Dynamic Programming: Longest Common Subsequence

Chapter 15.4

Subsequences

- A subsequence is a subcollection of the elements of a sequence, taken in the order they appear in the sequence.
  - For instance, if the sequence is ABCDEA, then ACDA is a subsequence (ABCDEA), but AEC is not.
  - One can also think of a subsequence as the original sequence with some elements crossed out: ABCDEA
- It is easiest to think of the sequences as strings, but in reality, any sequences can be used, such as sequences of integers or real numbers or customer records.
  - 4 15 24 14 19 33 8 17 has a subsequence 24 19 8 17.

Longest Common Subsequence

- The Longest Common Subsequence (LCS) problem is to find a longest sequence that is a subsequence of two given sequences.
- For example, suppose we have two DNA strands encoded as

X=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA

and

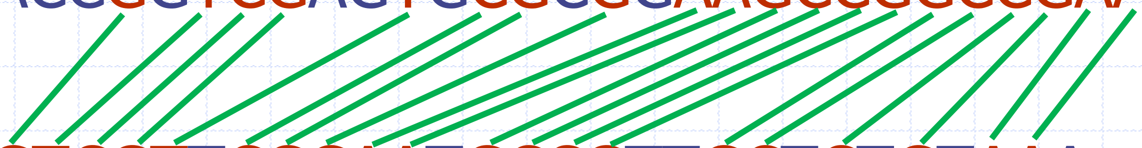
Y=GTCGTTCGGAATGCCGTTGCTCTGTAAA

The length of the LCS is then a measure of the similarity of the strands.

Longest Common Subsequence

- For these strings,

  X=ACCGGTCGAGTGCGCGGAAGCCGGCCGAA



  Y=GTCGTTCGGAATGCCGTTGCTCTGTAAA

  an LCS is GTCGTCGGAAGCCGGCCGAA.

- For X = SPRINGTIME and Y = PIONEER, an LCS is PINE (SPRINGTIME and PIONEER).

Structure of an LCS

If $X = \langle x_1 x_2 \ldots x_m \rangle$ is a sequence, let X_i denote the i^{th} prefix of X. In notation, $X_i = \langle x_1 x_2 \ldots x_i \rangle$.

Theorem:

Let $X = \langle x_1 x_2 \ldots x_m \rangle$ and $Y = \langle y_1 y_2 \ldots y_n \rangle$ be sequences, and let $Z = \langle z_1 z_2 \ldots z_k \rangle$ be any LCS of X and Y.

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 $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Structure of an LCS

Proof:

(1) If $z_k \neq x_m$ we could append $x_m = y_n$ to Z. $\rightarrow\leftarrow$
If Z_{k-1} were not an LCS of X_{m-1} and Y_{n-1}, we could replace it by an LCS W of X_{m-1} and Y_{n-1}; then Wz_k is longer than Z. $\rightarrow\leftarrow$

(2) If $z_k \neq x_m$ then Z is a common subsequence of X_{m-1} and Y. If Z were not an LCS of X_{m-1} and Y, we could replace Z by an LCS W of X_{m-1} and Y; then W (which is longer than Z) is a (longest) common subsequence of X and Y. $\rightarrow\leftarrow$

(3) Symmetric to (2).

DP properties of LCS

- The theorem shows that LCS has the optimal-substructure property. The LCS of X and Y involves the LCS of prefixes of X and Y.
- LCS also has the overlapping subproblems property: In determining an LCS for X_i and Y_{j-1}, we may use (case 2) the LCS for X_{i-1} and Y_{j-1}. But also, in determining an LCS for X_{i-1} and Y_j, we may also use (case 3) the LCS for X_{i-1} and Y_{j-1}.
- Thus, differing subproblems make use of the same subproblem.

Recursive formulation for the length of an LCS

Consider an LCS of X and Y.

Let c(i, j) be the length of an LCS of sequences X_i and Y_j. Then:

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

The first case is the basis. The second case is case (1) of the theorem. The third case is the combination of cases (2) and (3) of the theorem.

Formulation as code

```
LCS(X, Y) {
    return c(m, n);
}

c(i, j) {
    if (i=0 or j=0)
        return 0;
    if (X[i] = Y[j])
        return 1 + c(i-1, j-1);
    return max(c(i, j-1), c(i-1, j));
}
```

Analysis:

let T(s) be time for c(i, j) where s = i+j

- if (i=0 or j=0): O(1)
- return 0;: O(1)
- if (X[i] = Y[j]): O(1)
- return 1 + c(i-1, j-1);: O(1)+T(s-2)
- return max(c(i, j-1), c(i-1, j));: O(1)+2T(s-1)

$$T(s) = O(1) \quad \text{if } s=0$$
$$T(s) = O(1) + \max(T(s-2), 2T(s-1)) \quad \text{if not}$$

$$T(s) \in \Omega(2^s)$$

Memoized

```
LCS(X, Y) {
    allocate matrix memo[0..m,0..n] = ∞;
    return c(m, n);
}
c(i, j) {
    if memo[i, j] ≠ ∞
        return memo[i, j];
    if (i=0 or j=0)
        memo[i, j] = 0;
    else if (X[i] = Y[j])
        memo[i, j] = 1 + c(i-1, j-1);
    else
        memo[i, j] = max(c(i, j-1), c(i-1, j));
    return memo[i, j];
}
```

Memoized analysis

Consider all calls to c(i, j). At most (m+1)(n+1) of them make it past the memo-checking **if** statement. Each of these has the potential to call c() recursively twice. LCS() calls c() once. Thus there are at most 2(m+1)(n+1) + 1 calls of c(i, j).

The nonrecursive work in c(i, j) is O(1), so the total work in c(i, j) is at most (2(m+1)(n+1) + 1)·O(1) = O(mn).

The nonsubroutine work in LCS() is O(1) + matrix allocation work, which is O(mn) because the entries are initialized to something other than 0.

Thus, the total work is the work in LCS() + work in c() = O(mn) + O(mn) = O(mn).

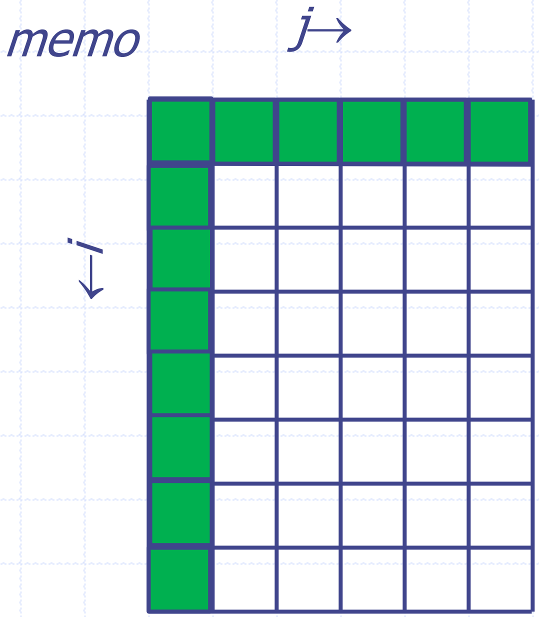
Exercise: One can avoid the O(mn) matrix allocation work by having the memo[i, j] store 1 + length(LCS(X_i,Y_j)) rather than length(LCS(X_i,Y_j)); this allows initialization of memo[] by 0. Write out the pseudocode for this. (Work out before viewing the next slide.)

Memoized Variant

```
LCS(X, Y) {
      allocate matrix memo[0..m,0..n] = 0;
      return c(m, n);
}
c(i, j) {
      if memo[i, j] ≠ 0
         return memo[i, j] - 1;
      if (i=0 or j=0)
         memo[i, j] = 0 + 1;                     // note: written 0+1 rather than 1 for clarity.
      else if (X[i] = Y[j])                      // In code, compiler would clean it up.
         memo[i, j] = 1 + c(i-1, j-1) + 1;
      else
         memo[i, j] = max(c(i, j-1), c(i-1, j)) + 1;
      return memo[i, j] - 1;
}
```

Dynamic Programming

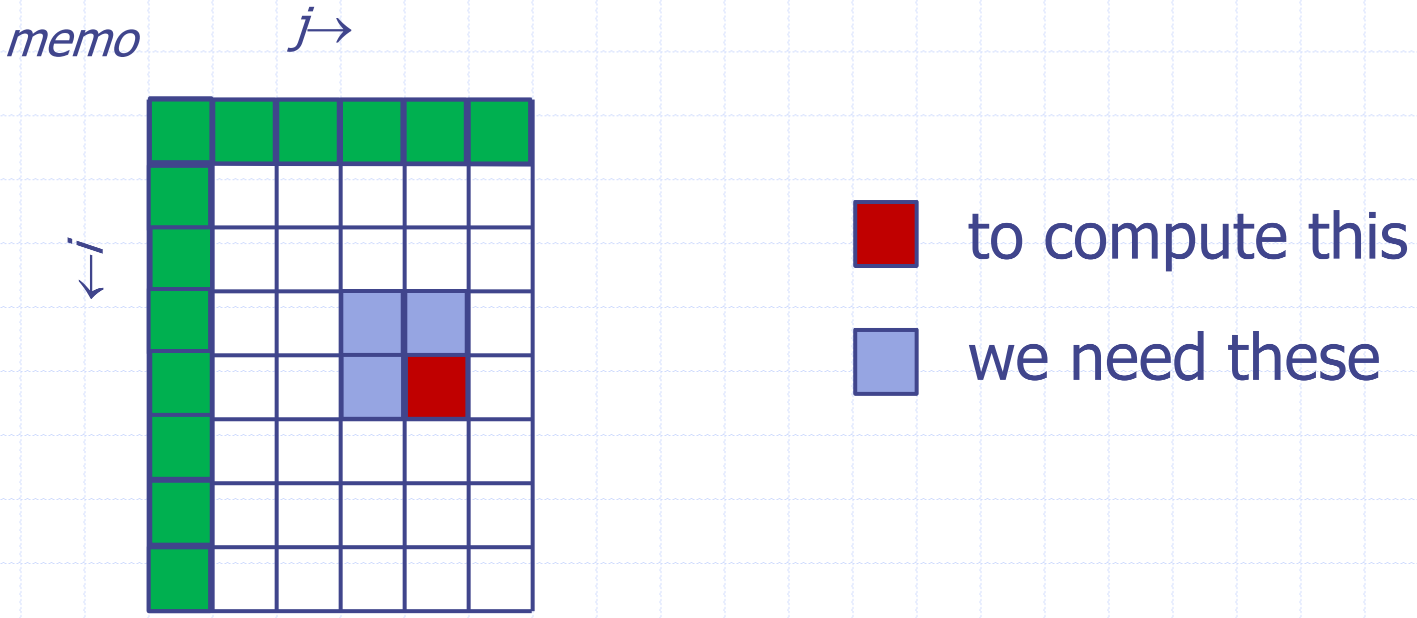
For dynamic programming, we must find the order in which to compute the memo table entries.



We start with the entries that have i=0 or j=0.

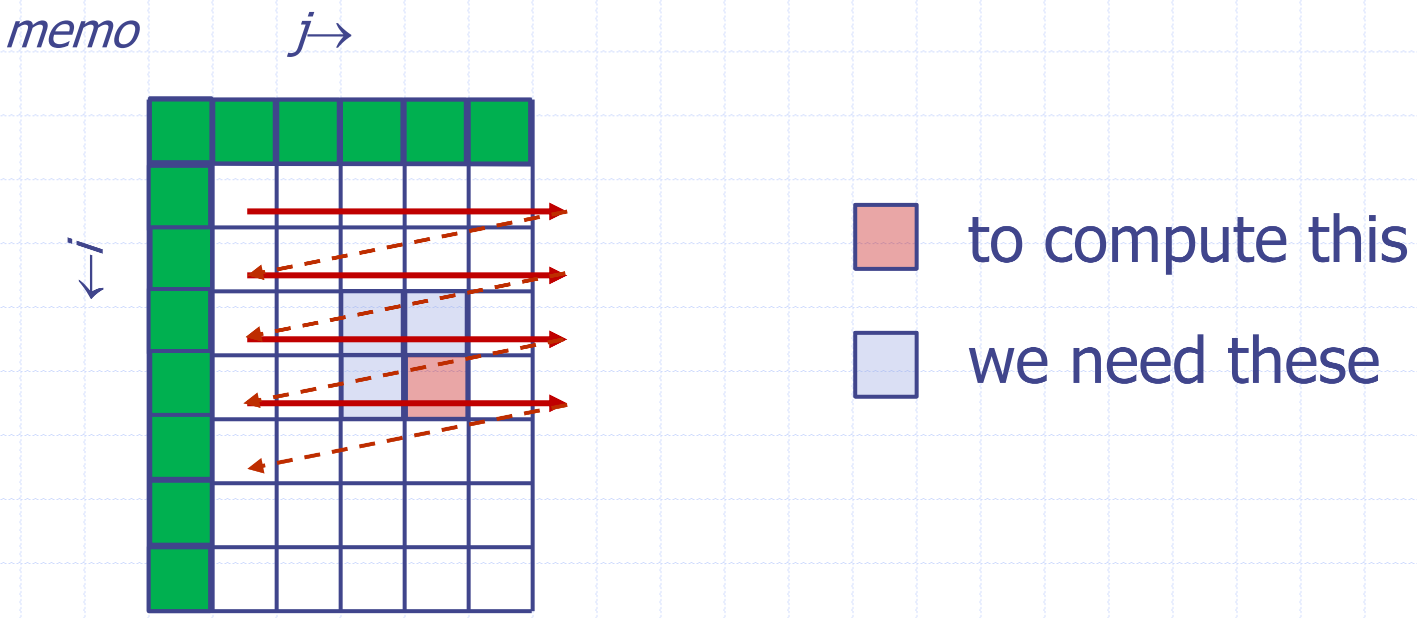
Dynamic Programming

To compute an entry memo[i, j], we may need memo[i-1, j-1], memo[i-1, j], and memo[i, j-1].



Dynamic Programming

A row-major order (or a column-major one) will ensure that we have the entries we need when computing memo[i, j].



Dynamic Programming

```
LCS(X, Y) {
        allocate matrix memo[0..m,0..n] = 0;
        for i = 1 to m
            memo[i, 0] = 0;
        for j = 0 to n
            memo[0, j] = 0;
        for i = 1 to m
            for j=1 to n
                if X[i] = Y[j]
                        memo[i, j] = memo[i-1, j-1] + 1;
                else
                        memo[i, j] = max(memo[i-1, j], memo[i, j-1]);
        return memo[m, n];
}
```

Analysis of DP

```
LCS(X, Y) {
    allocate matrix memo[0..m,0..n] = 0;          O(1)
    for i = 1 to m                                O(m) iterations
        memo[i, 0] = 0;                               O(1) per iteration
    for j = 0 to n                                O(n) iterations
        memo[0, j] = 0;                               O(1) per iteration
    for i = 1 to m                              }
        for j=1 to n                            } O(mn) iterations
            if X[i] = Y[i]                          }
                memo[i, j] = memo[i-1, j-1] + 1;    } O(1) work per
            else                                    } iteration
                memo[i, j] = max(memo[i-1, j], memo[i, j-1]);
    return memo[m, n];                            O(1)
}
```

$O(m) + O(n) + O(mn) + O(1)$

$= O(mn)$

Textbook notes

LCS on the previous slide is the equivalent of *LCS-LENGTH* in the text. *LCS-LENGTH* uses the array c[i, j] instead of memo[i, j] and uses b[i, j] to store the traceback information.

The text's *PRINT-LCS* is the traceback function.

In the section **Improving the code,** they note that the traceback information isn't really needed for optimal-time O(m+n) traceback in this problem.

They also show that one can reduce the memory space to O(min(m, n)) if traceback is not required, by keeping only two rows (or columns, in column-major order) of the table.

Demonstration

		s	p	a	n	k	i	n	g
	0	0	0	0	0	0	0	0	0
a	0								
m	0								
p	0								
u	0								
t	0								
a	0								
t	0								
i	0								
o	0								
n	0								

Demonstration

		s	p	a	n	k	i	n	g
	0	0	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1	1	1
m	0								
p	0								
u	0								
t	0								
a	0								
t	0								
i	0								
o	0								
n	0								

same letter

different letters

Demonstration

		s	*p*	*a*	*n*	*k*	*i*	*n*	*g*
	0	0	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1	1	1
m	0	0	0	1	1	1	1	1	1
p	0								
u	0								
t	0								
a	0								
t	0								
i	0								
o	0								
n	0								

Demonstration

		s	p	a	n	k	i	n	g
	0	0	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1	1	1
m	0	0	0	1	1	1	1	1	1
p	0	0	1	1	1	1	1	1	1
u	0	0	1	1	1	1	1	1	1
t	0	0	1	1	1	1	1	1	1
a	0	0	1	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	2
i	0	0	1	2	2	2	3	3	3
o	0	0	1	2	2	2	3	3	3
n	0	0	1	2	3	3	3	4	4

Demonstration

		s	*p*	*a*	*n*	*k*	*i*	*n*	*g*
	0	0	0	0	0	0	0	0	0
a	0	0	0	1	1	1	1	1	1
m	0	0	0	1	1	1	1	1	1
p	0	0	1	1	1	1	1	1	1
u	0	0	1	1	1	1	1	1	1
t	0	0	1	1	1	1	1	1	1
a	0	0	1	2	2	2	2	2	2
t	0	0	1	2	2	2	2	2	2
i	0	0	1	2	2	2	3	3	3
o	0	0	1	2	2	2	3	3	3
n	0	0	1	2	3	3	3	4	4
			p	*a*			*i*	*n*	