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» (ИУ7)**

НАПРАВЛЕНИЕ ПОДГОТОВКИ **09.03.04 Программная инженерия**

ОТЧЕТ

по лабораторной работе № 1

Название: Расстояние Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-52Б _____ А.А. Паклин
(Группа) (Подпись, дата) (И.О. Фамилия)

Преподаватель _____ Л.Л. Волкова
(Подпись, дата) (И.О. Фамилия)

Москва, 2020

Оглавление

Введение

Расстояние Левенштейна - минимальное количество редакторских операций, которое необходимо для превращения одной строки в другую(или другими словами - расстояние между строками).

Расстояние Левенштейна используется в следующих сферах:

- поисковики - строка поиска Яндекс: автоисправление, автозамена;
- бионформатика - в белке каждая молекула представляется буквой из ограниченного алфавита, и белки представляются строками, которые можно сравнивать.

Целью данной лабораторной работы является изучение метода динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна.

Задачами данной лабораторной являются:

1. изучение алгоритмов Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. применение метода динамического программирования для матричной реализации указанных алгоритмов;
3. получение практических навыков реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментальное подтверждение различий во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. описание и обоснование полученных результатов в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

1. Аналитическая часть

Задача по нахождению расстояния Левенштейна заключается в поиске минимального количества операций вставки/удаления/замены для превращения одной строки в другую.

При нахождении расстояния Дамерау — Левенштейна добавляется операция транспозиции (перестановки соседних символов).

Действия обозначаются так:

1. D (delete) — удалить;
2. I (insert) — вставить;
3. R (replace) — заменить;
4. M(match) - совпадение.

Пусть S_1 и S_2 — две строки (длиной M и N соответственно) над некоторым алфавитом, тогда расстояние Левенштейна можно подсчитать по следующей рекуррентной формуле, описанной в книге [1]:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(S_1[i], S_2[j-1]) + 1, & j > 0 \quad //I \\ D(S_1[i-1], S_2[j]) + 1, & i > 0 \quad //D \\ D(S_1[i-1], S_2[j-1]) + & \\ \left[\begin{array}{ll} 0, & \text{если} \quad S1[i] == S2[j], \quad //M \\ 1, & \text{иначе} \quad //R \end{array} \right. & \\ ) & \end{cases}$$

где $min()$ возвращает наименьший из аргументов.

Расстояние Дамерау-Левенштейна вычисляется по следующей рекуррентной формуле, описанной в книге [1]:

$$D(i,j) = \begin{cases} 0, & i = 0, j = 0 \\ i, & j = 0, i > 0 \\ j, & i = 0, j > 0 \\ min(& \\ D(S_1[i], S_2[j-1]) + 1, & j > 0 \quad //I \\ D(S_1[i-1], S_2[j]) + 1, & i > 0 \quad //D \\ D(S_1[i-1], S_2[j-1]) + & \\ \left[\begin{array}{ll} 0, & \text{если} \quad S1[i] == S2[j], \quad //M \\ 1, & \text{иначе} \quad //R \end{array} \right. & \\ D(S_1[i-2], S_2[j-2]) + 1 & \text{если } i = 1, j = 1, S_1[i] = S_2[j-1], S_1[i-1] = S_2[j] \\ ) & \end{cases}$$

Вывод

Была рассмотрена теоретичсекая часть алгоритмов нахождения минимального редакционного расстояния: Левенштейна и Дамерау-Левенштейна. Отличие которых — наличие дополнительной операции во втором алгоритме.

2. Конструкторская часть

В данном разделе описываются схемы алгоритмов для нахождения минимального редакционного расстояния.

2.1 Разработка алгоритмов

На рисунках 1 - 4 представлены схемы алгоритмов, сделанные на основе теоретического материала, представленного в книге [2]

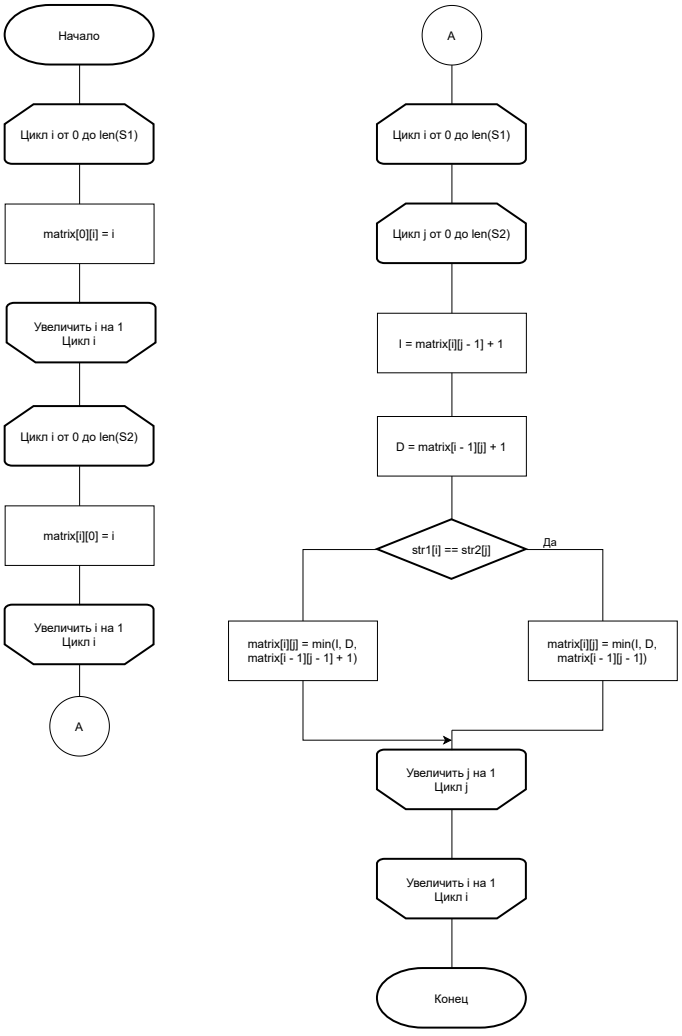


Рисунок 1: Схема матричного алгоритма Левенштейна

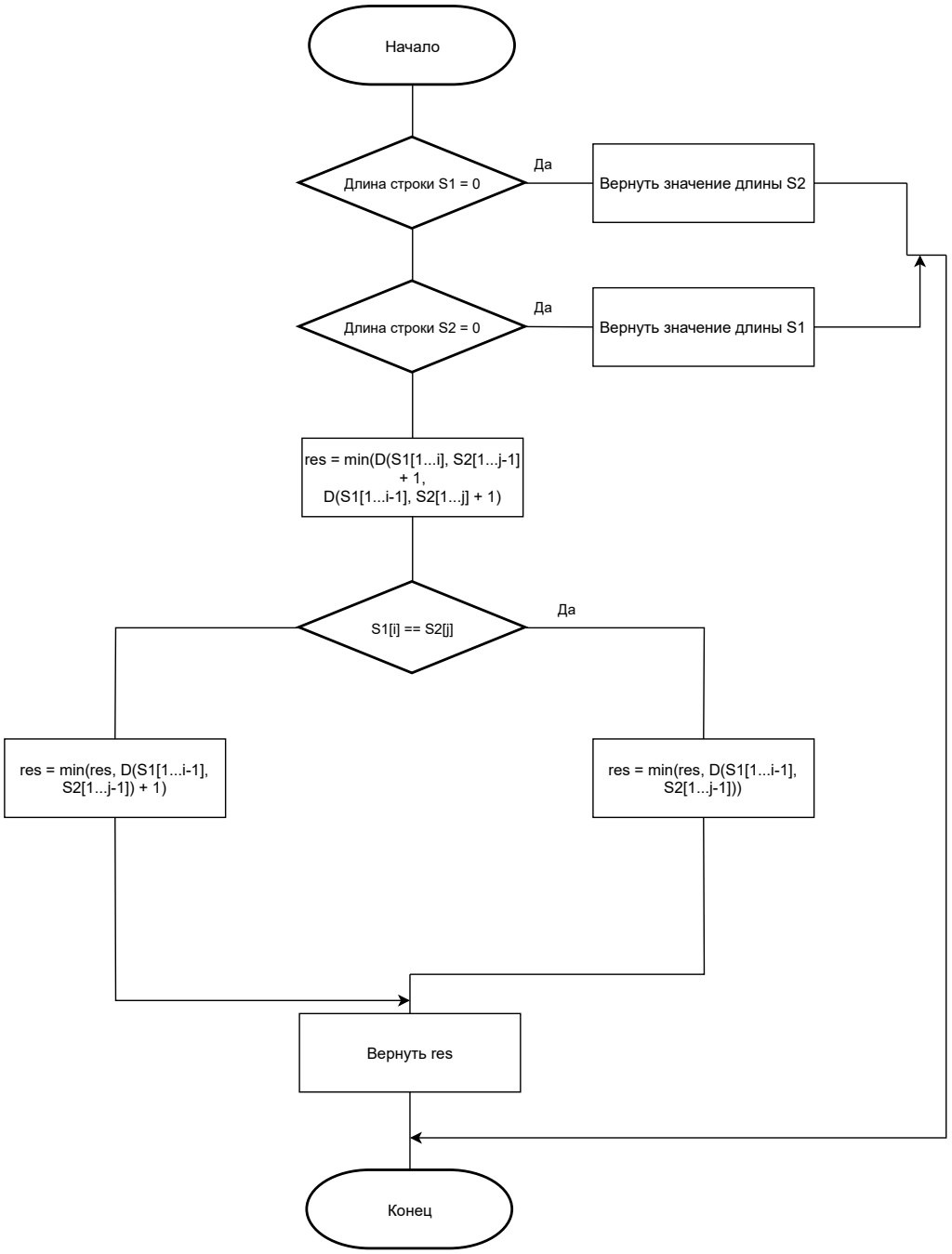


Рисунок 2: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

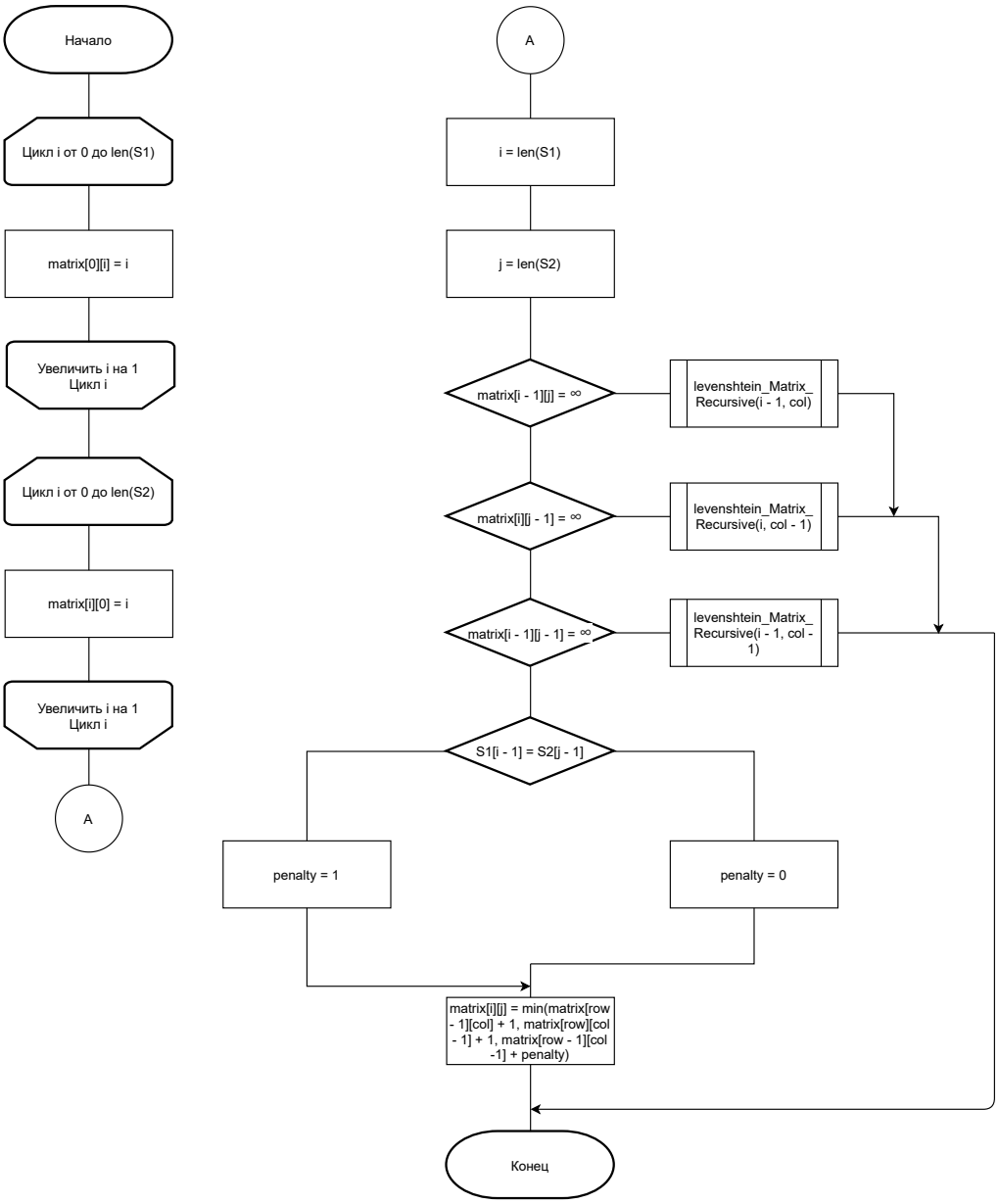


Рисунок 3: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с заполнением матрицы и дополнительными проверками

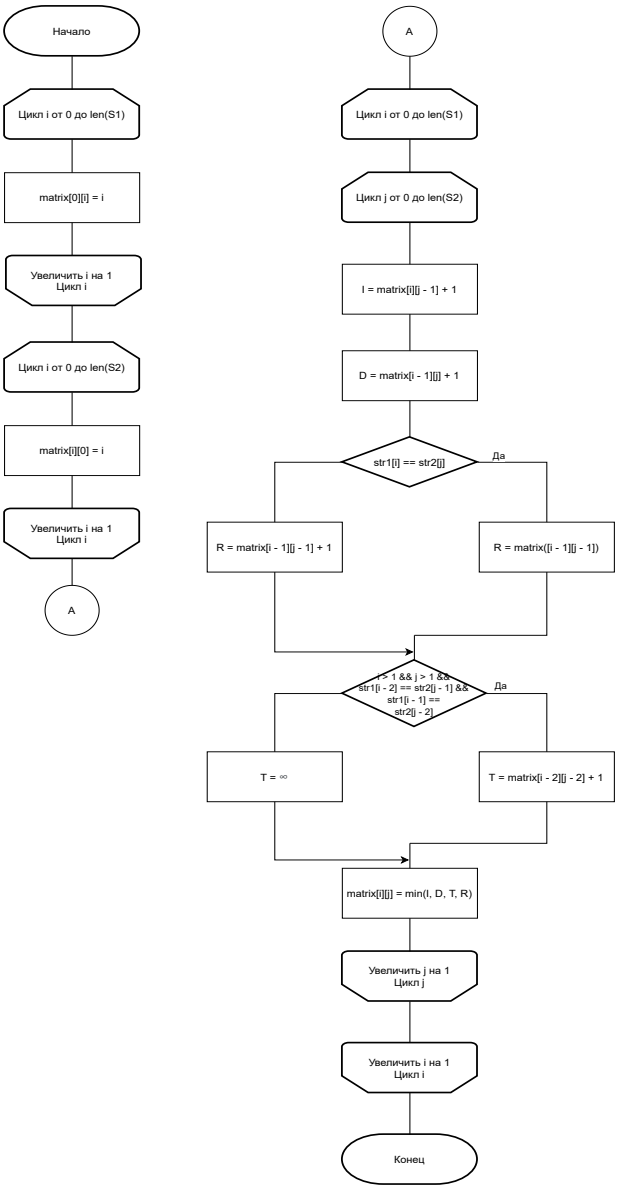


Рисунок 4: Схема матричного алгоритма нахождения расстояния Дамерау - Левенштейна

Вывод

В данном разделе были проиллюстрированы схемы алгоритмов нахождения минимального редакционного расстояния.

3. Технологическая часть

В данном разделе описываются средства реализации, рассматриваются требования к программному обеспечению, а также приводится реализация алгоритмов на языке C++.

3.1 Требования к программному обеспечению

Требования к вводу:

1. на вход подаются две строки;
2. буквы в верхнем и нижнем регистрах считаются разными.

Требования к программе: Две пустые строки - корректный ввод, программа не должна завершаться аварийно.

3.2 Средства реализации

Для выполнения лабораторной работы мною был выбран язык C++, так как он является одним из самых высокопроизводительных языков программирования. Для замера времени использовалась встроенная библиотека std::chrono[4].

3.3 Листинг кода алгоритмов

На листинге 3.1 представлена реализация стандартного алгоритма Левенштейна:

Листинг 3.1: Функция нахождения расстояния Левенштейна с использованием матрицы

```
Matrix<size_t> Levenstein(string_view first, string_view second) {
  auto rows = first.size() + 1;
  auto cols = second.size() + 1;
  Matrix<size_t> result(rows, cols);
  for (size_t i = 0; i < rows; i++) {
    for (size_t j = 0; j < cols; j++) {
      if (i * j == 0) {
        result[i][j] = i | j;
      } else if (first[i - 1] == second[j - 1]) {
        result[i][j] = result[i - 1][j - 1];
      } else {
        result[i][j] = 1 + u_min(
                                  result[i - 1][j - 1],
                                  result[i][j - 1],
                                  result[i - 1][j]
                                 );
      }
    }
  }
  return result;
}
```

На листинге 3.2 представлена реализация алгоритма Левенштейна с использованием рекурсии:

Листинг 3.2: Функция нахождения расстояния Левенштейна с использованием рекурсии

```
size_t RecursiveImpl(string_view first, string_view second,
              const size_t i, const size_t j)
{
  if (i * j == 0) {
    return i | j;
  } else if (first[i - 1] == second[j - 1]) {
    return RecursiveImpl(first, second, i - 1, j - 1);
  }
  return 1 + u_min(
                    RecursiveImpl(first, second, i, j - 1),
                    RecursiveImpl(first, second, i - 1, j),
                    RecursiveImpl(first, second, i - 1, j - 1)
                  );
}

size_t RecursiveLevenstein(string_view first, string_view secnd) {
  return RecursiveImpl(first, secnd, first.size() + 1, secnd.size() + 1);
}
```

На листинге 3.3 представлена реализация алгоритма Левенштейна с использованием рекурсии и заполнением матрицы:

Листинг 3.3: Функция нахождения расстояния Левенштейна с использованием рекурсии и заполнением матрицы

```
size_t RecursiveMatrImpl(Matrix<size_t>& matrix, string_view first,
          string_view second, const size_t i, const size_t j)
{
  if (matrix[i][j] != -1) {
    return matrix[i][j];
  }
  if (i * j == 0) {
    return matrix[i][j] = i | j;
  }
  auto cost = (first[i - 1] != second[j - 1]);
  return matrix[i][j] = cost + u_min(
                  RecursiveMatrImpl(matrix, first, second, i, j - 1),
                  RecursiveMatrImpl(matrix, first, second, i - 1, j),
                  RecursiveMatrImpl(matrix, first, second, i - 1, j - 1)
                                    );
}

Matrix<size_t> RecursiveMatrLevenstein(string_view first,
                                       string_view second)
{
  Matrix<size_t> result(first.size() + 1, second.size() + 1);
  result.assign(-1);
  RecursiveMatrImpl(result, first, second, first.size(), second.size());
  return result;
}
```

На листинге 3.4 представлена реализация стандартного алгоритма Дамерау-Левенштейна:

Листинг 3.4: Функция нахождения расстояния Дамерау-Левенштейна с использованием матрицы

```
Matrix<size_t> DamerauLevenstein(string_view first, string_view second) {
  auto rows = first.size() + 1;
  auto cols = second.size() + 1;
  Matrix<size_t> result(rows, cols);

  for (size_t i = 0; i < rows; i++) {
    for (size_t j = 0; j < cols; j++) {
      if (i * j == 0) {
        result[i][j] = i | j;
      } else if (first[i - 1] == second[j - 1]) {
        result[i][j] = result[i - 1][j - 1];
      } else {
        result[i][j] = 1 + u_min(
                                  result[i - 1][j - 1],
                                  result[i][j - 1],
                                  result[i - 1][j]
                                );
      }
      if (i > 1 && j > 1 && first[i - 1] == second[j - 2]
            && first[i - 2] == second[j - 1])
      {
          result[i][j] = u_min(
                                result[i][j],
                                result[i - 2][j - 2] + 1
                              );
      }
    }
  }
  return result;
}
```

Вывод

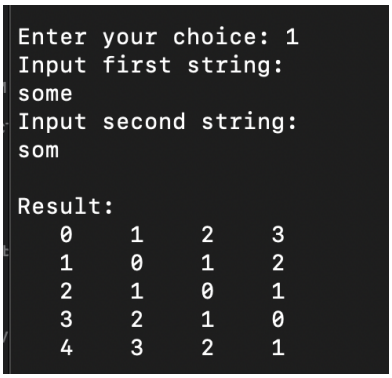
В данном разделе была приведены реализация алгоритмов поиска минимального редакционного расстояния на языке программирования C++.

4. Экспериментальная часть

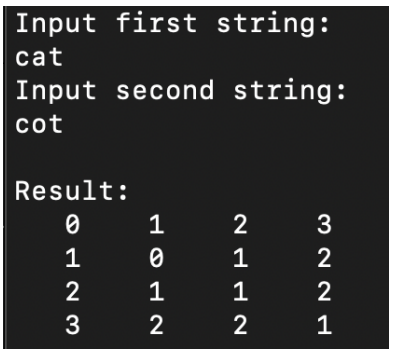
В данном разделе иллюстрируются примеры работы программы, результаты тестирования, а также описывается постановка эксперимента по замеру времени и памяти.

4.1 Тестирование программы

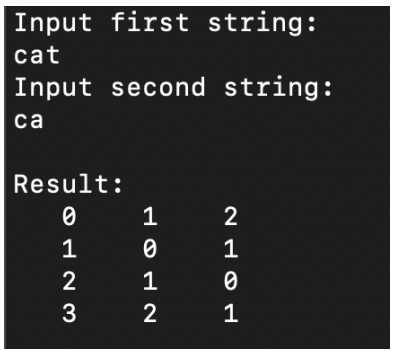
Примеры работы для каждой из операций:



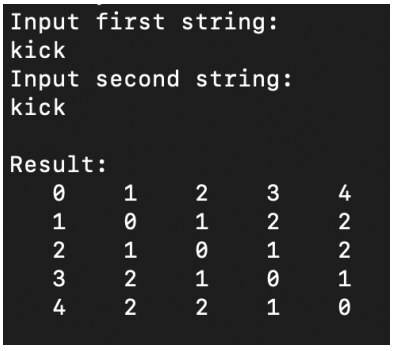
(1) удаление символов из строки



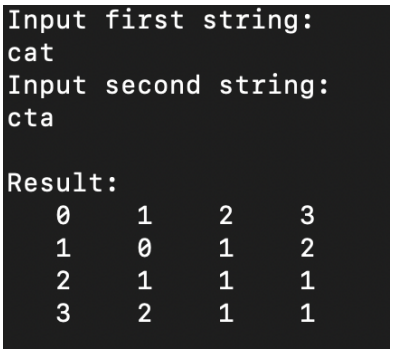
(2) замена



(3) добавление



(4) ввод одинаковых строк



(5) перестановка

Рисунок 5: Результаты работы программы

Также было проведено тестирование методом черного ящика, результаты показаны в таблице 4.2:

№	Первое слово	Второе слово	Ожидаемый результат(Левенштейн, Дамерау-Левенштейн)	Полученный результат
1	kot	skat	2 2	2 2
2	kate	ktae	2 1	2 1
3	abacaba	aabcaab	4 2	4 2
4	sobaka	sboku	3 3	3 3
5	qwerty	queue	4 4	4 4
6	apple	aplpe	2 1	2 1
7	bmstu	utsmb	4 4	4 4

Таблица 4.2: Таблица результатов тестов

4.2 Постановка эксперимента по замеру времени и памяти

В рамках данной лабораторной работы были проведены слудующие эксперименты:

1. теоретический замер потребляемой памяти всеми четырьмя алгоритмами на строках длины 10;
2. сравнение скорости работы матричных алгоритмов Левенштейна и Дамерау-Левенштейна на строках длиной от 1000 до 2000 включительно. Один эксперимент проводился 50 раз;
3. сравнение скорости работы рекурсивного алгоритма Левенштейна и рекурсивного алгоритма Левнештейна с матрицей на строках длиной от 1 до 12 включительно. Один эксперимент ставился 5 раз;
4. сравнение трех алгоритмов Левенштейна на строках длиной от 1 до 12 включительно. Один эксперимен проводился 5 раз.

4.3 Сравнительный анализ на материале экспериментальных данных

В таблице 4.3 представлен список основных сущностей, используемых в алгоритмах, и количество памяти, которое занимает каждая из них.

Сущность	Описание	Размер(64 битная архитектура) в байтах
Целое число	Машинно зависимая величина.	4
Массив рун	Руна - это псевдоним для целого беззнакового 32 битного числа	len(arr) * 4
Вектор / Строка	Данная сущность по сути представляет собой динамический массив.	24
Ссылка	Объект, указывающий на определенные данные, но не хранящий их.	4

Таблица 4.3: Память, занимаемая различными сущностями в C++

В таблице 4.4 видно, что потребляемая память у рекурсивных алгоритмов зависит от максимальной глубины рекурсии, которая равна максимальной длине из двух слов.

Алгоритм	Используемые сущности	Память, занимаемая каждой сущностью(в байтах)	Потребляемая память для всего алгоритма(64 битная архитектура)
Матричный Левенштейн	Две строки	len(s1) * 2 + len(s2) * 2 + 48	2 * (len(s1) + len(s2)) + 88 + (len(s1) + 1)*(24 + (len(s2) + 1)* 4)
	Вектор векторов для представления матрицы	(len(s1) + 1)*(24 + (len(s2) + 1)* 4) + 24	
	Четыре переменные типа size_t для итерации в цикле и хранения длины строк	16	
Рекурсивный Левенштейн	Две строки	len(s1) * 2 + len(s2) * 2 + 48	(2 * (len(s1) + len(s2)) + 48 + 68 * (макс.уровень рекурсии)
	Две переменные типа size_t для индексации строк	8	
	Указатель для адреса возврата	8	
	Переменная типа size_t для ответа	4	

Рекурсивный Левенштейн с Матрицей	Две строки	len(s1) * 2 + len(s2) * 2 + 48	(2 *(len(s1) + len(s2)) + 68 * (макс.уровень рекурсии) + (len(s1) + 1)*(24 + (len(s2) + 1)* 4) + 72
	Вектор векторов для представления матрицы	(len(s1) + 1)*(24 + (len(s2) + 1)* 4) + 24	
	Две переменные типа size_t для индексации строк	8	
	Указатель для адреса возврата	8	
	Переменная типа size_t для ответа	4	
Матричный Дамерау-Левенштейн	Две строки	len(s1) * 2 + len(s2) * 2 + 48	2 * (len(s1) + len(s2)) + 88 + (len(s1) + 1)*(24 + (len(s2) + 1)* 4)
	Вектор векторов для представления матрицы	(len(s1) + 1)*(24 + (len(s2) + 1)* 4) + 24	
	Четыре переменные типа size_t для итерации в цикле и хранения длины строк	16	

Таблица 4.4: Теоретическая оценка памяти, потребляемой алгоритмами

В таблице 4.5 видно, что рекурсивный алгоритм Левенштейна наименее затратный по памяти. Самым затратным же алгоритмом является рекурсивный Левенштейн с заполнением матрицы, так как помимо потворного вызова рекурсивной функции, в самом начале также аллоцируется память под матрицу, которая в итоге и заполняется.

Алгоритм	Теоретическое потребление памяти при len(s1) = len(s2) = 10
Матричный Левенштейн	876
Рекурсивный Левенштейн	768
Рекурсивный Левенштейн с матрицей	1540
Матричный Дамерау Левенштейн	876

Таблица 4.5: Теоретическое потребление памяти алгоритмами при строках длины 10

Взглянув на рисунок 6 можно сделать вывод, что матричный алгоритм Дамерау-Левенштейна работает медленнее матричного алгоритма Левенштейна, в силу того, что

в Дамерау на каждой итерации цикла нам нужно дополнительно проверять на истинность еще одно условие и в случае прохождения проверки производить дополнительные вычисления.

В силу того, что рекурсивный алгоритм работает достаточно медленно, график скорости работы двух рекурсивных алгоритмов был разбит на 2 рисунка. Сравнив рисунки 7 и 8 можно сделать вывод, что несмотря на то, что рекурсивный алгоритм Левенштейна с заполнением матрицы требует больше памяти, чем рекурсивный алгоритм без нее, благодаря использованию матрицы он значительно выигрываем в скорости, так как в ходе выполнения не происходит лишних вызовов на повторное вычисление значений, вместо этого они сразу берутся из матрицы.

Взглянув на рисунок 6 можно сделать вывод, что матричный алгоритм Дамерау-Левенштейна работает медленнее матричного алгоритма Левенштейна, в силу того, что в Дамерау на каждой итерации цикла нам нужно дополнительно проверять на истинность еще одно условие и в случае прохождения проверки производить дополнительные вычисления.

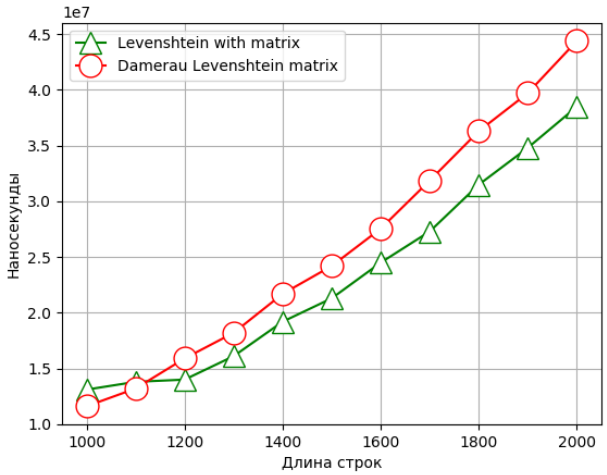


Рисунок 6: График работы матричных алгоритмов Левенштейна и Дамерау Левенштейна

В силу того, что рекурсивный алгоритм работает достаточно медленно, график скорости работы двух рекурсивных алгоритмов был разбит на 2 рисунка. Сравнив рисунки 7 и 8 можно сделать вывод, что несмотря на то, что рекурсивный алгоритм Левенштейна с заполнением матрицы требует больше памяти, чем рекурсивный алгоритм без нее, благодаря использованию матрицы он значительно выигрываем в скорости, так как в ходе выполнения не происходит лишних вызовов на повторное вычисление значений, вместо этого они сразу берутся из матрицы.

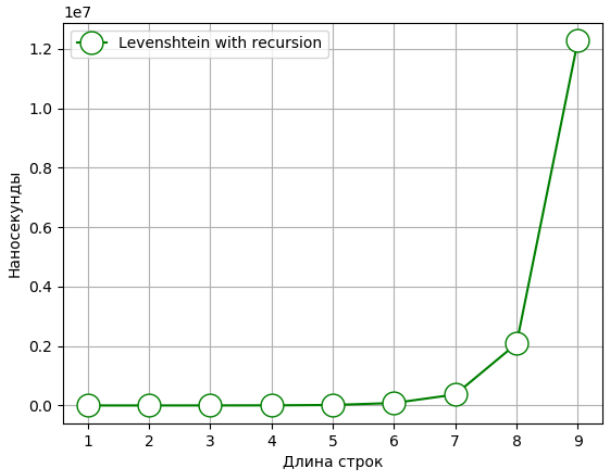


Рисунок 7: График работы рекурсивного алгоритма Левенштейна

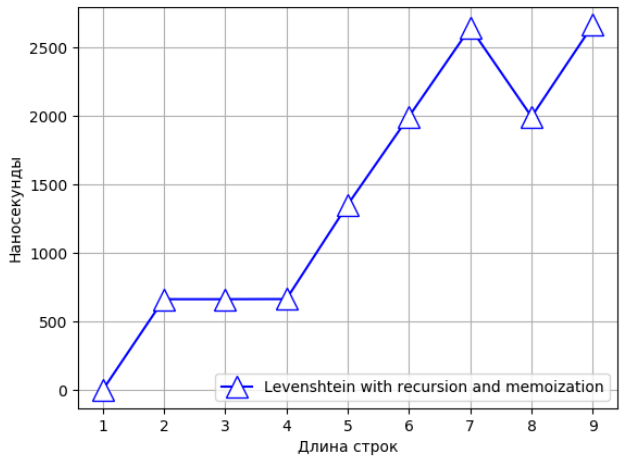


Рисунок 8: График работы рекурсивного алгоритма с заполнением матрицы Левенштейна

Сравнив рисунок 8 и рисунок 9 можно сделать вывод, что из трех алгоритмов Левенштейна самым медленным оказывается рекурсивный алгоритм, причем при увеличении длины слов появляется проблема того, что результат не может быть получен за приемлимое время. Однако как видно на рисунке 9 добавление матрицы существенно сокращает время выполнения.

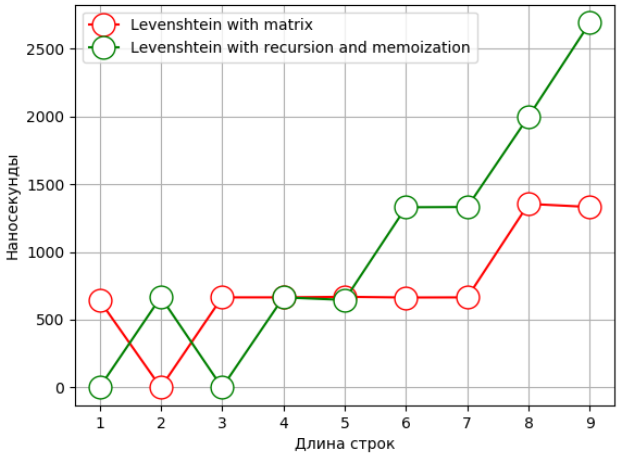


Рисунок 9: График работы алгоритмов Левенштейна

Вывод

В данном разделе были продемонстрированы результаты экспериментов и сделаны выводы в соответствии с анализом их результатов. Было установлено, что рекурсивный алгоритм работает медленнее остальных, но требует меньше памяти. Самым быстрым алгоритмом оказался матричный Левенштейн.

Заключение

В ходе выполнения лабораторной работы были выполнены следующие задачи:

1. изучил алгоритмы Левенштейна и Дамерау-Левенштейна нахождения расстояния между строками;
2. примененил метод динамического программирования для матричной реализации указанных алгоритмов;
3. получил практические навыкы реализации указанных алгоритмов: двух алгоритмов в матричной версии и одного из алгоритмов в рекурсивной версии;
4. провел сравнительный анализ линейной и рекурсивной реализаций выбранного алгоритма определения расстояния между строками по затрачиваемым ресурсам (времени и памяти);
5. экспериментально подтвердил различия во временно́й эффективности рекурсивной и нерекурсивной реализаций выбранного алгоритма определения расстояния между строками при помощи разработанного программного обеспечения на материале замеров процессорного времени выполнения реализации на варьирующихся длинах строк;
6. описал и обосновал полученные результаты в отчете о выполненной лабораторной работе, выполненного как расчётно-пояснительная записка к работе.

Список использованной литературы

[1] Скиена Стивен. Алгоритмы.Руководство к разработке. – М.:БХВ–Петербруг, 2018. – 720 с.

[2] Дэн Гасфилд. Строки, деревья и последовательности в алгоритмах. – М.:БХВ–Петербруг, 2003. – 654 с.

[3] Справка по C++ // cppreference URL: https://en.cppreference.com/w/ (дата обращения: 06.10.2020).

[4] Библиотека для замера времени std::chrono // cppreference URL: https://en.cppreference.com/w/cpp/header/chrono (дата обращения: 06.10.2020)