Programming Assignment 4 of CSCI 311

California State University, Chico

December 1st 11:59pm PST on Canvas

1 Assignment Overview

This assignment involves the implementation and analysis of algorithms using both **naive** and **dynamic programming (bottom-up)** approaches. Key aspects include:

- Solving problems with two distinct methodologies.
- Measuring and comparing the performance using the C++ chrono library.
- Documenting the time and space complexities of both approaches.
- Providing empirical data and a theoretical basis to compare the efficiency of the two approaches.
- You also have the freedom in how to design the API for each of the problem whether you use the naive or the bottom up approach. However we ask you to clearly mention in the code as part of the comments what inputs to the API are and what does it mean. For each of the problem I do provide a very high level idea of what these inputs must be and what the output of your function (solution) must be.

2 Performance Measurement and Complexity Analysis

2.1 Using C++ chrono Library

Measure the execution time of both the naive and dynamic programming approaches using the `std::chrono` library. This will provide empirical data for analysis. Keep in mind you may not be able to use the chrono library for large problem size atleast not for the naive approach. This is because the problem may take a very long time to solve or you may also run out of stack space causing stack overflow.

2.2 Complexity Analysis

- Time Complexity: Analyze how the execution time scales with the size of the input. Do this for the naive and for the dynamic programming approach.
- Space Complexity: Evaluate the memory usage in relation to the input size. Do this for the naive and for the dynamic programming approach.

3 Report Writing Guidelines

3.1 Algorithm Explanation

Include clear explanations for both the naive and dynamic programming approaches. Also refer to the notes available on Canvas on Dynamic Programming, in particular refer to the following document: Dynamic Programming by Professor Vazirani.

3.2 Empirical Data Presentation

Present the collected data in a structured format, such as tables or graphs, for clear visualization and comparison.

3.3 Comparative Evaluation

Discuss the optimization techniques used in dynamic programming and their impact on the overall performance compared to the naive approach.

3.4 Report Format

Your report should be well-structured with an introduction, body, and conclusion. In the body for each of the problem, present the time and space complexity of the naive implementation followed by the dynamic programming approach. Mention clearly how the dynamic programming approach is better than the naive implementation. Ensure clarity and conciseness in your writing.

4 Additional Considerations

- Ensure code optimization for accurate performance measurement.
- Use a variety of test cases to evaluate the algorithms comprehensively.
- Comment your code for clarity.
- Reflect on your learning and its applicability to real-world scenarios.

5 Problems

5.1 Matrix Chain Multiplication Problem

5.1.1 Objective

Determine the most efficient way to multiply a series of matrices to minimize scalar multiplications. First let me show you how to multiply any two matrices:

5.1.2 Matrix Multiplication Examples

Example 1: Multiplying a 2×3 Matrix with a 3×2 Matrix

Matrix A is (2×3) i.e., 2 rows and 3 columns:

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \end{pmatrix}$$

Matrix B (3×2) i.e., 3 rows and 2 columns:

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \end{pmatrix}$$

You can only multiply matrix A and B i.e., $A \times B$ if the number of columns of A are equal to the number of rows of matrix B. Also keep in mind that $A \times B \neq B \times A$

Multiplication:
The resultant Matrix C (2×2) for the above example is:
Each element of C is the sum of the products of corresponding elements of a row of A and a column of B.
For instance,

$$c_{11} = a_{11} \cdot b_{11} + a_{12} \cdot b_{21} + a_{13} \cdot b_{31}$$
$$c_{12} = a_{11} \cdot b_{12} + a_{12} \cdot b_{22} + a_{13} \cdot b_{32}$$
$$c_{21} = a_{21} \cdot b_{11} + a_{22} \cdot b_{21} + a_{23} \cdot b_{31}$$
$$c_{22} = a_{21} \cdot b_{12} + a_{22} \cdot b_{22} + a_{23} \cdot b_{32}$$

Scalar Multiplications:
Each element in C requires 3 scalar multiplications.
Total scalar multiplications $= 2 \times 2 \times 3 = 12$.

Example 2: Multiplying a 3×4 Matrix with a 4×2 Matrix
Matrix A (3×4):

$$A = \begin{pmatrix} a_{11} & a_{12} & a_{13} & a_{14} \\ a_{21} & a_{22} & a_{23} & a_{24} \\ a_{31} & a_{32} & a_{33} & a_{34} \end{pmatrix}$$

Matrix B (4×2):

$$B = \begin{pmatrix} b_{11} & b_{12} \\ b_{21} & b_{22} \\ b_{31} & b_{32} \\ b_{41} & b_{42} \end{pmatrix}$$

Multiplication: Resultant Matrix C (3×2):
Each element of C is obtained similarly by multiplying and summing corresponding elements of a row of A and a column of B.
Scalar Multiplications:
Each element in C requires 4 scalar multiplications.
Total scalar multiplications $= 3 \times 4 \times 2 = 24$.

5.1.3 Detailed Explanation of what you are expected to do

Now here is what you need to do for this question on your assignment. The Matrix Chain Multiplication problem involves finding the most efficient order of multiplying a given sequence of matrices. The goal is to minimize the number of scalar multiplications required.

Use Case 1

Input: $p = [10, 20, 30, 40, 30]$
The above input array implies that the following matrices are under consideration:

- A1 (10×20)
- A2 (20×30)
- A3 (30×40)
- A4 (40×30)

Output: 30000 scalar multiplications
Explanation: The specific order that minimizes the number of scalar multiplications is found through dynamic programming, leading to the minimum possible number of multiplications.
Optimal Order: Multiply $(A1 \times A2)$, then multiply the resultant matrix with $A3$, and finally multiply that result with $A4$. First, multiply $A1$ and $A2$: $10 \times 20 \times 30 = 6{,}000$ operations. Multiply the resultant matrix which is of size (10×30) with $A3$: $10 \times 30 \times 40 = 12{,}000$ operations. Finally, multiply the new resultant matrix (10×40) with A4: $10 \times 40 \times 30 = 12{,}000$ operations.
Total Scalar Multiplications: $6,000 + 12,000 + 12,000 = 30{,}000$.

Use Case 2

Input: $p = [30, 35, 15, 5, 10, 20, 25]$
The above input array implies that the following matrices are under consideration:

- A1 (30×35)
- A2 (35×15)
- A3 (15×5)
- A4 (5×10)
- A5 (10×20)
- A6 (20×25)

Output: 15125 scalar multiplications
Explanation: The most efficient way to multiply these matrices requires 15125 scalar multiplications.

Use Case 3

Input: $p = [5, 10, 3, 12, 5, 50, 6]$
The above input array implies that the following matrices are under consideration:

- A1 (5×10)
- A2 (10×3)
- A3 (3×12)
- A4 (12×5)
- A5 (5×50)
- A6 (50×6)

Output: 2010 scalar multiplications
Explanation: The minimum number of scalar multiplications needed to multiply these matrices is 2010.

1. You will write your code for the naive implementation in `Question1A.cpp`
2. You will write your code for the dynamic programming bottom up approach in `Question1B.cpp`

5.2 Rod Cutting Problem

5.2.1 Objective:

Maximize profit by cutting a rod of length n into smaller lengths, given a price table for each length.

5.2.2 Understanding the Problem

- *Basic Concept*: Given a rod of length n and a table of prices for rods of different lengths, the goal is to cut the rod into pieces to maximize total profit.
- *Constraints*:
  - Cuts must be in integer lengths.
  - Each piece can be sold according to the given price table.
  - The total length of the pieces must equal the length of the original rod.
- *Objective*: Find the optimal cut configuration that maximizes total profit.

5.2.3 Use Cases

Use Case 1:

- *Input*: length of the rod = 8 and price array = [1, 5, 8, 9, 10, 17, 17, 20]
- *Objective*: Maximize the total profit from selling the rod pieces.
- *Output*: 22
- *Explanation*: An optimal way is to cut the rod into pieces of lengths 2 and 6, yielding a profit of 22 (5 + 17).

Use Case 2:

- *Input*: length of the rod = 5, price array = [2, 5, 7, 8, 10]
- *Output*: 12
- *Explanation*: Cutting the rod into lengths 2 and 3 yields a profit of 12 (5 + 7), which is the maximum profit.

5.2.4 Approach to Solve the Problem

- *Dynamic Programming*: Solve the problem using dynamic programming, breaking it down into simpler subproblems.
- *Bottom-Up Approach*: Start with the smallest lengths, gradually building up to the solution for the original length while tracking the maximum profit.
- *Record Keeping*: For each length, consider all possible ways to cut the rod and choose the most profitable one.
- *Reconstructing the Solution*: Use the recorded decisions to determine the specific cuts that yield the maximum profit.

1. You will write your code for the naive implementation in `Question2A.cpp`
2. You will write your code for the dynamic programming bottom up approach in `Question2B.cpp`

5.3 Word Break Problem

5.3.1 Objective:

Determine if a given string can be segmented into one or more words found in a provided dictionary.

5.3.2 Details of the Problem

- *Input*: A string 's' and a list of words 'wordDict' that represents the dictionary.
- *Task*: Check if 's' can be segmented into a space-separated sequence of one or more dictionary words.
- *Constraints*:
  - The string 's' and the words in 'wordDict' consist only of lowercase English letters.
  - The same word in the dictionary may be reused multiple times in the segmentation.
  - The dictionary does not contain duplicate words.
- *Output*: Return *True* if the string can be segmented, *False* otherwise.

5.3.3 Use Cases

Use Case 1:

- *Input*: s = "applepenapple", wordDict = ["apple", "pen"]
- *Objective*: Determine if "applepenapple" can be segmented into words from the given dictionary.
- *Output*: True
- *Explanation*: The string can be segmented as "apple pen apple", with all words present in the dictionary.

Use Case 2:

- *Input*: s = "catsandog", wordDict = ["cats", "dog", "sand", "and", "cat"]
- *Output*: False
- *Explanation*: There is no way to segment "catsandog" into a sequence of words from the dictionary without leaving some letters unused.

5.3.4 Approach to Solve the Problem

- *Dynamic Programming*: This problem can be efficiently solved using a dynamic programming approach.
- *Subproblem Definition*: For a substring of 's', determine if it can be segmented into dictionary words. Use this information to solve for larger substrings.
- *Bottom-Up Solution*: Start with smaller substrings and build up to the entire string, keeping track of whether each substring can be segmented.
- *Backtracking*: If the current substring cannot be formed, backtrack and try different segmentations.

1. You will write your code for the naive implementation in `Question3A.cpp`
2. You will write your code for the dynamic programming bottom up approach in `Question3B.cpp`

5.4 Partition Problem

5.4.1 Objective:

Determine whether a given set of integers can be partitioned into two subsets such that the sum of elements in both subsets is equal.

5.4.2 Details of the Problem

- *Input*: A set of integers.
- *Task*: Check if it's possible to divide the set into two subsets with equal sums.
- *Constraints*:
  - The set may contain positive, negative, or zero values.
  - The number of elements in the set is not fixed.
  - The subsets do not need to have an equal number of elements, only equal sums.
- *Output*: Return *True* if such a partition exists, *False* otherwise.

5.4.3 Use Cases

Use Case 1:

- *Input*: set = [1, 5, 11, 5]
- *Output*: True
- *Explanation*: The set can be partitioned into two subsets [1, 5, 5] and [11], both with a sum of 11.

Use Case 2:

- *Input*: set = [1, 2, 3, 5]
- *Output*: False
- *Explanation*: There is no possible partition that divides the set into two subsets with equal sums.

Use Case 3:

- *Input*: set = [2, 3, 5]
- *Output*: True
- *Explanation*: The first set can have 2 and 3 which sums to 5 and the second set can have 5 by itself.

1. You will write your code for the naive implementation in `Question4A.cpp`
2. You will write your code for the dynamic programming bottom up approach in `Question4B.cpp`

5.5 Palindrome Partitioning

5.5.1 Objective:

Find the minimum number of cuts needed to partition a string such that every substring is a palindrome.

5.5.2 Concept

The goal is to split a given string into the fewest possible substrings where each substring is a palindrome (reads the same backward and forward). If the entire input string is already a palindrome (like "racecarracecar"), the minimum number of cuts required is zero because no partitioning is needed to meet the objective.

5.5.3 Use Cases

Use Case 1

- *Input*: s = "racecar"
- *Output*: 0
- *Explanation*: The entire string "racecar" is a palindrome, hence no cuts are needed.

Use Case 2:

- *Input*: s = "noonabbad"
- *Output*: 2
- *Explanation*: An optimal solution is "noon — abba — d", needing two cuts.

Use Case 3:

- *Input*: s = "banana"
- *Output*: 1
- *Explanation*: One solution is "b — anana", requiring one cut.

Approach to Solve the Problem

- *Dynamic Programming*: Use dynamic programming to minimize redundant computations.
- *Palindrome Check*: Implement a function to check if a substring is a palindrome.
- *Subproblem Definition*: Define a subproblem as finding the minimum cuts for a substring.
- *Bottom-Up Solution*: Start with small substrings, expanding to the entire string, optimizing the number of cuts.
- *Optimization*: Store results of subproblems to avoid repeated calculations.

6 Implementation

For each problem:

- Implement the naive solution using recursion or brute force.
- Implement the dynamic programming solution using the bottom-up approach.
- Use C++ for implementation.

7 Empirical Testing

- Use the C++ chrono library to measure the execution time of both solutions.
- Document cases where the naive solution may time out.

8 Report Writing

- Write a report comparing the naive and dynamic programming approaches.
- Discuss improvements in time and/or space complexity.
- Include empirical data and analysis.