**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**

Вариант 3a.

|  |  |  |
| --- | --- | --- |
| Студентка гр. 3388 |  | Басик В.В. |
| Преподаватель |  | Жангиров Т.Р. |

Санкт-Петербург

2025

**Цель работы:**

Изучить алгоритм Вагнера-Фишера для нахождения редакционного расстояния Левенштейна. Реализовать задание в соответствии с вариантом.

**Задание.**

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.

**Sample Input:**

pedestal

stien

**Sample Output:**

7

**Реализация**  
Программа реализует три вариации алгоритма вычисления редакционного расстояния (расстояния Левенштейна) с дополнительными оптимизациями и возможностями:

1. **Task1:** Базовое вычисление стоимости преобразования строк с поддержкой операции двойной вставки.
2. **Task2:** Расширенное вычисление с восстановлением точной последовательности операций редактирования.
3. **Task3:** Оптимизированная по памяти версия алгоритма.

### Структуры и константы

**Константы:**

* const long long INF = 1e18;  
  Значение для представления "бесконечности" в матрицах динамического программирования.

### **Функции**

#### void printDP(const vector<vector<long long>>& dp, const string& A, const string& B)

**Назначение:**  
Визуализирует матрицу динамического программирования в читаемом формате.  
**Параметры:**

* dp: Матрица стоимостей преобразования.
* A: Исходная строка.
* B: Целевая строка.  
  **Вывод:**
* Заголовок с символами строки B (или ε для пустой строки).
* Строки, соответствующие символам A (или ε), с выровненными значениями матрицы.  
  **Особенности:**
* Автоматически заменяет INF на "∞" для наглядности.

#### void task1()

**Назначение:**  
Вычисляет минимальную стоимость преобразования строки A в строку B с поддержкой **двойной вставки** (оптимизация для повторяющихся символов).  
**Входные данные:**

1. Строка с параметрами:
   * 3 параметра: [замена, вставка, удаление].
   * 4 параметра: [замена, вставка, удаление, двойная\_вставка].
2. Исходная строка A.
3. Целевая строка B.

**Процесс:**

1. **Инициализация матрицы dp:**
   * dp[i][0] = i \* cost\_delete (удаление всех символов A).
   * dp[0][j] = j \* cost\_insert (вставка всех символов B).
2. **Заполнение матрицы:**  
   Для каждой позиции (i, j) вычисляются варианты:
   * **Совпадение/замена:** dp[i-1][j-1] + (A[i-1]==B[j-1] ? 0 : cost\_replace).
   * **Вставка:** dp[i][j-1] + cost\_insert.
   * **Удаление:** dp[i-1][j] + cost\_delete.
   * **Двойная вставка (если применимо):** dp[i][j-2] + cost\_double\_insert.
3. **Логирование:**
   * Вывод матрицы после инициализации и каждой строки.
   * Подробные комментарии для каждого шага вычислений.  
     **Результат:**  
     Финальная стоимость преобразования dp[n][m].

#### void task2()

**Назначение:**  
Расширение task1 с **восстановлением последовательности операций** через матрицу предков.  
**Дополнительные структуры:**

* parent[i][j]: Хранит (предыдущая\_строка, предыдущий\_столбец, операция).

**Операции:**

* 'M': Совпадение (Match).
* 'R': Замена (Replace).
* 'I': Вставка (Insert).
* 'D': Удаление (Delete).
* 'W': Двойная вставка (double insert).

**Процесс:**

1. Инициализация матриц dp и parent аналогично task1.
2. Для каждой позиции (i, j):
   * Фиксируется операция с минимальной стоимостью и обновляется parent.
3. **Восстановление пути:**
   * Обратный проход от (n, m) до (0, 0) с сохранением операций.
   * Разворот последовательности операций.  
     **Вывод:**

* Последовательность операций (например, "MIIRDD").

#### void task3()

**Назначение:**  
Оптимизированная по памяти реализация редакционного расстояния (алгоритм Вагнера-Фишера).  
**Особенности:**

* Использует **два массива** (prev, curr) вместо полной матрицы.
* Стандартные стоимости операций: замена = 1, вставка = 1, удаление = 1.  
  **Процесс:**

1. **Инициализация:**
   * prev[j] = j (вставка j символов).
2. **Обновление массива:**  
   Для каждого символа S[i-1]:
   * curr[0] = i (удаление i символов).
   * Для j от 1 до m:
     + Совпадение: curr[j] = prev[j-1].
     + Несовпадение: min(prev[j] + 1, curr[j-1] + 1, prev[j-1] + 1).
3. **Логирование:**
   * Вывод текущего состояния массива после обработки строки.  
     **Результат:**  
     Редакционное расстояние prev[m].

**Функция main():**

* Запрашивает у пользователя номер задачи (1, 2, 3).
* Вызывает соответствующую функцию (task1, task2, task3).
* Обрабатывает некорректный ввод.

**Особенности ввода:**

* Для задач 1 и 2: параметры в первой строке, строки A и B — в следующих.
* Для задачи 3: строки S и T в отдельных строках.
* Использует cin.ignore() для корректной обработки перевода строк.

**Оценка сложности алгоритма:**

**Общая характеристика:**  
Все задачи реализуют алгоритм Вагнера-Фишера для вычисления редакционного расстояния с различными оптимизациями. Основные параметры:

* n - длина первой строки (A или S)
* m - длина второй строки (B или T)

### Задача 1: Базовое вычисление стоимости

**Временная сложность:**

1. **Инициализация матрицы:**
   * Заполнение первого столбца: O(n)
   * Заполнение первой строки: O(m)
   * Итого: O(n + m)
2. **Основной цикл:**
   * Обработка каждой ячейки матрицы (n \* m элементов)
   * Для каждой ячейки выполняется:
     + Сравнение символов (O(1))
     + Вычисление 3 вариантов операций (O(1))
     + Проверка двойной вставки (O(1))
   * **Итог:** O(n \* m)
3. **Вывод матрицы:**
   * Вызывается после обработки каждой строки (n раз)
   * Каждый вывод занимает O(n \* m)
   * **Итог:** O(n² \* m)

**Общая временная сложность:** O(n² \* m) (доминирует вывод)

**Пространственная сложность:**

* Матрица dp размером (n+1) x (m+1): O(n \* m)

### Задача 2: Восстановление операций

**Временная сложность:**

1. **Инициализация:**
   * Аналогично задаче 1: O(n + m)
2. **Основной цикл:**
   * Обработка n \* m ячеек
   * Для каждой ячейки:
     + Проверка 4 вариантов операций (O(1))
     + Обновление матрицы предков (O(1))
   * **Итог:** O(n \* m)
3. **Восстановление пути:**
   * Проход от (n, m) к (0, 0): O(n + m)
4. **Вывод матрицы:**
   * Аналогично задаче 1: O(n² \* m)

**Общая временная сложность:** O(n² \* m)

**Пространственная сложность:**

* Матрица dp: O(n \* m)
* Матрица parent: O(n \* m) (хранит координаты и операцию)
* Строка операций ops: O(n + m)

### Задача 3: Оптимизированная версия

**Временная сложность:**

1. **Инициализация:**
   * Заполнение массива prev: O(m)
2. **Основной цикл:**
   * Обработка n строк
   * Для каждой строки обработка m элементов:
     + Сравнение символов (O(1))
     + Вычисление минимума из 3 значений (O(1))
   * **Итог:** O(n \* m)
3. **Вывод:**
   * После каждой строки вывод массива: O(n \* m)

**Общая временная сложность:** O(n \* m)

**Пространственная сложность:**

* Два массива prev и curr длиной (m+1): O(m)
* Не требует хранения полной матрицы

**Тестирование**

Таблица 1. Тестирование.

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 1 1 1 2  kitten sitting | 3 |
| 1 1 1 1  abc  abbc | MIMM |

**Вывод**

В ходе лабораторной работы была написана программа, реализующая алгоритм Вагнера-Фишера для поиска редакционного расстояния.