**LECTURE 14**

Dynamic Programming (DP)

- Revisit our models of computation
- Hallmarks: Greedy vs. DP
- Simple example: Fibonacci
  - Illustrate: overlapping subproblems
  - Ordering the computation: $O(2^n) \rightarrow O(n)$
- Longest common subsequence
  - Recursive formulation & optimal substructure
  - Ordering the computation

Hallmarks of optimization problems

1. Optimal substructure
   - An optimal solution to a problem (instance) contains optimal solutions to subproblems.
2. Overlapping subproblems
   - A recursive solution contains a “small” number of distinct subproblems repeated many times.
3. Greedy choice property
   - Locally optimal choices lead to globally optimal solution
   - Greedy Choice is not possible
   - Globally optimal solution requires trace back through many choices

Overlapping subproblems example: Fibonacci

(not really an optimization problem)

Computing Fibonacci numbers: Top down

- Fibonacci numbers are defined recursively:
  - Python code
    ```python
def fibonacci(n):
    if n==2 or n==1: return 1
    return fibonacci(n-1) + fibonacci(n-2)
```
- Goal: Compute 7th Fibonacci number.
  - $F(2)=1, F(3)=2, F(4)=3, F(5)=5, F(6)=8, F(7)=13$...
  - $1, 2, 3, 5, 8, 13, 21, \ldots$
- Analysis:
  - $T(n) = T(n-1) + T(n-2) + \ldots + T(2) + T(1) \approx O(2^n)$
Computing Fibonacci numbers: Bottom up
• Top-down approach
  – Python code
  
  ```python
  def fibonacci(n):
      fib_table[1] = 1
      fib_table[2] = 1
      for i in range(3,n+1):
          fib_table[i] = fib_table[i-1]+fib_table[i-2]
      return fib_table[n]
  ```

  Analysis: $T(n) = O(n)$

  ![Fibonacci table](image)

Lessons from iterative Fibonacci algorithm
• What did the iterative solution do?
  – Reveal identical sub-problems
  – Order computation to enable result reuse
  – Systematically filled-in table of results
  – Expressed larger problems from their subparts
  
  • Ordering of computations matters
    – Naïve top-down approach very slow
      • results of smaller problems not available
      • repeated work
    – Systematic bottom-up approach successful
      • Systematically solve each sub-problem
      • Fill-in table of sub-problem results in order.
      • Look up solutions instead of recomputing

Longest Common Subsequence (LCS)
• Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

  ![LCS example](image)

  “a” not “the”

  $x$: A B C B D A B
  $y$: B D C A B A

  BCBA = LCS($x$, $y$)

  Functional notation, but not a function

  Three optimal solutions: BCBA, BDAB, BCAB.

  Note: substring (contiguous) vs. subsequence.

Applications of LCS
• Word processing
  – Document comparison
  – Spell-checking
  – Plagiarism detection
• Genomes
  – Sequence alignment
  – Evolutionary analysis
  – Edit distance vs. LCS

Brute-force LCS algorithm
Check every subsequence of $x[1 \ldots m]$ to see if it is also a subsequence of $y[1 \ldots n]$.

Analysis
• Checking = $O(n)$ time per subsequence.
• $2^m$ subsequences of $x$ (each bit-vector of length $m$ determines a distinct subsequence of $x$).

Worst-case running time = $O(n2^m) = \text{exponential time.}$
**Towards a better algorithm**

**Simplification:**
1. Look at the length of a longest-common subsequence.
2. Extend the algorithm to find the LCS itself.

**Notation:** Denote the length of a sequence \( s \) by \( |s| \).

**Strategy:** Consider prefixes of \( x \) and \( y \).

• Define \( c[i, j] = |\text{LCS}(x[1 \ldots i], y[1 \ldots j])| \).
• Then, \( c[m, n] = |\text{LCS}(x, y)| \).

---

**Recursive formulation of LCS**

\[
\text{LCS}(x, y, i, j) \quad \text{// ignoring base cases}
\]

\[
\begin{align*}
\text{if } x[i] &= y[j] \\
&\quad \text{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
&\quad \text{else } c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \\
&\quad \quad \quad \text{LCS}(x, y, i, j-1) \} \\
\text{return } c[i, j]
\end{align*}
\]

Relies on optimal substructure:
- Construct optimal solution recursively, based on optimal solutions to subproblems.

---

**Dynamic-programming hallmark #1**

**Optimal substructure**

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If \( z = \text{LCS}(x, y) \), then any prefix of \( z \) is an LCS of a prefix of \( x \) and a prefix of \( y \).

---

**Proof (continued)**

**Claim:** \( z[1 \ldots k–1] = \text{LCS}(x[1 \ldots i–1], y[1 \ldots j–1]) \).

Suppose \( w \) is a longer CS of \( x[1 \ldots i–1] \) and \( y[1 \ldots j–1] \), that is, \(|w| > k–1\). Then, cut and paste: \( w \parallel z[k] \) (\( w \) concatenated with \( z[k] \)) is a common subsequence of \( x[1 \ldots i] \) and \( y[1 \ldots j] \) with \(|w \parallel z[k]| > k\). Contradiction, proving the claim.

Thus, \( c[i–1, j–1] = k–1 \), which implies that \( c[i, j] = c[i–1, j–1] + 1 \).

Other cases are similar. □

---

**Analysis of recursive formulation**

\[
\text{LCS}(x, y, i, j) \quad \text{// ignoring base cases}
\]

\[
\begin{align*}
\text{if } x[i] &= y[j] \\
&\quad \text{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
&\quad \text{else } c[i, j] \leftarrow \max \{ \text{LCS}(x, y, i-1, j), \\
&\quad \quad \quad \text{LCS}(x, y, i, j-1) \} \\
\text{return } c[i, j]
\end{align*}
\]

Worse case: \( x[i] \neq y[j] \), in which case the algorithm evaluates two subproblems, each with only one parameter decremented.
Introduction to Algorithms, Lecture 14 4/1/2008

© Leiserson, Kellis

**Recursion tree**

\[ m = 7, n = 6: \]

\[
\begin{array}{c}
\vdots \\
7,6 \\
6,6 \\
6,6 \\
5,6 \\
5,5 \\
5,5 \\
5,5 \\
7,5 \\
7,5 \\
7,5 \\
\vdots \\
\end{array}
\]

Height \( m + n \Rightarrow \) work potentially exponential, but we’re solving the same subproblems repeatedly!

**Dynamic-programming hallmark #2**

Overlapping subproblems

A recursive solution contains a “small” number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths \( m \) and \( n \) is only \( mn \).

**Avoiding repeated work**

- Two solutions are possible
  - **Solution 1: Memoization**
    - Top-down approach, recursive.
    - Store information about partial solutions, and re-use them when they come up.
  - **Solution 2: Dynamic programming**
    - Bottom-up approach, iterative.
    - Order computation s.t. when solving a subproblem, the solutions to all needed subproblems are already available.

**Memoization algorithm**

**Memoization:** After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

\[
\begin{align*}
\text{LCS}(x, y, i, j) & \quad \text{if } c[i, j] = \text{NIL} \\
& \begin{cases}
\text{then if } x[i] = y[j] \\
& \text{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1 \\
& \text{else } c[i, j] \leftarrow \max \{ & \text{LCS}(x, y, i-1, j), & \text{LCS}(x, y, i, j-1) \}
\end{cases}
\end{align*}
\]

If already in the table, no work to do.

Otherwise: same as before

Time = \( \Theta(mn) \) = constant work per table entry.

Space = \( \Theta(mn) \).

**Representing the subproblem solutions: indexing by \( c[i, j] \)**

**Parameterization:**

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

**Memoization:**

Top-down recursive calls, storing intermediate results.

**Dyn Prog:**

Compute the table bottom-up.

**Representing the subproblem solutions: indexing by \( c[i, j] \)**

**Dyn Prog:**

Compute the table bottom-up.

**Memoization:**

Top-down recursive calls, storing intermediate results.

**For LCS, Dyn Prog is clearly superior as all subproblems must be computed.**

Time = \( \Theta(mn) \).
Introduction to Algorithms, Lecture 14

Dynamic Programming in Theory

- Hallmarks of Dynamic Programming
  - Optimal substructure: Optimal solution to problem (instance) contains optimal solutions to sub-problems
  - Overlapping subproblems: Limited number of distinct subproblems, repeated many many times
- Typically for optimization problems (unlike Fib example)
  - Optimal choice made locally: max(subsolution score)
  - Score is typically added through the search space
  - Traceback common, find optimal path from indiv. choices
- Middle of the road in range of difficulty
  - Easier: greedy choice possible at each step
  - DynProg: requires a traceback to find that optimal path
  - Harder: no opt. substr., e.g. subproblem dependencies

Dynamic Programming in Practice

- Setting up dynamic programming
  - Find ‘matrix’ parameterization (# dimensions, variables)
    - Example: All-pairs shortest paths: different parameterizations
      - Matrix multiplication: Compute optimal paths of length at most m, using optimal paths of length at most m-1.
      - Floyd-Warshall: Compute optimal paths using vertices {1..k}, using optimal paths using vertices {1..k-1}.
    - Today: Fibonacci=linear. LCS=two-dimensional.
  - Make sure sub-problem space is finite! (not exponential)
    - If not all subproblems are used, better off using memoization
    - If reuse not extensive, perhaps DynProg is not right solution!
  - Traversal order: sub-results ready when you need them
  - Computation order matters! (bottom-up, but not always obvious)
  - Recursion formula: larger problems = F(subparts)
    - Typically F() includes min() or max(): remember choices!
    - Systematically fill in table of results, finding optimal score
    - Trace-back from optimal score \rightarrow optimal solution

Summary

LECTURE 14
Dynamic Programming (DP)
- Revisit our models of computation
- Hallmarks: Greedy vs. DP
- Simple example: Fibonacci
  - Illustrate: overlapping subproblems
  - Ordering the computation: O(2^n) \rightarrow O(n)
- Longest common subsequence
  - Recursive formulation & optimal substructure
  - Ordering the computation

Trace-back to find optimal solution

Previous algo.
only finds score
of optimal soltn.

To find LCS:
• Remember choices
• Reconstruct by
trace back steps.

Space = \Theta(mn).
Exercise:
\Theta(\min\{m, n\}).

Duality: LCS ‘alignment’ \leftrightarrow path through the matrix

Goal: Find best path
through the matrix