Dynamic Programming II

Recall the maximum common subsequence problem from last day:

T A R M A C
C A T A M A R A N

More sophisticated: count # changes

e.g., You: Pythagorus
Google: Pythagoras?

You: recurrence
Google: recurrence?

A change is:
- add a letter \textit{gap}
- delete a letter
- replace a letter \textit{mismatch}

The problem comes up in bioinformatics for DNA strings. DNA is a sequence of chromosomes, i.e., a string over the alphabet \texttt{A, C, T, G}.

This is called \textit{edit distance}.

Two string can be aligned in different ways:

e.g. A A C A T
     A A A A G
     3 changes
     (2 gaps, 1 mismatch)

e.g. A A C A T
     A A A A G
     2 changes
     (2 mismatches)
Problem: Given 2 strings $x_1..x_m$ and $y_1..y_n$, compute their edit distance.
I.e., find the alignment that gives the minimum number of changes.

Dynamic Programming Algorithm

Subproblem: $M(i, j) =$ minimum number of changes to match $x_1..x_{i-1}x_i$ and $y_1..y_{j-1}y_j$.

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

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

where:

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

So far, we used $r = d = a = 1$ (i.e., count # changes).
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
  $r(a, c) = 2 \ldots$ not too close
In what order do we solve subproblems? Same as last day.

\[ M[0..m, 0..n] \]
for \( i = 0..m \): \( M(i, 0) = \text{id} \)  
for \( j = 0..n \): \( M(0, j) = ja \)  
for \( i = 1..m \)
for \( j = 1..n \)
\[ M(i, j) = \ldots \]

\{ fill matrix in order \[ \rightarrow \] \}
(or could do columns first)

Analysis: \( O(nm) \) time and \( O(nm) \) space

\( (nm \) subproblems, constant time each)
Recall Interval Scheduling aka Activity Selection: Given a set of intervals $I$, find a maximum size subset of disjoint intervals:

**Weighted Interval Scheduling**

Weighted Interval Scheduling: Given $I$ and weight $w(i)$ for each $i \in I$, find set $S \subseteq I$ such that no two intervals overlap and maximize $\sum_{i \in S} w(i)$.

e.g., you have preferences for certain activities.

A more general problem:

- $I$ is a set of element ("items")
- $w(i) =$ weight of item $i$
- some pairs $(i, j)$ conflict

Find a maximum weight subset $S \subseteq I$ with no conflicting pairs.

Can be modeled as a graph: vertex = item edge = conflict

Problem is Max Weight Independent Set and we will see later that it is NP-complete.
A general approach to finding max weight independent set.
Consider one item $i$. Either we choose it or not.

$$\text{OPT}(I) = \max\{\text{OPT}(I - \{i\}), w(i) + \text{OPT}(I')\} \quad \text{where} \quad I' = \text{intervals disjoint from } i$$

In general this recursive solution does not give polynomial time.

$$T(n) = 2T(n - 1) + O(1) \implies T(n) \in \Theta(2^n)$$

Essentially, we may end up solving subproblems for each of the $2^n$ subsets of $I$.

When $I =$ set of intervals, we can do better with dynamic programming.

Order intervals $1..n$ by right endpoint

something nice happens

<table>
<thead>
<tr>
<th>$i$</th>
<th>$i-1$</th>
<th>Intervals disjoint from interval $i$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$j$</td>
<td>$j-1$</td>
<td>$j$ for some $j$</td>
</tr>
</tbody>
</table>

For each $i$, let $p(i) =$ largest index $j < i$ s.t. interval $j$ is disjoint from interval $i$.

Let $M(i) =$ max weight subset of intervals $1..i$

$$M(i) = \max\{M(i - 1), w(i) + M(p(i))\}$$
A Dynamic Programming algorithm – computes the actual set, not just weight

Sort intervals 1..n by right endpoint.

\[ M(0) := 0; \quad S(0) := \emptyset \quad S \text{ stores the set} \]

\[ \text{for } i \text{ from 1 to } n \text{ do} \]

\[ p(i) := i - 1 \]

\[ \text{while } p(i) \neq 0 \text{ and intervals } i \text{ and } p(i) \text{ overlap do} \]

\[ p(i) := p(i) - 1 \]

\[ \text{od} \]

\[ \text{if } M(i) \geq w(i) + M(p(i)) \text{ then} \]

\[ M(i) := M(i - 1); \quad S(i - 1) := S(i - 1) \]

\[ \text{else} \]

\[ M(i) := w(i) + M(p(i)); \quad S(i) := \{i\} \cup S(p(i)) \]

\[ \text{fi} \]

\[ \text{od} \]

---

**Final answer:** weight \( M(n) \), set \( S(n) \)  
**Time:** \( n \) subproblems, each \( O(n) \)  
so total of \( O(n^2) + O(n \log n) \) to sort.  
**Space:** \( O(n^2) \) - storing \( n \) sets, each \( O(n) \)

---

**Next:**

1. computing all \( p(i) \) values before-hand to save time
2. computing \( S \) by backtracking to save space
How to compute $p(i)$: We use sorted order $1..n$ by right endpoint
AND sorted order $\ell_1..\ell_n$ by left endpoint

\[
\begin{align*}
j &:= n \\
\text{for } k \text{ from } n \text{ down to } 1 \text{ do} \\
\quad \text{while } \ell_k \text{ overlaps } j \text{ do} \\
\quad \\
\quad \quad j &:= j - 1 \\
\quad \text{od} \\
\quad p(\ell_k) &:= j \\
\text{od}
\end{align*}
\]

Run time $\Theta(n)$ after sorting

Final algorithm:

Sort intervals $1..n$ by right endpoint.
Sort intervals by left endpoint.
Compute $p(i)$ for all $i$.
$M(0) := 0$

for $i$ from 1 to $n$ do

\[
M(i) := \max\{M(i - 1), w(i) + M(p(i))\}
\]

od

Runtime: $O(n \log n) + O(n) + O(n \cdot c)$

\[\text{time per subproblem} \quad \# \text{subproblems}\]
Backtracking to compute $S$: Use recursive routine to $S$-OPT

```
S-OPT(i)
    if $i = 0$ then
        return $\emptyset$
    elif $M(i - 1) \geq w(i) + M(p(i))$ then
        return $S$-OPT($i - 1$)
    else
        return $\{i\} \cup S$-OPT($p(i)$)
fi
```

The set we want is $S$-OPT($n$).

Time: $O(n)$

Space: $O(n)$

Summary

- A general idea to find an optimal subset is to solve subproblems where one element is in or out
- Exponential in general; can sometimes be efficient
- Key ideas of dynamic programming:
  - Identify subproblems (not too many) together with
  - an order of solving them such that each subproblem can be solved by combining a few previously solved subproblems.