Programa feito em Python para a resolução de sistemas

```python
import math

class P_adic_integer:
    def __init__(self, base):
        if not(is_prime(base)):
            raise ValueError('base must be a prime number')
        self.base = base

        # digits = [a_0, a_1, a_2, a_3, ...]
        self.digits = []

    def __str__(self):
        s = "[..., "
        for i in range(len(self.digits)-1, -1, -1):
            s += str(self.digits[i])
            if(i != 0):
                s += ", "
        s += " ]"
        return s

    def add_digit(self, digit):
        if type(digit) != int or digit < 0 or digit >= self.base:
            raise ValueError("Invalid digit: {}".format(digit))
        self.digits.append(digit)

    def partial_value(self):
        # returns the partial value of the padic integer:
        # a_0 + a_1 * base + a_2 * base**2 + a_3 * base**3 + ...
        now = 0
        for i in range(len(self.digits)):
            now += self.digits[i] * self.base ** i
        return now

    def n_digits(self):
        return len(self.digits)

    def next_possibilities(self):
        # returns the possible next partial values of the padic integer,
        # considering the current value and the base of the number.
        now = self.partial_value()
        nextPower = len(self.digits)
        return range(now, now + self.base**(nextPower + 1), self.base**(nextPower))

    def clone(self):
        new = P_adic_integer(self.base)
        for digit in self.digits:
            new.add_digit(digit)
        return new



class Polynomial:
    def __init__(self):
        # [a, b, c, d, ...] -> a * x**0 + b * x**1 + c * x**2 + ...
        self.coefficients = []

    def __str__(self):
        s = ""
        for i in range(len(self.coefficients)):
            s += str(self.coefficients[i]) + " * x**" + str(i)
            if i != len(self.coefficients) - 1:
                s += " + "
        return s

    def add_term(self, coefficient, power = -1):
        if power < 0:
```

```python
            self.coefficients.append(coefficient)
        elif power < len(self.coefficients):
            self.coefficients[power] += coefficient
        else:
            while not(power < len(self.coefficients)):
                self.coefficients.append(0)
            self.coefficients[power] = coefficient

    def evaluate(self, x):
        value = 0
        for i in range(len(self.coefficients)):
            value += self.coefficients[i] * x**i
        return value

    @staticmethod
    def read():
        p = Polynomial()
        i = 0
        while True:
            leitura = input(f'coefficient for x**{i} (nothing to stop) = ').strip()
            if leitura == '':
                break
            p.add_term(int(leitura))
            i+=1
        return p


def generate_solutions(Polynomial, base, n_digits):
    partialSolutions = []
    solutions = []

    # start solutions finding possible x1's
    for x1 in range(base):
        if Polynomial.evaluate(x1) % base == 0:
            newSol = P_adic_integer(base)
            newSol.add_digit(x1) # a_0 = x1
            partialSolutions.append(newSol)

    # complete solutions finding compatible xn's, n >= 2.
    while len(partialSolutions) != 0:
        partSol = partialSolutions.pop(0)
        n = partSol.n_digits() + 1
        possible_xns = partSol.next_possibilities()
        for c in range(base):
            xn = possible_xns[c]
            if Polynomial.evaluate(xn) % base**n == 0:
                newSol = partSol.clone()
                newSol.add_digit(c)
                if newSol.n_digits() == n_digits:
                    solutions.append(newSol)
                else:
                    partialSolutions.append(newSol)

    return solutions

def is_prime(n):
  for i in range(2,int(math.sqrt(n))+1):
    if (n%i) == 0:
      return False
  return True

def find_next_prime(n):
    n += 1
    while not(is_prime(n)):
        n += 1
    return n

def find_n_solutions_in_lowest_base(Polynomial, n, n_digits):
```

```python
    base = 2
    while True:
        sols = generate_solutions(Polynomial, base, n_digits)
        if len(sols) == n:
            return sols
        base = find_next_prime(base)


def list_commands():
    print('''
Commands:
    set -> set the polynomial used to solve the congruence system.
    solve base n_digits -> solve the system and print the solutions.
    solveMin n_solutions n_digits -> solve the system in the the lowest prime base possible with n distinct
↪   solutions.
    exit -> stop execution.
''')

def printSolutions(solutions):
    abc = 'abcdefghijklmnopqrstuvwxyzABCDEFGHIJKLMNOPQRSTUVWXYZ'
    if len(solutions) == 0:
        print("No solutions found.")
    elif len(solutions) == 1:
        print(f"1 solution found.")
    else:
        print(f"{len(solutions)} distinct solutions found.")
    for i in range(len(solutions)):
        print(f"x_{'{' + str(i - len(abc)) + '}' if i >= len(abc) else abc[i]} = {solutions[i]}")

if __name__ == '__main__':
    list_commands()

    while True:
        command = input('Enter command: ').strip()
        command = command.split(' ')

        if command[0] == 'set':
            try:
                p = Polynomial.read()
                print(f"Polynomial set: {p}")
            except Exception as e:
                print(f"Error: {e}, polynomial set cancelled.")

        elif command[0] == 'solve':
            try:
                base = int(command[1])
                n_digits = int(command[2])
            except Exception:
                print(f"Syntax error...")
                list_commands()
                continue

            try:
                print(f"Solving for x: {p} equiv 0 (mod {base}**n)")
                sols = generate_solutions(p, base, n_digits)
                printSolutions(sols)
            except KeyboardInterrupt:
                print(f"Solve cancelled by user.")
            except Exception as e:
                print(f"Error: {e}, solve cancelled.")

        elif command[0] == 'solveMin':
            try:
                n_solutions = int(command[1])
                n_digits = int(command[2])
            except Exception:
                print(f"Syntax error...")
                list_commands()
```

```
            continue

        try:
            print(f"Solving for x: {p} equiv 0 (mod p**n)")
            sols = find_n_solutions_in_lowest_base(p, n_solutions, n_digits)
            print(f"Lowest base with {n_solutions} distinct solutions: {sols[0].base}.")
            printSolutions(sols)
        except KeyboardInterrupt:
            print(f"Solve cancelled by user.")
        except Exception as e:
            print(f"Error: {e}, solve cancelled.")

    elif command[0] == 'exit':
        print("\nExiting...")

    else:
        print("Command not recognized: {}".format(command[0]))
        list_commands()
        continue

    print('')
```