CSE 417
Dynamic Programming (pt 6)
Optimizations
Reminders

> HW5 due today

> HW6 will be posted shortly
  – solve a Knapsack-type problem
  – many additional wrinkles
  – will need some optimization
Dynamic Programming Review

> Apply the steps...
  1. Describe solution in terms of solution to any sub-problems
  2. Determine all the sub-problems you’ll need to apply this recursively
  3. Solve every sub-problem (once only) in an appropriate order

> Key question:
  1. Can you solve the problem by combining solutions from sub-problems?

> Count sub-problems to determine running time
  – total is number of sub-problems times time per sub-problem

**optimal substructure**: (small) set of solutions, constructed from solutions to sub-problems that is guaranteed to include the optimal one
Review From Previous Lectures

> Previously:
  > Find opt substructure by considering how the opt solution could use the last input.
  > Given multiple inputs, consider how opt uses last of either or both
  > Given clever choice of sub-problems, find opt substructure by considering new options
  > Alternatively, consider the shape of the optimal solution in general, e.g., tree structured

> Longest Common Subsequence / Edit Distance:
  > opt either uses last of first, last of second, both, or neither
  > edit distance is a generalization that allows substitutions & has arbitrary costs

> Pattern Matching
  > like max sub-array sum, switch to finding longest ending at each $i$
  > consider how last char of pattern matches: several cases
Outline for Today

- Space Considerations
- Divide & Conquer
- Sparseness
- Monotonicity
Simple Space Optimization

> Many DP algorithms only need to keep a small part of the table...
  – Knapsack only uses 1 .. n-1 with limit V ≤ W for 1 .. n with limit W
    > keep just the column of solutions for 1 .. n-1
    > O(W) space
  – All-pairs shortest path uses solution with intermediate nodes 1 .. k-1 for 1 .. k
    > keep all pairs distance
    > O(n^2) space (rather than O(n^3))
    > likewise for single-shortest shortest path with negative weights
Many DP algorithms only need to keep a small part of the table...

- Knapsack only uses 1 .. n-1 with limit \( V \leq W \) for 1 .. n with limit W
- All-pairs shortest path uses solution with intermediate nodes 1 .. k-1 for 1 .. k
- Longest common subsequence only needs prefixes of \( a_1, \ldots, a_n \) and \( b_1, \ldots, b_m \) that are at most 1 shorter
  > can just keep the previous row / column (whichever is smaller)
  > \( O(1) \) space for pattern matching
- Not just for 2D or 3D tables:
  > max sub-array sum only needs 1 .. n-1 for 1 .. n also
  > document layout in TeX only needs 1 .. j such that the words j+1 .. n-1 can fit on one line
  > \( O(1) \) space
Simple Space Optimization

> Unfortunately, this does apply to many of the others...
  – weighted interval scheduling needs (potentially) every prefix of 1 .. n
  – optimal breakout trades potentially needs every prefix of 1 .. n
  – all problems on trees:
    > optimal BSTs, matrix chain multiplication, optimal polygon triangulation
    > need to consider every choice of root...

```
+---+---+---+---+---+---+
| A | B | C | D | E | F |
+---+---+---+---+---+---+
| 1 | a | b | c | d | e |
+---+---+---+---+---+---+
| 2 | a | 1 | 4 | 10| 18| F2|
+---+---+---+---+---+---+
| 3 | b | 2 | 7 | 15| 26|   |
+---+---+---+---+---+---+
| 4 | c | 3 | 10| 20|   |   |
+---+---+---+---+---+---+
| 5 | d | 4 | 13|   |   |   |
+---+---+---+---+---+---+
| 6 | e |   |   | 5 |   |   |
+---+---+---+---+---+---+
```
Simple Space Optimization

- Unfortunately, this does apply to many of the others...
  
  - weighted interval scheduling needs (potentially) every prefix of 1 .. n
  
  - optimal breakout trades potentially needs every prefix of 1 .. n
  
  - all problems on trees:
    
    > optimal BSTs, matrix chain multiplication, optimal polygon triangulation
    
    > need to consider every choice of root

- No obvious way to improve space use for these
Finding Solutions

> Simple space optimizations only work if we just want **opt value**
  > saw that our formulas for opt value didn’t need earlier parts of the table
  > sometimes that is fine: max sub-array sum, edit distance, pattern matching

> Seemingly need whole table to find the **solution** achieving opt
  > often need solution: opt BST, knapsack, shortest path, etc.
Finding Solution Example

> Find longest common subsequence of
  A = [1, 2, 1, 5, 4, 3]
  B = [2, 1, 3, 2, 1, 4]

> Recall our formula
  \[
  \text{opt value for 1, \ldots, i and 1, \ldots, j} = \\
  \max( \text{opt value for 1, \ldots, i-1 and 1, \ldots, j}, \\
  \text{opt value for 1, \ldots, i and 1, \ldots, j-1}, \\
  (\text{opt value for 1, \ldots, i-1 and 1, \ldots, j-1}) + (1\text{ if } a_i = b_j \text{ else } 0))
  \]
Finding Solution Example

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

No matches with empty lists...
Finding Solution Example

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

match [2]...
## Finding Solution Example

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

match \([2, 1]\)...
Finding Solution Example

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

match [2, 1, 3]...
## Finding Solution Example

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>i</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
</tbody>
</table>
Finding Solution Example

> Table tells us the value of the optimal solution

> Walk **backward** through table to find solution achieving that value

> Each option from formula describes how the opt solution uses the last element(s):

\[
\text{opt value for } 1, \ldots, i \text{ and } 1, \ldots, j = \\
\max( \text{opt value for } 1, \ldots, i-1 \text{ and } 1, \ldots, j, \\
\text{opt value for } 1, \ldots, i \text{ and } 1, \ldots, j-1, \\
(\text{opt value for } 1, \ldots, i-1 \text{ and } 1, \ldots, j-1) + (1 \text{ if } a_i = b_j \text{ else } 0))
\]
Finding Solution Example

<table>
<thead>
<tr>
<th>i</th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
# Finding Solution Example

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

The solution is marked with circles and arrows connecting elements.
More Space Optimizations

> Simple space optimizations only work if we just want opt value
  – saw that our formulas for opt value didn’t need earlier parts of the table

> Seemingly need whole table to find the solution achieving opt
  – not an issue for max-subarray sum or pattern matching
  – BUT is for all of the others

> Particularly important for computational biology
  – DNA similarity (edit distance) and RNA secondary structure
  – inputs can be huge
Outline for Today

> Space Considerations
> Divide & Conquer
> Sparseness
> Monotonicity
Finding Solutions by D&C

> Simple optimization allows us to save a lot of space
  – e.g., go from $O(nm)$ to $O(n)$ or $O(m)$

> BUT we seem to lose the ability to get the actual solution
  – we usually need that

> In fact, we can get the best of both worlds using prior technique: divide & conquer
Finding Solutions by D&C

- Solution comes from the path along which we achieve opt value
- It’s not obvious what sub-problems would be useful...
Finding Solutions by D&C

> It’s not obvious what sub-problems would be useful...
  - these two would work **IF** the optimal solution goes through purple square
  - but there is no way to know if it does
Finding Solutions by D&C

> It’s not obvious what sub-problems would be useful...
>   - these two would work **IF** the optimal solution goes through purple square...
>   - but there is no way to know if it does (in fact, it does not in this case)
Finding Solutions by D&C

> **Problem:** Find out where the opt solution goes through middle col
  
  - if it crosses at \((i, m/2)\), then paths from sub-problem on \(1 .. i\) and \(1 .. m/2\) and sub-problem \(i .. n\) and \(m/2 .. m\) concatenate to give us the full path
Finding Solutions by D&C

> **Problem:** Find out where the opt solution goes through middle col
  
  - if it crosses at \((i, m/2)\), then paths from sub-problem on \(1 \ldots i\) and \(1 \ldots m/2\) and sub-problem \(i \ldots n\) and \(m/2 \ldots m\) concatenate to give us the full path

> **D&C algorithm:**
  
  - find index \(i\) where the opt path passes through \((i, m/2)\)
  - recursively find opt path on \(1 \ldots i\) and \(1 \ldots m/2\)
  - recursively find opt path on \(i \ldots n\) and \(m/2 \ldots m\)
  - return concatenation of those two paths which meet at \((i, m/2)\)
Finding Opt Path at Midpoint

> Find opt values with DP
> Find opt path with D&C assuming we can solve:

**Problem:** Find out where the opt solution goes through middle col

> Q: How do we solve this problem?
> A: dynamic programming
Finding Opt Path at Midpoint

> **Idea:** compute not just the opt value but also where the opt solution passed through the midpoint

\[
\text{opt-mid at } (i,j) = \\
\begin{align*}
\text{undefined} & \quad \text{if } j < \frac{m}{2} \quad \text{(don't know yet)} \\
i & \quad \text{if } j = \frac{m}{2} \\
\text{opt-mid}(i', j') & \quad \text{if } j > \frac{m}{2}
\end{align*}
\]

where opt solution at \((i, j)\) goes through \((i', j')\)

– we know opt solution goes through \((i', j')\) — some sub-problem
– we know opt solution from \((i', j')\) crosses mid at opt-mid\((i', j')\)
Finding Opt Path at Midpoint

![Diagram of a grid with a robot and coins, and a table of opt values.]

<p>| | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

opt values
Finding Opt Path at Midpoint

\[
\begin{array}{cccccccc}
2 & 2 & 2 & 2 & 2 & 3 & 3 & 3 \\
1 & 2 & 0 & 2 & 2 & 3 & 3 & 3 \\
1 & 2 & 2 & 2 & 0 & 1 & 1 & 1 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
0 & 0 & 0 & 1 & 0 & 1 & 0 & 2 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 2 \\
\end{array}
\]
Finding Opt Path at Midpoint

Opt values

<p>| | | | | | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

Opt mid-point
Finding Opt Path at Midpoint

<table>
<thead>
<tr>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>2</th>
<th>3</th>
<th>3</th>
<th>3</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>0</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
</tbody>
</table>

opt values

<table>
<thead>
<tr>
<th>6</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
</tbody>
</table>

opt path from purple spot goes left

opt mid-point
Finding Opt Path at Midpoint

Opt mid-point

Opt path from purple spot goes down
Finding Opt Path at Midpoint

opt values

opt mid-point

opt path from purple spot goes left or down
Finding Opt Path at Midpoint

Finding Opt Path at Midpoint
Finding Opt Path at Midpoint

opt values

opt mid-point

opt path from purple spot goes down
Finding Opt Path at Midpoint

Opt mid-point

Opt values

Opt path from purple spot goes down
Finding Opt Path at Midpoint

Opt values

Opt mid-point
Finding Opt Path at Midpoint

opt values

2 2 2 2 2 3 3 3 3
1 2 0 2 2 3 3 3 3
1 2 2 2 0 1 1 1 2
0 0 0 1 1 1 1 0 2
0 0 0 1 0 1 0 2 2
0 0 0 1 1 1 1 2 2

opt mid-point

6 5 5 5
5 5 5 5
4 3 3 3
3 3 3
2 1 1
1 1 1 1 1

W
Finding Opt Path at Midpoint

- **Opt Mid-point**
- **Opt Values**
  - 2 2 2 2 2 3 3 3 3
  - 1 2 0 2 2 3 3 3 3
  - 1 2 2 2 0 1 1 1 2
  - 0 0 0 1 1 1 1 0 2
  - 0 0 0 1 0 1 0 2 2
  - 0 0 0 1 1 1 1 2 2

- **Opt path from purple spot** goes down not left
Finding Opt Path at Midpoint

Opt values

Opt mid-point

Opt path from purple spot goes left not down
Finding Opt Path at Midpoint

opt values

2 2 2 2 2 3 3 3 3
1 2 0 2 2 3 3 3 3
1 2 2 2 0 1 1 1 2
0 0 0 1 1 1 1 0 2
0 0 0 1 0 1 0 2 2
0 0 0 1 1 1 1 2 2

opt mid-point

6 5 5 5 5
5 5 5 5 5
4 3 3 3 1
3 3 3 1
2 1 1 1
1 1 1 1 1

W
Finding Solutions by D&C: Summary

> Compute opt-mid in (at most) same amount of time as opt value
  – record not only which value in the set is the min/max but also which one it was
  – only changes total running time by a constant factor

> If we can compute the opt value in time $t(n)$ and space $s(n)$, we can compute the opt solution in time $O(t(n) \log n)$ and space $s(n)$

> Can still save space by keeping only last row / col, at the cost of $O(\log n)$ factor in running time
Outline for Today

> Space Considerations
> Divide & Conquer
> Sparseness
> Monotonicity
Sparse Tables

> Sometimes, only a fraction of the matrix has useful values
> We have seen one example of this already...
Sparse Tables

- Sometimes, only a fraction of the matrix has useful values
- We have seen one example of this already...
  - entries containing rocks are not useful since they cannot be used on a path
Sparse Tables

> Sometimes, only a fraction of the matrix has useful values
  – we have seen one example of this already... the robot problem
  – entries containing rocks are not useful since they cannot be used on a path

> Save space (and time) by not storing (or computing) these values
Sparse Tables

> Sometimes, only a fraction of the matrix has useful values
  – we have seen one example of this already... the robot problem
  – entries containing rocks are not useful since they cannot be used on a path

> The robot problem is trying to find the best path in a **directed acyclic graph** (DAG)
  – solve that by DP even with negative weights (since no cycles)
  – algorithm runs in $O(n + m)$ time for $n$ node, $m$ edge graph
    > (note that we need to process nodes in the right order)

> Storing data for nodes save space/time in **sparse** graphs
Sparseness in LCS

> Table entries can also be useless in other ways...
In longest common subsequence, the only “interesting” table entries are those for \((i, j)\) with \(a_i = b_j\).

<table>
<thead>
<tr>
<th></th>
<th>2</th>
<th>1</th>
<th>3</th>
<th>2</th>
<th>1</th>
<th>4</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>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>1</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Sparseness in LCS

- Only place where you add elements to solution...
- Solution is a path that only moves in up/left direction with each step.
Sparseness in LCS

> In longest common subsequence, the only “interesting” table entries are those for \((i, j)\) with \(a_i = b_j\)

> Could solve the problem by constructing a graph whose nodes are the interesting entries
  – if there are \(K\) such entries, we have \(K\) nodes and \(O(K^2)\) edges
  – DAG algorithm finds the best path in \(O(K^2)\) time
  – total time to build and solve is \(O(n + m + K^2)\)
    > put each sequence into a hash table... (or use sort & binary search)
  – this is faster provided that \(K < (nm)^{0.5}\)
In longest common subsequence, the only “interesting” table entries are those for \((i, j)\) with \(a_i = b_j\).

Non-interesting entries are just the max of the values left & above

- applied recursively...
- each such entry is the maximum of the interesting entries above / left
- this is a slow way to solve the problem if there are few interesting entries
Knapsack uses a lot of memory... would be nice to have sparseness.

As you’ll see in HW6, however, that is not what happens:
- column for prefix 1..j stores optimal solution for any subset of items 1–j
- there are $2^j$ such subsets, most of which have distinct values
- the columns quickly become dense

One special case: all numbers have a common divisor:
- then every sum is divisible by that number as well
- so there is no need to store table entries for other numbers
**Sparseness Summary**

> DP solves shortest path on DAGs in $O(n+m)$ time
  - (more details on how to order the nodes in the general case...)

> Can use that to speed up the computation when the entries used in the solution are limited to a small subset of “interesting” ones
  - for LCS, the optimal path goes through non-interesting entries
    BUT those do not contribute anything to the actual solution
  - non-interesting entries indicate an element was *not used*

> Don’t expect sparseness when solutions are subsets
  - number of possible subsets grows very quickly
Outline for Today

> Space Considerations
> Divide & Conquer
> Sparseness
> Monotonicity
There are also some general techniques for improving the time complexity of DP algorithms in certain cases
- see, e.g., Yao on “quadrangle inequalities”
- see, e.g., Galil & Park on exploiting convexity / concavity

We will look at the “Knuth optimization”
- probably the most famous of these
Knuth Optimization

> This technique applies where our matrix of solutions OPT satisfies:

\[ OPT[i, j-1] \leq OPT[i, j] \leq OPT[i+1, j] \quad \text{for all } i, j \]

> This holds for, e.g., the optimal binary search tree problem
  – (see Knuth’s “Art of Computer Programming” volume 3)
  – this is in no way obvious!
    > (in fact, it’s not even true as stated...
      true claim is about which element is the root, not the optimal value)

> With that, the optimization is straightforward...
Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>(i, j-1)</th>
<th>(i, j)</th>
<th>(i+1, j)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>≤</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>(i-1, j-2)</th>
<th>(i-1, j-1)</th>
<th>(i, j)</th>
<th>(i, j-1)</th>
<th>(i+1, j)</th>
</tr>
</thead>
<tbody>
<tr>
<td>(i-1, j-2)</td>
<td>≤</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i, j-1)</td>
<td>≺</td>
<td>≤</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i, j)</td>
<td>≺</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>(i+1, j)</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
### Knuth Optimization

<table>
<thead>
<tr>
<th>(i-1, j-2)</th>
<th>(i-1, j-1)</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>(i, j-1)</td>
<td>(i, j)</td>
</tr>
<tr>
<td></td>
<td>(i+1, j)</td>
<td>(i+1, j+1)</td>
</tr>
<tr>
<td></td>
<td>(i+2, j+1)</td>
<td></td>
</tr>
</tbody>
</table>

\[ \ldots \leq \text{OPT}[i-1, j-1] \leq \text{OPT}[i, j] \leq \text{OPT}[i+1, j+1] \leq \ldots \]

and each is limited to a **non-overlapping** range
Knuth Optimization

> Recall, in optimal BST problem, sub-problems are ranges $i \ldots j$

> Base cases are ranges of size 1 ($i \ldots i$)

> Solve problem on $i \ldots j$ using only subranges of $i \ldots j$
  - in particular, we need only shorter ranges to solve longer ones
  - ranges of the same length are along the same diagonal...
## Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>1 .. 1</th>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2 .. 2</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3 .. 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4 .. 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5 .. 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6 .. 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7 .. 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>1 .. 1</th>
<th>1 .. 2</th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 .. 2</td>
<td>2 .. 3</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 .. 3</td>
<td>3 .. 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 .. 4</td>
<td>4 .. 5</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 .. 5</td>
<td>5 .. 6</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 .. 6</td>
<td>6 .. 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td>7 .. 7</td>
</tr>
</tbody>
</table>

*size 2...*
Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>1 .. 1</th>
<th>1 .. 2</th>
<th>1 .. 3</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>2 .. 2</td>
<td>2 .. 3</td>
<td>2 .. 4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3 .. 3</td>
<td>3 .. 4</td>
<td>3 .. 5</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4 .. 4</td>
<td>4 .. 5</td>
<td>4 .. 6</td>
<td></td>
<td></td>
</tr>
<tr>
<td>5 .. 5</td>
<td>5 .. 6</td>
<td>5 .. 7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>6 .. 6</td>
<td>6 .. 7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7 .. 7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## Knuth Optimization

<table>
<thead>
<tr>
<th></th>
<th>1..1</th>
<th>1..2</th>
<th>1..3</th>
<th>1..4</th>
<th>1..5</th>
<th>1..6</th>
<th>1..7</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>2..2</td>
<td>2..3</td>
<td>2..4</td>
<td>2..5</td>
<td>2..6</td>
<td>2..7</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3..3</td>
<td>3..4</td>
<td>3..5</td>
<td>3..6</td>
<td>3..7</td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4..4</td>
<td>4..5</td>
<td>4..6</td>
<td>4..7</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5..5</td>
<td>5..6</td>
<td>5..7</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6..6</td>
<td>6..7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7..7</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Knuth Optimization

> Recall, in optimal BST problem, sub-problems are ranges $i .. j$
  - base cases are ranges of size 1 along the main diagonal
  - solve remaining problems along increasing upper diagonals

> Along each diagonal, values can only increase moving down / right
> AND each lies in a non-overlapping range
  - $\text{OPT}[i-1, j-2]$ to $\text{OPT}[i, j-1]$
  - $\text{OPT}[i, j-1]$ to $\text{OPT}[i+1, j]$
  - $\text{OPT}[i+1, j]$ to $\text{OPT}[i+2, j+1]$
  - ...
Knuth Optimization

\[
\begin{array}{c|c|c}
(i-1, j-2) & (i-1, j-1) & \vdots \\
\hline
(i, j-1) & (i, j) & \vdots \\
\hline
(i+1, j) & (i+1, j+1) & \vdots \\
\hline
(i+2, j+1) & \vdots \\
\end{array}
\]

\[\leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \quad \leq \]
Recall, in optimal BST problem, sub-problems are ranges $i \ldots j$
- base cases are ranges of size 1 along the main diagonal
- solve remaining problems along increasing upper diagonals

Along each diagonal, values can only increase AND each lies in a *non-overlapping* range

If all the values are in a range of size $O(n)$, we can compute them all in $O(n)$ time
- reduces the overall running time from $O(n^3)$ to $O(n^2)$