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»

Лабораторная работа № 1
По курсу: «Анализ алгоритмов»

Тема: «Расстояния Левенштейна и Дамерау-Левенштейна»

Студент Горячев В. Г.

Группа ИУ7-55Б
Оценка (баллы) _____

Преподаватель _____

Москва.
2020 г.

Оглавление

Введение

Расстояние Левенштейна – это минимальное количество редакторских операций, необходимых для преобразования одной строки в другую. В общем случае каждая операция имеет свою стоимость выполнения, что позволяет учитывать различные условия работы, например, когда стоимость операции удаления вдвое дороже, чем вставки.

Впервые эта задача была упомянута советским математиком В. И. Левенштейном в 1965 г[2]. Расстояние Левенштейна применяется в теории информации, компьютерной лингвистике и, возможно, иных областях. Конкретные примеры:

- исправление ошибок в слове;
- сравнение текстовых файлов – тогда в роли символов выступают целые строки;
- биоинформатика – генетические последовательности можно сравнивать и изучать этим методом, ввиду ограниченности составляющего их «алфавита».

Расстояние Дамерау-Левенштейна имеет минимальные отличия от расстояния Левенштейна: Фредерик Дамерау предложил учитывать дополнительную операцию, которая, применительно к строкам и работе с человеческими запросами, позволяет значительно уменьшить получаемое расстояние.

1. Аналитическая часть

1.1. Цель и задачи

Целью данной лабораторной работы является реализация и сравнение по временной и ёмкостной эффективности разных алгоритмов поиска расстояний Левенштейна и Дамерау-Левенштейна.

Задачи:

1. дать математическое описание расстояний Левенштейна и Дамерау — Левенштейна;
2. разработать алгоритмы поиска расстояний;
3. реализовать алгоритмы поиска расстояний;
4. провести эксперименты по замеру времени работы реализации алгоритмов;
5. провести сравнительный анализ реализации алгоритмов по затраченному времени;
6. дать теоретическую оценку максимальной затрачиваемой реализациями алгоритмов памяти.

1.2. Описание и формулы

Задача поиска расстояния Левенштейна заключается в нахождении минимального количества односимвольных операций для преобразования одной строки в другую.

Такими операциями считаются односимвольные:

- вставка – I (англ. insert);
- удаление – D (англ. delete);
- замена – R (англ. replace);
- совпадение – M (англ. Match, штраф 0).

Сразу можно сделать пояснение про отличие расстояния Дамерау-Левенштейна. Оно учитывается дополнительную операцию – перестановку X (англ. exchange), которая позволяет обнаружить два соседних символа, расположение которых в другой строке отличается только на один обмен между собой. Так, например, для строк «аб» и «ба» потребуется 1 перестановка вместо 2 замен.

Пусть S_1 и S_2 – две строки над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по рекуррентной формуле 1 (в рамках условия лабораторной работы можно пренебречь возможностью задавать особые стоимости операций, поэтому запись формул упрощается).

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

где $m(a,b)$ равна нулю, если $a = b$, и единице – в противном случае.

Пример нахождения расстояния Левенштейна с помощью таблицы приведен в табл. 1.1.

Таблица 1. 1. Пример вычисления расстояния Левенштейна с помощью матрицы

		М	А	Р	С
	0	1	2	3	4
М	1	0	1	2	3
О	2	1	1	2	3
Д	3	2	2	2	3

Таким образом, расстояние равно 3.

Расстояние Дамерау-Левенштейна вычисляется по рекуррентной формуле 2.

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

Вывод

В этом разделе были сформулированы цели и задачи работы, а также математическая сторона задачи поиска расстояния Левенштейна. Используя полученные формулы, можно сделать переход к дальнейшей разработке реализаций изучаемого алгоритма.

2. Конструкторская часть

Программа предъявляет следующие требования к вводу:

1. на вход подаются две строки;
2. заглавные и строчные буквы считаются разными.

2.1. Требования к программному обеспечению

На ввод подаются две строки произвольной длины.

Ввод двух пустых строк должен считаться корректным вводом, программа не должна завершаться аварийно.

Программа должна выводить результаты вычислений, а также дополнительную информацию, такую, как измерение времени работы.

2.2. Схемы алгоритмов

В данном пункте будут рассмотрены схемы алгоритмов (рис. 2.1–2.4).

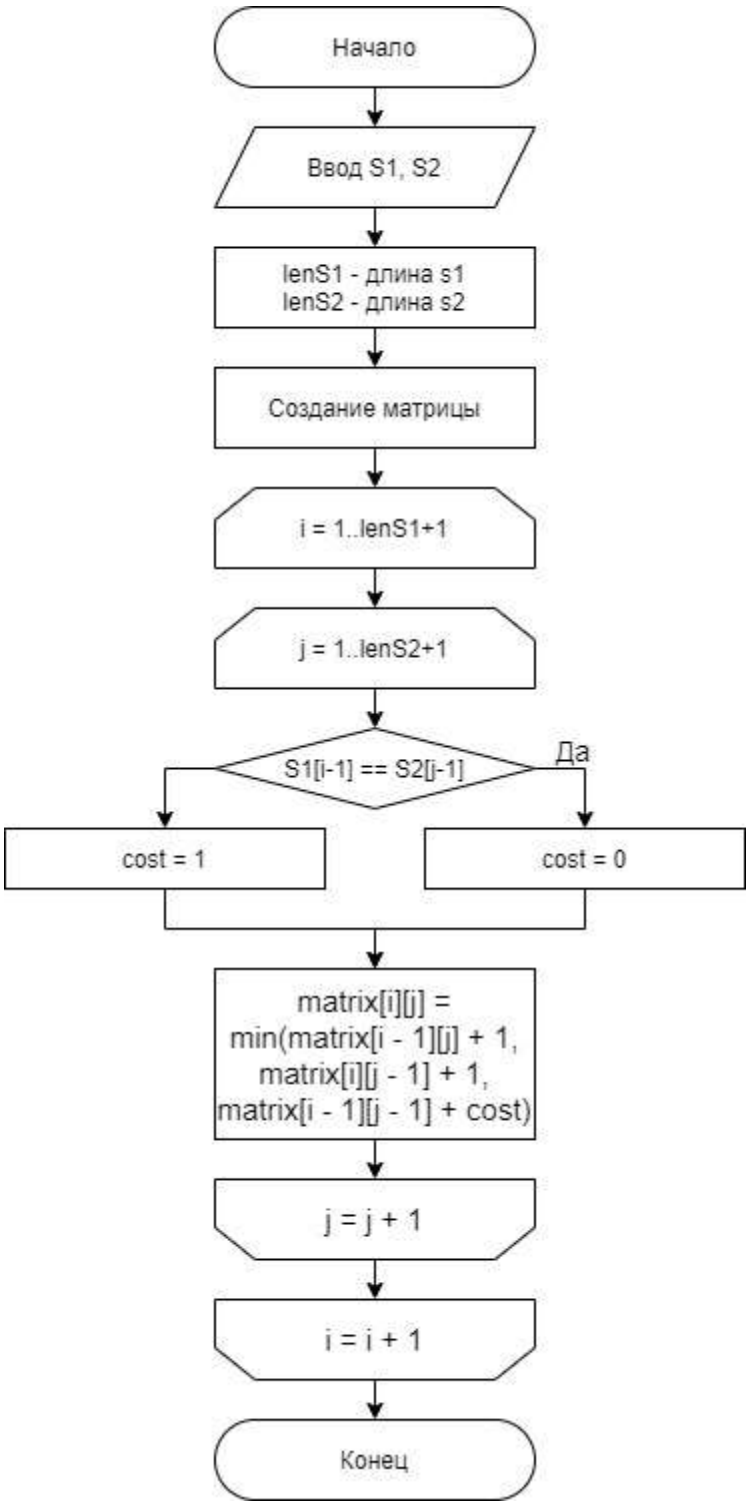

Рис. 2.1: Схема матричного алгоритма нахождения расстояния Левенштейна

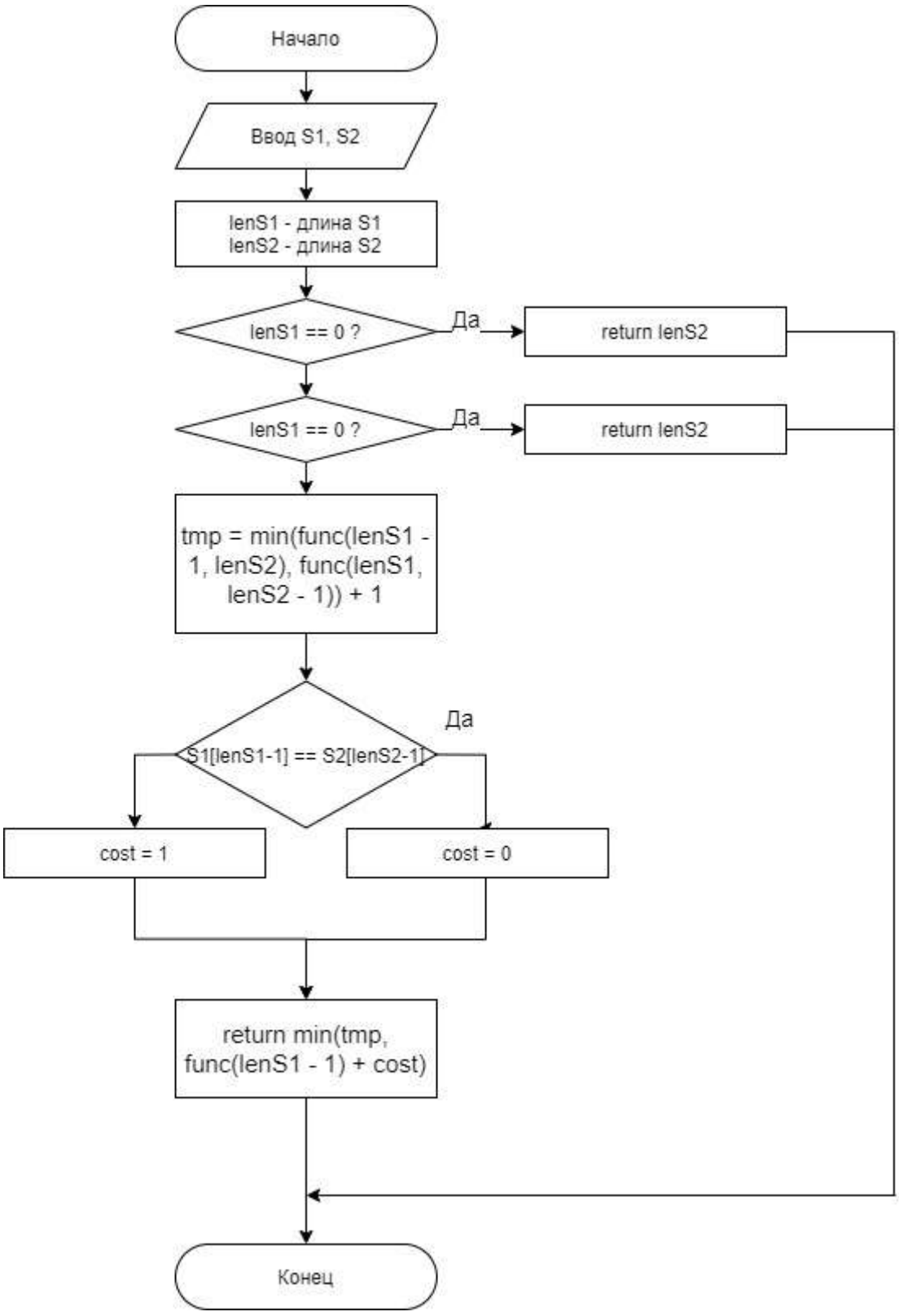

Рис. 2.2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

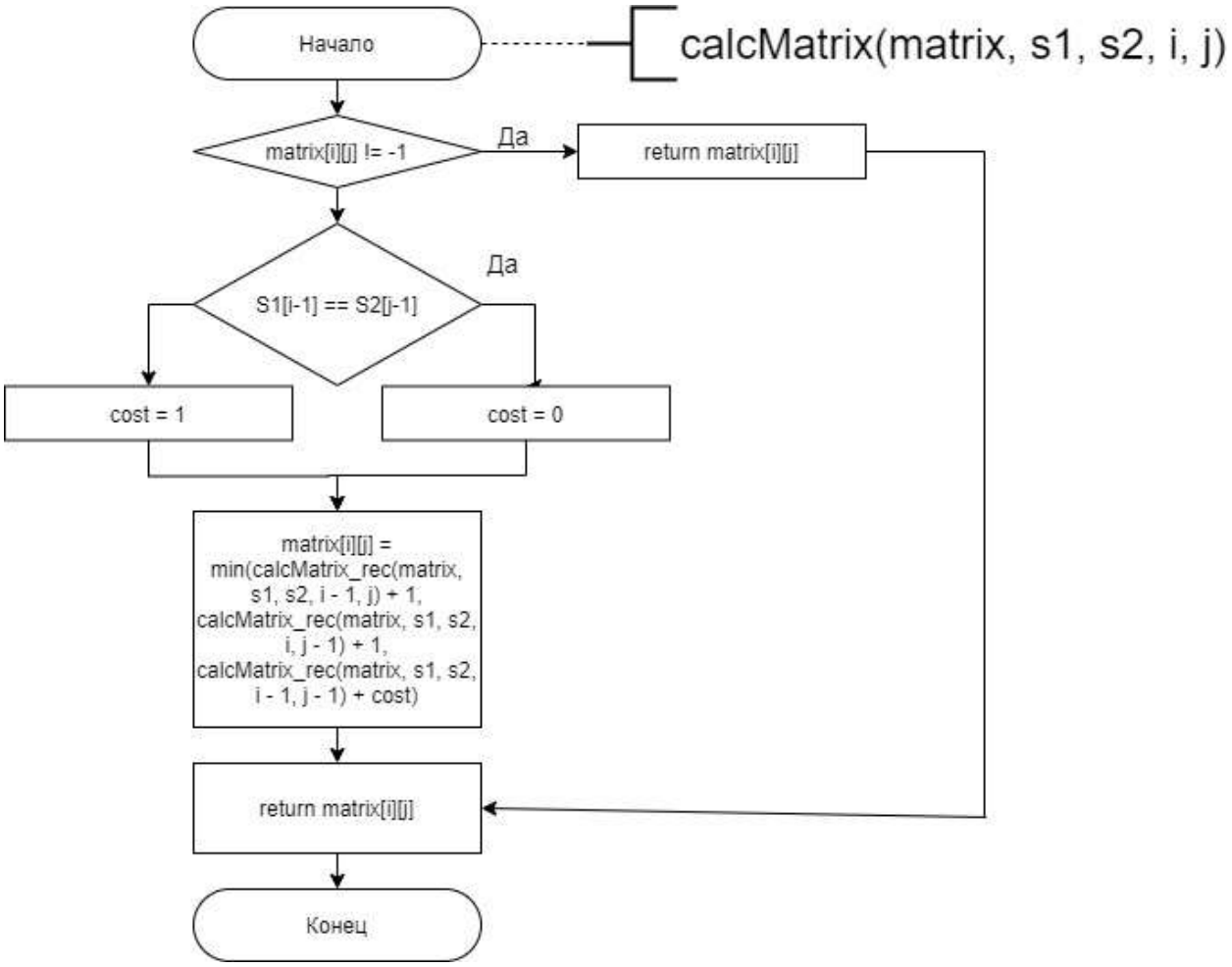

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с матрицей

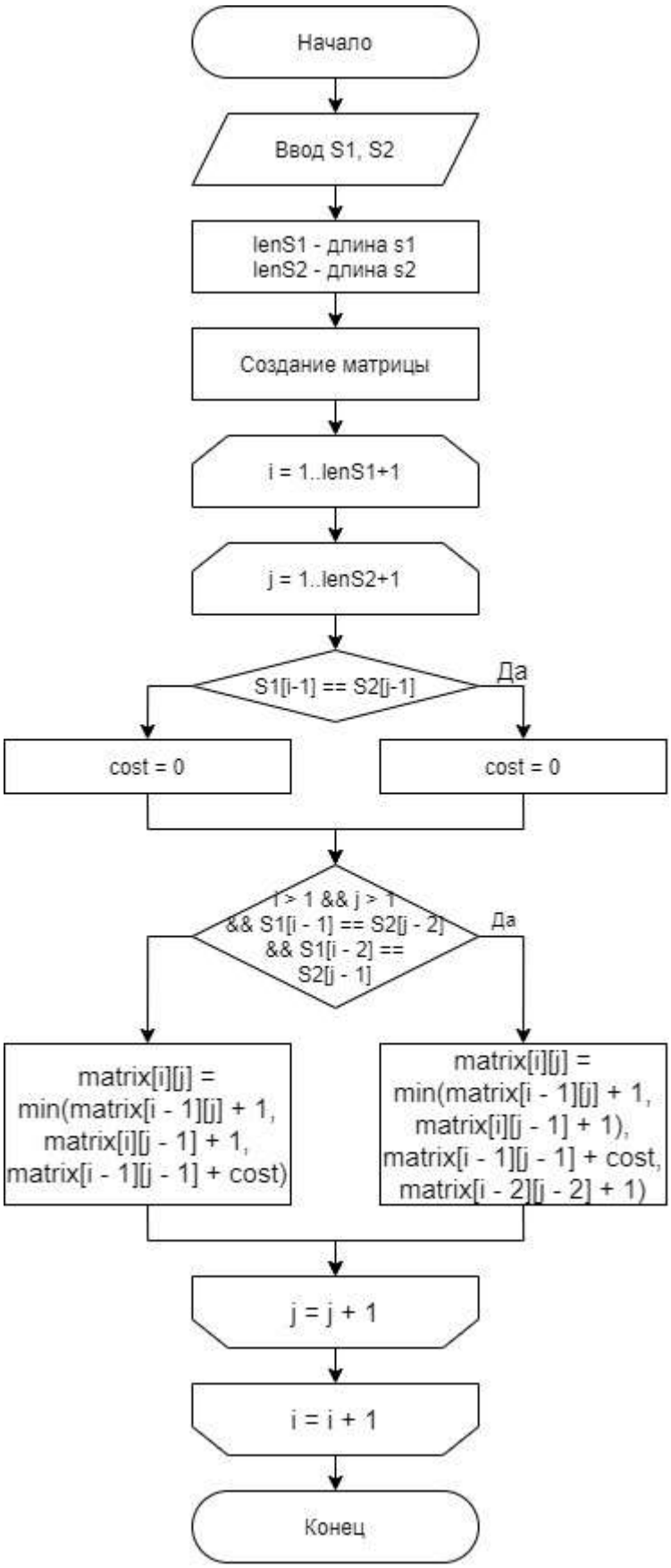

Рис. 2.4: Схема матричного алгоритма нахождения расстояния Дамерау-Левенштейна

Вывод

В ходе конструкторской части были сформулированы требования к программе и разработаны схемы алгоритмов, с помощью которых можно осуществить написание программы.

3. Технологическая часть

3.1. Выбор языка программирования

Для написания программных реализаций алгоритмов был выбран язык программирования C++, ввиду изучения оного на прошлых курсах, наличия некоторых элементов программы (реализация универсальной матрицы).

Среда разработки – Visual Studio 2019.

3.2. Реализация алгоритмов

Тип **Word** в приводимых ниже листингах – псевдоним для целочисленного типа short int, размер которого составляет 2 байта.

Основным элементом каждого варианта алгоритма является выбор минимума из трёх чисел, добавление стоимости шага и проверка двух символов на равенство. Поэтому будет правильно вынести эту повторяющуюся часть кода в отдельную функцию, реализация которой представлена в листинге 3.1.

```
inline Word minLevenstein(Word a, Word b, Word c, char& s1, char& s2) {
   return (Word) min(min(a, b) + 1, c + isNotEqual(s1, s2));
}
```

Листинг 3.1: Функция выбора минимального шага

Первый вариант реализации алгоритма Левенштейна основан на использовании матрицы для вычислений в ходе итеративного процесса. Код представлен в листинге 3.2.

```
Word findDistLev_matrix(string& word_vert, string& word_hor, bool print_flag)
{
   if (word_vert.length() == 0 || word_hor.length() == 0)
      return max(word_vert.length(), word_hor.length());

   size_t n = word_vert.length() + 1, m = word_hor.length() + 1;
   Matrix<Word> cost_matrix(n, m); {
      size_t i = 0;
      for (; i < min(n, m); cost_matrix[0][i] = cost_matrix[i][0] = i++);
      if (n < m) for (; i < m; cost_matrix[0][i] = i++);
      else for (; i < n; cost_matrix[i][0] = i++);
   }

   for (size_t i = 1; i < n; ++i)
      for (size_t j = 1; j < m; ++j)
         cost_matrix[i][j] = minLevenstein(cost_matrix[i][j - 1],
            cost_matrix[i - 1][j], cost_matrix[i - 1][j - 1],
               word_vert[i - 1], word_hor[j - 1]);
```

```
    if (print_flag)
        print_table(cost_matrix, word_vert, word_hor);

    return cost_matrix[n - 1][m - 1];
}
```

Листинг 3.2: Функция нахождения расстояния Левенштейна с матрицей

В листинге 3.3 представлен рекурсивный вариант реализации алгоритма Левенштейна.

```
Word findDistLev_rec(string& s1, string& s2, size_t n1, size_t n2) {
    if (n1 == 0) return n2;
    if (n2 == 0) return n1;

    return minLevenstein(findDistLev_rec(s1, s2, n1 - 1, n2), findDistLev_rec(s1,
s2, n1, n2 - 1),
        findDistLev_rec(s1, s2, n1 - 1, n2 - 1), s1[n1 - 1], s2[n2 - 1]);
}
```

Листинг 3.3: Функция нахождения расстояния Левенштейна рекурсивная

Последний вариант реализации алгоритма Левенштейна основан на использовании матрицы для вычислений, но процесс по своей сути рекурсивный. Матрица используется для того, чтобы избежать затрат времени на обработку повторяющихся вызовов. Код представлен в листинге 3.4.

```
Word findDistLev_rec_matr(string& s1, string& s2, bool print_flag) {
    size_t n = s1.length() + 1, m = s2.length() + 1;
    Matrix<Word> cost_matrix(n, m);
    for (size_t i = 0; i < n; ++i)
        for (size_t j = 0; j < m; ++j)
            cost_matrix[i][j] = (i && j) ? -1 : (i + j);

    calcMatr_rec(cost_matrix, s1, s2, s1.length(), s2.length());

    if (print_flag)
        print_table(cost_matrix, s1, s2);

    return cost_matrix[n - 1][m - 1];
}
Word calcMatr_rec(Matrix<Word>& matrix, string& s1, string& s2, size_t i,
size_t j)
```

```
{
   if (matrix[i][j] != -1)  return matrix[i][j];

   matrix[i][j] = minLevenstein(calcMatr_rec(matrix, s1, s2, i - 1, j),
calcMatr_rec(matrix, s1, s2, i, j - 1),
      calcMatr_rec(matrix, s1, s2, i - 1, j - 1), s1[i - 1], s2[j - 1]);
   return matrix[i][j];
}
```

Листинг 3.4: Функция нахождения расстояния Левенштейна рекурсивная с матрицей вместе с логикой, вынесенной в отдельную функцию

Матричная реализация алгоритма представлена в листинге 3.5.

```
Word findDistDamLev_matrix(string& s1, string& s2, bool print_flag) {
   if (s1.length() == 0 || s2.length() == 0)
      return max(s1.length(), s2.length());

   size_t n = s1.length() + 1, m = s2.length() + 1;
   Matrix<Word> cost_matrix(n, m); {
      size_t i = 0;
      for (; i < min(n, m); cost_matrix[0][i] = cost_matrix[i][0] = i++);
      if (n < m) for (; i < m; cost_matrix[0][i] = i++);
      else for (; i < n; cost_matrix[i][0] = i++);
   }

   for (size_t i = 1; i < n; ++i)
      for (size_t j = 1; j < m; ++j) {
         cost_matrix[i][j] = minLevenstein(cost_matrix[i][j - 1],
            cost_matrix[i - 1][j], cost_matrix[i - 1][j - 1],
            s1[i - 1], s2[j - 1]);
         if ((i > 1 && j > 1) && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2[j - 1])
            cost_matrix[i][j] = min(cost_matrix[i][j], (Word) (cost_matrix[i - 2][j -
2] + 1));
      }

   if (print_flag)
      print_table(cost_matrix, s1, s2);

   return cost_matrix[n - 1][m - 1];
}
```

Листинг 3.5: Функция нахождения расстояния Дамерау-Левенштейна с матрицей

3.3. Тестирование

Формат и количество вводимых данных: две строки. Проводится по принципу чёрного ящика.

В таблице отображён основной результат работы алгоритмов, общий для всех вариаций. Помимо расстояния между строками программа выводит матрицы и дополнительную информацию.

Результаты тестирования группы реализаций алгоритмов относятся к поиску расстояния Левенштейна, они представлены в табл. 3.1.

Таблица 3.1. Результаты работы реализаций алгоритма поиска расстояний Левенштейна

Вводимые данные	**Результат**
Две пустых строки	0
“kit”, “skat”	2
“skat”, “kit”	2
“kit”, “”	3
“”, “skat”	4

Результаты тестирования реализации алгоритма поиска расстояния Дамерау-Левенштейна представлены в табл. 3.2.

Таблица 3.2. Результаты работы реализации алгоритма поиска расстояния Дамерау-Левенштейна

Вводимые данные	**Результат**
Две пустых строки	0
“kit”, “skat”	2
“skat”, “kit”	2
“kit”, “”	3
“”, “skat”	4
“ab”, “ba”	1 – у алгоритма поиска расстояния Дамерау-Левенштейна, 2 – у остальных

Все результаты работы программы соответствуют ожидаемым результатам.

Вывод

В ходе технологической части были успешно реализованы и протестированы все варианты алгоритма поиска расстояния Левенштейна и Дамерау-Левенштейна, что позволит произвести измерения времени работы и перейти к следующей части.

4. Исследовательская часть

4.1. Сравнительный анализ на основе замеров времени работы алгоритмов

Для удобства берутся два слова одинаковой длины. Для замеров быстродействия проводится 1000 повторных вызовов функций.

Быстродействие алгоритмов измеряется в процессорном времени, которое более точно отражает время работы алгоритмов, с помощью функции _rdtsc()[1].

Полученные данные представлены в табл. 4.1.

Таблица 4.1. Результаты тестовых измерений скорости работы алгоритма (в тактах процессора) в зависимости от длины слов

Длина	Л. матрица	Л. рекурсив.	Л. матр+рек	Д.-Л. матр.
1	5251	94	5648	5235
2	8348	475	9622	8353
3	11949	2347	14987	12208
4	18450	13984	24773	19265
5	22035	63979	30845	23122
6	28841	354007	41639	30703
7	34310	1890262	52074	35620
8	43068	10318472	66878	44683

Можно проиллюстрировать табличные данные графиком (рис. 4.1):

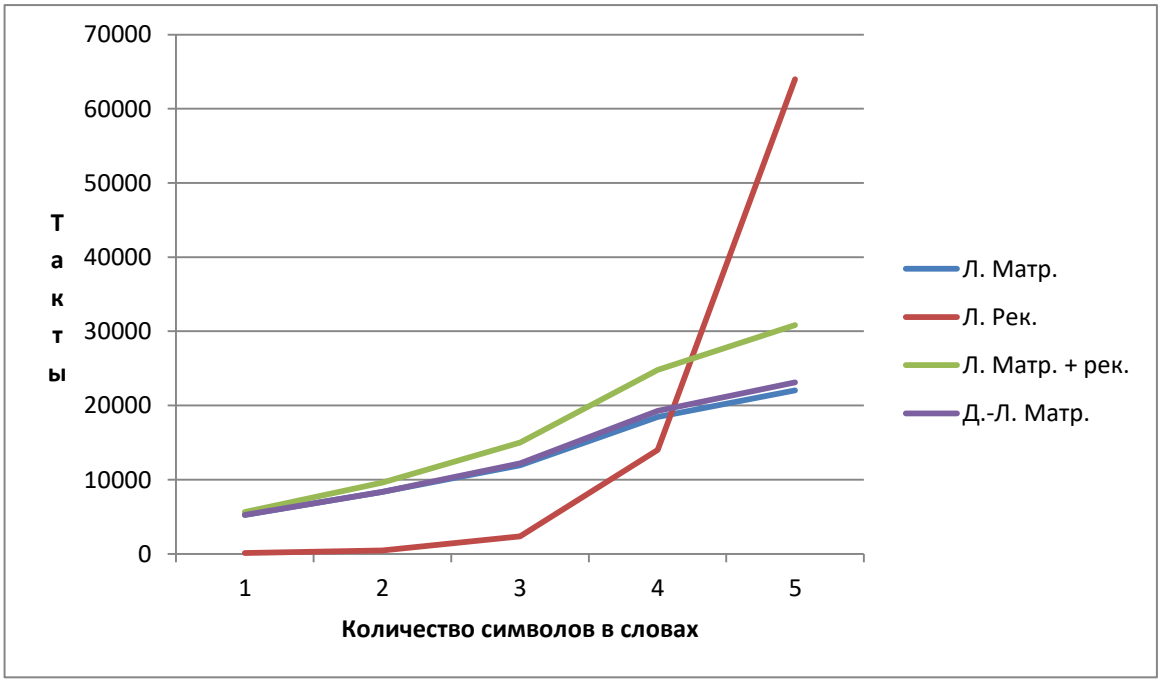

Рис. 4.1: Измерения времени работы реализаций алгоритмов

Поскольку функция времени рекурсивной реализации растёт очень близко, график приводится лишь до 5 символов в слове. Видно, что до 4 символов рекурсивная реализация показывает себя эффективнее или не хуже по времени, чем другие варианты, однако потом затрачиваемое на обработку время растёт экспоненциально. Рекурсивная реализация с матрицей также требует больше времени, чем прочие, однако скорость роста этой функции сопоставима с функциями времени матричной реализации и матричной реализации алгоритма поиска расстояний Дамерау-Левенштейна, которые требуют время на создание и инициализацию матриц.

4.2. Сравнительный анализ на основе теоретической оценки используемой памяти

Исследуемые четыре алгоритма можно разбить на три группы, чтобы рассчитать используемую ими память.

Во всех расчётах n и m – длина первой и второй строк соответственно. Расчёты проводятся для 64-разрядной версии операционной системы.

В первую группу можно определить алгоритмы Левенштейна и Дамерау-Левенштейна с матрицей. Используемая ими память (в байтах) может быть описана формулами 4.1 и 4.3, а рассчитана в формулах 4.2 и 4.4.

$$\text{указатель 1} + \text{указатель 2} + bool + \text{адрес возврата} + \text{результат} = \text{стоимость вызова функции} \quad (4.1)$$

$$8 + 8 + 1 + 8 + 2 = 27 \quad (4.2)$$

$$\text{размер } n + \text{размер } m + \text{указатель} + (n+1) * (m+1) * \text{размер типа} + (n+1) * \text{размер указателя} = \text{размер матрицы в памяти} \quad (4.3)$$

$8 + 8 + 8 + (n+1) * (m+1) * 2 + (n+1) * 8 = 32 + 2 * (n+1)(m+1) + 8 * n$ (4.4) – основные затраты памяти приходятся именно на хранимую матрицу. Возможны дополнительные незначительные расходы памяти на организацию циклов и вызов вспомогательных функций, но, ввиду нерекурсивной природы этой реализации, их вычисление не имеет смысла.

В результате получается формула 4.5, удобная для расчётов.

$$59 + 2 * (n+1)(m+1) + 8 * n \quad (4.5)$$

Во вторую группу можно отнести рекурсивную реализацию алгоритма Левенштейна. Используемая память на один вызов функции описывается формулой 4.4.

$$\text{указатель 1} + \text{указатель 2} + \text{длина строки} + \text{длина строки} + \text{адрес возврата} + \text{результат} = \text{стоимость вызова функции} \quad (4.4)$$

$$8 + 8 + 8 + 8 + 8 + 2 = 42$$

Внутри этой рекурсивной функции может производиться вызов дополнительной функции, однако будем считать, что компилятор полноценно исполняет директиву Inline. В таком случае, нам необходимо оценить только потребление памяти в зависимости от длины слова.

Хотя рекурсивные вызовы представляют собой, по сути, троичное дерево, по факту в памяти может содержаться, в худшем случае, только одна «ветвь», количество узлов в которой может доходить до числа, равного сумме длин обоих слов.

Таким образом, необходимая для работы память описывается формулой 4.5.

$$42 * (n + m) \qquad (4.5)$$

В третью группу относится рекурсивная реализация с матрицей, которая сочетает в себе особенности потребления памяти от предыдущих групп.

Изначально она требует памяти столько же, сколько вызов матричной функции, что описывается формулой 4.3, данной выше.

Потом к этому значению прибавляется максимальная оценка рекурсивной реализации, рассчитываемая по формуле 4.5.

Здесь стоит учитывать, что с некоторой вероятностью памяти на рекурсивные вызовы будет затрачено меньше, чем в варианте без матрицы. Но в общем случае общее потребление будет рассчитываться по формуле 4.6.

$$59 + 2 * (n + 1)(m + 1) + 8 * n + 42 * (\mathrm{n} + \mathrm{m}) \qquad (4.6)$$

Многое во второй и третьей группах будет зависеть от размеров слов и вида получающегося дерева.

Вывод

В целом, видно, что рекурсивная реализация алгоритма поиска расстояния Левенштейна самая эффективная по использованию памяти, если оценивать получившиеся функции, но на небольших размерах слов она может уступать матричной реализации.

Таким образом, среди оцениваемых реализаций алгоритмов Левенштейна итеративная матричная реализация оказалась практически лучшей. Она превосходит все прочие по скорости работы, и затрачиваемое на вычисления время растёт линейно в зависимости от длины слов, а затрачиваемая на хранение данных память суммарно меньше, чем объём памяти, необходимый для хранения рекурсивных вызовов, на всех тестовых размерах слов. К тому моменту, как рекурсивная реализация алгоритма без

матрицы сравняется и превзойдёт по эффективности использования памяти матричную итеративную реализацию, вычисления будут требовать значительно больше времени. Кроме того, данные в матричной реализации могут храниться в динамически выделенной памяти, а в рекурсивной версии – только в стеке программы, размер которого фиксирован, из-за чего может возникнуть ошибка переполнения стека.

Заключение

В ходе лабораторной работы были реализованы и изучены различные реализации алгоритма поиска расстояний Левенштейна и Дамерау-Левенштейна, проведены экспериментальное сравнение по затраченному времени и теоретическая оценка затрачиваемой памяти.

Проведённое исследование позволяет сделать вывод, что, если принимать за стандарт реализацию матричного алгоритма поиска расстояния Левенштейна при двух строках длиной в 7 символов, будет получен следующий результат:

1) рекурсивная реализация без матрицы требует на 550% больше времени для вычислений;
2) на 142% хуже по памяти при выбранном значении;
3) рекурсивная реализация с матрицей требует на 52% больше времени;
4) рекурсивная реализация с матрицей требует на 242% больше памяти.

Таким образом, среди оцениваемых реализаций алгоритмов Левенштейна лидирует итеративная матричная реализация. Помимо абсолютных значений к её достоинствам стоит отнести практически линейный рост затрат времени на вычисления и, в конкретном случае, эффективность использования памяти при работе со словами длиной до приблизительно 40-45 символов. Другим её достоинством можно посчитать возможность размещения в динамически выделенной памяти. Иные реализации используют стек программы, что может послужить причиной ошибки переполнения стека.

Список литературы

1. Функция __rdtsc() – документация Майкрософт [эл. ресурс]. Режим доступа: https://docs.microsoft.com/ru-ru/cpp/intrinsics/rdtsc?view=vs-2019 (дата обращения: 27.09.2020).
2. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. Доклады Академий Наук СССР, 1965. 163.4:845- 848.