**Analysis:**

**sparseMatrix.hpp**

This implementation uses CSR or Yale format. It has a elements, col\_index, and row\_index to store sparseMatrix.

sparseMatrx(int row, int cols): construct empty sparseMatrix object.

SparseMatrix(Matrix<T> & matrix): construct sparseMatrix from Matrix class. (a conversion)

sparseMatrix(const sparseMatrix<T> & other>: copy constructor.

~sparseMatrix(): default deconstructor

ConvertToDense(sparseMatrix): convert a sparseMatrix to dense Matrix.

Thus, this allows a conversion between sparseMatrix and Matrix class with ease.

**matrix.hpp**

Reshaping: in order to reshape we simply change the value of row and col if the product of the new value equal to the original size.

Slicing: The slicing operation is done in two steps. First, we find the index of the element that will be present in the slicing. This is done by finding the all the row and column index in two set. The entries of the slice are the cross product of these two sets, that is (Set of row index) x (Set of column index).

Sum: The sum function simply sums up all value of the elements in the matrix.

The Average function, Min function and Max function have two overloading. The zero-argument version compute the average, min and max for all element in the matrix. While the other version takes one argument called axis. When axis = 0, each function returns their corresponding answer along each column which result in a matrix with 1 row. When axis = 1, each function returns their corresponding answer along each row, result in a matrix of 1 column.

The average function, Min function and Max function also have a slightly different implementation when the matrix typename is not a primitive type. Two version of each function exist. Each called to the function will check if the typename is a reference type. If this is true, it will choose the implementation suited for that datatype. For instance, if the typename is complex, we will compare the modulus of two complex number instead of applying the > operator directly to the object.

Convolution: The convolution of two matrix is similar to a dot product between two matrices. In the operation, the second operand is called the kernel whose dimension must be smaller than the first operand. To get each entry of the convolution we place the kernel on the first matrix, then multiply and sum their overlapping element. Then we move the kernel and repeat the process until the kernel overlap the lower right corner of the matrix.

**templateUtil.h**

is\_same: using std::is\_same, it is a function that checks if two types are the same.

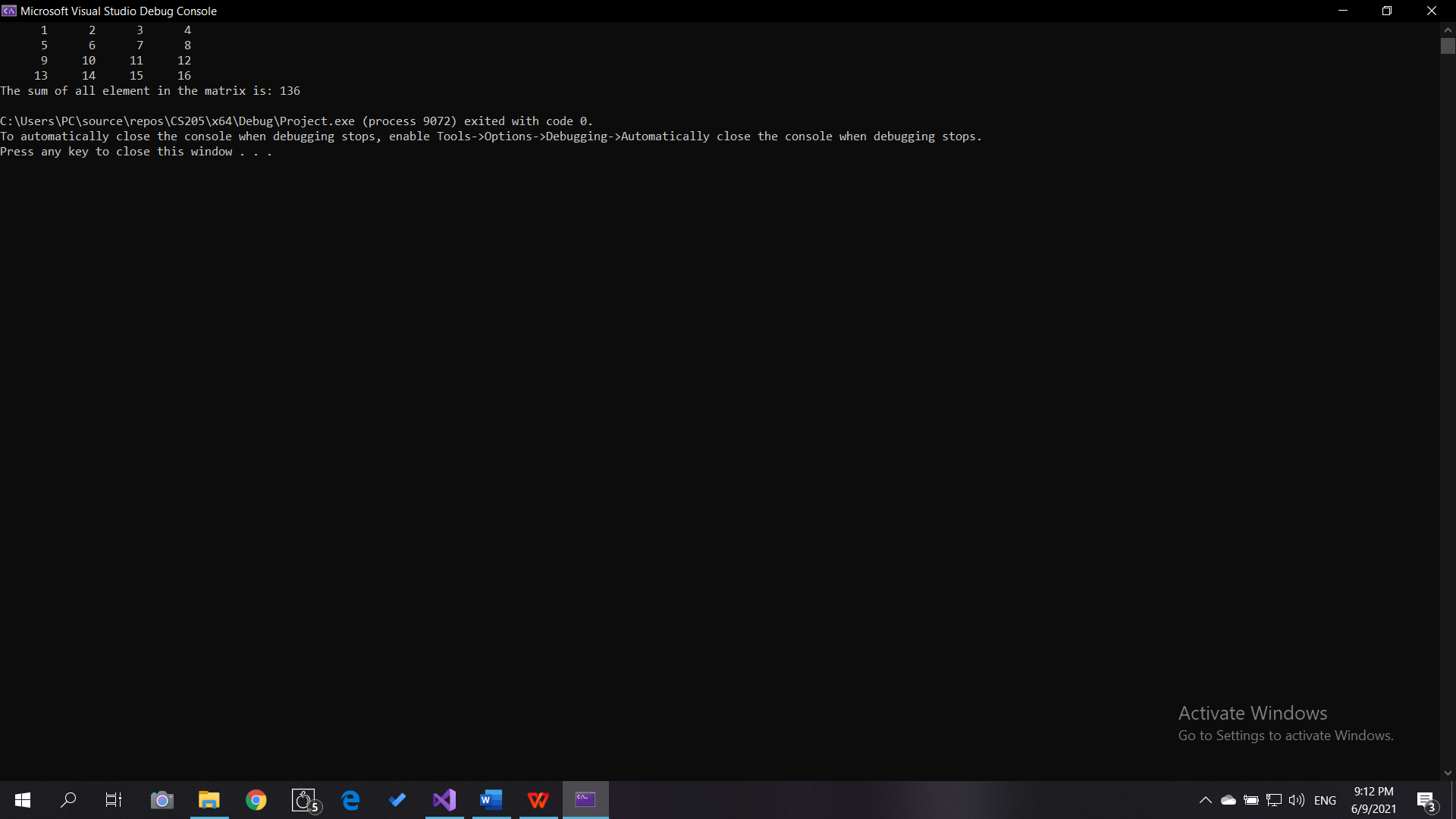
Is\_complex: by first initializing the variable is\_complex\_t to false initially, it returns true if and only if the type inside the angular bracket is std::complex<T>.

is\_arithmetic: : using std::is\_arithmetic , it is a function that checks if a type is arithmetic.

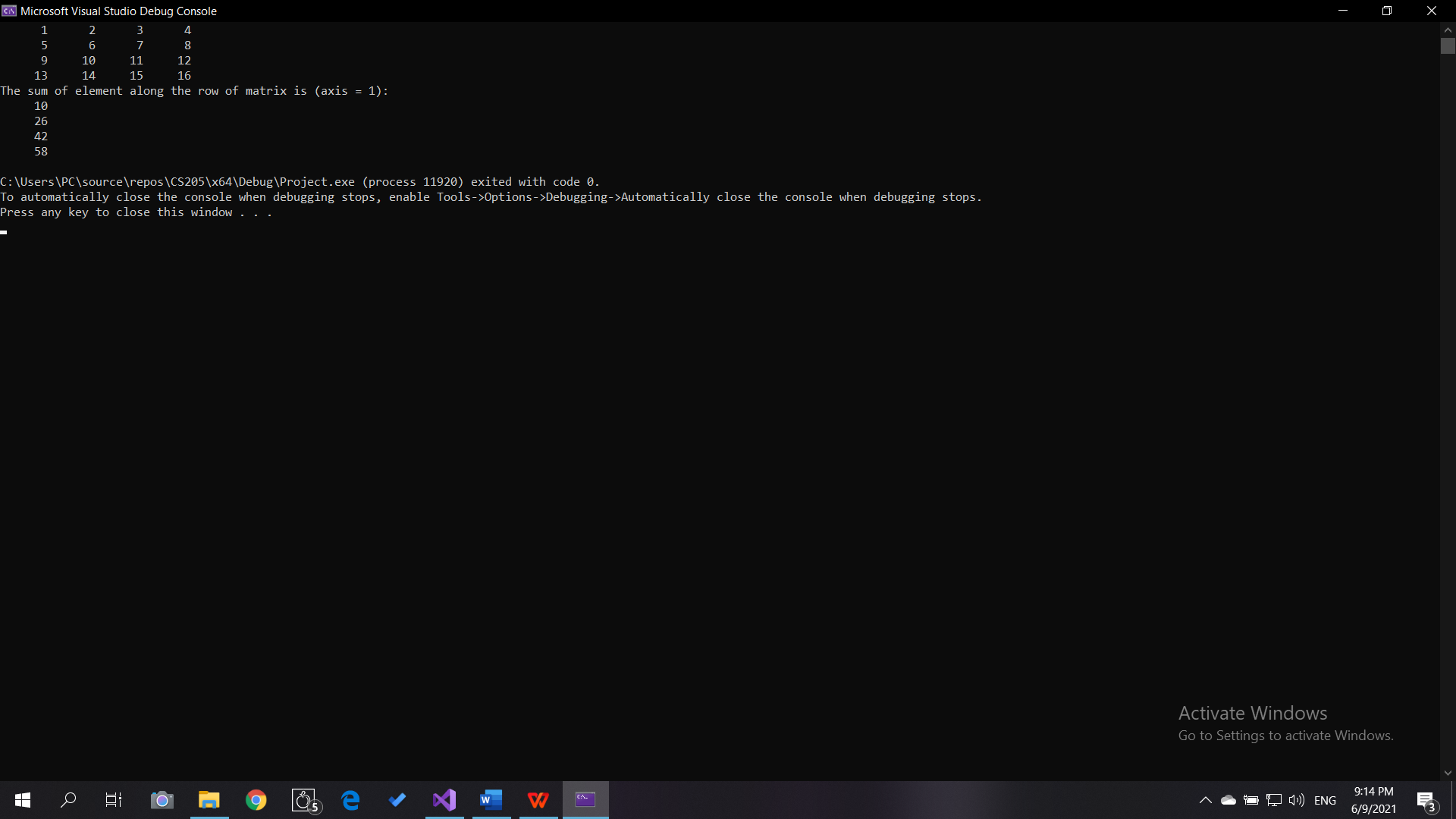
#define if (…) : macro using std:: enable\_if, it enables when the type matches.

**Result and verification:**

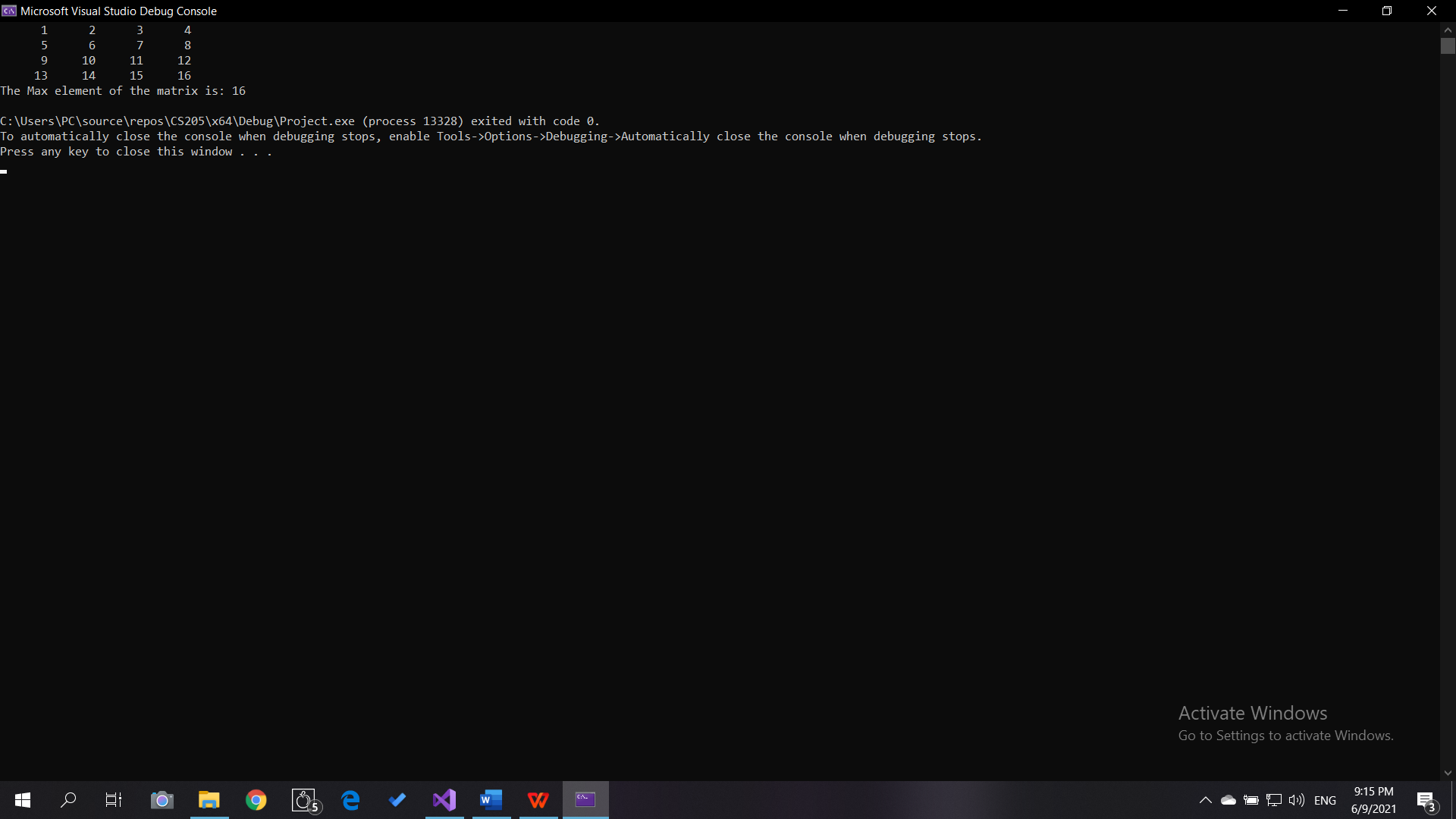
Result of Sum():



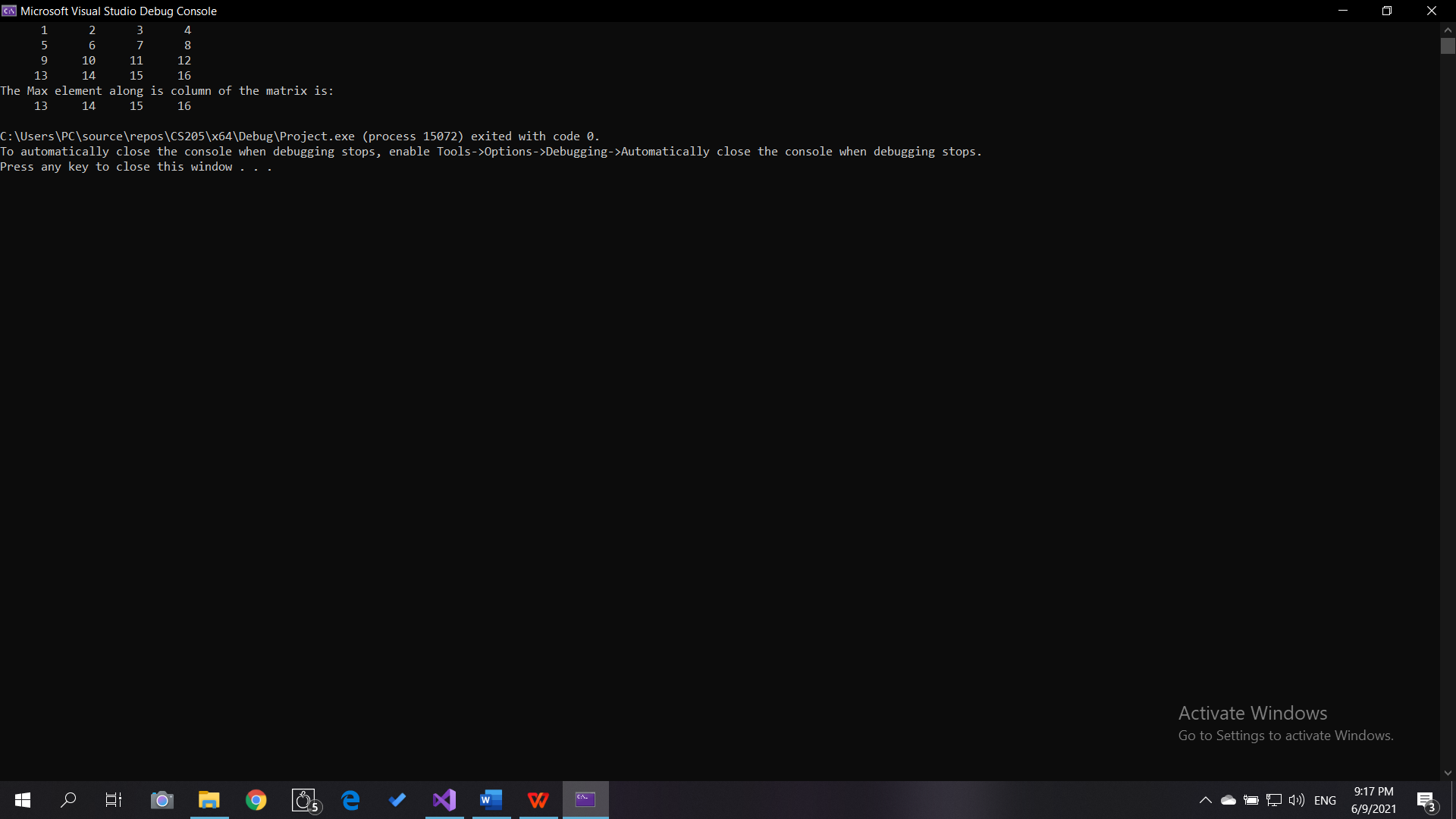
Result of Sum(1):



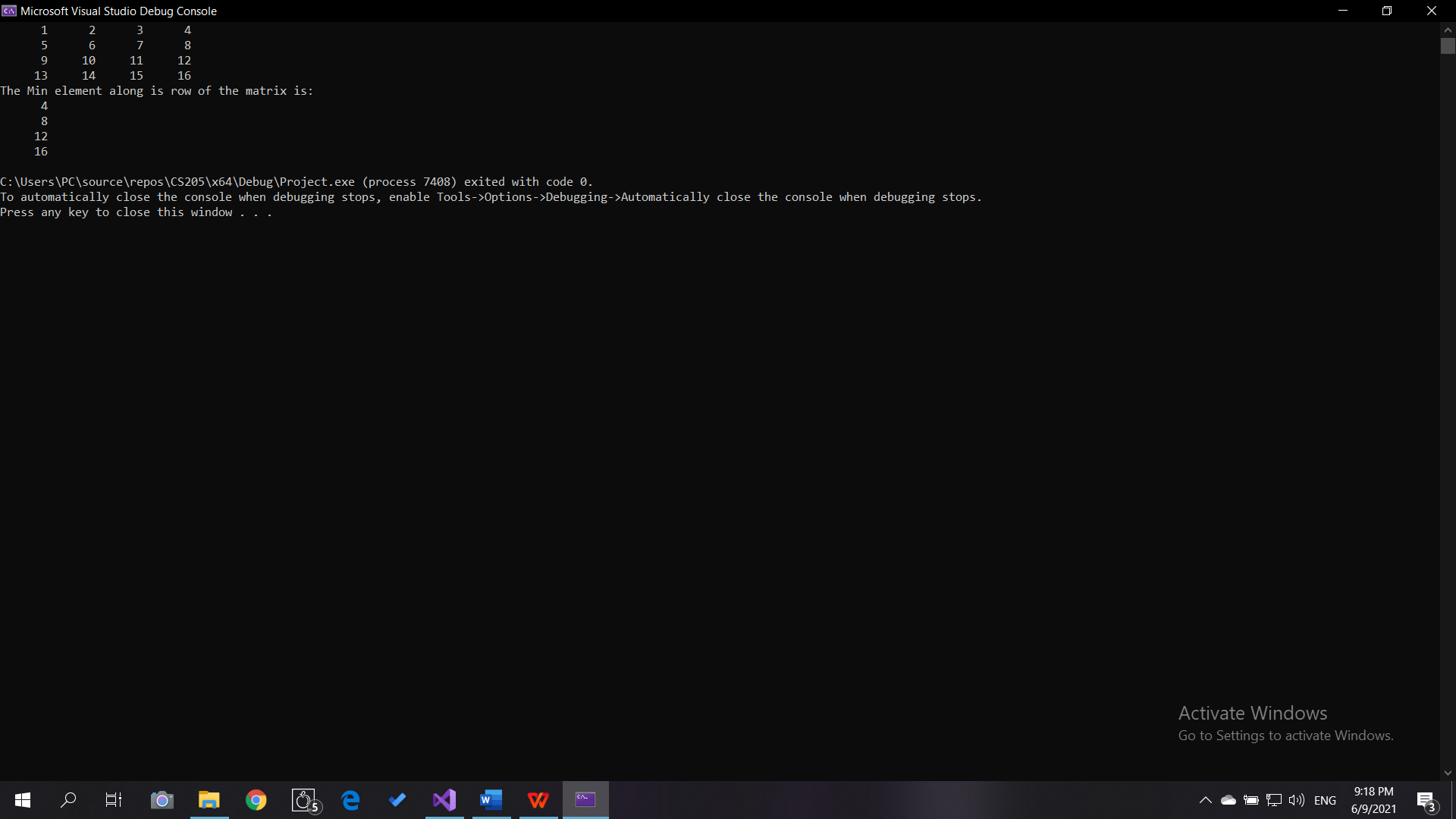
Result of Max():



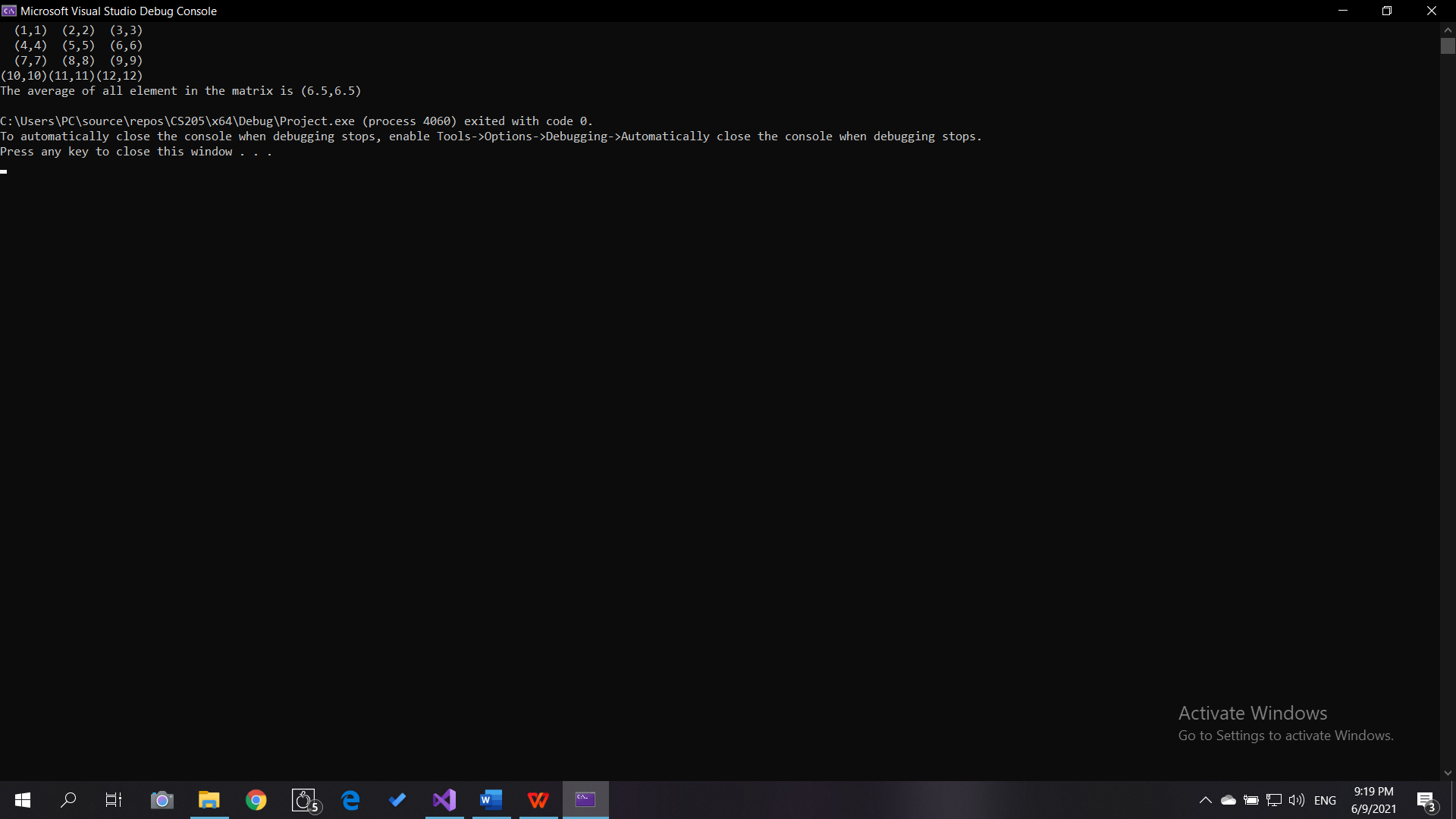
Result of Max(0):



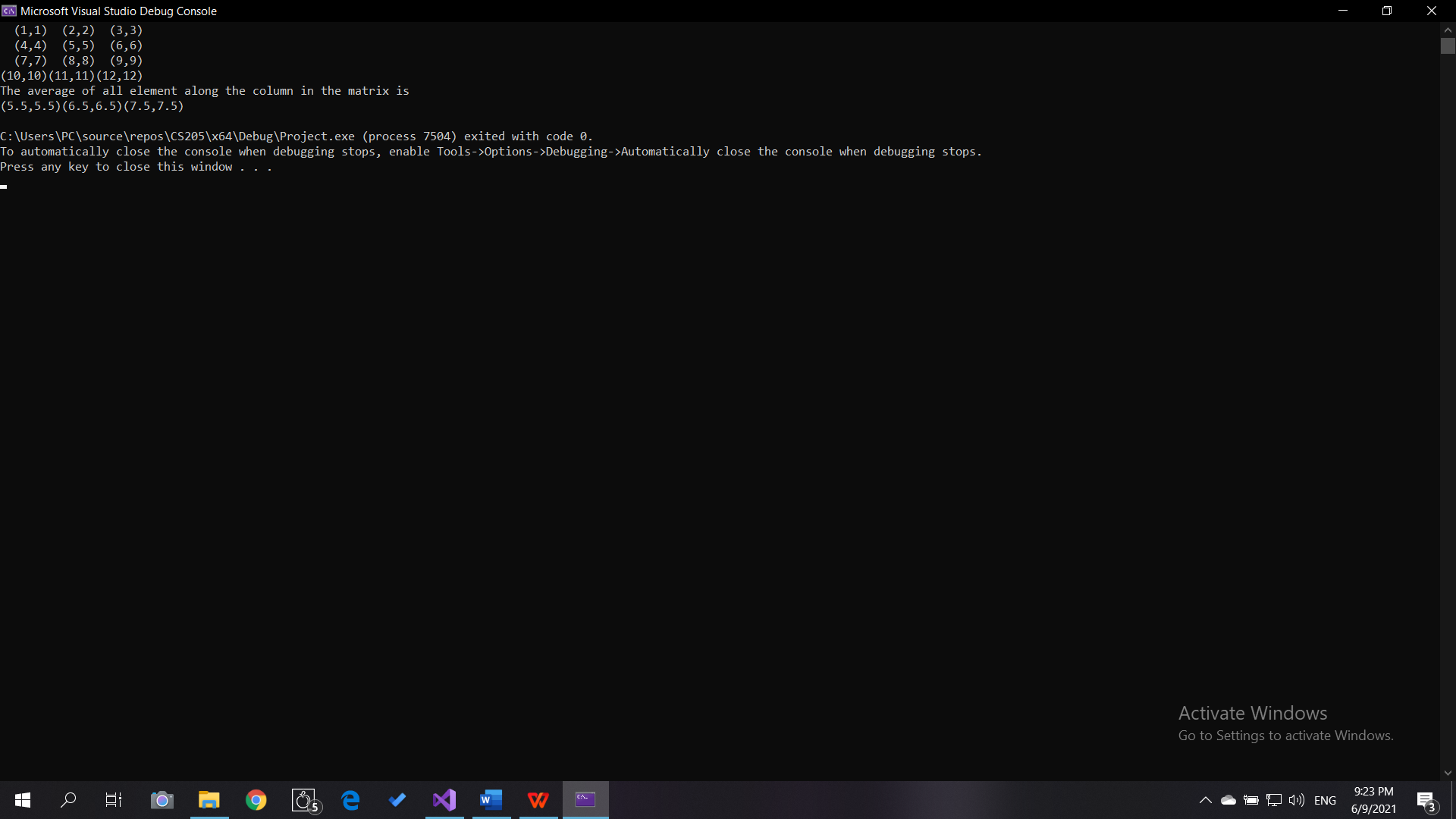
Result of Min(1):



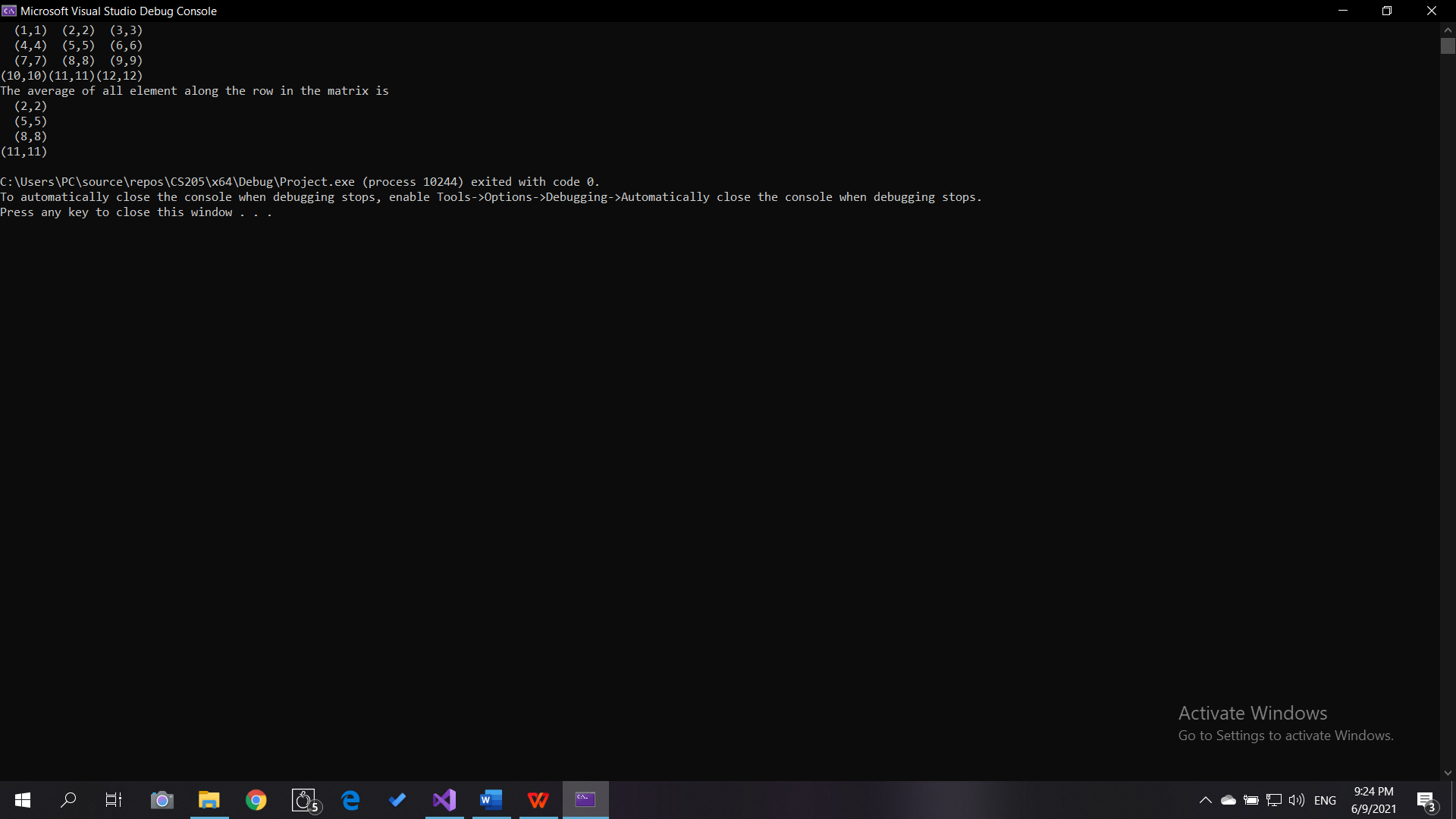
Result of Average() with complex entries:



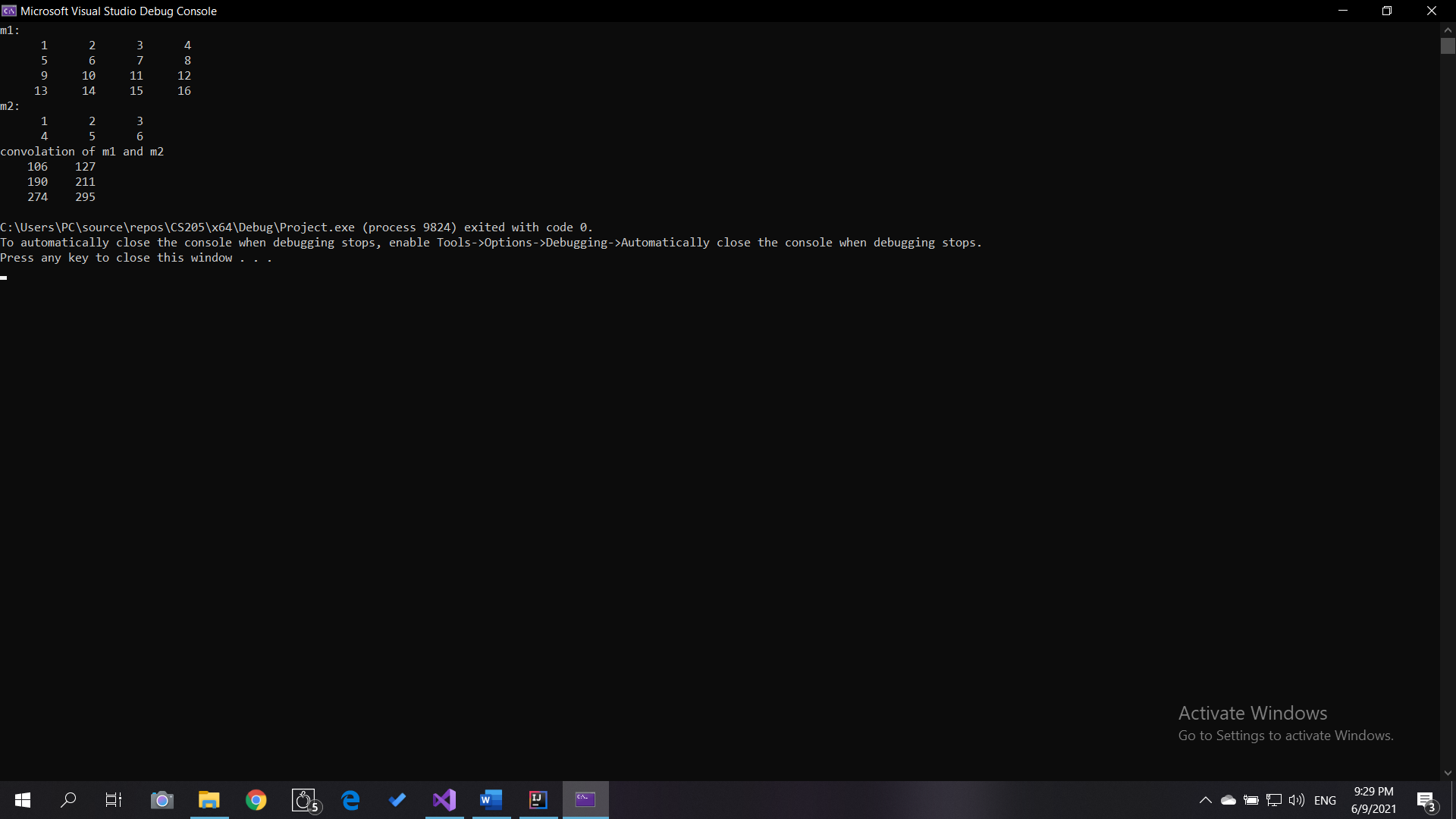
Result of Average(0) with complex entries:



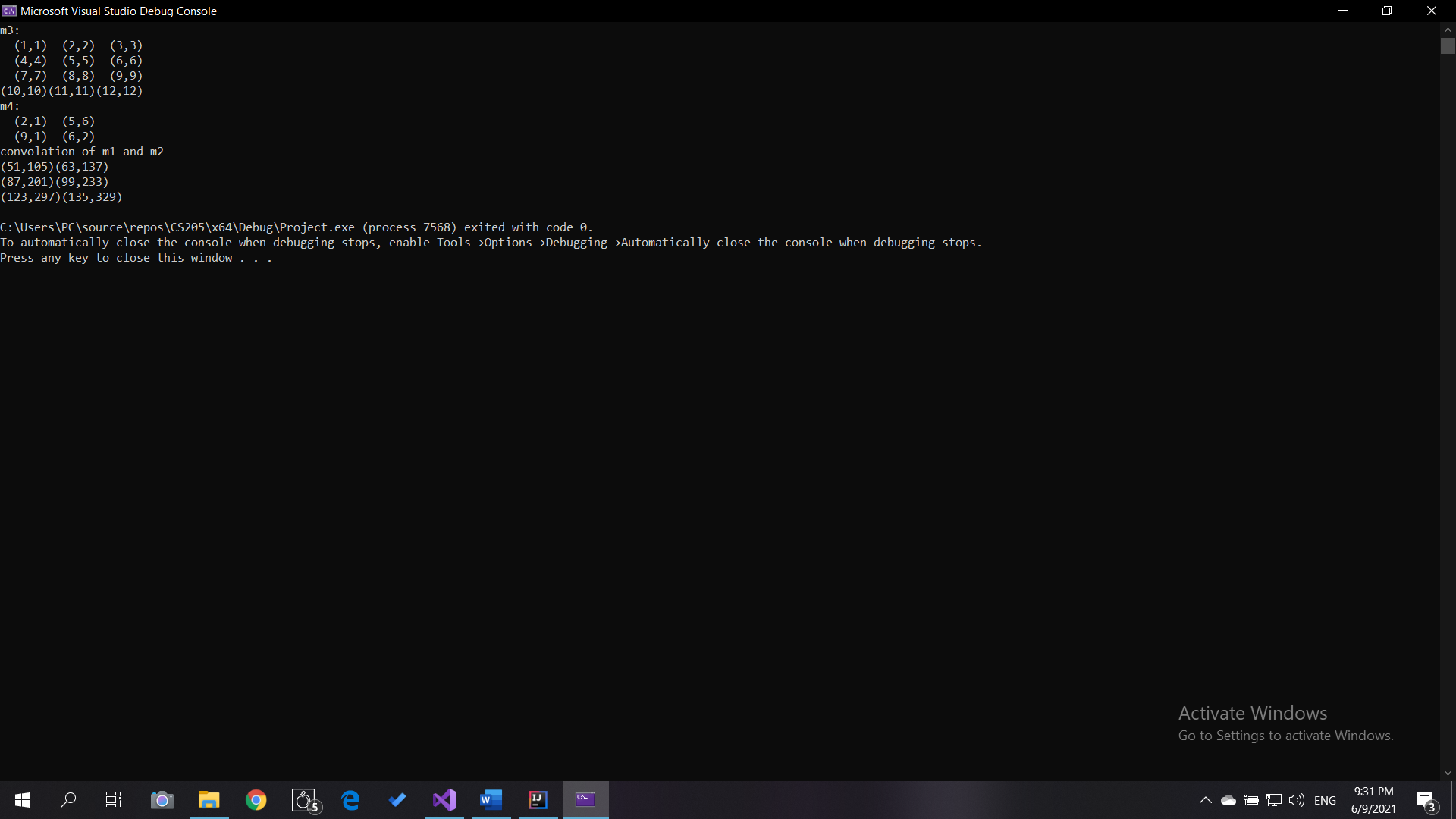
Result of Average(1) with complex entries:



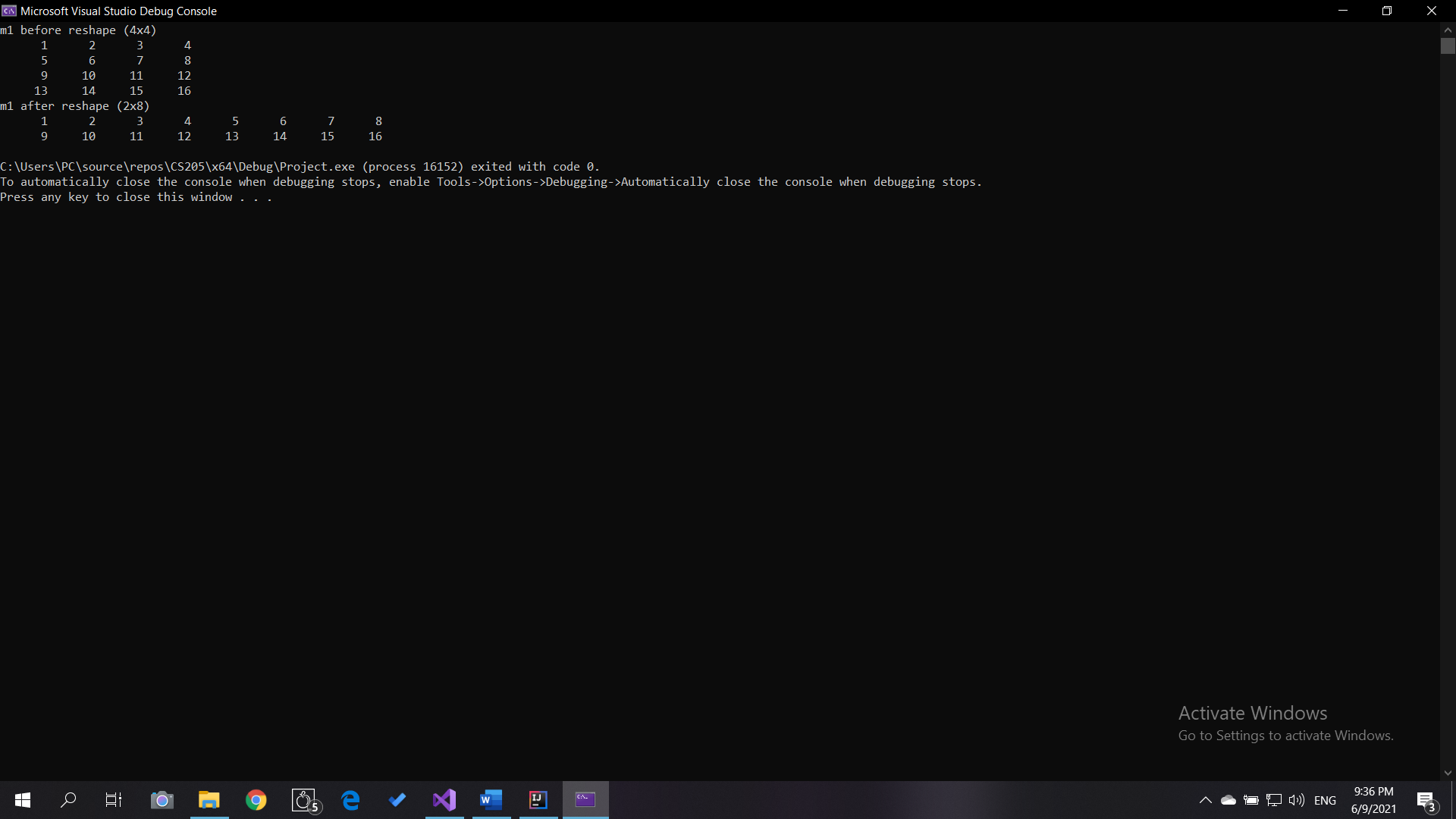
Convolution of m1 and m2:



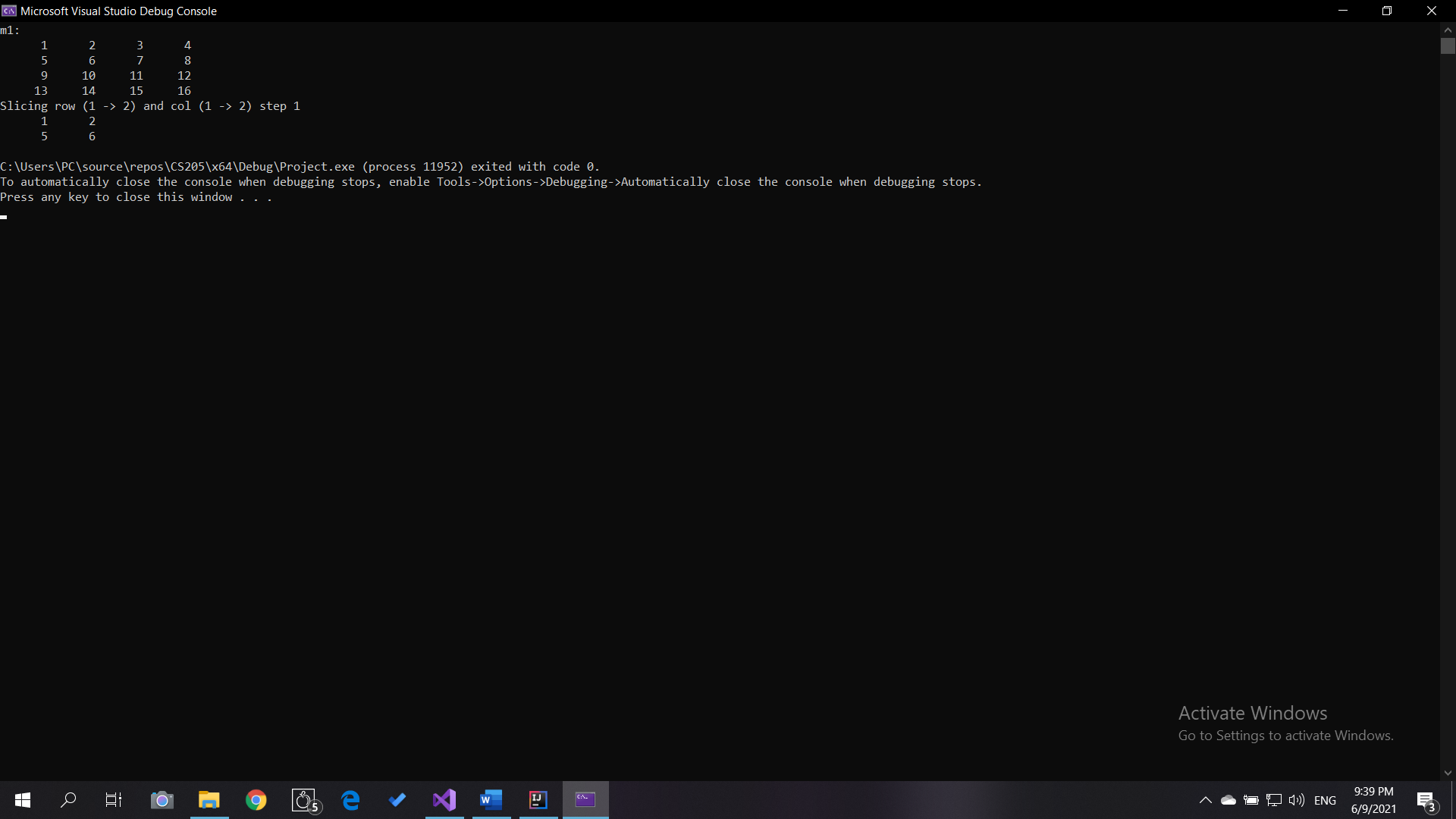
Convolution of m3 and m4 (complex entries)



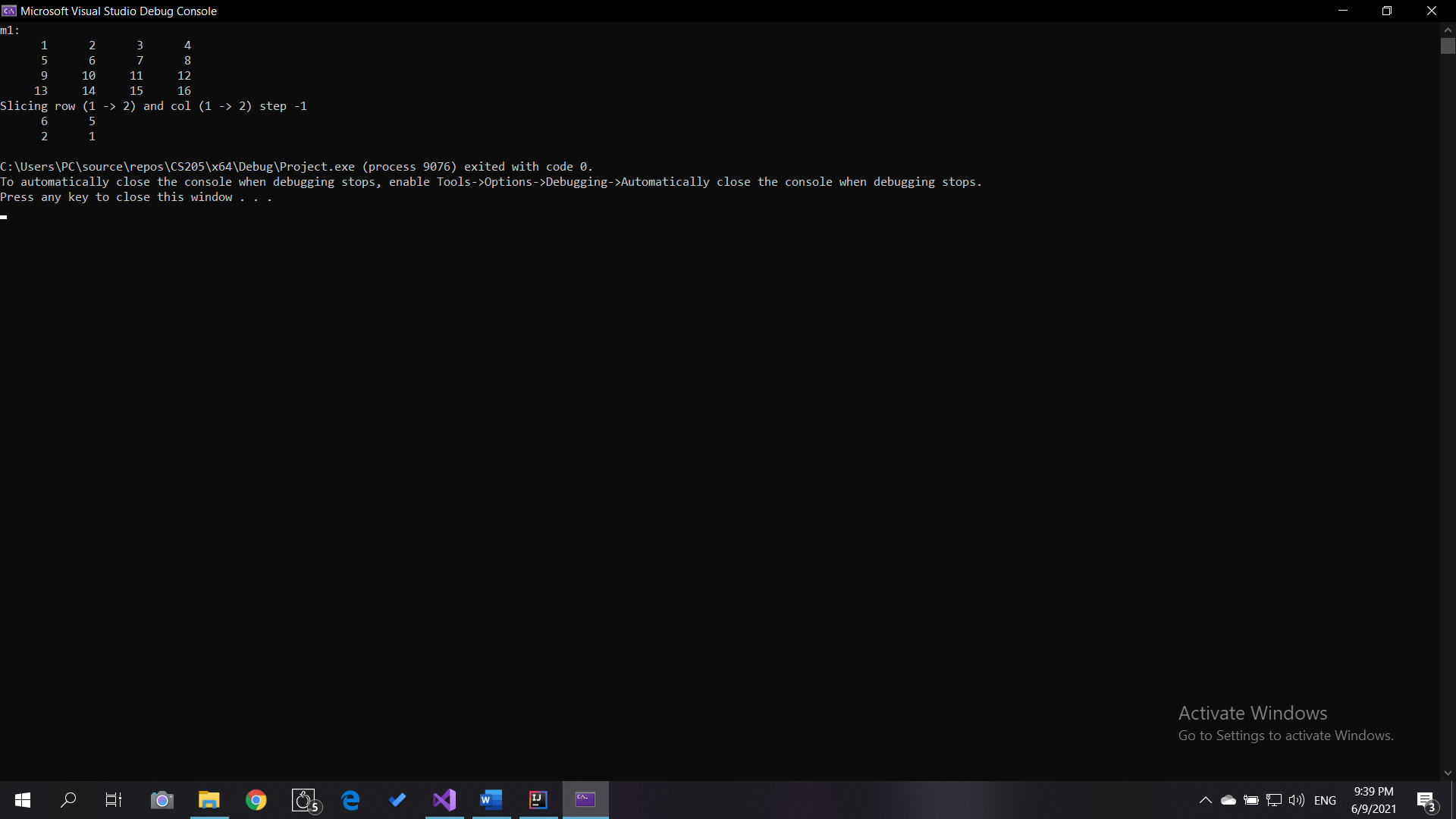
Result of Reshape:

Result of Slicing:

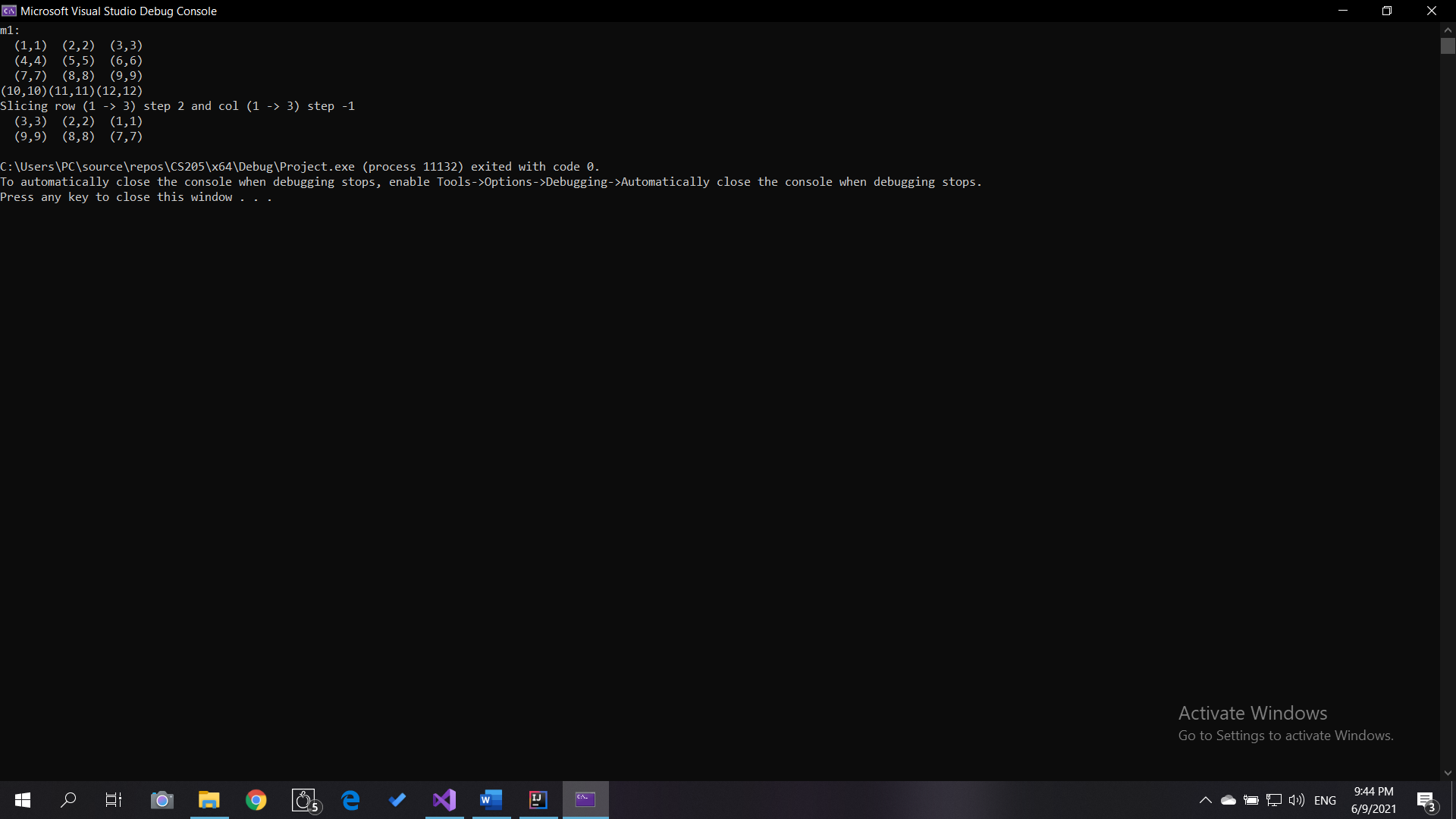
Slice(0,1,-1,0,1-1)



Slice(0,1,-1,0,1-1), both the step for row and column is negative so the slice are arrange backward.



Slice(0,2,2,0,2,-1) with complex entries



**Problem and Solution:**

There was an issue with Average, max and min function when the type name is not primitive type. So, we have to have separate definition of these function for reference type. We will check the type template T to determine which version we should use.