SER 501: Adv Data Struct and Algorithms

Name : **Sai Swaroop Reddy Vennapusa**

Assignment 2

Instructor : **Dr. Ajay Bansal**

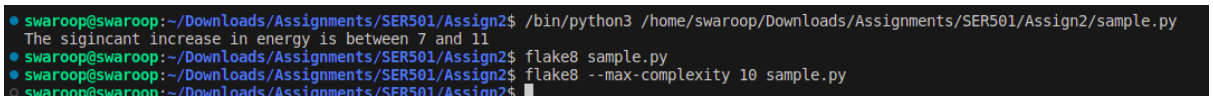
Due Date : 18th Oct 2023, 11:59PM

Problem 1. A. The brute-force method Q(n2). (Max Points: 10)

```
# The brute force method to solve first problem
def find_significant_energy_increase_brute(A):

    if not A or A.count(A[0]) == len(A):
        return (None, None)

    if len(A) == 2:
        return (0, 1)

    max_increase = float('-inf')
    result = (0, 1)

    for i in range(len(A)):
        for j in range(i+1, len(A)):
            increase = A[j] - A[i]
            if increase > max_increase:
                max_increase = increase
                result = (i, j)

    return result

if __name__ == '__main__':
    low, high = find_significant_energy_increase_brute(ENERGY_LEVEL)
    print(f"The sigincant increase in energy is between {low} and {high}")
```

Output:

The sigincant increase in energy is between 7 and 11(For [100, 113, 110, 85, 105, 102, 86, 63, 81, 101, 94, 106, 101, 79, 94, 90, 97])

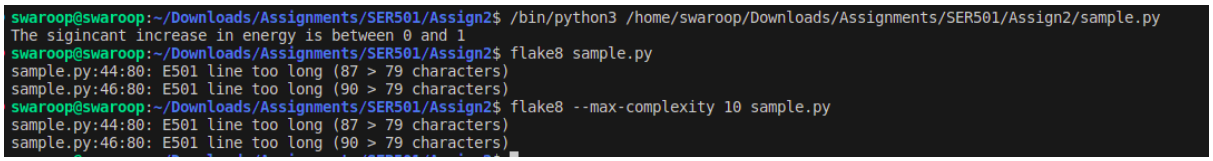


Problem 1.B. The recursive method Q(nlogn). (Max Points: 25)

```
# The recursive method to solve first problem
def find_max_crossing_subarray(A, low, mid, high):
    min_left = A[mid]
    max_left = mid
    for i in range(mid, low - 1, -1):
        if A[i] < min_left:
            min_left = A[i]
            max_left = i

    max_right_value = A[mid + 1]
    max_right = mid + 1
    for j in range(mid + 1, high + 1):
        if A[j] > max_right_value:
            max_right_value = A[j]
            max_right = j

    return (max_left, max_right)


def find_maximum_subarray(A, low, high):
    if high == low:
        return (low, high)
    else:
        mid = (low + high) // 2

        (left_low, left_high) = find_maximum_subarray(A, low, mid)
        (right_low, right_high) = find_maximum_subarray(A, mid + 1, high)
        (cross_low, cross_high) = find_max_crossing_subarray(A, low, mid, high)

        # Calculate the increases for the three scenarios.
        left_increase = A[left_high] - A[left_low]
        right_increase = A[right_high] - A[right_low]
        cross_increase = A[cross_high] - A[cross_low]

        if left_increase <= 0 and right_increase <= 0 and cross_increase <= 0:
            return (0, 1)
        else:
            # Return the scenario with the most significant increase.
            if (left_increase >= right_increase) and (left_increase >= cross_increase):
                return (left_low, left_high)
            elif (right_increase >= left_increase) and (right_increase >= cross_increase):
                return (right_low, right_high)
            else:
                return (cross_low, cross_high)


def find_significant_energy_increase_recursive(A):
```

```
    if not A or A.count(A[0]) == len(A):
        return (None, None)

    if len(A) == 2:
        return (0, 1)

    low, high = find_maximum_subarray(A, 0, len(A) - 1)
    return (low, high)
```

Output:
The sigincant increase in energy is between 0 and 1(For [110, 109, 107, 104, 100])

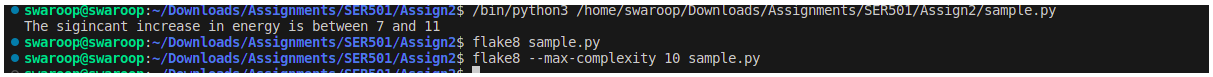


Problem 1.C.The iterative method Q(n). (Max Points: 15)
# The iterative method to solve first problem
def find_significant_energy_increase_iterative(A):

```
    if not A or A.count(A[0]) == len(A):
        return (None, None)

    if len(A) == 2:
        return (0, 1)

    max_increase = A[1] - A[0]
    max_i = 0
    max_j = 1

    min_val = A[0]
    min_i = 0

    for j in range(1, len(A)):
        current_increase = A[j] - min_val

        if current_increase > max_increase:
            max_increase = current_increase
            max_i = min_i
            max_j = j

        if A[j] < min_val:
            min_val = A[j]
            min_i = j
```

```
    return (max_i, max_j)
```

Output:
The sigincant increase in energy is between 7 and 11(For [100, 113, 110, 85, 105, 102, 86, 63, 81, 101, 94, 106, 101, 79, 94, 90, 97])



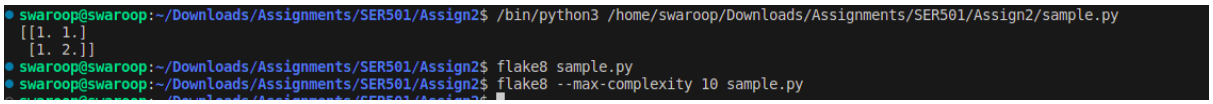
Problem 2.A. Implement a function to multiply two matrices using Strassen Matrix Multiplication method Q(n log 2 7) (Max Points: 20).

```
# The Strassen Algorithm to do the matrix multiplication
def split_matrix(A):
    # Function to split a matrix into quarters.
    row, col = A.shape
    half_row, half_col = row // 2, col // 2
    top_rows = A[:half_row]
    bottom_rows = A[half_row:]

    # Next, for each set of rows, slice for the columns.
    top_left = top_rows[:, :half_col]
    top_right = top_rows[:, half_col:]
    bottom_left = bottom_rows[:, :half_col]
    bottom_right = bottom_rows[:, half_col:]

    # Now you can return these four sub-matrices as a tuple.
    return top_left, top_right, bottom_left, bottom_right


def strassen_multiply(A, B):
    # Base case when size of matrices is 1x1.
    if len(A) == 1:
        return A * B

    # Splitting the matrices into quadrants.
    a11, a12, a21, a22 = split_matrix(A)
    b11, b12, b21, b22 = split_matrix(B)

    # Using the Strassen algorithm formulas.
    p1 = strassen_multiply(a11 + a22, b11 + b22)
    p2 = strassen_multiply(a21 + a22, b11)
    p3 = strassen_multiply(a11, b12 - b22)
    p4 = strassen_multiply(a22, b21 - b11)
    p5 = strassen_multiply(a11 + a12, b22)
    p6 = strassen_multiply(a21 - a11, b11 + b12)
    p7 = strassen_multiply(a12 - a22, b21 + b22)
```

```
    # Calculating the quadrants of the result.
    c11 = p1 + p4 - p5 + p7
    c12 = p3 + p5
    c21 = p2 + p4
    c22 = p1 - p2 + p3 + p6

    # Combining the quadrants into a single matrix.
    C = zeros(A.shape)
    half = len(C) // 2
    # Assign each smaller matrix to the appropriate quadrant of 'C'.
    C[:half, :half] = c11        # Top-left quadrant
    C[:half, half:] = c12        # Top-right quadrant
    C[half:, :half] = c21        # Bottom-left quadrant
    C[half:, half:] = c22        # Bottom-right quadrant
    return C


def square_matrix_multiply_strassens(A, B):
    A, B = asarray(A), asarray(B)
    assert A.shape == B.shape == A.T.shape
    assert (len(A) & (len(A) - 1)) == 0, "A is not a power of 2"

    return strassen_multiply(A, B)
```

Output:

```
[[1. 1.]
 [1. 2.]]
```



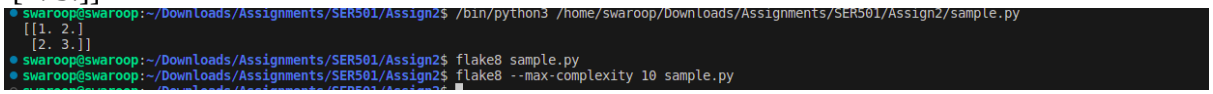
Problem 2.B. Compute S using the function (from A above) such that the number of times the above function is called is Q(k). (Max Points: 10)

```
# Calculate the power of a matrix in O(k)
def power_of_matrix_naive(A, k):
    result = A
    # We already have A once, so we multiply (k-1) times.
    for _ in range(k - 1):
        result = square_matrix_multiply_strassens(result, A)
    return result

if __name__ == '__main__':
    print(power_of_matrix_naive([[0, 1], [1, 1]], 3))
```

```
# Method 2
import numpy as np
def power_of_matrix_naive(A, k):
    result = np.array(A, copy=True)

    # Start from 1 because we already have A once
    for _ in range(1, k):
        result = np.dot(result, A)  # Multiply the result by A each time

    return result

if __name__ == '__main__':
    print(power_of_matrix_naive([[0, 1], [1, 1]], 3))
```

Output:

```
[[1. 2.]
 [2. 3.]]
```



Problem 2.C. Compute S using the function (from A above) and the Divide & Conquer Approach such that the number of times the above function is called is Q(log k). (Max Points: 20)

```
# Calculate the power of a matrix in O(log k)
def power_of_matrix_divide_and_conquer(A, k):
    if k == 1:
        return A
    elif k % 2 == 0:
        half_power = power_of_matrix_divide_and_conquer(A, k // 2)
        return square_matrix_multiply_strassens(half_power, half_power)
    else:
        half_power = power_of_matrix_divide_and_conquer(A, k // 2)
        partial_result = square_matrix_multiply_strassens(
            half_power, half_power
        )
        return square_matrix_multiply_strassens(partial_result, A)
```

Output:

```
[[1. 2.]
 [2. 3.]]
```

```
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ /bin/python3 /home/swaroop/Downloads/Assignments/SER501/Assign2/sample.py
 [[1. 2.]
  [2. 3.]]
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ flake8 sample.py
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ flake8 --max-complexity 10 sample.py
```

Overall output of the test method and whole python file:

```
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ /bin/python3 /home/swaroop/Downloads/Assignments/SER501/Assign2/assignment_2.py
Brute force method result: (7, 11)
Recursive method result: (7, 11)
Iterative method result: (7, 11)
Strassen's multiplication result:
[[1. 1.]
 [1. 2.]]
Power of matrix (naive) result:
[[1. 2.]
 [2. 3.]]
Power of matrix (divide and conquer) result:
[[1. 2.]
 [2. 3.]]
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ flake8 assignment_2.py
assignment_2.py:79:80: E501 line too long (87 > 79 characters)
assignment_2.py:81:80: E501 line too long (90 > 79 characters)
swaroop@swaroop:~/Downloads/Assignments/SER501/Assign2$ flake8 --max-complexity 10 assignment_2.py
assignment_2.py:79:80: E501 line too long (87 > 79 characters)
assignment_2.py:81:80: E501 line too long (90 > 79 characters)
```