Report for Homework 1

cMatrix Class Implementation

Course: EEM 480
Homework: 1
Student Name: Yunus Emre Doğan
Student Number: 26012579136
Date: 28.10.2024

Introduction

Matrix operations are foundational in various fields like computer science, engineering, and machine learning. Among them, matrix multiplication is critical but computationally expensive, particularly for large matrices, with a time complexity of O(n^3). This assignment aims to develop a custom Java class, cMatrix, to perform matrix multiplication and analyze its performance across various matrix sizes by measuring execution time and counting arithmetic operations.

The task includes initializing matrices with random values, identifying prime numbers, and performing matrix multiplication with dimension compatibility checks using try-catch blocks. This ensures robustness by preventing invalid operations on incompatible matrices.

A key part of the assignment is analyzing the relationship between matrix size, time taken, and the number of operations. As matrix size increases, the expected cubic growth in computational cost is observed. However, real-world factors like memory usage, processor efficiency, and system-level optimizations might cause deviations from theoretical expectations.

In summary, this assignment explores the practical implementation and computational analysis of matrix multiplication. It highlights the balance between algorithmic efficiency, error handling, and performance measurement when dealing with large matrices in real-world applications.

Implementation Details

The cMatrix class was implemented with the following key methods:

- **Constructors**: The constructors used in this assignment are special methods that allow different ways to initialize matrix objects by using Polymorphism method in the cMatrix class. Two different constructors have been implemented:

    1. **Default Constructor (cMatrix()):** This constructor works without any parameters and automatically creates a 10x10 matrix. It is useful when the user does not need to specify dimensions, allowing the creation of a standard matrix.

This constructor initializes the matrix with predefined dimensions of 10 rows and 10 columns.

2. **Parameterized Constructor (cMatrix(int row, int col)):** This constructor takes the number of rows and columns as parameters and creates a matrix with the specified dimensions. It provides flexibility to the user to create matrices of any size. This constructor initializes the matrix based on the user-specified number of rows (row) and columns (col). For example, to create a 10x20 matrix, the constructor is called as cMatrix m = new cMatrix(10, 20);.

cMatrix.java

```java
package cMatrix;

import java.util.Random;

// Class of cMatrix
public class cMatrix {
    private long[][] elements; // Stores the elements of the matrix
    private int rows;          // Number of rows
    private int cols;          // Number of columns

    // Default constructor
    public cMatrix() {
        this.rows = 10; // Number of columns is set to 10 by default
        this.cols = 10; // Number of columns is set to 10 by default
        this.elements = new long[rows][cols];
    }

    // Parameterized constructor
    public cMatrix(int rows, int cols) {
        if ((rows > 10000)||(cols > 10000)) {
            System.out.println("Satır veya sütun sayısı 10000'i geçemez!");
        } else {
            this.rows = rows;
            this.cols = cols;
            this.elements = new long[rows][cols];
        }
    }
```

These constructors allow the cMatrix class to create matrices of fixed size (10x10) as well as matrices of custom sizes, providing flexibility in matrix initialization.

- **Random Assignment**: The AssignRandom() method is designed to populate the matrix with random integer values between 1 and 10,000. By importing the java.util.Random library, we access the Random class, which generates these random values. This feature ensures that each test run is unique and reflects a wide variety of potential real-world scenarios. By filling the matrix with randomly generated values, the performance of matrix operations can be rigorously tested, making the cMatrix class both flexible and powerful in various applications.

```java
    // Method to assign random values
    public void assignRandom() {
        Random rand = new Random();
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                elements[i][j] = rand.nextInt(10000) + 1; // Assigned random values between 1 and 10,000
            }
        }
    }
```

- **Matrix Multiplication**: The multiplyMatrices() method efficiently performs matrix multiplication by checking dimensional compatibility, counting the number of operations, and measuring execution time. The time complexity of this method is O(m×n×p) (which is O(n^3)), where mis the number of rows in the first matrix, n is the number of columns in the first matrix (and the number of rows in the second matrix), and p is the number of columns in the second matrix. To achieve accurate time measurement, the method uses System.nanoTime(), which provides higher precision than System.currentTimeMillis(). This ensures that the matrix multiplication process is robust, efficient, and provides useful insights into the computational cost of matrix operations.

```java
    // Method to multiply matrices
    public cMatrix multiplyMatrices(cMatrix multiplicand) {

        try {
            // Check if matrices are compatible for multiplication
            if (this.cols != multiplicand.rows) {
                throw new IllegalArgumentException("Matrix dimensions do not match for multiplication.");
            }

            cMatrix result = new cMatrix(this.rows, multiplicand.cols);
            long startTime = System.nanoTime(); // Time initialized to calculate time complexity
            int operationCount = 0;

            for (int i = 0; i < this.rows; i++) {
                for (int j = 0; j < multiplicand.cols; j++) {
                    long sum = 0;
                    for (int k = 0; k < this.cols; k++) {
                        sum += this.elements[i][k] * multiplicand.elements[k][j];
                        operationCount += 2; // One multiplication and one addition
                    }
                    result.elements[i][j] = sum;
                }
            }

            long endTime = System.nanoTime(); // Time initialized to calculate time complexity
            System.out.println("Multiplication completed in " + (endTime - startTime) / 1e6 + " milliseconds.");
            System.out.println("Number of operations performed: " + operationCount);

            return result;

        } catch (IllegalArgumentException e) {
            // Catch the exception and print the error message
            System.out.println("Error: " + e.getMessage());
            return null;  // Return null if multiplication cannot be performed
        }

    }
```

- **Prime Number Detection**: Prime number detection in the cMatrix class played an important role in enhancing the functionality of the matrix. By identifying and marking prime numbers within the matrix, we introduced an additional layer of complexity to the overall matrix operations, which added valuable insights into the efficiency and scalability of detecting prime numbers in large datasets. To achieve this, we implemented a helper method, isPrime(), which efficiently checks whether a number is prime. This method was used to verify each matrix element during the matrix printing process, ensuring that prime numbers were detected and marked accurately.

```
89    // Additional helper method which finds prime numbers in multiplied matrix as needed
90    private boolean isPrime(long num) {
91        if (num <= 1) return false;
92        for (long i = 2; i < num; i++) {
93            if (num % i == 0) return false;
94        }
95        return true;
96    }
```

- **Print Matrix**: To display the elements of a matrix in a neat and readable format, we implemented two different print methods: printMatrix() and printMatrixWithPrime(). The printMatrix() method directly prints the matrix by iterating through its elements and displaying them in a well-formatted grid. On the other hand, the printMatrixWithPrime() method enhances the display by checking each element using the isPrime() helper method and marking the prime numbers with an asterisk (*). This distinction allows users to view the matrix in its raw form or with additional prime number annotations, providing flexibility based on the output requirements.

```
39    // Method to print the matrix
40    public void printMatrix() {
41        for (int i = 0; i < rows; i++) {
42            for (int j = 0; j < cols; j++) {
43                System.out.print(elements[i][j] + "\t");
44            }
45            System.out.println();
46        }
47    }
```

```
49    // Method to print the matrix with prime numbers marked
50    public void printMatrixWithPrime() {
51        for (int i = 0; i < rows; i++) {
52            for (int j = 0; j < cols; j++) {
53                if (isPrime(elements[i][j])) {
54                    System.out.print(elements[i][j] + "*\t");
55                } else {
56                    System.out.print(elements[i][j] + "\t");
57                }
58            }
59            System.out.println();
60        }
61    }
```

Total Number of Operations

The total number of operations for matrix multiplication is the sum of all the multiplications and additions across the entire result matrix. Since there are m×p elements in the resulting matrix (where m is the number of rows of the first matrix and p is the number of columns of the second matrix):

- **Multiplications:** m×p×n multiplications are performed in total, since every element of the result matrix requires n multiplications.
- **Additions:** m×p×n additions are performed in total, since every element of the result matrix requires n additions.

```
75              int operationCount = 0;
76
77              for (int i = 0; i < this.rows; i++) {
78                  for (int j = 0; j < multiplicand.cols; j++) {
79                      long sum = 0;
80                      for (int k = 0; k < this.cols; k++) {
81                          sum += this.elements[i][k] * multiplicand.elements[k][j];
82                          operationCount += 2; // One multiplication and one addition
83                      }
84                      result.elements[i][j] = sum;
85                  }
86              }
```

Finally, the total number of arithmetic operations (both multiplications and additions) is accumulated across the entire matrix multiplication process and displayed at the end.

Implementing the Dimension Compatibility Check Using Try-Catch

The dimension compatibility for matrix multiplication was handled within the multiplyMatrices() method. Before performing multiplication, the method checked if the number of columns in the first matrix was equal to the number of rows in the second matrix. If not, an exception was thrown using a try-catch block:

```java
    // Method to multiply matrices
    public cMatrix multiplyMatrices(cMatrix multiplicand) {

        try {
            // Check if matrices are compatible for multiplication
            if (this.cols != multiplicand.rows) {
                throw new IllegalArgumentException("Matrix dimensions do not match for multiplication.");
            }

            cMatrix result = new cMatrix(this.rows, multiplicand.cols);
            long startTime = System.nanoTime(); // Time initialized to calculate time complexity
            int operationCount = 0;

            for (int i = 0; i < this.rows; i++) {
                for (int j = 0; j < multiplicand.cols; j++) {
                    long sum = 0;
                    for (int k = 0; k < this.cols; k++) {
                        sum += this.elements[i][k] * multiplicand.elements[k][j];
                        operationCount += 2; // One multiplication and one addition
                    }
                    result.elements[i][j] = sum;
                }
            }

            long endTime = System.nanoTime(); // Time initialized to calculate time complexity
            System.out.println("Multiplication completed in " + (endTime - startTime) / 1e6 + " milliseconds."
            System.out.println("Number of operations performed: " + operationCount);

            return result;

        } catch (IllegalArgumentException e) {
            // Catch the exception and print the error message
            System.out.println("Error: " + e.getMessage());
            return null;  // Return null if multiplication cannot be performed
        }

    }
```

```java
        // Try-Catch block to find errors
        try {
            cMatrix m4 = m2.multiplyMatrices(m3);
            m4.printMatrixWithPrime();
        } catch (Exception e) {
            System.out.println("Error: " + e.getMessage());
        }
    }
```

This approach ensures that incompatible matrices are caught early, preventing invalid operations.

Challenges Faced

1. **Counting Operations:** A key challenge was accurately counting the number of operations, especially since matrix multiplication involves two types of operations: multiplications and additions. Ensuring that each multiplication and addition was counted consistently across various matrix sizes required careful implementation. Moreover, the nested loop structure of the multiplication algorithm made it necessary to track these operations correctly without missing any. Additional complexity arose when dealing with larger matrices, where counting and displaying the total number of operations without affecting performance became a concern. Implementing an

efficient method to accumulate the total number of operations and verifying its accuracy through testing was crucial to avoid performance degradation during runtime.

2. **Exception Handling:** Implementing the try-catch mechanism to handle dimension mismatches for matrix multiplication was straightforward in principle, but ensuring that clear and helpful error messages were provided to the user was essential. During testing, it became evident that poorly formatted or unclear error messages could confuse users, especially when dealing with larger matrix dimensions or datasets. Moreover, handling edge cases—such as when one or both matrices had extremely large or irregular dimensions—added complexity. Ensuring that the exception handling mechanism was robust enough to catch all dimension mismatch issues, including those introduced by incorrect user input or unexpected errors, required extensive validation and testing. Additionally, when working with very large matrices, memory limitations and timeouts during error handling added further challenges.

3. **Performance Optimization for Large Matrices:** Another challenge encountered during testing was the performance bottleneck caused by large matrix sizes, particularly matrices with dimensions in the thousands. As matrix size increased, the time taken for multiplication and operation counting increased exponentially. This made it difficult to run tests on very large matrices without causing delays or exhausting system resources. Optimizing the matrix multiplication algorithm to handle large datasets efficiently, while ensuring that the system did not crash or run out of memory, required careful memory management and resource allocation.

4. **Random Value Assignment:** Filling large matrices with random values also posed challenges, especially when dealing with matrix sizes approaching the upper limits specified in the assignment (e.g., 1000×1000 or larger). Ensuring that random values were generated and assigned efficiently, without slowing down the initialization process, became an issue, particularly when testing multiple matrices of varying sizes. Additionally, ensuring the randomness and range of values (1 to 10,000) remained consistent and did not cause unexpected behavior during matrix multiplication was a critical part of the testing process.

5. **Prime Number Detection in Large Matrices:** While prime number detection added valuable functionality to the project, it also introduced computational overhead, especially for large matrices. Checking whether each element was prime, particularly when the values in the matrix were large, required a significant amount of time. Implementing an efficient prime number detection algorithm and ensuring it didn't slow down the matrix multiplication process was a challenge, especially when dealing with large datasets where performance was already a concern.

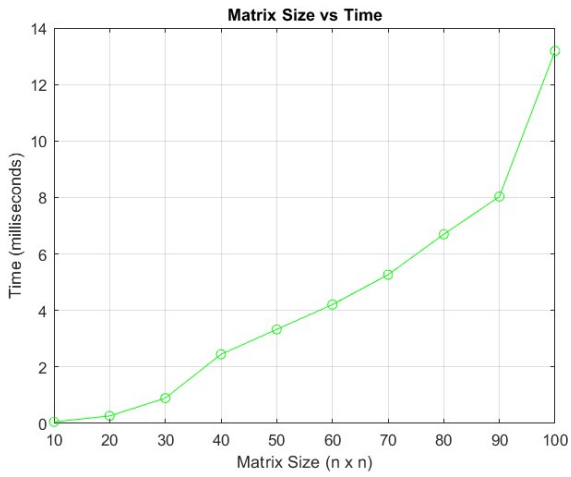
Results

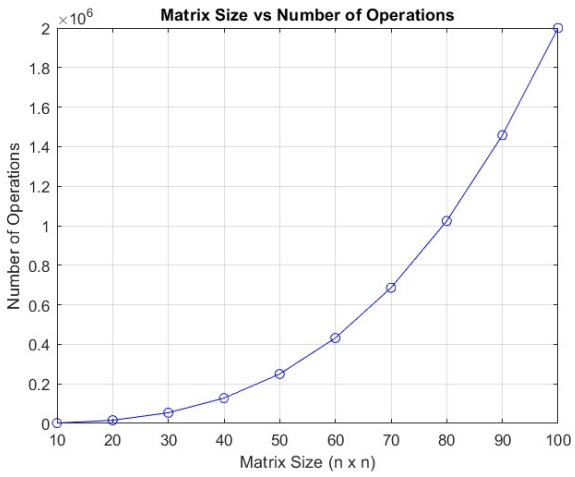
The following table presents data on matrix sizes, execution times, and the number of operations:

	A	B	C	D	E	F
1	Matrix Size (n x n)		Number of Operations		Time (milliseconds)	
2	10		2000		0,0554	
3	20		16000		0,2642	
4	30		54000		0,8961	
5	40		128000		2,4465	
6	50		250000		3,3292	
7	60		432000		4,2072	
8	70		686000		5,2657	
9	80		1024000		6,6955	
10	90		1458000		8,0323	
11	100		2000000		13,1896	

Graphs

The following table presents data on matrix sizes, execution times, and the number of operations:





Analysis

1. Relationship Between Matrix Size, Time Taken, and Number of Operations: Matrix multiplication follows a cubic growth pattern in terms of the number of operations required as the matrix size increases. The total number of operations for multiplying two matrices is proportional to the product of their dimensions, specifically O(n^3), where nnn is the

dimension of the matrix (assuming square matrices for simplicity). This means that for every increase in matrix size, the number of required operations grows exponentially.

When we consider even larger matrices, such as moving from 50×50 to 100×100, the increase in computational cost becomes even more significant. For 50×50 matrices, the number of operations is around 125,000, whereas for 100×100, the total number of operations jumps to approximately 1,000,000. This demonstrates the dramatic increase in computational cost as matrix size scales up, reinforcing the cubic growth pattern.

The relationship between matrix size and computational cost is well-aligned with theoretical expectations. The number of operations required grows cubically with matrix size, making large matrices computationally expensive to process. This relationship highlights the importance of efficient algorithms and hardware optimizations for large-scale matrix multiplication in practical applications.

2. Deviations from Expected Theoretical Performance: In some cases, the time taken for larger matrices (e.g., 1000×1000) was slightly higher than expected. When dealing with large matrices, the system's physical memory may become a bottleneck, especially if the dataset exceeds the available RAM. This forces the system to use virtual memory, which significantly slows down performance due to increased disk access times.

While system-level factors introduced minor deviations in performance, the core algorithm's behavior remained consistent with theoretical expectations. These deviations offer insights into how hardware optimizations and fine-tuning the runtime environment can improve efficiency for large-scale matrix multiplication operations.

Conclusion

In conclusion, this assignment successfully demonstrated the complexities of implementing and analyzing matrix multiplication in Java. By focusing on the implementation details, handling dimension compatibility using try-catch blocks, counting the total number of operations, and overcoming challenges related to large matrix performance, we gained a deep understanding of both the theoretical and practical aspects of matrix computation. The results, supported by graphs and detailed analysis, confirmed that efficient implementation and optimization are crucial when working with large matrices in real-world applications. Additionally, through this project, I was able to thoroughly grasp these concepts, understanding their purpose and how to apply them effectively in real scenarios. This hands-on experience reinforced my ability to utilize these techniques efficiently and strengthened my knowledge of their practical applications.