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 Динамическое программирование

Студент Бугаков И. С.

Группа ИУ7-54Б

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

Москва, 2024

СОДЕРЖАНИЕ

ВВЕДЕНИЕ

Расстояние Левенштейна — метрика, показывающая, разницу между двумя последовательностями символов. Данное редакционное расстояние основано на минимальном количестве операций вставки, удаления, изменения символов необходимом для преобразования одной строки в другую.

Впервые задача была поставлена в 1965 году Владимиром Левенштейном при исследовании 0 – 1 последовательностей [6]. Позднее была обобщена на произвольные алфавиты. В дальнейшем Фредерик Дамерау добавил операцию транспозиции, показав, что 80% ошибок при печати представлены этими операциями [4].

Нахождение редакционного расстояния применяется для решения следующих задач:

— исправление ошибок при вводе текста в поисковых системах, базах данных, при автоматическом распознавании речи;

— сравнение текстовых файлов утилитой diff;

— сравнение хромосом, геном, белков в биоинформатике.

Цель данной работы — исследование алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна.

Задачи лабораторной работы:

1) описать алгоритмы поиска расстояний Левенштейна и Дамерау—Левенштейна;
2) реализовать следующие алгоритмы:
    - нерекурсивный алгоритм поиска расстояния Левенштейна;
    - рекурсивный алгоритм поиска расстояния Левенштейна;
    - рекурсивный алгоритм поиска расстояния Левенштейна с мемоизацией;
    - нерекурсивный алгоритм поиска расстояния Дамерау—Левенштейна;
3) определить подходящие для измерения процессорного времени инструменты;
4) провести анализ временных и емкостных затрат различных реализаций.

1 Аналитическая часть

1.1 Расстояние Левенштейна

1.1.1 Описание алгоритма

Расстояние Левенштейна — одна из разновидностей редакционного расстояния [5], метрика, которая отражает разницу между двумя строками. Расстояние Левенштейна показывает наименьшее количество операций вставки, удаления, замены символа необходимых для преобразования одной строки к другой.

В общем случае каждая операция имеет свою собственную стоимость:

— $\omega(a, b),\ a \neq b$ — стоимость замены буквы a на b;

— $\omega(\lambda, a)$ — стоимость вставки буквы a;

— $\omega(a, \lambda)$ — стоимость удаления буквы a.

В дальнейшем стоимость каждой из операций принимается равной 1:

— $\omega(a, b) = 1,\ a \neq b$;

— $\omega(\lambda, a) = 1$;

— $\omega(a, \lambda) = 1$.

Отдельно введен случай совпадения символов. Его стоимость равна 0 — $\omega(a, a) = 0$.

Пусть функция $D(i, j)$ - расстояние Левенштейна между префиксами рассматриваемых строк, т. е. $s_1[1 \dots i]$ и $s_2[1 \dots j]$. Тогда расстояние Левенштейна для двух строк s_1 и s_2 длины n и m соответственно, может быть вычислено по рекурентной формуле:

$$D(i,j) = \begin{cases} 0 & i = 0,\ j = 0 \\ i & i > 0,\ j = 0 \\ j & i = 0,\ j > 0 \\ min \begin{cases} D(i-1,j) \\ D(i,j-1) \\ D(i-1,j-1) + match(s_1[i], s_2[j]) \end{cases} & i > 0,\ j > 0 \end{cases} \tag{1.1}$$

где $match(a, b)$ - функция совпадения символов:

$$match(a,b) = \begin{cases} 0 & a \neq b \\ 1 & a = b \end{cases} \tag{1.2}$$

1.1.2 Рекурсивная реализация

Данная реализация представляет собой в точности имплементацию формулы (1.1).

1.1.3 Рекурсивная реализация с мемоизацией

Для вычисления минимума в формуле (1.1) происходит многократный пересчет промежуточных значений. Чтобы избежать повторных вычислений, введена матрица $A[(n+1) \times (m+1)]$ в ячейке $A_{i,j}$ которой будет записываться значение $D(i,j)$ при первом расчете. При повторной необходимости вычислить $D(i,j)$, значение будет подставлено из матрицы.

1.1.4 Нерекурсивная реализация

Рекурсивная реализация с мемоизацией имеет дополнительные временные и емкостные затраты на рекурсивные вызовы. Поэтому для нахождения расстояния Левенштейна был предложен подход динамического программирования. При таком подходе предполагается итерационное заполнение матрицы A. При этом порядок обхода возможен как по строкам, так и по столбцам, так как все необходимые значения для текущего шага итерации уже будут вычислены. Расстояние Левенштейна для исходных строк будет при этом находиться в $A_{n,m}$.

1.2 Расстояние Дамерау—Левенштейна

1.2.1 Описание алгоритма

Расстояние Дамерау — Левенштейна — также одна из разновидностей редакционных расстояний. Представляет собой модификацию расстояния Левенштейна, путем добавления операции транспозиции, т. е. перестановки букв в строке для совпадения с другой строкой.

Расстояние Дамерау—Левенштейна также может быть вычислено по рекурентной формуле:

$$DL(i,j) = \begin{cases} 0 & i = 0,\ j = 0 \\ i & i > 0,\ j = 0 \\ j & i = 0,\ j > 0 \\ min \begin{cases} DL(i-1,j) \\ DL(i,j-1) \\ DL(i-1,j-1) + match(s_1[i], s_2[j]) \\ DL(i-2,j-2) + 1 \end{cases} & i > 1,\ j > 1, \\ & s_1[i] = s_2[j-1], \\ & s_1[i-1] = s_2[j] \\ min \begin{cases} DL(i-1,j) \\ DL(i,j-1) \\ DL(i-1,j-1) + match(s_1[i], s_2[j]) \end{cases} & \text{иначе} \end{cases} \tag{1.3}$$

где $match(a, b)$ — по-прежнему функция совпадения символов (1.2), $DL(i, j)$ — расстояние Дамерау — Левенштейна для префиксов $s_1[1 \dots i]$ и $s_2[1 \dots j]$.

1.2.2 Нерекурсивная реализация

Данный алгоритм аналогичен нерекурсивному поиску расстояния Левенштейна. В ячейку $A_{i,j}$ матрицы $A[(n+1) \times (m+1)]$ записывается значение $DL(i, j)$. Заполнение также может производиться по строкам или столбцам, а окончательный ответ по-прежнему будет в $A_{n,m}$.

Вывод

Рассмотрены алгоритмы нахождения расстояний Левенштейна и Дамерау—Левенштейна. Так как оба алгоритма заданы рекурентными формулами, возможна их рекурсивная или итеративная реализация с использованием подхода динамического программирования.

2 Конструкторская часть

Ниже приведены схемы алгоритмов, упомянутых в аналитической части.

2.1 Поиск расстояния Левенштейна

Ниже приведены схемы различных реализаций алгоритма поиска расстояния Левенштейна:

— рисунок 2.1 - схема рекурсивной реализации;

— рисунок 2.2 - схема рекурсивной реализации с мемоизацией;

— рисунок 2.3 - схема итеративной реализации.

2.1.1 Рекурсивная реализация

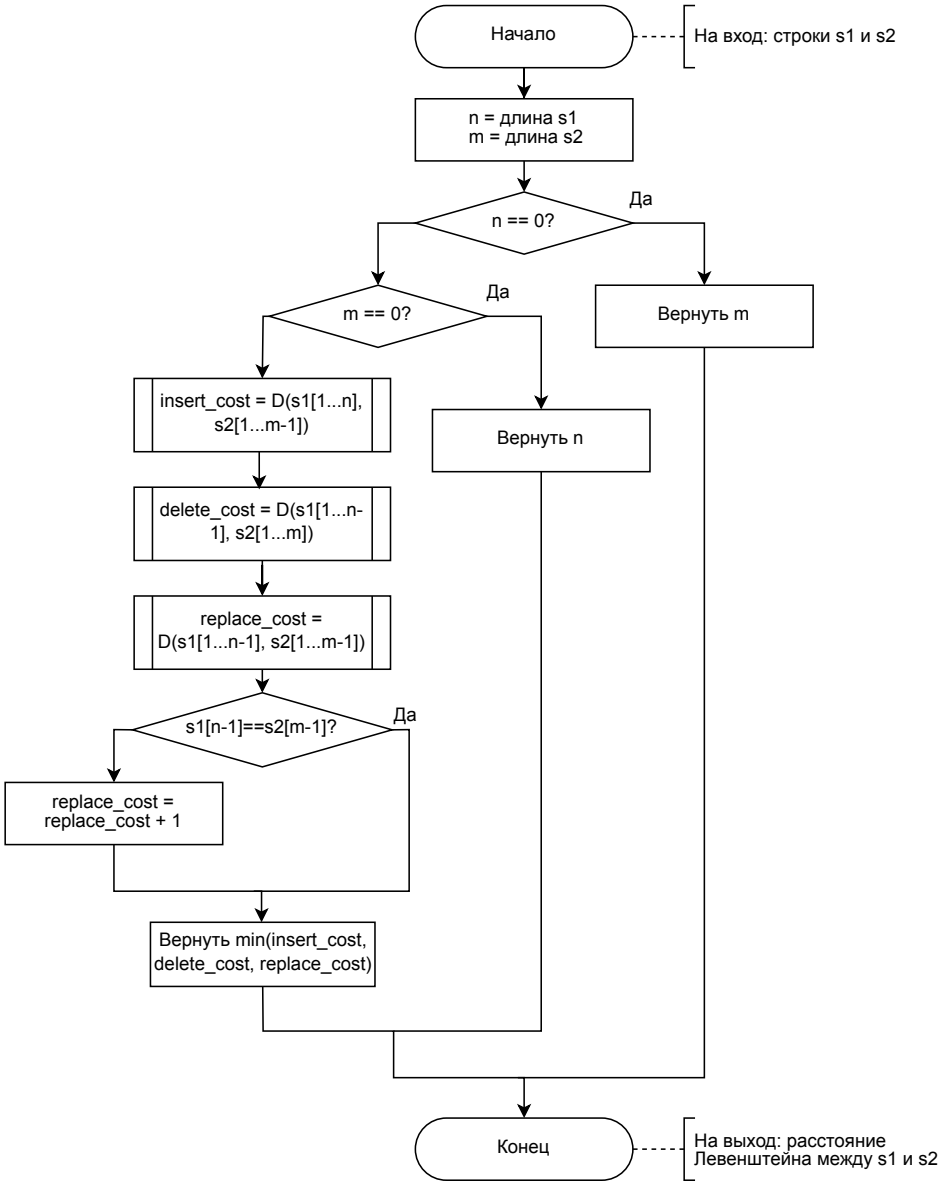

Рисунок 2.1 — Схема рекурсивной реализации алгоритма поиска расстояния Левенштейна

2.1.2 Рекурсивная реализация с мемоизацией

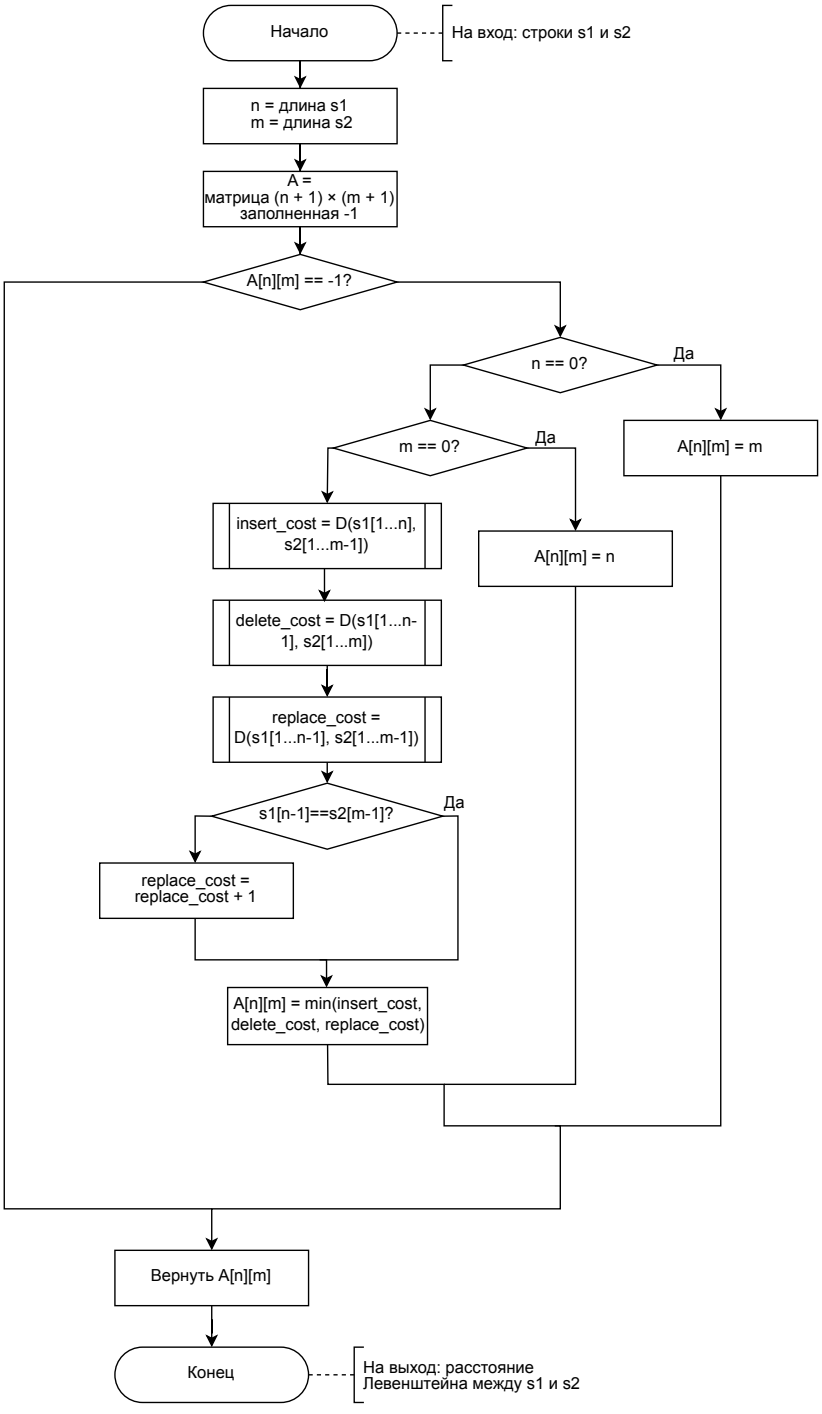

Рисунок 2.2 — Схема рекурсивной реализации алгоритма поиска расстояния Левенштейна с мемоизацией

2.1.3 Нерекурсивная реализация

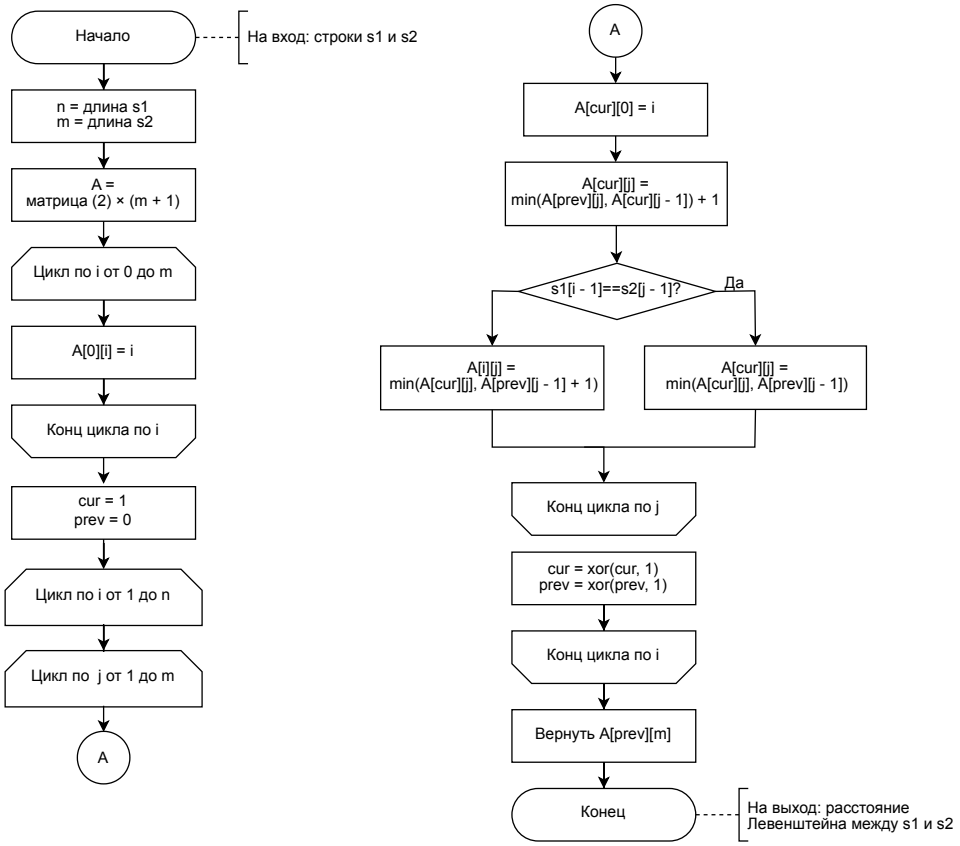

Рисунок 2.3 — Схема нерекурсивной реализации алгоритма поиска расстояния Левенштейна

2.2 Поиск расстояния Дамерау—Левенштейна

На рисунке 2.4 приведена схема итеративной реализации алгоритма поиска расстояния Дамерау-Левенштейна.

2.2.1 Нерекурсивная реализация

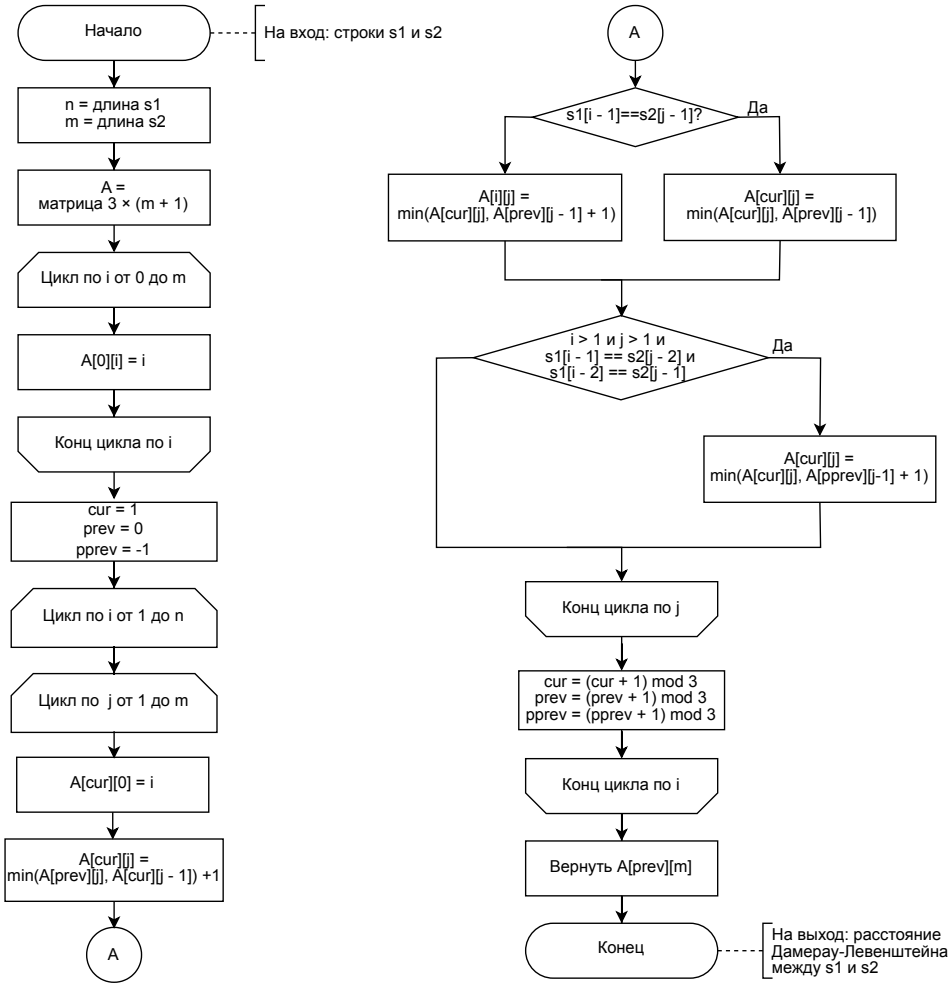

Рисунок 2.4 — Схема нерекурсивной реализации алгоритма поиска расстояния Дамерау-Левенштейна

Вывод

На основе аналитической части построены схемы алгоритмов для реализации.

3 Технологическая часть

3.1 Выбор языка программирования

Для реализации указанных алгоритмов и выполнения лабораторной работы был выбран язык C++, т. к. программа на нем может быть запущена на микроконтроллере для замеров времени работы алгоритма. Также данный язык предоставляет возможность работать с динамической памятью с помощью контейнерных классов.

Была выбрана IDE Arduino-IDE, т. к. она дает возможность запускать код на выбранном языке напрямую на микроконтроллере, обладает функциями автодополнения и подсветки синтаксиса.

3.2 Реализации алгоритмов

В листингах ниже представлены реализации указанных алгоритмов:

— в листинге 3.1 — рекурсивная реализация алгоритма поиска расстояния Левенштейна;

— в листинге 3.2 и 3.3 — рекурсивные реализации алгоритма с мемоизацией. Первая функция является интерфейсной, вторая необходима для использования матрицы с уже рассчитанными значениями;

— в листинге 3.4 — итерационная реализация с использованием подхода динамического программирования;

— в листинге 3.5 — итерационная реализация алгоритма поиска расстояния Дамерау—Левенштейна с использованием подхода динамического программирования.

Листинг 3.1 — Рекурсивная функция поиска расстояния Левенштейна

```
int levenstein_recursion(const string &s1, const string &s2) {
  if (s1.empty())
    return static_cast<int>(s2.size());
  else if (s2.empty())
    return static_cast<int>(s1.size());
  else {
    int ans = min(levenstein_recursion(s1.substr(0, s1.size() - 1),
       s2),
    levenstein_recursion(s1, s2.substr(0, s2.size() - 1))) + 1;
    if (s1[s1.size() - 1] == s2[s2.size() - 1])
      return min(ans, levenstein_recursion(s1.substr(0, s1.size() -
         1), s2.substr(0, s2.size() - 1)));
    else
      return min(ans, levenstein_recursion(s1.substr(0, s1.size() -
         1), s2.substr(0, s2.size() - 1)) + 1);
  }
}
```

Листинг 3.2 — Вызываемая функция рекурсивного поиска расстояния Левенштейна с мемоизацией

```
int levenstein_recursion_memoization(const string &s1, const string &
   s2) {
  int n = static_cast<int>(s1.size()) + 1, m = static_cast<int>(s2.
     size()) + 1;
  if (s1.empty())
    return static_cast<int>(s2.size());
  else if (s2.empty())
    return static_cast<int>(s1.size());
  else {
    vector <vector<int>> dp(n, vector<int>(m, -1));
    return dp[s1.size()][s2.size()] =
       levenstein_recursion_memoization(s1, s2, dp);
  }
}
```

Листинг 3.3 — Перегруженная функция рекурсивного поиска расстояния Левенштейна с мемоизацией

```
int levenstein_recursion_memoization(const string &s1, const string &
   s2, vector <vector<int>> &dp) {
  if (dp[s1.size()][s2.size()] == -1) {
    if (s1.empty())
```

```
      return static_cast<int>(s2.size());
    else if (s2.empty())
      return static_cast<int>(s1.size());
    else {
      dp[s1.size()][s2.size()] = min(levenstein_recursion_memoization
         (s1.substr(0, s1.size() - 1), s2, dp),
      levenstein_recursion_memoization(s1, s2.substr(0, s2.size() -
         1), dp)) +
      1;
      if (s1[s1.size() - 1] == s2[s2.size() - 1])
        return dp[s1.size()][s2.size()] = min(dp[s1.size()][s2.size()
           ],
        levenstein_recursion(s1.substr(0, s1.size() - 1),
        s2.substr(0, s2.size() - 1)));
      else
        return dp[s1.size()][s2.size()] = min(dp[s1.size()][s2.size()
           ],
        levenstein_recursion(s1.substr(0, s1.size() - 1),
        s2.substr(0, s2.size() - 1)) + 1);
    }
  }
  return dp[s1.size()][s2.size()];
}
```

Листинг 3.4 — Функция нерекурсивного поиска расстояния Левенштейна

```
int levenstein(const string &s1, const string &s2) {
  int n = static_cast<int>(s1.size()) + 1,
  m = static_cast<int>(s2.size()) + 1;
  vector <vector<int>> dp(2, vector<int>(m));
  for (int j = 1; j < m; ++j)
  dp[0][j] = j;
  int cur = 1, prev = 0;
  for (int i = 1; i < n; ++i) {
    dp[cur][0] = i;
    for (int j = 1; j < m; ++j) {
      dp[cur][j] = min(dp[cur][j - 1], dp[prev][j]) + 1;
      if (s1[i - 1] == s2[j - 1])
        dp[cur][j] = min(dp[cur][j], dp[prev][j - 1]);
      else
        dp[cur][j] = min(dp[cur][j], dp[prev][j - 1] + 1);
    }
    cur ^= 1, prev ^= 1;
  }
  return dp[prev][m - 1];
}
```

Листинг 3.5 — Функция нерекурсивного поиска расстояния Дамерау—Левенштейна

```
int damerau_levenstein(const string &s1, const string &s2) {
  int n = static_cast<int>(s1.size()) + 1, m = static_cast<int>(s2.
     size()) + 1;
  vector <vector<int>> dp(3, vector<int>(m));
  for (int i = 1; i < m; ++i) {
    dp[0][i] = i;
  }
  int cur = 1, prev = 0, pprev = -1;
  for (int i = 1; i < n; ++i) {
    dp[cur][0] = i;
    for (int j = 1; j < m; ++j) {
      dp[cur][j] = min(dp[prev][j], dp[cur][j - 1]) + 1;
      if (s1[i - 1] == s2[j - 1])
        dp[cur][j] = min(dp[cur][j], dp[prev][j - 1]);
      else
        dp[cur][j] = min(dp[cur][j], dp[prev][j - 1] + 1);
      if (i > 1 && j > 1 && s1[i - 1] == s2[j - 2] && s1[i - 2] == s2
         [j - 1])
```

```
            dp[cur][j] = min(dp[cur][j], dp[pprev][j - 2] + 1);
        }
        cur = static_cast<int>((cur + 1) % dp.size()),
        prev = static_cast<int>((prev + 1) % dp.size()),
        pprev = static_cast<int>((pprev + 1) % dp.size());
    }
    return dp[prev][m - 1];
}
```

3.3 Тестирование

Для тестирования алгоритмов, тестовые случаи были разбиты на классы эквивалентности. В таблице 3.1 приведены тесты для алгоритма поиска расстояния Левенштейна, в таблице 3.2 — для поиска расстояния Дамерау-Левенштейна.

Таблица 3.1 — Тестовые случаи для алгоритма поиска расстояния Левенштейна

Классы эквивалентности	Входные данные	Ожидаемые выходные данные	Полученные выходные данные
Равные строки	«abracadabra» «abracadabra»	0	0
Пустая первая строка	«» «code»	4	4
Пустая вторая строка	«code» «»	4	4
Замена букв	«cap» «cat»	1	1
Добавление/удаление букв	«fake» «break»	4	4

Таблица 3.2 — Тестовые случаи для алгоритма поиска расстояния Дамерау — Левенштейна

Классы эквивалентности	Входные данные	Ожидаемые выходные данные	Полученные выходные данные
Равные строки	«abracadabra» «abracadabra»	0	0
Пустая первая строка	«» «code»	4	4
Пустая вторая строка	«code» «»	4	4
Замена букв	«cap» «cat»	1	1
Добавление/удаление букв	«fake» «break»	4	4
Транспозиция букв	«cpu» «cup»	1	1

Для проведения тестирования использовалась библиотека Google Tests [1]. Пример модульного тестирования в листинге 3.6.

Листинг 3.6 — Тестовый набор и запуск модульных тестов

```
TEST(DAM_LEV, equal_string) {
  total_tests++;
  string s1 = "abracadabra", s2 = "abracadabra";
  int corr_ans = 0, ans = damerau_levenstein(s1, s2);
  ASSERT_EQ(corr_ans, ans);
}

TEST(DAM_LEV, empty_string_1) {
  total_tests++;
  string s1 = "", s2 = "code";
  int corr_ans = 4, ans = damerau_levenstein(s1, s2);
  ASSERT_EQ(corr_ans, ans);
}

TEST(DAM_LEV, empty_string_2) {
  total_tests++;
  string s1 = "code", s2 = "";
  int corr_ans = 4, ans = damerau_levenstein(s1, s2);
  ASSERT_EQ(corr_ans, ans);
}

TEST(DAM_LEV, swapped_letter_strings)
{
  total_tests++;
  string s1 = "cap", s2 = "cat";
  int corr_ans = 1, ans = damerau_levenstein(s1, s2);
  ASSERT_EQ(corr_ans, ans);
}

TEST(DAM_LEV, complex_test_strings)
{
  total_tests++;
  string s1 = "fake", s2 = "break";
  int corr_ans = 4, ans = damerau_levenstein(s1, s2);
  ASSERT_EQ(corr_ans, ans);
}

TEST(DAM_LEV, transposition_letters_strings)
{
  total_tests++;
```

```
    string s1 = "cpu", s2 = "cup";
    int corr_ans = 1, ans = damerau_levenstein(s1, s2);
    ASSERT_EQ(corr_ans, ans);
}

int main(int argc, char **argv) {
    testing::InitGoogleTest(&argc, argv);
    int failed_cnt = RUN_ALL_TESTS();
    cout<<"TESTS_PASSED "<<total_tests-failed_cnt<<'\n'<<"TESTS_FAILED
        "<<failed_cnt<<'\n';
    return failed_cnt;
}
```

Вывод

Были реализованы алгоритмы поиска редакционных расстояний и проведено их тестирование. Все тесты были успешно пройдены.

4 Исследовательская часть

Замеры выполнялись на микроконтроллере STM32, со следующими характеристиками [3]:

— 32-битный процессор архитектуры ARM с частотой 256 МГц;

— оперативная память: 512 Кбайт.

4.1 Замеры процессорного времени

Для замеров использовались случайно сгенерированные строки длинной до 10 символов. Для каждой пары строк совпадающей длины замеры проводились 50 раз. Замеры проводились с помощью функции micros() [2]. В таблице 4.1 представлены результаты замеров.

Таблица 4.1 — Результаты замеров времени для различных реализаций алгоритмов поиска и длин входных строк в секундах для 50 запусков

Длины строк	Расстояние Левенштейна			Расстояние Дамерау—Левенштейна
	Рекурсивная реализация	Рекурсивная реализация с мемоизацией	Итеративная реализация	Итеративная реализация
1	0.000009	0.000033	0.000059	0.000086
2	0.000041	0.000059	0.000058	0.000074
3	0.000186	0.000180	0.000062	0.000088
4	0.000968	0.000729	0.000074	0.000098
5	0.005069	0.003389	0.000073	0.000105
6	0.028604	0.020497	0.000081	0.000083
7	0.133681	0.113024	0.000089	0.000098
8	0.696408	0.501769	0.000099	0.000108
9	3.987760	2.801120	0.000123	0.000119
10	23.923900	12.220100	0.000156	0.000123

На рисунке 4.1 представлены графики замеров времени для каждой из трех реализаций алгоритма поиска расстояния Левенштейна.

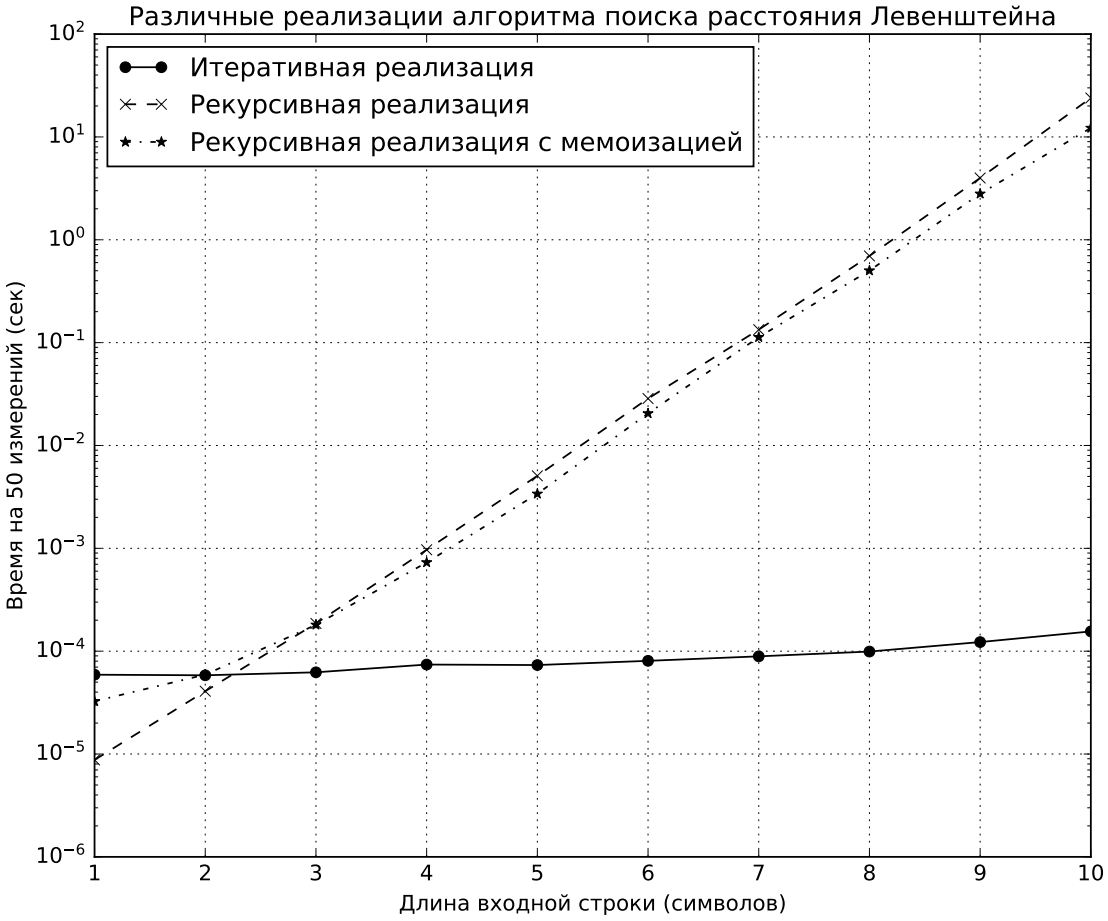

Рисунок 4.1 — Графики замеров времени для представленных реализаций поиска расстояния Левенштейна

На рисунке 4.2 представлены графики замеров времени для итеративных реализаций алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна.

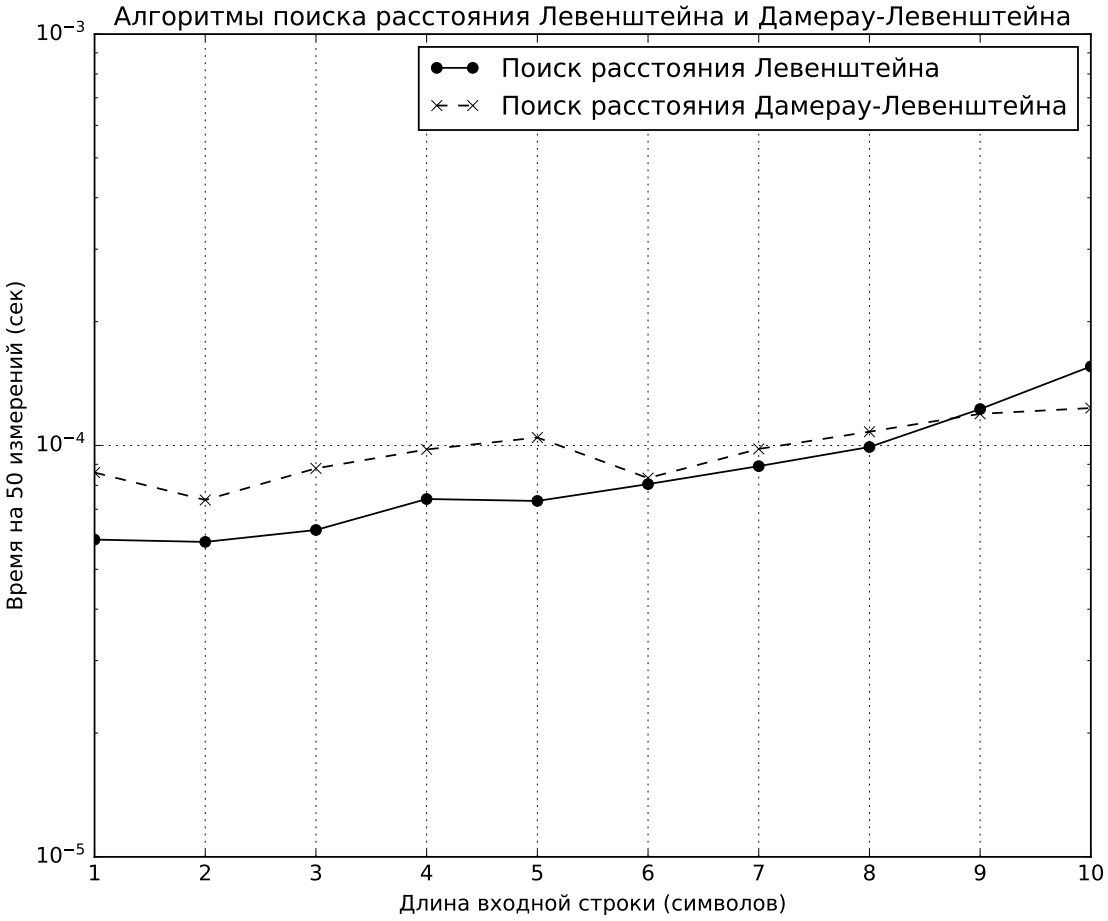

Рисунок 4.2 — Графики замеров времени для итеративных алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна

4.2 Оценка емкостных затрат

Для итеративных реализаций алгоритмов поиска редакционных расстояний оценка складывается из размера выделенной матрицы и некоторых дополнительных константных затрат:

— для алгоритма поиска расстояния Левенштейна

$$2 \cdot min(n, m) + 2 \cdot 4 + 2 \cdot 4 + 2 \cdot 4 = 2 \cdot min(n, m) + 24 \approx O(n) \tag{4.1}$$

где

- $2 \cdot min(n, m)$ — для хранения матрицы уже рассчитанных значений;
- $2 \cdot 4$ — для хранения размеров входных строк
- $2 \cdot 4$ — для хранения указателей в матрице значений
- $2 \cdot 4$ — для хранения счетчиков циклов

— для алгоритма поиска расстояния Дамерау—Левенштейна

$$3 \cdot min(n, m) + 2 \cdot 4 + 3 \cdot 4 + 2 \cdot 4 = 2 \cdot min(n, m) + 28 \approx O(n) \tag{4.2}$$

где слагаемые указаны аналогично предыдущему случаю

В рекурсивной реализации каждый вызов делает три рекурсивных вызова. Считая для оценки дерево полным, его высота $h = max(n, m)$. При этом, так как вызовы осуществляются последовательно, единовременно в стеке не более h вызовов. Каждый вызов принимает копии подстрок, то есть еще $2 \cdot max(n, m)$ памяти. Таким образом итоговая оценка $h \cdot 2 \cdot max(n, m) = 2 \cdot max(n, m)^2 = O(n^2)$.

4.3 Вывод

Рекурсивные реализации в среднем медленнее итеративных. Рекурсивная реализация алгоритма с мемоизацией быстрее, чем та же реализация без нее, так как избавляет от повторных вызовов для тех же входных значений.

Итеративная реализация алгоритма поиска расстояния Дамерау—Левенштейна медленнее, чем такая же для поиска расстояния Левенштейна в связи с необходимостью учета операции транспозиции и дополнительных связанных с этим проверок.

По памяти итеративные реализации эффективнее рекурсивных.

ЗАКЛЮЧЕНИЕ

Целью данной работы было исследование алгоритмов поиска расстояний Левенштейна и Дамерау—Левенштейна.

В ходе лабораторной работы были выполнены следующие задачи:

— описаны алгоритмы поиска расстояний Левенштейна и Дамерау — Левенштейна;

— имплементированы различные реализации алгоритмов поиска редакционных расстояний;

— выбраны инструменты для измерения процессорного времени;

— проведены замеры и анализ временных и емкостных затрат различных реализаций указанных алгоритмов. В ходе анализа было выявлено, что итеративные реализации в среднем быстрее рекурсивных и эффективнее их по памяти.

Поставленная цель исследования алгоритмов поиска редакционных расстояний была достигнута.

СПИСОК ИСПОЛЬЗОВАННЫХ ИСТОЧНИКОВ

1. Googletest user's guide. Режим доступа: `https://google.github.io/googletest/`. Дата обращения 8.10.2024.

2. micros(). Режим доступа: `https://reference.arduino.cc/reference/en/language/functions/time/micros/`. Дата обращения 10.10.2024.

3. Stm32f7 series. Режим доступа: `https://www.st.com/en/microcontrollers-microprocessors/stm32f7-series.html`. Дата обращения 10.10.2024.

4. Fred J. Damerau. A technique for computer detection and correction of spelling errors. *Commun. ACM*, 7(3):171–176, 1964.

5. Gonzalo Navarro. A guided tour to approximate string matching. *ACM Comput. Surv.*, 33(1):31–88, 2001.

6. В. И. Левенштейн. Двоичные коды с исправлением выпадений, вставок и замещений символов. *Доклады Академий Наук СССР*, 163(4):845–848, 1965.