Dynamic Programming

Recall Fibonacci

Recursive

\[ f(n) \]
if \( n = 0 \) return 0
if \( n = 1 \) return 1
else return \( f(n-1) + f(n-2) \)

\[ T(n) = T(n-1) + T(n-2) + c \]
so run time grows like the Fibonacci numbers

BAD!

Iterative

\[ f(0) = 0 \]
\[ f(1) = 1 \]
for \( i = 2 \ldots n \)
\[ f(i) = f(i-1) + f(i-2) \]

\( O(n) \) (assuming numbers are small enough)

GOOD!

- an example of dynamic programming.

Main idea of dynamic programming:

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

Run time:

\((\# \text{ subproblems}) \times (\text{time to solve one subproblem})\)
Text segmentation
Given a string of letters $A[1...n]$, $A[i] \in \{a,b,\ldots,z\}$
can you split it 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. THEMEMPTY splits into THEM EMPTY
Notes: a greedy solution might try to find
the shortest word $A[i..i]$ (prefix)
THE MEMPTY wrong
or the longest word $A[i..i]$
THEME MPTY wrong.
Can we do something like Fibonacci?
Suppose we knew

$$\text{Split }[k] = \begin{cases} \text{True} & \text{if } A[i..k] \text{ is splittable} \\ \text{False} & \text{otherwise} \end{cases}$$

for $k = 0..n-1$
Can we then find $\text{Split }[n]$?
Try $\text{Split }[j] \text{ AND Word } [j+1, n]$ for all $j = 0..n-1$
Claim $\text{Split }[n]$ iff at least one $j$ gives true
why? $\iff$ we have a way to split $A[i..n]$
$\implies$ if $A[i..n]$ is splittable,
take $A[j+1..n]$ as last word.
Resulting algorithm:

Split[0] ← True
for k = 1 to n
    Split[k] ← False
    for j = 0 to k - 1
        if Split[j] AND Word[j+1, k]
            then Split[k] ← True

Run time 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.

*Example:* $5 \ 2 \ 1 \ 4 \ 3 \ 1 \ 6 \ 9 \ 2$

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 — we need to see if $A[n]$ is large enough to add to a previous sequence.

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

Algorithm

$$LISE[1] = 1$$

for $k = 2 \ldots n$

$$LISE[k] = 1$$

for $j = 1 \ldots k-1$

if $A[k] > A[j]$ then

$$LISE[k] = \max \{LISE[k], LIS[j]+1\}$$

Example: Argue correctness

Runtime $O(n^2)$

Note: there is an $O(n \log n)$ time algorithm.
Example

\[
\begin{array}{cccccccccc}
1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 \\
5 & 2 & 1 & 4 & 3 & 1 & 6 & 9 & 2
\end{array}
\]

L1Se \begin{array}{ccccccc}
1 & 1 & 1 & 2 & 2 & 1 & 3
\end{array}
\begin{array}{ccc}
4 & 2
\end{array}

coming from \begin{array}{cccccccc}
0 & 0 & 0 & 2 & 2 & 0 & 4 & 7 & 3
\end{array}

Run Time \ O(n^2)

this array allows us to recover the seq.

How do we get the final answer?

- max entry in L1Se
- or add dummy entry \( A[n+1] = +\infty \)
- and return \( L1Se[n+1] - 1 \)

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

Recall pattern matching from CS 240

Given a long string $T$ and short string $P$
find occurrences of $P$ in $T$

Useful in grep, find, etc.

Also useful & given two long strings find
longest common subsequence

\[ \text{TARMAC} \quad \text{CATAMARAN} \]

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

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

Let \( M(i, j) = \text{length of longest common subsequence of } x_1 \ldots x_i \text{ and } y_1 \ldots y_j \)

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

**Choices:**

- match $x_i = y_j$
- skip $x_i$
- skip $y_j$

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

\[
M(1, 0) = 0, \quad M(0, 1) = 0
\]

Solve subproblems in any order with $M(i-1, j-1), M(i-1, j), M(i, j-1)$ before $M(i, j)$
Ex. Fill in the table.

Ex. Write out the pseudo-code

for \( i = 0 \cdots n \) \( M(i, 0) \leftarrow 0 \)
for \( j = 0 \cdots m \) \( M(0, j) \leftarrow 0 \)
for \( i = 1 \cdots n \)
    for \( j = 1 \cdots m \)
        \( M(i, j) = \max \left\{ \begin{array}{ll}
            1 + M(i-1, j-1) & \text{if } x_i = y_j \\
            M(i-1, j) & \\
            M(i, j-1) & 
        \end{array} \right. \)

Note that this is a correct ordering of \( i, j \)

In fact, if \( x_i = y_j \) we can use the first choice (no need to check max of other two choices)

Ex. prove this.
Run-time: $O(n \cdot m \cdot C)$

- # subproblems
- time to solve one subproblem (compare 3 possibilities)

To find the actual max. common subsequence:

work backwards from $M(m, n)$. Call $OPT(m, n)$.

$OPT(i, j)$ — recursive routine

if $M(i, j) = M(i-1, j)$ then $OPT(i-1, j)$
else if $M(i, j) = M(i, j-1)$ then $OPT(i, j-1)$
else -- we must have matched $i$ and $j$

output $i, j$

$OPT(i-1, j-1)$

or we can record, when we fill $M(i, j)$,

where the max came from

Next day: more sophisticated “edit” distance between strings.
Max common subsequence solves
longest increasing subsequence

$L = 5 \ (2) \ 9 \ 6 \ (3) \ (7) \ 4$
$S = 2 \ 3 \ 4 \ 5 \ 6 \ 7 \ 9$

increasing subsequence of length 3.

$S = \text{sort}\ L$.

Claim: longest increasing subsequence of $L$

$= \text{max common subsequence of } L \text{ and } S$