Name : Pratik Suryawanshi

ASU ID : 1213231238

ASURITE : psuryawa

Implement functions to solve the maximum sub-array problem

Using the brute-force method

Code :

```
def find_maximum_subarray_brute(A):
    theMaxSum = A[0]
    i = 0
    left = i
    right = i
    sum = 0
    while(i < len(A)):
        j = i
        sum = 0
        while(j < len(A)):
            sum += A[j]
            if(theMaxSum < sum):
                theMaxSum = sum
                left = i
                right = j
            j = j + 1
        i = i + 1
    print("find_maximum_subarray_brute output")
    print("Start Index : ", left)
    print("End Index : ", right)
    print("MaxSUM : ", theMaxSum)
    print("MaxSum sub array : ", tuple(A[left:right+1]))
    return (left, right)
```

Explaination : In above code, array is iterated for every index i and index j which is greater than i. All the non-repeating possible combinations of array are tried and sum is calculated within that range. This sum is compared with theMaxSum which is keeping track of max sum till current condition along with indexes of that max sum in left and right variables. Hence after outer while loop is done, theMaxSum will have max sum till now and left, right indexes will show array indexes enclosing max sum subarray. Complexity of this algorithm is **O(n^2)**

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

Output :

```
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501> flake8 --max-complexity 10 assignment_2.py
assignment_2.py:6:80: E501 line too long (89 > 79 characters)
PS C:\Users\Pratik\Desktop\501> python  assignment_2.py
find_maximum_subarray_brute output
Start Index :  7
End Index :  10
MaxSUM :  43
MaxSum sub array :  (18, 20, -7, 12)
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
```

B. Using the recursive method

Code :

```
def find_maximum_crossing_subarray(A, low, mid, high):
    leftSum = A[mid]
    tempLeftSum = A[mid]
    i = mid - 1
    lIndex = mid
    while (i >= low):
        tempLeftSum = tempLeftSum + A[i]
        if (leftSum < tempLeftSum):
            leftSum = tempLeftSum
            lIndex = i
        i = i - 1
```

```
    rightSum = A[mid+1]
    tempRightSum = A[mid+1]
    i = mid + 2
    rIndex = i - 1
    while (i <= high):
        tempRightSum = tempRightSum + A[i]
        if (rightSum < tempRightSum):
            rightSum = tempRightSum
            rIndex = i
        i = i + 1
    return ((lIndex, rIndex), leftSum + rightSum)


# The recursive method to solve max subarray problem
def find_maximum_subarray_recursive_helper(A, low=0, high=-1):
    if (low == high):
        return ((low, high), A[low])
    else:
        midPoint = int((low + high)/2)

    leftArr = find_maximum_subarray_recursive_helper(A, low, midPoint)
    rightArr = find_maximum_subarray_recursive_helper(A, midPoint + 1,
high)
    crossArr = find_maximum_crossing_subarray(A, low, midPoint, high)

    maxSum = max(leftArr[1], rightArr[1], crossArr[1])
    for i in (leftArr, rightArr, crossArr):
        if(i[1] == maxSum):
            return i
```

```
# The recursive method to solve max subarray problem
def find_maximum_subarray_recursive(A):
    output = find_maximum_subarray_recursive_helper(A, 0, len(A) - 1)
    print("\nfind_maximum_subarray_recursive output")
    print("start and end Index touple : ", output[0])
    print("max sum : ", output[1])
    print("max sub array : ", A[output[0][0]:output[0][1]+1])
    return output[0]
```

Explaination :

In above code, I have used divide and conquer approach to solve this problem. Given array is divided into two halves to find max subarray recursively (from leftmost to mid and mid+1 to rightmost index) and `find_maximum_crossing_subarray` function is used to get max sum array possible including middle element in linear time . Max sum from left array, right array and crossing subarray is compared to determine which is largest sum subarray in given array, index touple along with max sum is returned as output from helper function and only index touple is returned from `find_maximum_subarray_recursive` function.

Time complexity of this function can be give as $T(n) = 2T(n/2) + \Theta(n)$.

Solution of this recurrence from Master theorem is $\Theta(n \text{ Logn})$

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

Output:

```
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501> flake8 --max-complexity 10 assignment_2.py
assignment_2.py:6:80: E501 line too long (89 > 79 characters)
PS C:\Users\Pratik\Desktop\501> python  assignment_2.py

find_maximum_subarray_recursive output
start and end Index touple :  (7, 10)
max sum :  43
max sub array :  [18, 20, -7, 12]
PS C:\Users\Pratik\Desktop\501>
```

C. Using the iterative method

Code :

```
def find_maximum_subarray_iterative(A):
    maxSum = A[0]
    tempSum = 0
    left = 0
    right = 0
    index = 0
    i = 0
    while (i < len(A)):
        tempSum = tempSum + A[i]
        if (maxSum < tempSum):
            maxSum = tempSum
            left = index
            right = i
        if (tempSum < 0):
            tempSum = 0
            index = i + 1
        i = i + 1
    print("find_maximum_subarray_iterative output")
    print("left Index : ", left)
    print("right Index : ", right)
    print("maxSum : ", maxSum)
    print("MaxSum sub array : ", tuple(A[left:right+1]))

    return (left, right)
```

Explaination : In above code, linear algorithm is developed to find max sub array for array A. In code, tempSum keeps track of computed sum till now plus current elemet and maxSum is max sum till now. If adding next element increases the sum till now then element is added into maxSum and loop continues keeping right index updated. If element decreases sum till now then that element is skipped and array is iterated to find element greater than maxSum. If there exists such element then left index is updated and loop continues.

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

Output :

```
PROBLEMS    OUTPUT    DEBUG CONSOLE    TERMINAL    1:

PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501> flake8 --max-complexity 10 assignment_2.py
assignment_2.py:6:80: E501 line too long (89 > 79 characters)
PS C:\Users\Pratik\Desktop\501> python  assignment_2.py
find_maximum_subarray_iterative output
left Index :  7
right Index :  10
maxSum :  43
MaxSum sub array :  (18, 20, -7, 12)
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501>
```

implement functions to calculate the product AB:

Using matrix multiplication

Code :

```
def square_matrix_multiply(A, B):
    A = asarray(A)
    B = asarray(B)
    assert A.shape == B.shape
    assert A.shape == A.T.shape
    if (len(A) == 0 or len(B) == 0):
        return None
    w, h = len(A), len(A[0])
    Output = [[0 for x in range(w)] for y in range(h)]
    for i in range(len(A)):
        for j in range(len(B[0])):
            for k in range(len(B)):
                Output[i][j] += A[i][k] * B[k][j]
```

```
    print("matrix A :")
    print(A)
    print("matrix B : ")
    print(B)
    print("output : ")
    print(array(Output))
    return array(Output)
```

Explanation : In above code, simple matrix multiplication is done using 3 for loops. Initially assertions are added to check if A has same dimensions as B and that both are square matrices. Later simple matrix multiplication is performed stored in Output variable. Time complexity for this is $O(n^2)$

Input :

A = [[1, 2], [3, 4]]

B = [[4, 3], [2, 1]]

Output :

```
PS C:\Users\Pratik\Desktop\501>
PS C:\Users\Pratik\Desktop\501> flake8 --max-complexity 10 assignment_2.py
assignment_2.py:7:80: E501 line too long (89 > 79 characters)
PS C:\Users\Pratik\Desktop\501> python  assignment_2.py
matrix A :
[[1 2]
 [3 4]]
matrix B :
[[4 3]
 [2 1]]
output :
[[ 8  5]
 [20 13]]
PS C:\Users\Pratik\Desktop\501>
```

Using Strassen's Algorithm.

Code :

```
def add_matrix(A, B):
    length = len(A)
    result = [[0 for j in range(0, length)] for i in range(0, length)]
    for i in range(0, length):
        for j in range(0, length):
            result[i][j] = A[i][j] + B[i][j]
    return result
def substract_matrix(A, B):
    length = len(A)
    result = [[0 for j in range(0, length)] for i in range(0, length)]
    for i in range(0, length):
        for j in range(0, length):
            result[i][j] = A[i][j] - B[i][j]
    return result

def square_matrix_multiply_strassens(A, B):
    A = asarray(A)
    B = asarray(B)
    if len(A) == 0:
        return None
    assert A.shape == B.shape
    assert A.shape == A.T.shape
    if len(A) == 1:
        result = [0]
        result[0] = A[0]*B[0]
        return result
    assert (len(A) & (len(A) - 1)) == 0, "A is not a power of 2"
    length = len(A)
```

```
    newLen = int(length/2)

    A_top_left = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    A_top_right = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    A_bot_left = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    A_bot_right = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    B_top_left = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    B_top_right = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    B_bot_left = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    B_bot_right = [[0 for k in range(0, newLen)] for i in range(0,
newLen)]

    for i in range(0, newLen):

        for j in range(0, newLen):

            A_top_left[i][j] = A[i][j]

            A_top_right[i][j] = A[i][j + newLen]

            A_bot_left[i][j] = A[i + newLen][j]

            A_bot_right[i][j] = A[i + newLen][j + newLen]

            B_top_left[i][j] = B[i][j]

            B_top_right[i][j] = B[i][j + newLen]

            B_bot_left[i][j] = B[i + newLen][j]

            B_bot_right[i][j] = B[i + newLen][j + newLen]


    expA = [[0 for k in range(0, newLen)] for i in range(0, newLen)]

    expB = [[0 for k in range(0, newLen)] for i in range(0, newLen)]


    expA = add_matrix(A_top_left, A_bot_right)

    expB = add_matrix(B_top_left, B_bot_right)
```

```
m1 = square_matrix_multiply_strassens(expA, expB)

expA = add_matrix(A_bot_left, A_bot_right)
m2 = square_matrix_multiply_strassens(expA, B_top_left)

expB = substract_matrix(B_top_right, B_bot_right)
m3 = square_matrix_multiply_strassens(A_top_left, expB)

expB = substract_matrix(B_bot_left, B_top_left)
m4 = square_matrix_multiply_strassens(A_bot_right, expB)

expA = add_matrix(A_top_left, A_top_right)
m5 = square_matrix_multiply_strassens(expA, B_bot_right)

expA = substract_matrix(A_bot_left, A_top_left)
expB = add_matrix(B_top_left, B_top_right)
m6 = square_matrix_multiply_strassens(expA, expB)

expA = substract_matrix(A_top_right, A_bot_right)
expB = add_matrix(B_bot_left, B_bot_right)
m7 = square_matrix_multiply_strassens(expA, expB)

expA = add_matrix(m1, m4)
expB = add_matrix(expA, m7)
r11 = substract_matrix(expB, m5)
r12 = add_matrix(m3, m5)
r21 = add_matrix(m2, m4)

expB = add_matrix(add_matrix(m1, m3), m6)
r22 = substract_matrix(expB, m2)
```

```
    Result = [[0 for k in range(0, length)] for i in range(0, length)]
    for i in range(0, newLen):
        for j in range(0, newLen):
            Result[i][j] = r11[i][j]
            Result[i][j + newLen] = r12[i][j]
            Result[i + newLen][j] = r21[i][j]
            Result[i + newLen][j + newLen] = r22[i][j]
    return array(Result)
```

Explaination :

In above code, Strassen's matrix multiplication is implemented. This algorithm recursively divides the given matrices into 4 parts and applying certain formulas computed by m1,m2.. m7 determines output product of matrix A and B. Assertions are done to check if input matrices has dimensions in power of 3 and that it is square matrix. Next , matrices are divided into quarters and formula are applied. In formula whenever there is matrix multiplication, recursive call is done to get result. Finally output is calculated and stored in Result and returned.

`substract_matrix(A, B)` Used for substraction of 2 matrices

`add_matrix(A, B)` Used for addition of 2 matrices

Input :

A = [[1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1], [1, 1, 1, 1]]

B = [[2, 2, 2, 2], [2, 2, 2, 2], [2, 2, 2, 2], [2, 2, 2, 2]]

Output :