COMP 3270
Introduction to Algorithms
Programming Assignment

Due: July 19^{th}, Friday by 11:59 PM (midnight)
Late submissions will not be accepted even if you have a proper excuse because you have 39 days. Not submitting this HW will result in an F grade.

Objective: The empirical analysis of algorithms involves implementing, running, and analyzing the run-time data collected against theoretical predictions. This homework asks you to implement and theoretically and empirically determine the complexities of algorithms involving (a) Matrix Multiplication and (b) Matrix-Chain Multiplication.

Matrix Multiplication is a fundamental operation in many machine learning, scientific computing, graph theory, and pattern recognition problems. **For details on the matrix multiplication algorithms listed in this programming assignment, please see section 4.1 (page 80) and section 4.2 (page 85) of your textbook.**

The definition of matrix multiplication is as follows: if $C = AB$ for an $n \times m$ matrix A and $m \times p$ matrix B, then C is an $n \times p$ matrix with entries

$$c_{ij} = \sum_{k=1}^{m} a_{ik} b_{kj}. \tag{1}$$

Matrix-Chain Multiplication via Dynamic Programming: Given a sequence (chain)

$$\langle A_1, A_2, \ldots, A_n \rangle \tag{2}$$

of n matrices to be multiplied, where the matrices are not necessarily square, the goal is to compute the product

$$A_1 A_2 \ldots A_n \tag{3}$$

The matrix-chain multiplication involves minimizing the number of scalar multiplications. The problem can be solved by parenthesizing the expression in equation 3 and applying the pairwise matrix multiplication algorithm as a subroutine. Because matrix multiplication is associative, all parenthesizations produce the same outcome. For example, consider the matrix chain multiplication of $A_1 A_2 A_3 A_4$. The product can be fully parenthesized in the following ways, all of which yield the same result:

$$(A_1(A_2(A_3A_4))),\ (A_1((A_2A3)A_4)),\ ((A_1A_2)(A_3A_4)),\ ((A_1(A_2A_3))A_4),\ (((A_1A_2)A_3)A_4).$$

How you parenthesize the expression significantly affects the number of scalar multiplications necessary to evaluate the product. The **dynamic programming** algorithm that you will be implementing for this assignment generates an optimal ordering of the multiplications to minimize the number of scalar multiplications needed to evaluate the matrix-chain expression. Please **read pages 373 through 381 of the textbook** to understand the logic of the dynamic programming algorithm before attempting to solve the problem.

Requirements:

1. Implement the provided algorithms with respective complexity orders to solve this problem. You are required to use the **Python** language over the **Jupyter Notebook** platform (interactive development environment) and provide the notebook as part of your submission. `Jupyter Notebook` (i.e., JupyterLab) software is available for download at `https://jupyter.org/`. The implementation should be consistent with the algorithms provided in this homework. Besides, you are required to use the `NumPy` library (e.g., numpy.array) (see `https://numpy.org/`) for matrix operations specified in the algorithms along with the `pandas` python data analysis library (`https://pandas.pydata.org/`) for data collection, storage (e.g., pandas data frame), and visualization. You are also required to use the `markdown` cell text/comment feature of the `Jupyter` environment to practice the **Literate Programming** paradigm in software engineering. See `https://en.wikipedia.org/wiki/Literate_programming` for more on Literate Programming. If you turn in code that implements different algorithms, generated by AI or found on the web, then you will get a zero. For more on the AI policy and code reuse, please see the submission instructions.

2. Calculate **T(n)** and the order of complexity of each algorithm in the **big-Oh** or **Theta** notation using any of the methods we discussed in class. (a) Show your calculations and derivations step by step in detail, then (b) state the polynomial (or recurrence relation for the recursive algorithms) T(n) and (c) determine the complexity order of the algorithm. Incomplete answers to this part will get a zero, not a partial grade.

3. **Matrix Multiplication Problem**: Write `Python` scripts that carry out the following tasks, one after the other:

   (a) First, read from a file named **"input.txt"** containing two sequences of 16 comma-delimited integers in the first line. The second sequence should be delimited by a semicolon (e.g., $1, 2, 3, 4, 5, \ldots, 16; 1, 2, 3, 4, 5, \ldots, 16$). Create two 4×4 matrices containing the 16 integers from each sequence. The first sequence represents the first matrix, and the following represents the second matrix. Run each of the algorithms on the input matrices and print out the answer (calculated matrix) produced by each on the notebook as follows: "Algorithm-1: <answer>; Algorithm-2:<answer>; Algorithm-3:<answer>; Algorithm-4:<answer>; Algorithm-5:<answer> where <answer> is the result of the matrix multiplication as determined by the respective algorithm.

   (b) **Experiment - I**: The scripts of the steps listed below should be presented explicitly in the `Jupyter Notebook`. Each step should be preceded by a description (text) that explains the objective as well as the input and output specification (in natural language) of the following code using a `Jupyter` **markdown** cell.

   - Generate pairs of 30 square matrices of size 10×10, 20×20, 30×30, ..., 300×300, each containing randomly generated real numbers between 0 and 1.
   - Then use the system clock to measure time t_1, run one of the following three algorithms: **Algorithm-1**, **Algorithm-2**, **Algorithm-3**, measure time t_2, calculate the time it takes to complete the execution of the algorithm (i.e., $t_2 - t_1$). Do this for each of the algorithms executing on each of the $n \times n$ pairs of input matrices to generate a 30×7 data frame (i.e., `pandas` data frame). Each row in the data frame represents the input size and the empirical (and theoretical) time the three algorithms take when applied to the input matrices. Each row of the data frame table corresponds to the input size, s, corresponding to $s \times s$ input matrices, where s ranges from 10 to 300. The first four columns of the data frame are the size of the problem (e.g., 10 for 10 by 10 matrices), followed by the columns representing the execution (empirical) time for Algorithm-1, Algorithm-2, and Algorithm-3. To minimize variance, you can run each algorithm on a given input matrix multiple times (e.g., 10 times) and record the average time measurements.
   - Fill the last three columns of this data frame with values

     $$\lceil T_1(n) \rceil, \lceil T_2(n) \rceil, \lceil T_3(n) \rceil \tag{4}$$

     where n = each input size and $T(n)$ are the polynomials representing the theoretically calculated/predicted complexity of the three algorithms determined. So, column 2 will have measured running times of your first algorithm, and

column 5 will have the predicted (theoretical) value for the same algorithm; similarly for columns 3 & 6, 4 & 7. You may need to scale the complexity values (or use an appropriate time unit such as nano/micro/milli seconds for the measured running times) to bring all data into similar ranges.

- Print the data frame in the `Jupyter Notebook` and then visualize/plot a labeled graph/chart that shows the growth rates (both theoretical/predicted and empirical) as a function of the input size (x-axis ranging from 10 to 300) for each algorithm using the `Python` line plot library (or the line plot features of the `pandas` library). Label the curves appropriately.

(c) **Experiment - II**: The scripts of the steps listed below should be presented explicitly in the `Jupyter Notebook`. Each step should be preceded by a description (text) that explains the objective as well as the input and output specification (in natural language) of the following code using a `Jupyter` **markdown** cell.

- Generate 9 exponentially growing square matrices of size $2^i \times 2^i$, where $0 \leq i \leq 9$. Perform the same steps listed for Experiment-I above, but using instead the following algorithms: **Algorithm-1**, **Algorithm-3**, **Algorithm-4**, and **Algorithm-5**.
- Print the generated data frame in the `Jupyter Notebook` and then visualize the growth rates (theoretical/predicted and empirical) as a function of the input size for each algorithm using the Python line plot library (or the line plot features of the `pandas` library). Label the curves appropriately.

4. **Matrix-Chain Multiplication Problem**: Write Python scripts that carry out the following tasks:

- Generate 19 matrix chains, each containing 10 matrices: $\langle A_1, \ldots, A_{10} \rangle$. Each matrix contains randomly generated real numbers between 0 and 1. The dimensions of each one of the matrices in the chain are determined as follows: Let $(p_{j,i-1}, p_{j,i})$ be the dimensions of the matrix A_i in the j^{th} matrix chain ($2 \leq j \leq 20$), where p_i, $0 \leq i \leq 10$, is a random integer between 10 and $j \times 10$. Using random integer generation to determine the size of the matrices is critical to ensure that the matrices are rectangular (not square). Notice that each successive matrix chain will potentially contain relatively larger matrices than the previous one. In the j^{th} matrix chain, the dimensions of the matrices $\langle A_1, \ldots, A_{10} \rangle$ are defined as $P = (p_{j,0}, p_{j,1}, p_{j,2}, \ldots, p_{j,10})$. That is, for a given matrix chain, the first matrix A_1 has dimensions (p_0, p_1), the second matrix A_2 has dimensions (p_1, p_2), and the dimensions for A_{10} is (p_9, p_{10}).
- Use the system clock to measure time t_1, run the algorithms: **Algorithm-1**, **Chain-Matrix-Multiplication-Algorithm**, measure time t_2, calculate the time it takes to complete the execution of the algorithm (i.e., $t_2 - t_1$). `Algorithm-1` multiplies two matrices of size $(n \times m)$ and $(m \times p)$ to produce a new matrix of size $(n \times p)$. You

must perform chain multiplication by carrying out a sequence of pairwise multiplications and accumulating results. For instance, if $n = 3$ and the matrix chain is $\langle A_1, A_2, A_3 \rangle$, then your driver program will use Algorithm-1 to multiply A_1 and A_2 to produce a matrix of size (p_0, p_2) and then use Algorithm-1 again to multiply the result (i.e., $A_1.A_2$) with A_3 that has the dimensions (p_2, p_3) to produce a final matrix of size (p_0, p_3).

- The `MATRIX-CHAIN-ORDER` algorithm listed in this document (and also on page 378 of the textbook) computes the optimal parenthesization that minimizes the number of necessary scalar multiplications. Your task is to (a) implement the provided `MATRIX-CHAIN-ORDER` algorithm, (b) develop and implement a `Chain- Matrix-Multiply` algorithm that uses the information generated by `MATRIX-CHAIN-ORDER` to perform chain multiplication in the order determined by the optimal parenthesization. **Hint**: The `PRINT-OPTIMAL-PARENS` algorithm presented on page 381 of the textbook suggests a `recursive strategy` that can be adapted for the `Chain-Matrix- Multiply` algorithm.
- The performance analysis experiment should produce and print a data frame with 19 rows. In each row, the first column is the matrix chain ID (i.e., $(2 \leq j \leq 20)$, the second column is the upper bound on the matrix size (i.e., $j \times 10$), the third column is the performance (i.e., execution time) of the **sequential** multiplication of the matrices in the order they are listed in the matrix chain, and the last column is the execution time of the **optimal** order determined by the `MATRIX-CHAIN-ORDER` algorithm. Next, compare the performance of the sequential and optimal algorithms using a line plot visualization as a function of the matrix size upper bound.

SUBMISSION INSTRUCTIONS: If you have any doubts/questions, ask the instructor or the TA before you code. We expect that you know how to program by now. So, the instructor or TA will not debug your source code for you. We can help you with logic and algorithms.

Your submission must include:

- The `Jupyter Notebook` that is properly annotated using the `Markdown` cells that explain the objective of the scripts at each step and the input/output constraints – if you reuse code fragments from another source such as the Generative AI (e.g., ChatGPT, CoPilot), text, web site, etc., clearly identify the fragments and their source in comments)
- a Word (or PDF) document that includes the following:
  - Details of the complexity order calculation. For each algorithm, make sure to present the asymptotic growth rate in big-Oh notation. For iterative algorithms, perform approximate analysis and derive the complexity as a polynomial function of the input size. For recursive algorithms, explicitly show the use of the appropriate method (e.g., Master Method) in calculating theoretical complexity.

- The graphs and an explanation of what they show: Did the actual/empirical time match each algorithm's predicted/theoretical complexity as the input size increased? Why or why not? In the chain-matrix multiplication problem, does the optimal solution outperform the sequential multiplication, and by how much?

Submit via Canvas, following the instructions below (you may lose points otherwise), all required items before the deadline on the due date. Late submissions will not be graded. Do not send any executables.

- Include all files as a single zipped attachment.
- Include in your submission a README file that should contain your name, the names of all files in the zipped archive, an explanation of what `Jupyter` environment and `Python` kernel used, and the following certification statement (verbatim): "I certify that I wrote the code I am submitting. I did not copy the whole or parts of it from another student or have another person write the code for me. Any code I am reusing or generated with Artificial Intelligence aid in my program is marked as such with its source identified in comments." Without this certification, your grade will be zero
- If you used Generative AI technology, you should also include the following statement: "The author(s) would like to acknowledge the use of [Generative AI Tool Name], a language model developed by [Generative AI Tool Provider], in the preparation of this assignment. The [Generative AI Tool Name] was used in the following way(s) in this assignment [e.g., brainstorming, grammatical correction, citation, which portion of the assignment]."

You will receive an email if the TA experiences difficulty accessing your attachments. You must check your Canvas course account and sort out any such problem cooperatively with the TA. If this is not done promptly, your homework will not be graded.

Caution: Your work should follow the academic integrity guidelines stated in the syllabus. Do your own work; do not copy that of another. If we suspect copying, we will compare submissions using plagiarism detection tools. If AI technology is used, then you should explicitly and clearly explain what parts of the code and analyses were produced by an AI system and how you modified and customized the solutions generated by AI. If we find evidence of copying, besides giving you a zero for the assignment, we will refer your case to the university administration, and your certification will be used in the proceedings.

GRADING POLICY: We will test your program with our own **input.txt** file to verify the matrix multiplication results and execute all the scripts to regenerate the results and graphs. Your grade will depend on how your program works when we test it. We will not try to debug your program; you are responsible for sending a correct program/script that will compile (or be interpreted), run, not crash, and produce the correct output. Programs that are not well-organized, not commented properly using Markdown cells in the `Jupyter Notebook`, and not

written following the directions are likely to lose points. If you make additional assumptions about the input beyond the ones stated in this document, your program may not run correctly during our test.

The homework will be graded out of a maximum of 100 points. The minimum requirement is to turn in everything asked for. If you do not meet this requirement, you will get 0 points. If you do, your grade will be determined as follows:

- Matrix multiplication problem (45 pts)
  - 4 points for correct complexity order calculation of each algorithm: total 20 points max.
  - Experiment-1 (10 pts) – 10 points for the data frame listing, graph visualization, and explanation: total 10 points max
  - Experiment-2 (10 pts) – 10 points for the data frame listing, graph visualization, and explanation: total 10 points max
  - The use of `markdown cell`/text that explain each related step in the `Jupyter Notebook` (5 pts)
- Chain-Matrix Multiplication via Dynamic Programming (15 pts)
  - Implementation of the `MATRIX-CHAIN-ORDER` algorithm (5 pts)
  - Development and implementation of the optimal `Chain-Matrix-Multiply` algorithm that uses the information generated by `MATRIX-CHAIN-ORDER` (5pts)
  - The use of `markdown` cell/text that explains each related step in the `Jupyter Notebook` (5 pts)
- If your `Jupyter Notebook` scripts fail, you will get 0 out of the remaining 40 points.
- If the scripts run but produce no appropriate/expected output or completely wrong output, you will get 10 points.
- If the output is partially correct, you will get a partial grade higher than 10.
- If the output is fully correct, you will get an additional 40 points.

algorithm-1(A, B, C, p, q, r)
input: A is p x q, B is q x r, and C is p x r matrix

```
1 for i = 1 to p
2    for j = 1 to r
3        sum = 0
4        for k = 1 to q
5            sum = sum + Aik*Bkj
6        Cij = sum
```

algorithm-2(A, B, C, p, q, q, T=5)
input: A is p x q, B is q x r, and C is p x r matrix, T is step size

```
1 for I = 1 to p in steps of T
2    for J = 1 to r in steps of T
3        for K = 1 to q in steps of T
4            for i = I to min(I+T, p)
5              for j = J to min(J+T, r)
6                  sum = 0
7                  for k = K to min(K+T, q)
8                        sum = sum +  Aik*Bkj
9                  Cij = Cij + sum
```

algorithm-3(A, B, C, p, q, r)
input: A is p x q, B is q x r, and C is p x r matrix
strategy -- splits matrices in two instead of four submatrices (as is the case in algorithm-4). Splitting a matrix means dividing it into two parts of equal size, or as close to equal sizes as possible in the case of odd dimensions

if max(p,q,r) < 8 ---- you can choose another threshold)
  algorithm-1(A, B, C, p, q, r) ---- use the iterative algorithm
else
  if max(p,q,r) = p, split A horizontally:

$$C = \begin{pmatrix} A_1 \\ A_2 \end{pmatrix} B = \begin{pmatrix} A_1 B \\ A_2 B \end{pmatrix}$$

  else if max(p,q,r) = r, split B vertically

$$C = A(B_1 \quad B_2) = (AB_1 \quad AB_2)$$

  else split A vertically and B horizontally

$$C = (A_1 \quad A_2) \begin{pmatrix} B_1 \\ B_2 \end{pmatrix} = A_1 B_1 + A_2 B_2$$

algorithm-4(A, B, C)
input: A is 2^n x 2^n , B is 2^n x 2^n , and C is 2^n x 2^n matrix
strategy – block partitioning - works for all square matrices whose dimensions are powers of two, i.e., the shapes are $2^n \times 2^n$ for some *n >= 0*.

if n =0
C = [a_{11}*b_{11}]
else
partition A and B into 4 equal size blocks, each one of which is a $2^{n/2} \times 2^{n/2}$ matrix

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where

$$\begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix} = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix} = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

algorithm-5(A, B, C)
input: A is 2^n x 2^n , B is 2^n x 2^n , and C is 2^n x 2^n matrix

if n < 3
    algorithm-1(A, B, C) ---- use the iterative algorithm
else
    partition A and B into 4 equal size blocks, each one of which is a $2^{n/2} \times 2^{n/2}$ matrix

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix}, B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

$$C = \begin{pmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{pmatrix}$$

where

$$\begin{aligned} M_1 &= (A_{11} + A_{22}) \times (B_{11} + B_{22}) \\ M_2 &= (A_{21} + A_{22}) \times B_{11}; \\ M_3 &= A_{11} \times (B_{12} - B_{22}); \\ M_4 &= A_{22} \times (B_{21} - B_{11}); \\ M_5 &= (A_{11} + A_{12}) \times B_{22}; \\ M_6 &= (A_{21} - A_{11}) \times (B_{11} + B_{12}) \\ M_7 &= (A_{12} - A_{22}) \times (B_{21} + B_{22}) \end{aligned}$$

$$\begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix} = \begin{bmatrix} M_1 + M_4 - M_5 + M_7 & M_3 + M_5 \\ M_2 + M_4 & M_1 - M_2 + M_3 + M_6 \end{bmatrix}$$

(from page 378 of the textbook)

```
MATRIX-CHAIN-ORDER(p, n)
 1  let m[1:n, 1:n] and s[1:n-1, 2:n] be new tables
 2  for i = 1 to n                          // chain length 1
 3      m[i, i] = 0
 4  for l = 2 to n                          // l is the chain length
 5      for i = 1 to n - l + 1              // chain begins at A_i
 6          j = i + l - 1                   // chain ends at A_j
 7          m[i, j] = ∞
 8          for k = i to j - 1              // try A_{i:k} A_{k+1:j}
 9              q = m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j
10              if q < m[i, j]
11                  m[i, j] = q             // remember this cost
12                  s[i, j] = k             // remember this index
13  return m and s
```