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»

Студент гр. 3343 _____ Отмахов Д. В.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Изучить работу алгоритма Вагнера-Фишера для построения матрицы расстояния Левенштейна и нахождения редакционного предписания.

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 – 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:

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

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|≤25501≤|*S*|≤2550).

Вторая строка входных данных содержит строку из строчных латинских букв. (T, 1≤|T|≤25501≤|*T*|≤2550).

Параметры выходных данных:

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

Sample Input:

pedestal

stien

Sample Output:

7

Вариант 14б.

Методом динамического программирования вычислить: длину наибольшей общей подстроки двух строк, вывести и саму эту подстроку.

Выполнение работы

Описание алгоритма

Расстояние Левенштейна показывает минимальное количество операций, необходимых для преобразования одного слова в другое, а алгоритм Вагнера-Фишера — это метод нахождения этого расстояния путем построения матрицы расстояний.

Заполнение матрицы расстояний

Матрица расстояний строится на основе двух входных строк. Каждое значение в этой матрице представляет расстояние Левенштейна для двух подстрок, полученных путем обрезания исходных строк по индексам строки и столбца.

1. Инициализация:

- d[0][0] = 0 – расстояние между двумя пустыми строками равно нулю.
- d[0][j] = j • insertion_cost – чтобы преобразовать пустую строку в подстроку второй строки, нужно выполнить j вставок.
- d[i][0] = i • deletion_cost – чтобы преобразовать подстроку первой строки в пустую строку, нужно выполнить i удалений.

2. Заполнение остальных ячеек матрицы:

Для всех остальных ячеек d[i][j], где i > 0 и j > 0, значение ячейки вычисляется как минимум из трех вариантов:

- d[i][j-1] + insert_cost – это значение слева, которое означает операцию вставки символа.
- d[i-1][j] + delete_cost – это значение сверху, которое означает операцию удаления символа.
- d[i-1][j-1] + replace_cost – это значение слева-сверху, которое означает операцию замены символов. Если символы совпадают, то цена замены равна 0.

Таким образом, значение в правом нижнем углу матрицы d[n][m] будет равно расстоянию Левенштейна для рассматриваемых строк.

Поиск редакционного предписания

После того как матрица заполнена, необходимо найти редакционное предписание — последовательность операций, которые преобразуют одну строку в другую. Для этого мы начинаем с конечного значения матрицы (в правом нижнем углу) и двигаемся к начальной ячейке (в левом верхнем углу), выбирая наиболее оптимальный путь. В каждом шаге выбираем минимум из трех возможных операций (вставка, удаление, замена) на основе значений ячеек слева, сверху и слева-сверху. В случае совпадения символов замена не требуется, и соответствующая операция пропускается.

Оценка сложности

Временная сложность алгоритма – $O(n * m)$, где n – длина первой строки, а m – длина второй. Сложность квадратичная, поскольку программа составляет матрицу расстояний, размером $(n + 1) * (m + 1)$.

Пространственная сложность алгоритма – $O(n * m)$, так как необходимо хранить матрицу размером $(n + 1) * (m + 1)$.

Код программы содержит реализацию следующих методов:

- *buildEditDistanceMatrix(costs: Costs, source: String, target: String) -> [[Int]]* – строит и заполняет матрицу расстояний Левенштейна для двух строк. Матрица вычисляется на основе стоимости операций вставки, удаления и замены.
- *editDistanceWithTrace(costs: Costs, source: String, target: String) -> (operations: String, source: String, target: String)* – находит минимальное расстояние Левенштейна между двумя строками и возвращает последовательность операций, которые преобразуют одну строку в другую.
- *levenshteinDistance(costs: Costs, source: String, target: String) -> Int* – реализует алгоритм Левенштейна для вычисления расстояния между двумя строками без отслеживания операций.
- *longestCommonSubstring(_ source: String, _ target: String) -> (length: Int, substring: String)* – находит длину и самую длинную общую подстроку между

двумя строками, используя динамическое программирование для поиска максимального совпадения символов.

Тестирование

Программа была протестирована на различных входных данных. Результаты представлены в Таблице 1.

Таблица 1.

Входные данные	Выходные данные (минимальное редакционное расстояние, последовательность операций, наибольшая общая подстрока)
abc abcdefgh 1 1 1	5 MMMIIIII abc
abcdefgh abc 1 1 1	MMMDDDDD 5 abc
abcccccd abcdefg 1 1 1	MMDMRRRR 5 abc
aaaaaa bbbbbbbbb 1 1 1	IIIRRRRRR 9
abcdddf aaabc 1 1 1	MDDRRRR 6 abc

```
Enter the first string:
abc
Enter the second string:
abcdefg
Enter the operation costs (replace insert delete) separated by spaces:
1 1 1
Initial Matrix:
0   1   2   3   4   5   6   7
1   0   0   0   0   0   0   0
2   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0

Matrix after row 1:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0

Matrix after row 2:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   1   0   1   2   3   4   5
3   0   0   0   0   0   0   0

Matrix after row 3:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   1   0   1   2   3   4   5
3   2   1   0   1   2   3   4
```

```
Tracing back operations:
Insert: g
Insert: f
Insert: e
Insert: d
Match: c == c
Match: b == b
Match: a == a
Edit operations sequence: MMMIIII
Initial Matrix:
0   1   2   3   4   5   6   7
1   0   0   0   0   0   0   0
2   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0

Matrix after row 1:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   0   0   0   0   0   0   0
3   0   0   0   0   0   0   0

Matrix after row 2:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   1   0   1   2   3   4   5
3   0   0   0   0   0   0   0

Matrix after row 3:
0   1   2   3   4   5   6   7
1   0   1   2   3   4   5   6
2   1   0   1   2   3   4   5
3   2   1   0   1   2   3   4

Levenshtein distance: 4
Longest common substring length: 3
Longest common substring: "abc"
Program ended with exit code: 0
```

Рисунок 1 – Результат работы программы.

Исследование

Исследуем эффективность алгоритма Вагнера-Фишера для построения матрицы расстояний на различных объёмах входных данных. Результаты исследования приведены в Таблице 2.

Таблица 2.

Размер первого слова	Размер второго слова	Произведение размеров	Затраченное время, с
10	10	100	0.000531
10	1000	10000	0.004331
1000	10	10000	0.004577
100	10000	1000000	0.345308
10000	100	1000000	0.332744
1000	1000	1000000	0.333181
100000	10	1000000	0.394613
10	1000000	10000000	3.509299

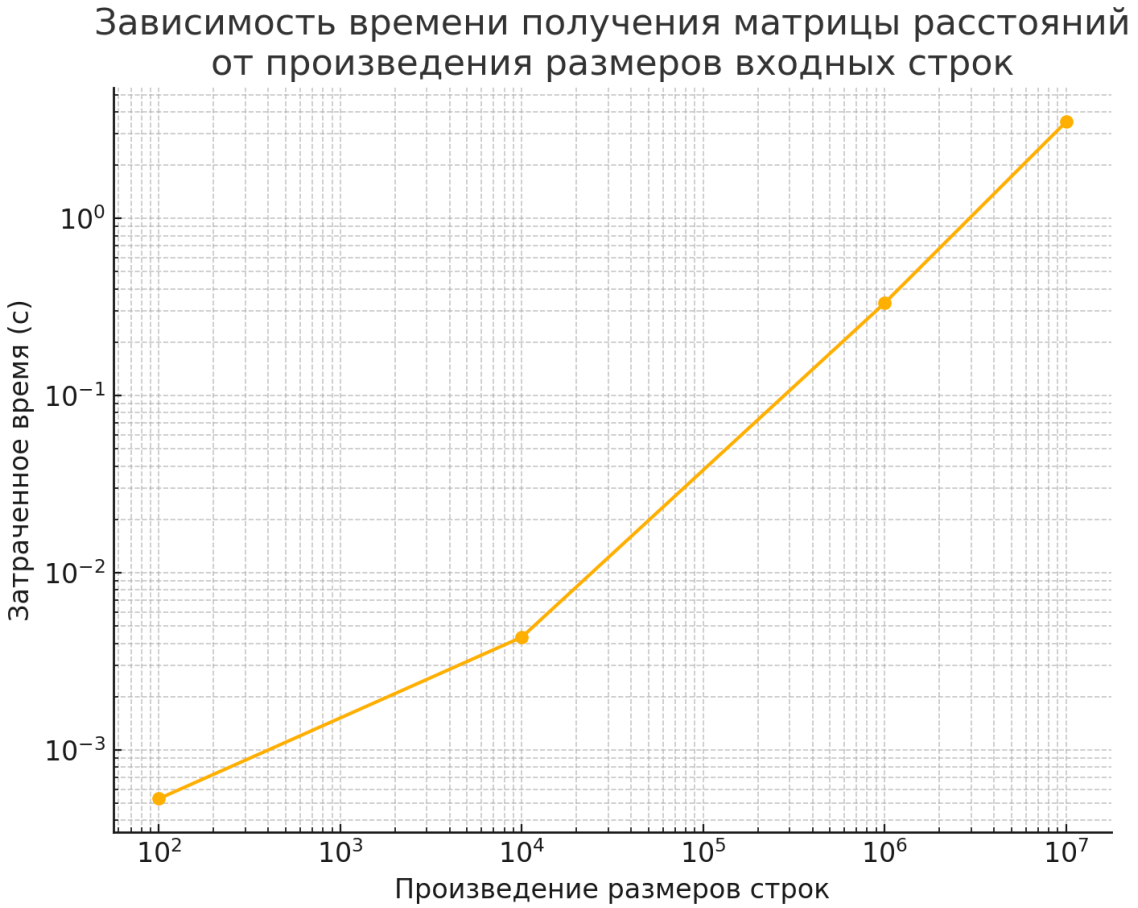

Рисунок 2 – График зависимости затраченного времени на получение матрицы расстояний от произведения размеров входных строк.

Можно сделать следующие выводы:

- Полученные результаты подтверждают временную оценку $O(n * m)$.
- Результат по времени не зависит от того, какая из строк длиннее.

Выводы

Во время выполнения лабораторной работы, была изучена работа алгоритма Вагнера-Фишера. Решены задачи поиска матрицы расстояния Левенштейна и нахождения редакционного предписания.

ПРИЛОЖЕНИЕ

ИСХОДНЫЙ КОД ПРОГРАММЫ

algorithms.swift

```
//
//  algorithms.swift
//  EditDistanceAlgorithms
//
//  Created by Danil Otmakhov on 27.04.2025.
//

import Foundation

enum EditOperation: String {
    case match = "M"
    case replace = "R"
    case insert = "I"
    case delete = "D"
}


struct Costs {
    let replace: Int
    let insert: Int
    let delete: Int

    init(replace: Int, insert: Int, delete: Int) {
        guard replace >= 0, insert >= 0, delete >= 0 else {
            fatalError("The cost of operations should be non-negative.")
        }
        self.replace = replace
        self.insert = insert
        self.delete = delete
    }
}


func buildEditDistanceMatrix(costs: Costs, source: String, target: String) ->
[[Int]] {
    let a = Array(source)
    let b = Array(target)
    let n = a.count
    let m = b.count

    var dp = Array(repeating: Array(repeating: 0, count: m + 1), count: n + 1)

    for i in 0...n {
        dp[i][0] = i * costs.delete
    }

    for j in 0...m {
        dp[0][j] = j * costs.insert
    }

    print("Initial Matrix:")
    printMatrix(dp)

    for i in 1...n {
        for j in 1...m {
            if a[i - 1] == b[j - 1] {
                dp[i][j] = dp[i - 1][j - 1]
```

```
            } else {
                let replace = dp[i - 1][j - 1] + costs.replace
                let insert = dp[i][j - 1] + costs.insert
                let delete = dp[i - 1][j] + costs.delete
                dp[i][j] = min(replace, insert, delete)
            }
        }
        print("Matrix after row \(i):")
        printMatrix(dp)
    }

    return dp
}


func editDistanceWithTrace(costs: Costs, source: String, target: String) -> (op-
erations: String, source: String, target: String) {
    let a = Array(source)
    let b = Array(target)
    let n = a.count
    let m = b.count

    let dp = buildEditDistanceMatrix(costs: costs, source: source, target: tar-
get)

    var operations: [EditOperation] = []
    var i = n
    var j = m

    print("Tracing back operations:")

    while i > 0 || j > 0 {
        if i > 0 && j > 0 && a[i - 1] == b[j - 1] {
            operations.append(.match)
            print("Match: \(a[i - 1]) == \(b[j - 1])")
            i -= 1
            j -= 1
        } else {
            if i > 0 && j > 0 && dp[i][j] == dp[i - 1][j - 1] + costs.replace {
                operations.append(.replace)
                print("Replace: \(a[i - 1]) -> \(b[j - 1])")
                i -= 1
                j -= 1
            } else if j > 0 && dp[i][j] == dp[i][j - 1] + costs.insert {
                operations.append(.insert)
                print("Insert: \(b[j - 1])")
                j -= 1
            } else if i > 0 && dp[i][j] == dp[i - 1][j] + costs.delete {
                operations.append(.delete)
                print("Delete: \(a[i - 1])")
                i -= 1
            }
        }
    }

    let operationsString = operations.reversed().map { $0.rawValue }.joined()

    return (operationsString, source, target)
}


func levenshteinDistance(costs: Costs, source: String, target: String) -> Int {
    let dp = buildEditDistanceMatrix(costs: costs, source: source, target: tar-
get)
```

```
    return dp[source.count][target.count]
}


func longestCommonSubstring(_ source: String, _ target: String) -> (length: Int,
substring: String) {
    if source.isEmpty || target.isEmpty {
        return (0, "")
    }

    let a = Array(source)
    let b = Array(target)
    let n = a.count
    let m = b.count

    var dp = Array(repeating: Array(repeating: 0, count: m + 1), count: n + 1)
    var maxLength = 0
    var endIndex = 0

    for i in 1...n {
        for j in 1...m {
            if a[i - 1] == b[j - 1] {
                dp[i][j] = dp[i - 1][j - 1] + 1
                if dp[i][j] > maxLength {
                    maxLength = dp[i][j]
                    endIndex = i
                }
            }
        }
    }

    let startIndex = endIndex - maxLength
    let substring = String(a[startIndex..<endIndex])

    return (maxLength, substring)
}


func printMatrix(_ matrix: [[Int]]) {
    for row in matrix {
        print(row.map { String($0) }.joined(separator: "\t"))
    }
    print()
```

main.swift

```
//
//  main.swift
//  EditDistanceAlgorithms
//
//  Created by Danil Otmakhov on 27.04.2025.
//

import Foundation

print("Enter the first string:")
guard let source = readLine(), !source.isEmpty else {
    fatalError("First string is empty.")
}

print("Enter the second string:")
```

```
guard let target = readLine(), !target.isEmpty else {
    fatalError("Second string is empty.")
}

print("Enter the operation costs (replace insert delete) separated by spaces:")
guard let costsLine = readLine(),
      !costsLine.isEmpty else {
    fatalError("Operation costs input is empty.")
}

let costValues = costsLine.split(separator: " ").compactMap { Int($0) }
guard costValues.count == 3 else {
    fatalError("You must enter exactly three integer costs.")
}

let costs = Costs(
    replace: costValues[0],
    insert: costValues[1],
    delete: costValues[2]
)

let (operations, alignedSource, alignedTarget) = editDistanceWithTrace(costs:
costs, source: source, target: target)
print("Edit operations sequence: \(operations)")

let distance = levenshteinDistance(costs: costs, source: source, target: target)
print("Levenshtein distance: \(distance)")

let (lcsLength, lcsSubstring) = longestCommonSubstring(source, target)
print("Longest common substring length: \(lcsLength)")
print("Longest common substring: \"\(lcsSubstring)\"")
```