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 № 1
по дисциплине «Анализ алгоритмов»

Тема Расстояние Левенштейна

Студент Равашдех Ф.Х.

Группа ИУ7-55Б

Преподаватели Строганов Ю.В.

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна (англ. Levenshtein distance) (также редакционное расстояние или дистанция редактирования) между двумя строками в теории информации и компьютерной лингвистике [1] — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой (также транспозиции двух соседних символов для расстояния Дамерау — Левенштейна [2]), необходимых для превращения одной строки в другую.

Цель лабораторной работы — сравнение алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна. Для достижения поставленной цели необходимо выполнить следующие задачи:

1) реализовать указанные алгоритмы поиска расстояния (два матричных, рекурсивный без кэширования и рекурсивный с кэшированием);
2) провести замеры затраченного процессорного времени выполнения реализованных алгоритмов;
3) провести исследование затрачиваемого процессорного времени и памяти при различных реализациях алгоритмов
4) провести сравнительный анализ реализаций алгоритмов по полученным данным;
5) описать результаты в отчете.

1 Аналитическая часть

1.1 Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние, дистанция редактирования) — метрика, измеряющая разность между двумя последовательностями символов.

Введены операции нескольких типов:

— I (англ. insert) — операция вставки символа;

— D (англ. delete) — операция удаления символа;

— R (англ.replace) — операция замены символа.

Символ λ обозначает пустой символ строки, не входящий ни в одну из рассматриваемых строк.

Стоимость каждой из вышеизложенных операций равна 1:

— $D(a, b) = 1$, $a \neq b$, в противном случае замена не происходит;

— $D(\lambda, b) = 1$;

— $D(a, \lambda) = 1$.

Совпадение символов обозначено как M (англ. match), таким образом $D(a, a) = 0$.

Для решения проблемы взаимного выравнивания строк (в случае когда строки различной длины, существует множество способов сопоставить соответствующие символы), была введена рекуррентная формула.

1) $L1$ — длина S_1;

2) $L2$ — длина S_2;

3) $S_1[1...i]$ — подстрока S_1 длиной i, начиная с первого символа;

4) $S_2[1...j]$ — подстрока S_2 длиной j, начиная с первого символа;

Расстояние Левенштейна между двумя строками S_1 и S_2 длиной M и N соответственно рассчитывается по рекуррентной формуле (1.1).

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} \max(i, j), & \text{если } \min(i, j) = 0, \\ \min(D(S_1[1..i], S_2[1..j-1]) + 1, & \\ \quad D(S_1[1..i-1], S_2[1..j]) + 1, & \text{иначе,} \\ \quad D(S_1[1..i-1], S_2[1..j-1]) + & \\ \quad +m(S_1[i], S_2[j])), & \end{cases} \tag{1.1}$$

где сравнение символов строк S_1 и S_2 рассчитывается как:

$$m(a, b) = \begin{cases} 0, & \text{если a = b,} \\ 1, & \text{иначе.} \end{cases} \tag{1.2}$$

Обозначим результат подсчета расстояния Левенштейна для подстрок $S_1[1..i], S_2[1..j]$ как $Lev(S_1[1..i], S_2[1..j])$.

1.2 Расстояние Дамерау — Левенштейна

В расстоянии Дамерау — Левенштейна была введена еще одна операция, которую обозначили как S (англ. swap), данная операция применима, только если $S_1[i] = S_2[j-1]$ и $S_1[i-1] = S_2[j]$. Рекуррентная формула данной метрики выглядит следующим образом:

$$D(S_1[1..i], S_2[1..j]) = \begin{cases} min \begin{cases} Lev(S_1[1..i], S_2[1..j]), \\ D(S_1[1..i-2], S_2[1..j-2]) + 1, \end{cases} & \text{если i > 1, j > 1,} \\ & S_1[i] = S_2[j-1], \\ & S_1[i-1] = S_2[j], \\ Lev(S_1[1..i], S_2[1..j]) & \text{иначе.} \end{cases} \tag{1.3}$$

Идея в том, что после замены пары символов местами, полученные пары были поэлементно равны друг другу.

Вывод

В данной части были введены необходимые обозначения для поиска расстояний Левенштейна и Дамерау — Левенштейна и выведены рекуррентные отношения для поиска их значения.

2 Конструкторская часть

2.1 Нерекурсивный алгоритм нахождения расстояния Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Левенштейна представлена на рисунке 2.1.

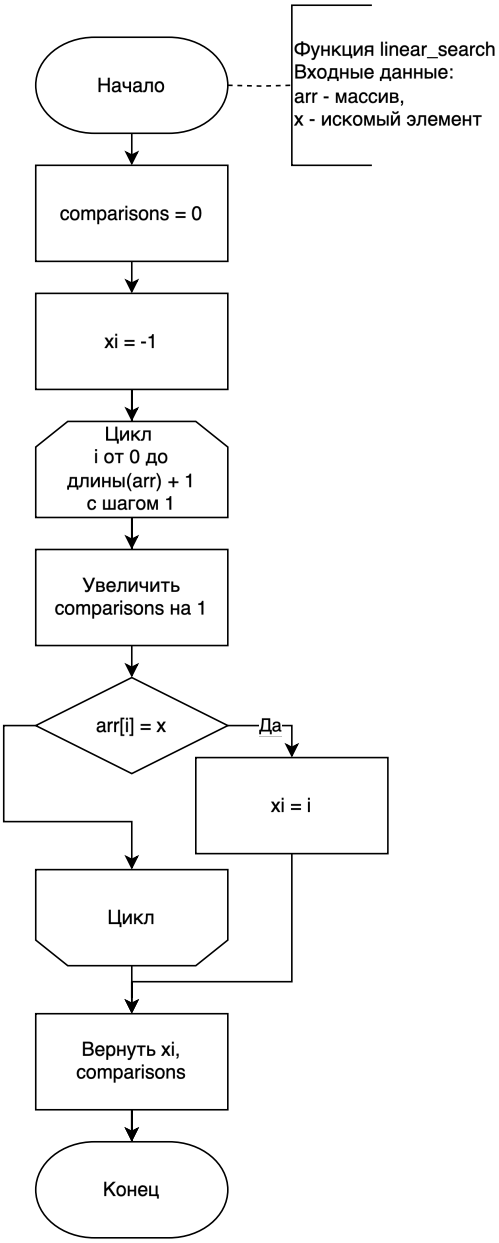


Рисунок 2.1 — Схема нерекурсивного алгоритма нахождения расстояния Левенштейна

2.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Схема рекурсивного алгоритма нахождения расстояния Левенштейна представлена на рисунке 2.2.

2.3 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем

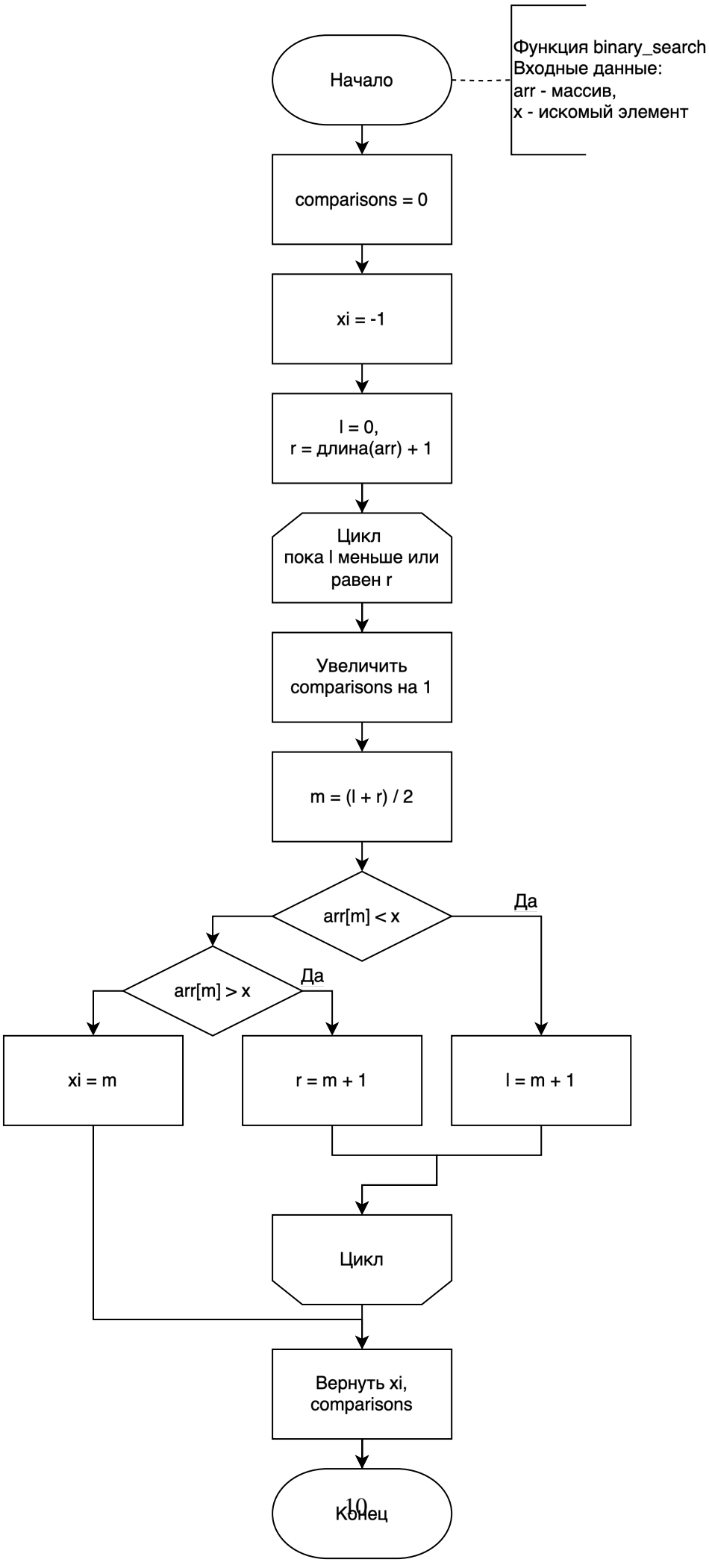
Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем представлена на рисунке 2.3.

2.4 Нерекурсивный алгоритм нахождения расстояния Дамерау — Левенштейна

Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна представлена на рисунке 2.4.

Вывод

В данной части работы были описаны алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна.


Начало
Функция binary_search
Входные данные:
arr - массив,
x - искомый элемент
comparisons = 0
xi = -1
l = 0,
r = длина(arr) + 1
Цикл
пока l меньше или
равен r
Увеличить
comparisons на 1
m = (l + r) / 2
arr[m] < x
Да
arr[m] > x
Да
xi = m
r = m + 1
l = m + 1
Цикл
Вернуть xi,
comparisons
Конец

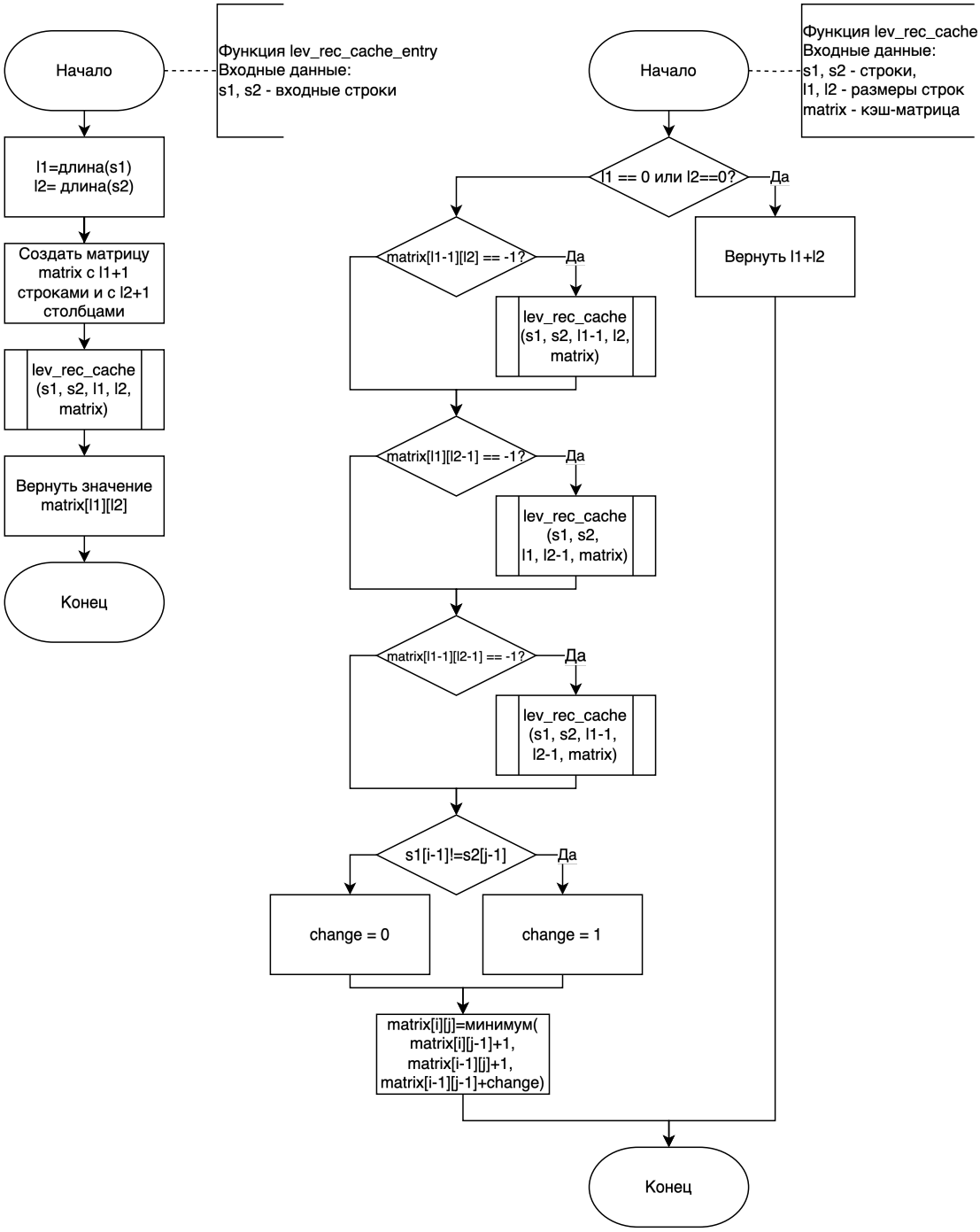


Рисунок 2.3 — Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем

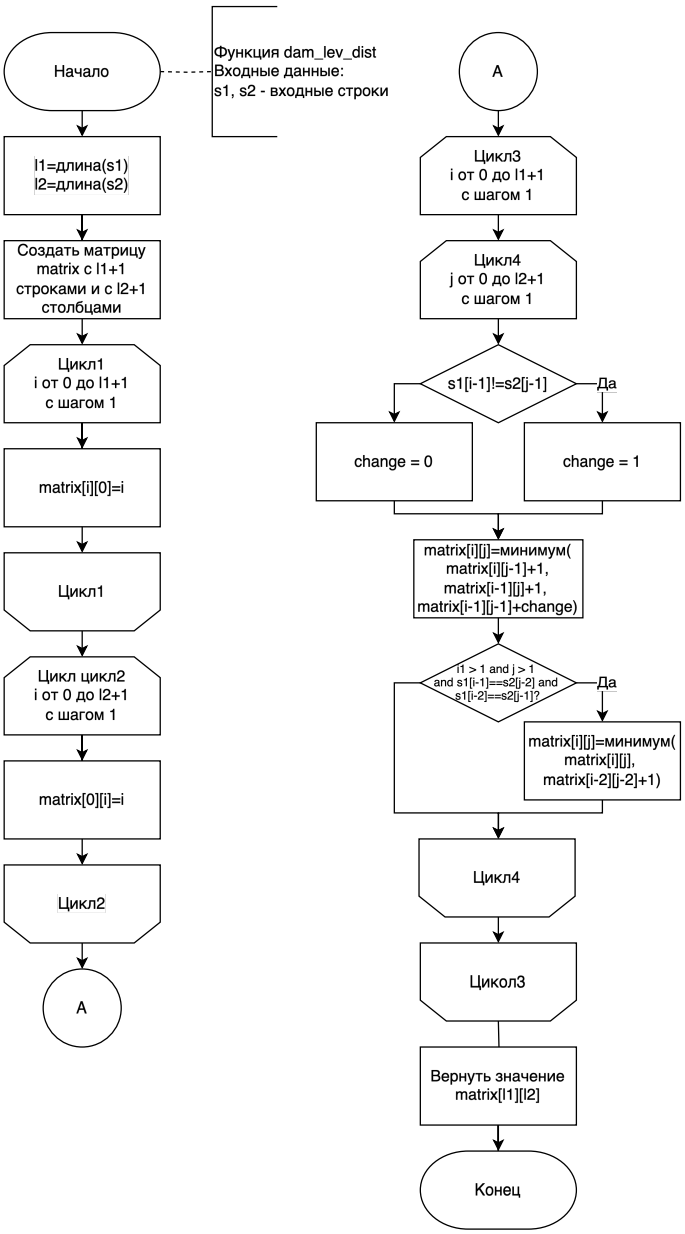


Рисунок 2.4 — Схема нерекурсивного алгоритма нахождения расстояния Дамерау — Левенштейна

3 Технологическая часть

3.1 Требования к программному обеспечению

Входные данные: две строки на русском или английском языке в любом регистре;

Выходные данные: искомое расстояние для каждого метода.

3.2 Средства реализации

Алгоритмы для данной лабораторной работы были реализованы на языке MicroPython (реализация Python) с помощью функции замера времени ticks_ms [3].

3.3 Реализация алгоритмов

Реализация алгоритма нахождения расстояния Левенштейна и Дамерау — Левенштейна представлена в листинга (3.1) - (3.4).

Листинг 3.1 — Функция нахождения расстояния Левенштейна нерекурсивно

```
def lev_dist(s1, s2):
  l1, l2 = len(s1), len(s2)
  matrix = [[0 for _ in range(l2 + 1)] for _ in range(l1 + 1)]

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

  for i in range(1, l1 + 1):
    for j in range(1, l2 + 1):
      change = 1 if s1[i - 1] != s2[j - 1] else 0
      matrix[i][j] = min(
        matrix[i][j - 1] + 1,
        matrix[i - 1][j] + 1,
        matrix[i - 1][j - 1] + change
      )
  return matrix[l1][l2]
```

Листинг 3.2 — Функция нахождения расстояния Левенштейна рекурсивно

```
def lev_dist_rec(s1, s2, l1, l2):
  if l1 == 0:
    return l2
```

```
  if l2 == 0:
    return l1

  change = 1 if s1[l1 - 1] != s2[l2 - 1] else 0
  res = min(
    lev_dist_rec(s1, s2, l1, l2 - 1) + 1,
    lev_dist_rec(s1, s2, l1 - 1, l2) + 1,
    lev_dist_rec(s1, s2, l1 - 1, l2 - 1) + change
  )
  return res
```

Листинг 3.3 — Функция нахождения расстояния Левенштейна рекурсивно с кэшем

```
def lev_rec_cache_entry(s1, s2):
  l1, l2 = len(s1), len(s2)

  matrix = [[-1 for _ in range(l2 + 1)] for _ in range(l1 + 1)]
  lev_rec_cache(s1, s2, l1, l2, matrix)
  return matrix[l1][l2]

def lev_rec_cache(s1, s2, l1, l2, matrix):
  if l1 == 0:
    matrix[l1][l2] = l2
  elif l2 == 0:
    matrix[l1][l2] = l1
  else:
    if matrix[l1 - 1][l2] == -1:
      lev_rec_cache(s1, s2, l1 - 1, l2, matrix)
    if matrix[l1][l2 - 1] == -1:
      lev_rec_cache(s1, s2, l1, l2 - 1, matrix)
    if matrix[l1 - 1][l2 - 1] == -1:
      lev_rec_cache(s1, s2, l1 - 1, l2 - 1, matrix)

    change = 1 if s1[l1 - 1] != s2[l2 - 1] else 0
    matrix[l1][l2] = min(
      matrix[l1][l2 - 1] + 1,
      matrix[l1 - 1][l2] + 1,
      matrix[l1 - 1][l2 - 1] + change
    )
```

Листинг 3.4 — Функция нахождения расстояния Дамерау — Левенштейна нерекурсивно

```
def dam_lev_dist(s1, s2):
  l1, l2 = len(s1), len(s2)
  matrix = [[0 for _ in range(l2 + 1)] for _ in range(l1 + 1)]

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

  for i in range(1, l1 + 1):
    for j in range(1, l2 + 1):
      change = 1 if s1[i - 1] != s2[j - 1] else 0
      matrix[i][j] = min(
        matrix[i][j - 1] + 1,
        matrix[i - 1][j] + 1,
        matrix[i - 1][j - 1] + change
      )
      if i > 1 and j > 1 and s1[i - 1] == s2[j - 2] and s1[i - 2] == s2
         [j - 1]:
        matrix[i][j] = min(matrix[i][j], matrix[i - 2][j - 2] + 1)
  return matrix[l1][l2]
```

3.4 Тестирование

В таблице [3.1] представлены данные о результатах тестирования реализации алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна. Все тесты пройдены успешно.

Таблица 3.1 — Модульные тесты

Входные данные		**Алгоритм и расстояние**			
		Левенштейна			**Дамерау — Левенштейна**
Строка 1	**Строка 2**	**Итеративный**	**Рекурсивный**		**Итеративный**
			Без кэша	**С кэшем**	
polynomial	exponential	6	6	6	6
silent	listen	4	4	4	4
кот	скат	2	2	2	2
ab	ba	2	2	2	1

Вывод

В данном разделе были рассмотрены требования к программному обеспечению, используемые средства реализации, приведена реализация алгоритмов и результаты тестирования.

4 Исследовательская часть

4.1 Технические характеристики

Характеристики устройства (STM32F767 [4]), на котором выполнялись замеры времени:

1) процессор Arm® 32-bit Cortex®-M7 CPU, 216 МГц;
2) оперативная память: 512 Кбайт.

4.2 Время выполнения алгоритмов

Замеры времени работы алгоритмов для каждой длины строк проводились по 100 раз для каждого алгоритма (строки случайно сгенерируются на каждой итерации, т.е. на итерации все алгоритмы работают с одинаковыми строками), затем значение времени усреднялось. Данные представлены в таблице 4.1.

Таблица 4.1 — Таблица замеров времени работы алгоритмов (в мс)

Длина строк	Левенштейн	Лев. рек.	Лев. рек. с кэшем	Дамерау — Левенштейн
0	0.09	0.01	0.12	0.11
1	0.15	0.06	0.14	0.14
2	0.20	0.23	0.28	0.24
3	0.35	1.15	0.57	0.40
4	0.58	5.95	0.94	0.70
5	0.82	30.97	1.39	1.03

На рисунке 4.1 показан график зависимости времени работы алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна от длины строк.

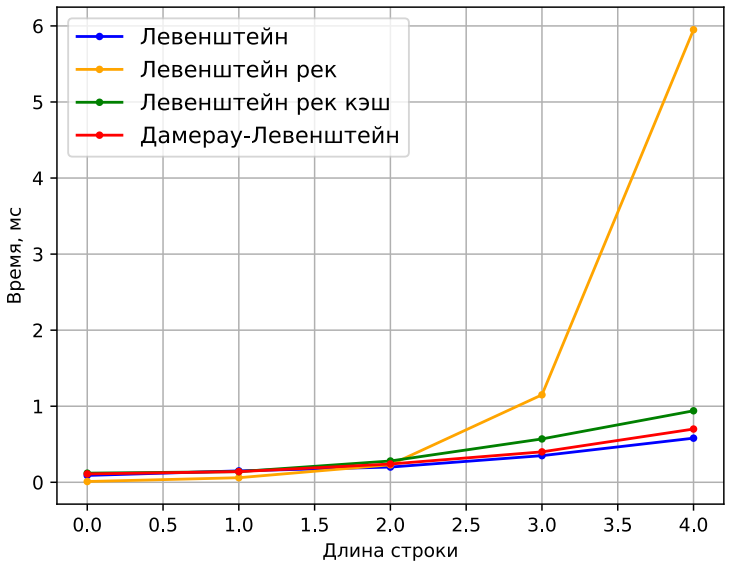


Рисунок 4.1 — График сравнения алгоритмов на строках до длины 4

На рисунке 4.2 показан график зависимости времени работы тех же алгоритмов, что и на рисунке 4.1, но без учета рекурсивного алгоритма без кэширования.

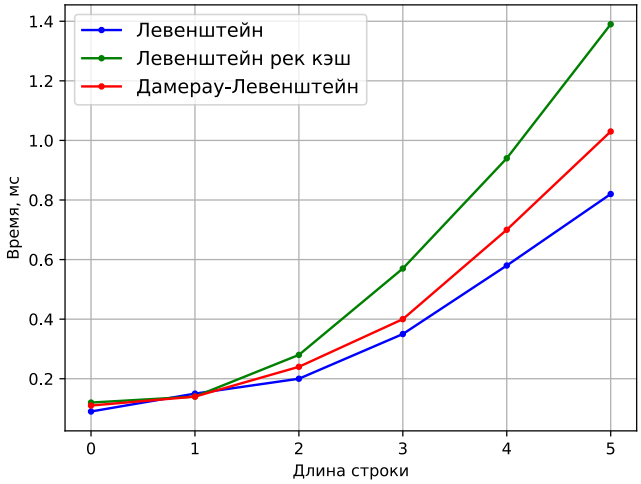


Рисунок 4.2 — График сравнения алгоритмов (кроме рекурсивного без кэша)

На рисунке [4.3] показан график зависимости времени работы нерекурсивных алгоритмов от длины строк.

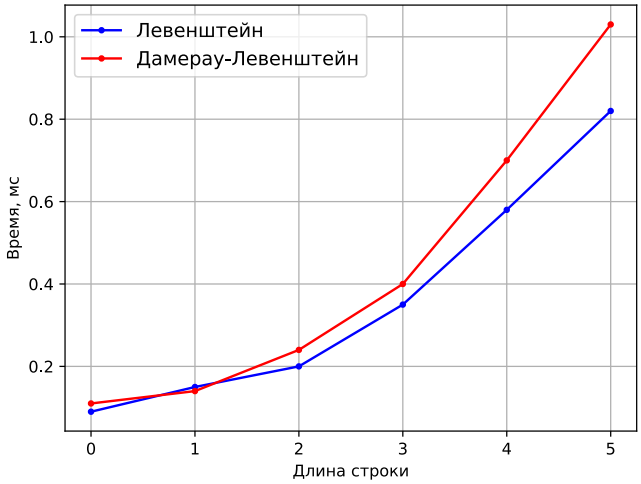


Рисунок 4.3 — График сравнения нерекурсивных алгоритмов нахождения расстояний Левенштейна и Дамерау — Левенштейна

4.3 Вывод

В данном разделе были приведены технические характеристики и время выполнения алгоритмов. Рекурсивный алгоритм без кэширования, вычисляющий расстояние Левенштейна, выполнялся дольше других вариантов. Нерекурсивный алгоритм вычисления расстояния Дамерау — Левенштейна выполнялся медленнее, чем нерекурсивный алгоритм вычисления расстояния Левенштейна, т.к. содержал дополнительные действия.

ЗАКЛЮЧЕНИЕ

Цель работы достигнута: сравнение алгоритмов нахождения расстояния Левенштейна и Дамерау — Левенштейна было проведено.

В ходе выполнения лабораторной работы были решены все задачи:

1) реализованы указанные алгоритмы поиска расстояния (два матричных, рекурсивный без кэширования и рекурсивный с кэшированием);

2) проведены замеры затраченного процессорного времени выполнения реализованных алгоритмов;

3) проведено исследование затрачиваемого процессорного времени и памяти при различных реализациях алгоритмов

4) проведен сравнительный анализ реализаций алгоритмов по полученным данным;

5) описаны результаты в отчете.

Сравнительный анализ реализаций показал:

1) самый быстрый — нерекурсивный алгоритм поиска расстояния Левенштейна;

2) самый медленный — рекурсивный алгоритм поиска расстояния Левенштейна;

3) нерекурсивный алгоритм поиска расстояния Левенштейна быстрее нерекурсивного алгоритма поиска расстояния Дамерау — Левенштейна;

4) нерекурсивный алгоритм поиска расстояния Дамерау — Левенштейна быстрее рекурсивного алгоритма поиска расстояния Левенштейна с кэшированием;

5) рекурсивный алгоритм поиска расстояния Левенштейна с кэшированием быстрее рекурсивного алгоритма поиска расстояния Левенштейна без кэширования.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Задача о расстоянии Левенштейна [Электронный ресурс]. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_редакционном_расстоянии,_алгоритм_Вагнера-Фишера (дата обращения 18.09.24)

2. Задача о расстоянии Дамерау — Левенштейна [Электронный ресурс]. Режим доступа: https://neerc.ifmo.ru/wiki/index.php?title=Задача_о_расстоянии_Дамерау-Левенштейна (дата обращения 18.09.24)

3. Функция time_ms для micropython [Электронный ресурс]. Режим доступа: https://docs.micropython.org/en/v1.15/library/utime.html#utime.ticks_ms (дата обращения 19.09.24)

4. STM32F767 datasheet [Электронный ресурс]. Режим доступа: https://www.st.com/resource/en/datasheet/stm32f765bi.pdf (дата обращения 20.09.24)