Dynamic programming

- Dynamic programming is an algorithm design technique that is often applied to combinatorial optimization problems where

  (1) an optimal solution can be decomposed into an optimal solution to a subproblem of the same form, and

  (2) a sequential structure can be imposed on a solution.

Eg
- Maximum-Sum Subarray
- Longest Common-Subsequence
- Matrix-Chain Multiplication

Dynamic programming framework

Solving a problem by dynamic programming consists of carrying out 4 steps:

(1) Characterize the recursive structure of an optimal solution.

<u>Aside</u> To characterize the structure of a solution, ask the $\$10^6$ question: "How does an optimal solution end?"

(2) Write a recurrence for the value of an optimal solution.

<u>Aside</u> To derive a recurrence for the solution value, first determine how to describe a subproblem.

(Evaluation phase) (3) Evaluate the recurrence bottom-up in a table.

(Recovery phase) (4) Recover an optimal solution from the table of solution values.

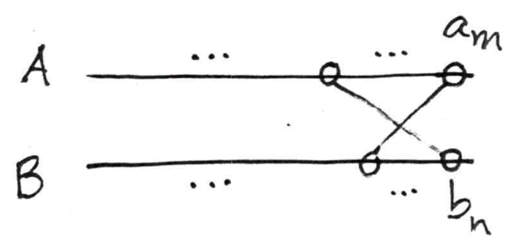
Computing an LCS by dynamic programming

(1) The structure of an LCS

For input strings $A[1:m] = a_1 a_2 \cdots a_m$ and $B[1:n] = b_1 b_2 \cdots b_n$, there are 3 ways an LCS of A and B could end:

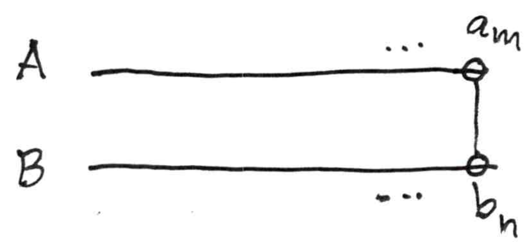
<u>Case 1</u> The LCS of A and B ends by using both characters a_m and b_n :

In this situation, characters a_m and b_n cannot be matched to other characters in B and A in the LCS,



Impossible, since matched char^s must appear in same order in A and B

so we must have



which further implies that we must have $a_m = b_n$.

Case 1 cont'd

Thus for this case the LCS has the form:

$$\underbrace{\begin{matrix} a_1 & a_2 & \cdots & a_{m-1} \\ b_1 & b_2 & \cdots & b_{n-1} \end{matrix}}_{} \quad \begin{matrix} a_m \\ | \\ b_n \end{matrix}$$

Must be an LCS of
A[1 : m-1] and B[1 : n-1]

(since otherwise, replacing the initial portion by the LCS of A[1 : m-1] and B[1 : n-1] would yield a longer solution — a contradiction).

Case 2

The LCS of A and B does not use a_m:

$$\underbrace{\begin{matrix} a_1 & a_2 & \cdots & a_{m-1} & \\ b_1 & b_2 & \cdots & b_{n-1} & b_n \end{matrix}}_{} \quad a_m$$

Must be an LCS of A[1 : m-1] and B[1:n]

Case 3

The LCS does not use b_n:

Symmetric to Case 2, the LCS must be an LCS of A[1 : n] and B[1:m-1]

(2) Recurrence for the value of an LCS

- The recursive subproblem that arises is one of computing an LCS over a prefix of A and a prefix of B.

  This subproblem can be specified by giving the lengths i, j of the prefixes.

- So let

$$L(i,j) := \text{the length of an LCS of } A[1:i] \text{ and } B[1:j].$$

  Then by the 3 cases,

Case 1 --->
Case 2 --->
Case 3 --->

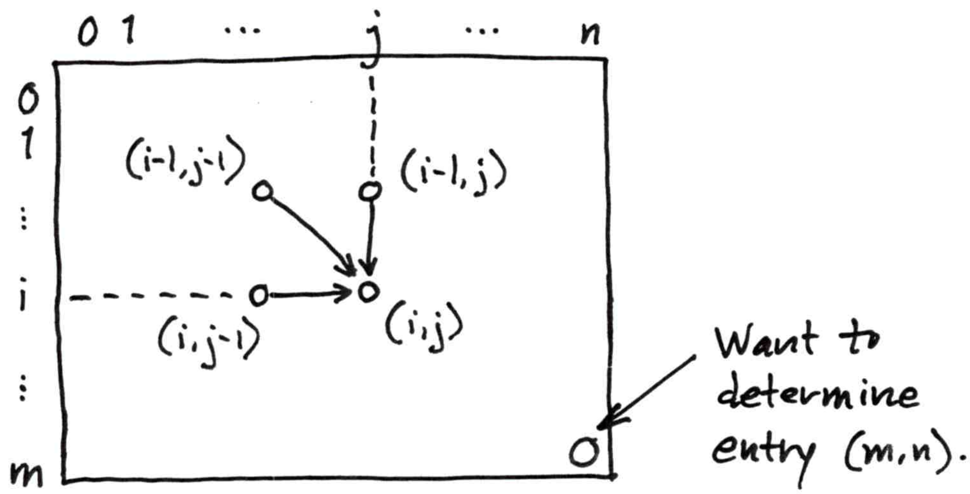
$$L(i,j) = \begin{cases} \max \begin{Bmatrix} L(i-1,j-1)+1 \text{ if } A[i]=B[j], \\ L(i-1,j), \\ L(i,j-1) \end{Bmatrix}, & i \geq 1 \text{ and } j \geq 1; \\ 0, & i \leq 0 \text{ or } j \leq 0. \end{cases}$$

- The solution value for the original problem is

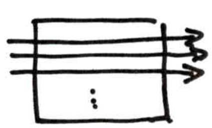
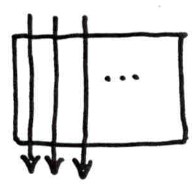
$$L(m,n).$$

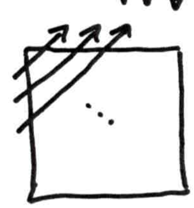
(3) Compute the solution value bottom-up, using a table

- We evaluate $L(i,j)$ in a table $L[0:m,\ 0:n]$.



In general, entry (i,j) depends on the 3 entries $(i-1,j)$, $(i,j-1)$, and $(i-1,j-1)$.

- If we fill in the table in row-major order  or column-major order 

or anti-diagonal-major order , then

these 3 entries will have already been evaluated when evaluating entry (i,j).

```
procedure EvaluateLCS(A, B, L, m, n) begin
                                              · Fill in table L[0:m, 0:n]
                                                for strings A[1:m], B[1:n].
  Θ(m+n) time {
    L[0,0] := 0
    for j := 1 to n do                        · Initialize boundary values
      L[0,j] := 0
    for i := 1 to m do
      L[i,0] := 0
  }
  Θ(mn) time {
    for i := 1 to m do                        · Evaluate L(i,j) by the
      for j := 1 to n do                        recurrence in row-major
                                                order.
        if A[i] = B[j] then
          L[i,j] := max{L[i-1,j-1]+1, L[i-1,j], L[i,j-1]}
        else
          L[i,j] := max{L[i-1,j], L[i,j-1]}
  }
end
```

Analysis

- $\Theta(m+n+mn) = \Theta(mn)$ time (expensive, but tolerable)
- $\Theta(mn)$ space (very costly for long strings)

(4) Recover an LCS from the table of solution values

```
procedure RecoverLCS (A, B, L, i, j) begin
                                        · Outputs the LCS of
                                        A[1:i] and B[1:j] using
                                        the L-table.
   if i ≤ 0 or j ≤ 0 then
      return
   if A[i] = B[j] and L[i,j] = L[i-1, j-1] + 1 then begin
      RecoverLCS (A, B, L, i-1, j-1)
      output A[i]
   end else if L[i,j] = L[i-1, j] then
      RecoverLCS (A, B, L, i-1, j)
   else
      RecoverLCS (A, B, L, i, j-1)
end
```

Analysis

- Each call decrements i or j and expends $\Theta(1)$ time.
- Starting with $i = m$ and $j = n$ takes $\Theta(m+n)$ total time.

(4) Recovering an LCS, cont'd

- Putting it all together, the whole algorithm is,

$$
\left.\begin{array}{l}\Theta(mn) \text{ time} \\ \Theta(mn) \text{ space}\end{array}\right\{
$$

<u>procedure</u> LCS (A, B, m, n) <u>begin</u>

    L := Array(0, m, 0, n)
    Evaluate LCS (A, B, L, m, n)
    Recover LCS (A, B, L, m, n)
    Destroy (L)

<u>end</u>