МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра информационной безопасности

ОТЧЕТ

по лабораторной работе №2
по дисциплине «Алгоритмы и структуры данных»
Тема: Экспериментальное нахождение средней временной сложности алгоритма
Вариант: Быстрое возведение квадратной матрицы в степень

Студент гр. 4364 _____ Карпеев М.А.

Студент гр. 4364 _____ Котельников Т.А.

ТЧ	РА	ЭЧ	ИП	ТП	О/В

защищено ___ . ___ . _____
с _____ попытки
с оценкой _____

Преподаватель _____ Абросимов И.К.

Санкт-Петербург
2025

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

Студенты Карпеев М.А., Котельников Т.А группы 4364

Тема работы: экспериментальное нахождение средней временной сложности алгоритма быстрого возведения квадратной матрицы в степень

Задание на лабораторную работу:

1) Реализовать и отладить на языке С структуры данных матрицы с коэффициентами из конечного поля и операциями над ней;

2) Реализовать алгоритм быстрого возведения квадратной матрицы в степень с использованием структуры данных.

3) Выполнить экспериментальное нахождение средней сложности исследуемого алгоритма

Содержание пояснительной записки: разделы «СОДЕРЖАНИЕ», «ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ» с подразделами «Описание исследуемого алгоритма» и «Методика экспериментального исследования», «РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ» с подразделами «Реализация исследуемого алгоритма», «Тестирование реализации алгоритма», «Проведение эксперимента» и «Обработка результатов эксперимента», «ЗАКЛЮЧЕНИЕ», «СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ»

Предполагаемый объем пояснительной записки: не менее 110 страниц.

Дата выдачи задания: 20.10.2025

Дата сдачи пояснительной записки: _____

Задание получил _____ Карпеев М.А.

Задание получил _____ Котельников Т.А.

Задание выдал _____ Абросимов И.К.

СОДЕРЖАНИЕ

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ 4
1.1 Описание исследуемого алгоритма 4
1.2 Методика экспериментального исследования 7
2 РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ 10
2.1 Реализация исследуемого алгоритма 10
2.2 Реализация исследуемого алгоритма 49
2.3 Тестирование реализации алгоритма 53
2.4 Проведение эксперимента 59
2.5 Обработка результатов эксперимента 69
ЗАКЛЮЧЕНИЕ 73
СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ 74
ПРИЛОЖЕНИЕ А. Доказательства используемых утверждений 75
ПРИЛОЖЕНИЕ Б. Используемые функции 76
ПРИЛОЖЕНИЕ В. Разработанные функции 78
ПРИЛОЖЕНИЕ Г. Исходный код разработанной программы 81

1 ТЕОРЕТИЧЕСКОЕ ВВЕДЕНИЕ

Цель работы: экспериментальное нахождение средней временной сложности алгоритма быстрого возведения матрицы в степень на языке С.

1.1 Описание исследуемого алгоритма

Быстрое возведение квадратной матрицы в степень над конечным полем — алгоритм, основанный на принципе быстрого возведения числа в степень по модулю [1, с. 1-5]. Пусть дана квадратная матрица $A \in \mathrm{M}_{k\times k}(\mathbb{F})$ над конечным полем $\mathbb{F}$ и целое число $n \geq 0$. Требуется вычислить значение A^n.

Основная идея алгоритма состоит в использовании двоичного представления показателя степени (m − количество бит). Пусть

$$n = \sum_{i=0}^{m-1} b_i 2^i \,, b_i \in 0{,}1, \tag{1.1}$$

Где $m = \lfloor log^2(n) \rfloor + 1$. Тогда

$$A^n = \prod_{i=0}^{m-1} (A^{2^i})^{b_i}. \tag{1.2}$$

В данной работе используется право направленный бинарный алгоритм (Right-to-Left Binary Exponentiation), при котором обработка двоичного разложения показателя n начинается с младшего бита. На каждом шаге, если текущий бит $b_i = 1$, промежуточный результат умножается на текущее значение степени A^{2^i}. После обработки бита показатель делится на два с помощью операции двоичного сдвига вправо, а текущая степень матрицы возводится в квадрат:

$$A^{2^{i+1}} = (A^{2^i})^2. \tag{1.3}$$

Каждая операция возведения в квадрат и перемножения матриц выполняется в поле $\mathbb{F}$, причём элементы матриц представлены встроенным типом данных unsigned long long в диапазоне от 2^{19} до $2^{20} - 1$. Для предотвращения переполнения при вычислениях после каждой операции умножения и сложения выполняется редукция по модулю.

Входные данные: квадратная матрица порядка n с элементами из поля $\mathbb{F}$ и показатель степени n.

Выходные данные: каждая строка выходного файла содержит четыре значения — размер квадратной матрицы, показатель степени, размер конечного поля и время вычисления в наносекундах.

Алгоритм завершает свою работу, когда показатель степени k становится равен нулю.

Чтобы оценить нижнюю границу числа шагов, заметим, что если показатель степени n является степенью двойки, например $n = 2^p$, то выполняется ровно p операций возведения в квадрат (так как ни один бит, кроме старшего, не равен единице). В этом случае количество матричных умножений минимально.

Верхняя граница достигается, когда все биты показателя равны единице (например, $n = 2^p - 1$); тогда каждое возведение в квадрат сопровождается дополнительным умножением на матрицу A.

Алгоритм быстрого возведения квадратной матрицы в степень основывается на свойствах возведения матриц в целую неотрицательную степень (1.4) – (1.7):

$$A^0 = E_k; \tag{1.4}$$

$$A^1 = A\,; \tag{1.5}$$

$$A^{m+n} = A^m \cdot A^n,\ \ m, n \geq 0; \tag{1.6}$$

$$(A^m)^n = A^{m \cdot n},\ \ m, n \geq 0, (1.4). \tag{1.7}$$

При этом свойства (1.4) и (1.5) являются базовыми и применяются однократно при обработке показателей степени 0 или 1. Свойства (1.6) и (1.7) лежат в основе метода «разделяй и властвуй», позволяя раскладывать показатель степени на части и использовать многократное возведение в квадрат для уменьшения количества умножений матриц.

Используя свойства (1.4) – (1.7) и учитывая, что (1.4) и (1.5) применяются однократно, можно построить рекурсивную форму алгоритма быстрого возведения матрицы A в степень $n \geq 0$:

$$\mathrm{P(A,n)} = \begin{cases} E_k, & n = 0, \\ A, & n = 1, \\ \left(P\left(A, \frac{n}{2}\right)\right)^2, & n > 1 \text{ и } n_i \bmod 2 = 0 \\ A \cdot \left(P\left(A, \frac{n}{2}\right)\right)^2, & n > 1 \text{ и } n_i \bmod 2 = 1 \end{cases} \quad (1.8)$$

Здесь $P(A, n)$— результат возведения матрицы A в степень n, E_k— единичная матрица порядка k

Пусть:

- R_i — результат на i-м шаге,
- P_i — текущая степень матрицы (то, что будем возводить в квадрат на следующем шаге),
- n_i — оставшийся показатель степени на i-м шаге.

Тогда итеративная форма будет выглядеть так:

$$\begin{cases} R_{i+1} = \begin{bmatrix} R_i \cdot P_i, & n_i mod 2 = 1, \\ R_i, & n_i mod 2 = 0, \end{bmatrix} R_0 = E_k, \\ P_{i+1} = P_i \cdot P_i, \ P_0 = A, \\ n_{i+1} = \lfloor n_i/2 \rfloor, \ n_0 = n. \end{cases} \quad (1.9)$$

Условием завершения итеративной формы алгоритма является достижение нулевого показателя степени $n = 0$, при этом результатом является матрица A^n. Количество шагов алгоритма оценивается как

$$T_A(n) = \lfloor log^2 n \rfloor + \sum_{i=0}^{\lfloor log^2 n \rfloor} b_i, \quad (1.10)$$

Где первое слагаемое — количество операций возведения в квадрат, второе — количество единиц в двоичной записи числа или операций «Result на Temp», т.е. $R_i \cdot P_i$.

Доказательства утверждений (1.4) – (1.7) приведены в разделе «ПРИЛОЖЕНИЕ А. Доказательства используемых утверждений».

В данной работе исследуется, как распределены средние временные затраты алгоритма быстрого возведения в степень для матриц, элементы которых принадлежат диапазону от 2^{19}до $2^{20}-1$. Минимальное количество шагов соответствует случаям, когда показатель степени — степень двойки, а максимальное — числам, состоящим из всех единичных битов в двоичном представлении.

1.2 Методика экспериментального исследования

Экспериментальное исследования проводится по методике [2, с. 4-6].

В работе исследуется средняя временная сложность алгоритма – среднее количество временных затрат алгоритма A на множестве входных данных заданного размера n, обозначаемый как $\overline{T_A}(n)$

$$\overline{T_A}(n) = \frac{1}{|X|}\sum_{x \in X_n}^{n} \dot{C_A^T}(x), \qquad (1.11)$$

где $\mathrm{X_n} = \mathrm{x} \mid \|x\| = n -$ множество входных данных размера n, $C_A^T(x)$ – временные затраты алгоритма, равные количеству времени, которое затрачивается на выполнение алгоритма A при входных данных $\mathrm{x} \in X_n$ размера $\|x\| = n$. В работе $C_A^T(x)$ измеряется непосредственно, например, с помощью собственной функции `get_time_ns()`, на основе ф-ции `clock_gettime(clockid_t clock_id, struct timespec *tp)` из заголовочного файла `#include <time.h>`. Выбор такой ф-ции стало необходимая точность измерения в наносекундах.

Нахождение функции $\overline{T_A}(n)$ в виде выражения с конкретными константами слишком трудоемко, поскольку требует аналитического построения распределения временных затрат входных данных из множества X_n. Поэтому вместо нахождения точного вида описывающей $\overline{T_A}(n)$ функции часто ограничиваются определением некоторого множества функций, к которому принадлежит $\overline{T_A}(n)$.

Для оценивания временной сложности в сверху используется символ «о большое». Говорят, что f$(n) \in O\big(g(n)\big)$ тогда и только тогда, когда существует положительное вещественное c и натуральное N такое, что для любого $n > N$ выполняется неравенство

$$|f(n)| \leq c \cdot |g(n)|. \tag{1.12}$$

Для оценивания алгоритмов обычно служат следующие функции: константа, логарифм, степень, произведение степени на логарифм, экспонента, факториал. При этом чем быстрее растут значения оцениваемой функции, тем больше времени потребует выполнение алгоритма.

Для определения класса функции, которому принадлежит временная сложность, обычно используются случайно выбранные наборы входных данных фиксированного размера $n_1, n_2, \dots, n_k$, где $k \geq 5$. Пусть $t_1, t_2, \dots, t_k$ – измеренное время работы алгоритма при соответствующем размере входных данных, f – функция такая, что $T(n) \in O\big(f(n)\big)$, тогда $t_i = T(n_i) = C \cdot f(n_i)$, где $C > 0$ – некоторая константа и функцию f можно определить следующим образом. Выберем размеры входных данных так, чтобы было справедливо приближенное равенство $\frac{t_{i+1}}{t_i} \approx \alpha$, где α – некоторая константа. Тогда, с учетом

$$\frac{t_{i+1}}{t_i} = \frac{C \cdot f(n_{i+1})}{C \cdot f(n_i)} = \frac{f(n_{i+1})}{f(n_i)}. \tag{1.13}$$

имеем $f(n_{i+1}) = \alpha \cdot f(n_i)$ при $i \in \overline{1, k-1}$, тогда значение данного отношения характеризует скорость роста функции f, по которой можно определить саму функцию f, как решение системы

$$\begin{cases} n_{i+1} = g(n_i), \\ f(n_{i+1}) = \alpha \cdot f(n_i). \end{cases} \tag{1.14}$$

причем общий вид функции f известен (например, степенная, показательная).

Если известно, что алгоритм выполняется за полиномиальное время, то размер входных данных после каждого измерения времени рекомендуется удваивать. Если же алгоритм выполняется за время, не меньшее экспоненциального, то размер входных данных рекомендуется увеличивать на

небольшую константу. Таким образом, n_{i+1} и n_i обычно связаны функцией g одного из следующих видов

$$g(n_i) = 2n_i. \quad (1.15)$$

$$g(n_i) = n_i + m. \quad (1.16)$$

где m – фиксированная положительная константа.

Особый случай возникает, когда $\frac{t_{i+1}}{t_i} \neq const$ даже при выборе $n_{i+1} = n_i + 1$. В этом случае следует выполнить интерполяцию отношений $\frac{t_{i+1}}{t_i}$ для всех $i \in \overline{1, k-1}$. Пусть $\alpha(n)$ – результат такой интерполяции, тогда

$$f(n+1) = \alpha(n) \cdot f(n). \quad (1.17)$$

Это означает, что алгоритм имеет сложность, сравнимую с факториальной или даже превосходящую ее.

Размер входных данных при проведении экспериментов рекомендуется выбирать таким, чтобы, с одной стороны, они были достаточно «большими», так как оценивается функция $f(n)$ в предположении большого размера n входных данных, с другой стороны, не чрезмерно большими, чтобы время измерения было разумным (не более двадцати минут на все измерения времени).

В случае, когда время работы алгоритма настолько мало, что его нельзя измерить средствами используемого языка программирования, следует выполнять генерацию некоторого количества N входных данных фиксированного размера, измерять общее время t_i' выполнения алгоритма для всех сгенерированных входах, после чего вычислить время работы на одном входе как $t_i = \frac{t_i'}{N}$.

В случае, когда входных параметров у алгоритма несколько, среди них выбирается тот, который вносит наибольший вклад с точки зрения анализа сложности. Если такой выбор сделать не удается, то следует анализировать поведение каждого параметра отдельно, либо в совокупности (например, связав длины входных данных алгоритма при помощи некоторой функции).

2 РЕШЕНИЕ ПОСТАВЛЕННОЙ ЗАДАЧИ

2.1 Реализация исследуемого алгоритма

В данной работе необходимо использовать структуру данных «Матрица с коэффициентами из конечного поля». Ниже приведено описание структуру данных с указанием названием, типов и назначением полей.

Таблица 2.1 — Описание полей структуру данных `Matrix`

Название	Тип данных	Назначение
`data`	`ULL**`	Указатель на матрицу (двумерный динамический массив)
`rows`	`int`	Количество строк в матрице
`cols`	`int`	Количество столбцов в матрице
`field_size`	`ULL`	Размер конечного поля, по которому выполняются арифметические операции (если 0 — без модуля)

```
typedef unsigned long long ULL;

/* Структура данных для матрицы */
typedef struct Matrix
{
    ULL** data;      /* указатель на массив строк */
    int rows;                       /* число строк */
    int cols;                       /* число столбцов */
    ULL field_size;  /* модуль (размер конечного поля) */
} Matrix;
```

Также, в рамках исследования данного алгоритма, были реализованы восемь основных методов работы со структурой данных `Matrix`: сложение, разность, умножение на число (скаляр), транспонирование, умножение на матрицу согласованного размера, получение подматрицы (через левую, правую, верхнюю и нижнюю границы), преобразование матрицы в строку типа `char*` и преобразование строки типа `char*` в матрицу. Дополнительно были созданы функции для динамического создания (`matrix_create`) и освобождения памяти (`matrix_free`) структуры `Matrix`.

```
/* ---------- Функции (матрицы) ---------- */

/*
 * Создать матрицу rows x cols, поле field_size.
 * result — выходной параметр (адрес указателя на Matrix).
 * [IN] rows - количество строк матрицы
 * [IN] cols - количество столбцов матрицы
 * [IN] field_size - размер данных (по умолчанию, 0)
 * [OUT] result - указатель на созданную матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_create(int rows, int cols, ULL field_size,
    Matrix** result);

/*
 * Освободить память, выделенную под матрицу.
 * Уничтожает структуру и внутренние данные, предотвращая
 * утечку памяти.
 * Безопасно при передаче NULL (не вызывает разыменования NULL).
 * [IN] matrix — указатель на созданную матрицу для удаления
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_free(Matrix** matrix);
```

```
/*
 * Создать копию матрицы src и вернуть её через result.
 * Полностью дублирует размеры, поле field_size и данные.
 * [IN] src — исходная матрица
 * [OUT] result — указатель на указатель, куда будет записана
 * новая копия
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_copy(const Matrix* src, Matrix** result);

/*
 * Сложить две матрицы одинакового размера: a + b.
 * Операция выполняется в поле field_size, если оно задано.
 * [IN] a — первая матрица
 * [IN] b — вторая матрица
 * [OUT] result — указатель на новую матрицу, содержащую сумму
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_sum(const Matrix* a, const Matrix* b,
    Matrix** result);

/*
 * Вычесть матрицу b из матрицы a: a - b.
 * Операция выполняется в поле field_size, если оно задано.
 * [IN] a — уменьшаемое
 * [IN] b — вычитаемое
 * [OUT] result — указатель на матрицу результата
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_subtract(const Matrix* a, const Matrix* b,
    Matrix** result);
```

```
/*
 * Умножить матрицу a на скаляр scalar в поле field_size.
 * Каждое значение элемента матрицы умножается на scalar.
 * [IN] a — исходная матрица
 * [IN] scalar — множитель
 * [OUT] result — указатель на матрицу результата
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_scalar_multiply(const Matrix* a, ULL scalar,
    Matrix** result);

/*
 * Транспонировать матрицу a (перевернуть строки и столбцы).
 * Результирующая матрица имеет размеры cols x rows.
 * [IN] a — исходная матрица
 * [OUT] result — указатель на транспонированную матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_transpose(const Matrix* a, Matrix** result);

/*
 * Вырезать подматрицу из матрицы a по указанным индексам.
 * Диапазоны [start_row, end_row) и [start_col, end_col)
 * должны быть корректными.
 * [IN] a — исходная матрица
 * [IN] start_row — начальный индекс строки
 * [IN] end_row — конечный индекс строки (не включая)
 * [IN] start_col — начальный индекс столбца
 * [IN] end_col — конечный индекс столбца (не включая)
 * [OUT] result — указатель на вырезанную подматрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_submatrix(const Matrix* a, int start_row, int
    end_row, int start_col, int end_col, Matrix** result);
```

```
/*
 * Перемножить матрицы a и b: a × b.
 * Количество столбцов в a должно совпадать с количеством
 * строк в b.
 * [IN] a — первая матрица (левый множитель)
 * [IN] b — вторая матрица (правый множитель)
 * [OUT] result — указатель на новую матрицу с произведением
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result);

/*
 * Умножить два числа по модулю (a * b) % mod.
 * Используется в арифметике поля field_size, предотвращая
 * переполнение.
 * [IN] a — первый множитель
 * [IN] b — второй множитель
 * [IN] mod — модуль (если 0, операция выполняется без модуля)
 * [OUT] result - указатель на результат умножения
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int multiply_mod(ULL a, ULL b, ULL mod, ULL* result);

/*
 * Возвести квадратную матрицу base в степень exponent.
 * Используется метод бинарного возведения для эффективности.
 * Операции выполняются в поле field_size.
 * [IN] base — квадратная матрица (n x n)
 * [IN] exponent — показатель степени
 * [OUT] result — указатель на результирующую матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_power(const Matrix* base, ULL exponent,
    Matrix** result);
```

```
/*
 * Напечатать матрицу в стандартный поток вывода.
 * Формат вывода зависит от field_size (по модулю или
 * обычные значения).
 * Используется для отладки и проверки корректности.
 * [IN] matrix — указатель на матрицу для печати
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_print(const Matrix* matrix);

/*
 * Преобразовать матрицу в строковый формат.
 * Формат: (a11,a12;a21,a22,...)
 * Строка выделяется через malloc, необходимо освободить
 * через free.
 * [IN] matrix — указатель на исходную матрицу
 * [OUT] result — указатель на строку с результатом
 * [RETURN] STRING_SUCCESS или код ошибки STRING_STATUS
 */
int matrix_to_string(const Matrix* matrix, char** result);

/*
 * Преобразовать строку вида "(...)" в матрицу.
 * Строка должна содержать элементы через ',' и строки
 * через ';'.
 * [IN] str — входная строка с данными матрицы
 * [IN] field_size — размер поля для модульной арифметики
 * [OUT] result — указатель на созданную матрицу
 * [RETURN] STRING_SUCCESS или код ошибки STRING_STATUS
 */
int string_to_matrix(const char* str, unsigned long long
    field_size, Matrix** result);
```

Основными методами, применяемыми непосредственно в алгоритме быстрого возведения квадратной матрицы в степень, являются: создание матрицы (`matrix_create`), освобождение памяти (`matrix_free`), сложение (`matrix_sum`) и умножение (`matrix_multiply`).

Ниже приведено подробное описание разработки необходимых 8 функций (с учётом `matrix_create` и `matrix_free`):

Создание матрицы:

```
int matrix_create(int rows, int cols, ULL field_size,
    Matrix** result)
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.2).

Таблица 2.2 – Переменные, используемые для реализации функции

Название	Назначение
`rows`	Количество строк создаваемой матрицы
`cols`	Количество столбцов создаваемой матрицы
`field_size`	Размер поля для арифметики по модулю (0 — обычная арифметика)
`result`	Указатель на указатель для возвращаемой матрицы
`matrix`	Указатель на создаваемую матрицу типа `Matrix`
`i`	Индекс текущей строки при выделении памяти для элементов
`j`	Индекс ранее выделенных строк при освобождении памяти в случае ошибки

```
int matrix_create(int rows, int cols, ULL field_size,
    Matrix** result)
{
    if (!result)
        return MATRIX_ERROR_NULL_POINTER;

    if (rows < 1 || cols < 1)
```

```
    return MATRIX_ERROR_INVALID_SIZE;

Matrix* matrix = malloc(sizeof(Matrix));
if (!matrix)
    return MATRIX_ERROR_CREATION;

matrix->rows = rows;
matrix->cols = cols;
matrix->field_size = field_size;
matrix->data = NULL;

matrix->data = (ULL**)malloc(rows * sizeof(ULL*));
if (!matrix->data)
{
    free(matrix);
    return MATRIX_ERROR_CREATION;
}

for (int i = 0; i < rows; i++)
{
    matrix->data[i] = (ULL*)calloc(cols, sizeof(ULL));
    if (!matrix->data[i])
    {
        for (int j = 0; j < i; j++)
        {
            free(matrix->data[j]);
        }
        free(matrix->data);
        free(matrix);
        return MATRIX_ERROR_CREATION;
    }
}

*result = matrix;
return MATRIX_SUCCESS;
```

```
}
```

Освобождение памяти матрицы

```
int matrix_free(Matrix** matrix_ptr)
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.3).

Таблица 2.3 – Переменные, используемые для реализации функции

Название	Назначение
`matrix_ptr`	Указатель на указатель матрицы для освобождения
`matrix`	Локальная переменная для хранения указателя на матрицу перед освобождением
`i`	Индекс текущей строки при освобождении памяти для элементов

```
int matrix_free(Matrix** matrix_ptr)
{
    if (!matrix_ptr)
    {
        return MATRIX_SUCCESS;
    }

    Matrix* matrix = *matrix_ptr;
    if (!matrix)
    {
        return MATRIX_SUCCESS;
    }

    if (matrix->data)
    {
        for (int i = 0; i < matrix->rows; i++)
        {
            if (matrix->data[i])
            {
                free(matrix->data[i]);
```

```
                matrix->data[i] = NULL;
            }
        }
        free(matrix->data);
        matrix->data = NULL;
    }

    free(matrix);
    *matrix_ptr = NULL;

    return MATRIX_SUCCESS;
}
```

Сложение матриц

```
int matrix_sum(const Matrix* a, const Matrix* b, Matrix** result)
```

Для реализации функции применим итеративную формулу (2.1)

$$\begin{cases} C_{i,j} = A_{i,j} + B_{i,j}, & field = 0, \\ C_{i,j} = \left(\left(A_{i,j} mod field\right) + \left(B_{i,j} mod field\right)\right) mod\ field, & field \neq 0. \end{cases} \quad (2.1)$$

Где $i \in \overline{0, rows-1}, j \in \overline{0, cols-1}; A, B, C \in \mathrm{M}_{m \times n}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.9) соответствует функция:

```
int matrix_sum0(const Matrix* A, const Matrix* B, Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, A->cols, A->field_size, &C);
    for (int p = 0; p < A->rows; p++)
    {
        for (int q = 0; q < A->cols; q++)
        {
            ULL m = A->data[p][q];
            ULL n = B->data[p][q];
            ULL r;
            if (A->field_size == 0)
            {
```

```
                r = m + n;
            }
            else
            {
                r = (m % A->field_size + n % A->field_size)
                % A->field_size;
            }
            C->data[p][q] = r;
        }
    }
    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.4).

Таблица 2.4 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
A	a	Указатель на первую исходную матрицу
B	b	Указатель на вторую исходную матрицу
R	result	Указатель на указатель матрицы результата
C	sum_matrix	Указатель на создаваемую матрицу суммы
p	i	Индекс текущей строки матрицы
q	j	Индекс текущего столбца матрицы
m	x	Значение элемента матрицы A в позиции (i,j)
n	y	Значение элемента матрицы B в позиции (i,j)
r	r	Вычисленное значение суммы элементов для позиции (i,j)

```
int matrix_sum(const Matrix* a, const Matrix* b, Matrix** result)
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1 || b->rows < 1 ||
        b->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    if (a->rows != b->rows || a->cols != b->cols)
    {
        return MATRIX_ERROR_DIMENSION;
    }

    if (a->field_size != b->field_size)
    {
        return MATRIX_ERROR_INVALID_FIELD;
    }

    int error;
    Matrix* sum_matrix;
    error = matrix_create(a->rows, a->cols, a->field_size,
        &sum_matrix);
    if (error != MATRIX_SUCCESS)
    {
        return error;
```

```
    }

    for (int i = 0; i < a->rows; i++)
    {
        for (int j = 0; j < a->cols; j++)
        {
            ULL x = a->data[i][j];
            ULL y = b->data[i][j];

            if (a->field_size == 0)
            {
                if (x > ULLONG_MAX - y)
                {
                    matrix_free(&sum_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }
                sum_matrix->data[i][j] = x + y;
            }
            else
            {
                x %= a->field_size;
                y %= a->field_size;

                if (x > ULLONG_MAX - y)
                {
                    matrix_free(&sum_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }
                sum_matrix->data[i][j] = (x+y) % a->field_size;
            }
        }
    }
    *result = sum_matrix;
    return MATRIX_SUCCESS;
}
```

Вычитание матриц

```
int matrix_subtract(const Matrix a, const Matrix b,
    Matrix** result);
```

Для реализации функции применим итеративную формулу (2.2):

$$\begin{cases} C_{i,j} = A_{i,j} - B_{i,j}, & field = 0, \\ C_{i,j} = \left(\left(A_{i,j} mod\ field\right) - \left(B_{i,j} mod\ field\right)\right) mod\ field, & field \neq 0. \end{cases} \quad (2.2)$$

Где $i \in \overline{(0, rows - 1)}, j \in \overline{(0, cols - 1)}; A, B, C \in \mathrm{M}_{m \times n}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.2) соответствует функция:

```
int matrix_subtract0(const Matrix* A, const Matrix* B,
    Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, A->cols, A->field_size, &C);

    for (int p = 0; p < A->rows; p++)
    {
        for (int q = 0; q < A->cols; q++)
        {
            ULL m = A->data[p][q];
            ULL n = B->data[p][q];
            ULL r;

            if (A->field_size == 0)
            {
                r = m - n;
            }
            else
            {
                r = ((m % A->field_size) -
                    (n % A->field_size)) % A->field_size;
            }

            C->data[p][q] = r;
        }
    }

    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.5).

Таблица 2.5 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
A	a	Указатель на первую исходную матрицу
B	b	Указатель на вторую исходную матрицу
R	result	Указатель на указатель матрицы результата
C	sub_matrix	Указатель на создаваемую матрицу разности
p	i	Индекс текущей строки матрицы
q	j	Индекс текущего столбца матрицы
m	x	Значение элемента матрицы A в позиции (i,j)
n	y	Значение элемента матрицы B в позиции (i,j)
r	r	Вычисленное значение разности элементов для позиции (i,j)

```
int matrix_subtract(const Matrix* a, const Matrix* b, Matrix**
result)
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1 || b->rows < 1 ||
        b->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    if (a->rows != b->rows || a->cols != b->cols)
    {
        return MATRIX_ERROR_DIMENSION;
```

```
    }

    if (a->field_size != b->field_size)
    {
        return MATRIX_ERROR_INVALID_FIELD;
    }

    int error;
    Matrix* sub_matrix;
    error = matrix_create(a->rows, a->cols, a->field_size,
        &sub_matrix);
    if (error != MATRIX_SUCCESS)
    {
        return error;
    }

    for (int i = 0; i < a->rows; i++)
    {
        for (int j = 0; j < a->cols; j++)
        {
            ULL x = a->data[i][j];
            ULL y = b->data[i][j];

            if (a->field_size == 0)
            {
                if (x < y)
                {
                    matrix_free(&sub_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }
                sub_matrix->data[i][j] = x - y;
            }
            else
            {
                x %= a->field_size;
                y %= a->field_size;
                sub_matrix->data[i][j] =
                    (x + a->field_size - y) % a->field_size;
            }
        }
    }

    *result = sub_matrix;
    return MATRIX_SUCCESS;
}
```

Умножение матриц

```
int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result)
```

Для реализации функции применим итеративную формулу (2.3)

$$C_{i,j} = \begin{cases} \sum_{k=0}^{n-1} A_{i,k} \cdot B_{k,j}\,, & field = 0, \\ \sum_{k=0}^{n-1} \left(\left(A_{i,j}\ mod\ field\right) \cdot \left(B_{i,j}\ mod\ field\right) \right) mod\ field\,, & field \neq 0. \end{cases} \quad (2.3)$$

Где $i \in \overline{0, rows-1}$, $j \in \overline{0, cols-1}$, $k \in \overline{0, n-1}$, $A \in \mathrm{M}_{m\times n}(\mathbb{F})$, $B \in \mathrm{M}_{n\times p}(\mathbb{F})$, $C \in \mathrm{M}_{m\times p}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.9) соответствует функция:

```
int matrix_multiply0(const Matrix* A, const Matrix* B,
    Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, B->cols, A->field_size, &C);

    for (int i = 0; i < A->rows; i++)
    {
        for (int j = 0; j < B->cols; j++)
        {
            ULL sum = 0;

            for (int k = 0; k < A->cols; k++)
            {
                ULL term;
                if (A->field_size == 0)
                {
                    term = A->data[i][k] * B->data[k][j];
                    sum += term;
                }
                else
                {
```

```
                    term = (A->data[i][k] % A->field_size *
                            B->data[k][j] % A->field_size)
                            % A->field_size;
                    sum = (sum + term) % A->field_size;
                }
            }

            C->data[i][j] = sum;
        }
    }

    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.6).

Таблица 2.6 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
A	a	Указатель на первую исходную матрицу
B	b	Указатель на вторую исходную матрицу
R	result	Указатель на указатель матрицы результата
C	product	Указатель на создаваемую матрицу произведения
i	i	Индекс текущей строки матрицы A / результата
j	j	Индекс текущего столбца матрицы B / результата
k	k	Индекс суммируемого элемента для произведения
sum	sum	Накопленная сумма произведений в позиции (i,j)
term	term	Текущий элемент произведения A[i][k] * B[k][j]

```
int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result)
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if(a->rows < 1 || a->cols < 1 || b->rows < 1 || b->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    if (a->cols != b->rows)
    {
        return MATRIX_ERROR_DIMENSION;
    }

    if (a->field_size != b->field_size)
    {
        return MATRIX_ERROR_INVALID_FIELD;
    }

    int error;
    Matrix* product;
    error = matrix_create(a->rows, b->cols, a->field_size,
        &product);
    if (error != MATRIX_SUCCESS)
    {
```

```
    return error;
}

for (int i = 0; i < a->rows; i++)
{
    for (int j = 0; j < b->cols; j++)
    {
        ULL sum = 0;

        for (int k = 0; k < a->cols; k++)
        {
            ULL term;

            if (a->field_size == 0)
            {
                if ((a->data[i][k] != 0) && (
                    b->data[k][j] > ULLONG_MAX /
                    a->data[i][k]))
                {
                    matrix_free(&product);
                    return MATRIX_ERROR_OVERFLOW;
                }

                term = a->data[i][k] * b->data[k][j];

                if (sum > ULLONG_MAX - term)
                {
                    matrix_free(&product);
                    return MATRIX_ERROR_OVERFLOW;
                }

                sum += term;
            }
            else
            {
```

```
                    error = multiply_mod(a->data[i][k],
                        b->data[k][j],a->field_size, &term);
                    if (error != MATRIX_SUCCESS)
                    {
                        matrix_free(&product);
                        return error;
                    }

                    sum = (sum + term) % a->field_size;
                }
            }

            product->data[i][j] = sum;
        }
    }

    *result = product;
    return MATRIX_SUCCESS;
}
```

Умножение на скаляр

```
int matrix_scalar_multiply(const Matrix* a, ULL scalar,
    Matrix** result)
```

Для реализации функции применим итеративную формулу (2.4):

$$\alpha \cdot C_{i,j} = \begin{cases} \sum_{k=0}^{n-1} \alpha \cdot A_{i,k}, & field = 0, \\ \sum_{i=0}^{n-1} \Big((\alpha \bmod field) \cdot \big(A_{i,j} \bmod field\big) \Big) \bmod field, & field \neq 0. \end{cases} \quad (2.4)$$

Где $i \in \overline{0, rows-1}$, $j \in \overline{0, cols-1}$; $k \in \overline{0, n-1}$; $A, C \in \mathrm{M}_{m \times n}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.4) соответствует функция:

```
int matrix_scalar_multiply0(const Matrix* A, ULL s, Matrix** R)
{
    Matrix* C;
```

```
    matrix_create(A->rows, A->cols, A->field_size, &C);

    for (int i = 0; i < A->rows; i++)
    {
        for (int j = 0; j < A->cols; j++)
        {
            ULL x = A->data[i][j];
            ULL r;

            if (A->field_size == 0)
            {
                r = x * s;
            }
            else
            {
                r = (x % A->field_size * s % A->field_size)
                % A->field_size;
            }

            C->data[i][j] = r;
        }
    }

    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.7).

Таблица 2.7 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
A	a	Указатель на исходную матрицу

s	scalar	Скалярный множитель
R	result	Указатель на указатель матрицы результата
C	scaled_matrix	Указатель на создаваемую матрицу результата
i	i	Индекс текущей строки матрицы
j	j	Индекс текущего столбца матрицы
x	x	Элемент исходной матрицы в позиции (i,j)
r	res	Вычисленное значение элемента после умножения

```
int matrix_scalar_multiply(const Matrix* a, ULL scalar,
    Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    int error;
    Matrix* scaled_matrix;
    error = matrix_create(a->rows, a->cols, a->field_size,
        &scaled_matrix);
    if (error != MATRIX_SUCCESS)
```

```
{
    return error;
}

for (int i = 0; i < a->rows; i++)
{
    for (int j = 0; j < a->cols; j++)
    {
        ULL x = a->data[i][j];
        ULL res;

        if (a->field_size == 0)
        {
            if (x != 0 && scalar > ULLONG_MAX / x)
            {
                matrix_free(&scaled_matrix);
                return MATRIX_ERROR_OVERFLOW;
            }
            res = x * scalar;
        }
        else
        {
            error = multiply_mod(x, scalar,
                a->field_size, &res);
            if (error != MATRIX_SUCCESS)
            {
                matrix_free(&scaled_matrix);
                return error;
            }
        }

        scaled_matrix->data[i][j] = res;
    }
}
*result = scaled_matrix;
```

```
    return MATRIX_SUCCESS;
}
```

Транспонирование матриц

```
int matrix_transpose(const Matrix* a, Matrix** result)
```

Для реализации функции применим итеративную формулу (2.5)

$$C_{j,i} = A_{i,j}, \tag{2.5}$$

Где $i \in \overline{0, cols - 1}, j \in \overline{0, rows - 1}; A, C \in \mathrm{M}_{m \times n}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.9) соответствует функция:

```
int matrix_transpose0(const Matrix* A, Matrix** R)
{
    Matrix* C;
    matrix_create(A->cols, A->rows, A->field_size, &C);

    for (int i = 0; i < A->rows; i++)
    {
        for (int j = 0; j < A->cols; j++)
        {
            C->data[j][i] = A->data[i][j];
        }
    }

    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.8).

Таблица 2.8 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
`A`	`a`	Указатель на исходную матрицу
`R`	`result`	Указатель на указатель матрицы результата

C	transposed	Указатель на создаваемую транспонированную матрицу
i	i	Индекс текущей строки матрицы A
j	j	Индекс текущего столбца матрицы A

```
int matrix_transpose(const Matrix* a, Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    int error;
    Matrix* transposed;
    error = matrix_create(a->cols, a->rows, a->field_size,
        &transposed);
    if (error != MATRIX_SUCCESS)
    {
        return error;
    }

    for (int i = 0; i < a->rows; i++)
    {
        for (int j = 0; j < a->cols; j++)
```

```
        {
            transposed->data[j][i] = a->data[i][j];
        }
    }

    *result = transposed;
    return MATRIX_SUCCESS;
}
```

Нахождение подматрицы (через левую, правую, верхнюю и нижнюю границу)

```
int matrix_submatrix(const Matrix a, int start_row, int end_row, int start_col, int end_col, Matrix* result)
```

Для реализации функции применим итеративную формулу (2.6):

$$C_{i,j} = A_{startrow+i,startcol+j}, \tag{2.6}$$

Где $i \in \overline{0, endrows - startrows}$, $j \in \overline{0, endcols - startcols}$;

$A \in \mathrm{M}_{m \times n}(\mathbb{F}), C \in \mathrm{M}_{(endrows-startrows+1)\times(endcols-startcols+1)}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.6) соответствует функция:

```
int matrix_submatrix0(const Matrix* A, int sr, int er, int sc,
    int ec, Matrix** R)
{
    int sub_rows = er - sr + 1;
    int sub_cols = ec - sc + 1;
    Matrix* C;
    matrix_create(sub_rows, sub_cols, A->field_size, &C);

    for (int i = 0; i < sub_rows; i++)
    {
        for (int j = 0; j < sub_cols; j++)
        {
            C->data[i][j] = A->data[sr + i][sc + j];
        }
    }

    *R = C;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.9).

Таблица 2.9 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
A	a	Указатель на исходную матрицу
R	result	Указатель на указатель матрицы результата
C	submatrix	Указатель на создаваемую подматрицу
sr	start_row	Верхняя граница подматрицы
er	end_row	Нижняя граница подматрицы
sc	start_col	Левая граница подматрицы
ec	end_col	Правая граница подматрицы
i	i	Индекс текущей строки подматрицы
j	j	Индекс текущего столбца подматрицы

```
int matrix_submatrix(const Matrix* a, int start_row,
    int end_row, int start_col, int end_col,
    Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }
    if (start_row < 0 || end_row >= a->rows || start_col < 0
        || end_col >= a->cols ||
        start_row > end_row || start_col > end_col)
    {
        return MATRIX_ERROR_DIMENSION;
    }
```

```
    int sub_rows = end_row - start_row + 1;
    int sub_cols = end_col - start_col + 1;

    int error;
    Matrix* submatrix;
    error = matrix_create(sub_rows, sub_cols, a->field_size,
        &submatrix);
    if (error != MATRIX_SUCCESS) return error;

    int i, j;
    for (i = 0; i < sub_rows; i++)
    {
        for (j = 0; j < sub_cols; j++)
        {
            submatrix->data[i][j] = a->data[start_row + i]
            [start_col + j];
        }
    }

    *result = submatrix;
    return MATRIX_SUCCESS;
}
```

Преобразование матрицы в строку

```
int matrix_to_string(const Matrix* matrix, char** result)
```

Для реализации функции применим итеративную формулу (2.7)

$$S = \sum_{i=0}^{rows-1} \sum_{j=0}^{cols-1} str(\mathrm{M}_{i,j}) + \begin{cases} ; & : j < cols - 1, \\ , & : i < rows - 1, \quad j = cols - 1, \\ `space` & : i = rows - 1, \quad j < cols - 1. \end{cases} \quad (2.7)$$

Где $i \in \overline{0, rows-1}, j \in \overline{0, cols-1}; S_{\text{строка}}$ соответствует $C \in \mathrm{M}_{m \times n}(\mathbb{F})$.

Согласно схеме итераций, формуле (2.7)(2.9) соответствует функция:

```
int matrix_to_string0(const Matrix* A, char** S)
{
```

```
int total_chars = 2;

for (int i = 0; i < A->rows; i++)
{
    for (int j = 0; j < A->cols; j++)
    {
        ULL num = A->data[i][j];
        if (A->field_size != 0)
            num %= A->field_size;

        int digits = (num == 0) ? 1 : 0;
        ULL t = num;
        while (t > 0)
        {
            digits++;
            t /= 10;
        }

        total_chars += digits;
        if (j < A->cols - 1) total_chars++;
    }

    if (i < A->rows - 1) total_chars++;
}

total_chars++;
char* res = malloc(total_chars);
if (!res) return STRING_ERROR_CONVERSION;

int pos = 0;
res[pos++] = '(';

for (int i = 0; i < A->rows; i++)
{
    for (int j = 0; j < A->cols; j++)
```

```
        {
            ULL num = A->data[i][j];
            if (A->field_size != 0)
                num %= A->field_size;

            pos += sprintf(res + pos, "%llu", num);

            if (j < A->cols - 1)
                res[pos++] = ',';
        }

        if (i < A->rows - 1)
            res[pos++] = ';';
    }

    res[pos++] = ')';
    res[pos] = '\0';
    *S = res;
    return 0;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.10).

Таблица 2.10 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
m	matrix	Указатель на исходную матрицу
s	result	Указатель на результирующую строку
n	num	Значение текущего элемента матрицы
r	str_result	Буфер для сборки строки
p	pos	Текущая позиция записи в строке

d	digit_count	Количество цифр текущего числа
tmp	temp	Временная копия числа для обработки цифр
buf	buffer	Промежуточный буфер для хранения цифр числа
bp	buf_pos	Позиция записи в промежуточном буфере
dig	digits	Массив цифр числа в обратном порядке
dc	digits_count	Количество цифр, записанных в массив dig

```
int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result)
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if(a->rows < 1 || a->cols < 1 || b->rows < 1 || b->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    if (a->cols != b->rows)
    {
        return MATRIX_ERROR_DIMENSION;
    }

    if (a->field_size != b->field_size)
```

```
{
    return MATRIX_ERROR_INVALID_FIELD;
}

int error;
Matrix* product;
error = matrix_create(a->rows, b->cols, a->field_size,
    &product);
if (error != MATRIX_SUCCESS)
{
    return error;
}

for (int i = 0; i < a->rows; i++)
{
    for (int j = 0; j < b->cols; j++)
    {
        ULL sum = 0;

        for (int k = 0; k < a->cols; k++)
        {
            ULL term;

            if (a->field_size == 0)
            {
                if ((a->data[i][k] != 0) && (
                    b->data[k][j] > ULLONG_MAX /
                    a->data[i][k]))
                {
                    matrix_free(&product);
                    return MATRIX_ERROR_OVERFLOW;
                }

                term = a->data[i][k] * b->data[k][j];
```

```
                    if (sum > ULLONG_MAX - term)
                    {
                        matrix_free(&product);
                        return MATRIX_ERROR_OVERFLOW;
                    }

                    sum += term;
                }
                else
                {
                    error = multiply_mod(a->data[i][k],
                        b->data[k][j],a->field_size, &term);
                    if (error != MATRIX_SUCCESS)
                    {
                        matrix_free(&product);
                        return error;
                    }

                    sum = (sum + term) % a->field_size;
                }
            }

            product->data[i][j] = sum;
        }
    }

    *result = product;
    return MATRIX_SUCCESS;
}
```

Преобразование строки в матрицу

```
int string_to_matrix(const char* str, ULL field_size,
        Matrix** result)
```

Для реализации функции применим итеративную формулу (2.8):

$$C_{i,j} = \boldsymbol{S}[p_{i,j}], \tag{2.8}$$

Где $i \in \overline{0, rows-1}, j \in \overline{0, cols-1}; C \in M_{k\times k}(\mathbb{F}); p_{i,j}$ — индекс в строке, на котором начинается число матрицы $C_{i,j}$, $\boldsymbol{S}[p_{i,j}]$ — число в строке, которое находится по индексу $p_{i,j}$.

Согласно схеме итераций, формуле (2.8) соответствует функция:

```
int string_to_matrix0(const char* input_str, ULL field_size,
    Matrix** matrix)
{
    if (!input_str)
        return STRING_ERROR_NULL_POINTER;

    int rows = 1, cols = 1;
    for (int i = 1; i < (int)strlen(input_str) - 1; i++)
    {
        if (input_str[i] == ';') rows++;
        else if (input_str[i] == ',' && rows == 1) cols++;
    }

    int error = matrix_create(rows, cols, field_size, matrix);
    if (error != MATRIX_SUCCESS)
        return STRING_ERROR_CONVERSION;

    int row_index = 0, col_index = 0;
    char buffer[32];
    int buffer_pos = 0;
    int inside_parens = 0;

    for (int i = 0; i < (int)strlen(input_str); i++)
    {
        char c = input_str[i];
        if (c == '(') { inside_parens = 1; continue; }
```

```
    else if (c == ')') { inside_parens = 0; break; }
    if (!inside_parens) continue;

    if (c == ',' || c == ';' || c == ')')
    {
        if (buffer_pos > 0)
        {
            buffer[buffer_pos] = '\0';
            ULL value = strtoull(buffer, NULL, 10);
            if (field_size != 0) value %= field_size;

            (*matrix)->data[row_index][col_index] = value;
            buffer_pos = 0;
            col_index++;
        }

        if (c == ';') { row_index++; col_index = 0; }
    }
    else if (c != ' ')
    {
        if (buffer_pos >= 31)
        {
            matrix_free(*matrix);
            *matrix = NULL;
            return STRING_ERROR_BUFFER_OVERFLOW;
        }
        buffer[buffer_pos++] = c;
    }
}

if (buffer_pos > 0)
{
```

```
            buffer[buffer_pos] = '\0';
            ULL value = strtoull(buffer, NULL, 10);
            if (field_size != 0) value %= field_size;
            (*matrix)->data[row_index][col_index] = value;
        }

        return STRING_SUCCESS;
}
```

При написании окончательной версии функции используются следующие целочисленные переменные (Таблица 2.11).

Таблица 2.11 – Переменные, используемые для реализации функции

Старое название	Новое название	Описание
input_str	str_input	Входная строка с матрицей
matrix	matrix	Указатель на результирующую матрицу
field_size	field_size	Размер конечного поля для операции по модулю
rows	rows_count	Количество строк в матрице
cols	cols_count	Количество столбцов в матрице
buffer	buffer	Буфер для накопления цифр числа
buffer_pos	buf_index	Текущая позиция в буфере
row_index	row	Индекс текущей строки
col_index	col	Индекс текущего столбца
inside_parens	inside_parentheses	Флаг нахождения внутри скобок
value	value	Текущее число для записи в матрицу

```
int string_to_matrix(const char* str, ULL field_size,
    Matrix** result)
```

```
{
    if (!str)
    {
        return STRING_ERROR_NULL_POINTER;
    }
    if (strlen(str) < 3)
    {
        return STRING_ERROR_INVALID_FORMAT;
    }

    int rows = 1;
    int cols = 1;

    for (int i = 1; i < (int)strlen(str) - 1; i++)
    {
        if (str[i] == ';')
        {
            rows++;
        }
        else if (str[i] == ',' && rows == 1)
        {
            cols++;
        }
    }

    int matrix_error;
    matrix_error = matrix_create(rows, cols, field_size,
        result);
    if (matrix_error != MATRIX_SUCCESS)
    {
        return STRING_ERROR_CONVERSION;
    }

    int row = 0, col = 0;
    char buffer[32];
    int buf_index = 0;
    int inside_parentheses = 0;

    for (int i = 0; i < (int)strlen(str); i++)
    {
        if (str[i] == '(')
        {
            inside_parentheses = 1;
            continue;
        }
        else if (str[i] == ')')
        {
            inside_parentheses = 0;
            break;
        }

        if (!inside_parentheses) continue;
```

```
        if (str[i] == ';' || str[i] == ',' || str[i] == ')')
        {
            if (buf_index > 0)
            {
                buffer[buf_index] = '\0';
                ULL value = strtoull(buffer, NULL, 10);
                if (field_size != 0)
                {
                    value %= field_size;
                }
                (*result)->data[row][col] = value;

                buf_index = 0;
                col++;
            }

            if (str[i] == ';')
            {
                row++;
                col = 0;
            }
        }
        else if (str[i] != ' ')
        {
            if (buf_index >= 31)
            {
                matrix_free(*result);
                *result = NULL;
                return STRING_ERROR_BUFFER_OVERFLOW;
            }
            buffer[buf_index++] = str[i];
        }
    }

    if (buf_index > 0)
    {
        buffer[buf_index] = '\0';
        ULL value = strtoull(buffer, NULL, 10);
        if (field_size != 0)
        {
            value %= field_size;
        }
        (*result)->data[row][col] = value;
    }

    return STRING_SUCCESS;
}
```

2.2 Реализация исследуемого алгоритма

Согласно заданию, исследование проводится для матриц с элементами в диапазоне от 2^{19} до $2^{20}-1$, поэтому при реализации алгоритма быстрого возведения матрицы в степень можно использовать встроенный в C тип данных `unsigned long long` и операции над переменными этого типа. Перечень используемых операций с типом `unsigned long long` приведен в разделе «ПРИЛОЖЕНИЕ Б. Используемые функции».

Для реализации быстрого возведения матрицы в степень применим рекурсивную формулу

$$\mathrm{P}(A,n)=\begin{cases} E_k, & n=0, \\ A, & n=1, \\ \left(P\left(A,\frac{n}{2}\right)\right)^2, & n>1 \text{ и } n_i \bmod 2=0 \\ A\cdot\left(P\left(A,\frac{n}{2}\right)\right)^2, & n>1 \text{ и } n_i \bmod 2=1 \end{cases} \tag{2.9}$$

Согласно схеме итераций [3, с. 8-9], формуле (2.9) соответствует функция

```
int matrix_power0(const Matrix* A, ULL n, Matrix** R)
{
    if (!A || !R)
        return MATRIX_ERROR_NULL_POINTER;

    if (A->rows != A->cols)
        return MATRIX_ERROR_NOT_SQUARE;

    if (*R != NULL)
        matrix_free(R);

    int err;
    Matrix* result = NULL;
    Matrix* temp = NULL;
```

```
if (n == 0)
{
    err = matrix_create(A->rows, A->cols, A->field_size,
        &result);
    if (err != MATRIX_SUCCESS) return err;
    for (int i = 0; i < A->rows; i++)
        result->data[i][i] = 1;

    *R = result;
    return MATRIX_SUCCESS;
}

if (n == 1)
    return matrix_copy(A, R);

err = matrix_create(A->rows, A->cols, A->field_size,
    &result);
if (err != MATRIX_SUCCESS) return err;
for (int i = 0; i < A->rows; i++)
    result->data[i][i] = 1;

err = matrix_copy(A, &temp);
if (err != MATRIX_SUCCESS)
{
    matrix_free(&result);
    return err;
}

ULL exp = n;
while (exp > 0)
{
```

```
    if (exp & 1)
    {
        Matrix* new_result = NULL;
        err = matrix_multiply(result, temp, &new_result);
        if (err != MATRIX_SUCCESS)
        {
            matrix_free(&result);
            matrix_free(&temp);
            return err;
        }
        matrix_free(&result);
        result = new_result;
    }

    exp >>= 1;

    if (exp > 0)
    {
        Matrix* new_temp = NULL;
        err = matrix_multiply(temp, temp, &new_temp);
        if (err != MATRIX_SUCCESS)
        {
            matrix_free(&result);
            matrix_free(&temp);
            return err;
        }
        matrix_free(&temp);
        temp = new_temp;
    }
}

matrix_free(&temp);
```

```
    *R = result;
    return MATRIX_SUCCESS;
}
```

При написании функции `matrix_power0` используются следующие переменные (Таблица 2.12).

Таблица 2.12 – Переменные, используемые для реализации алгоритма

Название	Описание
`A`	Входная матрица — основание возведения в степень, соответствует A_0 в рекуррентной формуле (2.1).
`n`	Показатель степени, соответствует e_0 в рекуррентной формуле (2.1).
`R`	Указатель на результирующую матрицу, в которую записывается A^n.
`err`	Код ошибки, возвращаемый функциями работы с матрицами (`matrix_create, matrix_multiply, matrix_copy`).
`result`	Текущая накопленная матрица результата A_i, соответствующая частичному произведению в итеративной форме (2.1).
`temp`	Промежуточная матрица степени P_i, используемая для возведения в квадрат на каждом шаге алгоритма.
`exp`	Рабочая копия показателя степени e_i, уменьшающаяся при каждом цикле делением на 2 (побитовым сдвигом вправо).
`new_result`	Временная матрица для хранения промежуточного произведения при умножении `result * temp`.
`new_temp`	Временная матрица для хранения квадрата матрицы `temp * temp` на следующем шаге итерации.

2.3 Тестирование реализации алгоритма

Функция `manual_test` позволяет проверить корректность работы функции `matrix_power` для нескольких заранее известных матриц и показателей степени (Таблица 2.15).

Таблица 2.15 - Граничные значения для проверки в функции `matrix_power0`

№	Матрица	n	$\mathbb{F}$	Ожидаемый результат A^n
1	$\begin{pmatrix}1 & 2\\3 & 4\end{pmatrix}$.	2	100	$\begin{pmatrix}7 & 10\\15 & 22\end{pmatrix}$
2	$\begin{pmatrix}1 & 1\\1 & 0\end{pmatrix}$.	10	100	$\begin{pmatrix}89 & 55\\55 & 34\end{pmatrix}$
3	$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$.	5	100	$\begin{pmatrix}1 & 0\\0 & 1\end{pmatrix}$

```
int manual_test()
{
    printf("=== ТЕСТИРОВАНИЕ С ИЗВЕСТНЫМИ ДАННЫМИ ===\n");
    printf("\nТест 1: Матрица 2x2 в степени 2\n");
    Matrix* m1;
    int str_error = string_to_matrix("(1,2;3,4)", 100, &m1);
    if (str_error == STRING_SUCCESS)
    {
        Matrix* r1;
        int mat_error = matrix_power(m1, 2, &r1);
        if (mat_error == MATRIX_SUCCESS)
        {
            matrix_print(m1);
            printf("^2 =\n");
            matrix_print(r1);
            matrix_free(r1);
        }
        else
```

```
    {
        printf("Ошибка создания матрицы: %s\n",
            get_string_error_message(str_error));
    }
    matrix_free(m1);
}
else
{
    printf("Ошибка создания матрицы: %s\n",
        get_string_error_message(str_error));
}
printf("\nТест 2: Матрица 2x2 в степени 10\n");
Matrix* m2;
str_error = string_to_matrix("(1,1;1,0)", 100, &m2);
if (str_error == STRING_SUCCESS)
{
    Matrix* r2;
    int mat_error = matrix_power(m2, 10, &r2);
    if (mat_error == MATRIX_SUCCESS)
    {
        matrix_print(m2);
        printf("^10 =\n");
        matrix_print(r2);
        matrix_free(r2);
    }
    else
    {
        printf("Ошибка создания матрицы: %s\n",
            get_string_error_message(str_error));
    }
    matrix_free(m2);
}
else
{
    printf("Ошибка создания матрицы: %s\n",
```

```
            get_string_error_message(str_error));
    }
    printf("\nТест 3: Единичная матрица в степени 5\n");
    Matrix* m3;
    str_error = string_to_matrix("(1,0;0,1)", 100, &m3);
    if (str_error == STRING_SUCCESS)
    {
        Matrix* r3;
        int mat_error = matrix_power(m3, 5, &r3);
        if (mat_error == MATRIX_SUCCESS)
        {
            matrix_print(m3);
            printf("^5 =\n");
            matrix_print(r3);
            matrix_free(r3);
        }
        else
        {
            printf("Ошибка создания матрицы: %s\n",
                get_string_error_message(str_error));
        }
        matrix_free(m3);
    }
    else
    {
        printf("Ошибка создания матрицы: %s\n",
            get_string_error_message(str_error));
    }
    return UI_SUCCESS;
}
```

Результаты работы данной функции приведены ниже:

```
=== ТЕСТИРОВАНИЕ С ИЗВЕСТНЫМИ ДАННЫМИ ===
Тест 1: Матрица 2x2 в степени 2
```

```
Matrix 2x2 (field size: 100):
  1 2
  3 4
^2 =
Matrix 2x2 (field size: 100):
  7 10
  15 22
Тест 2: Матрица 2x2 в степени 10
Matrix 2x2 (field size: 100):
  1 1
  1 0
^10 =
Matrix 2x2 (field size: 100):
  89 55
  55 34
Тест 3: Единичная матрица в степени 5
Matrix 2x2 (field size: 100):
  1 0
  0 1
^5 =
Matrix 2x2 (field size: 100):
  1 0
  0 1
```

Во-вторых, для проверки других матриц и показателей степени реализована функция `input_test`, которая при вызове запрашивает у пользователя ввод параметров: размер матрицы, размер конечного поля, показатель степени и саму матрицу в формате `(a11,a12,...;a21,a22,...)`.

```
int input_test()
{
    printf("=== РУЧНОЕ ТЕСТИРОВАНИЕ ===\n");

    char buffer[32767];
```

```
int size;
ULL exponent, field_size;

printf("Введите размер матрицы:");
if (scanf("%d", &size) != 1 || size <= 0)
{
    printf("Ошибка ввода размера матрицы\n");
    return UI_ERROR_INPUT;
}

printf("Введите размер конечного поля:");
if (scanf("%llu", &field_size) != 1 || field_size == 0)
{
    printf("Ошибка ввода размера поля\n");
    return UI_ERROR_INPUT;
}

printf("Введите степень:");
if (scanf("%llu", &exponent) != 1)
{
    printf("Ошибка ввода степени\n");
    return UI_ERROR_INPUT;
}
getchar();
printf("Введите матрицу в формате (a11,a12,...;a21,"
       "a22,...):");
if (fgets(buffer, sizeof(buffer), stdin) == NULL)
{
    printf("Ошибка ввода матрицы\n");
    return UI_ERROR_INPUT;
}
buffer[strcspn(buffer, "\n")] = 0;
Matrix* matrix;
int string_error = string_to_matrix(buffer, field_size,
    &matrix);
```

```
if (string_error != STRING_SUCCESS)
{
    printf("Ошибка преобразования строки в матрицу: %s\n",
        get_string_error_message(string_error));
    return UI_ERROR_INPUT;
}
printf("\nИсходная матрица:\n");
matrix_print(matrix);
int64_t t0 = 0, t1 = 0;
if (get_time_ns(&t0) != 0) t0 = 0;
Matrix* R = NULL;
int pow_err = matrix_power(matrix, exponent, &R);
if (get_time_ns(&t1) != 0) t1 = t0;
ULL dt_ns = t1 - t0;
if (pow_err == MATRIX_SUCCESS)
{
    printf("\nРезультат возведения в степень %llu:\n",
        exponent);
    matrix_print(R);
    char* result_str;
    string_error = matrix_to_string(R, &result_str);
    if (string_error == STRING_SUCCESS)
    {
        printf("\nРезультат в строковом формате: %s\n",
            result_str);
        free(result_str);
    }
    printf("Время выполнения: %llu наносекунд\n", dt_ns);
    matrix_free(R);
}
else
{
    printf("Ошибка возведения в степень: %s\n",
        get_matrix_error_message(pow_err));
}
```

```
    matrix_free(matrix);
    return UI_SUCCESS;
}
```

2.4 Проведение эксперимента

Для проведения эксперимента реализована окончательная версия функции алгоритма быстрого возведения квадратной матрицы в степень `matrix_power`, с поддержкой проверки корректности входных данных и работы с конечным полем. Также реализована функция `generate_test_cases`, предназначенная для автоматической генерации и проверки тестовых случаев.

```
int matrix_power(const Matrix* base, ULL exponent,
    Matrix** result)
{
    if (!base || !result)
        return MATRIX_ERROR_NULL_POINTER;

    if (base->rows != base->cols)
        return MATRIX_ERROR_NOT_SQUARE;

    if (*result != NULL)
    {
        matrix_free(result);
    }

    int error;
    Matrix* res_matrix = NULL;
    Matrix* temp_power = NULL;

    if (exponent == 0)
    {
        error = matrix_create(base->rows, base->cols,
```

```
            base->field_size, &res_matrix);
        if (error != MATRIX_SUCCESS)
            return error;

        for (int i = 0; i < base->rows; i++)
            res_matrix->data[i][i] = 1;

        *result = res_matrix;
        return MATRIX_SUCCESS;
    }

    if (exponent == 1)
        return matrix_copy(base, result);

    error = matrix_create(base->rows, base->cols, base-
>field_size,
        &res_matrix);
    if (error != MATRIX_SUCCESS)
        return error;

    for (int i = 0; i < base->rows; i++)
        res_matrix->data[i][i] = 1;

    error = matrix_copy(base, &temp_power);
    if (error != MATRIX_SUCCESS)
    {
        matrix_free(&res_matrix);
        return error;
    }

    ULL exp = exponent;
    while (exp > 0)
    {
        if (exp & 1)
        {
```

```
            Matrix* temp_result = NULL;
            error = matrix_multiply(res_matrix, temp_power,
                &temp_result);
            if (error != MATRIX_SUCCESS)
            {
                matrix_free(&res_matrix);
                matrix_free(&temp_power);
                return error;
            }
            matrix_free(&res_matrix);
            res_matrix = temp_result;
        }

        exp >>= 1;

        if (exp > 0)
        {
            Matrix* temp_square = NULL;
            error = matrix_multiply(temp_power, temp_power,
                &temp_square);
            if (error != MATRIX_SUCCESS)
            {
                matrix_free(&res_matrix);
                matrix_free(&temp_power);
                return error;
            }
            matrix_free(&temp_power);
            temp_power = temp_square;
        }
    }

    matrix_free(&temp_power);
    *result = res_matrix;
    return MATRIX_SUCCESS;
}
```

```
int generate_test_cases(const char* filename, int min_size,
    int max_size, int num_tests, ULL min_exponent,
    ULL max_exponent, ULL field_size)
{
    if (!filename)
        return TEST_ERROR_FILE_WRITE;
    if (min_size <= 0 || max_size <= 0 || min_size > max_size)
        return TEST_ERROR_INVALID_PARAMS;
    if (num_tests <= 0)
        return TEST_ERROR_INVALID_PARAMS;
    if (min_exponent > max_exponent)
        return TEST_ERROR_INVALID_PARAMS;

    FILE* csv = fopen(filename, "w");
    if (!csv) return TEST_ERROR_FILE_WRITE;

    FILE* short_out = fopen("output-short.txt", "w");
    if (!short_out)
    {
        fclose(csv);
        return TEST_ERROR_FILE_WRITE;
    }

    fprintf(csv, "matrix_size,exponent,field_size,"
                 "computation_time_ns\n");
    fprintf(short_out, "matrix_size exponent field_size "
                       "computation_time_ns\n");

    srand((unsigned)time(NULL));
    static int count_tests = 1;
    int successful_tests = 0;

    for (int test_idx = 0; test_idx < num_tests; test_idx++)
    {
        int size = min_size + (rand32() % (max_size - min_size
```

```
    + 1));
ULL exponent = min_exponent;
if (min_exponent != max_exponent)
{
    exponent = min_exponent + (rand64() % (max_exponent
        - min_exponent + 1));
}

Matrix* M = NULL;
int create_err = generate_random_matrix(size,
    field_size, &M);
if (create_err != MATRIX_SUCCESS)
{
    printf("\nFailed to create matrix: %s\n",
        get_matrix_error_message(create_err));
    continue;
}

char* matrix_str = NULL;
if (matrix_to_string(M, &matrix_str) != STRING_SUCCESS)
    matrix_str = strdup("SERIALIZE_ERROR");

int64_t t0 = 0, t1 = 0;
if (get_time_ns(&t0) != 0) t0 = 0;
Matrix* R = NULL;
int pow_err = matrix_power(M, exponent, &R);
if (get_time_ns(&t1) != 0) t1 = t0;
ULL dt_ns = t1 - t0;

char* result_str = NULL;
if (pow_err == MATRIX_SUCCESS)
{
    if (matrix_to_string(R, &result_str) !=
        STRING_SUCCESS)
        result_str = strdup("SERIALIZE_ERROR");
```

```
        }
        else
        {
            const char* msg = get_matrix_error_message(pow_err);
            result_str = strdup(msg ? msg : "POWER_ERROR");
        }

        printf("%6d %6d %12llu %12llu %12lld\n", count_tests,
            size, exponent, field_size, dt_ns);

        fprintf(short_out, "%d %llu %llu %llu\n", size,
            exponent, field_size, dt_ns);

        fprintf(csv, "%d,%llu,%llu,%lld\n",
                size, exponent, field_size,
                dt_ns);

        free(matrix_str);
        free(result_str);
        if (R) matrix_free(&R);
        matrix_free(&M);
        count_tests++;
        successful_tests++;
    }

    fclose(csv);
    fclose(short_out);

    if (successful_tests == 0)
        return TEST_ERROR_GENERATION;

    printf("Generated and ran %d tests (output: %s and "
           "output-short.txt)\n", successful_tests, filename);
    return TEST_SUCCESS;
}
```

Для организации эксперимента реализована функция `file_operations_test`, предназначенная для выбора режима генерации тестов и создания файлов с тестовыми данными. Функция позволяет пользователю выбрать один из двух режимов генерации:

1. Фиксация по степени — все тестовые матрицы возводятся в одну и ту же степень, а размеры матриц выбираются случайным образом в заданном диапазоне.

2. Фиксация по размеру матрицы — все тестовые матрицы имеют одинаковый размер, а показатель степени выбирается случайным образом в заданном диапазоне.

Пользователю последовательно предлагается ввести:

- количество тестов;
- фиксированное значение степени или размера матрицы в зависимости от выбранного режима;
- диапазон размеров матриц или показателей степени;
- размер конечного поля (0 — без модулей).

После ввода параметров вызывается функция `generate_test_cases`, которая создает тестовые матрицы, возводит их в указанные степени с помощью функции `matrix_power`, измеряет время выполнения и сохраняет результаты в CSV-файл `matrix_power_tests.csv` и TXT-файл `output-short.txt`.

Функция обеспечивает контроль корректности ввода и выводит сообщения об ошибках при нарушении условий (например, некорректный диапазон значений или ошибки генерации тестов).

```
int file_operations_test()
{
    printf("=== ВЫБОР РЕЖИМА ГЕНЕРАЦИИ ТЕСТОВ ===\n");
    printf("1) Фиксация по степени "
           "(случайный размер матрицы)\n");
    printf("2) Фиксация по размеру матрицы "
```

```
       "(случайная степень)\n");
printf("Выберите режим [1-2]:");

int mode = 0;
if (scanf("%d", &mode) != 1 || (mode != 1
    && mode != 2))
    return UI_ERROR_INPUT;
while (getchar() != '\n');

int min_size = 0, max_size = 0;
ULL min_exp = 0, max_exp = 0, static_size = 0,
field_size = 0;
int num_tests = 0;

printf("\nВведите количество тестов [1-10000]:");
if (scanf("%d", &num_tests) != 1 || num_tests < 1
    || num_tests > 10000)
    return UI_ERROR_INPUT;
while (getchar() != '\n');

if (mode == 1) // фиксированная степень
{
    printf("\nВведите фиксированную степень "
           "[1-1000000]:");
    if (scanf("%llu", &static_size) != 1 || static_size < 1
        || static_size > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');
    min_exp = max_exp = static_size;

    printf("\nВведите минимальный размер матрицы "
           "[1-1000000]:");
    if (scanf("%d", &min_size) != 1 || min_size < 1
        || min_size > 1000000)
        return UI_ERROR_INPUT;
```

```
    while (getchar() != '\n');

    printf("\nВведите максимальный размер матрицы "
           "[%d-1000000]:", min_size);
    if (scanf("%d", &max_size) != 1 || max_size < min_size
        || max_size > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');
}
else // фиксированный размер матрицы
{
    printf("\nВведите фиксированный размер матрицы "
           "[1-1000000]:");
    if (scanf("%llu", &static_size) != 1 || static_size < 1
        || static_size > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');
    min_size = max_size = static_size;

    printf("\nВведите минимальную степень [1-1000000]:");
    if (scanf("%llu", &min_exp) != 1 || min_exp < 1 ||
        min_exp > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');

    printf("\nВведите максимальную степень [%llu-1000000]:",
        min_exp);
    if (scanf("%llu", &max_exp) != 1 || max_exp < min_exp
        || max_exp > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');
}

ULL max_number = 1ULL << (EXP + 1);
```

```
    printf("\nВведите размер поля [0-%llu] (0 = без модулей):",
        max_number);
    if (scanf("%llu", &field_size) != 1)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');

    printf("\nНачало генерации тестов...\n");

    int test_error = generate_test_cases(
        "matrix_power_tests.csv", min_size, max_size,
        num_tests,min_exp, max_exp, field_size);

    if (test_error == TEST_SUCCESS)
    {
        printf("\nТестовые данные успешно сохранены в"
               " matrix_power_tests.csv\n");
    }
    else
    {
        printf("\nОшибка при генерации тестов: %d\n",
            test_error);
    }

    return UI_SUCCESS;
}
```

Перечень реализованных в процессе выполнения лабораторной работы функций приведен в разделе «ПРИЛОЖЕНИЕ В. Исходный код разработанной программы», а исходный код разработанной программы – в разделе «ПРИЛОЖЕНИЕ Г. Исходный код разработанной программы».

2.5 Обработка результатов эксперимента

Таблица 2.2 - Результаты экспериментов

Степень (размер = 64, поле = 1048576)	t_i, нс	t_i, с	$\alpha \approx \frac{t_{i+1}}{t_i} = \frac{f(n_{i+1})}{f(n_i)}$
10-99	173824326	0,174	-
100-999	281413124	0,281	1,618951331
1000-9999	380898743	0,381	1,353521604
10000-99999	483345803	0,483	1,268961402
100000-999999	574778256	0,575	1,189165712
1000000-9999999	686833406	0,687	1,194953704

По экспериментальны данным (Таблица 2.2) построена гистограмма распределения временных затрат алгоритма быстрого возведения матрицы в степень (Рисунок 2.1)

Рисунок 2.1 - Гистограмма распределения временных затрат алгоритма быстрого возведения матрицы в степень, полученная в результате экспериментов

Также при помощи встроенной в Excel анализатора данных, можно дополнить график линиями тренда: линейной, полиномиальной (3 степени) и степенной (Рисунок 2.2)

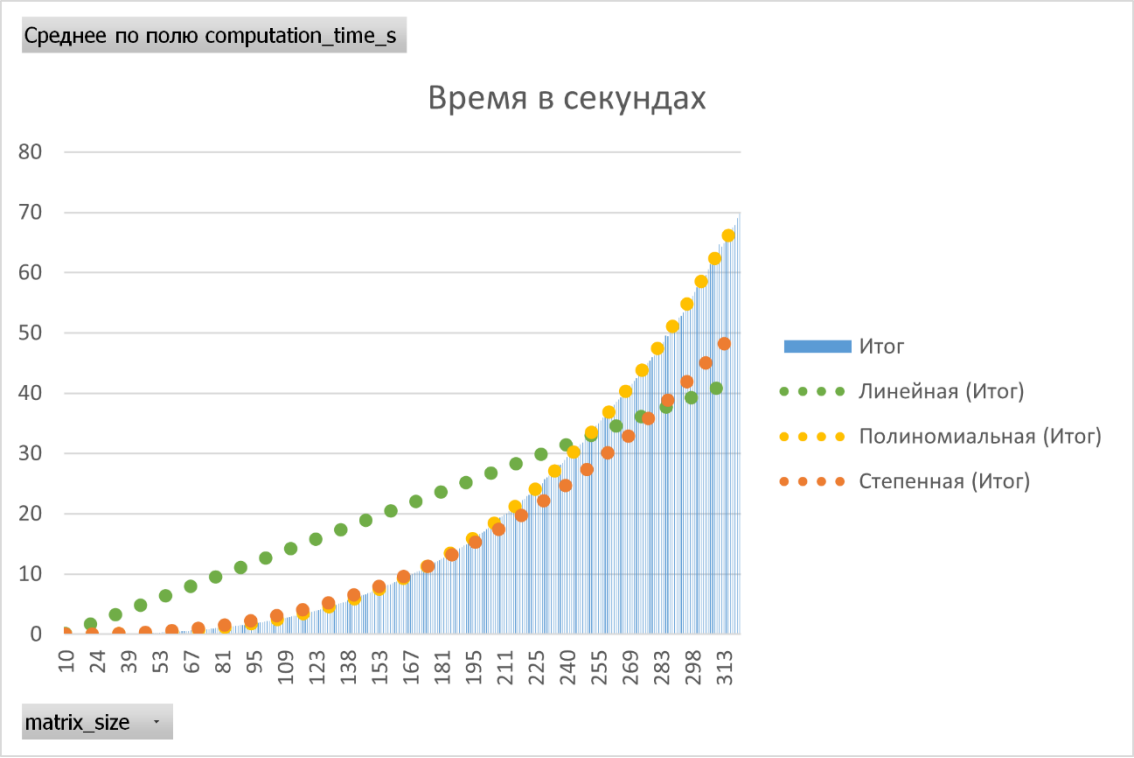

Рисунок 2.2 - Гистограмма распределения временных затрат алгоритма быстрого возведения матрицы в степень, полученная в результате экспериментов, дополненная линиями тренда

Отношения $\frac{t_{i+1}}{t_i}$ приблизительно постоянны и близки к значению $\alpha \approx 8$. Согласно методике [2], это свидетельствует о полиномиальном характере роста функции сложности $f(n)$. Поскольку размер входных данных удваивался ($n_{i+1} = 2n_i$), имеет место соотношение:

$$\frac{t_{i+1}}{t_i} = \frac{f(2n_i)}{f(n_i)} \approx 8$$

Предполагая, что $f(n) = n^a$, получаем:

$$\frac{f(2n_i)}{f(n_i)} = \frac{(2n_i)^a}{n_i^a} = 2^a \approx 8$$

$$2^a = 8 \Rightarrow a = \log_2 8 = 3$$

Экспериментально установлено, что средняя временная сложность алгоритма быстрого возведения квадратной матрицы в степень (при фиксации степени) относится к классу $O(n^3)$.

Этот результат полностью согласуется с теоретической оценкой, так как базовой операцией в алгоритме является умножение матриц, имеющее кубическую сложность $O(n^3)$, а количество таких операций в алгоритме быстрого возведения в степень не меняется (в силу фиксации степени), но логарифмически зависит от показателя степени, что в асимптотике не меняет доминирующую компоненту n^3.

ЗАКЛЮЧЕНИЕ

В ходе исследования алгоритма быстрого возведения квадратной матрицы в степень для размеров матриц n ∈ [10, 320] с элементами типа ULL из конечного поля $[2^{19}; 2^{20} - 1]$ получены следующие результаты:

Максимальное наблюдаемое время выполнения алгоритма для матрицы размера $n = 320$ составило 70,57 секунд, минимальное для матрицы размера $n = 10$ составило 0,0019 секунды.

Экспериментально установлено, что средняя временная сложность алгоритма (с фиксацией по степени) относится к классу $O(n^3)$. Это подтверждается тем, что при удвоении размера матрицы время выполнения возрастает приблизительно в 8 раз ($\alpha \approx 8$), что соответствует соотношению $\frac{(2n)^3}{n^3} = 8$.

СПИСОК ИСПОЛЬЗУЕМЫХ ИСТОЧНИКОВ

1. Marouf I., Asad M.M., Abu Al-Haija Q. Comparative Study of Efficient Modular Exponentiation Algorithms // COMPUSOFT, An international journal of advanced computer technology. – 2017. – Vol. 6, Issue VIII. – P. 2381-2389. – URL: https://www.researchgate.net/publication/320084242_Comparative_Study_of_Efficient_Modular_Exponentiation_Algorithms#read (дата обращения: 04.11.2025).
2. Абросимов И.К. АиСД 2025 ЛР 1 ПЗ.pdf – Яндекс Документы [Электронный ресурс] // Яндекс Диск: [сайт]. URL: disk.yandex.ru/i/wHghNib8Q_33Vw (дата обращения: 04.11.2025).
3. Абросимов И.К. АиСД 2025 02-03.pdf – Яндекс Документы [Электронный ресурс] // Яндекс Диск: [сайт]. URL: disk.yandex.ru/i/emuHbgfwxNa60w (дата обращения: 04.11.2025).

Белоусов И. В. МАТРИЦЫ И ОПРЕДЕЛИТЕЛИ: учебное пособие по линейной алгебре / И. В. Белоусов. – Кишинев: Институт прикладной физики, Академия наук Республики Молдова, 2006. – 2-е изд., исправл. и доп. – URL: https://eqworld.ipmnet.ru/ru/library/books/Belousov2006ru.pdf (дата обращения: 06.11.2025).

ПРИЛОЖЕНИЕ А. Доказательства используемых утверждений

Пусть A — квадратная матрица порядка k, а E_k— единичная матрица того же порядка. Обозначим через A^n произведение n матриц A, если $n \geq 1$, и положим $A^0 = E_k$.

Утверждение 1 (нулевая степень матрицы).

$$A^0 = E_k.$$

Доказательство.

Полное доказательство приведено в [4, с. 19-20].

Утверждение 2 (единичная степень матрицы).

$$A^1 = A.$$

Доказательство.

Полное доказательство приведено в [4, с. 19-20].

Утверждение 3 (свойство суммы показателей).

$$\mathrm{A^{m+n}} = \mathrm{A^m} \cdot \mathrm{A^n}, \qquad \mathrm{m, n} \geq 0.$$

Доказательство.

Полное доказательство приведено в [4, с. 19-20].

Утверждение 4 (свойство произведения показателей).

$$(A^m)^n = A^{m \cdot n}, \qquad m, n \geq 0.$$

Доказательство.

Полное доказательство приведено в [4, с. 19-20].

ПРИЛОЖЕНИЕ Б. Используемые функции

Используются нижеприведенные операторы для стандартных типов языка С, стандарт С11:

Оператор	Назначение
=	Присваивание: присваивает правое значение левой переменной.
==	Сравнение на равенство: возвращает ненулевое значение (true), если операнды равны, иначе 0.
!=	Сравнение на неравенство: возвращает ненулевое значение (true), если операнды различны, иначе 0.
>	Сравнение "больше": 1, если левое больше правого, иначе 0.
<	Сравнение "меньше": 1, если левое меньше правого, иначе 0.
+	Сложение: возвращает сумму двух чисел
-	Вычитание и унарный минус: используется для вычитания и как знак отрицательного числа.
*	Умножение
/	Целочисленное деление
%	Остаток от деления
+=	Сложение с присваиванием

Используются нижеприведенные функции из заголовочного файла <cstdlib.h>.

`int rand()`
Возвращает псевдослучайное число в диапазоне от 0 до RAND_MAX.

`void srand(unsigned int seed)`	
Инициализирует генератор псевдослучайных чисел `rand()`	
`seed`	инициализатор, значение которого однозначно определяет последовательность генерируемых `rand()` псевдослучайных чисел

`void* malloc(size_t size)`	
Выделяет `size` байт динамической памяти и возвращает указатель на начало. В коде используется для выделения массивов `numbers` и `bitmap`.	
`size`	инициализатор, значение которого выделяет память

`void free(void* ptr)`	
Освобождает ранее выделенную память, указанную в `ptr`. В коде используется для освобождения `numbers` и `bitmap`.	
`*ptr`	инициализатор, значение которого является объектом выделенной при помощи ф-ции `malloc`, который необходимо освободить

ПРИЛОЖЕНИЕ В. Разработанные функции

Функции, необходимые для реализации исследуемого алгоритма.

`int Multiply(ULL a, ULL b, ULL *Result)`	
Умножает a на b методом сложения и сдвигов (алгоритм «умножение через двоичное разложение»).	
`a`	инициализатор, первый множитель
`b`	инициализатор, второй множитель
`*Result`	указатель на результат
`Multiply_OK (0)`	Функция завершила свою работу без ошибок
`ERROR_Multiply_NullPointer(1)`	Нулевой указатель в аргументе ф-ции

`uint64_t rand64()`
Формирует 64-битное псевдослучайное значение путём объединения нескольких вызовов `rand()`

`int main()`
Основная функция: создаёт/проверяет `input.txt, output.txt, output-excel.csv`, выполняет ручной ввод/тесты, для каждого числа из `input.txt` вызывает kollats и записывает результаты в `output.txt` и `output-excel.csv`.

Функции, необходимые для тестирования исследуемого алгоритма.

`int generate_unique_numbers_file()`
Создаёт (по запросу пользователя) файл `input.txt` с `NUM_NUMBERS` уникальными случайными числами в диапазоне `[2^K, 2^(K+1))`. Используется bitmap-индексы для отслеживания занятых значений

GUNF_file_OK (0)	Функция завершила свою работу без ошибки
ERROR_GUNF_RangeTooSmall (1)	Маленький объём чисел
ERROR_GUNF_AllocPool (2)	Ошибка выделения динамической памяти
ERROR_GUNF_Input_OpenFail (3)	Ошибка открытия файла

void manual_test_collatz()
Набор ручных тестов для kollats0 на фиксированном списке входных значений (массив input из Q элементов)

int input_test()	
Спрашивает у пользователя, хочет ли он ввести число вручную; если да — читает целое, проверяет введённое число, вызывает kollats0 и печатает результат	
IT_OK (0)	Функция завершила свою работу без ошибки
ERROR_IT_InvalidInput (1)	Ошибка в считывании числа с клавиатуры
ERROR_IT_kollats0Failed (2)	Ошибка в вызове ф-ции kollats0

Функции, необходимые для исследования алгоритма.

int kollats0(ULL x, ULL* y)	
Подсчитывает количество шагов последовательности Коллатца, необходимое, чтобы x превратилось в 1, не записывая промежуточные значения.	
x	начальное число
*y	указатель, по которому запишется количество шагов

kollats0_OK (0)	Функция завершила свою работу без ошибок
ERROR_kollats0_NullPointer (1)	Нулевой указатель в аргументе ф-ции

int kollats(ULL Number, ULL* Steps, FILE* OutputFile)	
Выполняет последовательность Коллатца для Number, выводит нечётные промежуточные значения в OutputFile и в конце записывает длину последовательности (количество шагов)	
Number	начальное число
Steps	указатель, по которому запишется длина
*OutputFile	указатель на открытый файловый поток для вывода.
kollats_OK (0)	Функция завершила свою работу без ошибки
ERROR_kollats_NullPointer_Steps(1)	Нулевой указатель в аргументе ULL* steps
ERROR_kollats_NullPointer_File(2)	Нулевой указатель в аргументе ULL* OutputFile
ERROR_kollats_MultiplyFailed(3)	Ошибка в вызове ф-ции Multiply

ПРИЛОЖЕНИЕ Г. Исходный код разработанной программы

Файл `include\common.h`.

```
#ifndef LAB2_COMMON_H
#define LAB2_COMMON_H


#include <stdint.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <time.h>
#include <limits.h>


/* ---------- Коды ошибок (общие / для UI) ---------- */
/* MAIN_STATUS — общий код возврата для main/утилит */
enum MAIN_STATUS
{
    SUCCESS = 0,
    ERROR_MEMORY_ALLOCATION,
    ERROR_INVALID_INPUT,
    ERROR_FILE_OPERATION,
    ERROR_USER_INPUT
};

/* MATRIX_STATUS — коды ошибок для операций с матрицами */
enum MATRIX_STATUS
{
    MATRIX_SUCCESS = 0,
    MATRIX_ERROR_DIMENSION,
    MATRIX_ERROR_INVALID_SIZE,
    MATRIX_ERROR_NOT_SQUARE,
    MATRIX_ERROR_CREATION,
```

```
    MATRIX_ERROR_NULL_POINTER,
    MATRIX_ERROR_INVALID_FIELD,
    MATRIX_ERROR_INVALID_NUMBER,
    MATRIX_ERROR_INVALID_ARGUMENT,
    MATRIX_ERROR_OVERFLOW
};

/* STRING_STATUS — коды ошибок в работе со строками/парсингом */
enum STRING_STATUS
{
    STRING_SUCCESS = 0,
    STRING_ERROR_CONVERSION,
    STRING_ERROR_INVALID_FORMAT,
    STRING_ERROR_BUFFER_OVERFLOW,
    STRING_ERROR_NULL_POINTER
};

/* TEST_STATUS — коды ошибок модуля генерации тестов */
enum TEST_STATUS
{
    TEST_SUCCESS = 0,
    TEST_ERROR_GENERATION,
    TEST_ERROR_FILE_WRITE,
    TEST_ERROR_INVALID_PARAMS,
    TEST_ERROR_CLOCK,
    TEST_ERROR_OVERFLOW
};

/* UI_STATUS — коды ошибок интерфейса/печати */
enum UI_STATUS
{
    UI_SUCCESS = 0,
    UI_ERROR_INPUT,
    UI_ERROR_MENU,
    UI_ERROR_DISPLAY
```

```
};

/*
 * Получить текстовое сообщение для кода ошибки матрицы.
 * Возвращает строку с описанием ошибки.
 * Если код ошибки неизвестен, возвращает "Unknown matrix error"
 * [IN] error — числовой код ошибки из enum MATRIX_STATUS
 * [RETURN] const char* — указатель на строку с сообщением
 */
const char* get_matrix_error_message(int error);

/*
 * Получить текстовое сообщение для кода ошибки работы
 * со строками.
 * Возвращает строку с описанием ошибки.
 * Если код ошибки неизвестен, возвращает "Unknown string error"
 * [IN] error — числовой код ошибки из enum STRING_STATUS
 * [RETURN] const char* — указатель на строку с сообщением
 */
const char* get_string_error_message(int error);

#endif //LAB2_COMMON_H
```

Файл `include\matrix.h`.

```
#ifndef LAB2_MATRIX_H
#define LAB2_MATRIX_H

#include "common.h"

typedef unsigned long long ULL;

/* Структура данных для матрицы */
typedef struct Matrix
{
```

```
    ULL** data;      /* указатель на массив строк */
    ULL field_size;  /* модуль (размер конечного поля) */
    int rows;                       /* число строк */
    int cols;                       /* число столбцов */
} Matrix;

/* ---------- Функции (матрицы) ---------- */

/*
 * Создать матрицу rows x cols, поле field_size.
 * result — выходной параметр (адрес указателя на Matrix).
 * [IN] rows - количество строк матрицы
 * [IN] cols - количество столбцов матрицы
 * [IN] field_size - размер данных (по умолчанию, 0)
 * [OUT] result - указатель на созданную матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_create(int rows, int cols, ULL field_size,
    Matrix** result);

/*
 * Освободить память, выделенную под матрицу.
 * Уничтожает структуру и внутренние данные, предотвращая
 * утечку памяти.
 * Безопасно при передаче NULL (не вызывает разыменования NULL).
 * [IN] matrix — указатель на созданную матрицу для удаления
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_free(Matrix** matrix);

/*
 * Создать копию матрицы src и вернуть её через result.
 * Полностью дублирует размеры, поле field_size и данные.
 * [IN] src — исходная матрица
 * [OUT] result — указатель на указатель, куда будет записана
```

```
 * новая копия
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_copy(const Matrix* src, Matrix** result);

/*
 * Сложить две матрицы одинакового размера: a + b.
 * Операция выполняется в поле field_size, если оно задано.
 * [IN] a — первая матрица
 * [IN] b — вторая матрица
 * [OUT] result — указатель на новую матрицу, содержащую сумму
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_sum(const Matrix* a, const Matrix* b,
    Matrix** result);

/*
 * Вычесть матрицу b из матрицы a: a - b.
 * Операция выполняется в поле field_size, если оно задано.
 * [IN] a — уменьшаемое
 * [IN] b — вычитаемое
 * [OUT] result — указатель на матрицу результата
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_subtract(const Matrix* a, const Matrix* b,
    Matrix** result);

/*
 * Умножить матрицу a на скаляр scalar в поле field_size.
 * Каждое значение элемента матрицы умножается на scalar.
 * [IN] a — исходная матрица
 * [IN] scalar — множитель
 * [OUT] result — указатель на матрицу результата
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
```

```
int matrix_scalar_multiply(const Matrix* a, ULL scalar,
    Matrix** result);

/*
 * Транспонировать матрицу a (перевернуть строки и столбцы).
 * Результирующая матрица имеет размеры cols x rows.
 * [IN] a — исходная матрица
 * [OUT] result — указатель на транспонированную матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_transpose(const Matrix* a, Matrix** result);

/*
 * Вырезать подматрицу из матрицы a по указанным индексам.
 * Диапазоны [start_row, end_row) и [start_col, end_col)
 * должны быть корректными.
 * [IN] a — исходная матрица
 * [IN] start_row — начальный индекс строки
 * [IN] end_row — конечный индекс строки (не включая)
 * [IN] start_col — начальный индекс столбца
 * [IN] end_col — конечный индекс столбца (не включая)
 * [OUT] result — указатель на вырезанную подматрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_submatrix(const Matrix* a, int start_row, int
    end_row, int start_col, int end_col, Matrix** result);

/*
 * Перемножить матрицы a и b: a × b.
 * Количество столбцов в a должно совпадать с количеством
 * строк в b.
 * [IN] a — первая матрица (левый множитель)
 * [IN] b — вторая матрица (правый множитель)
 * [OUT] result — указатель на новую матрицу с произведением
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
```

```
 */
int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result);

/*
 * Умножить два числа по модулю (a * b) % mod.
 * Используется в арифметике поля field_size, предотвращая
 * переполнение.
 * [IN] a — первый множитель
 * [IN] b — второй множитель
 * [IN] mod — модуль (если 0, операция выполняется без модуля)
 * [OUT] result - указатель на результат умножения
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int multiply_mod(ULL a, ULL b, ULL mod, ULL* result);

/*
 * Возвести квадратную матрицу base в степень exponent.
 * Используется метод бинарного возведения для эффективности.
 * Операции выполняются в поле field_size.
 * [IN] base — квадратная матрица (n x n)
 * [IN] exponent — показатель степени
 * [OUT] result — указатель на результирующую матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int matrix_power(const Matrix* base, ULL exponent,
    Matrix** result);

/*
 * Напечатать матрицу в стандартный поток вывода.
 * Формат вывода зависит от field_size (по модулю или
 * обычные значения).
 * Используется для отладки и проверки корректности.
 * [IN] matrix — указатель на матрицу для печати
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
```

```
 */
int matrix_print(const Matrix* matrix);

#endif //LAB2_MATRIX_H
```

Файл include\tests.h.

```
#ifndef LAB2_TESTS_H
#define LAB2_TESTS_H

#include "string_utils.h"

/*
 * Сгенерировать случайную квадратную матрицу size x size.
 * Все элементы — случайные числа по модулю field_size.
 * [IN] size — размер матрицы (количество строк и столбцов)
 * [IN] field_size — размер конечного поля для модульной
 * арифметики
 * [OUT] result — указатель на созданную матрицу
 * [RETURN] MATRIX_SUCCESS или код ошибки MATRIX_STATUS
 */
int generate_random_matrix(int size, ULL field_size,
    Matrix** result);

/*
 * Сгенерировать набор тестов и сохранить в CSV-файл.
 * CSV содержит: размер матрицы, степень, поле, время выполнения
 * - Измерение времени делается через
 *   clock_gettime(CLOCK_MONOTONIC).
 * - Формируется два файла:
 *     output-short.txt   (matrix_size exponent field_size
 *     computation_time_ns)
 *     filename (CSV)     (matrix_size,exponent,field_size,
 *     computation_time_ns)
```

```
 * [IN] filename — имя выходного CSV-файла
 * [IN] min_size, max_size — диапазон размеров матриц
 * [IN] num_tests — количество тестов
 * [IN] min_exponent, max_exponent — диапазон степеней
 *  для возведения матрицы
 * [IN] field_size — размер конечного поля
 * [RETURN] TEST_SUCCESS или код ошибки TEST_STATUS
 */

int generate_test_cases(const char* filename, int min_size,
    int max_size, int num_tests,
    unsigned long long min_exponent,
    unsigned long long max_exponent,
    unsigned long long field_size);

/*
 * Генерация тестов с взаимодействием с пользователем (CSV-файл)
 * Выполняется ввод параметров через консоль и генерация тестов.
 * [RETURN] UI_SUCCESS или код ошибки UI_STATUS
 */
int file_operations_test(void);

/*
 * Ручной ввод матрицы и параметров для тестирования функций.
 * Позволяет пользователю ввести матрицу и степень, выводит
 *  результат.
 * [RETURN] UI_SUCCESS или код ошибки UI_STATUS
 */
int input_test(void);

/*
 * Предопределённые тесты для проверки функций работы с
 * матрицами.
 * Используется для быстрого тестирования без пользовательского
 * ввода.
```

```
 * [RETURN] UI_SUCCESS или код ошибки UI_STATUS
 */
int manual_test(void);

#endif //LAB2_TESTS_H
```

Файл include\string_utils.h.

```
#ifndef LAB2_STRING_UTILS_H
#define LAB2_STRING_UTILS_H

#include "matrix.h"

/*
 * Преобразовать матрицу в строковый формат.
 * Формат: (a11,a12;a21,a22,...)
 * Строка выделяется через malloc, необходимо освободить
 * через free.
 * [IN] matrix — указатель на исходную матрицу
 * [OUT] result — указатель на строку с результатом
 * [RETURN] STRING_SUCCESS или код ошибки STRING_STATUS
 */
int matrix_to_string(const Matrix* matrix, char** result);

/*
 * Преобразовать строку вида "(...)" в матрицу.
 * Строка должна содержать элементы через ',' и строки
 * через ';'.
 * [IN] str — входная строка с данными матрицы
 * [IN] field_size — размер поля для модульной арифметики
 * [OUT] result — указатель на созданную матрицу
 * [RETURN] STRING_SUCCESS или код ошибки STRING_STATUS
 */
int string_to_matrix(const char* str, unsigned long long
    field_size, Matrix** result);
```

```
#endif //LAB2_STRING_UTILS_H
```

Файл `src\common.c`.

```
#include "../include/common.h"

const char* get_matrix_error_message(int error)
{
    const char* messages[] = {
        "\nEN: Operation completed successfully \n"
        "RU: Операция выполнена успешно\n",
        "\nEN: Matrix dimensions mismatch \n"
        "RU: Несовпадение размеров матриц\n",
        "\nEN: Invalid matrix size \n"
        "RU: Недопустимое количество строк или столбцов\n",
        "\nEN: Matrix is not square \n"
        "RU: Матрица не является квадратной\n",
        "\nEN: Memory allocation failed \n"
        "RU: Ошибка выделения памяти\n",
        "\nEN: Null pointer passed to function \n"
        "RU: Передан NULL указатель\n",
        "\nEN: Field/modulus mismatch or invalid \n"
        "RU: Несовпадение поля/модуля или"
        " недопустимое значение\n"
    };
    return ( (error >= 0) && (error < sizeof(messages)/
        sizeof(messages[0])) ) ? messages[error] :
    "EN: Unknown matrix error \n"
    "RU: Неизвестная ошибка матрицы";
}

const char* get_string_error_message(int error)
{
    const char* messages[] = {
```

```
        "\nEN: Operation completed successfully \n"
        "RU: Операция выполнена успешно\n",
        "\nEN: String conversion failed \n"
        "RU: Ошибка преобразования строки\n",
        "\nEN: Invalid string format \n"
        "RU: Недопустимый формат строки\n",
        "\nEN: String buffer overflow \n"
        "RU: Переполнение буфера строки\n",
        "\nEN: Null pointer passed to function \n"
        "RU: Передан NULL указатель\n"
    };
    return ((error >= 0) && (error < sizeof(messages)/
        sizeof(messages[0]))) ?
            messages[error] : "\nEN: Unknown string error \n"
                              "RU: Неизвестная ошибка строки\n";
}
```

Файл `src\matrix.c`.

```
#include "../include/matrix.h"
#include "../include/common.h"

int matrix_create(int rows, int cols, ULL field_size,
    Matrix** result)
{
    if (!result)
        return MATRIX_ERROR_NULL_POINTER;

    if (rows < 1 || cols < 1)
        return MATRIX_ERROR_INVALID_SIZE;

    Matrix* matrix = malloc(sizeof(Matrix));
    if (!matrix)
        return MATRIX_ERROR_CREATION;
```

```
    matrix->rows = rows;
    matrix->cols = cols;
    matrix->field_size = field_size;
    matrix->data = NULL;

    matrix->data = (ULL**)malloc(rows * sizeof(ULL*));
    if (!matrix->data)
    {
        free(matrix);
        return MATRIX_ERROR_CREATION;
    }

    for (int i = 0; i < rows; i++)
    {
        matrix->data[i] = (ULL*)calloc(cols, sizeof(ULL));
        if (!matrix->data[i])
        {
            for (int j = 0; j < i; j++)
            {
                free(matrix->data[j]);
            }
            free(matrix->data);
            free(matrix);
            return MATRIX_ERROR_CREATION;
        }
    }

    *result = matrix;
    return MATRIX_SUCCESS;
}

int matrix_free(Matrix** matrix_ptr)
{
    if (!matrix_ptr)
    {
```

```
        return MATRIX_SUCCESS;
    }

    Matrix* matrix = *matrix_ptr;
    if (!matrix)
    {
        return MATRIX_SUCCESS;
    }

    if (matrix->data)
    {
        for (int i = 0; i < matrix->rows; i++)
        {
            if (matrix->data[i])
            {
                free(matrix->data[i]);
                matrix->data[i] = NULL;
            }
        }
        free(matrix->data);
        matrix->data = NULL;
    }

    free(matrix);
    *matrix_ptr = NULL;

    return MATRIX_SUCCESS;
}

int matrix_copy(const Matrix* src, Matrix** result)
{
    if (!src || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }
```

```
    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (src == *result)
    {
        return MATRIX_ERROR_INVALID_ARGUMENT;
    }

    int error;
    Matrix* dest;
    error = matrix_create(src->rows, src->cols, src->field_size,
        &dest);
    if (error != MATRIX_SUCCESS)
    {
        return error;
    }

    for (int i = 0; i < src->rows; i++)
    {
        for (int j = 0; j < src->cols; j++)
        {
            dest->data[i][j] = src->data[i][j];
        }
    }

    *result = dest;
    return MATRIX_SUCCESS;
}


int multiply_mod(ULL a, ULL b, ULL mod, ULL* result)
{
```

```
    if (!result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    *result = 0;

    if (mod == 0)
    {
        if (a != 0 && b > ULLONG_MAX / a)
        {
            return MATRIX_ERROR_OVERFLOW;
        }
        *result = a * b;
        return MATRIX_SUCCESS;
    }

    ULL res = 0;
    a %= mod;
    b %= mod;

    while (b > 0)
    {
        if (b & 1)
        {
            res = (res + a) % mod;
        }
        a = (a * 2) % mod;
        b >>= 1;
    }

    *result = res;
    return MATRIX_SUCCESS;
}
```

```
int matrix_sum0(const Matrix* A, const Matrix* B, Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, A->cols, A->field_size, &C);

    for (int p = 0; p < A->rows; p++)
    {
        for (int q = 0; q < A->cols; q++)
        {
            ULL m = A->data[p][q];
            ULL n = B->data[p][q];
            ULL r;

            if (A->field_size == 0)
            {
                r = m + n;
            }
            else
            {
                r = (m % A->field_size + n % A->field_size)
                % A->field_size;
            }

            C->data[p][q] = r;
        }
    }

    *R = C;
    return 0;
}


int matrix_sum(const Matrix* a,const Matrix* b,Matrix** result)
{
    if (!a || !b || !result)
```

```
{
    return MATRIX_ERROR_NULL_POINTER;
}

if (*result != NULL)
{
    matrix_free(result);
}

if (a->rows < 1 || a->cols < 1 || b->rows < 1 ||
    b->cols < 1)
{
    return MATRIX_ERROR_INVALID_SIZE;
}

if (a->rows != b->rows || a->cols != b->cols)
{
    return MATRIX_ERROR_DIMENSION;
}

if (a->field_size != b->field_size)
{
    return MATRIX_ERROR_INVALID_FIELD;
}

int error;
Matrix* sum_matrix;
error = matrix_create(a->rows, a->cols, a->field_size,
    &sum_matrix);
if (error != MATRIX_SUCCESS)
{
    return error;
}

for (int i = 0; i < a->rows; i++)
```

```
    {
        for (int j = 0; j < a->cols; j++)
        {
            ULL x = a->data[i][j];
            ULL y = b->data[i][j];

            if (a->field_size == 0)
            {
                if (x > ULLONG_MAX - y)
                {
                    matrix_free(&sum_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }
                sum_matrix->data[i][j] = x + y;
            }
            else
            {
                x %= a->field_size;
                y %= a->field_size;

                if (x > ULLONG_MAX - y)
                {
                    matrix_free(&sum_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }

                sum_matrix->data[i][j] = (x+y) % a->field_size;
            }
        }
    }

    *result = sum_matrix;
    return MATRIX_SUCCESS;
}
```

```
int matrix_subtract0(const Matrix* A, const Matrix* B,
    Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, A->cols, A->field_size, &C);

    for (int p = 0; p < A->rows; p++)
    {
        for (int q = 0; q < A->cols; q++)
        {
            ULL m = A->data[p][q];
            ULL n = B->data[p][q];
            ULL r;

            if (A->field_size == 0)
            {
                r = m - n;
            }
            else
            {
                r = ((m % A->field_size) -
                    (n % A->field_size)) % A->field_size;
            }

            C->data[p][q] = r;
        }
    }

    *R = C;
    return 0;
}


int matrix_subtract(const Matrix* a, const Matrix* b,
    Matrix** result)
```

```
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1 || b->rows < 1
        || b->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }

    if (a->rows != b->rows || a->cols != b->cols)
    {
        return MATRIX_ERROR_DIMENSION;
    }

    if (a->field_size != b->field_size)
    {
        return MATRIX_ERROR_INVALID_FIELD;
    }

    int error;
    Matrix* sub_matrix;
    error = matrix_create(a->rows, a->cols, a->field_size,
        &sub_matrix);
    if (error != MATRIX_SUCCESS)
    {
        return error;
    }
```

```
    for (int i = 0; i < a->rows; i++)
    {
        for (int j = 0; j < a->cols; j++)
        {
            ULL x = a->data[i][j];
            ULL y = b->data[i][j];

            if (a->field_size == 0)
            {
                if (x < y)
                {
                    matrix_free(&sub_matrix);
                    return MATRIX_ERROR_OVERFLOW;
                }
                sub_matrix->data[i][j] = x - y;
            }
            else
            {
                x %= a->field_size;
                y %= a->field_size;
                sub_matrix->data[i][j] =
                    (x + a->field_size - y) % a->field_size;
            }
        }
    }

    *result = sub_matrix;
    return MATRIX_SUCCESS;
}

int matrix_scalar_multiply0(const Matrix* A, ULL s, Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, A->cols, A->field_size, &C);
```

```
    for (int i = 0; i < A->rows; i++)
    {
        for (int j = 0; j < A->cols; j++)
        {
            ULL x = A->data[i][j];
            ULL r;

            if (A->field_size == 0)
            {
                r = x * s;
            }
            else
            {
                r = (x % A->field_size * s % A->field_size)
                % A->field_size;
            }

            C->data[i][j] = r;
        }
    }

    *R = C;
    return 0;
}

int matrix_scalar_multiply(const Matrix* a, ULL scalar,
    Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
```

```
{
    matrix_free(result);
}

if (a->rows < 1 || a->cols < 1)
{
    return MATRIX_ERROR_INVALID_SIZE;
}

int error;
Matrix* scaled_matrix;
error = matrix_create(a->rows, a->cols, a->field_size,
    &scaled_matrix);
if (error != MATRIX_SUCCESS)
{
    return error;
}

for (int i = 0; i < a->rows; i++)
{
    for (int j = 0; j < a->cols; j++)
    {
        ULL x = a->data[i][j];
        ULL res;

        if (a->field_size == 0)
        {
            if (x != 0 && scalar > ULLONG_MAX / x)
            {
                matrix_free(&scaled_matrix);
                return MATRIX_ERROR_OVERFLOW;
            }
            res = x * scalar;
        }
        else
```

```
            {
                error = multiply_mod(x, scalar,
                    a->field_size, &res);
                if (error != MATRIX_SUCCESS)
                {
                    matrix_free(&scaled_matrix);
                    return error;
                }
            }

            scaled_matrix->data[i][j] = res;
        }
    }

    *result = scaled_matrix;
    return MATRIX_SUCCESS;
}

int matrix_transpose(const Matrix* a, Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }

    if (a->rows < 1 || a->cols < 1)
    {
        return MATRIX_ERROR_INVALID_SIZE;
    }
```

```
    int error;
    Matrix* transposed;
    error = matrix_create(a->cols, a->rows, a->field_size,
        &transposed);
    if (error != MATRIX_SUCCESS)
    {
        return error;
    }

    for (int i = 0; i < a->rows; i++)
    {
        for (int j = 0; j < a->cols; j++)
        {
            transposed->data[j][i] = a->data[i][j];
        }
    }

    *result = transposed;
    return MATRIX_SUCCESS;
}

int matrix_multiply0(const Matrix* A, const Matrix* B,
    Matrix** R)
{
    Matrix* C;
    matrix_create(A->rows, B->cols, A->field_size, &C);

    for (int i = 0; i < A->rows; i++)
    {
        for (int j = 0; j < B->cols; j++)
        {
            ULL sum = 0;

            for (int k = 0; k < A->cols; k++)
            {
```

```
                ULL term;
                if (A->field_size == 0)
                {
                    term = A->data[i][k] * B->data[k][j];
                    sum += term;
                }
                else
                {
                    term = (A->data[i][k] % A->field_size *
                            B->data[k][j] % A->field_size)
                            % A->field_size;
                    sum = (sum + term) % A->field_size;
                }
            }

            C->data[i][j] = sum;
        }
    }

    *R = C;
    return 0;
}

int matrix_multiply(const Matrix* a, const Matrix* b,
    Matrix** result)
{
    if (!a || !b || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }

    if (*result != NULL)
    {
        matrix_free(result);
    }
```

```
if(a->rows < 1 || a->cols < 1 || b->rows < 1 || b->cols < 1)
{
    return MATRIX_ERROR_INVALID_SIZE;
}

if (a->cols != b->rows)
{
    return MATRIX_ERROR_DIMENSION;
}

if (a->field_size != b->field_size)
{
    return MATRIX_ERROR_INVALID_FIELD;
}

int error;
Matrix* product;
error = matrix_create(a->rows, b->cols, a->field_size,
    &product);
if (error != MATRIX_SUCCESS)
{
    return error;
}

for (int i = 0; i < a->rows; i++)
{
    for (int j = 0; j < b->cols; j++)
    {
        ULL sum = 0;

        for (int k = 0; k < a->cols; k++)
        {
            ULL term;
```

```
    if (a->field_size == 0)
    {
        if ((a->data[i][k] != 0) && (
            b->data[k][j] > ULLONG_MAX /
            a->data[i][k]))
        {
            matrix_free(&product);
            return MATRIX_ERROR_OVERFLOW;
        }

        term = a->data[i][k] * b->data[k][j];

        if (sum > ULLONG_MAX - term)
        {
            matrix_free(&product);
            return MATRIX_ERROR_OVERFLOW;
        }

        sum += term;
    }
    else
    {
        error = multiply_mod(a->data[i][k],
            b->data[k][j],a->field_size, &term);
        if (error != MATRIX_SUCCESS)
        {
            matrix_free(&product);
            return error;
        }

        sum = (sum + term) % a->field_size;
    }
}

product->data[i][j] = sum;
```

```
        }
    }

    *result = product;
    return MATRIX_SUCCESS;
}

int matrix_submatrix0(const Matrix* A, int sr, int er, int sc,
    int ec, Matrix** R)
{
    int sub_rows = er - sr + 1;
    int sub_cols = ec - sc + 1;
    Matrix* C;
    matrix_create(sub_rows, sub_cols, A->field_size, &C);

    for (int i = 0; i < sub_rows; i++)
    {
        for (int j = 0; j < sub_cols; j++)
        {
            C->data[i][j] = A->data[sr + i][sc + j];
        }
    }

    *R = C;
    return 0;
}

int matrix_submatrix(const Matrix* a, int start_row,
    int end_row, int start_col, int end_col,
    Matrix** result)
{
    if (!a || !result)
    {
        return MATRIX_ERROR_NULL_POINTER;
    }
```

```
    if (start_row < 0 || end_row >= a->rows || start_col < 0
        || end_col >= a->cols ||
        start_row > end_row || start_col > end_col)
    {
        return MATRIX_ERROR_DIMENSION;
    }

    int sub_rows = end_row - start_row + 1;
    int sub_cols = end_col - start_col + 1;

    int error;
    Matrix* submatrix;
    error = matrix_create(sub_rows, sub_cols, a->field_size,
        &submatrix);
    if (error != MATRIX_SUCCESS) return error;

    int i, j;
    for (i = 0; i < sub_rows; i++)
    {
        for (j = 0; j < sub_cols; j++)
        {
            submatrix->data[i][j] = a->data[start_row + i]
            [start_col + j];
        }
    }

    *result = submatrix;
    return MATRIX_SUCCESS;
}

int matrix_power(const Matrix* base, ULL exponent, Matrix**
result)
{
    if (!base || !result)
        return MATRIX_ERROR_NULL_POINTER;
```

```
    if (base->rows != base->cols)
        return MATRIX_ERROR_NOT_SQUARE;

    if (*result != NULL)
    {
        matrix_free(result);
    }

    int error;
    Matrix* res_matrix = NULL;
    Matrix* temp_power = NULL;

    if (exponent == 0)
    {
        error = matrix_create(base->rows, base->cols,
            base->field_size, &res_matrix);
        if (error != MATRIX_SUCCESS)
            return error;

        for (int i = 0; i < base->rows; i++)
            res_matrix->data[i][i] = 1;

        *result = res_matrix;
        return MATRIX_SUCCESS;
    }

    if (exponent == 1)
        return matrix_copy(base, result);

    error   =   matrix_create(base->rows,   base->cols,   base-
>field_size, &res_matrix);
    if (error != MATRIX_SUCCESS)
        return error;
```

```
    for (int i = 0; i < base->rows; i++)
        res_matrix->data[i][i] = 1;

    error = matrix_copy(base, &temp_power);
    if (error != MATRIX_SUCCESS)
    {
        matrix_free(&res_matrix);
        return error;
    }

    ULL exp = exponent;
    while (exp > 0)
    {
        if (exp & 1)
        {
            Matrix* temp_result = NULL;
            error   =   matrix_multiply(res_matrix,   temp_power,
&temp_result);
            if (error != MATRIX_SUCCESS)
            {
                matrix_free(&res_matrix);
                matrix_free(&temp_power);
                return error;
            }
            matrix_free(&res_matrix);
            res_matrix = temp_result;
        }

        exp >>= 1;

        if (exp > 0)
        {
            Matrix* temp_square = NULL;
            error   =   matrix_multiply(temp_power,   temp_power,
&temp_square);
```

```
            if (error != MATRIX_SUCCESS)
            {
                matrix_free(&res_matrix);
                matrix_free(&temp_power);
                return error;
            }
            matrix_free(&temp_power);
            temp_power = temp_square;
        }
    }

    matrix_free(&temp_power);
    *result = res_matrix;
    return MATRIX_SUCCESS;
}

int matrix_print(const Matrix* matrix)
{
    if (!matrix)
    {
        printf("NULL matrix\n");
        return UI_ERROR_DISPLAY;
    }

    printf("Matrix %dx%d (field size: %llu):\n",
        matrix->rows, matrix->cols, matrix->field_size);
    for (int i = 0; i < matrix->rows; i++)
    {
        printf("  ");
        for (int j = 0; j < matrix->cols; j++)
        {
            printf("%llu ", matrix->data[i][j]);
        }
        printf("\n");
    }
```

```
    return UI_SUCCESS;
}

int matrix_power0(const Matrix* A, ULL n, Matrix** R)
{
    if (!A || !R)
        return MATRIX_ERROR_NULL_POINTER;

    if (A->rows != A->cols)
        return MATRIX_ERROR_NOT_SQUARE;

    if (*R != NULL)
        matrix_free(R);

    int err;
    Matrix* result = NULL;
    Matrix* temp = NULL;

    if (n == 0)
    {
        err  =  matrix_create(A->rows,  A->cols,  A->field_size,
&result);
        if (err != MATRIX_SUCCESS) return err;
        for (int i = 0; i < A->rows; i++)
            result->data[i][i] = 1;

        *R = result;
        return MATRIX_SUCCESS;
    }

    if (n == 1)
        return matrix_copy(A, R);

    err = matrix_create(A->rows, A->cols, A->field_size, &result);
    if (err != MATRIX_SUCCESS) return err;
```

```
for (int i = 0; i < A->rows; i++)
    result->data[i][i] = 1;

err = matrix_copy(A, &temp);
if (err != MATRIX_SUCCESS)
{
    matrix_free(&result);
    return err;
}

ULL exp = n;
while (exp > 0)
{
    if (exp & 1)
    {
        Matrix* new_result = NULL;
        err = matrix_multiply(result, temp, &new_result);
        if (err != MATRIX_SUCCESS)
        {
            matrix_free(&result);
            matrix_free(&temp);
            return err;
        }
        matrix_free(&result);
        result = new_result;
    }

    exp >>= 1;

    if (exp > 0)
    {
        Matrix* new_temp = NULL;
        err = matrix_multiply(temp, temp, &new_temp);
        if (err != MATRIX_SUCCESS)
        {
```

```
                matrix_free(&result);
                matrix_free(&temp);
                return err;
            }
            matrix_free(&temp);
            temp = new_temp;
        }
    }

    matrix_free(&temp);
    *R = result;
    return MATRIX_SUCCESS;
}
```

Файл `src\tests.c`.

```
#include "../include/tests.h"

#define EXP 19 // [2^EXP;(2^EXP)-1)

static inline uint64_t rand64(void)
{
    return ((uint64_t)(rand() & 0xFFFF) << 48) |
           ((uint64_t)(rand() & 0xFFFF) << 32) |
           ((uint64_t)(rand() & 0xFFFF) << 16) |
           (uint64_t)(rand() & 0xFFFF);
}

static inline uint32_t rand32(void)
{
    return ((uint32_t)(rand() & 0xFFFF) << 16) |
           (uint32_t)(rand() & 0xFFFF);
}

static int get_time_ns(int64_t* out_ns)
```

```
{
    if (out_ns == NULL)
    {
        return TEST_ERROR_INVALID_PARAMS;
    }

    struct timespec ts;

    if (clock_gettime(CLOCK_MONOTONIC, &ts) != 0)
    {
        return TEST_ERROR_CLOCK;
    }
    if (ts.tv_sec > (INT64_MAX / 1000000000LL))
    {
        return TEST_ERROR_OVERFLOW;
    }

    *out_ns = (int64_t)ts.tv_sec * 1000000000LL +
        (int64_t)ts.tv_nsec;

    return 0;
}

int generate_random_matrix(int size, ULL field_size,
    Matrix** result)
{
    if (result == NULL)
        return MATRIX_ERROR_NULL_POINTER;
    if (size <= 0)
        return MATRIX_ERROR_INVALID_SIZE;

    int err = matrix_create(size, size, field_size, result);
    if (err != MATRIX_SUCCESS)
        return err;
```

```
    ULL min_number = 1ULL << EXP;
    ULL max_number = 1ULL << (EXP + 1);

    ULL range = max_number - min_number + 1;

    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            ULL random_value;
            ULL r = rand64();
            if (field_size == 0)
            {

                random_value = min_number + (r % range);
            }
            else
            {
                random_value = min_number + (r % range);
                random_value %= field_size;
            }

            (*result)->data[i][j] = random_value;
        }
    }

    return MATRIX_SUCCESS;
}

int generate_test_cases(const char* filename, int min_size,
    int max_size, int num_tests, ULL min_exponent,
    ULL max_exponent, ULL field_size)
{
    if (!filename)
        return TEST_ERROR_FILE_WRITE;
```

```
if (min_size <= 0 || max_size <= 0 || min_size > max_size)
    return TEST_ERROR_INVALID_PARAMS;
if (num_tests <= 0)
    return TEST_ERROR_INVALID_PARAMS;
if (min_exponent > max_exponent)
    return TEST_ERROR_INVALID_PARAMS;

FILE* csv = fopen(filename, "w");
if (!csv) return TEST_ERROR_FILE_WRITE;

FILE* short_out = fopen("output-short.txt", "w");
if (!short_out)
{
    fclose(csv);
    return TEST_ERROR_FILE_WRITE;
}

fprintf(csv, "matrix_size,exponent,field_size,"
             "computation_time_ns\n");
fprintf(short_out, "matrix_size exponent field_size "
                   "computation_time_ns\n");

srand((unsigned)time(NULL));
static int count_tests = 1;
int successful_tests = 0;

for (int test_idx = 0; test_idx < num_tests; test_idx++)
{
    int size = min_size + (rand32() % (max_size - min_size
        + 1));
    ULL exponent = min_exponent;
    if (min_exponent != max_exponent)
    {
        exponent = min_exponent + (rand64() % (max_exponent
            - min_exponent + 1));
```

```
}

Matrix* M = NULL;
int create_err = generate_random_matrix(size,
    field_size, &M);
if (create_err != MATRIX_SUCCESS)
{
    printf("\nFailed to create matrix: %s\n",
        get_matrix_error_message(create_err));
    continue;
}

char* matrix_str = NULL;
if (matrix_to_string(M, &matrix_str) != STRING_SUCCESS)
    matrix_str = strdup("SERIALIZE_ERROR");

int64_t t0 = 0, t1 = 0;
if (get_time_ns(&t0) != 0) t0 = 0;
Matrix* R = NULL;
int pow_err = matrix_power(M, exponent, &R);
if (get_time_ns(&t1) != 0) t1 = t0;
ULL dt_ns = t1 - t0;

char* result_str = NULL;
if (pow_err == MATRIX_SUCCESS)
{
    if (matrix_to_string(R, &result_str) !=
        STRING_SUCCESS)
        result_str = strdup("SERIALIZE_ERROR");
}
else
{
    const char* msg = get_matrix_error_message(pow_err);
    result_str = strdup(msg ? msg : "POWER_ERROR");
}
```

```
            printf("%6d %6d %12llu %12llu %12lld\n", count_tests,
                size, exponent, field_size, dt_ns);

            fprintf(short_out, "%d %llu %llu %llu\n", size,
                exponent, field_size, dt_ns);

            fprintf(csv, "%d,%llu,%llu,%lld\n",
                    size, exponent, field_size,
                    dt_ns);

            free(matrix_str);
            free(result_str);
            if (R) matrix_free(&R);
            matrix_free(&M);
            count_tests++;
            successful_tests++;
        }

        fclose(csv);
        fclose(short_out);

        if (successful_tests == 0)
            return TEST_ERROR_GENERATION;

        printf("Generated and ran %d tests (output: %s and "
               "output-short.txt)\n", successful_tests, filename);
        return TEST_SUCCESS;
    }

    int file_operations_test()
    {
        printf("=== ВЫБОР РЕЖИМА ГЕНЕРАЦИИ ТЕСТОВ ===\n");
        printf("1) Фиксация по степени "
               "(случайный размер матрицы)\n");
```

```
printf("2) Фиксация по размеру матрицы "
       "(случайная степень)\n");
printf("Выберите режим [1-2]:");

int mode = 0;
if (scanf("%d", &mode) != 1 || (mode != 1
    && mode != 2))
    return UI_ERROR_INPUT;
while (getchar() != '\n');

int min_size = 0, max_size = 0;
ULL min_exp = 0, max_exp = 0, static_size = 0,
field_size = 0;
int num_tests = 0;

printf("\nВведите количество тестов [1-10000]:");
if (scanf("%d", &num_tests) != 1 || num_tests < 1
    || num_tests > 10000)
    return UI_ERROR_INPUT;
while (getchar() != '\n');

if (mode == 1) // фиксированная степень
{
    printf("\nВведите фиксированную степень "
           "[1-1000000]:");
    if (scanf("%llu", &static_size) != 1 || static_size < 1
        || static_size > 1000000)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');
    min_exp = max_exp = static_size;

    printf("\nВведите минимальный размер матрицы "
           "[1-1000000]:");
    if (scanf("%d", &min_size) != 1 || min_size < 1
        || min_size > 1000000)
```

```
            return UI_ERROR_INPUT;
        while (getchar() != '\n');

        printf("\nВведите максимальный размер матрицы "
               "[%d-1000000]:", min_size);
        if (scanf("%d", &max_size) != 1 || max_size < min_size
            || max_size > 1000000)
            return UI_ERROR_INPUT;
        while (getchar() != '\n');
    }
    else // фиксированный размер матрицы
    {
        printf("\nВведите фиксированный размер матрицы "
               "[1-1000000]:");
        if (scanf("%llu", &static_size) != 1 || static_size < 1
            || static_size > 1000000)
            return UI_ERROR_INPUT;
        while (getchar() != '\n');
        min_size = max_size = static_size;

        printf("\nВведите минимальную степень [1-1000000]:");
        if (scanf("%llu", &min_exp) != 1 || min_exp < 1 ||
            min_exp > 1000000)
            return UI_ERROR_INPUT;
        while (getchar() != '\n');

        printf("\nВведите максимальную степень [%llu-1000000]:",
            min_exp);
        if (scanf("%llu", &max_exp) != 1 || max_exp < min_exp
            || max_exp > 1000000)
            return UI_ERROR_INPUT;
        while (getchar() != '\n');
    }

    ULL max_number = 1ULL << (EXP + 1);
```

```
    printf("\nВведите размер поля [0-%llu] (0 = без модулей):",
        max_number);
    if (scanf("%llu", &field_size) != 1)
        return UI_ERROR_INPUT;
    while (getchar() != '\n');

    printf("\nНачало генерации тестов...\n");

    int test_error = generate_test_cases(
        "matrix_power_tests.csv", min_size, max_size,
        num_tests,min_exp, max_exp, field_size);

    if (test_error == TEST_SUCCESS)
    {
        printf("\nТестовые данные успешно сохранены в"
               " matrix_power_tests.csv\n");
    }
    else
    {
        printf("\nОшибка при генерации тестов: %d\n",
            test_error);
    }

    return UI_SUCCESS;
}

int input_test()
{
    printf("=== РУЧНОЕ ТЕСТИРОВАНИЕ ===\n");

    char buffer[32767];
    int size;
    ULL exponent, field_size;
```

```
printf("Введите размер матрицы:");
if (scanf("%d", &size) != 1 || size <= 0)
{
    printf("Ошибка ввода размера матрицы\n");
    return UI_ERROR_INPUT;
}

printf("Введите размер конечного поля:");
if (scanf("%llu", &field_size) != 1 || field_size == 0)
{
    printf("Ошибка ввода размера поля\n");
    return UI_ERROR_INPUT;
}

printf("Введите степень:");
if (scanf("%llu", &exponent) != 1)
{
    printf("Ошибка ввода степени\n");
    return UI_ERROR_INPUT;
}

getchar();
printf("Введите матрицу в формате (a11,a12,...;a21,"
       "a22,...):");
if (fgets(buffer, sizeof(buffer), stdin) == NULL)
{
    printf("Ошибка ввода матрицы\n");
    return UI_ERROR_INPUT;
}
buffer[strcspn(buffer, "\n")] = 0;

Matrix* matrix;
int string_error = string_to_matrix(buffer, field_size,
    &matrix);
if (string_error != STRING_SUCCESS)
```

```
{
    printf("Ошибка преобразования строки в матрицу: %s\n",
        get_string_error_message(string_error));
    return UI_ERROR_INPUT;
}

printf("\nИсходная матрица:\n");
matrix_print(matrix);

int64_t t0 = 0, t1 = 0;
if (get_time_ns(&t0) != 0) t0 = 0;
Matrix* R = NULL;
int pow_err = matrix_power(matrix, exponent, &R);
if (get_time_ns(&t1) != 0) t1 = t0;
ULL dt_ns = t1 - t0;

if (pow_err == MATRIX_SUCCESS)
{
    printf("\nРезультат возведения в степень %llu:\n",
        exponent);
    matrix_print(R);

    char* result_str;
    string_error = matrix_to_string(R, &result_str);
    if (string_error == STRING_SUCCESS)
    {
        printf("\nРезультат в строковом формате: %s\n",
            result_str);
        free(result_str);
    }

    printf("Время выполнения: %llu наносекунд\n", dt_ns);

    matrix_free(&R);
}
```

```
    else
    {
        printf("Ошибка возведения в степень: %s\n",
            get_matrix_error_message(pow_err));
    }

    matrix_free(&matrix);
    return UI_SUCCESS;
}

int manual_test()
{
    printf("=== ТЕСТИРОВАНИЕ С ИЗВЕСТНЫМИ ДАННЫМИ ===\n");

    printf("\nТест 1: Матрица 2x2 в степени 2\n");
    Matrix* m1;
    int str_error = string_to_matrix("(1,2;3,4)", 100, &m1);
    if (str_error == STRING_SUCCESS)
    {
        Matrix* r1;
        int mat_error = matrix_power(m1, 2, &r1);
        if (mat_error == MATRIX_SUCCESS)
        {
            matrix_print(m1);
            printf("^2 =\n");
            matrix_print(r1);
            matrix_free(&r1);
        }
        else
        {
            printf("Ошибка: %s\n",
                get_matrix_error_message(mat_error));
        }
        matrix_free(&m1);
    }
```

```
else
{
    printf("Ошибка создания матрицы: %s\n",
        get_string_error_message(str_error));
}

printf("\nТест 2: Матрица 2x2 в степени 10\n");
Matrix* m2;
str_error = string_to_matrix("(1,1;1,0)", 100, &m2);
if (str_error == STRING_SUCCESS)
{
    Matrix* r2;
    int mat_error = matrix_power(m2, 10, &r2);
    if (mat_error == MATRIX_SUCCESS)
    {
        matrix_print(m2);
        printf("^10 =\n");
        matrix_print(r2);
        matrix_free(&r2);
    }
    else
    {
        printf("Ошибка: %s\n",
            get_matrix_error_message(mat_error));
    }
    matrix_free(&m2);
}
else
{
    printf("Ошибка создания матрицы: %s\n",
        get_string_error_message(str_error));
}

printf("\nТест 3: Единичная матрица в степени 5\n");
Matrix* m3;
```

```
    str_error = string_to_matrix("(1,0;0,1)", 100, &m3);
    if (str_error == STRING_SUCCESS)
    {
        Matrix* r3;
        int mat_error = matrix_power(m3, 5, &r3);
        if (mat_error == MATRIX_SUCCESS)
        {
            matrix_print(m3);
            printf("^5 =\n");
            matrix_print(r3);
            matrix_free(&r3);
        }
        else
        {
            printf("Ошибка: %s\n",
                get_matrix_error_message(mat_error));
        }
        matrix_free(&m3);
    }
    else
    {
        printf("Ошибка создания матрицы: %s\n",
            get_string_error_message(str_error));
    }

    return UI_SUCCESS;
}
```

Файл `src\string_utils.c`.

```
#include "../include/string_utils.h"

int string_to_matrix0(const char* input_str, ULL field_size,
Matrix** matrix)
{
```

```
if (!input_str || !matrix)
    return STRING_ERROR_NULL_POINTER;

if (*matrix != NULL)
    matrix_free(matrix);

int rows = 1, cols = 1;
size_t len = strlen(input_str);
for (size_t i = 1; i < len - 1; i++)
{
    if (input_str[i] == ';') rows++;
    else if (input_str[i] == ',' && rows == 1) cols++;
}

int error = matrix_create(rows, cols, field_size, matrix);
if (error != MATRIX_SUCCESS)
    return STRING_ERROR_CONVERSION;

int row_index = 0, col_index = 0;
char buffer[32];
int buffer_pos = 0;
int inside_parens = 0;

for (size_t i = 0; i < len; i++)
{
    char c = input_str[i];
    if (c == '(') { inside_parens = 1; continue; }
    else if (c == ')') { break; }
    if (!inside_parens) continue;

    if (c == ',' || c == ';' || c == ')')
    {
        if (buffer_pos > 0)
        {
            buffer[buffer_pos] = '\0';
```

```
                ULL value = strtoull(buffer, NULL, 10);
                if (field_size != 0) value %= field_size;

                (*matrix)->data[row_index][col_index] = value;
                buffer_pos = 0;
                col_index++;
            }

            if (c == ';') { row_index++; col_index = 0; }
        }
        else if (c != ' ')
        {
            if (buffer_pos >= 31)
            {
                matrix_free(matrix);
                *matrix = NULL;
                return STRING_ERROR_BUFFER_OVERFLOW;
            }
            buffer[buffer_pos++] = c;
        }
    }

    if (buffer_pos > 0)
    {
        buffer[buffer_pos] = '\0';
        ULL value = strtoull(buffer, NULL, 10);
        if (field_size != 0) value %= field_size;
        (*matrix)->data[row_index][col_index] = value;
    }

    return STRING_SUCCESS;
}

int string_to_matrix(const char* str, ULL field_size, Matrix**
result)
```

```
{
    if (!str || !result)
        return STRING_ERROR_NULL_POINTER;

    if (*result != NULL)
        matrix_free(result);

    size_t len = strlen(str);
    if (len < 3)
        return STRING_ERROR_INVALID_FORMAT;

    int rows = 1, cols = 1;
    for (size_t i = 1; i < len - 1; i++)
    {
        if (str[i] == ';') rows++;
        else if (str[i] == ',' && rows == 1) cols++;
    }

    int matrix_error = matrix_create(rows, cols, field_size,
result);
    if (matrix_error != MATRIX_SUCCESS)
        return STRING_ERROR_CONVERSION;

    int row = 0, col = 0;
    char buffer[32];
    int buf_index = 0;
    int inside_parentheses = 0;

    for (size_t i = 0; i < len; i++)
    {
        char c = str[i];
        if (c == '(') { inside_parentheses = 1; continue; }
        else if (c == ')') { break; }
        if (!inside_parentheses) continue;
```

```
    if (c == ';' || c == ',' || c == ')')
    {
        if (buf_index > 0)
        {
            buffer[buf_index] = '\0';
            ULL value = strtoull(buffer, NULL, 10);
            if (field_size != 0) value %= field_size;

            (*result)->data[row][col] = value;
            buf_index = 0;
            col++;
        }

        if (c == ';') { row++; col = 0; }
    }
    else if (c != ' ')
    {
        if (buf_index >= 31)
        {
            matrix_free(result);
            *result = NULL;
            return STRING_ERROR_BUFFER_OVERFLOW;
        }
        buffer[buf_index++] = c;
    }
}

if (buf_index > 0)
{
    buffer[buf_index] = '\0';
    ULL value = strtoull(buffer, NULL, 10);
    if (field_size != 0) value %= field_size;
    (*result)->data[row][col] = value;
}
```

```
    return STRING_SUCCESS;
}

int matrix_to_string0(const Matrix* m, char** s)
{
    if (!m || !s)
        return STRING_ERROR_NULL_POINTER;

    if (*s != NULL)
    {
        free(*s);
        *s = NULL;
    }

    if (m->rows <= 0 || m->cols <= 0)
        return STRING_ERROR_INVALID_FORMAT;

    int total_chars = 2; // ( )

    for (int i = 0; i < m->rows; i++)
    {
        for (int j = 0; j < m->cols; j++)
        {
            ULL n = m->data[i][j];
            if (m->field_size != 0)
                n %= m->field_size;

            int digits = (n == 0) ? 1 : 0;
            ULL tmp = n;
            while (tmp > 0)
            {
                digits++;
                tmp /= 10;
            }
```

```
            total_chars += digits;
            if (j < m->cols - 1)
                total_chars++; // ,
        }
        if (i < m->rows - 1)
            total_chars++; // ;
    }

    total_chars++; // '\0'
    char* result_str = malloc(total_chars);
    if (!result_str)
        return STRING_ERROR_CONVERSION;

    int pos = 0;
    result_str[pos++] = '(';

    for (int i = 0; i < m->rows; i++)
    {
        for (int j = 0; j < m->cols; j++)
        {
            ULL n = m->data[i][j];
            if (m->field_size != 0)
                n %= m->field_size;

            char buffer[32];
            int buf_pos = 0;

            if (n == 0)
                buffer[buf_pos++] = '0';
            else
            {
                ULL tmp = n;
                char digits[32];
                int digit_count = 0;
                while (tmp > 0)
```

```
                {
                    digits[digit_count++] = '0' + (tmp % 10);
                    tmp /= 10;
                }
                for (int k = digit_count - 1; k >= 0; k--)
                    buffer[buf_pos++] = digits[k];
            }

            for (int k = 0; k < buf_pos; k++)
                result_str[pos++] = buffer[k];

            if (j < m->cols - 1)
                result_str[pos++] = ',';
        }
        if (i < m->rows - 1)
            result_str[pos++] = ';';
    }

    result_str[pos++] = ')';
    result_str[pos] = '\0';
    *s = result_str;

    return STRING_SUCCESS;
}

int matrix_to_string(const Matrix* matrix, char** result)
{
    if (!matrix || !result)
        return STRING_ERROR_NULL_POINTER;

    if (*result != NULL)
    {
        free(*result);
        *result = NULL;
    }
```

```
if (matrix->rows <= 0 || matrix->cols <= 0)
    return STRING_ERROR_INVALID_FORMAT;

int total_chars = 2; // '(' и ')'

for (int i = 0; i < matrix->rows; i++)
{
    for (int j = 0; j < matrix->cols; j++)
    {
        ULL num = matrix->data[i][j];
        if (matrix->field_size != 0)
            num %= matrix->field_size;

        int digits = (num == 0) ? 1 : 0;
        ULL tmp = num;
        while (tmp > 0)
        {
            digits++;
            tmp /= 10;
        }

        total_chars += digits;
        if (j < matrix->cols - 1)
            total_chars++;
    }
    if (i < matrix->rows - 1)
        total_chars++;
}

total_chars++; // '\0'

char* str_result = malloc(total_chars);
if (!str_result)
    return STRING_ERROR_CONVERSION;
```

```
int pos = 0;
str_result[pos++] = '(';

for (int i = 0; i < matrix->rows; i++)
{
    for (int j = 0; j < matrix->cols; j++)
    {
        ULL num = matrix->data[i][j];
        if (matrix->field_size != 0)
            num %= matrix->field_size;

        char buffer[32];
        int buf_pos = 0;

        if (num == 0)
            buffer[buf_pos++] = '0';
        else
        {
            ULL tmp = num;
            char digits[32];
            int digit_count = 0;
            while (tmp > 0)
            {
                digits[digit_count++] = '0' + (tmp % 10);
                tmp /= 10;
            }
            for (int k = digit_count - 1; k >= 0; k--)
                buffer[buf_pos++] = digits[k];
        }

        for (int k = 0; k < buf_pos; k++)
            str_result[pos++] = buffer[k];

        if (j < matrix->cols - 1)
```

```
                str_result[pos++] = ',';
        }
        if (i < matrix->rows - 1)
            str_result[pos++] = ';';
    }

    str_result[pos++] = ')';
    str_result[pos] = '\0';
    *result = str_result;

    return STRING_SUCCESS;
}
```

Файл `src\main.c`.

```
#include "../include/common.h"
#include "../include/tests.h"

int main()
{
    printf("БЫСТРОЕ ВОЗВЕДЕНИЕ КВАДРАТНОЙ МАТРИЦЫ В СТЕПЕНЬ\n");
    printf("================================================\n"
           "\n");

    printf("СПРАВКА:\n");
    printf("1. Ручное тестирование - ввод матрицы и параметров "
           "вручную\n");
    printf("2. Тестирование с известными данными - "
           "предопределенные тесты\n");
    printf("3. Генерация тестовых данных - создание CSV файла "
           "с результатами\n");
    printf("4. Выход - завершение программы\n\n");

    int choice;
    int ui_error;
```

```
do
{
    printf("Меню:\n");
    printf("1. Ручное тестирование\n");
    printf("2. Тестирование с известными данными\n");
    printf("3. Генерация тестовых данных\n");
    printf("4. Выход\n");
    printf("Выберите опцию:");

    if (scanf("%d", &choice) != 1)
    {
        printf("Ошибка ввода\n");
        while (getchar() != '\n');
        continue;
    }

    switch (choice)
    {
        case 1:
            ui_error = input_test();
            if (ui_error != UI_SUCCESS)
            {
                printf("Ошибка ручного тестирования: %d\n",
                    ui_error);
            }
            break;
        case 2:
            ui_error = manual_test();
            if (ui_error != UI_SUCCESS)
            {
                printf("Ошибка предопределенного "
                       "тестирования: %d\n", ui_error);
            }
            break;
```

```
            case 3:
                ui_error = file_operations_test();
                if (ui_error != UI_SUCCESS)
                {
                    printf("Ошибка работы с файлами: %d\n",
                        ui_error);
                }
                break;
            case 4:
                printf("Выход...\n");
                break;
            default:
                printf("Неверный выбор. Попробуйте снова."
                       "\n");
        }
    } while (choice != 4);

    return SUCCESS;
}
```