Problem Set 5

Due date: Electronic submission of the pdf file of this homework is due on **2/23/2025 before 11:59pm** on canvas.

Name: Chayce Leonard

Resources. Class Textbook
Wikipedia
GeeksForGeeks (Dynamic Programming Article)

On my honor, as an Aggie, I have neither given nor received any unauthorized aid on any portion of the academic work included in this assignment. Furthermore, I have disclosed all resources (people, books, web sites, etc.) that have been used to prepare this homework.

Signature: Chayce Leonard _____

Make sure that you describe all solutions in your own words. Typesetting in LaTeX is required. Read chapters 14 and 15 in our textbook.

Problem 1 (20 points)**.** Determine an LCS of $X = \langle R, H, U, B, A, R, B \rangle$ and $Y = \langle S, T, R, A, W, B, E, R, R, Y \rangle$ using the dynamic programming algorithm that was discussed in class. Make sure that you explain your answer step-by-step, in detail, rather than just giving an LCS. [Make sure that you list X vertically, and Y horizontally when constructing the table. If $x_i \neq y_j$, and $c[i-1, j] = c[i, j-1]$, choose $c[i-1, j]$. Explain row-by-row how the table is constructed.]

Why do we care? For every algorithm, you should make sure that you can work it out step-by-step.

Solution. To find the Longest Common Subsequence (LCS) of $X = \langle R, H, U, B, A, R, B \rangle$ and $Y = \langle S, T, R, A, W, B, E, R, R, Y \rangle$, we'll use the dynamic programming algorithm discussed in class. We'll construct a table $c[i, j]$ where i represents the index in X and j represents the index in Y. The value in each cell $c[i, j]$ will represent the length of the LCS for the prefixes $X[1..i]$ and $Y[1..j]$ [[4]].

Now, let's construct the table step-by-step, explaining each row:

		S	T	R	A	W	B	E	R	R	Y
	0	0	0	0	0	0	0	0	0	0	0
R	0	0	0	1	1	1	1	1	1	1	1
H	0	0	0	1	1	1	1	1	1	1	1
U	0	0	0	1	1	1	1	1	1	1	1
B	0	0	0	1	1	1	2	2	2	2	2
A	0	0	0	1	2	2	2	2	2	2	2
R	0	0	0	2	2	2	2	2	3	3	3
B	0	0	0	2	2	2	3	3	3	3	3

Let's explain the construction of each row:

1. **Row 0 (Initialization):** We initialize the first row and column with 0s, as the LCS of any string with an empty string is 0 [[5]].

2. **Row 1 (R):** We compare 'R' with each character in Y. When we reach the 'R' in Y (column 3), we get a match, so we add 1 to the value in the cell diagonally up and left ($0 + 1 = 1$). This value propagates to the right as it's the maximum.

3. **Row 2 (H):** 'H' doesn't match any character in Y, so we take the maximum of the cell above or to the left for each position. The row remains the same as the previous row.

4. **Row 3 (U):** Similar to 'H', 'U' doesn't match any character, so this row is identical to the previous one.

5. **Row 4 (B):** 'B' matches with 'B' in Y (column 6). At this position, we add 1 to the value diagonally up and left ($1 + 1 = 2$). This new maximum propagates to the right.

6. **Row 5 (A):** 'A' matches with 'A' in Y (column 4). We get a new maximum of 2 at this position, which continues to the right.

7. **Row 6 (R):** 'R' matches twice in Y (columns 3 and 9). The first match doesn't increase the maximum, but the second match does, giving us a new maximum of 3 which propagates to the right.

8. **Row 7 (B):** 'B' matches with 'B' in Y (column 6), increasing the maximum to 3, which continues to the end of the row.

Now that we have constructed the table, we can find the LCS by backtracking from the bottom-right cell $(7, 10)$ [[6]]:

- Start at $c[7, 10] = 3$
- Move diagonally up-left when $x_i = y_j$, otherwise move up or left to the larger value
- The path: $(7, 10) \to (6, 9) \to (5, 4) \to (4, 6)$
- Characters on this path: B, R, A, B

Therefore, an LCS of X and Y is $\langle B, R, A, B \rangle$.

Note: There might be multiple valid LCSs of the same maximum length. The one we found is determined by our choice to prefer $c[i-1, j]$ when $x_i \neq y_j$ and $c[i-1, j] = c[i, j-1]$, as specified in the problem statement.

Problem 2 (20 points)**.** Find an optimal parenthesization of a matrix-chain product whose sequence of dimensions is $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$. Explain how you found the solution. [Hint: You might find it worthwhile to implement the dynamic programming algorithm, and print the relevant tables to aid in your explanations. However, you should make sure that you are able to solve it by hand as well.]

Solution. To find an optimal parenthesization of the matrix-chain product with dimensions $\langle 5, 10, 3, 12, 5, 50, 6 \rangle$, we'll use the dynamic programming approach. This method involves creating two tables: $m[i][j]$ for the minimum number of scalar multiplications, and $s[i][j]$ for the split points.

Step 1: Define the matrices

We have six matrices with the following dimensions:

- $A_1 : 5 \times 10$
- $A_2 : 10 \times 3$
- $A_3 : 3 \times 12$

- $A_4 : 12 \times 5$
- $A_5 : 5 \times 50$
- $A_6 : 50 \times 6$

Step 2: Create and fill the dynamic programming tables

We fill the tables m and s for increasing chain lengths from 2 to 6. The m table stores the minimum number of scalar multiplications, while the s table stores the optimal split points.

The m table:

	1	2	3	4	5	6
1	0	150	330	405	1655	2010
2		0	360	330	2430	1950
3			0	180	930	1770
4				0	3000	1860
5					0	1500
6						0

The s table:

	1	2	3	4	5	6
1	0	1	1	1	3	1
2		0	2	2	3	3
3			0	3	3	3
4				0	4	4
5					0	5
6						0

Step 3: Interpret the results

The minimum number of scalar multiplications needed is $m[1][6] = 2010$.
To find the optimal parenthesization, we trace back through the s table:

- Start at $s[1][6] = 1$, which means split between A_1 and A_2
- For the right part, look at $s[2][6] = 3$, which means split between A_3 and A_4
- For the rightmost part, $s[4][6] = 4$, which means split between A_4 and A_5

Step 4: Optimal parenthesization

The optimal parenthesization is: $((A_1A_2)((A_3A_4)(A_5A_6)))$
This means the optimal order of matrix multiplication is:

1. Multiply A_1 and A_2
2. Multiply A_3 and A_4
3. Multiply A_5 and A_6
4. Multiply the result of (A_3A_4) with (A_5A_6)

5. Finally, multiply (A_1A_2) with the result of step 4

Explanation of the process:

The dynamic programming algorithm works by considering all possible ways to split the matrix chain and choosing the one that minimizes the number of scalar multiplications. It starts with smaller subchains and builds up to the full chain.

For example, $m[1][2] = 150$ because multiplying $A_1(5 \times 10)$ with $A_2(10 \times 3)$ requires $5 \times 10 \times 3 = 150$ scalar multiplications.

The algorithm then considers longer chains. For instance, when computing $m[1][3]$, it compares:

- $(A_1A_2)A_3$: $m[1][2] + 5 \times 3 \times 12 = 150 + 180 = 330$
- $A_1(A_2A_3)$: $m[2][3] + 5 \times 10 \times 12 = 360 + 600 = 960$

It chooses the smaller value (330) and records the split point (1) in $s[1][3]$.

This process continues until the entire chain is considered, resulting in the optimal solution of 2010 scalar multiplications.

The dynamic programming approach efficiently solves this problem by avoiding redundant calculations and systematically building the solution from smaller subproblems to the complete problem.

Problem 3 (20 points)**.** Use a proof by induction to show that the solution to the recurrence

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

is $\Omega(2^n)$.

Why do we care? This is a simple lower bound on the number of parenthesizations of a chain of n matrices. It will remind you about the true meaning of $\Omega(2^n)$. The result serves as a reminder why a brute-force solution is not attractive.

Solution. We will prove by induction that the solution to the recurrence

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

is $\Omega(2^n)$. This means we need to show that there exist positive constants c and n_0 such that $P(n) \geq c \cdot 2^n$ for all $n \geq n_0$.

Base Case: For $n = 1$, we have $P(1) = 1 = 2^0 = 2^1/2$. Let $c = 1/2$, then $P(1) \geq c \cdot 2^1$.

Inductive Hypothesis: Assume that for some $k \geq 1$ and for all $1 \leq i \leq k$, we have $P(i) \geq c \cdot 2^i$, where $c = 1/2$.

Inductive Step: We need to prove that $P(k+1) \geq c \cdot 2^{k+1}$.

For $k + 1 \geq 2$, we have:

$$
\begin{aligned}
P(k+1) &= \sum_{i=1}^{k} P(i)P(k+1-i) \\
&\geq \sum_{i=1}^{k} (c \cdot 2^i)(c \cdot 2^{k+1-i}) \quad \text{(by inductive hypothesis)} \\
&= c^2 \cdot 2^{k+1} \sum_{i=1}^{k} 1 \\
&= c^2 \cdot 2^{k+1} \cdot k \\
&= \frac{1}{4} \cdot 2^{k+1} \cdot k \quad \text{(since } c = 1/2\text{)} \\
&\geq \frac{1}{2} \cdot 2^{k+1} = c \cdot 2^{k+1} \quad \text{(for } k \geq 2\text{)}
\end{aligned}
$$

Thus, we have shown that $P(k+1) \geq c \cdot 2^{k+1}$ for $k \geq 2$, which means the statement holds for all $n \geq 3$.

Conclusion: By the principle of mathematical induction, we have proved that $P(n) \geq c \cdot 2^n$ for all $n \geq 1$, where $c = 1/2$ and $n_0 = 1$. Therefore, $P(n) = \Omega(2^n)$.

This lower bound demonstrates that the number of parenthesizations for a chain of n matrices grows at least exponentially with n. Consequently, a brute-force approach to finding the optimal parenthesization would be highly inefficient, as it would need to consider $\Omega(2^n)$ different possibilities, leading to exponential time complexity.

Problem 4 (20 points)**.** Let $R(i, j)$ be the number of times that table entry $m[i, j]$ is referenced while computing other table entries in a call of MATRIX-CHAIN-ORDER, see [CLRS, Section 14.2]. Show that the total number of references for the entire table is

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3}.$$

[Hint: Re-read the definition of $R(i, j)$ a couple of times.]

Why do we care? This is a key statistics for the run-time of the dynamic-programming solution. The manipulation of such sums is an essential skill that you need to train.

Solution. To prove that

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \frac{n^3 - n}{3},$$

we'll follow a step-by-step approach based on the structure of the MATRIX-CHAIN-ORDER algorithm.

Step 1: Understanding $R(i, j)$

$R(i, j)$ represents the number of times the table entry $m[i, j]$ is referenced while computing other entries in the MATRIX-CHAIN-ORDER algorithm. From the algorithm's structure, we know that $m[i, j]$ is referenced when computing $m[k, j]$ for all $k < i$.

Step 2: Expressing $R(i, j)$

Based on this understanding:

- $R(i, j) = i - 1$ for $i < j$,
- $R(i, i) = 0$ for all i.

Step 3: Formulating the double summation

We can express the double summation as:

$$\sum_{i=1}^{n} \sum_{j=i}^{n} R(i, j) = \sum_{i=1}^{n} \sum_{j=i+1}^{n} (i - 1) + \sum_{i=1}^{n} R(i, i).$$

The second sum $\sum_{i=1}^{n} R(i, i) = 0$ because $R(i, i) = 0$ for all i.

Step 4: Simplifying the summation

We can rewrite the first sum as:

$$\sum_{i=1}^{n} \sum_{j=i+1}^{n} (i - 1) = \sum_{i=1}^{n-1} (n - i)(i - 1).$$

This is because for each i, the inner sum runs $(n - i)$ times.

Step 5: Expanding the summation

$$\begin{aligned} \sum_{i=1}^{n-1} (n - i)(i - 1) &= \sum_{i=1}^{n-1} (ni - i^2 - n + i) \\ &= n \sum_{i=1}^{n-1} i - \sum_{i=1}^{n-1} i^2 - n(n - 1) + \sum_{i=1}^{n-1} i. \end{aligned}$$

Step 6: Using known summation formulas

We can use the well-known formulas:

$$\sum_{i=1}^{n-1} i = \frac{n(n - 1)}{2}, \quad \sum_{i=1}^{n-1} i^2 = \frac{(n - 1)n(2n - 1)}{6}.$$

Step 7: Substituting and simplifying

Substituting these formulas and simplifying:

$$
\begin{aligned}
& n\frac{n(n-1)}{2} - \frac{(n-1)n(2n-1)}{6} - n(n-1) + \frac{n(n-1)}{2} \\
&= \frac{3n^2(n-1) - (n-1)n(2n-1) - 6n(n-1) + 3n(n-1)}{6} \\
&= \frac{(n-1)(3n^2 - n(2n-1) - 3n)}{6} \\
&= \frac{(n-1)(n^2-2n)}{6} \\
&= \frac{n^3 - 3n^2 + 2n}{6} \\
&= \frac{n(n^2-3n+2)}{6} \\
&= \frac{n(n-1)(n-2)}{6} \\
&= \frac{n^3-n}{3}.
\end{aligned}
$$

Conclusion:
Thus, we have proven that

$$\sum_{i=1}^{n}\sum_{j=i}^{n} R(i,j) = \frac{n^3-n}{3}.$$

This result aligns with the cubic time complexity $O(n^3)$ of the MATRIX-CHAIN-ORDER algorithm, as demonstrated in the algorithm analysis. The total number of references to table entries grows as a cubic function of n, with the $-n$ term becoming negligible for large n. This confirms the algorithm's $O(n^3)$ complexity and provides insight into its performance characteristics, particularly its efficiency for small to medium-sized inputs and potential limitations for very large matrix chains.

Problem 5 (20 points)**.** Give an $O(n^2)$-time algorithm to find the longest monotonically increasing subsequence of a sequence of n numbers.

Why do we care? You should learn how to apply the algorithms from the lecture. This is a good opportunity to hone your problem solving skills. Make sure that you solve it yourself without any help!

Solution. Longest Increasing Subsequence (LIS) Problem

The problem is to find the longest monotonically increasing subsequence of a sequence of n numbers using an $O(n^2)$ time algorithm.

Algorithm Description: We use a dynamic programming approach with the following components:

- $dp[i]$: stores the length of LIS ending at index i

- $prev[i]$: stores the previous index in the LIS ending at i

The algorithm proceeds as follows:

1. Initialize $dp[i] = 1$ and $prev[i] = -1$ for all i
2. For each i from 1 to n:
    - For each j from 1 to $i - 1$:
        - If $A[i] > A[j]$ and $dp[i] < dp[j] + 1$:
            - Update $dp[i] = dp[j] + 1$
            - Set $prev[i] = j$
3. Find the index max_index with the maximum value in dp
4. Reconstruct the LIS by tracing back from max_index using $prev$

Complexity Analysis:

- Time Complexity: $O(n^2)$
    - The outer loop runs n times
    - The inner loop runs up to i times for each i
    - This results in a total of $\frac{n(n-1)}{2}$ iterations, which is $O(n^2)$
- Space Complexity: $O(n)$
    - We use two additional arrays (dp and $prev$) of size n

Example Test Cases:

1. Input: $[10, 22, 9, 33, 21, 50, 41, 60, 80]$
   LIS: $[10, 22, 33, 50, 60, 80]$ with length 6
2. Input: $[3, 10, 2, 1, 20]$
   LIS: $[3, 10, 20]$ with length 3
3. Input: $[1, 2, 3, 4, 5]$
   LIS: $[1, 2, 3, 4, 5]$ with length 5
4. Input: $[5, 4, 3, 2, 1]$
   LIS: $[5]$ with length 1
5. Input: $[2, 2, 2, 2]$
   LIS: $[2]$ with length 1

Notes:

- This algorithm is simple to implement and understand.
- However, its $O(n^2)$ time complexity may be slow for large inputs.

- More efficient algorithms exist for larger datasets, such as those with $O(n \log n)$ complexity.

Discussions on canvas are always encouraged, especially to clarify concepts that were introduced in the lecture. However, discussions of homework problems on canvas should not contain spoilers. It is okay to ask for clarifications concerning homework questions if needed. Make sure that you write the solutions in your own words.

Checklist:

☐ Did you add your name?
☐ Did you disclose all resources that you have used?
(This includes all people, books, websites, etc. that you have consulted)
☐ Did you sign that you followed the Aggie honor code?
☐ Did you solve all problems?
☐ Did you submit the pdf file of your homework?