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. 3344 _____ Атоян М.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

Цель работы.

Написании программы вычисления редакционного расстояния и предписания алгоритмом Вагнера-Фишера.

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. (SS, $1 \le |S| \le 25501 \le |S| \le 2550$).

Вторая строка входных данных содержит строку из строчных латинских букв. (TT, $1 \le |T| \le 25501 \le |T| \le 2550$).

Параметры выходных данных:

Одно число LL, равное расстоянию Левенштейна между строками SS и TT.

Sample Input:

pedestal

stien

Sample Output:

7

Описание алгоритма.

Алгоритм Вагнера-Фишера — это алгоритм динамического программирования, предназначенный для вычисления редакционного расстояния (расстояния Левенштейна) между двумя строками. Это минимальное количество операций вставки, удаления или замены символов, необходимых для преобразования одной строки в другую.

Основные шаги алгоритма:

1) Инициализация матрицы:
Создаётся матрица размером (n+1) × (m+1), где n и m — длины строк.
Первая строка заполняется числами от 0 до n (стоимость удаления символов).
Первый столбец заполняется числами от 0 до m (стоимость вставки символов).
2) Заполнение матрицы:
Для каждой ячейки [i][j] (где i — символ первой строки, j — второй) вычисляется минимальная стоимость операций:
Удаление: cost = matrix[i-1][j] + 1
Вставка: cost = matrix[i][j-1] + 1
Замена:
Если символы совпадают (str1[i-1] == str2[j-1]), то замена бесплатна: cost = matrix[i-1][j-1].
Иначе: cost = matrix[i-1][j-1] + 1.
Выбирается минимальное из трёх значений.
3) Результат:
Значение в правом нижнем углу матрицы (matrix[n][m]) — искомое редакционное расстояние.
4) Сложность:
По времени: **O(n*m)**, т.к. алгоритм заполняет матрицу размером (n+1) * (m+1), где: n — длина первой строки, m — длина второй строки.Пространственная: **O(n·m),** т.к для хранения матрицы потребуется память пропорциональная n*m.

Алгоритм поиска редакционного предписания

1) Построение матрицы расстояний

Создаётся матрица D размером (n+1) × (m+1), где n и m — длины

строк str1 и str2.

D[i][j] — расстояние между подстроками str1[0..i-1] и str2[0..j-1].

Заполнение матрицы аналогично алгоритму Вагнера-Фишера.

2) Восстановление операций

Начиная с ячейки D[n][m], двигаемся к D[0][0], выбирая путь с минимальной стоимостью.

Для каждой ячейки D[i][j] определяем, какая операция была применена:

Шаг вверх (D[i-1][j] → D[i][j]): удаление символа str1[i-1].

Шаг влево (D[i][j-1] → D[i][j]): вставка символа str2[j-1].

Шаг по диагонали (D[i-1][j-1] → D[i][j]):

Если str1[i-1] == str2[j-1]: совпадение (операция не требуется).

Иначе: замена str1[i-1] на str2[j-1].

Описание функций и структур данных.

1. *func levenshteinDistance(s, t []rune) (int, []Operation)*

Функция инициализирует матрицу *dp [][]int,* что занимает len(s)*len(t) памяти, заполняет её значениями по алгоритму Вагнера-Фишера, что занимает len(s)*len(t) времени, и возвращает левое нижнее значение матрице(редакционное расстояние между двумя строками), а также список операций, необходимых для преобразования строки s в строку t.

2. *func min(a, b, c int) (int, Operation)*

Функция возвращает наименьшее переданное в аргументы редакционное расстояние и операцию, которая соответствует выбранному расстоянию.

Исследование.

Теоретическая сложность алгоритма по времени составляет O(len(str1) * len(str2)). Это обусловлено тем, что в ходе работы алгоритм итерируется по матрице размером len(str1) на len(str2).

Теоретическая сложность алгоритма по памяти составляет O(lens(str1) * len(str2)). Это обусловлено тем, что в ходе работы алгоритм создаёт матрицу размером len(str1) на len(str2).

Для проверки теоретических значений были собраны бэнчмарки для разных длин строк(два числа через дефис после названия функции):

```
mikhail@DESKTOP-KN4CJD8:~/PiAA/lab3/benchmarking$ go test -bench=. -benchmem -benchtime=3s
goos: linux
goarch: amd64
pkg: lab3/benchmarking
cpu: AMD Ryzen 5 6600H with Radeon Graphics
BenchmarkLevenshtein100_100-12             97521             36307 ns/op           93184 B/op        102 allocs/op
BenchmarkLevenshtein400_100-12             21772            165330 ns/op          369026 B/op        402 allocs/op
BenchmarkLevenshtein400_200-12             10000            324138 ns/op          728323 B/op        402 allocs/op
BenchmarkLevenshtein800_400-12              2529           1329760 ns/op         2788750 B/op        802 allocs/op
BenchmarkLevenshtein1600_200-12             2862           1217373 ns/op         2909967 B/op       1602 allocs/op
PASS
ok      lab3/benchmarking       19.591s
```

Рисунок 1 – Бенчмарки

Наглядно видно, что время работы и аллоцированная память одних бенчмарков относятся ко времени работы и алоцированной памяти других бэнчмарков, как произведения длин строк разных этит бэнчмарков, переданных в аргументы. Теоретическая сложность по времени и по памяти совпадает с практической.

Тестирование.

Таблица 1 – Результаты тестирования для первого задания

№	Входные данные	Выходные данные	Комментарий
1	ab abfagfab	6	Верно
2	hello world	4	Верно
3	ВАСИЛЬЕВ КВАСЮТИН	6	Верно
4	pedestal stien	7	Верно
5	connect conehead	4	Верно

Выводы.

Был реализован алгоритм Вагнера-Фишера для вычисления редакционного расстояния и предписания между двумя строками, определяя минимальное количество операций (вставки, удаления, замены) для преобразования одной строки в другую. Алгоритм эффективно решает задачи сравнения строк, исправления опечаток и других приложений, связанных с обработкой текста.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: levenshtein.go

```
   package levenshtein

type Operation int

const (
      Replace Operation = iota
      Insert
      Delete
      Match
)

func (op Operation) String() string {
      return [...]string{"R", "I", "D", "M"}[op]
}

func min(a, b, c int) (int, Operation) {
      minVal := a
      op := Replace
      if b < minVal {
            minVal = b
            op = Insert
      }
      if c < minVal {
            minVal = c
            op = Delete
      }
      return minVal, op
}

func LevenshteinDistance(s, t []rune) int {
      m, n := len(s), len(t)
      dp := make([][]int, m+1)
      ops := make([][]Operation, m+1)

      for i := range dp {
            dp[i] = make([]int, n+1)
            ops[i] = make([]Operation, n+1)
      }

      for i := 0; i <= m; i++ {
            dp[i][0] = i
            ops[i][0] = Delete
            printTable(dp, s, t, i, 0)
      }

      for j := 0; j <= n; j++ {
            dp[0][j] = j
            ops[0][j] = Insert
            printTable(dp, s, t, 0, j)
      }

      for i := 1; i <= m; i++ {
            for j := 1; j <= n; j++ {
```

```
                if s[i-1] == t[j-1] {
                        dp[i][j] = dp[i-1][j-1]
                        ops[i][j] = Match
                } else {
                        val, op := min(
                                dp[i-1][j-1]+1, // Replacement
                                dp[i][j-1]+1,   // Insertion
                                dp[i-1][j]+1,   // Deletion
                        )
                        dp[i][j] = val
                        ops[i][j] = op
                        printTable(dp, s, t, i, j)
                }
        }
    }

    i, j := m, n
    var operations []Operation
    for i > 0 || j > 0 {
        op := ops[i][j]
        operations = append(operations, op)
        switch op {
        case Match, Replace:
            i--
            j--
        case Insert:
            j--
        case Delete:
            i--
        }
    }

    for i, j := 0, len(operations)-1; i < j; i, j = i+1, j-1 {
        operations[i], operations[j] = operations[j], operations[i]
    }

    printTable(dp, s, t, len(dp), len(dp[0]))
    printAlignment(s, t, operations)

    return dp[m][n]
}
```

Имя файла visualization.go:

```
package levenshtein

import (
    "fmt"
    "os"
    "os/exec"
    "time"
)

func clearScreen() {
    cmd := exec.Command("clear")
    if os.Getenv("OS") == "Windows_NT" {
        cmd = exec.Command("cmd", "/c", "cls")
    }
    cmd.Stdout = os.Stdout
```

```
	cmd.Run()
}

func printAlignment(s, t []rune, operations []Operation) {
	fmt.Println("\nAlignment Steps:")
	fmt.Print("  ")
	for _, op := range operations {
		fmt.Printf(" %-3s", op)
	}
	fmt.Println("\n  " + repeat("———", len(operations)))

	fmt.Print("S ")
	for i, op := range operations {
		switch op {
		case Match, Replace:
			fmt.Printf(" %-2c", s[0])
			s = s[1:]
		case Delete:
			fmt.Printf(" %-2c", s[0])
			s = s[1:]
		default:
			fmt.Print(" * ")
		}
		if i < len(operations)-1 {
			fmt.Print("|")
		}
	}
	fmt.Println("\n  " + repeat("———", len(operations)))

	fmt.Print("T ")
	for i, op := range operations {
		switch op {
		case Match, Replace:
			fmt.Printf(" %-2c", t[0])
			t = t[1:]
		case Insert:
			fmt.Printf(" %-2c", t[0])
			t = t[1:]
		default:
			fmt.Print(" * ")
		}
		if i < len(operations)-1 {
			fmt.Print("|")
		}
	}
	fmt.Println("\n")
}

func repeat(s string, n int) string {
	result := ""
	for i := 0; i < n; i++ {
		result += s
	}
	return result
}

func printTable(dp [][]int, s, t []rune, i, j int) {
	clearScreen()
```

```
	fmt.Print("       ")
	for _, ch := range t {
		fmt.Printf(" %c ", ch)
	}
	fmt.Println()

	for row := 0; row < len(dp); row++ {
		if row == 0 {
			fmt.Print("   ")
		} else {
			fmt.Printf(" %c ", s[row-1])
		}

		for col := 0; col < len(dp[row]); col++ {
			if row == i && col == j {
				fmt.Printf(" \033[1;32m%2d\033[0m", dp[row][col])
			} else {
				fmt.Printf(" %2d", dp[row][col])
			}
		}
		fmt.Println()
	}
	fmt.Println()
	time.Sleep(300 * time.Millisecond) // Pause for visualization
}
```

Имя файла main.go:

```
package main

import (
	"fmt"
	"lab3/levenshtein"
)

func main() {
	var s1, s2 string
	fmt.Print("Enter first string: ")
	fmt.Scanln(&s1)
	fmt.Print("Enter second string: ")
	fmt.Scanln(&s2)

	distance:= levenshtein.LevenshteinDistance([]rune(s1), []rune(s2))

	fmt.Printf("\nLevenshtein distance: %d\n", distance)
}
```