Dynamic Programming

Topics:
- Elements of DP Algorithms
- Longest Common Subsequence

Longest Common Subsequence

- Similarity can be defined in different ways:
  - Two DNA strands are similar if one is a substring of the other.
  - Two strands are similar if the number of changes needed to turn one into the other is small.
  - There is a third strand $S_3$ in which the bases in $S_3$ appear in each of $S_1$ and $S_2$; these bases must appear in the same order, but not necessarily consecutively. The longer the strand $S_1$ we can find, the more similar $S_1$ and $S_2$ are. (we focus on this)

Longest Common Subsequence

- In biological applications, we often want to compare the DNA of two (or more) different organisms.
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine (A, G, C, T).
- Comparison of two DNA strings
  
  $S_1 = \text{ACCGGTCGAGTCGCCGGAAGCCGCGAA}$
  $S_2 = \text{GTCGTTCGGAATGCCGTTGCTCTGTAAA}$
  $S_3 = \text{GTCGTCCGAAAGCCGCCGAA}$

Dynamic programming for LCS

- Longest common subsequence (LCS)
  - Given two sequences $X[1..m]$ and $Y[1..n]$, find a longest subsequence common to them both.

  $X: A \quad B \quad C \quad B \quad D \quad A \quad B$
  $Y: B \quad D \quad C \quad A \quad B \quad A$

  $\text{LCS}(X, Y) = \text{BCBA}$

  functional notation, but not a function

"a" not "the"
Brute-force LCS algorithm

- For every subsequence of \( X = <x_1, \ldots, x_m> \), check whether it's a subsequence of \( Y = <y_1, \ldots, y_n> \).

- Time: \( O(n2^m) \).
  - \( 2^m \) subsequences of \( X \) (each bit-vector of length \( m \) determines a distinct subsequence of \( X \)).
  - Each subsequence takes \( O(n) \) time to check: scan \( Y \) for first letter, from there scan for second, and so on.

Towards a better algorithm

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

- Strategy: Consider prefixes of \( X \) and \( Y \).
  - Define \( c[i, j] = \) length of LCS of \( X_i \) and \( Y_j \).
  - Then, \( c[m, n] = \) length of LCS of \( X \) and \( Y \).

Optimal substructure

- Notation. \( X_i = <x_1, \ldots, x_i>, Y_j = <y_1, \ldots, y_j> \).
- Theorem. Let \( Z = <z_1, z_2, \ldots, z_k> \) be any LCS of \( X = <x_1, x_2, \ldots, x_m> \) and \( Y = <y_1, y_2, \ldots, y_n> \).
  - 1. If \( x_m = y_n \) then \( z_k = x_m = y_n \) and \( Z \) is an LCS of \( X_{m-1} \) and \( Y_{n-1} \).
  - 2. If \( x_m \neq y_n \) then \( Z \) is an LCS of \( X_{m-1} \) and \( Y \).
  - 3. If \( x_m \neq y_n \) then \( Z \) is an LCS of \( X \) and \( Y_{n-1} \).
- Proof.

Recursive formulation

- Theorem.
  \[
  c[i, j] = \begin{cases} 
  c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j \\
  \max(c[i, j-1], c[i-1, j]) & \text{otherwise}
  \end{cases}
  \]

  Proof. Case \( x_i = y_j \):
  \[
  \begin{array}{cccccc}
  x: & 1 & 2 & \ldots & i & \ldots & m \\
  \hline
  y: & 1 & 2 & \ldots & j & \ldots & n \\
  \end{array}
  \]

  Let \( z = z[1..k] = <z_1, \ldots, z_k> = \text{LCS}(x[1..i], y[1..j]) \), where \( c[i, j] = k \). Then \( z_k = x_i \), or else \( z \) could be extended. Thus, \( z[1..k-1] = \text{CS of } x[1..i-1] \) and \( y[1..j-1] \).
Proof (continued)

1. First show that $z_k = x_m = y_n$. Suppose not. Then make a subsequence $Z' = <z_1, \ldots, z_k, x_m>$. It's a common subsequence of $X$ and $Y$ and has length $k+1 \Rightarrow Z'$ is a longer common subsequence than $Z \Rightarrow Z$ contracts $Z$ being an LCS.

Now show $Z_{k-1}$ is an LCS of $X_{m-1}$ and $Y_{n-1}$. Clearly, it's a common subsequence. Now suppose there exists a common subsequence $W$ of $X_{m-1}$ and $Y_{n-1}$ that's longer than $Z_{k-1} \Rightarrow$ length of $W \geq k$. Make subsequence $W'$ by appending $x_m$ to $W$. $W'$ is a subsequence of $X$ and $Y$, has length $\geq k+1 \Rightarrow$ contracts $Z$ being an LCS.

2. If $z_k \neq x_m$, then $Z$ is a common subsequence of $X_{m-1}$ and $Y$. Suppose there exists a subsequence $W$ of $X_{m-1}$ and $Y$ with length $> k$. Then $W$ is a common subsequence of $X$ and $Y \Rightarrow$ contracts $Z$ being an LCS.


Therefore, an LCS of two sequences contains as a prefix an LCS of the sequences.

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 $Y$.

Recursive algorithm for LCS

Recursive formulation
Define $c[i, j] =$ length of LCS of $X_i$ and $Y_j$. We want $c[m, n]$.

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

Again, we could write a recursive algorithm based on this formulation.

• Lots of repeated subproblems.
• Height $= m + n \Rightarrow$ work potentially exponential, but we’re solving subproblems already solved.
• Instead of re-computing, store in a table.

Recursive tree

$m = 3, n = 4:$

$\text{same subproblem}$

$\text{height} = m + n$
The number of distinct LCS subproblems for two strings $m$ and $n$ is only $mn$.

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

Computing the length of an LCS

- The sequences are $X = <A, B, C, B, D, A, B>$ and $Y = <B, D, C, A, B, A>$

<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>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>$x_i$</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>$A$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>$B$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>$C$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td>$D$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>5</td>
<td>$A$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>6</td>
<td>$B$</td>
<td>0</td>
<td>1</td>
<td>2</td>
<td>3</td>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Time = $\Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space = $\Theta(mn)$.

Constructing an LCS

- The procedure takes time $O(m + n)$.
<s>Memoization algorithm</s>

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

MEMO-LCS(X,Y,i,j)
1 if c[i,j] = NIL
2 then if x[i] = y[j]
3 then c[i,j] ← MEMO-LCS(X,Y,i-1,j-1)+1
4 else c[i,j] ← max{MEMO-LCS(X,Y,i-1,j), MEMO-LCS(X,Y,i,j-1)}
5 return c[i,j]

Computing the length of LCS

MEMOIZED-LCS(X,Y)
1 for i ← 1 to m
2 do for j ← i to n
3 do c[i,j] ← NIL
4 return MEMO-LCS(X,Y,m,n)

• Assuming that initially ∀i,j: c[i,j] = NIL, to get the length of LCS of X and Y, MEMO-LCS(X, Y, length[X], length[Y]) should be called.

Components of DP Algorithms

• Elements of dynamic programming
  – Optimal substructure
  – Overlapping subproblems

• Optimal substructure varies across problem domains in two ways:
  – How many subproblems are used in an optimal solution to the original problem
  – How many choices we have in determining which subproblem(s) to use in an optimal solution.

• Optimal substructure
  – Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
  – Suppose that you are given this last choice that leads to an optimal solution.
  – Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
  – Show that the solution to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
    » Suppose that one of the subproblem solutions is not optimal
    » Cut it out
    » Paste in an optimal solution
    » Get a better solution to the original problem. Contradicts optimality of problem solution.

• Elements of DP Algorithms
  – Optimal substructure
  – Overlapping subproblems

Components of DP Algorithms

• Elements of dynamic programming
  – Optimal substructure
  – Overlapping subproblems

• Optimal substructure varies across problem domains in two ways:
  – How many subproblems are used in an optimal solution to the original problem
  – How many choices we have in determining which subproblem(s) to use in an optimal solution.

• Optimal substructure
  – Show that a solution to a problem consists of making a choice, which leaves one or subproblems to solve.
  – Suppose that you are given this last choice that leads to an optimal solution.
  – Given this choice, determine which subproblems arise and how to characterize the resulting space of subproblems.
  – Show that the solution to the subproblems used within the optimal solution must themselves be optimal. Usually use cut-and-paste:
    » Suppose that one of the subproblem solutions is not optimal
    » Cut it out
    » Paste in an optimal solution
    » Get a better solution to the original problem. Contradicts optimality of problem solution.
• That was optimal substructure.
  – Need to ensure that you consider a wide enough range of choices and subproblems that you get them all. Try all the choices, solve all the subproblems resulting from each choice, and pick the choice whose solution, along with subproblem solutions, is best.
  – How to characterize the space of subproblems?
    » Keep the space as simple as possible.
    » Expand it as necessary.

• In assembly-line scheduling
  – We had $\Theta(n)$ subproblems overall and only two choices to examine for each, yielding a $\Theta(n)$ running time.
    $$f_i[j] = \begin{cases} e_{1,1} + a_{1,1} & \text{if } j = 1 \\ \min(f_i[1]+a_{i,j}+f_i[j-1]+t_{2,1}+a_{1,j}) & \text{if } j \geq 2 \end{cases}$$

  – $f^* = \min(f_1[n]+x_1, f_2[n]+x_2)$

• For matrix-chain multiplication
  – There were $\Theta(n^2)$ subproblems overall, and in each we had at most $n-1$ choices, giving an $O(n^3)$ running time.
    $$m[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min_{i<k<j} \{ m[i,k] + m[k+1,j] + \overset{p_i,p_{k},p_j}{q,r} \} & \text{if } i < j \end{cases}$$

• Does longest path have optimal substructure?
  – It seems like it should.
  – It does not.

• Consider $q \rightarrow r \rightarrow t = \text{longest path } q \rightarrow t$. Are its subpaths longest paths? No!
  – Subpath $q \rightarrow r$ is $q \rightarrow r$.
  – Longest simple path $q \rightarrow r$ is $q \rightarrow s \rightarrow t \rightarrow r$.
  – Subpath $r \rightarrow t$ is $r \rightarrow t$.
  – Longest simple path $r \rightarrow t$ is $r \rightarrow q \rightarrow s \rightarrow t$. 
Optimal substructure

• Not only isn’t there optimal substructure, but we can’t even assemble a legal solution from solutions to subproblems.
• Combine longest simple paths:
  \[ q \rightarrow s \rightarrow t \rightarrow r \rightarrow q \rightarrow s \rightarrow t \]
• Not simple!
• In fact, this problem is NP-complete (so it probably has no optimal substructure to find)

Optimal substructure

• What’s the big difference between shortest path and longest path?.
  – Shortest path has independent subproblems.
  – Solution to one subproblem does not affect solution to another subproblem of the same problem.
  – Longest simple path: subproblems are not independent.
  – Consider subproblems of longest simple paths: \( q \rightarrow r \) and \( r \rightarrow t \).
  – Longest simple path \( q \rightarrow r \) uses \( s \) and \( t \).
  – Cannot use \( s \) and \( t \) to solve longest simple path \( r \rightarrow t \), since if we do, the path isn’t simple.
  – But we have to use \( t \) to find longest simple path \( r \rightarrow t \).
  – Using resources (vertices) to solve one subproblem renders them unavailable to solve the other subproblem.