CS 3430: S22 Scientific Computing
Assignment 09
Solving Linear Congruences of First Degree
and
Chinese Remainder Theorem

Vladimir Kulyukin
Department of Computer Science
Utah State University

March 26, 2022

Learning Objectives

1. Extended Euclid Algorithm;
2. Equivalence Classes Modulo n;
3. Multiplicative Inverses Modulo n;
4. Solving Congruences of First Degree;
5. Chinese Remainder Theorem (CRT).

Introduction

In this assignment, we'll solve congruence equations of 1st degree. We'll approach this problem in five steps. First, we'll implement the Extended Euclid algorithm. Second, we'll learn how to represent equivalence classes with Python generators. Third, we'll implement an algorithm to find multiplicative inverses modulo n. Finally, we'll implement an algorithm to solve linear congruence equations (aka congruences of first degree). Fifth, we'll implement the algorithm of the Chinese Remainder Theorem (CRT).

You'll save your solutions in `cs3430_s22_hw09.py`. You can use the unit tests in `cs3430_s22_hw09_uts.py` to stress test your code.

Problem 0: (0 points)

Review your notes or the slides for Lectures 17, 18, 19 in Canvas and become comfortable with the Division Theorem, Euclid's two algorithms for the greatest common divisor (GCD), equivalence classes modulo n, multiplicative inverses modulo n, the algorithm to solve linear congruence equations that we discussed in Lecture 18, and the Chinese Remainder Theorem we talked about in Lecture 19.

Problem 1: (1 point)

Implement the Extended Euclid algorithm whose pseudocode is given on Slide 24 of Lecture 17 and save your solution in `xeuc(a,b)` in `cs3430_s22_hw09.py`. Test your implementation with `test_xeuc()` in `cs3430_s22_hw09_uts.py`. This test uses random numbers to stress test your implementation. You output should look as follows.

```
...
1 = 29356*-32808 + 70937*13577
gcd(29356,70937) = 1 = 29356-32808 + 7093713577
4 = 64588*8074 + 75348*-6921
gcd(64588,75348) = 4 = 645888074 + 75348-6921
68 = 69700*138 + 79492*-121
gcd(69700,79492) = 68 = 69700138 + 79492-121
5 = 51725*653 + 7270*-4646
gcd(51725,7270) = 5 = 51725653 + 7270-4646
4 = 13084*94 + 3628*-339
gcd(13084,3628) = 4 = 1308494 + 3628-339
1 = 87739*20472 + 93703*-19169
gcd(87739,93703) = 1 = 8773920472 + 93703-19169
...
```

Feel free to adjust the lower and upper bounds of the range from which random integers are chosen or the number of tests.

Problem 2: (1 point)

In Lecture 17, solved the congruence $3x \equiv 1 \pmod{10}$ (Slides 34 and 35 of Lecture 17) and discovered that the adjective *unique* applies to individual integers but to equivalence classes modulo 10 (i.e., $[x]_{10}$). Each equivalence class modulo n has infinitely many integers. Therefore, we have to confront centuries-old problem of mathematics, science, and philosophy: how do we represent the infinite through the finite?

One way to do it in Python is to use generators. I discovered Python generators when I started doing GB/TB-size data (mostly audio and video) analysis and processing projects back in 2014. I thought that generators are possible only in Lisp (and other functional programming languages) until a colleague of mine (and a fellow Lisper and a life-long aficionado of functional programming) told me that Python had generators, too. His remark immediately caught my attention, because I'm always on the lookout for NoSQL solutions to various data analysis/management problems. In general, the less SQL, the happier I get. It's a matter of taste, of course! Anyway, after playing with Python generators for a couple of days I was hooked. Besides, generators are not just useful; in my opinion, they are one of the coolest features of functional programming and yield excellent insights into programmatic aspects of infinity.

Let me say a few words about them for the sake of those who've never used generators before. Generators can be thought of as lazy iterators, except one doesn't have to worry about the Python iterator protocol. Python iterables are containers that contain all items (e.g., a dictionary or a list) that can be iterated through. Generators, on the other hand, never contain all items when their creation. Rather, they generate items on demand, which makes them an excellent choice for representing infinite sets of objects.

How do we create a generator? In Python, anytime you see the keyword `yield` instead of `return` in a function, you're dealing with a generator. Here's a simple generator that generates the first five primes (i.e., 2, 3, 5, 7, 11).

```
def gen_primes():
    yield 2
    yield 3
    yield 5
    yield 7
    yield 11
```

I put the above generation in `generators.py`. Once we've defined a generator, we can create it and run the Python `next` function to extract objects from the generator one by one. If a generator is finite (i.e., can generate only finitely many objects), a `StopIteration` exception is thrown after the last object is produced. Here's an example of using our prime generator.

```
>>> from generators import *
>>> pg = gen_primes()
>>> next(pg)
2
>>> next(pg)
3
>>> next(pg)
5
>>> next(pg)
7
>>> next(pg)
11
>>> next(pg)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
StopIteration
```

Nothing terrible about that exception. If we need to generate the first 5 primes again, all we need to do is to call `gen_primes()` to create another generator object and get the primes out of it. In other words, a generator can run only once.

We don't even need to worry about exception handling, because many Python constructs (e.g., the for-loop) handle `StopIteration` for us. Here's an example of using `gen_primes()` in a for-loop. No exception is raised. Actually, it's raised and caught by the loop, at which point the iteration stops.

```
>>> for p in gen_primes():
        print(p)
2
3
5
7
11
```

Here's an infinite generator to generate the set of natural numbers.

```
def gen_naturals():
    n = 0
    while True:
        yield n
        n += 1
```

Here's how we can use it.

```
>>> from generators import *
>>> gn = gen_naturals()
>>> next(gn)
0
>>> next(gn)
1
>>> next(gn)
2
>>> next(gn)
3
>>> next(gn)
4
>>> lst = [next(gn) for _ in range(10)]
>>> lst
[5, 6, 7, 8, 9, 10, 11, 12, 13, 14]
```

I wouldn't run the for-loop below if I were you.

```
for n in gen_naturals():
    print(n)
```

So, let's use the generators to represent equivalence classes modulo n. We'll use the generators (a finite computationa unit) to represent infinite objects (i.e., equivalence classes). Recall that an equivalence class of an integer a modulo n ($n > 1$) is defined as

$$[a]_n = \{a + kn\},$$

where k is an integer. For example, $[3]_7 = \{..., -11, -4, 3, 10, 17, ...\}$. Here's a function that defines a generator for $[a]_n$. It's an infinite generator that starts with $a + 0n$ and then alternates between positive and negative values of k ever increasing in absolute magnitude.

```
def make_equiv_class_mod_n(a, n):
    assert n > 0
    def gen_equiv_class(k):
        kk = k
        while True:
            yield a + kk*n
            if kk == 0:
                kk += 1
            elif kk > 0:
                kk *= -1
            elif kk < 0:
                kk *= -1
                kk += 1
    return gen_equiv_class(0)
```

The above definition is in `cs3430_s22_hw09.py`. Here's how we can use it to generate some elements of $[3]_7$.

```
>>> from cs3430_s22_hw09 import *
>>> ec = make_equiv_class_mod_n(3, 7)
>>> for i in range(10):
        print(next(ec))
```

```
3
10
-4
17
-11
24
-18
31
-25
38
>>> rslt = [next(ec) for _ in range(15)]
>>> rslt
[-32, 45, -39, 52, -46, 59, -53, 66, -60, 73, -67, 80, -74, 87, -81]
```

Implement the function `mult_inv_mod_n(a, n)` in `cs3430_s22_hw09.py` that takes integers a and b and returns the equivalence class of a multiplicative inverse of a modulo n if it exists (i.e., if $ax \equiv 1 \pmod{n}$ has a solution) or `None` when a doesn't have a multiplicative inverse modulo n. In other words, it returns a generator for $[x]_n$ if $ax \equiv 1 \pmod{n}$ has a solution and `None`, otherwise.

Use `test_mult_inv_mod_n_01`, `test_mult_inv_mod_n_02`, and `test_mult_inv_mod_n_03` to test your solution. For example, `test_mult_inv_mod_n_01` tests your implementation on example 1 on Slide 11 in Lecture 18. Below is the output where `<>` stands for $\equiv$.

```
3*-3    <>      1       (mod 10)

3*7     <>      1       (mod 10)

3*-13   <>      1       (mod 10)

3*17    <>      1       (mod 10)

3*-23   <>      1       (mod 10)

3*27    <>      1       (mod 10)

3*-33   <>      1       (mod 10)

3*37    <>      1       (mod 10)

3*-43   <>      1       (mod 10)

3*47    <>      1       (mod 10)
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

Problem 3: (1 point)

Implement the function `solve_cong(a, b, m, tmax=10)` in `cs3430_s22_hw09.py` that runs the algorithm on Slide 16 in Lecture 18 for solving linear congruences $ax \equiv b \pmod{m}$. The value of the keyword parameter `tmax` specifies the maximum number of the equivalence classes (saved in a list) that this function returns. Step 4 on Slide 16 tells us that then number of equivalence classes

is g, where $g = gcd(a, m)$. When $g \leq tmax$, the length of the list of the equivalence classes is g. Otherwise, it's $tmax$. I just realized that what I'm trying to say is that the length of the list is $min(g, tmax)$. We can also do some random sampling.

The method `test_solve_cong_01()` tests your implementation on the example on Slide 17 in Lecture 18. A call to `solve_cong()` returns a list of five equivalence classes (i.e., $[6]_{50}$, $[16]_{50}$, $[26]_{50}$, $[36]_{50}$, $[46]_{50}$) and tests 5 elements of each equivalence classes by plugging them into $35x \equiv 10 \pmod{50}$. Here's my output.

```
equivalence class 0
35*6    <>      10      (mod 50)
35*56   <>      10      (mod 50)
35*-44  <>      10      (mod 50)
35*106  <>      10      (mod 50)
35*-94  <>      10      (mod 50)

equivalence class 1
35*16   <>      10      (mod 50)
35*66   <>      10      (mod 50)
35*-34  <>      10      (mod 50)
35*116  <>      10      (mod 50)
35*-84  <>      10      (mod 50)

equivalence class 2
35*26   <>      10      (mod 50)
35*76   <>      10      (mod 50)
35*-24  <>      10      (mod 50)
35*126  <>      10      (mod 50)
35*-74  <>      10      (mod 50)

equivalence class 3
35*36   <>      10      (mod 50)
35*86   <>      10      (mod 50)
35*-14  <>      10      (mod 50)
35*136  <>      10      (mod 50)
35*-64  <>      10      (mod 50)

equivalence class 4
35*46   <>      10      (mod 50)
35*96   <>      10      (mod 50)
35*-4   <>      10      (mod 50)
35*146  <>      10      (mod 50)
35*-54  <>      10      (mod 50)
.
----------------------------------------------------------------------
Ran 1 test in 0.000s
OK
```

The methods `test_solve_cong_02()` and `test_solve_cong_03()` randomize tests of `solve_cong()` with smaller and larger values, respectively, of a, b, and m. For example, when you run `test_solve_cong_03()`, you should see outputs similar to the ones below.

```
65902*x <>      38292   (mod 25588) has 2 solution(s)
```

```
        equivalence class 0
        65902*-38923818 <>      38292   (mod 25588)
        65902*-38898230 <>      38292   (mod 25588)

        equivalence class 1
        65902*-38911024 <>      38292   (mod 25588)
        65902*-38885436 <>      38292   (mod 25588)

72202*x <>       98000   (mod 99533) has 1 solution(s)

        equivalence class 0
        72202*3864434000        <>      98000   (mod 99533)
        72202*3864533533        <>      98000   (mod 99533)

74218*x <>      42667   (mod 69884) has no solution
...
```

Problem 4 (2 point)

Implement the function `solve_congs(mvals, avals, num_sols=1)` as the static method of the `crt` class in `crt.py` that runs the Chinese Remainder Theorem (CRT) algorithm on slides 10 – 16 in Lecture 19. The parameter `mvals` is an array of the relatively prime integers $m_1, m_2, ..., m_r$; the parameter `avals` is an array of integers $a_1, a_2, ..., a_r$; and the keyword argument `num_sols` specifies the number of returned solutions to the congruence system

1) $x \equiv a_1 \pmod{m_1}$;

2) $x \equiv a_2 \pmod{m_2}$;

...

r) $x \equiv a_r \pmod{m_r}$.

The function returns the list of `num_sols` integers, each of which solves the system (i.e., it solves each of the r congruences of the system). The `test_crt_01()` tests `solve_congs()` on the following system.

1. $x \equiv 10 \pmod{3}$;
2. $x \equiv 11 \pmod{4}$;
3. $x \equiv 12 \pmod{5}$.

Here's my output.

```
test_crt_01()...
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=10
m/mj=15, bj=-1, aj=11
m/mj=12, bj=-2, aj=12
x <> 10 = 3
```

```
x <> 11 = 4
x <> 12 = 5
[7, 67, 127, 187, 247, 307, 367, 427, 487, 547]
test_crt_01() passed...
```

This test returns 10 solutions: 7, 67, 127, 187, 247, 307, 367, 427, 487, 547. Let's check the first solution 7.

1. $7 \equiv 10 \pmod 3$, because $3|(10-7)$;
2. $7 \equiv 11 \pmod 4$, because $4|(11-4)$;
3. $7 \equiv 12 \pmod 5$, because $5|(12-7)$.

Here's my output of `test_crt_02()`.

```
====== test_crt_02()...
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=1
m/mj=15, bj=-1, aj=1
m/mj=12, bj=-2, aj=1

System 1
x <> 1 = 3
x <> 1 = 4
x <> 1 = 5
[1, 61, 121]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=1
m/mj=15, bj=-1, aj=1
m/mj=12, bj=-2, aj=2

System 2
x <> 1 = 3
x <> 1 = 4
x <> 2 = 5
[37, 97, 157]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=1
m/mj=15, bj=-1, aj=2
m/mj=12, bj=-2, aj=1

System 3
x <> 1 = 3
x <> 2 = 4
```

```
x <> 1 = 5
[46, 106, 166]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=2
m/mj=15, bj=-1, aj=1
m/mj=12, bj=-2, aj=1

System 4
x <> 2 = 3
x <> 1 = 4
x <> 1 = 5
[41, 101, 161]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=2
m/mj=15, bj=-1, aj=2
m/mj=12, bj=-2, aj=1

System 5
x <> 2 = 3
x <> 2 = 4
x <> 1 = 5
[26, 86, 146]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=2
m/mj=15, bj=-1, aj=1
m/mj=12, bj=-2, aj=2

System 6
x <> 2 = 3
x <> 1 = 4
x <> 2 = 5
[17, 77, 137]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=1
m/mj=15, bj=-1, aj=2
m/mj=12, bj=-2, aj=2

System 7
x <> 1 = 3
x <> 2 = 4
x <> 2 = 5
```

```
[22, 82, 142]
m=60
b=-1
b=-1
b=-2
m/mj=20, bj=-1, aj=2
m/mj=15, bj=-1, aj=2
m/mj=12, bj=-2, aj=2

System 8
x <> 2 = 3
x <> 2 = 4
x <> 2 = 5
[2, 62, 122]
====== test_crt_02() passed...
```

What To Submit

Save your source code in `cs3430_s22_hw09.py` and `crt.py` and submit in Canvas along with all the auxiliary files necessary to run your code.

Happy Hacking!