**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 Левенштейна**

Вариант – 14б

|  |  |  |
| --- | --- | --- |
| Студент гр. 3388 |  | Березовский М.А. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы:**

Изучить теоретические основы алгоритма Левенштейна.

**Задание:**

**Реализовать алгоритм Левенштейна (частный случай алгоритма Вагнера-Фишера)**

**Параметры входных данных:**

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

**Параметры выходных данных:**

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

**Sample Input:**

pedestal

stien

**Sample Output:**

7

**Выполнение работы**

Описание алгоритма для решения задачи

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

Алгоритм выполняется в два этапа:  
Изначально на вход поступают две строки – S и T

i — текущий символ строки S

j — текущий символ строки T

**1. Инициализация таблицы dp**

- Создаётся двумерный массив dp[n+1][m+1], где n = S.length(), m = T.length().

- Строка S - первая строка, которую нужно превратить, строка T - цель.

- Для каждой позиции (i, j) таблица будет хранить минимальное количество операций, чтобы превратить подстроку S[0..i-1] в подстроку T[0..j-1].

- Первая строка и первый столбец заполняются числами 0..n и 0..m соответственно:

- dp[i][0] = i → нужно удалить i символов из S, чтобы получить пустую строку.

- dp[0][j] = j → нужно вставить j символов, чтобы получить подстроку T[0..j-1].

Таким образом, таблица готова для пошагового заполнения.

**2. Заполнение таблицы dp**

- Алгоритм проходит по всем символам строк S и T

- На каждой итерации выполняется сравнение символов S[i-1] и T[j-1]:

1) Совпадение символов  
 - Если S[i-1] == T[j-1], значит, для этих позиций операций не нужно.

- Тогда dp[i][j] = dp[i-1][j-1]

- Идея: текущее редактирование не увеличивает количество операций, мы просто переносим значение предыдущей диагонали.

2) Несовпадение символов

- Если символы различны, возможны три операции:

- Замена: dp[i-1][j-1] + 1

- Удаление: dp[i-1][j] + 1

- Вставка: dp[i][j-1] + 1

- Выбирается минимальное из трёх значений: dp[i][j] = min(replace, delete, insert)

- Таким образом, на каждом шаге выбирается минимальное количество действий, необходимое для приведения подстроки S[0..i-1] к подстроке T[0..j-1].

После заполнения всей таблицы значение dp[n][m] — это минимальное количество операций, чтобы превратить строку S в строку T.

**Вариант 14б**

Методом динамического программирования вычислить: длину наибольшей общей подстроки двух строк, вывести и саму эту подстроку.

**Алгоритм**

Используется динамическое программирование. Создаётся матрица dp2 размера (n+1) на (m+1), где dp2[i][j] - длина наибольшей общей подстроки, которая оканчивается в S[i-1] и T[j-1]

Переход:

- если S[i-1] == T[j-1] -> dp2[i][j] = dp2[i-1][j-1] + 1 (продление диагонали)

- иначе -> dp2[i][j] = 0 (непрерывность оборвалась)

Параллельно поддержим:

* maxLen – текущая максимальная длина
* endIndex – индекс i в S, где заканчивается найденная максимальная подстрока

После заполнения: подстрока = S.substring(endIndex - maxLen, endIndex)

**Оценка сложности алгоритма:**

**Оценка времени выполнения:**

Алгоритм Левенштейна строит таблицу dp, где строки соответствуют символам первой строки S, а столбцы - символам второй строки T. Для каждой позиции в этой таблице алгоритм сравнивает символы двух строк и выбирает минимальное число операций: вставка, удаление или замена. Так как всего таких позиций (n+1) × (m+1), где n - длина первой строки, а m - длина второй, каждая клетка обрабатывается один раз, значит, общее количество операций примерно равно n \* m – то есть сложность O(n \* m)

**Оценка использования памяти:**

Основное место в памяти занимает двумерная таблица dp для хранения промежуточных значений. Дополнительно используются несколько переменных для хранения текущих символов и вычислений — их количество не зависит от длины строк. Таким образом, память растёт примерно пропорционально количеству клеток таблицы: (n+1) × (m+1) – O(n \* m)

**Тестирование**

Таблица 1. Тестирование.

|  |  |
| --- | --- |
| *Входные данные* | *Выходные данные* |
| abc  abc | Расстояние Левенштейна = 0  Длина наибольшей общей подстроки = 3  Общая подстрока = abc |
| abc  xyz | Расстояние Левенштейна = 3  Длина наибольшей общей подстроки = 0  Общая подстрока отсутствует |
| pedestal  stien | Расстояние Левенштейна = 7  Длина наибольшей общей подстроки = 2  Общая подстрока = st |
| abcde  xbcdy | Расстояние Левенштейна = 2  Длина наибольшей общей подстроки = 3  Общая подстрока = bcd |

**Вывод**

В ходе лабораторной работы был реализован алгоритм вычисления расстояния Левенштейна между двумя строками. Программа создаёт двумерную таблицу dp, где хранится минимальное количество операций (вставка, удаление, замена), необходимых для превращения одной строки в другую. Алгоритм пошагово заполняет таблицу, сравнивая символы строк и выбирая оптимальное действие. Код был проверен на строках с совпадениями, заменами, вставками, удалениями и полностью различными строками — результаты подтвердили корректность работы.

Дополнительно в индивидуальной части (задание 14б) был реализован поиск наибольшей общей подстроки. Для этого строится отдельная таблица динамического программирования, которая отслеживает длину текущего непрерывного совпадения. В результате программа не только вычисляет расстояние Левенштейна, но и находит саму длиннейшую подстроку, общую для двух строк.