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 ЛАБОРАТОРНОЙ РАБОТЕ №1

по дисциплине Дискретная математика

Калькулятор «большой» конечной арифметики

Вариант 1

Студент,
группы 5130201/40003 _____ Адиатуллин Т.Р

Доцент
_____ Востров А.В

«____»_____ 2025 г.

Санкт-Петербург, 2026

Содержание

Введение

Курсовая работа посвящена разработке калькулятора большой конечной арифметики $\langle Z_8; +, * \rangle$ для четырёх арифметических операций: сложения, вычитания, умножения и деления. Калькулятор построен на основе малой конечной арифметики, в которой задано правило ”+1” и выполняются свойства коммутативности сложения и умножения, ассоциативности этих операций, дистрибутивности умножения относительно сложения. В системе заданы аддитивная единица «a» и мультипликативная единица «b», а также выполняется свойство $x \times a = a$ для любого элемента x.

Правило ”+1” определяет переход от текущего символа к следующему и задано в соответствии с таблицей 1.

Таблица 1: Правило ”+1”

x	a	b	c	d	e	f	g	h
$x+1$	b	c	e	g	d	h	f	a

1. Математическое описание

1.1. «Малая» и «большая» конечные арифметики

Множество Z вместе с набором операций $\Sigma = \{\varphi_1, \ldots, \varphi_m\}$, $\varphi_i : Z^{n_i} \to Z$, где n_i - арность операции φ_i, называется алгебраической структурой, универсальной алгеброй или просто алгеброй.

Коммутативное кольцо с единицей - алгебраическая структура $\langle Z; +, * \rangle$, в которой выполняются следующие аксиомы:

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. Существование противоположного элемента: $\forall a \in Z \ \exists (-a) \in Z : a + (-a) = 0$
4. Коммутативность сложения: $a + b = b + a$
5. Ассоциативность умножения: $(a * b) * c = a * (b * c)$
6. Дистрибутивность: $a * (b + c) = a * b + a * c$
7. Коммутативность умножения: $a * b = b * a$
8. Существование единичного элемента: $\exists 1 \in Z \ (\forall a \in Z : a * 1 = 1 * a = a)$

«Малая» конечная арифметика - конечное коммутативное кольцо с единицей $\langle Z_8; +, * \rangle$, на котором определены действия вычитания и деления, причём деление определено частично (только для элементов, имеющих мультипликативную инверсию).

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

- Нулевой элемент (нейтральный по сложению): a
- Единичный элемент (нейтральный по умножению): b
- Отношение порядка (правило инкремента «+1»):
  $a \to b \to c \to e \to d \to g \to f \to h$

«Большая» конечная арифметика - конечное коммутативное кольцо с единицей $\langle Z_i^n; +, * \rangle$, элементами которого являются слова длины до n над алфавитом Z_i. Операции определены позиционно с учётом переноса разрядов. Деление определено с остатком.

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

Каждому символу сопоставлен индекс (позиционное значение):

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

1.2. Таблицы операций малой арифметики

На рисунках 1 и 2 представлены таблицы операций и переносов для сложения и умножения соответственно

	a	b	c	d	e	f	g	h
a	a	b	c	d	e	f	g	h
b	b	c	e	g	d	h	f	a
c	c	e	d	f	g	a	h	b
d	d	g	f	a	h	c	b	e
e	e	d	g	h	f	b	a	c
f	f	h	a	c	b	d	e	g
g	g	f	h	b	a	e	c	d
h	h	a	b	e	c	g	d	f

	a	b	c	d	e	f	g	h
a	a	a	a	a	a	a	a	a
b	a	a	a	a	a	a	a	b
c	a	a	a	a	a	b	a	b
d	a	a	a	b	a	b	b	b
e	a	a	a	a	a	b	b	b
f	a	a	b	b	b	b	b	b
g	a	a	a	b	b	b	b	b
h	a	b	b	b	b	b	b	b

Рис. 1: Таблица сложения и таблица переносов для малой арифметики

	a	b	c	d	e	f	g	h
a	a	a	a	a	a	a	a	a
b	a	b	c	d	e	f	g	h
c	a	c	d	a	f	d	c	f
d	a	d	a	a	d	a	d	d
e	a	e	f	d	b	c	h	g
f	a	f	d	a	c	d	f	c
g	a	g	c	d	h	f	b	e
h	a	h	f	d	g	c	e	b

	a	b	c	d	e	f	g	h
a	a	a	a	a	a	a	a	a
b	a	a	a	a	a	a	a	a
c	a	a	a	b	a	b	b	b
d	a	a	b	c	b	e	c	e
e	a	a	a	b	b	c	b	c
f	a	a	b	e	c	d	e	g
g	a	a	b	c	b	e	e	d
h	a	a	b	e	c	g	d	f

Рис. 2: Таблица умножения и таблица переносов для малой арифметики

2. Особенности реализации

2.1. Структура проекта

Проект имеет следующую структуру:

- `SmallArithmetic.hpp` и `SmallArithmetic.cpp`: содержат реализацию малой конечной арифметики на основе диаграммы Хассе. Класс выполняет операции над однозначными элементами алфавита.
- `BigArithmeticCalc.hpp` и `BigArithmeticCalc.cpp`: содержат реализацию большой конечной арифметики. Класс выполняет операции столбиком над многозначными числами.
- `UI.hpp` и `UI.cpp`: реализуют пользовательский интерфейс, обработку команд и арифметических выражений.
- `config.hpp`: содержит конфигурацию системы - мощность алфавита, правило ”+1” и сам алфавит.
- `main.cpp`: точка входа в программу.
- `CMakeLists.txt`: автоматизация сборки проекта.

2.2. Структуры данных

Для реализации конечной арифметики используются два класса: `SmallArithmetic` и `BigArithmeticCalc`.

2.2.1. Класс SmallArithmetic

Класс хранит следующие основные структуры:

- `map<string, string> plusOneRule`: правило перехода к следующему элементу алфавита.
- `vector<string> alphabet`: набор всех элементов алфавита.
- `map<string, string> inverseMap`: таблица обратных элементов для деления.
- `map<tuple<string, string, string>, pair<string, string>> additionTableWithCarry`: таблица сложения с переносом для операций столбиком.
- `vector<vector<string>> addTable, mulTable, subTable, divTabl` таблицы операций малой арифметики.

- `int N`: мощность алфавита.
- `static const int MAX_DIGITS = 8`: максимальное количество разрядов в числах.

2.2.2. Класс BigArithmeticCalc

Класс содержит:

- `SmallArithmetic small`: объект малой арифметики для выполнения посимвольных операций.

2.3. Класс SmallArithmetic

2.3.1. Назначение и структура данных

Класс `SmallArithmetic` реализует малую конечную арифметику на заданном алфавите с правилом перехода к следующему элементу (правило "+1"). Все операции выполняются через построение диаграммы Хассе.

```
class SmallArithmetic {
private:
    int N;
    map<string, string> plusOneRule;
    vector<string> alphabet;

    map<string, string> inverseMap;

    map<tuple<string, string, string>, pair<string, string>>
   additionTableWithCarry;
    map<string, string> negationMap;

    vector<vector<string>> addTable;
    vector<vector<string>> mulTable;
    vector<vector<string>> subTable;
    vector<vector<string>> divTable;

    string additiveIdentity;
    string multiplicativeIdentity;
    string universum;
    string emptySet;

    static const int MAX_DIGITS = 8;

    string addByHasse(const string& a, const string& b) const;
    string multiplyByHasse(const string& a, const string& b) const;
    string subtractByHasse(const string& a, const string& b) const;
    string divideByHasse(const string& a, const string& b) const;

    optional<string> findMultiplicativeInverse(const string& x);
    void buildInverseMap();
```

```
31
32      void buildAddTable();
33      void buildMulTable();
34      void buildSubTable();
35      void buildDivTable();
36
37      void buildAdditionTableWithCarry();
38      bool checkCarry(const string& start, const string& steps)
        const;
39      void buildNegationMap();
40
41      string nextSymbol(const string& current) const;
42      int compareSymbols(const string& a, const string& b) const;
43
44      void printTable(const vector<vector<string>>& table) const;
45
46  public:
47      SmallArithmetic(int n,
48                      const map<string, string>& rule,
49                      const vector<string>& alph,
50                      const string& addId = "a",
51                      const string& mulId = "b");
52
53      const vector<string>& getAlphabet() const;
54      int getN() const;
55
56      const string& getAdditiveIdentity() const;
57      const string& getMultiplicativeIdentity() const;
58      const string& getUniversum() const;
59      const string& getEmptySet() const;
60      int getMaxDigits() const;
61      string getMinNumber() const;
62      string getMaxNumber() const;
63
64      bool isValidElement(const string& elem) const;
65      int compareElems(const string& a, const string& b) const;
66      string nextElem(const string& current) const;
67      pair<string, string> addWithCarry(const string& c1,
68                                        const string& c2,
69                                        const string& carry_in)
        const;
70      string smallSubtract(const string& a, const string& b) const;
71
72      void printAddTable() const;
73      void printMulTable() const;
74      void printSubTable() const;
75      void printDivTable() const;
76      void printAllTables() const;
77      void printHasseDiagram() const;
78  };
```

Listing 1: Интерфейс класса SmallArithmetic

Ключевые методы

2.3.2. Метод nextSymbol(const string& current)

Назначение: Возвращает следующий элемент алфавита согласно правилу ”+1”. Используется для обхода диаграммы Хассе при выполнении операций сложения и умножения.

Вход:

- `const string& current`: Текущий символ алфавита.

Выход:

- `string`: Следующий символ по правилу ”+1”.

```
string SmallArithmetic::nextSymbol(const string& current) const {
    auto it = plusOneRule.find(current);
    if (it == plusOneRule.end()) {
        throw runtime_error("invalid symbol: " + current);
    }
    return it->second;
}
```

Listing 2: Реализация nextSymbol()

2.3.3. Метод compareSymbols(const string& a, const string& b)

Назначение: Сравнивает два элемента алфавита, определяя их относительный порядок в диаграмме Хассе.

Вход:

- `const string& a`: Первый элемент для сравнения.
- `const string& b`: Второй элемент для сравнения.

Выход:

- `int`: 0 (если $a = b$), -1 (если $a < b$), 1 (если $a > b$).

```
int SmallArithmetic::compareSymbols(const string& a, const string&
    b) const {
    if (a == b) return 0;

    string current = additiveIdentity;
    while (true) {
        if (current == a) return -1;
        if (current == b) return 1;
        current = nextSymbol(current);
        if (current == additiveIdentity) break;
    }
    return 0;
}
```

Listing 3: Реализация compareSymbols()

Алгоритм последовательно проходит по циклу правила "+1", начиная с нейтрального элемента, и определяет, какой из элементов встречается раньше.

2.3.4. Метод addByHasse(const string& a, const string& b)

Назначение: Выполняет сложение двух элементов алфавита путём последовательного применения правила "+1". Результат вычисляется как $a + b = \underbrace{(a + 1) + 1 \ldots + 1}_{b}$.

Вход:

- `const string& a`: Первое слагаемое.
- `const string& b`: Второе слагаемое.

Выход:

- `string`: Результат сложения $a + b$.

```
string SmallArithmetic::addByHasse(const string& a, const string&
   b) const {
    if (b == additiveIdentity) return a;

    string counter = additiveIdentity;
    string result = a;

    while (counter != b) {
        result = nextSymbol(result);
        counter = nextSymbol(counter);
    }

    return result;
}
```

Listing 4: Реализация addByHasse()

Алгоритм работает следующим образом:

1. Если второе слагаемое является нейтральным элементом, возвращается первое слагаемое;
2. Инициализируется счётчик нейтральным элементом;
3. В цикле счётчик увеличивается до значения второго слагаемого;
4. Одновременно результат увеличивается на ту же величину;
5. Возвращается итоговое значение.

2.3.5. Метод multiplyByHasse(const string& a, const string& b)

Назначение: Выполняет умножение двух элементов алфавита через повторное сложение. Результат вычисляется как $a \times b = \underbrace{a + a + \ldots + a}_{b}$.

Вход:

- `const string& a`: Множитель.
- `const string& b`: Множитель.

Выход:

- `string`: Результат умножения $a \times b$.

```
string SmallArithmetic::multiplyByHasse(const string& a, const
   string& b) const {
    // специальный случай: a * a = универсум
    if (a == additiveIdentity && b == additiveIdentity) {
        return universum;
    }

    if (a == additiveIdentity || b == additiveIdentity) {
        return additiveIdentity;
    }

    string counter = additiveIdentity;
    string result = additiveIdentity;

    // буквенный счетчик
    while (counter != b) {
        result = addByHasse(result, a);
        counter = nextSymbol(counter);
    }

    return result;
}
```

Listing 5: Реализация multiplyByHasse()

Алгоритм:

1. Обработка краевого случая: произведение двух нейтральных элементов по сложению даёт универсум;
2. Если хотя бы один множитель - нейтральный элемент, результат - нейтральный элемент;
3. Иначе выполняется повторное сложение множимого с самим собой b раз.

2.3.6. Метод subtractByHasse(const string& a, const string& b)

Назначение: Выполняет вычитание путём поиска такого элемента c, что $b + c = a$. Реализует обратную операцию к сложению.

Вход:

- `const string& a`: Уменьшаемое.
- `const string& b`: Вычитаемое.

Выход:

- `string`: Результат вычитания $a - b$ или строка ошибки.

```
string SmallArithmetic::subtractByHasse(const string& a, const
   string& b) const {
    for (const auto& candidate : alphabet) {
        if (addByHasse(b, candidate) == a) {
            return candidate;
        }
    }
    return "ERR:  with subtract";
}
```

Listing 6: Реализация subtractByHasse()

Алгоритм перебирает все элементы алфавита и проверяет, какой из них при сложении с вычитаемым даёт уменьшаемое.

2.3.7. Метод buildAdditionTableWithCarry()

Назначение: Строит таблицу для сложения с учётом входящего переноса. Таблица используется в большой арифметике при сложении столбиком. Для каждой тройки (a, b, c_{in}) вычисляется пара (sum, c_{out}).

Вход:

- vector<string> alphabet

Выход:

- map<tuple<string, string, string>, pair<string, string» additionTableWithCarry - таблица для сложения с учётом входящего переноса.

```
void SmallArithmetic::buildAdditionTableWithCarry() {
    for (const auto& c1 : alphabet) {
        for (const auto& c2 : alphabet) {
            for (const auto& carry_in : {additiveIdentity,
   multiplicativeIdentity}) {
                string sum1 = addByHasse(c1, c2); // сумма без
   переноса
                string final_sum = addByHasse(sum1, carry_in); //
   финальная сумма с учетом переноса

                bool carry1 = checkCarry(c1, c2); // проверяем
   перенос c1 + c2
                bool carry2 = checkCarry(sum1, carry_in); //
   проверяем перенос sum1 + carry_in

                string carry_out = (carry1 || carry2) ?
   multiplicativeIdentity : additiveIdentity; // выставляем перенос

                additionTableWithCarry[make_tuple(c1, c2,
   carry_in)] = make_pair(final_sum, carry_out);
            }
        }
    }
}
```

Listing 7: Реализация buildAdditionTableWithCarry()

Таблица содержит тройки (a, b, c_{in}) и соответствующие им пары (sum, c_{out}), где:

- a, b - слагаемые разрядов;
- c_{in} - входящий перенос из младшего разряда;
- sum - результат сложения в текущем разряде;
- c_{out} - исходящий перенос в старший разряд.

2.4. Класс BigArithmeticCalc

2.4.1. Назначение и структура данных

Класс `BigArithmeticCalc` реализует большую конечную арифметику - операции над многозначными числами. Внутри использует класс `SmallArithmetic` для посимвольных операций.

```
class BigArithmeticCalc {
private:
    SmallArithmetic small;

    bool isNegative(const string& num) const;
    string removeSign(const string& num) const;
    string addSign(const string& num, bool negative) const;

    bool isValidNumber(const string& num) const;

    string deleteTrashZeros(const string& num) const;

    bool isOverflow(const string& num) const;

    string addBigUnsigned(const string& a, const string& b) const;
    string subtractBigUnsigned(const string& a, const string& b)
   const;
    string multiplyBigUnsigned(const string& a, const string& b)
   const;
    pair<string, string> divideBigUnsigned(const string& a, const
   string& b) const;

    string multiplyByDigit(const string& num, const string& digit)
   const;

    int compareBigUnsigned(const string& a, const string& b) const;

public:
    BigArithmeticCalc(int n,
                      const map<string, string>& rule,
                      const vector<string>& alph,
                      const string& addId = "a",
                      const string& mulId = "b");

    const vector<string>& getAlphabet() const;
    string getMinNumber() const;
    string getMaxNumber() const;

    string add(const string& a, const string& b) const;
    string multiply(const string& a, const string& b) const;
    string subtract(const string& a, const string& b) const;
    string divide(const string& a, const string& b) const;

    void printAddTable() const;
    void printMulTable() const;
```

```
42      void printSubTable() const;
43      void printDivTable() const;
44      void printAllTables() const;
45      void printInfo() const;
46      void printHasseDiagram() const;
47      void printHelp() const;
48  };
```

Listing 8: Интерфейс класса BigArithmeticCalc

Класс хранит только один объект малой арифметики, через который выполняются все операции с разрядами.

Вспомогательные методы

2.4.2. Метод isNegative(const string& num)

Назначение: Проверяет, является ли число отрицательным (начинается ли строка с символа минус).

Вход:

- `const string& num`: Число для проверки.

Выход:

- `bool`: `true`, если число отрицательное, иначе `false`.

```
1  bool BigArithmeticCalc::isNegative(const string& num) const {
2      return ! num.empty() && num[0] == '-';
3  }
```

Listing 9: Реализация isNegative()

2.4.3. Метод removeSign(const string& num)

Назначение: Удаляет знак минус из начала строки, возвращая абсолютное значение числа.

Вход:

- `const string& num`: Число (возможно, с минусом).

Выход:

- `string`: Число без знака.

```
1  string BigArithmeticCalc::removeSign(const string& num) const {
2      if (isNegative(num)) {
3          return num.substr(1);
4      }
5      return num;
6  }
```

Listing 10: Реализация removeSign()

2.4.4. Метод addSign(const string& num, bool negative)

Назначение: Добавляет знак минус к числу, если указан флаг отрицательности. Нейтральный элемент по сложению всегда возвращается без знака.

Вход:

- `const string& num`: Число без знака.
- `bool negative`: Флаг отрицательности.

Выход:

- `string`: Число со знаком (если `negative = true`) или без знака.

```
string BigArithmeticCalc::addSign(const string& num, bool
   negative) const {
    if (num == small.getAdditiveIdentity()) return num;
    return negative ? "-" + num : num;
}
```

Listing 11: Реализация addSign()

2.4.5. Метод isValidNumber(const string& num)

Назначение: Проверяет, что все символы числа принадлежат алфавиту системы.

Вход:

- `const string& num`: Число для проверки.

Выход:

- `bool`: `true`, если все символы валидны, иначе `false`.

```
bool BigArithmeticCalc::isValidNumber(const string& num) const {
    if (num.empty()) return false;

    string unsign_num = removeSign(num);

    for (char c : unsign_num) {
        if (!small.isValidElement(string(1, c))) return false;
    }
    return true;
}
```

Listing 12: Реализация isValidNumber()

2.4.6. Метод isOverflow(const string& num)

Назначение: Проверяет, не превышает ли количество разрядов числа максимально допустимое значение (MAX_DIGITS = 8).

Вход:

- `const string& num`: Число для проверки.

Выход:

- `bool`: `true`, если произошло переполнение, иначе `false`.

```
bool BigArithmeticCalc::isOverflow(const string& num) const {
    string unsign_num = removeSign(num);
    string withoutZeros = deleteTrashZeros(unsign_num);
    return withoutZeros.length() >
   static_cast<size_t>(small.getMaxDigits());
}
```

Listing 13: Реализация isOverflow()

2.4.7. Метод deleteTrashZeros(const string& num)

Назначение: Удаляет ведущие нули из числа. Если число состоит только из нулей, возвращает нейтральный элемент по сложению.

Вход:

- `const string& num`: Число с возможными ведущими нулями.

Выход:

- `string`: Число без ведущих нулей.

```
string BigArithmeticCalc::deleteTrashZeros(const string& num)
    const {
    if (num.empty()) return small.getAdditiveIdentity();

    bool negative = isNegative(num);
    string unsign_num = removeSign(num);

    size_t firstNonZero = 0;
    while (firstNonZero < unsign_num.length() &&
           string(1, unsign_num[firstNonZero]) ==
    small.getAdditiveIdentity()) {
        firstNonZero++;
    }

    if (firstNonZero == unsign_num.length()) {
        return small.getAdditiveIdentity();
    }

    string result = unsign_num.substr(firstNonZero);

    return addSign(result, negative);
}
```

Listing 14: Реализация deleteTrashZeros()

Алгоритм:

1. Удаляется знак (если есть);
2. Подсчитывается количество ведущих нулей;
3. Если все символы - нули, возвращается нейтральный элемент;
4. Иначе возвращается подстрока без ведущих нулей со знаком (если необходимо).

2.4.8. Метод compareBigUnsigned(const string& a, const string& b)

Назначение: Сравнивает два беззнаковых многозначных числа.
Вход:

- `const string& a`: Первое число.
- `const string& b`: Второе число.

Выход:

- `int`: 0 (если $a = b$), 1 (если $a > b$), -1 (если $a < b$).

```
int BigArithmeticCalc::compareBigUnsigned(const string& a, const
   string& b) const {
    string na = deleteTrashZeros(a);
    string nb = deleteTrashZeros(b);

    if (na.length() != nb.length()) {
        return na.length() > nb.length() ? 1 : -1;
    }

    for (size_t i = 0; i < na.length(); ++i) {
        int cmp = small.compareElems(string(1, na[i]), string(1,
   nb[i]));
        if (cmp != 0) return cmp;
    }

    return 0;
}
```

Listing 15: Реализация compareBigUnsigned()

Алгоритм:

1. Нормализация чисел (удаление ведущих нулей);
2. Сравнение по количеству разрядов;
3. При равной длине - посимвольное сравнение слева направо.

Операции большой арифметики

2.4.9. Метод addBigUnsigned(const string& a, const string& b)

Назначение: Выполняет сложение двух беззнаковых многозначных чисел столбиком с учётом переноса между разрядами.

Вход:

- `const string& a`: Первое слагаемое.
- `const string& b`: Второе слагаемое.

Выход:

- `string`: Сумма $a + b$ без ведущих нулей.

```
string BigArithmeticCalc::addBigUnsigned(const string& a, const
   string& b) const {
    if (!isValidNumber(a) || !isValidNumber(b)) {
        return "ERR: invalid number";
    }

    string num1 = a;
    string num2 = b;

    // добавляем нули
    size_t max_len = max(num1.length(), num2.length());
    while (num1.length() < max_len) {
        num1 = small.getAdditiveIdentity() + num1;
    }
    while (num2.length() < max_len) {
        num2 = small.getAdditiveIdentity() + num2;
    }

    string result;
    string carry = small.getAdditiveIdentity();

    for (int i = static_cast<int>(max_len) - 1; i >= 0; --i) {
        string digit1(1, num1[i]);
        string digit2(1, num2[i]);

        auto res = small.addWithCarry(digit1, digit2, carry);
        if (res.first == "ERR") {
            return "ERR: invalid addition";
        }

        string sum = res.first;
        carry = res.second;

        result = sum + result;
    }

```

```
    if (carry != small.getAdditiveIdentity()) {
        result = carry + result;
    }

    if (isOverflow(result)) {
        return "ERR: overflow";
    }

    return deleteTrashZeros(result);
}
```

Listing 16: Реализация addBigUnsigned()

Алгоритм выполняет сложение справа налево (от младших разрядов к старшим), используя таблицу `additionTableWithCarry` для учёта переносов.

2.4.10. Метод subtractBigUnsigned(const string& a, const string& b)

Назначение: Выполняет вычитание двух беззнаковых многозначных чисел столбиком с учётом заимствования из старших разрядов. Требует, чтобы $a \geq b$.

Вход:

- `const string& a`: Уменьшаемое ($a \geq b$).
- `const string& b`: Вычитаемое.

Выход:

- `string`: Разность $a - b$ без ведущих нулей.

```
string BigArithmeticCalc::subtractBigUnsigned(const string& a,
   const string& b) const {
    if (!isValidNumber(a) || !isValidNumber(b)) {
        return "ERR: invalid number";
    }

    // сравниваем числа
    int cmp = compareBigUnsigned(a, b);
    if (cmp == 0) {
        return small.getAdditiveIdentity();
    }

    // определяем большее и меньшее число
    string larger = (cmp >= 0) ? a : b;
    string smaller = (cmp >= 0) ? b : a;

    // выравниваем длины
    size_t max_len = max(larger.length(), smaller.length());
    while (larger.length() < max_len) {
        larger = small.getAdditiveIdentity() + larger;
    }
    while (smaller.length() < max_len) {
```

```
        smaller = small.getAdditiveIdentity() + smaller;
    }

    string result;
    bool borrow = false; // флаг заема

    // вычитаем справа налево
    for (int i = static_cast<int>(max_len) - 1; i >= 0; --i) {
        string current(1, larger[i]);
        string digit2(1, smaller[i]);

        if (borrow) {
      if (current == small.getAdditiveIdentity()) {
                current = small.getAlphabet().back();
            } else {
                current = small.smallSubtract(current,
small.getMultiplicativeIdentity());
                borrow = false;
            }
        }

        int cmp_digits = small.compareElems(current, digit2);

        if (cmp_digits >= 0) {
            string result_digit = small.smallSubtract(current,
digit2);
            result = result_digit + result;
        } else {
            string temp = current; // аналог 10 + a если( нужем
заем)
            for (int j = 0; j < small.getN(); j++) {
                temp = small.nextElem(temp);
            }

            // теперь вычетаем
            string result_digit = small.smallSubtract(temp,
digit2);
            result = result_digit + result;
            borrow = true; // флаг
        }
    }
    if (borrow) {
        return "ERR: borrow after subtraction";
    }
    return deleteTrashZeros(result);
}
```

Listing 17: Реализация subtractBigUnsigned()

Алгоритм:

1. Сравниваем числа, чтобы определить, какое больше, и выравниваем их по длине добавлением ведущих нулей.

2. Вычитаем цифры поразрядно, начиная с младших разрядов, учитывая заем при необходимости.

3. Если текущая цифра меньше вычитаемой, берем заем у старшего разряда и корректируем результат.

4. Формируем итоговую строку результата и убираем ведущие нули.

2.4.11. Метод multiplyBigUnsigned(const string& a, const string& b)

Назначение: Выполняет умножение двух беззнаковых многозначных чисел столбиком. Число a умножается на каждую цифру числа b, промежуточные результаты сдвигаются и суммируются.

Вход:

- `const string& a`: Множитель.
- `const string& b`: Множитель.

Выход:

- `string`: Произведение $a \times b$ без ведущих нулей.

```
string BigArithmeticCalc::multiplyBigUnsigned(const string& a,
   const string& b) const {
    if (!isValidNumber(a) || !isValidNumber(b)) {
        return "ERR: invalid number";
    }

    if (a == small.getAdditiveIdentity() || b ==
   small.getAdditiveIdentity()) {
        return "[" + getMinNumber() + ";" + getMaxNumber() + "]";
    }

    string result = small.getAdditiveIdentity();

    // умножение столбиком: проходим по каждой цифре множителя
   справа налево
    for (int i = b.length() - 1; i >= 0; --i) {
        string multiplier_digit(1, b[i]);

        if (multiplier_digit != small.getAdditiveIdentity()) {
            string partial_product = multiplyByDigit(a,
   multiplier_digit);
            if (partial_product.substr(0, 4) == "ERR:") return
   partial_product;

            // сдвиг влево добавляем( нули справа)
            int shift_count = b.length() - 1 - i;
            for (int j = 0; j < shift_count; j++) {
                partial_product += small.getAdditiveIdentity();
```

```
            }

            result = addBigUnsigned(result, partial_product);
            if (result.substr(0, 4) == "ERR:") return result;
        }
    }

    return deleteTrashZeros(result);
}
```

Listing 18: Реализация multiplyBigUnsigned()

Алгоритм:

1. Умножение числа a на каждую цифру числа b;
2. Сдвиг промежуточных результатов влево на соответствующее количество разрядов;
3. Суммирование всех промежуточных результатов.

2.4.12. Метод multiplyByDigit(const string& num, const string& digit)

Назначение: Умножает многозначное число на однозначное число (один элемент алфавита). Используется в методе `multiplyBigUnsigned`.

Вход:

- `const string& num`: Многозначное число.
- `const string& digit`: Однозначное число (элемент алфавита).

Выход:

- `string`: Произведение $num \times digit$.

```
string BigArithmeticCalc::multiplyByDigit(const string& num, const
   string& digit) const {
    if (digit == small.getAdditiveIdentity()) return
   small.getAdditiveIdentity();
    if (digit == small.getMultiplicativeIdentity()) return num;

    string result = small.getAdditiveIdentity();
    string counter = small.getAdditiveIdentity();

    // умножаем через многократное сложение с буквенным счетчиком
    while (counter != digit) {
        result = addBigUnsigned(result, num);
        if (result.substr(0, 4) == "ERR:") return result;
        counter = small.nextElem(counter);
    }

    return result;
```

```
16 }
```

Listing 19: Реализация multiplyByDigit()

2.4.13. Метод divideBigUnsigned(const string& a, const string& b)

Назначение: Выполняет деление двух беззнаковых многозначных чисел ”уголком”. Возвращает частное и остаток.

Вход:

- `const string& a`: Делимое.
- `const string& b`: Делитель ($b \neq 0$).

Выход:

- `pair<string, string>`: Пара (частное, остаток).

```
pair<string, string> BigArithmeticCalc::divideBigUnsigned(const
   string& a, const string& b) const {
    if (!isValidNumber(a) || !isValidNumber(b)) {
        return {"ERR", "ERR"};
    }

    // деление на ноль
    if (b == small.getAdditiveIdentity()) {
        // 0/0 = [min;max]
        if (a == small.getAdditiveIdentity()) {
            return {small.getUniversum(), small.getUniversum()};
        }
        return {small.getEmptySet(), small.getEmptySet()};
    }

    // если делимое равно нулю
    if (a == small.getAdditiveIdentity()) {
        return {small.getAdditiveIdentity(),
   small.getAdditiveIdentity()};
    }

    string dividend = a;
    string divisor = b;

    // если делимое меньше делителя
    if (compareBigUnsigned(dividend, divisor) < 0) {
        return {small.getAdditiveIdentity(), dividend};
    }

    // деление в столбик
    string q_str;
    string curr_r = small.getAdditiveIdentity();

    // проходим по каждой цифре делимого слева направо
    for (size_t i = 0; i < dividend.length(); i++) {
        // добавляем следующую цифру к остатку
        curr_r = curr_r + string(1, dividend[i]);
        curr_r = deleteTrashZeros(curr_r);
```

```
37
38          // ищем максимальную цифру частного
39          string q_digit = small.getAdditiveIdentity();
40
41          // пробуем все возможные цифры от 1 до максимальной
42          string test_digit = small.getMultiplicativeIdentity();
43          while (test_digit != small.getAdditiveIdentity()) {
44              string test_product = multiplyByDigit(divisor,
    test_digit);
45
46              if (compareBigUnsigned(test_product, curr_r) > 0) {
47                  break;
48              }
49
50              q_digit = test_digit;
51              test_digit = small.nextElem(test_digit);
52          }
53
54          q_str += q_digit;
55
56          // вычитаем произведение из остатка
57          if (q_digit != small.getAdditiveIdentity()) {
58              string product = multiplyByDigit(divisor, q_digit);
59              curr_r = subtractBigUnsigned(curr_r, product);
60          }
61      }
62
63      string q = deleteTrashZeros(q_str);
64      string r = deleteTrashZeros(curr_r);
65
66      return {q, r};
67  }
```

Listing 20: Реализация divideBigUnsigned()

Алгоритм выполняет деление ”уголком”, последовательно вычитая делитель из текущего остатка.

Публичные операции с учётом знаков

2.4.14. Метод add(const string& a, const string& b)

Назначение: Выполняет сложение двух чисел с учётом знаков. Обрабатывает случаи сложения чисел с одинаковыми и разными знаками.

Вход:

- `const string& a`: Первое слагаемое.
- `const string& b`: Второе слагаемое.

Выход:

- `string`: Сумма $a + b$ со знаком.

```
string BigArithmeticCalc::add(const string& a, const string& b)
   const {
    bool neg_a = isNegative(a);
    bool neg_b = isNegative(b);

    string unsign_a = removeSign(a);
    string unsign_b = removeSign(b);

    // a + b оба( положительные)
    if (!neg_a && !neg_b) {
        return addBigUnsigned(unsign_a, unsign_b);
    }

    // -a + (-b) = -(a + b)
    if (neg_a && neg_b) {
        string sum = addBigUnsigned(unsign_a, unsign_b);
        if (sum.substr(0, 4) == "ERR:") return sum;
        return addSign(sum, true);
    }

    // a + (-b) = a - b
    if (!neg_a && neg_b) {
        int cmp = compareBigUnsigned(unsign_a, unsign_b);
        if (cmp >= 0) {
            return subtractBigUnsigned(unsign_a, unsign_b);
        } else {
            string diff = subtractBigUnsigned(unsign_b, unsign_a);
      cout << "Debug:" << diff << endl;
            return addSign(diff, true);
        }
    }

    // -a + b = b - a
    if (neg_a && !neg_b) {
        int cmp = compareBigUnsigned(unsign_b, unsign_a);
        if (cmp >= 0) {
            return subtractBigUnsigned(unsign_b, unsign_a);
        } else {
            string diff = subtractBigUnsigned(unsign_a, unsign_b);
            return addSign(diff, true);
        }
    }

    return "ERR: unknown case";
}
```

Listing 21: Реализация add()

Алгоритм:

1. Определяем знаки чисел и отделяем их абсолютные значения.

2. Если оба числа положительные, складываем их напрямую.

3. Если оба числа отрицательные, складываем их абсолютные значения и присваиваем результату знак минус.

4. Если одно число отрицательное, а другое положительное, выполняем вычитание меньшего числа из большего и присваиваем результату знак числа с большим абсолютным значением.

5. Возвращаем ошибку в случае некорректного входа или непредвиденной ситуации.

2.4.15. Метод subtract(const string& a, const string& b)

Назначение: Выполняет вычитание двух чисел с учётом знаков. Сводится к сложению с противоположным числом: $a - b = a + (-b)$.

Вход:

- `const string& a`: Уменьшаемое.
- `const string& b`: Вычитаемое.

Выход:

- `string`: Разность $a - b$ со знаком.

```
string BigArithmeticCalc::subtract(const string& a, const string&
   b) const {
    string neg_b = isNegative(b) ? removeSign(b) : "-" + b;
    return add(a, neg_b);
}
```

Listing 22: Реализация subtract()

2.4.16. Метод multiply(const string& a, const string& b)

Назначение: Выполняет умножение двух чисел с учётом знаков. Знак результата определяется правилом: $(-)\times(-) = (+)$, $(+)\times(-) = (-)$, $(+)\times(+) = (+)$.

Вход:

- `const string& a`: Множимое.
- `const string& b`: Множитель.

Выход:

- `string`: Произведение $a \times b$ со знаком.

```
string BigArithmeticCalc::multiply(const string& a, const string&
   b) const {
    bool neg_a = isNegative(a);
    bool neg_b = isNegative(b);

    string unsign_a = removeSign(a);
    string unsign_b = removeSign(b);

    string product = multiplyBigUnsigned(unsign_a, unsign_b);
    if (product.substr(0, 4) == "ERR:") return product;

    // результат отрицательный, если знаки разные
    bool result_negative = (neg_a != neg_b);

    return addSign(product, result_negative);
}
```

Listing 23: Реализация multiply()

2.4.17. Метод divide(const string& a, const string& b)

Назначение: Выполняет деление двух чисел с учётом знаков. Возвращает частное и остаток.

Вход:

- `const string& a`: Делимое.
- `const string& b`: Делитель ($b \neq 0$).

Выход:

- `string`: Частное a/b со знаком и остаток.

```
string BigArithmeticCalc::divide(const string& a, const string& b)
   const {
    bool neg_a = isNegative(a);
    bool neg_b = isNegative(b);

    string unsign_a = removeSign(a);
    string unsign_b = removeSign(b);

    auto [q, r] = divideBigUnsigned(unsign_a, unsign_b);

    // обработка специальных случаев
    if (q == small.getEmptySet() || q == small.getUniversum()) {
        return "Q: " + q + " | R: " + r;
    }

    if (q.substr(0, 3) == "ERR") {
        return q;
```

```
    }

    // если делимое отрицательное, а делитель положительный
    // то -a / b = -(a/b + 1), остаток = b - r
    if (neg_a && !neg_b && r != small.getAdditiveIdentity()) {
        // добавляем единицу к частному
        q = addBigUnsigned(q, small.getMultiplicativeIdentity());
        // вычисляем новый остаток: b - r
        r = subtractBigUnsigned(unsign_b, r);
        q = addSign(q, true);
    }
    // если делимое положительноем, а делитель отрицательный
    // то a / (-b) = -(a/b), остаток остается r
    else if (!neg_a && neg_b) {
        q = addSign(q, true);
    } else if (neg_a && !neg_b) { // -a / b = -(a/b) без остатка
        q = addSign(q, true);
    }
    // если оба отрицательные: -a / (-b) = a/b
    else if (neg_a && neg_b) {}

    return "Q: " + q + " | R: " + r;
}
```

Listing 24: Реализация divide()

Алгоритм:

1. Вычисляем частное и остаток для абсолютных значений чисел.
2. Корректируем знак частного и остатка в зависимости от знаков делимого и делителя.
3. Возвращаем результат в формате ”Q: <частное> | R: <остаток>”.

3. Результаты работы программы

3.1. Запуск программы

При запуске программа выводит приветственное сообщение и информацию о командах. Пользователь может вводить команды или арифметические выражения.

```
=========== Калькулятор БОЛЬШОЙ конечной арифметики Z8 ===========

Введите 'help' для справки по командам
Введите 'info' для информации о системе

=========== ИНФОРМАЦИЯ О КОНЕЧНОЙ АРИФМЕТИКЕ ===========

размерность: Z8
аддитивная единица (0): 'a'
мультипликативная единица (1): 'b'

максимальное количество разрядов: 8
диапазон чисел: [-hhhhhhhh; hhhhhhhh]

специальные правила:
  • a * a = [-hhhhhhhh; hhhhhhhh] (универсум)
  • число / a = ∅ (пустое множество)
  • a / a = [-hhhhhhhh; hhhhhhhh] (диапазон всех чисел)
  • переполнение (> 8 разрядов) = ERR: overflow
=========== ДИАГРАММА ХАССЕ (правило +1) ===========

Отношение порядка элементов Z8:

  a => b => c => e => d => g => f => h => a (цикл)

=====================================================
```

Рис. 3: Приветственное сообщение при запуске программы

3.2. Сценарий 1: Просмотр справки

3.2.1. Команда help

При вводе команды `help` программа выводит список всех доступных команд и их описание.

```
=========== ПОМОЩЬ ===========

доступные команды:


  БОЛЬШАЯ АРИФМЕТИКА (до 8 разрядов, с отрицательными):
    <число> + <число>       - сложение больших чисел
    <число> - <число>       - вычитание больших чисел
    <число> * <число>       - умножение больших чисел
    <число> / <число>       - деление больших чисел

    примечание: отрицательные числа пишутся с '-' (например: -abc)

  ИНФОРМАЦИЯ:
    info                    - информация об арифметике
    hasse                   - диаграмма хассе
    tables                  - все таблицы операций
    add_table               - таблица сложения
    mul_table               - таблица умножения
    sub_table               - таблица вычитания
    div_table               - таблица деления
    help                    - эта справка
    exit / quit             - выход

примеры:
  abd + ghf               - сложение положительных
  -abd + ghf              - сложение отрицательного и положительного
  hhhhhhhh + b            - переполнение (ERR: overflow)
  -c / d                  - деление отрицательного на положительное
  a / a                   - результат:[-hhhhhhhh; hhhhhhhh]
  g / a                   - результат: ∅ (пустое множество)

особенности деления с остатком:
  • при делении -a / b: частное увеличивается на 1, остаток = b - r
  • остаток может быть многозначным

 ============================================================
```

Рис. 4: Результат выполнения команды help

3.2.2. Команда info

При вводе команды `info` программа выводит информацию о конфигурации системы.

```
=========== ИНФОРМАЦИЯ О КОНЕЧНОЙ АРИФМЕТИКЕ ===========

размерность: Z8
аддитивная единица (0): 'a'
мультипликативная единица (1): 'b'

максимальное количество разрядов: 8
диапазон чисел: [-hhhhhhhh; hhhhhhhh]

специальные правила:
  • a * a = [-hhhhhhhh; hhhhhhhh] (универсум)
  • число / a = ∅ (пустое множество)
  • a / a = [-hhhhhhhh; hhhhhhhh] (диапазон всех чисел)
  • переполнение (> 8 разрядов) = ERR: overflow
```

Рис. 5: Результат выполнения команды info

3.3. Сценарий 2: Просмотр диаграммы Хассе

3.3.1. Команда hasse

При вводе команды `hasse` программа выводит диаграмму Хассе - визуальное представление отношения порядка на алфавите.

```
=========== ДИАГРАММА ХАССЕ (правило +1) ===========

Отношение порядка элементов Z8:

  a => b => c => e => d => g => f => h => a (цикл)

================================================
```

Рис. 6: Диаграмма Хассе

Диаграмма показывает циклический порядок элементов алфавита. Каждый элемент связан со следующим по правилу ”+1”.

3.4. Сценарий 3: Операции с большими числами

3.4.1. Сложение положительных чисел

Пользователь вводит выражение для сложения двух положительных чисел.

```
calc> abc + def

=========== Результат большой арифметики ===========
  abc + def = dga
====================================================
```

Рис. 7: Сложение положительных чисел

Результат вычисляется столбиком с учётом переносов между разрядами.

3.4.2. Сложение с переносами

Пример, демонстрирующий механизм переноса в старшие разряды.

```
calc> hhhh + b

=========== Результат большой арифметики ===========
  hhhh + b = baaaa
====================================================
```

Рис. 8: Сложение с переносом

При сложении `f + b` происходит переход через границу алфавита, что вызывает перенос в следующий разряд.

3.4.3. Сложение с отрицательными числами

Пример сложения чисел с разными знаками.

```
calc> abc + -abc

=========== Результат большой арифметики ===========
  abc + -abc = a
====================================================

calc> -def + bcd

=========== Результат большой арифметики ===========
  -def + bcd = -ebc
====================================================

calc> -bcd + def

=========== Результат большой арифметики ===========
  -bcd + def = ebc
====================================================
```

Рис. 9: Сложение с отрицательными числами

Программа правильно обрабатывает случаи:

- Сложение противоположных чисел (результат - нейтральный элемент)
- Сложение чисел с разными знаками (вычитание модулей)

3.4.4. Вычитание положительных чисел

```
calc> def - abc

=========== Результат большой арифметики ===========
  def - abc = dcd
====================================================

calc> abc - def

=========== Результат большой арифметики ===========
  abc - def = -dcd
====================================================
```

Рис. 10: Вычитание положительных чисел

При вычитании меньшего из большего результат положительный, при вычитании большего из меньшего - отрицательный.

3.4.5. Вычитание с заимствованием

Пример, демонстрирующий механизм заимствования из старших разрядов.

```
calc> baaa - hhh

=========== Результат большой арифметики ===========
  baaa - hhh = b
====================================================
```

Рис. 11: Вычитание с заимствованием

При вычитании `a - b` происходит заимствование из старшего разряда.

3.4.6. Умножение положительных чисел

```
calc> abc * h

=========== Результат большой арифметики ===========
  abc * h = baf
====================================================

calc> bc * de

=========== Результат большой арифметики ===========
  bc * de = gef
====================================================
```

Рис. 12: Умножение положительных чисел

Умножение выполняется столбиком с промежуточным суммированием.

3.4.7. Умножение с отрицательными числами

```
calc> -bc * de

=========== Результат большой арифметики ===========
  -bc * de = -gef
====================================================

calc> -abc * -h

=========== Результат большой арифметики ===========
  -abc * -h = baf
====================================================

calc> abc * -h

=========== Результат большой арифметики ===========
  abc * -h = -baf
====================================================
```

Рис. 13: Умножение с отрицательными числами

Знак результата определяется правилом:

- $+ \times + = +$
- $- \times - = +$
- $+ \times - = -$
- $- \times + = -$

3.4.8. Деление положительных чисел

```
calc> gef / bc

=========== Результат большой арифметики ===========
  gef / bc = Q: de | R: a
====================================================

calc> abcd / ef

=========== Результат большой арифметики ===========
  abcd / ef = Q: c | R: ea
====================================================
```

Рис. 14: Деление положительных чисел

Деление выполняется ”уголком” с получением целой части.

3.4.9. Деление с отрицательными числами

```
calc> -gef / bc

=========== Результат большой арифметики ===========
  -gef / bc = Q: -de | R: a
====================================================

calc> gef / -bc

=========== Результат большой арифметики ===========
  gef / -bc = Q: -de | R: a
====================================================

calc> -gef / -bc

=========== Результат большой арифметики ===========
  -gef / -bc = Q: de | R: a
====================================================
```

Рис. 15: Деление с отрицательными числами

3.5. Сценарий 4: Обработка ошибок

3.5.1. Некорректный формат числа

При вводе числа, содержащего символы вне алфавита, программа выводит сообщение об ошибке.

```
calc> abx / f

=========== Результат большой арифметики ===========
  abx / f = ERR
====================================================
```

Рис. 16: Ошибка: некорректный формат числа

Символ `x` не принадлежит алфавиту системы.

3.5.2. Деление на ноль

При попытке деления на нейтральный элемент по сложению (аналог нуля) программа выводит ошибку.

```
calc> abc / a

=========== Результат большой арифметики ===========
  abc / a = Q: ø | R: ø
====================================================
```

Рис. 17: Ошибка: деление на ноль

3.5.3. Переполнение разрядов

При превышении максимального количества разрядов (MAX_DIGITS = 8) программа выводит ошибку переполнения.

```
calc> hhhhhhhhhhhhh * hh

=========== Результат большой арифметики ===========
  hhhhhhhhhhhhh * hh = ERR: overflow
====================================================
```

Рис. 18: Ошибка: переполнение

Результат операции превышает 8 разрядов, что не допускается системой.

3.5.4. Неизвестная команда

При вводе неизвестной команды программа выводит сообщение об ошибке.

```
calc> home
Ошибка: неверный формат команды. Введите 'help' для справки
```

Рис. 19: Ошибка: неизвестная команда

3.6. Выход из программы

При вводе команды `exit` или `quit` программа завершает работу с прощальным сообщением.

```
calc> home
Ошибка: неверный формат команды. Введите 'help' для справки
```

Рис. 20: Завершение работы программы

Заключение

В ходе курсовой работы был разработан калькулятор большой конечной арифметики $\langle Z_8^8; +, * \rangle$ с поддержкой восьми разрядов. Калькулятор выполняет четыре арифметические операции: сложение, вычитание, умножение и деление над многозначными числами в заданной системе счисления.

Основные результаты

В программной части была реализована малая конечная арифметика с операциями над однозначными элементами через диаграмму Хассе. Построены таблицы операций малой арифметики для сложения, умножения. Для выполнения операций столбиком создана таблица сложения с переносом.

На основе малой арифметики реализована большая арифметика с операциями над многозначными числами до восьми разрядов. Добавлена поддержка отрицательных чисел с корректной обработкой знаков при выполнении всех операций.

Разработана система команд для просмотра информации о системе, диаграммы Хассе. Реализована обработка ошибок, включая деление на ноль, переполнение разрядов и некорректный формат чисел.

Реализованные алгоритмы показали корректность работы для различных входных данных.

Достоинства реализации

Достоинством программы является модульная архитектура: малая арифметика работает с однозначными элементами, а большая арифметика использует её для операций столбиком. Код разделён по файлам, каждый из которых отвечает за свою часть функционала.

Использование предвычисленных таблиц операций делает вычисления эффективными: все операции малой арифметики вычисляются один раз при инициализации, а затем используются для быстрого доступа к значениям. Это позволяет избежать повторных вычислений при работе с многозначными числами.

Недостатки реализации

Конфигурация системы (алфавит, правило "$+1$", нейтральные элементы) задаётся в файле `config.hpp` и требует перекомпиляции при изменении. Пользователь не может менять параметры во время работы программы.

Также программа ограничена восьмью разрядами, что не позволяет работать с очень большими числами. При превышении лимита возникает ошибка переполнения.

Возможности масштабирования и улучшения

Все операции выполняются через консольный интерфейс. Визуализация таблиц и диаграммы Хассе была бы удобнее через GUI при помощи фреймворка Qt.

Можно добавить возможность загружать конфигурацию из файла, чтобы пользователь мог менять алфавит и правило ”+1” без повторной компиляции программы.

Список литературы

[1] Сайт кафедры с учебными материалами по курсу *«Дискретная математика»*. Ссылка: https://tema.spbstu.ru/dismath/ (Дата обращения: 21.10.2025).

[2] Новиков Ф.А. *Дискретная математика*. Учебник. Ссылка на PDF: https://stugum.wordpress.com/wp-content/uploads/2014/03/novikov.pdf (Дата обращения: 21.10.2025).