Государственное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н. Э. Баумана»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ОТЧЁТ ПО АНАЛИЗУ АЛГОРИТМОВ

к лабораторной работе №1 на тему:

Расстояние Левенштейна. Расстояние Дамерау-Левенштейна

Студент: Квасников А.В. ИУ7-56

Москва 2018

Оглавление

Введение

В лабораторной работе изучаются расстояние Левенштейна и расстояние Дамерау-Левенштейна. Задачи для лабораторной работы.

1. Изучение алгоритмов нахождения расстояний Левенштейна и Дамерау-Левенштейна между строками.

2. Применение метода динамического программирования для матричной реализации указанных алгоритмов

3. Получение практических навыков реализации указанных алгоритмов

4. Сравнительный анализ линейной и рекурсивной реализации выбранного алгоритма

5. Описание и обоснование полученных результатов в отчете о выполненной лабораторной работе.

1. Аналитическая часть

В данном разделе представлено описание расстояния Левенштейна и расстояния Дамерау-Левенштейна.

1.1 Расстояние Левенштейна

Расстояние Левенштейна (также редакционное расстояние) — это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для превращения одной строки в другую.

Расстояние Левенштейна применяется для исправления ошибок в слове поискового запроса (поисковые системы, поиск по базе данных, распознавание рукописного текста и устной речи).

Расстояние Левенштейна использует понятие цена операции. Операций всего 3: замена, вставка, удаление. В общем случае:

- *w(a, b) — цена замены символа a на символ b;*
- *w(ε, b) — цена вставки символа b;*
- *w(a, ε) — цена удаления символа a.*

Необходимо найти последовательность замен, минимизирующую суммарную цену.

- *w(a, a) = 0;*
- *w(a, b) = 1 при a≠b;*
- *w(ε, b) = 1;*
- *w(a, ε) = 1.*

1.2 Расстояние Дамерау-Левенштейна

Расстояние Дамерау-Левенштейна — это мера разницы двух строк символов, определяемая как минимальное количество операций вставки, удаления, замены и транспозиции (перестановки двух соседних символов), необходимых для перевода одной строки в другую. Является модификацией расстояния Левенштейна: к операциям вставки, удаления и замены символов, определенных в расстоянии Левенштейна, добавлена операция транспозиции (перестановки) символов. Т.е. если перестановка возможна и необходима, символы строки меняются местами. Эта модификация основывается на том, что люди часто делают опечатки в словах, меняя позиции соседних букв, и для того, чтобы исправить их, достаточно лишь поменять такие символы местами. Для таких случаев редакционное расстояние, определенное алгоритмом Дамерау-Левенштейна, будет меньше, чем при использовании классического алгоритма Левенштейна.

2. Конструкторская часть

В данном разделе представлено математическое описание алгоритмов, а также их блок-схемы

2.1 Математическое описание

Пусть имеются две строки $s1$ и $s2$ длиной m и n. Тогда расстояние Левенштейна ($D(s1, s2)$) можно подсчитать по рекуррентной формуле:

$$D(s1[1...m], s2[1...n]) = min(D(s1[1...m-1], s2[1...n]) + 1,$$
$$D(s1[1...m], s2[1...n-1]) + 1,$$
$$D(s1[1...m-1], s2[1...n-1]) + \alpha),$$

где $\alpha = 0$, если $s1[m] = s2[n]$, иначе $\alpha = 1$

Классификация разрешенных операций и штрафы на выполнение операции:

а) Замена символа - $R = 1$;

б) Вставка символа - $I = 1$;

в) Удаление символа - $D = 1$;

г) Совпадение символа - $M = 0$.

В алгоритме Дамерау-Левенштейна добавляется еще одна операция - транспозиция символа $T = 1$, а в рекуррентную формулу добавляется еще один член:

$$D(s1[1...m-1],s2[1...n-1]+\beta,$$

где $\beta = 1$,если $s1[m-1] = s2[n]$ и s2[n-1] = s1[m], иначе $\beta = 0$

2.2 Схемы алгоритмов

Матричная реализация алгоритма определения расстояния Левенштейна описана блок-схемой на рис. 2.

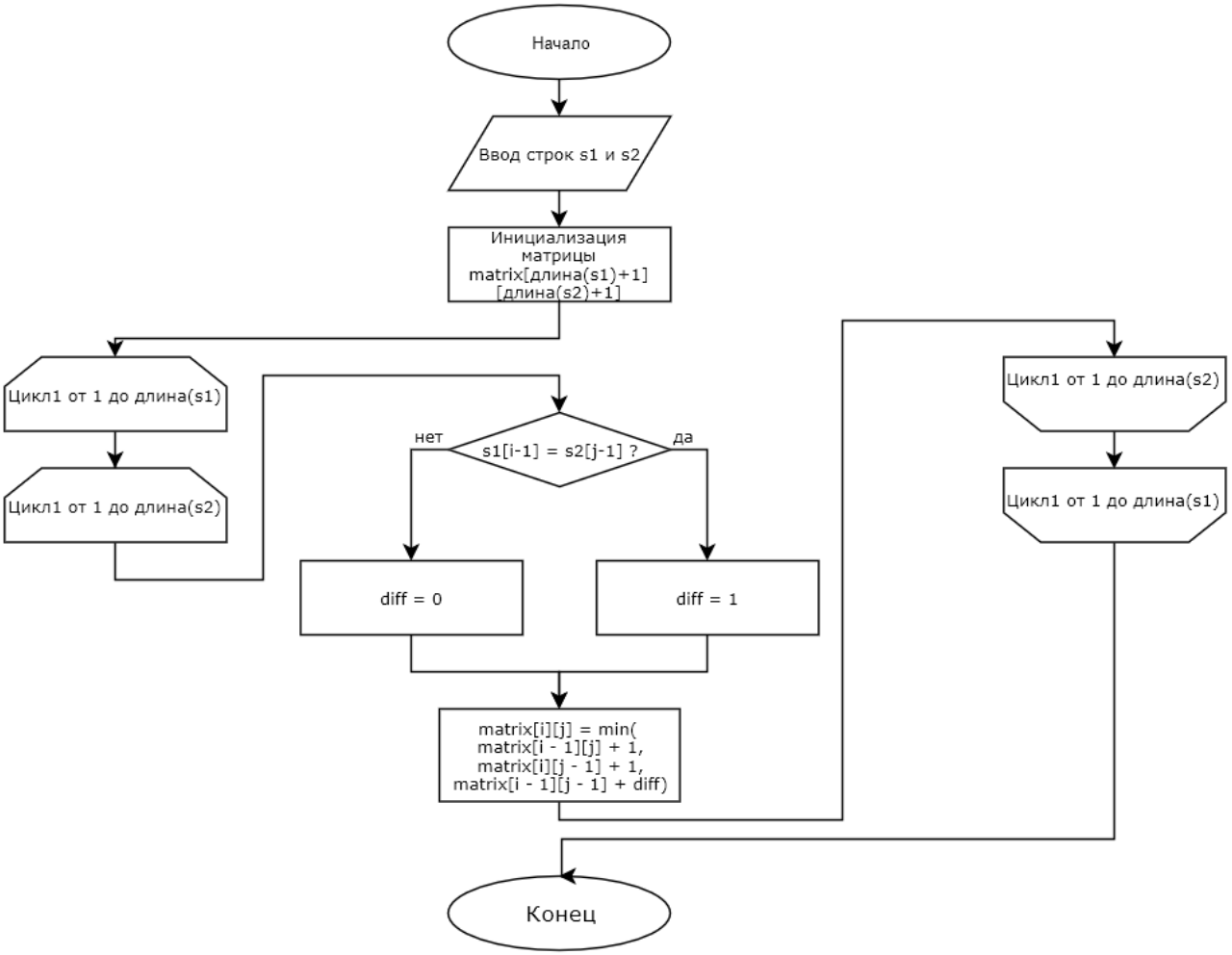


Рис. 2 – Схема алгоритма определения расстояния Левенштейна в матричной реализации

Рекурсивная реализация алгоритма определения расстояния Левенштейна

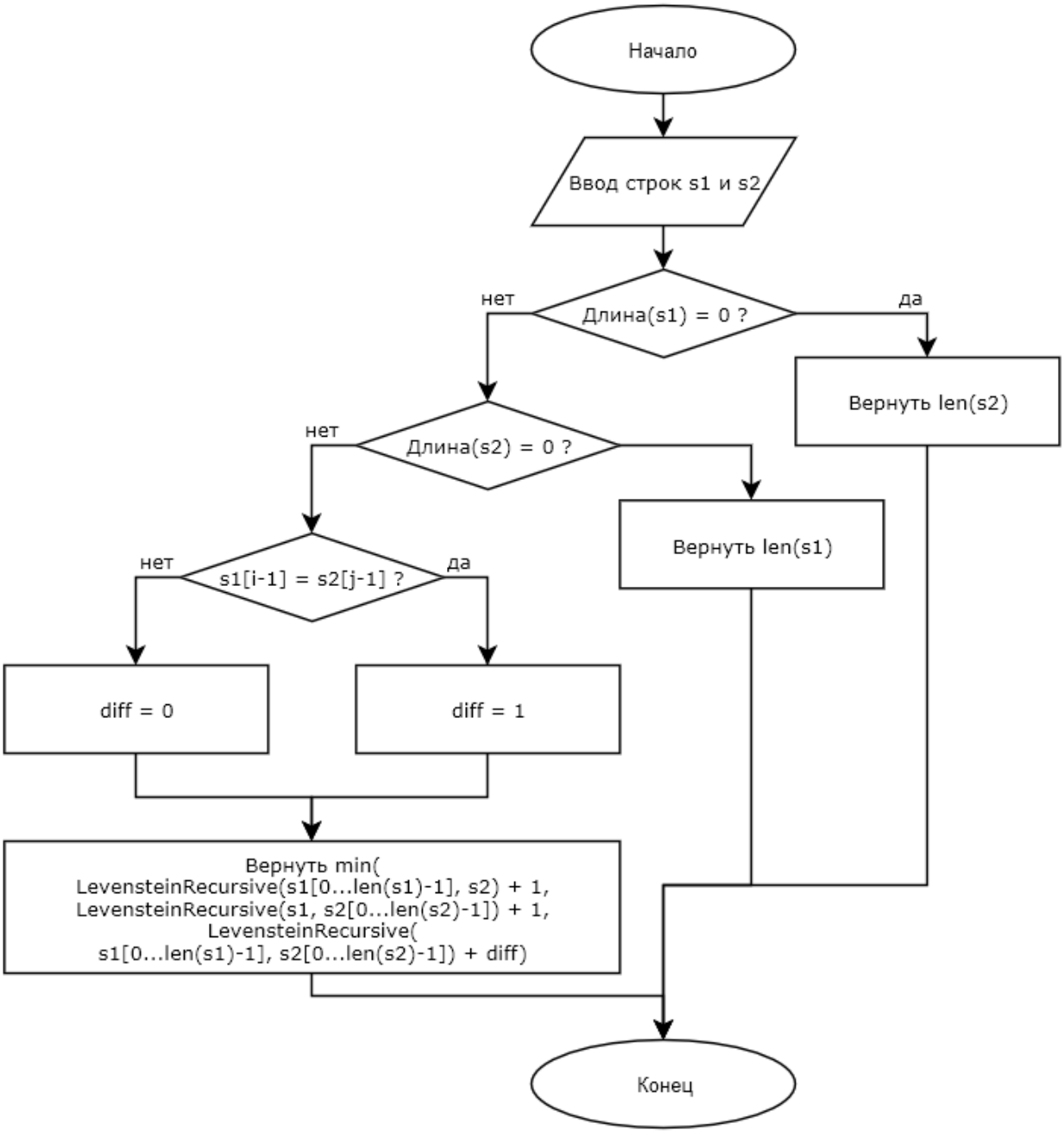


Рис. 3 – Схема алгоритма определения расстояния Левенштейна в рекурсивной реализации

Матричная реализация алгоритма определения расстояния Дамерау-Левенштейна

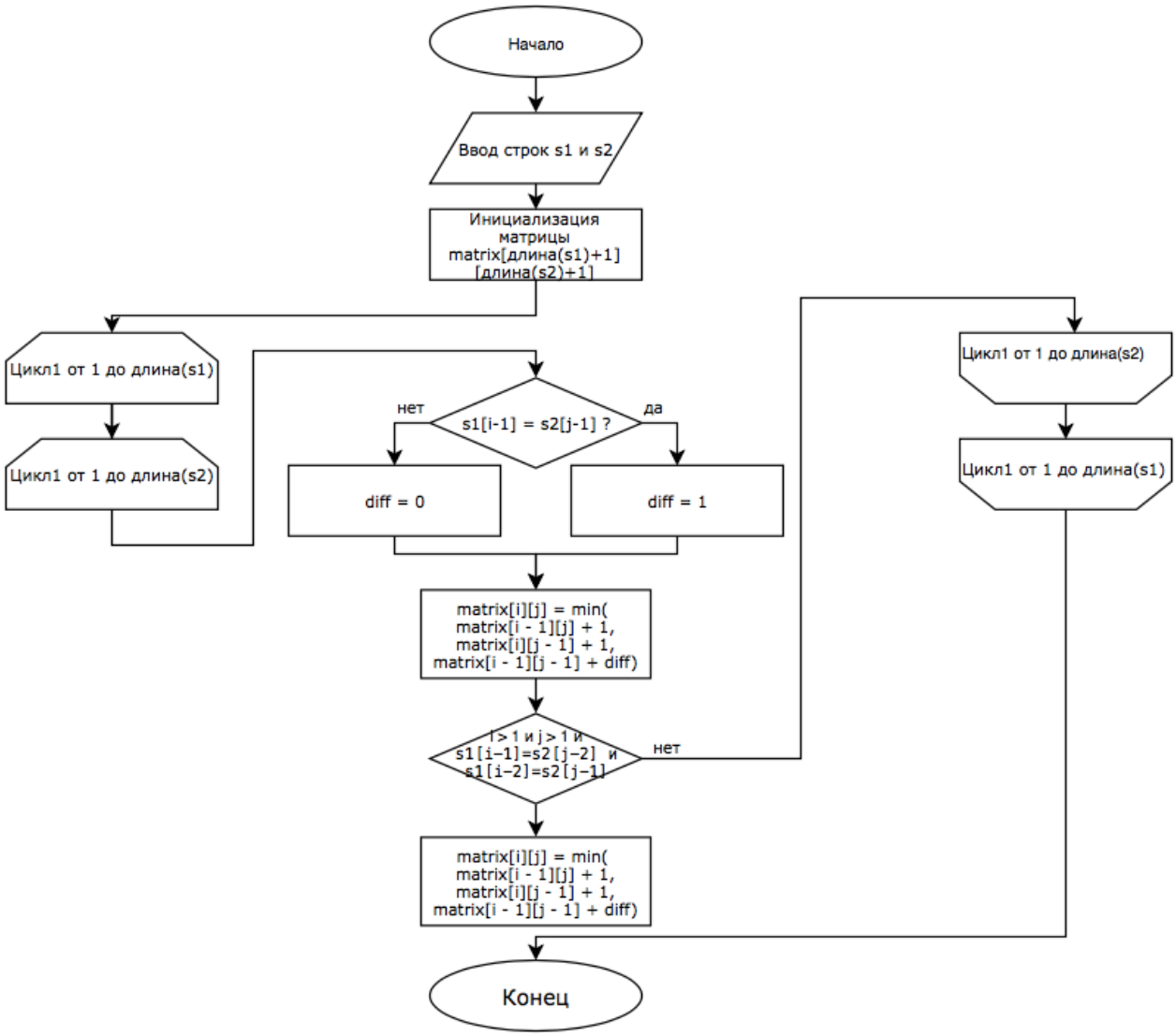


Рис. 4 – Схема алгоритма определения расстояния Дамерау-Левенштейна в матричной реализации

2.3 Сравнительный анализ рекурсивной и не рекурсивной реализации

Рекурсивная реализация затрачивает гораздо больше времени, чем не рекурсивная. Но, в некоторых случаях, она требует меньше памяти.

3. Технологическая часть

В данном разделе описаны требования к программному обеспечению, а также приведен листинг кода.

3.1 Требования к программному обеспечению

Программа для данной лабораторной работы разрабатывалась на языке C++ в среде XCode, поддерживаемой операционной системой Mac OS. Для запуска программы необходима среда XCode.

3.2 Листинг кода

Нерекурсивная (матричная) реализация алгоритма Левенштейна.

```
int levenshtein_distance(std::string s1, std::string s2) {
    int sizeS1 = s1.length();
    int sizeS2 = s2.length();
    if (sizeS1 == 0) {
        return sizeS2;
    }
    if (sizeS2 == 0) {
        return sizeS1;
    }

    std::vector< std::vector<int>>matrix(sizeS1 + 1);
    for (int i = 0; i <= sizeS1; i++) {
        matrix[i].resize(sizeS2 + 1);
        matrix[i][0] = i;
    }
    for (int i = 0; i <= sizeS2; i++) {
        matrix[0][i] = i;
    }
    int turn = 0;
    for (int i = 1; i <= sizeS1; i++) {
        for(int j = 1; j <= sizeS2; j++) {
            turn = s1[i - 1] == s2[j - 1] ? 0 : 1;
            matrix[i][j] = std::min(std::min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1), matrix[i - 1][j - 1] +
turn);
        }
    }
    return matrix[sizeS1][sizeS2];
}
```

Рекурсивная реализация алгоритма Левенштейна.

```
int levenshtein_recurs(std::string s1, std::string s2)
{
   int sizeS1 = s1.length();
   int sizeS2 = s2.length();
   int cost = 0;
   if (sizeS1 == 0) {
      return sizeS2;
   }
   if (sizeS2 == 0) {
      return sizeS1;
   }

   cost = s1[0] == s2[0] ? 0 : 1;

   int result = std::min(std::min(levenshtein_recurs(s1.substr(1, sizeS1), s2) + 1,
levenshtein_recurs(s1, s2.substr(1, sizeS2)) + 1), levenshtein_recurs(s1.substr(1, sizeS1),
s2.substr(1, sizeS2)) + cost);

   return result;
}
```

Нерекурсивная (матричная) реализация алгоритма Дамерау-Левенштейна.

```
int damerau_levenshtein(std::string s1, std::string s2)
{
   int sizeS1 = s1.length();
   int sizeS2 = s2.length();
   if (sizeS1 == 0) {
      return sizeS2;
   }
   if (sizeS2 == 0) {
      return sizeS1;
   }

   std::vector< std::vector<int>>matrix(sizeS1 + 1);
   for (int i = 0; i <= sizeS1; i++) {
      matrix[i].resize(sizeS2 + 1);
      matrix[i][0] = i;
   }
   for (int i = 0; i <= sizeS2; i++) {
      matrix[0][i] = i;
   }
   int turn = 0;
   for (int i = 1; i <= sizeS1; i++) {
      for(int j = 1; j <= sizeS2; j++) {
         turn = s1[i - 1] == s2[j - 1] ? 0 : 1;
         if ((i > 1 && j > 1) && (s1[i-1] == s2[j-2] && s1[i-2] == s2[j-1]) && (turn == 1))
            matrix[i][j] = std::min(std::min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1), std::min(matrix[i - 1][j - 1]
+ turn, matrix[i - 2][j - 2] + 1));
         else
            matrix[i][j] = std::min(std::min(matrix[i - 1][j] + 1, matrix[i][j - 1] + 1), matrix[i - 1][j - 1] + turn);
      }
   }

   return matrix[sizeS1][sizeS2];
}
```

4. Экспериментальная часть

4.1 Пример работы

Примеры работы программы с введенными данными (Слово 1 и Слово 2). В качестве вывода приведены результаты работы 3 реализованных алгоритмов.

Таблица 1.1 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
doors	**odors**	2 2 1	2 2 1	+
doors	**doors**	0 0 0	0 0 0	+
doors	**door**	1 1 1	1 1 1	+
doors	**norway**	5 5 5	5 5 5	+

Таблица 1.2 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
abracadabra	**arbadacarba**	6 6 4	6 6 4	+
abracadabra	**abracadabra**	0 0 0	0 0 0	+
abracadabra	**arbitrary**	7 7 7	7 7 7	+
abracadabra	**barack**	8 8 7	8 8 7	+

Таблица 1.3 — Данные тестов

Слово 1	Слово 2	Вывод	Ожидаемый вывод	Результат
decade	**facade**	2 2 2	2 2 2	+
decade	**decade**	0 0 0	0 0 0	+
decade	**decaed**	2 2 1	2 2 1	+
decade	**deacon**	4 4 3	4 4 3	+

4.2 Постановка эксперимента

Алгоритмы были протестированы по скорости работы.

Замеры времени отражены на рисунке 1.

Память - на рисунке 2.

Рисунок 1 — Время работы программы в зависимости от длины входных слов, меньше — лучше

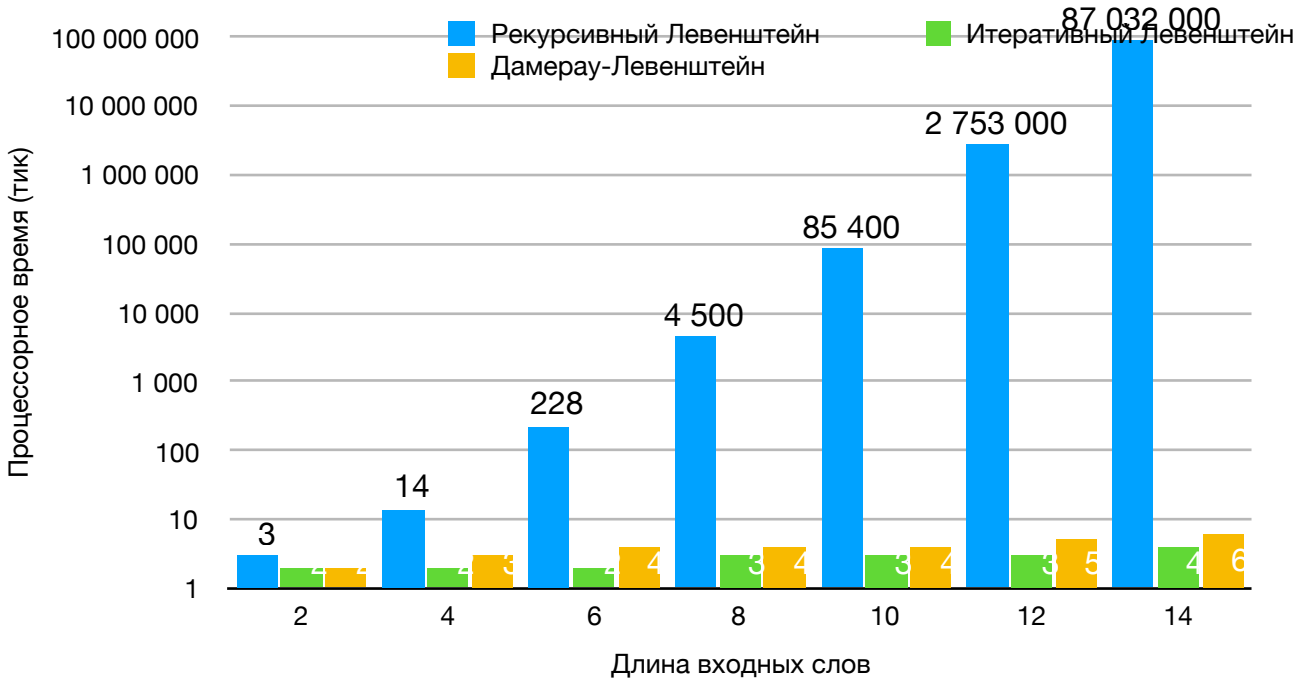
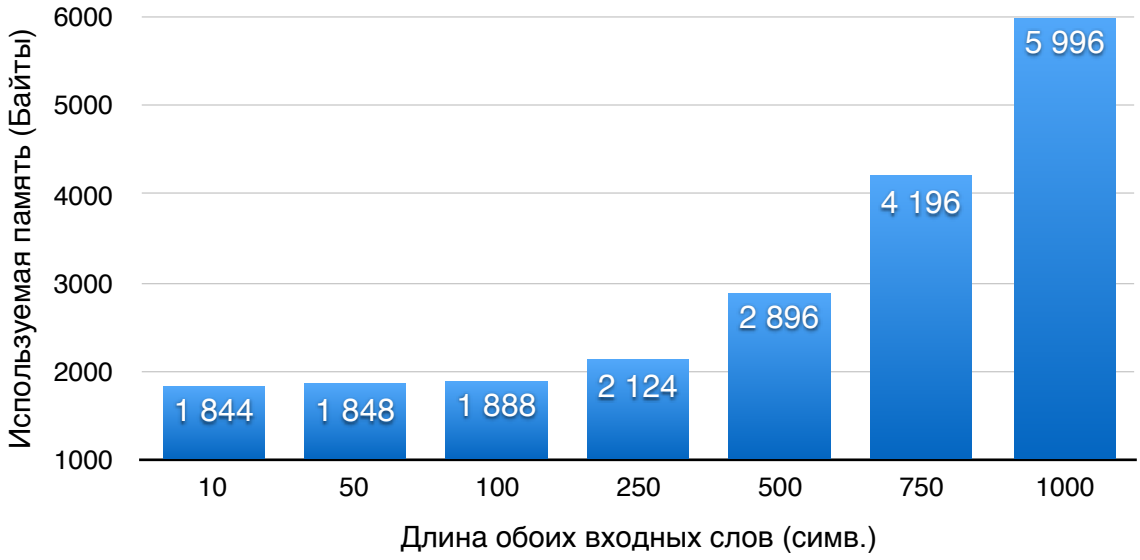


Рисунок 2 — Память, потребляемая программой



4.3 Сравнительный анализ на материале данных эксперимента

На примере алгоритма Левенштейна было показано, что данная рекурсивная реализация алгоритма поиска редакционного расстояния очень сильно проигрывает по времени работы, демонстрируя экспоненциальный рост процессорного времени.

На практике, большинство ошибок при печати представляют собой транспозицию соседних символов, и алгоритм Дамерау-Левенштейна, имеющий соответствующую проверку, предпочтительнее. Однако, у алгоритма Дамерау-Левенштейна также может быть рекурсивная реализация.

Принимая во внимание первый абзац, можно сделать вывод о том, что подобная реализация рекурсивного алгоритма Дамерау-Левенштейна будет демонстрировать такой же экспоненциальный рост времени работы и нецелесообразен для обработки даже не очень больших объёмов текста.

Таким образом, предпочтительней всего для исправления ошибок при пользовательском вводе использовать итеративный алгоритм Дамерау-Левенштейна.

Заключение

В данной работе были реализованы и протестированы самые популярные алгоритмы поиска редакционного расстояния - Левенштейна и Дамерау-Левенштейна. Проведён сравнительный анализ рекурсивной и нерекурсивной (итеративной) реализаций.

Итеративная реализация продемонстрировала наилучший результат по времени работы и лишена гипотетической проблемы переполнения стека вызовов.

Рекурсивная же реализация выигрывает лишь по простоте воплощения на языке программирования и удобочитаемости кода.