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»

Отчет по лабораторной работе № 1 по курсу "Анализ алгоритмов"

Тема Расстояние Левенштейна и Дамерау-Левенштейна

Студент Цветков И.А.

Группа ИУ7-53Б

Оценка (баллы) __________

Преподаватель Волкова Л. Л.

Москва — 2021 г.

Оглавление

Введение

Операции работы со строками являются очень важной частью всего программирования. Часто возникает потребность в использовании строк для различных задач - обычные статьи, записи в базу данных и так далее. Отсюда возникает несколько важных задач, для решения которых нужны алгоритмы сравнения строк. Об этих алгоритмах и пойдет речь в данной работе. Подобные алгоритмы используются при:

- исправлении ошибок в тексте, предлагая заменить введенное слово с ошибкой на наиболее подходящее;
- поиске слова в тексте по подстроке;
- сравнении целых текстовых файлов.

Цель работы: изучение, реализация и исследование алгоритмов нахождения расстояний Левенштейна и Дамерау–Левенштейна.

Задачи работы.

1. Изучить и реализовать алгоритмы Левенштейна и Дамерау–Левенштейна.
2. Провести тестирование по времени и по памяти для алгоритмов Левенштейна и Дамерау-Левенштейна.
3. Провести сравнительный анализ по времени рекурсивной и матричной реализации алгоритма нахождения расстояния Левенштейна.
4. Провести сравнительный анализ по времени матричной и с кешем реализации алгоритма нахождения расстояния Левенштейна.
5. Провести сравнительный анализ по времени алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.
6. Описать и обосновать полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1 Аналитическая часть

В данном разделе будут разобраны алгоритмы нахождения расстояния - алгоритмы Левенштейна и Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна [1] между двумя строками - метрика, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую (каждая операция имеет свою "цену штраф).

Редакционное предписание - последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену (и является расстоянием Левенштейна).

Пусть S_1 и S_2 - две строки, длиной N и M соответственно. Введем следующие обозначения:

- I (англ. Insert) - вставка символа в произвольной позиции ($w(\lambda, b) = 1$);
- D (англ. Delete) - удаление символа в произвольной позиции ($w(\lambda, b) = 1$);
- R (англ. Replace) - замена символа на другой ($w(a, b) = 1,\ a \neq b$);
- M (англ. Match) - совпадение двух символов ($w(a, a) = 0$).

С учетом введенных обозначений, расстояние Левенштейна может быть подсчитано по следующей рекуррентной формуле:

$$
D(i,j) = \begin{cases}
0 & \text{i = 0, j = 0} \\
i & \text{j = 0, i > 0} \\
j & \text{i = 0, j > 0} \\
\min\{ & \\
\qquad D(i, j-1) + 1 & \\
\qquad D(i-1, j) + 1 & \text{i > 0, j > 0} \\
\qquad D(i-1, j-1) + m(a[i], b[j]) & (1.2) \\
\} &
\end{cases} \tag{1.1}
$$

Функция 1.2 определена как:

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

1.2 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм вычисления расстояния Левенштейна реализует формулу 1.1

Минимальная цена преобразования - минимальное значение приведенных вариантов.

Если полагать, что *a', b'* - строки a и b без последнего символа соответственно, то цена преобразования из строки a в b может быть выражена так:

1. сумма цены преобразования строки a' в b и цены проведения операции удаления, которая необходима для преобразования a' в a;
2. сумма цены преобразования строки a в b' и цены проведения операции вставки, которая необходима для преобразования b' в b;
3. сумма цены преобразования из a' в b' и операции замены, предполагая, что a и b оканчиваются на разные символы;

4. цена преобразования из a' в b', предполагая, что a и b оканчиваются на один и тот же символ.

1.3 Матричный алгоритм нахождения расстояния Левенштейна

Рекурсивный алгоритм вычисления расстояния Левенштейна может быть не эффективен при больших i и j, так как множество промежуточных значений $D(i,j)$ вычисляются не один раз, что сильно замедляет время выполнения программы.

В качестве оптимизации можно использовать *матрицу* для хранения промежуточных значений. Матрица имеет размеры:

$$(length(S1)+1)*((length(S2)+1), \tag{1.3}$$

где $length(S)$ – длина строки S

Значение в ячейке $[i,j]$ равно значению $D(S1[1...i], S2[1...j])$. Первая строка и первый столбец тривиальны.

Всю таблицу (за исключением первого столбца и первой строки) заполняем в соответствии с формулой 1.4.

$$A[i][j] = min\begin{cases} A[i-1][j]+1 \\ A[i][j-1]+1 \\ A[i-1][j-1]+m(S1[i],S2[j]) \quad (1.5) \end{cases}. \tag{1.4}$$

Функция 1.5 определена как:

$$m(S1[i],S2[j]) = \begin{cases} 0, & \text{если } S1[i-1]=S2[j-1], \\ 1, & \text{иначе} \end{cases}. \tag{1.5}$$

Результат вычисления расстояния Левенштейна будет ячейка матрицы с индексами $i = length(S1)$ и $j = length(S2)$.

1.4 Рекурсивный алгоритм нахождения расстояния Левенштейна с использованием кеша

В качестве оптимизации рекурсивного алгоритма заполнения можно использовать *кеш*, который будет представлять собой матрицу.

Суть оптимизации - при выполнении рекурсии происходит параллельное заполнение матрицы.

Если рекурсивный алгоритм выполняет прогон для данных, которые еще не были обработаны, то результат нахождения заносится в матрицу. Иначе, если обработанные данные встречаются снова, то для них расстояние не находится и алгоритм переходит к следующему шагу.

1.5 Расстояние Дамерау — Левенштейна

Расстояние Дамерау-Левенштейна [2] между двумя строками, состоящими из конечного числа символов — это минимальное число операций вставки, удаления, замены одного символа и транспозиции двух соседних символов, необходимых для перевода одной строки в другую.

Является модификацией расстояния Левенштейна - добавлена операции *транспозиции*, то есть перестановки, двух символов.

Расстояние Дамерау — Левенштейна может быть найдено по формуле 1.6, которая задана как

$$
d_{a,b}(i,j) = \begin{cases}
\max(i,j), & \text{если } \min(i,j) = 0, \\
\min\{ & \\
\qquad d_{a,b}(i,j-1)+1, & \\
\qquad d_{a,b}(i-1,j)+1, & \\
\qquad d_{a,b}(i-1,j-1)+m(a[i],b[j]), & \text{иначе} \\
\qquad \left[\begin{array}{ll} d_{a,b}(i-2,j-2)+1, & \text{если } i,j > 1; \\ & a[i] = b[j-1]; \\ & b[j] = a[i-1] \\ \infty, & \text{иначе} \end{array} \right. & \\
\} &
\end{cases}, \tag{1.6}
$$

Формула выводится по тем же соображениям, что и формула (1.1).

Вывод

В данном разделе были теоретически разобраны формулы Левенштейна и Дамерау-Левенштейна, которые являются рекуррентными, что позволяет реализовать их как рекурсивно, так и итерационно.

2 Конструкторская часть

В этом разделе будут представлены требования к вводу и программе, а также схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

2.1 Требования к вводу

1. На вход подаются две строки.
2. Буквы верхнего и нижнего регистров считаются различными.

2.2 Требования к программе

1. На вход подается две строки - корректный случай, программа должна верно обрабатывать такую ситуацию.
2. В качестве результата работы программа должна вывести число, которое будет являться расстояние Левенштейна (Дамерау-Левенштейна), а также при необходимости матрицу.

2.3 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3 и 2.4 представлены схемы алгоритмов вычисления расстояния Левенштейна и Дамерау-Левенштейна.

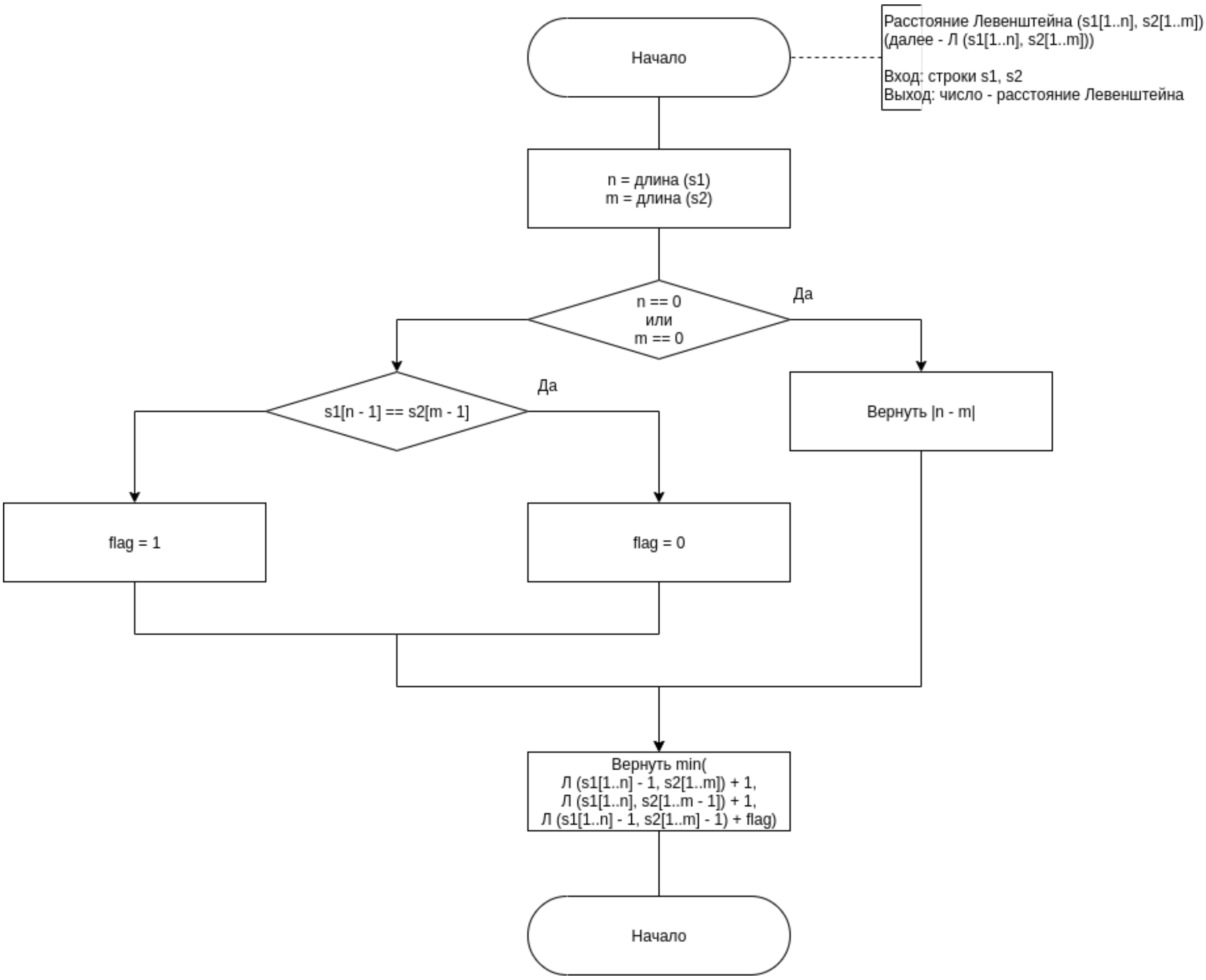

Рисунок 2.1 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна

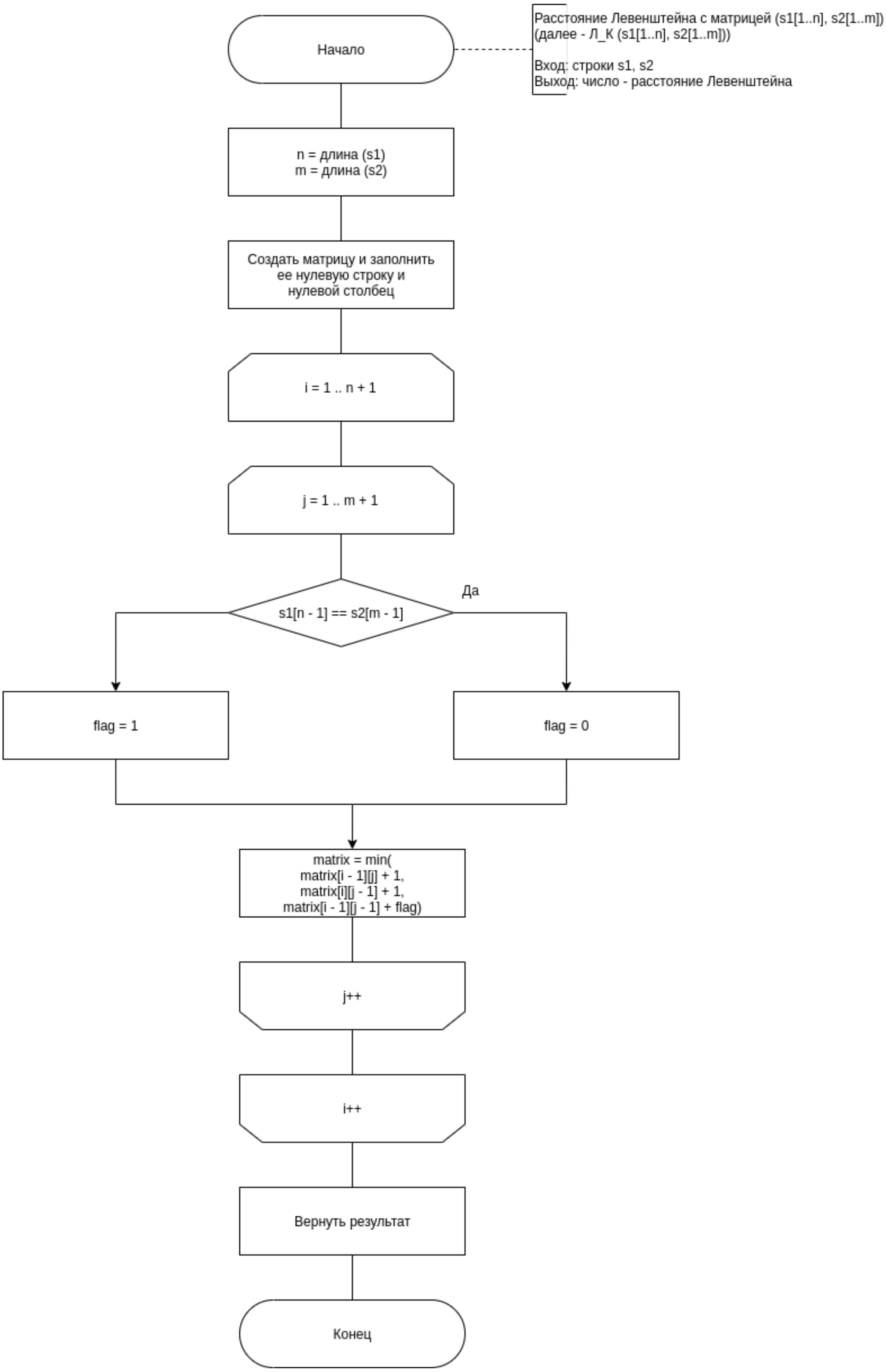

Рисунок 2.2 – Схема матричного алгоритма нахождения расстояния Левенштейна

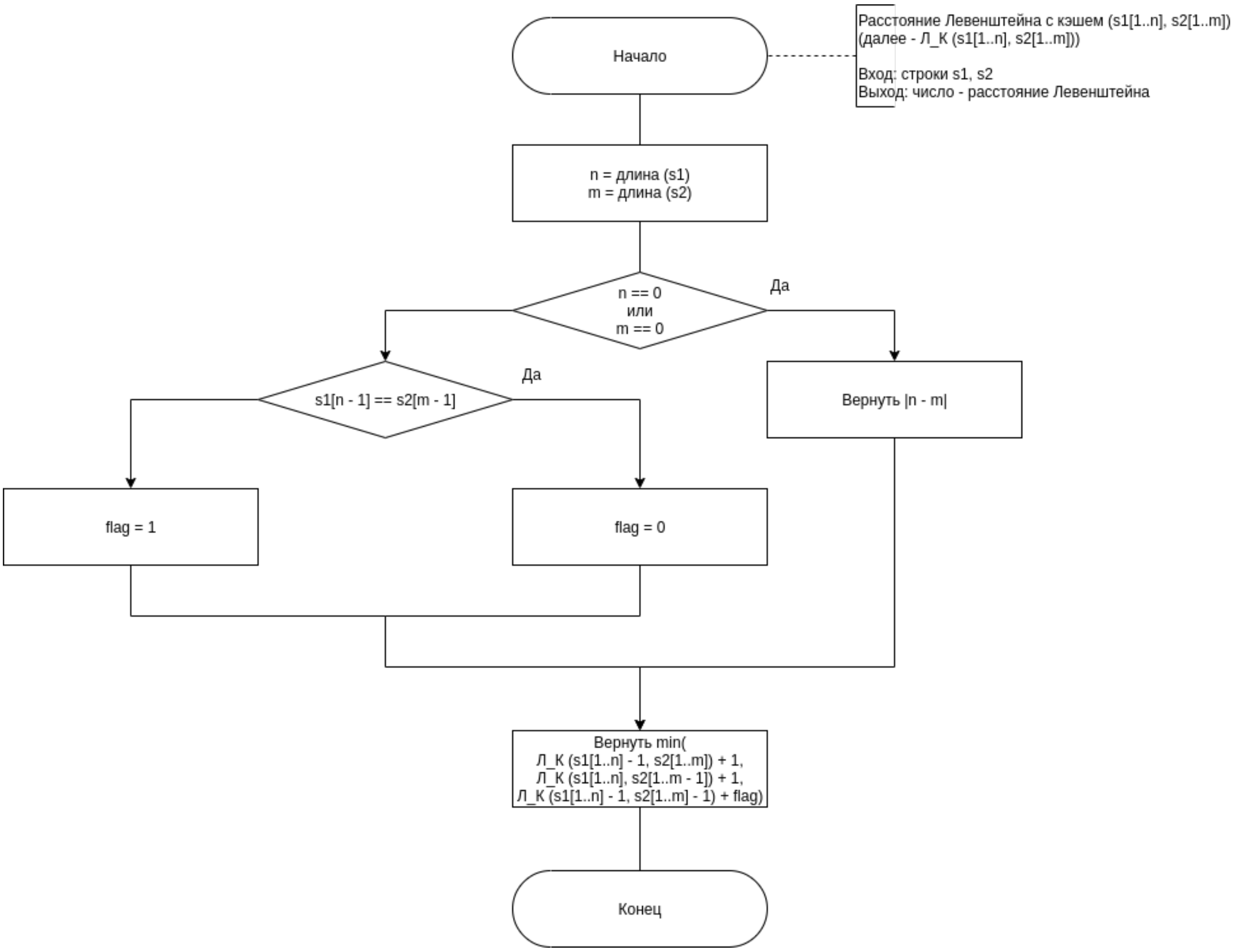

Рисунок 2.3 – Схема рекурсивного алгоритма нахождения расстояния Левенштейна с использованием кеша (матрицы)

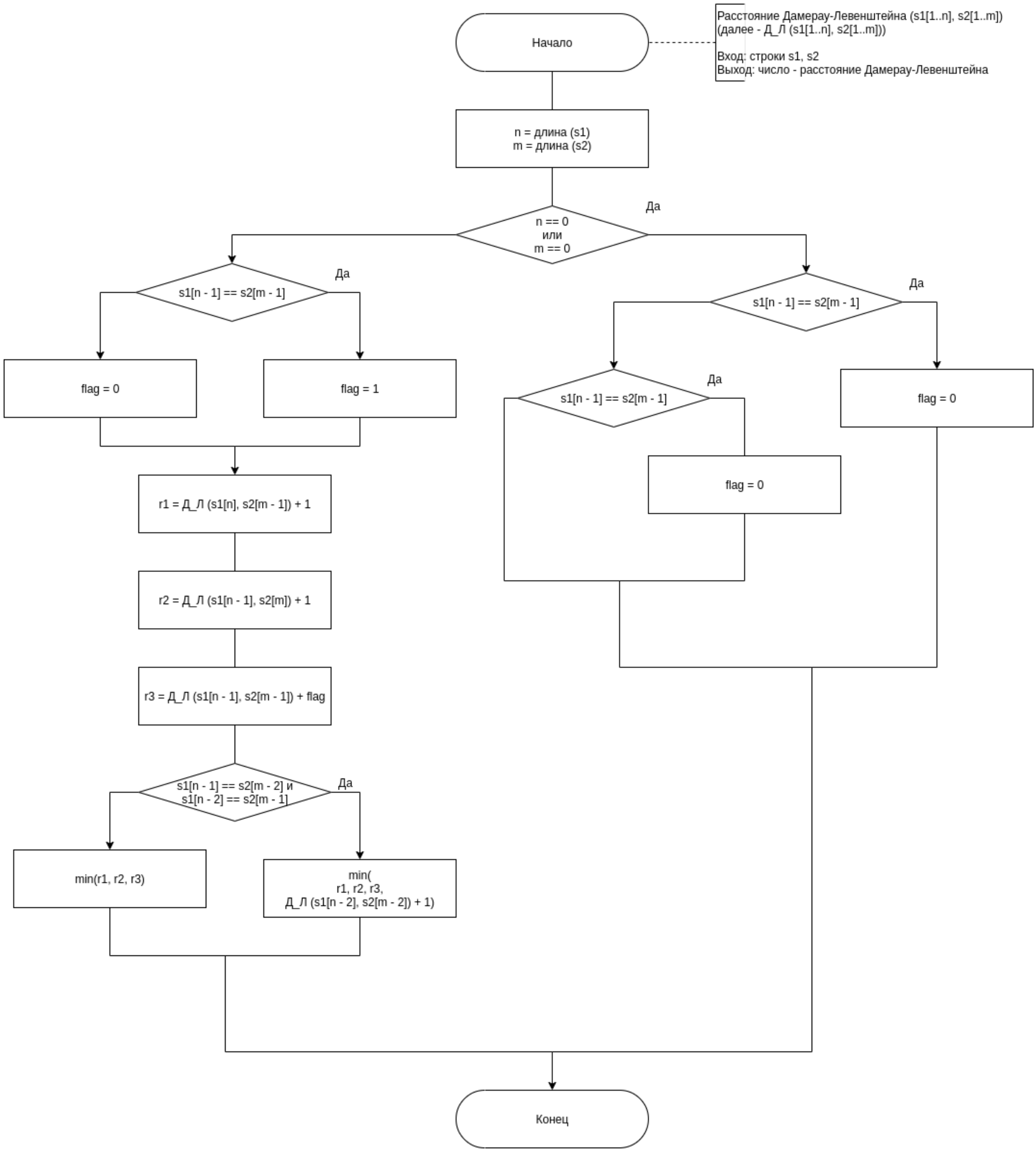

Рисунок 2.4 – Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

Вывод

В данном разделе были представлены требования к вводу и программе, а также схемы алгоритмов, рассматриваемых в лабораторной работе.

3 Технологическая часть

В данном разделе будут рассмотрены средства реализации, а также представлены листинги сортировок.

3.1 Средства реализации

В данной работе для реализации был выбран язык программирования *Python*[3]. В текущей лабораторной работе требуется замерить процессорное время для выполняемой программы, а также построить графики. Все эти инструменты присутствуют в выбранном языке программирования.

Время работы было замерено с помощью функции *process_time(...)* из библиотеки *time*[4].

3.2 Сведения о модулях программы

Программа состоит из двух модулей:

- *main.py* - файл, содержащий весь служебный код;
- *algorythms.py* - файл, содержащий код всех алгоритмов.

3.3 Листинги кода

В листингах 3.1, 3.2, 3.3, 3.4 представлены реализации алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна.

Листинг 3.1 – Алгоритм нахождения расстояния Левенштейна (реккурсивный)

```
def levenstein_recursive(str1, str2):
    n = len(str1)
    m = len(str2)

    if ((n == 0) or (m == 0)):
        return abs(n - m)

    flag = 0

    if (str1[-1] != str2[-1]):
        flag = 1

    min_distance = min(levenstein_recursive(str1[:-1], str2) + 1,
                       levenstein_recursive(str1, str2[:-1]) + 1,
                       levenstein_recursive(str1[:-1], str2[:-1]) + flag)

    return min_distance
```

Листинг 3.2 – Алгоритм нахождения расстояния Левенштейна (матричный)

```
def levenstein_matrix(str1, str2):
    n = len(str1)
    m = len(str2)
    matrix = create_lev_matrix(n + 1, m + 1)

    for i in range(1, n + 1):
        for j in range(1, m + 1):
            add = matrix[i - 1][j] + 1
            delete = matrix[i][j - 1] + 1
            change = matrix[i - 1][j - 1]

            if (str1[i - 1] != str2[j - 1]):
                change += 1

            matrix[i][j] = min(add, delete, change)

    return matrix[n][m]
```

Листинг 3.3 – Алгоритм нахождения расстояния Левенштейна с использованием кеша в виде матрицы

```
def recursive_for_levenstein_cache(str1, str2, n, m, matrix):
    if (matrix[n][m] != -1):
        return matrix[n][m]

    if (n == 0):
        matrix[n][m] = m
        return matrix[n][m]

    if ((n > 0) and (m == 0)):
        matrix[n][m] = n
        return matrix[n][m]

    add = recursive_for_levenstein_cache(str1, str2, n - 1, m, matrix) + 1
    delete = recursive_for_levenstein_cache(str1, str2, n, m - 1, matrix) + 1
    change = recursive_for_levenstein_cache(str1, str2, n - 1, m - 1, matrix)

    if (str1[n - 1] != str2[m - 1]):
        change += 1 # flag

    matrix[n][m] = min(add, delete, change)

    return matrix[n][m]

def levenstein_cache_matrix(str1, str2):
    n = len(str1)
    m = len(str2)

    matrix = create_lev_matrix(n + 1, m + 1)

    for i in range(n + 1):
        for j in range(m + 1):
            matrix[i][j] = -1

    recursive_for_levenstein_cache(str1, str2, n, m, matrix)

    return matrix[n][m]
```

Листинг 3.4 – Алгоритм нахождения расстояния Дамерау-Левенштейна (реккурсивный)

```
def damerau_levenstein_recursive(str1, str2):
    n = len(str1)
    m = len(str2)

    if ((n == 0) or (m == 0)):
        if (n != 0):
            return n

        if (m != 0):
            return m

        return 0

    flag = 0 if (str1[-1] == str2[-1]) else 1

    add = damerau_levenstein_recursive(str1[:n - 1], str2) + 1
    delete = damerau_levenstein_recursive(str1, str2[:m - 1]) + 1
    change = damerau_levenstein_recursive(str1[:n - 1], str2[:m - 1]) + flag
    extra_change = damerau_levenstein_recursive(str1[:n - 2], str2[:m - 2]) + 1

    if ((n > 1) and (m > 1) and (str1[-1] == str2[-2]) and (str1[-2] == str2[-1])):
        minimum  = min(add, delete, change, extra_change)

    else:
        minimum = min(add, delete, change)

    return minimum
```

3.4 Функциональные тесты

В таблице 3.1 приведены тесты для функций, реализующих алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна. Тесты *для всех алгоритмов* пройдены успешно.

Таблица 3.1 – Функциональные тесты

			Ожидаемый результат	
№	Строка 1	Строка 2	Левенштейн	Дамерау-Л.
1	крот	кот	1	1
2	тестировщик	тесто	6	6
3	программист	программа	2	1
4	"пустая строка"	привет	6	6
5	октябрь	"пустая строка"	8	8
8	ремонт	емонт	1	1
9	гигиена	иена	3	3
10	нисан	автоваз	6	6
11	спасибо	пожалуйста	9	9
12	что	кто	1	1
13	ты	тыква	3	3
14	есть	кушать	4	4
15	вот	в	2	2

Вывод

Были представлены всех алгоритмов нахождения расстояния Левенштейна и Дамерау-Левенштейна, которые были описаны в предыдущем разделе. Также в данном разделе была приведена информации о выбранных средствах для разработки алгоритмов.

4 Исследовательская часть

В данном разделе будут приведены примеры работы программа, а также проведен сравнительный анализ алгоритмов при различных ситуациях на основе полученных данных.

4.1 Технические характеристики

Технические характеристики устройства, на котором выполнялось тестирование представлены далее:

- операционная система: Ubuntu 20.04.3 [5] Linux [6] x86_64;
- память: 8 GiB;
- процессор: Intel® Core™ i5-7300HQ CPU @ 2.50GHz [7].

При тестировании ноутбук был включен в сеть электропитания. Во время тестирования ноутбук был нагружен только встроенными приложениями окружения, а также системой тестирования.

4.2 Демонстрация работы программы

На рисунке 4.1 представлен результат работы программы.

```
      Меню

     1. Расстояние Левенштейна (рекурсивно)
     2. Расстояние Левенштейна (матрица)
     3. Расстояние Левенштейна (рекурсивно с кешем)
     4. Росстояние Дамерау-Левенштейна (рекурсивно)
     5. Замерить времени
     0. Выход

        Выбор:     2

Введите 1-ую строку:    попытка
Введите 2-ую строку:    непытка

Матрица, с помощью которой происходило вычисление расстояния Левенштейна:

0  0  н  е  п  ы  т  к  а
0  0  1  2  3  4  5  6  7
п  1  1  2  2  3  4  5  6
о  2  2  2  3  3  4  5  6
п  3  3  3  2  3  4  5  6
ы  4  4  4  3  2  3  4  5
т  5  5  5  4  3  2  3  4
к  6  6  6  5  4  3  2  3
а  7  7  7  6  5  4  3  2

Результат:  2
```

Рисунок 4.1 – Пример работы программы

4.3 Время выполнения алгоритмов

Как было сказано выше, используется функция замера процессорного времени process_time(...) из библиотеки time на Python. Функция возвращает пользовательское процессорное время типа float.

Использовать функцию приходится дважды, затем из конечного времени нужно вычесть начальное, чтобы получить результат.

Замеры проводились для длины слова от 0 до 9 по 100 раз на различных входных данных.

Результаты замеров приведены в таблице 4.1 (время в мс).

Таблица 4.1 – Результаты замеров времени

Длина	Л.(рек)	Л.(матр.)	Л.(рек с матр.)	Д.-Л.(рек.)
0	0.0033	0.0074	0.0089	0.0032
1	0.0081	0.0111	0.0156	0.0092
2	0.0261	0.0191	0.0273	0.0280
3	0.0647	0.0115	0.0192	0.0570
4	0.1654	0.0113	0.0226	0.2544
5	0.8275	0.0141	0.0260	1.2789
6	4.1587	0.0199	0.0365	7.1449
7	24.6731	0.0309	0.0568	42.2119
8	125.5106	0.0317	0.0613	227.0448
9	651.1621	0.0393	0.0752	1278.5680

Также на рисунках 4.2, 4.3, 4.4 приведены графические результаты замеров.

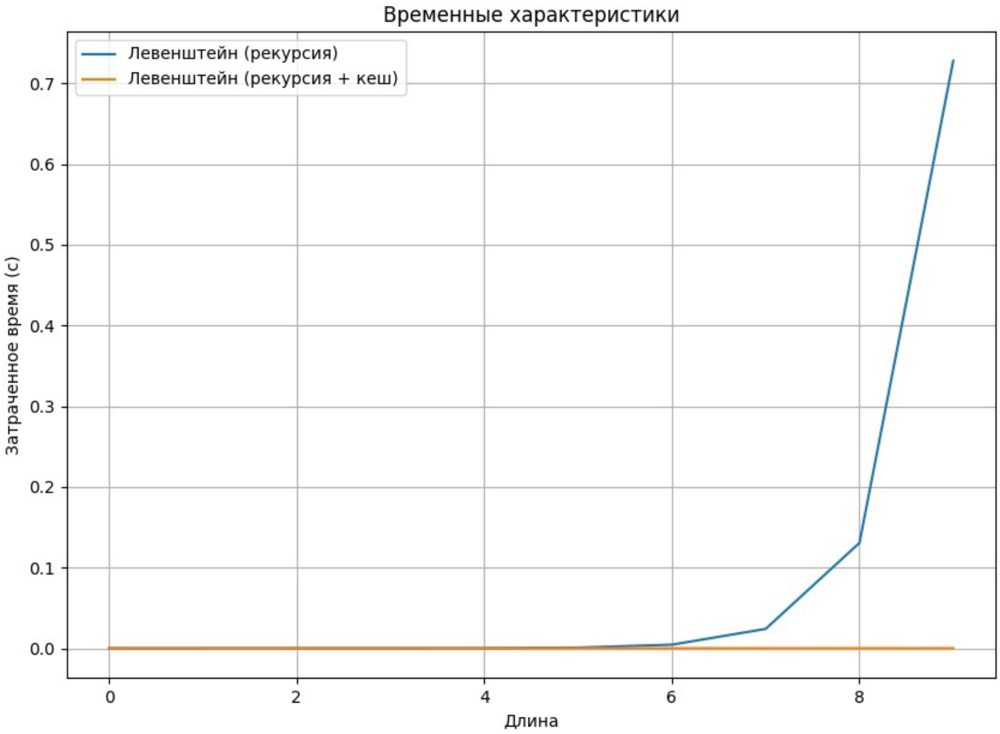

Рисунок 4.2 – Сравнение по времени алгоритмов Левенштейна с использованием рекурсии и с использованием кеша (матрица + рекурсия)

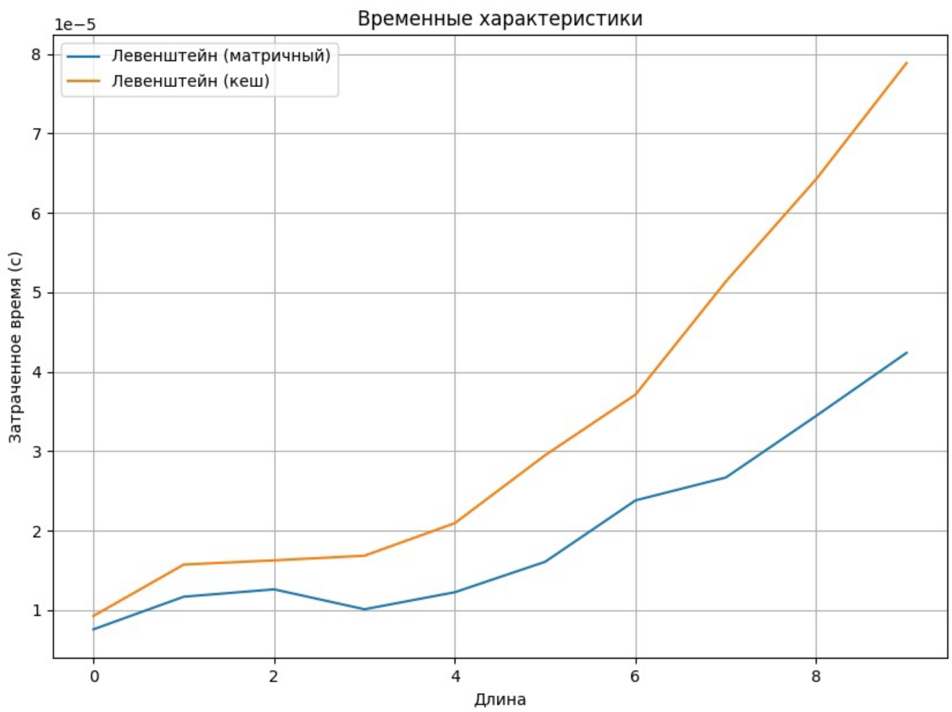

Рисунок 4.3 – Сравнение алгоритмов нахождения расстояния Левенштейна матричного и с кешем в виде матрицы

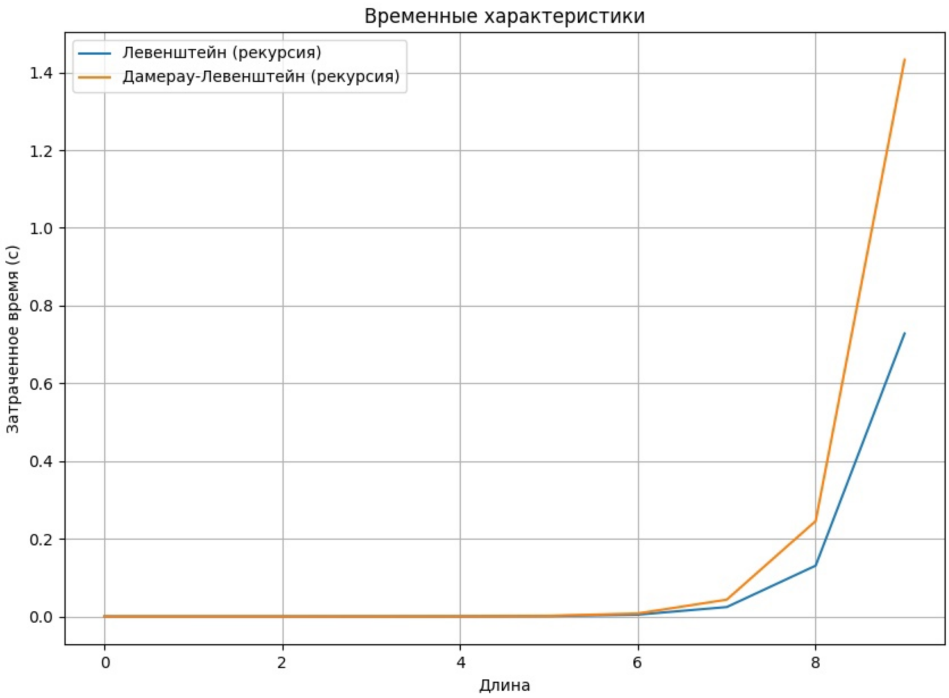

Рисунок 4.4 – Сравнение по времени рекурсивных алгоритмов Левенштейна и Дамерау-Левенштейна

4.4 Использование памяти

С точки зрения замеров и сравнения используемой памяти, алгоритмы Левенштейна и Дамерау-Левенштейна не отличаются друг от друга. Тогда рассмотрим только рекурсивную и матричную реализации данных алгоритмов.

Пусть:

- n - длина строки S1
- m - длина строки S2

Тогда затраты по памяти будут такими:

- алгоритм нахождения расстояния Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 - (n + m) * sizeof(char)
  - для n, m - 2 * sizeof(int)
  - доп. переменные - 2 * sizeof(int)
  - адрес возврата
- алгоритм нахождения расстояния Левенштейна с использованием кеша в виде матрицы (память на саму матрицу: ((n + 1) * (m + 1)) * sizeof(int)) (рекурсивный), где для каждого вызова:
  - для S1, S2 - (n + m) * sizeof(char)
  - для n, m - 2 * sizeof(int)
  - доп. переменные - 2 * sizeof(int)
  - ссылка на матрицу - 8 байт
  - адрес возврата
- алгоритм нахождения расстояния Дамерау-Левенштейна (рекурсивный), где для каждого вызова:
  - для S1, S2 - (n + m) * sizeof(char)

  - для n, m - 2 * sizeof(int)
  - доп. переменные - 2 * sizeof(int)
  - адрес возврата

- алгоритм нахождения расстояния Левенштейна (матричный):
  - для матрицы - ((n + 1) * (m + 1)) * sizeof(int))
  - текущая строка матрицы - (n + 1) * sizeof(int)
  - для S1, S2 - (n + m) * sizeof(char)
  - для n, m - 2 * sizeof(int)
  - доп. переменные - 3 * sizeof(int)
  - адрес возврата

Вывод

Исходя из замеров по памяти, итеративные алгоритмы проигрывают рекурсивным, потому что максимальный размер памяти в них растет, как произведение длин строк, а в рекурсивных - как сумма длин строк.

Исходя из полученных результатов замера времени, при увеличении длины слова рекурсивная реализация алгоритма Левенштейна начинает сильно проигрывать матричной реализации - при длине слова в 8 символов эта разница превышает несколько порядков.

Алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна примерно сопоставимы между собой, поскольку оба имеют рекурсивную реализацию, поэтому расхождение по времени у них не большое.

Заключение

Цель, которая была поставлена в начале лабораторной работы была достигнута, а также в ходе выполнения лабораторной работы были решены следующие задачи:

- были изучены и реализованы алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна;
- были также изучены матричная реализация, а также реализация с использованием кеша в виде матрицы для алгоритма Левенштейна;
- проведен сравнительный анализ алгоритмов Левенштейна и Дамерау-Левенштейна, а также сравнение рекурсивной и матричной реализаций, матричной и с кешом реализаций алгоритма Левенштейна
- подготовлен отчет о лабораторной работе.

Список литературы

[1] Вычисление редакционного расстояния [Электронный ресурс]. Режим доступа: `https://habr.com/ru/post/117063/` (дата обращения: 4.10.2021).

[2] Нечёткий поиск в тексте и словаре [Электронный ресурс]. Режим доступа: `https://habr.com/ru/post/114997/` (дата обращения: 4.10.2021).

[3] Welcome to Python [Электронный ресурс]. Режим доступа: `https://www.python.org` (дата обращения: 04.10.2021).

[4] time — Time access and conversions [Электронный ресурс]. Режим доступа: `https://docs.python.org/3/library/time.html#functions` (дата обращения: 04.10.2021).

[5] Ubuntu 20.04.3 LTS (Focal Fossa) [Электронный ресурс]. Режим доступа: `https://releases.ubuntu.com/20.04/` (дата обращения: 04.10.2021).

[6] Linux [Электронный ресурс]. Режим доступа: `https://www.linux.org/forums/#linux-tutorials.122` (дата обращения: 04.10.2021).

[7] Процессор Intel® Core™ i5-7300HQ [Электронный ресурс]. Режим доступа: `https://ark.intel.com/content/www/ru/ru/ark/products/97456/intel-core-i5-7300hq-processor-6m-cache-up-to-3-50-ghz.html` (дата обращения: 04.10.2021).