МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ
РОССИЙСКОЙ ФЕДЕРАЦИИ

Федеральное государственное автономное образовательное учреждение
высшего образования «Санкт-Петербургский политехнический университет
Петра Великого»

Институт компьютерных наук и кибербезопасности

Направление: 02.03.01 Математика и компьютерные науки

Отчет по курсовой работе

по дисциплине: «Дискретная математика»

«Калькулятор большой конечной арифметики».

Вариант 48

Студент,
группы 5130201/40003 _____ Селезнев К. Д.

Руководитель,
Преподаватель _____ Востров А. В.

«____»_____ 2025 г.

Санкт-Петербург, 2025

Содержание

Введение

Курсовая работа посвящена реализации калькулятора большой конечной арифметики на основе малой конечной арифметики. Работа выполнена на языке программирования C++17.

Задача состоит в построении калькулятора для конечного коммутативного кольца с единицей $\langle Z_8; +, * \rangle$, где операции определены над словами длины до 8 символов. Базовые действия реализованы через малую арифметику $\langle Z_8; +, * \rangle$, в которой задано правило $+1$.

В рамках работы реализованы:

Система правил кольца на основе заданного правила $+1$.

Модуль малой арифметики для действий над односимвольными элементами.

Модуль большой арифметики для многоразрядных чисел с управлением переносами и контролем переполнения.

Интерактивный интерфейс калькулятора с меню действий.

Набор тестов для проверки математических свойств.

Калькулятор поддерживает четыре действия: сложение, вычитание, умножение и деление с остатком. Все вычисления выполняются в рамках заданного конечного кольца с ограничением на максимальное количество разрядов.

1 Математическое описание

1.1 Малая и большая конечные арифметики

Алгебраическая структура представляет собой множество значений с определенными на нем операциями и отношениями.

Коммутативное кольцо с единицей $\langle Z; +, * \rangle$ удовлетворяет следующим аксиомам:

1. Ассоциативность сложения: $(a + b) + c = a + (b + c)$
2. Существование нулевого элемента: $\exists 0 \in Z \ (\forall a \in Z : a + 0 = 0 + a = a)$
3. Коммутативность сложения: $a + b = b + a$
4. Ассоциативность умножения: $(a * b) * c = a * (b * c)$
5. Дистрибутивность: $a * (b + c) = a * b + a * c$
6. Коммутативность умножения: $a * b = b * a$

Малая конечная арифметика представляет собой конечное коммутативное кольцо с единицей $\langle Z_8; +, * \rangle$, на котором определены действия вычитания и деления.

В данной работе используется кольцо $Z_8 = \{a, b, c, d, e, f, g, h\}$ размера 8, где:

Нулевой элемент: a

Единичный элемент: b

Отношение порядка задается правилом «+1» согласно варианту 48:

$a \to b \to e \to d \to g \to c \to f \to h \to a$

Большая конечная арифметика представляет собой конечное коммутативное кольцо с единицей $\langle Z_8^n; +, * \rangle$, элементами которого являются слова длины до n над алфавитом Z_8. Действия определены позиционно с учетом переноса разрядов. Деление определено с остатком.

В данной работе $n = 8$, поэтому итоговая структура имеет вид $\langle Z_8^8; +, * \rangle$.

Каждому символу сопоставлен индекс:

Символ	a	b	e	d	g	c	f	h
Индекс	0	1	2	3	4	5	6	7

1.2 Примеры арифметических действий

Малая арифметика Z_8

Пример 1: $a + c = c$ (сложение с нулем: $0 + 5 = 5$)

Вычисление: начинаем с элемента a и применяем действие +one c раз:

$a \xrightarrow{+1} b \xrightarrow{+1} e \xrightarrow{+1} d \xrightarrow{+1} g \xrightarrow{+1} c$

Результат: $a + c = c$.

Пример 2: $a * c = a$ (умножение на ноль: $0 * 5 = 0$)

Вычисление: умножение реализуется как повторное сложение. Нужно добавить a к результату c раз.

Результат: $a * c = a$.

Пример 3: $b + b = e$ (сложение единиц: $1 + 1 = 2$)

Вычисление: начинаем с b и применяем действие +one b раз:

$b \xrightarrow{+1} e$

Результат: $b + b = e$.

Пример 4: $e + d = g$ (сложение: $2 + 3 = 5$)

Вычисление: начинаем с e и применяем действие +one d раз:

$e \xrightarrow{+1} d \xrightarrow{+1} g \xrightarrow{+1} c$

Результат: $e + d = c$.

Пример 5: $e * e = g$ (умножение: $2 * 2 = 4$)

Вычисление: сложим e два раза.

$e \xrightarrow{+1} d \xrightarrow{+1} g$

Поэтому результат: $e + e = g$.

Большая арифметика: пример сложения

В большой арифметике сложение выполняется поразрядно справа налево с учетом переноса.

Рассмотрим сложение чисел $fff + eee$ в позиционной записи Z_8 (вариант 48), где $f = 6$ и $e = 2$.

	Номер разряда			
	3	2	1	0
Перенос (вход)		b	b	
Операнд 1		f	f	f
Операнд 2		e	e	e
Сумма		b	b	a
Перенос (выход)	b	b	b	b
Итог	b	b	b	a

Пошаговое вычисление:

Разряд 0: $f + e = 6 + 2 = 8 \equiv 0 \pmod 8$, результат $= a$, перенос $= b$.

Разряд 1: $f + e + b = 6 + 2 + 1 = 9 \equiv 1 \pmod 8$, результат $= b$, перенос $= b$.

Разряд 2: $f + e + b = 6 + 2 + 1 = 9 \equiv 1 \pmod 8$, результат $= b$, перенос $= b$.

Разряд 3: итоговый перенос записывается в старший разряд: b.

Итоговый результат: $fff + eee = bbba$

2 Особенности реализации

Программа написана на C++17 с использованием стандартной библиотеки. Сборка проекта осуществляется через Makefile.

2.1 Структура проекта

Файловая структура организована следующим образом:

constants.h и constants.cpp содержат определения констант кольца.

utils.h и utils.cpp содержат вспомогательные функции и таблицы действий.

arithmetic.h и arithmetic.cpp содержат базовые операции малой арифметики.

BigFiniteNumber.h и BigFiniteNumber.cpp содержат класс больших чисел.

main.cpp содержит интерфейс калькулятора.

Makefile содержит правила сборки проекта.

2.2 Архитектура классов

Система строится на взаимодействии основных компонентов.

2.2.1 Модуль constants

Отвечает за хранение констант кольца.

Основные константы: `SYMBOLS` — упорядоченный список элементов кольца, `MOD` — размер кольца (8), `ADDITIVE_UNIT` — символ нуля (`'a'`), `MULTIPLICATIVE_UNIT` — символ единицы (`'b'`), `MAX_DIGITS` — максимальное количество разрядов (8).

Определение констант:

```
const std::vector<char> SYMBOLS ={'a', 'b', 'e', 'd', 'g', 'c', 'f', 'h'};
const int MOD = SYMBOLS.size();
const char ADDITIVE_UNIT = SYMBOLS[0];
const char MULTIPLICATIVE_UNIT = SYMBOLS[1];
const int MAX_DIGITS = 8;
```

2.2.2 Модуль utils

Реализует таблицы действий и вспомогательные функции.

Основные таблицы:

`NEXT_SYMBOL_MAP` — таблица переходов для правила «+1».

`SYMBOL_NEGATION_MAP` — таблица аддитивных инверсий.

`ADDITION_TABLE` — таблица сложения с переносом.

Функция init_symbols_maps

Вход: map<char, char> NEXT_SYMBOL_MAP (таблица переходов).

Выход: заполняет глобальные таблицы.

Описание: инициализирует все необходимые таблицы действий. Сначала заполняется таблица переходов. Затем вычисляются аддитивные инверсии для всех элементов. После этого строится полная таблица сложения с учетом переноса для всех возможных комбинаций входных данных.

Таблица переходов NEXT_SYMBOL_MAP

Эта таблица хранит правило «+1» для варианта 48. Для каждого символа кольца определен следующий символ в циклической последовательности.

```
NEXT_SYMBOL_MAP['a'] = 'b';
NEXT_SYMBOL_MAP['b'] = 'e';
NEXT_SYMBOL_MAP['c'] = 'f';
NEXT_SYMBOL_MAP['d'] = 'g';
NEXT_SYMBOL_MAP['e'] = 'd';
NEXT_SYMBOL_MAP['f'] = 'h';
NEXT_SYMBOL_MAP['g'] = 'c';
NEXT_SYMBOL_MAP['h'] = 'a';
```

Таблица переходов используется для реализации базового действия «+1», на основе которого строятся все остальные действия кольца.

Таблица отрицаний SYMBOL_NEGATION_MAP

Эта таблица хранит аддитивные инверсии элементов. Для каждого элемента x находится такой элемент $-x$, что $x + (-x) = a$ (нулевой элемент).

Построение таблицы отрицаний: для каждого элемента кольца применяется действие «+1» до тех пор, пока не получится нулевой элемент. Таким образом, количество сделанных шагов соответствует аддитивной инверсии исходного элемента.

```
for (char c : SYMBOLS){
    char current = c;
    char counter = get_additive_unit();

    while (current != get_additive_unit()){
        current = NEXT_SYMBOL_MAP.at(current);
        counter = NEXT_SYMBOL_MAP.at(counter);
    }
    SYMBOL_NEGATION_MAP[c] = counter;
}
```

Листинг 1: Построение таблицы отрицаний

Пример: для элемента b нужно выполнить 7 шагов по «+1», чтобы вернуться к a. Поэтому $-b = h$.

Таблица сложения ADDITION_TABLE

Эта таблица хранит результаты сложения с переносом для всех возможных комбинаций двух цифр и входного переноса.

Структура записи в таблице: $(digit_1, digit_2, carry\_in) \to (sum, carry\_out)$, где $digit_1$ — первая цифра, $digit_2$ — вторая цифра, $carry\_in$ — входной перенос из предыдущего разряда, sum — результат сложения в текущем разряде, $carry\_out$ — выходной перенос в следующий разряд.

Построение таблицы сложения: таблица строится для всех комбинаций двух операндов и входного переноса. Для каждой тройки $(c_1, c_2, carry\_in)$ вычисляется сумма и выходной перенос.

Алгоритм вычисления: к первому операнду c_1 последовательно применяется действие «+1» столько раз, сколько шагов от нуля до c_2. Затем к полученному результату применяется действие «+1» столько раз, сколько шагов от нуля до $carry\_in$. Если во время вычисления происходит переход через нулевой элемент, фиксируется переполнение и устанавливается выходной перенос.

```
for (char c1 : SYMBOLS){
    for (char c2 : SYMBOLS){
        for (char carry_in : SYMBOLS){
            char result = c1;
            int steps = 0;

            char temp_c2 = get_additive_unit();
            while (temp_c2 != c2){
                result = NEXT_SYMBOL_MAP.at(result);
                temp_c2 = NEXT_SYMBOL_MAP.at(temp_c2);
                steps++;
                if (result == get_additive_unit()){
                    steps = MOD + steps;
                }
            }

            char temp_carry = get_additive_unit();
            while (temp_carry != carry_in){
                result = NEXT_SYMBOL_MAP.at(result);
                temp_carry = NEXT_SYMBOL_MAP.at(temp_carry);
                steps++;
                if (result == get_additive_unit() &&
                    temp_carry != carry_in){
                    steps = MOD + steps;
```

```
                }
            }

            char carry_out = get_additive_unit();
            if (steps >= MOD){
                carry_out = get_multiplicative_unit();
            }

            ADDITION_TABLE[std::make_tuple(c1, c2, carry_in)] =
                std::make_pair(result, carry_out);
        }
    }
}
```

Листинг 2: Построение таблицы сложения с переносом

Пример использования таблицы: для вычисления $f + e$ с входным переносом b обращаемся к таблице. Вызов `ADDITION_TABLE[(f, e, b)]` возвращает пару `(a, b)`, где первый элемент — результат сложения в текущем разряде, второй элемент — перенос в следующий разряд.

Таблица сложения содержит $8 * 8 * 8 = 512$ записей, охватывая все возможные комбинации входных данных для кольца Z_8.

2.2.3 Модуль arithmetic

Реализует базовые операции малой арифметики. Основные функции: `next_symbol`, `prev_symbol`, `symbolic_add`, `symbolic_negate`, `symbolic_multiply`.

Функция `next_symbol`

Вход: символ `c`.

Выход: следующий символ в кольце.

Описание: возвращает следующий элемент согласно правилу «+1». Использует таблицу переходов `NEXT_SYMBOL_MAP`.

```
char next_symbol(char current_char){
    if (!is_valid_digit(current_char)){
        throw std::logic_error("Invalid character");
    }
    return NEXT_SYMBOL_MAP.at(current_char);
}
```

Листинг 3: Функция next_symbol

Функция `symbolic_add`

Вход: два символа `a` и `b`.

Выход: их сумма.

Описание: сложение выполняется через циклическое прибавление единицы. К первому аргументу применяется `next_symbol` столько раз, сколько шагов от нуля до второго аргумента. Использует таблицу сложения для получения результата без переноса.

```
char symbolic_add(char char1, char char2){
    auto [sum, carry] = ADDITION_TABLE[std::make_tuple(char1,
       char2, get_additive_unit())];
    return sum;
}
```

Листинг 4: Функция symbolic_add

Функция `symbolic_negate`

Вход: символ `c`.

Выход: аддитивная инверсия.

Описание: возвращает противоположный элемент из таблицы отрицаний. Для элемента x находится такой элемент $-x$, что $x + (-x) = 0$.

```
char symbolic_negate(char c){
    return SYMBOL_NEGATION_MAP.at(c);
}
```

Листинг 5: Функция symbolic_negate

Функция `symbolic_multiply`

Вход: два символа `a` и `b`.

Выход: произведение.

Описание: умножение сводится к многократному сложению. Первый операнд прибавляется к результату столько раз, сколько единиц содержится в операнде b. Обрабатываются специальные случаи умножения.

```
char symbolic_multiply(char char1, char char2){
    if (char1 == get_additive_unit() ||
        char2 == get_additive_unit()){
        return get_additive_unit();
```

```
    }

    if (char1 == get_multiplicative_unit()) return char2;
    if (char2 == get_multiplicative_unit()) return char1;

    char result = get_additive_unit();
    char counter = get_additive_unit();

    while (counter != char2){
        result = symbolic_add(result, char1);
        counter = next_symbol(counter);
    }
    return result;
}
```

Листинг 6: Функция symbolic_multiply

2.2.4 Класс BigFiniteNumber

Отвечает за операции над многоразрядными числами.

Основные поля:
`value` — строка символов (цифр) числа.
`is_negative` — флаг отрицательности числа.

Основные методы: конструкторы, операторы сравнения, арифметические действия, вспомогательные методы нормализации и проверки переполнения.

Метод `normalize`

Вход: строка числа.

Выход: строка без ведущих нулей.

Описание: удаляет ведущие нулевые элементы из строки числа. Если все символы являются нулевыми, возвращается строка с одним нулевым элементом.

Метод `operator+`

Вход: два числа `a` и `b`.

Выход: результат сложения.

Описание: основной метод сложения. Анализирует знаки операндов. Если знаки совпадают, вызывается сложение по разрядам с учетом переноса.

Иначе задача сводится к вычитанию меньшего по модулю числа из большего.

Алгоритм сложения чисел одного знака:
Выравнивание длины операндов добавлением нулевых элементов слева.
Проход по разрядам справа налево.
Для каждого разряда используется таблица сложения с входным переносом.
Сохранение суммы разряда и обновление переноса.
Если после обработки всех разрядов остался перенос, он добавляется как новый старший разряд.

```
BigFiniteNumber BigFiniteNumber::operator+(const BigFiniteNumber&
    other) const{
    if (this->is_negative == other.is_negative){
        int max_len = std::max(
            this->value.length(),
            other.value.length()
        );
        std::string s1 = pad_left(this->value, max_len);
        std::string s2 = pad_left(other.value, max_len);

        std::string result;
        char perenos = get_additive_unit();
        for (int i = max_len - 1; i >= 0; --i){
            auto [sum, new_perenos] = ADDITION_TABLE[
                std::make_tuple(s1[i], s2[i], perenos)
            ];
            result = std::string(1, sum) + result;
            perenos = new_perenos;
        }

        if (perenos != get_additive_unit()){
            result = std::string(1, perenos) + result;
        }

        BigFiniteNumber res =
            BigFiniteNumber::from_internal_string(
                normalize(result),
                this->is_negative
            );
        res.check_overflow();
        return res;
    } else{
        if (this->is_negative){
            BigFiniteNumber positive_this =
                BigFiniteNumber::from_internal_string(
                    this->value, false
                );
            return other - positive_this;
```

```
38          } else{
39              BigFiniteNumber positive_other =
40                  BigFiniteNumber::from_internal_string(
41                      other.value, false
42                  );
43              return *this - positive_other;
44          }
45      }
46  }
```

Листинг 7: Перегрузка operator'а +

Метод `operator-`

Вход: два числа `a` и `b`.

Выход: разность `a - b`.

Описание: реализует вычитание с учетом знаков. Если знаки разные, задача сводится к сложению. Если знаки одинаковые, выполняется вычитание меньшего по модулю из большего с установкой соответствующего знака результата.

Алгоритм вычитания:
Определение большего и меньшего по модулю операндов.
Выравнивание длины операндов.
Проход по разрядам справа налево с учетом заема.
Для каждого разряда проверяется возможность вычитания без заема.
Если вычитание невозможно, выполняется заем из старшего разряда.
Нормализация результата и установка знака.

Метод `operator*`

Вход: два числа `a` и `b`.

Выход: произведение.

Описание: использует алгоритм умножения столбиком. В цикле перебираются разряды второго множителя справа налево. Первый множитель умножается на текущую цифру второго путем многократного сложения. Результат сдвигается влево на номер разряда добавлением нулевых элементов справа. Все частичные произведения суммируются.

```
1  BigFiniteNumber BigFiniteNumber::operator*(const BigFiniteNumber&
       other) const{
```

```
    if (this->value == std::string(1, get_additive_unit()) &&
       other.value == std::string(1, get_additive_unit())){
        throw std::domain_error(INFINITY_INTERVAL_STR);
    }

    if (this->value == std::string(1, get_additive_unit()) ||
       other.value == std::string(1, get_additive_unit())){
        return BigFiniteNumber::from_internal_string(std::string
           (1, get_additive_unit()), false);
    }

    bool result_negative = (this->is_negative != other.
       is_negative);

    BigFiniteNumber result = BigFiniteNumber::
       from_internal_string(std::string(1, get_additive_unit()),
       false);
    BigFiniteNumber multiplicand = BigFiniteNumber::
       from_internal_string(this->value, false);

    for (size_t i = 0; i < other.value.length(); i++){
        int pos = other.value.length() - 1 - i;
        char multiplier_digit = other.value[pos];

        if (multiplier_digit != get_additive_unit()){
            BigFiniteNumber partial_product = BigFiniteNumber::
               from_internal_string(std::string(1,
               get_additive_unit()), false);
            char counter = get_additive_unit();
            while (counter != multiplier_digit){
                partial_product = partial_product + multiplicand;
                counter = next_symbol(counter);
            }

            std::string sdvig_value = partial_product.value;
            for (size_t j = 0; j < i; j++){
                sdvig_value += get_additive_unit();
            }

            BigFiniteNumber sdvig_product = BigFiniteNumber::
               from_internal_string(sdvig_value, false);
            result = result + sdvig_product;
        }
    }

    BigFiniteNumber res = BigFiniteNumber::from_internal_string(
       result.value, result_negative);
    res.check_overflow();
    return res;
}
```

Листинг 8: Перегрузка operator'a *

Метод `divide`

Вход: делимое `a` и делитель `b`.

Выход: пара (частное, остаток).

Описание: реализует деление столбиком. Алгоритм обрабатывает делимое слева направо, добавляя по одному разряду к текущему остатку. Для каждой позиции находится максимальная цифра частного, при умножении на которую делитель не превышает текущий остаток. Найденная цифра добавляется к частному, а произведение вычитается из остатка.

Обработка специальных случаев:
Деление на ноль: если делитель равен нулю и делимое не ноль, возвращается ошибка пустого множества. Если оба нуля, возвращается диапазон всех значений.
Деление отрицательного на положительное: выполняется коррекция частного и остатка для обеспечения неотрицательности остатка.

```
std::pair<BigFiniteNumber, BigFiniteNumber> BigFiniteNumber::
   divide(const BigFiniteNumber& other) const{
    if (other.value == std::string(1, get_additive_unit())){
        if (this->value == std::string(1, get_additive_unit())){
            throw std::domain_error(INFINITY_INTERVAL_STR);
        } else{
            throw std::domain_error(EMPTY_SET_STR);
        }
    }

    if (this->value == std::string(1, get_additive_unit())){
        BigFiniteNumber zero = BigFiniteNumber::
           from_internal_string(std::string(1, get_additive_unit
           ()), false);
        return {zero, zero};
    }

    BigFiniteNumber dividend_abs = BigFiniteNumber::
       from_internal_string(this->value, false);
    BigFiniteNumber divisor_abs = BigFiniteNumber::
       from_internal_string(other.value, false);

    if (dividend_abs < divisor_abs){
        if (this->is_negative && !other.is_negative){
            BigFiniteNumber minus_one = BigFiniteNumber::
               from_internal_string(std::string(1,
               get_multiplicative_unit()), true);
            BigFiniteNumber remainder = divisor_abs -
               dividend_abs;
```

```
            return {minus_one, remainder};
        } else{
            BigFiniteNumber zero = BigFiniteNumber::
               from_internal_string(std::string(1,
               get_additive_unit()), false);
            return {zero, dividend_abs};
        }
    }

    std::string quotient_str;
    BigFiniteNumber current_remainder = BigFiniteNumber::
       from_internal_string(std::string(1, get_additive_unit()),
       false);

    for (size_t i = 0; i < dividend_abs.value.length(); i++){
        current_remainder = BigFiniteNumber::from_internal_string
           (
            current_remainder.value + std::string(1, dividend_abs
               .value[i]), false
        );
        current_remainder.value = normalize(current_remainder.
           value);

        char quotient_digit = get_additive_unit();
        for (char test_digit = get_multiplicative_unit();
           test_digit != get_additive_unit(); test_digit =
           next_symbol(test_digit)){
            BigFiniteNumber test_product = divisor_abs *
               BigFiniteNumber::from_internal_string(std::string
               (1, test_digit), false);
            if (test_product > current_remainder) break;
            quotient_digit = test_digit;
        }
        quotient_str += quotient_digit;
        if (quotient_digit != get_additive_unit()){
            BigFiniteNumber product = divisor_abs *
               BigFiniteNumber::from_internal_string(std::string
               (1, quotient_digit), false);
            current_remainder = current_remainder - product;
        }
    }

    BigFiniteNumber quotient_abs = BigFiniteNumber::
       from_internal_string(normalize(quotient_str), false);

    bool quotient_negative = (this->is_negative != other.
       is_negative);
    BigFiniteNumber quotient = BigFiniteNumber::
       from_internal_string(quotient_abs.value, quotient_negative
       );
    BigFiniteNumber remainder = current_remainder;

```

```
57      if (this->is_negative && !other.is_negative && remainder.
           value != std::string(1, get_additive_unit())){
58          BigFiniteNumber one = BigFiniteNumber::
               from_internal_string(std::string(1,
               get_multiplicative_unit()), false);
59          quotient = quotient - one;
60          remainder = divisor_abs - remainder;
61      }
62
63      remainder.is_negative = false;
64
65      return {quotient, remainder};
66  }
```

Листинг 9: Действие деление

2.3 Тестирование

Программа включает набор тестов для проверки математических свойств кольца.

Тест коммутативности сложения

Проверяет выполнение равенства $a+b = b+a$ для всех пар тестовых чисел.

Тест ассоциативности сложения

Проверяет выполнение равенства $(a+b)+c = a+(b+c)$ для трех тестовых чисел.

Тест дистрибутивности

Проверяет выполнение равенства $a*(b+c) = a*b+a*c$ для тройки чисел.

Тест свойства умножения на ноль

Проверяет выполнение равенства $x * a = a$ для различных значений x.

Тест деления

Проверяет корректность деления с остатком, включая случай деления отрицательного числа на положительное.

3 Результаты работы программы

3.1 Запуск калькулятора

После запуска программы командой `./calculator` пользователю отображается начальный экран с информацией о системе и доступными операциями (см. рисунок 1).

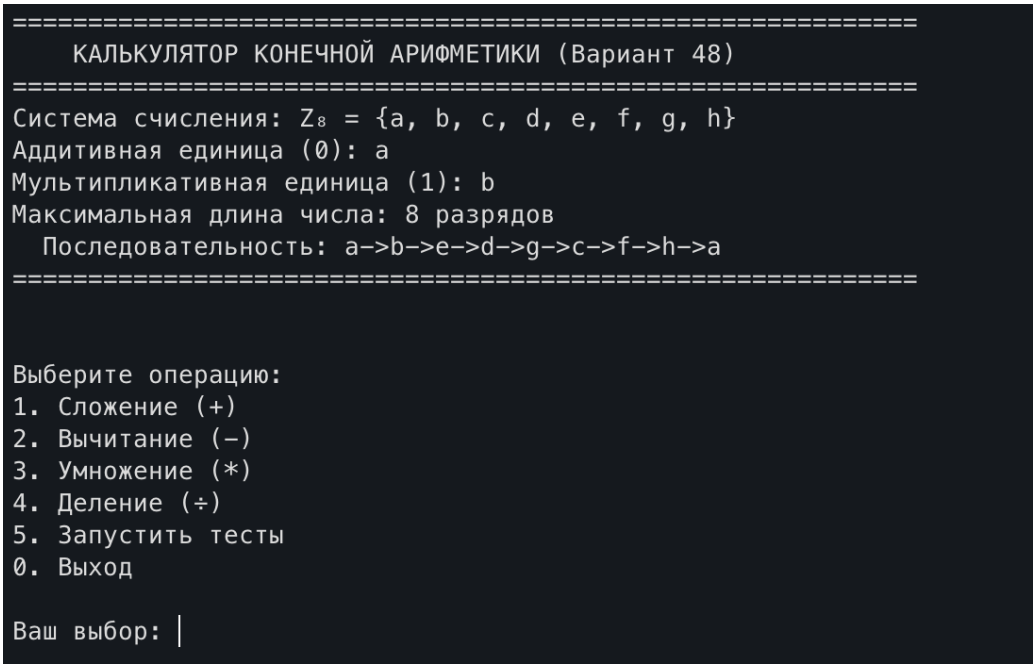

Рис. 1: Начальное меню

На начальном экране указаны:
Используемая система счисления и множество элементов.
Значения аддитивной и мультипликативной единиц.
Максимальная длина чисел в разрядах.
Правило инкремента для данного варианта.
Список доступных операций с номерами для выбора.

3.2 Выполнение операций

Пользователь вводит номер операции, затем два числа. Программа вычисляет результат и выводит его на экран (см. рисунок 2).

Пример сложения: $b + e = d$

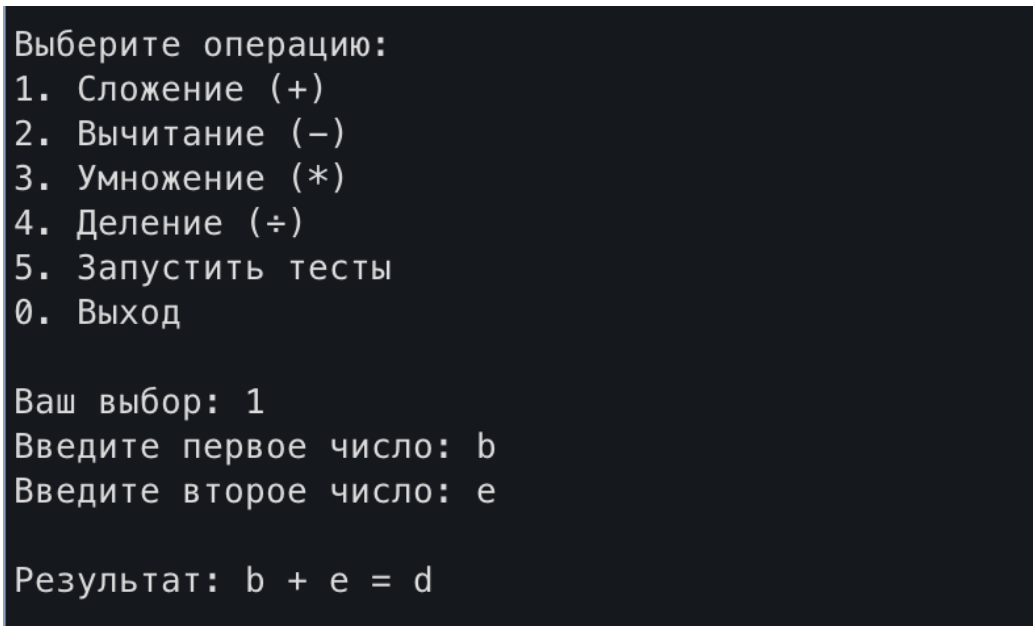

Рис. 2: Действие сложения

Пользователь выбирает операцию 1, вводит первое число `b` и второе число `e`.

Пример сложения с отрицательными числами: $-b + f = c$

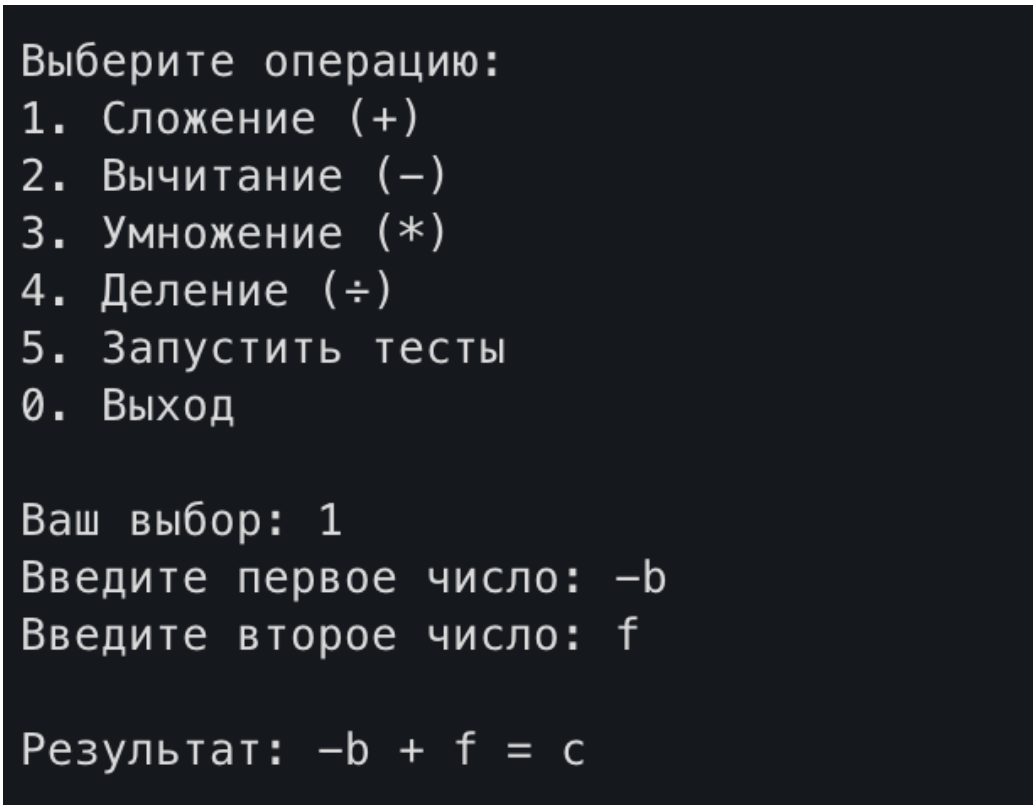

Рис. 3: Действие сложения с отрицательными значениями

Программа корректно обрабатывает отрицательные числа (см. рисунок. 3). Минус указывается перед числом.

Пример вычитания: $f - c = b$

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 2
Введите первое число: f
Введите второе число: c

Результат: f - c = b
```

Рис. 4: Действие вычитания

Вычитание реализовано через сложение с противоположным элементом (см. рисунок. 4).

Пример умножения: $c * e = be$

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 3
Введите первое число: c
Введите второе число: e

Результат: c * e = be
```

Рис. 5: Действие умножения

Умножение выполняется через повторное сложение с учетом разрядности результата (см. рисунок. 5).

Пример деления: $h \div c = b(e)$

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 4
Введите первое число: h
Введите второе число: c

Результат: h ÷ c = b(e)
```

Рис. 6: Действие деления

При делении программа выводит частное и остаток в скобках. Остаток всегда неотрицательный (см. рисунок 6).

Пример деления отрицательного числа на положительное: $-h \div c = -e(d)$

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 4
Введите первое число: -h
Введите второе число: c

Результат: -h ÷ c = -e(d)
```

Рис. 7: Действие деления отрицательного на положительное

Программа корректно выполняет коррекцию частного и остатка при делении отрицательного числа на положительное (см. рисунок 7).

3.2.1 Необычные случаи

Пример деления аддитивной единицы на аддитивную: $a \div a$

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 4
Введите первое число: a
Введите второе число: a

Результат: a ÷ a = [-hhhhhhhh, hhhhhhhh]
```

Рис. 8: Обрабатываемый случай деления

Программа корректно выполняет деление и выдает правильный результат при операндах аддитивных единиц (см. рисунок 8).

Пример деления аддитивной единицы на любой другой символ нашей системы счисления: $a \div b$ и $a \div h$

```
Ваш выбор: 4
Введите первое число: a
Введите второе число: b

Результат: a ÷ b = a(a)

Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 4
Введите первое число: a
Введите второе число: h

Результат: a ÷ h = a(a)
```

Рис. 9: Случай деления «a» на «x»

Программа дает корректный результат (см. рисунок 9), а именно: «a», что и требовалось ожидать.

Пример умножения аддитивной единица на любой символ: $a * x = a$
Этот пункт как раз должен выполняться, так как он учитывался в составленном задании к курсовой работе (см. рисунок 10).

```
Ваш выбор: 3
Введите первое число: a
Введите второе число: c

Результат: a * c = a

Выберите операцию:
1. Сложение (+)
2. Вычитание (-)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 3
Введите первое число: a
Введите второе число: g

Результат: a * g = a
```

Рис. 10: Случай умножения «a» на «x»

Показанный результат дает понять, что свойство: для любого x | x * a = a

3.3 Запуск тестов

Для запуска встроенных тестов пользователь выбирает операцию 5. Результат показан на рисунке 11.

Программа последовательно проверяет все математические свойства кольца:

Проверка правила «+1» для всех элементов.

Коммутативность сложения.

Ассоциативность сложения.

Коммутативность умножения.

Ассоциативность умножения.

Дистрибутивность умножения относительно сложения.

Свойство умножения на нулевой элемент.

Для каждого теста выводится проверяемое выражение и результат проверки.

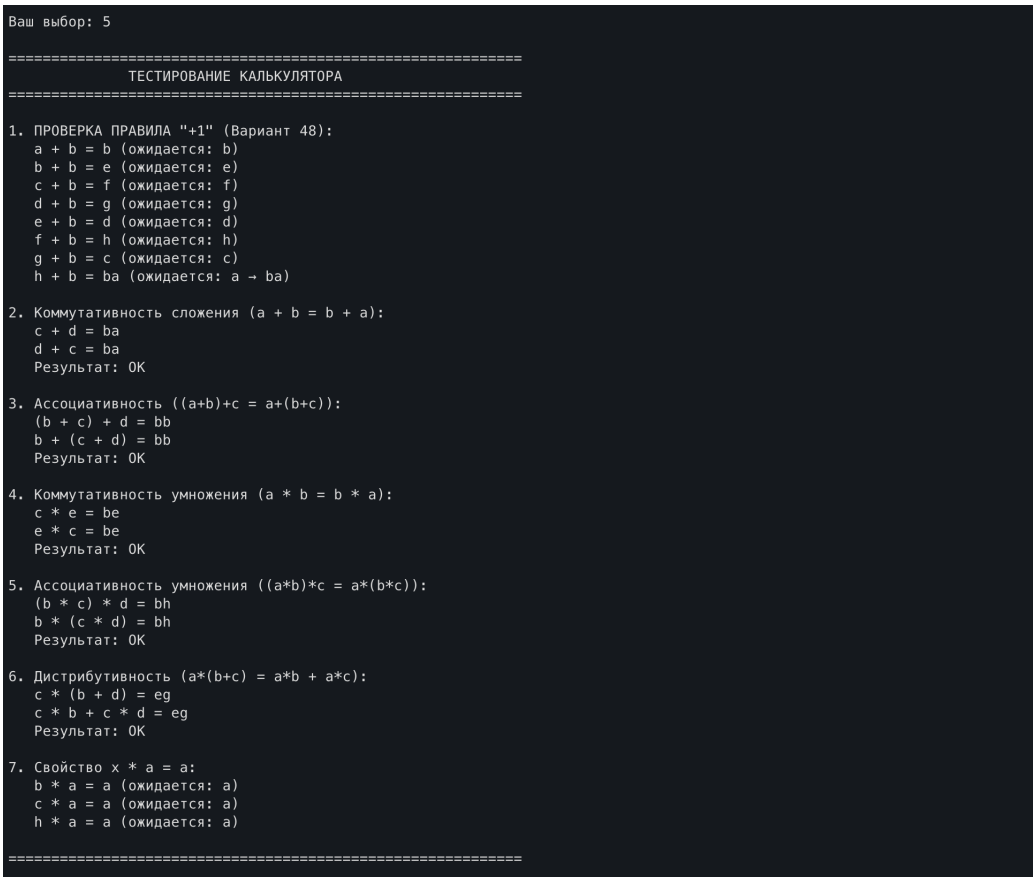

Рис. 11: Все тесты

3.4 Обработка ошибок

Программа корректно обрабатывает некорректный ввод и выводит соответствующие сообщения об ошибках.

Деление на ноль

При попытке деления ненулевого числа на ноль программа выводит сообщение о пустом множестве (см. рисунок 12).

Пример: $c \div a = \{\}$

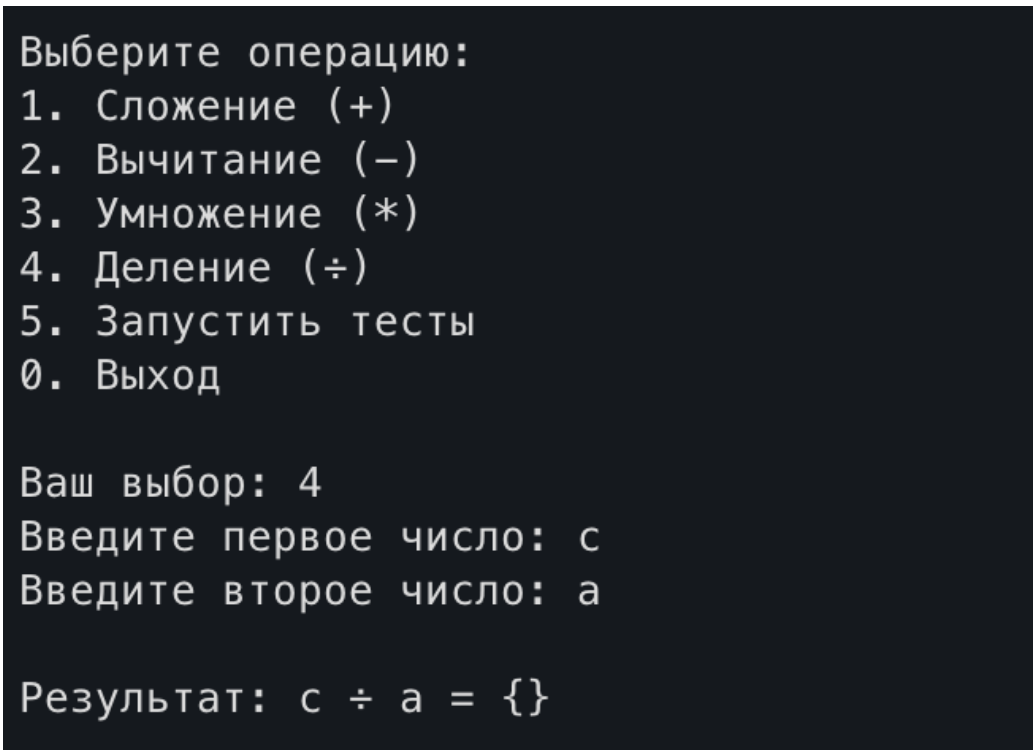

Рис. 12: Деление ненулевого числа на аддитивную единицу

Переполнение

Если результат операции превышает максимальное количество разрядов, программа выводит сообщение о переполнении (см. рисунок 13).

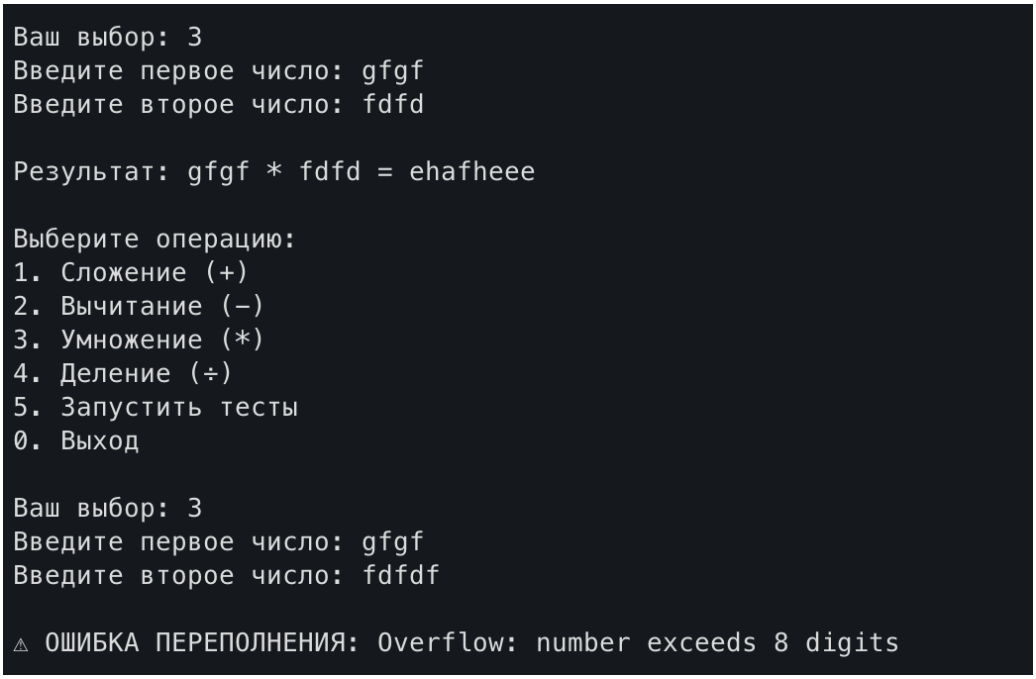

Рис. 13: Переполнение

На скриншоте явно показан пример близкий к переполнению, и что меняется, чтоб мы вышли за пределы 8 символов.

Некорректный символ

При вводе символа, не принадлежащего кольцу, программа выводит сообщение об ошибке и заменяет число на нулевое значение (см. рисунок 14).

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (−)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 1
Введите первое число: k
Введите второе число: j
ERROR: Invalid character 'k' in input 'k'. Defaulting to 'a'.
ERROR: Invalid character 'j' in input 'j'. Defaulting to 'a'.

Результат: a + a = a
```

Рис. 14: Ошибка: некорректный символ

Некорректный ввод в меню

При вводе некорректного номера действия программа выводит сообщение об ошибке и запрашивает ввод повторно (см. рисунок 15).

```
Выберите операцию:
1. Сложение (+)
2. Вычитание (−)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор: 99
Неверный выбор! Введите число от 0 до 5.

Выберите операцию:
1. Сложение (+)
2. Вычитание (−)
3. Умножение (*)
4. Деление (÷)
5. Запустить тесты
0. Выход

Ваш выбор:
```

Рис. 15: Ошибка: некорректный номер

3.5 Сборка проекта

Сборка проекта выполняется через Makefile. Команды для сборки и запуска:

```
# Сборка проекта
make

# Запуск калькулятора
make run

# Очистка файлов
make clean

# Пересборка
make rebuild
```

После выполнения команды `make` создается исполняемый файл `calculator`. Результат виден на изображении 16.

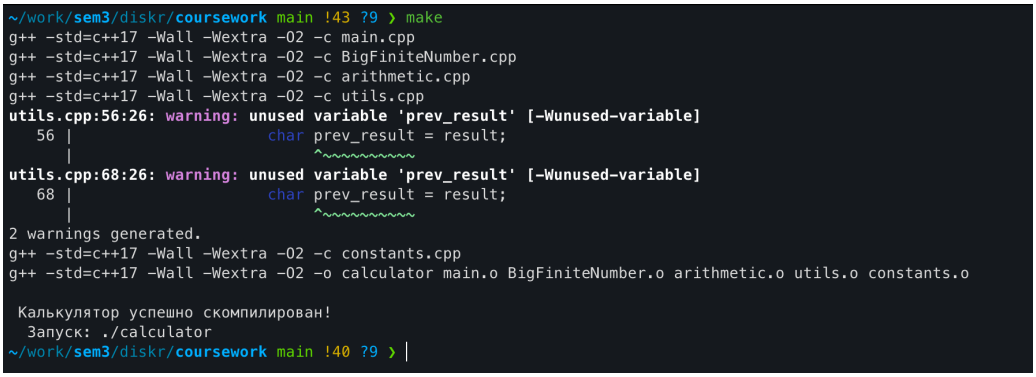

Рис. 16: Сборка проекта

Заключение

В ходе курсовой работы разработан калькулятор большой конечной арифметики $\langle Z_8^8; +, * \rangle$ на основе малой конечной арифметики $\langle Z_8; +, * \rangle$. Калькулятор поддерживает четыре действия: сложение, вычитание, умножение и деление с остатком.

Реализованная функциональность

Архитектура программы состоит из четырех модулей: constants для хранения констант кольца, utils для таблиц действий, arithmetic для базовых действий над символами, BigFiniteNumber для действий над многоразрядными числами.

Система таблиц действий построена на основе правила «+1» для варианта 48. Таблица сложения содержит 512 записей для всех комбинаций операндов и переносов.

Интерфейс позволяет выполнять действия через меню и запускать тесты проверки кольца.

Тестирование проверяет коммутативность сложения и умножения, ассоциативность сложения и умножения, дистрибутивность умножения относительно сложения, свойство умножения на нулевой элемент.

Деление реализовано алгоритмом деления столбиком с вычислением частного и остатка. Остаток всегда неотрицательный.

Особенности реализации

Малая арифметика реализует действия над односимвольными элементами. Все действия построены на базовом действии «+1».

Большая арифметика работает с многоразрядными числами. Сложение и вычитание выполняются поразрядно с управлением переносами и заемами.

Отрицательные числа обрабатываются через анализ знаков операндов. Действия сводятся к работе с абсолютными значениями с последующей установкой знака результата.

Деление отрицательного числа на положительное выполняет коррекцию частного и остатка для соблюдения условия неотрицательности остатка.

Недостатки программы

Умножение и деление имеют сложность $O(N \cdot M)$ из-за реализации через многократное сложение. Для чисел размера 8 разрядов это приводит к выполнению до 64 действий сложения на одно умножение.

Максимальная длина чисел фиксирована в коде константой MAX_DIGITS. Изменение требует перекомпиляции программы.

Таблица сложения занимает 512 записей в памяти. При увеличении размера кольца память растет кубически.

Масштабируемость

Программа может быть расширена добавлением действий возведения в степень, нахождения НОД, вычисления НОК через НОД.

Возможно увеличение максимальной длины чисел через параметризацию константы MAX_DIGITS и динамическое выделение памяти для результатов действий.

Интерфейс может быть дополнен графическим режимом работы (GUI) для удобства использования.

Список литературы

1. Сайт кафедры с учебными материалами по курсу *«Дискретная математика»*. URL: https://tema.spbstu.ru/dismath/ (дата обращения: 8.12.2025).

2. Новиков Ф.А. *Дискретная математика*. Учебник. – 384 с. Ссылка на PDF: https://stugum.wordpress.com/wp-content/uploads/2014/03/novikov.pdf (дата обращения: 10.12.2025).

3. «Большая» конечная арифметика. Ссылка на сайт: https://studfile.net/preview/9751743/page:22/ (дата обращения: 10.12.2025).