**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. 3342 |  | Мохамед М.Х. |
| Преподаватель |  | Виноградова Е.В. |

Санкт-Петербург

2025

**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 – строка A.

Третья строка – строка B.

**Выходные данные:**

Одно число – минимальная стоимость операций.

**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.

**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.

**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| ≤ 2550).

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

**Параметры выходных данных:**

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

Индивидуальный вариант 5а. Добавляется 4-я операция со своей стоимостью: удаление двух последовательных разных символов.

**Описание алгоритма**

Алгоритм вычисления редакционного расстояния, известный как расстояние Левенштейна, позволяет определить минимальное число правок, требуемых для приведения двух текстовых последовательностей к идентичному виду. В основе метода лежит подсчет элементарных правок: добавления, исключения и модификации отдельных знаков. Данный подход нашел применение в различных сферах, включая анализ генетических цепочек, лингвистические исследования и системы автоматического исправления опечаток.

Суть метода, предложенного Вагнером и Фишером, базируется на последовательном вычислении значений специальной таблицы. Каждый элемент D[i][j] этой таблицы содержит значение минимальных затрат на преобразование начального отрезка первой строки длиной i в начальный отрезок второй строки длиной j. Для заполнения таблицы применяется принцип динамического программирования, при котором решение сложной задачи разбивается на последовательность более простых подзадач.

Исходные значения таблицы определяются следующим образом:

* Первая строка соответствует стоимости последовательного добавления всех символов целевой строки
* Первый столбец отражает затраты на последовательное удаление всех символов исходной строки

Конечный результат вычислений содержится в последней ячейке таблицы и показывает минимальную совокупную стоимость преобразования всей исходной строки в целевую.

Описание работы алгоритма (задания 1, 3)

**1. Инициализация матрицы:**

* Создается двумерная матрица dp размером (lenA+1) × (lenB+1), где lenA и lenB - длины строк A и B соответственно
* Заполняются базовые случаи:
  + Первый столбец: стоимость преобразования строки A в пустую строку через операции удаления (dp[i][0] = i \* cost\_delete)
  + Первая строка: стоимость преобразования пустой строки в строку B через операции вставки (dp[0][j] = j \* cost\_insert)
* При активированной визуализации выводится начальное состояние матрицы

**2. Заполнение матрицы:**

* **Для каждого символа строки A (индекс i от 1 до lenA):**
  + Для каждого символа строки B (индекс j от 1 до lenB):
    - Совпадение символов:
      * Значение берется по диагонали (dp[i][j] = dp[i-1][j-1])
      * Визуализация показывает совпадающие символы и перенос значения
    - Несовпадение символов:
      * Вычисляются три варианта операций:
        1. Замена: dp[i-1][j-1] + cost\_replace
        2. Вставка: dp[i][j-1] + cost\_insert
        3. Удаление: dp[i-1][j] + cost\_delete
      * Выбирается минимальная стоимость
      * Визуализация отображает расчет всех вариантов и выбор оптимального
    - Специальная операция (если активирована):
      * Для последовательности двух разных символов рассматривается операция парного удаления
      * Сравнивается с текущим минимумом и при необходимости обновляется значение
  + После обработки каждого элемента выводится текущее состояние матрицы (при визуализации)

**3. Завершение:**

* Результат берется из правой нижней ячейки матрицы (dp[lenA][lenB])
* Визуализация показывает итоговое значение редакционного расстояния
* Алгоритм возвращает минимальную стоимость преобразования строки A в строку B с учетом всех возможных операций

Описание работы алгоритма (задание 2)

**1. Инициализация**

* Считываются стоимости операций:
  + cost\_replace - замена символа
  + cost\_insert - вставка символа
  + cost\_delete - удаление символа
  + cost\_delete\_two\_diff - удаление пары разных символов (если ENABLE\_SPECIAL\_OPERATION активирован)
* Создаются две матрицы размером (lenA+1)×(lenB+1):
  + dp - хранит минимальные стоимости преобразований
  + operations - фиксирует выполненные операции ('M', 'R', 'I', 'D', 'P')

**2. Базовые случаи**

* Первый столбец dp[i][0] инициализируется стоимостью удаления i символов
* Первая строка dp[0][j] инициализируется стоимостью вставки j символов

**3. Заполнение матрицы**

Для каждой пары символов (i,j):

1. При совпадении символов:
   * dp[i][j] = dp[i-1][j-1]
   * operations[i][j] = 'M' (match)
2. При несовпадении:
   * Вычисляются стоимости трех операций:
     + Замена: dp[i-1][j-1] + cost\_replace
     + Вставка: dp[i][j-1] + cost\_insert
     + Удаление: dp[i-1][j] + cost\_delete
   * Выбирается операция с минимальной стоимостью
3. При активированной специальной операции:
   * Если возможно удаление пары разных символов (i ≥ 2 и A[i-1] ≠ A[i-2]):
     + Вычисляется стоимость dp[i-2][j] + cost\_delete\_two\_diff
     + Если эта стоимость меньше текущей, обновляется значение и операция ('P')

**4. Восстановление последовательности операций**

* Обратный проход от dp[lenA][lenB] к dp[0][0]:
  + По матрице operations определяется тип операции
  + В зависимости от операции изменяются индексы i и j
  + Операции сохраняются в обратном порядке
* Полученная последовательность разворачивается

**5. Результат**

* Возвращается:
  + Последовательность операций преобразования
  + Исходная строка A
  + Целевая строка B

**Сложность по времени:**

**Задание 1, 3**

**Общая сложность:**O(n \* m), где n - длина строки A, m - длина строки B.

**Пояснение:**

1. Инициализация DP-таблицы:
   * Создание таблицы размером (n+1) x (m+1): O(n \* m)
   * Заполнение базовых случаев (первая строка и столбец): O(n + m)
2. Заполнение DP-таблицы:
   * Вложенные циклы по i (1..n) и j (1..m): O(n \* m)
   * Внутри циклов выполняются операции за O(1):
     + Проверка на совпадение символов
     + Вычисление стоимости replace/insert/delete
     + Опциональная проверка специальной операции (если ENABLE\_SPECIAL\_OPERATION)

**Задание 2**

**Общая сложность:** O(n \* m), где n — длина строки A, m — длина строки B.

**Пояснение:**

**1. Инициализация DP-таблиц**

* dp (размер (n+1) × (m+1)) и operations (размер (n+1) × (m+1)).
* Заполнение базовых случаев:
  + dp[i][0] (удаление всех символов из **A**): **O(n)**
  + dp[0][j] (вставка всех символов в **B**): **O(m)**
* **Итого:** **O(n + m)** (не доминирует над **O(n × m)**).

**2. Заполнение DP-таблицы**

* **Вложенные циклы** по i (от 1 до n) и j (от 1 до m): **O(n × m)**.
* Внутри каждой итерации:
  + Проверка совпадения символов (A[i-1] == B[j-1]): **O(1)**
  + Вычисление стоимости операций (replace, insert, delete): **O(1)**
  + Опциональная проверка **специальной операции** (удаление пары символов):
    - Проверка if (ENABLE\_SPECIAL\_OPERATION && i >= 2 && A[i-1] != A[i-2]): **O(1)**
    - Обновление стоимости: **O(1)**
  + **Итого на одну итерацию:** **O(1)** (не влияет на асимптотику).

**3. Восстановление последовательности операций**

* while (i > 0 || j > 0) — в худшем случае **O(n + m)** (двигаемся по диагонали).
* reverse(sequence) — **O(k)**, где k = O(n + m) (длина последовательности).
* **Итого:** **O(n + m)** (не доминирует).

**Сложность по памяти:**

**Задание 1 и 3**

* **O(n × m)** (хранение DP-матрицы dp размером (n+1) × (m+1))

**Пояснение:**

* 1. dp — матрица размером (lenA + 1) × (lenB + 1) = O(n × m).
  2. Входные строки A и B: **O(n + m)** *(не доминирует)*.
  3. Остальные переменные (cost\_\*, i, j и т.д.): **O(1)**.

**Итог:** Доминирующий фактор — **матрица**dp → **O(n × m)**.

**Задание 2**

* **O(n × m)** (хранение двух матриц dp и operations размером (n+1) × (m+1))

**Пояснение:**

1. dp — матрица стоимости операций: **O(n × m)**
2. operations — матрица для хранения операций: **O(n × m)**
3. Остальное (A, B, sequence и временные переменные): **O(n + m)** *(не доминирует)*

**Итог:** Память определяется размером DP-таблиц → **O(n × m)**.

**Описание функций и структур данных**

**1. Функция**printMatrix

**Назначение:** Визуализирует матрицу динамического программирования и операций в табличном формате.

**Параметры:**

* dp – матрица расстояний (тип: vector<vector<int>>)
* ops – матрица операций (тип: vector<vector<char>>)
* A, B – исходные строки (тип: string)
* i, j – координаты текущей ячейки для подсветки (тип: int)

**Возвращаемое значение:** void

**2. Функция**main

**Назначение:** Основная функция, реализующая алгоритм Левенштейна с возможностью визуализации и поддержкой специальных операций.  
 **Параметры (ввод через**cin**):**

* cost\_replace, cost\_insert, cost\_delete – стоимости базовых операций (тип: int)
* cost\_delete\_two\_diff – стоимость удаления пары разных символов (тип: int, опционально)
* A, B – входные строки для сравнения (тип: string)

**Возвращаемое значение:** int (код завершения программы, 0 при успешном выполнении)

Разработанный программный код см. в приложении А.

**Тестирование**

Результаты тестирования для заданий 1, 2 и 3 представлены в табл. 1, 2 и 3 соответственно. Результаты тестирования заданий с учетом варианта индивидуализации представлены в таблице 4.

Таблица 1 – Результаты тестирования для задания 1

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | 4 5 6  locker  pocket | 8 |
|  | 1 1 1  entrance  reenterable | 5 |
|  | 1 2 3  cat  mat | 1 |

Таблица 2 – Результаты тестирования для задания 2

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | 4 5 6  locker  pocket | RMMMMR  locker  pocket |
|  | 1 1 1  entrance  reenterable | IIMMMIMMRRM  entrance  reenterable |
|  | 1 2 3  cat  mat | RMM  cat  mat |

Таблица 3 – Результаты тестирования для задания 3

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | locker  pocket | 2 |
|  | entrance  reenterable | 5 |
|  | cat  mat | 1 |

Таблица 4 – Результаты тестирования с учетом варианта индивидуализации

|  |  |  |
| --- | --- | --- |
| № п/п | Входные данные | Выходные данные |
|  | 3 5 5  4  abcd  ad | 4 |
|  | 3 4 5  1  abcdef  af | MPPM  abcdef  af |

**Приложение А  
Исходный код программы**

Название файла: task1.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <climits>

#include <iomanip>

// Feature flags - can be toggled true/false

#define ENABLE\_SPECIAL\_OPERATION false // Enables pair deletion operation

#define ENABLE\_VISUALIZATION false // Enables step-by-step DP matrix display

using namespace std;

// Visualizes the DP matrix with current cell highlighted

void visualizeDP(const vector<vector<int>>& dp, const string& A, const string& B, int i, int j) {

cout << "\nDP matrix state at (" << i << "," << j << "):\n";

cout << " ";

for (char c : B) cout << setw(4) << c; // Print B string as header

cout << endl;

for (int x = 0; x < dp.size(); ++x) {

cout << (x > 0 ? A[x-1] : ' ') << " "; // Print A chars as row labels

for (int y = 0; y < dp[x].size(); ++y) {

if (x == i && y == j) {

cout << "[" << setw(2) << dp[x][y] << "]"; // Highlight current cell

} else {

cout << setw(4) << dp[x][y]; // Regular cell display

}

}

cout << endl;

}

cout << endl;

}

int main() {

// Read operation costs

int cost\_replace, cost\_insert, cost\_delete;

cin >> cost\_replace >> cost\_insert >> cost\_delete;

int cost\_delete\_two\_diff = 0;

// Read special operation cost if enabled

if (ENABLE\_SPECIAL\_OPERATION) {

cin >> cost\_delete\_two\_diff;

}

// Read input strings

string A, B;

cin >> A >> B;

int lenA = A.size();

int lenB = B.size();

// DP table initialization

vector<vector<int>> dp(lenA + 1, vector<int>(lenB + 1, 0));

// Base cases:

// Converting to empty string (delete all)

for (int i = 0; i <= lenA; ++i) {

dp[i][0] = i \* cost\_delete;

}

// Building from empty string (insert all)

for (int j = 0; j <= lenB; ++j) {

dp[0][j] = j \* cost\_insert;

}

// Show initial DP state if visualization enabled

if (ENABLE\_VISUALIZATION) {

cout << "Initial DP matrix:" << endl;

visualizeDP(dp, A, B, 0, 0);

}

// Fill DP table

for (int i = 1; i <= lenA; ++i) {

for (int j = 1; j <= lenB; ++j) {

if (A[i-1] == B[j-1]) {

// Characters match - carry previous diagonal value

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

if (ENABLE\_VISUALIZATION) {

cout << "Characters match: A[" << i-1 << "] = B[" << j-1 << "] = " << A[i-1] << endl;

cout << "dp[" << i << "][" << j << "] = dp[" << i-1 << "][" << j-1 << "] = " << dp[i][j] << endl;

}

} else {

// Calculate standard operations costs

int replace\_cost = dp[i-1][j-1] + cost\_replace;

int insert\_cost = dp[i][j-1] + cost\_insert;

int delete\_cost = dp[i-1][j] + cost\_delete;

dp[i][j] = min({replace\_cost, insert\_cost, delete\_cost});

if (ENABLE\_VISUALIZATION) {

cout << "Calculating dp[" << i << "][" << j << "]:" << endl;

cout << " Replace cost: " << replace\_cost << endl;

cout << " Insert cost: " << insert\_cost << endl;

cout << " Delete cost: " << delete\_cost << endl;

cout << " Selected minimum: " << dp[i][j] << endl;

}

}

// Special operation: delete two different consecutive chars

if (ENABLE\_SPECIAL\_OPERATION && i >= 2 && A[i-1] != A[i-2]) {

int special\_delete\_cost = dp[i-2][j] + cost\_delete\_two\_diff;

dp[i][j] = min(dp[i][j], special\_delete\_cost);

if (ENABLE\_VISUALIZATION) {

cout << " Special delete pair cost: " << special\_delete\_cost << endl;

cout << " Updated minimum: " << dp[i][j] << endl;

}

}

// Display current DP state if visualization enabled

if (ENABLE\_VISUALIZATION) {

visualizeDP(dp, A, B, i, j);

cout << "----------------------------------------" << endl;

}

}

}

// Output final result

if(ENABLE\_VISUALIZATION) {

cout << "Final edit distance: ";

}

cout << dp[lenA][lenB] << endl;

return 0;

}

Название файла: task2.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <iomanip>

// Feature flags - can be toggled true/false

#define ENABLE\_SPECIAL\_OPERATION false // Enables pair deletion operation

#define ENABLE\_VISUALIZATION false // Enables step-by-step DP matrix display

using namespace std;

void printMatrix(const vector<vector<int>>& dp, const vector<vector<char>>& ops,

const string& A, const string& B, int i, int j) {

// Print header with B string

cout << " ";

for (char c : B) cout << setw(4) << c;

cout << "\n +" << string(5\*(B.size()+1), '-') << "+\n";

// Print matrix with A string as row labels

for (int x = 0; x <= A.size(); ++x) {

cout << (x > 0 ? A[x-1] : ' ') << " |";

for (int y = 0; y <= B.size(); ++y) {

// Highlight current cell

if (x == i && y == j) cout << "[";

else cout << " ";

// Print value and operation

cout << setw(2) << dp[x][y] << ops[x][y];

if (x == i && y == j) cout << "]";

else cout << " ";

}

cout << "|\n";

}

cout << " +" << string(5\*(B.size()+1), '-') << "+\n\n";

}

int main() {

// Read operation costs

int cost\_replace, cost\_insert, cost\_delete;

cin >> cost\_replace >> cost\_insert >> cost\_delete;

int cost\_delete\_two\_diff = 0;

// Read special operation cost if enabled

if (ENABLE\_SPECIAL\_OPERATION) {

cin >> cost\_delete\_two\_diff;

}

string A, B;

cin >> A >> B;

int lenA = A.size();

int lenB = B.size();

// DP table for costs

vector<vector<int>> dp(lenA + 1, vector<int>(lenB + 1, 0));

// Table to track operations

vector<vector<char>> operations(lenA + 1, vector<char>(lenB + 1, ' '));

// Initialize base cases

for (int i = 0; i <= lenA; ++i) {

dp[i][0] = i \* cost\_delete;

operations[i][0] = 'D'; // Delete operation

}

for (int j = 0; j <= lenB; ++j) {

dp[0][j] = j \* cost\_insert;

operations[0][j] = 'I'; // Insert operation

}

if (ENABLE\_VISUALIZATION) {

cout << "=== INITIAL MATRIX ===\n";

printMatrix(dp, operations, A, B, 0, 0);

}

// Fill DP tables

for (int i = 1; i <= lenA; ++i) {

for (int j = 1; j <= lenB; ++j) {

if (A[i-1] == B[j-1]) {

// Characters match

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

operations[i][j] = 'M'; // Match operation

if (ENABLE\_VISUALIZATION) {

cout << "MATCH at (" << i << "," << j << "): "

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

}

} else {

// Calculate standard operation costs

int replace\_cost = dp[i-1][j-1] + cost\_replace;

int insert\_cost = dp[i][j-1] + cost\_insert;

int delete\_cost = dp[i-1][j] + cost\_delete;

// Find minimum cost operation

if (replace\_cost <= insert\_cost && replace\_cost <= delete\_cost) {

dp[i][j] = replace\_cost;

operations[i][j] = 'R';

} else if (insert\_cost <= replace\_cost && insert\_cost <= delete\_cost) {

dp[i][j] = insert\_cost;

operations[i][j] = 'I';

} else {

dp[i][j] = delete\_cost;

operations[i][j] = 'D';

}

if (ENABLE\_VISUALIZATION) {

cout << "OPERATIONS at (" << i << "," << j << "):\n"

<< " Replace: " << replace\_cost << "\n"

<< " Insert: " << insert\_cost << "\n"

<< " Delete: " << delete\_cost << "\n"

<< " Selected: " << operations[i][j]

<< " (cost=" << dp[i][j] << ")\n";

}

// Check for special pair deletion operation

if (ENABLE\_SPECIAL\_OPERATION && i >= 2 && A[i-1] != A[i-2]) {

int pair\_delete\_cost = dp[i-2][j] + cost\_delete\_two\_diff;

if (pair\_delete\_cost < dp[i][j]) {

dp[i][j] = pair\_delete\_cost;

operations[i][j] = 'P'; // Pair delete operation

if (ENABLE\_VISUALIZATION) {

cout << " SPECIAL PAIR DELETE: " << pair\_delete\_cost

<< " (delete " << A[i-2] << A[i-1] << ")\n";

}

}

}

}

if (ENABLE\_VISUALIZATION) {

printMatrix(dp, operations, A, B, i, j);

cout << "----------------------------------------\n";

}

}

}

// Reconstruct operation sequence

string sequence;

int i = lenA, j = lenB;

while (i > 0 || j > 0) {

char op = operations[i][j];

sequence += op;

if (op == 'M' || op == 'R') {

i--; j--;

} else if (op == 'I') {

j--;

} else if (op == 'D') {

i--;

} else if (op == 'P') {

i -= 2;

}

}

reverse(sequence.begin(), sequence.end());

// Output results

if (ENABLE\_VISUALIZATION) {

cout << "\n=== FINAL RESULTS ===\n";

cout << "Edit distance: " << dp[lenA][lenB] << "\n";

cout << "Operation sequence: " << sequence << "\n";

cout << "Legend: M=Match, R=Replace, I=Insert, D=Delete, P=PairDelete\n";

cout << "Original string: " << A << "\n";

cout << "Target string: " << B << "\n";

} else {

cout << sequence << "\n" << A << "\n" << B << "\n";

}

return 0;

}

Название файла: task3.cpp

#include <iostream>

#include <vector>

#include <algorithm>

#include <string>

#include <climits>

#include <iomanip>

// Feature flags - can be toggled true/false

#define ENABLE\_SPECIAL\_OPERATION false // Enables pair deletion operation

#define ENABLE\_VISUALIZATION false // Enables step-by-step DP matrix display

using namespace std;

// Visualizes the DP matrix with current cell highlighted

void visualizeDP(const vector<vector<int>>& dp, const string& A, const string& B, int i, int j) {

cout << "\nDP matrix state at (" << i << "," << j << "):\n";

cout << " ";

for (char c : B) cout << setw(4) << c; // Print B string as header

cout << endl;

for (int x = 0; x < dp.size(); ++x) {

cout << (x > 0 ? A[x-1] : ' ') << " "; // Print A chars as row labels

for (int y = 0; y < dp[x].size(); ++y) {

if (x == i && y == j) {

cout << "[" << setw(2) << dp[x][y] << "]"; // Highlight current cell

} else {

cout << setw(4) << dp[x][y]; // Regular cell display

}

}

cout << endl;

}

cout << endl;

}

int main() {

// Read operation costs

int cost\_replace, cost\_insert, cost\_delete;

cost\_replace = 1;

cost\_insert = 1;

cost\_delete = 1;

int cost\_delete\_two\_diff = 0;

// Read special operation cost if enabled

if (ENABLE\_SPECIAL\_OPERATION) {

cin >> cost\_delete\_two\_diff;

}

// Read input strings

string A, B;

cin >> A >> B;

int lenA = A.size();

int lenB = B.size();

// DP table initialization

vector<vector<int>> dp(lenA + 1, vector<int>(lenB + 1, 0));

// Base cases:

// Converting to empty string (delete all)

for (int i = 0; i <= lenA; ++i) {

dp[i][0] = i \* cost\_delete;

}

// Building from empty string (insert all)

for (int j = 0; j <= lenB; ++j) {

dp[0][j] = j \* cost\_insert;

}

// Show initial DP state if visualization enabled

if (ENABLE\_VISUALIZATION) {

cout << "Initial DP matrix:" << endl;

visualizeDP(dp, A, B, 0, 0);

}

// Fill DP table

for (int i = 1; i <= lenA; ++i) {

for (int j = 1; j <= lenB; ++j) {

if (A[i-1] == B[j-1]) {

// Characters match - carry previous diagonal value

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

if (ENABLE\_VISUALIZATION) {

cout << "Characters match: A[" << i-1 << "] = B[" << j-1 << "] = " << A[i-1] << endl;

cout << "dp[" << i << "][" << j << "] = dp[" << i-1 << "][" << j-1 << "] = " << dp[i][j] << endl;

}

} else {

// Calculate standard operations costs

int replace\_cost = dp[i-1][j-1] + cost\_replace;

int insert\_cost = dp[i][j-1] + cost\_insert;

int delete\_cost = dp[i-1][j] + cost\_delete;

dp[i][j] = min({replace\_cost, insert\_cost, delete\_cost});

if (ENABLE\_VISUALIZATION) {

cout << "Calculating dp[" << i << "][" << j << "]:" << endl;

cout << " Replace cost: " << replace\_cost << endl;

cout << " Insert cost: " << insert\_cost << endl;

cout << " Delete cost: " << delete\_cost << endl;

cout << " Selected minimum: " << dp[i][j] << endl;

}

}

// Special operation: delete two different consecutive chars

if (ENABLE\_SPECIAL\_OPERATION && i >= 2 && A[i-1] != A[i-2]) {

int special\_delete\_cost = dp[i-2][j] + cost\_delete\_two\_diff;

dp[i][j] = min(dp[i][j], special\_delete\_cost);

if (ENABLE\_VISUALIZATION) {

cout << " Special delete pair cost: " << special\_delete\_cost << endl;

cout << " Updated minimum: " << dp[i][j] << endl;

}

}

// Display current DP state if visualization enabled

if (ENABLE\_VISUALIZATION) {

visualizeDP(dp, A, B, i, j);

cout << "----------------------------------------" << endl;

}

}

}

// Output final result

if(ENABLE\_VISUALIZATION) {

cout << "Final edit distance: ";

}

cout << dp[lenA][lenB] << endl;

return 0;

}