Assignment 2
Name: Terng Yan Long (唐延龍)
ID: 111550883

1. Results

A. Pure Recursion[1]

	Number of scalar multiplications	Time (ns)
#1	-	Time Limit Exceeded
#2	-	Time Limit Exceeded
#3	-	Time Limit Exceeded
Average	-	Time Limit Exceeded

B. Recursion with memoization

	Number of scalar multiplications	Time (ns)
#1	848199	3812200
#2	848199	3796400
#3	848199	5301700
Average	848199	4303433

C. Nested-for loops

	Number of scalar multiplications	Time (ns)
#1	848199	5256800
#2	848199	5869300
#3	848199	4939600
Average	848199	5355233

```
e0543630@DESKTOP-FM88NND:/mnt/c/Users/yanlo/OneDrive/Desktop/
atrixChainMultiplication/src$ java mcm.Main
B. Recursion with memoization
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 3812200

C. Nested For Loops
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 5256800

e0543630@DESKTOP-FM88NND:/mnt/c/Users/yanlo/OneDrive/Desktop/
atrixChainMultiplication/src$ java mcm.Main
B. Recursion with memoization
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 3796400

C. Nested For Loops
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 5869300

e0543630@DESKTOP-FM88NND:/mnt/c/Users/yanlo/OneDrive/Desktop/
atrixChainMultiplication/src$ java mcm.Main
B. Recursion with memoization
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 5301700

C. Nested For Loops
------------------------------------------------------------
Total Number of Multiplications: 848199
Elapsed Run Time (in nanoseconds): 4939600
```

Figure 1: Screenshot of the obtained result

[1] Initially, the program was executed for 2 days straight but still did not terminate. After much consideration, I decided to label it as “Time Limit Exceeded”, as I feel it was sufficient in displaying the exponential nature of such algorithm.

Assignment 2
Name: Terng Yan Long (唐延龍)
ID: 111550883

2. Discussions and Observations

In this assignment, we were tasked to solve the Matrix Chain Multiplication problem by using 3 different approaches (namely Pure Recursion, Recursion with memoization, and Nested-for loops). The 100 input instances were also given as follows:

```
public static final int[] INPUT = {107, 161, 142, 37, 77, 90, 142, 84, 97, 165, 45, 13, 15, 139, 16, 158, 93, 11,
        155, 42, 127, 92, 164, 190, 115, 156, 44, 85, 56, 41, 186, 39, 80, 171, 160, 195, 65, 37, 119, 14, 93, 198,
        71, 104, 99, 178, 12, 184, 30, 71, 15, 193, 79, 152, 116, 73, 26, 62, 80, 171, 16, 125, 3, 91, 90, 40, 133,
        114, 185, 196, 103, 19, 78, 6, 180, 158, 43, 86, 65, 164, 1, 50, 129, 152, 23, 87, 127, 132, 36, 163, 114,
        65, 43, 91, 150, 9, 135, 25, 96, 22, 81, 150};
```

Figure 2: Input instances[2]

The dimensions for matrix M_i is defined to be INPUT[i-1] x INPUT[i]. In addition, the number of scalar multiplications between 2 matrices (M_i and M_{i+1}) can be easily computed by using the formula INPUT[i-1] * INPUT[i] * INPUT[i+1]. Since matrix multiplication is not commutative in nature, the order in which different matrices are operated on will affect the final result (total number of scalar multiplications). Hence, different approaches have been implemented and will be analysed, in terms of their correctness as well as their efficiency.

2.1 Pure Recursion

The first approach involves the use of the naïve **recursion** method. For a set of n matrices, there are n-1 ways in which the set can be partitioned into 2 separate chains. For example, the set containing matrices {A, B, C} can be partitioned into either {(A, B), C} or {A, (B, C)}, which equates to 3 – 1 = 2 ways. Subsequently, each sub-groups can be further partitioned by using the same process, which leads to recursion.

In my implementation, the main recurse function takes in 3 different parameters (i, j, arr), which represent the start index, the end index, and the input array respectively. The base case can be trivially defined since the multiplication of a single matrix does not contribute to the total number of operations. Next, the outer for-loop is defined such that each possible partition is iterated in order to obtain the minimum number of operations required. The result for each partition is simply the sum of the result of the 2 sub-problems, as well as the number of multiplication needed to "combine" the resulting matrices from the sub-problems.

[2] Even though input instances are provided for 101 matrices, only the first 100 will be used in this assignment.

Assignment 2
Name: Terng Yan Long (唐延龍)
ID: 111550883

```java
// Base Case: Only 1 matrix left -> number of operations = 0
if (i == j) {
    return 0;
}
int count = Integer.MAX_VALUE;
for (int k = i; k < j; k++) {
    int recurseOperations = recurse(i, k, arr) + recurse( i: k+1, j, arr);
    int currOperations = arr[i-1] * arr[k] * arr[j];
    int totalOperations = currOperations + recurseOperations;

    if (totalOperations < count) {
        count = totalOperations;
    }
}
return count;
```

Figure 3: Implementation of recursion-based approach

This process can be modelled as a tree-like structure as follows:

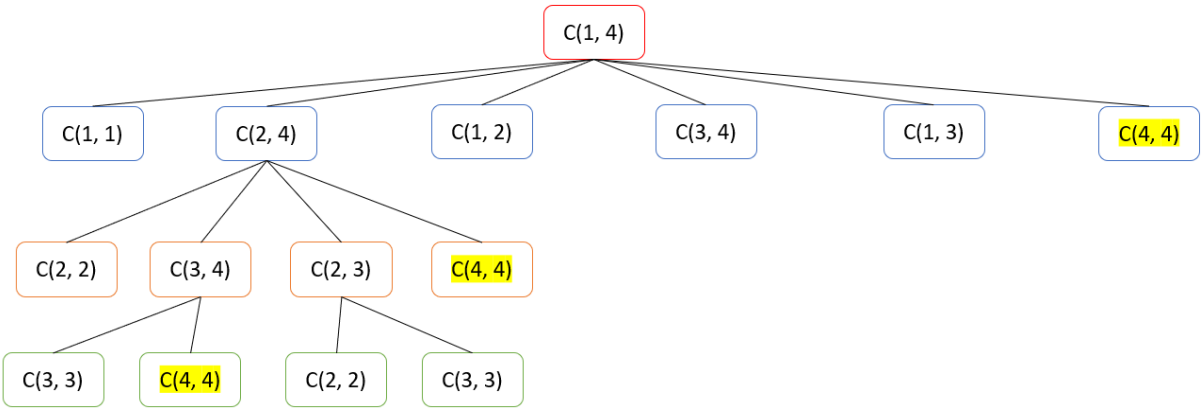

Figure 4: Part of the solving process for a simpler 4 matrices problem

As seen in this diagram, the minimum cost for computing $M_1 * M_2 * M_3 * M_4$ requires repeated computations of C(4,4). In this case, each matrix can either be used or unused as a partition. Since there are n matrices and each matrix consist of two choices, it will result in a time complexity of $O(2^n)$, where n is the number of matrices. The exponentiality is due to the top-down nature of recursion as well as the additional cost incurred from performing redundant work. From the Results section, it is observed that the computation takes a large amount of time, which coincides with the expected theoretical time complexity.

On the other hand, the space complexity would be O(1), as no additional memory is required.

Assignment 2
Name: Terng Yan Long (唐延龍)
ID: 111550883

2.2 Recursion with Memoization

The next approach involves the use of memoization to eliminate redundant duplicated computations. This can be achieved by introducing an additional table (in the form of a two-dimensional (2D) array) to store the intermediate results from each sub-problem.

The main idea of the algorithm is similar to the Pure Recursion implementation. However, the only significant difference is that the table is referenced before each computation. In the event that the corresponding cell already contains the required value, the result can be used directly, and the following recursive computation can be skipped. During each recursive process, the table is updated whenever the current result is lower than the stored value.

```
// Base Case: Only 1 matrix left -> number of operations = 0
if (i == j) {
    return 0;
}

// Check if stored value exists in the table
if (memoizedTable[i][j] != -1) {
    return memoizedTable[i][j];
}

memoizedTable[i][j] = Integer.MAX_VALUE;
for (int k = i; k < j; k++) {
    int recurseOperations = memoize(i, k, arr) + memoize( i: k+1, j, arr);
    int currOperations = arr[i-1] * arr[k] * arr[j];
    int totalOperations = currOperations + recurseOperations;

    if (totalOperations < memoizedTable[i][j]) {
        memoizedTable[i][j] = totalOperations;
    }
}
return memoizedTable[i][j];
```

Figure 5: Implementation of memoization-based approach

In this case, the time complexity would be $O(n^3)$, where n is the number of matrices, as mentioned in the lecture slides. This is because, it takes O(n) to calculate each entry in the table (given by the for loop), and there is a total of n^2 entries in the table.[3] On the other hand, the space complexity would be $O(n^2)$ due to the use of the 2D array.

[3] To be more accurate, we will only need to fill $\frac{n*(n-1)}{2}$ entries, in which i <= j

Assignment 2
Name: Terng Yan Long (唐延龍)
ID: 111550883

2.3 Nested-for loops

Lastly, the bottom-up iterative approach consisting of nested-for loops can be used as well. In this approach, every cell in the table is computed and filled in, starting with the smallest sub-problem and eventually building up to form the final result. Similar to the previous approach, the current value is computed by referring to existing values in the table.

This process can be illustrated with a simple example consisting of 5 matrices (Figure 6). The value of each cell denotes the minimum multiplication costs of the matrices in the specified range. For example, the value in [1, 3] represents the total number of multiplication needed to combine the first 3 matrices. In addition, the cells are filled diagonally as seen from the coloured arrows, in the order of yellow, red, blue, and finally, purple which denotes the final result.

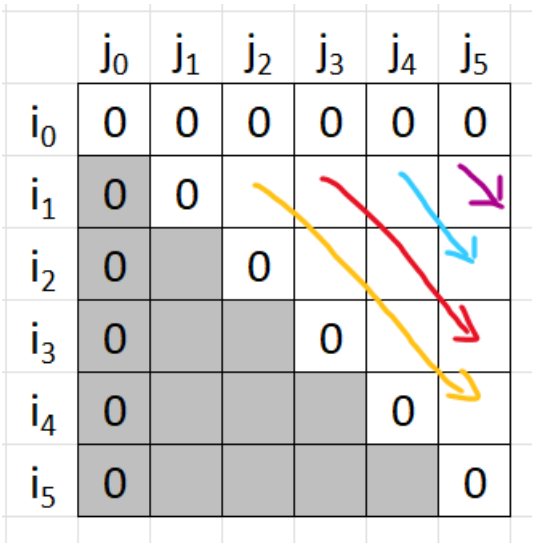

Figure 6: General solving process for the iterative loop-based approach

This approach can be implemented by using 3 nested for-loops:

```
public static int iterate(int start, int end, int[] arr) {
    // Let diff be the difference between the value of the current row and column
    for (int diff = 1; diff <= NUMBER_OF_MATRICES - 1; diff++) {
        for (int i = start; i <= NUMBER_OF_MATRICES - diff; i++) {
            int j = i + diff;
            iterativeTable[i][j] = Integer.MAX_VALUE;
            for (int k = i; k < j; k++) {
                int currOperations = iterativeTable[i][k] + iterativeTable[k+1][j] + (arr[i-1] * arr[k] * arr[j]);

                if (currOperations < iterativeTable[i][j]) {
                    iterativeTable[i][j] = currOperations;
                }
            }
        }
    }
    return iterativeTable[1][NUMBER_OF_MATRICES];
```

Figure 7: Implementation of iterative loop-based approach

In this case, the time complexity would also be $O(n^3)$, where n is the number of matrices. This can be derived by using the same explanation from the previous section. In addition, since each loop runs for a maximum of n times, $n*n*n = n^3$ would be a good guess such that it results in a sufficiently tight bound.

Likewise, the space complexity would be $O(n^2)$.

Conclusion

Even though the pure recursion approach seems more intuitive and easier to implement, it is significantly more inefficient as reflected in the total computational time at the beginning of this report. To overcome this issue, dynamic programming can be introduced such that the time complexity is reduced from $O(2^n)$ to $O(n^3)$, resulting in a much faster computation while taking advantage of the optimal sub-structure of such a problem.

Although the time complexity of the memoization and the iterative approach are both $O(n^3)$ seems to be the same, the memoization method proved to be slightly more efficient (albeit not that significant). This is because the constant coefficient might be different and is “left out” when deriving the time complexity. Various factors can result in such an observation, such as not needing to fill up the entire table in the memoization approach, causing it to take less time to complete the computation.