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

Студент группы **ИУ7-51Б** ____________________ **Н.Ю. Баринов**

(Подпись, дата) (И.О. Фамилия)

Преподаватель ____________________ **Л.Л. Волкова**

(Подпись, дата) (И.О. Фамилия)

***2022** г.*

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ

Для того, чтобы сравнить 2 числа, можно найти из разность, чтобы сравнить пары чисел на координатной плоскости - найти гипотенузу.

Как сравнить строки? Для этого существуют редкционные расстояния.

Расстояние Левенштейна - минимальное количество операций, необходимых для преобразования одной строки в другую.

Редакционные операции:

1) вставка 1 символа (insert);
2) удаление 1 символа (delete);
3) замена 1 символа (replace).

Существует несколько алгортимов, реализующих расчет расстояния Левенштейна.

Дамерау, заметив, что одной из самых частых ошибок при печати является перестановка дву соседних букв, предложил включить в число редакционных операций операцию перестановки двух соседних символов.

1 Аналитическая часть

1.1 Цель и задачи

Изучение метода динамического программирования на материале расстояний Левенштейна и Дамерау-Левенштейна.

Для поставленной цели требуется решить следующие задачи:

1) изученить расстояния Левенштейна и Дамерау-Левенштейна;
2) разработать алгортимы поиска расстояний Левенштейна и Дамерау-Левенштейна;
3) реализовать алгортимы поиска расстояний Левенштейна и Дамерау-Левенштейна;
4) выполнить оценку реализованных алгоритмов по памяти;
5) выполнить замеры процессорного времени работы реализованных алгортимов;
6) выполнить сравнительный анализ нерекурсивных алгортимов поиска расстояний Левенштейна и Дамерау-Левенштейна;
7) выполнить сравнительный анализ алгортимов поиска расстояния Дамерау-Левенштейна.

1.2 Обзор существующих алгоритмов

1.2.1 Расстояние Левенштейна

Пусть строки s1 = s1[1…l1], s2 = s2[1…l2]

l1, l2 - длины строк

s1[1…i] - подстрока s1 длиной i

s2[1…j] - подстрока s2 длиной j

Тогда расстояние Левенштейна

$$D(\text{s1[1…i], s2[1…j]}) = \begin{cases} 0, i = 0, j = 0 \\ j, i = 0, j > 0 \\ i, i > 0, j = 0 \\ min\Big(D(\text{s1[1…i], s2[1…j-1]}) + 1, \\ \qquad D(\text{s1[1…i-1], s2[1…j]}) + 1, \\ \qquad D(\text{s1[1…i-1], s2[1…j-1]}) + \begin{cases} 0, s1[i] = s2[i] \\ 1, \text{иначе} \end{cases} \Big), \\ \text{иначе.} \end{cases} \tag{1}$$

1.2.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна

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

1.2.3 Общие подходы

Существуют несколько подходов к реализации алгоритма поиска расстояний Левенштейна и Дамерау-Левенштейна.

Иттерационный подход:

Начиная с левого верхнего угла матрицы, описывающей расстояния между подстроками исхоных строк, считаются последующие значения до правого-нижнего угла, где и будет записано искомое растояние.

Рекурсивный подход:

Ищется сразу искомое расстояние с использованием рекусивных вызовов функции от необходимых подстрок. Имеются минусы: придется несколько раз пересчитывать уже вычисленные растояния.

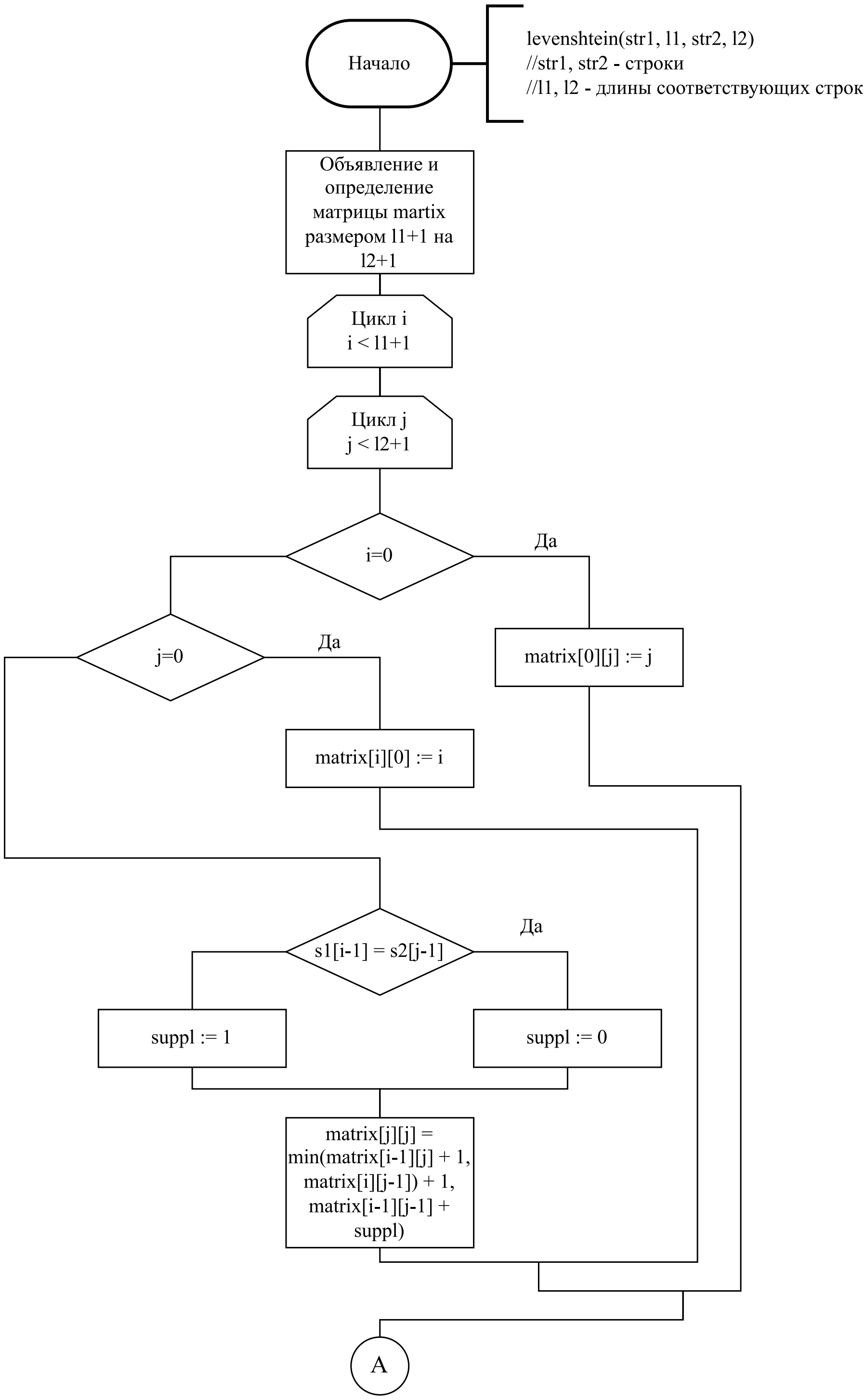
Рекурсивный подход с кешем:

Подход такой же, как и рекусивный, но также существует матрица, в которой хранятся уже вычисленные расстояния. При каждом вызове проверяется не вычисленно ли уже искомое расстояние.

2 Конструкторская часть

2.1 Алгоритмы поиска редакционных расстояний

2.1.1 Иттерационный алгоритм поиска расстояния Левенштейна



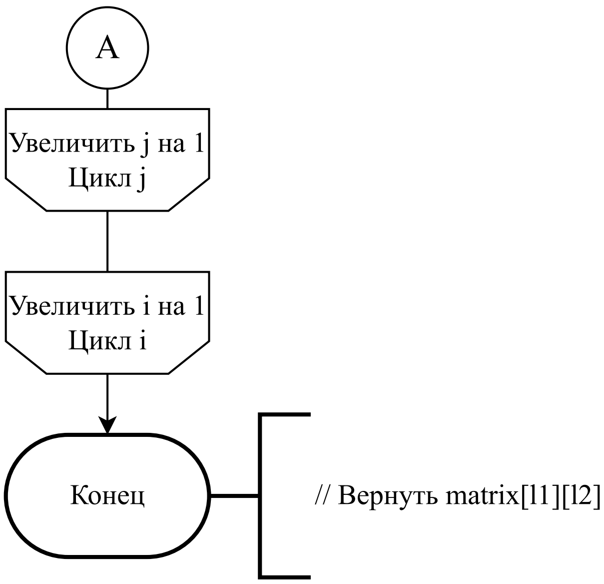


Рисунок 1 – Иттерационный алгоритм поиска расстояния Левенштейна

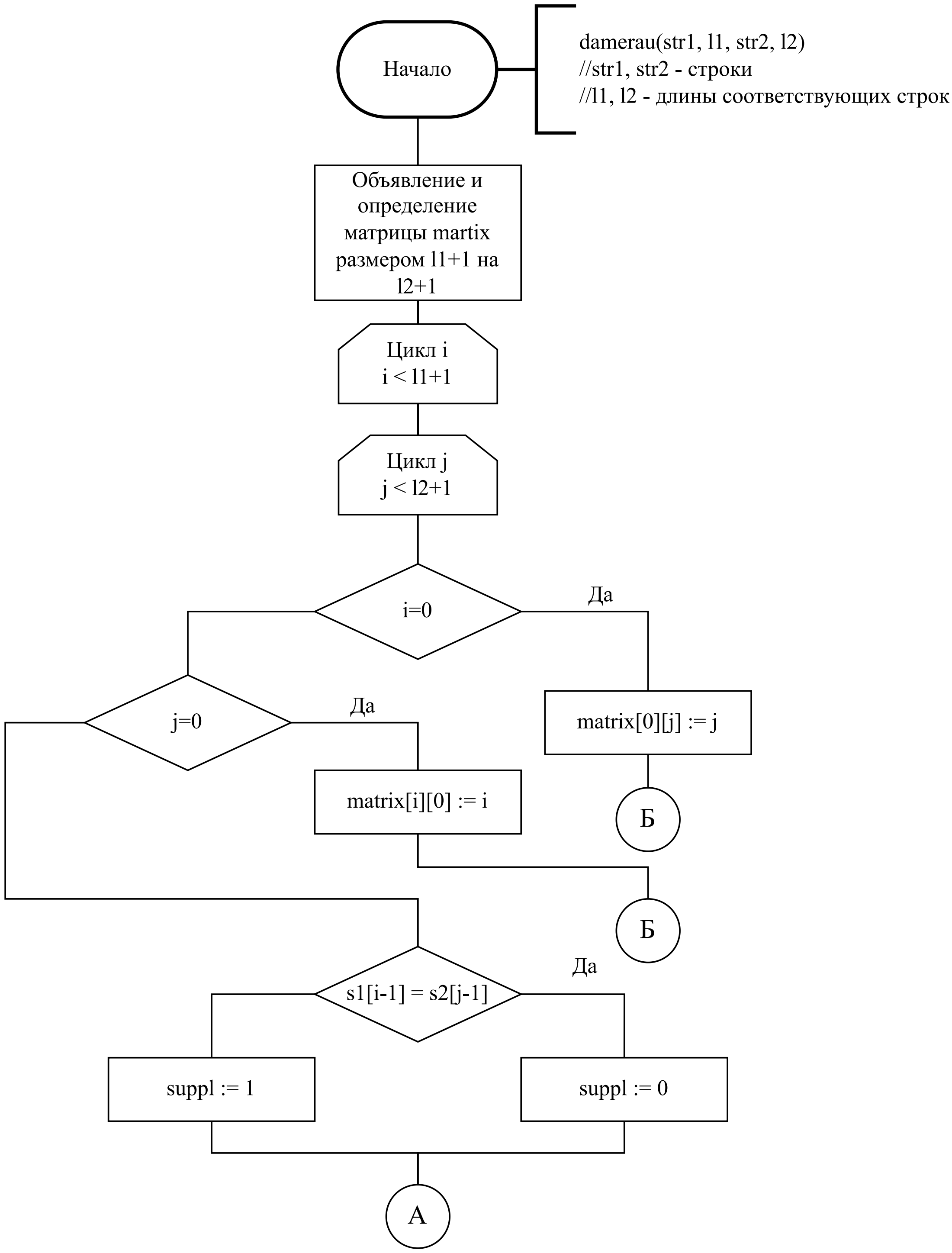
Оценка памяти алгоритма:

Пусть n - длина строки s1, m - длина строки s2

Тогда затраты по памяти будут:

- для матрицы - ((n+1)*(m+1)*sizeof(int))
- для s1, s2 - ((n+m)*sizeof(char))
- для n, m - (2*sizeof(int))
- для переменных индексации - (2*sizeof(int))
- для предыдущих символов строк- (2*sizeof(char))
- адрес возврата - (sizeof(int*))

2.1.2 Иттерационный алгоритм поиска расстояния Дамерау-Левенштейна



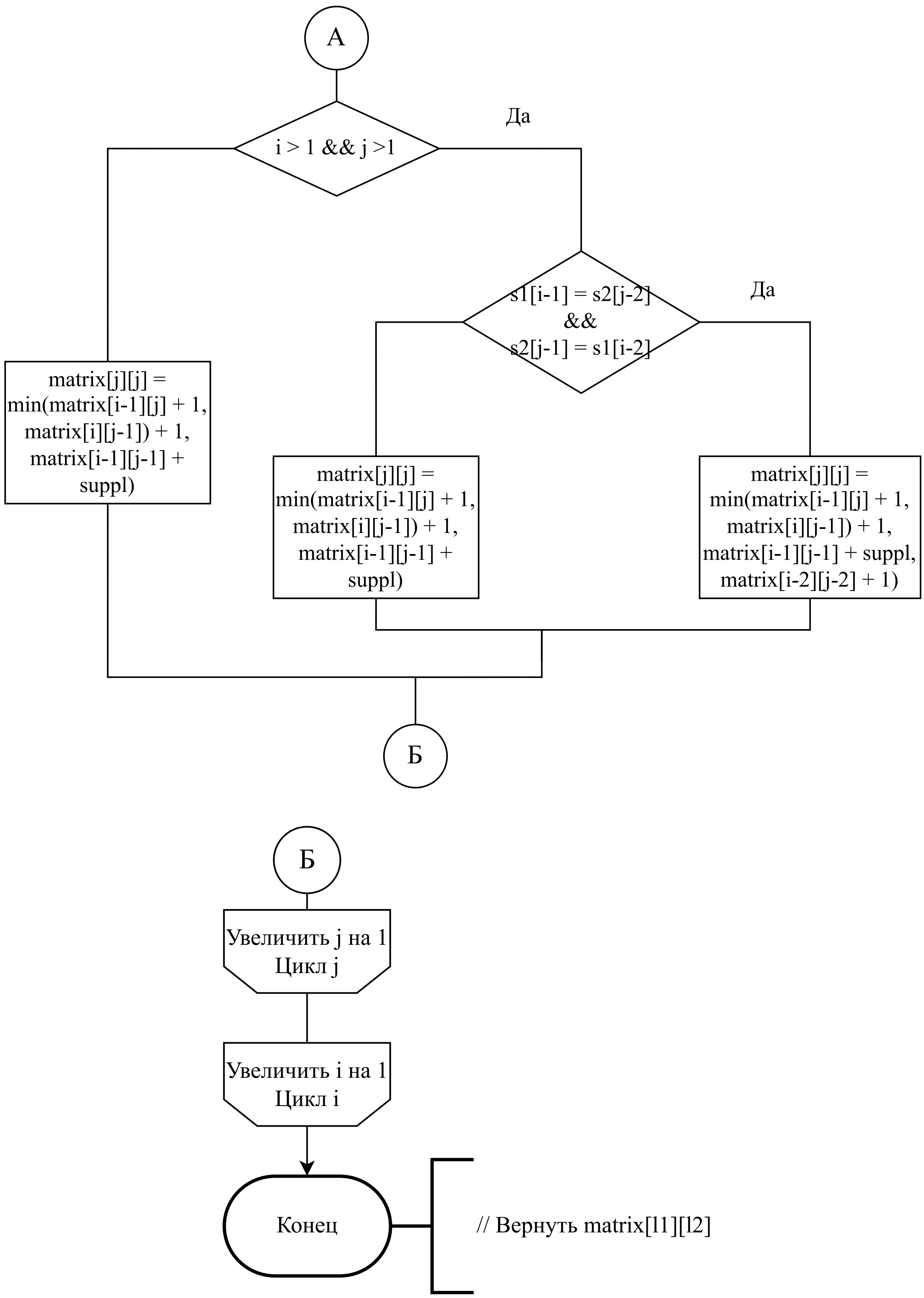


Рисунок 2 – Иттерационный алгоритм поиска расстояния Дамерау-Левенштейна

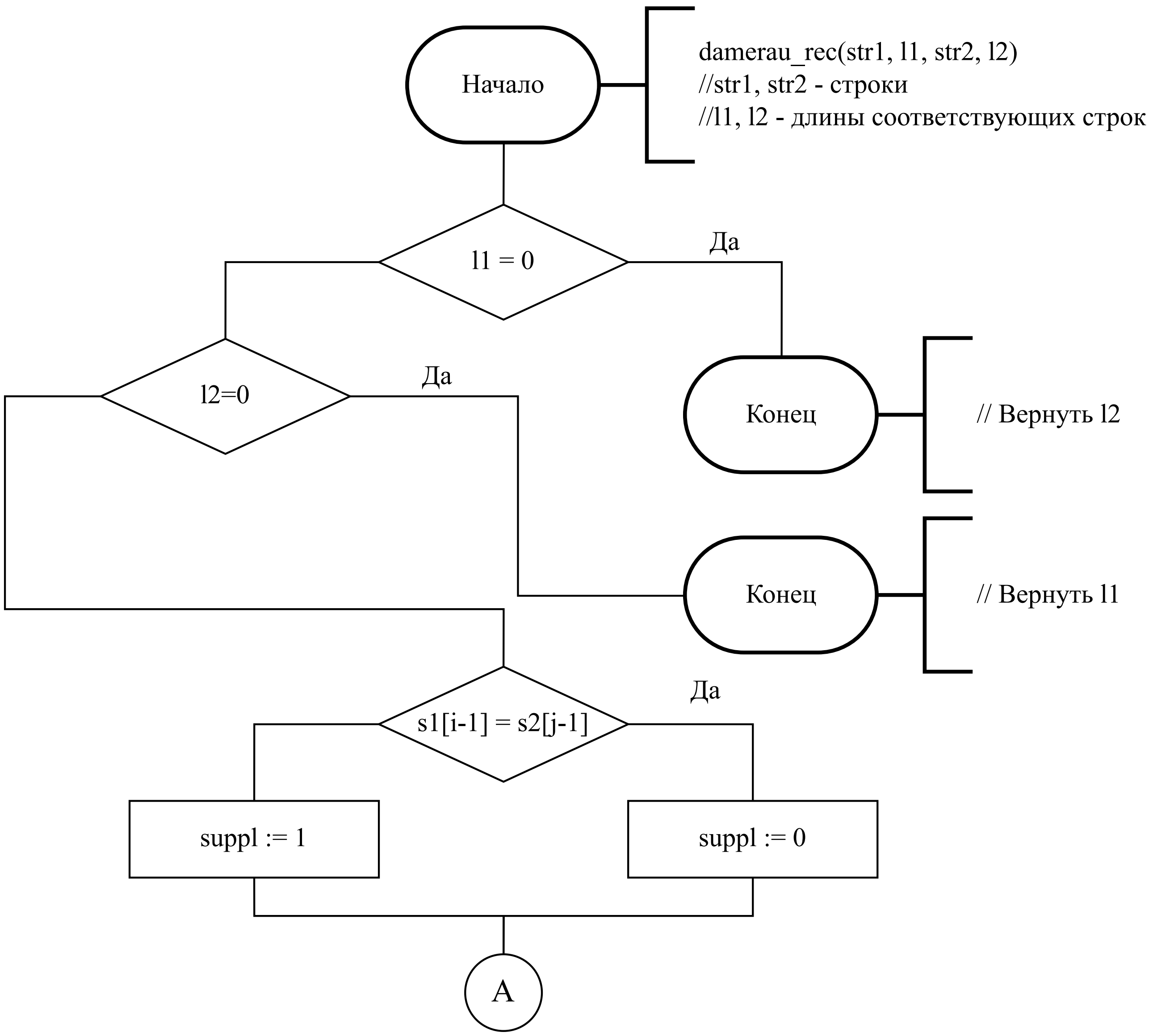
Оценка памяти алгоритма:

Пусть n - длина строки s1, m - длина строки s2

Тогда затраты по памяти будут:

- для матрицы - ((n+1)*(m+1)*sizeof(int))
- для s1, s2 - ((n+m)*sizeof(char))
- для n, m - (2*sizeof(int))
- для переменных индексации - (2*sizeof(int))
- для предыдущих символов строк- (4*sizeof(char))
- адрес возврата - (sizeof(int*))

2.1.3 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна



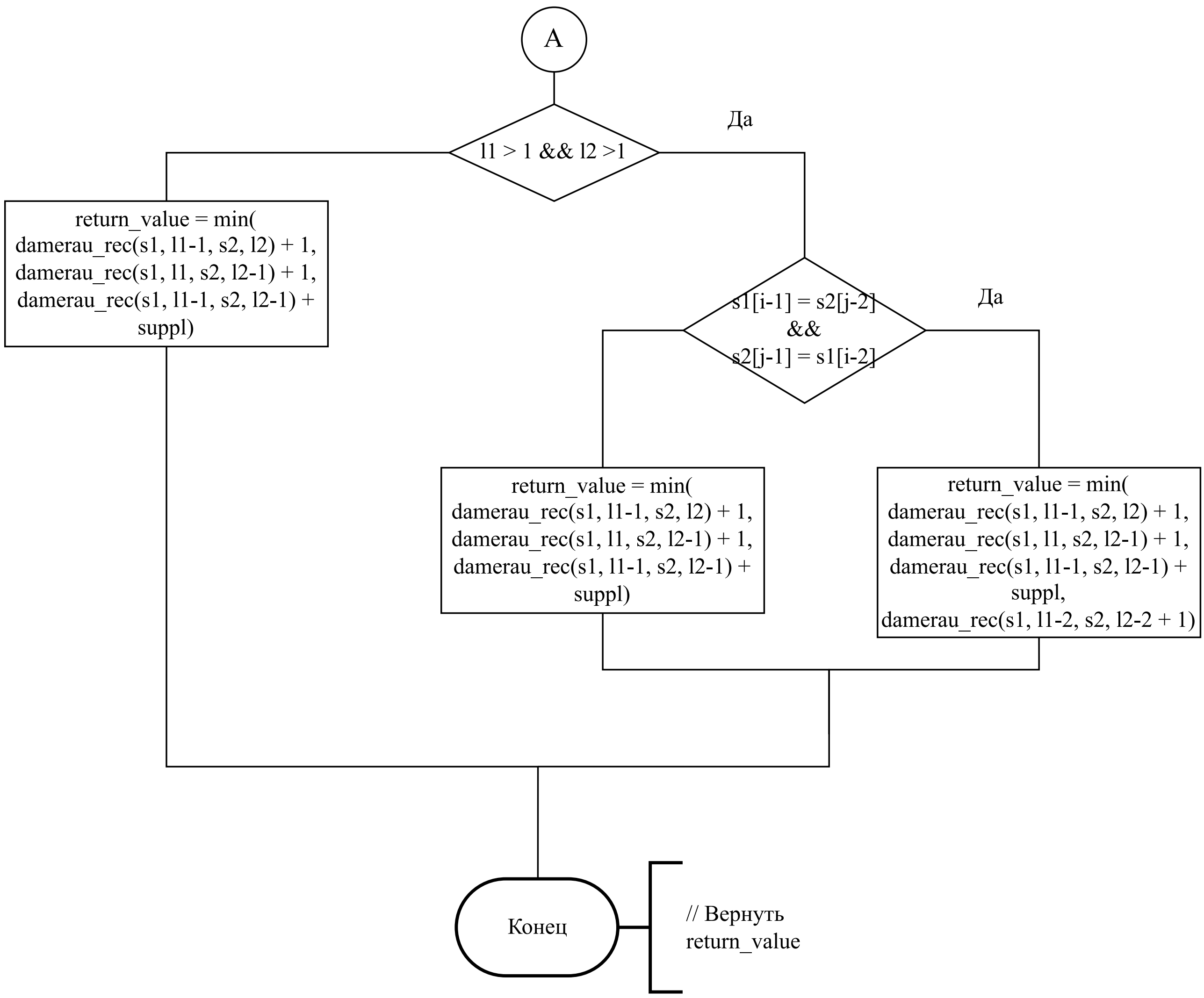


Рисунок 3 – Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

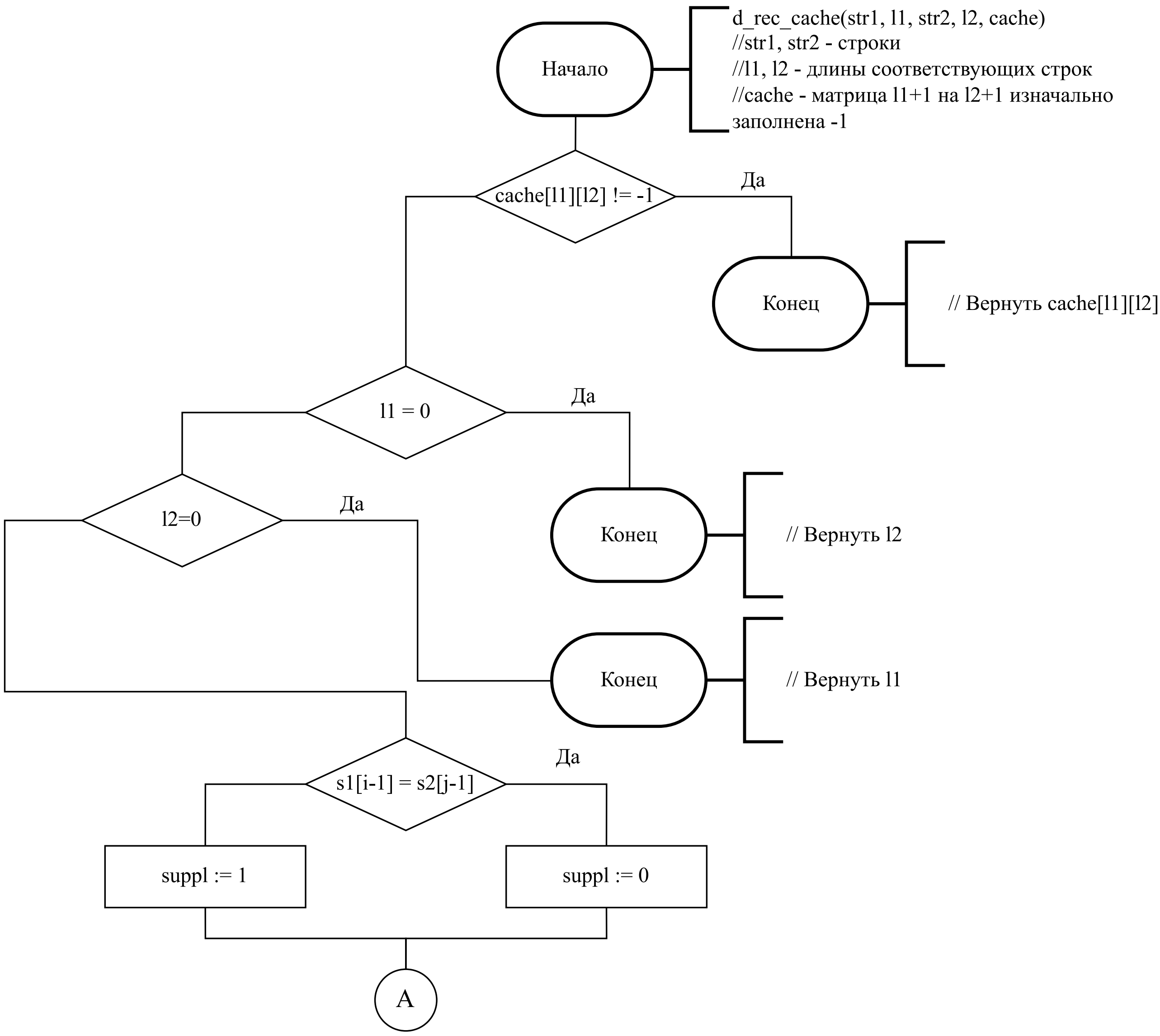
Оценка памяти алгоритма:

Пусть n - длина строки s1, m - длина строки s2

Тогда затраты по памяти **для каждого вызова** будут:

- для s1, s2 - ((n+m)*sizeof(char))
- для n, m - (2*sizeof(int))
- для предыдущих символов строк- (4*sizeof(char))
- адрес возврата - (sizeof(int*))

2.1.4 Рекурсивный с кэшем алгоритм поиска расстояния Дамерау-Левенштейна



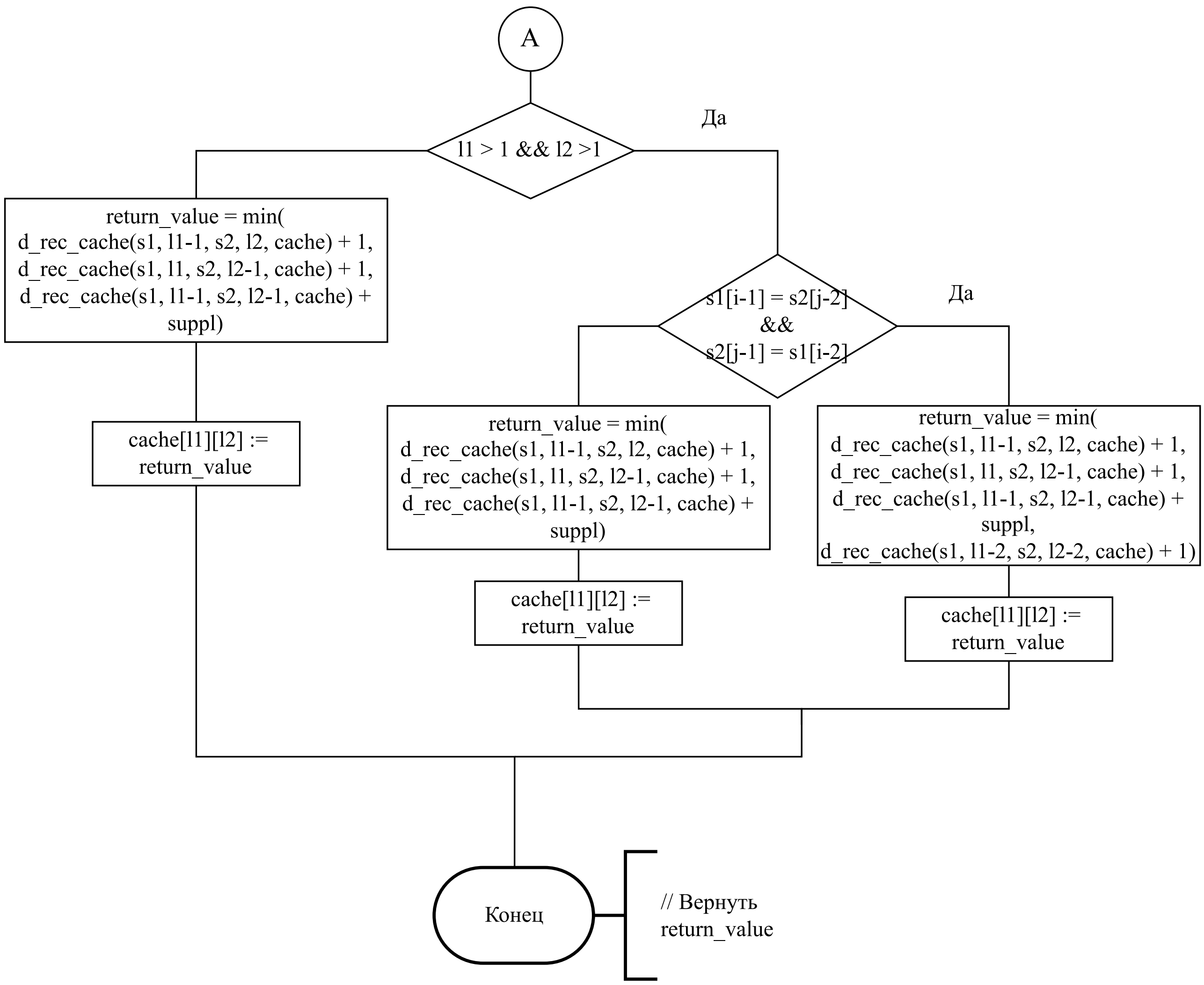


Рисунок 4 – Рекурсивный с кэшем алгоритм поиска расстояния Дамерау-Левенштейна

Оценка памяти алгоритма:

Пусть n - длина строки s1, m - длина строки s2

Тогда затраты по памяти помимо матрицы((n+1)*(m+1)*sizeof(int)) **для каждого вызова** будут:

- для s1, s2 - ((n+m)*sizeof(char))
- для n, m - (2*sizeof(int))
- для предыдущих символов строк- (4*sizeof(char))
- адрес возврата - (sizeof(int*))
- ссылка на матрицу - (sizeof(int**))

3 Технологическая часть

3.1 Выбор языка программирования, средств разработки

Для реализации алгоритмов был выбран язык C(c99). В данной лабораторной работе необходимо замерить процессорное время, такую возможность дает стандартная библиотека **time.h**[1]. Компилятор - **GCC**[2].

3.2 Реализация алгоритмов

В листингах 1-4 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 1: Иттерационный алгоритм поиска расстояния Левенштейна

```
int levenshtein(char *s1, int l1, char *s2, int l2)
{
    int matrix[l1 + 1][l2 + 1];
    for (int i = 0; i < l1 + 1; i++)
    {
        for (int j = 0; j < l2 + 1; j++)
        {
            if (i == 0)
            {
                matrix[0][j] = j;
                continue;
            }
            if (j == 0)
            {
                matrix[i][0] = i;
                continue;
            }
            char ch1 = s1[i - 1];
            char ch2 = s2[j - 1];
            matrix[i][j] = min3(matrix[i - 1][j] + 1,
                                matrix[i][j - 1] + 1,
                                matrix[i - 1][j - 1]
                                + (ch1 == ch2 ? 0 : 1));
        }
    }
    return matrix[l1][l2];
}
```

Листинг 2: Иттерационный алгоритм поиска расстояния Дамерау-Левенштейна

```
int damerau(char *s1, int l1, char *s2, int l2)
{
    int matrix[l1 + 1][l2 + 1];
    for (int i = 0; i < l1 + 1; i++)
    {
        for (int j = 0; j < l2 + 1; j++)
        {
            if (i == 0)
            {
                matrix[0][j] = j;
                continue;
            }
            if (j == 0)
            {
                matrix[i][0] = i;
                continue;
            }
            char ch1 = s1[i - 1];
            char ch2 = s2[j - 1];
            if (i > 1 && j > 1)
            {
                char pch1 = s1[i - 2];
                char pch2 = s2[j - 2];
                if (ch1 == pch2 && ch2 == pch1)
                {
                    matrix[i][j] = min4(matrix[i - 1][j] + 1,
                                        matrix[i][j - 1] + 1,
                                        matrix[i - 1][j - 1]
                                        + (ch1 == ch2 ? 0 : 1),
                                        matrix[i - 2][j - 2]
                                        + 1);
                }
                else
```

```
                {
                    matrix[i][j] = min3(matrix[i - 1][j] + 1,
                                        matrix[i][j - 1] + 1,
                                        matrix[i - 1][j - 1]
                                        + (ch1 == ch2 ?0:1));
                }
            }
            else
            {
                matrix[i][j] = min3(matrix[i - 1][j] + 1,
                                    matrix[i][j - 1] + 1,
                                    matrix[i - 1][j - 1]
                                    + (ch1 == ch2 ? 0 : 1));
            }
        }
    }
    return matrix[l1][l2];
}
```

Листинг 3: Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

```
int damereu_rec(char *s1, int l1, char *s2, int l2)
{
    if (l1 == 0)
        return l2;
    if (l2 == 0)
        return l1;
    char ch1 = s1[l1 - 1];
    char ch2 = s2[l2 - 1];
    if (l1 > 1 && l2 > 1)
    {
        char pch1 = s1[l1 - 2];
        char pch2 = s2[l2 - 2];
```

```
        if (ch1 == pch2 && ch2 == pch1)
        {
            return min4(damereu_rec(s1, l1 - 1, s2, l2) + 1,
                        damereu_rec(s1, l1, s2, l2 - 1) + 1,
                        damereu_rec(s1, l1 - 1, s2, l2 - 1)
                        + (ch1 == ch2 ? 0 : 1),
                        damereu_rec(s1, l1 - 2, s2, l2 - 2)
                        + 1);
        }
        else
        {
            return min3(damereu_rec(s1, l1 - 1, s2, l2) + 1,
                        damereu_rec(s1, l1, s2, l2 - 1) + 1,
                        damereu_rec(s1, l1 - 1, s2, l2 - 1)
                        + (ch1 == ch2 ? 0 : 1));
        }
    }
    else
    {
        return min3(damereu_rec(s1, l1 - 1, s2, l2) + 1,
                    damereu_rec(s1, l1, s2, l2 - 1) + 1,
                    damereu_rec(s1, l1 - 1, s2, l2 - 1)
                    + (ch1 == ch2 ? 0 : 1));
    }
}
```

Листинг 4: Рекурсивный с кэшем алгоритм поиска расстояния Дамерау-Левенштейна

```
int damereu_rec_cache(char *s1, int l1, char *s2, int l2)
{
    int **cache = malloc((l1 + 1) * sizeof(int*));
    for (int i = 0; i < l1 + 1; i++)
```

```
    {
        cache[i] = malloc((l2 + 1) * sizeof(int));
    }
    for (int i = 0; i < l1 + 1; i++)
    {
        for (int j = 0; j < l2 + 1; j++)
        {
            cache[i][j] = -1;
        }
    }
    return do_rec(s1, l1, s2, l2, cache, l1 + 1, l2 + 1);
}

int do_rec(char *s1, int l1, char *s2, int l2, int **cache,
            int m, int n)
{
    if (cache[l1][l2] != -1)
        return cache[l1][l2];
    if (l1 == 0)
        return l2;
    if (l2 == 0)
        return l1;
    char ch1 = s1[l1 - 1];
    char ch2 = s2[l2 - 1];
    if (l1 > 1 && l2 > 1)
    {
        char pch1 = s1[l1 - 2];
        char pch2 = s2[l2 - 2];
        if (ch1 == pch2 && ch2 == pch1)
        {
            int dist = min4(do_rec(s1, l1 - 1, s2, l2,
                                    cache, m, n) + 1,
                            do_rec(s1, l1, s2, l2 - 1,
                                    cache, m, n) + 1,
                            do_rec(s1, l1 - 1, s2, l2 - 1,
```

```
                                        cache, m, n)
                                + (ch1 == ch2 ? 0 : 1),
                                do_rec(s1, l1 - 2, s2, l2 - 2,
                                        cache, m, n) + 1);
                cache[l1][l2] = dist;
                return dist;
            }
            else
            {
                int dist = min3(do_rec(s1, l1 - 1, s2, l2,
                                        cache, m, n) + 1,
                                do_rec(s1, l1, s2, l2 - 1,
                                        cache, m, n) + 1,
                                do_rec(s1, l1 - 1, s2, l2 - 1,
                                        cache, m, n)
                                + (ch1 == ch2 ? 0 : 1));
                cache[l1][l2] = dist;
                return dist;
            }
        }
        else
        {
            int dist = min3(do_rec(s1, l1 - 1, s2, l2,
                                    cache, m, n) + 1,
                            do_rec(s1, l1, s2, l2 - 1,
                                    cache, m, n) + 1,
                            do_rec(s1, l1 - 1, s2, l2 - 1,
                                    cache, m, n)
                            + (ch1 == ch2 ? 0 : 1));
            cache[l1][l2] = dist;
            return dist;
        }
    }
```

3.3 Тестирование

В таблице 1 приведены тесты для функций, реализующих алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна. Тесты для всех алгоритмов пройдены успешно.

Таблица 1 – Проведенные тесты

№	Строка 1	Строка 2	Результат Левенштейн	Результат Дамерау-Л.
1	""	""	0	0
2	""	mama	4	4
3	babushka	""	8	8
4	moloko	oloko	1	1
5	baton	at	3	3
6	remont	pat	5	5
7	rov	kit	3	3
8	tovar	to	3	3
9	stakan	satkan	2	1
10	aabb	abab	2	1

4 Экспериментальная часть

В данном разделе будут приведены замеры процессорного времени работы функций, а также проведен сравнительный анализ алгоритмов.

4.1 Технические характеритики

Операционная система - **Ubuntu 22.04 LTS**[3]. Процессор - **AMD Ryzen 5 3500U**. Оперативная память - 16 Гб.

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Замеры процессорного времени

Для измерения процессорного времени используется функция **clock()** из стандартной библиотеки **time.h**. Найти время в секундах позволяет конструкция:

Листинг 5: Пример использования функции clock()

```
int start = clock();
// some code
int end = clock();
cpu_time_used += ((double)(end - start)) / CLOCKS_PER_SEC;
```

Замеры проводились 100 раз для всех функций кроме Дамерау-Левенштейна с рекурсией, для него число замеров равно 5. Результаты замеров приведены в таблице 2 (время в секундах).

Таблица 2 – Замеры времени

Дл. строки	Л.	Д-Л.	Д-Л.(рек)	Д-Л (рек+кэш)
0	0,000001	0,000001	0,000001	0,000001
1	0,000001	0,000001	0,000001	0,000001
2	0,000001	0,000001	0,000001	0,000002
3	0,000001	0,000001	0,000002	0,000002
4	0,000002	0,000001	0,000006	0,000003
5	0,000002	0,000002	0,000039	0,000004
6	0,000003	0,000002	0,000201	0,000006
7	0,000003	0,000002	0,000994	0,000007
8	0,000004	0,000004	0,006436	0,000008
9	0,000004	0,000005	0,015010	0,000008
10	0,000004	0,000005	0,053566	0,000007
11	0,000004	0,000005	0,291436	0,000008
12	0,000005	0,000005	1,624380	0,000009
13	0,000005	0,000006	9,086192	0,000009

Также на рисунках 5-8 приведены графические результаты замеров.

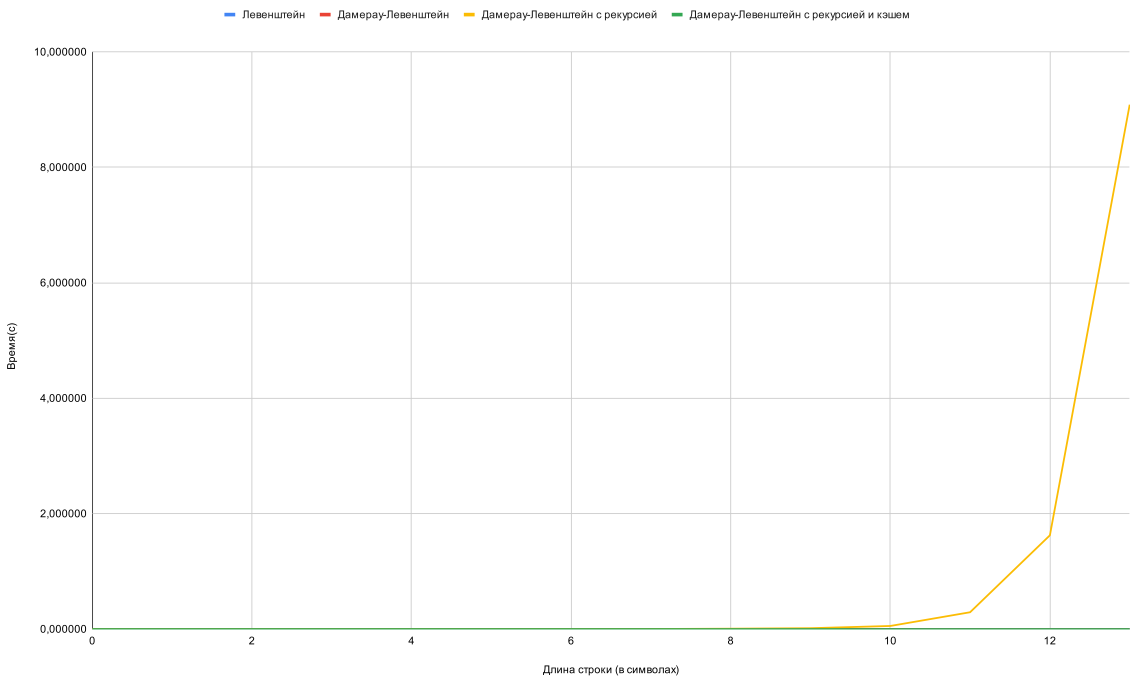


Рисунок 5 – Сравнение по времени всех алгортимов

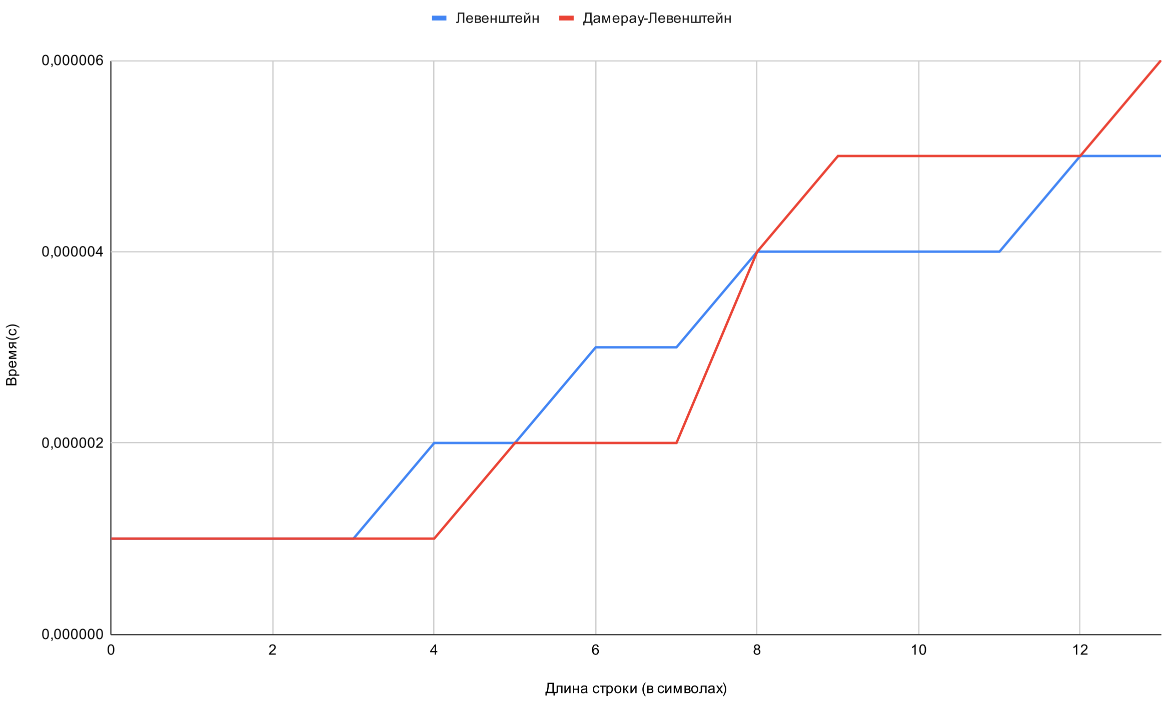


Рисунок 6 – Сравнение по времени нерекурсивных алгортимов поиска расстояний Левенштейна и Дамерау-Левенштейна

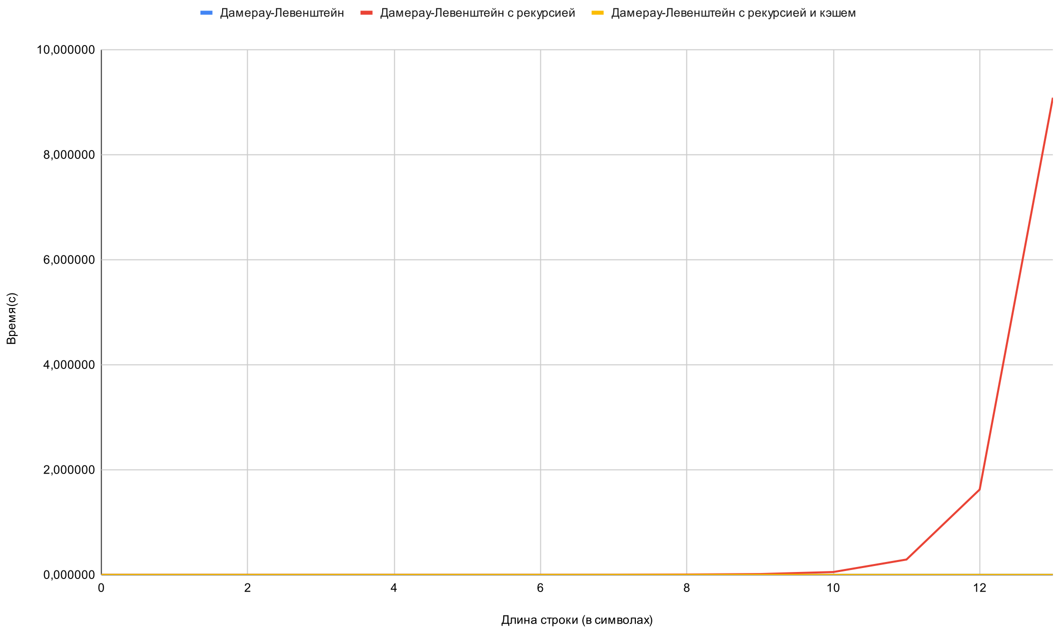


Рисунок 7 – Сравнение по времени алгортимов поиска расстояния Дамерау-Левенштейна

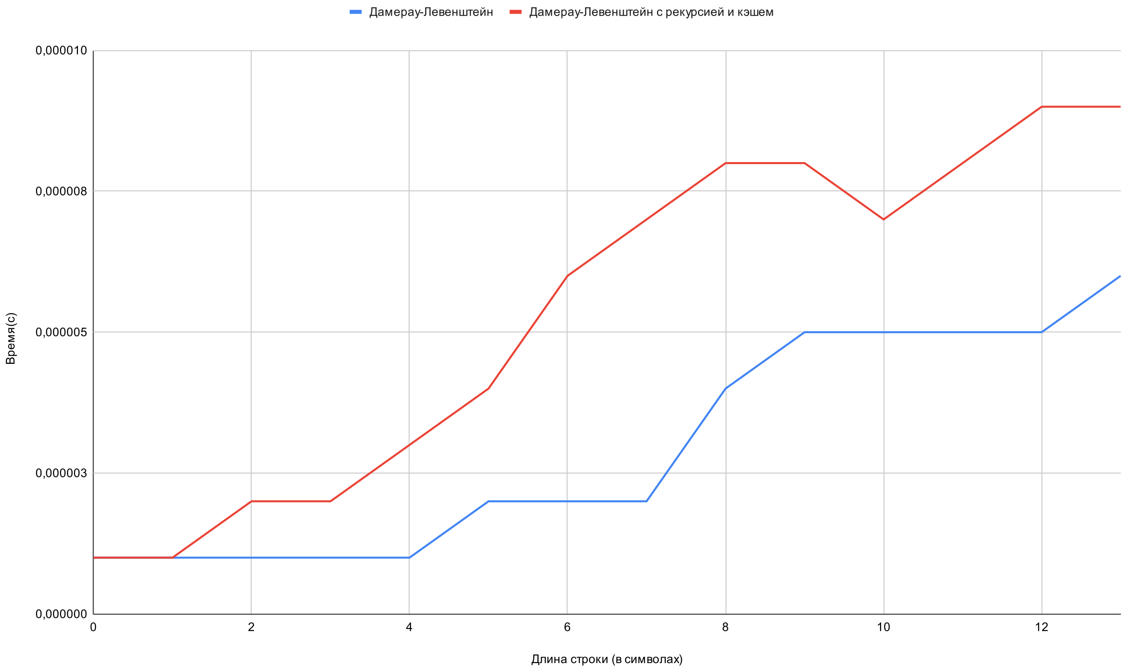


Рисунок 8 – Сравнение по времени нерекурсивного и рекусивного с кешем алгортимов поиска расстояния Дамерау-Левенштейна

4.3 Выводы

Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных - как сумма длин строк.

В результате эксперимента было получено, что при длине строк более 10 символов рекусивный алгортим поиска выполнятся намного дольше других. Его не следует использовать уже при длине строк в 3 символа, так как уже при этой длине он алгоритм медленнее остальных в 3 раза.

Также при проведении эксперимента было выявлено, что нерекурсивные алгортимы поиска расстояний Левенштейна и Дамерау-Левенштейна имеют примерно одинаковую скорость выполнения, поэтому выбирать между ними нужно исходя из задачи.

Рекурсивный алгортим поиска расстояния Дамерау-Левенштейна с кешем проигрывает нерекурсивному по времени, но выигрывает по памяти. Осуществлять выбор между этими алгоритмами нужно исходя из того, что важнее - память или время.

ЗАКЛЮЧЕНИЕ

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

1) изучены расстояния Левенштейна и Дамерау-Левенштейна;
2) разработаны алгортимы поиска расстояний Левенштейна и Дамерау-Левенштейна;
3) реализованы алгортимы поиска расстояний Левенштейна и Дамерау-Левенштейна;
4) выполнена оценку реализованных алгоритмов по памяти;
5) выполнены замеры процессорного времени работы реализованных алгортимов;
6) выполнен сравнительный анализ нерекурсивных алгортимов поиска расстояний Левенштейна и Дамерау-Левенштейна;
7) выполнен сравнительный анализ алгортимов поиска расстояния Дамерау-Левенштейна.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. ISO/IEC 9899:1999 [Электронный ресурс]. – Режим доступа: https://www.open-std.org/jtc1/sc22/WG14/www/docs/n1256.pdf, свободный – (11.10.2022)

2. GCC, the GNU Compiler Collection [Электронный ресурс]. – Режим доступа: https://gcc.gnu.org, свободный – (11.10.2022)

3. Ubuntu for desktops [Электронный ресурс]. – Режим доступа: https://ubuntu.com/desktop, свободный – (11.10.2022)