Problem 1.1.1. Write code for multiplying a pair of $n \times n$ matrices ($C \leftarrow AB$):

```
for $i = 1, 2, \ldots, n$ do
    for $j = 1, 2, \ldots, n$ do
        $c_{ij} \leftarrow 0$
        for $k = 1, 2, \ldots, n$ do
            $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
        end for
    end for
end for
```

Do this in your favorite interpreted language (MATLAB, Python, Ruby, $\cdots$). Use your code for multiplying a pair of 100×100 matrices. If your language has a built-in operation for performing matrix multiplication, use this to compute the matrix product of your test matrices. How much faster is the built-in operation?

Solution.

Python will be the interpreted language of choice. Moreover, I wanted to generalize the function to allow for non-square matrices. The working code is available in *1.1.1 Code.py*; hoIver, the code will also be listed below.

```python
def matrix_multiply(A: List[List[float]], B: List[List[float]]) -> List[List[float]]:
"""
Performs matrix multiplication using base Python. I also generalize it a bit
further than what Problem 1.1.1 asks of us. I allow for rectangular matrices
(not just square). This function will assume each input is not jagged.

Parameters
----------
A: List[List[float]]
An n x m matrix

B: List[List[float]]
An m x p matrix

Returns
-------
A: List[List[float]]
An n x p matrix

Raises
------
ValueError
If the dimensions are in valid. That is, when the width of matrix A does
```

```
not match the length of B
"""

WIDTH_A = len(A[0])
LENGTH_B = len(B)

# Verify dimension are correct
if WIDTH_A != LENGTH_B:
  raise ValueError("Invalid dimensions.")

# Start multiplication
n = len(A) # length of output matrix
p = len(B[0]) # width of output matrix


C = [[0 for _ in range(p)] for _ in range(n)]
for i in range(n):
  for j in range(p):
    for k in range(min(n, p)):
      C[i][j] += A[i][k]*B[k][j]
return C

```

To verify the implementation was correct, I inputted the following matrices for A and B and computed $C = AB$:

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, C = \begin{pmatrix} 7 & 10 \\ 15 & 22 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \\ 5 & 6 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, C = \begin{pmatrix} 7 & 10 \\ 15 & 22 \\ 23 & 34 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 & 5 \\ 3 & 4 & 6 \end{pmatrix}, C = \begin{pmatrix} 7 & 10 & 17 \\ 15 & 22 & 39 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}, C = \begin{pmatrix} 1 & 2 \\ 3 & 4 \end{pmatrix}$$

$$A = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 5 & 6 & 7 & 8 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix},\ B = \begin{pmatrix} 16 & 15 & 14 & 13 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{pmatrix},\ C = \begin{pmatrix} 80 & 70 & 60 & 50 \\ 240 & 214 & 188 & 162 \\ 400 & 358 & 316 & 274 \\ 560 & 502 & 444 & 386 \end{pmatrix}$$

$$A = \begin{pmatrix} 0 & -1 & 0 & -0.5 \\ 12 & 11 & 10 & 9 \\ 8 & 7 & 6 & 5 \\ 4 & 3 & 2 & 1 \end{pmatrix},\ B = \begin{pmatrix} 1 & 2 & 3 & 4 \\ 0 & -1 & -0.5 & 0 \\ 9 & 10 & 11 & 12 \\ 13 & 14 & 15 & 16 \end{pmatrix},\ C = \begin{pmatrix} -6.5 & -6.0 & -7.0 & -8.0 \\ 219 & 239 & 275.5 & 312 \\ 127 & 139 & 161.5 & 184 \\ 35 & 39 & 47.5 & 56 \end{pmatrix}$$

For the "built-in" matrix multiplication, I will be using the matrix multiplication feature in NumPy. To test to see if there is a speed up, two 100×100 matrices were generated at random where each entry is an integer from 0 to 100; The matrices I used are in the *1.1.1* Folder in *1.1 Code and Data.* For the function I implemented, the operation was performed 100 times and the execution times for each trial were recorded. This was repeated for the multiplication built into NumPy. The execution times and the code used to calculate the execution times can be found in the *1.1.1* Folder.

I found that my implementation had a mean execution time of 0.537128028869629 seconds while using NumPy had a mean execution time of $9.753704071044921 \times 10^{-6}$ seconds. That is a 99.99818410070843% speedup compared to our implementation. ■

Problem 1.1.2. The matrix operation $C \leftarrow C + AB$ on $n \times n$ matrices can be written as three nested loops: The order of the loops can be changed; in fact, any order of "`for i`", "`for j`", and "`for k`" can be used

```
for $i = 1, 2, \ldots, n$ do
    for $j = 1, 2, \ldots, n$ do
        for $k = 1, 2, \ldots, n$ do
            $c_{ij} \leftarrow c_{ij} + a_{ik}b_{kj}$
        end for
    end for
end for
```

giving a total of $3! = 6$ possible orderings. In a *compiled language* (such as Fortran, C/C++, Java, or Julia), time the six possible orderings. Which one is faster. Can you explain why?

Problem 1.1.3. A rule of thumb for high-performance computing is that when a data item is read in, UC should use it as much as possible, rather than re-reading that data item many times and doing a little computing on it each time. In matrix multiplication, this can be achieved by subdividing each matrix into $b \times b$ blocks. For $n \times n$ matrices A and B, UC can let $m = n/b$ and write:

$$AB = \begin{pmatrix} A_{11} & A_{12} & \cdots & A_{1m} \\ A_{21} & A_{22} & \cdots & A_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ A_{m1} & A_{m2} & \cdots & A_{mm} \end{pmatrix} \begin{pmatrix} B_{11} & B_{12} & \cdots & B_{1m} \\ B_{21} & B_{22} & \cdots & B_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ B_{m1} & B_{m2} & \cdots & B_{mm} \end{pmatrix} = \begin{pmatrix} C_{11} & C_{12} & \cdots & C_{1m} \\ C_{21} & C_{22} & \cdots & C_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ C_{m1} & C_{m2} & \cdots & C_{mm} \end{pmatrix} = C$$

Then for $i, j = 1, 2, \ldots, n$, $C_{ij} \leftarrow \sum_{k=1}^{m} A_{ik} B_{kj}$. As long as UC can keep all of A_{ik}, B_{kj}, and C_{ij} in cache memory at once, the update $C_{ij} \leftarrow C_{ij} + A_{ik} B_{kj}$ can be done without any memory transfers once A_{ik} and B_{kj} have been loaded into memory. Write out a blocked version of the matrix multiplication algorithm and count the number of memory transfers with the blocked and original matrix multiplication algorithms.

Problem 1.1.4. Implement the original and unrolled inner product algorithms in Algorithm 1.1.3 as a function in your favorite programming language. Time the function for taking the product of two vectors of 10^6 entries. Use the simpler version to check the correctness of the unrolled version. Note that there may be differences due to roundoff error. Put the function call inside a loop to perform the inner product 10^3 times. Is there any difference in the times of the two algorithms? Note that interpreted languages, such as Python and MATLAB, may see little or no difference in timings. This is probably due to the fact that the time savings for the unrolled version is negligible compared to the overhead of interpretation. Also, interpreted languages will typically "box" and "unbox" the raw floating point value, converting it to and from a data structure that contains information about the type of the object as UCll as the object itself.

Problem 1.1.5. The following pseudo-code is designed to provoke a "stack overflow" error:

```
function OVERFLOW(n)
    if $n = 2^k$ for some $k$ then
        print($n$)
    end if
    OVERFLOW($n + 1$)
end function
```

Implement in your favorite programming language. How does it behave on your computer? How large a value of n is printed out before overflow occurs?

Problem 1.1.6. Dynamic memory allocation is memory allocation that occurs at run-time. It is an essential in all modern computational systems. Pick a programming language. How does this language allocate memory or objects? Do you need to explicitly de-allocate memory or objects in your programming language? How is this done?

Problem 1.1.7. There are different ways of automatically de-allocating unusable objects (*garbage collection*) in programming systems. In this exercise, UC look at two of them. Describe reference counted garbage collection and “mark and sUCep” garbage collection. What are their strengths and UCaknesses?

Problem 1.1.8. *Memory leaks* are a kind of bug that can be hard to find and remove. These arise when memory is allocated for objects that are never de-allocated, and so eventually take up all available memory. Even if a system has garbage collection, this can still occur. Explain how this might happen.

Problem 1.1.9. Memory allocation and de-allocation can lead to fragmentation of the memory allocation system over time, so that there may be a great deal of memory available, but no large object can be allocated because the available memory is fragmented into small pieces. Read about and describe the SmallTalk double indirection scheme that allows SmallTalk to de-fragment the memory allocation system.