Dynamic Programming

Tim Dosen DAA
Semester Genap 2021/2022

References

Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, **Introduction to Algorithms *Third Edition,*** The MIT Press, 2009

Slide DAA Pak Stef – DAA 8

Dynamic Programming

- Like the divide-and-conquer method, dynamic programming solves problems by combining the solutions to subproblems.
- "Programming" here refers to a tableau (tabular) method, not to writing computer programs.
- As we saw in previous lectures, algorithms based on the divide-and-conquer method partition a problem into independent subproblems, solve the subproblems recursively, and then combine their solutions to solve the original problem.

  In contrast, dynamic programming is applicable when the subproblems are not independent, that is, when subproblems share subsubproblems.

Dynamic Programming

- When the subproblems are not independent, a divide-and-conquer algorithm does more work than necessary, that is, repeatedly solving the common subsubproblems.
- A dynamic-programming algorithm solves every subsubproblem just once and then saves its answer in a table, for later uses.

- Dynamic programming is typically applied to optimization problems. In such problems there can be many possible solutions. Each solution has a value, and we want to find a solution with the optimal (minimum or maximum) value.

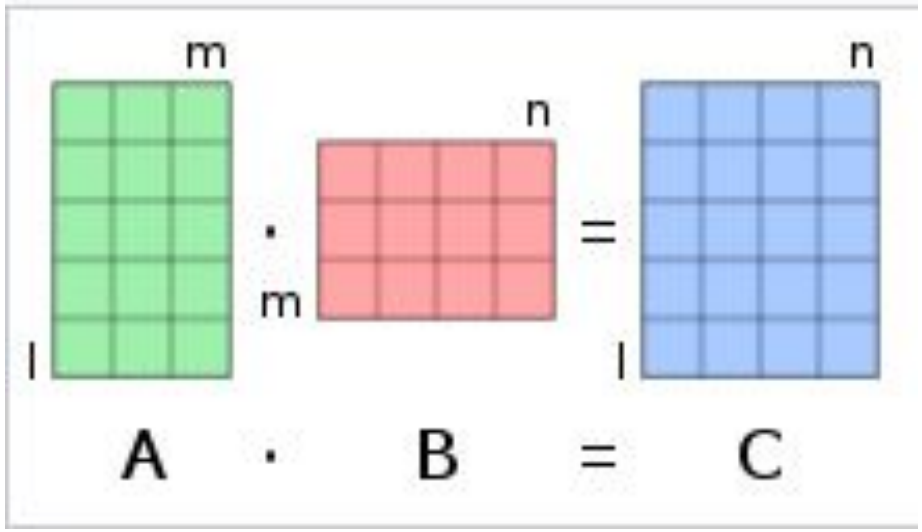
Dynamic Programming

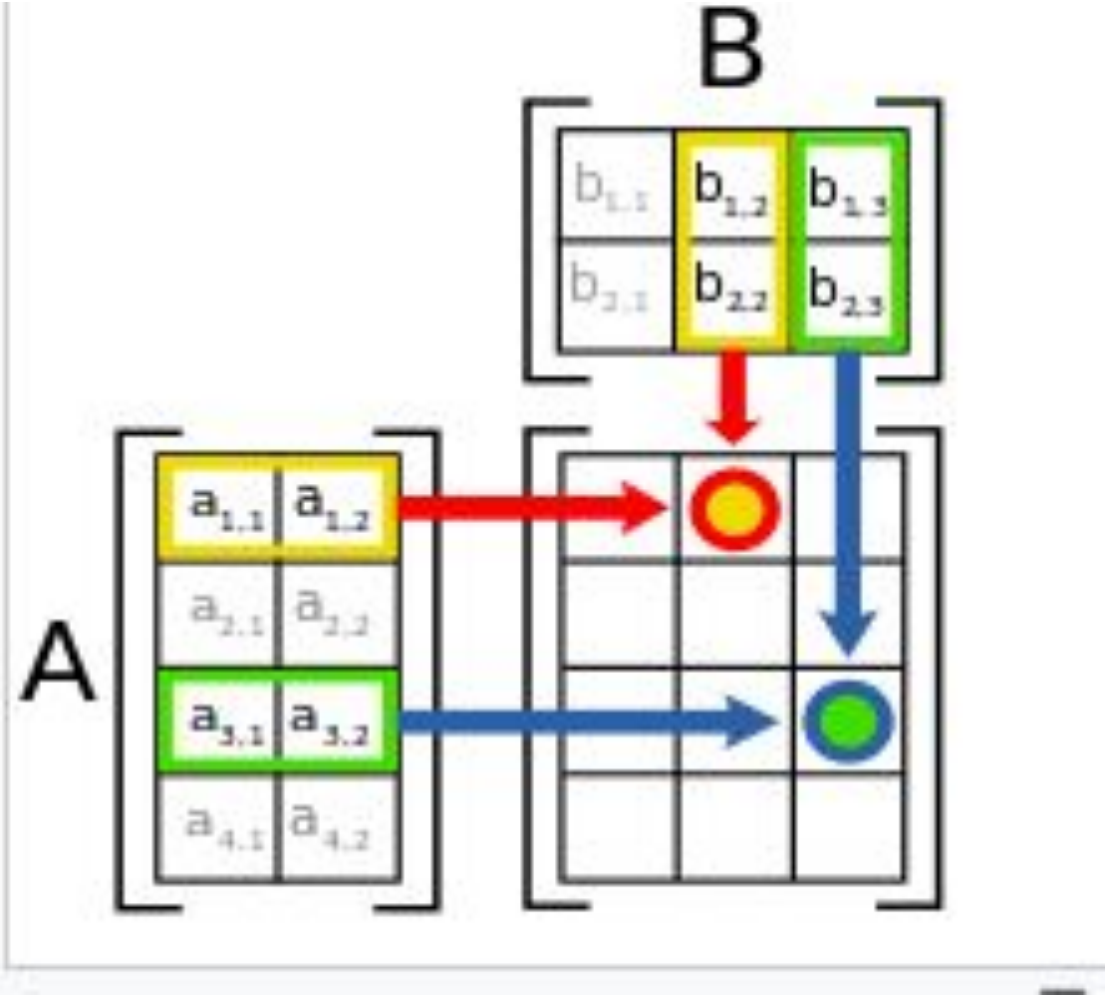
- The development of a dynamic-programming algorithm usually consists of 4 steps:

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

Matrix-chain Multiplication Problem

Matrix Multiplication





- The standard algorithm for multiplying two matrices is given by the following pseudocode.

```
MATRIX-MULTIPLY( A, B)
1.  if ncolumns[A] ≠ nrows[B]
2.      then error "incompatible dimensions"
3.      else  for i ← 1 to nrows[A]
4.                  do  for j ← 1 to ncolumns[B]
5.                          do  C[i, j] ← 0
6.                              for k ← 1 to ncolumns[A]
7.                                  do  C[i, j]← C[i, j] + A[i, k]·B[k, j]
8.              return C

//ncolumns = number of columns
//nrows = number of rows
```

Matrix Multiplication

- We can multiply two matrices A and B only if they are compatible: the number of columns of A is equal to the number of rows of B.
- If A is a (p x q) matrix and B is a (q x r) matrix, then the resulting matrix C is a (p x r) matrix.
- The time to compute C is dominated by the number of scalar multiplications in line 7, which is p·q·r in total.

Matrix-chain Multiplication Problem

Problem:

Given a sequence (a chain) $\langle A_1, A_2, \ldots, A_n \rangle$ of n matrices to be multiplied, where for $i = 1, 2, \ldots, n$, matrix A_i has dimension $p_{i-1} \times p_i$,

fully parenthesize the product $A_1 A_2 \cdots A_n$ in a way that minimizes the number of scalar multiplications.

Matrix-chain Multiplication Problem

- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.
- Matrix multiplication is associative, and therefore all parenthesizations yield the same product. But different parenthesizations can result in different costs in terms of the number of scalar multiplications. See the examples.

Matrix-chain Multiplication Problem

- Example:
  - consider the problem of a chain A1, A2, A3 of three matrices.
  - Suppose:
    - A1: 10 x 100
    - A2: 100 x 5
    - A3: 5 x 50
  - Parenthesization ((A1 A2) A3) needs:
    - 10 x 100 x 5 = 5000 to multiply (A1 A2) plus 10 x 5 x 50 = 2500 to multiply the output of (A1 A2) by A3 --> total 7500 scalar multiplications
  - Parenthesization (A1 (A2 A3)) needs:
    - 100 x 5 x 50 = 25000 to multiply (A2 A3) plus 10 x 100 x 50 = 50000 to multiply A1 by the output of (A2 A3) --> total 75000 scalar multiplications
  - Thus, computing the product according to the first parenthesization is 10 times faster.

Counting the number of parenthesizations

- P(n) = the number of alternative parenthesizations of a sequence of n matrices

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2. \end{cases}$$

- By the method of generating functions, it can shown that the solution is

  P(n) = C(n-1)

  where

$$C(n) = \frac{1}{n+1}\binom{2n}{n}$$
$$= \Omega(4^n / n^{3/2})$$

- C(n) is the nth Catalan number.

n	C(n)
1	1
2	2
3	5
4	14
5	42
6	132

Applying dynamic programming to MCM Problem

Step 1: The structure of an optimal parenthesization

- The first step in the dynamic-programming paradigm is to find the optimal substructure and then use it to construct an optimal solution to the problem from optimal solutions to subproblems.
- For the problem of matrix-chain multiplications, we can perform this step as follows.
- Notation: $A_{i..j}$ denotes the matrix that results from evaluating the product $A_i A_{i+1} \cdots A_j$, where $i \le j$.
- If the problem is nontrivial (i.e. $i < j$), then any parenthesization of the product $A_i A_{i+1} \cdots A_j$ must split the product between A_k and A_{k+1} for some k in the range $i \le k < j$. The cost of this parenthesization is (the cost of computing the matrix $A_{i..k}$) + (the cost of computing $A_{k+1..j}$) + (the cost of multiplying them together).

Step 1: The structure of an optimal parenthesization (2)

- The optimal substructure:

  Suppose that an optimal parenthesization of $A_i A_{i+1} \cdots A_j$ splits the product between A_k and A_{k+1}. Then the parenthesization of the subchain $A_i A_{i+1} \cdots A_k$ within this optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_i A_{i+1} \cdots A_k$.

  Similarly the parenthesization of the subchain $A_{k+1} A_{k+2} \cdots A_j$ within the optimal parenthesization of $A_i A_{i+1} \cdots A_j$ must be an optimal parenthesization of $A_{k+1} A_{k+2} \cdots A_j$.

  Why?

Step 1: The structure of an optimal parenthesization (3)

- Now we use the optimal substructure to show that we can construct an optimal solution to the problem from optimal solutions to subproblems.
- Any solution to a nontrivial instance of the matrix-chain multiplication problem needs to split the product, and any optimal solution contains optimal solutions to subproblem instances.
- Thus, an optimal solution to the problem can be built by:
  - splitting the problem into two subproblems (i.e., optimally parenthesizing $A_i A_{i+1} \cdots A_k$ and $A_{k+1} A_{k+2} \cdots A_j$);
  - finding optimal solutions to subproblem instances;
  - combining these optimal subproblem solutions.

When we search for the correct place to split the product, we have to consider all possible places so that we are sure of having found the optimal one.

Step 2: A recursive solution

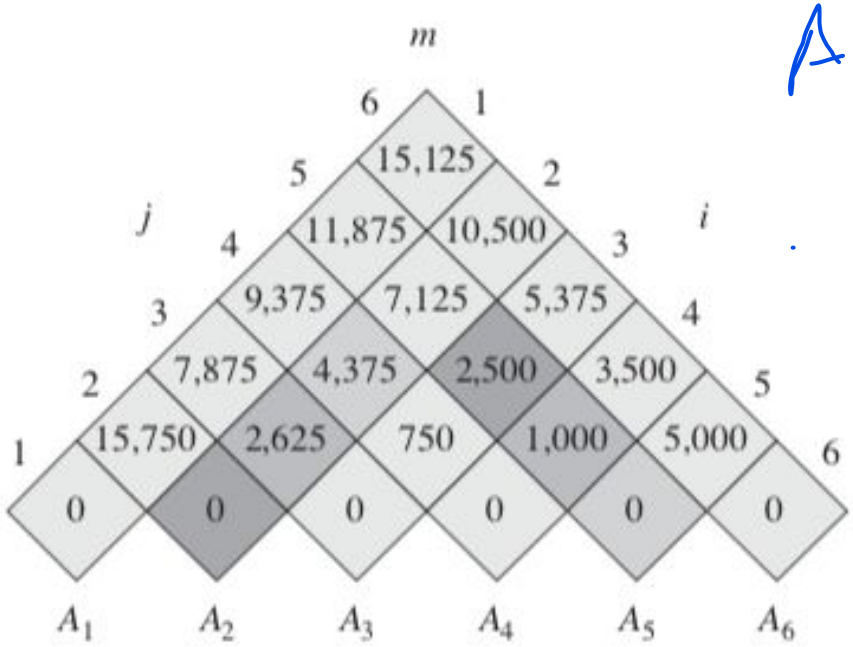
$$m[i,j] = \begin{cases} 0 & \text{if } i = j\ , \\ \min\limits_{i \le k < j} \{m[i,k] + m[k+1,j] + p_{i-1} p_k p_j\} & \text{if } i < j\ . \end{cases}$$

Step 3: Computing the optimal costs

- Instead of computing the solution to recurrence recursively, we compute the optimal cost by using **a tabular, bottom-up approach**.

```
MATRIX-CHAIN-ORDER(p)
 1  n = p.length − 1
 2  let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
 3  for i = 1 to n
 4      m[i, i] = 0
 5  for l = 2 to n                    // l is the chain length
 6      for i = 1 to n − l + 1
 7          j = i + l − 1
 8          m[i, j] = ∞
 9          for k = i to j − 1
10              q = m[i, k] + m[k + 1, j] + p_{i−1} p_k p_j
11              if q < m[i, j]
12                  m[i, j] = q
13                  s[i, j] = k
14  return m and s
```

Step 3: Computing the optimal costs (2)



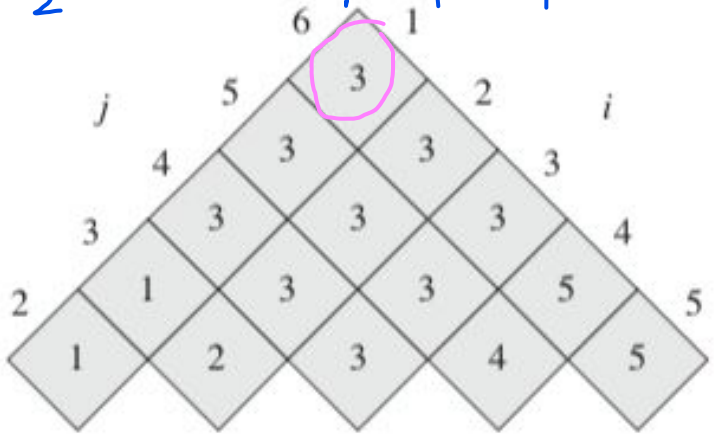


Figure 15.5 The m and s tables computed by MATRIX-CHAIN-ORDER for $n = 6$ and the following matrix dimensions:

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

karena A1...A6, berarti dari 1-6, kita bisa cek baris i = 1, j = 6, dimana pada tabel S, menunjukkan angka 3 (warna pink)

awal = (A1 A2 A3 A4 A5 A6) berarti hasilnya

(A1 A2 A3) (A4 A5 A6)

terus karena ada 2 partisi 1-3 dan 4-6, maka cek untuk partisi 1-3:
i = 1, j = 3, k = 1

Oleh karena itu dibuat

(A1 (A2 A3)) (A4 A5 A6)

Terus cek partisi kedua a4-6
i = 4, j = 7, dimana k = 5)

Oleh karena itu

dibuat (A1 (A2 A3)) ((A4 A5) A6))

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

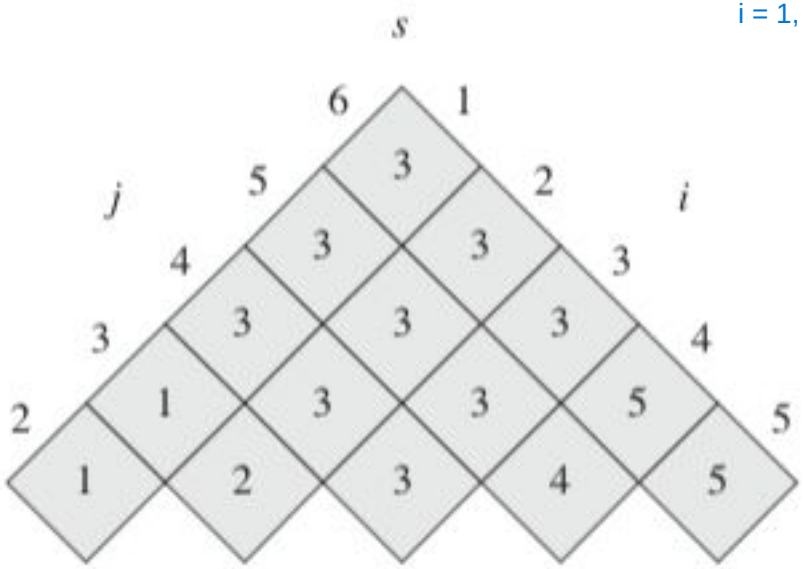
Step 4: Constructing an optimal solution

(A1 A2 A3 A4 A5 A6) i = 1, j = 6, s[i, j] = k = 3

i = 1, j = 3 (dari k tadi), s[i, j] = 1 (A1 (A2 A3) A4 A5 A6)

i = 1, j = 1, ya diemin aja or make something like this ((A1) (A2 A3) A4 A5 A6)

terus i = 4, j = 6




```
PRINT-OPTIMAL-PARENS(s, i, j)
1  if i == j
2      print "A"_i
3  else print "("
4      PRINT-OPTIMAL-PARENS(s, i, s[i, j])
5      PRINT-OPTIMAL-PARENS(s, s[i, j] + 1, j)
6      print ")"
```

final result (i think):
((A1 (A2 A3)) ((A4 A5) A6))

Recursive MCM

Recursive MCM

```
RECURSIVE-MATRIX-CHAIN(p, i, j)
1  if i == j
2      return 0
3  m[i, j] = ∞
4  for k = i to j − 1
5      q = RECURSIVE-MATRIX-CHAIN(p, i, k)
           + RECURSIVE-MATRIX-CHAIN(p, k + 1, j)
           + p_{i−1} p_k p_j
6      if q < m[i, j]
7          m[i, j] = q
8  return m[i, j]
```

- $T(n) = \Omega(2^n)$

Recursive MCM (2)

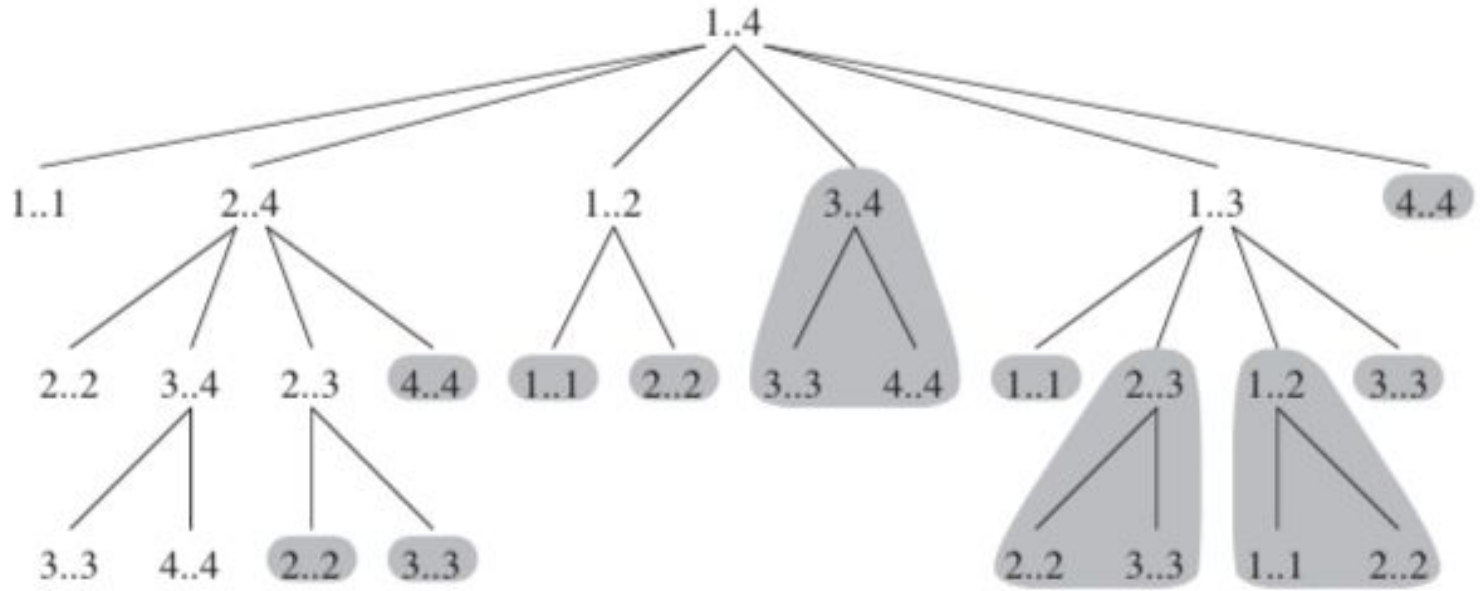


Figure 15.7 The recursion tree for the computation of RECURSIVE-MATRIX-CHAIN(p, 1, 4). Each node contains the parameters i and j. The computations performed in a shaded subtree are replaced by a single table lookup in MEMOIZED-MATRIX-CHAIN.

Memoization

Memoization

- Memoization is a variation of dynamic programming that often offers the efficiency of the usual dynamic-programming approach while maintaining a top-down strategy.
- As in ordinary dynamic programming, we maintain a table with subproblem solutions, but the process for filling in the table is more like the recursive algorithm.
- A memoized recursive algorithm maintains an entry in a table for the solution to each subproblem. Each table entry initially contains a special value to indicate that the entry has yet to be filled in. When the subproblem is first encountered during the execution of the recursive algorithm, its solution is computed and then stored in the table. Each subsequent time that the subproblem is encountered, the value stored in the table is simply looked up and returned.

MCM with Memoization

- $T(n) = O(n^3)$

MCM with Memoization

```
MEMOIZED-MATRIX-CHAIN(p)
1  n = p.length − 1
2  let m[1..n, 1..n] be a new table
3  for i = 1 to n
4      for j = i to n
5          m[i, j] = ∞
6  return LOOKUP-CHAIN(m, p, 1, n)

LOOKUP-CHAIN(m, p, i, j)
1  if m[i, j] < ∞
2      return m[i, j]
3  if i == j
4      m[i, j] = 0
5  else for k = i to j − 1
6          q = LOOKUP-CHAIN(m, p, i, k)
               + LOOKUP-CHAIN(m, p, k + 1, j) + p_{i−1} p_k p_j
7          if q < m[i, j]
8              m[i, j] = q
9  return m[i, j]
```

Bottom-up DP vs. Memoization

- In general practice, if all subproblems must be solved **at least once**, a bottom-up dynamic-programming algorithm usually outperforms the corresponding top-down memoized algorithm by a constant factor, because:
  - the bottom-up algorithm has no overhead for recursion and less overhead for maintaining the table.
  - Moreover, for some problems we can exploit the regular pattern of table accesses in the dynamic programming algorithm to reduce time or space requirements even further.
- Alternatively, if some subproblems in the subproblem space **need not be solved at all**, the memoized solution has the advantage of solving only those subproblems that are definitely required.

Exercise

1. List all possible parenthesizations of ABCD. How many are there?
2. Using your own words, highlight the differences (implementation strategy, complexity, etc.) of the algorithms for Matrix Chain Multiplication:
   a. using recursive (backtracking algorithm),
   b. bottom-up DP, and
   c. memoization.
3. **Do Worksheet Dynamic Programming: Matrix Chain Multiplication.**

Longest common subsequence (LCS)

Longest common subsequence (LCS) (1)

- Application in bioinformatics: comparing the DNA (deoxyribo-nucleic acid) of two or more different organisms.
- A strand of DNA consists of a string of molecules called bases, where the possible bases are adenine, guanine, cytosine, and thymine. Representing each of these bases by their initial letters, a strand of DNA can be expressed as a string over the finite set {A, C, G, T}.
- For example, the DNA of one organism may be

  S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA,

  while the DNA of another organism may be

  S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.
- One goal of comparing two strands of DNA is to determine how "similar" the two strands are, as some measure of how closely related the two organisms are.

Longest common subsequence (LCS) (2)

- One way to measure the similarity of strands S1 and S2 is by finding the longest common strand S3. The bases in S3 must appear in S1 and S2 in the same order, but not necessarily consecutively.

  The longest common strand S3 is GTCGTCGGAAGCCGGCCGAA.

- This notion of similarity can be formalize as the longest-common-subsequence problem.

- A subsequence of a given sequence is just the given sequence with zero or more elements left out.

  For example, Z = ⟨B, C, D, B⟩ is a subsequence of X = ⟨A, B, C, B, D, A, B⟩.

Longest common subsequence (LCS) (3)

- Given two sequences X and Y, we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y.

  For example, if X = ⟨A, B, C, B, D, A, B⟩ and Y = ⟨B, D, C, A, B, A⟩, the sequence ⟨B, C, A⟩ is a common subsequence of both X and Y, and so is the sequence ⟨B, C, B, A⟩.

  The sequence ⟨B, C, B, A⟩ is a longest common subsequence (LCS) of X and Y, since there is no common subsequence of X and Y which has length greater than 4.

  The sequence ⟨B, D, A, B⟩ is also an LCS of X and Y.

- Longest-Common-Subsequence (LCS) problem:

  Given two sequences X and Y, find a maximum-length common subsequence of X and Y.

Longest common subsequence (LCS) (4)

- A brute-force approach to solving the LCS problem is to enumerate all subsequences of $X = \langle x_1, x_2, \ldots, x_m \rangle$ and check each subsequence to see if it is also a subsequence of Y, keeping track of the longest subsequence found.

  There are 2^m subsequences of X, so this approach requires exponential time.

LCS with Dynamic Programming

Step 1. Characterizing an LCS (1)

- Notation: Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, we define the kth prefix of X, for $k = 0, 1, 2, \ldots, m$, as $X_k = \langle x_1, x_2, \ldots, x_k \rangle$.

  For example, if $X = \langle A, B, C, B, D, A, B \rangle$, then $X_4 = \langle A, B, C, B \rangle$ and X_0 is the empty sequence $\langle\ \rangle$.
- The LCS problem has an optimal-substructure property.

Step 1. Characterizing an LCS (2)

Theorem 15.1 (Optimal substructure of an LCS)
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}.

- This characterization shows that an LCS of two sequences contains within it an LCS of prefixes of the two sequences.
- Thus, the LCS problem has an optimal-substructure property.
- Also note that the LCS problem has the overlapping-subproblems property. To find an LCS of X and Y, we may need to find the LCS's of X and Y_{n-1} and of X_{m-1} and Y. But each of these subproblems has the subsubproblem of finding the LCS of X_{m-1} and Y_{n-1}.

Step 2. A recursive solution

Let us define c[i, j] to be the length of an LCS of the sequences X_i and Y_j.

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

Step 3: Computing the length of an LCS

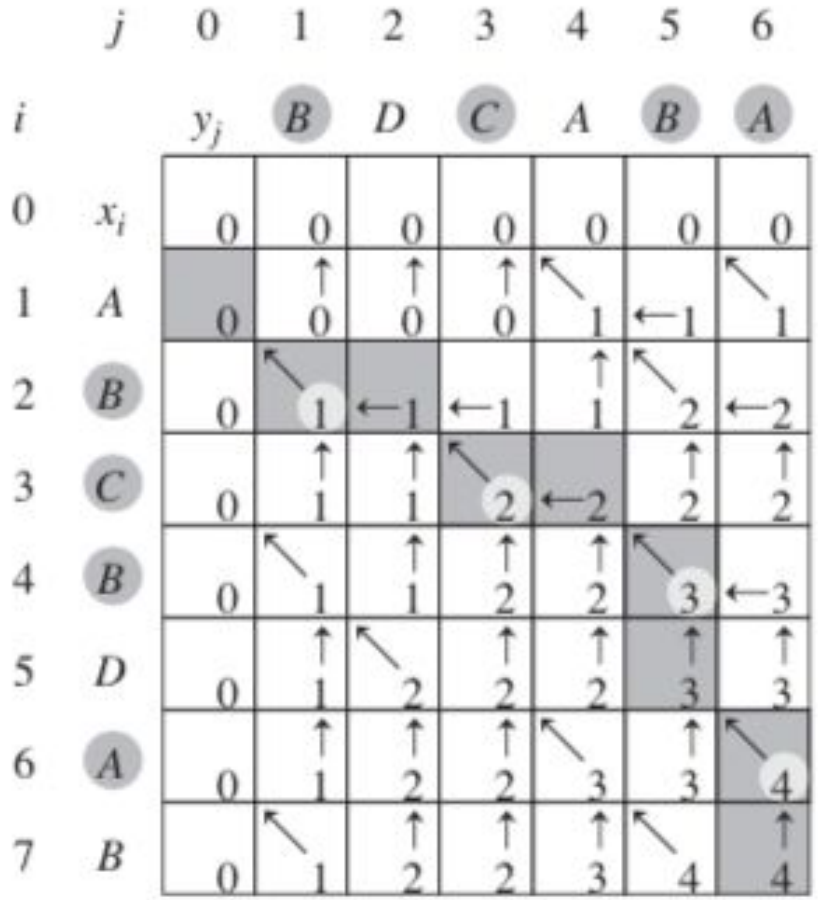
- The following LCS-LENGTH procedure takes two sequences X = $\langle x_1, x_2, \ldots, x_m \rangle$ and Y = $\langle y_1, y_2, \ldots, y_n \rangle$ as inputs. It stores the c[i,j] values in a table c[0 .. m, 0 .. n] whose entries are computed in row-major order. It also maintains the table b[1 .. m, 1 .. n] to simplify the construction of an optimal solution.

  b[i, j] points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i, j].

  c[m, n] contains the length of an LCS of X and Y.

```
LCS-LENGTH(X, Y)
 1  m = X.length
 2  n = Y.length
 3  let b[1..m, 1..n] and c[0..m, 0..n] be new tables
 4  for i = 1 to m
 5      c[i, 0] = 0
 6  for j = 0 to n
 7      c[0, j] = 0
 8  for i = 1 to m
 9      for j = 1 to n
10          if x_i == y_j
11              c[i, j] = c[i − 1, j − 1] + 1
12              b[i, j] = "↖"
13          elseif c[i − 1, j] ≥ c[i, j − 1]
14              c[i, j] = c[i − 1, j]
15              b[i, j] = "↑"
16          else c[i, j] = c[i, j − 1]
17              b[i, j] = "←"
18  return c and b
```

Step 3: Computing the length of an LCS (2)



i		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
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

- The running time of LCS-LENGTH is $O(mn)$, because each table entry takes $O(1)$ time to compute.

Step 4: Constructing an LCS

```
PRINT-LCS(b, X, i, j)
1  if i == 0 or j == 0
2       return
3  if b[i, j] == "↖"
4       PRINT-LCS(b, X, i − 1, j − 1)
5       print x_i
6  elseif b[i, j] == "↑"
7       PRINT-LCS(b, X, i − 1, j)
8  else PRINT-LCS(b, X, i, j − 1)
```

- The table b can be used to construct quickly an LCS of X and Y. We simply begin at b[m, n] and trace through the table following the arrows.
- An " ↖" in the entry b[i, j] implies that $x_i = y_j$ is an element of the LCS.
- The recursive procedure PRINT-LCS prints out an LCS of X and Y in the proper order.
- The initial call is PRINT-LCS(b, X, length[X], length[Y]).
- The running time of PRINT-LCS is O(m+n), because at least one of i and j is decremented in each stage of the recursion.

Exercise

1. **Do Worksheet Dynamic Programming: Longest Common Subsequence.**
2. (at home) Design a DP algorithm for longest common subsequence of three sequences (three strings).
   Example: FASILKOM, PUSILKOM, and FISIP as longest common subsequence SI.