[i - 1, t - jd_i] : 0 \leq j \leq [t/d_i]\} & \text{if } i \geq 2 \\ t & \text{if } i = 1. \text{ OR } t = 0 \end{cases}$
FILLING THE ARRAY $N[1 \ldots n, 1 \ldots T]$:

We only look at the previous $i$-row!

Consider cell $N[i, t]$

It is sufficient to fill: row $i=1$ (base case), then for $i = 2 \ldots n$, for $t = 0 \ldots T$

$N[i, t] = \begin{cases} \min\{j + N[i-1, t - jd_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\ t & \text{if } i = 1. \quad \text{OR } t = 0 \end{cases}$
Algorithm: Coin Changing ($d_1, \ldots, d_n$)

```plaintext
comment: $d_1 = 1$

for $t \leftarrow 0$ to $T$
  do 
    $N[1, t] \leftarrow t$
    $A[1, t] \leftarrow t$

for $i \leftarrow 2$ to $n$
  do 
    for $t \leftarrow 0$ to $T$
      do 
        $N[i, t] \leftarrow N[i - 1, t]$
        $A[i, t] \leftarrow 0$
        do 
          for $j \leftarrow 1$ to $\lfloor t/d_i \rfloor$
            do 
              if $j + N[i - 1, t - j d_i] < N[i, t]$
                then 
                  $N[i, t] \leftarrow j + N[i - 1, t - j d_i]$
                  $A[i, t] \leftarrow j$

return $(N[n, T])$
```

Base case (row $i=1$): using unit-valued coins

General case ($i \geq 2$): using other coin types

Additionally, we maintain $A[i, t] = \# \text{ of coins of type } d_i \text{ used in } N[i, t]$

Complexity?:

$$N[i, t] = \begin{cases} 
\min\{j + N[i - 1, t - j d_i] : 0 \leq j \leq \lfloor t/d_i \rfloor\} & \text{if } i \geq 2 \\
0 & \text{if } i = 1.
\end{cases}$$
Recall: an algorithm is "polynomial time" only if its runtime is polynomial in the size (number of bits) in the input!

The input size $s$ is: $\text{bits}(d_1) + \text{bits}(d_2) + \ldots + \text{bits}(d_n) + \text{bits}(T)$.

Suppose coin denominations are small constants, so each $d_i$ fits in a machine word, and $\text{bits}(d_i) \in \Theta(1)$.

But, maybe $T$ is very large and can't fit in a machine word. So, $\text{bits}(T) \in \Theta \left( \log_2 T \right)$.

Then, $s \in n \cdot \Theta(1) + \Theta(\log_2 T) = \Theta(n + \log_2 T)$.

So, for example, if $n = 1$, then $s \in \Theta(\log_2 T)$, which means $T \in \Theta(2^s)$. $T$ is exponential in the input size $s$, and hence so is the running time!
Computing the Optimal Set of Coins

We trace back through the table to compute the optimal set of coins.

There are two possible approaches:

Recompute the relevant table entries \( N[i, t] \) during the traceback.

Store relevant extra information, while the table \( N[i, t] \) is being constructed, in another table \( A[i, t] \).

Suppose we follow the second approach.

The \( A[i, t] \) values make it easy to determine number of coins of each denomination in the optimal solution \( N[i, T] \).

This is kind of similar to 0-1 Knapsack.
Exercise for home: compute the correct output without using \( A[i,t] \)

\[
\text{Algorithm: } \text{Trace}(d_1, \ldots, d_n, T, A) \\
t \leftarrow T \\
\text{for } i \leftarrow n \text{ downto } 1 \\
\quad \text{do } \begin{cases} 
    a_i \leftarrow A[i,t] \\
    t \leftarrow t - a_i d_i 
\end{cases} \\
\text{return } (a_1, \ldots, a_n)
\]

Note that we only need the \( A[i,t] \) values to do the traceback.
Problem 5.3

Longest Common Subsequence

Instance: Two sequences $X = (x_1, \ldots, x_m)$ and $Y = (y_1, \ldots, y_n)$ over some finite alphabet $\Gamma$.

Find: A maximum length sequence $Z$ that is a subsequence of both $X$ and $Y$.

$Z = (z_1, \ldots, z_\ell)$ is a subsequence of $X$ if there exist indices $1 \leq i_1 < \cdots < i_\ell \leq m$ such that $z_j = x_{i_j}$, $1 \leq j \leq \ell$.

Similarly, $Z$ is a subsequence of $Y$ if there exist (possibly different) indices $1 \leq h_1 < \cdots < h_\ell \leq n$ such that $z_j = y_{h_j}$, $1 \leq j \leq \ell$. 
EXAMPLES

- $X=\text{aaaaa}$  $Y=\text{bbbb}$  $Z=\text{LCS}(X,Y)=$?
  - $Z=\epsilon$ (empty sequence)
- $X=\text{abcde}$  $Y=\text{bcd}$  $Z=\text{LCS}(X,Y)=$?
  - $Z=\text{bcd}$
- $X=\text{abcde}$  $Y=\text{labe}$  $Z=\text{LCS}(X,Y)=$?
  - $Z=\text{abe}$
POSSIBLE GREEDY SOLUTIONS?

- Alg: for each \( x_i \in X \), try to choose a matching \( y_j \in Y \) that is to the right of all previously chosen \( y_j \) values

  - \( X=\text{abcde} \) \quad Y=\text{labef} 
  - \( X=\text{ab}cde \) \quad Y=\text{labef} 
  - \( X=\text{abc}de \) \quad Y=\text{labef} \ [\text{no suitable} \ y_j \ \text{found}] 
  - \( X=\text{abcd}e \) \quad Y=\text{labef} \ [\text{no suitable} \ y_j \ \text{found}] 
  - \( X=\text{abcde} \) \quad Y=\text{labef} 
  - \( Z=\text{abe} \) \quad \text{Optimal?}
POSSIBLE GREEDY SOLUTIONS?

- Alg: for each $x_i \in X$, try to choose a matching $y_j \in Y$ that is to the right of all previously chosen $y_j$ values
  - $X=\text{azbracadabra}$, $Y=\text{abracadabraz}$
  - $X=\text{azbracadabra}$, $Y=\text{abracadabraz}$
  - $X=\text{azbracadabra}$, $Y=\text{abracadabraz}$ [no $y_j$ after $z$]
  - $X=\text{azbracadabra}$, $Y=\text{abracadabraz}$ [no $y_j$ after $z$]

Blindly taking $z$ is bad. How to decide whether to take or leave $z$?

Try both possibilities! (Brute force / dynamic programming)

Optimal?

Similar greedy alg that goes right-to-left works for this input, but fails for other inputs.
THINKING ABOUT SUBPROBLEMS

- Entire problem: # characters in LCS(X,Y)
- How to reduce problem size? Reduce size of X or Y.
- Define X’ and Y’ as follows

<table>
<thead>
<tr>
<th>X</th>
<th>x₁</th>
<th>x₂</th>
<th>x₃</th>
<th>x₄</th>
<th>...</th>
<th>xₘ₋₁</th>
<th>xₘ</th>
</tr>
</thead>
<tbody>
<tr>
<td>X’</td>
<td>x₁</td>
<td>x₂</td>
<td>x₃</td>
<td>x₄</td>
<td>...</td>
<td>xₘ₋₁</td>
<td></td>
</tr>
<tr>
<td>Y</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>...</td>
<td>yₙ₋₁</td>
<td>yₙ</td>
</tr>
<tr>
<td>Y’</td>
<td>y₁</td>
<td>y₂</td>
<td>y₃</td>
<td>y₄</td>
<td>...</td>
<td>yₙ₋₁</td>
<td></td>
</tr>
</tbody>
</table>
Consider an **optimal solution** Z

Can we express Z in terms of X' and Y' instead of X and Y?

- By definition, $Z = LCS(X, Y)$

<table>
<thead>
<tr>
<th>Z</th>
<th>$z_1$</th>
<th>$z_2$</th>
<th>$z_3$</th>
<th>$z_{\ell-1}$</th>
<th>$z_\ell$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>X</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>...</th>
<th>$x_{m-1}$</th>
<th>$x_m$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>X'</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>$x_5$</th>
<th>...</th>
<th>$x_{m-1}$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>$y_5$</th>
<th>...</th>
<th>$y_{n-1}$</th>
<th>$y_n$</th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>Y'</th>
<th>$y_1$</th>
<th>$y_2$</th>
<th>$y_3$</th>
<th>$y_4$</th>
<th>$y_5$</th>
<th>...</th>
<th>$y_{n-1}$</th>
</tr>
</thead>
</table>
CONSIDER AN OPTIMAL SOLUTION Z

- By definition, \( Z = \text{LCS}(X,Y) \)
- Suppose \( z_\ell \) matches both \( x_m \) and \( y_n \)

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
Z & z_1 & z_2 & \cdots & z_{\ell-1} & z_\ell \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c}
\hline
X & x_1 & x_2 & x_3 & x_4 & \cdots & x_{m-1} & x_m \\
\hline
Y & y_1 & y_2 & y_3 & y_4 & \cdots & y_{n-1} & y_n \\
\hline
\end{array}
\]

\[
\begin{array}{c|c|c|c|c|c|c|c|c|c|c}
\hline
X' & x_1 & x_2 & x_3 & x_4 & \cdots & x_{m-1} \\
\hline
Y' & y_1 & y_2 & y_3 & y_4 & \cdots & y_{n-1} \\
\hline
\end{array}
\]

Can we express \( Z \) in terms of \( X' \) and \( Y' \) instead of \( X \) and \( Y \)?

Then \( Z = \text{LCS}(X', Y') + z_\ell \)

Consumed by being matched with \( y_n \)

Consumed by being matched with \( x_m \)
CONSIDER AN OPTIMAL SOLUTION Z

- By definition, $Z = \text{LCS}(X,Y)$
- Suppose $z_\ell$ matches only $x_m$ (so $x_m \neq y_n$)

Can we express $Z$ in terms of $X'$ and $Y'$ instead of $X$ and $Y$?

Then $Z = \text{LCS}(X, Y')$

Maybe still needed by $Z$

(Might be matched with something in $Y'$)

Not needed by $Z$

Remove to shrink problem size!
Consider an **optimal solution** Z

- By definition, \( Z = \text{LCS}(X,Y) \)
- Suppose \( z_\ell \) **matches only** \( y_n \) (so \( x_m \neq y_n \))

Then \( Z = \text{LCS}(X',Y) \)

**Not needed by Z**

**Maybe still needed by Z**

Can we express \( Z \) in terms of \( X' \) and \( Y' \) instead of \( X \) and \( Y \)?
CONSIDER AN **OPTIMAL SOLUTION** Z

- By definition, $Z = LCS(X, Y)$
- Suppose $z_{\ell}$ matches neither.

<table>
<thead>
<tr>
<th>$X$</th>
<th>$x_1$</th>
<th>$x_2$</th>
<th>$x_3$</th>
<th>$x_4$</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>$X'$</td>
<td>$x_1$</td>
<td>$x_2$</td>
<td>$x_3$</td>
<td>$x_4$</td>
<td>...</td>
</tr>
<tr>
<td>$Y$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$y_4$</td>
<td>$y_n$</td>
</tr>
<tr>
<td>$Y'$</td>
<td>$y_1$</td>
<td>$y_2$</td>
<td>$y_3$</td>
<td>$y_4$</td>
<td>...</td>
</tr>
</tbody>
</table>

**Note that** $x_m \neq y_n$, or else we could improve $Z$ by adding them!

Can we express $Z$ in terms of $X'$ and $Y'$ instead of $X$ and $Y$?

Take $Z = LCS(X', Y')$

*Not needed by Z*

*Not needed by Z*
FOUR CASES

- **Case** \( z_{\ell} \text{ matches both} \) (so \( x_m = y_n \)): \( Z = LCS(X', Y') + z_{\ell} \)
- **Case** \( z_{\ell} \text{ matches only} \) \( x_m \) (so \( x_m \neq y_n \)): \( Z = LCS(X, Y') \)
- **Case** \( z_{\ell} \text{ matches only} \) \( y_n \) (so \( x_m \neq y_n \)): \( Z = LCS(X', Y) \)
- **Case** \( z_{\ell} \text{ matches neither} \) (recall \( x_m \neq y_n \)): \( Z = LCS(X', Y') \)

- We don’t know \( z_{\ell} \)! How to identify case 1 vs 2-4? (If \( x_m = y_n \))
- How to differentiate between cases 2-4 without knowing \( z_{\ell} \)?

- Try all 3 possibilities in the recurrence and maximize length!

---

Let \( X_i = (x_1, \ldots, x_i) \), \( Y_j = (y_1, \ldots, y_j) \) and \( c[i, j] = |LCS(X_i, Y_j)| \)

In-class exercise: **derive the recurrence** for \( c[i, j] \) (part 1) and **give pseudocode** to solve the problem (part 2)
WE DID NOT GO PAST HERE IN CLASS

• But you are encouraged to attempt the exercise.
• If you want to teach yourself memorization (for the assignment), see slides 32-34.
IN-CLASS EXERCISE PART 1: DERIVE $c[i,j]$

- Case $z_\ell$ matches both (so $x_m = y_n$): $Z = LCS(X', Y') + z_\ell$
- Case $z_\ell$ matches only $x_m$ (so $x_m \neq y_n$): $Z = LCS(X, Y')$
- Case $z_\ell$ matches only $y_n$ (so $x_m \neq y_n$): $Z = LCS(X', Y)$
- Case $z_\ell$ matches neither (recall $x_m \neq y_n$): $Z = LCS(X', Y')$

Let $X_i = (x_1, ..., x_i), Y_j = (y_1, ..., y_j)$ and $c[i,j] = |LCS(X_i, Y_j)|$

$$c[i,j] = \begin{cases} ??? & \text{if } i = 0 \text{ or } j = 0 \\ ??? & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\ ??? & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j \end{cases}$$
IN-CLASS EXERCISE PART 1: DERIVE $c[i, j]$

- Case $z_\ell$ matches both (so $x_m = y_n$): $Z = LCS(X', Y') + z_\ell$
- Case $z_\ell$ matches only $x_m$ (so $x_m \neq y_n$): $Z = LCS(X, Y')$
- Case $z_\ell$ matches only $y_n$ (so $x_m \neq y_n$): $Z = LCS(X', Y)$
- Case $z_\ell$ matches neither (recall $x_m \neq y_n$): $Z = LCS(X', Y')$

Let $X_i = (x_1, ..., x_i)$, $Y_j = (y_1, ..., y_j)$ and $c[i, j] = |LCS(X_i, Y_j)|$

$\begin{align*}
c[i, j] &= \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
c[i - 1, j - 1] + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
\max\{c[i, j - 1], c[i - 1, j], c[i - 1, j - 1]\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j
\end{cases}
\end{align*}$

Can simplify! Observe that $c[i - 1, j - 1] \leq c[i, j - 1]$, because the former only has a subset of the input to the latter!

Therefore, it can't be the max.
IN-CLASS EXERCISE PART 1: DERIVE $c[i, j]$

- Case $z_\ell$ matches both (so $x_m = y_n$): $Z = LCS(X', Y') + z_\ell$
- Case $z_\ell$ matches only $x_m$ (so $x_m \neq y_n$): $Z = LCS(X, Y')$
- Case $z_\ell$ matches only $y_n$ (so $x_m \neq y_n$): $Z = LCS(X', Y)$
- Case $z_\ell$ matches neither (recall $x_m \neq y_n$): $Z = LCS(X', Y')$

Let $X_i = (x_1, \ldots, x_i)$, $Y_j = (y_1, \ldots, y_j)$ and $c[i, j] = |LCS(X_i, Y_j)|$

$$c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
c[i - 1, j - 1] + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
\max\{c[i, j - 1], c[i - 1, j]\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j 
\end{cases}$$
EXERCISE PART 2: PSEUDOCODE

Give pseudocode to compute $c[i,j]$ for all $i,j$ and return the length of $LCS(X,Y)$.

Algorithm: $LCS1(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

Remaining code:
Assume $c[]$ already exists.

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

Complexity?
Space? Time?

$\theta(nm)$ for both
COMPUTING THE LCS (NOT ITS LENGTH)

To make it easy to find the actual LCS (not just its length), we keep track of three possible cases that can arise in the recurrence relation:

UL $x_i = y_j$, include this symbol in the LCS, denote by $\swarrow$

L $x_i \neq y_j$, $c[i, j-1] > c[i-1, j]$, denote by $\leftarrow$

U $x_i \neq y_j$, $c[i, j-1] \leq c[i-1, j]$, denote by $\uparrow$

As a mnemonic aid, U denotes “up” and L denotes “left”.

$$c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
   c[i-1, j-1] + 1 & \text{if } i, j \geq 1 \text{ and } x_i = y_j \\
\max\{c[i, j-1], c[i-1, j]\} & \text{if } i, j \geq 1 \text{ and } x_i \neq y_j
\end{cases}$$
**Algorithm:** $LCS2(X = (x_1, \ldots, x_m), Y = (y_1, \ldots, y_n))$

1. for $i \leftarrow 0$ to $m$ do $c[i, 0] \leftarrow 0$
2. for $j \leftarrow 0$ to $n$ do $c[0, j] \leftarrow 0$
3. for $i \leftarrow 1$ to $m$
   - for $j \leftarrow 1$ to $n$
     - if $x_i = y_j$
       - then $c[i, j] \leftarrow c[i - 1, j - 1] + 1$
     - else if $c[i, j - 1] > c[i - 1, j]$
       - then $c[i, j] \leftarrow c[i, j - 1]$
     - else $c[i, j] \leftarrow c[i - 1, j]$

4. return $(c, \pi)$;

The table for the dynamic programming matrix $c[i,j]$ is:

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

+1 means $x_i$ is in the LCS!

If there are multiple possible sequences with the same length $|LCS(X,Y)|$ then $x_i$ is in some such sequence.
Suppose \( X = \text{gdvegta} \) and \( Y = \text{gvcekst} \).

How to obtain LCS=gvet from this table?
Algorithm: $\text{FindLCS}(c, \pi, v)$

1. $seq \leftarrow ()$
2. $i \leftarrow m$
3. $j \leftarrow n$
4. while $\min\{i, j\} > 0$
   a. if $\pi[i, j] = \text{UL}$
      i. $seq \leftarrow x_i \parallel seq$
      ii. $i \leftarrow i - 1$
      iii. $j \leftarrow j - 1$
   b. else if $\pi[i, j] = \text{L}$ then $j \leftarrow j - 1$
   c. else $i \leftarrow i - 1$

return $(seq)$

Complexity of this trace-back:
Space? Time?

Recall: $|X| = m$, $|Y| = n$

- space: $O(nm)$
- time: $O(n+m)$
MEMOIZATION: AN ALTERNATIVE TO DP

Recall that the goal of dynamic programming is to eliminate solving subproblems more than once.

Memoization is another way to accomplish the same goal.

Memoization is a recursive algorithm based on same recurrence relation as would be used by a dynamic programming algorithm.

The idea is to remember which subproblems have been solved; if the same subproblem is encountered more than once during the recursion, the solution will be looked up in a table rather than being re-calculated.

This is easy to do if initialize a table of all possible subproblems having the value undefined in every entry.

Whenever a subproblem is solved, the table entry is updated.
EXAMPLE: USING MEMORIZATION TO COMPUTE FIBONACCI NUMBERS EFFICIENTLY

main
  for $i \leftarrow 2$ to $n$
    do $M[i] \leftarrow -1$
  return $(RecFib(n))$

procedure $RecFib(n)$
  if $n = 0$ then $f \leftarrow 0$
  else if $n = 1$ then $f \leftarrow 1$
  else if $M[n] \neq -1$ then $f \leftarrow M[n]$
    $f_1 \leftarrow RecFib(n - 1)$
    $f_2 \leftarrow RecFib(n - 2)$
    $f \leftarrow f_1 + f_2$
    $M[n] \leftarrow f$
  else
    $f \leftarrow f_1 + f_2$
    $M[n] \leftarrow f$
  return $(f)$;
Algorithm: \( \text{BadFib}(n) \)

if \( n = 0 \) then \( f \leftarrow 0 \)
else if \( n = 1 \) then \( f \leftarrow 1 \)
else \(
\begin{align*}
  f_1 & \leftarrow \text{BadFib}(n - 1) \\
  f_2 & \leftarrow \text{BadFib}(n - 2) \\
  f & \leftarrow f_1 + f_2
\end{align*}
\)
return \( f \);

procedure \( \text{RecFib}(n) \)

if \( n = 0 \) then \( f \leftarrow 0 \)
else if \( n = 1 \) then \( f \leftarrow 1 \)
else if \( M[n] \neq -1 \) then \( f \leftarrow M[n] \)
else \(
\begin{align*}
  f_1 & \leftarrow \text{RecFib}(n - 1) \\
  f_2 & \leftarrow \text{RecFib}(n - 2) \\
  f & \leftarrow f_1 + f_2
\end{align*}
\)
return \( f \);

If \( M[n] \) is already computed, \textbf{don’t} recurse!

Memoization reduces this tree to a line with right-hanging leaves. 
\# recursive calls = \( O(n) \) instead of \( \sim 2^n \)

Calls \textbf{not} needed because of memoization
NEXT TIME

• More dynamic programming problems
  • Minimum length triangulation (maybe)
  • Optimal matrix multiplication order (maybe)
• Maybe: **big-picture** overview of the algorithmic design paradigms we’ve seen so far
  • Brute force, divide and conquer, dynamic programming, greedy
• Pros/cons of each? When to use each?