**МИНОБРНАУКИ РОССИИ**

**Санкт-Петербургский государственный**

**электротехнический университет**

**«ЛЭТИ» им. В.И. Ульянова (Ленина)**

**Кафедра МО ЭВМ**

отчет

**по лабораторной работе №3**

**по дисциплине Построение и анализ алгоритмов**

Тема: «Редакционное расстояние»

|  |  |  |
| --- | --- | --- |
| Студент гр. 3343 |  | Калиберов Н.И |
| Преподаватель |  | Жангиров Т. Р |

Санкт-Петербург

2025

# Цель работы

Изучить работу алгоритма Вагнера-Фишера для построения матрицы расстояния Левенштейна и нахождения редакционного предписания.

# Задание 1

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.
2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).
3. *delete(ε, b)* – удалить из строки символ *b*.

Каждая операция может иметь некоторую цену выполнения (*положительное число*).

Даны две строки A и B, а также три числа, отвечающие за цену каждой операции. Определите минимальную стоимость операций, которые необходимы для превращения строки A в строку B.

**Входные данные**: первая строка – три числа: цена операции *replace*, цена операции *insert*, цена операции *delete*; вторая строка – A; третья строка – B.

**Выходные данные**: одно число – минимальная стоимость операций.

**Sample Input:**

1 1 1

entrance

reenterable

**Sample Output:**

5

# Задание 2

Над строкой ε (будем считать строкой непрерывную последовательность из латинских букв) заданы следующие операции:

1. *replace(ε, a, b)* – заменить символ *a* на символ *b*.
2. *insert(ε, a)* – вставить в строку символ *a* (на любую позицию).
3. *delete(ε, b)* – удалить из строки символ *b*.

Каждая операция может иметь некоторую цену выполнения (*положительное число*).

Даны две строки A и B, а также три числа, отвечающие за цену каждой операции. Определите последовательность операций (редакционное предписание) с минимальной стоимостью, которые необходимы для превращения строки A в строку B.

| Пример (все операции стоят одинаково) | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **M** | **M** | **R** | **I** | **M** | **R** | **R** |
| **C** | **O** | **N** | **N** |  | **E** | **C** | **T** |
| C | O | N | **E** | **H** | E | A | D |

| Пример (цена замены 3, остальные операции по 1) | | | | | | | | | | |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **M** | **M** | **M** | **D** | **M** | **I** | **I** | **I** | **I** | **D** | **D** |
| **C** | **O** | **N** | **N** | **E** |  |  |  |  | **C** | **T** |
| C | O | N |  | E | **H** | **E** | **A** | **D** |  |  |

**Входные данные**: первая строка – три числа: цена операции *replace*, цена операции *insert*, цена операции *delete*; вторая строка – A; третья строка – B.

**Выходные данные**: первая строка – последовательность операций (M – совпадение, ничего делать не надо; R – заменить символ на другой; I – вставить символ на текущую позицию; D – удалить символ из строки); вторая строка – исходная строка A; третья строка – исходная строка B.

**Sample Input:**

1 1 1

entrance

reenterable

**Sample Output:**

IMIMMIMMRRM

entrance

reenterable

# Задание 3

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

Разработайте программу, осуществляющую поиск расстояния Левенштейна между двумя строками.

**Пример:**

Для строк pedestal и stien расстояние Левенштейна равно 7:

* Сначала нужно совершить четыре операции удаления символа: pedestal -> stal.
* Затем необходимо заменить два последних символа: stal -> stie.
* Потом нужно добавить символ в конец строки: stie -> stien.

**Параметры входных данных:**

Первая строка входных данных содержит строку из строчных латинских букв. (S, 1≤∣S∣≤25501≤∣*S*∣≤2550).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, 1≤∣T∣≤25501≤∣*T*∣≤2550).

**Параметры выходных данных:**

Одно число L, равное расстоянию Левенштейна между строками S и T.

**Sample Input:**

pedestal

stien

**Sample Output:**

7

**Вариант 3б.**

Добавляется 4-я операция со своей стоимостью: последовательная вставка

двух символов, выполнять её разрешается только в 2 случаях:

- в начале строки;

- если первый из вставляемых символов равен предшествующему символу

получаемой строки.

## Выполнение работы

Описание алгоритма

Расстояние Левенштейна показывает, в какое количество действий с символами одно слово можно преобразовать в другое, а Алгоритм Вагнера-Фишера – это алгоритм нахождения этого расстояния путём составления матрицы расстояний.

Сначала идёт заполнение матрицы расстояний. Она строится на основе двух рассматриваемых слов. Каждое значение в ней – расстояние Левенштейна для двух подстрок, полученных путём обрезания оригинальных строк по индексам строки и столбца. Мы преобразуем первое поданное слово во второе. Рассмотрим основные случаи при заполнении:

– расстояние между двумя пустыми строками – нулевое.

+ *insertion\_cost* (первая строка) – чтобы получить из пустой строки вторую (или её подстроку), нужно выполнить j вставок.

+ *deletion\_cost* (первый столбец) – чтобы получить из начальной строки (или её подстроки) пустую, нужно выполнить i удалений.

– для остальных ячеек матрицы берётся минимум из определённых вычисленных значений + цена соответствующей операции. Среди рассматриваемых операций:

значение слева () + цена вставки текущего символа,

значение сверху () + цена удаление текущего символа,

значение слева-сверху () + цена замены, если символы совпадают, то цена замены = 0.

Таким образом, значение в правом нижнем углу матрицы – расстояние Левенштейна для рассматриваемых строк.

Затем находится редакционное предписание – последовательность операций, которые преобразуют первую строку во вторую. Для этого необходимо из конечного значения матрицы (справа снизу) вернуться в начальную (слева сверху) по наиболее оптимальному пути. Берётся минимум из значения + цены операции для левой, верхней и левой верхней ячеек, которые соответственно обозначают операции вставки, удаления и замены (в случае совпадении символов операция не требуются).

Оценка сложности

Временная сложность алгоритма – , где n – длина первой строки, а m – длина второй. Сложность квадратичная, поскольку программа составляет матрицу расстояний, размером .

Пространственная сложность алгоритма – поскольку нам необходимо хранить непосредственно матрицу размером .

Код программы содержит реализацию следующих функций:

* *levenshteinDistanceWithDoubleInsert(s1, s2)* – вычисляет минимальное расстояние между строками *s1* и *s2*, используя замену (*replace\_cost = 1*), вставку (*insert\_cost = 1*), удаление (*delete\_cost = 1*), двойную вставку (*double\_insert\_cost = 1*).

# Тестирование

Программа была протестирована на различных входных данных. Составлена соответствующая таблица:

Таблица 1.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| (пустая строка)  ab | 1 |
| aa  aabb | 1 |
| abc  aabc | 1 |
| abbc  abbbbc | 1 |
| zxc  cxz | 2 |

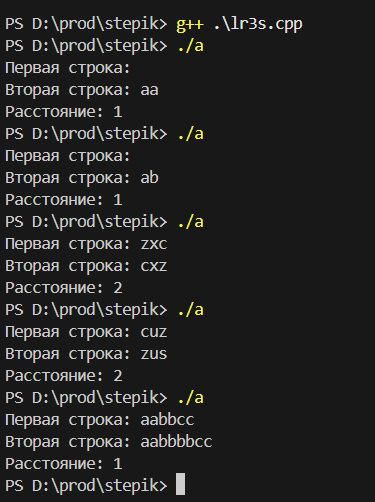


Рисунок 1 – Результатs работы программы.

# Выводы

Во время выполнения лабораторной работы, была изучена работа алгоритма Вагнера-Фишера. Решены задачи поиска матрицы расстояния Левенштейна и нахождения редакционного предписания.

**ПРИЛОЖЕНИЕ**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: lr3dop.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <climits>

#include <windows.h>

using namespace std;

int levenshteinDistanceWithDoubleInsert(const string& s1, const string& s2) {

    const int replace\_cost = 1;

    const int insert\_cost = 1;

    const int delete\_cost = 1;

    const int double\_insert\_cost = 1;

    const size\_t m = s1.length();

    const size\_t n = s2.length();

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (size\_t i = 1; i <= m; ++i) {

        dp[i][0] = dp[i - 1][0] + delete\_cost;

    }

    for (size\_t j = 1; j <= n; ++j) {

        // проверка на пустую строку

        if (j >= 2) {

            dp[0][j] = min(dp[0][j - 1] + insert\_cost,

                          dp[0][j - 2] + double\_insert\_cost);

        } else {

            dp[0][j] = dp[0][j - 1] + insert\_cost;

        }

    }

    // Заполнение матрицы

    for (size\_t i = 1; i <= m; ++i) {

        for (size\_t j = 1; j <= n; ++j) {

            if (s1[i - 1] == s2[j - 1]) {

                dp[i][j] = dp[i - 1][j - 1];

            } else {

                int cost = min({

                    dp[i - 1][j - 1] + replace\_cost,

                    dp[i][j - 1] + insert\_cost,

                    dp[i - 1][j] + delete\_cost

                });

                // Проверка возможности двойной вставки

                if (j >= 2 && s2[j - 1] == s2[j - 2]) {

                    cost = min(cost, dp[i][j - 2] + double\_insert\_cost);

                }

                dp[i][j] = cost;

            }

        }

    }

    return dp[m][n];

}

int main() {

    SetConsoleOutputCP(CP\_UTF8);

    SetConsoleCP(CP\_UTF8);

    string s1, s2;

    cout << "Первая строка: ";

    getline(cin, s1);

    cout << "Вторая строка: ";

    getline(cin, s2);

    int distance = levenshteinDistanceWithDoubleInsert(s1, s2);

    cout << "Расстояние: " << distance << endl;

    return 0;

}

Имя файла: lr3spetik.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

using namespace std;

int levenshteinDistance(const string& s1, const string& s2) {

    const size\_t m = s1.length();

    const size\_t n = s2.length();

    vector<vector<int>> dp(m + 1, vector<int>(n + 1, 0));

    for (size\_t i = 1; i <= m; ++i) {

        dp[i][0] = i;

    }

    for (size\_t j = 1; j <= n; ++j) {

        dp[0][j] = j;

    }

    for (size\_t i = 1; i <= m; ++i) {

        for (size\_t j = 1; j <= n; ++j) {

            if (s1[i - 1] == s2[j - 1]) {

                dp[i][j] = dp[i - 1][j - 1];

            } else {

                dp[i][j] = min({

                    dp[i - 1][j - 1] + 1,

                    dp[i][j - 1] + 1,

                    dp[i - 1][j] + 1

                });

            }

        }

    }

    return dp[m][n];

}

int main() {

    string s1, s2;

    cin >> s1 >> s2;

    cout << levenshteinDistance(s1, s2) << endl;

    return 0;

}