LECTURE 12
Dynamic Programming
• Fibonacci Numbers
• Weighted Interval Scheduling
• Longest Common Subsequence

Sofya Raskhodnikova
Dynamic Programming

“Those who cannot remember the past are doomed to repeat it.”

George Santayana, *The Life of Reason, Book I: Introduction and Reason in Common Sense*
Design Techniques So Far

- **Greedy.** Build up a solution incrementally, myopically optimizing some local criterion.
- **Recursion / divide & conquer.** Break up a problem into subproblems, solve subproblems, and combine solutions.
- **Dynamic programming.** Break problem into overlapping subproblems, and build up solutions to larger and larger subproblems.
Dynamic Programming History

• Bellman. [1950s] Pioneered the systematic study of dynamic programming.

• Etymology.
  – Dynamic programming = planning over time.
  – Secretary of Defense was hostile to mathematical research.
  – Bellman sought an impressive name to avoid confrontation.

"it's impossible to use dynamic in a pejorative sense"
"something not even a Congressman could object to"

Dynamic Programming Applications

- Areas.
  - Bioinformatics.
  - Control theory.
  - Information theory.
  - Operations research.
  - Computer science: theory, graphics, AI, compilers, systems, ….

- Some famous dynamic programming algorithms.
  - Unix diff for comparing two files.
  - Viterbi for hidden Markov models / decoding convolutional codes
  - Smith-Waterman for genetic sequence alignment.
  - Bellman-Ford for shortest path routing in networks.
  - Cocke-Kasami-Younger for parsing context free grammars.
Fibonacci Sequence
Fibonacci Sequence

• Sequence defined by
  • \( a_1 = 1 \)
  • \( a_2 = 1 \)
  • \( a_n = a_{n-1} + a_{n-2} \)

1, 1, 3, 5, 8, 13, 21, 34, ...

• How should you compute the Fibonacci sequence?

• Recursive algorithm:

  1. If \( n = 1 \) or \( n = 2 \), then
  2. return 1
  3. Else
  4. \( a = \text{Fib}(n-1) \)
  5. \( b = \text{Fib}(n-2) \)
  6. return \( a + b \)

• Running Time?
Review Question

• Prove that the solution to the recurrence
  \[ T(n) = T(n-1) + T(n-2) + \Theta(1) \] is exponential in \( n \).
Review Question

• Prove that the solution to the recurrence
  \( T(n) = T(n-1) + T(n-2) + \Theta(1) \) is exponential in \( n \).

Easy to show: \( \Omega \left( (\sqrt{2})^n \right) \) by inspecting the recursion tree

Later in the course (if time permits):
  \( \Theta(\phi^n) \) where \( \phi \approx 1.618 \) is the golden ratio
Computing Fibonacci Sequence Faster

- **Observation:** Lots of redundancy! The recursive algorithm only solves $n-1$ different subproblems
- “**Memoization**”: Store the values returned by recursive calls in a sub-table
- **Resulting Algorithm:**

  ```
  Fib(n)
  1. If n = 1 or n=2, then
  2. return 1
  3. Else
  5. For i=3 to n
  6. f[i] ← f[i-1]+f[i-2]
  7. return f[n]
  ```

- **Running Time?**

  $O(n)$ if integer operations take constant time.
Computing Fibonacci Sequence Faster

• Observation: Fibonacci recurrence is linear

\[
\begin{pmatrix}
a_n \\
a_{n-1}
\end{pmatrix} = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix} \begin{pmatrix} a_{n-1} \\
a_{n-2} \end{pmatrix} = \cdots = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}^{n-2} \begin{pmatrix} a_2 \\
a_1 \end{pmatrix}
\]

• Can compute \( A^n \) using only \( O(\log n) \) matrix multiplications; each one takes \( O(1) \) integer multiplications and additions.

• Total running time?

\( O(\log n) \) integer operations. Exponential improvement!

• Exercise: how big an improvement if we count bit operations?

– Multiplying \( k \)-bit numbers takes \( O(k \log k) \) time.

• How many bits needed to write down \( a_n \)?
Weighted Interval Scheduling
Weighted Interval Scheduling

- Weighted interval scheduling problem.
  - Job $j$ starts at $s_j$, finishes at $f_j$, and has weight or value $v_j$.
  - Two jobs compatible if they don't overlap.
  - **Goal**: find maximum weight subset of mutually compatible jobs.
Unweighted Interval Scheduling Review

• **Recall.** Greedy algorithm works if all weights are 1.
  – Consider jobs in ascending order of finish time.
  – Add job to subset if it is compatible with previously chosen jobs.

• **Observation.** Greedy algorithm can fail spectacularly with weights.
Weighted Interval Scheduling

Notation. Label jobs by finishing time: \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Define: \( p(j) = \) largest index \( i < j \) such that job \( i \) is compatible with \( j \).

\[ = \text{largest } i \text{ such that } f_i \leq s_j \]

Example: \( p(8) = 5, p(7) = 3, p(2) = 0. \)
Dynamic Programming: Binary Choice

- **Notation.** \( \text{OPT}(j) = \text{value of optimal solution to the problem consisting of job requests } 1, 2, \ldots, j. \)
  - **Case 1:** \( \text{OPT} \) selects job \( j \).
    - collect profit \( v_j \)
    - can't use incompatible jobs \( \{ p(j) + 1, p(j) + 2, \ldots, j - 1 \} \)
    - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, p(j) \)
  - **Case 2:** \( \text{OPT} \) does not select job \( j \).
    - must include optimal solution to problem consisting of remaining compatible jobs \( 1, 2, \ldots, j-1 \)

\[
\text{OPT}(j) = \begin{cases} 
0 & \text{if } j = 0 \\
\max \{ v_j + \text{OPT}(p(j)), \text{OPT}(j-1) \} & \text{otherwise}
\end{cases}
\]
Weighted Interval Scheduling: Brute Force

- Brute force algorithm.

**Input**: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

**Sort** jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

**Compute** \( p(1), p(2), \ldots, p(n) \)

**Compute-Opt** \( (n) \)

**Compute-Opt** \( (j) \) {
    *if* \( (j = 0) \)
    *return* 0
    *else*
    *return* \( \max(v_j + \text{Compute-Opt}(p(j)), \text{Compute-Opt}(j-1)) \)
}

10/7/2016

S. Raskhodnikova; based on slides by E. Demaine, C. Leiserson, A. Smith, K. Wayne
Memoization. Store results of each sub-problem in a table; lookup as needed.

Input: \( n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n \)

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

for \( j = 1 \) to \( n \)
  \( M[j] = \) empty

\( M[0] = 0 \)

Compute-Opt(\( n \))

M-Compute-Opt(\( j \)) {
  if (\( M[j] \) is empty)
    \( M[j] = \max(w_j + M\text{-Compute-Opt}(p(j)), M\text{-Compute-Opt}(j-1)) \)
  return \( M[j] \)
}
Weighted Interval Scheduling: Run Time

• **Claim.** Memoized version of algorithm takes $O(n \log n)$ time.
  
  – Sort by finish time: $O(n \log n)$.
  
  – Computing $p(\cdot)$: $O(n \log n)$ via repeated binary search.

  – $M$–Compute–Opt$(j)$: each invocation takes $O(1)$ time and either
    
    • (i) returns an existing value $M[j]$
    
    • (ii) fills in one new entry $M[j]$ and makes two recursive calls

  – Case (ii) occurs at most $n$ times $\Rightarrow$ at most $2n$ recursive calls overall

  – Overall running time of $M$–Compute–Opt$(n)$ is $O(n)$. ⊲

• **Remark.** $O(n)$ if jobs are pre-sorted by start and finish times.
Equivalent algorithm: Bottom-Up

- Bottom-up dynamic programming. Unwind recursion.

\[
\text{Input: } n, s_1, \ldots, s_n, f_1, \ldots, f_n, v_1, \ldots, v_n
\]

Sort jobs by finish times so that \( f_1 \leq f_2 \leq \ldots \leq f_n \).

Compute \( p(1), p(2), \ldots, p(n) \)

Iterative-Compute-Opt {
    \[ M[0] = 0 \]
    \[ \text{for } j = 1 \text{ to } n \]
    \[ \text{M}[j] = \max(v_j + \text{M}[p(j)], \text{M}[j-1]) \]
}

return \( \text{M}[n] \)

Total time = (sorting + computing \( p(j) \)) + \( O(n) = O(n \log(n)) \)

S. Raskhodnikova; based on slides by E. Demaine, C. Leiserson, A. Smith, K. Wayne
Weighted Interval Scheduling: Finding a Solution

- Q. Dynamic programming algorithms computes optimal value. What if we want the solution itself?
- A. Do some post-processing.

```plaintext
Run M-Compute-Opt(n)
Run Find-Solution(n)

Find-Solution(j) {
  if (j = 0)
    output nothing
  else if (v_j + M[p(j)] > M[j-1])
    print j
    Find-Solution(p(j))
  else
    Find-Solution(j-1)
}
```

- # of recursive calls \( \leq n \Rightarrow O(n) \).
Longest Common Subsequence

A.k.a. “sequence alignment”

“edit distance”

…
Longest Common Subsequence (LCS)

- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

```
x:  A  B  C  B  D  A  B
y:  B  D  C  A  B  A
```

```
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</th>
<th>D</th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>D</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>B</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

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

"a" not "the"
Longest Common Subsequence (LCS)

- Given two sequences $x[1 \ldots m]$ and $y[1 \ldots n]$, find a longest subsequence common to them both.

```
x:  A  B  C  B  D  A  B
y:  B  D  C  A  B
```

“a” not “the”

```
BCAB = LCS(x, y)
```

S. Raskhodnikova; based on slides by E. Demaine, C. Leiserson, A. Smith, K. Wayne

10/7/2016
Motivation

• Approximate string matching [Levenshtein, 1966]
  – Search for “occurance”, get results for “occurrence”
• Computational biology [Needleman-Wunsch, 1970’s]
  – Simple measure of genome similarity

  cgtacggtacgtagtacggtacggtacggtacggtatcgtacgt
  acgtagtacgtagtacgtagtacgtagtacgtagtacgtagt
Motivation

• Approximate string matching [Levenshtein, 1966]
  – Search for “occurance”, get results for “occurrence”

• Computational biology [Needleman-Wunsch, 1970’s]
  – Simple measure of genome similarity

\[
\begin{align*}
\text{acgta} & \text{cgta} \\
\text{a} & \text{cgta}
\end{align*}
\]

\[
\begin{align*}
\text{acgta} & \text{cgta} \\
\text{cgta}
\end{align*}
\]

• \( n - \text{length}(\text{LCS}(x,y)) \) is called the “edit distance”
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 = $O(n2^m)$
= exponential time.
Simplification:
1. Look for 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

\[ c[i, j] = \begin{cases} 
  c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\
  \max\{c[i-1, j], c[i, j-1]\} & \text{otherwise.}
\end{cases} \]

Base case: \( c[i, j] = 0 \) if \( i = 0 \) or \( j = 0 \).

Case \( x[i] = y[j] \): Without loss of generality, optimal solution matches \( x[i] \) to \( y[j] \)
Recursive formulation

\[
c[i, j] = \begin{cases} 
c[i-1, j-1] + 1 & \text{if } x[i] = y[j], \\
\max \{ c[i-1, j], c[i, j-1] \} & \text{otherwise.}
\end{cases}
\]

Case \( x[i] \neq y[j] \): best matching might use \( x[i] \) or \( y[j] \) (or neither) but not both.
Dynamic-programming hallmark #1

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

If $z = LCS(x, y)$, then any prefix of $z$ is an LCS of a prefix of $x$ and a prefix of $y$. 

10/7/2016
S. Raskhodnikova; based on slides by E. Demaine, C. Leiserson, A. Smith, K. Wayne
Recursive algorithm for LCS

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

\[
\text{if } x[i] = y[j] \\
\quad \text{then } c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1
\]

\[
\text{else } c[i, j] \leftarrow \max\{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \}
\]

\text{return } c[i, j]

\[
\text{Worse case: } x[i] \neq y[j], \text{ in which case the algorithm evaluates two subproblems, each with only one parameter decremented.}
\]
Recursion tree

$m = 7, n = 6$:

Height $= m + n \Rightarrow$ work potentially exponential.
Recursion tree

$m = 7, n = 6$:

Height $= m + n \Rightarrow$ work potentially exponential, but we’re solving subproblems already solved!
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 $mn$. 
Memoization algorithm

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

\[
\text{LCS}(x, y, i, j) =
\begin{cases} 
  \text{NIL} & \text{if } c[i, j] = \text{NIL} \\
  \text{LCS}(x, y, i-1, j-1) + 1 & \text{if } x[i] = y[j] \\
  \max \{ \text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1) \} & \text{otherwise}
\end{cases}
\]

**Time** = $\Theta(mn) = \text{constant work per table entry}.$  
**Space** = $\Theta(mn).$
### Dynamic-programming algorithm

**Idea:**
Compute the table bottom-up.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</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>
<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>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
**IDEA:**
Compute the table bottom-up.

Time = $\Theta(mn)$. 

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</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>
<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>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</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>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>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</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>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
**Dynamic-programming algorithm**

**Idea:**
Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>B</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>
<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>
<td>1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>E</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>F</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>G</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
</tr>
</tbody>
</table>
## Idea:
Compute the table bottom-up.

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Multiple solutions are possible.
**Dynamic-programming algorithm**

**Idea:**

1. Compute the table bottom-up.
2. Time = $\Theta(mn)$.
3. Reconstruct LCS by tracing backwards.
4. Space = $\Theta(mn)$.

**Section 6.7:** $O(\min\{m, n\})$
# Saving space

- Why is space important?
  - Cost of storage
  - Cache misses

- To compute the **cost** of the optimal solution
  - Top to bottom
  - Remember only the previous row
  - $O(m)$ space in addition to input

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