Today:

- Dynamic programming
  - Longest common subsequence
  - Optimal BST
  - (Memoizing)

Dynamic Programming

- Dynamic programming is a metatechnique (not an algorithm) - like divide and conquer.
- Here, “programming” isn’t computer programming; word comes from table-based solution method.
- Divide & Conquer: break up into smaller problems
- Dynamic Programming: solve all smaller problems but only reuse optimal subproblem solutions

Example: Longest common subsequence (LCS)

§16.2, 16.3

Problem: Given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence common to both.

$x$: A B C B D A B

/   /   \\   \ \\

$y$: B D C A B A

$\Rightarrow$ B C B A.

Brute-force algorithm: For every subsequence of $x$, check if it’s a subsequence of $y$.

Worst-case running time: $\Theta(n2^m)$

($2^m$ subsequences of $x$ to check; each check takes $\Theta(n)$ time)

Recursive Algorithm

Better way:

For now, compute only length, not actual sequence.

Define $c[i, j] =$ length of LCS of “prefixes” $x[1..i]$ and $y[1..j]$.

Then $c[m, n] =$ length of LCS of $x$ and $y$.

Theorem:

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

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

Proof of case where \( x[i] = y[j] \)

Let \( z[1 .. k] \) be LCS of \( x[1 .. i] \) and \( y[1 .. j] \)
(i.e., \( c[i, j] = k \)).

Now, \( z[k] = x[i](= y[j]) \) (else could extend \( z \) by \( x[i] \)).
Thus \( z[1 .. k-1] \) is LCS of \( x[1 .. i-1] \) and \( y[1 .. j-1] \).
If \( \exists \) a LCS \( w \) longer than \( z[1 .. k-1] \),
\( \langle w, x[i] \rangle \) would be a LCS longer than \( z \).
Thus \( z[1 .. k-1] \) is LCS of \( x[1 .. i-1] \) and \( y[1 .. j-1] \)
(i.e., \( c[i-1, j-1] = k-1 \)).

Dynamic Programming Algorithm

LCS-LENGTH(X, Y)
1 \( m \leftarrow \text{length}[X] \)
2 \( n \leftarrow \text{length}[Y] \)
3 \text{for } i \leftarrow 1 \text{ to } m
4 \quad \text{do } c[i, 0] \leftarrow 0
5 \text{for } j \leftarrow 0 \text{ to } n
6 \quad \text{do } c[0, j] \leftarrow 0
7 \text{for } i \leftarrow 1 \text{ to } m
8 \quad \text{do for } j \leftarrow 1 \text{ to } n
9 \quad \text{do if } x[i] = y[j]
10 \quad \quad \text{then } c[i, j] \leftarrow c[i-1, j-1] + 1
11 \quad \quad \text{else if } c[i-1, j] \geq c[i, j-1]
12 \quad \quad \quad \text{then } c[i, j] \leftarrow c[i-1, j]
13 \quad \quad \text{else if } c[i-1, j-1] \geq c[i, j-1]
14 \quad \quad \quad \text{then } c[i, j] \leftarrow c[i-1, j-1]
15 \quad \quad \quad \text{else if } c[i, j-1] \geq c[i-1, j]
16 \quad \quad \quad \text{then } c[i, j] \leftarrow c[i, j-1]
17 \quad \text{return } c \text{ and } b

Example

\( X = \langle A, B, C, B, D, A, B \rangle \)
\( Y = \langle B, D, C, A, B, A \rangle \)
Which table entries must be known to compute \( c[i, j] \)?

<table>
<thead>
<tr>
<th>( i )</th>
<th>0</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>( j )</td>
<td></td>
<td>B</td>
<td>D</td>
<td>C</td>
<td>A</td>
<td>B</td>
<td>A</td>
</tr>
<tr>
<td>0</td>
<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>A</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>5</td>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>6</td>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>7</td>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>

Recursive Algorithm

Use recursive formulation directly

Example (for length 3, length 4 sequence):

What is running time of this algorithm? (Why?)
How many distinct subproblems are there?
One for each position in strings \( x, y \), for \( O(mn) \).
Each encountered many times in recursion tree!
Dynamic Programming

When can you apply it?

- Computational task must have recursive formulation.
- No cycles in formulation (usually, problem should be reduced to "smaller" problems).
- Total number of problem instances to be solved must be small, say N.
- Then running time is \( O(N \times \text{time to compute recursion rule}) \).

Example 2: Optimal BST (Sedgewick)

Suppose we are doing string searching, and know relative key frequencies

Example: spell; cc

Use Dynamic Programming to optimize BST

Reconstructing the Sequence

```plaintext
PRINT-LCS(b, X, i, j)
1 if i = 0 or j = 0
2 then return
3 if b[i, j] = "\""
4 then PRINT-LCS(b, X, i - 1, j - 1)
5 print \( x_i \)
6 elseif b[i, j] = "\""
7 then PRINT-LCS(b, X, i - 1, j)
8 else PRINT-LCS(b, X, i, j - 1)
```

Memoizing

Same idea, different implementation:
store optimal subanswers directly

```plaintext
MEMOIZED-LCS-LENGTH(X, Y)
1 m \leftarrow \text{length}[X]
2 n \leftarrow \text{length}[Y]
3 for i \leftarrow 0 to m
4 \quad \text{do for } j \leftarrow 0 to n
5 \quad \text{do } c[i, j] \leftarrow -1
6 \text{return } \text{LOOKUP-LCS}(m, n)
```

```plaintext
LOOKUP-LCS(i, j)
1 if c[i, j] \geq 0
2 then return c[i, j]
3 if x_i = y_j
4 then c[i, j] \leftarrow \text{LOOKUP-LCS}(i - 1, j - 1) + 1
5 else c[i, j] \leftarrow \max(\text{LOOKUP-LCS}(i - 1, j), \text{LOOKUP-LCS}(i, j - 1))
6 return c[i, j]
```
**Weighted Internal Path Length**

Example BST, with search frequencies

![BST Diagram]

Define “cost” of tree as frequency-weighted sum of node distances from root
This is WIPL; here

\[ 1 \times 1 + 4 \times 2 + 2 \times 2 + 2 \times 3 + 3 \times 3 + 1 \times 3 + 5 \times 4 = 51 \]

**Intuition**

Want to put highest-frequency keys near root
But must balance against cost of increased path lengths
First try: insert in order of decreasing frequency

WIPL is

\[ 1 \times 5 + 4 \times 2 + 2 \times 2 + 3 \times 3 + 1 \times 3 + 2 \times 4 + 5 \times 1 = 42 \]

Can we do better?

**Algorithm**

Given: keys \( K_1 < K_2 < \ldots < K_n \), 1 \( \leq i \leq n \)
and frequencies \( f_i, 1 \leq i \leq n \)
Find BST that minimizes, over all keys, of frequency times distance from root (access cost)
Idea: for subsequences of length 1, 2, 3, \( \ldots \) N-1
Find optimal BST for each subsequence. How?

- For each \( k, i \leq k \leq i + j \) \( [j + 1 \text{ is length}] \)
- Place \( K_k \) at root of T
- Lookup optimal BST \( L \) of \( K_i \ldots K_{k-1} \) (left)
- Lookup optimal BST \( R \) of \( K_{k+1} \ldots K_{k+j} \) (right)
- \( \text{WIPL}(T) = \text{WIPL}(L) + \text{WIPL}(R) + \sum_{i=1}^{i=j} f_i \) (why?)
  so compare to current optimum and update

**Example (N = 4)**

![Example BST Diagram]

Which trees are enumerated?
Which trees are not enumerated?
When to use dynamic programming?

Optimal substructure: optimal solution to problem instance contains optimal solutions to subinstances. Overlapping subproblems: total number of distinct subproblems small compared to recursive run time. (Alternative: “Memoizing,” a top-down approach.)