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
  2. guess (part of solution)
  3. relate subprob. solutions
  4. recurse & memoize
  5. build DP table bottom-up
    - check subprobs. acyclic/topological order
  or 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 $x[i:]$
  - prefixes $x[:i]$
  - substrings $x[i:j]$

L20

\[ \Theta(1x) \text{ cheaper} \Rightarrow \text{use if possible} \]
Parenthesization: optimal evaluation of an associative expression \(A[\emptyset] \cdot A[1] \cdot \ldots \cdot A[n-1]\)
- e.g. multiplying rectangular matrices

\[
\begin{array}{c|c|c}
A & B & C \\
\cdot & \cdot & \cdot \\
\hline
\end{array} \quad \begin{array}{c|c|c}
(A \cdot B) \cdot C & \text{vs.} & A \cdot (B \cdot C) \\
\theta(n^2) \text{ time} & \theta(n) \text{ time} \\
\end{array}
\]

\(\theta(n^2)\) time

2. guessing = outermost multiplication: \((\ldots)(\ldots)\)
\[\Rightarrow \text{# choices} = O(n)\]

1. subproblems = prefixes & suffixes? \(\text{NO}\)
\[= \text{cost of subtring} \ A[i:j]\]
\[\Rightarrow \text{# subproblems} = \Theta(n^2)\]

3. recurrence:
\[\text{DP}[i, j] = \min \left( \text{DP}[i, k] + \text{DP}[k, j] + \text{cost of} \right) \]
\[\quad \left( (A[i] \ldots A[k-1]) \cdot (A[k] \ldots A[j-1]) \right) \]
\[\text{for } k \text{ in range}(i+1, j)\]
\[\text{DP}[i, i+1] = \emptyset \]
\[\Rightarrow \text{cost per subproblem} = \Theta(j-i) = O(n)\]

4. topological order: increasing substring size
\[\Rightarrow \text{total time} = \Theta(n^3)\]

5. original problem = \text{DP}[\emptyset, n]
\[(& \text{use parent pointers to recover parens.})\]
Note: Above DP is not shortest paths in the Subproblem DAG! Two dependencies \( \Rightarrow \) not path.

Edit distance: (used for DNA comparison, diff, CVS/SVN\ldots, 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 \( c \rightarrow c' \)

Cost of edit depends only on characters \( c, c' \)
- E.g. in DNA, \( C \rightarrow G \) common mutation \( \Rightarrow \) low cost
- Cost of sequence = sum of costs of edits (0 if match)

- If insert & delete cost 1, replace costs \( \infty \)
- Min edit distance equivalent to finding longest common subsequence
  - Sequential but not necessarily contiguous
- E.g.: Hieroglyphology \( \Rightarrow \) Hello vs. Michaelangelo

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 \( 0 \leq i < |x|, 0 \leq j < |y| \)
   \( \Rightarrow \Theta(|x| \cdot |y|) \) subproblems

2. **guess whether to turn \( x \) into \( y \):**
   - \( x[i] \) deleted
   - \( y[j] \) inserted
   - \( x[i] \) replaced by \( y[j] \)
   \( \Rightarrow 3 \) choices
   - \( x[i] = y[j] \)
   - \( \text{free if } x[i] \neq y[j] \)

3. **recurrence:**
   \[ c(i, j) = \min \{ \]
   \[ \quad \text{cost (delete } x[i]) + c(i+1, j) \text{ if } i < |x|, \]
   \[ \quad \text{cost (insert } y[j]) + c(i, j+1) \text{ if } j < |y|, \]
   \[ \quad \text{cost (replace } x[i] \rightarrow y[j]) + c(i+1, j+1) \text{ if } \]
   \[ i < |x| \text{ & } j < |y| \}
   \[ \Rightarrow \Theta(1) \text{ time per subproblem} \]

4. **topological order:**
   - DAG in 2D table:
     - bottom up or right to left
     - only need to keep 2 rows/columns (last & current)
   \( \Rightarrow \) 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:
   - $DP[i] = \max(DP[i+1], v_i + DP[i+1] \text{ if } s_i \leq S?!)$
   - not enough information to know whether item $i$ fits — how much space is left?

Correct:
1. subproblem = value for suffix $i$:
   given knapsack of size $X$
   $\Rightarrow$ #subproblems = $O(nS)$ (!)
3. recurrence:
   - $DP[i,X] = \max(DP[i+1,X], v_i + DP[i+1,X-s_i] \text{ if } s_i \leq X)$
   - $DP[n,X] = \emptyset$
   $\Rightarrow$ time per subproblem = $O(1)$
4. topological order: for $i$ in $n, \ldots, 0$: 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 $O(n)$ if number $S$ fits in a word
- $O(n \log S)$ in general
- $S$ is exponential in $\log S$ (not polynomial)

**Pseudopolynomial time** = polynomial in the problem size and the numbers in input
- $O(nS)$ is pseudopolynomial

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