МИНОБРНАУКИ РОССИИ
ФГБОУ ВО «СГУ ИМЕНИ Н. Г. ЧЕРНЫШЕВСКОГО»

АРИФМЕТИЧЕСКИЕ ОПЕРАЦИИ В ЧИСЛОВЫХ ПОЛЯХ

ЛАБОРАТОРНАЯ РАБОТА

студента 5 курса 531 группы
направления 100501 — Компьютерная безопасность
факультета КНиИТ
Улитина Ивана Владимировича

Проверил
профессор ______________ В. А. Молчанов

Саратов 2023

1 Постановка задачи

Цель работы - изучение основных операций в числовых полях и их программная реализация.

Порядок выполнения работы:

1. Разобрать обычный, бинарный и расширенный алгоритмы Евклида вычисления наибольшего общего делителя целых чисел и привести их программную реализацию;
2. Разобрать алгоритмы решения систем сравнений и привести их программную реализацию;
3. Рассмотреть метод Гаусса решения систем линейных уравнений над конечными полями и привести его программную реализацию.

2 Теоретические сведения

2.1 Алгоритм Евклида

Алгоритм Евклида вычисления наибольшего общего делителя целых чисел a и $b > 0$ состоит из следующих этапов. Положим $a_0 = a$, $a_1 = b$ и выполним последовательно деления с остатком a_i на a_{i+1}:

$$a_0 = a_1 q_1 + a_2, 0 \leq a_2 < a_1,$$

$$a_1 = a_2 q_2 + a_3, 0 \leq a_3 < a_2,$$

$$\dots$$

$$a_{k-2} = a_{k-1} q_{k-1} + a_k, 0 \leq a_k < a_{k-1},$$

$$a_{k-1} = a_k q_k.$$

Так как остатки выполняемых делений образуют строго убывающую последовательность $a_1 > a_2 > a_3 > \dots \geq 0$, то этот процесс обязательно остановится в результате получения нулевого остатка деления. Легко видеть, что НОД(a_0, a_1) = НОД(a_1, a_2) = $\dots$ = НОД(a_{k-1}, a_k) = a_k. Значит, последний ненулевой остаток a_k = НОД(a, b).

2.2 Расширенный алгоритм Евклида

Расширенный алгоритм Евклида позволяет не только вычислять наибольший общий делитель целых чисел a и $b > 0$, но и представлять его в виде НОД$(a, b) = ax + by$ для некоторых $x, y \in Z$. Значения x, y находятся в результате обратного прохода этапов алгоритма Евклида, в каждом из которых уравнение разрешается относительно остатка a_i, который представляется в форме $a_i = ax_i + by_i$ для некоторых $x_i, y_i \in Z$. В результате получается следующая последовательность вычислений:

$$a_0 = a, a_0 = ax_0 + by_0,$$

$$a_1 = b, a_0 = ax_1 + by_1,$$

$$a_2 = a_0 - a_1 q_1, a_2 = ax_2 + by_2,$$

$$a_3 = a_1 - a_2 q_2, a_3 = ax_3 + by_3,$$

$$\dots$$

$$a_i = a_{i-2} - a_{i-1}q_{i-1}, a_i = ax_i + by_i,$$

$$\dots$$

$$a_k = a_{k-2} - a_{k-1}q_{k-1}, a_k = ax_k + by_k,$$

$$0 = a_{k-1} - a_k q_k, 0 = ax_{k+1} + by_{k+1}$$

В правом столбце все элементы $a_k, a_{k-1}, a_{k-2}, ..., a_1, a_0$ представляются в виде $a_i = ax_i + by_i$. Очевидно, что $x_0 = 1, y_0 = 0, x_1 = 0, y_1 = 1$ и выполняются равенства: $a_i = a_{i-2} - a_{i-1}q_{i-1}, x_i = x_{i-2} - x_{i-1}q_{i-1}, y_i = y_{i-2} - y_{i-1}q_{i-1}$. Отсюда последовательно получаются искомые представления всех элементов $a_k, a_{k-1}, a_{k-2}, \dots, a_1, a_0$ и, в частности, представление НОД$(a, b) = a_k = ax_k + by_k$.

2.3 Бинарный алгоритм Евклида

Бинарный алгоритм Евклида является одним из эффективных методов нахождения НОД. Пусть даны целые числа $a > b > 0$. Вычисляется последовательность упорядоченных пар (x_k, y_k) неотрицательных чисел, где $(x_1, y_1) = (a, b)$, и если уже вычислена пара (x_i, y_i), то:

1. находится число e со свойством $2^e y_i \leq x_i \leq 2^{e+1} y_i$;
2. вычисляется $t = \min\{2^{e+1}y_i - x_i, x_i - 2^e y_i\} \geq 0$;
3. если при этом $t \geq y_i$, то тогда полагаем $(x_{i+1}, y_{i+1}) = (y_i, t)$, а если $t > y_i$, то полагаем $(x_{i+1}, y_{i+1}) = (t, y_i)$.

Алгоритм заканчивает свою работу, как только очередное значение y_m оказывается равным нулю. При этом наибольшим общим делителем чисел a и b является число x_m.

Сам по себе бинарный алгоритм Евклида основан на следующих свойствах:

1. НОД($2 \cdot a, 2 \cdot b$) = 2$\cdot$ НОД(a, b);
2. НОД($2 \cdot a, 2 \cdot b + 1$) = НОД($a, 2 \cdot b + 1$)
3. НОД($-a, b$) = НОД(a, b).

2.4 Греко-китайская теорема об остатках

Теорема. Пусть $m_1, m_2, \dots, m_k$ – попарно взаимно простые целые числа и $M = m_1 m_2 \cdots m_k$. Тогда система линейных сравнений

$$\begin{cases} x \equiv a_1 (mod\, m_1) \\ x \equiv a_2 (mod\, m_2) \\ x \equiv a_3 (mod\, m_3) \\ \dots \\ x \equiv a_k (mod\, m_k) \end{cases} . \tag{1}$$

имеет единственное неотрицательное решение по модулю M. При этом, если для каждого $1 \leq j \leq n$ число $\frac{M}{m_j}$ и сравнение $M_j x \equiv a_j (mod\, m_j)$ имеет решение z_j, то решением системы линейных уравнений является остаток по модулю M числа $x = M_1 z_1 + M_2 z_2 + \dots + M_k z_k$.

2.5 Алгоритм Гарнера

Пусть $M = \prod_{i=1}^{k} m_i$, числа $m_1, \dots, m_k$ попарно взаимно просты, и $c_{ij} \equiv m_i^{-1} (mod\, m)_j, i \neq j, i, j \in 1, \dots, k$. Тогда решение системы может быть представлено в виде

$$x = q_1 + q_2 m_1 + q_3 m_2 + \dots + q_k m_1 \dots m_k,$$

где $0 \leq q_i < m_i, i \in 1, \dots, k$, и числа q_i вычисляются по формулам

$$q_1 = u_1 (mod\, m_1)$$

$$q_2 = (u_2 - q_1) c_{12} (mod\, m_2)$$

$$\dots$$

$$q_= (((u_k - q_1) c_{1k} - q_2) c_{2k} - \dots - q_{k-1}) c_{k-1k} (mod\, m)$$

2.6 Метод Гаусса решения систем линейных уравнений над конечными полями

Пусть $P = (P, +, \times, 1, 0)$ — произвольное поле.

Системой n линейных уравнений с m неизвестными $x_1, \dots, x_m$ называется выражение вида:

$$\begin{cases} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1m}x_m = b_1 \ (1) \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2m}x_m = b_2 \ (2) \\ \cdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nm}x_m = b_n \ (n), \end{cases} . \tag{2}$$

где $(1), (2), \ldots, (n)$ — линейные уравнения с неизвестными $x_1, \ldots, x_m$, коэффициентами $a_{11}, a_{12}, \ldots, a_{nm} \in P$ (первый индекс указывает номер уравнения, второй индекс – номер неизвестного) и свободными членами $b_1, \ldots, b_n \in P$ (индекс – номер уравнения). При этом числа $a_{11}, a_{12}, \ldots, a_{nm}$ называются также коэффициентами системы и $b_1, \ldots, b_n$ — свободными членами системы.

Система называется однородной, если $b_1 = \cdots = b_n = 0$. Система (2) кратко записывается в виде

$$\sum_{j=1}^{m} a_{ij}x_j = b_i(i = 1, \ldots, n).$$

Решением системы (2) называется такой упорядоченный набор $\zeta_1, \ldots, \zeta_m \in P$ из m элементов, что при подстановке в уравнения $(1) - (n)$ значений $x_1 = \zeta_1, \ldots, x_m = \zeta_m$ получаются верные равенства $\sum_{j=1}^{m} a_{ij}\zeta_j = b_i(i = 1, \ldots, n)$. Такое решение сокращенно записывается в виде элемента $\zeta = (\zeta_1, \ldots, x_m = \zeta_m)$ множества P^n.

Метод решения системы (2) заключается в равносильном преобразовании ее в систему линейных уравнений с противоречивым уравнением или в разрешенную систему линейных уравнений вида:

$$\begin{cases} x_1 + \cdots + \cdots + a'_{1,r+1}x_{r+1} + \cdots + a'_{1,m}x_m = b'_1 \ (1) \\ x_2 + \cdots + a'_{2,r+1}x_{r+1} + \cdots + a'_{2,m}x_m = b'_2 \ (2) \\ \cdots \\ x_r + a'_{r,r+1}x_{r+1} + \cdots + a'_{r,m}x_m = b'_r \ (r), \end{cases} . \tag{3}$$

где $r \leq n$, так как в процессе элементарных преобразований исходной системы удаляются тривиальные уравнения. В этом случае неизвестные $x_1, \ldots, x_r$ называются разрешенными (или базисными) и $x_{r+1}, \ldots, x_m$ — свободными.

Преобразование системы (2) в равносильную ей разрешенную систему (3)

осуществляется по методу Гаусса с помощью последовательного выполнения следующих Жордановых преобразований:

1. выбираем один из коэффициентов системы $a_{ij} \neq 0$;
2. умножаем i-ое уравнение системы на элемент a_{ij}^{-1};
3. прибавляем к обеим частям остальных k-ых уравнений системы (здесь $k = 1, \ldots, n, k \neq i$) соответствующие части нового i-ого уравнения, умноженные на коэффициент — a_{kj};
4. удаляем из системы тривиальные уравнения (нулевые строки);

При этом выбранный ненулевой элемент a_{ij} называется разрешающим, строка и столбец, содержащие элемент a_{ij}, также называются разрешающими. Такие действия удобнее осуществлять над таблицей коэффициентов системы (2), которая представляется в виде:

$\overline{A} = \begin{pmatrix} a_{11} & \cdots & a_{1m} & b_1 \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & \cdots & a_{nm} & b_n \end{pmatrix}$ и называется матрицей системы (2).

Конечной целью применения метода Гаусса к системе линейных уравнений (2) является преобразование с помощью Жордановых преобразований системы (2) в равносильную ей разрешенную систему (3).

Матрица $\overline{A'}$ такой разрешенной системы (3) имеет вид:

$$\overline{A'} = \begin{pmatrix} 1 & 0 & \cdots & 0 & a'_{1,r+1} & \cdots & a'_{1m} & b'_1 \\ 0 & 1 & \cdots & 0 & a'_{2,r+1} & \cdots & a'_{2m} & b'_2 \\ \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \cdots & \\ 0 & 0 & \cdots & 1 & a'_{r,r+1} & \cdots & a'_{rm} & b'_r \end{pmatrix}$$

Единичные столбцы матрицы $\overline{A'}$ будем называть разрешенными (или базисными), остальные столбцы с коэффициентами a'_{ij} — свободными. Строки, содержащие единицы базисных столбцов, называются разрешенными. Матрица называется разрешенной, если все ее строки разрешенные.

Применение метода Гаусса к системе линейных уравнений (2) в матричной форме равносильно преобразованию матрицы $\overline{A}$ этой системы в эквивалентную ей разрешенную матрицу $\overline{A'}$. При этом на каждом шаге метода Гаусса в преобразуемой матрице с помощью элементарных преобразований формируется новый единичный столбец.

3 Результаты работы

3.1 Описание алгоритмов Евклида вычисления НОД целых чисел

Алгоритм 1 - алгоритм Евклида

Вход: целые числа a, b.

Выход: $d =$ НОД (a, b).

Шаг 1. Положить $a_0 = a, a_1 = b, i = 1$.

Шаг 2. Найти остаток a_{i+1} от деления a_{i-1} на a_i.

Шаг 3. Если $a_{i+1} = 0$, то положить $d = a_i$. Иначе — положить $i = i + 1$ и вернуться к шагу 2.

Шаг 4. Результат: $d =$ НОД (a, b).

Псевдокод:

```
Алгоритм Евклида(a, b):
если b = 0 то
    вернуть a
иначе
    вернуть Алгоритм Евклида(b, a % b)
```

Трудоемкость алгоритма $O(\log(\max\{a, b\}))$.

Алгоритм 2 - расширенный алгоритм Евклида

Вход: целые числа a, b.

Выход: $d =$ НОД (a, b) и коэффициенты x, y.

Шаг 1. Положить $a_0 = a, a_1 = b, x_0 = 1, y_0 = 0, x_1 = 0, x_1 = 1, i = 1$.

Шаг 2. Найти остаток a_{i+1} от деления a_{i-1} на a_i.

Шаг 3. Найти $x_{i+1} = x_{i-1} - (\frac{a_{i-1}}{a_i} \cdot x_i)$.

Шаг 4. Найти $y_{i+1} = y_{i-1} - (\frac{a_{i-1}}{a_i} \cdot y_i)$.

Шаг 5. Если $a_{i+1} = 0$, то положить $d = a_i, x = x_i, y = y_i$. Иначе положить $i = i + 1$ и перейти к шагу 2.

Шаг 6. Результат: $d =$ НОД (a, b) и коэффициенты x, y.

Псевдокод:

```
Расширенный алгоритм Евклида(a, b):
если b = 0 то
    вернуть (a, 1, 0)
```

```
иначе
    (d, x, y) := Расширенный алгоритм Евклида(b, a % b)
    вернуть (d, y, x - (a / b) * y)
```

Трудоемкость алгоритма $O(\log(\max\{a, b\}))$.

Алгоритм 3 - бинарный алгоритм Евклида

Вход: целые числа a, b.

Выход: $d =$ НОД (a, b).

Шаг 1. Положить $a_0 = a, a_1 = b$.

Шаг 2. Если $a = 0$, положить $d = b$.

Шаг 3. Если $b = 0$, положить $d = a$.

Шаг 4. Если $a = b$, положить $d = a$.

Шаг 5. Если $a = 1$ или $b = 1$, положить $d = 1$.

Шаг 6. Если a и b четные, положить $d = 2\cdot$ НОД $(a/2, b/2)$, где НОД — бинарный алгоритм Евклида.

Шаг 7. Если a четное и b нечетное, положить $d =$ НОД $(a/2, b)$.

Шаг 8. Если a нечетное и b четное, положить $d =$ НОД $(a, b/2)$.

Шаг 9. Если a и b нечетные и $b > a$, положить $d =$ НОД $((b - a)/2, a)$.

Шаг 10. Если a и b нечетные и $a > b$, положить $d =$ НОД $((a - b)/2, b)$.

Шаг 11. Результат: d.

Псевдокод:

```
Бинарный алгоритм Евклида(a, b):
если a = b то
    вернуть a
если a = 0 то
    вернуть b
если b = 0 то
    вернуть a
если a чётное и b чётное то
    вернуть 2 * Бинарный алгоритм Евклида(a/2, b/2)
если a чётное и b нечётное то
    вернуть Бинарный алгоритм Евклида(a/2, b)
если a нечётное и b чётное то
    вернуть Бинарный алгоритм Евклида(a, b/2)
если a и b нечётные то
    если a > b то
```

```
            вернуть Бинарный алгоритм Евклида((a-b)/2, b)
        иначе
            вернуть Бинарный алгоритм Евклида((b-a)/2, a)
```

Трудоемкость алгоритма $O(\log(\max\{a,b\})^2)$.

3.2 Описание алгоритмов решения систем сравнений

Алгоритм 4 - решение системы сравнений с помощью греко-китайской теоремы об остатках

Вход: целые числа $a_1, a_2, \ldots, a_n$ и $m_1, m_2, \ldots, m_n$, где любые m_i, m_j попарно простые числа при $0 < i, j < n$.

Выход: целое число x — решение системы сравнений.

Шаг 1. Определить $M = \prod_{i=1}^{n} m_i$.

Шаг 2. Определить $c_1, \ldots, c_n$, где $c_i = \frac{M}{m_i}(1 \leq i \leq n)$.

Шаг 3. Определить $d_1, \ldots, d_n$, где $d_i = c_i^{-1}(mod\, m_i)(1 \leq i \leq n)$. Обратный элемент находится с помощью расширенного алгоритма Евклида (алгоритм 2).

Шаг 4. Результат: $x = \sum_{i=1}^{n} c_i d_i a_i(mod\, M)$.

Псевдокод:

```
Китайская Теорема Об Остатках(сравнения):
// сравнения - список кортежей (a, m), где a - остаток, m - модуль
M := 1
для каждого кортежа (a, m) в сравнениях do
    M := M * m
для каждого кортежа (a, m) в сравнениях do
    Mi := M / m
    (d, x, y) := Расширенный Алгоритм Евклида(Mi, m)
    Mi_inv := y
    x := x + a * Mi * Mi_inv
x := x mod M
вернуть x
```

Трудоемкость алгоритма $O(n^2 b^2)$, где b — число двоичных знаков, с помощью которых записываются числа $c_i d_i a_i$.

Алгоритм 5 - алгоритм Гарнера

Вход: целые числа $a_1, a_2, \ldots, a_n$ и $m_1, m_2, \ldots, m_n$, где любые m_i, m_j попарно простые числа при $0 < i, j < n$.

Выход: целое число x — решение системы сравнений.

Шаг 1. Определить $c_{11}, c_{12}, \ldots, c_{21}, c_{22}, \ldots, c_{nn}$, где $c_{ij} = m_i^{-1}(mod\, m_j)(1 \leq i, j \leq n)$. Обратный элемент находится с помощью расширенного алгоритма Евклида (алгоритм 2).

Шаг 2. Определить последовательность Q, которая изначально состоит из одного элемента q_1 и имеет длину $l = 1$. Положить $i = 0$.

Шаг 3. Положить $i = i + 1$, $q = u_i$.

Шаг 4. Для $j = 1, \ldots, l$ выполнить $q = (q - q_j) \cdot c_{ji}$.

Шаг 5. Добавить $q(mod\, m_i)$ в Q и положить $l = l + 1$. Если $i \leq n$, перейти к шагу 3.

Шаг 6. Положить $x = q_1$. Результат: $x = x + \sum_{i=2}^{n}(q_i \prod_{j=1}^{i-1} m_i)$.

Псевдокод:

```
Алгоритм Гарнера(сравнения):
// сравнения - список кортежей (ai, mi), где ai - остаток, mi -
↪ модуль

n := количество элементов в сравнениях
b := [0, 0, ..., 0] // Инициализация списка b нулями длиной n
coeff := [1, 1, ..., 1] // Инициализация списка коэффициентов
↪ coeff нулями длиной n

// Вычисляем коэффициенты coeff
для i от 1 до n-1 включительно do
    для j от 0 до i-1 включительно do
        coeff[i] := coeff[i] * mj % mi

x := a0

// Решаем для каждого i
для i от 1 до n-1 включительно do
    // Вычисляем b[i]
    b[i] := (ai - x) * Расширенный Алгоритм Евклида(coeff[i], mi)
    ↪ % mi

    // Корректировка b[i]
    для j от 0 до i-1 включительно do
        b[i] := (b[i] - b[j]) * обратный_элемент(mj, mi) % mi
```

```
        // Обновляем x
        x := x + b[i] * coeff[i]

    вернуть x
```

Трудоемкость алгоритма $O(n^2b^2)$.

Алгоритм 6 - метод Гаусса решения систем линейных уравнений над конечными полями

Вход: Модуль конечного поля p, коэффициенты системы $a_{00}, \ldots, a_{n-1,m-1}$, свободные члены системы $b_0, \ldots, b_{n-1}$.

Выход: Если у системы есть решение, то выход: список $X = (\zeta_0, \ldots, \zeta_{m-1})$, который является решением системы уравнений. Если решения у системы нет, выход: сообщение ”Система не имеет решений”.

Шаг 1. Определить список X длиной m, состоящий из 0.

Шаг 2. Положить $i = 0$.

Шаг 3. Проверить, образуют ли $a_{i0}, \ldots, a_{i,m-1}$ и b_i тривиальную строку. Если $a_{i0} = \cdots = a_{i,m-1} = 0$, но $b_i \neq 0$, то вывести в качестве результата ”Система не имеет решений”.

Шаг 4. Положить $inv = a_{ii}^{-1}(mod\, p)$.

Шаг 5. При $t = i, \ldots, m : a_{it} = a_{it} \cdot inv(mod\, p)$.

Шаг 6. Положить $b_i = b_i \cdot inv(mod\, p)$.

Шаг 7. Положить $k = i + 1$.

Шаг 8. Положить $fact = a_{ki}$, $b_k = (b_k - b_i \cdot fact)(mod\, p)$. Если $b_k < 0$, то $b_k = b_k + p$.

Шаг 9. Положить $j = 0$.

Шаг 10. Положить $a_{kj} = (a_{kj} - a_{ij} \cdot fact)(mod\, p)$. Если $a_{kj} < 0$, то $a_{kj} = a_{kj} + p$.

Шаг 11. Если $j < m$, положить $j = j + 1$ и перейти к шагу 10.

Шаг 12. Если $k < n$, положить $k = k + 1$ и перейти к шагу 8.

Шаг 13. Если $i < n$, положить $i = i + 1$ и перейти к шагу 3.

Шаг 14. Положить $j = m - 1$.

Шаг 15. Положить $x_j = b_j$.

Шаг 16. Положить $k = j + 1$.

Шаг 17. Положить $x_j = (x_j - a_{jk} \cdot x_k)(mod\, p)$. Если $x_j < 0$, то положить $x_j = x_j + p$.

Шаг 18. Если $k < m$, положить $k = k + 1$ и перейти к шагу 17.

Шаг 19. Положить $inv = a_{jj}^{-1} (mod\, p)$.
Шаг 20. Положить $x_j = (x_j \cdot inv)(mod\, p)$.
Шаг 21. Если $j > 0$, положить $j = j - 1$ и перейти к шагу 15.
Шаг 22. Результат: список X.

Псевдокод:

```
Метод Гаусса(m, n, a_values, terms, p)
// m - количество переменных
// n - количество уравнений
// a_values - матрица коэффициентов системы размером n x m
// terms - вектор правых частей уравнений размером n
// p - простое число (поле)

Создать вектор x размером m и заполнить нулями

// Прямой ход:
Для каждого уравнения i от 0 до n-1 выполнить:
    Проверить, является ли уравнение тривиальным:
        is_trivial = check_if_trivial(a_values[i], terms[i])
        Если is_trivial == -1:
            Вывести "Система не имеет решений!"
            Завершить выполнение

    inv = Расширенный Алгоритм Евклида(a_values[i][i], p)

    Для каждого столбца j от i до m выполнить:
        a_values[i][j] = (a_values[i][j] * inv) % p

    terms[i] = (terms[i] * inv) % p

    Для каждого уравнения k от i+1 до n выполнить:
        fact = a_values[k][i]
        terms[k] = (terms[k] - terms[i] * fact) % p
        Если terms[k] < 0:
            terms[k] += p

        Для каждого столбца j от 0 до m выполнить:
            a_values[k][j] = (a_values[k][j] - a_values[i][j] *
             ↪  fact) % p
            Если a_values[k][j] < 0:
                a_values[k][j] += p
```

```
// Обратный ход
Для каждого уравнения j от n-1 до 0 с шагом -1 выполнить:
    x[j] = terms[j]
    Для каждого столбца k от j+1 до m выполнить:
        x[j] = (x[j] - a_values[j][k] * x[k]) % p
        Если x[j] < 0:
            x[j] += p
    inv = Расширенный Алгоритм Евклида(a_values[j][j], p)
    x[j] = (x[j] * inv) % p

вернуть x
```

Трудоемкость алгоритма $O(n^2 \cdot m)$

3.3 Код программы, реализующей рассмотренные алгоритмы

```rust
use std::io;

fn read_integer() -> i32 {
    let mut n = String::new();
    io::stdin()
        .read_line(&mut n)
        .expect("failed to read input.");
    let n: i32 = n.trim().parse().expect("invalid input");
    n
}


fn euclid_gcd(a: i32, b: i32) -> i32 {
    if b == 0 {
        return a;
    };
    let r = a % b;
    return euclid_gcd(b, r);
}


fn euclid_gcd_extended(a: i32, b: i32, _x: i32, _y: i32) -> (i32, i32, i32, i32) {
    if a == 0 {
```

```
        return (a, b, 0, 1);
    }
    else {
        let (_, d, x1, y1) = euclid_gcd_extended(b % a, a, 0, 0);
        let x = y1 - (b / a) * x1;
        let y = x1;
        return (0, d, x, y)
    }
}


fn euclid_gcd_binary(a: i32, b: i32) -> i32 {
    if a == 0 {
        return b;
    }
    if b == 0 {
        return a;
    }
    if a == b {
        return a;
    }

    if a == 1 || b == 1 {
        return 1;
    }

    if a % 2 == 0 && b % 2 == 0 {
        return 2 * euclid_gcd_binary(a / 2, b / 2);
    }

    if a % 2 == 0 && b % 2 != 0 {
        return euclid_gcd_binary(a / 2, b);
    }

    if a % 2 != 0 && b % 2 == 0 {
        return euclid_gcd_binary(a, b / 2);
    }

    if a % 2 != 0 && b % 2 != 0 && b > a {
        return euclid_gcd_binary((b - a) / 2, a);
    }
```

```
    return euclid_gcd_binary((a - b) / 2, b);
}


fn solve_euclid() {
    println!("Введите число a:");
    let a = read_integer();

    println!("Введите число b:");
    let b = read_integer();

    println!("Алгоритм Евклида: {}", euclid_gcd(a, b));
    println!("Расширенный алгоритм Евклида: {:?}",
             euclid_gcd_extended(a, b, 0, 0));
    println!("Бинарный алгоритм Евклида: {}", euclid_gcd_binary(a, b));
}


fn print_array(arr: Vec<i32>) {
    print!("[");
    for i in 0..arr.len() {
        print!("{}", arr[i]);
        if i < arr.len() - 1 {
            print!(", ");
        }
    }
    println!("]");
}


fn check_coprime(modules: Vec<i32>, n: i32) -> bool {
    for i in 0..n {
        for j in 0..n {
            let m_i = modules[i as usize];
            let m_j = modules[j as usize];
            if euclid_gcd(m_i, m_j) != 1 && i != j {
                return false;
            }
        }
    }
```

```
    return true;
}


fn chinese_remainder_theorem(u_values: Vec<i32>, modules: Vec<i32>, n: i32) -> i32 {
    let mut big_m: i32 = 1;
    for i in 0..n {
        big_m *= modules[i as usize];
    }

    let mut params: Vec<i32> = Vec::new();
    let mut bezouts: Vec<i32> = Vec::new();
    for i in 0..n {
        let cur_m: i32 = modules[i as usize];
        let cur_param: i32 = big_m / cur_m;
        params.push(cur_param);

        let (_, _, inv, _) = euclid_gcd_extended(cur_param, cur_m, 0, 0);
        bezouts.push(inv);
    }

    let mut solution: i32 = 0;
    for i in 0..n {
        solution += u_values[i as usize]
                  * bezouts[i as usize]
                  * params[i as usize];
    }

    println!("{}", solution.rem_euclid(big_m));
    return 0;
}


fn harner_algorithm(u_values: Vec<i32>, modules: Vec<i32>, n: i32) -> i32 {
    let mut coeffs: Vec<Vec<i32>> = Vec::new();

    for i in 0..n {
        let mut cur_cs: Vec<i32> = Vec::new();
        let m_i: i32 = modules[i as usize];
        for j in 0..n {
```

```
                let m_j: i32 = modules[j as usize];
                let (_, _, cur_c, _) = euclid_gcd_extended(m_i, m_j, 0, 0);
                cur_cs.push(cur_c.rem_euclid(m_j));
            }
            coeffs.push(cur_cs);
        }

        let mut qs: Vec<i32> = Vec::new();

        for i in 0..n {
            let mut cur_q: i32 = u_values[i as usize];
            for j in 0..qs.len() {
                cur_q = (cur_q - qs[j as usize]) * coeffs[j as usize][i as usize];
    ↪
            }
            qs.push(cur_q.rem_euclid(modules[i as usize]));
        }

        let mut solution: i32 = qs[0];

        for i in 1..n {
            let mut mult: i32 = 1;
            for j in 0..i {
                mult *= modules[j as usize];
            }
            solution += qs[i as usize] * mult;
        }

        print!("{}", solution);
        return 0;
    }


    fn solve_comparison() -> () {
        println!("Введите число сравнений: ");
        let n = read_integer();


        println!("Введите значения u: ");
        let mut u_values: Vec<i32> = Vec::new();
        for _i in 0..n {
```

```
        let u = read_integer();
        u_values.push(u);
    }

    println!("Введите модули сравнений: ");
    let mut modules: Vec<i32> = Vec::new();
    for _i in 0..n {
        let m = read_integer();
        modules.push(m);
    }

    if !check_coprime(modules.clone(), n) {
        println!("Некоторые модули не взаимнопросты!");
        return ();
    }

    println!("Греко-китайская теорема: ");
    chinese_remainder_theorem(u_values.clone(), modules.clone(), n);
    println!("Алгоритм Гарнера: ");
    harner_algorithm(u_values.clone(), modules.clone(), n);
    return ();
}


fn check_if_trivial(coeffs: Vec<i32>, b: i32) -> i32 {
    let mut is_cf_zeros: bool = true;
    for i in 0..coeffs.len() {
        if coeffs[i as usize] != 0 {
            is_cf_zeros = false;
            break;
        }
    }

    if is_cf_zeros && b == 0 {
        return 1;
    }

    if is_cf_zeros && b != 0 {
        return -1;
    }

```

```
    return 0;
}


fn check_correct(a_values: Vec<Vec<i32>>, b: Vec<i32>, x: Vec<i32>) -> i32 {
    for i in 0..a_values.len() {
        let mut ans = 0;
        for j in 0..a_values[i as usize].len() {
            ans += a_values[i as usize][j as usize] * x[j as usize];
        }
        if ans != b[i as usize] {
            return 0;
        }
    }
    return 1;
}

fn mod_inverse(a: i32, n: i32) -> i32 {
    let (_, g, x, _) = euclid_gcd_extended(a, n, 0, 0);
    if g != 1 {
        return -1;
    } else {
        let result = (x % n + n) % n;
        return result;
    }
}

fn gauss () -> () {
    println!("Введите число уравнений: ");
    let n = read_integer();

    println!("Введите число неизвестных: ");
    let m = read_integer();

    println!("Введите модуль: ");
    let p = read_integer();

    println!("Введите коэффициенты: ");
    let mut a_values: Vec<Vec<i32>> = Vec::new();
    for i in 0..n {
        println!("Введите коэффициенты {}-го уравнения:", i + 1);
```

```
        let mut a_line: Vec<i32> = Vec::new();
        for _j in 0..m {
            let a = read_integer();
            a_line.push(a);
        }
        a_values.push(a_line);
    }

    println!("Введите свободные коэффициенты:");
    let mut terms: Vec<i32> = Vec::new();
    for _j in 0..n {
        let t = read_integer();
        terms.push(t);
    }

    let mut x: Vec<i32> = vec![0; m as usize];

    for i in 0..n {
        let is_trivial: i32 = check_if_trivial(a_values[i as usize].clone(),
                                               terms[i as usize].clone());
        if is_trivial == -1 {
            println!("Система не имеет решений!");
            return ();
        }

        let inv: i32 = mod_inverse(a_values[i as usize][i as usize], p);

        for j in i..m {
            a_values[i as usize][j as usize] = (a_values[i as usize][j as usize] * inv).rem_euclid(p);
        }

        terms[i as usize] = (terms[i as usize] * inv).rem_euclid(p);

        for k in (i + 1)..n {
            let fact: i32 = a_values[k as usize][i as usize];
            terms[k as usize] = (terms[k as usize] - terms[i as usize]  * fact).rem_euclid(p);
            if terms[k as usize] < 0 {
                terms[k as usize] += p;
            }
```

```

            for j in 0..m {
                a_values[k as usize][j as usize] = (a_values[k as usize][j as usize] - a_values[i as usize][j as usize] * fact).rem_euclid(p);

                if a_values[k as usize][j as usize] < 0 {
                    a_values[k as usize][j as usize] += p;
                }
            }
        }
    }

    for j in (0..terms.len()).rev() {
        x[j as usize] = terms[j as usize];
        for k in (j + 1)..m as usize{
            x[j as usize] = (x[j as usize] - a_values[j as usize][k as usize] * x[k as usize]).rem_euclid(p);
            if x[j as usize] < 0 {
                x[j as usize] += p;
            }
        }

        let inv: i32 = mod_inverse(a_values[j as usize][j as usize], p);
        x[j as usize] = (x[j as usize] * inv).rem_euclid(p);
    }

    let a_bef_matrix: Vec<Vec<i32>> = a_values.clone();

    println!("Методом Гаусса получена разряженная матрица:");
    for i in 0..n {
        let cur_line = &mut a_values[i as usize];
        cur_line.push(terms[i as usize]);
        print_array(cur_line.clone());
    }

    if check_correct(a_bef_matrix.clone(), terms.clone(), x.clone()) == 0 {
        println!("Система не имеет решений!");
        return ();
    }

    println!("Частное решение системы:");
```

```
    print_array(x.clone());
    println!("Общее решение системы:");
    for i in 0..m {
        if x[i as usize] == 0 {
            println!("x_{} = {}", i + 1, 0)
        }
        else {
            let mut ans = String::from("");
            let x_inv: i32 = mod_inverse(a_values[i as usize][i as usize], p);
            for j in 0..m {
                let cur_a = (a_values[i as usize][j as usize] * x_inv).rem_euclid(p);

                if i != j {
                    if cur_a > 0 {
                        ans.push_str(" - ");
                        ans.push_str(&cur_a.to_string());
                    }
                    if cur_a < 0 {
                        ans.push_str(" + ");
                        ans.push_str(&(-cur_a).to_string());
                    }
                    if cur_a != 0
                    {
                        ans.push('x');
                        ans.push_str(&(j + 1).to_string());
                    }
                }
            }
            println!("x_{} = {}{}", i + 1, terms[i as usize], ans);
        }
    }
    return ();
}


fn main() {
    println!("Выберите опцию:");
    println!("1 - алгоритмы Евклида;");
    println!("2 - Греко-китайская теорема и алгоритм Гарнера;");
    println!("3 - алгоритм Гаусса;");
```

```

    let n = read_integer();

    match n {
        1 => solve_euclid(),
        2 => solve_comparison(),
        3 => gauss(),
        _ => println!("Введено неверное число!"),
    }
}
```

3.4 Результаты тестирования программ

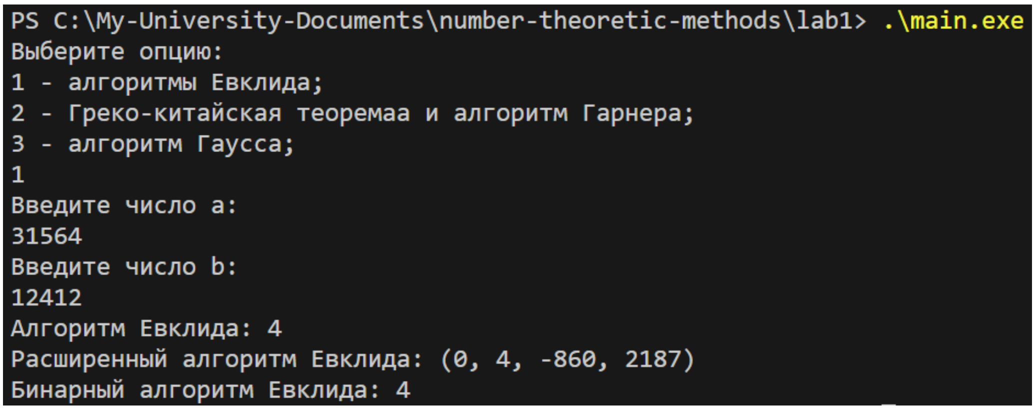

Рисунок 1 – Тест алгоритмов Евклида

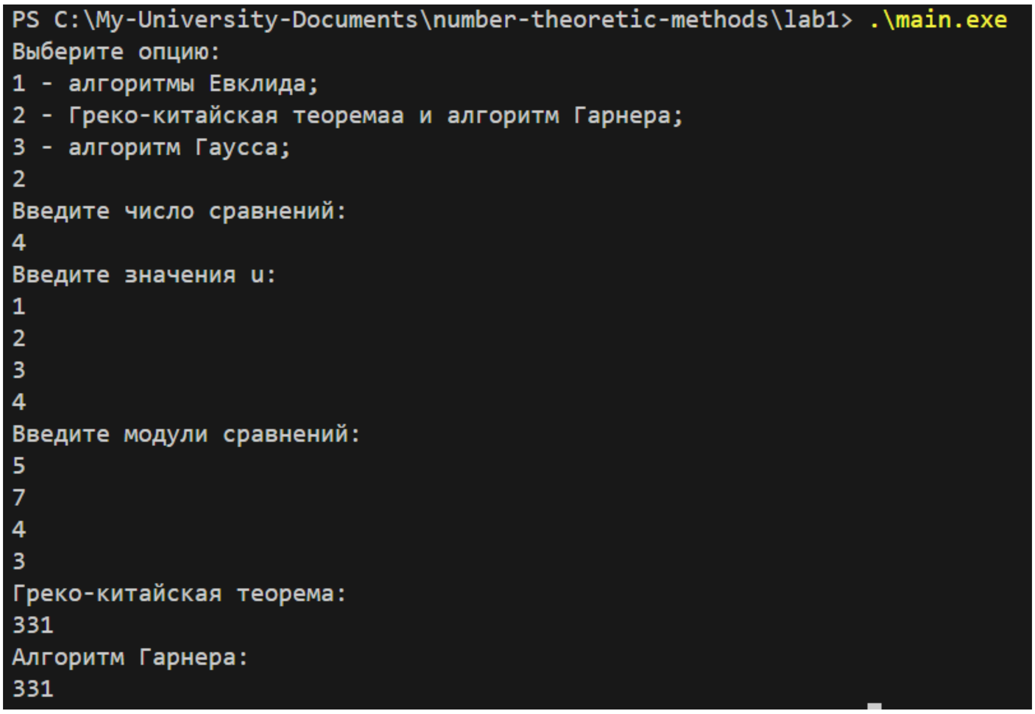

Рисунок 2 – Тест алгоритмов решения систем сравнений

```
Введите число уравнений:
3
Введите число неизвестных:
4
Введите модуль:
7
Введите коэффициенты:
Введите коэффициенты 1-го уравнения:
1
0
-2
-4
Введите коэффициенты 2-го уравнения:
2
-2
0
-3
Введите коэффициенты 3-го уравнения:
0
2
-4
-5
Введите свободные коэффициенты:
2
1
3
Методом Гаусса получена разряженная матрица:
[1, 0, 5, 3, 2]
[0, 1, 5, 1, 5]
[0, 0, 0, 0, 0]
Частное решение системы:
[2, 5, 0, 0]
Общее решение системы:
x_1 = 2 - 5x3 - 3x4
x_2 = 5 - 5x3 - 1x4
x_3 = 0
x_4 = 0
```

Рисунок 3 – Тест реализации метода Гаусса

ЗАКЛЮЧЕНИЕ

В данной лабораторной работе были рассмотрены теоретические сведения об обычном, расширенном и бинарном алгоритме Евклида, греко-китайская теорема об остатках, алгоритм Гарнера и метод Гаусса решения линейных уравнений над конечными полями. На их основе были составлены соответствующие алгоритмы. Была произведена оценка сложности созданных алгоритмов. Они послужили фундаментом для программной реализации, которая впоследствии успешно прошла тестирование, результаты которого были прикреплены к отчету вместе с листингом программы, написанной на языке Rust с использованием стандартных библиотек языка.