Dynamic Programming

Recall Fibonacci

**recursive**

\[
    f(n) = \begin{cases} 
        0 & \text{if } n = 0 \\
        1 & \text{if } n = 1 \\
        f(n-1) + f(n-2) & \text{else} 
    \end{cases}
\]

\[T(n) = T(n-1) + T(n-2) + \epsilon\]

so runtime grows like the Fibonacci numbers. BAD!

**iterative**

\[
    f(0) := 0 \\
    f(1) := 1 \\
    \text{for } i \text{ from 2 to } n \text{ do} \\
    \quad f(i) := f(i-1) + f(i-2) \\
    \text{od}
\]

\[O(n)\] arithmetic operations. GOOD!

- an example of dynamic programming

Main idea of dynamic programming:

solve “subproblems” from smaller to larger (bottom up) storing solutions

Runtime: \((\# \text{ subproblems}) \times (\text{time to solve one subproblem})\)
Text segmentation

Given a string of letters $A[1..n]$, $A[i] \in \{A, B, \ldots, Z\}$, can you split into words? Assume you have a test

$$\text{Word}[i, j] = \begin{cases} 
\text{True} & \text{if } A[i..j] \text{ is a word} \\
\text{False} & \text{otherwise}
\end{cases}$$

where each call takes $O(1)$

e.g., THEMEMEMPTY splits into THEM EM EMPTY

Note: a greedy solution might try to find

- the shortest word $A[1..i]$ (prefix): THE MEMEMPTY wrong
- or the longest word $A[1..i]$: THEME MPTY wrong

Can we do something like Fibonacci? Suppose we knew

$$\text{Split}[k] = \begin{cases} 
\text{True} & \text{if } A[1..k] \text{ is splittable} \\
\text{False} & \text{otherwise}
\end{cases} \text{ for } k = 0..n - 1$$

Can we then find $\text{Split}[n]$? Try $\text{Split}[j]$ and $\text{Word}[j + 1, n]$ for all $j = 0..n - 1$.

Claim: $\text{Split}[n]$ if and only if at least one $j$ gives True. Why?

$\Leftarrow$ we have a way to split $A[1..n]$

$\Rightarrow$ if $A[1..n]$ is splittable, take $A[j + 1..n]$ as last word
Resulting algorithm:

\[
\text{Split}[0] := \text{True} \\
\text{for } k \text{ from } 1 \text{ to } n \text{ do} \\
\quad \text{Split}[k] := \text{False} \\
\quad \text{for } j \text{ from } 0 \text{ to } k - 1 \text{ do} \\
\quad\quad \text{if Split}[j] \text{ and Word}[j + 1, k] \text{ then} \\
\quad\quad\quad \text{Split}[k] := \text{True} \\
\quad\quad \text{fi} \\
\quad \text{od} \\
\text{od}
\]

Runtime: $O(n^2)$

Ex. Show how to compute the actual split
Longest Increasing Subsequence

Given a sequence of numbers, $A[1..n]$, $A[i] \in \mathbb{N}$, find the longest increasing subsequence.

E.g., $5 \ 2 \ 1 \ 4 \ 3 \ 1 \ 6 \ 9 \ 2$  
Increasing subsequence of length 4

Following previous approach, what if we set

$LIS[k] = \text{length of longest increasing subsequence of } A[1..k]$?

This does not seem to give enough info to get $LIS[n]$ from previous $LIS[k]$'s.  
→ need to see if $A[n]$ is large enough to add to a previous sequence

Better Idea: Let $LISe[k] = \text{length of longest increasing subsequence of } A[1..k]$  
that ends with $A[k]$.

Algorithm

\[
\begin{align*}
LISe[1] & := 1 \\
\text{for } k \text{ from } 2 \text{ to } n \text{ do } \\
& \quad LISe[k] := 1 \\
& \quad \text{for } j \text{ from } 1 \text{ to } k - 1 \text{ do } \\
& \quad \quad \text{if } A[k] > A[j] \text{ then } \\
& \quad \quad \quad LISe[k] := \max\{LISe[k], LISe[j] + 1\} \\
& \quad \quad \text{fi} \\
& \quad \text{od} \\
& \text{od} \\
\end{align*}
\]

Ex. Argue correctness  
Runtime $O(n^2)$
Example

Run time: $O(n^2)$

How do we get the final answer?

- maximum entry in LIS$_e$

OR

- add dummy entry $A[n + 1] = +\infty$ and return $\text{LIS}e[n + 1] - 1$

Note: there is an $O(n \log n)$ time algorithm
Longest Common Subsequence

Recall pattern matching from CS 240: Given a long string $T$ and short pattern $P$ find occurrences of $P$ in $T$.

Useful in grep, find, etc.

Also useful: given two long strings find longest common subsequence

$\mathbf{x} = \text{T } \text{A } \text{R } \text{M } \text{A } \text{C}$

$\mathbf{y} = \text{C } \text{A } \text{T } \text{A } \text{M } \text{A } \text{R } \text{A } \text{N}$

Note that we can skip letters in both strings, but must preserve ordering.

Given strings $x_1 \ldots x_n$ and $y_1 \ldots y_m$,

Let $M(i, j) =$ length of longest common subsequence of $x_1 \cdots x_{i-1}x_i$ and $y_1 \cdots y_{j-1}y_j$.

How can we solve this subproblem based on solutions to “smaller” subproblems?

Choices: match $x_i = y_j$, skip $x_i$, skip $y_j$

$M(i, 0) = 0$

$M(0, j) = 0$

$M(i, j) = \max \begin{cases} 1 + M(i - 1, j - 1) & \text{if } x_i = y_j \\ M(i - 1, j) \\ M(i, j - 1) \end{cases}$

Solve subproblems in any order with $M(i - 1, j - 1)$, $M(i - 1, j)$, $M(i, j - 1)$ before $M(i, j)$
<table>
<thead>
<tr>
<th></th>
<th>y</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>9</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ø</td>
<td>CAT</td>
<td>TAM</td>
<td>MAR</td>
<td>AN</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>x</th>
<th>Ø</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
<th>0</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>T</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>Ø</td>
</tr>
<tr>
<td>2</td>
<td>A</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>R</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>M</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

for \(i = 0..n\): \(M(i, 0) := 0\)
for \(j = 0..m\): \(M(0, j) := 0\)
for \(i = 1..n\)
  for \(j = 1..m\)
    \[
    M(i, j) := \max \begin{cases} 
      1 + M(i - 1, j - 1) & \text{if } x_i = y_j \\
      M(i - 1, j) & \\
      M(i, j - 1) & 
    \end{cases}
    \]

Note that this is a correct ordering of \(i\) and \(j\).
In fact, if \(x_i = y_j\) we can use the first choice (no need to check max of other two choices).
Runtime: $O(n \cdot m \cdot c)$

To find the actual max. common subsequence: work backwards from $M(n, m)$.
→ Call OPT($n, m$).

OPT($i, j$) — recursive routine
if $i = 0$ or $j = 0$ then done fi
if $M(i, j) = M(i - 1, j)$ then
  OPT($i - 1, j$)
elif $M(i, j) = M(i, j - 1)$ then
  OPT($i, j - 1$)
else — we must have matched $i$ and $j$
  output $i, j$
  OPT($i - 1, j - 1$)
fi

Or we can record, when we fill $M(i, j)$, where the max comes from.

Next day: more sophisticated “edit” distance between strings.
Maximum common subsequence solves

Longest increasing subsequence

\[ L = 5 \ 2 \ 9 \ 6 \ 3 \ 7 \ 4 \]
\[ S = 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 9 \]

\[ S = \text{sort } L \]

Claim: Longest increasing subsequence of \( L = \) maximum common subsequence of \( L \) and \( S \).