



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**

(Подпись, дата)

Н.Ю. Баринов

(И.О. Фамилия)

Преподаватель

(Подпись, дата)

Л.Л. Волкова

(И.О. Фамилия)

2022 г.

ОГЛАВЛЕНИЕ

ВВЕДЕНИЕ	3
1 Аналитическая часть	4
1.1 Цель и задачи	4
1.2 Обзор существующих алгоритмов	5
1.2.1 Расстояние Левенштейна	5
1.2.2 Расстояние Дамерау-Левенштейна	6
1.2.3 Общие подходы	6
2 Конструкторская часть	8
2.1 Алгоритмы поиска редакционных расстояний	8
2.1.1 Итерационный алгоритм поиска расстояния Левенштейна	8
2.1.2 Итерационный алгоритм поиска расстояния Дамерау-Левенштейна	10
2.1.3 Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна	12
2.1.4 Рекурсивный с кэшем алгоритм поиска расстояния Дамерау-Левенштейна	14
3 Технологическая часть	16
3.1 Выбор языка программирования, средств разработки	16
3.2 Реализация алгоритмов	16
3.3 Тестирование	23
4 Экспериментальная часть	24
4.1 Технические характеристики	24
4.2 Замеры процессорного времени	24
4.3 Выводы	24
ЗАКЛЮЧЕНИЕ	25
СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ	26

ВВЕДЕНИЕ

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

Как сравнить строки? Для этого существуют редакционные 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 алгоритмов, реализующих расчет расстояния Левенштейна.

Дамерау, заметив, что одной из самых частых ошибок при печати является перестановка двух 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 Дamerau-Левенштейна.

Для поставленной цели требуется решить следующие задачи:

- 1) изучить расстояния Левенштейна и Дamerau-Левенштейна;
- 2) разработать алгоритмы поиска расстояний Левенштейна и Дamerau-Левенштейна;
- 3) реализовать алгоритмы поиска расстояний Левенштейна и Дamerau-Левенштейна;
- 4) выполнить оценку реализованных алгоритмов по памяти;
- 5) выполнить замеры процессорного времени работы реализованных алгоритмов;
- 6) выполнить сравнительный анализ нерекурсивных алгоритмов поиска расстояний Левенштейна и Дamerau-Левенштейна;
- 7) выполнить сравнительный анализ алгоритмов поиска расстояния Дamerau-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

$$D(s1[1...i], s2[1...j]) = \begin{cases} 0, i = 0, j = 0 \\ j, i = 0, j > 0 \\ i, i > 0, j = 0 \\ \min \left(D(s1[1...i], s2[1...j-1]) + 1, \right. \\ \quad D(s1[1...i-1], s2[1...j]) + 1, \\ \quad \left. D(s1[1...i-1], s2[1...j-1]) + \begin{cases} 0, s1[i] = s2[i] \\ 1, \text{ иначе} \end{cases} \right) \\ \text{иначе.} \end{cases} \quad (1)$$

1.2.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна

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

1.2.3 Общие подходы

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

Итерационный подход:

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

Рекурсивный подход:

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

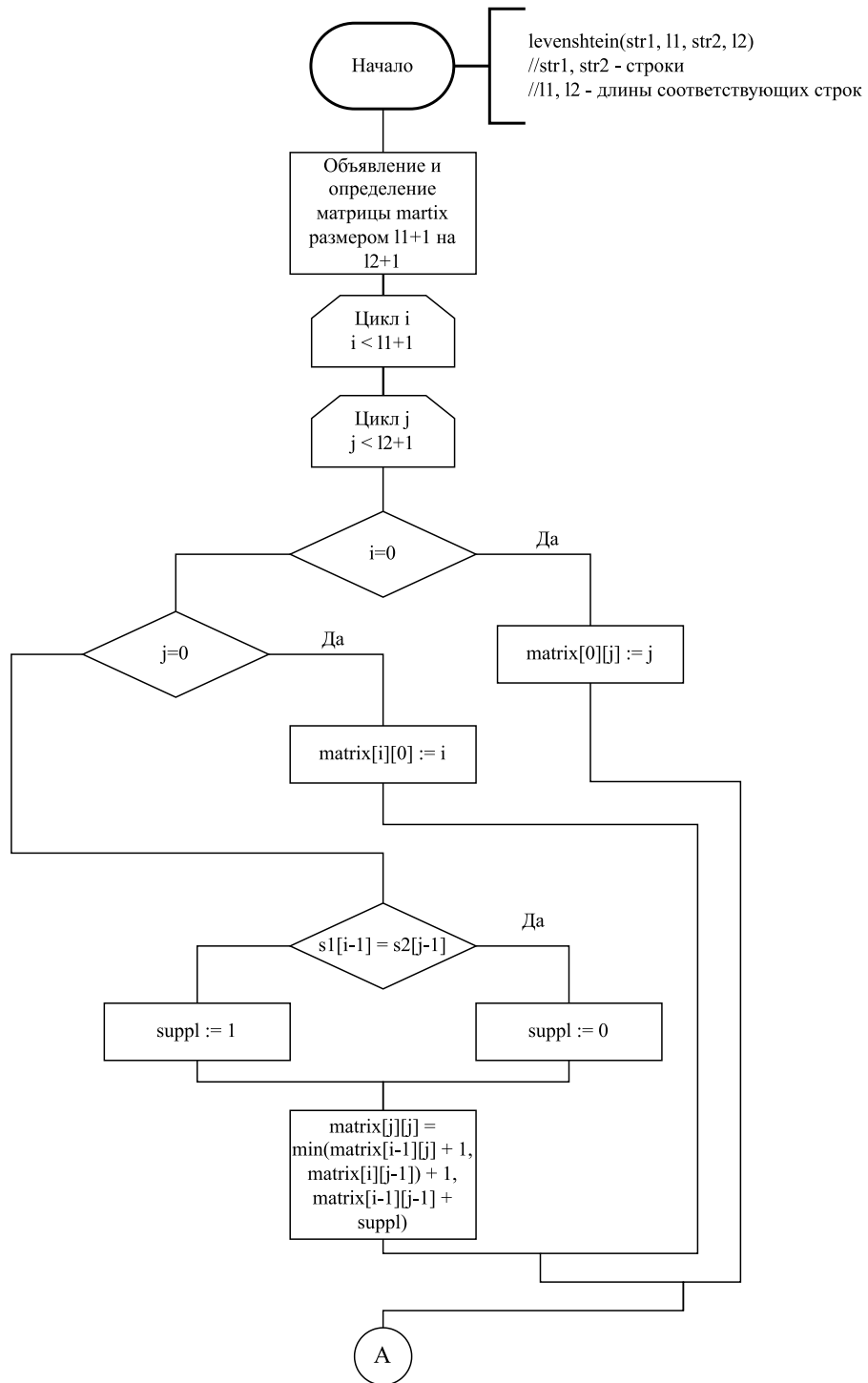
Рекурсивный подход с кешем:

Подход такой же, как и рекурсивный, 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 Итерационный алгоритм поиска расстояния Левенштейна



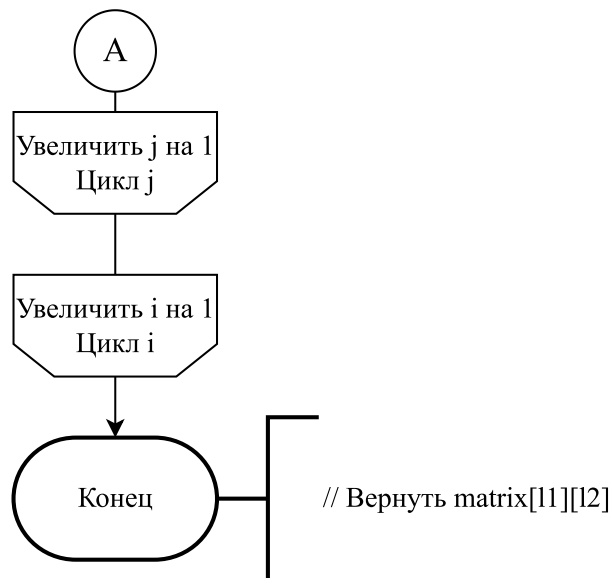


Рисунок 1 – Итерационный алгоритм поиска расстояния Левенштейна

Оценка памяти алгоритма:

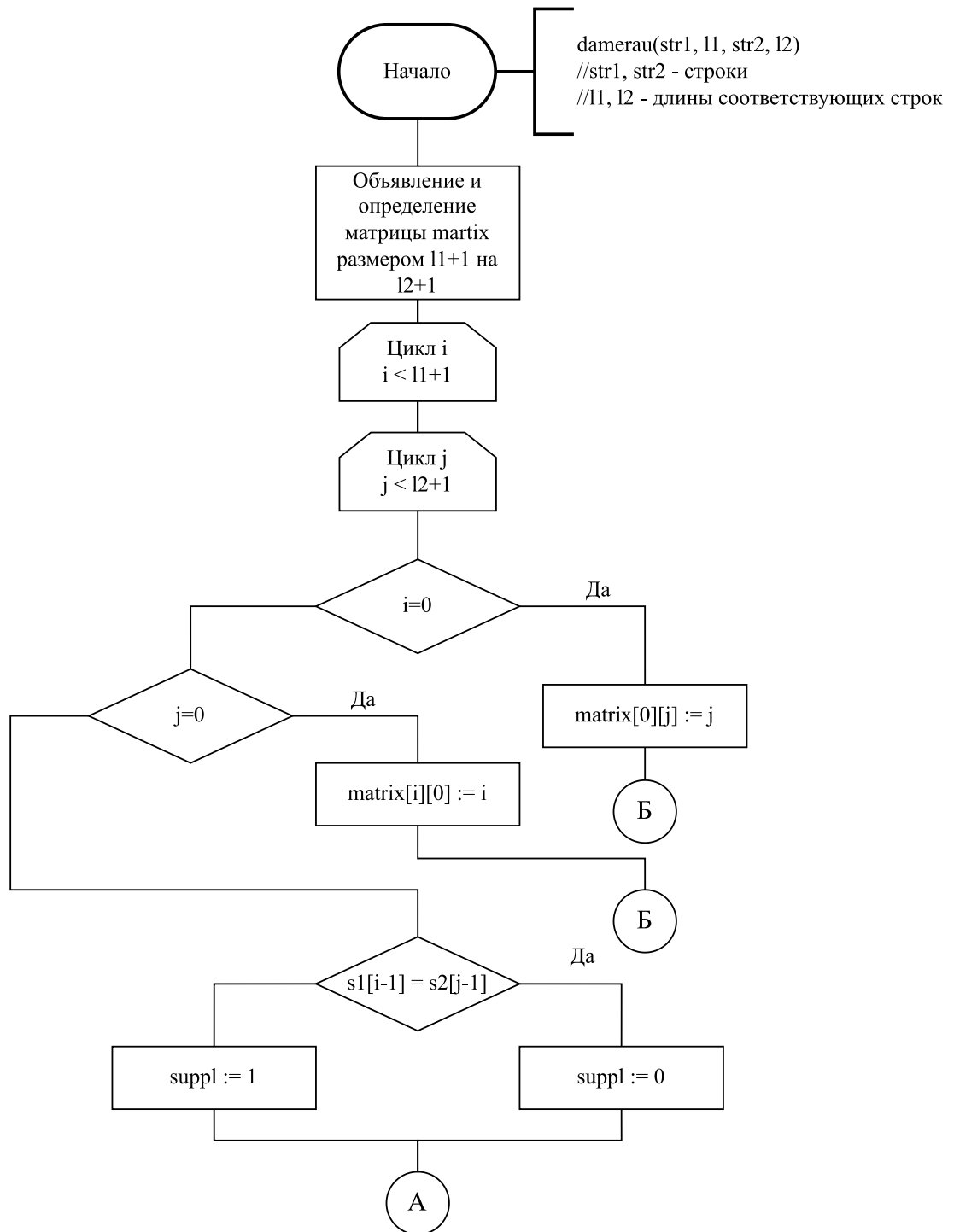
Пусть n - длина строки $s1$, m - длина строки $s2$

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 Итерационный алгоритм поиска расстояния

Дамерау-Левенштейна



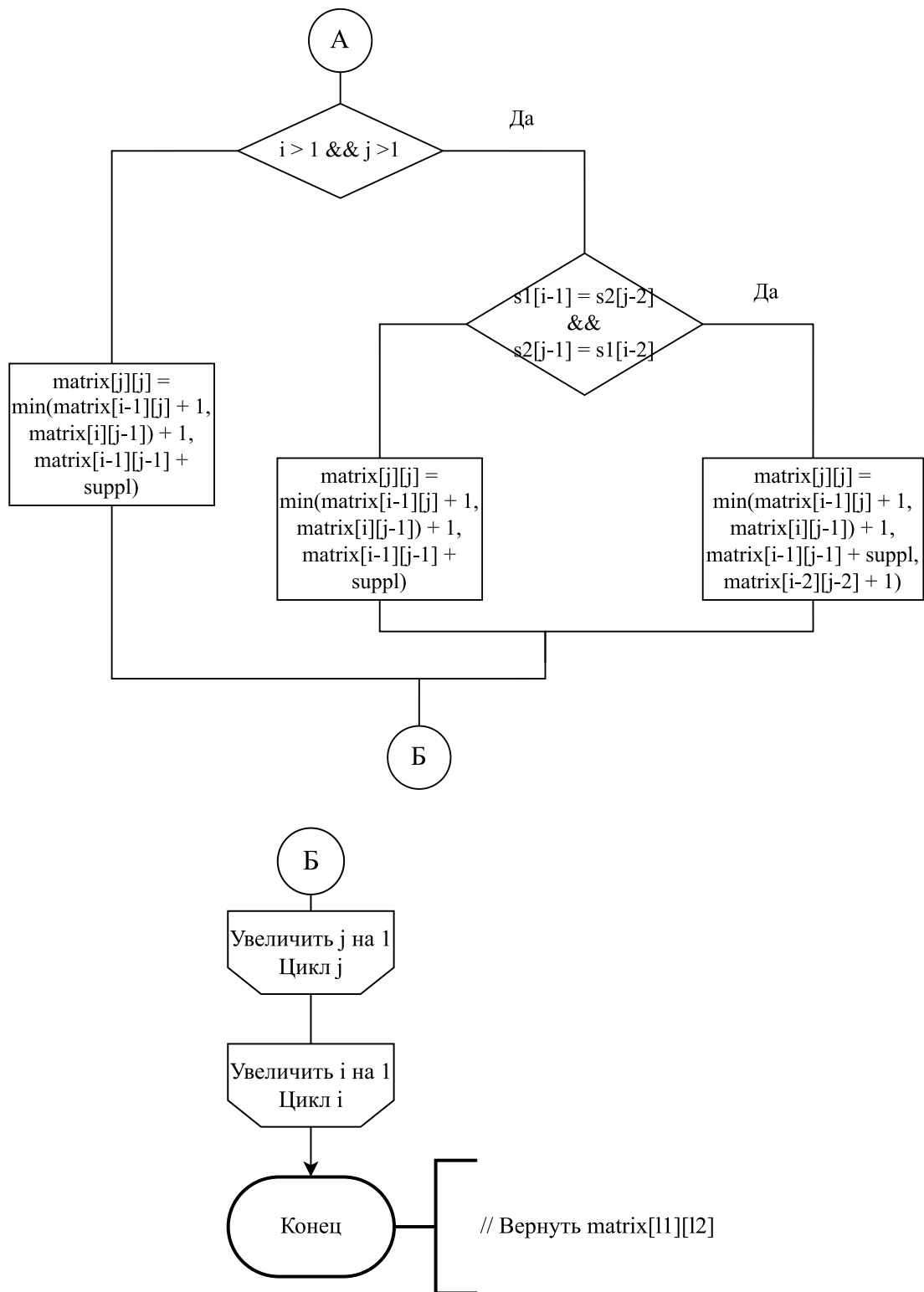


Рисунок 2 – Итерационный алгоритм поиска расстояния Дameraу-Левенштейна

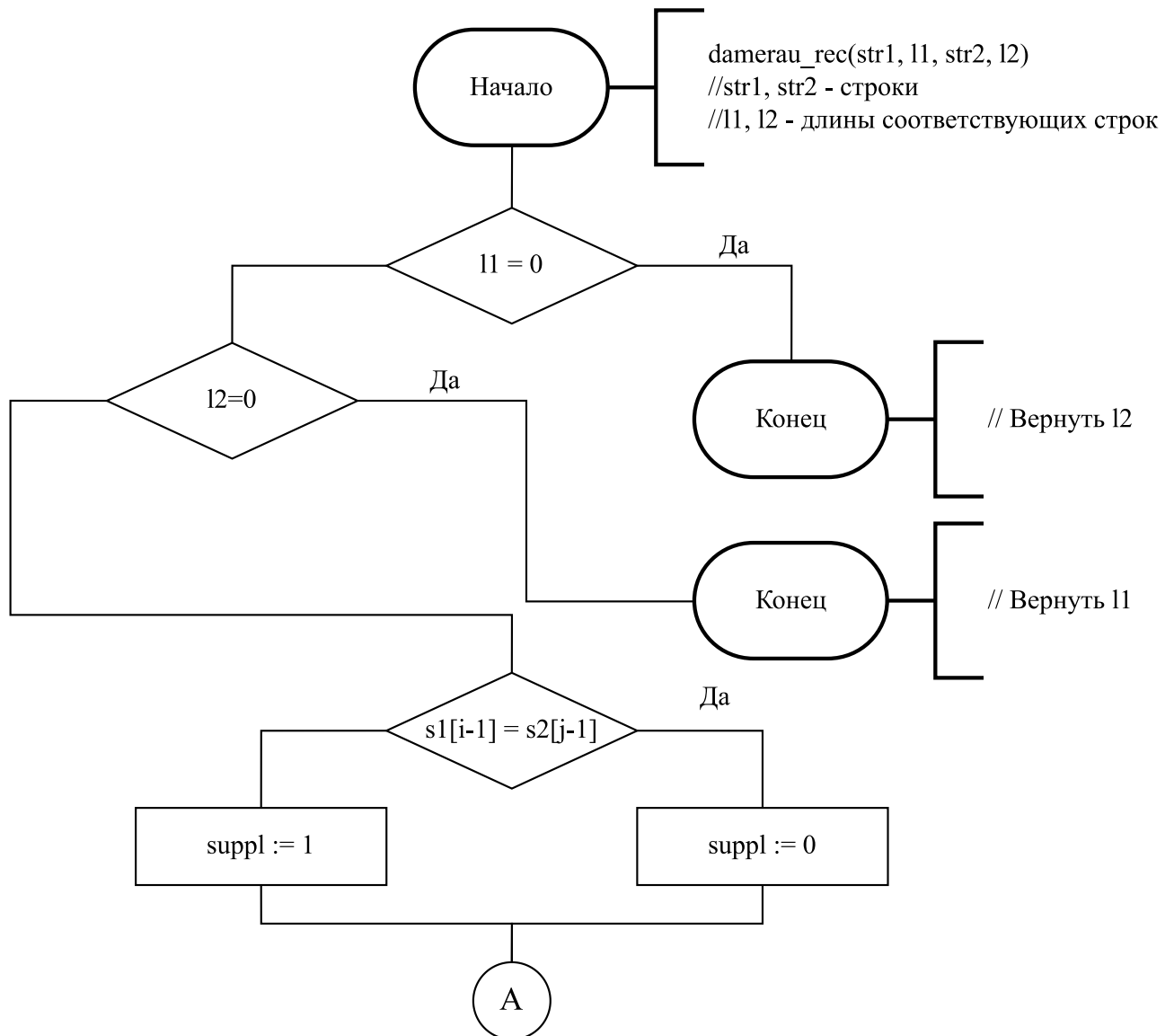
Оценка памяти алгоритма:

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 Дameraу-Левенштейна



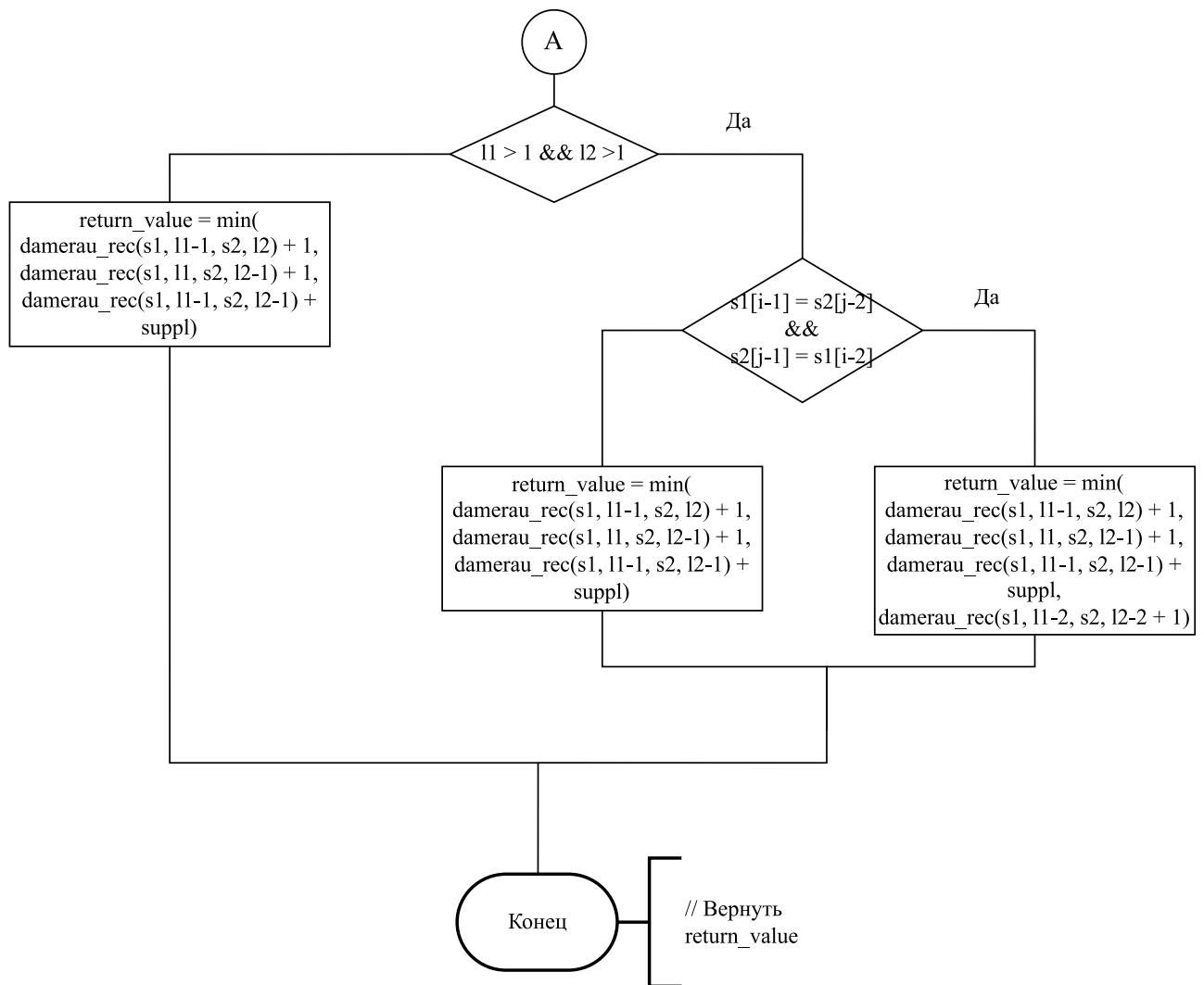


Рисунок 3 – Рекурсивный алгоритм поиска расстояния Дameraу-Левенштейна

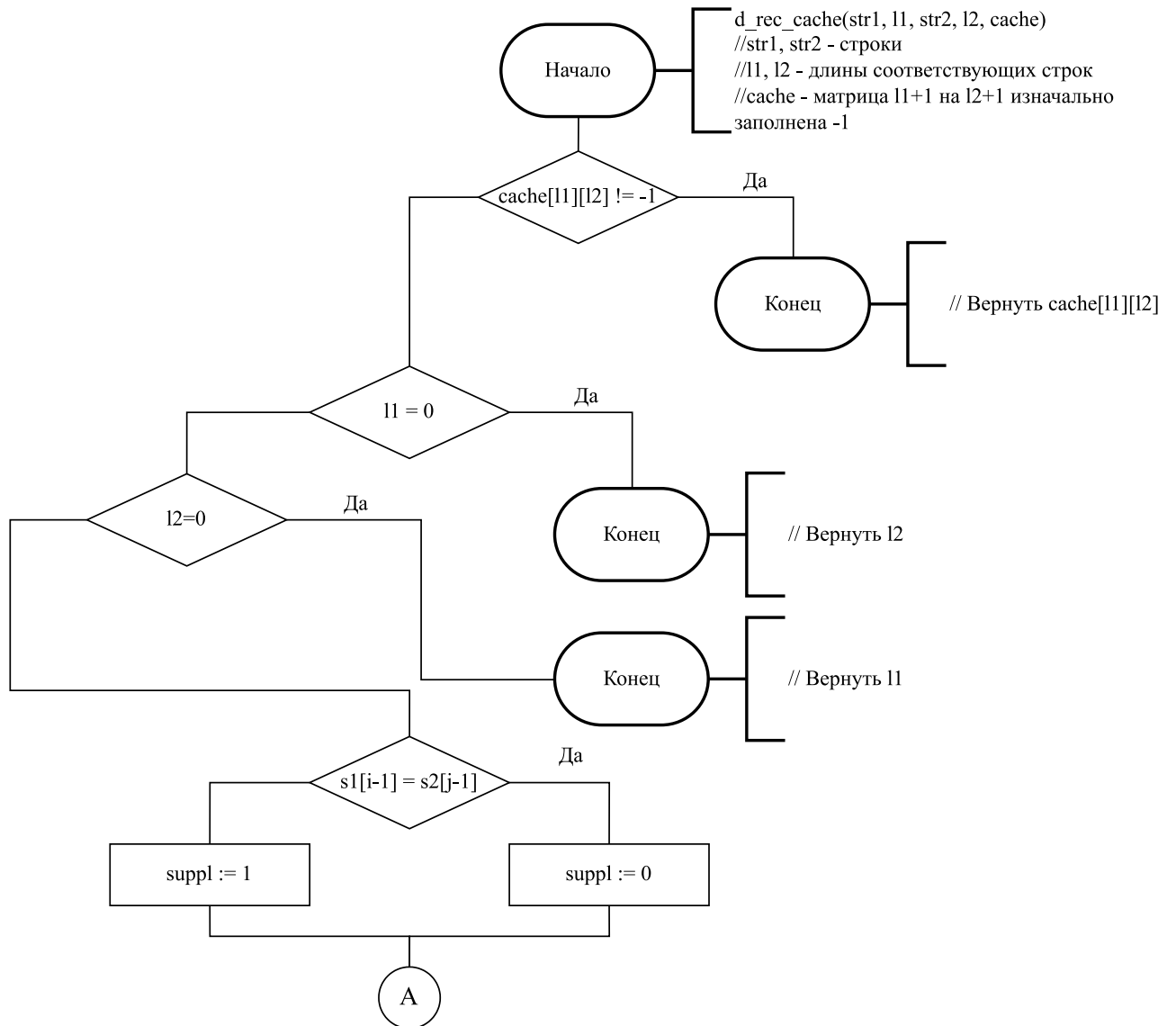
Оценка памяти алгоритма:

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 Дameraу-Левенштейна



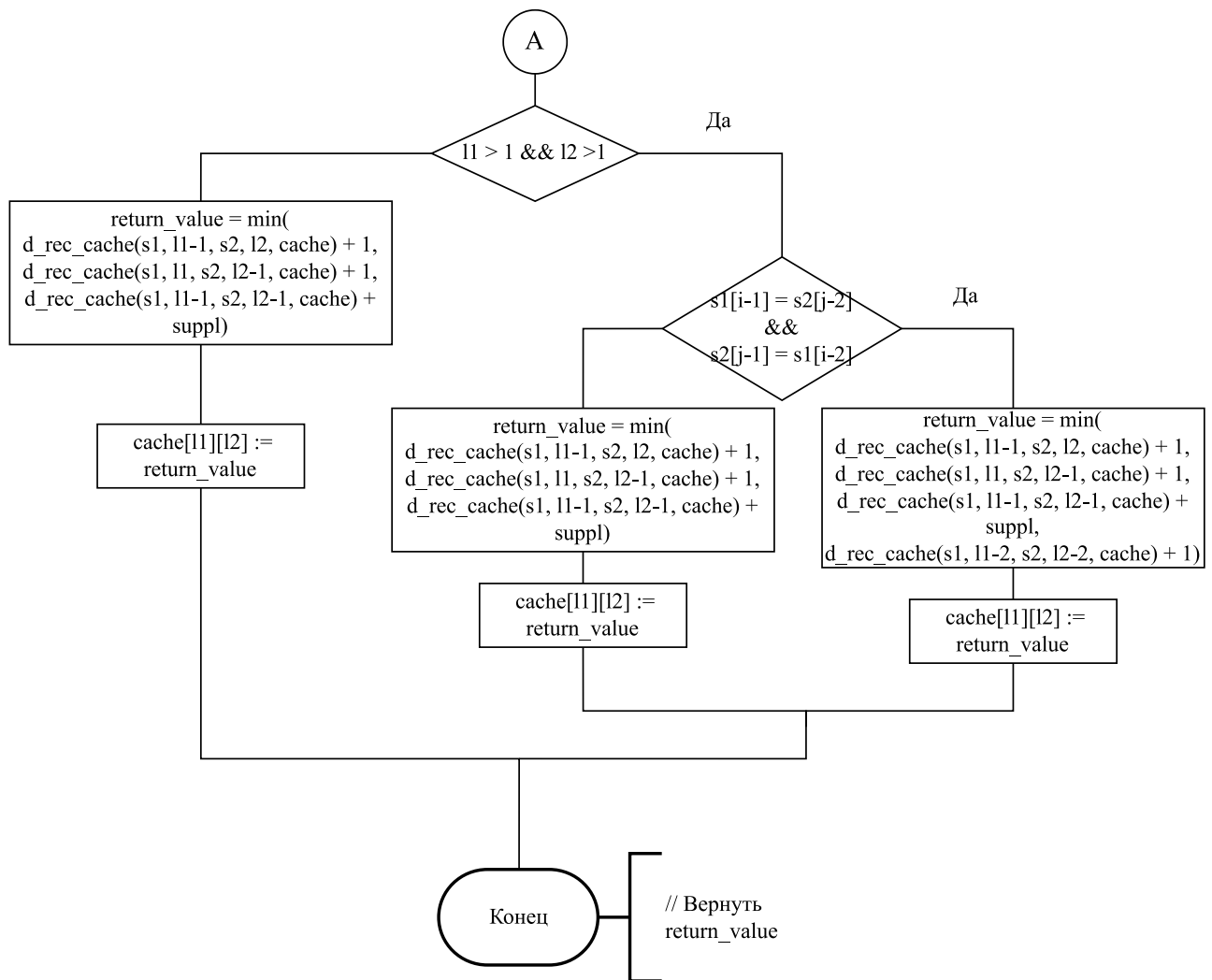


Рисунок 4 – Рекурсивный с кэшем алгоритм поиска расстояния Дameraу-Левенштейна

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: Итерационный алгоритм поиска расстояния Левенштейна

```
1  int levenshtein(char *s1, int l1, char *s2, int l2)
2  {
3      int matrix[l1 + 1][l2 + 1];
4      for (int i = 0; i < l1 + 1; i++)
5      {
6          for (int j = 0; j < l2 + 1; j++)
7          {
8              if (i == 0)
9              {
10                 matrix[0][j] = j;
11                 continue;
12             }
13             if (j == 0)
14             {
15                 matrix[i][0] = i;
16                 continue;
17             }
18             char ch1 = s1[i - 1];
19             char ch2 = s2[j - 1];
20             matrix[i][j] = min3(matrix[i - 1][j] + 1,
21                                matrix[i][j - 1] + 1,
22                                matrix[i - 1][j - 1]
23                                + (ch1 == ch2 ? 0 : 1));
24         }
25     }
26     return matrix[l1][l2];
27 }
```

Листинг 2: Итерационный алгоритм поиска расстояния Дameraу-Левенштейна

```

1  int damerau(char *s1, int l1, char *s2, int l2)
2  {
3      int matrix[l1 + 1][l2 + 1];
4      for (int i = 0; i < l1 + 1; i++)
5      {
6          for (int j = 0; j < l2 + 1; j++)
7          {
8              if (i == 0)
9              {
10                 matrix[0][j] = j;
11                 continue;
12             }
13             if (j == 0)
14             {
15                 matrix[i][0] = i;
16                 continue;
17             }
18             char ch1 = s1[i - 1];
19             char ch2 = s2[j - 1];
20             if (i > 1 && j > 1)
21             {
22                 char pch1 = s1[i - 2];
23                 char pch2 = s2[j - 2];
24                 if (ch1 == pch2 && ch2 == pch1)
25                 {
26                     matrix[i][j] = min4(matrix[i - 1][j] + 1,
27                                           matrix[i][j - 1] + 1,
28                                           matrix[i - 1][j - 1]
29                                           + (ch1 == ch2 ? 0 : 1),
30                                           matrix[i - 2][j - 2]
31                                           + 1);
32                 }
33             else

```

```

34         {
35             matrix[i][j] = min3(matrix[i - 1][j] + 1,
36                                 matrix[i][j - 1] + 1,
37                                 matrix[i - 1][j - 1]
38                                 + (ch1 == ch2 ? 0 : 1));
39         }
40     }
41     else
42     {
43         matrix[i][j] = min3(matrix[i - 1][j] + 1,
44                               matrix[i][j - 1] + 1,
45                               matrix[i - 1][j - 1]
46                               + (ch1 == ch2 ? 0 : 1));
47     }
48 }
49 }
50 return matrix[l1][l2];
51 }

```

Листинг 3: Рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна

```

1  int damereu_rec(char *s1, int l1, char *s2, int l2)
2  {
3      if (l1 == 0)
4          return l2;
5      if (l2 == 0)
6          return l1;
7      char ch1 = s1[l1 - 1];
8      char ch2 = s2[l2 - 1];
9      if (l1 > 1 && l2 > 1)
10     {
11         char pch1 = s1[l1 - 2];
12         char pch2 = s2[l2 - 2];

```

```

13         if (ch1 == pch2 && ch2 == pch1)
14         {
15             return min4(damereu_rec(s1, l1 - 1, s2, l2) + 1,
16                         damereu_rec(s1, l1, s2, l2 - 1) + 1,
17                         damereu_rec(s1, l1 - 1, s2, l2 - 1)
18                         + (ch1 == ch2 ? 0 : 1),
19                         damereu_rec(s1, l1 - 2, s2, l2 - 2)
20                         + 1);
21         }
22         else
23         {
24             return min3(damereu_rec(s1, l1 - 1, s2, l2) + 1,
25                         damereu_rec(s1, l1, s2, l2 - 1) + 1,
26                         damereu_rec(s1, l1 - 1, s2, l2 - 1)
27                         + (ch1 == ch2 ? 0 : 1));
28         }
29     }
30     else
31     {
32         return min3(damereu_rec(s1, l1 - 1, s2, l2) + 1,
33                     damereu_rec(s1, l1, s2, l2 - 1) + 1,
34                     damereu_rec(s1, l1 - 1, s2, l2 - 1)
35                     + (ch1 == ch2 ? 0 : 1));
36     }
37 }

```

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

```

1  int damereu_rec_cache(char *s1, int l1, char *s2, int l2)
2  {
3      int **cache = malloc((l1 + 1) * sizeof(int*));
4      for (int i = 0; i < l1 + 1; i++)

```

```

5      {
6          cache[i] = malloc((l2 + 1) * sizeof(int));
7      }
8      for (int i = 0; i < l1 + 1; i++)
9      {
10         for (int j = 0; j < l2 + 1; j++)
11         {
12             cache[i][j] = -1;
13         }
14     }
15     return do_rec(s1, l1, s2, l2, cache, l1 + 1, l2 + 1);
16 }
17
18 int do_rec(char *s1, int l1, char *s2, int l2, int **cache,
19           int m, int n)
20 {
21     if (cache[l1][l2] != -1)
22         return cache[l1][l2];
23     if (l1 == 0)
24         return l2;
25     if (l2 == 0)
26         return l1;
27     char ch1 = s1[l1 - 1];
28     char ch2 = s2[l2 - 1];
29     if (l1 > 1 && l2 > 1)
30     {
31         char pch1 = s1[l1 - 2];
32         char pch2 = s2[l2 - 2];
33         if (ch1 == pch2 && ch2 == pch1)
34         {
35             int dist = min4(do_rec(s1, l1 - 1, s2, l2,
36                                   cache, m, n) + 1,
37                             do_rec(s1, l1, s2, l2 - 1,
38                                   cache, m, n) + 1,
39                             do_rec(s1, l1 - 1, s2, l2 - 1,

```

```

40         cache, m, n)
41         + (ch1 == ch2 ? 0 : 1),
42         do_rec(s1, l1 - 2, s2, l2 - 2,
43             cache, m, n) + 1);
44     cache[l1][l2] = dist;
45     return dist;
46 }
47 else
48 {
49     int dist = min3(do_rec(s1, l1 - 1, s2, l2,
50         cache, m, n) + 1,
51         do_rec(s1, l1, s2, l2 - 1,
52             cache, m, n) + 1,
53         do_rec(s1, l1 - 1, s2, l2 - 1,
54             cache, m, n)
55         + (ch1 == ch2 ? 0 : 1));
56     cache[l1][l2] = dist;
57     return dist;
58 }
59 }
60 else
61 {
62     int dist = min3(do_rec(s1, l1 - 1, s2, l2,
63         cache, m, n) + 1,
64         do_rec(s1, l1, s2, l2 - 1,
65             cache, m, n) + 1,
66         do_rec(s1, l1 - 1, s2, l2 - 1,
67             cache, m, n)
68         + (ch1 == ch2 ? 0 : 1));
69     cache[l1][l2] = dist;
70     return dist;
71 }
72 }

```

3.3 Тестирование

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

№	Строка 1	Строка 2	Результат Левенштейн	Результат Дameraу-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 характеристики

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

4.3 Выводы

ЗАКЛЮЧЕНИЕ

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)