CS 561, Lecture 6

Jared Saia
University of New Mexico
Dynamic Programming

- Intro
- String Alignment
- Matrix Multiplication
- Longest Common Subsequence
"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 (1905)

What is Dynamic Programming?

- Dynamic Programming is basically “Divide and Conquer” with memorization
- Basic Trick is: Don’t solve the same problem more than once!
Consider the following procedure for computing the $n$-th Fibonacci number:

Fib(n) {
    if (n < 2)
        return n;
    else
        return Fib(n-1) + Fib(n-2);
}

• Q: What is the runtime of Fib?
• A: Except for recursive calls, the entire algorithm takes a constant number of steps. If $T(n)$ is the run time of the algorithm on input $n$, then we can say that:

$T(0) = T(1) = 1$, $T(n) = T(n - 2) + T(n - 1) + 1$

• It’s easy to show by induction that $T(n) = 2F_{n+1} - 1$. This is very bad!
• Q: How can we solve $T(n)$ exactly?
• A: We solved this recurrence using annihilators in the last lecture to get $T(n) = c_1 \phi^n + c_2 \hat{\phi}^n + c_3 1^n$ where $\phi = \frac{1+\sqrt{5}}{2}$ and $\hat{\phi} = \frac{1-\sqrt{5}}{2}$. 

Aside II

- If we solve for constants, we get that:

\[
T(0) = 1 = c_1 + c_2 + c_3
\]

\[
T(1) = 1 = c_1 \phi + c_2 \hat{\phi} + c_3
\]

\[
T(2) = 3 = c_1 \phi^2 + c_2 \hat{\phi}^2 + c_3
\]

Solving this system of linear equations (using Gaussian elimination) gives:

\[
c_1 = 1 + \frac{1}{\sqrt{5}}, \quad c_2 = 1 - \frac{1}{\sqrt{5}}, \quad c_3 = -1,
\]
So our final solution is

\[ T(n) = \left( 1 + \frac{1}{\sqrt{5}} \right) \phi^n + \left( 1 - \frac{1}{\sqrt{5}} \right) \hat{\phi}^n - 1 = \Theta(\phi^n). \]
The Problem

- The reason Fib is so slow is that it computes the same Fibonacci numbers over and over
- In general, there are $F_{k-1}$ recursive calls to Fib(n-k)
- We can greatly speed up the algorithm by writing down the results of the recursive calls and looking them up if needed
DP-Fib(n)
{
    if (n<2)
        return n;
    else{
        if (F[n] is undefined){
            F[n] = DP-Fib(n-1) + DP-Fib(n-2);
        }
        return F[n];
    }
}
Analysis

- For every value of $x$ between 1 and $n$, DP-Fib(x) is called exactly one time.
- Each call does constant work
- Thus runtime of DP-Fib(n) is $\Theta(n)$ - a huge savings
Take Away

Dynamic Programming is different than Divide and Conquer in the following way:

- “Divide and Conquer” divides problem into independent subproblems, solves the subproblems recursively and then combines solutions to solve original problem
- Dynamic Programming is used when the subproblems are not independent, i.e. the subproblems share subsubproblems
- For these kinds of problems, divide and conquer does more work than necessary
- Dynamic Programming solves each subproblem once only and saves the answer in a table for future reference
The Pattern

- **Formulate the problem recursively.** Write down a formula for the whole problem as a simple combination of answers to smaller subproblems.
- **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order.

Note: Dynamic Programs store the results of intermediate subproblems. This is frequently *but not always* done with some type of table.
The *edit distance* between two words is the minimum number of letter insertions, letter deletions, and letter substitutions required to transform one word into another. For example, the edit distance between *FOOD* and *MONEY* is at most four:

\[
\text{FOOD} \rightarrow \text{MOOD} \rightarrow \text{MOND} \rightarrow \text{MONED} \rightarrow \text{MONEY}
\]
Better way to display this process:

- Place the words one above the other in a table
- Put a gap in the first word for every insertion and a gap in the second word for every deletion
- Columns with two different characters correspond to substitutions
- Then the number of editing steps is just the number of columns that don’t contain the same character twice
Example

- String Alignment for “FOOD” and “MONEY”:
  
  \[
  \begin{array}{cccc}
  F & O & O & D \\
  M & O & N & E \end{array}
  \]

- It’s not too hard to see that we can’t do better than four for the edit distance between “Food” and “Money”
• Unfortunately, it can be more difficult to compute the edit distance exactly. Example:

    A L G O R I T H M
    A L T R U I S T I C
Key Observation

- If we remove the last column in an optimal alignment, the remaining alignment must also be optimal.
- Easy to prove by contradiction: Assume there is some better subalignment of all but the last column. Then we can just paste the last column onto this better subalignment to get a better overall alignment.
- Note: The last column can be either: 1) a blank on top aligned with a character on bottom, 2) a character on top aligned with a blank on bottom or 3) a character on top aligned with a character on bottom.
To develop a DP algorithm for this problem, we first need to find a recursive definition.

Assume we have a \( m \) length string \( A \) and an \( n \) length string \( B \).

Let \( E(i, j) \) be the edit distance between the first \( i \) characters of \( A \) and the first \( j \) characters of \( B \).

Then what we want to find is \( E(n, m) \).
• Say we want to compute $E(i, j)$ for some $i$ and $j$
• Further say that the “Recursion Fairy” can tell us the solution to $E(i', j')$, for all $i' \leq i$, $j' \leq j$, except for $i' = i$ and $j' = j$
• Q: Can we compute $E(i, j)$ efficiently with help from the our fairy friend?
There are three possible cases:

- **Insertion**: $E(i, j) = 1 + E(i - 1, j)$
- **Deletion**: $E(i, j) = 1 + E(i, j - 1)$
- **Substitution**: If $a_i = b_j$, $E(i, j) = E(i - 1, j - 1)$, else $E(i, j) = E(i - 1, j - 1) + 1$
Let $I(A[i] \neq B[j]) = 1$ if $A[i]$ and $B[j]$ are different, and 0 if they are the same. Then:

$$E(i, j) = \min \left\{ \begin{array}{l} E(i - 1, j) + 1, \\ E(i, j - 1) + 1, \\ E(i - 1, j - 1) + I(A[i] \neq B[j]) \end{array} \right\}$$
It’s not too hard to see that:

- $E(0, j) = j$ for all $j$, since the $j$ characters of $B$ must be aligned with blanks
- Similarly, $E(i, 0) = i$ for all $i$
• We now have enough info to directly create a recursive algorithm.
• The run time of this recursive algorithm would be given by the following recurrence:

\[
T(m, 0) = T(0, n) = O(1)
\]

\[
T(m, n) = T(m, n - 1) + T(m - 1, n) + T(n - 1, m - 1) + O(1)
\]

• Solution: \( T(n, n) = \Theta(1 + \sqrt{2^n}) \), which is terribly, terribly slow.
We can build up a $m \times n$ table which contains all values of $E(i,j)$
We start by filling in the base cases for this table: the entries in the 0-th row and 0-th column
To fill in any other entry, we need to know the values directly above, to the left and above and to the left.
Thus we can fill in the table in the standard way: left to right and top down to ensure that the entries we need to fill in each cell are always available
Example Table

- Bold numbers indicate places where characters in the strings are equal
- Arrows represent predecessors that define each entry: horizontal arrow is deletion, vertical is insertion and diagonal is substitution.
- Bold diagonal arrows are “free” substitutions of a letter for itself
- Any path of arrows from the top left to the bottom right corner gives an optimal alignment (there are three paths in this example table, so there are three optimal edit sequences).
<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>L</th>
<th>G</th>
<th>O</th>
<th>R</th>
<th>I</th>
<th>T</th>
<th>H</th>
<th>M</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td></td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td>↓</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>1</td>
<td>1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>5</td>
<td>3</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td>4</td>
<td>4</td>
<td>3</td>
<td>3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
<td>5</td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>9</td>
<td>9</td>
<td>8</td>
<td>7</td>
<td>7</td>
<td>7</td>
<td>6</td>
<td>5</td>
<td>5</td>
<td></td>
</tr>
<tr>
<td>10</td>
<td>10</td>
<td>9</td>
<td>8</td>
<td>8</td>
<td>8</td>
<td>7</td>
<td>6</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
The code

```c
EditDistance(A[1,...,m],B[1,...,n]){
    for (i=1;i<=m;i++){
        Edit[i,0] = i;
    }
    for (j=1;j<=n;j++){
        Edit[0,j] = j;
    }
    for (i=1;i<=m;i++){
        for (j=1;j<=n;j++){
            if (A[i]==B[j]){  
                Edit[i,j] = min(Edit[i-1,j]+1,
                                Edit[i,j-1]+1,
                                Edit[i-1,j-1]);
            }else{
                Edit[i,j] = min(Edit[i-1,j]+1,
                                Edit[i,j-1]+1;
                                Edit[i-1,j-1]+1);  
            }
        }
    }
    return Edit[m,n];
}
```
Analysis

- Let $n$ be the length of the first string and $m$ the length of the second string.
- Then there are $\Theta(nm)$ entries in the table, and it takes $\Theta(1)$ time to fill each entry.
- This implies that the run time of the algorithm is $\Theta(nm)$.
- Q: Can you find a faster algorithm?
Reconstructing an optimal alignment

- In this code, we do not keep info around to reconstruct the optimal alignment.
- However, it is a simple matter to keep around another array which stores, for each cell, a pointer to the cell that was used to achieve the current cell’s minimum edit distance.
- To reconstruct a solution, we then need only follow these pointers from the bottom right corner up to the top left corner.
Create a string alignment table for the two strings “abba” and “bab”. Put “abba” at the top of the table and “bab” on the left side.

Qi: \((i = 1, 2, \ldots, 5)\) What is the \(i\)-th row of your table?

Q6: What is the minimum edit distance and how many alignments achieve it?
Take Away

• To solve the string alignment problem, we did the following:
  1) formulated the problem recursively 2) built a solution to
     the recurrence from the bottom up
• Next we’ll see how a similar technique can be used to solve
  the matrix multiplication problem.
Matrix Chain Multiplication

Problem:

- We are given a sequence of \( n \) matrices, \( A_1, A_2, \ldots, A_n \), where for \( i = 1, 2, \ldots, n \), matrix \( A_i \) has dimension \( p_{i-1} \times p_i \).
- We want to compute the product, \( A_1A_2, \ldots, A_n \) as quickly as possible.
- In particular, we want to fully parenthesize the expression above so there are no ambiguities about the how the matrices are multiplied.
- A product of matrices is fully parenthesized if it is either a single matrix, or the product of two fully parenthesized matrix products, surrounded by parentheses.
Paranthesizing Matrices

- There are many ways to paranthesize the matrices.
- Each way gives the same output (because of associativity of matrix multiplications).
- However, the way we paranthesize will affect the time to compute the output.
- Our Goal: Find a paranthesization which requires the minimal number of scalar multiplications.
Example

- In this example, it’s much better to multiply the last two matrices first (this gives us a short, narrow matrix on the right)
- Worse to multiply the first two matrices first (this gives us a short wide matrix on the left)
- In general, our goal is to find ways to always create narrow and short resulting matrices.
A Problem

Problem: There can be many ways to paranthesize. E.g.

- \((A_1(A_2(A_3A_4)))\)
- \((A_1((A_2A_3)A_4))\)
- \(((A_1A_2)(A_3A_4))\)
- \(((A_1(A_2A_3))A_4)\)
- \(((A_1A_2)A_3)A_4)\)
A Problem

- Let $P(n)$ be the number of ways to paranthesize $n$ matrices. Then $P(1) = 1$
- For $n \geq 2$, we know that a fully paranthesized product is the product of two fully paranthesized products, and the split can occur anywhere from $k = 1$ to $k = n - 1$.
- Hence for $n \geq 2$:

\[
P(n) = \sum_{k=1}^{n-1} P(k)P(n-k)
\]

- In the hw, you will show that the solution to this recurrence is $\Omega(2^n)$
Q: Can we develop a DP Solution to this problem?

- **Formulate the problem recursively.** Write down a formula for the whole problem as a simple combination of answers to smaller subproblems
- **Build solutions to your recurrence from the bottom up.** Write an algorithm that starts with the base cases of your recurrence and works its way up to the final solution by considering the intermediate subproblems in the correct order.
Key Observation

- Let $A_{i..j}$ (for $i \leq j$) be the matrix that results from evaluating the product $A_iA_{i+1} \ldots A_j$
- Imagine we are computing $A_{i..j}$
- The last multiplication we do must look like this:

$$A_{i..j} = (A_{i..k}) \times (A_{k+1..j})$$

for some $k$ between $i$ and $j-1$
- Then total cost to compute $A_{i..j}$ is:

- cost to compute $A_{i..k}$ +
- cost to compute $A_{k+1..j}$ +
- cost to multiply $A_{i..k}$ and $A_{k+1..j}$
Recursive Formulation

- For any integers $x, y$, let $m(x, y)$ be the minimum cost of computing $A_{x..y}$
- Then for any $k$ between $i$ and $j - 1$,
  \[ m(i, j) \leq \text{optimal cost to compute } A_{i..k} + \text{optimal cost to compute } A_{k+1..j} + \text{cost to multiply } A_{i..k} \text{ and } A_{k+1..j} \]
- In other words:
  \[ m(i, j) \leq m(i, k) + m(k + 1, j) + \text{cost to multiply } A_{i..k} \text{ and } A_{k+1..j} \]
Cost to Multiply

- $A_{i..k}$ is a $p_{i-1}$ by $p_k$ matrix
- $A_{k+1..j}$ is a $p_k$ by $p_j$ matrix
- Thus multiplying $A_{i..k}$ and $A_{k+1..j}$ takes $p_{i-1}p_kp_j$ operations
- Hence we have:

\[
m(i, j) \leq m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j
\]
Recursive Formulation

- We’ve shown that $m(i, j) \leq m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j$
  for any $k = i, i+1, \ldots, j-1$
- Further note that the optimal parenthesization must use some value of $k = i, i+1, \ldots, j-1$. So we need only pick the best
- Thus we have:

\[
\begin{align*}
m(i, j) &= 0 \text{ if } i = j \\
m(i, j) &= \min_{i \leq k < j} \{ m(i, k) + m(k + 1, j) + p_{i-1}p_kp_j \}
\end{align*}
\]
The Recursive Algorithm

- We now have enough information to write a recursive function to solve the problem.
- The recursive solution will have runtime given by the following recurrence:
  - $T(1) = 1$,
  - $T(n) = 1 + \sum_{k=1}^{n-1}(T(k) + T(n-k) + 1)$
- Unfortunately, the solution to this recurrence is $\Omega(2^n)$ (as shown on p. 346 of the text).
DP Algorithm

- Note that we must solve one subproblem for each choice of $i$ and $j$ satisfying $1 \leq i \leq j \leq n$
- This is only $\binom{n}{2} + n = \Theta(n^2)$ subproblems
- The recursive algorithm encounters each subproblem many times in the branches of the recursion tree.
- However, we can just compute these subproblems from the bottom up, storing the results in a table (this is the DP solution)
Pseudocode

Matrix-Chain-Order(int p[]){
    n = p.length - 1;
    for (i=1;i<=n;i++){
        m(i,i) = 0;
    }
    for (l=2;l<=n;l++){ \l is chain length
        for (i=1;i<=n-l+1;i++){
            j = i+l-1;
            m[i,j] = MAXINT;
            for(k=i;k<=j-1;k++){
                q = m[i,k] + m[k+1,j] + p[i-1]*p[k]*p[j];
                if(q<m[i,j]){m[i,j] = q; s[i,j] = k; }
            }
        }
    }
}
Psuedocode

• This code computes both the optimal cost and a parenthesization that achieves that cost
• It uses an $m$ array to store the optimal costs of computing $m(i, j)$. It also uses a $s$ array, where $s(i, j)$ stores the $k$ value which gives $m(i, j)$
• The parenthesization can be recovered from the $s$ array using the pseudocode in the book on p. 338.
Analysis

- This code has three nested loops, each of which takes on at most $n - 1$ values, and the inner loop takes $O(1)$ time.
- Thus the runtime is $O(n^3)$
- The algorithm also requires $\Theta(n^2)$ space
Example

- Consider the sequence of three matrices, $A_1, A_2, A_3$ whose dimensions are given by the sequence 3, 1, 2, 1 (i.e. $p_0 = 3$, $p_1 = 1$, $p_2 = 2$, $p_3 = 1$)
- Let’s construct the tables giving the optimal parenthesization
- The $(i, j)$ entry of the first table will give the optimal cost for computing $A_{i..j}$, the $(i, j)$ entry of the second table will give a $k$ value which achieves this optimal cost
Computations

- $m(1, 1) = m(2, 2) = m(3, 3) = 0$
- $m(1, 2) = p_0p_1p_2 = 6$
- $m(2, 3) = p_1p_2p_3 = 2$
$m(1, 3) = \min \left\{ \begin{array}{l} m(1, 1) + m(2, 3) + p_0p_1p_3, \\ m(1, 2) + m(3, 3) + p_0p_2p_3 \end{array} \right\}$

$= \min \left\{ \begin{array}{l} 0 + 2 + 3, \\ 6 + 0 + 6 \end{array} \right\}$

$= 5$
Example, m array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
Example, s array

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

• Thus an optimal parenthesization is \((A_1(A_2A_3))\)
• The cost of this is 5
Example II

- Consider the sequence of three matrices, $A_1, A_2, A_3, A_4$ whose dimensions are given by the sequence 3, 1, 2, 1, 2 (i.e. $p_0 = 3$, $p_1 = 1$, $p_2 = 2$, $p_3 = 1$, $p_4 = 2$)
- Let’s construct the tables giving the optimal parenthesization
- The $(i,j)$ entry of the first table will give the optimal cost for computing $A_{i..j}$, the $(i,j)$ entry of the second table will give a $k$ value which achieves this optimal cost
Example II, $m$ array

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>6</td>
<td>5</td>
<td>10</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>0</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>0</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>0</td>
</tr>
</tbody>
</table>
**Example II, s array**

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>-</td>
<td>-</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>3</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Example Computation

\[ m(1, 4) = \min \left\{ m(1, 1) + m(2, 4) + p_0p_1p_4, \right. \]
\[ \left. m(1, 2) + m(3, 4) + p_0p_2p_4, m(1, 3) + m(4, 4) + p_0p_3p_4 \right\} \]

\[ = \min \left\{ 0 + 4 + 6, 6 + 4 + 12, 5 + 0 + 6 \right\} \]

\[ = 10 \]

This minimum is achieved when \( k = 1 \)
Example II

- Thus an optimal parenthesization is \((A_1((A_2A_3)A_4))\)
- The cost of this is 10
In-Class Exercise

- Consider the sequence of three matrices, $A_1, A_2, A_3$ whose dimensions are given by the sequence $1, 2, 1, 2$ (i.e. $p_0 = 1$, $p_1 = 2$, $p_2 = 1$, $p_3 = 2$)
- Q1: What are the $m$ array and $s$ array for these inputs?
- Q2: What is the optimal parenthesization?
Subsequence Definition

- Assume given sequence \( X = \langle x_1, x_2, \ldots, x_m \rangle \)
- Let \( Z = \langle z_1, z_2, \ldots, z_l \rangle \)
- Then \( Z \) is a *subsequence* of \( X \) if there exists a strictly increasing sequence \( \langle i_1, i_2, \ldots, i_k \rangle \) of indices such that for all \( j = 1, 2, \ldots, k \), \( x_{i_j} = z_j \)
Example

- Let $X = \langle A, B, C, B, A, B, D, C \rangle$,
- $Z = \langle A, C, A, B, C \rangle$
- Then, $Z$ is a subsequence of $X$
Common Subsequence

- Given two sequences $X$ and $Y$, we say that $Z$ is a common subsequence of $X$ and $Y$ if $Z$ is a subsequence of $X$ and $Z$ is a subsequence of $Y$
- Example: $X = \langle A, B, D, C, B, A, B, C \rangle$, $Y = \langle A, D, B, C, D, B, A, B \rangle$
- Then $Z = \langle A, B, B, A, B \rangle$ is a common subsequence
- $Z$ is not a longest common subsequence (LCS) of $X$ and $Y$ though since the common subsequence $Z' = \langle A, B, C, B, A, B \rangle$ is longer
- Q: Is $Z'$ a longest common subsequence?
LCS Problem

- We are given two sequences $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$
- Goal: Find a maximum-length common subsequence of $X$ and $Y$
Brute Force

- Brute Force approach is to enumerate all possible subsequences of $X$, check to see if its a subsequence of $Y$, and then keep track of the longest common subsequence of both $X$ and $Y$
- This is slow.
- Q: How many subsequences of $X$ are there?
• Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, for $i = 0, 1, \ldots, m$, let $X_i$ be the $i$-th prefix of $X$ i.e. $X_i = \langle x_1, x_2, \ldots, x_i \rangle$
• Example: if $X = \langle A, B, D, C \rangle$, $X_0 = \langle \rangle$ and $X_3 = \langle A, B, D \rangle$
Lemma 1: Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and let $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be sequences and let $Z = \langle z_1, z_2, \ldots, z_k \rangle$ be any LCS of $X$ and $Y$. Then:

- If $x_m = y_n$, then $z_k = x_m = y_n$ and $Z_{k-1}$ is a LCS of $X_{m-1}$ and $Y_{n-1}$
- If $x_m \neq y_n$, then $z_k \neq x_m$ implies that $Z$ is a LCS of $X_{m-1}$ and $Y$
- If $x_m \neq y_n$, then $z_k \neq y_n$ implies that $Z$ is an LCS of $X$ and $Y_{n-1}$
In-Class Exercise

- Prove each of the three statements in the previous slide
- Hint: Use proof by contradiction
Recursive Solution

- Let $X = \langle x_1, x_2, \ldots, x_m \rangle$ and $Y = \langle y_1, y_2, \ldots, y_n \rangle$ be arbitrary sequences.
- Based on Lemma 1, there are two main possibilities for the LCS of $X$ and $Y$:
  - If $x_m = y_n$, $LCS(X, Y)$ is $LCS(X_{m-1}, Y_{n-1})$ appended to $x_m = y_n$.
  - Otherwise, either $LCS(X, Y)$ is $LCS(X_{m-1}, Y)$ or $LCS(X, Y_{n-1})$ (whichever is larger).
Recursive solution

- Let $c(i, j)$ be the length of an LCS of the sequence $X_i, Y_j$
- Note that $c(i, j) = 0$ if $i$ or $j$ is 0
- Thus we have:

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

- This is already enough to write up a recursive function, however the naive recursive function will take exponential time.
- Instead, we can use dynamic programming and solve from the bottom up.
- Code for doing this is on p. 353 and 355 of the text, basically it uses the same ideas we’ve seen before of filling in entries in a table from the bottom up.
Example

• Consider $X = \langle A, B, D, C, B, A, B, C \rangle$, $Y = \langle A, D, B, C, D, B, A, B \rangle$
• The next slide gives the table constructed by the DP algorithm for computing the LCS of $X$ and $Y$
• The bold numbers represent one possible path giving a LCS.
• The arrows keep track of where the minimum is obtained from
### Example

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>D</th>
<th>C</th>
<th>B</th>
<th>A</th>
<th>B</th>
<th>C</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>
<td>0</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</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>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>D</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>3</td>
<td>4</td>
<td>4</td>
<td>4</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
<td>6</td>
<td>6</td>
</tr>
</tbody>
</table>
Reconstruction

- To find a reconstruction, first find a path of edges leading from the bottom right corner to the top left corner.
- In this path, the target of each diagonal arrow gives a character to include in the LCS.
- In our example, ⟨A, B, C, B, A, B⟩ is the LCS we get by following the edges along the only path from the bottom right to the top left.
Take Away

- We’ve seen four different DP type algorithms
- In each case, we did the following 1) found a recurrence for the solution 2) built solutions to the recurrence from the bottom up
- You should be prepared to do this on your own now!