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»

Лабораторная работа № 2
по дисциплине «Анализ алгоритмов»

Тема Алгоритмы умножения матриц

Студент Могилин Н. С.

Группа ИУ7-51Б

Преподаватели Волкова Л. Л., Строганов Д. В., Строганов Ю. В.

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

В настоящей лабораторной работе рассматриваются алгоритмы умножения матриц.

Целью данной работы является изучение, реализация, оценка и сравнение эффективности стандартного алгоритма умножения матриц и алгоритма Винограда.

Задачи работы:

— Изучить алгоритмы умножения матриц: стандартный и Винограда;

— Реализовать стандартный алгоритм умножения матриц, а также алгоритмы Винограда: с оптимизацией по варианту и без нее;

— Ввести модель вычислений и выполнить в ней оценку трудоемкости реализованных алгоритмов;

— Получить и сравнить экспериментальную эффективность полученных реализаций с теоретической.

Оптимизация алгоритма Винограда по варианту:

— Двоичный сдвиг вместо умножения на 2 при вычислении значений результирующей матрицы;

— Введение инкремента при вычислении вспомогательных массивов вместо умножения на 2;

— Вынос начальной итерации из внутреннего цикла при вычислении значений результирующей матрицы.

1 Аналитическая часть

1.1 Операция умножения матриц

Пусть даны две матрицы: $A = ||a_{ij}||$ размером $N \times M$ и $B = ||b_{ij}||$ размером $M \times Q$. Тогда произведением AB называется матрица $C = ||c_{ij}||$ размером $N \times Q$, элементы c_{ij} которой вычисляются по правилу умножения i-ой строки матрицы A на j-ый столбец матрицы B:

$$C = AB = \begin{pmatrix} a_{11} & \dots & a_{1k} & \dots & a_{1M} \\ a_{21} & \dots & a_{2k} & \dots & a_{2M} \\ \dots & \dots & \dots & \dots & \dots \\ a_{N1} & \dots & a_{Nk} & \dots & a_{NM} \end{pmatrix} \begin{pmatrix} b_{11} & \dots & b_{1k} & \dots & b_{1Q} \\ b_{21} & \dots & b_{2k} & \dots & b_{2Q} \\ \dots & \dots & \dots & \dots & \dots \\ b_{M1} & \dots & b_{Mk} & \dots & b_{MQ} \end{pmatrix} = \\ = \begin{pmatrix} a_{11}b_{11} + \dots + a_{1M}b_{M1} & \dots & a_{11}b_{1Q} + \dots + a_{1M}b_{MQ} \\ a_{21}b_{11} + \dots + a_{2M}b_{M1} & \dots & a_{21}b_{1Q} + \dots + a_{2M}b_{MQ} \\ \dots & \dots & \dots \\ a_{N1}b_{11} + \dots + a_{NM}b_{M1} & \dots & a_{N1}b_{1Q} + \dots + a_{NM}b_{MQ} \end{pmatrix}. \tag{1.1}$$

Элемент c_{ij} с индексами i, j матрицы C может быть описан следующим образом:

$$c_{ij} = a_{i1}b_{1j} + a_{i2}b_{2j} + \dots + a_{iM}bMj = \sum_{k=1}^{M} a_{ik}b_{kj}, \quad i = \overline{1..N}, j = \overline{1..Q}. \tag{1.2}$$

Таким образом, операция матричного умножения матриц A, B определена, если число столбцов матрицы A совпадает с числом строк матрицы B [1].

1.2 Алгоритмы умножения матриц

1.2.1 Стандартный алгоритм умножения матриц

Стандартный алгоритм умножения матриц реализует формулу 1.2, то есть для каждого элемента матрицы C с индексами i, j ищется сумма поэлементных произведений чисел строки i матрицы A на числа столбца j матрицы B.

1.2.2 Алгоритм Винограда умножения матриц

Ключевая идея алгоритма Винограда – снизить долю затратных операций умножения. Для этого рассмотрим формулу 1.2 в терминах скалярного произведения строки матрицы A на столбец матрицы B. Пусть

$$U_i = A_i = (u_1u_2 \dots u_M) = (a_{i1}a_{i2} \dots a_{iM}), \\ V_j = B_j = (v_1v_2 \dots v_M) = (b_{1j}b_{2j} \dots b_{Mj}), \tag{1.3}$$

где U_i – i-ая строка матрицы A, V_j – j-ый столбец матрицы B.

Тогда формулу 1.2 можно переписать следующим образом:

$$c_{ij} = U_i \cdot V_j. \tag{1.4}$$

Пусть $M = 4$. Распишем скалярное произведение 1.4:

$$c_{ij} = U_i \ \cdot \ V_j = u_1v_1 + u_2v_2 + u_3v_3 + u_4v_4. \tag{1.5}$$

Равенство 1.5 можно переписать в следующем виде:

$$c_{ij} = (u_1 + v_2)(u_2 + v_1) + (u_3 + v_4)(u_4 + v_3) - u_1u_2 - u_3u_4 - v_1v_2 - v_3v_4. \tag{1.6}$$

На 4 слагаемых в правой части равенства 1.5 приходится 4 умножения. При этом для первых двух слагаемых правой части равенства 1.6 требуется 2 умножения. Несложно заметить, что слагаемые $-u_1u_2 - u_3u_4$ не зависят от элементов V_j, а слагаемые $-v_1v_2 - v_3v_4$ не зависят от элементов U_i, то есть могут быть рассчитаны предварительно для каждой матрицы. Таким образом, в самой вложенной части алгоритма (в части расчета c_{ij}) будет производиться 2 операции умножения и 6 операций сложения, в то время как в стандартном алгоритме – 4 операции умножения и 4 операции сложения.

1.2.3 Возможные оптимизации алгоритма Винограда

Стратегии оптимизации алгоритма сводятся к использованию более быстрых операций: инкремент вместо сложения, двоичный сдвиг вместо умножения на 2, использование дополнительного буфера для накопления значения при суммировании с целью минимизации обращений к ячейке матрицы. Более подробно оптимизация алгоритма будет рассмотрена далее.

1.2.4 Модель вычислений

Для оценки трудоемкости алгоритмов необходимо ввести систему стоимостей базовых операций, оператора цикла и условного оператора:

— Примем единичной трудоемкость следующих операций: $=$, $+$, $-$, $+=$, $-=$, $==$, $!=$, $<$, $>$, $<=$, $>=$, [], $<<$, $>>$, $++$, $--$, &&, ||, !, &, |;

— Трудоемкость операций $*$, $/$, $*=$, $/=$, $\%$, $\%=$ примем равной 2;

— Трудоемкость оператора цикла примем равной:

$$f_{\text{ц}} = f_{\text{иниц}} + f_{\text{сравн}} + n \cdot (f_{\text{инкр}} + f_{\text{сравн}} + f_{\text{тела}}); \tag{1.7}$$

— Трудоемкость условного оператора примем равной:

$$f_{if} = f_{\text{усл}} + \begin{cases} \min(f_1, f_2), & \text{лучший случай (л.с.)} \\ \max(f_1, f_2), & \text{худший случай (х.с.)} \end{cases}, \tag{1.8}$$

где $f_{\text{усл}}$ – трудоемкость вычисления истинности условия, f_1 – трудоемкость блока, выполняемого при истинном условии, f_2 – при ложном условии.

Вывод

В этом разделе были рассмотрены алгоритмы вычисления произведения матриц, а также была введена модель вычислений трудоемкости алгоритма для дальнейшей оценки реализуемых программ.

2 Конструкторская часть

В данном разделе приводятся схемы алгоритмов, реализуемых в рамках лабораторной работы.

2.1 Схемы реализуемых алгоритмов

На рисунках 2.1-2.5 приведены схемы реализуемых алгоритмов.

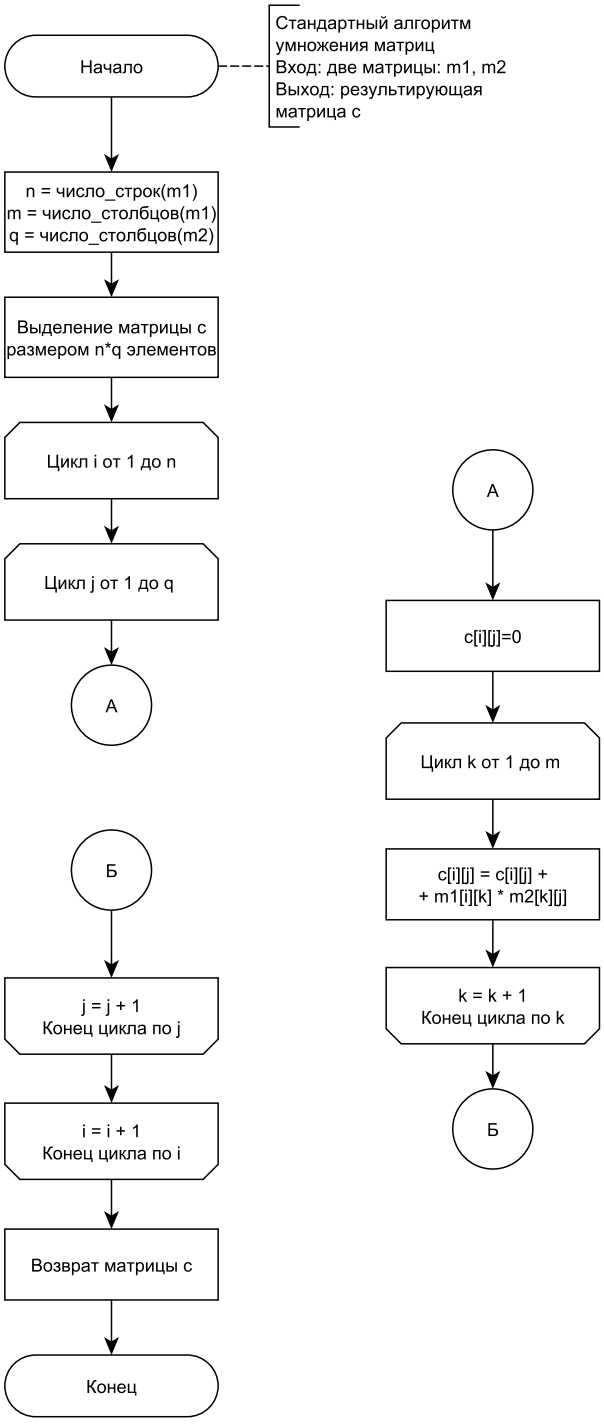

Рисунок 2.1 — Схема стандартного алгоритма умножения матриц

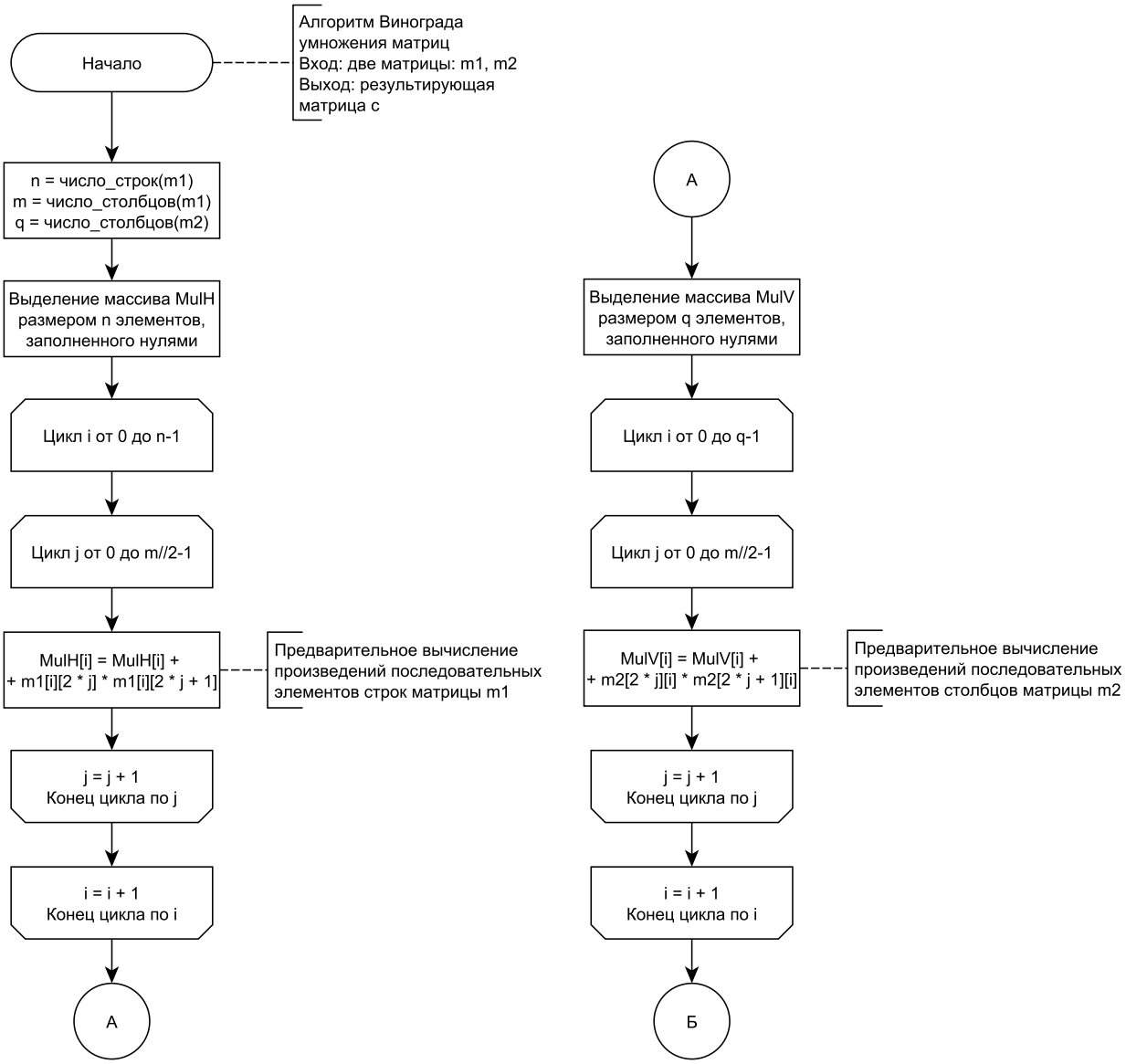

Рисунок 2.2 — Алгоритм Винограда, часть 1: предварительный расчет произведений

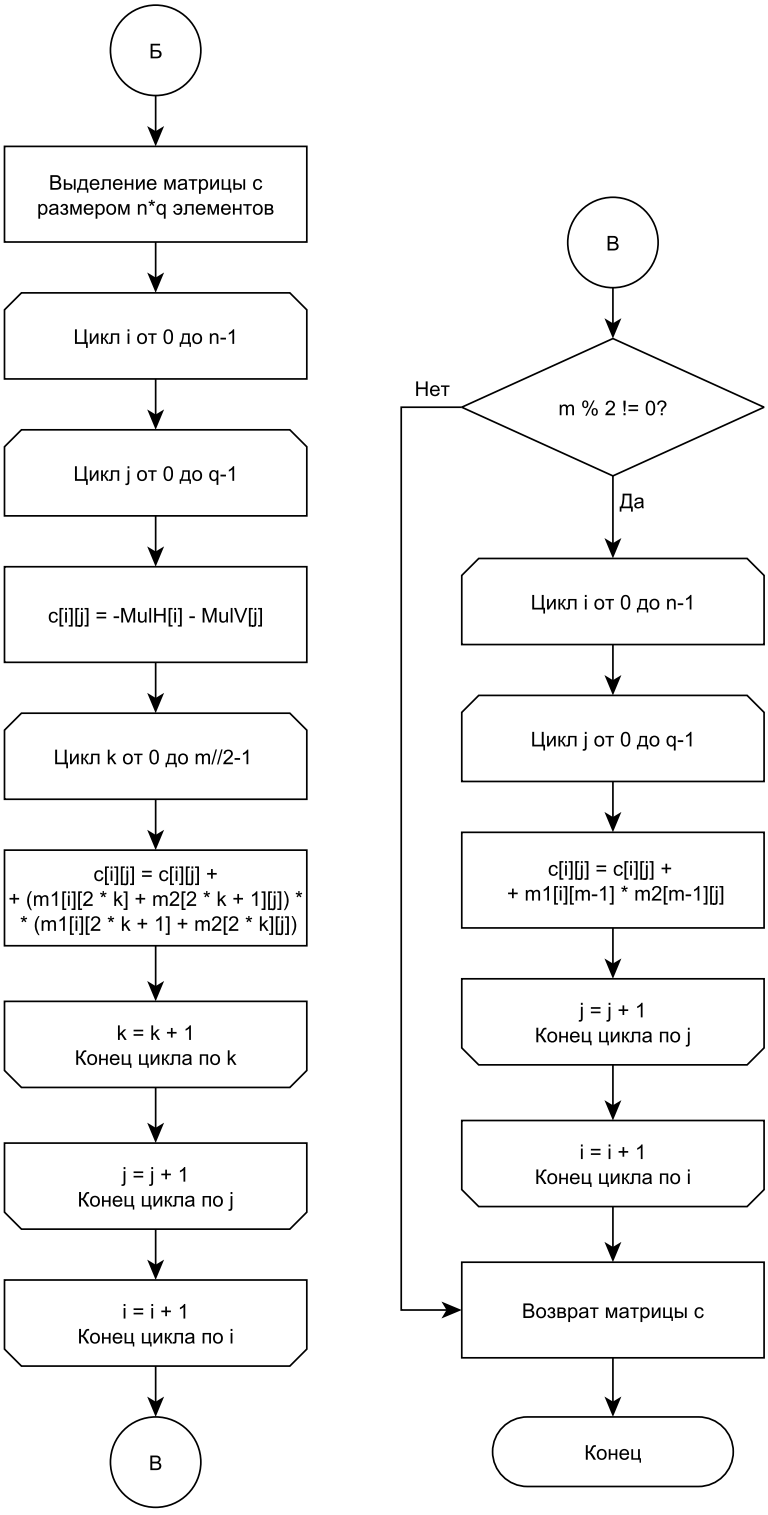

Рисунок 2.3 — Алгоритм Винограда, часть 2: расчет результирующей матрицы

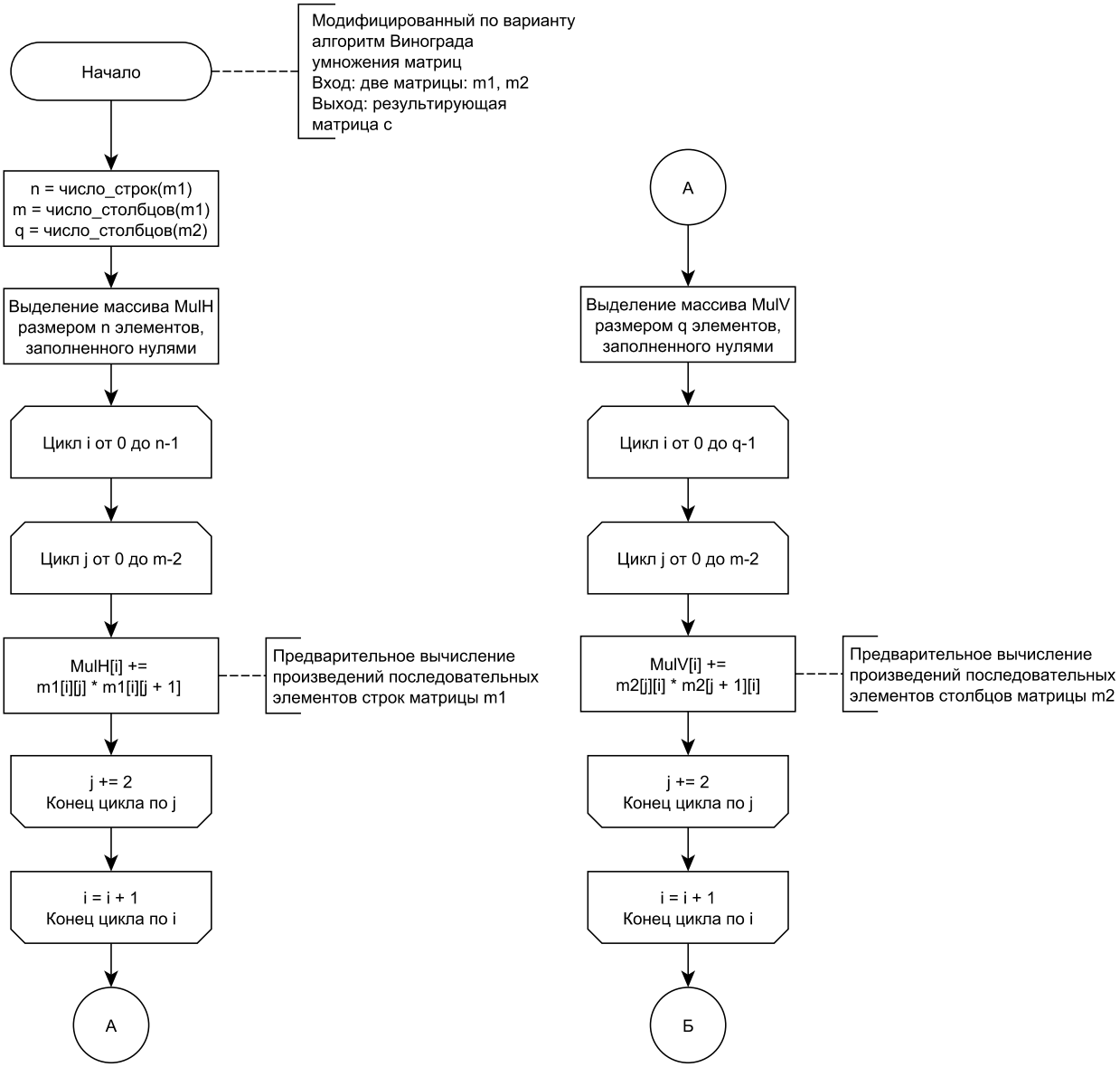

Рисунок 2.4 — Оптимизированный по варианту алгоритм Винограда, часть 1: предварительный расчет произведений

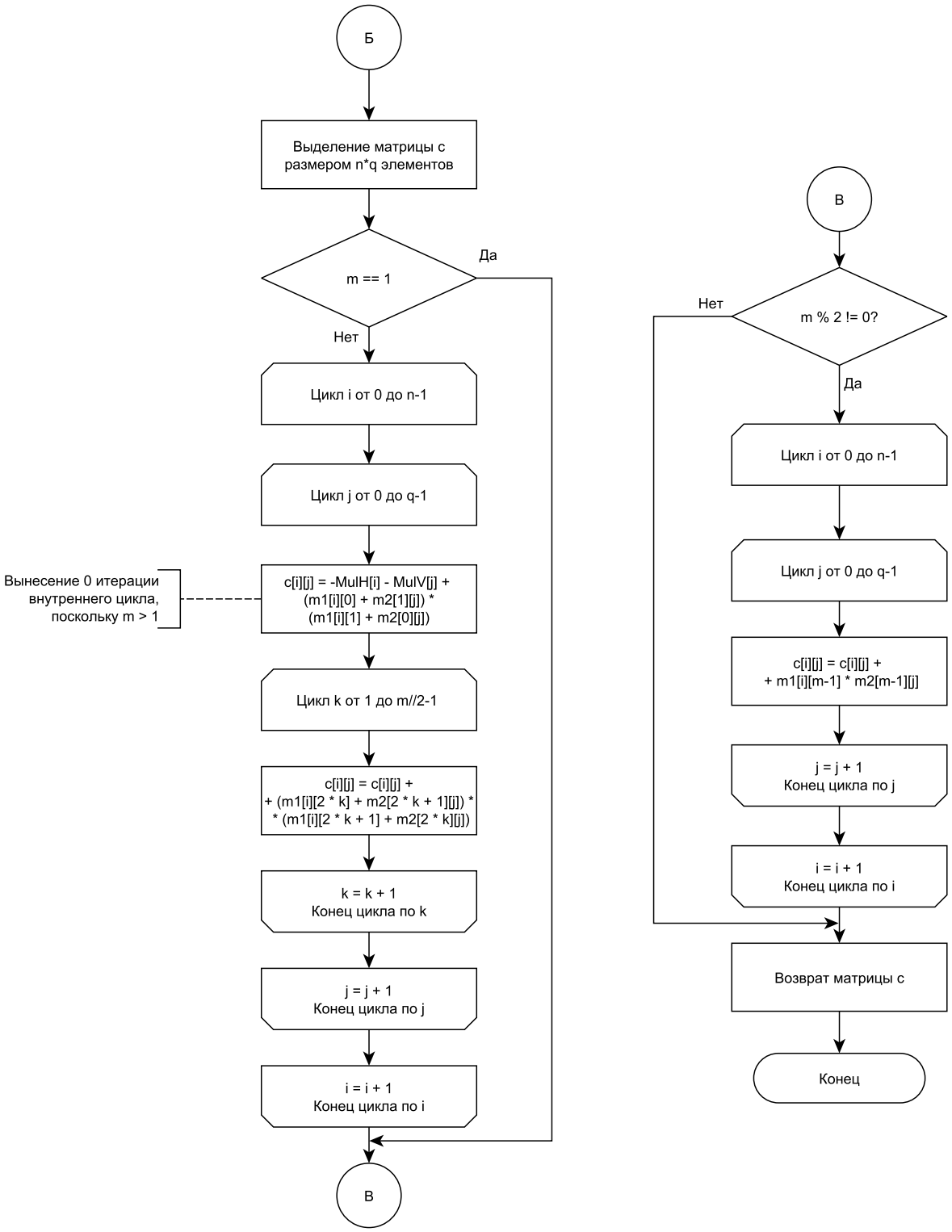

Рисунок 2.5 — Оптимизированный по варианту алгоритм Винограда, часть 2: расчет результирующей матрицы

К алгоритму Винограда применимы следующие оптимизации:

— Введение инкремента на значение, то есть использование $a+ = b$ вместо $a = a + b$. Выигрыш – 1 условная единица трудоемкости;

— Двоичный сдвиг влево на 1 вместо умножения на 2. Выигрыш – 1 усл. ед.;

— Альтернатива к предыдущему пункту: использование инкремента счетчика цикла на

2 в заголовке вместо того, чтобы на каждой итерации производить умножение на 2;

— Использование буфера для накопления значений $MulH[i]$, $MulV[i]$, $c[i][j]$. Выигрыш на каждой итерации – 1;

— Вынесение первой итерации из самого вложенного цикла алгоритма, разбиение на случаи $m == 1$ и $m > 1$.

В настоящей работе применены первая оптимизация, вторая оптимизация для расчета матрицы c, третья оптимизация при расчете вспомогательных массивов и последняя оптимизация.

Вывод

В текущем разделе были рассмотрены схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе рассматриваются средства реализации, а также листинги алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

3.1 Средства реализации

Для написания лабораторной работы использовался язык программирования *Python* [3], предоставляющий необходимый инструментарий для постановки замерного эксперимента и построения графиков: библиотеки *utime* [4] и *matplotlib* [5].

Выбор библиотеки для микроконтроллеров utime обосновывается тем, что замеры времени выполнения алгоритмов в данной лабораторной работе производились на отладочной плате NUCLEO-F767ZI [6], на которой можно запускать программы на языке *MicroPython* [7].

3.2 Листинги алгоритмов

В листингах 3.1-3.4 приведены реализации алгоритмов Левенштейна и Дамерау-Левенштейна.

```
1  def lev_recursive(str1: str, str2: str) -> int:
2    l1 = len(str1)
3    l2 = len(str2)
4    if not l1 and not l2:
5      return 0
6    if (l2 == 0) != (l1 == 0):
7      return l1 + l2
8
9    flag = 0 if str1[l1 - 1] == str2[l2 - 1] else 1
10
11   return min(lev_recursive(str1, str2[:l2 - 1]) + 1,
12              lev_recursive(str1[:l1 - 1], str2) + 1,
13              lev_recursive(str1[:l1 - 1], str2[:l2 - 1]) + flag)
```

Листинг 3.1 — Рекурсивный алгоритм нахождения расстояния Левенштейна

```
def lev_recursive_cached_rec(str1: str, str2: str, matrix) -> int:
  l1 = len(str1)
  l2 = len(str2)
  if not l1 and not l2:
    return 0
  if l2 and not l1:
    return l2
  if l1 and not l2:
    return l1

  str1_s = str1[:l1 - 1]
  str2_s = str2[:l2 - 1]
  if matrix[l1][l2 - 1] == -1:
    matrix[l1][l2 - 1] = lev_recursive_cached_rec(str1, str2_s,
       matrix)
  if matrix[l1 - 1][l2] == -1:
    matrix[l1 - 1][l2] = lev_recursive_cached_rec(str1_s, str2,
       matrix)
  if matrix[l1 - 1][l2 - 1] == -1:
    matrix[l1 - 1][l2 - 1] = lev_recursive_cached_rec(str1_s, str2_s,
        matrix)

  flag = 0 if str1[l1 - 1] == str2[l2 - 1] else 1

  return min(matrix[l1][l2 - 1] + 1,
             matrix[l1 - 1][l2] + 1,
             matrix[l1 - 1][l2 - 1] + flag)

def lev_recursive_cached(str1: str, str2: str) -> int:
  matrix = []
  for i in range(len(str1) + 1):
    matrix.append([-1] * (len(str2) + 1))
  return lev_recursive_cached_rec(str1, str2, matrix)
```

Листинг 3.2 — Рекурсивный алгоритм нахождения расстояния Левенштейна с меморизацией

```
def lev_dynamic(str1: str, str2: str) -> int:
  l1 = len(str1)
  l2 = len(str2)

  matrix = []
  for i in range(l1 + 1):
```

```
      matrix.append([0] * (l2 + 1))

    for i in range((l2 + 1)):
      matrix[0][i] = i
    for i in range(l1 + 1):
      matrix[i][0] = i

    for i in range(1, l1 + 1):
      for j in range(1, l2 + 1):
        matrix[i][j] = min(
        matrix[i - 1][j] + 1,
        matrix[i][j - 1] + 1,
        matrix[i - 1][j - 1] + (str1[i - 1] != str2[j - 1]))
    return matrix[l1][l2]
```

Листинг 3.3 — Матричный алгоритм нахождения расстояния Левенштейна

```
def dam_lev_dynamic(str1: str, str2: str) -> int:
    matrix = []
    for i in range(len(str1) + 1):
      matrix.append([0] * (len(str2) + 1))

    for i in range((len(str2) + 1)):
      matrix[0][i] = i
    for i in range(len(str1) + 1):
      matrix[i][0] = i

    for i in range(1, len(str1) + 1):
      for j in range(1, len(str2) + 1):
        matrix[i][j] = min(
        matrix[i - 1][j] + 1,
        matrix[i][j - 1] + 1,
        matrix[i - 1][j - 1] + (str1[i - 1] != str2[j - 1]))
        if i > 1 and j > 1 and str1[i - 1] == str2[j - 2] and str1[i -
           2] == str2[j - 1]:
          matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1)

    return matrix[len(str1)][len(str2)]
```

Листинг 3.4 — Матричный алгоритм нахождения расстояния Дамерау-Левенштейна

3.3 Функциональное тестирование

Для тестирования выделим следующие классы эквивалентности:

1) Обе строки пусты;

2) Одна из строк имеет нулевую длину;

3) Две непустные строки равны;

4) Две разные непустые строки, при расчете расстояния между которыми перестановки не играют роли;

5) Две разные непустые строки, при расчете расстояния между которыми используются перестановки.

В таблице 3.1 приведены функциональные тесты для программы, реализующей алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Все тесты пройдены программой успешно.

Таблица 3.1 — Функциональные тесты

№	Строка 1	Строка 2	Ожидаемый результат	
			Левенштейн	Дамерау-Левенштейн
1	Пустая строка	Пустая строка	0	0
2	Пустая строка	"Привет, Мир!"	12	12
3	"Hello, world!"	Пустая строка	13	13
4	"Равные"	"Равные"	0	0
5	"Равные"	"Разные"	1	1
6	"Hello!"	"Greetings!"	8	8
7	"123456"	"654321"	6	5
8	"12"	"21"	2	1

3.4 Демонстрация работы программы

На рисунке 3.1 показан результат работы программы на функциональном тесте №5 из таблицы 3.1.

```
Режимы работы программы:
1. Ввод двух строк и расчет расстояния между ними;
2. Генерация случайных строк и проведение замерного эксперимента.
Режим работы: 1

Введите первую строку: Равные
Введите вторую строку: Разные
Расстояние Левенштейна (рекурсия): 1
Расстояние Левенштейна (рекурсия с меморизацией): 1
Кэш-матрица алгоритма:
/  0  Р  а  з  н  ы  е
0  0  1  2  3  4  5  6
Р  1  0  1  2  3  4  5
а  2  1  0  1  2  3  4
в  3  2  1  1  2  3  4
н  4  3  2  2  1  2  3
ы  5  4  3  3  2  1  2
е  6  5  4  4  3  2  1

Расстояние Левенштейна (матричный): 1
Матрица алгоритма:
/  0  Р  а  з  н  ы  е
0  0  1  2  3  4  5  6
Р  1  0  1  2  3  4  5
а  2  1  0  1  2  3  4
в  3  2  1  1  2  3  4
н  4  3  2  2  1  2  3
ы  5  4  3  3  2  1  2
е  6  5  4  4  3  2  1

Расстояние Дамерау-Левенштейна (матричный): 1
Матрица алгоритма:
/  0  Р  а  з  н  ы  е
0  0  1  2  3  4  5  6
Р  1  0  1  2  3  4  5
а  2  1  0  1  2  3  4
в  3  2  1  1  2  3  4
н  4  3  2  2  1  2  3
ы  5  4  3  3  2  1  2
е  6  5  4  4  3  2  1
```

Рисунок 3.1 — Демонстрация работы программы на одном из функциональных тестов

Вывод

В этом разделе были рассмотрены средства реализации и листинги алгоритмов поиска редакционного расстояния.

4 Исследовательская часть

В данном разделе будет описано проведение замерного эксперимента, приведены его результаты, а также сформулированы выводы на их основе. <цель исследования, на какой машине делали (указать ЦПУ, ОЗУ, ОС), желательно написать, как исследовали и при каких условиях; что получили в результате (таблицы + графики)>

4.1 Постановка замерного эксперимента

Замерный эксперимент был поставлен на отладочной плате NUCLEO-F767ZI [6], позволяющей запускать программы на языке *MicroPython* [7].

Технические характеристики NUCLEO-F767ZI:

— Разрядность шины данных — 32 бита;

— Процессор — Arm Cortex-M7;

— Объем ОЗУ — 512 Кбайт;

Замеры времени проводились с помощью функции *ticks_us* [8] библиотеки *utime* [4]. Данная функция позволяет получить процессорное время в микросекундах, отсчитываемое от некоторой референтной точки.

Для рекурсивного алгоритма Левенштейна замеры времени проводились на длинах строк от 1 до 4, так как максимальная глубина рекурсии при работе на отладочной плите ограничена. В силу ограничений по памяти на матричных алгоритмах были проведены замеры на длинах строк от 1 до 64, при этом для алгоритма Левенштейна с меморизацией максимально допустимая длина строк составила 20.

4.2 Результаты замерного эксперимента

В результате проведения эксперимента были получены данные, приведенные в таблице 4.1.

Таблица 4.1 — Результаты проведения замерного эксперимента на отладочной плате. Все числа приведены в миллисекундах

Длина строк	Рекурсия Левенштейн	Рекурсия с кэшированием	матричный Левенштейн	матричный Дамерау-Левенштейн
1	2	2	2	2
2	6	5	3	4
3	27	10	5	6
4	139	16	7	9
5	-	32	10	13
6	-	46	13	17
7	-	65	17	22
8	-	88	21	28
10	-	142	31	41
12	-	229	43	58
14	-	320	58	77
16	-	428	75	102
18	-	622	92	125
20	-	819	112	157
22	-	-	134	189
24	-	-	160	229
26	-	-	188	267
28	-	-	216	313
30	-	-	246	352
32	-	-	281	408
34	-	-	319	456
36	-	-	356	512
38	-	-	397	570
40	-	-	438	643
42	-	-	486	707
44	-	-	531	783
46	-	-	579	848
48	-	-	633	928
50	-	-	693	1008
52	-	-	739	1108
54	-	-	807	1197
56	-	-	868	1277
58	-	-	936	1384
60	-	-	1000	1483
62	-	-	1072	1587
64	-	-	1145	1706

Визуализация и сравнение рекурсивных и нерекурсивных алгоритмов на полученных данных продемонстрировано на рисунках 4.1-4.2.

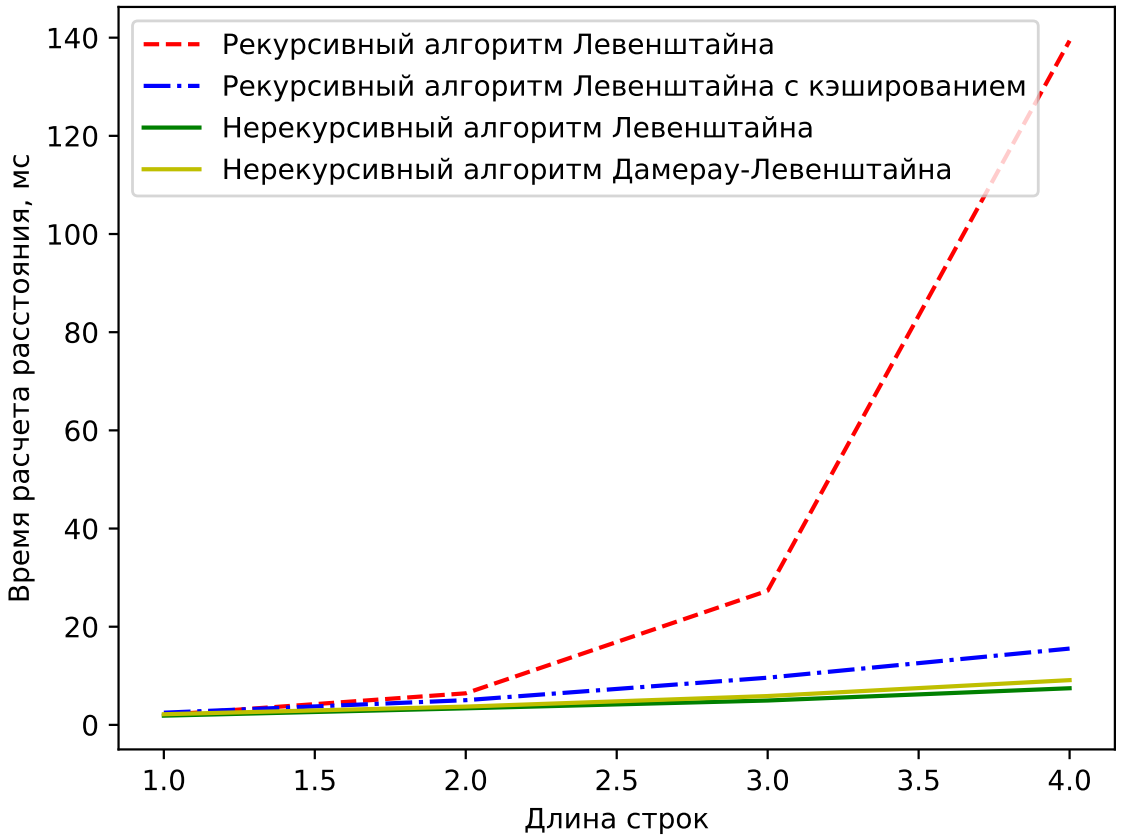

Рисунок 4.1 — Сравнение времени работы алгоритмов поиска редакционного расстояния

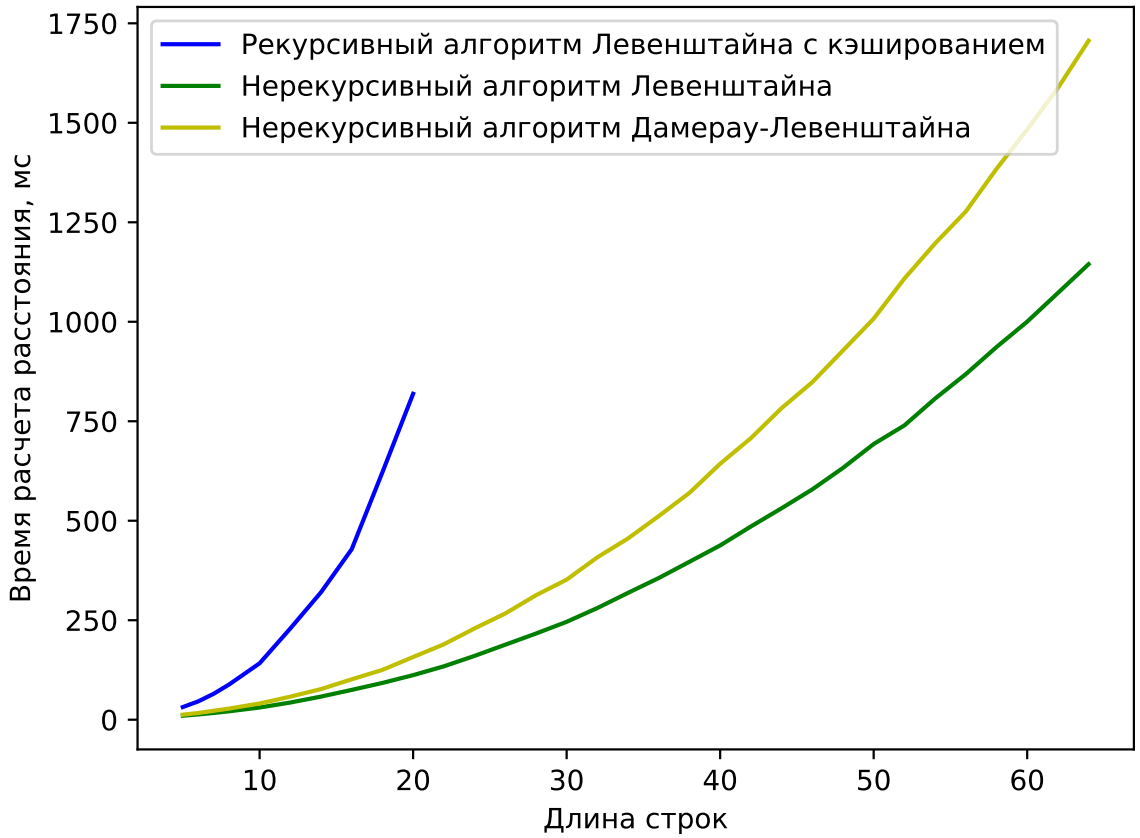

Рисунок 4.2 — Сравнение времени работы матричных алгоритмов поиска редакционного расстояния

Из полученных данных видно, что рекурсивный алгоритм вычисления расстояния Левенштейна является наиболее медленным в силу большого числа рекурсивных вызовов. Рекурсивный алгоритм с меморизацией работает быстрее реализации без запоминания, поскольку не происходит повторных вычислений, но все еще медленнее матричных алгоритмов из-за затрат, связанных с многократным вызовом функции (выделение кадра стека, копирование параметров и адреса возврата, выполенение передачи управления).

На небольших размерах строк матричные алгоритмы достаточно близки друг к другу по затратам времени, однако с ростом длин обрабатываемых строк растет и преимущество в эффективности алгоритма Левенштейна. Матричные алгоритмы Левенштейна и Дамерау-Левенштейна отличаются по временной эффективности в 1.2-1.5 раза на длинах строк до 64. Это наблюдение можно объяснить тем, что в алгоритме Дамерау-Левенштейна содержится дополнительная проверка на возможность транспозиции и в случае успешности проверки дополнительное вычисление минимума двух чисел.

4.3 Вывод

В результате проведения замерного эксперимента было выявлено, что матричная нерекурсивная реализация алгоритма Левенштейна является наиболее эффективной по времени выполнения по сравнению с другими рассмотреными реализациями.

При этом если размер оперативной памяти достаточно велик и допустимо удерживать большой объем памяти долгое время, лучше подойдет матричный алгоритм. Если же имеется достаточный объем памяти (по абсолютному значению больший, чем в предыдущем случае), но нет возможности удерживать максимальный объем занимаемой памяти длительное время, лучше подойдет рекурсивный алгоритм.

ЗАКЛЮЧЕНИЕ

В ходе выполнения лабораторной работы были изучены алгориты Левенштейна и Дамерау-Левенштейна, реализованы их рекурсивные (с кешированием и без) и нерекурсивные реализации, проведены замерные эксперименты, в которых было выявлено, что матричные реализации алгоритмов Левенштейна и Дамерау-Левенштейна являются наиболее эффективными по времени выполнения, однако занимают значительный объем памяти в течение всего времени выполнения программы. Матричные алгоритмы различаются между собой по временной эффективности в 1.2-1.5 раза в пользу алгоритма Левенштейна за счет меньшего числа производимых операций.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Расстояние Левенштейна [Электронный ресурс]. URL: https://www.codecamp.ru/blog/levenshtein-distance-in-python (дата обращения 25.09.24);

2. Расстояние Дамерау-Левенштайна [Электронный ресурс]. URL: https://programm.top/c-sharp/algorithm/damerau-levenshtein-distance (дата обращения 28.09.24);

3. Язык программирования Python [Электронный ресурс]. URL: https://www.python.org (дата обращения 28.09.24);

4. Библиотека utime [Электронный ресурс]. URL: https://docs.micropython.org/en/v1.15/library/utime.ht (дата обращения 28.09.24);

5. Библиотека Matplotlib [Электронный ресурс]. URL: https://matplotlib.org (дата обращения 28.09.24);

6. Отладочная плата NUCLEO-F767ZI [Электронный ресурс]. URL: https://www.st.com/en/evaluation-tools/nucleo-f767zi.html (дата обращения 28.09.24);

7. MicroPython [Электронный ресурс]. URL: https://micropython.org (дата обращения 28.09.24);

8. Описание функции ticks_us библиотеки utime [Электронный ресурс]. URL: https://docs.micropython.org/en/v1.15/library/utime.html#utime.ticks_us (дата обращения 28.09.24);