Dynamic Programming

Recall the maximum common subsequence problem from last day.

```
TARMAC
CATAMARAN
```

More sophisticated: count # changes from 1st string to 2nd.

**E.g.:**
- You: Pythagoras
- Google: Pythagoros
- a change is:
  - add a letter
  - delete a letter
  - replace a letter — mismatch
- This is called *edit distance*.

This problem comes up in bioinformatics for DNA strings.

DNA is a sequence of chromosomes, i.e. string over A, C, T, G.

Two strings can be aligned in different ways.

**E.g.:**
```
<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>G</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

- 3 changes
  - delete C
  - change T to A
  - add G

```
<table>
<thead>
<tr>
<th>A</th>
<th>A</th>
<th>C</th>
<th>A</th>
<th>T</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>A</td>
<td>A</td>
<td>G</td>
<td></td>
</tr>
</tbody>
</table>
```

- 2 changes
Problem: Given 2 strings $x_1 \ldots x_m$ and $y_1 \ldots y_n$ compute their edit distance.

i.e. find the alignment that gives min # changes.

Dynamic Programming Algorithm

Subproblem for $x_i \ldots x_i$ and $y_j \ldots y_j$

$M(i,j) = \min \text{ # changes from } x_i \text{ to } y_j$

choices:
- match $x_i$ to $y_j$, pay replacement cost if they differ
- match $x_i$ to blank (delete $x_i$)
- match $y_j$ to blank (add $y_j$)

$M(i,j) = \min \begin{cases} M(i-1,j-1) & \text{if } x_i = y_j \\
 r + M(i-1,j-1) & \text{if } x_i \neq y_j \\
 d + M(i-1,j) & \text{match } x_i \text{ to blank} \\
 a + M(i,j-1) & \text{match } y_j \text{ to blank} \end{cases}$

where $r =$ replacement cost
$d =$ delete cost
$a =$ add cost

So far, we used $r = d = a = 1$

More sophisticated: $r(x_i, y_j)$ - replacement cost depends on the letters

e.g. $r(A,S) = 1$ because these keys are close on typewriter

$r(A,C) = 2$ --- not so close
In what order do we solve subproblems?
Same as last day.

\[
\begin{bmatrix}
1 & \cdots & j \\
\vdots & \ddots & \vdots \\
0 & \cdots & 0
\end{bmatrix}
\text{need these 3 subproblems}
\]

\[
M = \left[ \begin{array}{ccc}
0 & \cdots & m \\
0 & \cdots & n
\end{array} \right]
\]
for \( i = 0 \ldots m \) \( M(i, 0) = i \cdot d \) - delete \( i \) letters
for \( j = 0 \ldots n \) \( M(0, j) = j \cdot a \) - add \( j \) letters
for \( i = 1 \ldots m \)
\[
\text{for } j = 1 \ldots n
\]
\[
M(i, j) = \ldots
\]

\[
\text{fill matrix in order } \left[ \begin{array}{c}
\vdots \\
\vdots \\
\vdots \\
\end{array} \right]
\]
(or could do columns first)

Analysis \( O(n \cdot m) \) time \( (n \cdot m \text{ subproblems, constant time each}) \)
\( O(n \cdot m) \) space

A different application \([\text{OPTIONAL}]\)
Music pattern matching

\[
\text{match this } \Rightarrow \#
\begin{array}{ccccccc}
\cdot & \cdot & \cdot & \cdot & \cdot & \cdot & \cdot
\end{array}
\]

\[
\text{to this } \Rightarrow d \quad p \quad \ell \quad \ell \quad \ell \quad \ell
\]

Use replacement rules that allow \( d \Rightarrow \ddots \)
Coin changing problem

Recall: given coin denominations
\[ c_1, c_2, \ldots, c_k \] (e.g., penny, nickel, etc)
and a value \( V \)
make change for \( V \) using min. # coins.

Greedy does not always work.

A dynamic programming algorithm:

Given \( V \), we might try each coin \( c_i \leq V \)
Then must make change for \( V - c_i \)

Subproblems \( C(u) \) for each \( u = 0 \cdots V \)

\[ C(0) = 0 \]

for \( u = 1 \cdots V \)

\[ C(u) = \min_{c_i \leq V} \frac{1}{1 + C(u - c_i)} : i = 1 \cdots k \]

end

in more detail:

\[ C(u) \leftarrow \infty \]

for \( i = 1 \cdots k \)

if \( c_i \leq u \) and \( (1 + C(u - c_i)) < C(u) \)

end then \( C(u) \leftarrow 1 + C(u - c_i) \)

Run Time \( O(V \cdot k) \)

# subproblems \( \cdot \) time for each
An algorithm runs in polynomial time if run-time is $O(n^c)$, $c$ a constant on input of size $n$.

The above algorithm is NOT polynomial time because size of $V$ is $\log V$ but run-time depends on $V$ (not $\log V$).

This is called a pseudo-polynomial time algorithm.

More on these ideas later in the course.
Constructing optimum binary search trees

given items 1 \ldots n
probabilities p_1 \ldots p_n
construct a binary search tree (items in leaves)
to minimize search cost \sum_i p_i \text{depth}(i)

eg. p_1 = \ldots = p_4 = \frac{1}{4}

\text{search cost} = 4 \cdot \frac{1}{4} \cdot 3 = 3

p_1 = .7 p_2 = p_3 = p_4 = .1

(\cdot 7) 3 + 3 (\cdot 1) 3
= 2.1 + .9
= 3

\begin{align*}
(\cdot 7) 2 + (\cdot 1) 3 + 2 (\cdot 1) 4 \\
= 1.4 + .3 + .8 \\
= 2.5
\end{align*}

[In case you've seen optimum Huffman trees, this is different in that leaf ordering is fixed.]

To apply dynamic programming:

subproblems: opt. binary search tree for items i \cdots j

order subproblems by # items, i.e. by j - i
to solve i \cdots j

Try all choices for k

\begin{align*}
&i \cdots k \\
&j - k + 1 \cdots j
\end{align*}
Details

\[ M[i,j] = \min_{k=i\ldots j-1} \left( \sum_{t=i}^{j} M[i,k] + M[k+1,j] \right) + \sum_{t=i}^{j} P_t \]

\[ \text{because every node gets } 1 \text{ deeper} \]

\[ \text{ind. of choice of } k \]

How to compute \( \sum_{t=i}^{j} P_t \)

First compute \( P[i] = \sum_{j=1}^{i} P_j \)

then we can get \( \sum_{t=i}^{j} P_t \) as \( P[j] - P[i-1] \)

for \( i=1\ldots n \)

\[ M[i,i] \leftarrow P_i \]

for \( r=1\ldots n-1 \)

\[ \text{for } i=1\ldots n-r \]

/* solve for \( M[i,i+r] \)

\[ \text{best} \leftarrow M[i,i] + M[i+1,i+r] \]

\[ \text{for } k=i+1\ldots i+r-1 \]

\[ \text{temp} \leftarrow M[i,k] + M[k+1,i+r] \]

\[ \text{if } \text{temp} < \text{best} \text{ then } \text{best} \leftarrow \text{temp} \]

end

\[ M[i,i+r] \leftarrow \text{best} + P[i+r] - P[i-1] \]

end

end \# \text{subproblems}

Run time \( O(n^2 \cdot n) = O(n^3) \) time per subproblem