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` You : `recur ance`

Google : `Pythagoras` ? Google : `recurrence` ?

A change is:
- add a letter
- delete a letter
- replace a letter

This is called edit distance.

The problem comes up in bioinformatics for DNA strings. DNA is a sequence of chromosones, i.e., a string over the alphabet **A, C, T, G**.

Two strings 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 two strings $x_1 \dots x_m$ and $y_1 \dots y_n$, 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 $x_1 \dots x_{i-1}x_i$ and $y_1 \dots y_{j-1}y_j$.

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)

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

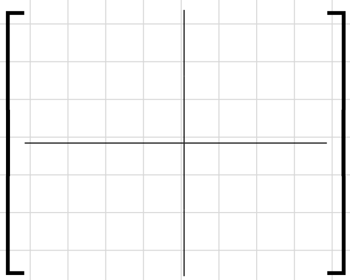
where:
$r =$ replacement cost
$d =$ delete cost
$a =$ add cost

So far, we used $r = d = a = 1$ (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
$r(a,c) = 2$ ... not too close

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

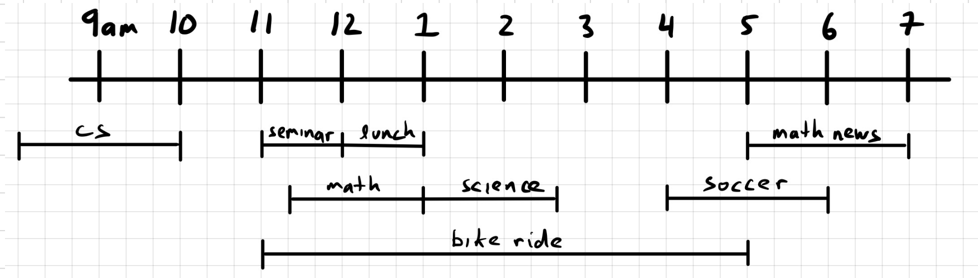
$M[0 \ldots m, 0 \ldots n]$
for i **from** 0 **to** m **do** $M(i,0) = id$
for j **from** 0 **to** n **do** $M(0,j) = ja$
for i **from** 1 **to** m **do**
    for j **from** 1 **to** n **do**
        $M(i,j) :=$ ...



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:



<u>Weighted Interval Scheduling</u>

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) <u>conflict</u>

Find a maximum weight subset $S \subset I$ 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.

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

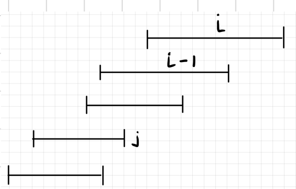
In general this recursive solution does not give polynomial time.

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

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

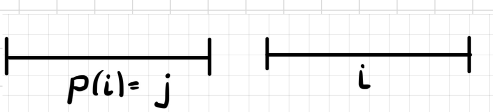
When $I =$ set of intervals, we can do better with dynamic programming.

Order intervals $1 \ldots n$ by right endpoint.
something nice happens



Intervals disjoint from interval i are $1 \ldots j$ for some j.

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



Now we can solve subproblems.

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

$$M(i) = \max\{M(i-1), w(i) + M(p(i))\}$$

A Dynamic Programming algorithm – computes the actual set, not just weight

Sort intervals $1 \dots n$ by right endpoint.
$M(0) := 0$
$S(0) := \emptyset$
for i **from** 1 **to** n **do**
  $p(i) := i - 1$
  while $p(i) \neq 0$ **and** intervals i and $p(i)$ overlap **do** $p(i) := p(i) - 1$
  if $M(i-1) \geq w(i) + M(p(i))$ **then**
    $M(i) := M(i-1)$
    $S(i) := S(i-1)$
  else
    $M(i) := w(i) + M(p(i))$
    $S(i) := \{i\} \cup S(p(i))$

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.

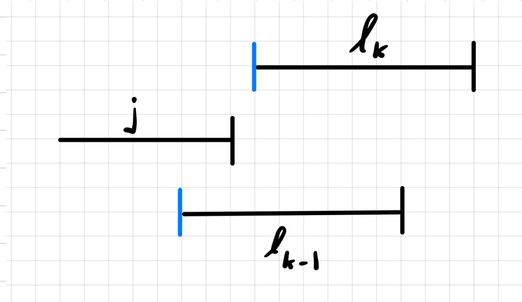
Space: $O(n^2)$ - storing n sets, each $O(n)$

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 $1 \ldots n$ by right endpoint
and sorted order $\ell_1 \ldots \ell_n$ by left endpoint

$j := n$
for k **from** n **downto** 1 **do**
  while ℓ_k overlaps j **do** $j := j - 1$
  $p(\ell_k) := j$



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

Run-time: $\underbrace{O(n \log n)}_{\text{sort}} + \underbrace{O(n)}_{p(*)} + O(n \cdot c)$

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($p(i)$)

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.