

Assignment-2

(Total Points: 50)

Instructions:

- Download the attached python file assignment_2.py.
- Follow the instructions and replace all TODO comments in the scaffolding code.
- Test your code as much as you can to make certain it is correct.
- Run flake8 in addition to testing your code; I expect professional and clear code with minimal flake8 warnings and having McCabe complexity (<10) from all of you.
- Create a write up with formatted code and screenshots of your code, output, running the McCabe complexity command and error free console.
- Save the write-up as a PDF and submit it along with your python code as
- separate attachments before the due date on blackboard.

Note: Running flake 8

flake8 path/to/your/file (for warnings and errors)

flake8 --max-complexity 10 path/to/your/file (for complexity)

Problem 1: Given an array having both positive and negative integers, (using the scaffolding code provided in the 'assignment_2.py' file) implement functions to solve the maximum sub-array problem:

A. Using the brute-force method. (Max Points: 5)

B. Using the recursive method (you would need to implement a function to solve the maximum crossing sub-array problem). (Max Points: 15)

C. Using the iterative method. (Max Points: 10)

Problem 2: Given two square matrices A and B, (using the scaffolding code provided in the 'assignment_2.py' file) implement functions to calculate the product AB:

A. Using matrix multiplication. (Max Points: 10)

B. Using Strassen's Algorithm. (Max Points: 10)

Source code:

```
# -*- coding: utf-8 -*-

from numpy import * # NOQA
from numpy import zeros
from numpy import array_equal
from numpy import random
from numpy import asarray
from numpy import inf
from numpy import floor

STOCK_PRICE_CHANGES = [13, -3, -25, 20, -3, -16,
                        -23, 18, 20, -7, 12, -5, -22, 15, -4, 7]

BLUE = '\033[94m'
BOLD = '\033[1m'
UNDERLINE = '\033[4m'
END = '\033[0m'

# The brute force to solve max subarray problem
def find_maximum_subarray_brute(A, low=0, high=-1):
    """
    Return a tuple (i,j) where A[i:j] is the maximum subarray.
    time complexity = O(n^2)
    """

    maxTotal = float(-inf)
    inputArr = asarray(A)
    n = inputArr.size

    if (low == high):
        return (low, high)
    if (high == -1 and n == 1):
        return (0, 0)
    else:
        for i in range(0, n):
            currentTotal = 0
            for j in range(i, n):
                currentTotal = currentTotal + inputArr[j]
                if (currentTotal > maxTotal):
                    maxTotal = currentTotal
                    low = i
                    high = j
            return (low, high)

# The maximum crossing subarray for solving the max subarray problem
def find_maximum_crossing_subarray(A, low, mid, high):
    """
    Find the maximum subarray that crosses mid
    Return a tuple (i,j) where A[i:j] is the maximum subarray.
    """

    inputArr = asarray(A)
    leftTotal = float(-inf)
    maxLeft = 0
    rightTotal = float(-inf)
    maxRight = 0
    currentTotal = 0

    for i in range(mid, low - 1, -1):
        currentTotal = currentTotal + inputArr[i]
```

```

        if (currentTotal > leftTotal):
            leftTotal = currentTotal
            maxLeft = i
    currentTotal = 0
    for j in range(mid + 1, high + 1):
        currentTotal = currentTotal + inputArr[j]
        if (currentTotal > rightTotal):
            rightTotal = currentTotal
            maxRight = j
    return (maxLeft, maxRight)

# The recursive to solve max subarray problem
def find_maximum_subarray_recursive(A, low=0, high=-1):
    """
    Return a tuple (i,j) where A[i:j] is the maximum subarray.
    """

    inputArr = asarray(A)
    if (high == low):
        return (low, high)
    if (high == -1 and inputArr.size == 1):
        return (0, 0)
    else:
        # Here we are assuming that if no "high" value was specified,
        # then we are looking for max sub-array of complete array
        if (high == -1):
            high = inputArr.size - 1
        mid = int(floor((high + low) / 2))

        lefts = find_maximum_subarray_recursive(inputArr, low, mid)
        leftTotal = getTotal(inputArr, lefts[0], lefts[1])
        rights = find_maximum_subarray_recursive(inputArr, mid + 1, high)
        rightTotal = getTotal(inputArr, rights[0], rights[1])
        crosses = find_maximum_crossing_subarray(inputArr, low, mid, high)
        crossTotal = getTotal(inputArr, crosses[0], crosses[1])

        if (leftTotal >= rightTotal and leftTotal >= crossTotal):
            return lefts
        if (rightTotal >= leftTotal and rightTotal >= crossTotal):
            return rights
        else:
            return crosses

def getTotal(A, low, high):
    """
    Return a value representing the sum of terms from A[low:high].
    """
    subArrTotal = 0
    for i in range(low, high + 1):
        subArrTotal += A[i]
    return subArrTotal

# The iterative to solve max subarray problem
def find_maximum_subarray_iterative(A, low=0, high=-1):
    """
    Return a tuple (i,j) where A[i:j] is the maximum subarray.
    """

    inputArr = asarray(A)
    if (high == low):

```

```

        return (low, high)
    if (high == -1 and inputArr.size == 1):
        return (0, 0)

    left = 0
    right = 0
    maxTotal = float(-inf)
    currentLeft = 0
    currentTotal = 0

    for i in range(0, inputArr.size):
        currentTotal = currentTotal + inputArr[i]
        if (currentTotal > maxTotal):
            maxTotal = currentTotal
            left = currentLeft
            right = i
        if (currentTotal <= 0):
            currentTotal = 0
            currentLeft = i + 1
    return (left, right)

# =====

def square_matrix_multiply(A, B):
    """
    Return the product AB of matrix multiplication.
    """

    matrix1 = asarray(A)
    matrix2 = asarray(B)
    assert matrix1.shape == matrix2.shape
    assert matrix1.shape == matrix1.T.shape
    assert matrix1.size, "matrix1 is empty"

    n = int(matrix1.size ** (0.5))
    resultMatrix = zeros((n, n))

    if (n == 1):
        resultMatrix[0] = matrix1[0] * matrix2[0]
    else:
        for i in range(0, n):
            for j in range(0, n):
                resultMatrix[i, j] = 0
                for k in range(0, n):
                    resultMatrix[i, j] = resultMatrix[i, j] + \
                        matrix1[i, k] * matrix2[k, j]

    return resultMatrix

# =====

def square_matrix_multiply_strassen(A, B):
    """
    Return the product AB of matrix multiplication.
    Assume len(A) is a power of 2
    """

    matrix1 = asarray(A)
    matrix2 = asarray(B)
    assert matrix1.shape == matrix2.shape
    assert matrix1.shape == matrix1.T.shape
    assert (len(matrix1) & (len(matrix1) - 1)) == 0, "matrix1 is " \

```

"not a power of 2"

```
n = int(matrix1.size ** (0.5))
resultMatrix = zeros((n, n))
if (n == 1):
    resultMatrix[0] = matrix1[0] * matrix2[0]
else:
    A11 = zeros((int(n / 2), int(n / 2)))
    A12 = zeros((int(n / 2), int(n / 2)))
    A21 = zeros((int(n / 2), int(n / 2)))
    A22 = zeros((int(n / 2), int(n / 2)))
    B11 = zeros((int(n / 2), int(n / 2)))
    B12 = zeros((int(n / 2), int(n / 2)))
    B21 = zeros((int(n / 2), int(n / 2)))
    B22 = zeros((int(n / 2), int(n / 2)))
    S1 = zeros((int(n / 2), int(n / 2)))
    S2 = zeros((int(n / 2), int(n / 2)))
    S3 = zeros((int(n / 2), int(n / 2)))
    S4 = zeros((int(n / 2), int(n / 2)))
    S5 = zeros((int(n / 2), int(n / 2)))
    S6 = zeros((int(n / 2), int(n / 2)))
    S7 = zeros((int(n / 2), int(n / 2)))
    S8 = zeros((int(n / 2), int(n / 2)))
    S9 = zeros((int(n / 2), int(n / 2)))
    S10 = zeros((int(n / 2), int(n / 2)))
    P1 = zeros((int(n / 2), int(n / 2)))
    P2 = zeros((int(n / 2), int(n / 2)))
    P3 = zeros((int(n / 2), int(n / 2)))
    P4 = zeros((int(n / 2), int(n / 2)))
    P5 = zeros((int(n / 2), int(n / 2)))
    P6 = zeros((int(n / 2), int(n / 2)))
    P7 = zeros((int(n / 2), int(n / 2)))
    C11 = zeros((int(n / 2), int(n / 2)))
    C12 = zeros((int(n / 2), int(n / 2)))
    C21 = zeros((int(n / 2), int(n / 2)))
    C22 = zeros((int(n / 2), int(n / 2)))

    for i in range(0, int(n / 2)):
        for j in range(0, int(n / 2)):
            A11[i, j] = matrix1[i, j]
            A12[i, j] = matrix1[i, j + int(n / 2)]
            A21[i, j] = matrix1[i + int(n / 2), j]
            A22[i, j] = matrix1[i + int(n / 2), j + int(n / 2)]
            B11[i, j] = matrix2[i, j]
            B12[i, j] = matrix2[i, j + int(n / 2)]
            B21[i, j] = matrix2[i + int(n / 2), j]
            B22[i, j] = matrix2[i + int(n / 2), j + int(n / 2)]

    for i in range(0, int(n / 2)):
        for j in range(0, int(n / 2)):
            S1[i, j] = B12[i, j] - B22[i, j]
            S2[i, j] = A11[i, j] + A12[i, j]
            S3[i, j] = A21[i, j] + A22[i, j]
            S4[i, j] = B21[i, j] - B11[i, j]
            S5[i, j] = A11[i, j] + A22[i, j]
            S6[i, j] = B11[i, j] + B22[i, j]
            S7[i, j] = A12[i, j] - A22[i, j]
            S8[i, j] = B21[i, j] + B22[i, j]
            S9[i, j] = A11[i, j] - A21[i, j]
            S10[i, j] = B11[i, j] + B12[i, j]

    P1 = square_matrix_multiply_strassens(A11, S1)
    P2 = square_matrix_multiply_strassens(S2, B22)
```

```

P3 = square_matrix_multiply_strassens(S3, B11)
P4 = square_matrix_multiply_strassens(A22, S4)
P5 = square_matrix_multiply_strassens(S5, S6)
P6 = square_matrix_multiply_strassens(S7, S8)
P7 = square_matrix_multiply_strassens(S9, S10)

for i in range(0, int(n / 2)):
    for j in range(0, int(n / 2)):
        C11[i, j] = P5[i, j] + P4[i, j] - P2[i, j] + P6[i, j]
        C12[i, j] = P1[i, j] + P2[i, j]
        C21[i, j] = P3[i, j] + P4[i, j]
        C22[i, j] = P5[i, j] + P1[i, j] - P3[i, j] - P7[i, j]

    for i in range(0, int(n / 2)):
        for j in range(0, int(n / 2)):
            resultMatrix[i, j] = C11[i, j]
            resultMatrix[i, j + int(n / 2)] = C12[i, j]
            resultMatrix[i + int(n / 2), j] = C21[i, j]
            resultMatrix[i + int(n / 2), j + int(n / 2)] = C22[i, j]

return resultMatrix

# =====

def test():
    # Unacceptable formats of input (like, empty arrays) will not be
    # tested as such situations are taken care by assertion fails within
    # functions.

    print("=====")
    print("Below are the Test cases for each function, whose result is ")
    print("either PASSED or FAILED.")
    print("Data from CLRS as well as randomly generated data is used. ")
    print("=====")
    print("First, we test our maximum sub-array functions.")
    print(" ")

    random.seed(2)
    #
    print(BLUE + BOLD + "Case1: For the book data:" + END)
    if (find_maximum_subarray_brute(STOCK_PRICE_CHANGES) == (7, 10)):
        print("    Brute Force Method ::-> " + UNDERLINE + "PASSED" + END)
    else:
        print("    Brute Force Method ::-> FAILED")
    if (find_maximum_subarray_recursive(STOCK_PRICE_CHANGES) == (7, 10)):
        print("    Recursive Method ::-> " + UNDERLINE + "PASSED" + END)
    else:
        print("    Recursive Method ::-> FAILED")
    if (find_maximum_subarray_iterative(STOCK_PRICE_CHANGES) == (7, 10)):
        print("    Iterative Method ::-> " + UNDERLINE + "PASSED" + END)
    else:
        print("    Iterative Method ::-> FAILED")

    #
    print(BLUE + BOLD + "Case2: For large inputs in large range:" + END)
    print("(Verifying if recursive & Iterative methods is same as brute method)")
    example2 = random.randint(-150, 150, 300)

```

```

if (find_maximum_subarray_recursive(example2)
    == find_maximum_subarray_brute(example2)):
    print("    Recursive Method ::-> " + UNDERLINE + "PASSED" + END)
else:
    print("    Recursive Method ::-> FAILED")
if (find_maximum_subarray_iterative(example2)
    == find_maximum_subarray_brute(example2)):
    print("    Iterative Method ::-> " + UNDERLINE + "PASSED" + END)
else:
    print("    Iterative Method ::-> FAILED")

print(" ")
print("Test cases for square matrix multiply methods.")
print(" ")

print(BLUE + BOLD + "Case3: For large inputs:" + END)
examplem5a = asarray([[1, 2, 3, 4],
                      [1, 2, 3, 4],
                      [1, 2, 3, 4],
                      [1, 2, 3, 4]])

examplem5b = asarray([[1, 2, 3, 4],
                      [1, 2, 3, 4],
                      [1, 2, 3, 4],
                      [1, 2, 3, 4]])

expectedm5 = asarray([[10, 20, 30, 40],
                      [10, 20, 30, 40],
                      [10, 20, 30, 40],
                      [10, 20, 30, 40]])

solutionm5a = square_matrix_multiply(examplem5a, examplem5b)
solutionm5b = square_matrix_multiply_strassen(examplem5a, examplem5b)

if (array_equal(solutionm5a, expectedm5)):
    print("    Brute Force Method ::-> " + UNDERLINE + "PASSED" + END)
else:
    print("    Brute Force Method ::-> FAILED")

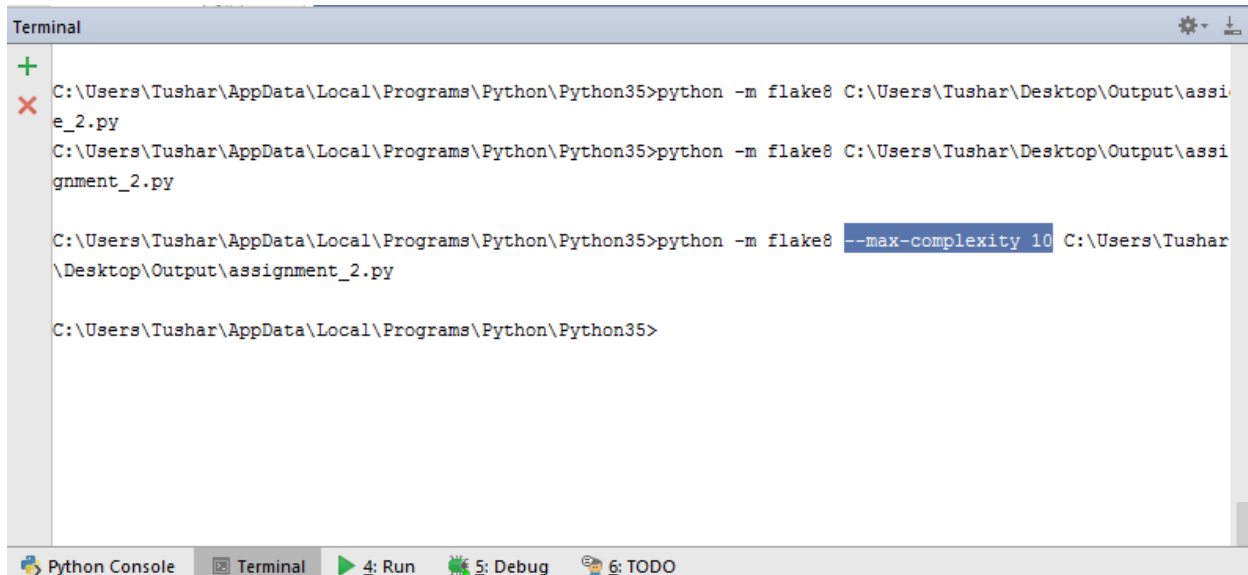
if (array_equal(solutionm5b, expectedm5)):
    print("    Strassen's Method ::-> " + UNDERLINE + "PASSED" + END)
else:
    print("    Strassen's Method ::-> FAILED")

if __name__ == '__main__':
    test()
#=====

```

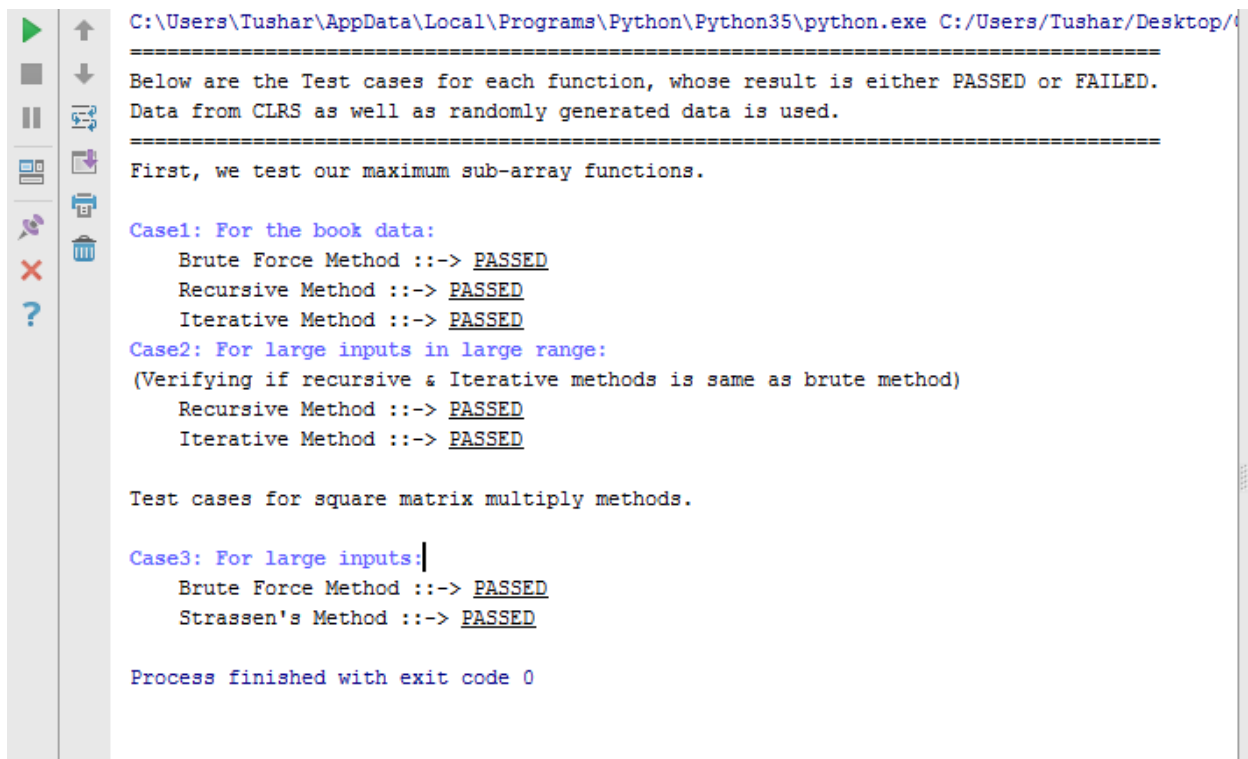
Test Results:

➤ Running the McCabe complexity command and error free console



```
Terminal
C:\Users\Tushar\AppData\Local\Programs\Python\Python35>python -m flake8 C:\Users\Tushar\Desktop\Output\assignment_2.py
C:\Users\Tushar\AppData\Local\Programs\Python\Python35>python -m flake8 --max-complexity 10 C:\Users\Tushar\Desktop\Output\assignment_2.py
C:\Users\Tushar\AppData\Local\Programs\Python\Python35>
```

➤ Screenshots of output



```
C:\Users\Tushar\AppData\Local\Programs\Python\Python35\python.exe C:/Users/Tushar/Desktop/4
=====
Below are the Test cases for each function, whose result is either PASSED or FAILED.
Data from CLRS as well as randomly generated data is used.
=====
First, we test our maximum sub-array functions.

Case1: For the book data:
  Brute Force Method ::-> PASSED
  Recursive Method ::-> PASSED
  Iterative Method ::-> PASSED

Case2: For large inputs in large range:
(Verifying if recursive & Iterative methods is same as brute method)
  Recursive Method ::-> PASSED
  Iterative Method ::-> PASSED

Test cases for square matrix multiply methods.

Case3: For large inputs:
  Brute Force Method ::-> PASSED
  Strassen's Method ::-> PASSED

Process finished with exit code 0
```