Discrete Mathematics Fall 2019

Coding Assignment 2: Prime Numbers

Due Date: Thursday November 7th, 2019

Academic Honesty Policy:

You are expected to submit your own work. Assignments are **to be completed individually**, NOT in groups. No collaboration is permitted, unless otherwise specified. Please do not include any of your code snippets or algorithms in public Piazza posts. You cannot not use solutions from any external source. You are not permitted to publish code or solutions online, nor post the course questions on forums including stack overflow. We run plagiarism detection software. If you have any questions at all about this policy please ask the course staff before submitting your assignment.

Read Carefully:

- Feel free to import any standard Python libraries you need as far as output and input of functions meet the requirement. Use the provided template to implement the functions.
- You **must name your file** "coding2.py". Any submission that does not follow this naming will not be graded and **will receive a zero.**
- A skeleton of each function has been provided to you. **You are expected to ONLY write code in the functions and blocks specified.** In any case, DO NOT modify the function signatures, return variables, or any code that is not specified to be modifiable (we will include comments in the code to distinguish these sections) for any reason. Also, though you may modify the `main` function to test other cases, **do not turn in an assignment with a modified main function, i.e. anything below the line shown below.** This will be run by our autograder, so any unexpected modifications that make it malfunction **will receive a zero.**

```
### DO NOT TURN IN AN ASSIGNMENT WITH ANYTHING BELOW HERE MODIFIED ###
if __name__ == "__main__":
```

- Test cases will be provided in the `main` function in the code. Three of the test cases for each part will be provided to you, and the others will be hidden test cases on our side (all graded via autograder).
- Only Python 3.x versions will be graded. To check your Python version, open a terminal window and enter:

```
python --version
```

- Although you will not be graded on your coding style, it is good to familiarize yourself with PEP8 style guide for Python. Learn more about PEP8 here:
`https://www.python.org/dev/peps/pep-0008/`
- To receive points, make sure your code runs. We recommend using Spyder, Pycharm or Google colab. They all allow you to download .py files. Be aware that if you write your code in some platforms like Codio and copy and paste it in a text file, there may be spurious characters in your code, and your code will not compile. **Always ensure that your .py compiles. Code that does not run will not receive any points.**

Part A: Euclid's Algorithm

Euclid's Algorithm was invented by the Greek mathematician Euclid about 2,300 years ago. The algorithm calculates the greatest common divisor/factor (GCD) of two integers. (Please refer to your lecture for more details about how and why it works).

Your task will be to implement Euclid's Algorithm in Python in order to calculate the GCD of any two numbers.

(a) Write a Python function `euclid` that takes two numbers as input and outputs their GCD. You may assume that inputs are valid (the inputs are two integers).

Your Python function `euclid` should take two arguments `num1` and `num2`, which are type `int`. It should return an `int` that is the GCD of the two numbers.

```
def euclid(num1, num2):
    # WRITE YOUR CODE HERE
    return #your GCD
```

For example, these lines print the values for $GCD(355, 55)$ and $GCD(182, 19833)$.

```
print(euclid(355,55)) # should print 5
print(euclid(182, 19833)) # should print 1
```

Your algorithm should also print out (but not return) the intermediate steps. By intermediate steps, we mean each equivalent calculation of the GCD of the two numbers. The Euclidean algorihtm reduces your initial GCD of two numbers into subsequent equivalent GCDs, if implemented correctly. For instance, for 270 and 192:

```
GCD(270,192) = GCD(192,78) = GCD(78,36) = GCD(36,6) = GCD(6,0) = 6.
```

Your goal is to print out

```
GCD(270,192) = GCD(192,78) = GCD(78,36) = GCD(36,6) = GCD(6,0) = 6
```

precisely as shown above (with `=` separating each printed statement), on a **single line.**

(b) Run your algorithm on the specified inputs in the main function to make sure your algorithm works. You may modify the main function to try other numbers but make sure to change it back, and **do not turn in a modified main function.**

Part B: Prime Number Generation

Prime number generation is a crucial to many subfields of math and computer science, notably cryptography and security. Here, you will implement another ancient algorithm known as the Sieve of Eratosthenes to generate prime numbers.

(a) Write a Python function `prime_gen` that implements the Sieve of Eratosthenes. Your function will take in an integer $n > 1$ and will output all prime numbers up to (and including) n.

Your function `prime_gen` will take in a single `int` as its argument and output a `list` of prime numbers (each also of type `int`). **Make sure your output is a list, or your code will get zero points.**

```
def prime_gen(n):
    # WRITE YOUR CODE HERE
    return # your list of prime numbers
```

For example, the following lines print the primes up to 5 and 10:

```
print(prime_gen(5))
print(prime_gen(10))
```

where the output is:

```
[2, 3, 5]
[2, 3, 5, 7]
```

(b) Run your algorithm on the specified inputs in the main function to make sure your algorithm works. You may modify the main function to try other numbers but make sure to change it back, and **do not turn in a modified main function.**

Part C: Primality Testing

Another useful algorithm to have for prime numbers is a primality checker. Primality checking is, again, important to fields such as cryptography and computer security. Here, you will implement three different primality tests.

(a) Trial Division. Write a function `prime_check_trial(n)` that checks if a given input integer n is a prime number using trial division. Trial division simply checks, for a given input integer n, whether there is a number between 2 and $\sqrt{n}$ that divides n. The function should take an `int` n and return a `bool`: `True` if n is a prime, or `False` if n is not a prime.

```
def prime_check_trial(n):
    # WRITE YOUR CODE HERE
    return # bool value (True or False) depending on prime or not
```

(b) Sieve of Eratosthenes. Write a function `prime_check_sieve(n)` that checks if a given input integer n is a prime number using the Sieve of Eratosthenes. The function should take an `int` n and return a `bool`: `True` if n is a prime, or `False` if n is not a prime.

```
def prime_check_sieve(n):
    # WRITE YOUR CODE HERE
    return # bool value (True or False) depending on prime or not
```

(c) Fermat's Little Theorem. Write a function `prime_check_fermat(n)` that checks if a given input integer n is a prime number using the Fermat's Little Theorem. Fermat's Little Theorem states:

If p is a prime number, then for any integer a, the number $a^p - a$ is an integer multiple of p.

The function should take an `int` n and return a `bool`: `True` if n is a prime, or `False` if n is not a prime.

```
def prime_check_fermat(n):
    # WRITE YOUR CODE HERE
    return # bool value (True or False) depending on prime or not
```

(d) Run your algorithms on the specified inputs in the main function to make sure your algorithm works. You may modify the main function to try other numbers but make sure to change it back, and **do not turn in a modified main function.**

Part D: Checking the Goldbach's conjecture (1742)

As seen in class, the Goldbach conjecture is still not proven or disproven and can be stated as follows:

Every even integer greater than two can be written as the sum of two prime numbers.

(a) Write a function `check_goldbach(n)` that verifies the conjecture for a given (potentially large) even integer n sent as a parameter to the function. The function should return the two primes that sum up to n, in the form of a `list` with two entries of type `int`. **As before, make sure your return types are correct, or your code will receive zero points.** You may assume that your input will be a positive even integer. Also note that Goldbach decompositions are not unique, so there may be more than one correct answer for a given integer n (any satisfying sum will be correct). Hint: use your function `prime_gen(n)`.

```
def check_goldbach(n):
    # WRITE YOUR CODE HERE
    return # two prime numbers that sum up to n
```

(b) Run your algorithm on the specified inputs in the main function to make sure your algorithm works. You may modify the main function to try other numbers but make sure to change it back, and **do not turn in a modified main function.**