6.006	Lecture 21	Dec. 1, 2011

TODAY: Dynamic Programming III (of 4)
- subproblems for strings
- parenthesization
- edit distance (& longest common subseq.)
- knapsack
- pseudopolynomial time

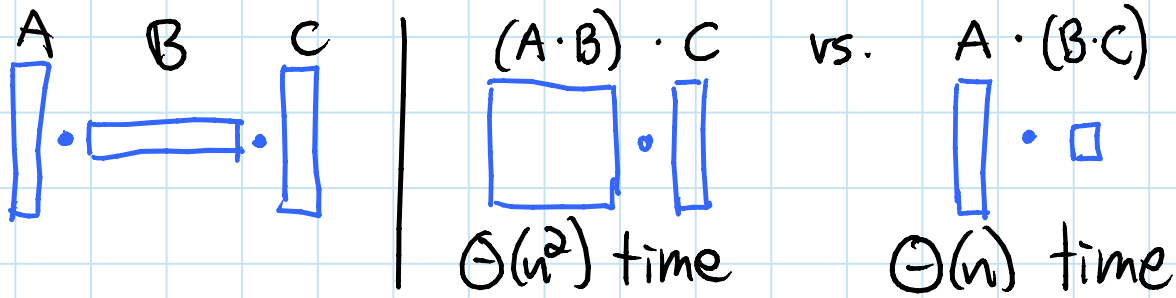
* 5 easy steps to dynamic programming:
  1. define subproblems — count # subprobs.
  2. guess (part of solution) — count # choices
  3. relate subprob. solutions — compute time/subprob.
  4. recurse + memoize — time = time/subprob. · #subprobs.
     OR build DP table bottom-up
     - check subprobs. acyclic/topological order
  5. solve original problem: = a subproblem
     OR by combining subprob. solutions (⇒ extra time)

- problems from L20 (text justification, Blackjack) are on sequences (words, cards)

* useful subproblems for strings/sequences x:
  - suffixes x[i:] } Θ(|x|) ← cheaper ⇒ use if possible
  - prefixes x[:i] } Θ(|x|)
  - substrings x[i:j] } Θ(|x|²)

L20

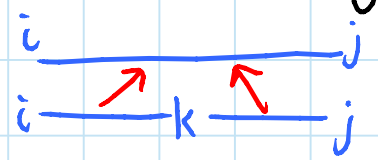
Parenthesization: optimal evaluation of an associative expression $A[0] \cdot A[1] \cdot \cdots \cdot A[n-1]$
- e.g. multiplying rectangular matrices



② guessing = outermost multiplication: $(\cdots)(\cdots)$ (k−1, k)
⇒ # choices = $O(n)$

① subproblems = ~~prefixes & suffixes?~~ NO
= cost of substring $A[i:j]$
⇒ # subproblems = $\Theta(n^2)$

③ recurrence:
- $DP[i,j] = \min($ $DP[i,k] + DP[k,j] +$ cost of $(A[i] \cdots A[k-1]) \cdot (A[k] \cdots A[j-1])$ for k in range$(i+1, j)$ $)$

DAG: 

- $DP[i, i+1] = \emptyset$

⇒ cost per subproblem = $\Theta(j-i) = O(n)$

④ topological order: increasing substring size
- total time = $\Theta(n^3)$

⑤ original problem = $DP[0, n]$
(& use parent pointers to recover parens.)

NOTE: Above DP is not shortest paths in the subproblem DAG! Two dependencies ⇒ not path!

Edit distance: (used for DNA comparison, diff, CVS/SVN/..., spellchecking (typos), plagiarism detection, etc.)

given two strings x & y, what's the cheapest possible sequence of character edits to transform x into y?

insert c | delete c | replace c→c'

- cost of edit depends only on characters c, c'
- e.g. in DNA, C→G common mutation ⇒ low cost
- cost of sequence = sum of costs of edits

- if insert & delete cost 1, replace costs ∞, min. edit distance equivalent to finding longest common subsequence

  sequential but not necessarily contiguous

- e.g.: HIEROGLYPHOLOGY vs. MICHAELANGELO → HELLO

Subproblems for multiple strings/sequences: combine suffix/prefix/substring subproblems

- multiply state spaces
- still polynomial for O(1) strings

Edit distance DP:

① subproblems: $c(i,j) = \text{edit-distance}(x[i:], y[j:])$
for $0 \le i < |x|$, $0 \le j < |y|$
$\Rightarrow \Theta(|x| \cdot |y|)$ subproblems

② guess whether, to turn x into y,
- $x[i]$ deleted
- $y[j]$ inserted
- $x[i]$ replaced by $y[j]$

} 3 choices

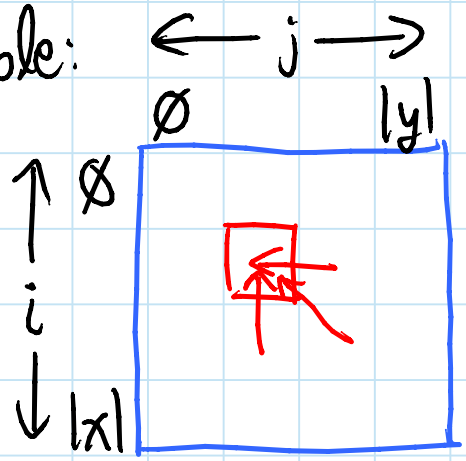
③ recurrence: $c(i,j) = \max\{$
$\text{cost}(\text{delete } x[i]) + c(i+1, j)$ if $i < |x|$,
$\text{cost}(\text{insert } y[j]) + c(i, j+1)$ if $j < |y|$,
$\text{cost}(\text{replace } x[i] \to y[j]) + c(i+1, j+1)$
if $i < |x|$ & $j < |y|\}$

- base case: $c(|x|, |y|) = \emptyset$

$\Rightarrow \Theta(1)$ time per subproblem

④ topological order: DAG in 2D table:
- bottom up OR right to left
- only need to keep last 2 rows/columns

$\Rightarrow$ linear space
- total time $= \Theta(|x| \cdot |y|)$



⑤ original problem: $c(\emptyset, \emptyset)$

Knapsack

of size S you want to pack
- item i has integer size s_i & real value v_i
- goal: choose subset of items of max. total value subject to total size $\leq S$

First attempt:

1. ~~subproblem = value for suffix i:~~ WRONG
2. guessing = whether to include item i
   $\Rightarrow$ #choices = 2
3. recurrence:
   - $DP[i] = \max(DP[i+1], v_i + DP[i+1]$ if $s_i \leq S$?!)
   - not enough information to know whether item i fits — how much space is left? GUESS!

Correct:

1. subproblem = value for suffix i: given knapsack of size X
   $\Rightarrow$ # subproblems $= O(nS)$ (!)
3. recurrence:
   - $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X-s_i]$ if $s_i \leq X)$
   - $DP[n, X] = \emptyset$
   $\Rightarrow$ time per subproblem $= O(1)$
4. topological order: for i in $n, \ldots, \emptyset$: for X in $0, \ldots, S$
   - total time $= O(nS)$
5. original problem $= DP[\emptyset, S]$
   (& use parent pointers to recover subset)

AMAZING: effectively trying all possible subsets!
... but is this actually fast?

Polynomial time = polynomial in input size
- here $\Theta(n)$ if number S fits in a word
  $O(n \lg S)$ in general
- S is exponential in $\lg S$ (not polynomial)

Pseudopolynomial time = polynomial in the problem size AND the numbers in input
here: S, s_i's, v_i's
- $\Theta(nS)$ is pseudopolynomial

$\Longrightarrow$

Remember: polynomial – GOOD
exponential – BAD
pseudopoly. – SO SO

6.006 Introduction to Algorithms
Fall 2011