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. Вар. 14а Методом динамического программирования вычислить: длину наибольшей общей подпоследовательности двух строк.

Студентка гр. 3343 _____ Синицкая Д.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучить и реализовать алгоритм вычисления редакционного расстояния между двумя строками.

Задание.

3.1 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

3.2 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

3.3 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. ($1 \leq |S| \leq 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. ($1 \leq |T| \leq 2550$).

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

Выполнение работы.

3.1

Основные шаги алгоритма:

1. Ввод данных: ввод стоимости операций (замены, вставки, удаления) и две строки A и B, для которых нужно найти редакционное расстояние.

2. Инициализация таблицы dp: создаётся матрица размером (m+1) × (n+1), где m и n — длины строк.

3. Заполнение базовых случаев: первая строка и первый столбец таблицы заполняются с учётом стоимости операций удаления и вставки.

4. Вычисление редакционного расстояния:

Если символы строки A[i - 1] и B[j - 1] совпадают — операция не требуется.

Если символы различаются — выбирается операция с минимальной стоимостью (замена, вставка, удаление).

5. Вычисление длины LCS (наибольшей общей подпоследовательности) — дополнительный алгоритм на основе динамического программирования.

6. Вывод таблиц и результата — осуществляется подробный вывод промежуточных значений и финальных метрик.

Описание функций и методов:

1. *void initializeBaseCases(vector<vector<int>>& dp, int m, int n, int delete_cost, int insert_cost)* — метод инициализирует первую строку и первый столбец таблицы dp.

Параметры:

dp — таблица для хранения стоимости редактирования.

m, n — длины строк A и B.

delete_cost, insert_cost — стоимости операций удаления и вставки.

2. *void computeEditDistance(vector<vector<int>>& dp, const string& A, const string& B, int replace_cost, int insert_cost, int delete_cost, int m, int n)* — метод вычисляет редакционное расстояние между строками A и B.

Параметры:

dp — таблица для хранения стоимости редактирования.

A, B — строки для сравнения.

replace_cost, insert_cost, delete_cost — стоимости операций.

m, n — длины строк A и B.

3. *int computeLCSLength(const string& A, const string& B, int m, int n)* — функция вычисляет длину наибольшей общей подпоследовательности строк A и B.

Параметры:

A, B — входные строки.

m, n — длины строк A и B.

Возвращаемое значение: *lcs[m][n]* — длина наибольшей общей подпоследовательности.

4. *void printDPTable(const vector<vector<int>>& dp, const string& A, const string& B, int m, int n)* и *void printLCSTable(const vector<vector<int>>& lcs, const string& A, const string& B, int m, int n)* — методы печатают таблицы *dp* и *lcs* соответственно в удобном для пользователя виде.

Оценка сложности алгоритма:

Временная сложность:

Редакционное расстояние:

Алгоритм использует двойной цикл по m и n, поэтому: O(m × n)

LCS (наибольшая общая подпоследовательность):

Алгоритм использует двойной цикл по m и n, поэтому: O(m × n)

Сложность по памяти:

1. Таблица dp. Хранит (m+1) × (n+1) значений: O(m × n)

2. Таблица lcs. Хранит (m+1) × (n+1) элементов: O(m × n)

3. Дополнительные переменные. Строки A, B, и несколько int переменных — пренебрежимо малы в сравнении с таблицами.

Общая пространственная сложность: O(m×n)

3.2

Основные шаги алгоритма:

1. Ввод данных: принимаются строки A, B и стоимости операций.

2. Инициализация таблиц:

dp — таблица стоимости преобразования.

op — таблица операций (M, R, I, D).

3. Заполнение таблицы dp согласно минимальной стоимости.

4. Восстановление пути операций — по таблице op строится строка операций.

5. Вычисление длины LCS — отдельно, через классический алгоритм LCS.

6. Вывод таблиц, операций и результата.

Описание функций и методов:

1. *void initializeDP(...)void initializeDP(vector<vector<int>>& dp, vector<vector<char>>& op, int m, int n, int cost_delete, int cost_insert)* — метод инициализирует начальные значения таблицы dp и op.

Параметры:

dp, op — таблицы для хранения стоимости и действий

m, n — размеры строк A, B.

cost_delete, cost_insert — стоимости операций.

2. *void fillDPTable(...)void fillDPTable(const string& A, const string& B, int cost_replace, int cost_insert, int cost_delete, vector<vector<int>>& dp, vector<vector<char>>& op)* — метод строит таблицу минимальных стоимостей преобразования строки A в B.

Параметры:

A, B — входные строки.

cost_replace, cost_insert, cost_delete — стоимости операций.

dp, op — таблицы для хранения стоимости и операций.

3. *string reconstructOperations(...)string reconstructOperations(const vector<vector<char>>& op, int m, int n)* — функция восстанавливает последовательность операций по таблице *op*.

Параметры:

op — таблица операций.

m, n — размеры строк.

Возвращаемое значение: *operations* — строка символов.

4. *int computeLCSLength(const string& A, const string& B, int m, int n)* — функция вычисляет длину наибольшей общей подпоследовательности строк A и B.

Параметры:

A, B — входные строки.

m, n — длины строк A и B.

Возвращаемое значение: *lcs[m][n]* — длина наибольшей общей подпоследовательности.

5. *void printDPTable(const vector<vector<int>>& dp, const string& A, const string& B, int m, int n)* и *void printLCSTable(const vector<vector<int>>& lcs, const string& A, const string& B, int m, int n)* — методы печатают таблицы *dp* и *lcs* соответственно в удобном для пользователя виде.

Оценка сложности алгоритма:

Временная сложность:

Алгоритм выполняется в двух этапах, оба со сложностью:

Инициализация + заполнение таблицы преобразования (dp): $O(m \times n)$

Вычисление LCS: $O(m \times n)$

Восстановление последовательности операций: O(m + n)

Итоговая временная сложность: O(m × n)

Сложность по памяти:

1. Таблица dp размера (m + 1) × (n + 1): O(m×n)

2. Таблица op размера (m + 1) × (n + 1): O(m×n)

3. Таблица lcs размера (m + 1)(n + 1): O(m×n)

Общая пространственная сложность: O(m×n)

3.3

Основные шаги алгоритма:

1. Ввод двух строк S и T

2. Инициализация таблицы dp размером (m+1) x (n+1) для вычисления расстояния Левенштейна

3. Заполнение таблицы dp:

Используются три операции: вставка, удаление, замена

При совпадении символов используется перенос диагонального значения

Вычисление расстояния Левенштейна — значение в правом нижнем углу таблицы dp

4. Вычисление длины LCS (наибольшей общей подпоследовательности):

Используется отдельная таблица lcs

При совпадении символов увеличивается длина LCS

5. Вывод таблиц dp и lcs с пояснениями

Описание функций и методов:

1.*void printDPTable(const vector<vector<int>>& dp, const string& A, const string& B, int m, int n)* и *void printLCSTable(const vector<vector<int>>& lcs, const string& A, const string& B, int m, int n)* — методы печатают таблицы *dp* и *lcs* соответственно в удобном для пользователя виде.

2. *int computeLCSLength(const string& A, const string& B, int m, int n)* — функция вычисляет длину наибольшей общей подпоследовательности строк A и B.

Параметры:

A, B — входные строки.

m, n — длины строк A и B.

Возвращаемое значение: *lcs[m][n]* — длина наибольшей общей подпоследовательности.

3. *void initializeDP(vector<vector<int>>& dp, int m, int n)* — метод заполняет первую строку и первый столбец значениями (базовые случаи).

Параметры:

dp — таблица расстояний

m, n — размеры таблицы

4. *void fillDPTable(const string& S, const string& T, vector<vector<int>>& dp)* — метод сравнивает символы двух строк, выбирает минимум из вставки, удаления и замены, заполняет таблицу dp шаг за шагом

Параметры:

S, T — строки

dp — таблица расстояний

Оценка сложности алгоритма:

Временная сложность:

Для расстояния Левенштейна:

Проходим по всей таблице dp размером (m + 1) × (n + 1): O(m × n)

Для LCS:

Проходим по всей таблице lcs размером (m + 1) × (n + 1): O(m × n)

Итоговая временная сложность: O(m × n)

Сложность по памяти:

1. Таблица dp размера (m + 1) × (n + 1): O(m×n)

2. Таблица lcs размера $(m + 1)(n + 1)$: $O(m \times n)$

Общая пространственная сложность: $O(m \times n)$

Тестирование.

Результаты тестирования представлены в таблицах 1-3.

Таблица 1 – Результаты тестирования задания 3.1

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2 1 1 abc abc	Редакционное расстояние: 0 Длина наибольшей подпоследовательности: 3	Строки уже совпадают. Не требуется ни одной операции. Результат соответствует ожидаемому.
2.	1 1 1 abc abd	Редакционное расстояние: 1 Длина наибольшей подпоследовательности: 2	Только одна замена. Результат соответствует ожидаемому.
3.	5 1 1 abc abcd	Редакционное расстояние: 1 Длина наибольшей подпоследовательности: 3	Вставка в конец. Результат соответствует ожидаемому.
4.	1 1 1 kitten sitting	Редакционное расстояние: 3 Длина наибольшей подпоследовательности: 4	Одна замена, одна вставка, одна вставка. Результат соответствует ожидаемому.
5.	2 2 2 aaaaa bbbbb	Редакционное расстояние: 10 Длина наибольшей подпоследовательности: 0	Все символы заменяются. Результат соответствует ожидаемому.

Таблица 2 – Результаты тестирования задания 3.2

№ п/п	Входные данные	Выходные данные	Комментарии
1.	1 1 1 abc abc	Последовательность операций: MMM Исходная строка A: abc Целевая строка B: abc Длина наибольшей подпоследовательности: 3	Строки уже совпадают. Не требуется ни одной операции. Результат соответствует ожидаемому.
2.	1 1 1 abc abd	Последовательность операций: MMR Исходная строка A: abc	Только одна замена. Результат соответствует ожидаемому.

		Целевая строка B: abd Длина наибольшей подпоследовательности: 2	
3.	5 1 1 abc abcd	Последовательность операций: MMMI Исходная строка A: abc Целевая строка B: abcd Длина наибольшей подпоследовательности: 3	Вставка в конец. Результат соответствует ожидаемому.
4.	1 1 1 kitten sitting	Последовательность операций: RMMMRMI Исходная строка A: kitten Целевая строка B: sitting Длина наибольшей подпоследовательности: 4	Классический случай: замены, совпадения и вставка.. Результат соответствует ожидаемому.
5.	2 2 2 aaaaa bbbbb	Последовательность операций: RRRRR Исходная строка A: aaaaa Целевая строка B: bbbbb Длина наибольшей подпоследовательности: 0	Все символы заменяются. Результат соответствует ожидаемому.

Таблица 3 – Результаты тестирования задания 3.3

№ п/п	Входные данные	Выходные данные	Комментарии
1.	abc abc	Расстояние Левенштейна между "abc" и "abc" = 0 Длина наибольшей подпоследовательности: 3	Строки идентичны. Результат соответствует ожидаемому.
2.	abc abd	Расстояние Левенштейна между "abc" и "abd" = 1 Длина наибольшей подпоследовательности: 2	Одна замена. Результат соответствует ожидаемому.

3.	abc abcd	Расстояние Левенштейна между "abc" и "abcd" = 1 Длина наибольшей подпоследовательности: 3	Одна вставка. Результат соответствует ожидаемому.
4.	abc ab	Расстояние Левенштейна между "abc" и "ab" = 1 Длина наибольшей подпоследовательности: 2	Одна операция удаления. Результат соответствует ожидаемому.
5.	pedestal stien	Расстояние Левенштейна между "pedestal" и "stien" = 7 Длина наибольшей подпоследовательности: 2	Классический случай. Результат соответствует ожидаемому.

Выводы.

В лабораторной работе была изучена и реализована реализация алгоритма вычисления редакционного расстояния между двумя строками. Реализация позволяет определить минимальное количество операций (вставка, удаление, замена), необходимых для преобразования одной строки в другую.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: lr_3_1_1.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <iomanip>
#include <algorithm>

using namespace std;

// метод печати таблицы dp
void printDPTable(const vector<vector<int>>& dp, const string& A,
const string& B, int m, int n) {

    cout << "\nТаблица стоимости редактирования:\n\n";
    cout << "    ";
    for (int j = 0; j <= n; ++j) {
        if (j == 0) cout << "  ";
        else cout << " " << B[j - 1] << " ";
    }
    cout << endl;

    for (int i = 0; i <= m; ++i) {
        if (i == 0) cout << "  ";
        else cout << A[i - 1] << " ";

        for (int j = 0; j <= n; ++j) {
            cout << setw(2) << dp[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

// метод печати таблицы LCS
void printLCSTable(const vector<vector<int>>& lcs, const string& A,
const string& B, int m, int n) {

    cout << "\nТаблица значений LCS:\n\n";
    cout << "    ";
    for (int j = 0; j <= n; ++j) {
        if (j == 0) cout << "  ";
        else cout << " " << B[j - 1] << " ";
    }
    cout << endl;

    for (int i = 0; i <= m; ++i) {
        if (i == 0) cout << "  ";
        else cout << A[i - 1] << " ";

        for (int j = 0; j <= n; ++j) {
            cout << setw(2) << lcs[i][j] << " ";
        }
        cout << endl;
```

```
        }
        cout << endl;
    }

    // метод инициализации базовых случаев для динамической таблицы
    void initializeBaseCases(vector<vector<int>>& dp, int m, int n, int
delete_cost, int insert_cost) {
        // заполнение первой колонки (число операций удаления символов)
        for (int i = 0; i <= m; ++i)
            dp[i][0] = i * delete_cost; // стоимость удаления i
символов

        // заполнение первой (число операций вставки символов)
        for (int j = 0; j <= n; ++j)
            dp[0][j] = j * insert_cost; // стоимость вставки j символов
    }

    // метод вычисления редакционного расстояния между строками A и B
    void computeEditDistance(vector<vector<int>>& dp, const string& A,
const string& B, int replace_cost, int insert_cost, int delete_cost, int
m, int n) {

        // применяем правило треугольника
        replace_cost = min(replace_cost, delete_cost + insert_cost);

        cout << "\nНачинаем вычисление редакционного расстояния:\n";

        // заполнение таблицы для хранения стоимости редактирования
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                // если текущие символы совпадают
                if (A[i - 1] == B[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1]; // стоимость остается
прежней

                    cout << "dp[" << i << "][" << j << "] (" << A[i -
1] << " == " << B[j - 1] << ") = dp[" << (i - 1) << "][" << (j - 1) << "]
= " << dp[i][j] << endl;

                } else {

                    // выбор минимальной стоимости из трех возможных
операций
                    dp[i][j] = min({
                        dp[i - 1][j - 1] + replace_cost, // стоимость
замены символа
                        dp[i][j - 1] + insert_cost,      // стоимость
вставки символа
                        dp[i - 1][j] + delete_cost       // стоимость
удаления символа
                    });

                    cout << "dp[" << i << "][" << j << "] (replace " <<
A[i - 1] << " -> " << B[j - 1]
                         << ") = min(replace=" << dp[i - 1][j - 1] +
replace_cost
                         << ", insert=" << dp[i][j - 1] + insert_cost
```

```
                        << ", delete=" << dp[i - 1][j] + delete_cost
                        << ") = " << dp[i][j] << endl;
                }
            }
        }

        printDPTable(dp, A, B, m, n);
    }

    // функция вычисление длины наибольшей общей подпоследовательности
    int computeLCSLength(const string& A, const string& B, int m, int
n) {

        vector<vector<int>> lcs(m + 1, vector<int>(n + 1, 0)); //
массив для хранения значений LCS

        cout << "\nНачинаем вычисление длины наибольшей общей
подпоследовательности (LCS):\n";

         // заполнение массива для хранения значений LCS
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                // если символы совпадают, увеличиваем длину на 1
                if (A[i - 1] == B[j - 1]) {
                    lcs[i][j] = lcs[i - 1][j - 1] + 1;

                    cout << "lcs[" << i << "][" << j << "] (match " <<
A[i - 1] << ") = "
                        << "lcs[" << (i - 1) << "][" << (j - 1) << "]
+ 1 = " << lcs[i][j] << endl;

                } else {

                    // иначе берем максимум между LCS без текущего
символа A или B
                    lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1]);

                    cout << "lcs[" << i << "][" << j << "] = max("
                        << "lcs[" << (i - 1) << "][" << j << "] = " <<
lcs[i - 1][j] << ", "
                        << "lcs[" << i << "][" << (j - 1) << "] = " <<
lcs[i][j - 1] << ") = "
                        << lcs[i][j] << endl;
                }
            }
        }

        printLCSTable(lcs, A, B, m, n);

        return lcs[m][n];
    }


    int main() {

        int replace_cost, insert_cost, delete_cost;                //
стоимости операций замены, вставки и удаления
```

```
        cin >> replace_cost >> insert_cost >> delete_cost;

        string A, B;    // строки A, B
        cin >> A >> B;

        int m = A.size();                                       // длина
строки A
        int n = B.size();                                       // длина
строки B
        vector<vector<int>>  dp(m  +  1,  vector<int>(n  +  1));    //
инициализация таблицы для хранения стоимости редактирования

        initializeBaseCases(dp,   m,   n,   delete_cost,   insert_cost);
// инициализация базовых случаев
        computeEditDistance(dp,   A,   B,   replace_cost,   insert_cost,
delete_cost, m, n);    // вычисление расстояния редактирования
        int    lcsLength    =    computeLCSLength(A,    B,    m,    n);
// вычисление длины наибольшей общей подпоследовательности

        cout << "\nРедакционное расстояние: " << dp[m][n] << endl;
        cout   <<   "\nДлина   наибольшей   подпоследовательности:   "   <<
lcsLength << endl;

        return 0;
    }
```

Название файла: lr_3_1_2.cpp

```
    #include <iostream>
    #include <vector>
    #include <string>
    #include <algorithm>
    #include <iomanip>

    using namespace std;

    // метод печати таблицы LCS
    void printLCSTable(const vector<vector<int>>& lcs, const string& A,
const string& B, int m, int n) {

        cout << "\nТаблица значений LCS:\n\n";
        cout << "    ";
        for (int j = 0; j <= n; ++j) {
            if (j == 0) cout << "  ";
            else cout << " " << B[j - 1] << " ";
        }
        cout << endl;

        for (int i = 0; i <= m; ++i) {
            if (i == 0) cout << "  ";
            else cout << A[i - 1] << " ";

            for (int j = 0; j <= n; ++j) {
                cout << setw(2) << lcs[i][j] << " ";
            }
            cout << endl;
        }
        cout << endl;
```

```
    }

    // функция вычисление длины наибольшей общей подпоследовательности
    int computeLCSLength(const string& A, const string& B, int m, int
n) {

        vector<vector<int>> lcs(m + 1, vector<int>(n + 1, 0)); //
массив для хранения значений LCS

        cout << "\nНачинаем вычисление длины наибольшей общей
подпоследовательности (LCS):\n";

         // заполнение массива для хранения значений LCS
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                // если символы совпадают, увеличиваем длину на 1
                if (A[i - 1] == B[j - 1]) {
                    lcs[i][j] = lcs[i - 1][j - 1] + 1;

                    cout << "lcs[" << i << "][" << j << "] (match " <<
A[i - 1] << ") = "
                         << "lcs[" << (i - 1) << "][" << (j - 1) << "]
+ 1 = " << lcs[i][j] << endl;

                } else {

                    // иначе берем максимум между LCS без текущего
символа A или B
                    lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1]);

                    cout << "lcs[" << i << "][" << j << "] = max("
                         << "lcs[" << (i - 1) << "][" << j << "] = " <<
lcs[i - 1][j] << ", "
                         << "lcs[" << i << "][" << (j - 1) << "] = " <<
lcs[i][j - 1] << ") = "
                         << lcs[i][j] << endl;
                }
            }
        }

        printLCSTable(lcs, A, B, m, n);

        return lcs[m][n];
    }

    // метод инициализации таблицы
    void initializeDP(vector<vector<int>>& dp, vector<vector<char>>&
op, int m, int n, int cost_delete, int cost_insert) {
        // инициализация первой колонки (строка A)
        for (int i = 0; i <= m; ++i) {
            dp[i][0] = i * cost_delete;    // стоимость удаления i
символов из A
            if (i > 0) op[i][0] = 'D';    // запоминаем операцию 'D'
(удаление)
        }
        // инициализация первой строки (строка B)
        for (int j = 0; j <= n; ++j) {
```

```
            dp[0][j] = j * cost_insert;    // стоимость вставки j
символов в A
            if (j > 0) op[0][j] = 'I';    // запоминаем операцию 'I'
(вставка)
        }
    }

    // метод заполнения таблицы
    void fillDPTable(const string& A, const string& B, int
cost_replace, int cost_insert, int cost_delete, vector<vector<int>>& dp,
vector<vector<char>>& op) {

        int m = A.size(), n = B.size();  // размеры строк A, B

        // проход по всем символам строк A и B
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                cout << "Сравниваем A[" << i - 1 << "] = " << A[i - 1]
<< " и B[" << j - 1 << "] = " << B[j - 1] << "\n";

                // если символы совпадают, цена остается прежней
                if (A[i - 1] == B[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];      // операция не
нужна
                    op[i][j] = 'M';                    // запоминаем
операцию 'M' (совпадение)

                    cout << "    Совпадают -> операция: M (match),
стоимость: " << dp[i][j] << "\n";

                } else {

                    // рассчет стоимости операций
                    int r = dp[i - 1][j - 1] + cost_replace; // замена
символа
                    int d = dp[i - 1][j] + cost_delete;          //
удаление символа из A
                    int ins = dp[i][j - 1] + cost_insert;    // вставка
символа в A

                    dp[i][j] = r;     // предполагаем, что замена -
наименьшая стоимость
                    op[i][j] = 'R'; // запоминаем операцию замены

                    cout << "  Не совпадают -> рассматриваем:\n";
                    cout << "    Замена (R): " << r << "\n";
                    cout << "    Удаление (D): " << d << "\n";
                    cout << "    Вставка (I): " << ins << "\n";

                    // если удаление дешевле, обновляем стоимость и
операцию
                    if (d < dp[i][j]) {
                        dp[i][j] = d;
                        op[i][j] = 'D'; // запоминаем операцию 'D'
(удаление)
                    }
```

```
                    // если вставка дешевле, обновляем стоимость и
операцию
                    if (ins < dp[i][j]) {
                        dp[i][j] = ins;
                        op[i][j] = 'I'; // запоминаем операцию 'I'
(вставка)
                    }

                    cout << "  Выбрана операция: " << op[i][j] << ",
стоимость: " << dp[i][j] << "\n";
                }
                cout << "-----------------------------------\n";
            }
        }
    }

    // вывод таблицы DP и операций
    void printTables(const vector<vector<int>>& dp, const
vector<vector<char>>& op, const string& A, const string& B) {
        int m = A.size(), n = B.size();

        cout << "\nТаблица стоимости операций (dp):\n    ";
        for (int j = 0; j <= n; ++j)
            cout << setw(4) << (j == 0 ? '-' : B[j - 1]);
        cout << "\n";

        for (int i = 0; i <= m; ++i) {
            cout << (i == 0 ? '-' : A[i - 1]) << " ";
            for (int j = 0; j <= n; ++j)
                cout << setw(4) << dp[i][j];
            cout << "\n";
        }

        cout << "\nТаблица операций (op):\n    ";
        for (int j = 0; j <= n; ++j)
            cout << setw(4) << (j == 0 ? '-' : B[j - 1]);
        cout << "\n";

        for (int i = 0; i <= m; ++i) {
            cout << (i == 0 ? '-' : A[i - 1]) << " ";
            for (int j = 0; j <= n; ++j)
                cout << setw(4) << (op[i][j] ? op[i][j] : ' ');
            cout << "\n";
        }
    }

    // функция восстановления последовательности операций
    string reconstructOperations(const vector<vector<char>>& op, int m,
int n) {
        string operations;  // строка для хранения последовательности
операций
        int i = m, j = n;

        // восстанавление операций (начиная с конца)
        while (i > 0 || j > 0) {
            char action = op[i][j]; // получение текущей операции
            operations += action;  // добавление операции в результат
```

```
            // переход в предыдущее состояние в зависимости от операции
            switch (action) {
                case 'M':   // совпадение
                case 'R':   // замена
                    --i;                        // переход к предыдущему
символу в A
                    --j;                        // переход к предыдущему
символу в B
                    break;
                case 'D':   // удаление
                    --i;                        // переход к предыдущему
символу в A
                    break;
                case 'I':   // вставка
                    --j;                        // переход к предыдущему
символу в B
                    break;
            }
        }

        // реверс операций, чтобы получить правильный порядок
        reverse(operations.begin(), operations.end());

        return operations;
    }

    int main() {

        int cost_replace, cost_insert, cost_delete;                 //
стоимости операций замены, вставки и удаления
        cin >> cost_replace >> cost_insert >> cost_delete;

        string A, B;        // строки A, B
        cin >> A >> B;

        int m = A.size(), n = B.size();  // длины строк A, B

        // создание таблиц для хранения стоимости и действий
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));
        vector<vector<char>> op(m + 1, vector<char>(n + 1));

        initializeDP(dp,   op,   m,   n,   cost_delete,   cost_insert);
// инициализация таблицы
        fillDPTable(A, B, cost_replace, cost_insert, cost_delete, dp,
op);  // заполнение таблицы
        string   operations   =   reconstructOperations(op,   m,   n);
// восстановление последовательности операций
        int    lcsLength    =    computeLCSLength(A,    B,    m,    n);
// вычисление длины наибольшей общей подпоследовательности

        cout  <<  "\nПоследовательность  операций:  "  <<  operations  <<
"\n";
        cout << "Исходная строка A: " << A << "\n";
        cout << "Целевая строка B: " << B << "\n";
        cout << "Длина наибольшей подпоследовательности: " << lcsLength
<< endl;

        return 0;
```

```
}
```

Название файла: lr_3_2.cpp

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <iomanip>

using namespace std;

// метод печати таблицы LCS
void printLCSTable(const vector<vector<int>>& lcs, const string& A,
const string& B, int m, int n) {

    cout << "\nТаблица значений LCS:\n\n";
    cout << "    ";
    for (int j = 0; j <= n; ++j) {
        if (j == 0) cout << "  ";
        else cout << " " << B[j - 1] << " ";
    }
    cout << endl;

    for (int i = 0; i <= m; ++i) {
        if (i == 0) cout << "  ";
        else cout << A[i - 1] << " ";

        for (int j = 0; j <= n; ++j) {
            cout << setw(2) << lcs[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
}

// функция вычисление длины наибольшей общей подпоследовательности
int computeLCSLength(const string& A, const string& B, int m, int
n) {

    vector<vector<int>> lcs(m + 1, vector<int>(n + 1, 0)); //
массив для хранения значений LCS

    cout << "\nНачинаем вычисление длины наибольшей общей
подпоследовательности (LCS):\n";

     // заполнение массива для хранения значений LCS
    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {

            // если символы совпадают, увеличиваем длину на 1
            if (A[i - 1] == B[j - 1]) {
                lcs[i][j] = lcs[i - 1][j - 1] + 1;

                cout << "lcs[" << i << "][" << j << "] (match " <<
A[i - 1] << ") = "
                     << "lcs[" << (i - 1) << "][" << (j - 1) << "]
+ 1 = " << lcs[i][j] << endl;
```

```
                } else {

                    // иначе берем максимум между LCS без текущего
символа A или B
                    lcs[i][j] = max(lcs[i - 1][j], lcs[i][j - 1]);

                    cout << "lcs[" << i << "][" << j << "] = max("
                         << "lcs[" << (i - 1) << "][" << j << "] = " <<
lcs[i - 1][j] << ", "
                         << "lcs[" << i << "][" << (j - 1) << "] = " <<
lcs[i][j - 1] << ") = "
                         << lcs[i][j] << endl;
                }
            }
        }

        printLCSTable(lcs, A, B, m, n);

        return lcs[m][n];
    }

    // Функция для вывода таблицы расстояний
    void printDPTable(const vector<vector<int>>& dp, const string& S,
const string& T) {
        int m = S.size();
        int n = T.size();

        cout << "\nТаблица расстояний (dp):\n\n";

        // Верхняя строка — символы строки T
        cout << setw(4) << " ";
        cout << setw(4) << "' '";  // пустой символ перед T
        for (char c : T)
            cout << setw(4) << c;
        cout << endl;

        // Вся таблица
        for (int i = 0; i <= m; ++i) {
            if (i == 0)
                cout << setw(4) << "' '";  // пустой символ перед S
            else
                cout << setw(4) << S[i - 1];

            for (int j = 0; j <= n; ++j) {
                cout << setw(4) << dp[i][j];
            }
            cout << endl;
        }
        cout << endl;
    }

    // метод инициализации таблицы
    void initializeDP(vector<vector<int>>& dp, int m, int n) {
        // первая строка соответствует расстоянию от пустой строки T к
префиксам строки S
        for (int i = 0; i <= m; ++i) dp[i][0] = i;   // расстояние до
пустой строки T
```

```
        // первая колонка соответствует расстоянию от пустой строки S к
префиксам строки T
        for (int j = 0; j <= n; ++j) dp[0][j] = j;  // расстояние до
пустой строки S
    }

    // метод заполнения таблицы расстояний Левенштейна
    void fillDPTable(const string& S, const string& T,
vector<vector<int>>& dp) {

        int m = S.size(), n = T.size(); // длины строк A и B

        // заполнение таблицы
        for (int i = 1; i <= m; ++i) {
            for (int j = 1; j <= n; ++j) {

                cout << "Сравниваем S[" << i - 1 << "] = " << S[i - 1]
<< " и T[" << j - 1 << "] = " << T[j - 1] << "\n";

                // если текущие символы совпадают, переносим значение
диагонально
                if (S[i - 1] == T[j - 1]) {
                    dp[i][j] = dp[i - 1][j - 1];

                    cout << "   Совпадают -> операция: M (match),
стоимость: " << dp[i][j] << "\n";

                } else {

                    // берем минимум из трех операций: удаление,
вставка, замена
                    dp[i][j] = min({
                        dp[i - 1][j] + 1,      // удаление (S[i-1] не
учитываем)
                        dp[i][j - 1] + 1,       // вставка (добавляем
T[j-1] к S)
                        dp[i - 1][j - 1] + 1  // замена (заменяем S[i-
1] на T[j-1])
                    });

                    cout << "  Не совпадают -> рассматриваем:\n";
                    cout << "    Удаление (D): " << dp[i - 1][j] + 1 <<
"\n";
                    cout << "    Вставка (I): " << dp[i][j - 1] + 1 <<
"\n";
                    cout << "    Замена (R): " << dp[i - 1][j - 1] + 1
<< "\n";

                    char op;
                    if (dp[i][j] == dp[i - 1][j - 1] + 1)
                        op = 'R';
                    else if (dp[i][j] == dp[i - 1][j] + 1)
                        op = 'D';
                    else
                        op = 'I';
```

```
                    cout << "    Выбрана операция: " << op << ",
стоимость: " << dp[i][j] << "\n";
                }
                cout << "-----------------------------------\n";
            }
        }

        printDPTable(dp, S, T);
    }

    int main() {
        string S, T;
        cin >> S >> T;

        int m = S.size(); // длина строки S
        int n = T.size(); // длина строки T

        vector<vector<int>> dp(m + 1, vector<int>(n + 1)); // таблица
для хранения расстояний
        initializeDP(dp, m, n);                                    //
инициализация таблицы
        fillDPTable(S, T, dp);                                     //
заполнение таблицы расстояний
        int lcsLength = computeLCSLength(S, T, m, n);              //
вычисление длины наибольшей общей подпоследовательности

        cout << "Расстояние Левенштейна между \"" << S << "\" и \"" <<
T << "\" = " << dp[m][n] << endl;
        cout << "Длина наибольшей подпоследовательности: " << lcsLength
<< endl;

        return 0;
    }
```