Computability & Complexity_Report

RC11_22161736

BARTLETT 23/24

Exercise One

1. Implement this algorithm in Python.

Use the NumPy ndarray object for your matrices;.

```
import time
import numpy as np


def multiply_matrices(A, B):
    dimension = A.shape[0]
    Result = np.zeros((dimension, dimension), dtype=int)
    for i in range(dimension):
        for j in range(dimension):
            Result[i, j] = sum(A[i, k] * B[k, j] for k in range(dimension))
    return Result
```

Implementation and Comparison

```
Matrix_A = np.array([[1, 2], [3, 4]])
Matrix_B = np.array([[5, 6], [7, 8]])
Matrix_C = np.array([[6, 3, 10, 4], [10, 3, 6, 10], [6, 9, 2, 5], [4, 2, 9, 4]])
Matrix_D = np.array([[i for i in range(1, 6)] for _ in range(5)])

print("Results from matmul with NumPy:")
start = time.time()
print('2 x 2 Matrix:', np.matmul(Matrix_A, Matrix_B))
print('4 x 4 Matrix:', np.matmul(Matrix_C, Matrix_C))
print('5 x 5 Matrix:', np.matmul(Matrix_D, Matrix_D))
run_time = time.time() - start
print('time: ', run_time)

start = time.time()
print("Results obtained by a custom matrix multiplication function:")
print('2 x 2 Matrix:', multiply_matrices(Matrix_A, Matrix_B))
print('4 x 4 Matrix:', multiply_matrices(Matrix_C, Matrix_C))
print('5 x 5 Matrix:', multiply_matrices(Matrix_D, Matrix_D))
run_time = time.time() - start
print('time: ', run_time)

print('Chained matrix multiplication:')
print(np.matmul(np.matmul(Matrix_A, Matrix_B), Matrix_A))
print(multiply_matrices(multiply_matrices(Matrix_A, Matrix_B), Matrix_A))
```

Output:

```
Results from matmul with NumPy:
2 x 2 Matrix: [[19 22]
 [43 50]]
4 x 4 Matrix: [[142 125 134 120]
 [166 113 220 140]
 [158  73 163 144]
 [114 107 106  97]]
5 x 5 Matrix: [[15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]]
time:  0.0
Results obtained by a custom matrix multiplication function:
2 x 2 Matrix: [[19 22]
 [43 50]]
4 x 4 Matrix: [[142 125 134 120]
 [166 113 220 140]
 [158  73 163 144]
 [114 107 106  97]]
5 x 5 Matrix: [[15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]
 [15 30 45 60 75]]
time:  0.0009999275207519531
Chained matrix multiplication:
[[ 85 126]
 [193 286]]
[[ 85 126]
 [193 286]]
```

Comparison: custom matrix multiplication function is more efficient(fast).

Exercise Two

1. Describe and explain the algorithm. It should contain at least the following:

How is it recursive:It is recursive because it calls itself to perform the task of matrix multiplication on smaller sub-problems. Each Matrix A &B is divided into 4 sub_matrices and then these sub_matrices are used as inputs for further recursive calls.

The base case of the recursion occurs when the matrix dimension n = 1.The matrix multiplication is trivial at the point, consisting of multiplying two scalar value a11 and b11 to produce c11. This would stop the further recursive calls.

How does the recursion step reduce to the base case:Each recursive call processes a smaller sub_matrix of size n/2 * n/2, reducing the problem size with each division until the base case n = 1 is reached. The division continues recursively until the size of the matrices being multplied is just a single element.

2. Implement the recursive algorithm in Python.

```
import numpy as np
```

```
def partition_matrix(matrix):
    n = len(matrix)
    mid = n // 2
    top_left = [row[:mid] for row in matrix[:mid]]
    top_right = [row[mid:] for row in matrix[:mid]]
    bottom_left = [row[:mid] for row in matrix[mid:]]
    bottom_right = [row[mid:] for row in matrix[mid:]]
    return top_left, top_right, bottom_left, bottom_right

def add_matrices(A, B):
    return [[A[i][j] + B[i][j] for j in range(len(A[i]))] for i in range(len(A))]

def combine_matrices(top_left, top_right, bottom_left, bottom_right):
    top_half = [left + right for left, right in zip(top_left, top_right)]
    bottom_half = [left + right for left, right in zip(bottom_left, bottom_right)]
    return top_half + bottom_half

def recursive_multiply_matrices(A, B):
    n = len(A)
    if n == 1:
        return [[A[0][0] * B[0][0]]]
    else:
        A11, A12, A21, A22 = partition_matrix(A)
        B11, B12, B21, B22 = partition_matrix(B)
        M1 = add_matrices(recursive_multiply_matrices(A11, B11), recursive_multiply_matrices(A12, B21))
        M2 = add_matrices(recursive_multiply_matrices(A11, B12), recursive_multiply_matrices(A12, B22))
        M3 = add_matrices(recursive_multiply_matrices(A21, B11), recursive_multiply_matrices(A22, B21))
        M4 = add_matrices(recursive_multiply_matrices(A21, B12), recursive_multiply_matrices(A22, B22))
        return combine_matrices(M1, M2, M3, M4)
```

```
Matrix_E = np.array([[1, 2], [3, 4]])
Matrix_F = np.array([[5, 6], [7, 8]])
Matrix_G = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])

print("Results using recursive matrix multiplication:")
print('2 x 2 Matrix:', recursive_multiply_matrices(Matrix_E, Matrix_F))
print('4 x 4 Matrix:', recursive_multiply_matrices(Matrix_G, Matrix_G))

print('Chained recursive matrix multiplication (2 x 2 Matrix):')
chained_result = recursive_multiply_matrices(recursive_multiply_matrices(Matrix_E, Matrix_F), Matrix_E)
print(chained_result)
```

```
Results using recursive matrix multiplication:
2 x 2 Matrix: [[19, 22], [43, 50]]
4 x 4 Matrix: [[90, 100, 110, 120], [202, 228, 254, 280], [314, 356, 398, 440], [426, 484, 542, 600]]
Chained recursive matrix multiplication (2 x 2 Matrix):
[[85, 126], [193, 286]]
```

Test and compare the practical speed with the non-recursive algorithm

```
import numpy as np
import time

# Create a large 64x64 matrix
Large_Matrix = np.array([[i for i in range(1, 65)] for _ in range(64)])

# Measure the time taken by the basic matrix multiplication
start_time = time.time()
result_basic = multiply_matrices(Large_Matrix, Large_Matrix)
basic_mul_time = time.time() - start_time

# Measure the time taken by the recursive matrix multiplication
start_time = time.time()
result_recursive = recursive_multiply_matrices(Large_Matrix.tolist(), Large_Matrix.tolist())
recursive_mul_time = time.time() - start_time

print(f"Basic multiply time: {basic_mul_time:.4f} s, Recursive multiply time: {recursive_mul_time:.4f} s")
```

```
Basic multiply time: 0.0703 s, Recursive multiply time: 0.3292 s
```

3. Do a complexity analysis for the SMMRec algorithm.

Divide step: Partition each matrix of size n × n into four submatrices, each of size is (n/2)* (n/2).

Conquer step: Recursively multiply these smaller matrices until the base case is reached 1×1 matrices, multiplication is very straightforward at this point.

Combine: The results of these recursive multiplications are summed where necessary and then combined to form the resulting matrix C.

For the recursive matrix multiplication, each decomposition creates four subproblems, each half the size of the primitive problem. For each pair of submatrices, we perform 8 multiplications. The recursive property produces a time complexity recursive relation represented as: $T(n)=8T(2/n)+\Theta(n^2)$

Exercise Three

1. Reflect on the difference between (complexity of) addition/subtraction and multiplication on matrices.

Implement and test the algorithm

```
import numpy as np
import time

def enhanced_matrix_split(matrix):
    size = len(matrix)
    mid_point = size // 2
    quadrant1 = [row[:mid_point] for row in matrix[:mid_point]]
    quadrant2 = [row[mid_point:] for row in matrix[:mid_point]]
    quadrant3 = [row[:mid_point] for row in matrix[mid_point:]]
    quadrant4 = [row[mid_point:] for row in matrix[mid_point:]]
    return quadrant1, quadrant2, quadrant3, quadrant4

def matrix_operation_add(submatrix1, submatrix2):
    return [[submatrix1[i][j] + submatrix2[i][j] for j in range(len(submatrix1))] for i in range(len(submatrix1))]

def matrix_operation_subtract(submatrix1, submatrix2):
    return [[submatrix1[i][j] - submatrix2[i][j] for j in range(len(submatrix1))] for i in range(len(submatrix1))]

def assemble_matrix(q1, q2, q3, q4):
    upper_half = [left + right for left, right in zip(q1, q2)]
    lower_half = [left + right for left, right in zip(q3, q4)]
    return upper_half + lower_half

def optimized_strassen_matrix_multiply(A, B):
    n = len(A)
    if n <= 2:  # Use the direct method for small matrices
        return recursive_multiply_matrices(A, B)
    else:
        mid = n // 2
        A11, A12, A21, A22 = enhanced_matrix_split(A)
        B11, B12, B21, B22 = enhanced_matrix_split(B)

        S1 = matrix_operation_add(A11, A22)
        S2 = matrix_operation_add(B11, B22)
        S3 = matrix_operation_add(A21, A22)
        S4 = B11
        S5 = A11
        S6 = matrix_operation_subtract(B12, B22)
        S7 = A22
        S8 = matrix_operation_subtract(B21, B11)
        S9 = matrix_operation_add(A11, A12)
        S10 = B22
        P1 = optimized_strassen_matrix_multiply(S1, S2)
        P2 = optimized_strassen_matrix_multiply(S3, S4)
        P3 = optimized_strassen_matrix_multiply(S5, S6)
        P4 = optimized_strassen_matrix_multiply(S7, S8)
        P5 = optimized_strassen_matrix_multiply(S9, S10)
        Q1 = matrix_operation_add(matrix_operation_subtract(matrix_operation_add(P1, P4), P5), P2)
        Q2 = matrix_operation_add(P3, P5)
        Q3 = matrix_operation_add(P2, P4)
        Q4 = matrix_operation_subtract(matrix_operation_subtract(matrix_operation_add(P1, P3), P2), P5)

        return assemble_matrix(Q1, Q2, Q3, Q4)
```

```
Matrix_H = np.array([[1, 2], [3, 4]])
Matrix_I = np.array([[5, 6], [7, 8]])
Matrix_J = np.array([[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12], [13, 14, 15, 16]])

print("Results using Strassen's algorithm:")
print('2 x 2 Matrix:', optimized_strassen_matrix_multiply(Matrix_H, Matrix_I))
print('4 x 4 Matrix:', optimized_strassen_matrix_multiply(Matrix_J, Matrix_J))
```

```
Results using Strassen's algorithm:
2 x 2 Matrix: [[19, 22], [43, 50]]
4 x 4 Matrix: [[604, 688, 110, 120], [764, 880, 254, 280], [314, 356, 136, 136], [426, 484, 72, 72]]
```

Compare the practical speed with the recursive algorithm

```
# Create a 128x128 matrix
Performance_Matrix = np.array([[i for i in range(1, 129)] for _ in range(128)])

# Start the timer for basic matrix multiplication
start_time = time.time()
result_from_basic = multiply_matrices(Performance_Matrix, Performance_Matrix)
time_for_basic = time.time() - start_time  # Calculate elapsed time

# Start the timer for recursive matrix multiplication
start_time = time.time()
result_from_recursive = recursive_multiply_matrices(Performance_Matrix.tolist(), Performance_Matrix.tolist())
time_for_recursive = time.time() - start_time  # Calculate elapsed time

print(f"Basic multiplication: {time_for_basic:.4f} s, Recursive multiplication: {time_for_recursive:.4f} s")
```

```
Basic multiplication: 0.5559 s, Recursive multiplication: 2.6781 s
```

2.Do a complexity analysis of the Strassen algorithm.

Standard Recursive Algorithm: The recurrence relation is T(n) = 8T(n/2)+Θ(n^2). This recurrence solves to O(n^3), indicating a cubic time comlexity typical for standard matrix multiplication according to the Master Theorem.

Strassen's Algorithm: In Strassen's method, the number of recursive calls is reduced to 7, instead of 8 as in the standard method. The recurrence relation for Strassen's algorithm is T(n) = 7T(n/2)+(n^2). Using the Master Theorem again, the solution to this recurrence is O(n^log7^2), which is aprroximately O(n^2.81). This demonstrates a more effcient algorithm than the standard approach, due to a lower exponent in the time complexity.

The Jupyter notebook file uploaded on Github