CS5800: Algorithms — Spring '21 — Virgil Pavlu

Homework 5
Submit via Gradescope

Name: Zerun Tian

Collaborators:

Instructions:

- Make sure to put your name on the first page. If you are using the LaTeX template we provided, then you can make sure it appears by filling in the `yourname` command.
- Please review the grading policy outlined in the course information page.
- You must also write down with whom you worked on the assignment. If this changes from problem to problem, then you should write down this information separately with each problem.
- Problem numbers (like Exercise 3.1-1) are corresponding to CLRS 3^{rd} edition. While the 2^{nd} edition has similar problems with similar numbers, the actual exercises and their solutions are different, so make sure you are using the 3^{rd} edition.

1. ***(15 points)*** *Exercise 15.2-1.*

Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $[5, 10, 3, 12, 5, 50, 6]$.

Solution:

To solve this problem with dynamic programming, we use the following five steps.

(1) Characterize the optimal solution:

Suppose an optimal solution parenthesizes $A_i A_{i+1} ... A_j$ by splitting between A_k and A_{k+1}. We claim that, for subchains $A_i ... A_k$ and $A_{k+1} ... A_j$, they have optimal parenthesization, respectively. If any of the subchains had an even better parenthesization, it could replace the existing one and get a smaller number of scalar multiplications overall. This contradicts that our given solution is optimal, so our claim has to be correct.

(2) Define the recurrence of the objective:

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

We use $C[i, j]$ to denote the number of scalar multiplications for $A_i A_{i+1} ... A_j$. For $i = j$, there is no multiplication to be performed. For $i < j$, we search for a k between i (inclusive) and j (exclusive) such that the split yields the minimum number of scalar multiplications considering the optimal solution to subchain $C[i, k]$ and $C[k + 1, j]$ plus $p_{i-1} p_k p_j$, which is the number of multiplications required for multiplying the two resulting matrices.

(3) Compute the value of an optimal solution bottom-up:

We refer to the pseudocode MATRIX-CHAIN-ORDER on page 375 of CLRS.

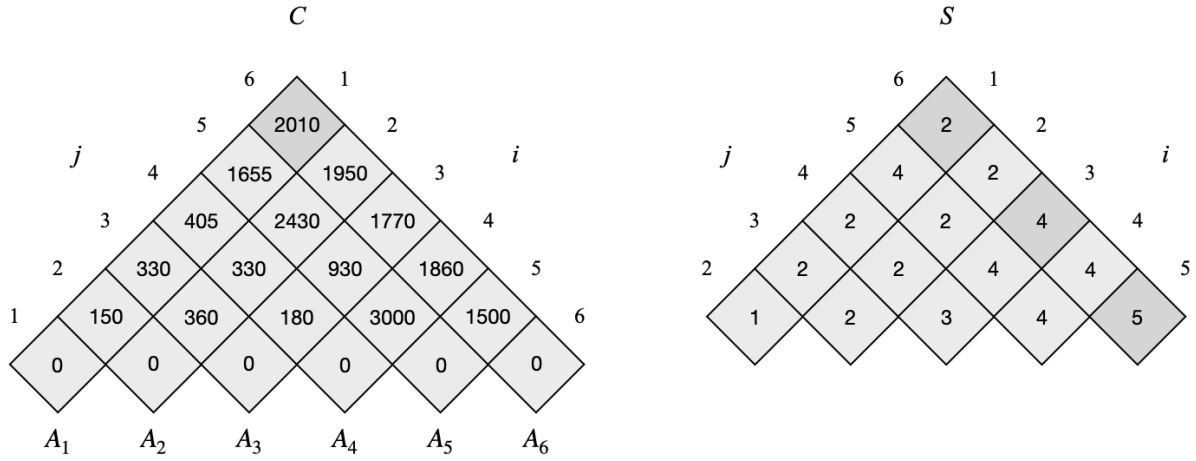
(4) Trace the optimal solution from the computed information:

In the procedure above, we have a table S that records the optimal split k at each cell, i.e. splitting between index k and $k + 1$. By inspecting where the optimal split k is at $S[1, n]$, we find the outermost split. Then, we continue find the splits at $S[1, k]$ and $S[k + 1, n]$ to split the two subchains further. The pseudocode PRINT-OPTIMAL-PARENS can be found on page 377 of CLRS.

(5) Runtime and space requirements:

The nested loop runs in $O(n^3)$ time which denominates the runtime of MATRIX-CHAIN-ORDER. In terms of space complexity, we have $O(n^2)$ because we use two tables C and S of size n^2.

After implementing the algorithm in code, we see the following results with the given dimensions.



Notice that for $C[i,i]$ whose values are zeros, we are working with single matrices. They are A_1 of size 5 by 10, A_2 of size 10 by 3, and so on so forth.

For $j-i=1$, $C[i,j]$ cells can be trivially computed by multiplying $p_{i-1}p_ip_j$. For example, $C[1,2] = p_0p_1p_2 = 5\cdot 10\cdot 3 = 150$. For $j-i=2$ cells, there are two possible parenthesizations. For example, for $C[1,3]$, we could have $A_1(A_2A_3)$ or $(A_1A_2)A_3$,

$$C[1,3] = \min\begin{cases} C[1,1]+C[2,3]+p0p1p3 = 0+360+5\cdot 10\cdot 12 = 960 \\ C[1,2]+C[3,3]+p0p2p3 = 150+0+5\cdot 3\cdot 12 = 150+180 = \mathbf{330} \end{cases}$$

The entry at $S[1,3]$ is 2 because we just found splitting between 2 and 3 results in less scalar multiplications. So on so forth, we fill the tables. For the final cell $C[1,6]$, we compare,

$$C[1,6] = \min\begin{cases} C[1,1]+C[2,6]+p_0p_1p_6 = 0+1950+5\cdot 10\cdot 6 = 2250 \\ C[1,2]+C[3,6]+p_0p_2p_6 = 150+1770+5\cdot 3\cdot 6 = \mathbf{2010} \\ C[1,3]+C[4,6]+p_0p_3p_6 = 330+1840+5\cdot 12\cdot 6 = 2530 \\ C[1,4]+C[5,6]+p_0p_4p_6 = 405+1500+5\cdot 5\cdot 6 = 2055 \\ C[1,5]+C[6,6]+p_0p_5p_6 = 1655+0+5\cdot 50\cdot 6 = 3155 \end{cases}$$

and found that first splitting between 2 and 3 results in the least number of scalar multiplications.

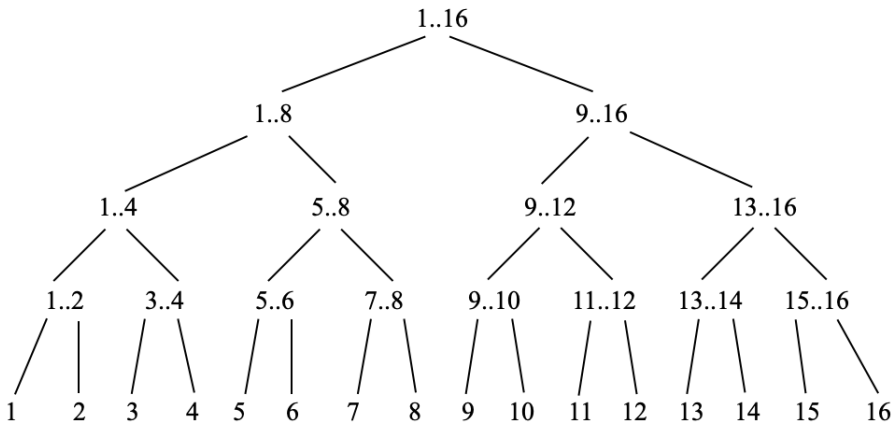
By looking at S, we know that the outermost split is between 2 and 3, so we have A_1A_2 and $A_3...A_6$. Then, we find that the optimal split for $A_3...A_6$ is between 4 and 5. This gives us an optimal parenthesization: $((A_1A_2)((A_3A_4)(A_5A_6)))$.

Code is listed in the last page of the document.

2. ***(15 points)*** *Exercise 15.3-2.*

Draw the recursion tree for the MERGE-SORT procedure from Section 2.3.1 on an array of 16 elements. Explain why memoization fails to speed up a good divide-and-conquer algorithm such as MERGE-SORT.

Solution:



From the recursion tree, we observe that for each range of indices *b..e*, it only appeared once in the tree. MERGE-SORT attempts to solve distinct inputs at every recursive call, which means there is no overlapping subproblem that needs to be stored to save computation time in a later call. Memoization in this problem even requires additional space.

3. ***(20 points)*** *Exercise 15.3-5.*

Suppose that in the rod-cutting problem of Section 15.1, we also had limit l_i on the number of pieces of length i that we are allowed to produce, for $i = 1, 2, ..., n$. Show that the optimal-substructure property described in Section 15.1 no longer holds.

Solution:

The optimal-substructure property states that the optimal solution to the rod-cutting problem incorporates optimal solutions to related subproblems, which may solve independently.

For example, prices for rods of different length are: $p_1 = 1$, $p_2 = 1$, $p_3 = 10$. Following the DP solution, a length-7 rod depends on optimal solutions to a length-3 rod and a length-4 rod. For the length-3 rod, we gain the maximum value when not cutting it. For the length-4 rod, we gain the maximum value when cutting it into pieces of length 3 and 1. Hence, by cutting the length-7 rod into two pieces of length 3 and one piece of length 1, we maximize the value: $10 + 10 + 1 = 21$. The optimal-substructure holds for the normal rod-cutting problem.

However, if we restrict the maximum number of pieces of length i that we can produce, we could not solve the subproblems independently. Using the same example and adding some limits: $l_1 = 10$, $l_2 = 10$, $l_3 = 1$, the above optimal solution does not apply because we can only produce one length-3 rod. The true optimal solution is cutting the length-7 rod into one length-3 and four length-1 pieces, which results in the value: $10 + 1 + 1 + 1 + 1 = 14$. This contradicts the optimal solution of DP. In this case, an optimal solution can no longer depends on optimal solutions to subproblems.

4. ***(20 points)*** *Problem 15-5.*

a. Given two sequences $x[1..m]$ and $y[1..n]$ and set of transformation-operation costs. Describe a dynamic-programming algorithm for the edit distance problem and prints an optimal operation sequence. Analyze the running time and space requirements of your algorithm.

Solution:

```
1:  function EDITDISTANCE(x, y)
2:      m = x.length, n = y.length
3:      let C[0..m, 0..n] and S[0..m, 0..n] be new tables.        ▷ tables initializtion
4:      for i = 0 to m do
5:          C[i, 0] = COST(delete) · i
6:          S[i, 0] = (i − 1, 0)
7:      for j = 0 to n do
8:          C[0, j] = COST(insert) · j
9:          S[0, j] = (0, j − 1)
10:     S[0, 0] = (0, 0)
11:     for i = 1 to m do
12:         for j = 1 to n do
13:             minCost = +inf
14:             c1 = COST(copy) + C[i − 1, j − 1]                 ▷ setting up the recurrence
15:             c2 = COST(replace) + C[i − 1, j − 1]
16:             c3 = COST(delete) + C[i − 1, j]
17:             c4 = COST(insert) + C[i, j − 1]
18:             c5 = COST(twiddle) + C[i − 2, j − 2]
19:             if x[i] == y[j] and c1 < minCost then    ▷ finding an operation to minimize cost
20:                 minCost = c1
21:                 S[i, j] = (i − 1, j − 1)
22:             if c2 < minCost then
23:                 minCost = c2
24:                 S[i, j] = (i − 1, j − 1)
25:             if c3 < minCost then
26:                 minCost = c3
27:                 S[i, j] = (i − 1, j)
28:             if c4 < minCost then
29:                 minCost = c4
30:                 S[i, j] = (i, j − 1)
31:             if c5 < minCost and i > 1 and j > 1 and x[i] == y[j − 1] and x[i − 1] == y[j] then
32:                 minCost = c5
33:                 S[i, j] = (i − 2, j − 2)
34:             C[i, j] = minCost
35:     return C and S
```

We use $C[i, j]$ to record the edit distance from $x[1..i]$ to $y[1..j]$; we store a pair of indices (p, q) at

$S[i, j]$ such that they identify the prefixes $x[1..p]$ and $y[1..q]$ after which the current operation is applied. For every i and j, we calculate its edit distance by finding the minimum cost among possible operations plus the edit distance of the subproblem prior to that operation. The recurrence of the objective is expressed in the above pseudocode from line 13 to 34.

With the resulting C and S tables, we are able to trace an optimal operation sequence by using the following procedure.

```
1:  function TRACEOPERATIONS(x, y, S)
2:      ops = []
3:      i = x.length
4:      j = y.length
5:      while i > 0 and j > 0 do
6:          p, q = S[i, j]                    ▷ backtrace the i, j to find the subproblem
7:          di = i − p
8:          dj = j − q
9:          if di == 1 and dj == 1 then
10:             if x[i] == y[j] then
11:                 ops.append("copy " + x[i])
12:             else
13:                 ops.append("replace by " + y[j])
14:         else if di == 1 and dj == 0 then
15:             ops.append("delete " + x[i])
16:         else if di == 0 and dj == 1 then
17:             ops.append("insert " + y[i])
18:         else if di == 2 and dj == 2 then
19:             ops.append("twiddle " + x[i − 1] + " and " + x[i])
20:         i = p
21:         j = q
22:     for op in ops.reversed do
23:         PRINT(op)
```

The running time of EDITDISTANCE is $O(mn)$ because the number of operations within the doubly-nested loop at line 11 is constant. The function TRACEOPERATIONS run in $O(m + n)$-time as how we backtrace the cells of table S. Overall, the DP algorithm runs in $O(mn)$-time. In terms of space, EDITDISTANCE takes $O(mn)$ primarily for the two tables C and S. The function TRACEOPERATIONS takes $O(m + n)$-space to keep track of the selected operations. Overall, the space complexity of the algorithm is $O(mn)$.

b. Explain how to cast the problem of finding an optimal alignment as an edit distance problem using a subset of the transformation operations copy, replace, delete, insert, twiddle, and kill.

Solution:

We are going to choose the subset: {copy, replace, delete, insert}. To formulate the alignment problem into an edit distance problem, we associate costs to operations such that the minimization of edit distance creates a well-matched and compact alignment.

When we delete a character from x, we think of it as adding a space to y at the corresponding spot for alignment. Similarly, when we insert a character to somewhere in y, we view it as adding a space to x for alignment. In these cases, alignment receives a -2 score, so we want to reduce such cases by associating a **cost of 2** with **delete** and **insert** operations. Furthermore, a **replace** operation implies that $x_j \neq y_j$, which receives a -1 in terms of alignment score. In this case, we want a **cost of 1** to be associated with **replace**. Finally, a **copy** operation implies that $x_j == y_j$ which results in a $+1$ for alignment score, so we associate a **cost -1**. We can solve the alignment problem of sequence x and y using the edit distance algorithm with those specific values for those operations. Finally, we find the *optimal-alignment-score* $= 0 -$ *edit-distance*.

5. ***(30 points)*** *(Note: you should decide to use Greedy or DP on this problem)*

Prof. Curly is planning a cross-country road-trip from Boston to Seattle on Interstate 90, and he needs to rent a car. ... Your algorithm or algorithms should output both the total cost of the trip and the various cities at which rental cars must be dropped off and/or picked up.

Solution:

Let's try to solve this using DP.

(1) Characterize the optimal solution:

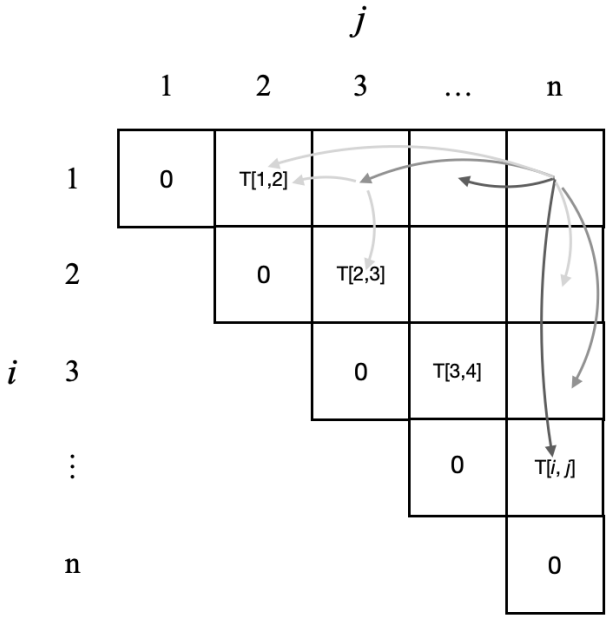
Suppose we are given an optimal solution: $[c_1, c_2, ..., c_m]$ where c_1 is Boston, c_m is Seattle, and c_2 .. c_{m-1} are cities where the professor should rent new cars to achieve an overall minimum cost. If we split it at any c_k for $1 < k < m$, $[c_1, ..., c_k]$ is an optimal solution to the subproblem of going from city c_1 to c_k. And, $[c_k, ..., c_m]$ is optimal to the subproblem of going from city c_k to c_m.

To show it is correct, we assume there exists a lower cost plan for a subproblem, so we could replace it in the given solution to achieve an overall lower cost. However, this means that the given solution is not optimal: a contradiction.

(2) Define the recurrence of the objective:

$$C[i,j] = \begin{cases} 0 & \text{if } i = j \\ \min\left\{T[i,j], \min_{i<k<j}\{C[i,k] + C[k,j]\}\right\} & \text{if } i < j \end{cases}$$

We use $C[i,j]$ to denote the minimum cost possible to travel from the city i to the city j. Because we don't consider backtrack, we omit cases of $i > j$. When $i < j$, the minimum cost is either going directly from city i to j or stopping at some city k that minimizes $C[i,k] + C[k,j]$.



(3) Compute the value of an optimal solution bottom-up:

```
1:  function PlanRentalCars(T)
2:      let C[1..n, 1..n], S[1..n, 1..n] be new tables of zeros      ▷ tables initialization
3:      for i = 1 to n do
4:          for j = i + 1 to n do
5:              p = j − i − 1
6:              q = j
7:              minCost = T[p, q]                                    ▷ setting up the recurrence
8:              S[p, q] = p
9:              for k = p + 1 to q − 1 do
10:                 cost = C[p, k] + C[k, q]
11:                 if cost < minCost then
12:                     minCost = cost
13:                     S[p, q] = k
14:             C[p, q] = minCost
15:     return C, S
```

(4) Trace the optimal solution from the computed information:

```
1:  function TraceSolution(cities, C, S)
2:      n = len(cities)
3:      e = n                                                        ▷ the index of ending city
4:      b = S[1][e]                                                  ▷ the index of beginning city
5:      Print("Total cost is " + C[1, e])
6:      plan = {1, e}                ▷ store a set of indices of cities that made to the final plan
7:      while b ≠ 1 do               ▷ continue unless the beginning city is Boston at index 1
8:          tmp = b
9:          b = S[tmp][e]
10:         e = tmp
11:         plan.add(e)                                              ▷ plan union {e}
12:     for i = 1 to n do
13:         if i in plan then
14:             Print(cities[i])
```

The TraceSolution first prints the total cost of an optimal solution at line 5. It then prints the cities from Boston to Seattle (from east to west) which are cities that Prof. Curly should drop off and/or pick up rental cars.

(5) Runtime and space requirements:

The runtime of PlanRentalCars has triple nested loops, each of which runs $\sim n$ times, resulting in $O(n^3)$. It requires $O(n^2)$-space for storing the C and S tables. The function TraceSolution runs in $O(n)$-time because the while loop at line 7 runs at most n iterations, and the for loop at line 12 runs n iterations each of which has a constant-time lookup. The space requirement is $O(n)$ for storing the indices of cities that are part of the final plan. Overall, to solve the problem and prints the solution, it takes $O(n^3)$-time with $O(n^2)$-space.

6. *(15 points)* *Exercise 15.4-5. (the last problem of hw4)*

Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Solution:

We create a new sequence $Y = [y_1, y_2, ..., y_n]$, which is the original sequence $X = [x_1, x_2, ..., x_n]$ sorted in increasing order. Then, we can apply a dynamic programming solution to find the longest common subsequence (LCS) of X and Y. These two steps together solve the problem. Since any subsequence of Y is monotonically increasing, any common subsequence of X and Y is guaranteed to be monotonically increasing. Our task now is reduced to finding a longest common subsequence of X and Y. To demonstrate a DP solution of LCS, we use the following steps:

(1) Characterize the optimal solution:

We are given a sequence $Z = [z_1, z_2, ..., z_k]$ that is an LCS of X and Y.

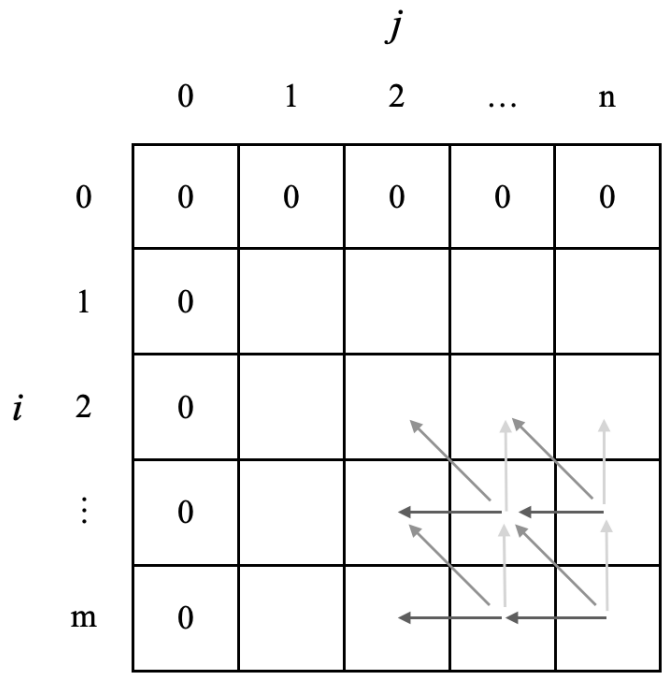
1. If $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
2. If $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
3. If $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Those observations can be proved correct via contradiction. For $x_m = y_n$, if z_k is not x_m, we can build a new sequence W of size $k+1$ by copying Z and appending x_m. This contradicts that Z is an optimal solution, so our $z_k \neq x_m$ assumption must be incorrect. Moreover, suppose there exists a W' that is an LCS of X_{m-1} and Y_{n-1} with length greater than $k-1$. By appending x_m to W', we effectively build a LCS of X and Y with length $> k$, which contradicts that Z of length k is optimal. Similarly, we can show the above claims 2 and 3 are correct (refer to CLRS p392).

(2) Define the recurrence of the objective:

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

The value at $C[i,j]$ represents the length of LCS of the prefixes X_i and Y_j. When either length of the prefixes is 0, there is no common subsequence. When $x_i = y_j$, we can build a longer common subsequence on top of what we found for prefixes X_{i-1} and Y_{j-1}. When $x_i \neq y_j$, we go with the longer subsequence so far.



(3) Compute the value of an optimal solution bottom-up:

```
1:  function LCS-LENGTH(X, Y)
2:      m = X.length
3:      n = Y.length
4:      let C[0..m, 0..n] be a new table
5:      for i = 0 to m do
6:          C[i, 0] = 0
7:      for j = 0 to n do
8:          C[0, j] = 0
9:      for i = 1 to m do
10:         for j = 1 to n do
11:             if x_i == y_j then
12:                 C[i, j] = C[i − 1, j − 1] + 1
13:             else if C[i − 1, j] ≥ C[i, j − 1] then
14:                 C[i, j] = C[i − 1, j]
15:             else
16:                 C[i, j] = C[i, j − 1]
17:     return C
```

(4) Trace the optimal solution from the computed information:

```
1:  function TRACE-LCS(X, Y, C)        ▷ C is the resulting matrix of LCS-LENGTH(X, Y)
2:      i = X.length
3:      j = Y.length
4:      k = C[i, j]
5:      ans = array of zeros of size k
```

```
6:  while i > 0 and j > 0 do
7:      if x_i == y_j then                ▷ when x_i equals y_j, it is a part of the final LCS
8:          ans[k] = x_i
9:          k = k − 1
10:         i = i − 1
11:         j = j − 1
12:     else if C[i − 1, j] ≥ C[i, j − 1] then
13:         i = i − 1
14:     else
15:         j = j − 1
16: return ans
```

We walk through the C table to trace an LCS of X and Y starting from $C[X.\text{length}, Y.\text{length}]$ to any $C[0, j]$ or $C[i, 0]$ by moving to smaller subproblems that yields a longer LCS.

(5) Runtime and space requirements:

The runtime for LCS-LENGTH is $O(mn)$ because the operations within the doubly-nested for loops are constant time. Its space requirement is also $O(mn)$ which is primary the C table. To trace the solution, TRACE-LCS runs in $O(m + n)$ time taking $O(k)$ spaces. Overall, the LCS problem can be solved in $O(mn)$-time.

Initially, we create Y that is a sorted copy of X, which takes $O(n \log n)$-time using merge sort. Now that the DP solution of LCS runs in $O(mn)$ time, and that X and Y are of the same size n, solving the problem takes $O(n^2)$-time.

```python
import sys
from os import linesep

def matrix_chain_order(p):
    n = len(p) - 1  # num of matrix
    C = [[0 for _ in range(n)] for _ in range(n)]
    S = [[0 for _ in range(n)] for _ in range(n)]

    for l in range(2, n + 1):  # l is chain length
        for i in range(n - l + 1):
            j = i + l - 1
            C[i][j] = sys.maxsize
            for k in range(i, j):
                q = C[i][k] + C[k + 1][j] + p[i] * p[k + 1] * p[j + 1]
                if q < C[i][j]:
                    C[i][j] = q
                    S[i][j] = k
    return C, S

def trace_opt_parens(S, i, j):
    if i >= j:
        return f'A_{i}'
    else:
        return f'({trace_opt_parens(S, i, S[i][j])}{trace_opt_parens(S, S[i][j] + 1, j)})'

def print_tables(X):
    print(linesep.join(', '.join(f'{col}' for col in row) for row in X))

def main():
    p = [5, 10, 3, 12, 5, 50, 6]
    C, S = matrix_chain_order(p)
    print_tables(C)
    print("")
    print_tables(S)
    print(trace_opt_parens(S, 0, len(S) - 1))


if __name__ == '__main__':
    main()
```