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 №1 по дисциплине "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Егорова П.А.

Группа ИУ7-54Б

Преподаватели Волкова Л.Л., Строганов Ю.В.

Москва — 2022 г.

Оглавление

Введение

Целью данной лабораторной работы является применение навыков динамического программирования в алгоритмах поиска расстояний Левенштейна и Дамерау-Левенштейна [1]. Расстояние Левенштейна (редакционное расстояние) можно описать, как последовательность действий, необходимых для получения второй строки из первой самым коротким способом.

Расстояние Левенштейна – минимальное количество редакционных операций (вставка, удаление, замена символа), необходимых для преобразования одной строки в другую.

Если текст был набран с клавиатуры, то вместо расстояния Левенштейна чаще используют расстояние Дамерау-Левенштейна, в котором добавляется еще одно возможное действие – перестановка двух соседних символов.

Расстояния Левенштейна и Дамерау-Левенштейна применяются в таких сферах, как:

- компьютерная лингвистика (автозамена в посиковых запросах, текстовая редактура);
- биоинформатика (последовательности белков);
- нечеткий поиск записей в базах (борьба с мошенниками и опечатками).

В рамках выполнения работы необходимо решить следующие задачи:

1) изучить расстояния Левенштейна и Дамерау-Левенштейна;
2) разработать алгоритмы поиска этих расстояний;
3) реализовать разработанные алгоритмы;
4) провести сравнительный анализ процессорного времени выполнения реализаций этих алгоритмов;
5) провести сравнительный анализ затрачиваемой реализованными алгоритмами пиковой памяти.

1 Аналитическая часть

Расстояния Левенштейна и Дамерау-Левенштейна [1] – это минимальное количество операций, необходимых для преобразования одной строки в другую. Различие между этими расстояниями заключается в наборе допустимых операций.

В расстоянии Левенштейна расссматриваются такие действия над символами, как вставка (I-insert), удаление (D-delete) и замена (R-replace). Также вводится операция, которая не требует никаких действий – совпадение (M-match).

В расстоянии Дамерау-Левенштейна в дополнение к перечисленным операциям вводится также перестановка соседних символов (X-xchange).

Данным операциям можно назначить цену (штраф). Часто используется следующий набор штрафов: для операции M он равен нулю, для остальных (I, D, R, X) – единице.

Тогда задача нахождения расстояний Дамерау-Левенштейна сводится к поиску последовательности действий, минимизирующих суммарный штраф. Это можно сделать с помощью рекуррентных формул.

1.1 Расстояние Левенштейна

Пусть дано две строки S_1 и S_2. Тогда расстояние Левенштейна можно найти по рекуррентной формуле (1.1):

$$D(S_1[1...i],S_2[1...j]) = \begin{cases} 0, \text{ если i} == 0, \text{ j} == 0 \\ \text{j, если i} == 0, \text{ j} > 0 \\ \text{i, если j} == 0, \text{ i} > 0 \\ min(\\ D(S_1[1...i],S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, \quad \text{j}>0, \text{ i}>0 \\ D(S_1[1...i-1], S_2[1...j-1])+ \\ \begin{cases} 0, \text{ если } S_1[\text{i}] == S_2[\text{j}] \\ 1, \text{ иначе} \end{cases} \\ ) \end{cases} \tag{1.1}$$

Первые три формулы в системе (1.1) являются тривиальными и подразумевают, соответственно: отсутствие действий (совпадение, так как обе строки пусты), вставку j символов в пустую S_1 для создания строки-копии S_2 длиной j, удаление всех i символов из строки S_1 для совпадения с пустой строкой S_2.

В дальнейшем необходимо выбирать минимум из штрафов, которые будут порождены операциями вставки символа в S_1 (первая формула в группе min), удаления символа из S_1, (вторая форумла в группе min), а также совпадения или замены, в зависимости от равенства рассматриваемых на данном этапе символов строк (третья формула в группе min).

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна между строками S_1 и S_2 рассчитывается по схожей с (1.1) рекуррентной формуле. Отличие состоит лишь в добавлении четвертого возможного варианта (1.2) в группу min:

$$\begin{cases} D(S_1[1...i-2],S_2[1...j-2]) + 1, \text{ если i, j}>1, \ a_i = b_{j-1}, b_j = a_{i-1} \\ \infty \text{ , иначе} \end{cases} \tag{1.2}$$

Этот вариант подразумевает перестановку соседних символов в S_1, если

длины обеих строк больше единицы, и соседние рассматриваемые символы в S_1 и S_2 крест-накрест равны. Если же хотя бы одно из условий не выполняется, то данная операция не учитывается при поиске минимума.

Итоговая же формула для поиска расстояния Дамерау-Левенштейна имеет следующий вид (1.3):

$$D(S_1[1...i],S_2[1...j]) = \begin{cases} 0, \text{ если i} == 0, \text{ j} == 0 \\ \text{j, если i} == 0, \text{ j} > 0 \\ \text{i, если j} == 0, \text{ i} > 0 \\ min(\\ D(S_1[1...i],S_2[1...j-1]) + 1, \\ D(S_1[1...i-1], S_2[1...j]) + 1, & \text{j}>0, \text{ i}>0 \\ D(S_1[1...i-1], S_2[1...j-1]) + \\ \begin{bmatrix} 0, \text{ если } S_1[\text{i}] == S_2[\text{j}] \\ 1, \text{ иначе} \end{bmatrix} \\ , \\ \begin{bmatrix} D(S_1[1...i-2],S_2[1...j-2]) + 1, \\ \text{если i, j}>1,\ a_i = b_{j-1}, b_j = a_{i-1}; \\ \infty \text{ , иначе} \end{bmatrix} \\ ) \end{cases} \tag{1.3}$$

Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов. Формулы Левенштейна и Дамерау-Левенштейна для рассчета расстояния между строками задаются рекурсивно, а следовательно, алгоритмы могут быть реализованы как рекурсивно, так и итеративно.

2 Конструкторская часть

Рекуррентные формулы, рассмотренные в предыдущем разделе, позволяют находить расстояние Левенштейна и Дамерау-Левенштейна. Однако при разработке алгоритмов, решающих эти задачи, можно использовать различные подходы: итеративный алгоритм, алгоритм рекурсии с кэшированием, алгоритм рекурсии без кэширования, которые будут рассмотрены в данном разделе.

2.1 Алгоритм поиска расстояния Левенштейна

Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой построчное заполнение матрицы.

На рисунке 2.1 приведена схема рассматриваемого алгоритма.

2.2 Алгоритмы поиска расстояния Дамерау-Левенштейна

Рассматриваются итеративный, рекурсивный без кэширования и рекурсивный с кэшированием алгоритмы поиска расстояния Дамерау-Левенштейна.

На рисунках 2.2 – 2.4 приведены схемы рассматриваемых алгоритмов.

Вывод

На основе теоретических знаний, полученных в аналитическом разделе, были разработаны схемы алгоритмов, благодаря которым могут быть найдены расстояния Левенштейна и Дамерау-Левенштейна.

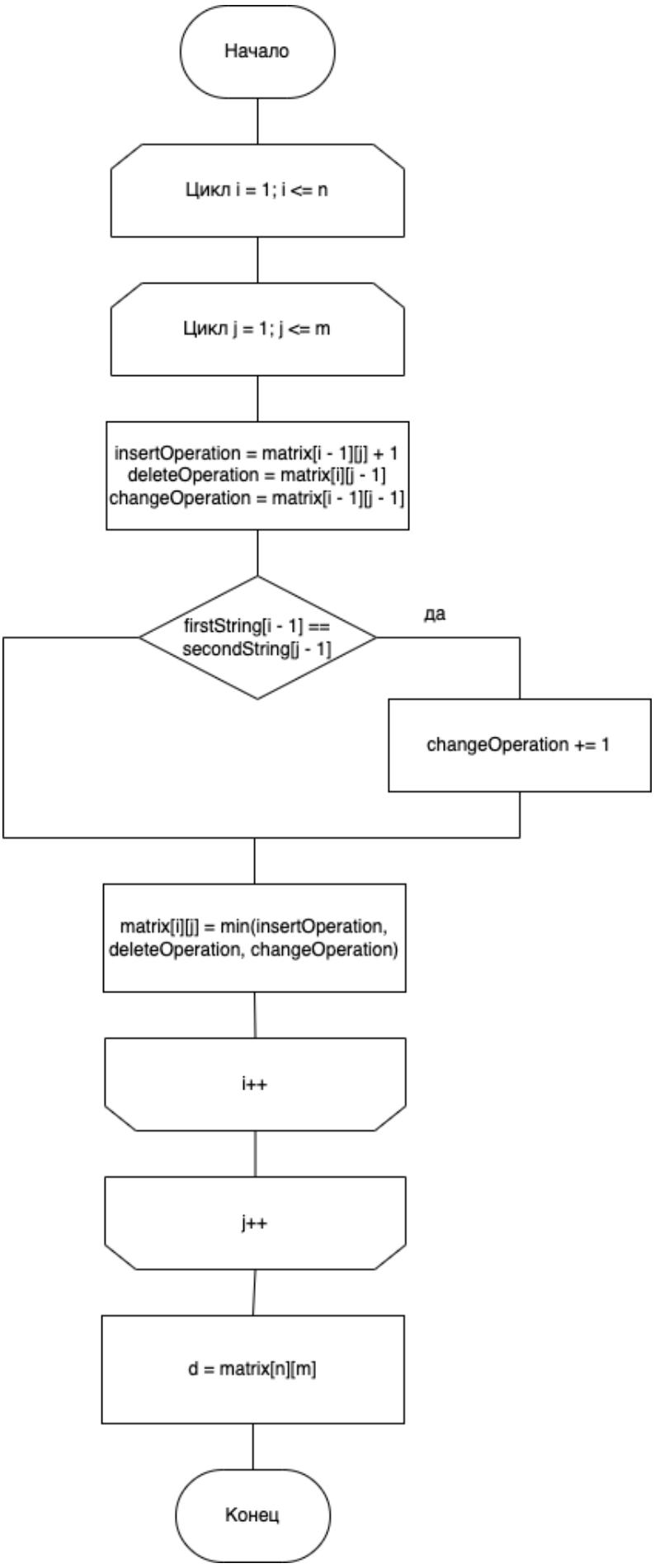

Рис. 2.1: Схема итеративного алгоритма поиска расстояния Левенштейна

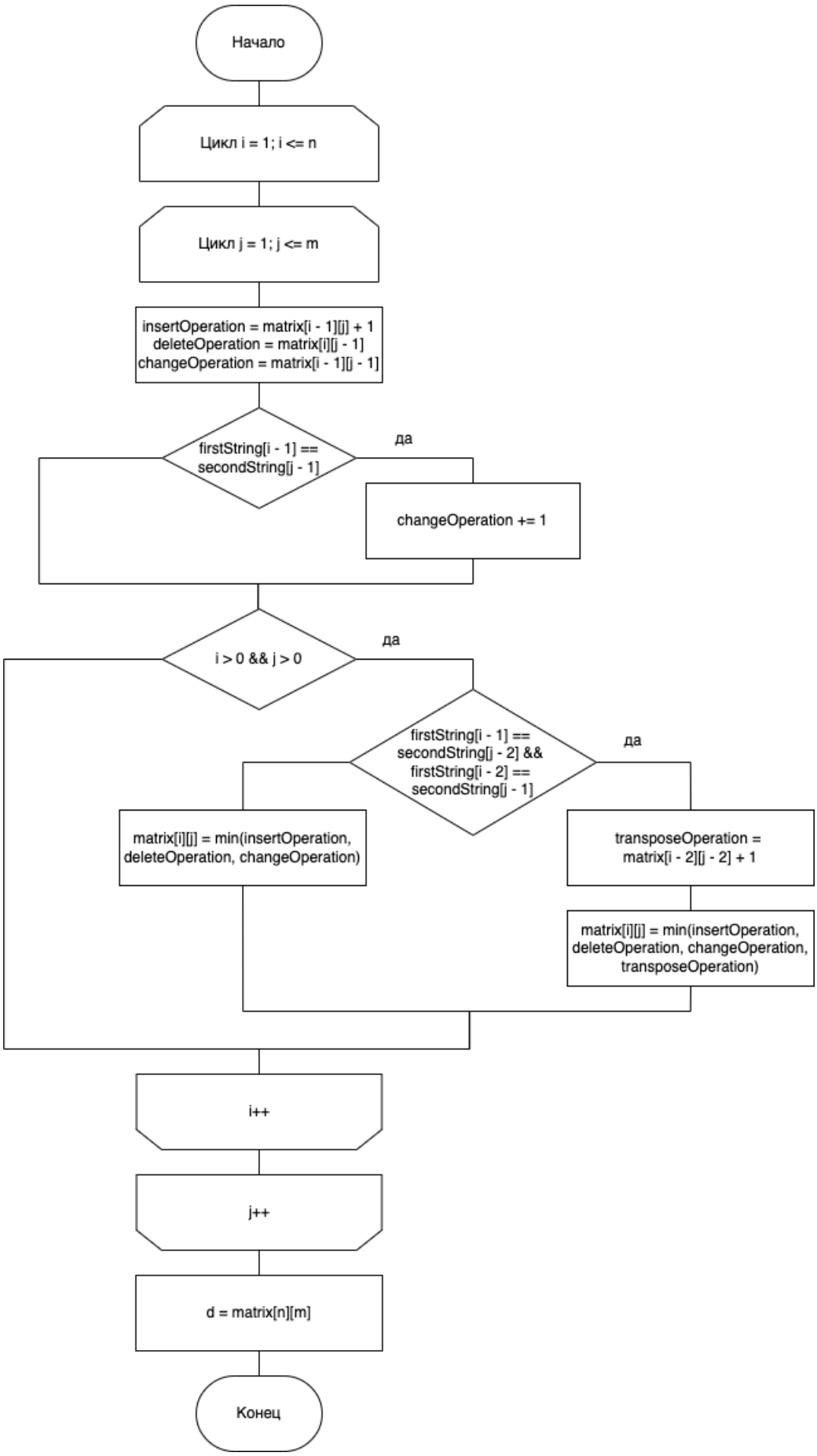

Рис. 2.2: Схема алгоритма поиска расстояния Дамерау-Левенштейна с заполнением матрицы расстояний

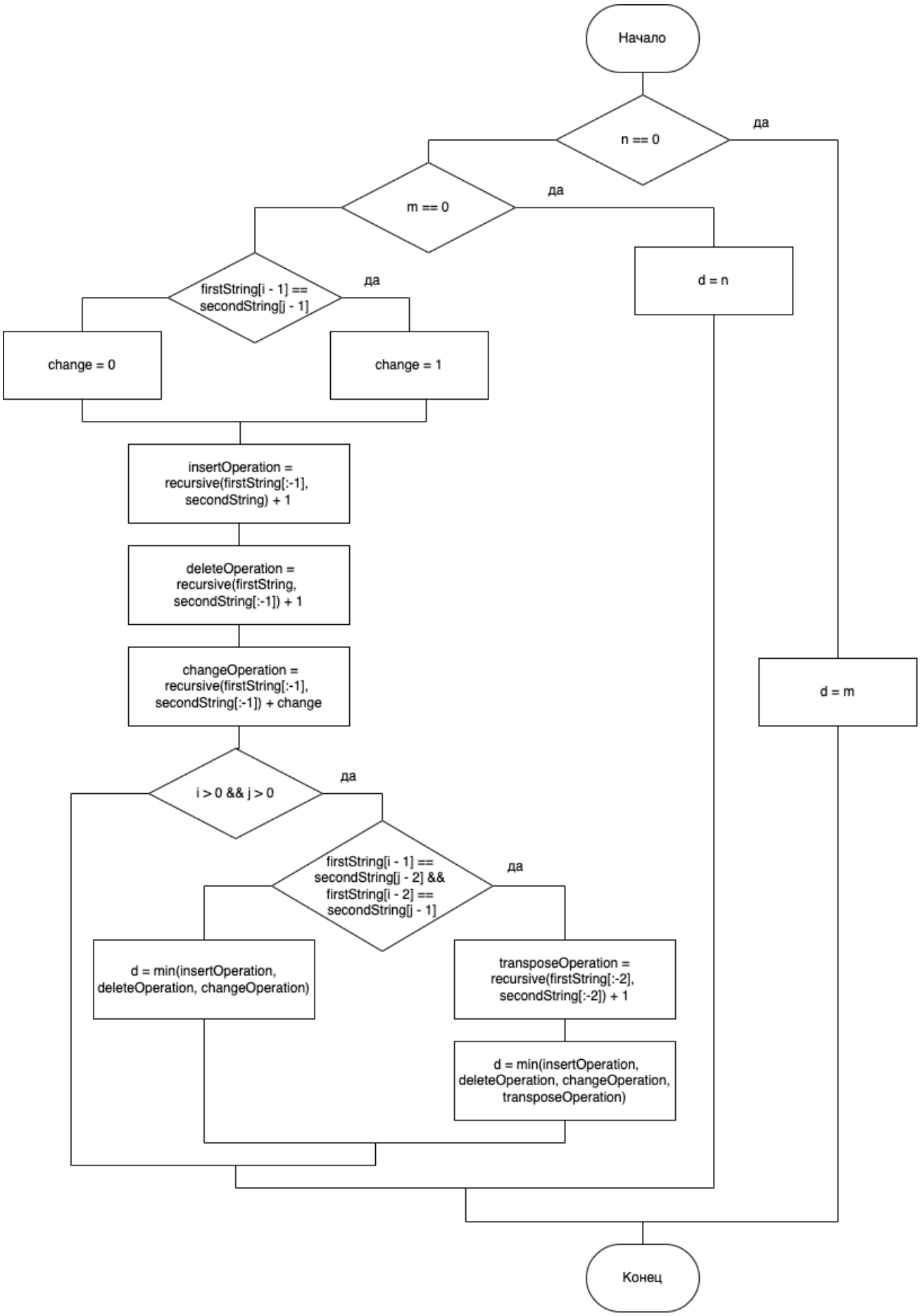

Рис. 2.3: Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кэширования

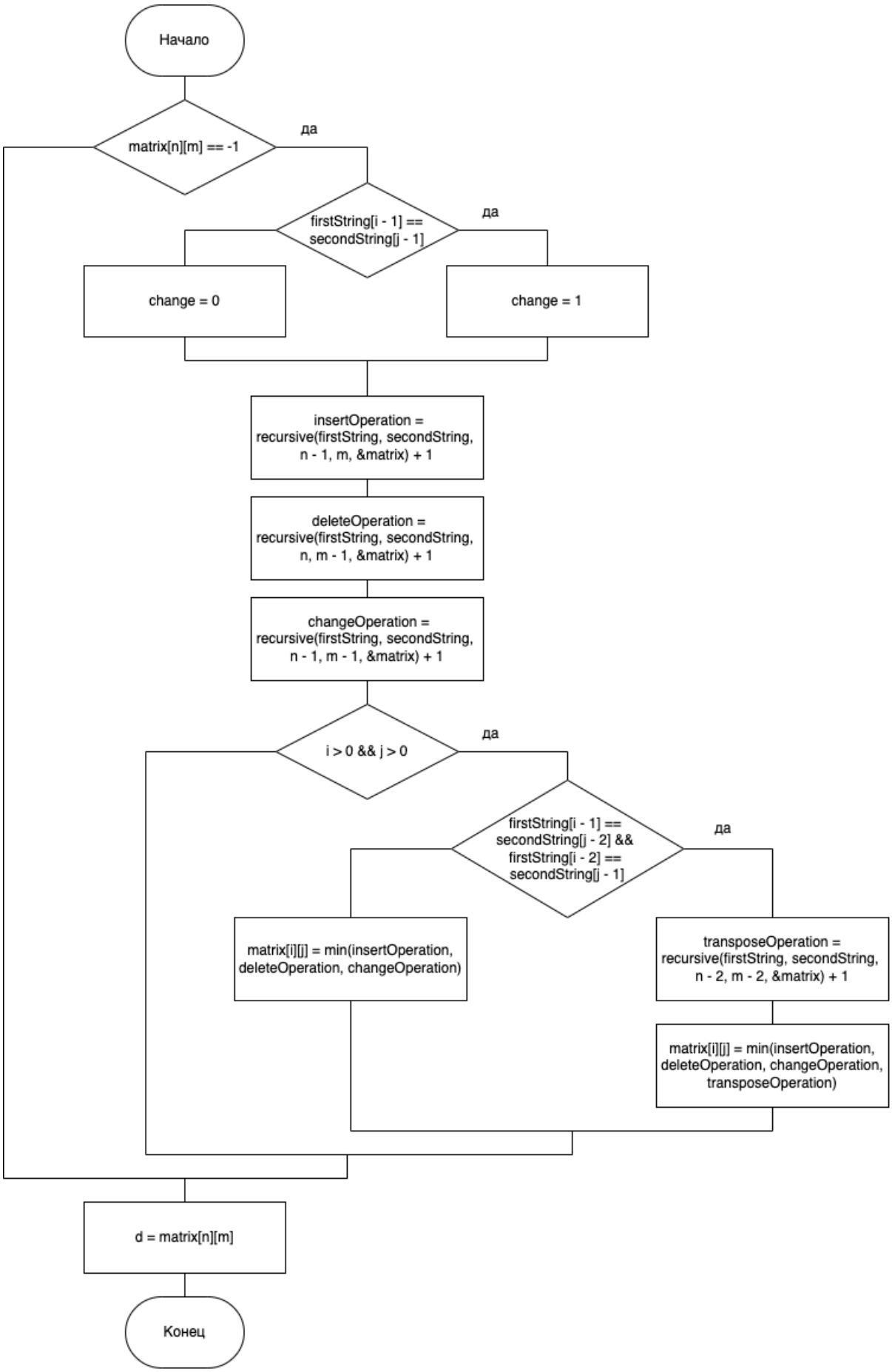

Рис. 2.4: Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна с кэшированием

3 Технологическая часть

В данном разделе производится выбор средств реализации, а также приводятся требования к программному обеспечению (ПО), листинги реализованных алгоритмов.

3.1 Требования к ПО

На вход программе подаются две строки, а на выходе должно быть получено искомое расстояние, рассчитанное с помощью каждого реализованного алгоритма: для расстояния Левенштейна – итеративный, для расстояния Дамерау-Левенштейна – итеративный, рекурсивный без кэша, рекурсивный с кэшем. Также необходимо вывести затраченное каждым алгоритмом процессорное время.

Интерфейс создаваемого приложения должен предоставлять возможность ввода двух строк и выбора алгоритма. В качестве результата наглядно должен быть представлен график работы алгоритмов: зависимость времени от количества повторений операции, а также искомое расстояние.

3.2 Выбор средств реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык Swift [2]. Данный язык создан компаний Apple для разработки программного обеспечения для macOS, iOS, watchOS и tvOS. Он содержит в себе большое количество инструментов, которые позволяют быстро создавать интерфейс приложений, а также реализовывать различные алгоритмы. Программное обеспечение, созданное посредством swift, позволит наглядно продемонстрировать скорость работы алгоритмов, а также упростить тестирование.

Кроме того, в Swift есть фреймворк CoreFoundation [3], который предоставляет функции для замера процессорного времени.

В качестве среды разработки выбран XCode [4]. Альтернативой ему выступает среда AppCode от JetBrains [5]. Однако Xcode, являясь офици-

альной средой разработки Apple, предоставляет возможность запуска симуляторов устройств Apple, что играет ключевую роль при создании масштабируемых приложений.

3.3 Листинги кода

В листингах 3.1 – 3.4 представлены реализации рассматриваемых алгоритмов.

3.4 Листинги кода

В Листинге 3.1 показана реализация матричного алгоритма нахождения расстояния Левенштейна.

```
1  func LevenshteinMatrix(_ firstString: String, _ secondString: String) ->
       Int {
2      let n = firstString.count
3      let m = secondString.count
4      var matrix = createMatrix(n: n + 1, m: m + 1, fill: 0)
5      for i in 1...n {
6          for j in 1...m {
7              let symbolN = firstString.index(firstString.startIndex,
       offsetBy: i - 1)
8              let symbolM = secondString.index(secondString.startIndex,
       offsetBy: j - 1)
9              let insertOperation = matrix[i - 1][j] + 1
10             let deleteOperation = matrix[i][j - 1] + 1
11             var changeOperation = matrix[i - 1][j - 1]
12
13             if firstString[symbolN] != secondString[symbolM] {
14                 changeOperation += 1
15             }
16
17             matrix[i][j] = min(insertOperation, deleteOperation,
       changeOperation)
18         }
19     }
20
21     return matrix[n][m]
22 }
```

Листинг 3.1: Функция нахождения расстояния Левенштейна матрично

В Листинге 3.2 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна.

```
func DamerauLevenshteinRecursive(_ firstString: String, _ secondString: String) -> Int {
    let n = firstString.count
    let m = secondString.count
    if n == 0 || m == 0 {
        if n != 0 { return n }
        if m != 0 { return m }

        return 0
    }

    let symbolN = firstString.index(before: firstString.endIndex)
    let symbolM = secondString.index(before: secondString.endIndex)
    let a = String(firstString[..<symbolN])
    let b = String(secondString[..<symbolM])

    var change = 0
    if firstString[symbolN] != secondString[symbolM] {
        change += 1
    }

    let insertOperation = DamerauLevenshteinRecursive(a, secondString) + 1
    let deleteOperation = DamerauLevenshteinRecursive(firstString, b) + 1
    let changeOperation = DamerauLevenshteinRecursive(a, b) + change

    if n > 1 && m > 1 {
        let symbolNMinusOne = firstString.index(firstString.endIndex, offsetBy: -2)
        let symbolMMinusOne = secondString.index(secondString.endIndex, offsetBy: -2)
        let c = String(firstString[..<symbolNMinusOne])
        let d = String(secondString[..<symbolMMinusOne])

        if firstString[symbolN] == secondString[symbolMMinusOne] && firstString[symbolNMinusOne] == secondString[symbolM] {
            let transposeOperation = DamerauLevenshteinRecursive(c, d) + 1
            return min(insertOperation, deleteOperation, changeOperation, transposeOperation)
        }
    }

    return min(insertOperation, deleteOperation, changeOperation)
}
```

Листинг 3.2: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

В Листинге 3.3 показана реализация матричного алгоритма нахождения расстояния Дамерау-Левенштейна.

```
func DamerauLevenshteinMatrix(_ firstString: String, _ secondString: String) -> Int {
    let n = firstString.count
    let m = secondString.count
    var matrix = createMatrix(n: n + 1, m: m + 1, fill: 0)

    for i in 1...n {
        for j in 1...m {
            let symbolN = firstString.index(firstString.startIndex, offsetBy: i - 1)
            let symbolM = secondString.index(secondString.startIndex, offsetBy: j - 1)

            let insertOperation = matrix[i - 1][j] + 1
            let deleteOperation = matrix[i][j - 1] + 1
            var changeOperation = matrix[i - 1][j - 1]
            if firstString[symbolN] != secondString[symbolM] {
                changeOperation += 1
            }

            if i > 1 && j > 1 {
                let symbolNMinusOne = firstString.index(firstString.startIndex, offsetBy: i - 2)
                let symbolMMinusOne = secondString.index(secondString.startIndex, offsetBy: j - 2)
                if firstString[symbolN] == secondString[symbolMMinusOne] && firstString[symbolNMinusOne] == secondString[symbolM] {
                    let transposeOperation = matrix[i - 2][j - 2] + 1
                    matrix[i][j] = min(insertOperation, deleteOperation, changeOperation, transposeOperation)
                } else {
                    matrix[i][j] = min(insertOperation, deleteOperation, changeOperation)
                }
            }
        }
    }
    return matrix[n][m]
}
```

Листинг 3.3: Функция нахождения расстояния Дамерау-Левенштейна матрично

В Листинге 3.4 показана реализация рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кэшем.

```
1  func DamerauLevenshteinRecursiveWithCash(_ firstString: String, _
       secondString: String) -> Int {
2      let n = firstString.count
3      let m = secondString.count
4      func recursive(firstString: String, secondString: String, n: Int, m:
       Int, matrix: inout [[Int]]) -> Int {
5          if matrix[n][m] != -1 { return matrix[n][m] }
6          let symbolN = firstString.index(firstString.startIndex, offsetBy:
       n - 1)
7          let symbolM = secondString.index(secondString.startIndex, offsetBy
       : m - 1)
8          var change = 0
9          if firstString[symbolN] != secondString[symbolM] { change = 1 }
10         let insertOperation = recursive(firstString, secondString, n - 1,
       m, &matrix) + 1
11         let deleteOperation = recursive(firstString, secondString, n, m -
       1, &matrix) + 1
12         let changeOperation = recursive(firstString, secondString, n - 1,
       m - 1, &matrix) + change
13         if n > 1 && m > 1 {
14             let symbolNMinusOne = firstString.index(firstString.startIndex
       , offsetBy: n - 2)
15             let symbolMMinusOne = secondString.index(secondString.
       startIndex, offsetBy: m - 2)
16             matrix[n][m] = min(insertOperation, deleteOperation,
       changeOperation)
17             if firstString[symbolN] == secondString[symbolMMinusOne] &&
       firstString[symbolNMinusOne] == secondString[symbolM] {
18                 let transposeOperation = recursive(firstString,
       secondString, n - 2, m - 2, &matrix) + 1
19                 matrix[n][m] = min(insertOperation, deleteOperation,
       changeOperation, transposeOperation)
20             }
21         } else {
22             matrix[n][m] = min(insertOperation, deleteOperation,
       changeOperation)
23         }
24         return matrix[n][m]
25     }
26     var matrix =  createMatrix(n: n + 1, m: m + 1, fill: -1)
27     _ = recursive(firstString, secondString, n, m, &matrix)
28     return matrix[n][m]
29 }
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно с кэшем

3.5 Тестовые данные

В таблице 3.1 приведены входные данные, на которых было протестированно разработанное ПО.

Таблица 3.1: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1	а	б	1 1 1 1	1 1 1 1
2	кот	скат	2 2 2 2	2 2 2 2
3	море	моер	2 1 1 1	2 1 1 1
4	абаваба	аабвааб	4 2 2 2	4 2 2 2
5	собака	собака	0 0 0 0	0 0 0 0
6	qwerty	queue	4 4 4 4	4 4 4 4
7	apple	aplpe	2 1 1 1	2 1 1 1
8	12	one	3 3 3 3	3 3 3 3
9	календула	конфета	6 6 6 6	6 6 6 6
10	конфета	календула	6 6 6 6	6 6 6 6

Вывод

Был производен выбор средств реализации и реализованы алгоритмы поиска расстояний: Левештнейна – итеративный, Дамерау-Левенштейна – итеративный, рекурсивный с кэшем и рекурсивный без кэша. Приведены листинги кода на выбранном языке программирования, а также представлена таблица, отображающая результаты работы программы на предложенных наборах тестовых данных.

4 Исследовательская часть

4.1 Интерфейс приложения

На рисунках 4.1 – 4.2 приведено изображение интерфейса главного экрана приложения.

Главный экран приложения дает возможность ввести две строки – слова, для которых будет вычисляться расстояние, а также выбрать метод, с помощью которого оно будет найдено. При нажатии на кнопку «Рассчитать», появляется второй экран, отображающий график зависимости времени (в секундах) от количества произведенных операций и результирующее расстояние.

4.2 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование:

- операционная система: iOS 14.5;
- оперативная память: 4 Гб;
- процессор: Apple A14 Bionic 2990 МГц [6];

Во время тестирования iPad не был подключен к другим устройствам и был включен в сеть питания.

4.3 Время выполнения реализаций алгоритмов

Все реализации алгоритмов сравнивались на строках длиной от 1 до 10 с шагом 100 в диапазоне повторений операций от 1 до 2000.

На рисунках 4.3 – 4.8 представлены результаты выполнения алгоритмов для строк «а» и «б», «кот» и «скат», «кенгуру» и «кентавр».

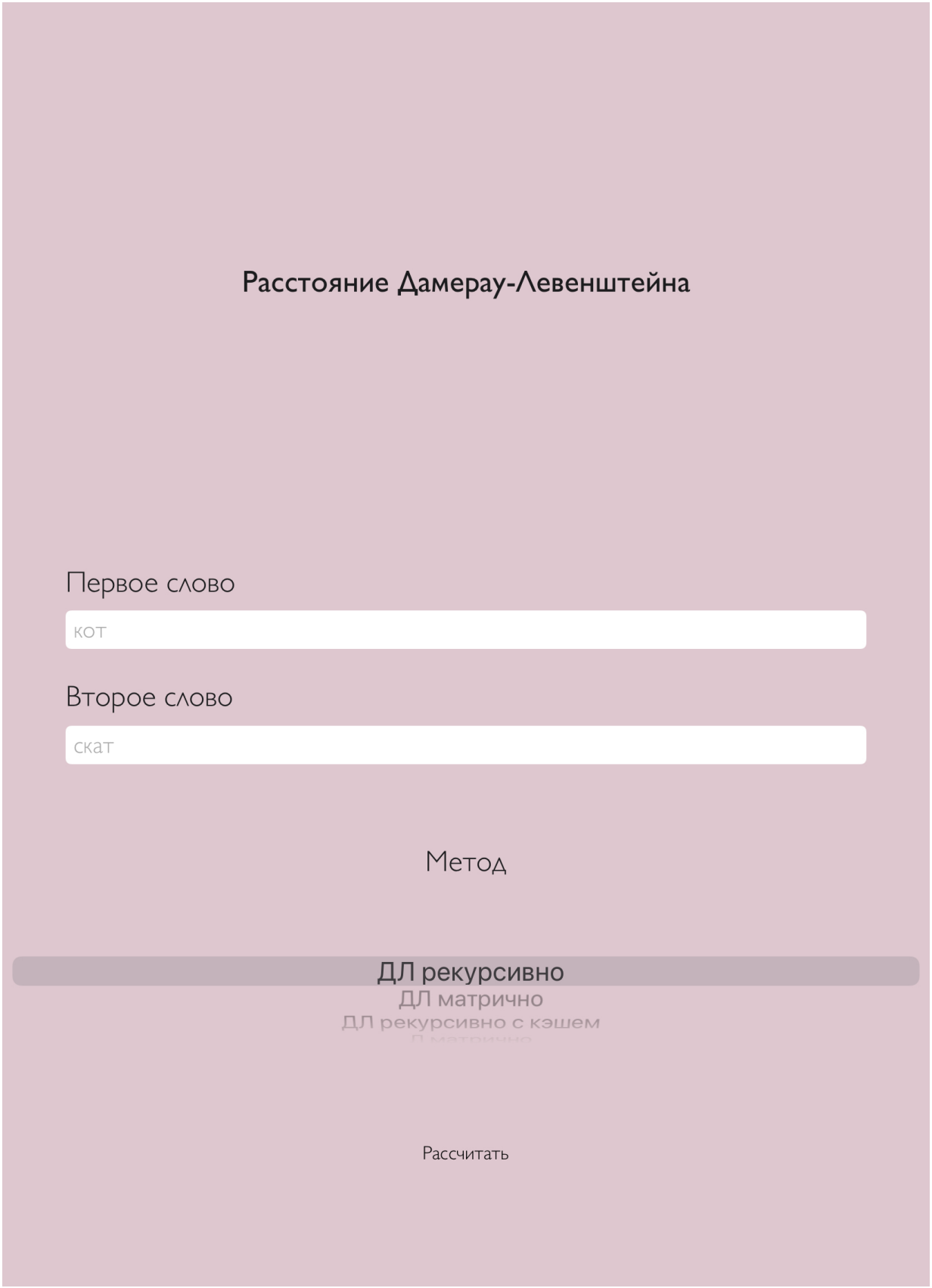

Рис. 4.1: Интерфейс

На рисунке 4.3 приведены результаты сравнения времени работы всех реализаций для слов малой длины, 4.5 – результаты сравнения времени ра-

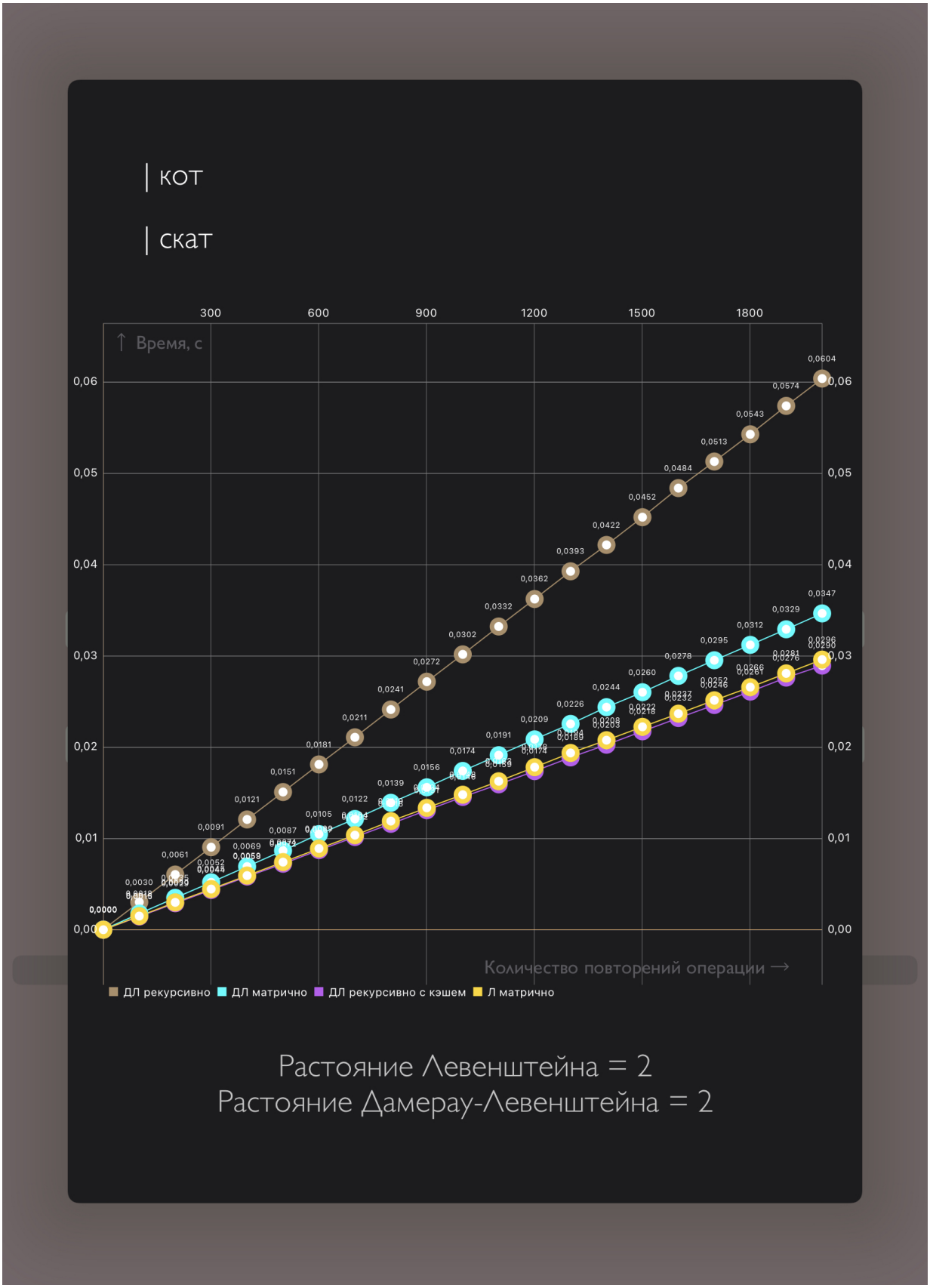

Рис. 4.2: Экран с результатом

боты всех реализаций для слов средней длины, 4.7 – результаты сравнения времени работы всех реализаций для слов большой длины.

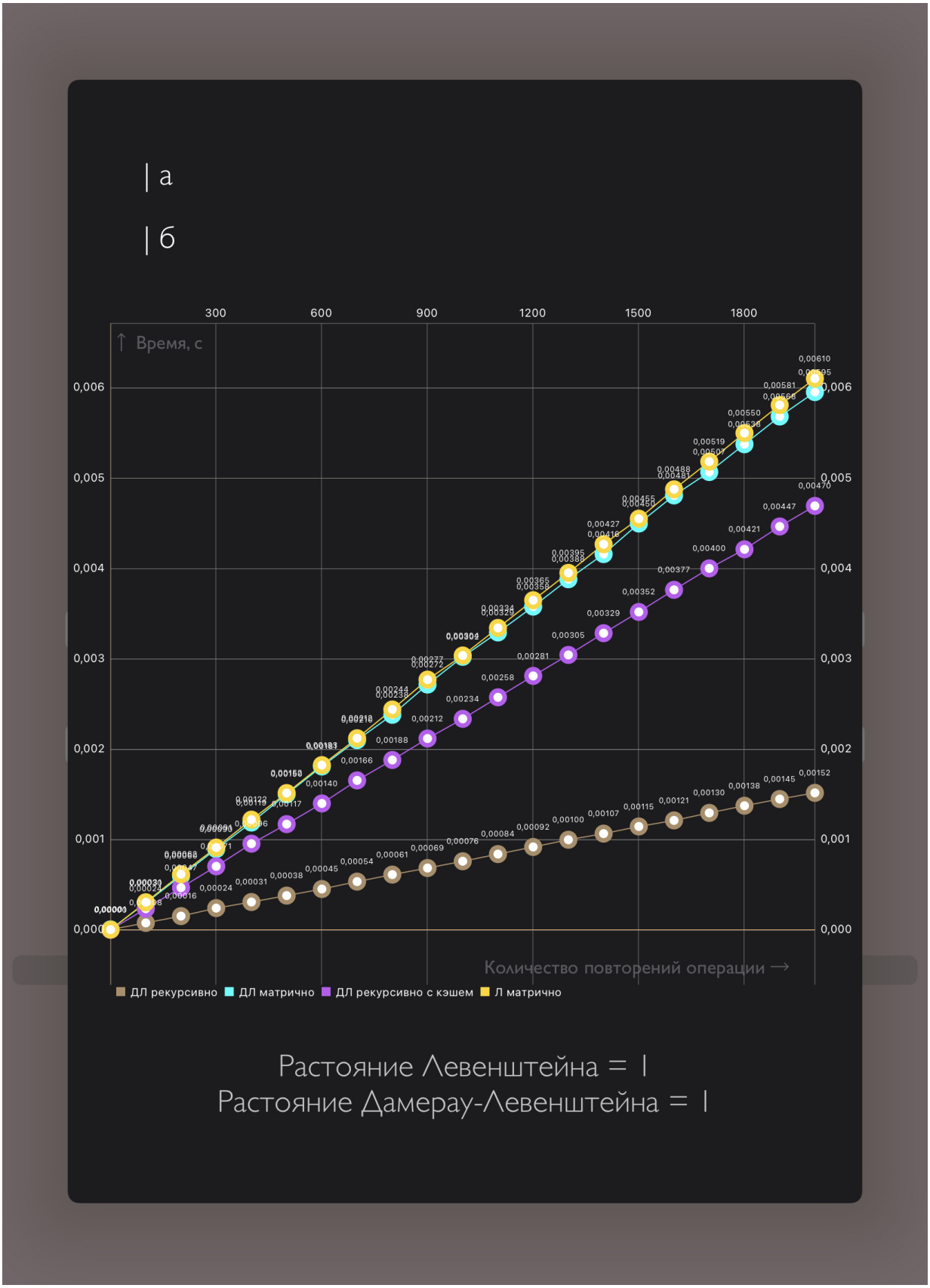

Рис. 4.3: Сравнение времени работы реализаций алгоритмов при малой длине слов (1-2 символа)

```
Входные данные №1: а
Входные данные №2: б
Матрица (Расстояние Левенштейна итеративно)

0 1
1 1
Расстояние Левенштейна итеративно -> 1

Расстояние Дамерау-Левенштейна рекурсивно -> 1

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1
1 1
Расстояние Дамерау-Левенштейна итеративно -> 1

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1
1 1
Расстояние Дамерау-Левенштейна рекурсивно с кэшем -> 1
```

Рис. 4.4: Матрицы алгоритмов при малой длине слов (1-2 символа)

При малой длине слов эффективнее всего использовать рекурсивный метод без кэша, а при средней и большой длине слов более рационально использование матричного алгоритма: рекурсивный алгоритм в данном случае проигрывает по времени. Причем при наибольшей рассматриваемой длине слов – в разы. Также можно отметить, что при малой длине слов алгоритм Левенштейна проигрывает по времени реализациям Дамерау-Левенштейна. Однако при увеличении длины слов уступает только рекурсивному алгоритму, использующему кэш.

4.4 Используемая память

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна не отличаются друг от друга с точки зрения использования памяти.

Пусть длина строки S1 - n, длина строки S2 - m, тогда затраты памяти для рекурсивного и итеративного алгоритмов будут следующими:

- матричный алгоритм Левенштейна:
  - строки S1, S2 – $(m + n) \cdot MemoryLayout < Character > .size$

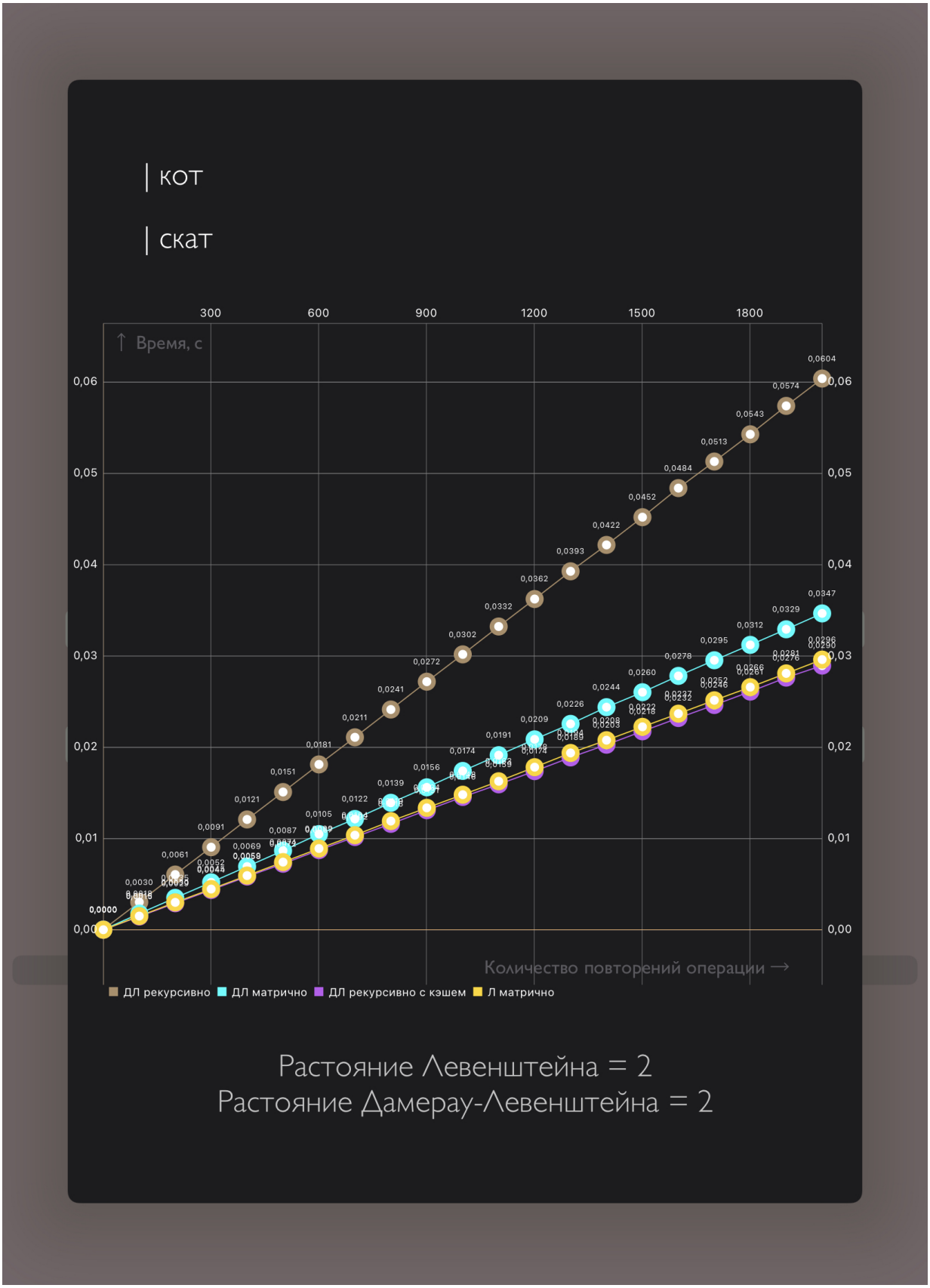

Рис. 4.5: Сравнение времени работы реализаций алгоритмов при средней длине слов (3-6 символов)

```
Входные данные №1: кот
Входные данные №2: скат
Матрица (Расстояние Левенштейна итеративно)

0 1 2 3 4
1 1 1 2 3
2 2 2 2 3
3 3 3 3 2
Расстояние Левенштейна итеративно -> 2

Расстояние Дамерау-Левенштейна рекурсивно -> 2

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1 2 3 4
1 1 1 2 3
2 2 2 2 3
3 3 3 3 2
Расстояние Дамерау-Левенштейна итеративно -> 2

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1 2 3 4
1 1 1 2 3
2 2 2 2 3
3 3 3 3 2
Расстояние Дамерау-Левенштейна рекурсивно с кэшем -> 2
```

Рис. 4.6: Матрицы алгоритмов при средней длине слов (3-6 символов)

  - матрица – $((m+1)\cdot(n+1))\cdot MemoryLayout<Int>.size$
  - текущая строка матрицы – $(n+1)\cdot MemoryLayout<Int>.size$
  - длины строк – $2\cdot MemoryLayout<Int>.size$
  - вспомогательные переменные – $3\cdot MemoryLayout<Int>.size$

- рекурсивный алгоритм Дамерау-Левенштейна (для каждого вызова):
  - строки S1, S2 – $(m+n)\cdot MemoryLayout<Character>.size$
  - длины строк – $2\cdot MemoryLayout<Int>.size$
  - вспомогательные переменные – $4\cdot MemoryLayout<Int>.size$
  - адрес возврата

Существенной является разница затрат памяти, используемой для хранения матрицы в итеративном алгоритме, и памяти, используемой для хранения строки в рекурсивном алгоритме. Очевидно, что произведение длин

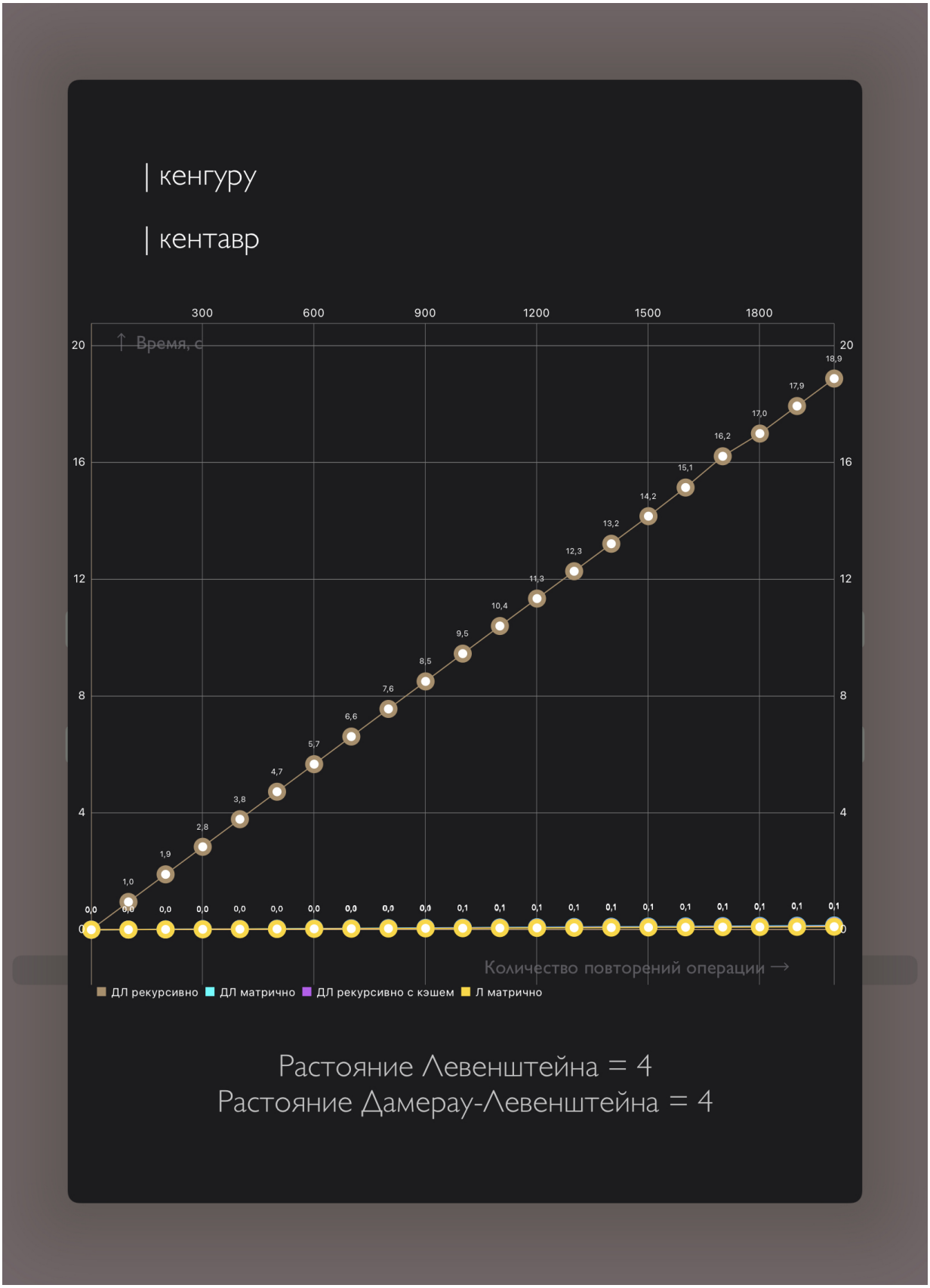

Рис. 4.7: Сравнение времени работы реализаций алгоритмов при большой длине слов (7-10 символов)

```
Входные данные №1: кенгуру
Входные данные №2: кентавр
Матрица (Расстояние Левенштейна итеративно)

0 1 2 3 4 5 6 7
1 0 1 2 3 4 5 6
2 1 0 1 2 3 4 5
3 2 1 0 1 2 3 4
4 3 2 1 1 2 3 4
5 4 3 2 2 2 3 4
6 5 4 3 3 3 3 3
7 6 5 4 4 4 4 4
Расстояние Левенштейна итеративно -> 4

Расстояние Дамерау-Левенштейна рекурсивно -> 4

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1 2 3 4 5 6 7
1 0 1 2 3 4 5 6
2 1 0 1 2 3 4 5
3 2 1 0 1 2 3 4
4 3 2 1 1 2 3 4
5 4 3 2 2 2 3 4
6 5 4 3 3 3 3 3
7 6 5 4 4 4 4 4
Расстояние Дамерау-Левенштейна итеративно -> 4

Матрица (Расстояние Дамерау-Левенштейна рекурсивно с кэшем)

0 1 2 3 4 5 6 7
1 0 1 2 3 4 5 6
2 1 0 1 2 3 4 5
3 2 1 0 1 2 3 4
4 3 2 1 1 2 3 4
5 4 3 2 2 2 3 4
6 5 4 3 3 3 3 3
7 6 5 4 4 4 4 4
Расстояние Дамерау-Левенштейна рекурсивно с кэшем -> 4
```

Рис. 4.8: Матрицы алгоритмов при при большой длине слов (7-10 символов)

строк требует больших затрат по памяти, нежели сумма. При этом память, затрачиваемая на хранение вспомогательных переменных, длин строк и прочего, не играет ключевой роли.

Вывод

Рекурсивные реализации алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна, не использующие кэширование, работают на порядок дольше итеративных реализаций. При применении кэширования они требуют меньше времени, однако все равно уступают по производительности итеративным алгоритмам, особенно при большой длине строк.

Но по расходу памяти итеративные реализации проигрывают рекурсивным: максимальный размер используемой памяти в них пропорционален произведению длин строк, в то время как в рекурсивных — сумме длин строк.

Если же применить к итеративным реализациям оптимизацию по памяти, то они будут выигрывать как по затрачиваемой памяти, так и по времени выполнения.

Заключение

В результате выполнения лабораторной работы при исследовании алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна были применены и отработаны навыки динамического программирования, а также создания программного обеспечения для iOS.

В ходе выполнения лабораторной работы были выполнены следующие задачи:

1) изучены расстояния Левенштейна и Дамерау-Левенштейна;

2) разработаны и реализованы алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна;

3) создан программный продукт, позволяющий протестировать реализованные алгоритмы;

4) проведен сравнительный анализ процессорного времени выполнения реализаций данных алгоритмов: выявлено, что реализация рекурсивного алгоритма без кэширование уступает матричному и алгоритму, использующему рекурсию с кэшем, при средней и большой длине слов, но является самым эффективным по памяти при малой длине слов;

5) проведен сравнительный анализ затрачиваемой алгоритмами памяти: выявлено, что итеративные реализации уступают рекурсивным по данному параметру;

6) был подготовлен отчет по лабораторной работе.

Литература

[1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады АН СССР, 1965. Т. 163. С. 845–848.

[2] Swift: Swift | Мультипарадигмальный компилируемый язык [Электронный ресурс]. Режим доступа: `https://www.apple.com/swift/`

[3] Core Foundation: Core Foundation | Time Utilities [Электронный ресурс]. Режим доступа: `https://developer.apple.com/documentation/corefoundation/time_utilities`

[4] Xcode: Xcode [Электронный ресурс]. Режим доступа: `https://developer.apple.com/xcode/`

[5] AppCode: AppCode | Smart Swift/Objective-C IDE for iOS [Электронный ресурс]. Режим доступа: `https://www.jetbrains.com/objc/`

[6] iPad: iPad | Apple A14 Bionic [Электронный ресурс]. Режим доступа: `https://developer.apple.com/xcode/`