Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЕТ

по Лабораторной работе №2

по курсу «Анализ Алгоритмов»

на тему: «Алгоритмы умножения матриц»

Студент группы ИУ7-51Б _____ (Подпись, дата) Савинова М. Г. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Волкова Л. Л. (Фамилия И.О.)

Преподаватель _____ (Подпись, дата) Строганов Ю. В.. (Фамилия И.О.)

Москва — 2023 г.

Содержание

Введение

Умножение матриц является основным инструментом линейной алгебры и имеет многочисленные применения в математике, физике, программировании [1].

Целью данной лабораторной работы является описание, реализация и исследование алгоритмов умножения матриц.

Для достижения поставленной цели необходимо выполнить следующие **задачи**:

1) описать следующие алгоритмы умножения матриц:
   - классический алгоритм умножения;
   - алгоритм Винограда;
   - оптимизированный алогритм Винограда;
2) релизовать описанные алгоритмы;
3) дать оценку трудоемкости алгоритмов;
4) провести замеры времени выполнения алгоритмов;
5) провести сравнительный анализ между алгоритмами.

1 Аналитическая часть

В данном разделе будут рассмотрены классический алгоритм умножениям матриц, алгоритм Винограда и его же оптимизированная версия.

1.1 Матрица

Матрицей размером $m \times n$ называют прямоугольную числовую таблицу, состоящую из $m \cdot n$ чисел, которые расположены в m строках и n столбцах. Состовляющие матрицу числа называют *элементами* этой *матрицы* [2].

Матрицу обозначают

$$\begin{pmatrix} a_{11} & a_{12} & \dots & a_{1n} \\ a_{21} & a_{22} & \dots & a_{2n} \\ \vdots & \vdots & \ddots & \vdots \\ a_{m1} & \vdots & \ddots & a_{mn} \end{pmatrix}. \tag{1.1}$$

Над матрицами возможны следующие операции:

— сложение матриц одинакового размера;

— произведение матрицы на число;

— произведение матриц, которое определено лишь в случае, когда количество *столбцов* первого сомножителя равно количеству *строк* второго [2].

1.2 Классический алгоритм

Пусть даны *матрица* $A = (a_{ij})$ *типа* $m \times n$ и *матрица* $B = (b_{ij})$ *типа* $n \times p$. *Произведением матриц* A и B называют матрицу $C = (c_{ij})$ типа $m \times p$ с *элементами*

$$c_{ij} = \sum_{k=1}^{n} a_{ik} b_{kj} \quad (i = \overline{1, m}, j = \overline{1, p}), \tag{1.2}$$

которую обозначают $C = AB$.

Классический алгоритм реализует формулу 1.2.

1.3 Алгоритм Винограда

Одним из самых эффективных по времени алгоритмов умножения матриц является алгоритм Винограда, имеющий асимптотическую сложность $O(n^{2,3755})$ [1].

Рассмотрим два вектора:

$$U = (u_1, u_2, u_3, u_4), \tag{1.3}$$

$$V = (v_1, v_2, v_3, v_4). \tag{1.4}$$

Их скалярное произведение равно:

$$U \times V = u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4, \tag{1.5}$$

что равносильно

$$U \times V = (u_1 + v_2)(u_2 + v_1) + (u_3 + v_4)(u_4 + v_3). \tag{1.6}$$

Возьмем упомянутые раннее матрицы A, B и . Скалярное произведение, по замыслу Винограда, 1.5 можно свести к следующему выражению:

$$c_{ij} = \sum_{k=1}^{n/2}(a_{i,2k-1} + b_{j,2k})(a_{i,2k} + b_{j,2k-1}) - \\ - \sum_{k=1}^{n/2} a_{i,2k-1}a_{i,2k} - \sum_{k=1}^{n/2} b_{2k-1,j}b_{2k,j}. \tag{1.7}$$

В целях экономии количества арифметических операций Виноград предложил находить второе и третье слагаемое в 1.7 заранее для каждой строки матрицы A и каждого столбца матрицы B.

Так, единожды вычислив для i-ой строки матрицы A значение выражения $\sum_{k=1}^{n/2} a_{i,2k-1}a_{i,2k}$, его можно использовать далее n раз для нахож-

дения элементов i-ой строки матрицы .

Аналогично, единожды вычислив для j-ой столбца матрицы B значение выражения $\sum_{k=1}^{n/2} b_{2k-1,j} b_{2k,j}$, его можно использовать далее n раз для нахождения элементов j-ой столбца матрицы [3].

Для примера, приведенного в формуле 1.6, в классическом умножении производится четыре умножения и три сложения; в алоритме Винограда — шесть умножений и девять сложений [3]. Но, несмотря на увеличение количества операций, выражение в правой части можно вычислить заранее и запомнить для каждой строки первой матрицы и каждого столбца второй матрицы. Это позволит выполнить лишь два умножения и пять сложений, складывая затем только лишь с двумя предварительно вычисленными суммами соседних элементов текущих строк и столбцов. Операция сложения выполняется быстрее, поэтому на практике алгоритм должен работать быстрее классического алгоритма умножения матриц.

При условии нечетного размера матрицы необходимо дополнительно добавить произведения крайних элементов соответствующих строк и столбцов.

1.4 Оптимизированный алгоритм Винограда

Для программной реализации алгоритма, рассмотренного в предыдущем пункте, можно выполнить следующие оптимизации:

1) значение $n/2$, используемое в качестве ограничения цикла подсчета предварительных данных, можно кэшировать;

2) операция умножения на 2 эффективнее реализовать как побитовый сдвиш влево на 1;

3) при условии существования операторов `+=, -=` в выбранном языке программирования, соответсвующие операции сложения и вычитания с присваиванием следует реализовывать с помощью данных операторов.

Вывод

В данном разделе были рассмотрены алгоритмы умножения матриц: классический, алгоритм Винограда. Также были рассмотрены оптимизации, которые можно учесть при програмной реализации алгоритма Винограда.

Основным отличием этих алгоритмов является наличие предварительных вичислений — как следствие, количество операций умножения и сложения также различно.

2 Конструкторская часть

В данном разделе будут реализованы схемы алгоритмов умножения матриц, приведено описание используемых типов данных, а также описана структура программного обеспечения.

2.1 Требования к программному обеспечению

К программе предъявлен ряд функциональных требований:

— на вход подаются две матрицы;

— матрицы располагаются в файлах, с расширением `*.txt`.

— на выходе — матрица, являющаяся результатом умножения матриц.

К программе предъявлен ряд требований:

— наличие интерфейса для выбора действия;

— наличие функциональных замеров процессорного времени выполнения алгоритмов умножения матриц;

— замеры процессорного времени выполнятся только для квадратных матриц.

2.2 Разработка алгоритмов

На рисунке 2.1 представлена схема алгоритма для стандартного умножения.

На рисунках 2.2–2.3 представлены схемы алгоритма умножения методом Винограда.

На рисунках 2.4–2.5 представлены схемы оптимизированного алгоритма умножения методом Винограда.

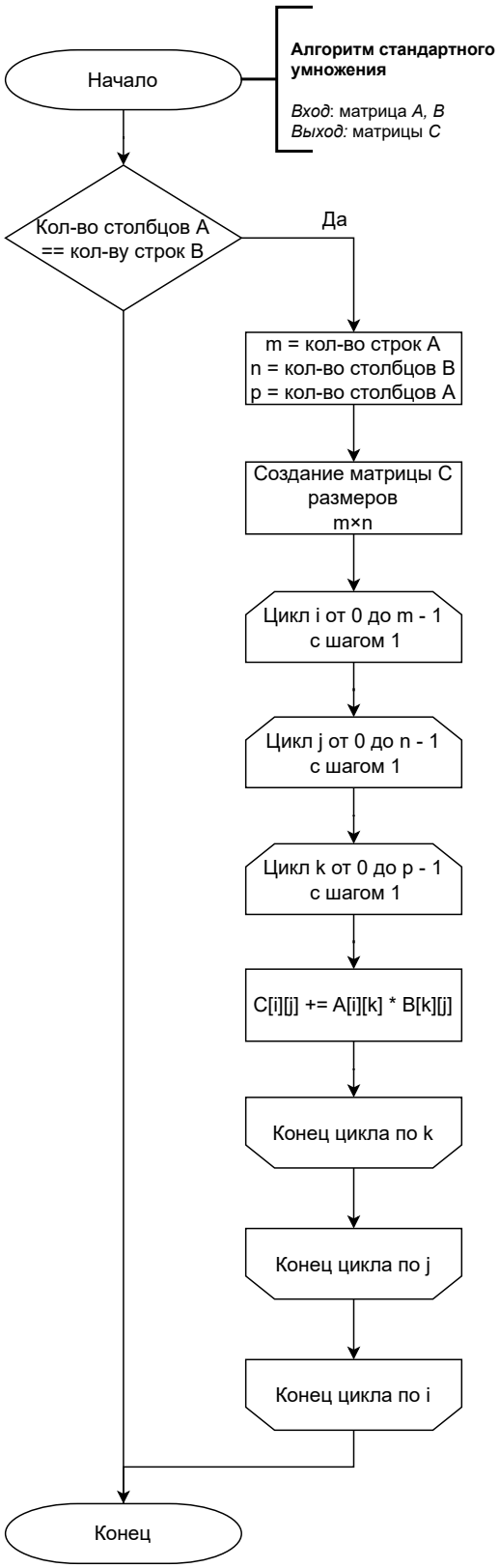

Рисунок 2.1 – Схема стандартного алгоритма умножения матриц

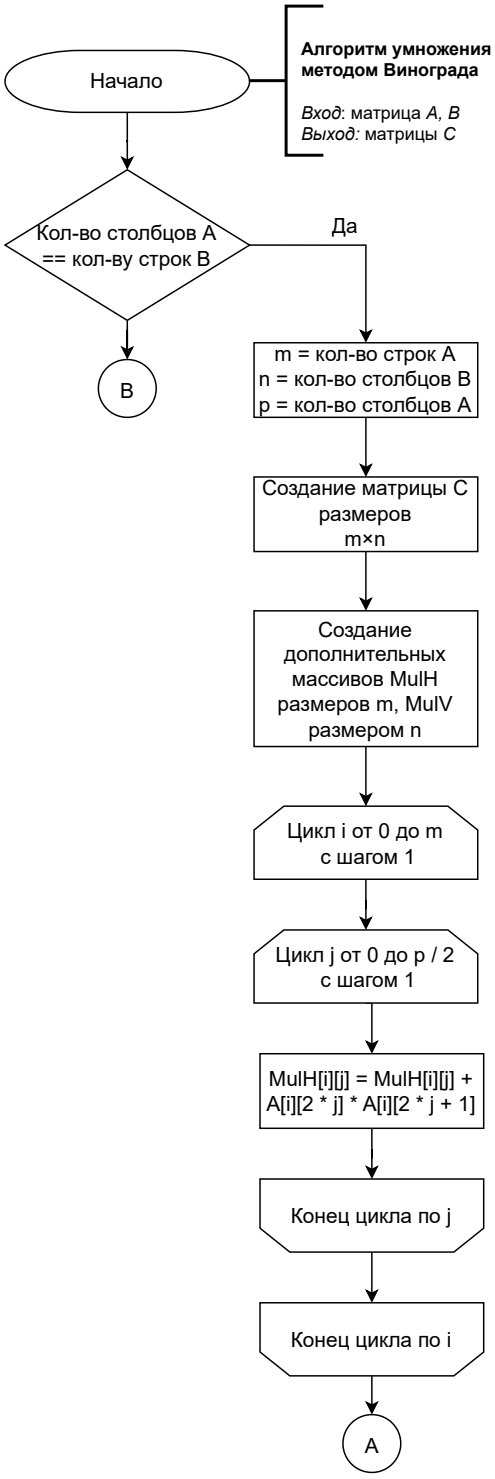

Рисунок 2.2 – Схема алгоритма умножения матриц методом Винограда (начало)

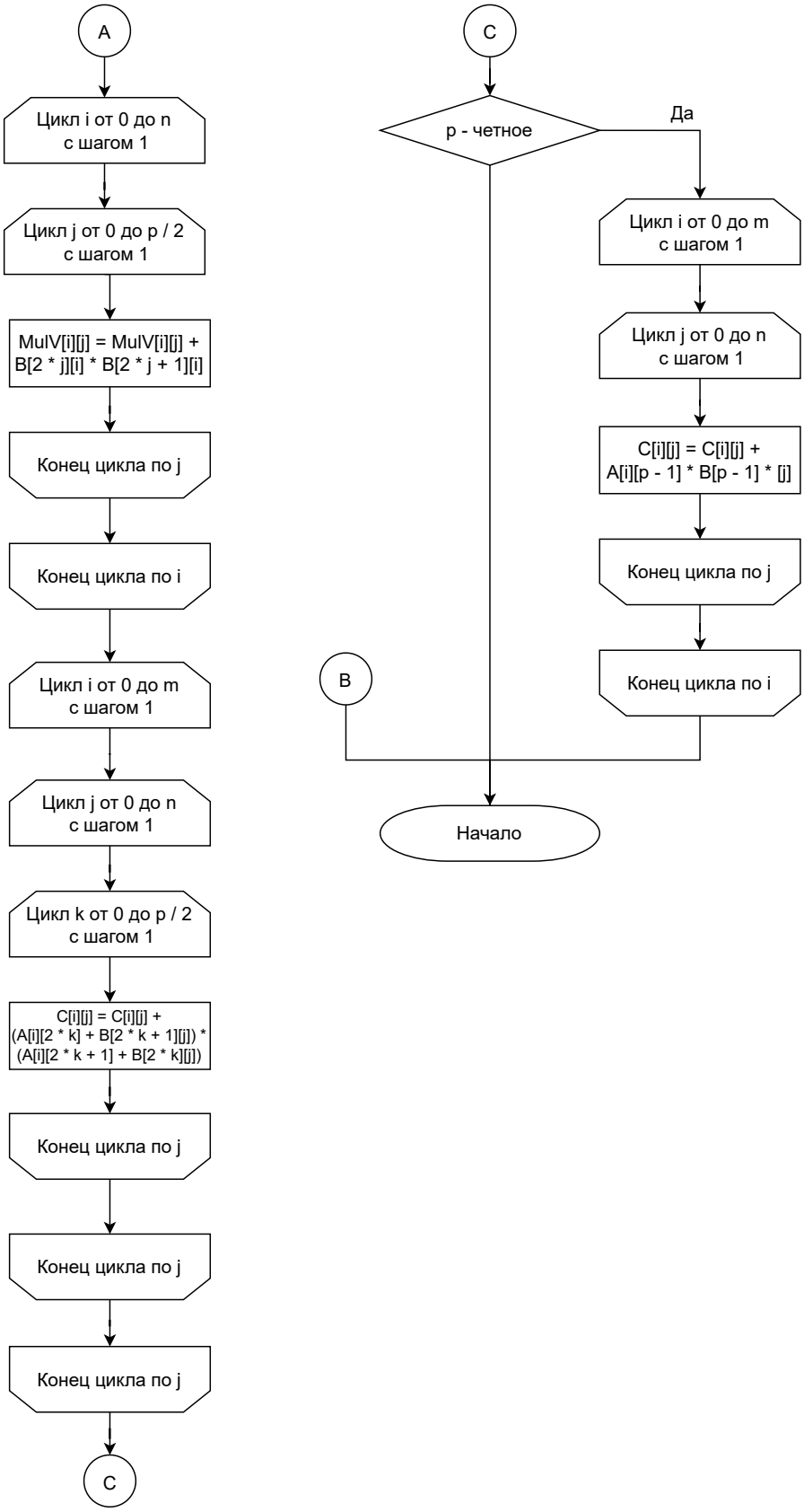

Рисунок 2.3 – Схема алгоритма умножения матриц методом Винограда (конец)

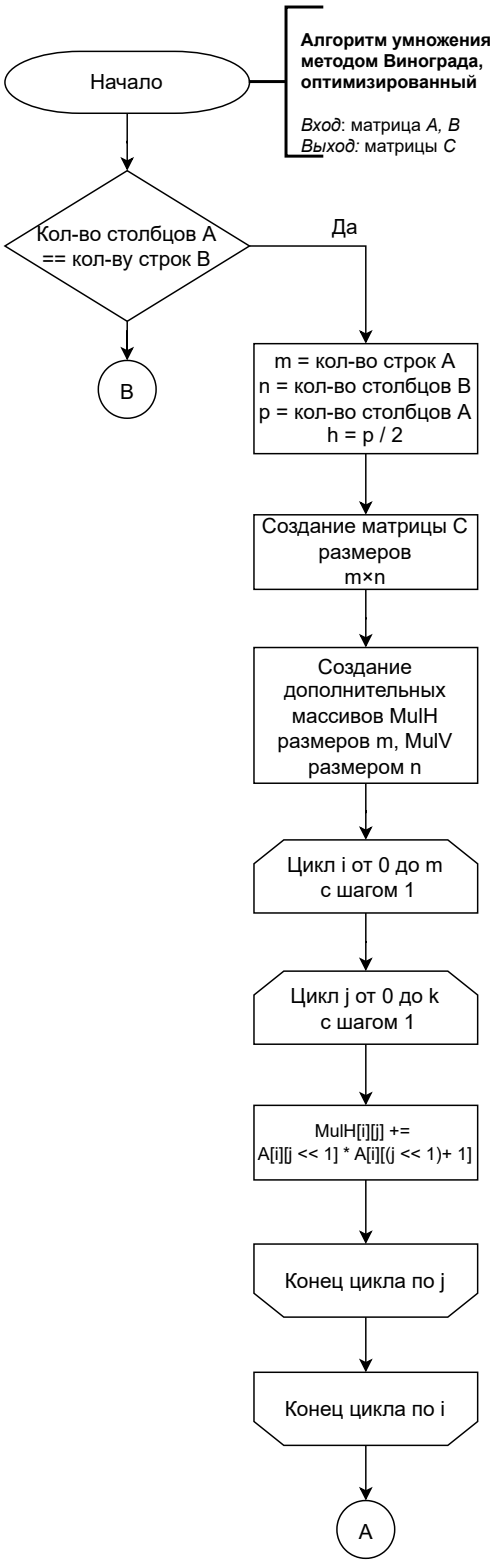

Рисунок 2.4 – Схема оптимизированного алгоритма умножения матриц методом Винограда (начало)

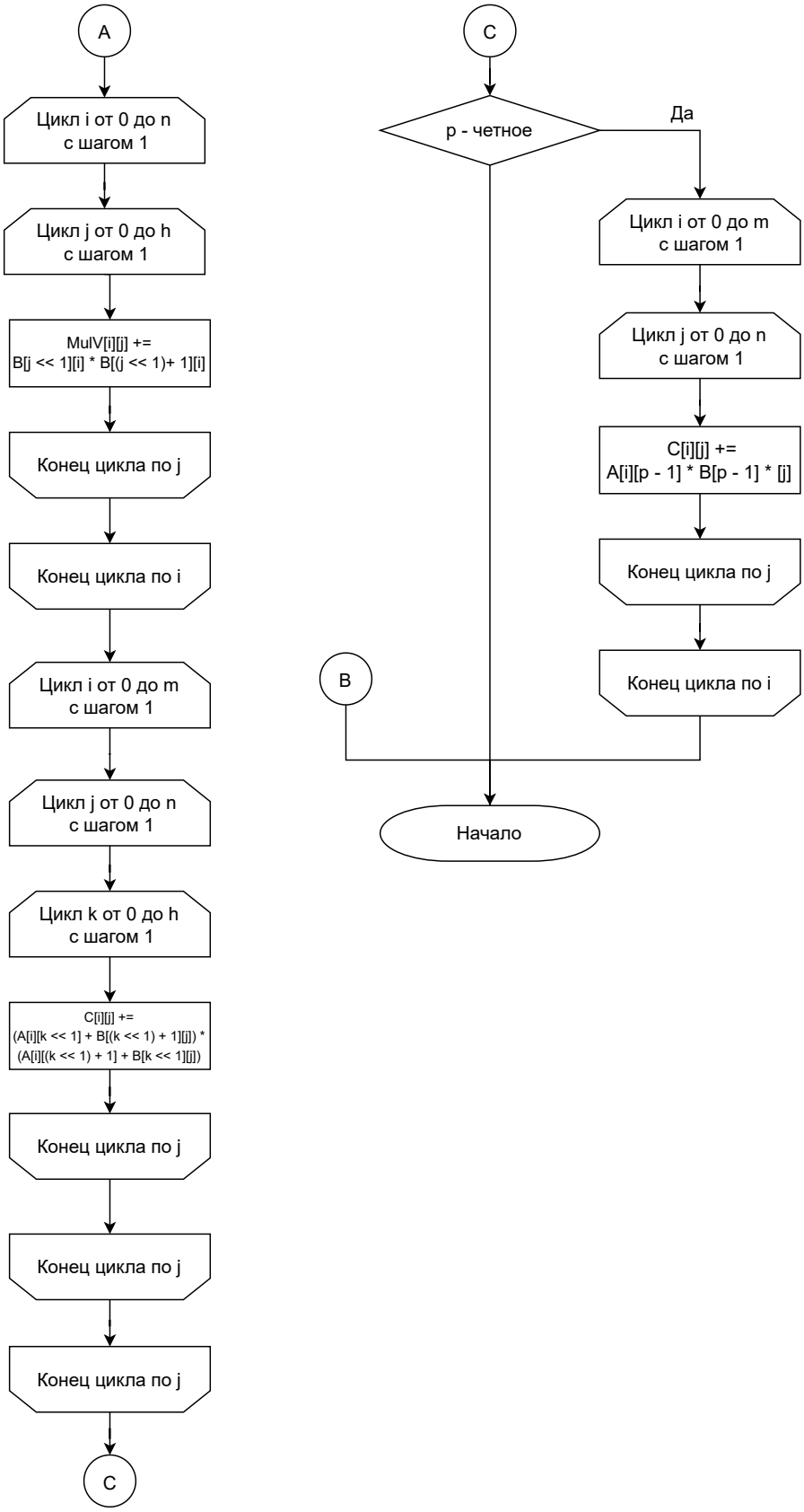

Рисунок 2.5 – Схема оптимизированного алгоритма умножения матриц методом Винограда (конец)

2.3 Модель вычислений для проведения оценки трудоемкости алгоритмов

Для последующего вычисления трудоемкости необходимо ввести модель вычислений:

1) операции из списка 2.1 имеют трудоемкость **1**;

$$+, -, =, + =, - =, ==, ! =, <, >, <=, >=, [], \\ ++, --, \&\&, >>, <<, ||, \&, | \tag{2.1}$$

2) операции из списка 2.2 имеют трудоемкость **2**;

$$*, /, \%, * =, / =, \% = \tag{2.2}$$

3) трудоемкость условного оператора `if условие then A else B` рассчитывается как 2.3;

$$f_{if} = f_{\text{условия}} + \begin{cases} f_A, & \text{в случае выполнеия условия,} \\ f_B, & \text{иначе.} \end{cases} \tag{2.3}$$

4) трудоемкость цикла рассчитывается как 2.4

$$f_{for} = f_{\text{инициализация}} + f_{\text{сравнения}} + M_{\text{итераций}} \cdot (f_{\text{тело}} + \\ + f_{\text{инкремент}} + f_{\text{сравнения}}) \tag{2.4}$$

5) трудоемкость вызова функции равна 0.

2.4 Трудоемкость алгоритмов

В следующих частях будут приведены рассчеты трудоемкостей алгоритмов для умножения матриц.

Стандартный алгоритм

Трудоемкость стандартного алгоритма умножения матриц состоит из:

— внешнего цикла по $i \in [1 \dots M]$, трудоёмкость которого: $f = 2 + M \cdot (2 + f_{body})$;

— цикла по $j \in [1 \dots N]$, трудоёмкость которого: $f = 2 + N \cdot (2 + f_{body})$;

— цикла по $k \in [1 \dots P]$, трудоёмкость которого: $f = 2 + 1K \cdot (2 + 12)$;

Так как трудоемкость стандартного алгоритма равно трудоемкости внешнего цикла —можно вычислить ее, подставив циклы тела:

$$\begin{gathered} f_{standart} = 2 + M \cdot (2 + 2 + N \cdot (2 + 2 + P \cdot (2 + 8 + 1 + 1 + 2))) = \\ = 2 + 4M + 4MN + 14MNP \approx 14MNP = O(N^3) \end{gathered} \tag{2.5}$$

Алгоритм Винограда

При вычислении трудоемкости алгоритма Винограда необходимо учесть следующее:

— трудоемкость создания и инициализации массивов $MulH$ и $MulV$:

$$f_{init} = f_{MulH} + f_{MulV}; \tag{2.6}$$

— трудоемкость заполнения массива $MulH$:

$$\begin{gathered} f_{MulH} = 2 + M \cdot (2 + 4 + \frac{P}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ = 2 + 6M + \frac{19MP}{2}; \end{gathered} \tag{2.7}$$

— трудоемкость заполнение массива $MulV$:

$$\begin{gathered} f_{MulV} = 2 + N \cdot (2 + 4 + \frac{P}{2} \cdot (4 + 6 + 1 + 2 + 3 \cdot 2)) = \\ = 2 + 6N + \frac{19NP}{2}; \end{gathered} \tag{2.8}$$

— трудоемкость цикла заполения для четных размеров:

$$f_{cycle} = 2 + M \cdot (4 + N \cdot (2 + 7 + 4 + \frac{P}{2} \cdot (4 + 28))) = \\ = 2 + 4M + 13MN + \frac{32MNP}{2} = 2 + 4M + 13MN + 16MNP \tag{2.9}$$

— трудоемкость дополнитеоьного цикла, в случае нечетного размера матрицы:

$$f_{check} = 3 + \begin{cases} 0, & \text{чётная} \\ 2 + M \cdot (4 + N \cdot (2 + 14)), & \text{иначе} \end{cases} \tag{2.10}$$

В итоге, для худшего случая (т. е. когда размер матрицы нечетный) получаем следующую трудоемкость:

$$f_{worst} = f_{MulH} + f_{MulV} + f_{cycle} + f_{check} \approx 16NMT = O(N^3) \tag{2.11}$$

Для лучшего случая (т. е. когда размер матрицы четный):

$$f_{best} = f_{MulH} + f_{MulV} + f_{cycle} + f_{check} \approx 16NMT = O(N^3) \tag{2.12}$$

Оптимизированный алгоритм Винограда

Вывод

В данном разделе на основе теоретических данных были перечислены требования к ПО, а также были построены схемы требуемых алгоритмов на основе теоретических данных, полученных на этапе анализа.

3 Технологическая часть

В данном разделе будут приведены средства реализации, листинг кода и функциональные тесты.

3.1 Средства реализации

Для реализации данной лабораторной работы был выбран язык `C++` [4], так как в нем есть стандартная библиотека `ctime` [5], которая позволяет производить замеры процессорного времени выполнения программы; тип данных `std::wstring`, позволяющий хранить как кириллические символы, так и латинские;

В качестве среды разработки был выбран *Visual Studio Code*: он является кроссплатформенным и предоставляет полный функционал для проектирования и отладки кода.

3.2 Сведения о модулях программы

Данная программа разбита на следующие модули:

— `main.cpp` — файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;

— `algorithms.cpp` — файл содержит функции поиска расстояния Левенштейна и Дамерау — Левенштейна;

— `matrix.cpp` — файл содержит функции динамического выделения и очищения памяти для матрицы, а так же ее вывод на экран;

— `measure.cpp` — файл содержит функции, замеряющие процессорное время выполнения алгоритмов поиска расстояния Левенштейна и Дамерау — Левенштейна;

3.3 Реализация алгоритмов

В листингах 3.2–3.7 приведены реализации алгоритмов поиска расстояний Левенштейна (только нерекурсивный алгоритм) и Дамерау — Левенштейна (нерекурсивный, рекурсивный и рекурсивный с кешированием).

Листинг 3.1 – Функция нахождения расстояния Левенштейна с использованием матрицы (начало)

```
1  int Algs::notRecursiveLev(wstring &word1, wstring &word2, bool
       print) {
2
3      int len1 = word1.length();
4      int len2 = word2.length();
5
6      int** mtr = Matrix::allocate(len2 + 1, len1 + 1);
7
8      if (!mtr)
9          return 0;
10
11     for (int i = 0; i <= len2; ++i) {
12
13         for (int j = 0; j <= len1; ++j) {
14
15             if (i == 0)
16                 mtr[i][j] = j;
17             else if (j == 0)
18                 mtr[i][j] = i;
19             else {
20                 int dif = (word1[j - 1] == word2[i - 1]) ? 0 :
                       1;
21
22                 mtr[i][j] = min(mtr[i - 1][j] + 1,
23                                 min(mtr[i][j - 1] + 1, mtr[i -
                                       1][j - 1] + dif));
24             }
25         }
26     }
```

Листинг 3.2 – Функция нахождения расстояния Левенштейна с использованием матрицы (конец)

```
1       if (print)
2           Matrix::print(mtr, word1, word2);
3
4       int res = mtr[len2][len1];
5       Matrix::release(mtr, len2 + 1);
6
7       return res;
8   }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау — Левенштейна с использованием матрицы (начало)

```
1  int Algs::notRecursiveDamLev(wstring &word1, wstring &word2,
       bool print) {
2
3       int len1 = word1.length();
4       int len2 = word2.length();
5
6       int** mtr = Matrix::allocate(len2 + 1, len1 + 1);
7
8       if (!mtr)
9           return 0;
10
11      for (int i = 0; i <= len2; ++i) {
12
13          for (int j = 0; j <= len1; ++j) {
14
15              if (i == 0)
16                  mtr[i][j] = j;
17              else if (j == 0)
18                  mtr[i][j] = i;
19              else {
20
21                  int dif = (word1[j - 1] == word2[i - 1]) ? 0 :
                        1;
22
23                  mtr[i][j] = min(mtr[i - 1][j] + 1,
24                                  min(mtr[i][j - 1] + 1, mtr[i -
                                        1][j - 1] + dif));
```

Листинг 3.4 – Функция нахождения расстояния Дамерау — Левенштейна с использованием матрицы (конец)

```
1                 if (word1[j - 2] == word2[i - 1] && word1[j -
                      1] == word2[i - 2])
2                     mtr[i][j] = min(mtr[i][j], mtr[i - 2][j -
                          2] + 1);
3             }
4         }
5     }
6
7     if (print)
8         Matrix::print(mtr, word1, word2);
9
10    int res = mtr[len2][len1];
11    Matrix::release(mtr, len2 + 1);
12
13    return res;
14 }
```

Листинг 3.5 – Функция нахождения расстояния Дамерау — Левенштейна рекурсивно

```
1  int Algs::recursive(wstring &word1, wstring &word2, int ind1,
       int ind2) {
2
3      if (min(ind1, ind2) == 0)
4          return max(ind1, ind2);
5
6      int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
7
8      int res = min(recursive(word1, word2, ind1 - 1, ind2 - 1) +
           dif,
9                    min(recursive(word1, word2, ind1 - 1, ind2) +
                         1,
10                       recursive(word1, word2, ind1, ind2 - 1) +
                             1));
11
12     if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
           2] && word1[ind1 - 2] == word2[ind2 - 1])
13         res = min(res, recursive(word1, word2, ind1 - 2, ind2 -
               2) + 1);
14
15     return res;
16 }
```

Листинг 3.6 – Функция вызова рекурсивного алгоритма с кешированием для поиска расстояния Дамерау — Левенштейна

```
1  int Algs::recursiveCash_Decor(wstring& word1, wstring& word2,
       bool print) {
2
3      int len1 = word1.length();
4      int len2 = word2.length();
5
6      int** cash = Matrix::allocate(len2 + 1, len1 + 1, true);
7
8      if (!cash)
9          return 0;
10
11     int res = recursiveCash(word1, word2, len1, len2, cash);
12
13     if (print)
14         Matrix::print(cash, word1, word2);
15
16     Matrix::release(cash, len2 + 1);
17
18     return res;
19 }
```

Листинг 3.7 – Функция нахождения расстояния Дамерау — Левенштейна рекурсивно с кешированием

```
1  int Algs::recursiveCash(wstring &word1, wstring &word2, int
       ind1, int ind2, int** cash) {
2
3      if (cash[ind2][ind1])
4          return cash[ind2][ind1];
5
6      if (min(ind1, ind2) == 0)
7          return cash[ind2][ind1] = max(ind1, ind2);
8
9      int dif = (word1[ind1 - 1] == word2[ind2 - 1]) ? 0 : 1;
10
11     int res = min(recursiveCash(word1, word2, ind1 - 1, ind2 -
           1, cash) + dif,
12                   min(recursiveCash(word1, word2, ind1 - 1,
                         ind2, cash) + 1,
13                       recursiveCash(word1, word2, ind1, ind2 -
                             1, cash) + 1));
14
15     if (ind1 > 1 && ind2 > 1 && word1[ind1 - 1] == word2[ind2 -
           2] && word1[ind1 - 2] == word2[ind2 - 1])
16         res = min(res, recursiveCash(word1, word2, ind1 - 2,
               ind2 - 2, cash) + 1);
17
18     cash[ind2][ind1] = res;
19
20     return res;
21 }
```

3.4 Функциональные тесты

Таблица 3.1 – Функциональные тесты

Входные данные		**Расстояние и алгоритм**			
		Левенштейна	**Дамерау — Левенштейна**		
Строка 1	**Строка 2**	**Итеративный**	**Итеративный**	**Рекурсивный**	
				Без кеша	**С кешем**
a	b	1	1	1	1
a	a	0	0	0	0
кот	скат	2	2	2	2
кот	кто	2	1	1	1
Австралия	Австрия	2	2	2	2
кот	ток	2	2	2	2
слон	слоны	1	1	1	1

Вывод

Были реализованы и протестированы алгоритмы поиска расстояния Левенштейна итеративно, а также поиска расстояния Дамерау–Левенштейна итеративно, рекурсивно и рекурсивного с кеширования. Проведено тестирование реализаций алгоритмов.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени, представлены далее.

— Процессор: AMD Ryzen 5 5500U – 2.10 ГГц;

— Оперативная память: 16 ГБайт;

— Операционная система: Windows 10 Pro 64-разрядная система версии 22H2.

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

На рисунке 4.1 представлена демонстрация работы разработанного ПО, а именно показаны результаты работы алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна на примере двух строк *«секста»* и *«септима»*.

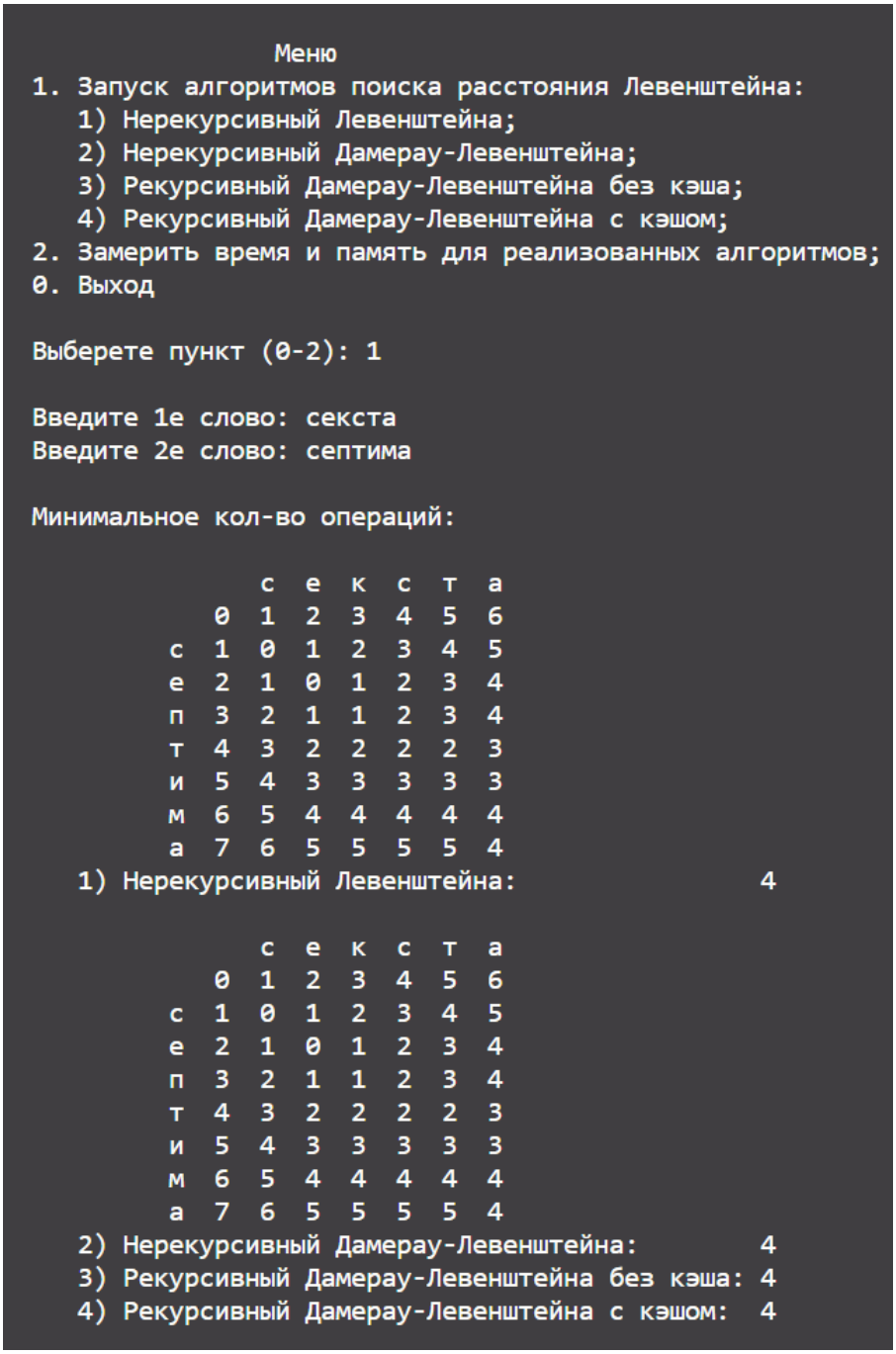

Рисунок 4.1 – Демонстрация работы программы

4.3 Временные характеристики

Все реализации алгоритмов сравнивались на случайно сгенерированных строках длиной:

— 0–10 с шагом 1 для всех алгоритмов;

— 10–200 с шагом 10 для нерекурсивных и рекурсивного с кешированием.

Поскольку замеры по времени имеют некоторую погрешность, для каждой строки и каждой реализации алгоритма замеры производились 1000 раз, а затем вычислялось среднее арифметическое значение.

На рисунке 4.2 представлен график, иллюстрирующий зависимость времени работы от длины строк для рекурсивных реализаций алгоритмов поиска расстояния Дамерау — Левенштейна с кешем и без.

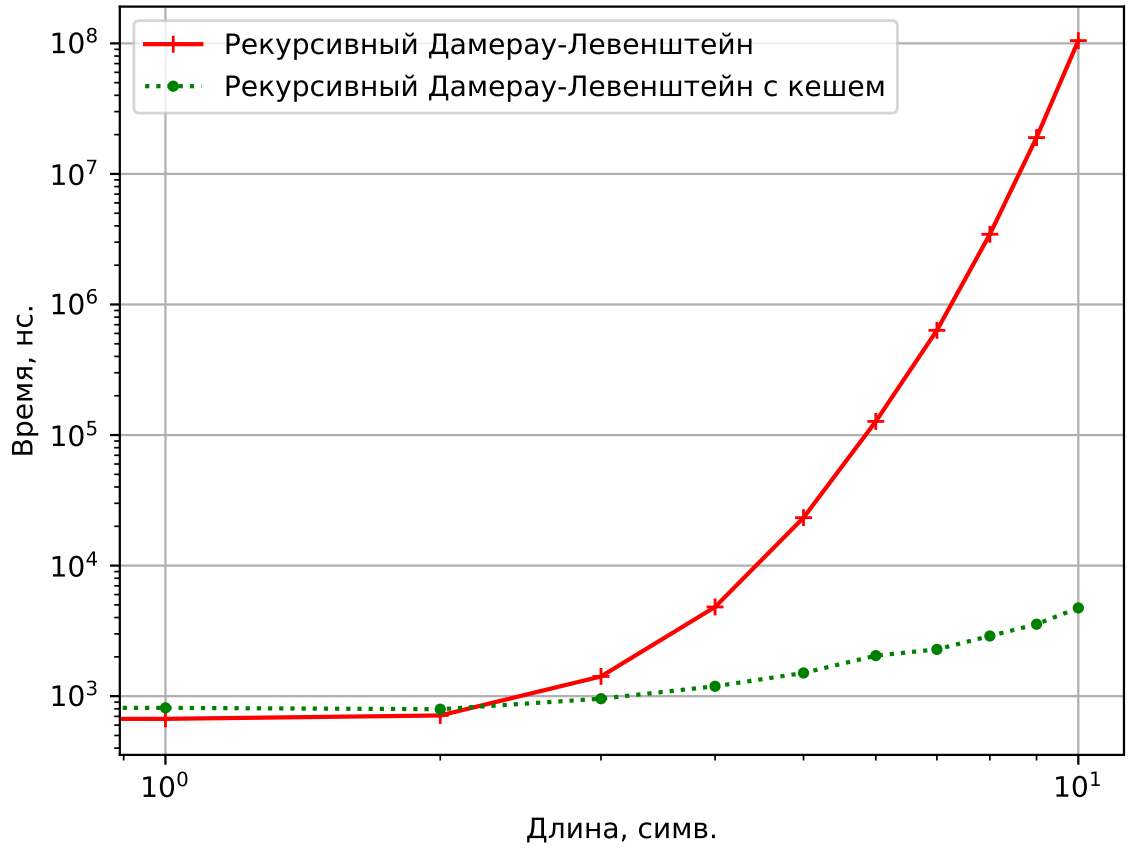

Рисунок 4.2 – Сравнение по времени рекурсивных реализаций алгоритмов поиска расстояния Дамерау — Левенштейна с кешем и без

На рисунке 4.3 представлен график, иллюстрирующий зависимость времени работы от длины строк для итеративных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна.

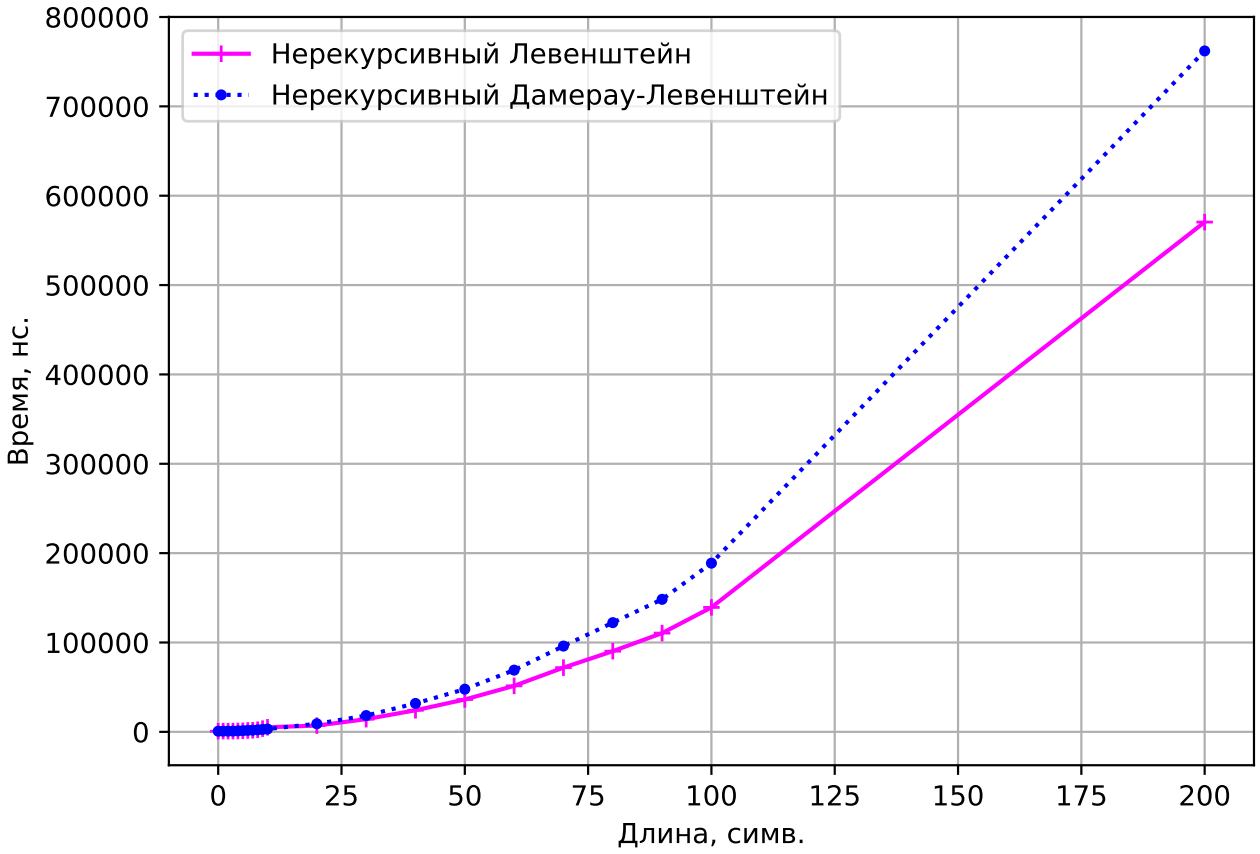

Рисунок 4.3 – Сравнение по времени итеративных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна

На рисунке 4.4 представлен график, иллюстрирующий зависимость времени работы от длины строк для итеративной реализации и рекурсивной реализации с использованием кеша алгоритма поиска расстояния Дамерау — Левенштейна.

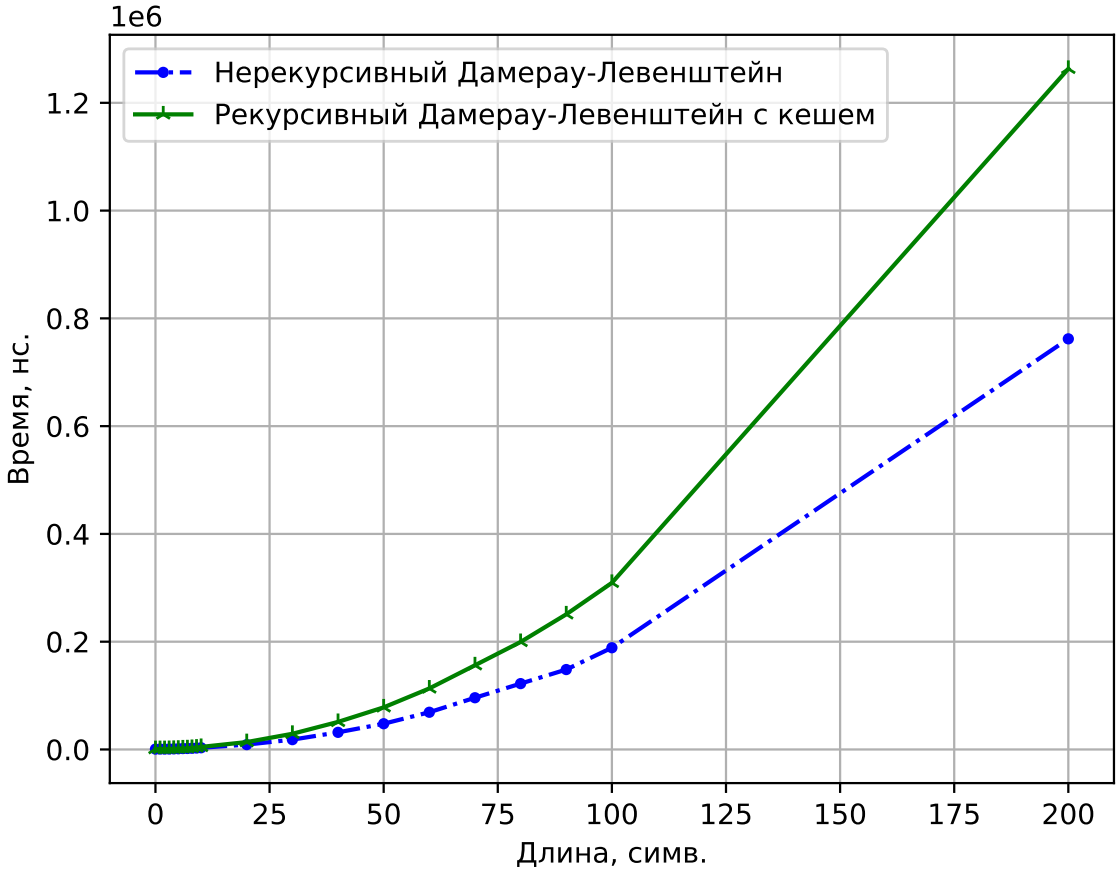

Рисунок 4.4 – Сравнение по времени итеративной реализации и рекурсивной реализации с использованием кеша алгоритма поиска расстояния Дамерау — Левенштейна

4.4 Характеристики по памяти

Введем следующие обозначения:

— n — длина строки S_1;

— m — длина строки S_2;

— $size()$ — функция вычисляющая размер в байтах;

— $char$ — тип, используемый для хранения символа строки;

— int — целочисленный тип.

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Левенштейна теоретически равно:

$$M_{iter} = (m+1)\cdot(n+1)\cdot size(int) + (n+m)\cdot size(char) + \\ +5\cdot size(int) + size(int**) + (n+1)\cdot size(int*), \quad (4.1)$$

где $(n+1)\cdot(m+1)\cdot size(int)$ — хранение матрицы;
$(n+m)\cdot size(char)$ — хранение двух строк;
$2\cdot size(int)$ — хранение размеров строк;
$3\cdot size(int)$ — дополнительные переменные;
$size(int**) + (n+1)\cdot size(int*)$ — указатель на матрицу.

Использование памяти при **итеративной реализации** алгоритма поиска расстояния Дамерау — Левенштейна идентично формуле (4.1).

Рассчитаем затраты по памяти для **рекурсивного** алгоритма поиска расстояния Дамерау — Левенштейна *(для каждого вызова)*:

$$M_{call} = (m+n)\cdot size(char) + 4\cdot size(int) + 8, \quad (4.2)$$

где $(n+m)\cdot size(char)$ — хранение двух строк;
$2\cdot size(int)$ — хранение размеров строк;
$2\cdot size(int)$ — дополнительные переменные;
8 байт — адрес возврата.

Макисмальная глубина стека вызовов при рекурсивной реализации равна сумме длин входящий строк, поэтому макисмальный расход памяти равен:

$$M_{rec} = (n+m)\cdot M_{call}, \quad (4.3)$$

где $n+m$ — максимальная глубина стека;
M_{call} — затраты по памяти для одного рекурсивного вызова.

Для рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с использованием кеша необходимо подсчитать размер самого кеша:

$$M_{cash} = (m+1)\cdot(n+1)\cdot size(int) + \\ + size(int**) + (n+1)\cdot size(int*), \quad (4.4)$$

где $(m+1)\cdot(n+1)$ — количество элементов в кеше;

$size(int**) + (n+1) \cdot size(int*)$ — хранение указателей.

Затраты по памяти для рекурсивного алгоритма поиска расстояния Дамерау — Левенштейна с учетом кеша:

$$M_{recCash} = M_{rec} + M_{cash}. \tag{4.5}$$

4.5 Вывод

По времени выполнения:

1) при малых длинах строк (< 5) рекурсивные реализации с кешем и без для поиска расстояния Дамерау — Левенштейна имеют приблизительно одинаковое время работы, но с увеличением длины строки реализация без кеша выполняется на порядок дольше, поскольу не происходит повторное вычисление значений (см рис. 4.2);

2) разница между итеративными реализациями алгоритмов поиска расстояний Левенштейна и Дамерау — Левенштейна незначительна, и обусловлена она дополнительным условием на проверку равенства соседних символов для расстояния Дамерау — Левенштейна (см рис. 4.3);

3) итеративная реализация работает на порядок быстрее рекурсивной с кешем для поиска расстояния Дамерау — Левенштейна (см рис. 4.4).

По затрачиваемой памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в итеративной реализации растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Заключение

Цель данной лабораторной работы была достигнута, а именно были изучены, реализованы и исследованы алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна.

В результате выполнения лабораторной работы для достижения этой цели были выполнены следующие задачи:

1) описаны алгоритмы поиска расстояния Левенштейна и Дамерау — Левенштейна;

2) разработаны и реализованы соответствующие алгоритмы;

3) создан программный продукт, позволяющий протестировать реализованные алгоритмы;

4) проведен сравнительный анализ процессорного времени выполнения реализованных алгоритмов;

5) проведен сравнительный анализ затрачиваемой алгоритмами памяти.

Список использованных источников

1 С. Анисимов Н., В. Строганов Ю. Реализация умножения матриц по Винограду на языке Haskell. // Новые информационные технологии в автоматизированных системах. 2018. URL: — Режим доступа: https://cyberleninka.ru/article/n/realizatsiya-algoritma-umnozheniya-matrits-po-vinogradu-na-yazyke-h (дата обращения 22.10.2023).

2 Н. Канатников А., П. Крищенко А. Аналитическая геометрия. Конспект лекций. 2009. Т. 132.

3 Головашкин Д. Л. Векторные алгоритмы вычислительной линейной алгебры: учеб. пособие. — Самара: Изд-во Самарского университета, 2019. — С. 28–35.

4 Документация по Microsoft C++ [Электронный ресурс]. — Режим доступа: https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017 (дата обращения: 25.09.2023).

5 C library function clock() [Электронный ресурс]. — Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2023).