Cosc 201
Algorithms and Data Structures
Lecture 22 (14/5/2025)
Dynamic Programming 2

Brendan McCane
brendan.mccane@otago.ac.nz

UNIVERSITY OF OTAGO
Te Whare Wānanga o Otāgo
NEW ZEALAND

String similarity

Let's consider two strings:

- $S_1 = \text{ACCGGTCGAGTGCGCGG}$
- $S_2 = \text{GTCGTTCGGAATGCC}$

Can we come up with some notion of how close these two strings are when the order of the letters is important?

Notions of similarity

- the longest substring common to both strings
- the number of changes to convert one string into another (this is called edit distance)
- the longest string, S_3, such that the characters in S_3 occur in both S_1 and S_2 in the same order, but not necessarily consecutively. This measure is called the longest common subsequence (LCS).

What is the LCS of:

$$S_1 = ACCGGTCGAGTGCGCGG$$
$$S_2 = GTCGTTCGGAATGCC$$

$$S_3 = GGTCGGTGCC$$

Longest common subsequence

Let $X = [x_1, x_2, ..., x_m]$ and $Y = [y_1, y_2, ..., y_n]$ be two strings. Further, let $Z = [z_1, z_2, ..., z_k]$ be an LCS of X and Y. Then the following statements hold:

1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$ then either:
   2.1 Z is an LCS of X_{m-1} and Y; or
   2.2 Z is an LCS of X_m and Y_{n-1}.

This gives us a very clear optimal substructure problem from which we can easily write a naive recursive solution. If we think about the length of the LCS, then the length can be defined as:

$$l_{i,j} = \begin{cases} 0, & \text{if } i = 0 \text{ or } j = 0, \\ l_{i-1,j-1} + 1, & \text{if } x_i = y_j, \\ \max(l_{i,j-1}, l_{i-1,j}), & \text{otherwise} \end{cases}$$

Naive algorithm

```
1:  function RecursiveLCS(X, Y)
2:      if X.length=0 or Y.length=0 then
3:          return ""
4:      end if
5:      if X[m]=Y[n] then
6:          return RecursiveLCS(X[1:m-1],Y[1:n-1])+X[m]
7:      end if
8:      lcs1 ← RecursiveLCS(X[1:m-1],Y)
9:      lcs2 ← RecursiveLCS(X, Y[1:n-1])
10:     if lcs1.length>lcs2.length then
11:         return lcs1
12:     else
13:         return lcs2
14:     end if
15: end function
```

Once again, the naive algorithm is exponential, so we have to memoize.

Memoized version

Initialise memo array to be full of -1.

```
1:  function MemoLCSRecurse(X, Y)
2:      if m = 0 or n = 0 then return ""
3:      end if
4:      if memo[m,n]≠"-1" then return memo[m,n]
5:      end if
6:      if X[m]=Y[n] then
7:          memo[m,n]← MemoLCSRecurse(X[1:m-1],Y[1:n-1])+X[m]
8:      else
9:          l1 ← MemoLCSRecurse(X[1:m-1],Y)
10:         l2 ← MemoLCSRecurse(X,Y[1:n-1])
11:         if l1.length>l2.length then
12:             memo[m,n] = l1
13:         else
14:             memo[m,n] = l2
15:         end if
16:     end if
17:     return memo[m,n]
18: end function
```

Memo table

For the strings:

$$s1 = "ACCGGTCGAGTGC"$$
$$s2 = "GTCGTTCGGAAT"$$

		G	T	C	G	T	T	C	G	G	A	A	T	
A											A	-1	-1	
C				C	C	C	C	C	C	C	C	-1	-1	
C				C	C	C	C	CC	CC	CC	CC	-1	-1	
G		G	G	G	CG	CG	CG	CG	CCG	CCG	CCG	-1	-1	
G		G	G	G	GG	GG	GG	GG	CGG	CCGG	CCGG	-1	-1	
T		G	GT	GT	GT	GGT	GGT	GGT	GGT	CCGG	CCGG	-1	-1	
C		G	GT	GTC	GTC	GTC	GTC	GGTC	GGTC	GGTC	GGTC	-1	-1	
G		G	GT	GTC	GTCG	GTCG	GTCG	GTCG	GGTCG	GGTCG	GGTCG	-1	-1	
A		G	GT	GTC	GTCG	GTCG	GTCG	GTCG	GGTCG	-1	GGTCGA	GGTCGA	-1	
G		G	GT	GTC	GTCG	GTCG	GTCG	GTCG	GTCGG	GGTCGG	GGTCGG	GGTCGG	-1	
T		-1	GT	GTC	-1	GTCGT	GTCGT	GTCGT	GTCGT	GGTCGG	GGTCGG	GGTCGG	GGTCGGT	
G		-1	-1	-1	GTCG	GTCGT	GTCGT	-1	GTCGTG	GTCGTG	GTCGTG	GTCGTG	GGTCGGT	
C		-1	-1	-1	-1	-1	-1	GTCGTC	GTCGTC	GTCGTC	GTCGTC	GTCGTC	GGTCGGT	

Iterative Version

- The memoized recursive version is fairly simple and efficient, but the strings do not need to be very long before we will blow the system stack.
- If we're comparing DNA for example, then we could expect strings millions or billions of characters long.
- Even with an iterative version, billions of characters will cause a problem. Why?
- Nevertheless, an iterative version will work for much larger problems than a recursive one.
- Once again, the iterative algorithm runs forwards from the smallest array indices to the largest, and fills up the whole array (compare to recursive backwards algorithm).

Iterative Algorithm

```
1: function ITERATIVELCS(X,Y)
2:     memo ← array of size m + 1 × n + 1
3:     for i ← 1 to m do
4:         for j ← 1 to n do
5:             if X[i]=Y[j] then
6:                 memo[i+1,j+1] ← memo[i,j]+X[i]
7:             else
8:                 if memo[i,j+1].len>memo[i+1,j].len then
9:                     memo[i+1,j+1] ← memo[i,j+1]
10:                else
11:                    memo[i+1,j+1] ← memo[i+1,j]
12:                end if
13:            end if
14:        end for
15:    end for
16:    return memo[m,n]
17: end function
```

Filled array

For the strings:

$$s1 = "ACCGGTCGAGTGC"$$
$$s2 = "GTCGTTCGGAAT"$$

		G	T	C	G	T	T	C	G	G	A	A	T	
A											A	A	A	
C				C	C	C	C	C	C	C	A	A	A	
C				C	C	C	C	CC	CC	CC	CC	CC	CC	
G		G	G	C	CG	CG	CG	CC	CCG	CCG	CCG	CCG	CCG	
G		G	G	C	CG	CG	CG	CC	CCG	CCGG	CCGG	CCGG	CCGG	
T		G	GT	GT	CG	CGT	CGT	CGT	CCG	CCGG	CCGG	CCGG	CCGGT	
C		G	GT	GTC	GTC	CGT	CGT	CGTC	CGTC	CCGG	CCGG	CCGG	CCGGT	
G		G	GT	GTC	GTCG	GTCG	GTCG	CGTC	CGTCG	CGTCG	CGTCG	CGTCG	CCGGT	
A		G	GT	GTC	GTCG	GTCG	GTCG	CGTC	CGTCG	CGTCG	CGTCGA	CGTCGA	CGTCGA	
G		G	GT	GTC	GTCG	GTCG	GTCG	CGTC	CGTCG	CGTCGG	CGTCGA	CGTCGA	CGTCGA	
T		G	GT	GTC	GTCG	GTCGT	GTCGT	GTCGT	CGTCG	CGTCGG	CGTCGA	CGTCGA	CGTCGAT	
G		G	GT	GTC	GTCG	GTCGT	GTCGT	GTCGT	GTCGTG	CGTCGG	CGTCGA	CGTCGA	CGTCGAT	
C		G	GT	GTC	GTCG	GTCGT	GTCGT	GTCGTC	GTCGTG	CGTCGG	CGTCGA	CGTCGA	CGTCGAT	

Genetics Example

Triticum aestivum HMW-glutenin subunit Glu-1Stx1 (Glu-1) gene

ATGGCTAAGCGGCTGGTCCTCTTTGCAGCAGTAGCCGTCGCCCTCGTGGCTCTCACCGCCGCTGAAGGTGAGGCCTCTGGGCAACTACAGTGT

GAGCGCGGGCTCCAGGAGCGCGAGCTCGAGGCGTGCCGACAGGTCGTGGACCAGCAGCTCCGAGACGCTAGCCCCGAGTGCCGCCCCGTCG

CCGTCAGCCCGGTCGCGAGACAATACGAGCAGCAAACCGTGGTGCCGCCCAAGGGCGGATCCTTCTACCCCGGCGAGACCACGCCACCGCAG

CAACTCCAACAAAGAATATTTTGGGGAATACCTACACTACTAAGAAGGTATTACCCAAGTGTAACTTCTCCGCGGCAGGGGTCATACTATCCAGGC

CAAGCTTCTCCGCAACGGCCAGGACAAGGACAGCAACCAGGACAAGGACAGCAGCCAGGACAAGGACAGCAACCAGGACAAGGGCAACAAAGT

CAGCAGCCAGGACAAGGGCAACATCCAGGACAAGGACAACAAGGGTACTACCCAACTTCTCCACAACAGCCAGGACAAGGGCAACAGCCAGGA

CAAGGGCAACAACCGGGAGAAGGACAACCAGGGTACTACCCAACTTCTCCACAGCAGCCAGGACAAGGGCAACAGCCAGGACAAGGGCAACCA

GGGTACTACCCAACTTCTTCGCAGCAGCCAGGACAAGGACAGCAGCCAGGACAAGGGCAGCAACCAGGACAAGGGCAACAAGGTCAGCAACCA

GGACAAGGGCAACAACCAGGACAAGGACAACAAGGGTACTACCCAACTTCTCCGCAACAGCTAGGACAAGGGCAACAGCCAGGACAATGGCAA

CAACCGGGACAAGGGCAACCAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGGGCAACCAGGGTACTACCCAACTTCTCCGCAGCAG

CCAGGACAATTGCAACAACCAGCACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGAGCAACAAGGGTACTACCCAA

CTTCTCCGCAGCAGCCAGGACAAGGGCAACAGCCAGGACAATGGCAACAACCAGGACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGC

AGCCAGGACAAGGGCAACAGCCAGGACAATGGCTGCAACCAGGACAAGAGCAACAGGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAG

GGCAACAGGCAGGACATGGGCAACAGCCAGGACAATGGCTCCAACCAGGACAAGGGCAACAAGGGTACTACCCAACTTCCCTGCAGCAGTCAG

GACAAGGGCAGCAATCAGGGCAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGTCAGGACAAGGGCAACAAGGCTACGACAGCCCAT

ACCATGTTAGCGCGGAGCACCAGGCGGCCAGCCTAAAGGTGGCAAAGGCACAGCAGCTCGCGGCACAGCTGCCGGCAATGTGCCGGCTGGAG

Genetics example 2

Thinopyrum intermedium HMW gluten subunit (1Aix3) gene

ATGGCTAAGCGGTTGGTCCTCTTTGCAGCAGTAGTCGTCGCCCTCGTGGCTCTCACCGCCGCTGAAGGTGAGGCCTCTGGGCAACTACAGTGTG

AGCGCGAGCTCGAGGCGTGCCGACAGGTCGTGGACCAGCAGCTCCGAGACGCTAGCCCCGAGTGCCGCCCCGTCGCCGTCAGCCCAGTCGCG

AGACAATACGAGCAGCAAACCGTGGTGCCGCCCAAGGGCGGATCCTTCTACCCCGGCGAGGCCACGCCACCGCAGCAACTCCAACAAAGAATA

TTTTGGGGAATACCTACACTACTAAGAAGGTATTACCCAAGTGTAACTTCTCCGCGGCAGGGGTCATACTATCCAGGCCAAGCTTTTCCGCAACG

GCCAGGACAAGGACAGCAACCAGGACAAGGGCAGCAACCAGGACAAGGGCAACAAAGTCAGCAGCCAGGACAAGGGCAACATCCAGGACAAG

GACAACAAGGGTACTACCCAACTTCTCCACAACAGCCAGGACAAGGGCAACAGCCAGGACAAGGGCAACAACCGGGAGAAGGGCAACCAGGGT

ACTACCCAACTTCTCCACAGCAGCCAGGACAAGGGCAACAACCAGGACAAGGGCAACCAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACA

AGGGCAGCAGCCAGGACAAGGACAGCAGCCAGGACAAGGGCAGCAACCAGGACAAGGGCAACAAGGTCAGCAACCAGGACAAGGGCAACAAC

CAGGACAAGGACAACAAGGGTACTACCCAACTTCTCCGCAACAGTTAGGACAAGGGCAACAGCCAGGACAATGGCAACAACCGGGACAAGGGC

AACCAGAGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGGGCAACCAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAATTGCAACA

ACCAGCACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGAGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGCCA

GGACAAGGGCAGCAGCCAGGACAATGGCAACAACCAGGACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGGGCAA

CAGCCAGGACAAGGGCTGCAACCAGGACAAGGGCAACAAGGGTACTACCCAACTTCTCCACAACAGCCAGGACAAGGGCAACGGCCAGGACAT

GGGCAACAGCCAGGACAATGGCTCCAACCAGGACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGTCAGGACAAGGGCAGCAATCA

GGACAAGGGCAACAAGGGTACTACCCAACTTCTCCGCAGCAGCCAGGACAAGGGCAACAAGGCTACGACAGCCCATACCATGTTAGCGCGGAG

CACCAGGCGGCCAGCCTAAAGGTGGCAAAGGCACAGCAGCTCGCGGCACAGCTGCCGGCAATGTGCCGGCTGGAGGGCGGCGACGCATTGTC

Efficiency

- Memo recurse took 164 milliseconds.
- Iterative DP took 62.9 milliseconds.
- I gave up on the naive recurse after 3 hours.
- For iterative DP, we can do better than this in terms of space efficiency.
- There is no need to keep the whole array in memory at one time
- We just need to reference the current row and the previous row.
- That means instead of space with $O(mn)$, we can use only $O(2n)$.

Next time

Edit distance and dynamic programming in general.