**Problem #1**

Subject: Printing a large integer with thousands separators (commas).

Modules: none

Filename: discrete_math.py

Humans have difficulty correctly interpreting long strings of uninterrupted numbers, which is why we use separators to group digits within a number. In English-speaking countries, the usual convention is to use a comma to separate the whole-number portion of a number into groups of three (and, hence, known as the 'thousands separator'. For example, the value 12345 is written as 12,345.
Since we will be working with relatively large numbers (a 64-bit number contains 19 or 20 digits), we want to be able to print them with thousands separators such that
9223372036854775808 = 9,223,372,036,854,775,808
Write a function, named pretty_int(n), that takes a single argument, assumed to be a non-negative integer, and returns a string representing that number and that includes the comma separators.
Place this function in the discrete_math module and include test code that prints out the values, 0, 1, 999, 1000, 2^16, and 2^64 using your pretty_int() function.
Your test code should ONLY execute if the module is run directly; it should NOT execute if the module is imported by another script.
HINT: This is NOT a hard problem – it can be done in five lines of code, but it requires a bit of planning. There are also multiple ways to go about it, some decidedly more elegant than others, but any way that works is acceptable. Also, don't let yourself get bogged down with this. If nothing else, start off with a function that merely returns the number converted to a string, move on, and accept the hit.

**WORK**

The pretty_int function in the module converts a non-negative integer into a string representation with thousands separators. It takes an integer `n` as input and returns a string with commas inserted for thousands. The function is implemented using Python's built-in string formatting method. In the main code block, it is tested whether the pretty_int function is implemented. If it is not implemented, a message is printed indicating that the function is not available. Otherwise, a list of integers is iterated over, and for each integer in the list, its string representation with thousands separators is printed.

**OUTPUT**

```
discrete_math.py : pretty_int() test

   0 = 0
   1 = 1
   999 = 999
   1000 = 1,000
   65536 = 65,536
   18446744073709551616 = 18,446,744,073,709,551,616
```

**CODE**

```
'''
PROGRAMMER: Jakob K. West
USERNAME: jwest21
PROGRAM: discrete_math.py

DESCRIPTION: Function to print large integers separated by commas
'''

    """""""""""""""""""""""
    PRETTY INT FUNCTION
    """""""""""""""""""""""

    print("discrete_math.py : pretty_int() test")
    print()

    def pretty_int(n):

        """
        Convert a non-negative integer to a string with thousands separators.

        Parameters:
        n (int): A non-negative integer.

        Returns:
        str: A string representing the number with comma separators for thousands.
        """
        return '{:,}'.format(n)

    try:
        pretty_int
    except NameError:
        print("    Function pretty_int() not implemented")
        print()
    else:
        n_list = [0, 1, 999, 1000, 2**16, 2**64]
        for n in n_list:
            print("  ", n, "=", pretty_int(n))
    print()
```

**Problem #2**

Subject: Finding a prime factor of a number

Modules: time
Filename: discrete_math.py

Add another function to your discrete_math module named prime_factor(n) that takes a single argument, n, that is assumed to be an integer and returns a 2-tuple in which the first member, p, is a prime factor of n and the second is n/p. The first number can be ANY prime factor of p, it does NOT have to be the smallest prime factor. You do not have to determine whether or not the second member is prime.

If the number is prime, then the return value should be (n, 1). If the function fails to find a prime factor, for any reason, it should return (1, n), which can be used to detect the failure since 1 is not a prime number. Reasons for returning this "error value" include the number passed not being integer-valued, the number not having any prime factors (which is the case for any integer smaller than 2, since 2 is the smallest prime number), or the function having insufficient time to find a prime factor.

How you choose to implement this function is up to you, but it will have a major impact on the time it takes you to factor the RSA numbers in the next part of the assignment. However, the RSA numbers you will be given have been chosen so that you do not need to be extremely efficient at factoring, but you also can't be extremely inefficient at it.

To support a time-expired feature, create a global variable (at the discrete_math module level) that the prime_factor() function checks regularly and, if that variable is set to a particular value (your choice), the function exits immediately, returning the error value.

Add test code that factors the following numbers, some of which are prime, some of which aren't, and some of which have no prime factors.

[ 0, 1, 2, 3, 12, 97 ]
Also factor the following RSA numbers.

5,782,475,771 4,698,643,325,249 253,049,136,761,293 18,241,119,882,520,216,117
37,530,294,278,910,762,919

The test code should also print out how long each takes and print the results in seconds with millisecond resolution. This test code should execute ONLY when the script is run directly, and NOT when the script is loaded as a module by another script.

HINT: While your script should function properly when run within whatever IDE you are using, you may find that the IDE significantly slows its performance – a factor of two is not uncommon. To test this out, you might try running your script directly from the command line as follows:

**C:\path_with_script>python discrete_math.py**

If this doesn’t work, then you probably need to add python to your system’s path.

**WORK**

Within the "PRETTY FACTOR FUNCTION" section of the **discrete_math.py** file, the **prime_factor()** function is defined to find a prime factor of a given integer **n**. It checks if **n** is less than 2 or not an integer, returning **(1, n)** if so. The function iterates through potential divisors, considering time constraints with a global variable **time_expired**. If a divisor is found, it returns the divisor and **n** divided by it; otherwise, it returns **(n, 1)**. Following the function definition, a test block checks the functionality of **prime_factor()** using various inputs, timing its execution, and printing the results, including prime factorization and computation time.

**OUTPUT**

```
discrete_math.py : prime_factor() test

    0 = (1)*(0) ( 0.000 seconds)
    1 = (1)*(1) ( 0.000 seconds)
    2 = (2)*(1) ( 0.000 seconds)
    3 = (3)*(1) ( 0.000 seconds)
    12 = (2)*(6) ( 0.000 seconds)
    97 = (97)*(1) ( 0.000 seconds)
    5,782,475,771 = (69,151)*(83,621) ( 0.005 seconds)
    4,698,643,325,249 = (1,264,447)*(3,715,967) ( 0.077 seconds)
    253,049,136,761,293 = (12,957,929)*(19,528,517) ( 0.823 seconds)
    18,241,119,882,520,216,117 = (320,019,647)*(57,000,000,011) ( 19.604 seconds)
    37,530,294,278,910,762,919 = (612,671)*(61,256,847,931,289) ( 0.039 seconds)
```

**CODE**

```
import time

    """""""""""""""""""""""""
    PRETTY FACTOR FUNCTION
    """""""""""""""""""""""""

    print("discrete_math.py : prime_factor() test")
    print()

    def prime_factor(n):

        """
        Find a prime factor of a number.

        Parameters:
        n (int): An integer to factor.

        Returns:
        tuple: A tuple containing two integers. The first integer is a prime factor
        of n, and the second integer is n divided by the prime factor. If the number
        is prime, then the return value is (n, 1). If the function fails to find a
        prime factor, it returns (1, n).
```

```
        """

        global time_expired

        if n < 2 or not isinstance(n, int):
            return (1, n)

        divisor = 2
        while divisor ** 2 <= n:
            if time_expired:
                return (1, n)
            if n % divisor == 0:
                return (divisor, n // divisor)
            divisor += 1

        return (n, 1)

    try:
        prime_factor
    except NameError:
        print("    Function pretty_factor() not implemented")
        print()
    else:

        for n in [0, 1, 2, 3, 12, 97]:
            start_time = time.time()
            result = prime_factor(n)
            end_time = time.time()
            print("    %s = (%s)*(%s) ( %.3f seconds)" \
                  % (pretty_int(n), pretty_int(result[0]),
                     pretty_int(result[1]), end_time - start_time))
        for n in [69_151*83_621, 1264447*3715967, 12957929*19528517,
320019647*57000000011, 61256847931289*612671]:
            start_time = time.time()
            result = prime_factor(n)
            end_time = time.time()
            print("    %s = (%s)*(%s) ( %.3f seconds)" \
                  % (pretty_int(n), pretty_int(result[0]),
                     pretty_int(result[1]), end_time - start_time))
```

**Problem #3**

Subject: Finding as many RSA prime factors given a .txt file
Modules: time, discrete_math.py, threading
Filename: discrete_math.py, hw04_03.py

This is where everything comes together. In this problem, the task is to factor as many of the RSA numbers contained in the file “rsa_numbers.txt” as possible in no more than 1200 seconds (20 minutes) by using the functions you have already written to write a new function, factor_list(), that exploits multithreading.
A driver script, hw04_03.py, is provided for you:

```
import discrete_math as dm
rsa_list = []
with open("rsa_numbers.txt", "rt") as fp:
    strings = fp.readlines()
for s in strings: rsa_list.append(int(s))
dm.factor_list(rsa_list, 1200)
```

The file contains twenty 64-bit numbers. At least one of these numbers has a relatively small prime factor, while the smallest prime factors of the others is approximately an order of magnitude larger. The “easy” number(s) is/are randomly placed within the file.
Your approach should be similar to that demonstrated in class, in which you have a master function in the main thread, factor_list(), that launches a new thread using another function, such as factor_thread(), as the target. Since threads cannot return values, their results need to be written into global variables that are then accessed and printed out by the master function.
NOTE:
Depending on your IDE, you may discover that, when time expires and your main thread ends, that the unfinished child threads may be unable to finish (if they are blocked) and your IDE may become unresponsive. Therefore, your program should have a way to end properly. This is the purpose of a global variable in the discrete_math module that signals the threads that they should end as quickly as possible, while the main thread should wait until all of the child threads have finished before ending itself. A threading function that might be of particular use for this purpose is active_count(). You should also consider creating a Lock object to synchronize your threads.

**WORK**

The function **factor_list** aims to factorize a list of numbers utilizing threading for parallel processing. Initially, the program prints a test indication for **factor_list()**. Subsequently, within the function **factor_list**, it initializes essential variables like **result**, **lock**, and **threads**. Iterating over each number in the input list, the function spawns threads for factorization, each assigned to the **factor_thread** target function. These threads are then started, and their references are appended to the **threads** list. The function then waits until either the specified **time_limit** is reached or all threads finish their tasks. During this waiting period, it ensures that the main thread sleeps for a short duration to avoid excessive resource consumption. Upon completion or timeout, the function prints informative messages regarding the factorization process. It displays details such as the length of the input list, the time limit, and for each number, its prime factors along with the time taken for factorization. If the time limit is exceeded, a message indicating "TIME EXPIRED" is printed. Additionally, it counts the successfully factored numbers and prints the count. Lastly, it prints the number of child threads terminated, along with any aborted factorization attempts and concludes with a message indicating the

completion of the program execution. The function concludes by testing its implementation against a predefined list of RSA numbers and a specified time limit.

**OUTPUT**

*** Please note that I had to comment out the try/except blocks in the discrete_math.py program, so that my computer could use all its computational power towards problem #3 ***

```
discrete_math.py : factor_list() test

Factoring a list of RSA numbers
List length: 10 numbers
Time limit: 8 seconds
263,538,196,264,043 = 6,186,619 * 42,598,097 --- ( 8.034 sec )
263,533,126,545,097 = 6,186,527 * 42,597,911 --- ( 8.034 sec )
263,531,430,756,403 = 6,186,493 * 42,597,871 --- ( 8.034 sec )
38,273,635,989,997 = 6,186,527 * 6,186,611 --- ( 8.034 sec )
263,532,029,957,197 = 6,186,503 * 42,597,899 --- ( 8.034 sec )
1,814,582,021,563,777 = 1 * 1,814,582,021,563,777 --- ( 8.034 sec )
1,814,590,285,559,783 = 1 * 1,814,590,285,559,783 --- ( 8.034 sec )
1,814,588,326,047,229 = 1 * 1,814,588,326,047,229 --- ( 8.034 sec )
1,814,589,007,617,233 = 1 * 1,814,589,007,617,233 --- ( 8.034 sec )
1,814,585,173,809,743 = 1 * 1,814,585,173,809,743 --- ( 8.034 sec )
TIME EXPIRED
Successfully factored 5 numbers.
Terminating 0 child threads.
Aborting: 1,814,582,021,563,777, 1,814,590,285,559,783, 1,814,588,326,047,229,
1,814,589,007,617,233, 1,814,585,173,809,743
Clean up complete, exiting program.

PS /Users/jakobwest/swe/academia/cs3080/hw04_jwest21/code> python3 hw04_03.py
discrete_math.py : pretty_int() test

discrete_math.py : prime_factor() test

discrete_math.py : factor_list() test

Factoring a list of RSA numbers
List length: 20 numbers
Time limit: 1200 seconds
11,805,923,586,610,988,473 = 343,597,721 * 34,359,726,113 --- ( 1200.184 sec )
TIME EXPIRED
Successfully factored 1 numbers.
Terminating 0 child threads.
Aborting: 11,806,022,799,552,001,063, 11,805,923,293,527,495,623,
11,806,025,046,700,345,621, 11,805,980,990,423,500,327, 11,805,974,626,997,379,923,
11,805,978,976,956,693,941, 11,805,989,573,503,495,133, 11,806,021,487,021,274,713,
11,805,978,873,922,178,711, 11,805,999,221,744,820,473, 11,805,948,774,717,201,407,
11,805,960,051,582,831,473, 11,805,997,469,372,260,019, 11,805,995,318,489,244,937,
11,805,965,782,830,781,577, 11,805,953,612,588,925,271, 11,805,993,621,138,293,323,
11,805,948,651,024,069,803, 11,806,060,533,634,512,017
Clean up complete, exiting program.
```

**CODE**

```
import threading


    """"""""""""""""""""""""
    FACTOR THREAD FUNCTION
    """"""""""""""""""""""""

    def factor_thread(num, result, lock):
        """
        Thread target function to factor a single number and store the result.

        Parameters:
        num (int): The number to factor.
        result (list): A list to store the factorization result.
        lock (threading.Lock): A lock to synchronize access to the result list.
        """

        global time_expired

        # Factor the number
        prime_factors = prime_factor(num)

        # Acquire lock to access result list
        lock.acquire()

        # Store the result
        result.append((num, prime_factors))

        # Release lock
        lock.release()


    """""""""""""""""""""""
    FACTOR LIST FUNCTION
    """""""""""""""""""""""

    print("discrete_math.py : factor_list() test")
    print()

    def factor_list(numbers, time_limit):
        """
        Factor a list of numbers using threading.

        Parameters:
        numbers (list): A list of numbers to factor.
        time_limit (int): The time limit in seconds.
        """

        global time_expired

        # Initialize variables
        result = []
```

```
        lock = threading.Lock()
        threads = []

        # Start threads for factorization
        for num in numbers:
            thread = threading.Thread(target=factor_thread, args=(num, result, lock))
            thread.start()
            threads.append(thread)

        # Wait until time limit or all threads are finished
        start_time = time.time()
        while time.time() - start_time < time_limit and threading.active_count() > 1:
            time.sleep(0.1)

        # Set time_expired flag to signal threads to end
        time_expired = True

        # Wait for all threads to finish
        for thread in threads:
            thread.join()

        # Print results
        print("Factoring a list of RSA numbers")
        print("List length:", len(numbers), "numbers")
        print("Time limit:", time_limit, "seconds")

        for num, factors in result:
            print(pretty_int(num), "=", pretty_int(factors[0]), "*",
pretty_int(factors[1]), "--- (", "{:.3f}".format(time.time() - start_time), "sec )")

        # Check if time expired
        if time.time() - start_time >= time_limit:
            print("TIME EXPIRED")

        # Count successfully factored numbers
        successes = sum(1 for _, factors in result if factors[0] != 1)
        print("Successfully factored", successes, "numbers.")
        print("Terminating", threading.active_count() - 1, "child threads.")
        print("Aborting:", ", ".join(str(pretty_int(num)) for num, factors in result
if factors[0] == 1))
        print("Clean up complete, exiting program.")

    try:
        factor_list
    except NameError:
        print("   Function factor_list() not implemented")
        print()
    else:

        rsa_p1 =
[6186493,42598097,6186503,6186527,42598099,42597899,6186611,42597917,6186619,42597871
]
        rsa_p2 =
[42597871,42597889,42597899,42597911,42597917,42597923,6186527,42597979,42598097,4259
8099]
```

```
        rsa_list = []
        for p1, p2 in zip(rsa_p1, rsa_p2):
            rsa_list.append(p1 * p2)
        factor_list(rsa_list, 8)


'''
PROGRAMMER: Jakob K. West
USERNAME: jwest21
PROGRAM: hw04_03.py

DESCRIPTION: Multithreaded RSA Factoring program
'''

import discrete_math as dm

rsa_list = []
with open("rsa_numbers.txt", "rt") as fp:
    strings = fp.readlines()

for s in strings:
    rsa_list.append(int(s))

dm.factor_list(rsa_list, 1200)
```