Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение
высшего образования
«Московский государственный технический университет
имени Н.Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»
КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

Отчёт

по лабораторной работе №1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент	ИУ7-55Б		Хетагуров П.К
	(Группа)	(Подпись, дата)	(И.О. Фамилия)
Преподователь			Л.Л. Волкова
		(Подпись, дата)	(И.О. Фамилия)

Москва, 2020

Содержание

Введение

В данной лабораторной работе будут рассмотренны и проанализированы такие реализации алгоритма поиска расстояния Левенштейна как:

1. матричная реализация;
2. рекурсивная без матрицы;
3. рекурсивная с матрицей;
4. матричная реализация алгоритма поиска расстояния Дамерау-Левенштейна.

1 Аналитическая часть

В данном разделе будут поставлены цели и задачи работы, будут рассмотренны основные теоритические сведения связанные с алгоритмами поиска расстояния Левенштейна и Дамерау-Левенштейна.

1.1 Цель и задачи работы

Цель работы:

Реализовать и сравнить по эффективности(емкостной, временной разные алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Задачи работы:

1) дать математическое описание расстояния Левенштейна и Дамерау-Левенштейна;
2) разработать алгоритмы поиска расстояний;
3) реализовать построенные алгоритмы;
4) провести эксперименты по замеру времени работы разработанных алгоритмов;
5) провести сравнения алгоритмов по затраченному времени и максимальной затраченной памяти;
6) дать теоритическую оценку затрачиваемой памяти.

1.2 Формула для нахождения расстояния Левенштейна

Расстояние Левенштейна (редакционное расстояние) - это минимальное количество редакционных операций, которое необходимо совершить для преобразования одной строки в другую.

Список редакционных операций:

- вставка (I);
- удаление (D);
- замена (R);
- совпадение (M).

При этом операции I, D, R имеют вес 1, а M - 0. Пусть есть две строки s_1 и s_2, индексируемые с 1. Тогда расстояние Левенштейна (L) определяется следующей реккурентной формулой (1):

$$D(s_1[1...i], s_2[1...j]) = \begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{i > 0, j = 0} \\ j, & \text{i = 0, j > 0} \\ min(D(s_1[1...i], s_2[i...j-1]) + 1, & \\ D(s_1[1...i-1], s_2[i...j-1]) + \begin{cases} 1, & \text{Если } s_1\text{[i] != } s_2\text{[j]}, \\ 0, & \text{Иначе} \end{cases} & \text{i > 0, j > 0} \\ D(s_1[1...i-1], s_2[i...j]) + 1) & \end{cases} \tag{1}$$

, где s_1[1...i] и s_2[1...j] - строки длинной i и j соответственно.

1.3 Формула для нахождения расстояния Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна является модификацией расстояния Левенштейна, к возможным редакторским операциям добавляется операция перестановки двух соседних символов (X) со штрафом 1.

Модифицированная формула 1 для нахождения расстояния Дамерау-Левенштейна:

$$D(s_1[1...i], s_2[1...j]) = \begin{cases} 0, & \text{i = 0, j = 0} \\ i, & \text{i > 0, j = 0} \\ j, & \text{i = 0, j > 0} \\ min(D(s_1[1...i], s_2[i...j-1]) + 1, & \\ D(s_1[1...i-1], s_2[i...j-1]) + \begin{cases} 1, & \text{Если } s_1\text{[i] != } s_2\text{[j]}, \\ 0, & \text{Иначе} \end{cases} & \text{i > 1, j > 1, } s_1\text{[i - 1] = } s_2\text{[j], } s_1\text{[i] = } s_2\text{[j - 1]} \\ D(s_1[1...i-1], s_2[i...j]) + 1), & \\ D(s_1[1...i-2], s_2[i...j-2]) + 1), & \\ & \\ min(D(s_1[1...i], s_2[i...j-1]) + 1, & \\ D(s_1[1...i-1], s_2[i...j-1]) + \begin{cases} 1, & \text{Если } s_1\text{[i] != } s_2\text{[j]}, \\ 0, & \text{Иначе} \end{cases} & \text{Иначе} \\ D(s_1[1...i-1], s_2[i...j]) + 1) & \end{cases} \tag{2}$$

, где s_1[1...i] и s_2[1...j] - строки длинной i и j соответственно.

Существуют и другие модификации, учитывающие расположение клавиш на клавиатуре и другие факторы, но в данной работе они не рассматриваются.

2 Конструкторская часть

В данном разделе будут рассмотренны схемы алгоритмов, требования к функциональности ПО и проведена теоритическая оценка затрачиваемой памяти.

2.1 Требования к ПО

ПО должно иметь два режима работы, выбираемыхиз меню:

1. режим демонстрации. В этом режиме должен осуществляться ввод двух слов и демонстрация работы на них всех реализованных алгоритмов, в том числе: вывод матрицы решения для алгоритмов, в которых это возможно, вывод найденного расстояния;
2. режим тестирования. В этом режиме должны проводится замеры эффективности (временной и емкостной) реализованных алгоритмов. Должен осуществляться вывод затраченного процессорного времени на случайным образом сгенерированных данных.

2.2 Схемы алгоритмов

Ниже представлены схемы следующих алгоритмов

- Матричный алгоритм поиска расстояния Левенштейна. Рисунок 1
- Рекурсивный алгоритм поиска расстояния Левенштейна без матрицы. Рисунок 2
- Рекурсивный алгоритм поиска расстояния Левенштейна с матрицей. Рисунок 3
- Матричный алгоритм поиска расстояния Дамерау-Левенштейна. Рисунок 4

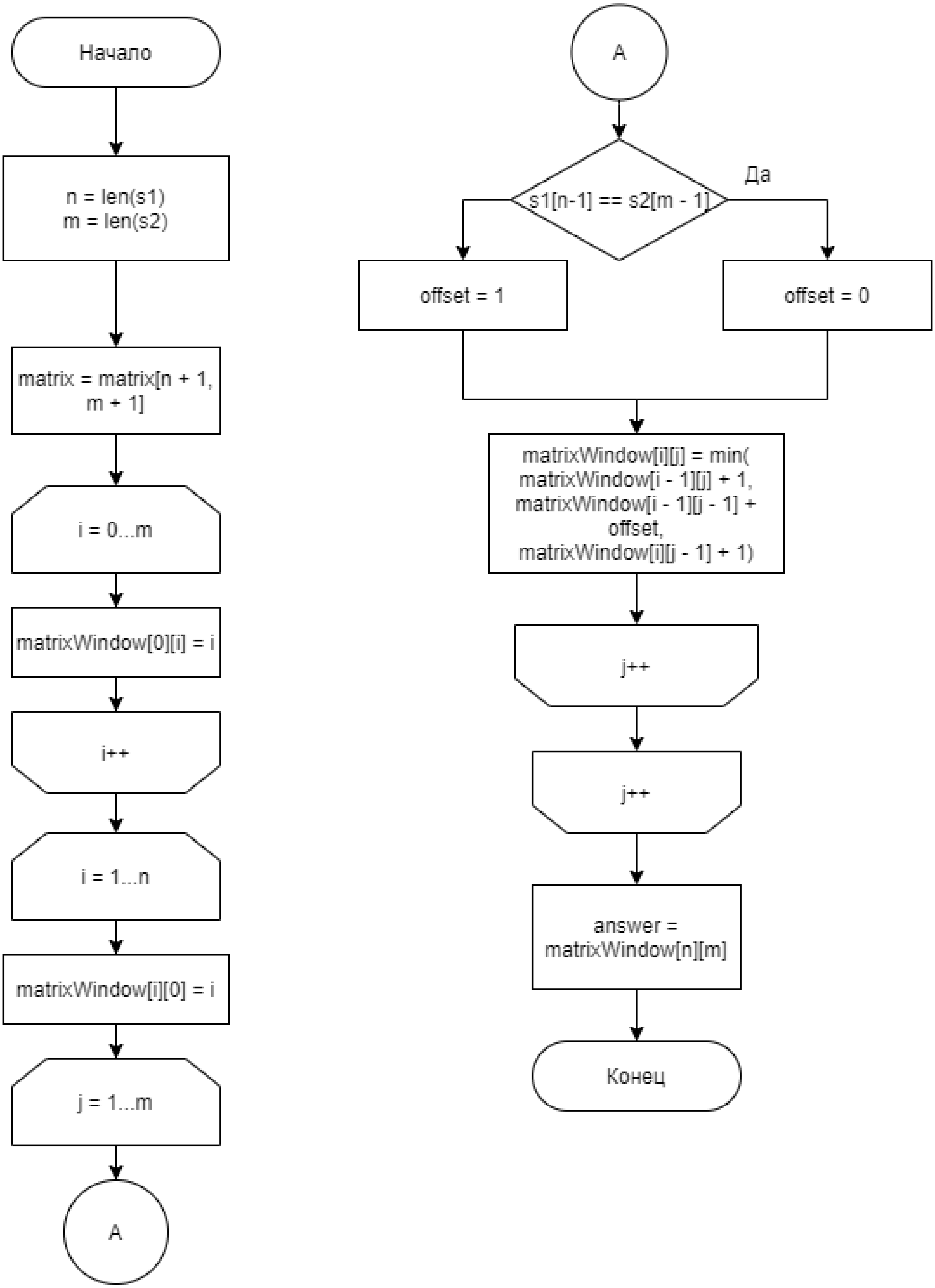


Рис. 1: Матричный алгоритм поиска расстояния Левенштейна

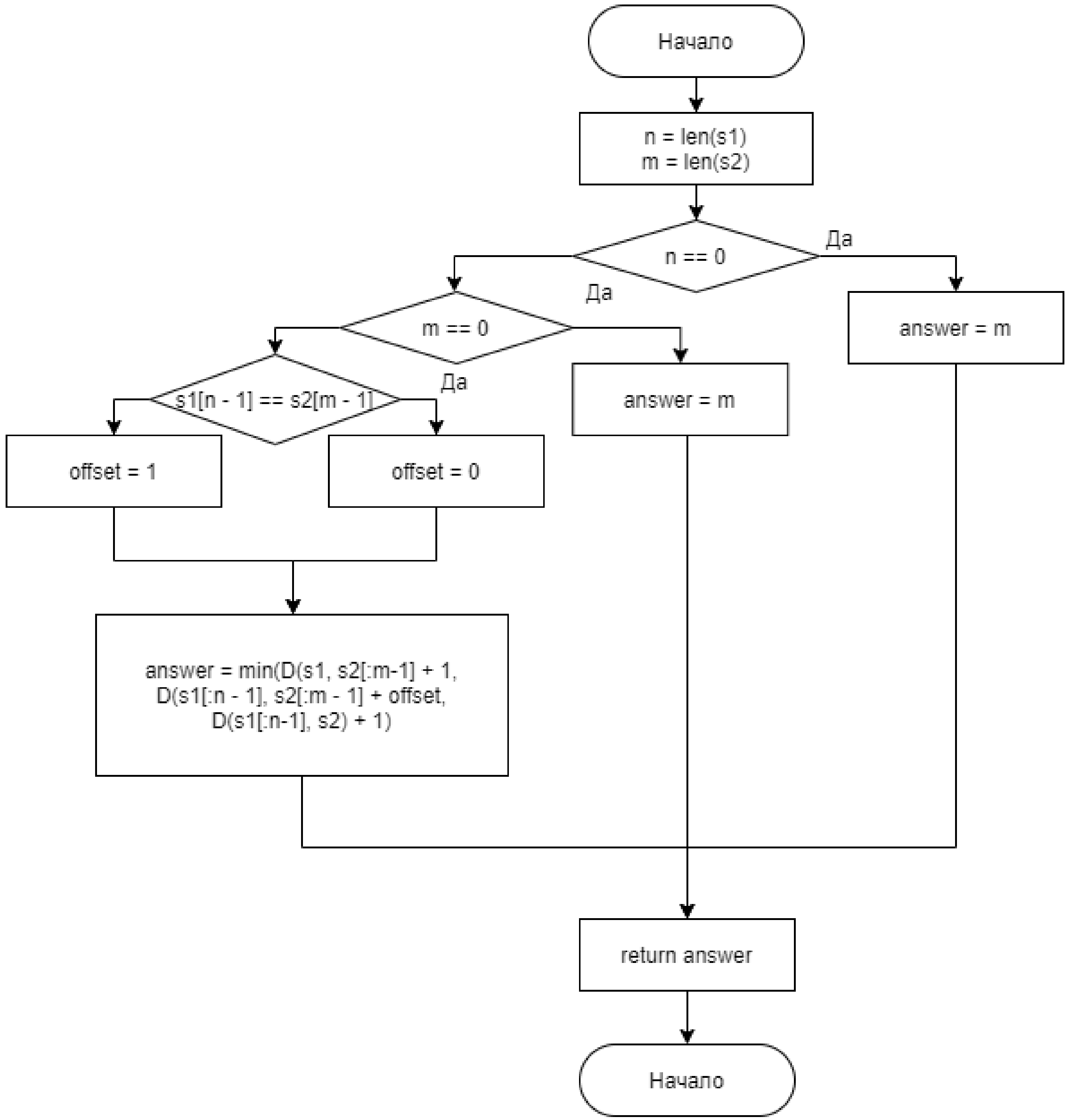


Рис. 2: Рекурсивный алгоритм поиска расстояния Левенштейна без матрицы

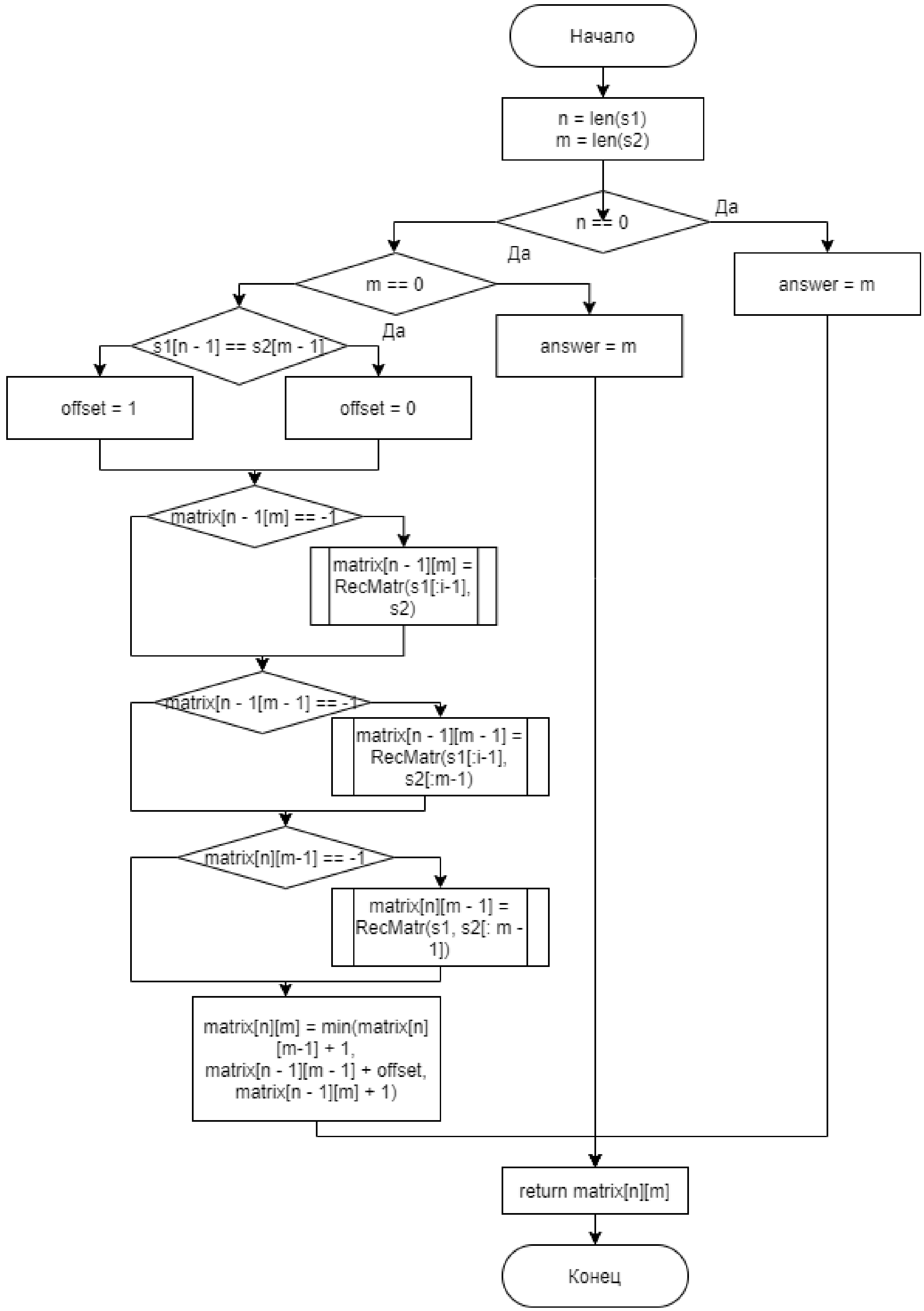


Рис. 3: Рекурсивный алгоритм поиска расстояния Левенштейна

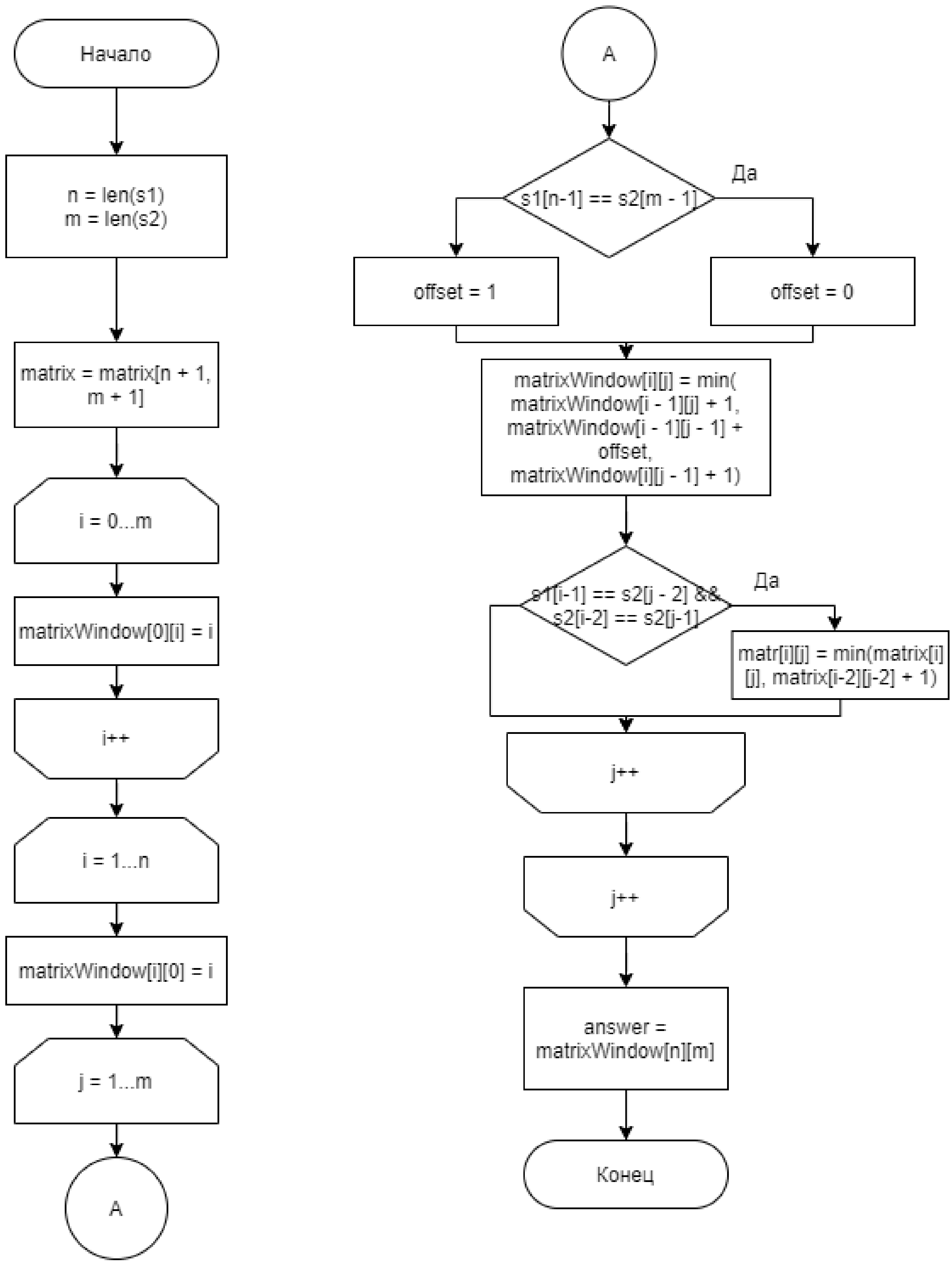


Рис. 4: Матричный алгоритм поиска расстояния Дамерау-Левенштейна

2.3 Оценка затрачиваемой памяти

При матричной реализации алгоритмов память занимает:

1. два входных слова длинами n и m;
2. матрица размером (n + 1)*(m + 1);
3. две переменные, хранящие длины слов;
4. четыре вспомагательные переменные. (i, j, answer, correction).

Таким образом размер занимаемой памяти составляет 3:

$$C_{\text{общ}} = C_1 * ((n+1)*(m+1)+2+4) + C_2 * (n+m) \tag{3}$$

При рекурсивной реализации при каждом вызове требуется выделять 4 переменных размером C_2. Максимальная глубина рекурсии равна (n + m). Максимальный объем занимаемой памяти - 4:

$$C_{\text{общ}} = (n+m)*(C_1*4 + C_2*(n+m)) + C_1*(n+1)*(m+1) \tag{4}$$

3 Технологическая часть

Ниже будут представлены средствы реализации и листинги реализованной программы.

3.1 Средства реализации

Выбранный язык программирования - Go, так как он структурный, в нем можно легко писать код, а требований по конкретнему языку не выдвигалось.[1] Среда разработки - Visual Studio Code.[2]

Функция вычисления процессорного времени использует функцию QueryPerfomanceCounter из библиотеки WinAPI.[3] Функция представлена на листинге 1.

Листинг 1. Функция замера процессорного времени.

```
func getProcessorTime() (int64) {
	dll, err := syscall.LoadDLL("kernel32.dll")
	if err != nil {
		fmt.Printf("Error 1")
		return 0;
	}
	qpc, err := dll.FindProc("QueryPerformanceCounter")
	if err != nil {
		fmt.Printf("Error 2")
		return 0;
	}
	var ctr int64
	ret, _, _ := qpc.Call(uintptr(unsafe.Pointer(&ctr)))
	if ret == 0 {
		return 0
	}
	return ctr
}
```

3.2 Реализации алгоритмов

Ниже представлены листинги реализаций следующего алгоритма:

- листинг 2. Матричный алгоритм нахождения расстояния Левенштейна,
- листинг 3. Рекурсивный алгоритм нахождения расстояния Левенштейна без матрицы,
- листинг 4. Рекурсивный алгоритм нахождения расстояния Левенштейна с матрицей,
- листинг 5. Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

Листинг 2. Реализация алгоритма поиска расстояния Левенштейна.

```
func LevenshtainMatrixNotWindow(s1, s2 []rune) (answer int) {
	firstLenght := len(s1)
	secondLenght := len(s2)
	if firstLenght == 0 && secondLenght > 0 {
		answer = secondLenght
	} else if secondLenght == 0 && firstLenght > 0 {
		answer = firstLenght
	} else {
		matrix := make([][]int, len(s1) + 1)
		for i := 0; i <= len(s1); i++ {
			matrix[i] = make([]int, len(s2) + 1)
		}
		for i := 0; i < secondLenght+1; i++ {
			matrix[0][i] = i
		}
		for i := 1; i < firstLenght+1; i++ {
			matrix[i][0] = i
			for j := 1; j < secondLenght+1; j++ {

				if matrix[i - 1][j] < matrix[i][j-1] {
					matrix[i][j] = matrix[i - 1][j]
				} else {
					matrix[i][j] = matrix[i][j-1]
				}
				matrix[i][j]++
				diagonalStep := matrix[i-1][j-1]
				if s1[i-1] != s2[j-1] {
					diagonalStep++
				}
				if diagonalStep < matrix[i][j] {
					matrix[i][j] = diagonalStep
				}
			}
		}
		answer = matrix[firstLenght][secondLenght]
	}
	return answer
```

```
}
```

Листинг 3. Реализация рекурсивного алгоритма поиска расстояния Левенштейна без матрицы.

```
func LevenshtainRecursiveMatrixless(s1, s2 []rune) (answer int) {
firstLenght := len(s1)
secondLenght := len(s2)

if firstLenght == 0 {
        answer = secondLenght
} else if secondLenght == 0 {
        answer = firstLenght
} else {
        lastSymbolFirst := s1[firstLenght - 1]
        lastSymbolSecond := s2[secondLenght - 1]

        correction := 1
        if (lastSymbolFirst == lastSymbolSecond) {
                correction = 0
        }
        s1 = s1[:firstLenght - 1]
        answer = LevenshtainRecursiveMatrixless(s1, s2) + 1
        s2 = s2[:secondLenght - 1]
        answerMiddle := LevenshtainRecursiveMatrixless(s1, s2) + correction
        s1 = append(s1, lastSymbolFirst)
        answerSecond := LevenshtainRecursiveMatrixless(s1, s2) + 1
        s1 = append(s2, lastSymbolSecond)

        if answerMiddle < answer {
                answer = answerMiddle
        }
        if answerSecond < answer {
                answer = answerSecond
        }
}
return answer
}
```

Листинг 4. Реализация рекурсивного алгоритма поиска расстояния Левенштейна с матрицей.

```
func LevenshtainRecursiveMatrixBody(s1, s2 []rune, matrix [][]int) (answer int) {
```

```
firstLenght := len(s1)
secondLenght := len(s2)

if firstLenght == 0 {
        answer = secondLenght
} else if secondLenght == 0 {
        answer = firstLenght
} else {
        lastSymbolFirst := s1[firstLenght - 1]
        lastSymbolSecond := s2[secondLenght - 1]

        correction := 1
        if (lastSymbolFirst == lastSymbolSecond) {
                correction = 0
        }
        s1 = s1[:firstLenght - 1]
        if (matrix[firstLenght - 1][secondLenght] == -1) {
                LevenshtainRecursiveMatrixBody(s1, s2, matrix)
        }
        answer = matrix[firstLenght - 1][secondLenght] + 1
        s2 = s2[:secondLenght - 1]
        if (matrix[firstLenght - 1][secondLenght - 1] == -1) {
                LevenshtainRecursiveMatrixBody(s1, s2, matrix)
        }
        answerMiddle := matrix[firstLenght - 1][secondLenght - 1] + correction
        s1 = append(s1, lastSymbolFirst)
        if (matrix[firstLenght][secondLenght - 1] == -1) {
                LevenshtainRecursiveMatrixBody(s1, s2, matrix)
        }
        answerSecond := matrix[firstLenght][secondLenght - 1] + 1
        s1 = append(s2, lastSymbolSecond)

        if answerMiddle < answer {
                answer = answerMiddle
        }
        if answerSecond < answer {
                answer = answerSecond
        }
```

```
        }
        matrix[firstLenght][secondLenght] = answer
        return answer
}
```

Листинг 5. Реализация матричного алгоритма поиска расстояния Дамерау-Левенштейна.

```
func DamerauLevenshtainMatrix(s1, s2 []rune) (answer int) {
firstLenght := len(s1)
secondLenght := len(s2)
if firstLenght == 0 && secondLenght > 0 {
        answer = secondLenght
} else if secondLenght == 0 && firstLenght > 0 {
        answer = firstLenght
} else {
        matrix := make([][]int, len(s1) + 1)
        for i := 0; i <= len(s1); i++ {
                matrix[i] = make([]int, len(s2) + 1)
        }

        for i := 0; i < secondLenght+1; i++ {
                matrix[0][i] = i
        }
        for i := 1; i < firstLenght+1; i++ {
                matrix[i][0] = i
                for j := 1; j < secondLenght+1; j++ {
                        if matrix[i-1][j] < matrix[i-1][j-1] {
                                matrix[i][j] = matrix[i-1][j]
                        } else {
                                matrix[i][j] = matrix[i][j-1]
                        }
                        matrix[i][j]++
                        diagonalStep := matrix[i-1][j-1]
                        if s1[i-1] != s2[j-1] {
                                diagonalStep++
                        }
                        if diagonalStep < matrix[i][j] {
                                matrix[i][j] = diagonalStep
                        }
```

```
                    if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] ==
                        if (matrix[i - 2][j - 2] + 1 < matrix[i][j]) {
                            matrix[i][j] = matrix[i - 2][j - 2] + 1
                        }
                    }
                }
            }
            answer = matrix[firstLenght][secondLenght]
        }
        return answer
}
```

3.3 Тестирование

Тестирование осуществляется по принципу “черного ящика”. Рассмотренные случаи:

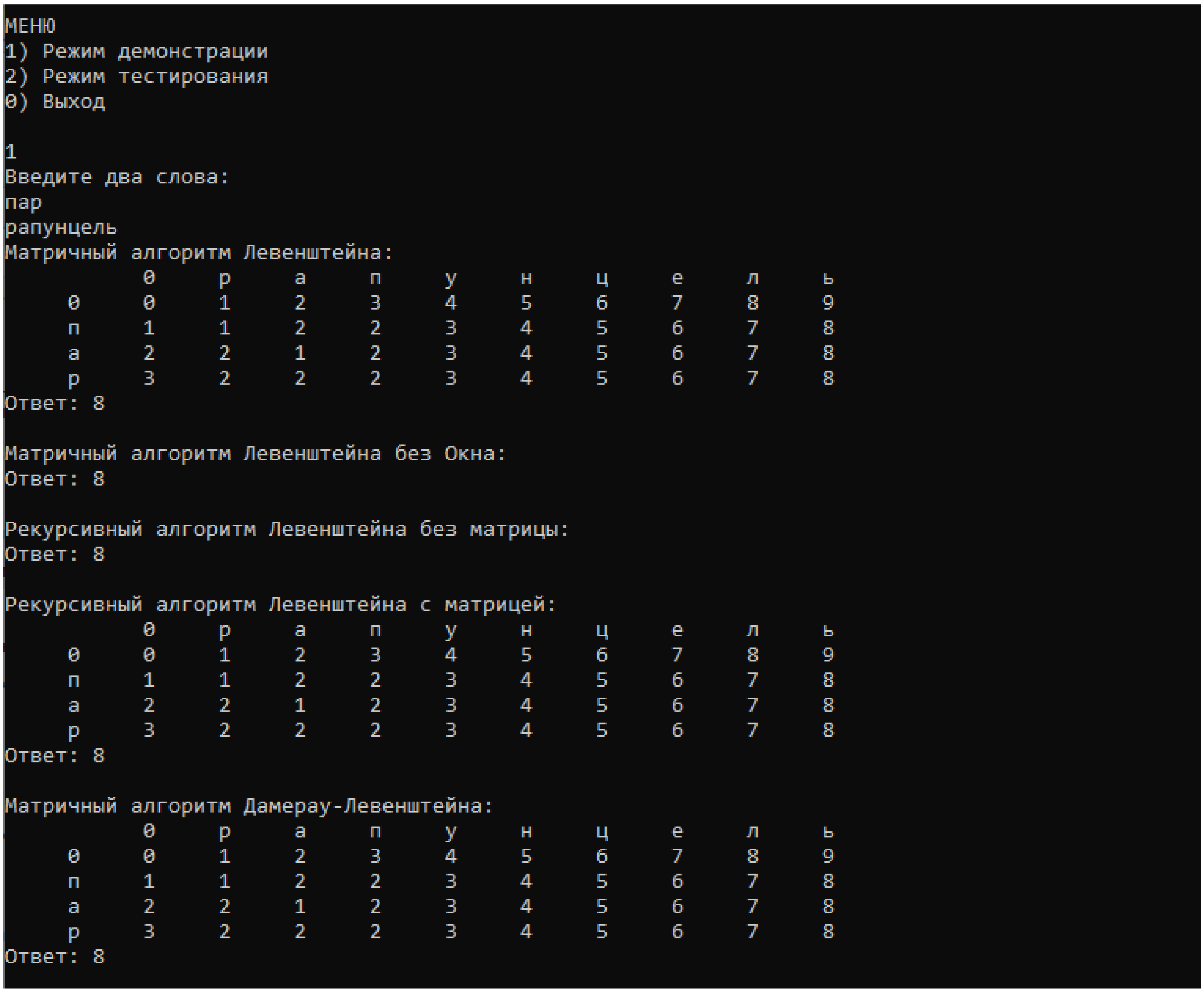
- одна строка пустая, вторая нет;
- обе строки пустые;
- строки эквивалентны,
- строки состоят их одного символа;
- строки состоят из произвольного количества символов.

4 Экспериментальная часть

В данной главе будут представлен пример работы программы, результат экспериментов по замеру времени и произведен сравнительный анализ алгоритмов по затрачиваемому времени.

4.1 Пример работы программы

Пример работы программы представлен на рисунке 5



```
МЕНЮ
1) Режим демонстрации
2) Режим тестирования
0) Выход

1
Введите два слова:
пар
рапунцель
Матричный алгоритм Левенштейна:
          0     р     а     п     у     н     ц     е     л     ь
     0    0     1     2     3     4     5     6     7     8     9
     п    1     1     2     2     3     4     5     6     7     8
     а    2     2     1     2     3     4     5     6     7     8
     р    3     2     2     2     3     4     5     6     7     8
Ответ: 8

Матричный алгоритм Левенштейна без Окна:
Ответ: 8

Рекурсивный алгоритм Левенштейна без матрицы:
Ответ: 8

Рекурсивный алгоритм Левенштейна с матрицей:
          0     р     а     п     у     н     ц     е     л     ь
     0    0     1     2     3     4     5     6     7     8     9
     п    1     1     2     2     3     4     5     6     7     8
     а    2     2     1     2     3     4     5     6     7     8
     р    3     2     2     2     3     4     5     6     7     8
Ответ: 8

Матричный алгоритм Дамерау-Левенштейна:
          0     р     а     п     у     н     ц     е     л     ь
     0    0     1     2     3     4     5     6     7     8     9
     п    1     1     2     2     3     4     5     6     7     8
     а    2     2     1     2     3     4     5     6     7     8
     р    3     2     2     2     3     4     5     6     7     8
Ответ: 8
```

Рис. 5: Пример работы программы со словами "пар" и "рапунцель"

4.2 Результаты тестирования

Результаты тестирования приведены в таблице 1.

Таблица 1: Результаты тестов

Входные строки	Ожидаемое результат	Полученный результат
“ “	0	0
‘aba‘ “	3	3
“ “	0	0
‘uwu‘ ‘uwu‘	0	0
‘random‘ ‘rndm‘	2	2
‘owl‘ ‘wolf‘	3/2	3/2

4.3 Сравнительный анализ алгоритмов по времени

Эксперименты проводятся на строках длины от 1 до 10 с шагом 2 (результаты на рисунке 6)

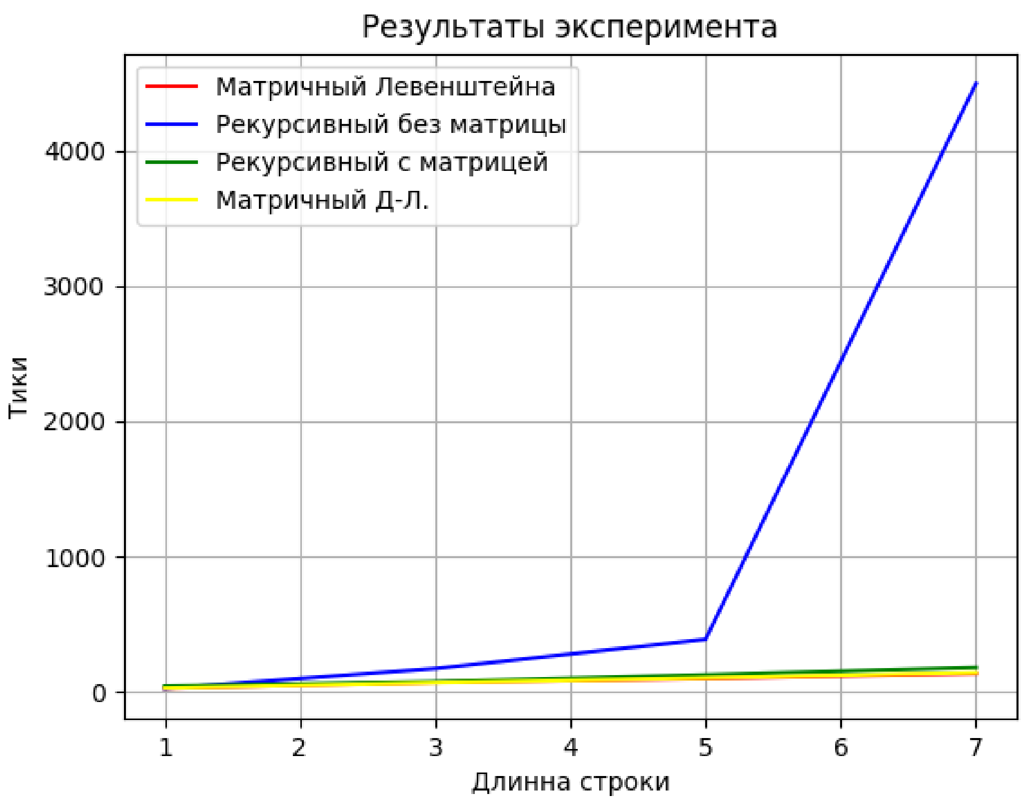


Рис. 6: Замеры времени на строках малой длины

И на строках длины от 50 до 550 с шагом 100 (результаты на рисунке 7)

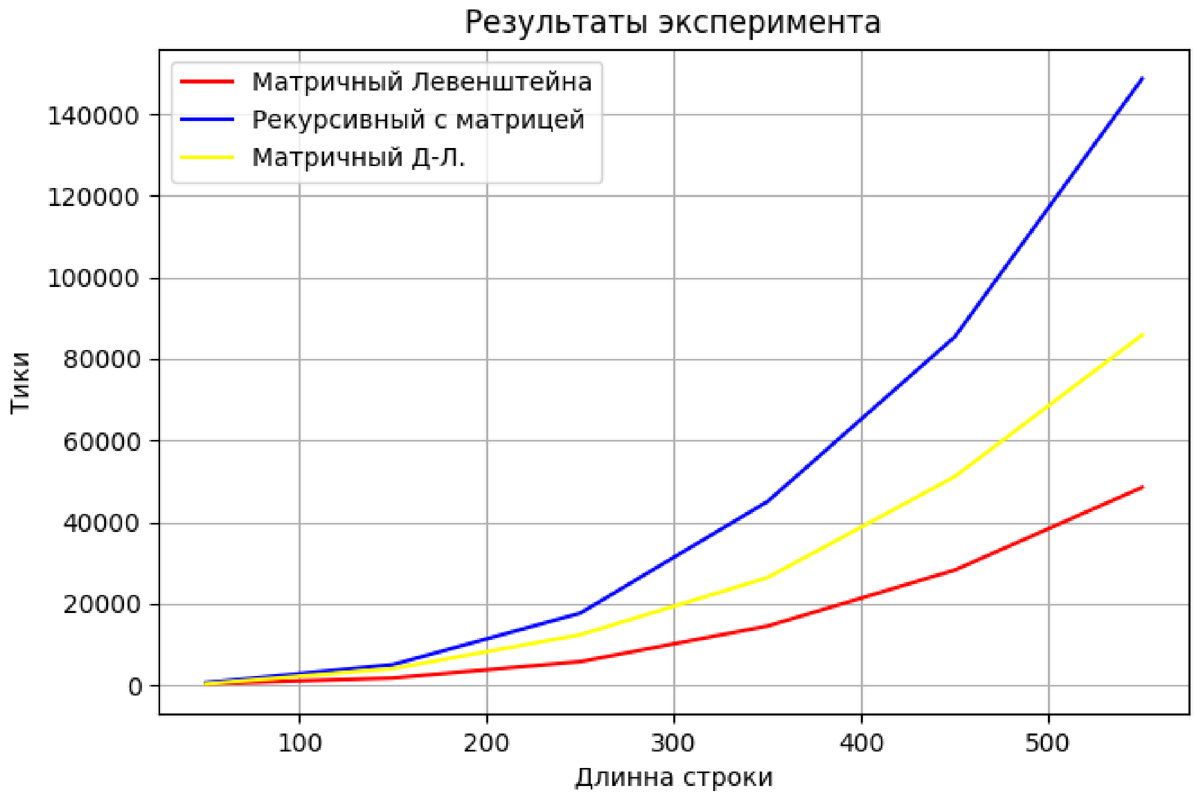


Рис. 7: Замеры времени на строках большой длины

Как видно из графиков, самым долгим алгоритмом является рекурсивный алгоритм без использования матрицы. Самым быстрым является матричный алгоритм. Рекурсивный матричный алгоритм быстрее простого рекурсивного, но сильно уступает матричным реализациям.

Заключение

В этой лабораторной работе были изложены теоретические основы расстояний Левенштейна и Дамерау-Левенштейна, были разработаны и реализованы алгоритмы мх поиска, проведены эксперименты по замеру времени работы разработанных алгоритмов и проведены сравнения алгоритмов по результатам эксперимента. Также была дана теоритическая оценка затрачиваемой памяти.

Список литературы

Список литературы

[1] Visual Studio Code [Электронный ресурс]. Режим доступа: (дата обращения - 02.10.2020) Свободный. URL: code.visualstudio.com

[2] Golang [Электронный ресурс]. Режим доступа: (дата обращения - 02.10.2020) Свободный. URL: http://golang-book.ru/

[3] WinAPI. Функция QueryPerformanceCounter [Электронный ресурс]. Режим доступа: (дата обращения - 02.10.2020) Свободный. URL: https://docs.microsoft.com/en-us/windows/win32/api/profileapi/nf-profileapi-queryperformancecounter