# Dynamic Programming

Alan Pilecki & Chad Maheu

## Outline

- Introduction to Dynamic Programming
- When is it used?
- Bioinformatics
- Examples in Bioinformatics
- Longest Common Subsequence
- Analysis of LCS

## Compared With Divide-and-conquer

- Dynamic Programming solves every subproblem just once and then saves its answer in a table.
- More efficient than Divide-and-conquer.
- Requires more memory for the table though.

## Typical Use

- Optimization Problems
  - Each solution has a value, we wish to find a solution with the optimal (max or min) value.

## Steps

1. Characterize the structure of an optimal solution.
2. Recursively define the value of an optimal solution in terms of the optimal solutions to subproblems.
3. Compute the value of an optimal solution in a bottom-up fashion.
4. Construct an optimal solution from the computed information.

## When it is used?

- When your problem exhibits:

1. Optimal Substructure
2. Overlapping Subproblems

## Optimal Substructure (1)

- Optimal substructure is the property stating that an optimal solution to a problem contains within it an optimal solution to subproblems.
- The solution to the problem consists of making a choice.
- Suppose you are given the choice that leads to an optimal solution.

## Optimal Substructure (2)

- You determine which subproblems to ensue.
- Show that the solutions to the subproblems used within the optimal solutions to the problem must themselves be optimal.

## Overlapping Subproblems

- A potential recursive algorithm will visit the same problem multiple times.
- Solutions to subproblems stored in a table with constant time lookup.
- Choices made are also stored in a table

## Bioinformatics

- The science of managing and analyzing biological data using advanced computing techniques.
- Rapidly growing field.

## Examples

- Analysis of Protein structures
- Determining Molecular structure
- Analyzing DNA sequences (Human Genome Project)

## Longest Common Subsequence

## Subsequence

- Subsequence - a substring of the sequence that maintains order but not necessarily consecutively
- Formally:

  $Z=<z_1,z_2,\ldots,z_k>$ is a subsequence of $X=<x_1,x_2,\ldots,x_m>$ if there exists a strictly increasing sequence $<i_1,i_2,\ldots i_k>$ of indices of X such that for all $j=1,2,\ldots k$ we have $x_{i_j}=z_j$

## Common Subsequence

- Common Subsequence – a subsequence Z such that Z is a subsequence of X and Y
- Example: In X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>, <B,C,A> is a common subsequence.
- For X, a sequence is $<i_1,i_2,i_3> = <2,3,6>$
- For Y, a sequence is $<i_1,i_2,i_3> = <1,3,4>$

## Longest Common Subsequence

- Since there is a longer subsequence in X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>, <B,C,A> is not the longest common subsequence.
- <B,C,B,A> is the longest subsequence of both X and Y and is of length 4
- Dynamic programming is used here since we need to find the optimal solution

## Characterizing an LCS

Theorem – Optimal Structure of an LCS

Let $X=<x_1,x_2,\ldots,x_m>$ and $Y=<y_1,y_2,\ldots,y_n>$

Let $Z=<z_1,z_2,\ldots,z_k>$ be an LCS of *X* and *Y*

1. If $x_m=y_n$ then $z_k=x_m=y_n$ and $z_{k-1}$ is an LCS of $x_{m-1}$, $y_{n-1}$
2. If $x_m \neq y_n$ then if $z_k \neq x_m$ implies that *z* is an LCS of $x_{m-1}, y_n$
3. If $x_m \neq y_n$ then if $z_k \neq y_n$ implies that *z* is an LCS of $x_m, y_{n-1}$

## Characterizing an LCS

- In plain English, if the last elements of the sequence match, that value is the last element of LCS.
- If the last elements of the sequence do not match, then each sequence must be compared to the other sequence disregarding that other sequences last element.

## Recursive Solution

- The problem lends itself to a recursive solution.
- c[i,j]=the length of the LCS of $x_i$ and $y_j$
  - c[i,j]=0 if i=0 or j=0
  - c[i,j]=c[i-1,j-1] + 1 if i,j > 0 and $x_i=y_j$
  - c[i,j]=max(c[i,j-1],c[i-1,j] if i,j > 0 and $x_i \neq y_j$
- The values stored in c are the results of the subproblems.

## Compute the Length of an LCS

```
LCS-LENGTH(X, Y)
m=length(X)
n=length(Y)
for i←0 to m
     c[i,0]←0
for j←0 to n
     c[0,j]←0
for i←1 to m
      for j←1 to n
              if (xi=yj) then
                          c[i,j]←c[i-1,j-1]+1
                          b[i,j]←"↖"
              else if c[i-1,j]≥c[i,j -1] then
                          c[i,j]←c[i-1,j]
                          b[i,j]←"↑"
              else
                          c[i,j]←c[i,j-1]
                          b[i,j]←"←"
return c and b
```

## Compute the Length of an LCS

- The resulting two tables contain all the information about the LCS of X and Y
- The table b is used to construct the value of the LCS of X and Y
- Table c is used to find the length of the LCS of X and Y

## Constructing the LCS

- To find the LCS, you start at b[m,n] and trace back through the arrows.
- When you encounter a "↖" at some b[i,j], it means that $x_i = y_j$ and is an element of the LCS
- Of course, this algorithm finds the LCS starting at the end

## Constructing the LCS

```
PRINT-LCS(b,X,i,j)
if i=0 or j=0 then
   return
if b[i,j]= "↖" then
   PRINT-LCS(b,X,i-1,j-1)
   print xi
else if b[i,j] = "↑" then
   PRINT-LCS(b,X,i-1,j)
else // b[i,j] = "←"
   PRINT-LCS(b,X,i,j-1)
```

## LCS Example

- X=<A,B,C,B,D,A,B> and Y=<B,D,C,A,B,A>
- Find the length of the LCS and the LCS itself

j

| | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| | | | $y_j$ | B | D | C | A | B | A |
| | 0 | $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 | A | 0 | | | | | | |
| | 2 | B | 0 | | | | | | |
| i | 3 | C | 0 | | | | | | |
| | 4 | B | 0 | | | | | | |
| | 5 | D | 0 | | | | | | |
| | 6 | A | 0 | | | | | | |
| | 7 | B | 0 | | | | | | |

| | | j | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| | | $y_j$ | B | D | C | A | B | A |
| | 0 $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| | 2 B | 0 | | | | | | |
| i | 3 C | 0 | | | | | | |
| | 4 B | 0 | | | | | | |
| | 5 D | 0 | | | | | | |
| | 6 A | 0 | | | | | | |
| | 7 B | 0 | | | | | | |

| | | j | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
| | | $y_j$ | B | D | C | A | B | A |
| | 0 $x_i$ | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| | 1 A | 0 | ↑ 0 | ↑ 0 | ↑ 0 | ↖ 1 | ← 1 | ↖ 1 |
| | 2 B | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 1 | ↖ 2 | ← 2 |
| i | 3 C | 0 | ↑ 1 | ↑ 1 | ↖ 2 | ← 2 | ↑ 2 | ↑ 2 |
| | 4 B | 0 | ↖ 1 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ← 3 |
| | 5 D | 0 | ↑ 1 | ↖ 2 | ↑ 2 | ↑ 2 | ↑ 3 | ↑ 3 |
| | 6 A | 0 | ↑ 1 | ↑ 2 | ↑ 2 | ↖ 3 | ↑ 3 | ↖ 4 |
| | 7 B | 0 | ↖ 1 | ↑ 2 | ↑ 2 | ↑ 3 | ↖ 4 | ↑ 4 |

## LCS Example

- Resulting answer is an LCS of length 4 because c[m,n] = 4
- LCS of <B,D,C,A,B,A> and <A,B,C,B,D,A,B> is <B,C,B,A>

## Analysis of LCS

- Without using Dynamic Programming it was $O(2^n)$, where n is the length of one of the sequences.
- Time of $2^n$ is needed to construct every subsequence of the sequence.

## Analysis of LCS

- Building the table c and b both requires O(mn), where m and n are the lengths of the two sequences and the time to compute each table entry is O(1)
- Retrieving the sequence from table b only requires O(n+m) since at each stage either i, j, or both are decremented.

## Improvements to LCS

- To save space, table b does not have to be constructed. Instead comparisons with elements in table c can allow the LCS to be constructed.
- The space saved is only $\Theta(mn)$, which is not an asymptotical decrease.

## Improvements to LCS

- Space can be reduced asymptotically by optimizations to table c.
- Since only two rows are being compared at a time, table c only has to consist of two rows.
- Disadvantage of this is that the LCS cannot be reconstructed from this information.

## References


- Our textbook, *Introduction to Algorithms, 2nd Edition.*
- *http://amber.cs.umd.edu/class/838-s04/articles.html*
- http://www.scs.carleton.ca/~nussbaum
- http://ranger.uta.edu/~cook/aa/lectures/applets/lcs/lcs.html


## Questions?