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 subproblems: acyclic/topological order

* 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 \( x[i:j] \)

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

\[
\begin{array}{ccc}
\begin{array}{c}
A \\
\cdot \\
B \\
\cdot \\
C
\end{array} & \begin{array}{c}
(A \cdot B) \cdot C \\
\Theta(n^2) \text{ time}
\end{array} & \begin{array}{c}
A \cdot (B \cdot C) \\
\Theta(n) \text{ time}
\end{array}
\end{array}
\]

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

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

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

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

5. original problem = \(DP[\emptyset,n]\)
   (& 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 ø, min edit distance equivalent to finding longest common subsequence sequential but not necessarily contiguous

- e.g.: **HIEROGLYPH OLOGY** vs. **MICHAE LANGELLO** ⇒ **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 \( 0 \leq i < |x|, 0 \leq j < |y| \)

   \( \Rightarrow \Theta(|x| \cdot |y|) \) subproblems

2. **Guess whether to turn** \( x \) **into** \( y \),
   - \( x[i] \) deleted
   - \( y[j] \) inserted
   - \( x[i] \) replaced by \( y[j] \)

   \( \{3 \text{ choices}\} \)

3. **Recurrence:**
   \( c(i, j) = \max \{ \)
   
   \( \quad \text{cost(delete } x[i]) + c(i+1, j) \text{ if } i < |x|, \)
   
   \( \quad \text{cost(insert } y[j]) + c(i, j+1) \text{ if } j < |y|, \)
   
   \( \quad \text{cost(replace } x[i] \Rightarrow y[j]) + c(i+1, j+1) \text{ if } i < |x| \& j < |y| \} \)

   - **Base case:** \( c(|x|, |y|) = 0 \)

   \( \Rightarrow \Theta(1) \) 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

   \( \Rightarrow \) Total time = \( \Theta(|x| \cdot |y|) \)

5. **Original problem:** \( c(0, 0) \)
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 $= O(nS)$
2. 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 $= O(1)$
3. **topological order**: for $i$ in $n, \ldots, 0$:
   for $X$ in $0, \ldots, S$
   - total time $= O(nS)$
4. **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
- $\Theta(nS)$ is pseudopolynomial

---

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