Dynamic Programming II

Recall the maximum common subsequence problem from last day:

TAR MAC
CATAMARAN

More sophisticated: count # changes

E.g., You: Pythagoras
Google: Pythagoras?
    ▼ Change

E.g., You: recurrence
Google: recurrence?
    ▼ 2 changes

E.g., You: Pythagoras
Google: Pythagoras?
    ▼ 1 change

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.,

\[
\begin{align*}
A & A C A T \quad \text{\$} \\
A & A \quad \text{\$} A A G
\end{align*}
\]

3 changes (2 gaps, 1 mismatch)

\[
\begin{align*}
A & A C A T \\
A & A A A A G
\end{align*}
\]

2 changes (2 mismatches)
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 \( x_i \) and \( y_j \):

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

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 \( y_j \))

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

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 too close
In what order do we solve subproblems?
Same as last day.

\[
\begin{bmatrix}
0 & \cdots & 0 & \cdots & n \\
\end{bmatrix}
\]

Matrix \( M \) needs these 3 subproblems

\[
M[i, 0] = i \cdot d \\
M(0, j) = j \cdot a \\
M[i, j] = 0 \\
\]

Analysis: \( O(n \cdot m) \) time, \( O(n \cdot m) \) space

A different application
Music pattern matching

Match this \# \# \# p \# p

to this \# d \# p \# p \# p

Use replacement rules that allow \( d \rightarrow \# \# \)
Weighted Interval Scheduling

Recall Interval Scheduling aka Activity Selection:
Given a set of intervals $I$, find a max size subset of disjoint intervals.

Weighted Interval Scheduling - Given $I$ and weights $w(i)$ for each $i \in I$, find set $S \subseteq I$ s.t. no two intervals in $S$ 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 elements ("items")
$w(i) =$ weight of item $i$

Some pairs $(i, j)$ conflict
Find a max 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
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.

\[ \text{OPT}(I) = \max \left\{ \text{OPT}(I - \{i\}), w(i) + \text{OPT}(I') \right\} \]

\( I' = \text{intervals disjoint from } i \)

In general this recursive solution does not give poly time.

\[ T(n) = 2T(n-1) + O(1) \quad \text{— exponential.} \]

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

When \( I = \text{set of intervals}, \) we can do better with dynamic programming.

Order intervals 1..n by right endpoint something nice happens

\[ \text{Intervals disjoint from interval } i \]

\[ \text{are 1..j for some } j \]

For each \( i, \) let \( p(i) = \text{largest index } j < i \)

such interval \( j \) is disjoint from interval \( i \)

\[ p(i) = j \]

Now we can solve subproblems

let \( M(i) = \text{max weight subset of intervals 1..i} \)

Then \( M(i) = \max \left\{ M(i-1), w(i) + M(p(i)) \right\} \)
A DP alg. computes the actual set, not just weight

sort intervals 1..n by right endpoint

\( M(0) = 0 \); \( S(0) = \emptyset \)

\( S \) stores the set

for \( i = 1 \ldots n \)

\[ p(i) \leftarrow i-1 \]

\( \text{compute} \begin{cases} \\
\text{while } p(i) \neq 0 \text{ and intervals } i \text{ and } p(i) \text{ overlap} \\
p(i) \leftarrow p(i) - 1 \\
\text{if } M(i-1) \geq w(i) + M(p(i)) \text{ then} \\
M(i) \leftarrow M(i-1) \\
S(i) \leftarrow S(i-1) \\
\text{else} \\
M(i) \leftarrow w(i) + M(p(i)) \\
S(i) \leftarrow S(i) \cup S(p(i)) \\
\end{cases} \]

end

\[ \text{final answer: } \frac{M(n)}{\text{weight}} \quad \frac{S(n)}{\text{set}} \]

n subproblems, each \( O(n) \)

so total of \( O(n^2) + O(n \log n) \) sort.

space \( O(n^2) \) - storing \( \ln 1 \) 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 $1, \ldots, n$ by right endpoint

AND sorted order $l_1, \ldots, l_n$ by left endpoint

\[
\begin{align*}
  &j \leq n \\
  &\text{for } k = n \ldots 1 \\
  &\text{while } l_k \text{ overlaps } j \text{ do } j \leftarrow j - 1 \\
  &\quad p(l_k) \leftarrow j \\
  &\text{end}
\end{align*}
\]

Run-time $\Theta(n)$ after sorting

Final algorithm

- sort intervals $1, \ldots, n$ by right endpoint
- sort intervals by left endpoint
- compute $p(i)$ for all $i$

\[
M(0) = 0
\]

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

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

end

Run-time $O(n \log n) + \Theta(n) + O(n \cdot c)$

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

$S$-OPT($i$)

if $i = 0$ return $\emptyset$
else if $M(i-1) \geq w(i) + M(p(i))$
    return $S$-OPT($i-1$)
else return $S$-OPT($p(i)$)

end

the set we want is $S$-OPT($n$)
Time $O(n)$  Space $O(n)$

Summary

• a general idea to find opt. subset: 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) and an order of solving them s.t. each subproblem can be solved by combining a few previously solved subproblems