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»

НАПРАВЛЕНИЕ ПОДГОТОВКИ «09.03.04 Программная инженерия»

ОТЧЕТ

ПО ЛАБОРАТОРНОЙ РАБОТЕ №1

Название: Расстояние Левенштейна и Дамерау – Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-54Б		С. Д. Параскун
	Группа	Подпись, дата	И. О. Фамилия
Преподаватель			Л. Л. Волкова
		Подпись, дата	И. О. Фамилия

Москва, 2021 г.

Содержание

Страница

Введение

В настоящее время перед компьютерной лингвистикой ставится множество задач. Одна из них - поиск редакционного расстояния между строками. Это определение минимального количества редакционных операций, необходимых для превращения одной строки в другую. Впервые эту задачу обозначил В. И. Левенштейн, имя которого закрепилось за ней.

При вычислении расстояния Левенштейна редакционные операции ограничиваются вставкой, удалением и заменой. В случае расстояния Дамерау - Левенштейна к операциям добавляется транспозиция - перестановка двух соседних символов.

Данные алгоритмы находят применение не только в компьютерной лингвистике для исправления ошибок или автозамены слов, но также в биоинформатике для определения разных участков ДНК и РНК.

Существует множество модификаций упомянутых алгоритмов. В данной работе будут рассмотрены лишь те, которые используют парадигмы динамического программирования.

Целью данной работы является получение навыков динамического программирования. Для достижения поставленной цели необходимо выполнить следующие задачи:

- Изучить реализацию алгоритмов Левенштейна и Дамерау-Левенштейна;
- Составить схемы данных алгоритмов;
- Применить методы динамического программирования при реализации алгоритмов Левенштейна и Дамерау-Левенштейна;
- Провести сравнительный анализ алгоритмов по затраченным ресурсам (время и память);
- Описать и обосновать полученные результаты.

1. Аналитический раздел

В данном разделе будут представлены описания алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Редакторским расстоянием или расстоянием Левенштейна [1] называется минимальное количество операций, необходимых для преобразования одной строки в другую. Множество операций состоит из вставки (I), удаления (D) и замены символов (R). Для каждой операции введен так называемый штраф (цена), равный 1. Также существет операция совпадение (M), оцениваемая нулевым штрафом. Суть алгоритма заключается в поиске такой последовательности операций, которая оценится минимальным штрафом.

1.2 Алгоритм с кэшем в форме двух строк

Расчет будем вести по рекуррентной формуле через расстояние между подстроками. Положим, что s1[1..i] - подстрока s1 длинной i, s2[1..j] - подстрока s2 длинной j. Тогда

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, i = 0, j = 0 \\ j, i = 0, j > 0 \\ i, i > 0, j = 0) \\ \alpha(s1, s2) \end{cases} \tag{1.1}$$

где:

$$\alpha(s1,s2) = min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, \text{если s1[i] = s2[j]} \\ 1, \text{иначе} \end{cases} \end{cases} \tag{1.2}$$

Исходя из данной рекуррентной формулы, мы можем составить кэш [2] в форме матрицы, который будет хранить ранее просчитанные варианты, однако объем используемой памяти можно сократить, если использовать не матрицу, а две строки. Всего при наличии двух строк мы сможем обращаться к результатам вычислений соседних подстрок, а значит мы сможем рассчитать нужное нам расстояние.

1.3 Рекурсивный алгоритм без кэша

Суть рекурсивного алгоритма заключается в том, что каждый раз при обработке основных строк s1[1..i] и s2[1..j] мы вызываем алгоритм для обработки трех вариантов изменения строк: D(s1[1..i - 1], s2[1..j]), D(s1[1..i - 1], s2[1..j - 1]), D(s1[1..i], s2[1..j - 1]).

Однако у данного варианта есть значительный недостаток - повторные вычисления. Каждый вызов будет обрабатываться индивидуально, а значит для одинаковых входных параметров пересчет будет вестись снова и снова. Чтобы избавиться от многократного определения расстояния можно использовать кэш.

1.4 Рекурсивный алгоритм с кэшем в форме матрицы

В отличие от предыдущего алгоритма, мы создаем матрицу, все элементы которой изначально инициализируем бесконечностью. Тогда в случае вычисления расстояния для какой-либо из подстрок, мы сможем сохранить

значение. Таким образом, если элемент матрицы будет равен бесконечности - он еще не рассчитан, а значит это действие необходимо выполнить; в обратном случае будет возвращен данный элемент.

1.5 Расстояние Дамерау - Левенштейна без кэша

В отличие от расстояния Левенштейна при поиске расстояния Дамерау-Левенштейна [3] ко множеству операций добавляется транспозиция (T) или по-другому обмен (X). Штраф для данной операции также будем считать за 1. Благодаря тому, что можно использовать перестановку 2-х соседних символов, редакционное расстояние может оказаться меньше. Если s1[i - 2] = s2[j - 1] и s1[i - 1] = s2[j - 2], формула (1.2) примет вид:

$$\alpha(s1, s2) = min \begin{cases} D(s1[1..i], s2[1..j-1]) + 1 \\ D(s1[1..i-1], s2[1..j]) + 1 \\ D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, \text{если s1[i] = s2[j]} \\ 1, \text{иначе} \end{cases} \\ D(s1[1..i-2], s2[1..j-2]) + 1 \end{cases} \tag{1.3}$$

1.6 Вывод

Все формулы подсчета редакционного расстояния рекурентны, реализовать их можно как итерационно, так и рекурсивно. На вход алгоритмам будут подаваться две строки и их длины, сравниваться могут строки как в английской, так и в русской раскладке, а также будут обрабатываться пустые строки. Реализуемое ПО будет работать в пользовательском режиме (вывод расстояний Левенштейна и Дамерау-Левенштейна), а также в экспериментальном (проведение замеров времени выполнения алгоритмов).

2. Конструкторский раздел

В данном разделе будут спроектированы схемы алгоритмов.

2.1 Схемы алгоритмов

На вход алгоритмов подаются строки s1 и s2, а также их длины i и j соотетственно. На выходе единственное число - искомое расстояние.

На рисунках 2.1 - 2.4 представлены схемы рассматриваемых алгоритмов.

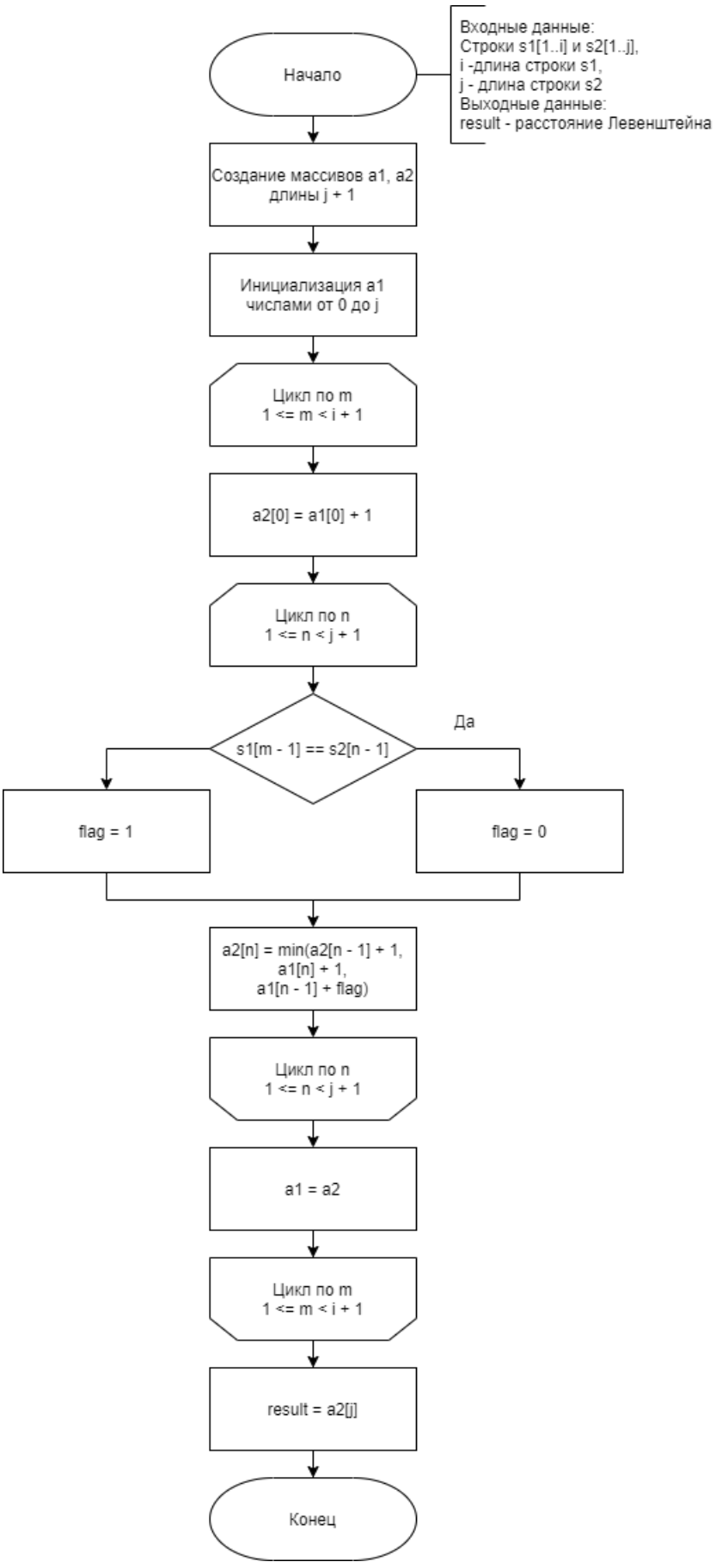

Рисунок 2.1 – Схема нерекурсивного алгоритма поиска расстояния Левенштейна с кэшем в форме двух строк

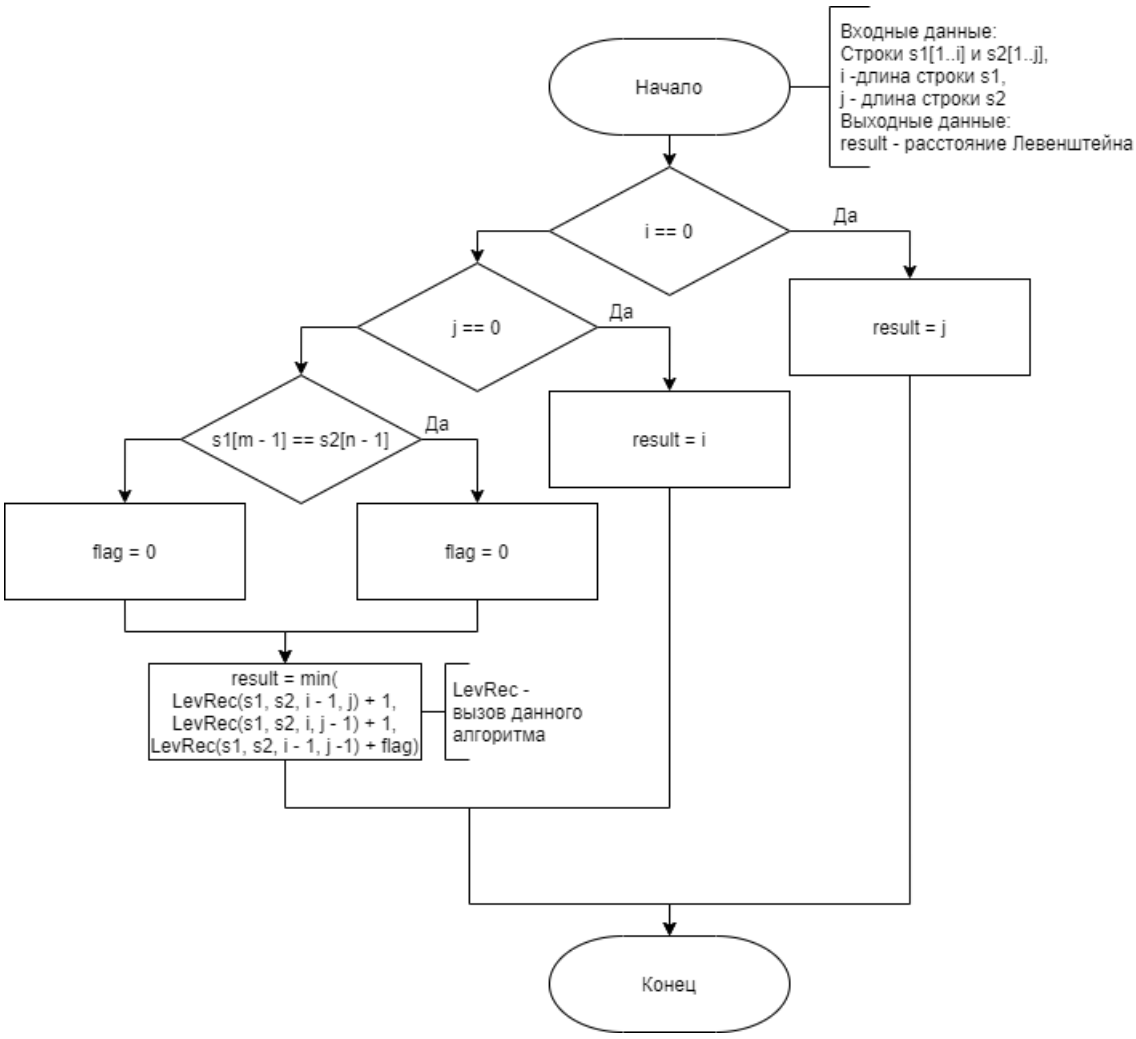

Рисунок 2.2 – Схема рекурсивного алгоритма поиска расстояния Левенштейна без кэша

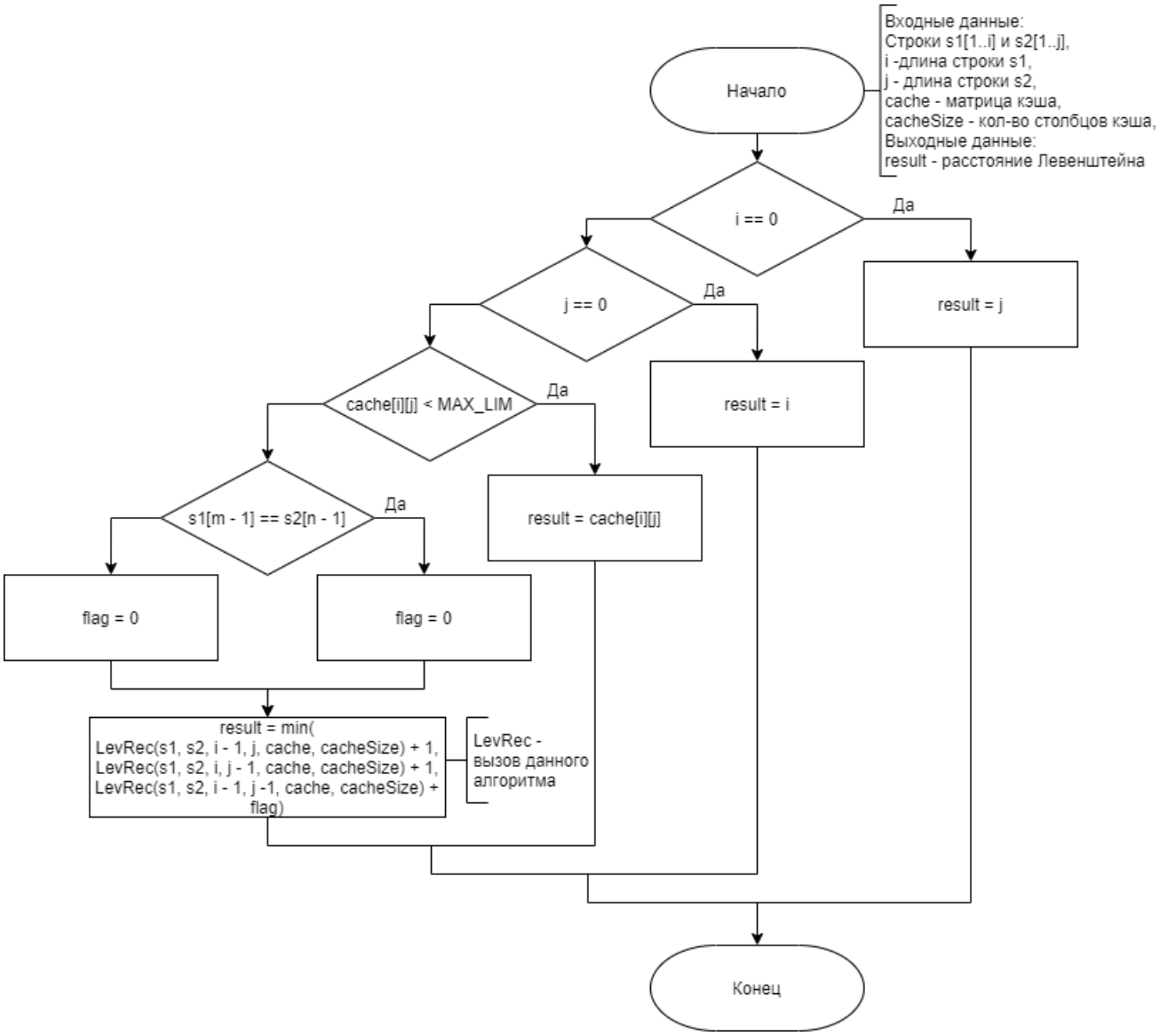

Рисунок 2.3 – Схема рекурсивного алгоритма поиска расстояния Левенштейна с кэшем в форме матрицы

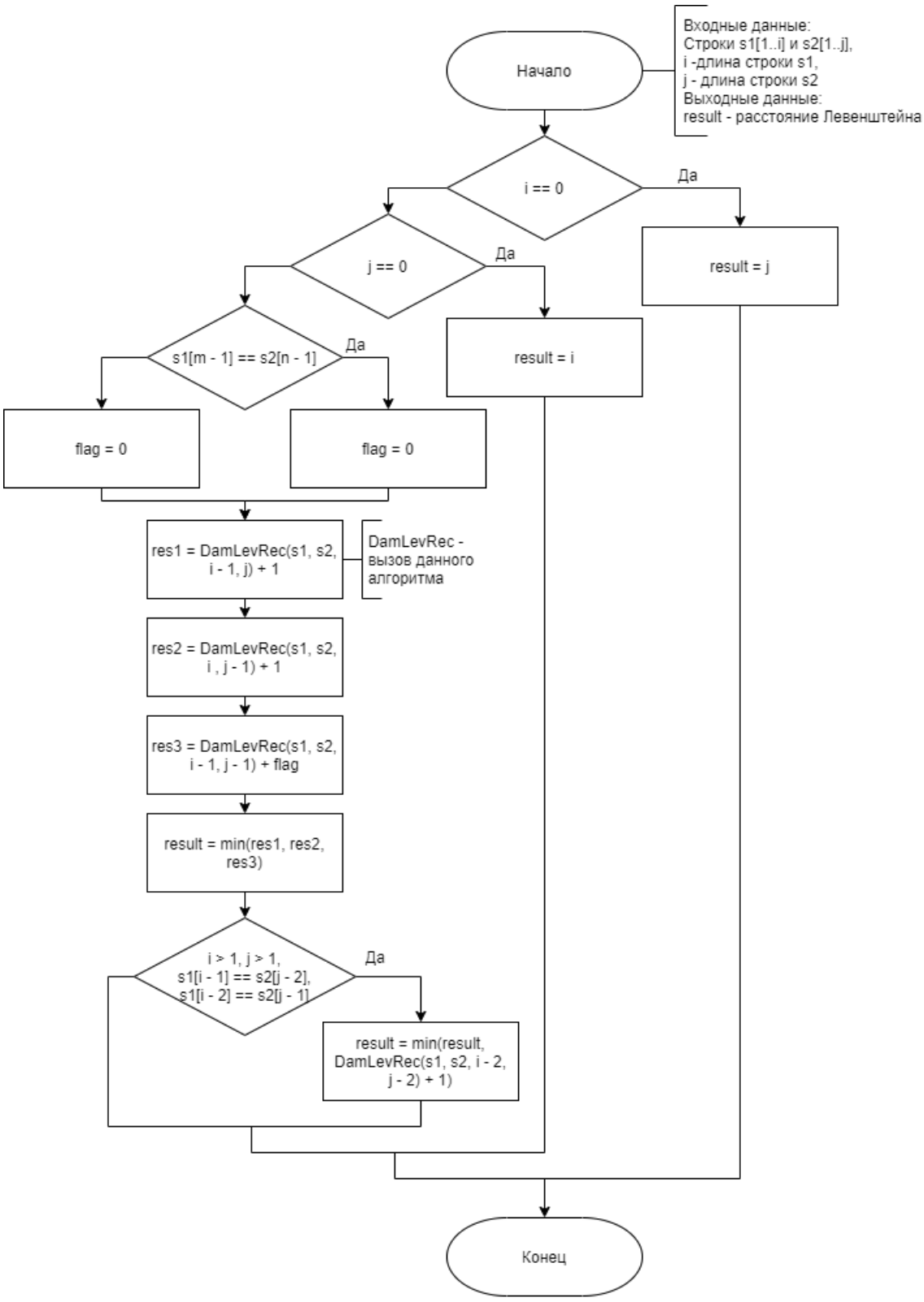

Рисунок 2.4 – Схема рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна без кэша

2.2 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- Строка - массив типа char размером длины строки;
- Длина строки - целое число int;
- Кэш в форме двух строк - два массива типа int размером длина второй строки + 1;
- Кэш в форме матрицы - матрица типа int размером (длина первой строки + 1) * (длина второй строки + 1).

2.3 Оценка памяти

Так как алгоритмы поиска расстояния Левенштейна и Дамерау-Левенштейна не отличаются друг от друга с точки зрения использования памяти, рассмотрим разницу рекурсивной и нерекурсивной реализации с кэшем.

Пусть s1 и s2 строки длины i и j соответственно.

Тогда для нерекурсивного алгоритма с кэшем в форме двух строк будет затрачена память на:

- Строки s1 и s2 - (i + j) * sizeof(char)
- Длины строк i и j - 2 * sizeof(int)
- Кэш в 2 строки размерностью j + 1 - 2 * (j + 1) * sizeof(int)
- вспомогательные переменные - 3 * sizeof(int)

Для рекурсивных реализаций глубина стека вызовов равна сумме длин строк i + j.

Для рекурсивного алгоритма Левенштейна без кэша будет затрачена память:

- Строки s1 и s2 - (i + j) * sizeof(char)
- Длины строк i и j - 2 * sizeof(int)
- Вспомогательные переменные - 2 * sizeof(int)
- Адрес возврата

Для рекурсивного алгоритма Левенштейна с кэшем в форме матрицы будет затрачена память:

- Строки s1 и s2 - (i + j) * sizeof(char)
- Длины строк i и j - 2 * sizeof(int)
- Вспомогательные переменные - 2 * sizeof(int)
- Память под матрицу кэша - (i + 1) * (j + 1) * sizeof(int)
- Указатель на матричный кэш - 4 байта
- Адрес возврата

Для рекурсивного алгоритма Дамерау-Левенштейна без кэша будет затрачена память:

- Строки s1 и s2 - (i + j) * sizeof(char)
- Длины строк i и j - 2 * sizeof(int)
- Вспомогательные переменные - 5 * sizeof(int)
- Адрес возврата

Рекурсивные алгоритмы выигрывают по затрачиваемой памяти, так как она растет как сумма длин строк, в то время как в итеративном алгоритме - произведение длин строк.

2.4 Структура ПО

ПО будет состоять из следующих модулей:

- Главный модуль - из него будет осуществляться запуск программы и выбор соответствующего режима работы;
- Модуль интерфейса - в нем будет описана реализация режимов работы программы;
- Модуль, содержащий реализации алгоритмов.

2.5 Вывод

На основе полученных в аналитическом разделе формул были спроектированы схемы алгоритмов, выбраны используемые типы данных, проведена оценка затрачиваемого объема памяти, а также описана структура ПО.

3. Технологический раздел

В данном разделе будут приведены требования к ПО, средства его реализации и листинга кода алгоритмов, а также рассмотрены тестовые случаи.

3.1 Требования к ПО

Программное обеспечение должно удовлетворять следующим требованиям:

- ПО принимает на вход текстовые данные в любой раскладке, длиной не более 100 символов;
- ПО возвращает редакционное расстояние.

3.2 Средства реализации

Для реализации ПО был выбран компилируемый язык C. В качестве среды разработки - QtCreator. Оба средства были выбраны из тех соображений, что навыки работы с ними были получены в более ранних курсах.

3.3 Листинги кода

В листингах 3.1 - 3.4 приведены реализации алгоритмов, изученных в аналитическом разделе.

Листинг 3.1 – Нерекурсивный алгоритм поиска расстояния Левенштейна с кэшем в форме 2 строк

```
int levCacheTwoRows(char *s1, int i, char *s2, int j)
{
    int *a1 = malloc(sizeof(int) * (j + 1));
    int *a2 = malloc(sizeof(int) * (j + 1));
    if (!a1 && !a2)
        return -1;

```

```
    for (int m = 0; m < j + 1; m++)
        a1[m] = m;

    int flag;
    for (int m = 1; m < i + 1; m++)
    {
        a2[0] = a1[0] + 1;
        for (int n = 1; n < j + 1; n++)
        {
            if (s1[m - 1] == s2[n - 1])
                flag = 0;
            else
                flag = 1;
            a2[n] = min(a2[n - 1] + 1, a1[n] + 1, a1[n - 1] + flag);
        }
        for (int n = 0; n < j + 1; n++)
            a1[n] = a2[n];
    }
    int result = a2[j];
    free(a1);
    free(a2);
    return result;
}
```

Листинг 3.2 – Рекурсивный алгоритм поиска расстояния Левенштейна без кэша

```
int levRecWithoutCache(char *s1, int i, char *s2, int j)
{
    if (i == 0)
        return j;
    else if (j == 0)
        return i;

    int flag;
    if (s1[i - 1] == s2[j - 1])
        flag = 0;
    else
        flag = 1;
    int result = min(levRecWithoutCache(s1, i - 1, s2, j) + 1,\
                     levRecWithoutCache(s1, i, s2, j - 1) + 1,\
                     levRecWithoutCache(s1, i - 1, s2, j - 1) + flag);
    return result;
}
```

Листинг 3.3 – Рекурсивный алгоритм поиска расстояния Левенштейна с кэшем в форме матрицы

```
int levRecWithCache(char *s1, int i, char *s2, int j, int *cache, int cacheSize)
{
    if (i == 0)
        return j;
    else if (j == 0)
        return i;

    if (cache[i * cacheSize + j] < MAX_LIM)
        return cache[i * cacheSize + j];

    int flag;
    if (s1[i - 1] == s2[j - 1])
        flag = 0;
    else
        flag = 1;
    cache[i * cacheSize + j] = min(levRecWithCache(s1, i - 1, s2, j,\
     cache, cacheSize) + 1,\
                    levRecWithCache(s1, i, s2, j - 1, cache,\
                     cacheSize) + 1,\
                    levRecWithCache(s1, i - 1, s2, j - 1, cache,\
                     cacheSize) + flag);
    return cache[i * cacheSize + j];
}
```

Листинг 3.4 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна без кэша

```
int damLevRecWithoutCache(char *s1, int i, char *s2, int j)
{
    if (i == 0)
        return j;
    else if (j == 0)
        return i;

    int flag;
    if (s1[i - 1] == s2[j - 1])
        flag = 0;
    else
        flag = 1;
    int res1 = damLevRecWithoutCache(s1, i - 1, s2, j) + 1;
    int res2 = damLevRecWithoutCache(s1, i, s2, j - 1) + 1;
    int res3 = damLevRecWithoutCache(s1, i - 1, s2, j - 1) + flag;

    int result = min(res1, res2, res3);
    if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
        result = min(result, damLevRecWithoutCache(s1, i - 2, s2,\
         j - 2) + 1, MAX_LIM);
```

```
21      return result;
22  }
```

3.4 Тестирование ПО

В таблице 3.1 приведены тестовые случаи для алгоритмов поиска редакционного расстояния. Случай 1 описывает ввод двух пустых строк, случай 2 - одна из строк пустая, другая нет, случаи 3 - 5 - расстояния Левенштейна и Дамерау-Левенштейна равны, случаи 6 - 7 - расстояния Левенштейна и Дамерау-Левенштейна дают разные результаты.

Таблица 3.1 – Тестовые случаи

№	Строка 1	Строка 2	Левенштейн	Д.-Левенштейн
1			0	0
2		вуз	3	3
3	скат	кот	2	2
4	дрова	двор	3	3
5	brown	born	2	2
6	музыка	мзуыка	2	1
7	abcdef	badcef	3	2

3.5 Вывод

На основе схем из конструкторского раздела были написаны реализации алгоритмов, а также было выполнено их тестирование.

4. Исследовательский раздел

В данном разделе будет проведен сравнительный анализ алгоритмов по времени и затрачиваемой памяти.

4.1 Технические характеристики

Тестирование выполнялось на устройстве со следующими характеристиками:

- Операционная система Windows 10 [4]
- Память 8 Гб
- Процессор Intel Core i3 7020U, 2.3 ГГц [5]

Во время проведения эксперимента устройство не было нагружено сторонними задачами, а также было подключено к блоку питания.

Замеры процессорного времени проводились с помощью ассемблерной вставки, вычисляющей затраченное процессорное время в тиках.

Листинг 4.1 – Ассемблерная вставка замера процессорного времени в тиках

```
unsigned long long tick(void)
{
    unsigned long long d;
    __asm__ __volatile__ ("rdtsc": "=A" (d));
    return d;
}
```

4.2 Оценка времени работы алгоритмов

В таблице 4.1 представлены замеры процессорного времени работы алгоритмов на словах длиной до 10 символов. Каждое значение было получено усреднением по 100 замерам.

Таблица 4.1 – Результаты замеров времени

Длина	Л.с 2 стр.	Рек.Л.без кэша	Рек.Л.с кэшем	Рек.Д.-Лев.без кэша
1	706	103	397	108
2	798	626	1186	653
3	1452	12172	2406	4179
4	1455	18138	2829	17291
5	2676	83102	4389	79969
6	3624	431378	5371	431347
7	5657	2262049	7843	2286692
8	9671	14404315	17058	14350247
9	12395	25646236	22531	25542013

На рисунке 4.1 представлен график сравнения времени работы рекурсивных алгоритмов поиска расстояния Левенштейна с кэшем в форме матрицы и без кэша. В результате эксперимента было получено, что на строках равной длины до 10 символов алгоритм с использованием кэша работает быстрее, чем алгоритм без кэша в 10^4 раз. В результате можно сделать вывод, что для описанных данных предпочтительно использовать алгоритмы с кэшем.

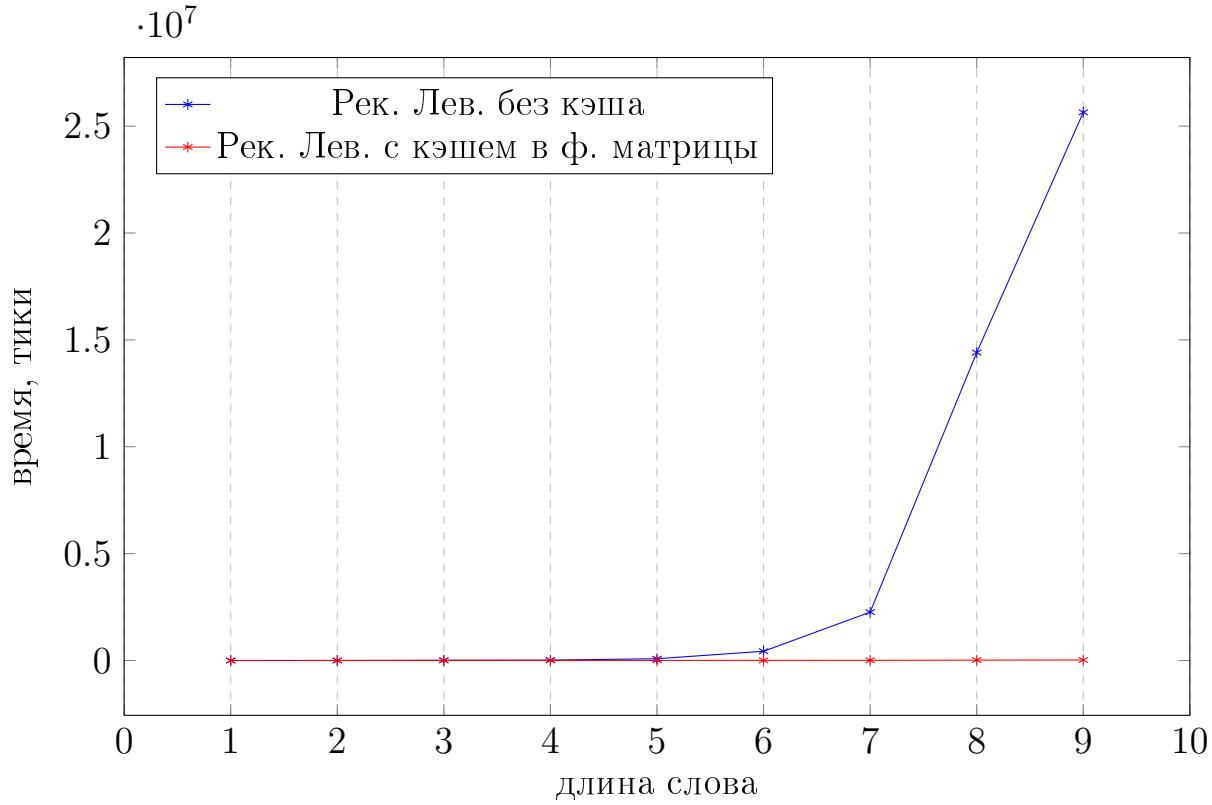

Рисунок 4.1 – Сравнение рекурсивных алгоритмов поиска расстояния Левенштейна с кэшем и без кэша

На рисунке 4.2 представлен график сравнения времени работы рекурсивных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна без использования кэша. Как можно заметить, они практически идентично накладываются друг на друга.

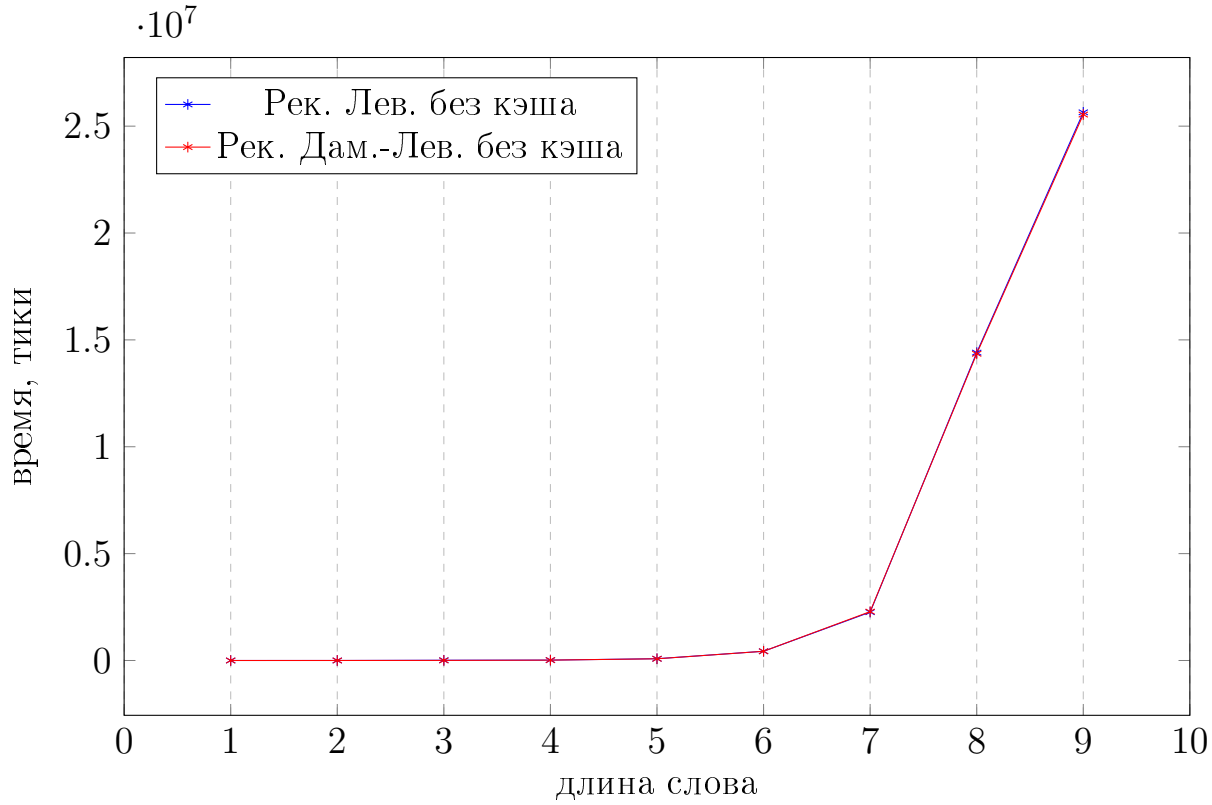

Рисунок 4.2 – Сравнение рекурсивных алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна без кэша

4.3 Вывод

Рекурсивные реализации поиска редакционного расстояния по времени работы оказываются на порядок дольше. На словах длиной до 10 символов виден выигрыш нерекурсивной реализации с кэшем в форме двух строк.

Также нетрудно заметить, что при добавлении кэша в рекурсивный алгоритм Левенштейна, время его выполнения значительно уменьшается за счет отсутствия повторных вычислений.

Касаемо алгоритма Дамерау-Левенштейна можно сказать что по скорости он сравним с рекурсивным Левенштейна без кэша.

Заключение

В ходе выполнения лабораторной работы были решены следующие задачи:

- Изучены реализации алгоритмов Левенштейна и Дамерау-Левенштейна;
- Составлены схемы алгоритмов;
- Применены методы динамического программирования при реализации алгоритмов Левенштейна и Дамерау-Левенштейна;
- Проведен сравнительный анализ алгоритмов по затраченным ресурсам (времени и памяти);
- Описаны и обоснованы полученные результаты.

В результате исследований можно прийти к выводу, что время выполнения рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна быстро растет при увеличении длины подаваемых на вход строк. Это связано с большим количеством повторных вычислений. Добавление кэша в рекурсивный алгоритм позволяет избавиться от них благодаря сохранению вычисленных ранее значений. Однако нерекурсивный алгоритм работает в 2 раза быстрее, но количество используемой им памяти увеличивается соразмерно произведению длин строк, что может оказаться крайне неэффективно на больших строках длиной больше 1000.

Список литературы

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

[2] Толковый словарь по информатике. – М.: Финансы и статистика, 1991. с. 543.

[3] Черненький В. М. Гапанюк Ю. Е. Методика идентификации пассажира по установочным данным. – М.: Вестник МГТУ им. Н.Э. Баумана. Сер. “Приборостроение”, 2012. Т. 163. С. 30–34.

[4] Windows 10 [Электронный ресурс]. Режим доступа: `https://www.microsoft.com/ru-ru/windows/windows-10-specifications` (дата обращения: 18.10.2021).

[5] Процессор Intel Core i3-7020u [Электронный ресурс]. Режим доступа: `https://www.intel.ru/content/www/ru/ru/products/sku/122590/intel-core-i37020u-processor-3m-cache-2-30-ghz/specifications.html` (дата обращения: 18.10.2021).