



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 курсу «Анализ Алгоритмов»
на тему: «Динамическое программирование»

Студент ИУ7-56Б
(Группа)

Преподаватели

Мансуров В. М.
(Фамилия И. О.)

Волкова Л. Л., Строганов Ю. В.
(Фамилия И. О.)

Москва — 2022 г.

Содержание

Введение	3
1 Аналитическая часть	4
1.1 Расстояние Левенштейна	4
1.1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна	5
1.1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кешированием	6
1.1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна	6
1.2 Расстояние Дамерау-Левенштейна	6
1.3 Вывод	7
2 Конструкторская часть	8
2.1 Функциональные требования	8
2.2 Разработка алгоритмов	8
2.3 Описание используемых типов данных	14
2.4 Вывод	14
3 Технологическая часть	15
3.1 Требования к программному обеспечению	15
3.2 Средства реализации	15
3.3 Сведения о модулях программы	15
3.4 Функциональные тесты	23
3.5 Вывод	23
4 Исследовательская часть	24
4.1 Технические характеристики	24
4.2 Демонстрация работы программы	24
4.3 Временные характеристики	26
4.4 Характеристики по памяти	28
4.5 Сравнительный анализ алгоритмов	32

4.6 Вывод	32
Заключение	33
Список использованных источников	34

Введение

В данной лабораторной работе будет рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замены), которое необходимо для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0-1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой diff;
- для сравнения генов, хромосом и белков в биоинформатике.

Целью данной лабораторной работы является изучение и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для поставленной цели необходимо выполнить следующие задачи:

- 1) изучить расстояния Левенштейна и Дамерау-Левенштейна;
- 2) создать ПО, реализующее следующие алгоритмы:
 - нерекурсивный метод поиска расстояния Левенштейна;
 - нерекурсивный метод поиска Дамерау-Левенштейна;
 - рекурсивный метод поиска Дамерау-Левенштейна;
 - рекурсивный с кэшированием метод поиска Дамерау-Левенштейна.
- 3) выбрать инструменты для замера процессорного времени выполнения реализаций алгоритмов;
- 4) Провести анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов.

Стоимость операций могут зависеть от вида операций:

1. $w(a, b)$ — цена замены символа a на b , R (от англ. replace);
2. $w(\lambda, b)$ — цена вставки символа b , I (от англ. insert);
3. $w(a, \lambda)$ — цена удаления символа a , D (от англ. delete).

Будем считать стоимость каждой вышеизложенной операции равной 1, то есть:

- $w(a, b) = 1, a \neq b$;
- $w(\lambda, b) = 1$;
- $w(a, \lambda) = 1$.

Введем понятие совпадения символов — M (от англ. match). Его стоимость будет равна 0, то есть $w(a, a) = 0$.

Расстояние Левенштейна между двумя строками S_1 и S_2 , длиной M и N соответственно. Расстояние Левенштейна рассчитывается по рекуррентной формуле 1.1:

$$D(i, j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ \min \begin{cases} D(i, j - 1) + 1 \\ D(i - 1, j) + 1 \\ D(i - 1, j - 1) + m(S_1[i], S_2[i]) \end{cases} & i > 0, j > 0 \end{cases} \quad (1.1)$$

$$m(a, b) = \begin{cases} 0 & \text{если } a = b, \\ 1 & \text{иначе} \end{cases}. \quad (1.2)$$

1.1.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм реализует формулу 1.1, функция D составлена таким образом, что:

1. Для передачи из пустой строки в пустую требуется ноль операций;
2. Для перевода из пустой строки в строку a требуется $|a|$ операций;
3. Для перевода из строки a в пустую строку требуется $|a|$ операций;
4. Для перевода из строки a в строку b требуется выполнить последовательно некоторое количество операций (удаление, вставка, замена) в некоторой последовательности. Последовательность поведения любых двух операций можно поменять, порядок поведения операций не имеет никакого значения. Пологая, что a' , b' - строки a и b без последнего символа соответственно, цена преобразования из строки a в строку b может быть выражена как:

- стоимость преобразования строки a в b и цены проведения операции удаления, которая необходима для преобразования a' в a ;
- стоимость преобразования строки a в b и цены проведения операции вставки, которая необходима для преобразования b' в b ;
- стоимость преобразования из a' в b' и операции замены, предполагая, что a' и b' оканчиваются разные символы;
- стоимость преобразования из a' в b' , предполагая, что a и b оканчиваются на один и тот же символ.

Минимальной стоимостью преобразования будет минимальное значение приведенных вариантов.

1.1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кешированием

Рекурсивная реализация алгоритма Левенштейна малоэффективна по времени при больших i, j , по причине проблемы повторных вычислений $D(i, j)$. Для оптимизации нахождения расстояния Левенштейна можно использовать матрицу в целях хранения соответствующих промежуточных значений. В таком случае алгоритм представляет собой рекурсивное заполнение матрицы $A_{|a|,|b|}$ значениями $D(i, j)$, такое хранение промежуточных данных можно назвать кэшем для рекурсивного алгоритма.

1.1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивная реализация алгоритма Левенштейна с кешированием малоэффективна по времени при больших i, j . Для оптимизации можно использовать итерационную реализацию заполнения матрицы промежуточными значениями $D(i, j)$.

Однако матричный алгоритм является малоэффективным по памяти при больших i, j , т.к. множество промежуточных значений $D(i, j)$ хранится в памяти после их использования. Для оптимизации нахождения расстояния Левенштейна можно использовать кэш, т.е. пару строк, содержащую значения $D(i, j)$, вычисленные в предыдущей итерации алгоритма и значения $D(i, j)$, вычисляемый в текущей итерации.

1.2 Расстояние Дameraу-Левенштейна

Расстояние Дameraу-Левенштейна (названо в честь ученых Фредерика Дameraу и Владимира Левенштейна) - это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиций (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния

Левенштейна, то есть к операциям добавляется операция транспозиция Т (от англ. transposition).

Расстояние Дамерау-Левенштейна может быть вычислено по рекуррентной формуле:

$$D(i, j) = \begin{cases} 0 & , j = 0, i = 0 \\ i & , j = 0, i > 0 \\ j & , j > 0, i = 0 \\ \min \begin{cases} D(i, j - 1) + 1 \\ D(i - 1, j) + 1 \\ D(i - 1, j - 1) + m(S_1[i], S_2[i]) \\ D(i - 2, j - 2) + m(S_1[i], S_2[i]) \end{cases} & \begin{matrix} , \text{если } i > 1, j > 1 \\ , S_1[i] = S_2[j - 1] \\ , S_1[j] = S_2[i - 1] \end{matrix} \\ \min \begin{cases} D(i, j - 1) + 1 \\ D(i - 1, j) + 1 \\ D(i - 1, j - 1) + m(S_1[i], S_2[i]) \end{cases} & , \text{иначе} \end{cases} \quad (1.3)$$

1.3 Вывод

В данном разделе были рассмотрены алгоритмы динамического программирования - алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна, формулы которых задаются рекуррентно, а следовательно, данные алгоритмы могут быть реализованы рекурсивно и итеративно. На вход алгоритмам поступают две строки, которые могут содержать как русские, так и английские буквы, также будет предусмотрен ввод пустых строк.

2 Конструкторская часть

В данном разделе будут приведены схемы алгоритмов нахождения расстояний Левенштейна и Дameraу-Левенштейна, приведено описание используемых типов данных, оценки памяти, а также описана структура ПО.

2.1 Функциональные требования

К программе предъявлены ряд требований:

- Входные данные - две строки;
- Выходные данные - результат работы всех алгоритмов поиска расстояний, целое число.

2.2 Разработка алгоритмов

На вход алгоритмов подаются строки S_1 и S_2

На рисунке 2.1 представлен схема алгоритма поиска расстояния Левенштейна.

На рисунках 2.2 - 2.5 представлены схемы алгоритмов поиска Дameraу-Левенштейна.

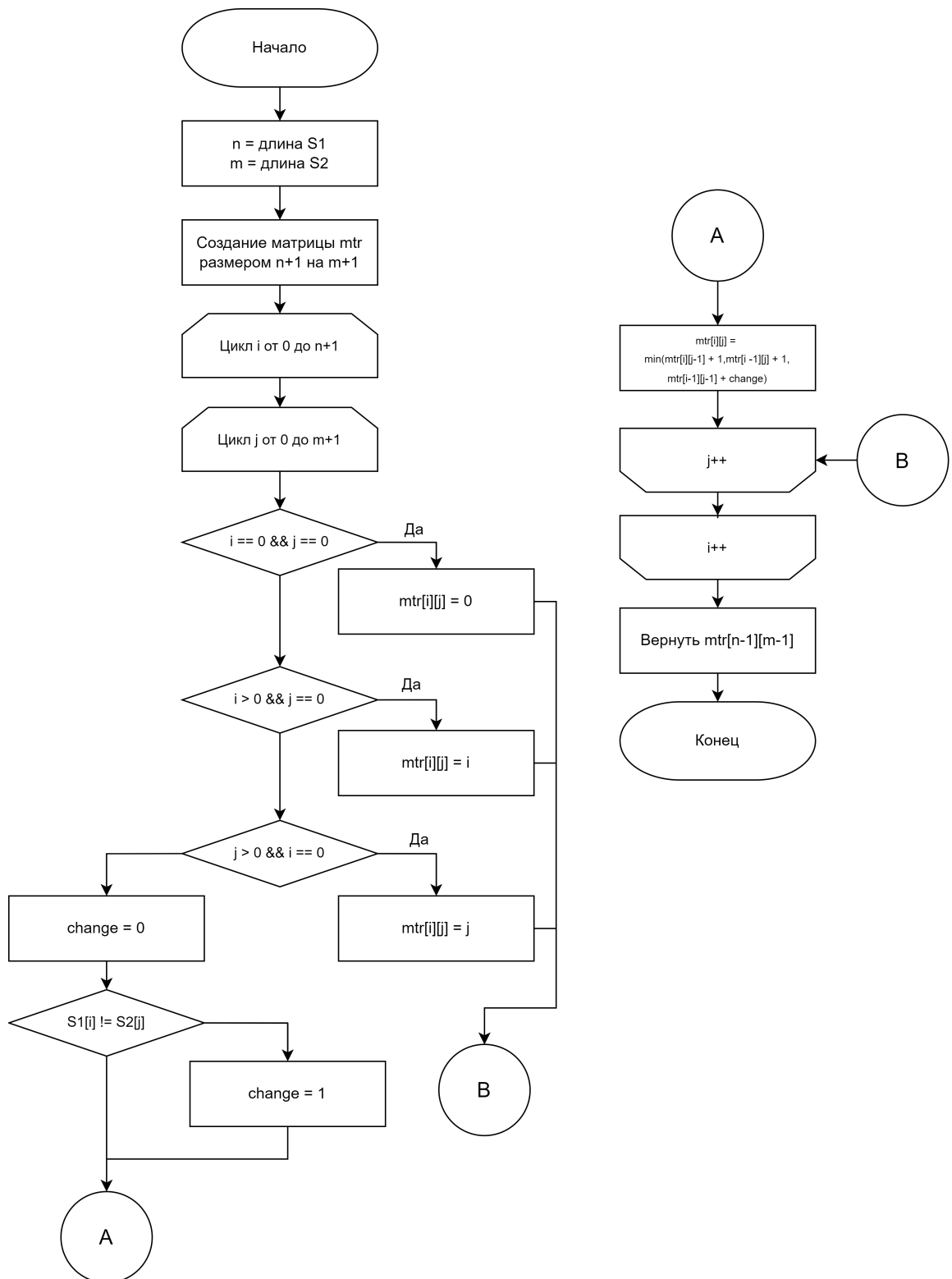


Рисунок 2.1 – Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

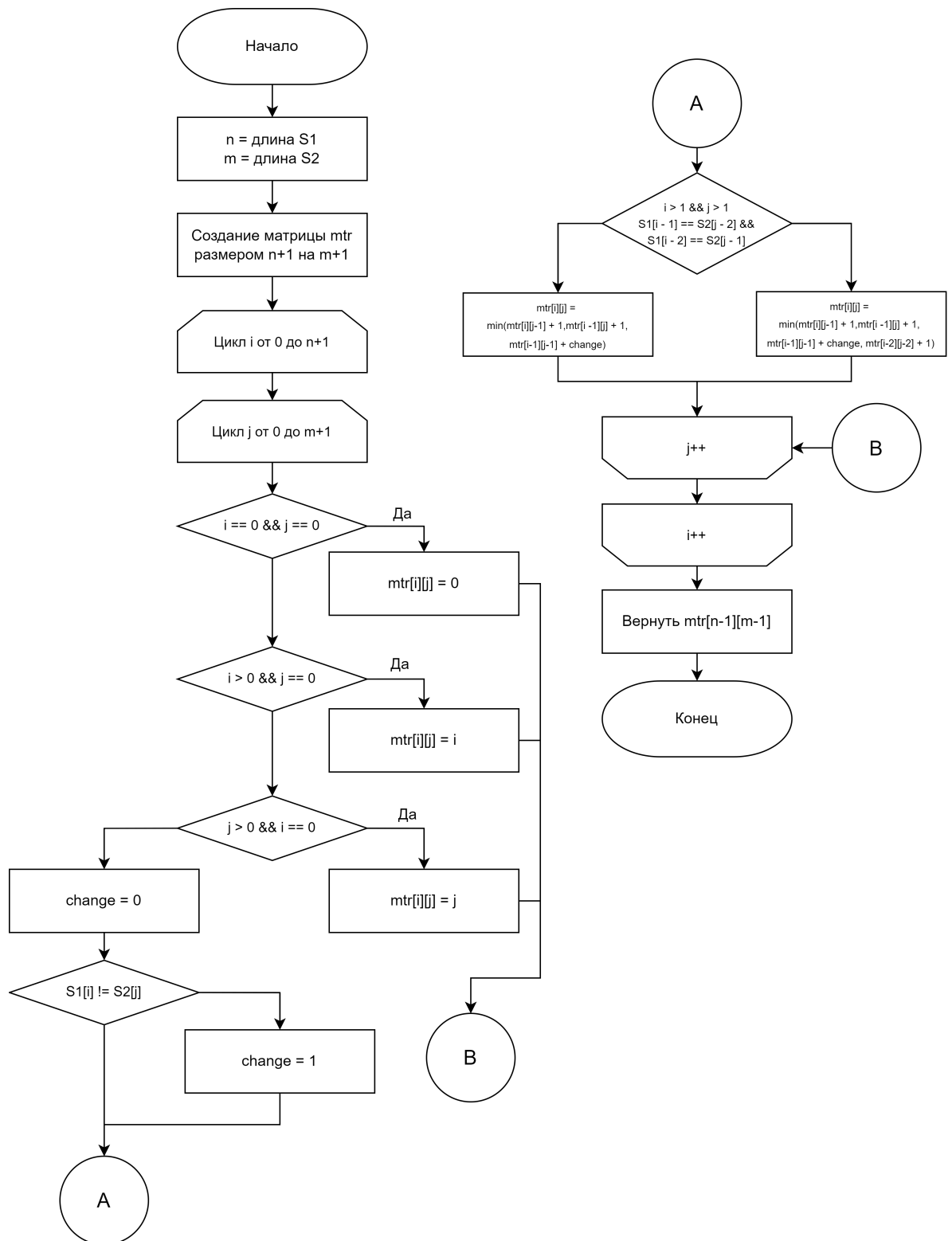


Рисунок 2.2 – Схема нерекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

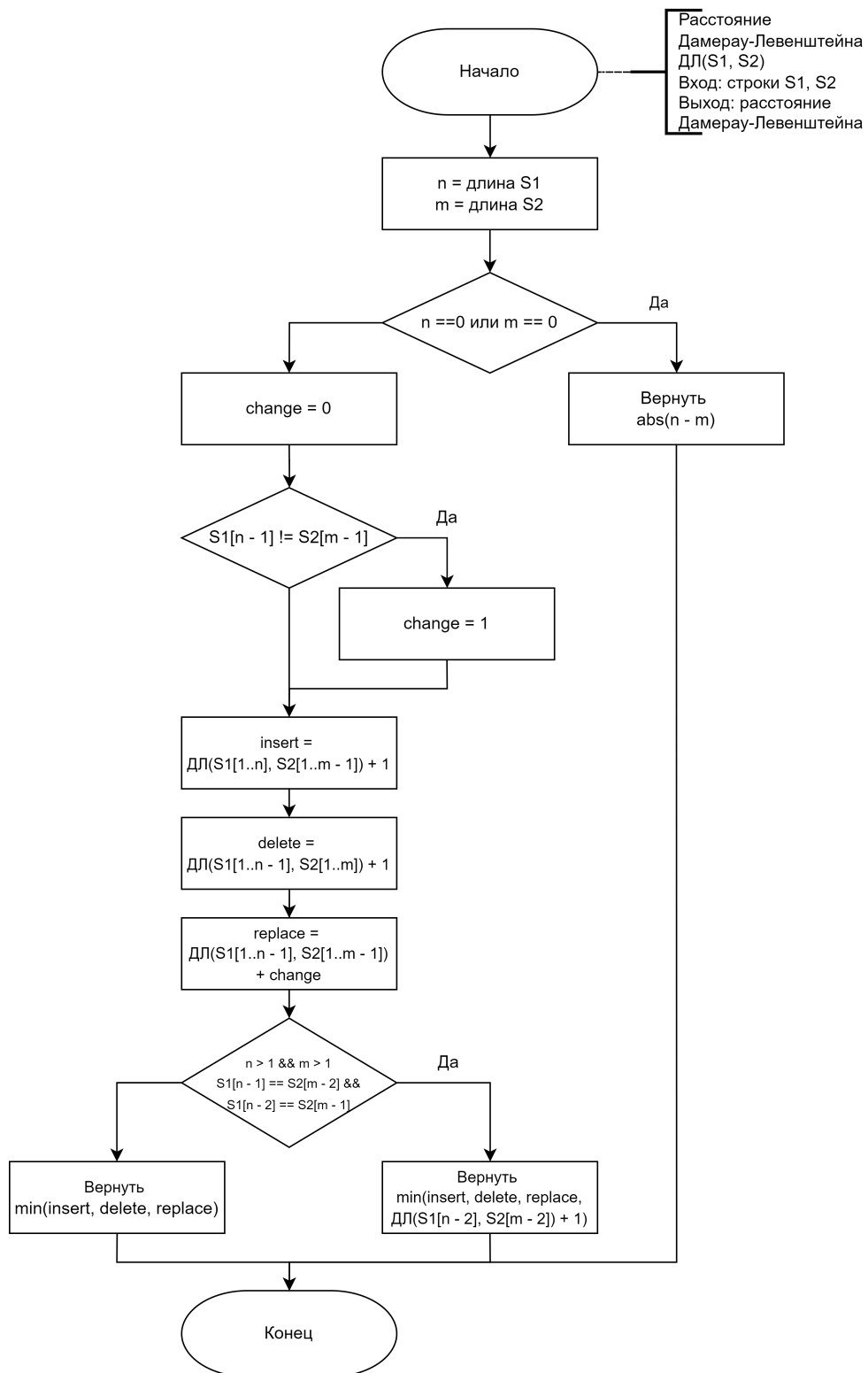


Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

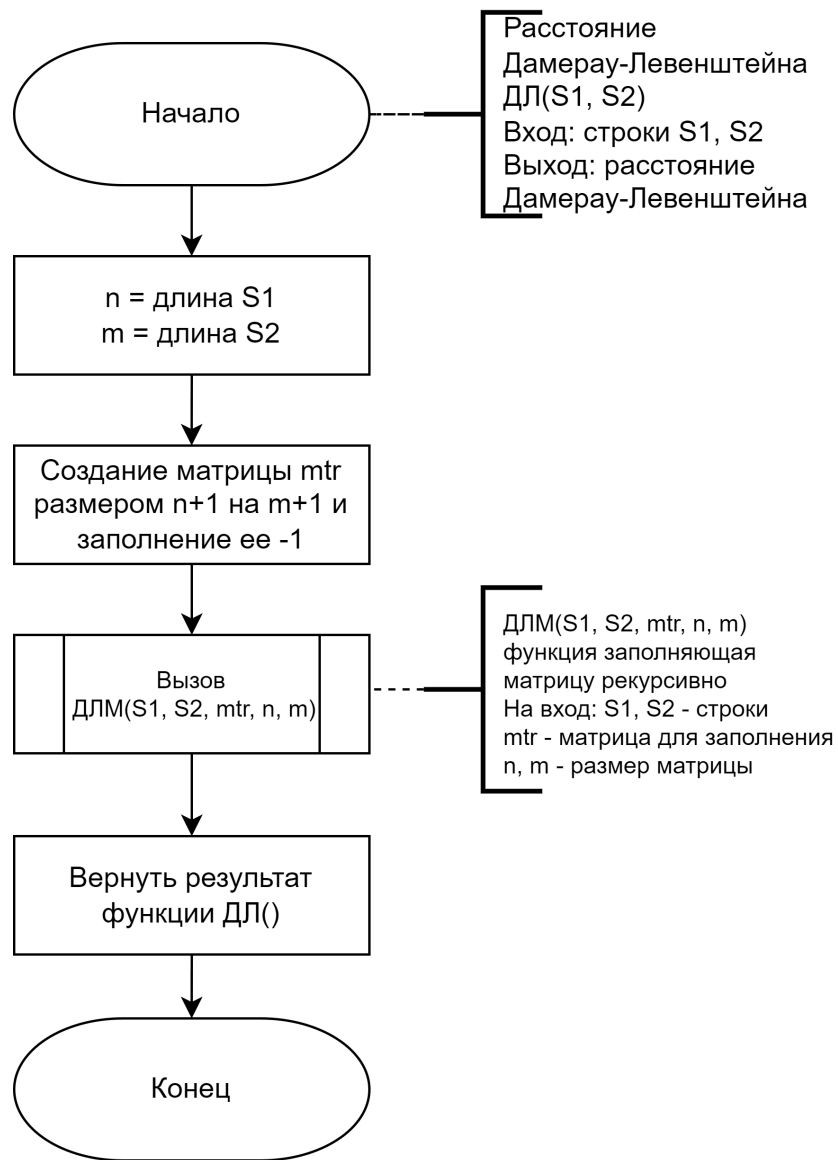


Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна с кешированием

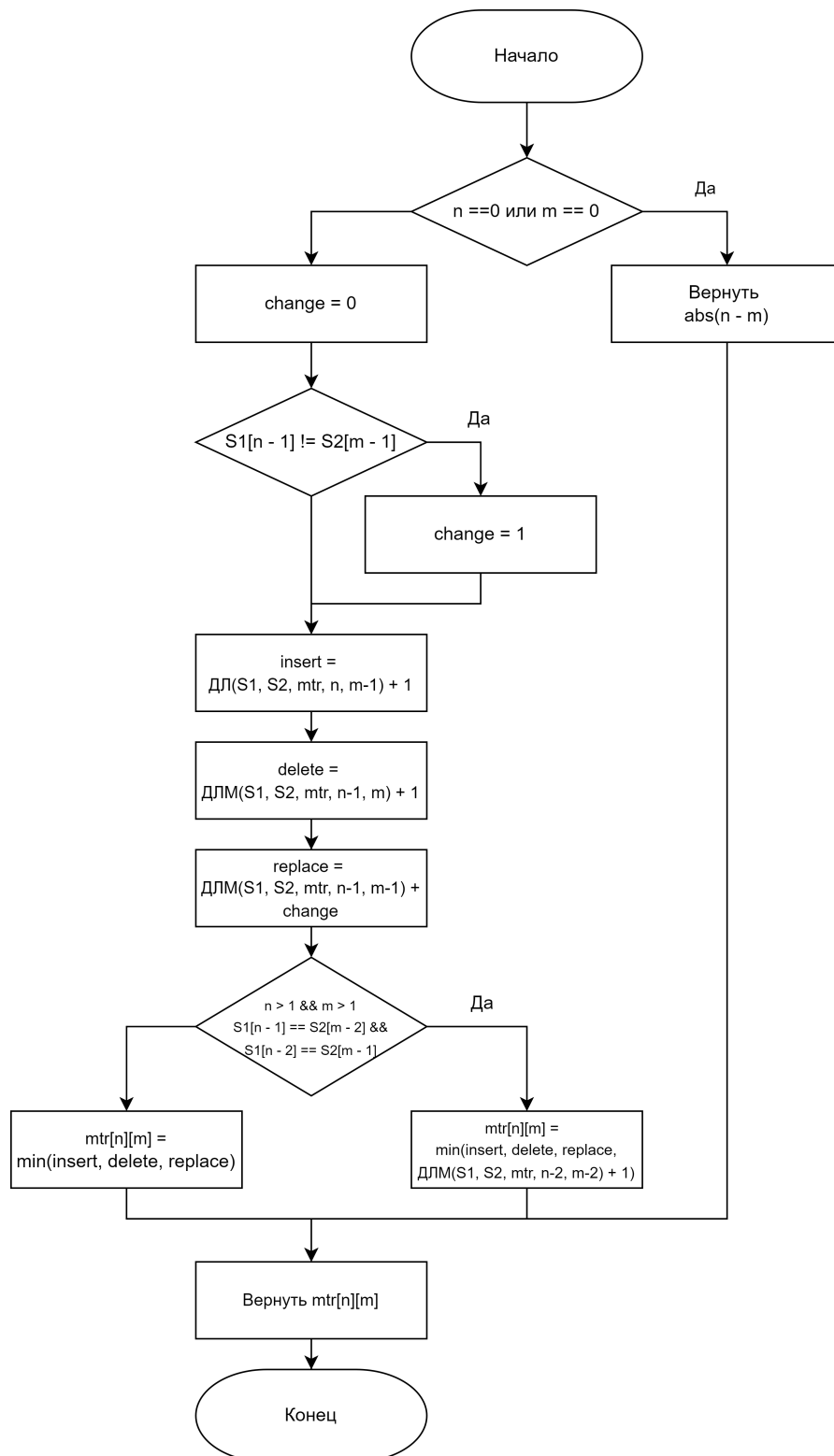


Рисунок 2.5 – Схема алгоритма рекурсивного заполнения матрицы путем поиска расстояния Дамерау-Левенштейна

2.3 Описание используемых типов данных

При реализации алгоритмов будут использованы следующие структуры данных:

- строка - массив типа *char* размером длины строки;
- длина строки - целое число типа *int*;
- матрица - двумерный массив типа *int*.

2.4 Вывод

В данном разделе на основе теоретических данных были построены схемы требуемых алгоритмов, выбраны используемые типы данных.

3 Технологическая часть

В данном разделе будут приведены требования к программному обеспечению, средства реализации, листинг кода и функциональные тесты.

3.1 Требования к программному обеспечению

К программе предъявлены ряд требований:

- должна иметь интерфейс для выбора действий;
- должна обрабатывать строки;
- должна хранить строку как на латинице, так и на кириллице;
- должна замерять процессорное время алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

3.2 Средства реализации

В качестве языка программирования для реализации данной лабораторной работы был выбран язык *C++* [2]. Данный выбор обусловлен тем, что я имею некоторый опыт разработки на нем, а так же наличием у языка встроенны библиотеки измерения процессорного времени и тип данных работающий как с кириллицей, так и с латиницей — *std::wstring*, которое соответствуют выдвинутым техническим требованиям.

Время работы было замерено с помощью функции *clock()* из библиотеки *ctime*[3]

3.3 Сведения о модулях программы

Данная программа разбита на следующие модули:

- *main.cpp* – Файл, содержащий точку входа в программу, из которой происходит вызов алгоритмов по разработанному интерфейсу;

- `algorithms.cpp` — Файл содержит функции поиска расстояния Левенштейна и Дamerau-Левенштейна.
- `allocate.cpp` — Файл содержит функции динамического выделения и очищения памяти для матрицы.
- `print_mtr_lev.cpp` — Файл содержит функцию вывода матрицы для итеративных алгоритмов поиска расстояния Левенштейна и Дamerau-Левенштейна, включая строки.
- `cru_time.cpp` — Файл содержит функции, измеряющее процессорное время алгоритмов поиска расстояния Левенштейна и Дamerau-Левенштейна.
- `memory.cpp` — Файл содержит функции, измеряющее память итеративного и рекурсивного алгоритмов поиска расстояния Левенштейна.

Листинг 3.1 – Функция нахождения расстояния Левенштейна с
использованием матрицы

```
1  int lev_mtr(wstring &str1, wstring &str2, bool print ) {
2      size_t n = str1.length();
3      size_t m = str2.length();
4      int **mtr = malloc_mtr(n + 1, m + 1);
5      int res = 0;
6      for (int i = 0; i <= n; i++)
7          for (int j = 0; j <= m; j++)
8              if (i == 0 && j == 0)
9                  mtr[i][j] = 0;
10             else if (i > 0 && j == 0)
11                 mtr[i][j] = i;
12             else if (j > 0 && i == 0)
13                 mtr[i][j] = j;
14             else {
15                 int change = 0;
16                 if (str1[i - 1] != str2[j - 1])
17                     change = 1;
18
19                 mtr[i][j] = std::min(mtr[i][j - 1] + 1,
20                                     std::min(mtr[i - 1][j] + 1,
21                                                 mtr[i - 1][j - 1] + change));
22             }
23             if (print)
24                 print_mtr_lev(str1, str2, mtr, n, m);
25             res = mtr[n][m];
26             free_mtr(mtr, n);
27             return res;
28 }
```

Листинг 3.2 – Функция нахождения расстояния Дameraу-Левенштейна с использованием матрицы

```
1  int dameray_lev_mtr(wstring &str1, wstring &str2, bool print)
2  {
3      size_t n = str1.length();
4      size_t m = str2.length();
5      int **mtr = malloc_mtr(n + 1, m + 1);
6      int res = 0;
7
8      for (int i = 0; i <= n; i++)
9      for (int j = 0; j <= m; j++) {
10         if (i == 0 && j == 0)
11             mtr[i][j] = 0;
12         else if (i > 0 && j == 0)
13             mtr[i][j] = i;
14         else if (j > 0 && i == 0)
15             mtr[i][j] = j;
16         else {
17             int change = 0;
18             if (str1[i - 1] != str2[j - 1])
19                 change = 1;
20
21             mtr[i][j] = min(mtr[i][j - 1] + 1,
22                             min(mtr[i - 1][j] + 1,
23                                 mtr[i - 1][j - 1] + change));
24
25             if (i > 1 && j > 1 &&
26                 str1[i - 1] == str2[j - 2] &&
27                 str1[i - 2] == str2[j - 1])
28                 mtr[i][j] = min(mtr[i][j], mtr[i - 2][j - 2] + 1);
29         }
30     }
31
32     if (print)
33         print_mtr_lev(str1, str2, mtr, n, m);
34     res = mtr[n][m];
35     free_mtr(mtr, n);
36
37     return res;
38 }
```

Листинг 3.3 – Функция нахождения расстояния Дамерау-Левенштейна
рекурсивно

```
1  int damerau_lev_rec_t(wstring &str1, wstring &str2, size_t n,  
   size_t m) {  
2      if (n == 0)  
3          return m;  
4      if (m == 0)  
5          return n;  
6  
7      int change = 0;  
8      int res = 0;  
9      if (str1[n - 1] != str2[m - 1])  
10         change = 1;  
11  
12         res = min(damerau_lev_rec_t(str1, str2, n, m - 1) + 1,  
13 min(damerau_lev_rec_t(str1, str2, n - 1, m) + 1,  
14 damerau_lev_rec_t(str1, str2, n - 1, m - 1) + change));  
15  
16         if (n > 1 && m > 1 &&  
17 str1[n - 1] == str2[m - 2] &&  
18 str1[n - 2] == str2[m - 1])  
19             res = std::min(res, damerau_lev_rec_t(str1, str2, n - 2, m  
   - 2) + 1);  
20         return res;  
21     }  
22  
23     int damerau_lev_rec(wstring &str1, wstring &str2)  
24     {  
25         return damerau_lev_rec_t(str1, str2, str1.length(),  
   str2.length());  
26     }
```

Листинг 3.4 – Функция нахождения расстояния Дамерау-Левенштейна
рекурсивно с кешированием

```
1  int damerau_lev_rec_hash_t(wstring &str1, wstring &str2, int
   **mtr, size_t n, size_t m)
2  {
3      if (n == 0)
4          return mtr[n][m] = m;
5      if (m == 0)
6          return mtr[n][m] = n;
7      int change = 0;
8      if (str1[n - 1] != str2[m - 1])
9          change = 1;
10     mtr[n][m] = min(damerau_lev_rec_hash_t(str1, str2, mtr, n,
        m - 1) + 1,
11     min(damerau_lev_rec_hash_t(str1, str2, mtr, n - 1, m) + 1,
12     damerau_lev_rec_hash_t(str1, str2, mtr, n - 1, m - 1) +
        change));
13     if (n > 1 && m > 1 &&
14     str1[n - 1] == str2[m - 2] &&
15     str1[n - 2] == str2[m - 1])
16         mtr[n][m] = min(mtr[n][m], damerau_lev_rec_hash_t(str1,
            str2, mtr, n - 2, m - 2) + 1);
17     return mtr[n][m];
18 }
19
20 int damerau_lev_rec_hash(wstring &str1, wstring &str2, bool
    print)
21 {
22     size_t n = str1.length();
23     size_t m = str2.length();
24     int **mtr = malloc_mtr(n + 1, m + 1);
25     for (int i = 0; i <= n; i++)
26         for (int j = 0; j <= m; j++) {
27             mtr[i][j] = -1;
28         }
29     int res = damerau_lev_rec_hash_t(str1, str2, mtr, n, m);
30     if (print)
31         print_mtr_lev(str1, str2, mtr, n, m);
32     free_mtr(mtr, n);
33     return res;
34 }
```

Листинг 3.5 – Функции динамического выделения и очищения памяти под матрицу

```
1  void free_mtr(int **mtr, std::size_t n) {
2      if (mtr != nullptr)
3      {
4          for (std::size_t i = 0; i < n; i++)
5              if (mtr[i] != nullptr)
6                  free(mtr[i]);
7          free(mtr);
8      }
9  }
10
11 int **malloc_mtr(std::size_t n, std::size_t m)
12 {
13     if (n == 0)
14         return nullptr;
15
16     int **mtr = static_cast<int **>(malloc(n * sizeof(int *)));
17     if (mtr != nullptr)
18         for (std::size_t i = 0; mtr[i] != nullptr && i < n; i++) {
19             mtr[i] = static_cast<int *>(malloc(m * sizeof(int)));
20             if (mtr[i] == nullptr)
21                 free_mtr(mtr, n);
22         }
23
24     return mtr;
25 }
```

Листинг 3.6 – Функции вывода матрицы для алгоритмов поиска расстояния
Левенштейна и Дамерау-Левенштейна

```
1  void print_mtr_lev(std::wstring str1, std::wstring str2,
2  int **mtr, std::size_t n, std::size_t m)
3  {
4      for(std::size_t i = 0; i <= n + 1; i++)
5      {
6          for(std::size_t j = 0; j <= m + 1; j++)
7          {
8              if (i == 0 && j == 0)
9                  std::wcout << "  ";
10             else if (i == 0)
11                 if (j == 1)
12                     std::wcout << "- ";
13             else
14                 std::wcout << str2[j - 2] << " ";
15             else if (j == 0)
16                 if (i == 1)
17                     std::wcout << "- ";
18             else
19                 std::wcout << str1[i - 2] << " ";
20             else
21                 std::wcout << mtr[i - 1][j - 1] << " ";
22         }
23         std::wcout << std::endl;
24     }
25 }
```

3.4 Функциональные тесты

В таблице 3.1 приведены функциональные тесты для алгоритмов вычисления расстояния Левенштейна и Дameraу—Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 – Функциональные тесты

Строка 1	Строка 2	Левенштейн	Дameraу-Левенштейн		
		Итеративный	Итеративный	Рекурсивный	
				Без кэша	С кэшем
а	Ь	1	1	1	1
а	а	0	0	0	0
кот	скат	2	2	2	2
друзья	рдузия	3	2	2	2
вагон	гонки	4	4	4	4
бар	раб	2	2	2	2
слон	слоны	1	1	1	1

3.5 Вывод

Были реализованы алгоритмы: поиска расстояния Левенштейна итеративно, а также поиска расстояния Дameraу– Левенштейна итеративно, рекурсивно и рекурсивного заполнением кеширования. Проведено тестирование разработанных алгоритмов на функциональных тестах.

4 Исследовательская часть

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялись замеры по времени представлены далее.

- Процессор: Intel(R) Core(TM) i5-10300H CPU @ 2.50 ГГц 2.50 ГГц. [4]
- Оперативная память: 16 ГБ.
- Операционная система: Windows 10 Pro 64-разрядная система версией 21H2. [5]

При замерах времени ноутбук был включен в сеть электропитания и был нагружен только системными приложениями.

4.2 Демонстрация работы программы

```

Выберите алгоритм поиска расстояния:
1 - 1) Нерекурсивный алгоритм Левенштейна;
   - 2) Нерекурсивный алгоритм Дамерау-Левенштейна;
   - 3) Рекурсивный алгоритм Дамерау-Левенштейна без кэша;
   - 4) Рекурсивный алгоритм Дамерау-Левенштейна с кэшем;
2 - Замерить время и память.

Выбор: 1
Введите первое слово: дурья
Введите второе слово: узяня
Минимальное количество операций:
- р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 2 2 3 4 5
у 3 2 2 2 3 4 5
з 4 3 3 2 3 4
ь 5 4 4 4 3 3 4
я 6 5 5 5 4 4 3
Нерекурсивный алгоритм Левенштейна: 3
- р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 1 2 3 4 5
у 3 2 2 1 2 3 4
з 4 3 3 2 1 2 3
ь 5 4 4 3 2 2 3
я 6 5 5 4 3 3 2
Нерекурсивный алгоритм Дамерау-Левенштейна: 2
Рекурсивный алгоритм Дамерау-Левенштейна без кэша: 2
- р д у з и я
- 0 1 2 3 4 5 6
д 1 1 1 2 3 4 5
р 2 1 1 2 3 4 5
у 3 2 2 1 2 3 4
з 4 3 3 2 1 2 3
ь 5 4 4 3 2 2 3
я 6 5 5 4 3 3 2
Рекурсивный алгоритм Дамерау-Левенштейна с кэшем: 2

```

Рисунок 4.1 – Демонстрация работы программы при поиске расстояний Левенштейна и Дамерау-Левенштейна

4.3 Временные характеристики

Результаты эксперимента замеров по времени приведены в Таблице 4.1. В данной таблице для значений. Значения для которых замеры по времени не выполнялись, в поле таблицы обозначены "-" это обусловлено тем для рекурсивной реализации алгоритмов при больших длин строк, алгоритм работает довольно долго.

Таблица 4.1 – Замер по времени для строк, размер которых от 1 до 200

Длина (символ)	Время, нс			
	Левенштейн	Дамерау-Левенштейн		
	Итеративный	Итеративный	Рекурсивный	
			Без кэша	С кэшем
1	0	0	0	0
2	0	0	0	0
3	0	0	0	0
4	0	0	0	0
5	0	15.656	15.656	15.656
6	0	0	62.625	62.625
7	15.656	15.656	328.78	391.41
8	0	0	1706.5	2004
9	15.656	15.656	9393.8	11288
10	15.656	15.656	51932	61545
20	15.656	15.656	-	-
30	31.313	15.656	-	-
40	31.313	31.313	-	-
50	31.313	46.969	-	-
60	62.625	78.282	-	-
70	78.282	78.282	-	-
80	93.938	109.59	-	-
90	109.59	140.91	-	-
100	140.91	172.22	-	-
200	673.22	751.5	-	-

Отдельно сравним итеративные алгоритмы поиска расстояний Левенштейна и Дамерау-Левенштейна. Сравнение будет производиться на основе данных, представленных в Таблице 4.1. Результат можно увидеть на Рисунке 4.2.

При длинах строк менее 30 символов разница по времени между итеративными реализациями незначительна, однако при увеличении длины строки алгоритм поиска расстояния Левенштейна оказывается быстрее вплоть до полутора раз (при длинах строк равных 200). Это обосновывается тем, что у

алгоритма поиска расстояния Дameraу-Левенштейна задействуется дополнительная операция, которая замедляет алгоритм.

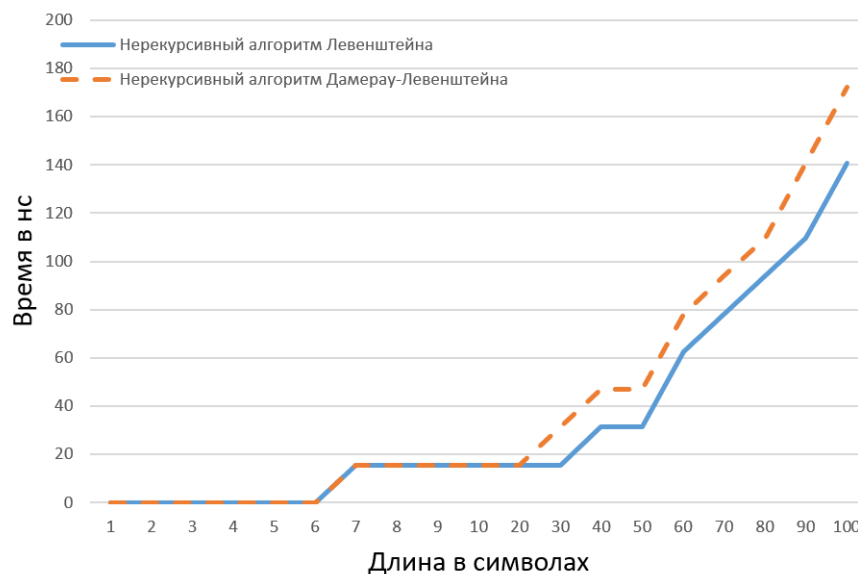


Рисунок 4.2 – Сравнение по времени нерекурсивных реализаций алгоритмов поиска расстояний Левенштейна и Дameraу-Левенштейна

Так же сравним рекурсивную и итеративную реализации алгоритма поиска расстояния Дameraу-Левенштейна. Данные представлены в Таблице 4.1 и отображены на Рисунке 4.3.



Рисунок 4.3 – Сравнение по времени алгоритмов поиска расстояния Дameraу-Левенштейна

На Рисунке 4.3 продемонстрировано, что рекурсивный алгоритм становится менее эффективным (вплоть до 21 раз при длине строк равной 7 элементов), чем итеративный.

Из этого можно сделать вывод о том, что рекурсивный алгоритм неэффективно использовать по времени.

Кроме того, согласно данным, приведенным в Таблице 4.1, рекурсивные алгоритмы при длинах строк более 10 элементов не пригодны к использованию в силу экспоненциально роста затрат процессорного времени, в то время, как затраты итеративных алгоритмов по времени линейны.

4.4 Характеристики по памяти

Введем следующие обозначения:

- n - длина строки S_1 ;
- m - длина строки S_2 ;
- $size()$ - функция вычисляющая размер в байтах;
- $string$ - строковый тип;
- int - целочисленный тип;
- $size_t$ - беззнаковый целочисленный тип.

Максимальная глубина стека вызовов при рекурсивной реализации нахождения расстояния Дамерау-Левенштейна равна сумме входящих строк, а на каждый вызов требуется 4 дополнительных переменных, соответственно, максимальный расход памяти.

$$(n + m) \cdot (2 \cdot size(string) + 3 \cdot size(int) + 2 \cdot sizeof(size_t)) \quad (4.1)$$

, где:

- хранение двух строк - $2 \cdot size(string)$;
- хранение размеров строк - $2 \cdot size(size_t)$;
- дополнительные переменные - $2 \cdot size(int)$;
- адрес возврата.

Для рекурсивного алгоритма Дамерау-Левенштейна будет теоретически тоже самое, что и в рекурсивном, но добавляется еще размер матрицы, соответственно, максимальный расход памяти:

$$(n + m) \cdot (2 \cdot \text{size}(\text{string}) + 3 \cdot \text{size}(\text{int}) + 2 \cdot \text{size}(\text{size_t})) + (n + 1) \cdot (m + 1) \cdot \text{size}(\text{int}) \quad (4.2)$$

Использование памяти при итеративной реализации расстояния Левенштейна теоретически равно:

$$(n + 1) \cdot (m + 1) \cdot \text{size}(\text{int}) + 2 \cdot \text{size}(\text{string}) + 2 \cdot \text{size}(\text{size_t}) + \text{size}(\text{int} **) + (n + 1) \cdot \text{size}(\text{int} *) + 2 \cdot \text{size}(\text{int}) \quad (4.3)$$

, где

- хранение двух строк - $2 \cdot \text{size}(\text{string})$;
- хранение размеров матрицы - $2 \cdot \text{size}(\text{size_t})$;
- хранение матрицы - $(n + 1) \cdot (m + 1) \cdot \text{size}(\text{int})$;
- указатель на матрицу - $\text{size}(\text{int} **) + (n + 1) \cdot \text{size}(\text{int} *)$;
- дополнительные переменные для хранения результата - $\text{size}(\text{int})$;
- адрес возврата.

Использование памяти при итеративной реализации расстояния Дамерау-Левенштейна теоретически равно:

$$(n + 1) \cdot (m + 1) \cdot \text{size}(\text{int}) + 2 \cdot \text{size}(\text{string}) + 2 \cdot \text{size}(\text{size_t}) + \text{size}(\text{int} **) + (n + 1) \cdot \text{size}(\text{int} *) + 3 \cdot \text{size}(\text{int}) \quad (4.4)$$

, где

- хранение двух строк - $2 * \text{size}(\text{string})$;
- хранение размеров матрицы - $2 \cdot \text{size}(\text{size_t})$;
- хранение матрицы - $(n + 1) \cdot (m + 1) \cdot \text{size}(\text{int})$;

- указатель на матрицу - $size(int **) + (n + 1) \cdot size(int*)$;
- дополнительные переменные - $2 \cdot size(int)$;
- адрес возврата.

По расходу памяти итеративные алгоритмы проигрывают рекурсивным: максимальный размер используемой памяти в них растет как произведение длин строк, в то время как у рекурсивного алгоритма — как сумма длин строк.

Таблица 4.2 – Замер памяти для строк, размером от 10 до 200

Длина (символ)	Размер в байтах			
	Левенштейн	Дамерау-Левенштейн		
	Итеративный	Итеративный	Рекурсивный	
			Без кэша	С кэшем
10	668	672	1840	2420
20	2028	2032	3680	5620
30	4188	4192	5520	9620
40	7148	7152	7360	14420
50	10908	10912	9200	20020
60	15468	15472	11040	26420
70	20828	20832	12880	33620
80	26988	26992	14720	41620
90	33948	33952	16560	50420
100	41708	41712	18400	60020
110	50268	50272	20240	70420
120	59628	59632	22080	81620
130	69788	69792	23920	93620
140	80748	80752	25760	106420
150	92508	92512	27600	120020
160	105068	105072	29440	134420
170	118428	118432	31280	149620
180	132588	132592	33120	165620
190	147548	147552	34960	182420
200	163308	163312	36800	200020

Из данных, приведенных в Таблице 4.2, видно, что рекурсивные алгоритмы являются более эффективными по памяти, так как используется только память под локальные переменные, передаваемые аргументы и возвращаемое значение, в то время как итеративные алгоритмы затрачивают память линейно пропорционально длинам обрабатываемых строк.

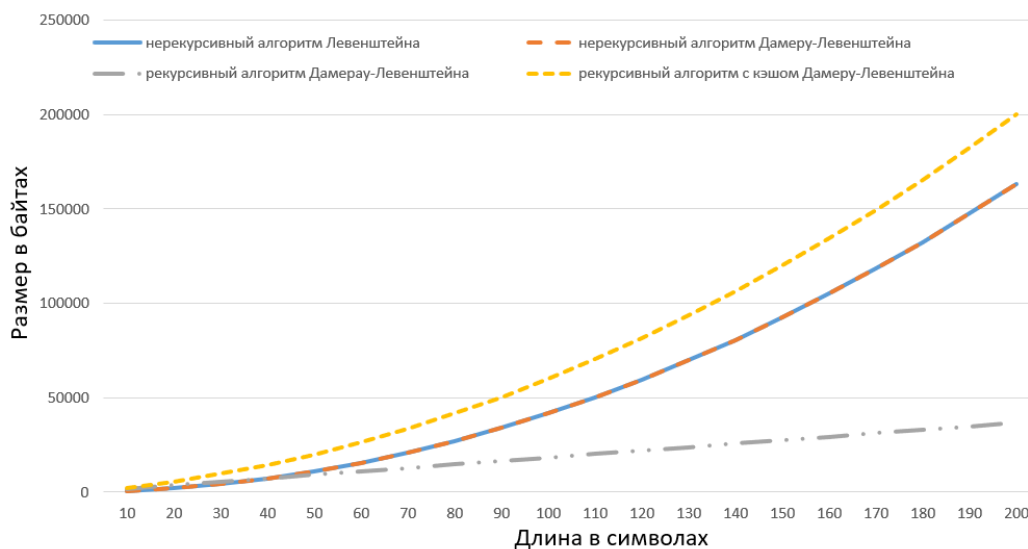


Рисунок 4.4 – Сравнение по памяти алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна — итеративной и рекурсивной реализации

Как видно из Рисунка 4.5, можно увидеть что рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна эффективная по памяти, чем итеративная.

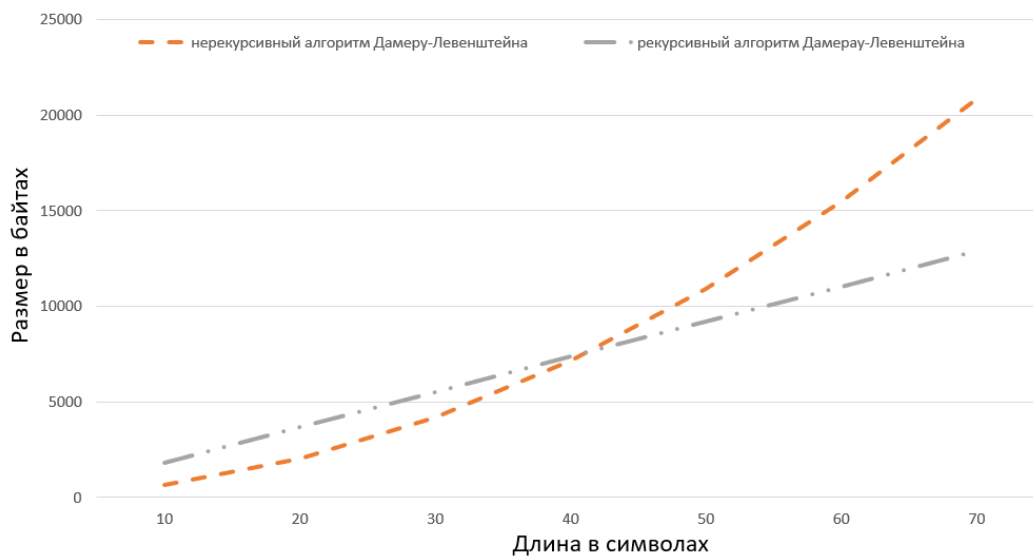


Рисунок 4.5 – Сравнение по памяти алгоритмов поиска расстояния Дамерау-Левенштейна — итеративной и рекурсивной реализации

4.5 Сравнительный анализ алгоритмов

Приведенные характеристики показывают нам, что рекурсивная реализация алгоритма очень сильно проигрывает по времени. В связи с этим, рекурсивные алгоритмы следует использовать лишь для малых размерностей строк (1-4 символа).

Так как во время печати очень часто возникают ошибки связанные с транспозицией букв, алгоритм поиска расстояния Дамерау-Левенштейна является наиболее предпочтительным, не смотря на то, что он проигрывает по времени и памяти алгоритму Левенштейна.

Можно сделать вывод о том, что рекуррентный алгоритм поиска расстояния Дамерау-Левенштейна будет более затратным по времени по сравнению с итеративной реализацией алгоритма поиска расстояния Дамерау-Левенштейна, но менее затратным по памяти по отношению к итеративному алгоритму.

4.6 Вывод

В данном разделе было произведено сравнение количества затраченного времени и памяти вышеизложенных алгоритмов. Наименее затратным по времени оказался итеративный алгоритм нахождения расстояния Левенштейна.

Для обработок малых длин строк (1-4 символа) предпочтительнее использовать рекурсивные алгоритмы, в то время как для остальных случаев рекомендуются использовать итеративные реализации. Однако, стоит учитывать дополнительные затраты по памяти, возникающие при использовании итеративных алгоритмов.

Заключение

В данной лабораторной работе были рассмотрено расстояние Левенштейна. Данное расстояние показывает минимальное количество операций (вставка, удаление, замены), которое необходимо для перевода одной строки в другую. Это расстояние помогает определить схожесть двух строк.

Впервые задачу поставил в 1965 году советский математик Владимир Левенштейн при изучении последовательностей 0-1, впоследствии более общую задачу для произвольного алфавита связали с его именем.

Расстояние Левенштейна применяется в теории информации и компьютерной лингвистике для:

- исправления ошибок в слове (в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
- сравнения текстовых файлов утилитой diff;
- для сравнения генов, хромосом и белков в биоинформатике.

Цели данной лабораторной работы были достигнуты, а именно изучение и исследование особенностей задач динамического программирования на алгоритмах Левенштейна и Дамерау-Левенштейна.

Для поставленной цели были выполнены следующие задачи:

- 1) изучено расстояние Левенштейна и Дамерау-Левенштейна;
- 2) создано ПО, реализующее следующие алгоритмы:
 - нерекурсивный метод поиска расстояния Левенштейна;
 - нерекурсивный метод поиска Дамерау-Левенштейна;
 - рекурсивный метод поиска Дамерау-Левенштейна;
 - рекурсивный с кэшированием метод поиска Дамерау-Левенштейна.
- 3) выбраны инструменты для замера процессорного времени выполнения реализаций алгоритмов;
- 4) Проведены анализ затрат работы программы по времени и по памяти, выяснить влияющие на них характеристики.

Список использованных источников

- [1] Левенштейн В. И. Двоичные коды с исправлением выпадений, вставок и замещений символов. – М.: Издательство «Наука», Доклады АН СССР, 1965. Т. 163. С. 845–848.
- [2] Документация по Microsoft C++ [Электронный ресурс]. Режим доступа: <https://learn.microsoft.com/ru-ru/cpp/?view=msvc-170&viewFallbackFrom=vs-2017> (дата обращения: 25.09.2022).
- [3] C library function clock() [Электронный ресурс]. Режим доступа: https://www.tutorialspoint.com/c_standard_library/c_function_clock.htm (дата обращения: 25.09.2022).
- [4] Intel [Электронный ресурс]. Режим доступа: <https://ark.intel.com/content/www/ru/ru/ark/products/201839/intel-core-i510300h-processor-8m-cache-up-to-4-50-ghz.html> (дата обращения: 25.09.2022).
- [5] Windows 10 Pro 2h21 64-bit [Электронный ресурс]. Режим доступа: <https://www.microsoft.com/ru-ru/software-download/windows10> (дата обращения: 25.09.2022).