Dynamic Programming Part 2

CS240 Spring 2022

Rui Fan

Matrix multiplication

$$\begin{bmatrix} 2 & 1 & 0 & 3 \\ 1 & 4 & 4 & 2 \end{bmatrix} \begin{bmatrix} 1 & 2 \\ 3 & 4 \\ 4 & 3 \\ 2 & 1 \end{bmatrix} = \begin{bmatrix} 11 & \boxed{11} \\ 33 & 32 \end{bmatrix}$$

- Let A be a p × q matrix, B be a q × r matrix. Let C = A × B.
  - C is a p × r matrix.
- C_{ij} is the entry in the i'th row and j'th column of C.
  - It's the dot product of the i'th row of A and j'th column of B.

$$C_{ij} = \sum_k A_{ik} \cdot B_{kj}$$

- Computing C takes O(pqr) time.
  - Time counted as number of multiplications.
  - C has pr entries.
  - Each entry formed by computing q products, then summing.

Multiplying many matrices

- Suppose we want to multiply A_1, A_2, A_3, A_4. This can be done in several ways.
  - $(A_1(A_2(A_3A_4)))$.
  - $(A_1((A_2A_3)A_4))$.
  - $((A_1A_2)(A_3A_4))$.
  - …
  - All give the same answer.
    - Because matrix multiplication is associative.
  - But they can take different amounts of time!

Same answer, different costs

- Multiplying a $p \times q$ matrix by a $q \times r$ matrix takes O(pqr) time.
- Let A_1, A_2, A_3 be 3 matrices with dimensions 10×100, 100×5, 5×50.
- Cost of $((A_1A_2)A_3)$ is 7500.
  - 10*100*5=5000 for A_1A_2, producing a 10×5 matrix.
  - Then another 10*5*50=2500 to multiply by A_3.
- Cost of $(A_1(A_2A_3))$ is 75,000.
  - 100*5*50=25000 for A_2A_3, producing a 100×50 matrix.
  - Then another 10*100*50=50000 to multiply by A_1.
- Same answer, but 10 times the cost!

Matrix-chain multiplication problem

- Given a sequence $A_1, A_2, \ldots, A_n$ of matrices, where A_i has dimensions $p_{i-1} \times p_i$, for i=1,…n, compute the product $A_1 \times A_2 \times \cdots \times A_n$ in a way that minimizes the cost.
- $A_1 \times A_2 \times \cdots \times A_n$ has dimensions $p_0 \times p_n$.
  - Same as $(((A_1 \times A_2) \times A_3) \ldots \times A_n)$.
  - $(A_1 \times A_2)$ has dimensions $p_0 \times p_2$.
  - $((A_1 \times A_2) \times A_3)$ has dimensions $p_0 \times p_3$.
  - $(((A_1 \times A_2) \times A_3) \times A_4)$ has dimensions $p_0 \times p_4$.
  - Etc.

Breaking into subproblems

- Suppose we want to multiply $A = A_1 \times A_2 \times A_3 \times A_4 \times A_5 \times A_6 \times A_7 \times A_8$ efficiently.
- Say we first compute $B = A_1 \times A_2 \times A_3 \times A_4$, then $C = A_5 \times A_6 \times A_7 \times A_8$, and then $A = B \times C$.
- B has dimensions $p_0 \times p_4$, C has dimensions $p_4 \times p_8$, so computing $B \times C$ takes $O(p_0 p_4 p_8)$ time.
- Let M(1,4) be the smallest cost to compute $A_1 \times A_2 \times A_3 \times A_4$, and M(5,8) the smallest cost to compute $A_5 \times A_6 \times A_7 \times A_8$.
- Then computing A by breaking it apart into B and C takes $M(1,4) + M(5,8) + O(p_0 p_4 p_8)$ time.
- Since we split A into two parts following A_4, we call this breaking at A_4.

Breaking into subproblems

- Alternatively, we can break at A_3.
- Compute $B' = A_1 \times A_2 \times A_3$, then $C' = A_4 \times A_5 \times A_6 \times A_7 \times A_8$, and then $A = B' \times C'$.
- B' has dimensions $p_0 \times p_3$, C' has dimensions $p_3 \times p_8$, so computing B'×C' takes $O(p_0 p_3 p_8)$ time.
- Let M(1,3) be the smallest cost to compute $A_1 \times A_2 \times A_3$, and M(4,8) the smallest cost to compute $A_4 \times A_5 \times A_6 \times A_7 \times A_8$.
- Then computing A by breaking it apart into B' and C' takes $M(1,3) + M(4,8) + O(p_0 p_3 p_8)$ time.

Breaking into subproblems

- Since there are 8 matrices, there are 7 ways we can break A into subproblems this way.
  - Breaking at A_1 has cost $C_1 = M(1,1) + M(2,8) + p_0 p_1 p_8$.
  - Breaking at A_2 has cost $C_2 = M(1,2) + M(3,8) + p_0 p_2 p_8$.
  - Breaking at A_3 has cost $C_3 = M(1,3) + M(4,8) + p_0 p_3 p_8$.
  - Breaking at A_4 has cost $C_4 = M(1,4) + M(5,8) + p_0 p_4 p_8$.
  - Breaking at A_5 has cost $C_5 = M(1,5) + M(6,8) + p_0 p_5 p_8$.
  - Breaking at A_6 has cost $C_6 = M(1,6) + M(7,8) + p_0 p_6 p_8$.
  - Breaking at A_7 has cost $C_7 = M(1,7) + M(8,8) + p_0 p_7 p_8$.

Breaking into subproblems

- Which split is best?
  - We don’t know.
  - But one of the splits gives the best way to multiply $A_1 \times A_2 \times \cdots \times A_8$.
  - Because no matter how we parenthesize $A_1 \times A_2 \times \cdots \times A_8$, there’s some multiplication that happens last.
    - This corresponds to the split position.
    - E.g. if we parenthesize as $\big(A_1 \times (A_2 \times A_3)\big) \times \Big(\big(A_4 \times A_5\big) \times \big(A_6 \times (A_7 \times A_8)\big)\Big)$, the split occurs after A_3.
- So, the minimum cost M(A) to multiply $A_1 \times A_2 \times \cdots \times A_8$ is the minimum cost from one of the splittings.
- So $M(A) = \min(C_1, C_2, C_3, C_4, C_5, C_6, C_7)$.

Dynamic programming equation

- Let M(i,j) be the smallest time to multiply matrices of $A_i \times A_{i+1} \times \cdots \times A_j$. Then

$$M(1,n) = \min_{1 \le j \le n-1} (M(1,j) + M(j+1,n) + p_0 p_j p_n)$$

Cost of multiplying all the matrices $A_1, \ldots, A_n$.

Choose the best break point j

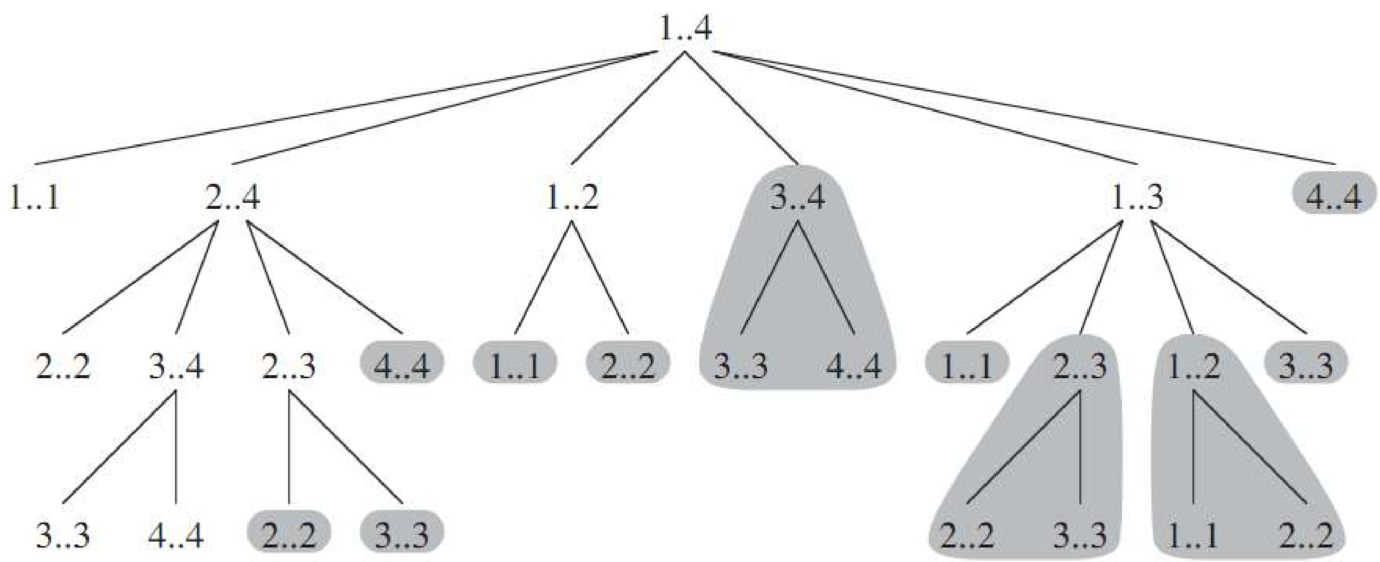
Cost of the first part

Cost of the second part

Cost of multiplying the matrices produced by the first and second parts

DP equation and subproblems

- Solving the main problem M(1,n) requires solving subproblems M(i,j), for $1 \leq i,j \leq n$.
- Solving each M(i,j) in turn requires solving smaller subproblems.
- Work bottom up. Solve smallest subproblems first, then combine the solutions to solutions of bigger subproblems.
- In the base case we have M(i,i)=0, for all i.
  - Because we just have matrix A_i, with no multiplications.

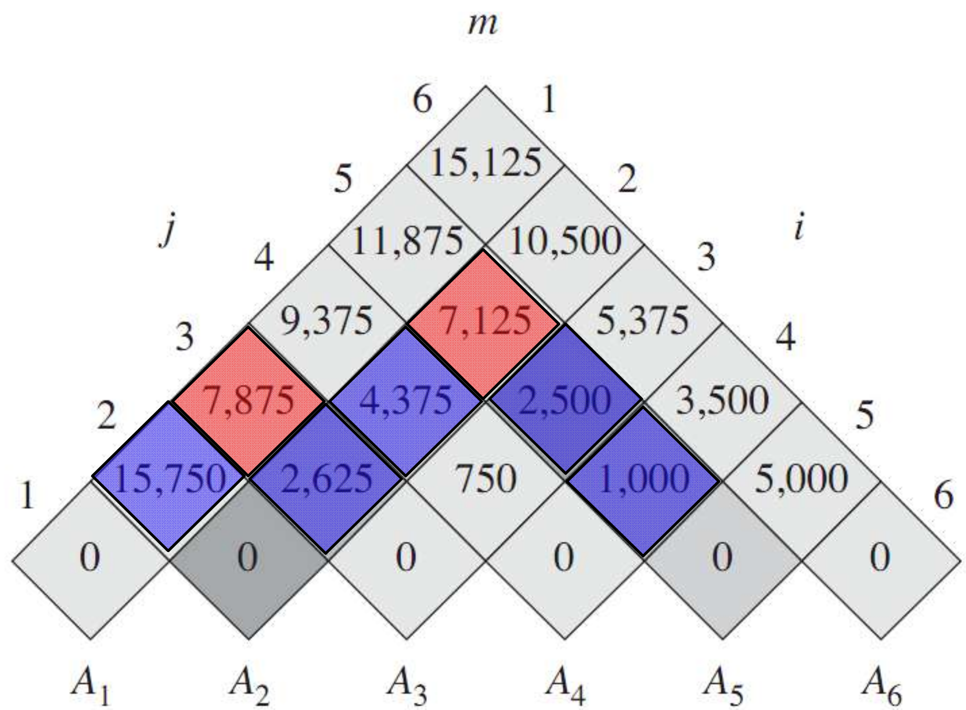


Source: Introduction to Algorithms, Cormen et al

The table method

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- ❑Compute bottom up, row by row.
- ❑Final answer we want is M(1,6).
- ❑Bottom row is all 0.
- ❑In second to bottom row, $A(i, i+1) = p_{i-1} p_i p_{i+1}$ because we multiply A_i and A_{i+1}.
- ❑Values in each row only depend on values in rows below, which we've already computed.



$$m[1,3] = \min \begin{cases} m[1,2] + m[3,3] + 30 \times 15 \times 5 = 18000 \\ m[1,1] + m[2,3] + 30 \times 35 \times 5 = 7875 \end{cases}$$

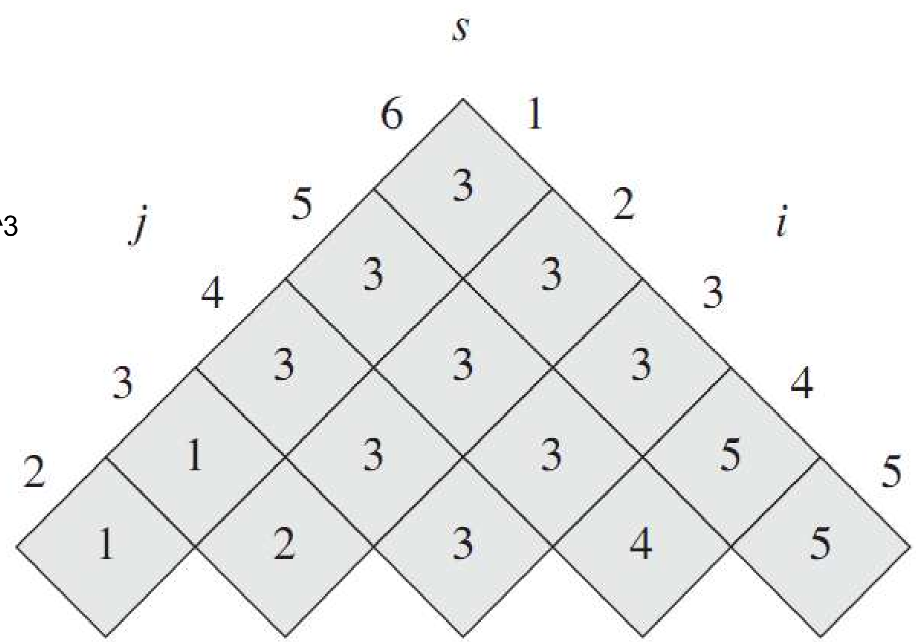
$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000 , \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125 , \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$

$$= 7125 .$$

The table method

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

- The s table says where the breaks should happen.
- This comes from which term produced the min cost in the m table.
- Ex To compute m[2,5], we saw breaking at A_3 gave the min cost. So s[2,5]=3.
- Ex To compute m[1,6] first break at 3.
- So the subproblems are m[1,3] and m[4,6].
- To compute m[1,3], break at 1.
- So the subproblems are m[1,1] and m[2,3].
- To compute m[4,6], break at 5.
- So the subproblems are m[4,5] and m[6,6].
- So altogether the optimal sequence is $\big(A_1(A_2A_3)\big)\big((A_4A_5)A_6\big)$.



Cost of the table method

- We have three nested loops, each of size O(n). So the total time cost is $O(n^3)$.
- More intuitively, we have n^2 table entries to fill in, where entry

  $$M(1,n) = \min_{1 \le j \le n-1} (M(1,j) + M(j+1,n) + p_0 p_j p_n)$$

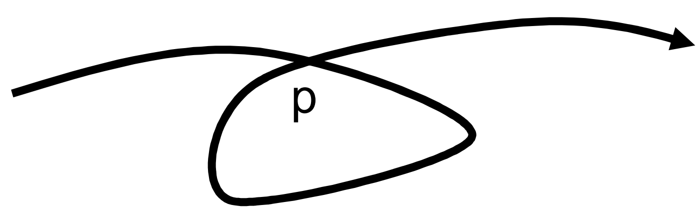
  involves checking O(n) other table entries. So the cost is $n^2 \times O(n) = O(n^3)$.
- Space complexity is $O(n^2)$, because we use two tables of size n^2.

Problems without optimal substructure

- Can we use dynamic programming to solve all problems efficiently?
- No. There are many problems we don't have any efficient solutions to.
- One main reason is, not all problems have the optimal substructure property.
  - Sometimes, solving a problem optimally involves solving subproblems nonoptimally!
- How is this possible?
  - One main reason is optimal solutions to subproblems might not be combinable to a solution for the original problem.

Problems with(out) optimal substructure

- Given a graph, find
  - The shortest path between two vertices.
  - The longest simple path between two vertices.
    - Simple path can't use a node twice.
    - Without the restriction, longest path can be infinite. Just go round and round forever.



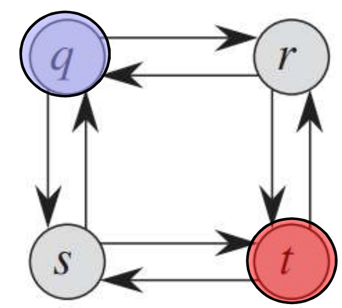
Not a simple path, cause there's a cycle, which uses p twice.

Shortest path

- Shortest paths has optimal substructure.
- Shortest path from x to y has to go through one of y's neighbors.
  - Subproblems are shortest paths from x to z, for every neighbor z of y.
  - Then $d(x,y) = \min_{(z,y)\in E}(d(x,z) + w(z,y))$.
  - I.e. shortest path from x to y is to take shortest path from x to one of y's neighbors, then go to y.

Longest simple path

- Longest simple path does not have optimal substructure.
  - I.e., you can't get the longest simple paths (LSP) from x to y by piecing together LSP's from x or y to other nodes.
- LSP from q to t is q,r,t.
  - LSP from q to r is q,s,t,r.
  - LSP from r to t is r,q,s,t.
  - But we can't combine the simple paths from q to r, and from r to t, into another simple path.
    - When subsolutions aren't combinable, we say the subproblems aren't independent.
- Not only can't we solve LSP using dynamic programming, LSP has no known efficient solution.



Longest common subsequence

- Given a string X, a subsequence of X is any string formed by removing some letters of X.
  - Ex “let” is a subsequence of “letters”.
    - So are “les” and “ees”.
  - Subsequence isn’t necessarily consecutive.
- A common subsequence of strings X and Y is a subsequence of both X and Y.
  - E.g. “ees” is a common subsequence of both “letters” and “cheers”.
  - It’s not the longest common subsequence (LCS).
  - The longest common subsequence (LCS) of “letters” and “cheers” is “eers”.
  - There may be several LCS’s for some strings.

Application

- Given two DNA sequences X and Y, how similar are they? This helps us understand
  - Will a person get a disease?
  - Are two people related?
  - What is the function of a new protein?
- One way is to look at the LCS of X and Y.
  - If the LCS is very long, then X and Y are probably similar.
- Many other ways to measure similarity.

Subproblems in LCS

In dynamic programming, we divide a problem into smaller subproblems.
For LCS, we divide the problem on a long string to the problem on a shorter string, by removing the last letter of the long string.

X **B D C A B A**

Y **A B C B D A**

The last letter of X and Y are both 'A'. So, 'A' is definitely in some LCS of X and Y.
Now, remove 'A' from X and Y to get X' and Y'.
Then the LCS of X and Y, is the LCS of X' and Y', appended by 'A'.

X' **B D C A B**

Y' **A B C B D**

The last letters of X' and Y' don't match. So these two letters can't both be in LCS of X and Y.
So either the last letter of X' isn't in the LCS, or the last letter of Y' isn't in the LCS.

We don't know which possibility is true, so the DP will try both.

X'' **B D C A**

Y' **A B C B D**

If 'B' not in LCS, we remove 'B' from X' to get X''.
The LCS of X' and Y' is the LCS of X'' and Y'.

X' **B D C A B**

Y'' **A B C B**

If 'D' not in LCS, we remove 'D' from Y' to get Y'', and the LCS of X' and Y' is the LCS of X' and Y''.

A dynamic program for LCS

- Let S[1,i] be the first i letters of a string S, and S[i] be the i'th letter of S.
  - Let S[i,0] be the empty string, for any i.
- Let LCS(X[1,i], Y[1,j]) be the LCS of X[1,i] and Y[1,j].
- Let c(i,j) be the length of LCS(X[1,i], Y[1,j]).
- We have the following dynamic programming equations.

c[i,0] is LCS of X[1,i] and Y[1,0]. Since Y[1,0] is empty string, it has no LCS with X[1,i]. Similarly for c[0,j].

$$c[i,0] = 0 \text{ for all } i$$
$$c[0,j] = 0 \text{ for all } j$$

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j], c[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

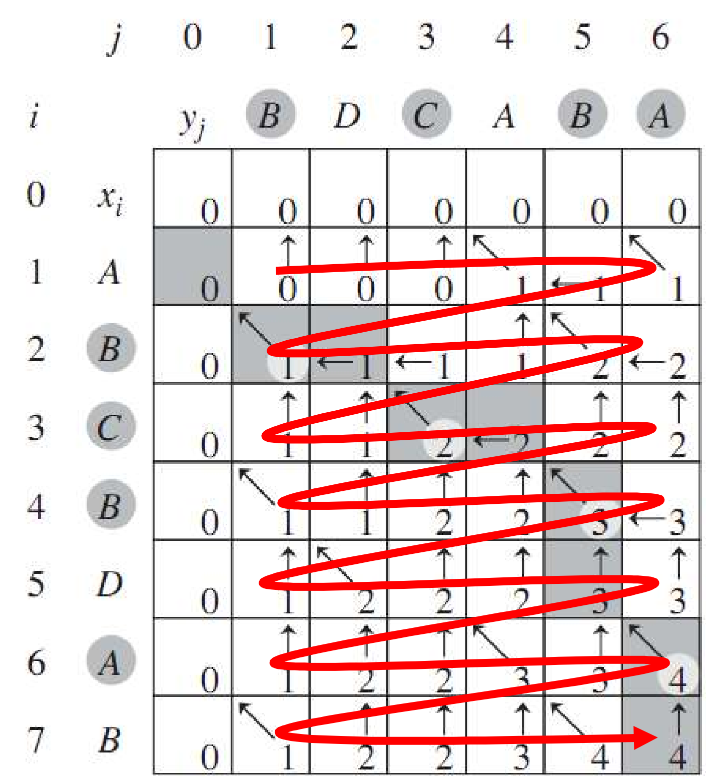
From argument on last slide.

The table method for LCS

- Make a table to record the c[i,j] values.
- Row is prefixes of Y, column is prefixes of X.
- c[i,j] is length of LCS(X[1,i],Y[1,j]).
- Start with the 0 column and 0 row. Fill in all 0's.
- Fill in rest of table from left to right, and from top to bottom. I.e. (1,1), (1,2),(1,3),…, (2,1),(2,2),…
  - Because a cell's value depends on vals in cells to the left, left-up, and up.

c[i,0] = 0 for all i.
c[0,j] = 0 for all j.

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j], c[i, j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$



Source: Introduction to Algorithms, Cormen et al.

The table method for LCS

- If the letters in row i and column j match, val of cell (i,j) = val of cell (i-1,j-1)+1.
  - Also, make a diagonal arrow, indicating LCS(X[1,i],Y[1,j]) is LCS(X[1,i-1],Y[1,j-1]) plus X[i] (or Y[j]).
- If letters don't match, val of cell (i,j) = max of vals in cells (i-1,j), (i,j-1).
  - Make an arrow to whichever cell gives has higher val (break ties arbitrarily).
  - Arrow indicates which way to go to find LCS.
- Length of LCS is in bottom right cell (4 in the example).
- LCS string given by following arrows starting from bottom right.
  - Each diagonal arrow corresponds to a matching letter.
  - The LCS is BCBA in the example.

c[i,0] = 0 for all i.
c[0,j] = 0 for all j.

$$c[i,j] = \begin{cases} c[i-1,j-1] + 1 & \text{if } X[i] = Y[j] \\ \max(c[i-1,j],\ c[i,j-1]) & \text{if } X[i] \neq Y[j] \end{cases}$$

	j	0	1	2	3	4	5	6
i		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
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

Cost of the LCS DP

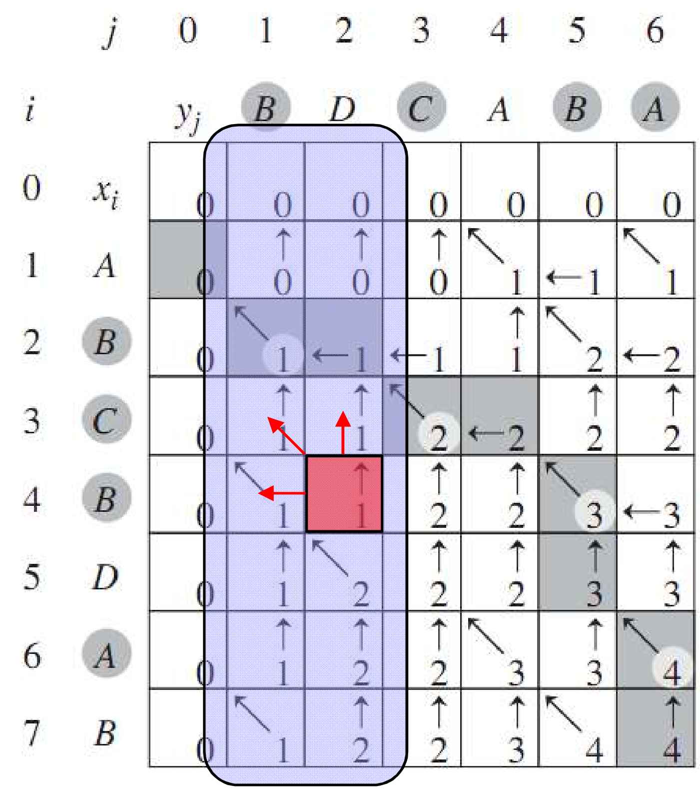
- There are $O(mn)$ entries in the c table.
- Filling each entry requires looking at 1 or 2 adjacent entries.
- So running time $O(mn)$.
- Amount of memory needed (space complexity) is also $O(mn)$, since tables have mn entries.

LCS in linear space

- For practical applications, e.g. in bioinformatics, m and n can be huge, $O(10^6)$.
- Running time can be $O(mn) = O(10^{12})$ or higher, which is feasible in practice.
- However, $O(10^{12})$ space complexity is often too much.
  - Can we solve LCS with linear, i.e. $O(m+n)$ space complexity?

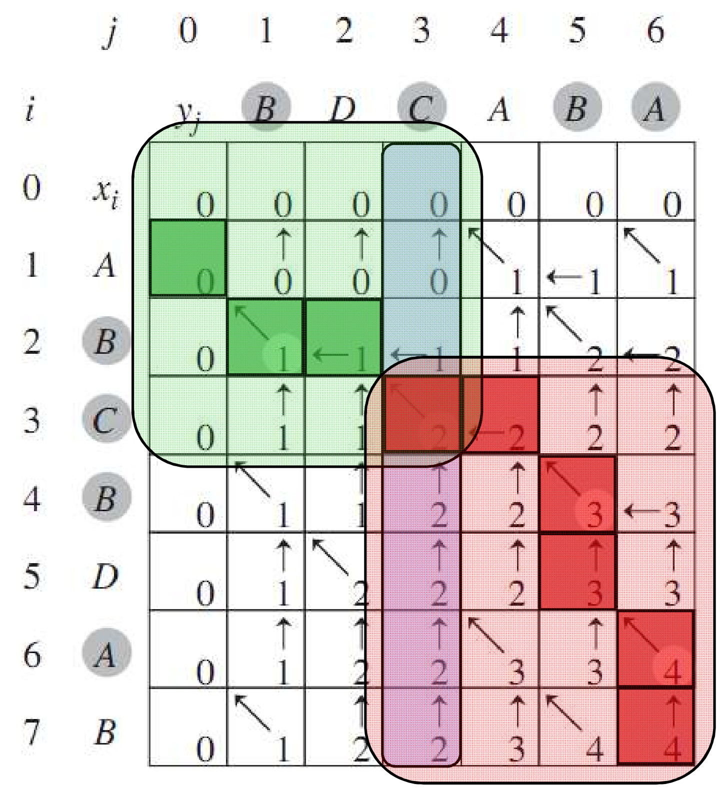
LCS in linear space

- Notice that we can easily compute the value of the LCS in linear space.
- Suppose $m \le n$. To compute the value of $c[i, j]$, only need values from $c[i-1, j], c[j-1, i]$ and $c[j-1, i-1]$.
- So when computing the j'th column, we only need values from the j'th and $j-1$'st columns.
  - So we only need to keep these two columns in memory.
  - So the memory complexity compute $c[m, n]$ is $O(m+n)$.
- But how do we compute the LCS string in linear space?
  - Retracing the LCS path seems to require storing entire matrix.



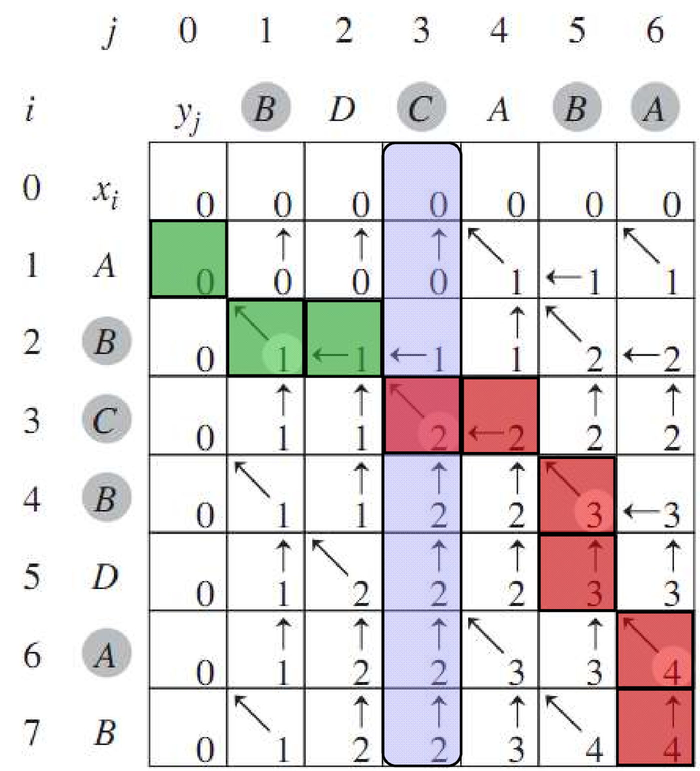
LCS in linear space

- Hirschberg (1975) proposed a way to find the LCS string in $O(mn)$ time and $O(m+n)$ space.
- The LCS path moves up and to the left, starting from (m, n).
- Suppose it crosses the middle column $n/2$ in row j.
  - We'll see later how to find j.
- This divides the LCS path into two parts, from (1,1) to (j,n/2), and (j,n/2) to (m,n).
- First part is an LCS for X[1,j] and Y[1,n/2].
- Second part is an LCS for Y[n,n/2] and X[m,j].
  - Y[n,n/2] is the reverse of Y[n/2,n], and X[m,j] is the reverse of X[j,m].



LCS in linear space

- How do we find the crossing row j of the LCS?
- Compute all the LCS values for $X[1,m]$ and $Y[1,n/2]$.
  - Let $p(i,n/2)$ be value in row i, column $n/2$.
- Compute all the LCS values for $X[m,1]$ and $Y[n,n/2]$.
  - Do this the same way as for normal LCS.
  - Let $q(i,n/2)$ be value in row i, column $n/2$.
- Then $j = \max_{1 \le i \le m} p\left(i,\frac{n}{2}\right) + q(i,\frac{n}{2})$.
- Computing all the LCS values and j takes $O(mn)$ time and $O(m+n)$ space.
- Once we find j, computing the green path takes $O(m+n)$ space, by induction.
- After computing the green path, we can reuse the same space to compute the red path.
- Thus, the entire computation takes $O(m+n)$ space.



Sequence Alignment: Running Time Analysis

Theorem. Let T(m, n) = max running time of algorithm on strings of length m and n. T(m, n) = O(mn).

Pf. (by induction on n)

- O(mn) time to compute f(•, n/2) and g (•, n/2) and find index q.
- T(q, n/2) + T(m - q, n/2) time for two recursive calls.
- Choose constant c so that:

$$\begin{aligned} T(m,\ 2) &\leq cm \\ T(2,\ n) &\leq cn \\ T(m,\ n) &\leq cmn + T(q,\ n/2) + T(m-q,\ n/2) \end{aligned}$$

- Claim: $T(m, n) \leq 2cmn$
  - Base cases: m = 2 or n = 2.
  - Inductive hypothesis: $T(m', n') \leq 2cm'n'$ with m'<m and n'<n

$$\begin{aligned} T(m,n) &\leq T(q,n/2) + T(m-q,n/2) + cmn \\ &\leq 2cqn/2 + 2c(m-q)n/2 + cmn \\ &= cqn + cmn - cqn + cmn \\ &= 2cmn \end{aligned}$$