Cairo University
Faculty of Engineering
Credit Hour System

SBEN429 Biomedical Data Analytics
Assignment 2

Name: **Sandra Adel Aziz Gebraiel**

ID: **1180059**

Date: **23/11/2021**

Naïve Divide and Conquer Approach of Matrix Multiplication:

1) Explanation of Code:

The division approach takes place by dividing each nxn matrices into 4 sub-matrices of size n/2 recursively, until nxn = 1x1.
The conquer approach is then satisfied by obtaining eight sub-problems of size n/2 (4 belonging to A_, and 4 belonging to B_, recursively)

So A_ = [[A, B], and B_ = [[E, F], So A_B_ = [[AE + BG, AF + BH],
[C, D]] [G, H]] [CE + DG, CF + DH]]

In which AE, BG, AF, BH, CE, DG, CF & DH are the 8 sub-problems/8 sub-matrices whose multiplications have to be computed.
Each sub-problem, when reaching base case size of 1x1, is computed by multiplying A_[ai, aj] x B_[bi, bj],
Where ai --> A's row pointer, aj → A's column pointer, bi → B's row pointer, bj → B's column pointer, according to these pointers' values in the reached recursive step.
Then each is summed and placed in its right place in the product matrix

RECURRANCE EQUATION: 8xT(n/2) + Kxn^2
TIME COMPLEXITY: Θ(n^3)

The previous algorithm only works if n is of power of 2 (to be able to divide each n x n matrix into 4 n/2 x n/2 sub-matrices), but n cannot be limited to these values.
Another issue is that the original matrix mult function does not have ai, aj, bi, bj among its parameters.
So, this is handled by placing the core algorithm in a wrapped matrix_mult function while the original call of matrix_mult handles these issues

In matrix_mult, n is checked if it is a power of 2: if log2(n) is a whole number in range 0-6, since the largest input n = 100, so the last n as power of 2 in range 0-100 is 64 = n^6)
If condition is true call the wrapped matrix_mult function with initial values of all pointers as zeros (begining of both matrices)
If it is false, then the nearest power of 2 larger than n is found, and both A and B matrices are padded with zero columns and rows, equal to the value of difference between this nearest power of 2 and original n, and the wrapped matrix_mult function is called with new zero padded A and B, and nearest power of 2 as new size
Then the result is sliced, to take only the original non-zero columns and rows of indices 0 → n-1

2) Pseudocode:

Wrapped_Matrix_Mult(A, B, n, ai, aj, bi, bj):

```
Initialize a 2D array 'product' of size nxn filled with zeros
If n =1:
        product[0,0] = A[ai,aj] x B[bi, bj]
        return product

AE = Wrapped_Matrix_Mult(A, B, n/2, ai, aj, bi, bj )
BG = Wrapped_Matrix_Mult(A, B, n/2, ai, aj + n/2, bi, bj + n/2 )
AF = Wrapped_Matrix_Mult(A, B, n/2, ai, aj, bi, bj + n/2)
BH = Wrapped_Matrix_Mult(A, B, n/2, ai, aj + n/2, bi + n/2, bj + n/2)
CE = Wrapped_Matrix_Mult(A, B, n/2, ai + n/2, aj, bi, bj)
DG = Wrapped_Matrix_Mult(A, B, n/2, ai + n/2, aj + n/2, bi + n/2, bj)
CF = Wrapped_Matrix_Mult(A, B, n/2, ai + n/2, aj, bi, bj + n/2)
DH = Wrapped_Matrix_Mult(A, B, n/2, ai + n/2, aj + n/2, bi + n/2, bj + n/2)

Place AE + BG in upper left part of product matrix
Place AF + BH in upper right part of product matrix
Place CE + DG in lower left part of product matrix
Place CF + DH in lower right part of product matrix
```

Matrix_Mult(A, B, n):

```
If n is not a power of 2:
        new_n = nearest power of 2 larger than n
        extra = new_n - n
        Initialize new_A of size new_n x new_n filled with zeros
        Place the values of A in upper left part of new_A
        Initialize new_B of size new_n x new_n filled with zeros
        Place the values of B in upper left part of new_B
        result = Wrapped_Matrix_Mult( new_A, new_B, new_n, 0, 0, 0, 0 )
        product = result[0 --> n-1 : 0 --> n-1]
else:
        product = Wrapped_Matrix_Mult( A, B, n, 0, 0, 0, 0 )

return product
```

3) Questions:

A)

```python
# Naive Solution:
# ******************
# Time Complexity of O(n^3):
# Because there are n^2 elements in the resultant matrix to be computed,
# each takes O(n) in its computation.

def matrix_mult_naive(A, B, n):

    product = np.zeros((n,n))

    for i in range(n):
        for j in range(n):
            for k in range(n):
                product[i,j] += A[i,k] * B[k,j]

    return product
```

B)

Recurrence Equation (Running Time): T(n) = 8xT(n/2) + Kxn^2

Where 8 → The problem is divided into 8 sub-problems:
(AE, BG, AF, BH, CE, DG, CF, DH)

T(n/2) → Each of size n/2

Kxn^2 → Rough run time of placement of sub-matrices in their right places in the product matrix

```python
def wrapped_matrix_mult(A, B, n, ai, aj, bi, bj):  → T(n)

    product = np.zeros((n,n))

    if n == 1:
        product[0,0] = A[ai, aj] * B[bi, bj]
        return product

    AE = wrapped_matrix_mult(A, B, n//2, ai, aj, bi, bj)  → T(n/2)
    BG = wrapped_matrix_mult(A, B, n//2, ai, aj + n//2, bi + n//2, bj)  → T(n/2)

    AF = wrapped_matrix_mult(A, B, n//2, ai, aj, bi, bj + n//2)  → T(n/2)
    BH = wrapped_matrix_mult(A, B, n//2, ai, aj + n//2, bi + n//2, bj + n//2)  → T(n/2)

    CE = wrapped_matrix_mult(A, B, n//2, ai + n//2, aj, bi, bj)  → T(n/2)
    DG = wrapped_matrix_mult(A, B, n//2, ai + n//2, aj + n//2, bi + n//2, bj)  → T(n/2)

    CF = wrapped_matrix_mult(A, B, n//2, ai + n//2, aj, bi, bj + n//2)  → T(n/2)
    DH = wrapped_matrix_mult(A, B, n//2, ai + n//2, aj + n//2, bi + n//2, bj + n//2)  → T(n/2)
```

```python
    pi = 0
    for i in range(n//2):
        pj = 0
        for j in range(n//2):
            product[pi, pj] += AE[i, j] + BG[i, j]
            pj += 1
        pi += 1
```

Kn^2

```python
    pi = 0
    for i in range(n//2):
        pj = n//2
        for j in range(n//2):
            product[pi, pj] += AF[i, j] + BH[i, j]
            pj += 1
        pi += 1
```

Kn^2

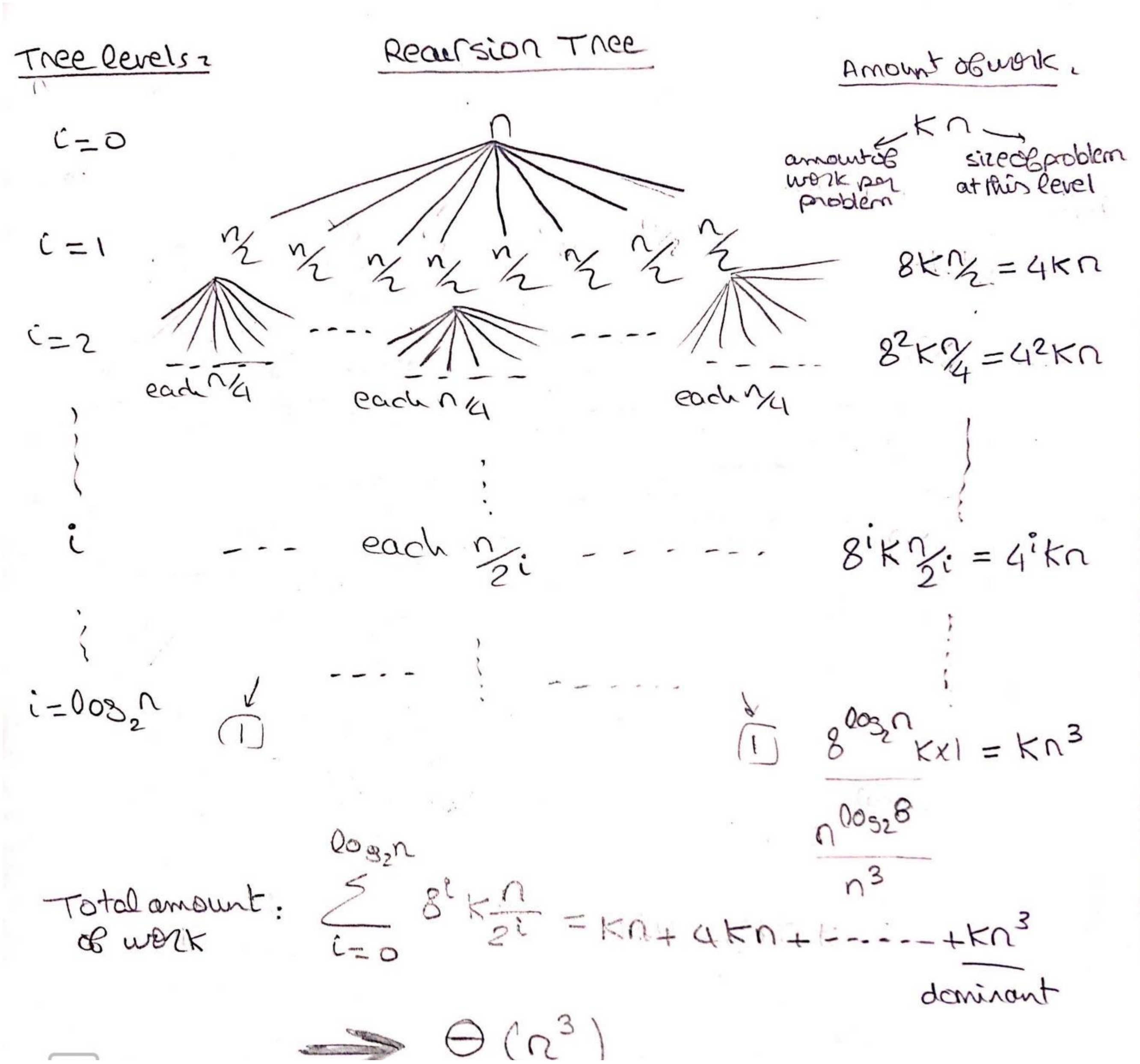
```python
pi = n//2
for i in range(n//2):
    pj = 0
    for j in range(n//2):
        product[pi, pj] += CE[i, j] + DG[i, j]
        pj += 1
    pi += 1
```

Kn^2

```python
pi = n//2
for i in range(n//2):
    pj = n//2
    for j in range(n//2):
        product[pi, pj] += CF[i, j] + DH[i, j]
        pj += 1
    pi += 1

return product
```

Kn^2



4) Stress Testing Code:

```
while 1:
    n = r.randint(1, 100)
    print(n)
    A = np.random.randint(-2 ** (10), 2 ** (10), (n,n))
    print(A)
    B = np.random.randint(-2 ** (10), 2 ** (10), (n,n))
    print(B)

    res1 = matrix_mult_naive(A, B, n)
    res2 = matrix_mult_divide_and_conquer_naive(A, B, n)
    print('OK')

    if (res1 != res2).all():
        print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
        break
    else:
        print('OK')
```

Optimized Divide and Conquer Approach of Matrix Multiplication (Bonus):

1) Explanation of Code:

Here, optimization is represented in dividing the problem into 7 sub-problems/sub-matrices of size n/2 x n/2 instead of 8, whose multiplication have to be computed, where A_ and B_ are sliced/partitioned into size n/2 as follows:

A_ = [[A, B],   and   B_ = [[E, F],
[C, D]]   [G, H]]

And the matrix multiplication of an algebraic expression of these partitions make up the 7 sub-problems: P1, P2, P3, P4, P5, P6, P7

P1 = A x (F-H), P2 = (A+B) x H, P3 = (C+D) x E, P4 = D x (G-E),
P5 = (A+D) x (E+H), P6 = (B-D) x (G+H), P7 = (A-C) x (E+F) → Strassen

Where each of P1-P7 is computed when the base case of both matrices to be multiplied (to be of size 1x1) is recursively reached.

Then, each sub-matrix is placed in its right place in the product matrix as follows:

A_B_ = [[P5+P4-P2+P6, P1+P2],
[P3+P4, P1+P5-P3-P7]]

Note: pointers are not used here, because in each recursive call, as each sub-problem is furtherly reduced to size n/2, both input matrices are sliced before the recursive function call, and are sent sliced in the function, until their size reaches 1x1, in which step, their multiplication is calculated.

WHEREAS, in the naive divide and conquer, the sub-problems are also recursively reduced to size n/2, but the whole matrices are sent in the function call without slicing, and the pointers indicate which element in both matrices (case of reaching size of 1x1) should be multiplied.

RECURRANCE EQUATION: 7xT(n/2) + Kxn^2
TIME COMPLEXITY: Θ(n^(log2(7))) ≈ Θ(n^2.81)

2) Pseudocode:

<u>Wrapped_Matrix_Mult_Fast(A, B, n):</u>

```
if n = 1:
        return A[0,0] x B[0,0] in a 2D array of size 1x1

A_ = A[0 → n/2 -1 , 0 → n/2 -1]
B_ = A[0 → n/2 - 1, n/2 → n -1]
C = A[n/2 → n - 1, 0 → n/2 - 1]
D = A[n/2 → n - 1, n/2 → n - 1]
E = B[0 → n/2 - 1, 0 → n/2 - 1]
F = B[0 → n/2 - 1, n/2 → n - 1]
G = B[n/2 → n - 1, 0 → n/2 -1]
H = B[n/2 → n - 1, n/2 →n -1]

P1 = Wrapped_Matrix_Mult_Fast(A_, F - H, n/2)
P2 = Wrapped_Matrix_Mult_Fast(A_ + B_, H, n/2)
P3 = Wrapped_Matrix_Mult_Fast(C + D, E, n/2)
P4 = Wrapped_Matrix_Mult_Fast(D, G - E, n/2)
P5 = Wrapped_Matrix_Mult_Fast(A_ + D, E + H, n/2)
P6 = Wrapped_Matrix_Mult_Fast(B_ - D, G + H, n/2)
P7 = Wrapped_Matrix_Mult_Fast(A_ - C, E + F, n/2)

Initialize a 2D array 'product' of size nxn filled with zeros

Place P5 + P4 - P2 + P6 in upper left part of product matrix
Place P1 + P2 in upper right part of product matrix
Place P3 + P4 in lower left part of product matrix
Place P1 + P5 - P3 - P7 in lower right part of product matrix

return product
```

<u>Matrix_Mult_Fast(A, B, n):</u>

```
If n is not a power of 2:
        new_n = nearest power of 2 larger than n
        extra = new_n - n
        Initialize new_A of size new_n x new_n filled with zeros
        Place the values of A in upper left part of new_A
        Initialize new_B of size new_n x new_n filled with zeros
        Place the values of B in upper left part of new_B
        result = Wrapped_Matrix_Mult_Fast( new_A, new_B, new_n )
        product = result[0 --> n-1 : 0 --> n-1]
```

```
else:
        product = Wrapped_Matrix_Mult_Fast( A, B, n )
return product
```

3) Questions:

A)

Recurrence Equation (Running Time): T(n) = 7xT(n/2) + Kxn^2

Where 7 → The problem is divided into 7 sub-problems (P1, P2, P3, P4, P5, P6, P7)

T(n/2) → Each of size n/2

Kxn^2 → Rough run time of placement of sub-matrices in their right places in the product matrix

```
def wrapped_matrix_mult_fast(A, B, n):    → T(n)

    if n == 1:
        return np.array([[A[0,0] * B[0,0]]])

    A_ = A[0:n//2, 0:n//2]
    B_ = A[0:n//2, n//2:n]
    C = A[n//2:n, 0:n//2]
    D = A[n//2:n, n//2:n]                  T(1)
    E = B[0:n//2, 0:n//2]
    F = B[0:n//2, n//2:n]
    G = B[n//2:n, 0:n//2]
    H = B[n//2:n, n//2:n]

    P1 = wrapped_matrix_mult_fast(A_, F - H, n//2)        → T(n/2)
    P2 = wrapped_matrix_mult_fast(A_ + B_, H, n//2)       → T(n/2)
    P3 = wrapped_matrix_mult_fast(C + D, E, n//2)         → T(n/2)
    P4 = wrapped_matrix_mult_fast(D, G - E, n//2)         → T(n/2)
    P5 = wrapped_matrix_mult_fast(A_ + D, E + H, n//2)    → T(n/2)
    P6 = wrapped_matrix_mult_fast(B_ - D, G + H, n//2)    → T(n/2)
    P7 = wrapped_matrix_mult_fast(A_ - C, E + F, n//2)    → T(n/2)

    product = np.zeros((n, n))

    pi = 0
    for i in range(n//2):
        pj = 0
        for j in range(n//2):                                                 Kn^2
            product[pi, pj] += ( P5[i, j] + P4[i, j] - P2[i, j] + P6[i, j] )
            pj += 1
        pi += 1

    pi = 0
    for i in range(n//2):
        pj = n//2
        for j in range(n//2):                                                 Kn^2
            product[pi, pj] += ( P1[i, j] + P2[i, j] )
            pj += 1
        pi += 1
```

```python
pi = n//2
for i in range(n//2):
    pj = 0
    for j in range(n//2):
        product[pi, pj] += ( P3[i, j] + P4[i, j] )
        pj += 1
    pi += 1
```

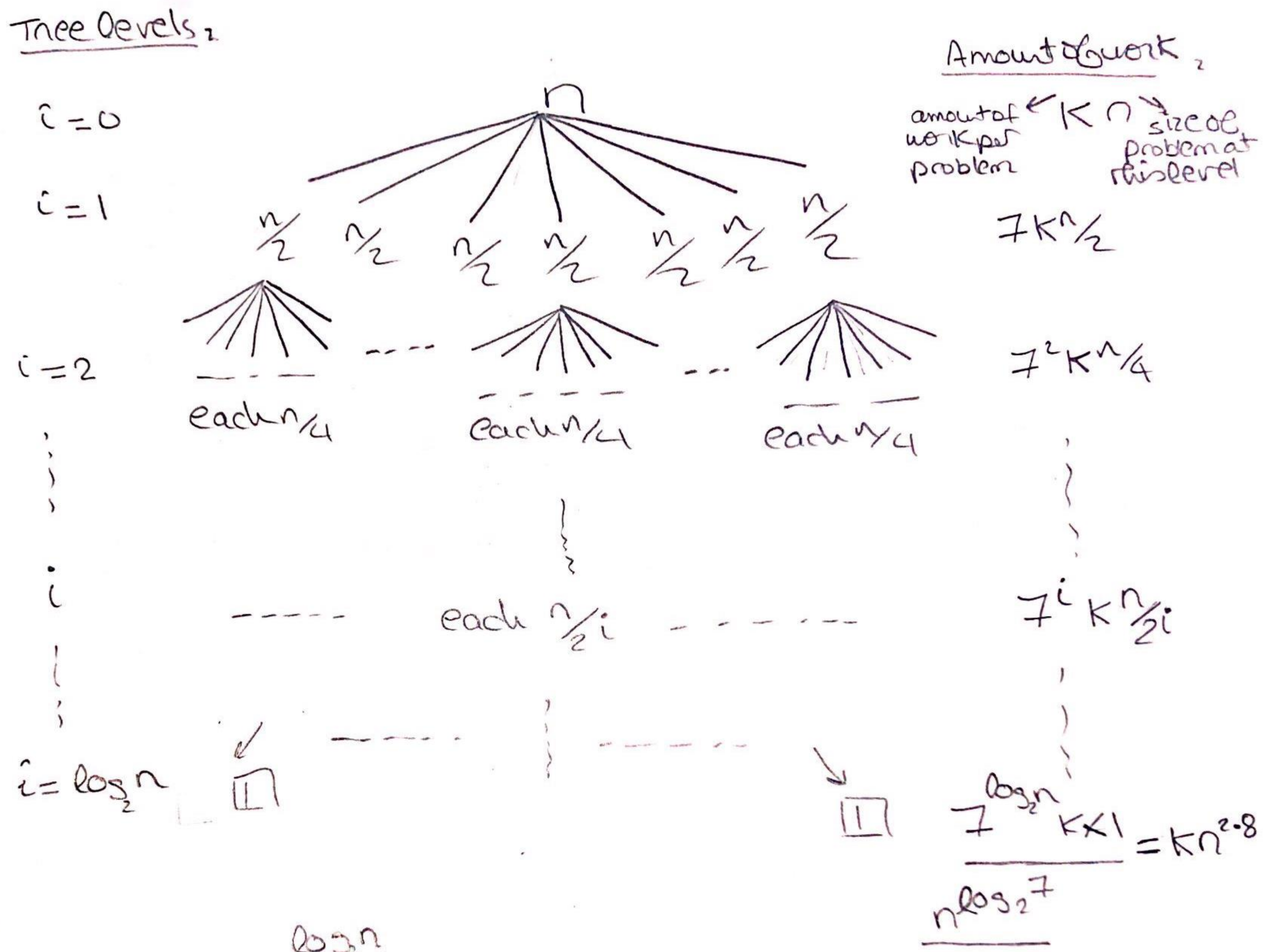
Kn^2

```python
pi = n//2
for i in range(n//2):
    pj = n//2
    for j in range(n//2):
        product[pi, pj] += ( P1[i, j] + P5[i, j] - P3[i, j] - P7[i, j] )
        pj += 1
    pi += 1

return product
```

Kn^2

Recursion Tree



Total amount of work: $\sum_{i=0}^{\log_2 n} 7^i K \frac{n}{2^i} = Kn + 7Kn/2 + \dots + \underbrace{Kn^{2.8}}_{\text{dominant}}$

$$\Rightarrow \Theta(n^{\log_2 7})$$

$$\simeq \Theta(n^{2.8})$$

4) Stress Testing Code:

```
while 1:
        n = r.randint(1, 100)
        print(n)
        A = np.random.randint(-2 ** (10), 2 ** (10), (n,n))
        print(A)
        B = np.random.randint(-2 ** (10), 2 ** (10), (n,n))
        print(B)

        res1 = matrix_mult_naive(A, B, n)
        res2 = matrix_mult_divide_and_conquer_fast(A, B, n)
        print('OK')

        if (res1 != res2).all():
            print('wrong answer'+ ' ' + str(res1) + ' ' + str(res2))
            break
        else:
            print('OK')
```

Resources:

MOOC: Algorithmic Toolbox Slides - Week 4

Sanjoy Dasgupta, Christos Papadimitriou, and Umesh Vazirani. Algorithms (1st Edition). McGraw-Hill Higher Education. 2008.