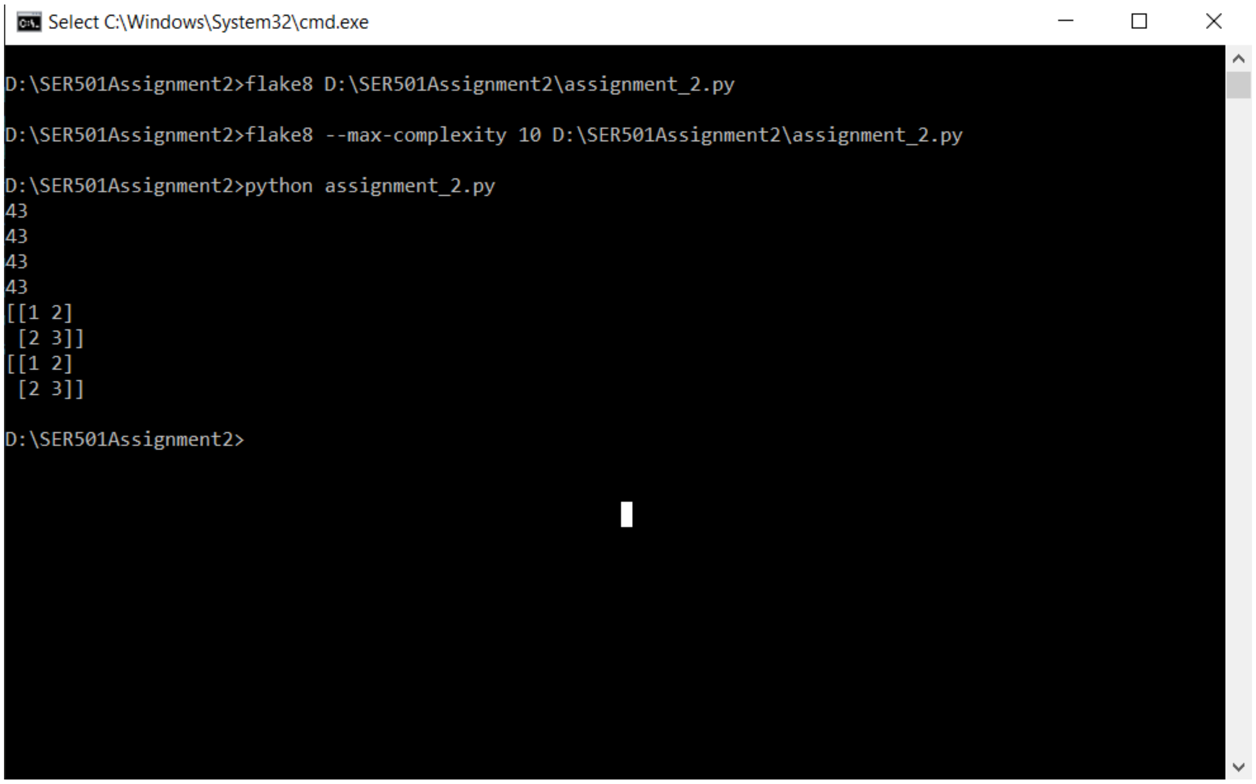
Below is a screenshot demonstrating the execution of the flake8 error checking, the execution of the flake8 max complexity checking, and the execution of assignment_2.py



```
D:\SER501Assignment2>flake8 D:\SER501Assignment2\assignment_2.py

D:\SER501Assignment2>flake8 --max-complexity 10 D:\SER501Assignment2\assignment_2.py

D:\SER501Assignment2>python assignment_2.py
43
43
43
43
[[1 2]
 [2 3]]
[[1 2]
 [2 3]]

D:\SER501Assignment2>
```

This demonstrates that I have successfully removed all warnings that could have been presented by flake8. The output of the program requires more explanation.

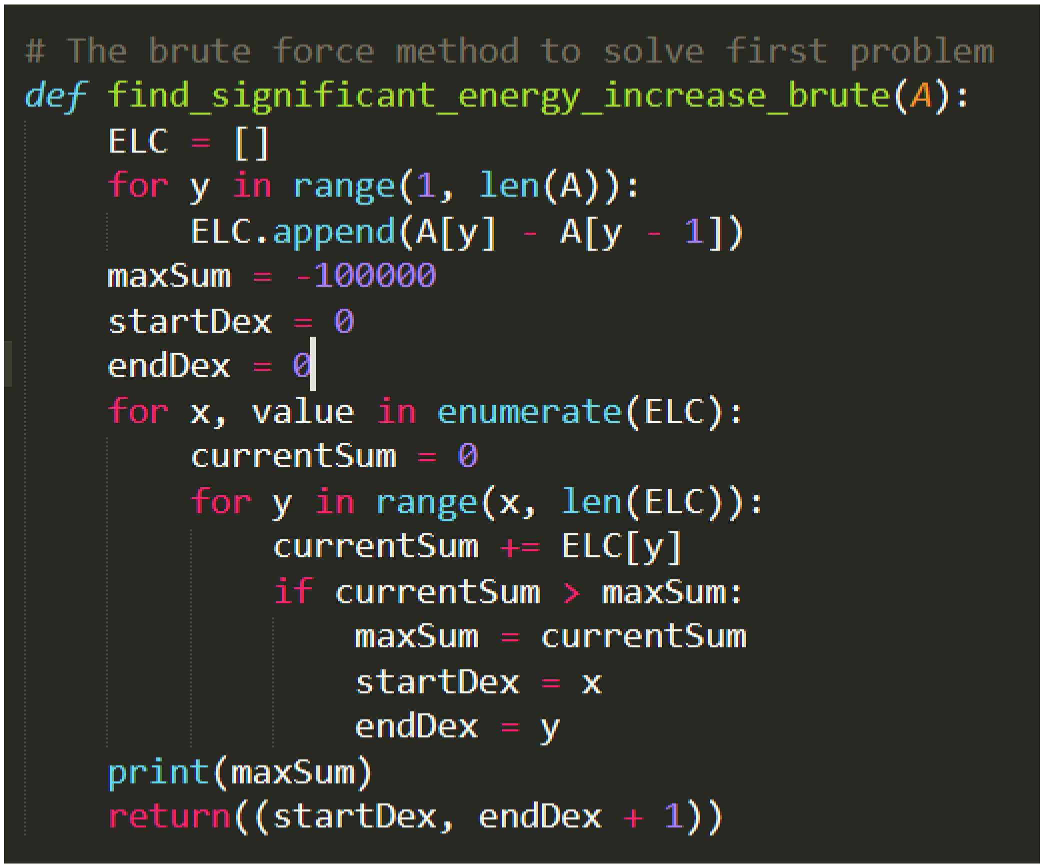
The first 43 represents the maximum sum of the subarray of solar energy level changes calculated by the brute force algorithm. The function correctly returns the tuple (7, 11), which is why there is no additional output as the result of a failed assertion.

The second and third 43's represent the execution of the recursive function used to find the maximum subarray of solar energy level changes. The first 43 printed demonstrates the correct calculation of the maximum sum of the subarray. The third 43 is printed by the helper function of the recursive function.

This helper function is used to return the correct indices containing the maximum sum. Once again, because the correct tuple is shown, no additional error output is displayed.

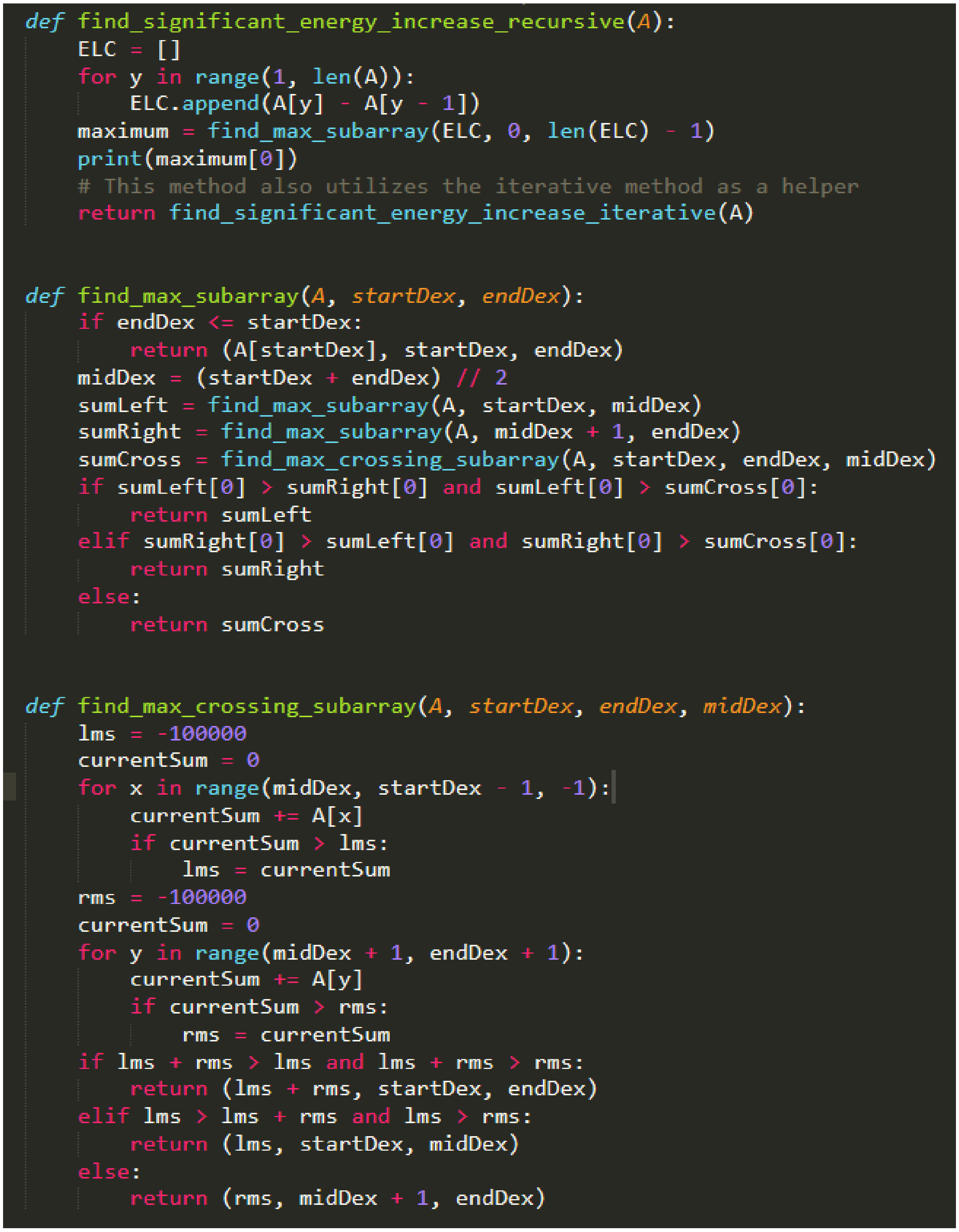
The final 43 represents the execution of the iterative function used to find the maximum subarray of solar energy level changes. As before, no errors are displayed since the correct tuple is returned.

The remaining two matrices relate to the matrix power functions, which rely on the Strassen matrix multiplication function to produce those outputs. These outputs match the expected outputs of the assert statements, which is why no additional erroneous output is shown.



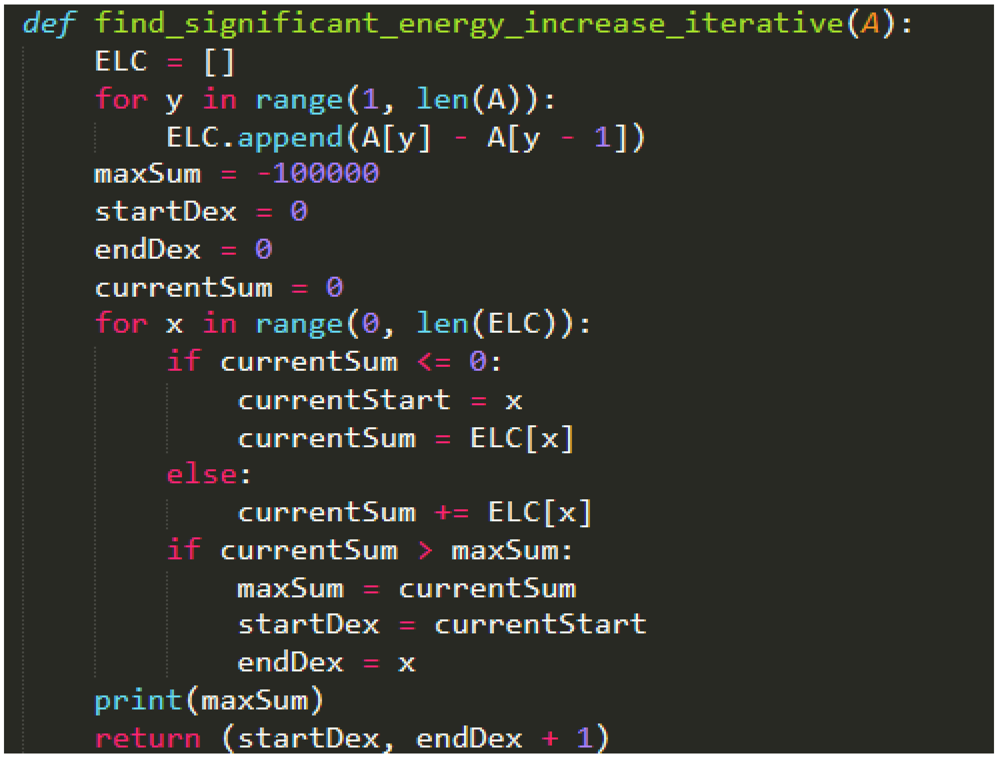
```python
# The brute force method to solve first problem
def find_significant_energy_increase_brute(A):
    ELC = []
    for y in range(1, len(A)):
        ELC.append(A[y] - A[y - 1])
    maxSum = -100000
    startDex = 0
    endDex = 0
    for x, value in enumerate(ELC):
        currentSum = 0
        for y in range(x, len(ELC)):
            currentSum += ELC[y]
            if currentSum > maxSum:
                maxSum = currentSum
                startDex = x
                endDex = y
    print(maxSum)
    return((startDex, endDex + 1))
```

The above image is the brute force function of finding the max subarray sum and its indices. This is clearly an O(n^2) complex algorithm since it must iterate over the entire list of changes for each individual change in energy. There is also a loop that iterates once over the original list of energy levels to calculate the changes in energy levels. This does not affect the complexity.

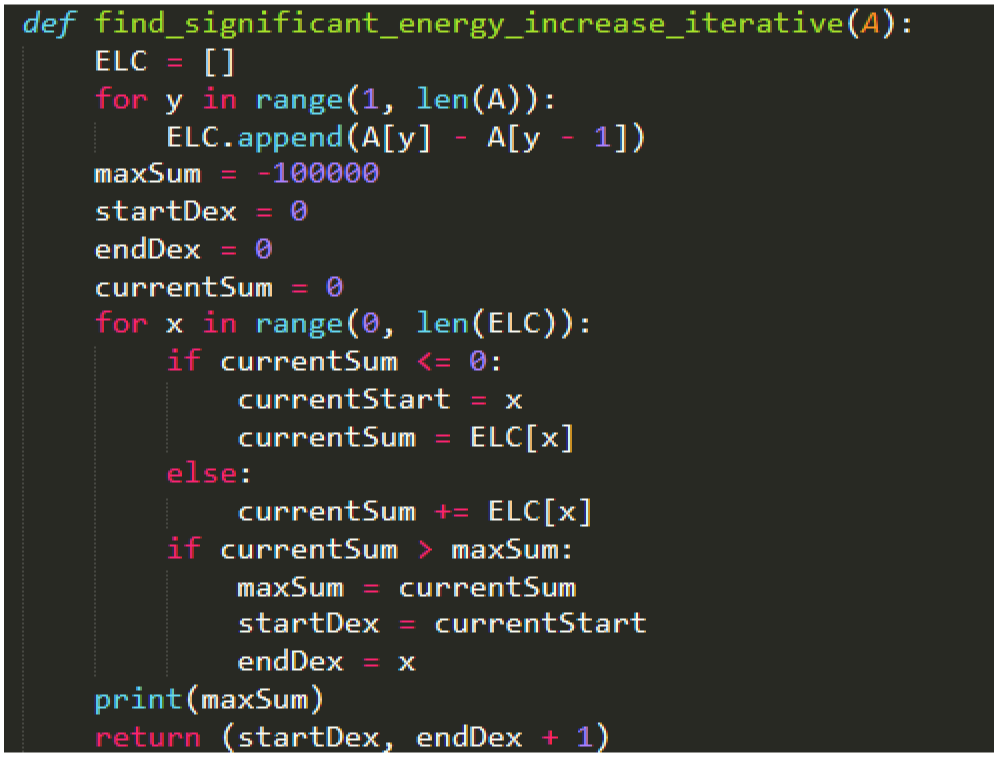


```python
def find_significant_energy_increase_recursive(A):
    ELC = []
    for y in range(1, len(A)):
        ELC.append(A[y] - A[y - 1])
    maximum = find_max_subarray(ELC, 0, len(ELC) - 1)
    print(maximum[0])
    # This method also utilizes the iterative method as a helper
    return find_significant_energy_increase_iterative(A)


def find_max_subarray(A, startDex, endDex):
    if endDex <= startDex:
        return (A[startDex], startDex, endDex)
    midDex = (startDex + endDex) // 2
    sumLeft = find_max_subarray(A, startDex, midDex)
    sumRight = find_max_subarray(A, midDex + 1, endDex)
    sumCross = find_max_crossing_subarray(A, startDex, endDex, midDex)
    if sumLeft[0] > sumRight[0] and sumLeft[0] > sumCross[0]:
        return sumLeft
    elif sumRight[0] > sumLeft[0] and sumRight[0] > sumCross[0]:
        return sumRight
    else:
        return sumCross


def find_max_crossing_subarray(A, startDex, endDex, midDex):
    lms = -100000
    currentSum = 0
    for x in range(midDex, startDex - 1, -1):
        currentSum += A[x]
        if currentSum > lms:
            lms = currentSum
    rms = -100000
    currentSum = 0
    for y in range(midDex + 1, endDex + 1):
        currentSum += A[y]
        if currentSum > rms:
            rms = currentSum
    if lms + rms > lms and lms + rms > rms:
        return (lms + rms, startDex, endDex)
    elif lms > lms + rms and lms > rms:
        return (lms, startDex, midDex)
    else:
        return (rms, midDex + 1, endDex)
```

The above three functions relate to the recursive, divide-and-conquer approach to finding the maximum sum of the subarray of energy level changes. The entire execution is O(nlogn) because of the division of the list into two separate lists and the iteration through the sublists when searching for the maximum sum across the midpoint of the list. Additionally, there is the same step taken as in the brute force function to calculate the changes in energy levels in the main function. This would be the main contributor to the n portion of the complexity of this algorithm. Worth noting is the use of the iterative function for finding the maximum subarray as a helper to find the indices containing the maximum sum. However, the divide-and-conquer function is proven to work since it correctly finds the maximum sum, which is 43.

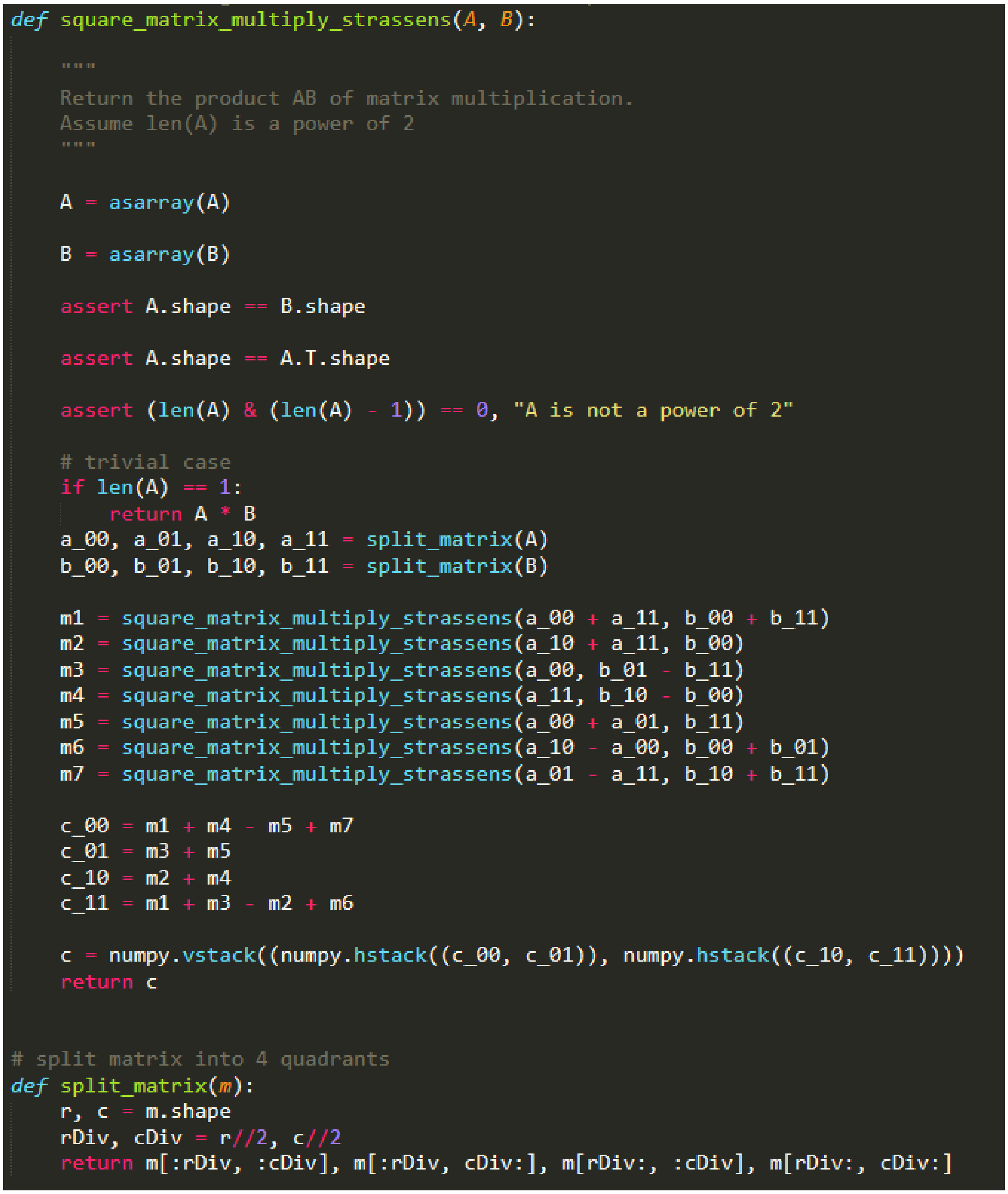


```
def find_significant_energy_increase_iterative(A):
    ELC = []
    for y in range(1, len(A)):
        ELC.append(A[y] - A[y - 1])
    maxSum = -100000
    startDex = 0
    endDex = 0
    currentSum = 0
    for x in range(0, len(ELC)):
        if currentSum <= 0:
            currentStart = x
            currentSum = ELC[x]
        else:
            currentSum += ELC[x]
        if currentSum > maxSum:
            maxSum = currentSum
            startDex = currentStart
            endDex = x
    print(maxSum)
    return (startDex, endDex + 1)
```



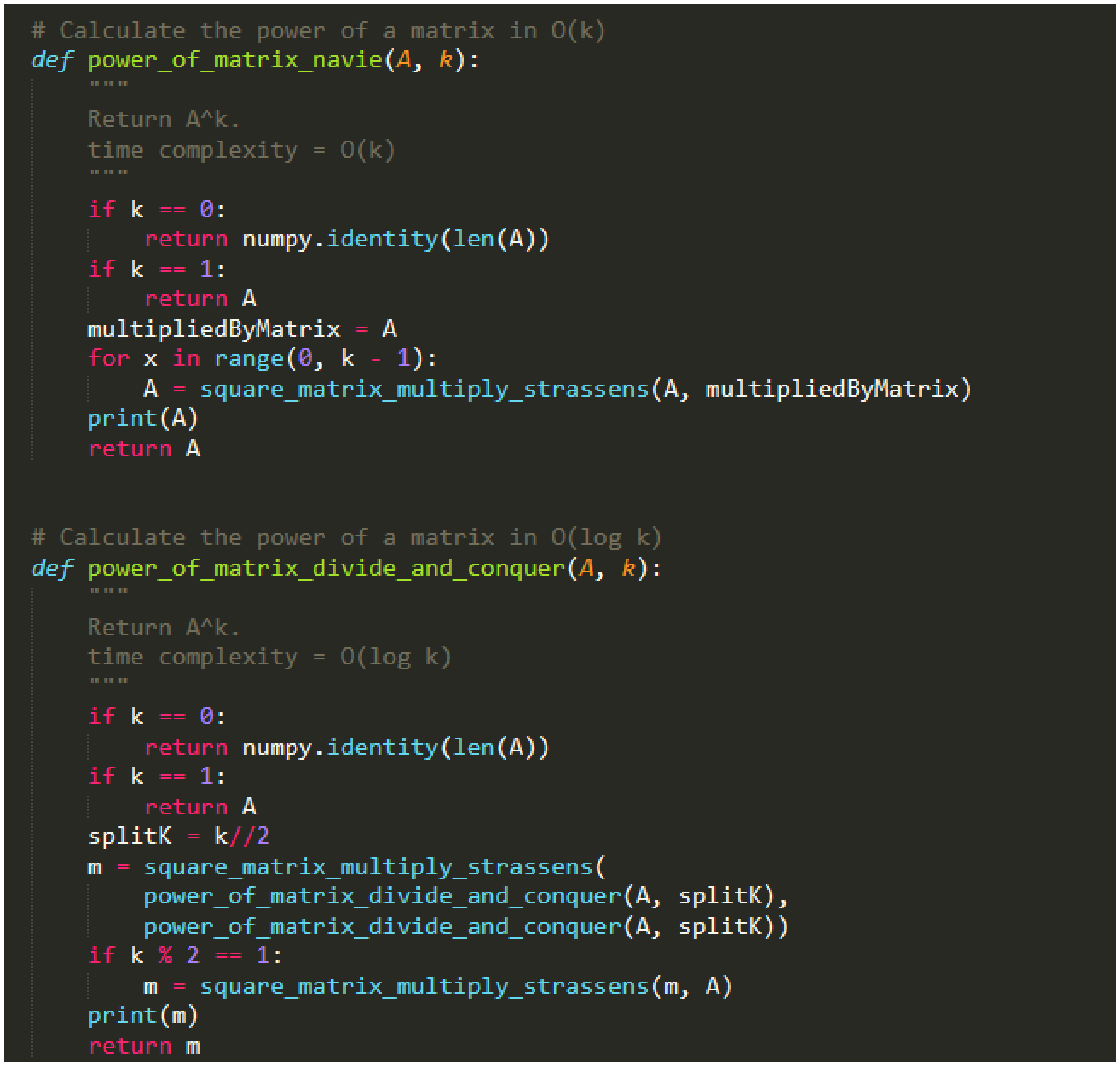
```python
def find_significant_energy_increase_iterative(A):
    ELC = []
    for y in range(1, len(A)):
        ELC.append(A[y] - A[y - 1])
    maxSum = -100000
    startDex = 0
    endDex = 0
    currentSum = 0
    for x in range(0, len(ELC)):
        if currentSum <= 0:
            currentStart = x
            currentSum = ELC[x]
        else:
            currentSum += ELC[x]
        if currentSum > maxSum:
            maxSum = currentSum
            startDex = currentStart
            endDex = x
    print(maxSum)
    return (startDex, endDex + 1)
```

Here is the iterative approach to finding the maximum sum from the subarray of energy level changes. This is O(n) since there are only two iterations through the entire list of energy levels. The first, as before, is to find the collection of energy level changes. The second is to determine the maximum sum of the subarray of energy level changes as well as the indices containing the subarray.



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

    A = asarray(A)

    B = asarray(B)

    assert A.shape == B.shape

    assert A.shape == A.T.shape

    assert (len(A) & (len(A) - 1)) == 0, "A is not a power of 2"

    # trivial case
    if len(A) == 1:
        return A * B
    a_00, a_01, a_10, a_11 = split_matrix(A)
    b_00, b_01, b_10, b_11 = split_matrix(B)

    m1 = square_matrix_multiply_strassens(a_00 + a_11, b_00 + b_11)
    m2 = square_matrix_multiply_strassens(a_10 + a_11, b_00)
    m3 = square_matrix_multiply_strassens(a_00, b_01 - b_11)
    m4 = square_matrix_multiply_strassens(a_11, b_10 - b_00)
    m5 = square_matrix_multiply_strassens(a_00 + a_01, b_11)
    m6 = square_matrix_multiply_strassens(a_10 - a_00, b_00 + b_01)
    m7 = square_matrix_multiply_strassens(a_01 - a_11, b_10 + b_11)

    c_00 = m1 + m4 - m5 + m7
    c_01 = m3 + m5
    c_10 = m2 + m4
    c_11 = m1 + m3 - m2 + m6

    c = numpy.vstack((numpy.hstack((c_00, c_01)), numpy.hstack((c_10, c_11))))
    return c


# split matrix into 4 quadrants
def split_matrix(m):
    r, c = m.shape
    rDiv, cDiv = r//2, c//2
    return m[:rDiv, :cDiv], m[:rDiv, cDiv:], m[rDiv:, :cDiv], m[rDiv:, cDiv:]
```

These two functions represent the Strassen matrix multiplication algorithm, which recursively splits the two matrices into four quadrants, multiplies those quadrants in a certain way as demonstrated by the m variables, and operates on those m variables to produce the 4 quadrants that make up the resulting matrix.



```python
# Calculate the power of a matrix in O(k)
def power_of_matrix_navie(A, k):
    """
    Return A^k.
    time complexity = O(k)
    """
    if k == 0:
        return numpy.identity(len(A))
    if k == 1:
        return A
    multipliedByMatrix = A
    for x in range(0, k - 1):
        A = square_matrix_multiply_strassens(A, multipliedByMatrix)
    print(A)
    return A


# Calculate the power of a matrix in O(log k)
def power_of_matrix_divide_and_conquer(A, k):
    """
    Return A^k.
    time complexity = O(log k)
    """
    if k == 0:
        return numpy.identity(len(A))
    if k == 1:
        return A
    splitK = k//2
    m = square_matrix_multiply_strassens(
        power_of_matrix_divide_and_conquer(A, splitK),
        power_of_matrix_divide_and_conquer(A, splitK))
    if k % 2 == 1:
        m = square_matrix_multiply_strassens(m, A)
    print(m)
    return m
```

The remaining two functions represent two approaches to finding the power of a matrix. Each function utilizes the previously defined Strassen matrix multiplication function, but executes that function in different ways and with different complexities. The naïve approach is to iterate up to the value of k and execute the matrix multiplication function on each iteration to produce the product. Because of this iterative approach, its time complexity is O(n). The divide-and-conquer approach splits the input k in half and multiplies the resulting splits with each other. If the input k is odd, the algorithm simply executes one more multiplication to produce the correct product.