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: Динамическое программирование

Студент гр. 3343 _____ Пименов П.В.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Реализовать алгоритм Вагнера-Фишера для нахождения редакционного расстояния и предписания.

Задание.

Индивидуальный вариант: 13

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.

Задание 1.

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

replace(ε, a, b) – заменить символ a на символ b.

insert(ε, a) – вставить в строку символ a (на любую позицию).

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:

replace(ε, a, b) – заменить символ a на символ b.

insert(ε, a) – вставить в строку символ a (на любую позицию).

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.

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, $1 \leq |S| \leq 2550$).

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

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

Выполнение работы.

Вариант 13. Для решения задач реализован алгоритм Вагнера-Фишера.

Описание реализованных функций:

- vector<vector<int>> getMatrix(const string& s1, const string& s2, int insertCost, int deleteCost, int replaceCost) – вычисляет матрицу расстояний с помощью алгоритма Вагнера-Фишера по заданным параметрам (двум строкам и ценам операций. Возвращает вычисленную матрицу. На каждом шаге сравнивает предыдущие символы, если они равны (match), то минимальное расстояние на текущий шаг – прошлое по диагонали. Если символы не равны, то берет минимальное из окрестности (по трем операциям: вставка, удаление, замена).

- string getPath(const vector<vector<int>>& D, const string& s1, const string& s2, int insertCost, int deleteCost, int replaceCost) – вычисляет редакционное предписание по заданной матрице расстояний. Действует «снизу вверх», начиная из самой правой нижней клетки (значение которой равно редакционному расстоянию). Действует циклом, пока не дойдем до пустой строки. На каждом шаге цикла вычисляет использованную операцию и записывает ее в строку операций. В конце работы разворачивает строку операций, поскольку шли «от конца».
- void getAllPaths(const vector<vector<int>>& D, const string& s1, const string& s2, int insertCost, int deleteCost, int replaceCost, int i, int j, string& currentPath, vector<string>& allPaths) – находит все редакционные предписания минимальной стоимости по заданным параметрам (задание индивидуального варианта). Действует методом бэктрекинга. Начинаем так же с нижней правой клетки и двигаемся во всех направлениях до левой верхней. Однако для того, чтобы найти все пути используется бэктрекинг (записываем операцию, запускаем метод опять в новой конфигурации, отменяем операцию). На каждом шаге проверяются именно подходящие по признакам операции, что дает на выходе только минимальные пути. Когда доходим до левой верхней клетки, записываем путь.
- void printPath(const string& path, const string& s1, const string& s2) – печатает редакционное предписание для двух строк в виде таблицы и строки превращений

Сложность алгоритма:

- По времени: O(MN), где M и N – размеры строк, поскольку требуется вычислить матрицу таких размеров для нахождения редакционного расстояния
- По памяти: O(MN) для хранения матрицы

Тестирование:

Входные данные	Выходные данные	Комментарий
1 1 1 abc ghj	3	Расстояние, успех
10 1 1 abc ghj	6	Расстояние, успех
1 2 1 abc bcgha	7	Расстояние, успех
1 1 1 abc ghj	RRR	Предписание, успех
10 1 1 abc ghj	IIIDDD	Предписание, успех
1 2 1 abc bcgha	DMMIII	Предписание, успех
2 1 1 abc bd	DMID DMDI DMR	Все предписания, успех

Выводы.

Реализован алгоритм Вагнера-Фишера для нахождения редакционного расстояния и предписания.

ПРИЛОЖЕНИЕ А

Файл main.cpp:

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

using namespace std;

vector<vector<int>> getMatrix(const string& s1, const string& s2,
                              int insertCost, int deleteCost, int replaceCost) {
    cout << "Calculating Levenshtein distance matrix" << endl << endl;
    size_t M = s1.size(), N = s2.size();
    vector<vector<int>> D(M + 1, vector<int>(N + 1, 0));

    for (int i = 1; i <= N; ++i) {
        D[0][i] = D[0][i - 1] + insertCost;
    }

    for (int i = 1; i <= M; ++i) {
        D[i][0] = D[i - 1][0] + deleteCost;
    }

    for (int i = 1; i <= M; ++i) {
        for (int j = 1; j <= N; ++j) {
            cout << "Comparing " << s1[i - 1] << " and " << s2[j - 1] << endl;
            if (s1[i - 1] != s2[j - 1]) {
                cout << "Symbols do not match" << endl;
                cout << "Insert cost: " << D[i][j - 1] + insertCost << endl;
                cout << "Delete cost: " << D[i - 1][j] + deleteCost << endl;
                cout << "Replace cost: " << D[i - 1][j - 1] + replaceCost
                     << endl;
                D[i][j] =
                    min(min(D[i - 1][j] + deleteCost, D[i][j - 1] + insertCost),
                        D[i - 1][j - 1] + replaceCost);
            } else {
                cout << "Symbols match" << endl;
                D[i][j] = D[i - 1][j - 1];
            }
            cout << "Cost D[" << i << "][" << j << "]: " << D[i][j] << endl
                 << endl;
        }
    }
    cout << "Matrix D:" << endl;
    for (int i = 0; i <= M; ++i) {
        for (int j = 0; j <= N; ++j) {
            cout << D[i][j] << " ";
        }
        cout << endl;
    }
    cout << endl;
    return D;
}

string getPath(const vector<vector<int>>& D, const string& s1, const string& s2,
               int insertCost, int deleteCost, int replaceCost) {
    string path;
    int i = s1.size(), j = s2.size();

    while (i > 0 || j > 0) {
        cout << "i: " << i << ", j: " << j << endl;
        if (i > 0 && j > 0 && s1[i - 1] == s2[j - 1] &&
```

```
            D[i][j] == D[i - 1][j - 1]) {  // Match
            cout << "Match" << endl;
            path.push_back('M');
            --i;
            --j;
        } else if (i > 0 && D[i][j] == D[i - 1][j] + deleteCost) {  // Delete
            cout << "Delete" << endl;
            path.push_back('D');
            --i;
        } else if (j > 0 && D[i][j] == D[i][j - 1] + insertCost) {  // Insert
            cout << "Insert" << endl;
            path.push_back('I');
            --j;
        } else if (i > 0 && j > 0 &&
                   D[i][j] == D[i - 1][j - 1] + replaceCost) {  // Replace
            cout << "Replace" << endl;
            path.push_back('R');
            --i;
            --j;
        }
        cout << endl;
    }

    reverse(path.begin(), path.end());
    cout << "Path: " << path << endl;
    return path;
}

void getAllPaths(const vector<vector<int>>& D, const string& s1,
                 const string& s2, int insertCost, int deleteCost,
                 int replaceCost, int i, int j, string& currentPath,
                 vector<string>& allPaths) {
    if (i == 0 && j == 0) {
        cout << "Reached start, storing path: " << currentPath << endl;
        allPaths.push_back(currentPath);  // Reached start, store the path
        return;
    }

    if (i > 0 && j > 0 && s1[i - 1] == s2[j - 1] &&
        D[i][j] == D[i - 1][j - 1]) {  // Match
        cout << "Match" << endl;
        currentPath.push_back('M');
        getAllPaths(D, s1, s2, insertCost, deleteCost, replaceCost, i - 1,
                    j - 1, currentPath, allPaths);
        currentPath.pop_back();
    }

    if (i > 0 && D[i][j] == D[i - 1][j] + deleteCost) {  // Delete
        cout << "Delete" << endl;
        currentPath.push_back('D');
        getAllPaths(D, s1, s2, insertCost, deleteCost, replaceCost, i - 1, j,
                    currentPath, allPaths);
        currentPath.pop_back();
    }

    if (j > 0 && D[i][j] == D[i][j - 1] + insertCost) {  // Insert
        cout << "Insert" << endl;
        currentPath.push_back('I');
        getAllPaths(D, s1, s2, insertCost, deleteCost, replaceCost, i, j - 1,
                    currentPath, allPaths);
        currentPath.pop_back();
    }
```

```
    if (i > 0 && j > 0 && s1[i - 1] != s2[j - 1] &&
        D[i][j] == D[i - 1][j - 1] + replaceCost) {  // Replace
        cout << "Replace" << endl;
        currentPath.push_back('R');
        getAllPaths(D, s1, s2, insertCost, deleteCost, replaceCost, i - 1,
                    j - 1, currentPath, allPaths);
        currentPath.pop_back();
    }
}

void printPath(const string& path, const string& s1, const string& s2) {
    cout << "Printing path: " << path << endl;
    string operations, top, bottom;
    int i = 0, j = 0;

    for (char c : path) {
        operations.push_back(c);
        switch (c) {
            case 'M':
            case 'R':
                top.push_back(s1[i++]);
                bottom.push_back(s2[j++]);
                break;
            case 'I':
                top.push_back(' ');
                bottom.push_back(s2[j++]);
                break;
            case 'D':
                top.push_back(s1[i++]);
                bottom.push_back(' ');
                break;
        }
    }

    for (char c : operations) {
        cout << c << " ";
    }
    cout << endl;
    for (char c : top) {
        cout << c << " ";
    }
    cout << endl;
    for (char c : bottom) {
        cout << c << " ";
    }
    cout << endl << endl;

    string current = s1;
    i = 0, j = 0;

    cout << current;

    for (char c : path) {
        switch (c) {
            case 'M':
                i++;
                j++;
                break;
            case 'R':
                current[i] = s2[j];
                i++;
                j++;
                break;
```

```
            case 'I':
                current.insert(current.begin() + i, s2[j]);
                i++;
                j++;
                break;
            case 'D':
                current.erase(current.begin() + i);
                break;
        }
        cout << " -> " << current;
    }
    cout << endl;
}

int main() {
    int insertCost = 1, deleteCost = 1, replaceCost = 1;
    string s1, s2;

    cin >> replaceCost >> insertCost >> deleteCost;
    cin >> s1 >> s2;

    size_t M = s1.size(), N = s2.size();

    vector<vector<int>> D =
        getMatrix(s1, s2, insertCost, deleteCost, replaceCost);

    // Levenstein Distance (Stepik 1)
    // cout << D[M][N] << endl;

    // Levenstein Path (Stepik 2)
    // string path = getPath(D, s1, s2, insertCost, deleteCost, replaceCost);
    // cout << path << endl << s1 << endl << s2 << endl;

    // All Levenstein Paths (Individual Task)
    string currentPath;
    vector<string> allPaths;
    cout << "Calculating all paths" << endl;
    getAllPaths(D, s1, s2, insertCost, deleteCost, replaceCost, s1.size(),
                s2.size(), currentPath, allPaths);
    cout << endl;
    for (auto& path : allPaths) {
        reverse(path.begin(), path.end());
    }

    for (const auto& path : allPaths) {
        cout << path << endl;
    }

    cout << endl;
    printPath(allPaths[0], s1, s2);

    return 0;
}
```