МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №3

по дисциплине «Построение и анализ алгоритмов»

ТЕМА: «РЕДАКЦИОННОЕ РАССТОЯНИЕ»

ВАРИАНТ 13

Гусакова К.А.

Студент гр. 3388 _____

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

ЗАДАНИЕ НА ЛАБОРАТОРНУЮ РАБОТУ

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.

Задание на вариант:

Вывести не одно, а все редакционные предписания с минимальной стоимостью. Для одного из предписаний продемонстрировать его применение для преобразования 1-ой строки во 2-ую.

Описание алгоритма для решения задачи

Алгоритм, реализованный в коде, решает задачу поиска самого дешевого преобразования одной строки в другую, используя операции с учётом их стоимости. Операции: замена символа, вставка и удаление. Каждая операция имеет заданную стоимость, и цель – получить строку B из строки A с минимальной суммарной стоимостью.

В основе алгоритма лежит метод динамического программирования, который строит таблицу (dp), где каждая ячейка содержит минимальную стоимость преобразования префикса строки A в префикс строки B, а также информацию о том, какая операция была применена (замена, вставка и т. д.).

Вот как работает алгоритм шаг за шагом:

Сначала инициализируется таблица dp, где dp[i][j] – это структура, содержащая:

- cost: минимальная стоимость преобразования первых i символов строки A в первые j символов строки B;
- operation: символ, обозначающий последнюю операцию на этом шаге (M – match, R – замена, I – вставка, D – удаление).

Пример начала инициализации:

```
dp[0][0] = { 0, 'M' }; // Нулевая стоимость: ничего не преобразуем
for (int i = 1; i <= m; ++i)
    dp[i][0] = { dp[i - 1][0].cost + cost_delete, 'D' }; // Удаляем все символы A
for (int j = 1; j <= n; ++j)
    dp[0][j] = { dp[0][j - 1].cost + cost_insert, 'I' }; // Вставляем все символы B
```

Это задаёт начальные состояния: чтобы получить пустую строку из A, нужно удалить все её символы; чтобы получить B из пустой строки, нужно вставить все её символы.

Далее заполняется остальная часть таблицы. На каждом шаге рассматриваются три варианта:

- Если символы A[i-1] и B[j-1] совпадают, операция – M, и стоимость не увеличивается.
- Иначе выбирается минимальная по стоимости из трёх операций:

  R (замена) – берётся стоимость ячейки dp[i-1][j-1] + cost_replace;

  I (вставка) – берётся dp[i][j-1] + cost_insert;

  D (удаление) – берётся dp[i-1][j] + cost_delete.

  Пример фрагмента:

```
if (A[i - 1] == B[j - 1]) {
    dp[i][j] = { dp[i - 1][j - 1].cost, 'M' };
} else {
    int replace_cost = dp[i - 1][j - 1].cost + cost_replace;
    int insert_cost = dp[i][j - 1].cost + cost_insert;
    int delete_cost = dp[i - 1][j].cost + cost_delete;
```

```
            if (replace_cost <= insert_cost && replace_cost <= delete_cost) {
                dp[i][j] = { replace_cost, 'R' };
            } else if (insert_cost <= replace_cost && insert_cost <= delete_cost) {
                dp[i][j] = { insert_cost, 'I' };
            } else {
                dp[i][j] = { delete_cost, 'D' };
            }
        }
```

После того как таблица dp полностью построена, итоговая стоимость – это значение dp[m][n].cost, где m и n – длины строк A и B соответственно.

Затем запускается отдельная рекурсивная функция, которая восстанавливает все возможные последовательности операций, давшие минимальную стоимость. Это делается путём обхода таблицы dp в обратном порядке – от dp[m][n] к dp[0][0].

На каждом шаге проверяется, какая операция могла привести к текущей ячейке при данной стоимости, и строится строка операций в обратном порядке. Если найден конец пути (ячейка dp[0][0]), строка операций сохраняется.

Пример восстановления пути:

```
        if (A[i - 1] == B[j - 1] && dp[i][j].cost == dp[i - 1][j - 1].cost) {
            find_all_paths(dp, A, B, i - 1, j - 1, current + 'M', all_paths, ...);
        }
        if (A[i - 1] != B[j - 1] && dp[i][j].cost == dp[i - 1][j - 1].cost + cost_replace)
{
            find_all_paths(dp, A, B, i - 1, j - 1, current + 'R', all_paths, ...);
        }
```

Затем показывается применение одной из таких последовательностей. Строка A по шагам превращается в B, и после каждой операции (замена, вставка, удаление) выводится текущее состояние строки. При этом

используется индекс i, который указывает на текущую позицию в строке, чтобы изменения применялись строго слева направо и только к оригинальным символам, а не к тем, что были добавлены во время.

Оценка сложности алгоритма по времени и памяти

Временная сложность:

1. build_dp()

```
for (int i = 1; i <= m; ++i)
   dp[i][0] = { dp[i - 1][0].cost + cost_delete, 'D' };
for (int j = 1; j <= n; ++j)
   dp[0][j] = { dp[0][j - 1].cost + cost_insert, 'I' };
for (int i = 1; i <= m; ++i) {
   for (int j = 1; j <= n; ++j) {
   }
}
```

Двойной цикл по i и j – O(m * n)

Каждый элемент вычисляется за O(1) времени

Временная сложность: O(m * n)

2. find_all_paths() – Нахождение всех путей преобразований

Рекурсивно находит все возможные пути с минимальной стоимостью редактирования от (m, n) до (0, 0) по матрице dp.

```
if (A[i - 1] == B[j - 1] && ...) // M
if (A[i - 1] != B[j - 1] && ...) // R
if (dp[i][j].cost == dp[i][j - 1].cost + cost_insert) // I
if (dp[i][j].cost == dp[i - 1][j].cost + cost_delete) // D
```

$O(3^{(m + n)})$в худшем случае, в каждой точке (i, j) может быть до трех рекурсивных вызовов (если все три операции допустимы и равны по стоимости). Глубина рекурсии максимум m + n(максимальная проходка).

Временная сложность: $O(3^{(m + n)})$

3. apply_operations() – пошаговое применение одного пути преобразования

O(m + n),

где m = A.length(), n = B.length()

Обрабатываем operations, длина которой не больше m + n (максимум m удалений + n вставок).

Каждая операция:

M, R – O(1)

I, D – вставка/удаление – O(k), где k – длина строки

Тк идет вывод каждой строки, то (m+n)(m+n)

Временная сложность**:** $O(m + n)^2$

Общая временная сложность программы:

$O(3^{(m+n)})$

Пространственная сложность:

1. build_dp() – Построение таблицы DP

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

- Хранится таблица размером (m + 1) × (n + 1).
- Каждый элемент (Cell) занимает **O(1)** памяти.

Пространственная сложность: O(m * n) – память под таблицу.

2. find_all_paths() – Нахождение всех путей преобразований

Основные ресурсы:

- Рекурсивный стек глубиной до m + n.
- all_paths – вектор строк, каждая длиной до m + n.
- Строка current – создается на каждом уровне рекурсии.

В худшем случае число путей экспоненциальное: до 3^(m+n).

Пространственная сложность: O(p * (m + n)), где p – количество оптимальных путей

3. apply_operations() – Применение одного пути преобразования

Используемая память:

- result – копия строки A O(m)
- operations – длина до m + n

- Дополнительно: временные строки при вставках/удалениях (в пределах O(m + n))

Пространственная сложность: O(m + n) – максимум длина промежуточной строки result.

Общая пространственная сложность программы:

$O(3^{m+n} * (m+n))$

Описание способа хранения частичных решений.

Частичные решения хранятся в таблице dp типа vector<vector<Cell>>.

```
struct Cell {
   int cost
   char operation;
};
```

Использованные оптимизации.

Сокращение глубины вызовов.

В find_all_paths() используется проверка:

```
if (i == 0 && j == 0)
```

которая завершает рекурсию как только достигнут старт пути (нижний угол таблицы).

Тестирование. Показ граничных случаев алгоритма.

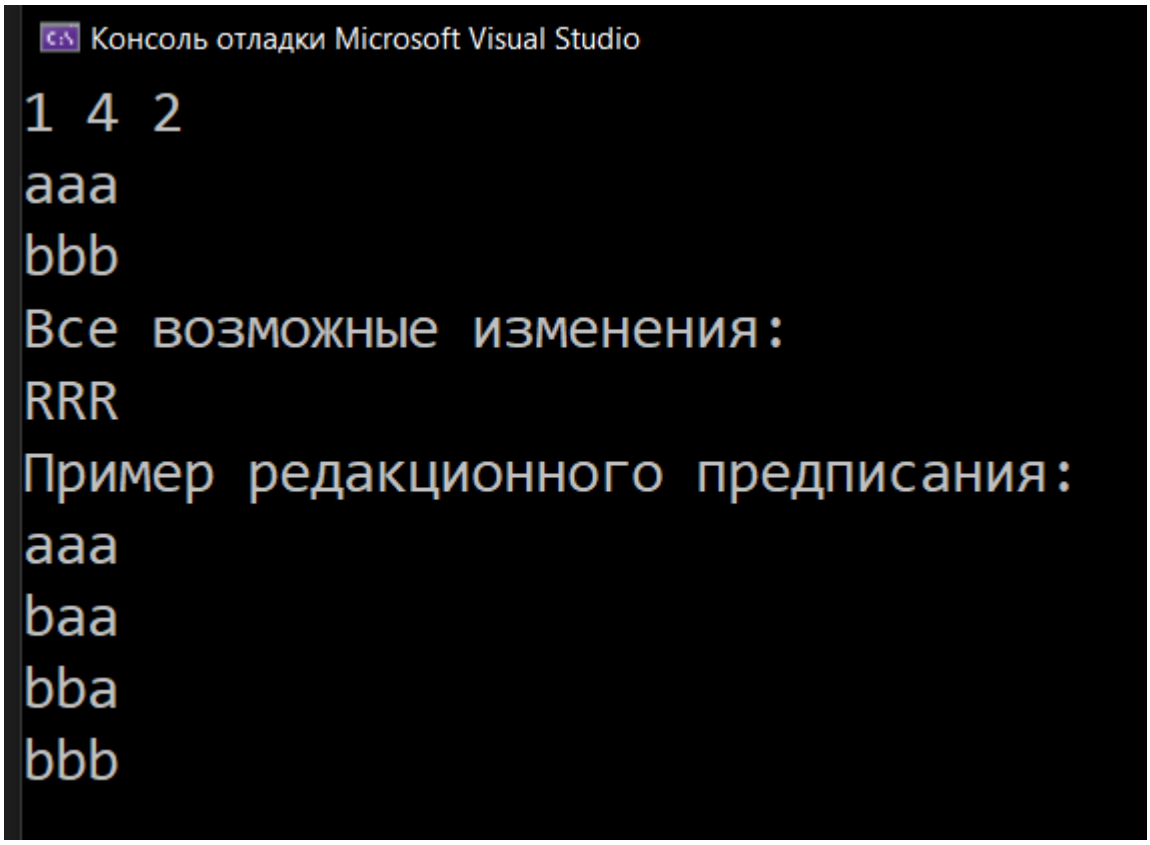

Рисунок 1 – показ базового случая с разной стоимостью

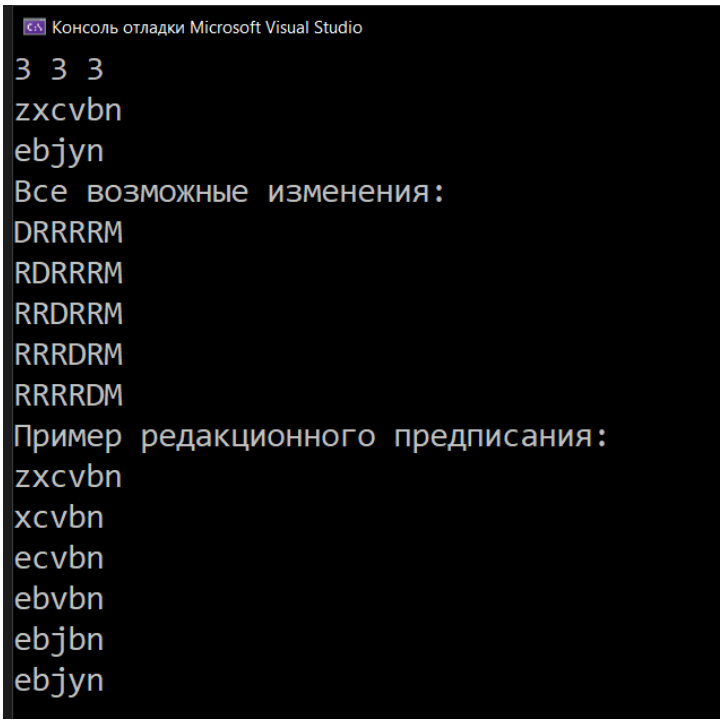

Рисунок 2 – показ с одинаковой стоимостью

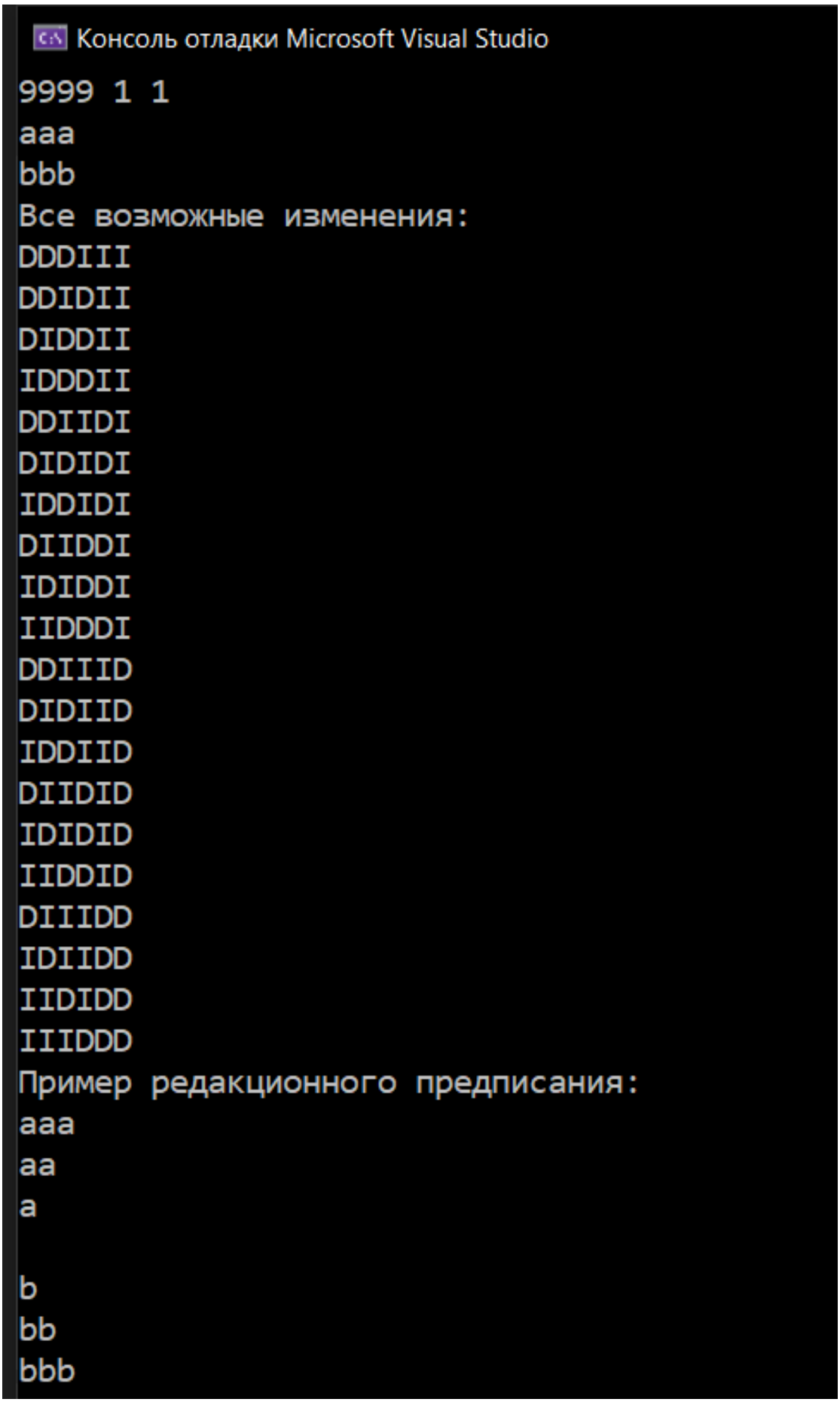

Рисунок 3 – показ случая с высокой стоимостью одной операции.

ВЫВОД

В ходе лабораторной работы был реализован алгоритм редактирования строк с учетом различных стоимостей операций, основанный на динамическом программировании. Алгоритм корректно определяет оптимальные пути преобразования и применяет их к строкам. Проведённый анализ подтвердил его эффективность и соответствие заявленной вычислительной сложности.

ПРИЛОЖЕНИЕ. ИСХОДНЫЙ КОД ПРОГРАММЫ

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

using namespace std;


struct Cell {
    int cost;
    char operation;
};


vector<vector<Cell>> build_dp(const string& A, const string& B,
int cost_replace, int cost_insert, int cost_delete) {
    int m = A.length();
    int n = B.length();
    vector<vector<Cell>> dp(m + 1, vector<Cell>(n + 1));

    dp[0][0] = { 0, 'M' };
    for (int i = 1; i <= m; ++i)
        dp[i][0] = { dp[i - 1][0].cost + cost_delete, 'D' };
    for (int j = 1; j <= n; ++j)
        dp[0][j] = { dp[0][j - 1].cost + cost_insert, 'I' };

    for (int i = 1; i <= m; ++i) {
        for (int j = 1; j <= n; ++j) {
            if (A[i - 1] == B[j - 1]) {
                dp[i][j] = { dp[i - 1][j - 1].cost, 'M' };
            }
            else {
                int replace_cost = dp[i - 1][j - 1].cost +
cost_replace;
                int insert_cost = dp[i][j - 1].cost +
cost_insert;
                int delete_cost = dp[i - 1][j].cost +
cost_delete;

                if (replace_cost <= insert_cost && replace_cost
<= delete_cost) {
                    dp[i][j] = { replace_cost, 'R' };
                }
                else if (insert_cost <= replace_cost &&
insert_cost <= delete_cost) {
                    dp[i][j] = { insert_cost, 'I' };
                }
                else {
                    dp[i][j] = { delete_cost, 'D' };
                }
            }
```

```
            }
        }

        return dp;
}

void find_all_paths(const vector<vector<Cell>>& dp, const
string& A, const string& B,
        int i, int j, string current, vector<string>& all_paths,
        int cost_replace, int cost_insert, int cost_delete) {
        if (i == 0 && j == 0) {
            reverse(current.begin(), current.end());
            all_paths.push_back(current);
            return;
        }

        if (i > 0 && j > 0) {
            if (A[i - 1] == B[j - 1] && dp[i][j].cost == dp[i - 1][j
- 1].cost) {
                find_all_paths(dp, A, B, i - 1, j - 1, current +
'M', all_paths, cost_replace, cost_insert, cost_delete);
            }
            if (A[i - 1] != B[j - 1] && dp[i][j].cost == dp[i - 1][j
- 1].cost + cost_replace) {
                find_all_paths(dp, A, B, i - 1, j - 1, current +
'R', all_paths, cost_replace, cost_insert, cost_delete);
            }
        }
        if (j > 0 && dp[i][j].cost == dp[i][j - 1].cost +
cost_insert) {
            find_all_paths(dp, A, B, i, j - 1, current + 'I',
all_paths, cost_replace, cost_insert, cost_delete);
        }
        if (i > 0 && dp[i][j].cost == dp[i - 1][j].cost +
cost_delete) {
            find_all_paths(dp, A, B, i - 1, j, current + 'D',
all_paths, cost_replace, cost_insert, cost_delete);
        }
}

void apply_operations(const string& A, const string& B, const
string& operations) {
        string result = A;
        int i = 0, j = 0;


        cout << result << endl;

        for (char op : operations) {
            if (op == 'M') {
                ++i;
                ++j;
            }
```

```
            else if (op == 'R') {
                result[i] = B[j];
                ++i;
                ++j;
                cout << result << endl;
            }
            else if (op == 'I') {
                result.insert(result.begin() + i, B[j]);
                ++i;
                ++j;
                cout << result << endl;
            }
            else if (op == 'D') {
                result.erase(result.begin() + i);
                cout << result << endl;
            }
        }
    }

    int main() {
        setlocale(LC_ALL, "Russian");
        int cost_replace, cost_insert, cost_delete;
        string A, B;

        cin >> cost_replace >> cost_insert >> cost_delete;
        cin >> A >> B;

        auto dp = build_dp(A, B, cost_replace, cost_insert,
    cost_delete);

        vector<string> all_paths;
        find_all_paths(dp, A, B, A.length(), B.length(), "",
    all_paths, cost_replace, cost_insert, cost_delete);

        cout << "Все возможные изменения:" << endl;
        for (const string& path : all_paths)
            cout << path << endl;

        if (!all_paths.empty()) {
            cout << "Пример редакционного предписания:" << endl;
            apply_operations(A, B, all_paths[0]);
        }

        return 0;
    }
```