МГТУ им. Баумана

Лабораторная работа №1

По курсу: "Анализ алгоритмов"

Расстояние Левенштейна

Работу выполнила: Лаврова Анастасия, ИУ7-55Б

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

Москва, 2019

Оглавление

Введение

Расстояние Левенштейна - минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове
- сравнения текстовых файлов утилитой diff
- в биоинформатике для сравнения генов, хромосом и белков

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);

5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;

6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (англ. delete) — удалить,
2. I (англ. insert) — вставить,
3. R (replace) — заменить,
4. M(match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(i, j-1) + 1, & \\ D(i-1, j) + 1, & j > 0, i > 0 \\ D(i-1, j-1) + m(S_1[i], S_2[j]) & \\ ), & \end{cases}$$

где $m(a,b)$ равна нулю, если $a = b$ и единице в противном случае; $min\{a, b, c\}$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекурентной формуле:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \\ D(i-2,j-2)+m(S_1[i],S_2[i]), \end{cases} & \text{, если } i,j > 0 \\ & \text{и } S_1[i] = S_2[j-1] \\ & \text{и } S_1[i-1] = S_2[j] \\ min \begin{cases} D(i,j-1)+1, \\ D(i-1,j)+1, \\ D(i-1,j-1)+m(S_1[i],S_2[i]), \end{cases} & \text{, иначе} \end{cases}$$

1.0.1 Вывод

В данном разделе были рассмотрены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, который является модификаций первого, учитывающего возможность перестановки соседних символов.

2 | Конструкторская часть

Требования к вводу:

1. На вход подаются две строки
2. uppercase и lowercase буквы считаются разными

Требования к программе:

1. Две пустые строки - корректный ввод, программа не должна аварийно завершаться

2.1 Схемы алгоритмов

В данной части будут рассмотрены схемы алгоритмов.

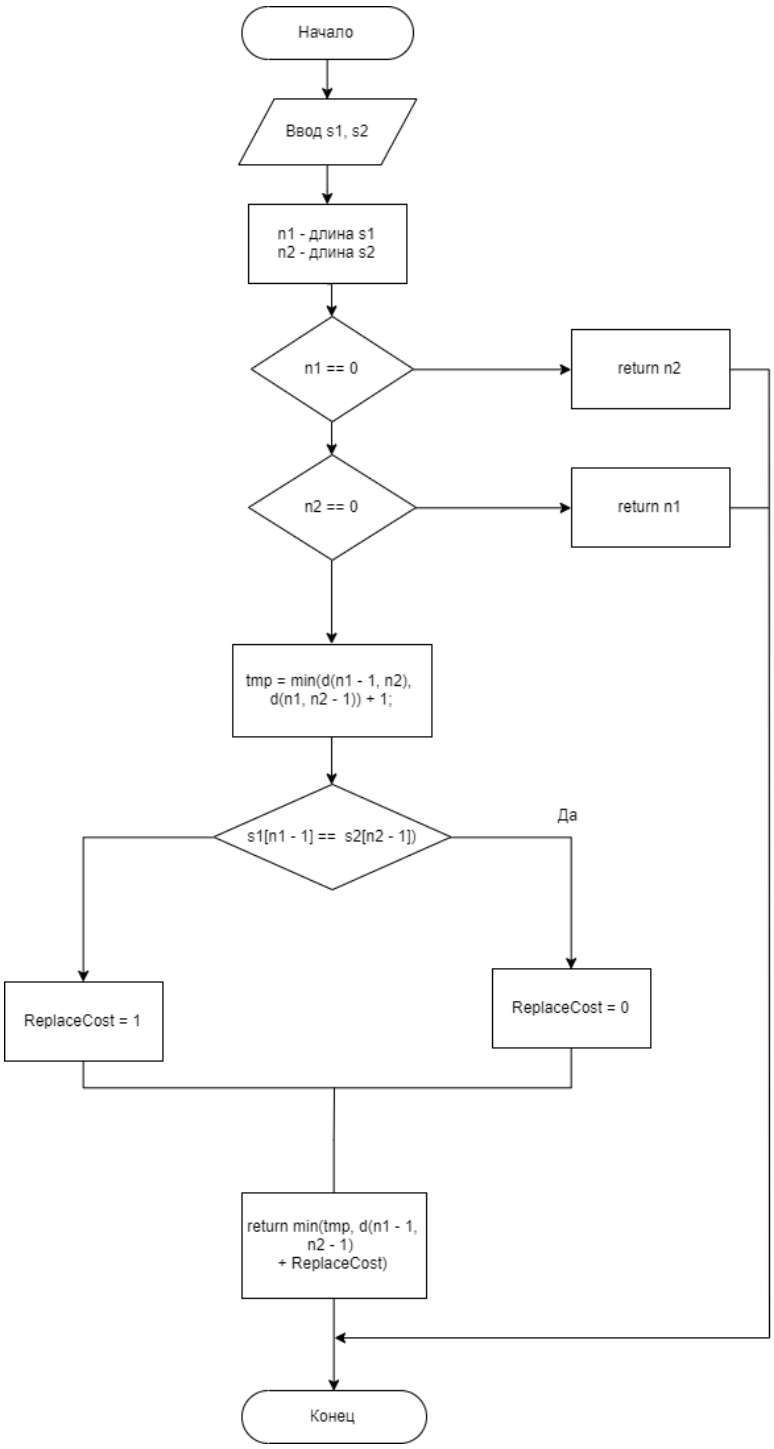


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

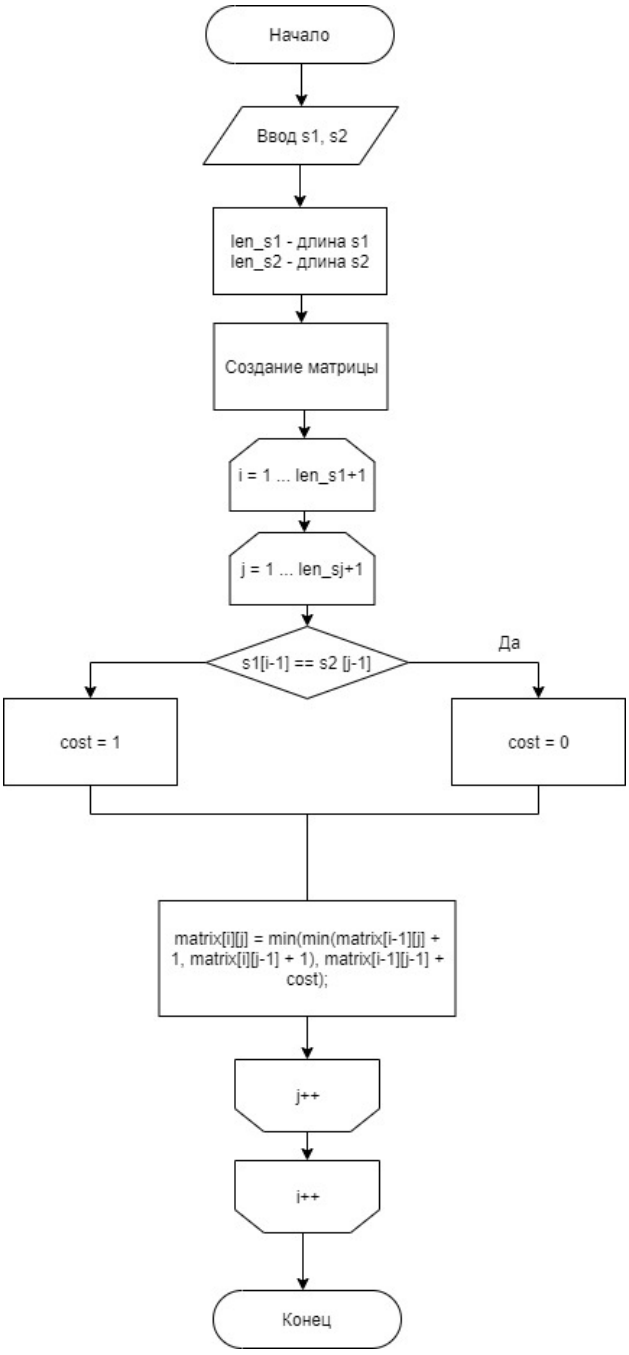


Рис. 2.2: Схема матричного алгоритма нахождения расстояния Левенштейна

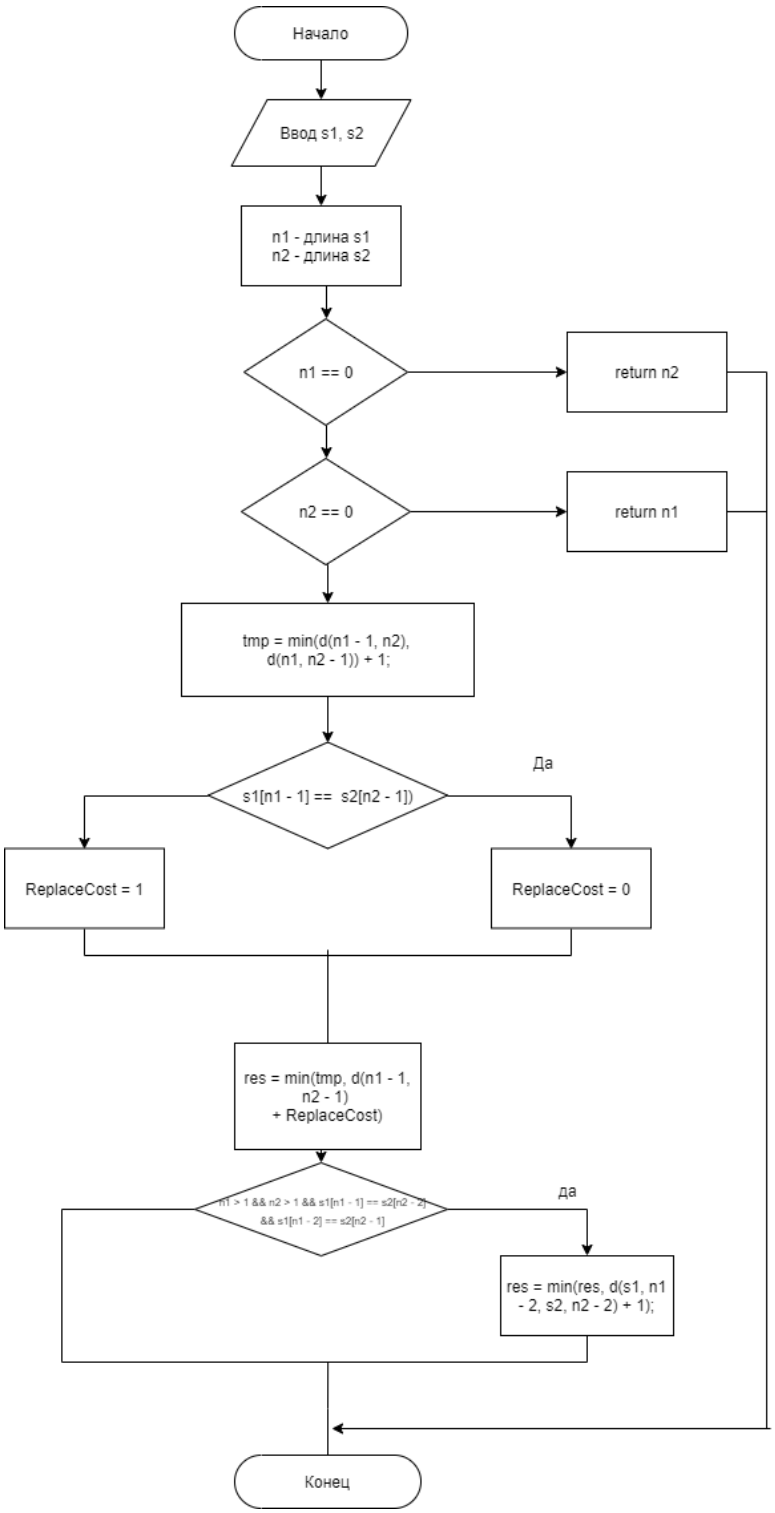


Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

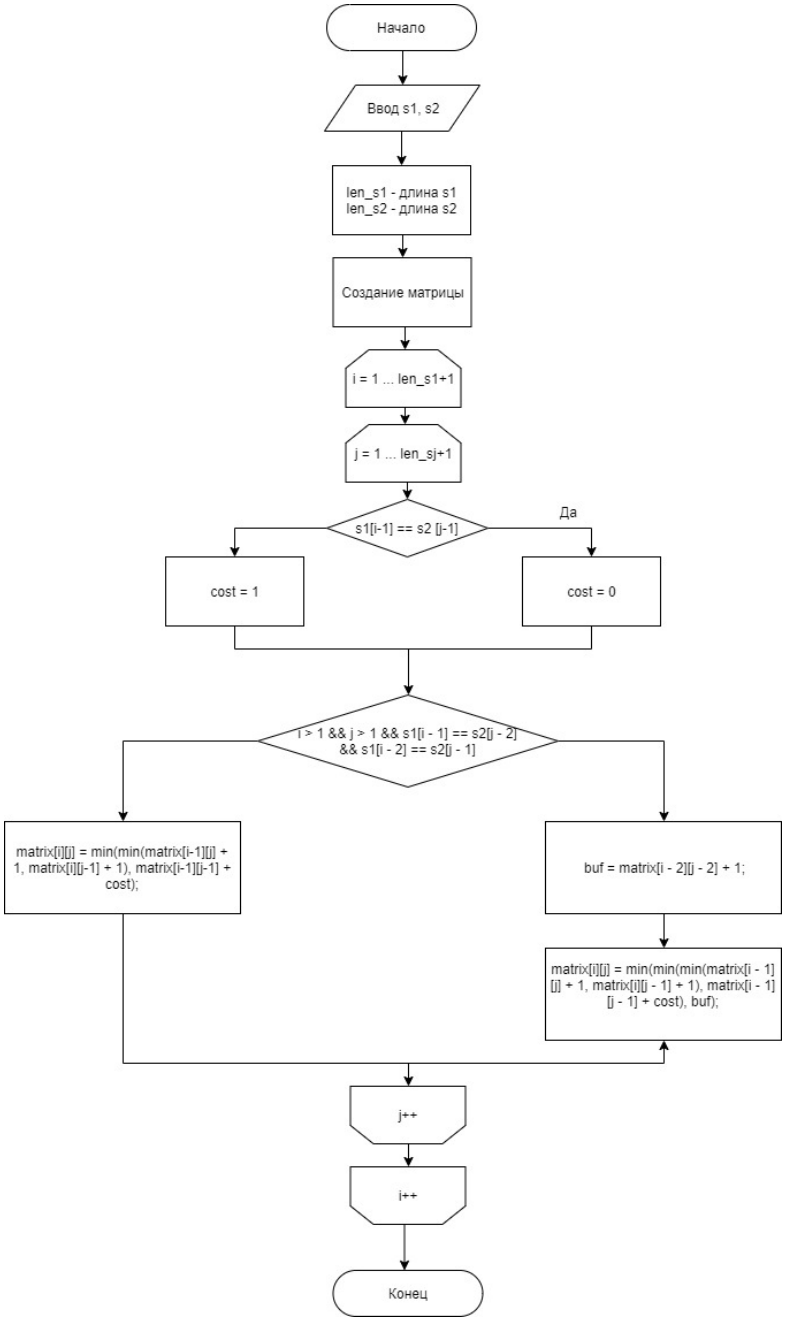


Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

3 | Технологическая часть

3.1 Выбор ЯП

Для реализации программ я выбрала язык программирования C++, так имею большой опыт работы с ним. Среда разработки - Visual Studio.

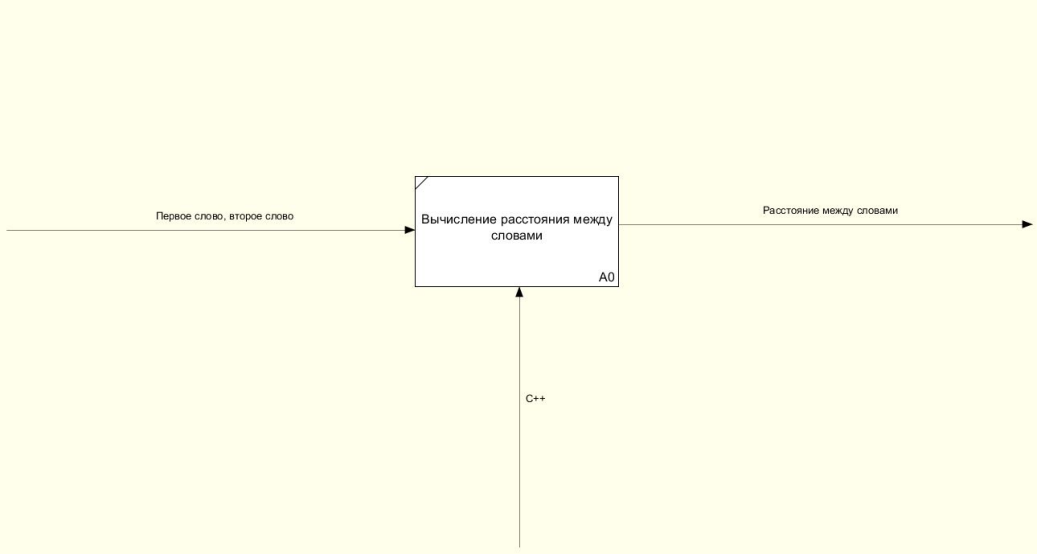


Рис. 3.1: IDEF0-диаграмма, описывающая алгоритм нахождения расстояния Левенштейна

3.2 Реализация алгоритма

Листинг 3.1: Функция нахождения расстояния Левенштейна рекурсивно

```
int levenshtein_rec(string &s1, int n1, string &s2, int n2)
{
  if (n1 == 0)
    return n2;
  if (n2 == 0)
    return n1;
  unsigned int tmp = min(levenshtein_rec(s1, n1 - 1, s2, n2
      ),
    levenshtein_rec(s1, n1, s2, n2 - 1)) + 1;
  return min(tmp, levenshtein_rec(s1, n1 - 1, s2, n2 - 1)
    + ReplaceCost(s1[n1 - 1], s2[n2 - 1]));
}
```

Листинг 3.2: Функция нахождения штрафа

```
unsigned int ReplaceCost(char a, char b)
{
  return (a != b);
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна матрично

```
int levenshtein_matrix(string s1, string s2)
{
  int len_s1 = s1.length();
  int len_s2 = s2.length();
  if (len_s1 == 0)
    return len_s2;
  if (len_s2 == 0)
    return len_s1;

  int **matrix = (int**)malloc((len_s1+1) * sizeof(int*));
  for (int i = 0; i <= len_s1; i++)
  {
    matrix[i] = (int*)malloc((len_s2 + 1) * sizeof(int));
  }

  for (int i = 0; i <= len_s1; i++)
  {
    for (int j = 0; j <= len_s2; j++)
```

```
19      {
20        if (i == 0)
21          matrix[i][j] = j;
22        if (j == 0)
23          matrix[i][j] = i;
24      }
25    }
26
27    int cost;
28    for (int i = 1; i < len_s1 + 1; i++)
29    {
30      for (int j = 1; j < len_s2 + 1; j++)
31      {
32        cost = s1[i-1] == s2[j-1] ? 0 : 1;
33        matrix[i][j] = min(min(matrix[i-1][j] + 1,
34          matrix[i][j-1] + 1), matrix[i-1][j-1] + cost);
35      }
36    }
37    for (int i = 0; i <= len_s1; i++)
38    {
39      for (int j = 0; j <= len_s2; j++)
40      {
41        cout << matrix[i][j] << " ";
42      }
43      cout << "\n";
44    }
45
46    int res = matrix[len_s1][len_s2];
47
48    for (int i = 0; i <= len_s1; i++)
49      free(matrix[i]);
50    free(matrix);
51    return res;
52  }
```

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна рекурсивно

```
1  unsigned int levenshtein_damerau_rec(string &s1, int n1,
       string &s2, int n2)
2  {
```

```
3   if (n1 == 0)
4     return n2;
5   if (n2 == 0)
6     return n1;
7   unsigned int tmp = min(levenshtein_damerau_rec(s1, n1 -
        1, s2, n2),   levenshtein_damerau_rec(s1, n1, s2, n2 -
         1)) + 1;
8   unsigned int res =  min(tmp, levenshtein_damerau_rec(s1,
        n1 - 1, s2, n2 - 1) + ReplaceCost(s1[n1 - 1], s2[n2 -
        1]));
9   if (n1 > 1 && n2 > 1 && s1[n1 - 1] == s2[n2 - 2] && s1[n1
         - 2] == s2[n2 - 1])
10  {
11    res = min(res, levenshtein_damerau_rec(s1, n1 - 2, s2,
          n2 - 2) + 1);
12  }
13  return res;
14 }
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна матрично

```
1 int levenshtein_damerau_matrix(string s1, string s2)
2 {
3   int len_s1 = s1.length();
4   int len_s2 = s2.length();
5   if (len_s1 == 0)
6     return len_s2;
7   if (len_s2 == 0)
8     return len_s1;
9
10  int **matrix = (int**)malloc((len_s1 + 1) * sizeof(int*))
        ;
11  for (int i = 0; i <= len_s1; i++)
12  {
13    matrix[i] = (int*)malloc((len_s2 + 1) * sizeof(int));
14  }
15
16  for (int i = 0; i <= len_s1; i++)
17  {
18    for (int j = 0; j <= len_s2; j++)
```

```
19      {
20        if (i == 0)
21          matrix[i][j] = j;
22        if (j == 0)
23          matrix[i][j] = i;
24      }
25    }
26
27    int cost, buf;
28    for (int i = 1; i < len_s1 + 1; i++)
29    {
30      for (int j = 1; j < len_s2 + 1; j++)
31      {
32        cost = s1[i - 1] == s2[j - 1] ? 0 : 1;
33        if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i
             - 2] == s2[j - 1])
34        {
35          buf = matrix[i - 2][j - 2] + 1;
36          matrix[i][j] = min(min(min(matrix[i - 1][j] + 1,
               matrix[i][j - 1] + 1), matrix[i - 1][j - 1] +
               cost), buf);
37        }
38        else
39        {
40          matrix[i][j] = min(min(matrix[i - 1][j] + 1,
41            matrix[i][j - 1] + 1), matrix[i - 1][j - 1] +
                 cost);
42        }
43      }
44    }
45    for (int i = 0; i <= len_s1; i++)
46    {
47      for (int j = 0; j <= len_s2; j++)
48      {
49        cout << matrix[i][j] << " ";
50      }
51      cout << "\n";
52    }
53
54    int res = matrix[len_s1][len_s2];
```

```
55
56    for (int i = 0; i <= len_s1; i++)
57      free(matrix[i]);
58    free(matrix);
59    return res;
60  }
```

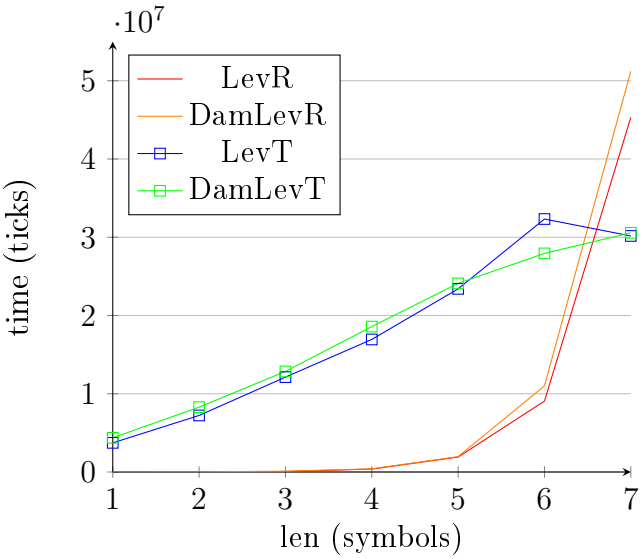
4 Исследовательская часть

4.1 Сравнительный анализ на основе замеров времени работы алгоритмов

Был проведен замер времени работы каждого из алгоритмов.

Таблица 4.1: Время работы алгоритмов (в тиках)

len	Lev(R)	Lev(T)	DamLev(R)	DamLev(T)
1	5928	3724258	7456	4367560
2	16865	7224736	21854	8286833
3	62333	12123365	105445	12852145
4	372661	16940041	407763	18585284
5	1909255	23402008	1966658	24103230
6	9065189	32328258	11002094	27935583
7	453325069	30166031	51219656	30567571



Наиболее эффективными по времени при маленькой длине слова являются рекурсивные реализации алгоритмов, но как только увеличивается длина слова, их эффективность резко снижается, что обусловлено большим количеством повторных рассчетов. Время работы алгоритма, использующего матрицу, намного меньше благодаря тому, что в нем требуется только (m + 1)*(n + 1) операций заполнения ячейки матрицы. Также установлено, что алгоритм ДамерауЛевенштейна работает немного дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки, однако алгоритмы сравнимы по временной эффективности.

4.2 Тестовые данные

Проведем тестирование программы. В столбцах "Ожидаемый результат"и "Полученный результат"4 числа соответсвуют рекурсивному алгоритму нахождения расстояния Левенштейна, матричному алгоритму нахождению расстоянию Левенштейна, рекурсивному алгоритму расстояния Дамерау-Левенштейна, матричному алгоритму нахождения расстояние Дамерау-Левенштейна.

Таблица 4.2: Таблица тестовых данных

№	Первое слово	Второе слово	Ожидаемый результат	Полученный результат
1			0 0 0 0	0 0 0 0
2	kot	skat	2 2 2 2	2 2 2 2
3	kate	ktae	2 2 1 1	2 2 1 1
4	abacaba	aabcaab	4 4 2 2	4 4 2 2
5	sobaka	sboku	3 3 3 3	3 3 3 3
6	qwerty	queue	4 4 4 4	4 4 4 4
7	apple	aplpe	2 2 1 1	2 2 1 1
8		cat	3 3 3 3	3 3 3 3
9	parallels		9 9 9 9	9 9 9 9
10	bmstu	utsmb	4 4 4 4	4 4 4 4

Заключение

Был изучен метод динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна. Также изучены алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками, получены практические навыки раелизации указанных алгоритмов в матричной и рекурсивных версиях.

Экспериментально было подтверждено различие во временной эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработаного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк.

В результате исследований я пришла к выводу, что матричная реализация данных алгоритмов заметно выигрывает по времени при росте длины строк, следовательно более применима в реальных проектах.