DYNAMIC PROGRAMMING II
Techniques for optimization

optimization problems typically involve making choices

backtracking: just try all solutions
  ▪ can be applied to almost all problems, but gives very slow algorithms
  ▪ try all options for first choice,
    for each option, recursively make other choices

greedy algorithms: construct solution iteratively, always make choice that seems best
  ▪ can be applied to few problems, but gives fast algorithms
  ▪ only try option that seems best for first choice (greedy choice),
    recursively make other choices

dynamic programming
  ▪ in between: not as fast as greedy, but works for more problems
5 steps in designing dynamic-programming algorithms

1. define subproblems
   [#subproblems]
2. guess first choice
   [#choices]
3. give recurrence for the value of an optimal solution
   [time/subproblem treating recursive calls as $\Theta(1)$]
   
   i. define subproblem in terms of a few parameters
   ii. define variable $m[..]$ = value of optimal solution for subproblem
   iii. relate subproblems by giving recurrence for $m[..]$

4. algorithm: fill in table for $m[..]$ in suitable order (or recurse & memoize)

5. solve original problem

Running time: #subproblems * time/subproblem
Correctness:  
(i) correctness of recurrence: relate OPT to recurrence
(ii) correctness of algorithm: induction using (i)

Today more examples: 
longest common subsequence and optimal binary search trees
5 steps: Matrix chain multiplication

1. define subproblems
   
   Given $i,j$ with $1 \leq i < j \leq n$, compute $A_i \cdot \ldots \cdot A_j$ in cheapest way

2. guess first choice
   
   Final multiplication

3. give recurrence for the value of an optimal solution
   
   $$m[i,j] = \begin{cases} 
   0 & \text{if } i = j \\
   \min_{i \leq k < j} \{ m[i,k] + m[k+1,j] + p_{i-1}p_kp_j \} & \text{if } i < j
   \end{cases}$$

4. algorithm: fill in table for $m[..]$ in suitable order (or recurse & memoize)

5. solve original problem: split markers $s[i,j]$}

Running time: $\Theta(n^3)$
Longest Common Subsequence
Comparing DNA sequences

DNA: string of characters  
A = adenine  
C = cytosine  
G = guanine  
T = thymine

Problem: measure the similarity of two given DNA sequences

\[ X = \text{A C C G T A A T C G A C G} \]
\[ Y = \text{A C G A T T G C A C T G} \]

How to measure similarity?

- edit distance: number of changes to transform X into Y
- length of longest common subsequence (LCS) of X and Y
- ...
Longest common subsequence

**Substring** of a string:
string that can be obtained by omitting zero or more characters

string: $A C C A T T G$
example substrings: $C C A T T G$ or $A C C T T G$ or ...

$LCS(X,Y)$
= a **longest common subsequence** of strings $X$ and $Y$
= a longest string that is a subsequence of $X$ and a subsequence of $Y$

$$X = \text{AC} \textcolor{red}{G} \text{GT} \textcolor{red}{T} \textcolor{red}{T} \textcolor{red}{T} \textcolor{red}{G} \text{A} \text{C} \text{G}$$
$$Y = \text{AC} \text{G} \text{A} \text{C} \text{G} \text{T} \text{T} \text{G} \text{A} \text{C} \text{G}$$

$LCS(X,Y) = \text{ACGTTGACG}$
(or $\text{ACGTTTCACG} \ldots$)
The LCS problem

Input: strings $X = x_1, x_2, \ldots, x_n$ and $Y = y_1, y_2, \ldots, y_m$

Output: (the length of) a longest common subsequence of $X$ and $Y$

So we have

valid solution = common subsequence

profit = length of the subsequence (to be maximized)
Let's study the structure of an optimal solution

- what are the choices that need to be made?
- what are the subproblems that remain? do we have optimal substructure?

**first choice:**
- is last character of $X$ matched to last character of $Y$?
- or is last character of $X$ unmatched?
- or is last character of $Y$ unmatched?

**greedy choice?**

**overlapping subproblems?**
5 steps in designing dynamic-programming algorithms

1. define subproblems
2. guess first choice
3. give recurrence for the value of an optimal solution
4. algorithm: fill in table for $c[..]$ in suitable order (or recurse & memoize)
5. solve original problem
3. Give recursive formula for the value of an optimal solution
   i. define subproblem in terms of a few parameters
      
      Find LCS of $X_i := x_1 \ldots x_i$ and $Y_j := y_1 \ldots y_j$, for parameters $i, j$ with $0 \leq i \leq n$ and $0 \leq j \leq m$?
      
      $X_0$ is empty string

   ii. define variable $c[\ldots]$ = value of optimal solution for subproblem
      
      $c[i,j] =$ length of LCS of $X_i$ and $Y_j$

   iii. give recursive formula for $c[\ldots]$
X_i := x_1 \ldots x_i \text{ and } Y_j := y_1 \ldots y_j, \text{ for } 0 \leq i \leq n \text{ and } 0 \leq j \leq m

we want recursive formula for c[i,j] = length of LCS of X_i and Y_j

Lemma: c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max \{ c[i-1,j], c[i,j-1] \} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases}
Lemma: \[ c[i,j] = \begin{cases} 0 & \text{if } i = 0 \text{ or } j = 0 \\ c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\ \max\{c[i-1,j], c[i,j-1]\} & \text{if } i,j > 0 \text{ and } x_i \neq y_j \end{cases} \]

Proof:

\( i = 0 \text{ or } j = 0: \) trivial (at least one of \( X_i \) and \( Y_j \) is empty)

\( i,j > 0 \text{ and } x_i = y_j: \)

We can extend LCS \((X_{i-1}, Y_{j-1})\) by appending \( x_i \), so \( c[i,j] \geq c[i-1,j-1] + 1 \).

On the other hand, by deleting the last character from LCS\((X_i, Y_j)\) we obtain a substring of \( X_{i-1} \) and \( Y_{j-1} \). Hence, \( c[i-1,j-1] \geq c[i,j] - 1 \).

It follows that \( c[i,j] = c[i-1,j-1] + 1 \).

\( i,j > 0 \text{ and } x_i \neq y_j: \)

If LCS\((X_i, Y_j)\) does not end with \( x_i \) then \( c[i,j] \leq c[i-1,j] \leq \max\{..\} \).

Otherwise LCS\((X_i, Y_j)\) cannot end with \( y_j \), so \( c[i,j] \leq c[i,j-1] \leq \max\{..\} \).

Obviously \( c[i,j] \geq \max\{..\} \). We conclude that \( c[i,j] = \max\{..\} \).
5 steps in designing dynamic-programming algorithms

1. define subproblems ✓
2. guess first choice ✓
3. give recurrence for the value of an optimal solution ✓
4. algorithm: fill in table for $c[..]$ in suitable order (or recurse & memoize)
5. solve original problem
4. Algorithm: fill in table for $c[..]$ in suitable order

\[
c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
 c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\
 \max \{ c[i-1,j], c[i,j-1] \} & \text{if } i,j > 0 \text{ and } x_i \neq y_j 
\end{cases}
\]
4. Algorithm: fill in table for $c[..]$ in suitable order

$$c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0 \\
(i-1,j-1) + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j \\
\max \{ c[i-1,j], c[i,j-1] \} & \text{if } i,j > 0 \text{ and } x_i \neq y_j 
\end{cases}$$

$LCS$-Length$(X,Y)$
1. $n \leftarrow \text{length}[X]$; $m \leftarrow \text{length}[Y]$
2. for $i \leftarrow 0$ to $n$ do $c[i,0] \leftarrow 0$
3. for $j \leftarrow 0$ to $m$ do $c[0,j] \leftarrow 0$
4. for $i \leftarrow 1$ to $n$
5. do for $j \leftarrow 1$ to $m$
6. do if $X[i] = Y[j]$
7. then $c[i,j] \leftarrow c[i-1,j-1] + 1$
8. else $c[i,j] \leftarrow \max \{ c[i-1,j], c[i,j-1] \}$
9. return $c[n,m]$
Analysis of running time

$LCS-\text{Length}(X,Y)$

$\Theta(1)$ 1. $n \leftarrow \text{length}[X]; m \leftarrow \text{length}[Y]$

$\Theta(n)$ 2. $\textbf{for } i \leftarrow 0 \textbf{ to } n \textbf{ do } c[i,0] \leftarrow 0$

$\Theta(m)$ 3. $\textbf{for } j \leftarrow 0 \textbf{ to } m \textbf{ do } c[0,j] \leftarrow 0$

4. $\textbf{for } i \leftarrow 1 \textbf{ to } n$

5. $\textbf{do for } j \leftarrow 1 \textbf{ to } m$

6. $\textbf{do if } X[i] = Y[j]$

7. $\textbf{then } c[i,j] \leftarrow c[i-1,j-1] + 1$

8. $\textbf{else } c[i,j] \leftarrow \max \{ c[i-1,j], c[i,j-1] \}$

$\Theta(1)$ 9. $\textbf{return } c[n,m]$

Lines 4 – 8: $\sum_{1 \leq i \leq n} \sum_{1 \leq j \leq m} \Theta(1) = \Theta(nm)$
5 steps in designing dynamic-programming algorithms

1. define subproblems
2. guess first choice
3. give recurrence for the value of an optimal solution
4. algorithm: fill in table for \( c[\ldots] \) in suitable order (or recurse & memoize)
5. solve original problem

Approaches for going from optimum value to optimum solution:
   i. store choices that led to optimum (e.g. \( s[i,j] \) for matrix multiplication)
   ii. retrace/deduce sequence of solutions that led to optimum backwards (next slide)
5. Extend algorithm so that it computes an actual solution that is optimal, and not just the value of an optimal solution.

Usual approach: extra table to remember choices that lead to optimal solution
It’s also possible to do without the extra table.

```
Print-LCS(c,X,Y,i,j)
1. if i=0 or j=0
2. then skip
3. else if X[i] = Y[j]
4. then Print-LCS(c,X,Y,i-1,j-1); print x_i
5. else if c[i-1,j] > c[i,j-1]
6. then Print-LCS(c,X,Y,i-1,j)
7. else Print-LCS (c,X,Y,i,j-1)
```

Initial call: $\text{Print-LCS}(c, X, Y, n, m)$

Advantage of avoiding extra table: saves memory
Disadvantage: extra work to construct solution
5 steps in designing dynamic-programming algorithms

1. define subproblems ✓
2. guess first choice ✓
3. give recurrence for the value of an optimal solution ✓
4. algorithm: fill in table for $c[..]$ in suitable order (or recurse & memoize) ✓
5. solve original problem

Approaches for going from optimum value to optimum solution:

i. store choices that led to optimum (e.g. $s[i,j]$ for matrix multiplication)

ii. retrace/deduce sequence of solutions that led to optimum backwards (next slide)
Optimal Binary Search Trees
Observation: balanced binary search tree does not necessarily have best average search time, if certain keys are searched for more often than others.

Example: dictionary for automatic translation of texts (keys are words)

Diagram:
- **Balanced**
  - porcupine
  - is
  - the

- Better average search time
  - porcupine
  - the
  - is
The problem of constructing optimal binary search trees

Input:

<p>| | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$d_0$</td>
<td>$k_1$</td>
<td>$d_1$</td>
<td>$k_2$</td>
<td>$d_2$</td>
<td>$k_3$</td>
<td>$d_3$</td>
</tr>
<tr>
<td>0.001</td>
<td>0.03</td>
<td>0.02</td>
<td>0.04</td>
<td>0.001</td>
<td>0.072</td>
<td>0.002</td>
</tr>
</tbody>
</table>

- keys $k_1, k_2, \ldots, k_n$ with $k_1 < k_2 < \ldots < k_n$
- probabilities $p_1, p_2, \ldots, p_n$ with $p_i = \text{probability that query key is } k_i$
- dummy keys $d_0, d_1, \ldots, d_n$
- probabilities $q_0, q_1, \ldots, q_n$ with $q_i = \text{probability that query key lies between } k_i \text{ and } k_{i+1}$ ($k_0 = -\infty$ en $k_{n+1} = \infty$), that is, that search ends in dummy $d_i$

Note: $\sum p_i + \sum q_i = 1$
Output: tree that minimizes expected search time (expected = weighted average)

```
   d0 k1 d1 k2 d2 k3 d3 k4 d4
  0.02 0.1 0.01 0.15 0.04 0.25 0.08 0.3 0.05
```

The expected cost of a query is given by:

\[
\text{expected cost of a query} = \sum p_i \cdot (1 + \text{depth of } k_i) + \sum q_i \cdot (1 + \text{depth of } d_i)
\]

or

\[
= 1 + \sum p_i \cdot (\text{depth of } k_i) + \sum q_i \cdot (\text{depth of } d_i)
\]

This is what we want to minimize.
Let's study the structure of an optimal solution

- what are the choices that need to be made?
- what are the subproblems that remain? do we have optimal substructure?

\[ d_0, k_1, d_1, k_2, d_2, k_3, d_3, \ldots, k_n, d_n \]

optimal tree for \( d_0, k_1, \ldots, d_{r-1} \)

optimal tree for \( d_r, k_{r+1}, \ldots, d_n \)
5 steps in designing dynamic-programming algorithms

1. define subproblems
2. guess first choice
3. give recurrence for the value of an optimal solution
4. algorithm: fill in table for $m[...]$ in suitable order (or recurse & memoize)
5. solve original problem
3. Give recursive formula for the value of an optimal solution
   i. define subproblem in terms of a few parameters

   Find tree for \( d_{i-1}, k_i, \ldots, d_j \) that minimizes expected search cost
   for parameters \( i, j \) with \( 0 \leq i - 1 \leq j \leq n \)?

   ii. define variable \( m[\ldots] = \) value of optimal solution for subproblem

   \[ m[i,j] = \) expected search cost of optimal tree for \( d_{i-1}, k_i, \ldots, d_j \)

   iii. give recursive formula for \( m[\ldots] \)
Recursive formula for $m[i,j] =$ expected search cost of optimal tree for $d_{i-1}, k_i, \ldots, d_j$

**Lemma:**

\[
\begin{align*}
\text{Expected search cost} \quad & m[i,j] = \\
& \begin{cases} 
q_{i-1} & \text{if } j = i-1 \\
\min \{ m[i,r-1] + m[r+1,j] + w(i,j) \} & \text{if } j > i -1 
\end{cases} \\
\end{align*}
\]

\[
\begin{align*}
\text{Define } w(i,j) &= \sum_{i \leq s \leq j} p_s + \sum_{i-1 \leq s \leq j} q_s \\
\end{align*}
\]
5 steps in designing dynamic-programming algorithms

1. define subproblems ✓
2. guess first choice ✓
3. give recurrence for the value of an optimal solution ✓
4. algorithm: fill in table for $m[..]$ in suitable order (or recurse & memoize)
5. solve original problem
4. Algorithm: fill in table for $m[..]$ in suitable order

\[
m[i,j] = \begin{cases} 
q_{i-1} & \text{if } j = i-1 \\
\min \{ m[i,r-1] + m[r+1,j] + w(i,j) \} & \text{if } j > i-1
\end{cases}
\]

$OptimalBST-Cost\ (p, q)\\n// p and q: arrays with probabilities $p_1, \ldots, p_n$ and $q_0, \ldots, q_n$

1. $n \leftarrow \text{length}[p]$
2. \textbf{for } $i \leftarrow 1$ \textbf{to } $n+1$
3. \textbf{do } $m[i,i-1] \leftarrow q_{i-1}$
4. \textbf{for } $i \leftarrow n$ \textbf{downto } 1
5. \textbf{do } \textbf{for } $j \leftarrow i$ \textbf{to } $n$
6. \textbf{do } $m[i,j] \leftarrow \min \{ m[i,r-1] + m[r+1,j] + w(i,j) \}$, $i \leq r \leq j$
7. \textbf{return } $m[1,n]$

\textbf{precompute}
Analysis of running time

OptimalBST-Cost (p, q)

// p and q: arrays with probabilities \( p_1, \ldots, p_n \) and \( q_0, \ldots, q_n \)
1. \( n \leftarrow \text{length}[p] \)
2. for \( i \leftarrow 1 \) to \( n+1 \)
3. do \( m[i, i-1] \leftarrow q_{i-1} \)
4. for \( i \leftarrow n \) downto 1
5. do for \( j \leftarrow i \) to \( n \)
6. do \( m[i, j] \leftarrow \min_{i \leq r \leq j} \{ m[i, r-1] + m[r+1, j] + w(i, j) \} \)
7. return \( m[1, n] \)

\( \Theta(n^3) \): similar to algorithm for matrix-chain multiplication
5 steps in designing dynamic-programming algorithms

1. define subproblems ✓
2. guess first choice ✓
3. give recurrence for the value of an optimal solution ✓
4. algorithm: fill in table for $m[..]$ in suitable order (or recurse & memoize) ✓
5. solve original problem
5 steps in designing dynamic-programming algorithms

1. define subproblems ✓
2. guess first choice ✓
3. give recurrence for the value of an optimal solution ✓
4. algorithm: fill in table for $m$ [..] in suitable order (or recurse & memoize) ✓
5. solve original problem Try it!
5 steps in designing dynamic-programming algorithms

1. define subproblems
2. guess first choice
3. give recurrence for the value of an optimal solution
   - define subproblem in terms of a few parameters
   - define variable $m[.]$ = value of optimal solution for subproblem
   - relate subproblems by giving recurrence for $m[.]$
4. algorithm: fill in table for $m[.]$ in suitable order (or recurse & memoize)
5. solve original problem

Running time: \#subproblems * time/subproblem
Correctness:
   - (i) correctness of recurrence: relate OPT to recurrence
   - (ii) correctness of algorithm: induction using (i)

A problem to think about: Knapsack with integer weights
- items with values $v_1, \ldots, v_n$ and integer weights $w_1, \ldots, w_n$
- select items to pack of total weight < $W$, maximizing total value
Summary of part I of the course

optimization problems
we want to find valid solution minimizing cost (or maximizing profit)

techniques for optimization

- backtracking: just try all solutions (pruning when possible)

- greedy algorithms: make choice that seems best, recursively solve remaining subproblem

- dynamic programming: give recursive formula, fill in table

always start by investigating structure of optimal solution