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: «Расстояния Левенштейна и Дамерау-Левенштейна»

Студент: Пронин А. С.

Группа: ИУ7-52Б

Преподаватель: Волкова Л. Л.

Оценка: _____

Москва

2021

Содержание

Введение

Цель работы – получить навык динамического программирования.
Задачи работы:

- изучить расстояния Левенштейна;
- изучить расстояния Дамерау-Левенштейна;
- разработать алгоритм вычисления расстояния Левенштейна обычным способом (матричным);
- разработать алгоритм вычисления расстояния Левенштейна рекурсивным способом;
- разработать алгоритм вычисления расстояния Левенштейна рекурсивным способом с кэшированием;
- разработать алгоритм вычисления расстояния Дамерау-Левенштейна обычным способом (матричным);
- реализовать алгоритмы;
- провести сравнительный анализ процессорного времени выполнения реализации алгоритмов;
- провести анализ пикового значения затрачиваемой памяти в программе.

1 Аналитический раздел

1.1 Расстояние Левенштейна

Расстояние Левенштейна (базовый вид редакторского расстояния)– это минимальное количество редакций необходимое для превращения одной строки в другую. [1]

Редакторские операции бывают:

- I (insert) - вставка
- D (delete) - удаление
- R (replace) – замена

У этих трёх операций штраф = 1.

Еще одна операция:

- M (Match) – совпадение

Эта операция не имеет штрафа (он равен нулю).

Пусть s1 и s2— две строки (длиной M и N соответственно) над некоторым афвалитом, тогда редакционное расстояние (расстояние Левенштейна) d(s1, s2) можно подсчитать по следующей рекуррентной

формуле: [2]

$$|x| = \begin{cases} 0, \text{если } i = 0, j = 0; \\ i, \text{если } i > 0, j = 0; \\ j, \text{если } i = 0, j > 0; \\ min \begin{pmatrix} D(s1[1 \dots i], s2[1 \dots j-1]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j-1]) + \begin{bmatrix} 0, s1[i] == s2[j] \\ 1, \text{иначе} \end{bmatrix} \end{pmatrix} \end{cases}$$

1.2 Расстояние Дамерау-Левенштейна

Вводится дополнительна операция: перестановка или транспозиция двух букв со штрафом 1. Если индексы позволяют и если две соседние буквы $s1[i] = s2[j-1] \wedge s1[i-1] = s2[j]$, то в минимум включается перестановка.

Пусть s1 и s2— две строки (длиной M и N соответственно) над некоторым афвалитом, тогда редакционное расстояние (расстояние Дамерау-Левенштейна) d(s1, s2) можно подсчитать по следующей рекуррентной

формуле: [3]

$$|x| = \begin{cases} 0, \text{если } i = 0, j = 0; \\ i, \text{если } i > 0, j = 0; \\ j, \text{если } i = 0, j > 0; \\ min \begin{pmatrix} D(s1[1 \dots i], s2[1 \dots j-1]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j-1]) + \begin{bmatrix} 0, s1[i] == s2[j] \\ 1, \text{иначе} \end{bmatrix} \\ D(s1[0 \dots i-2], s2[0 \dots j-2]) + 1 \end{pmatrix}, \\ \quad \text{если } i > 1, j > 1, s1[i-1] = s2[j-2], s1[i-2] = s2[j-1] \\ min \begin{pmatrix} D(s1[1 \dots i], s2[1 \dots j-1]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j]) + 1 \\ D(s1[1 \dots i-1], s2[1 \dots j-1]) + \begin{bmatrix} 0, s1[i] == s2[j] \\ 1, \text{иначе} \end{bmatrix} \end{pmatrix}, \text{иначе} \end{cases}$$

1.3 Применение

Расстояние Левенштейна и его обобщения активно применяются:

- при автозамене
- в поисковых строках
- "возможно вы имели ввиду"
- В биоинформатике (кодируем молекулы буквами)

2 Конструкторский раздел

2.1 Разработка алгоритмов

В разделе представлены схемы следующих алгоритмов вычисления расстояния:

- Левенштейна обычным способом (матричным) (рисунки 2.1-2.2)
- Левенштейна рекурсивным способом (рисунки 2.3-2.4)
- Левенштейна рекурсивным способом с кэшированием (рисунки 2.5-2.6)
- Дамерау-Левенштейна обычным способом (матричным) (рисунки 2.7-2.8)

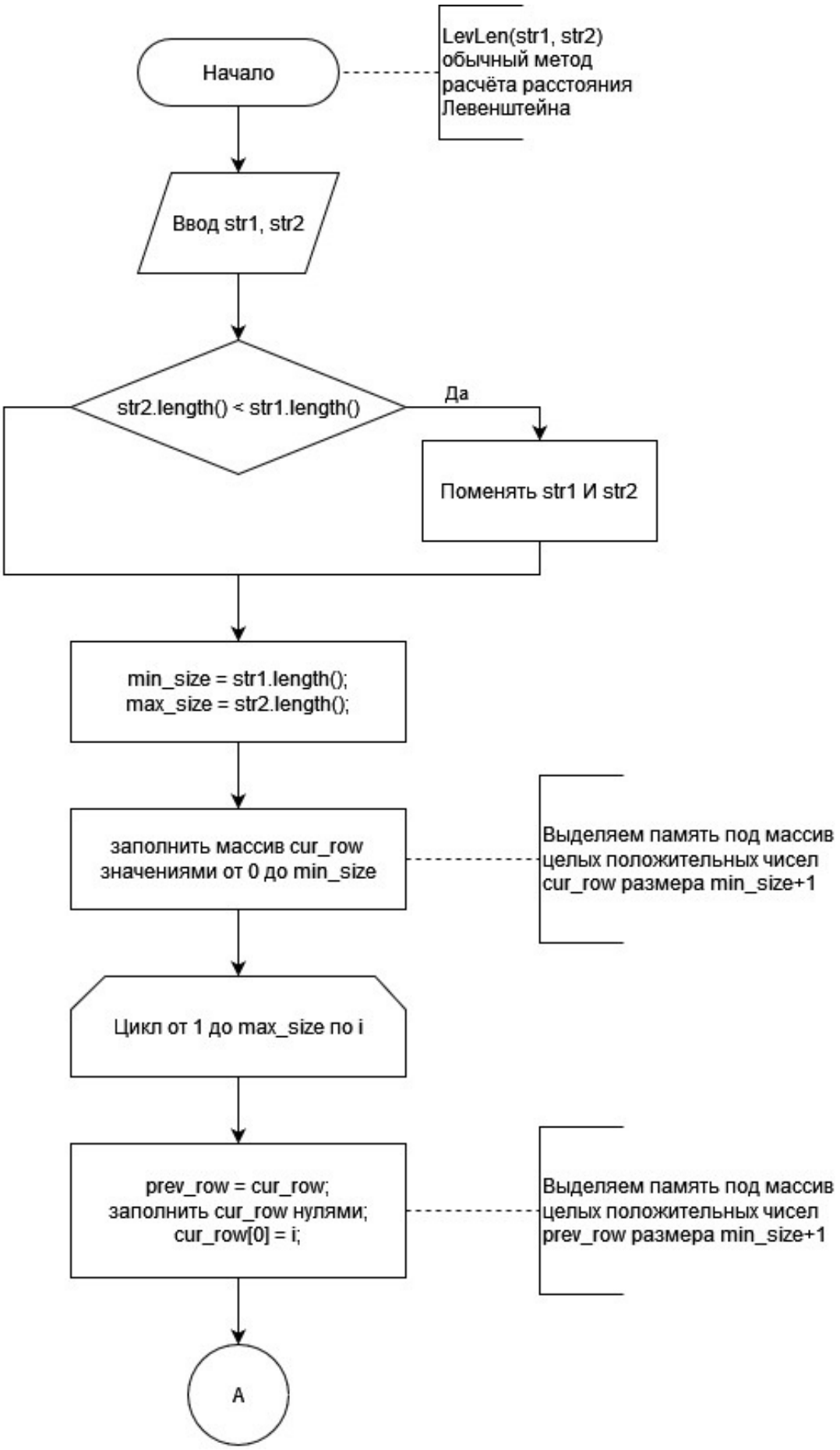


Рис. 2.1: Схема алгоритма Левенштейна, часть 1

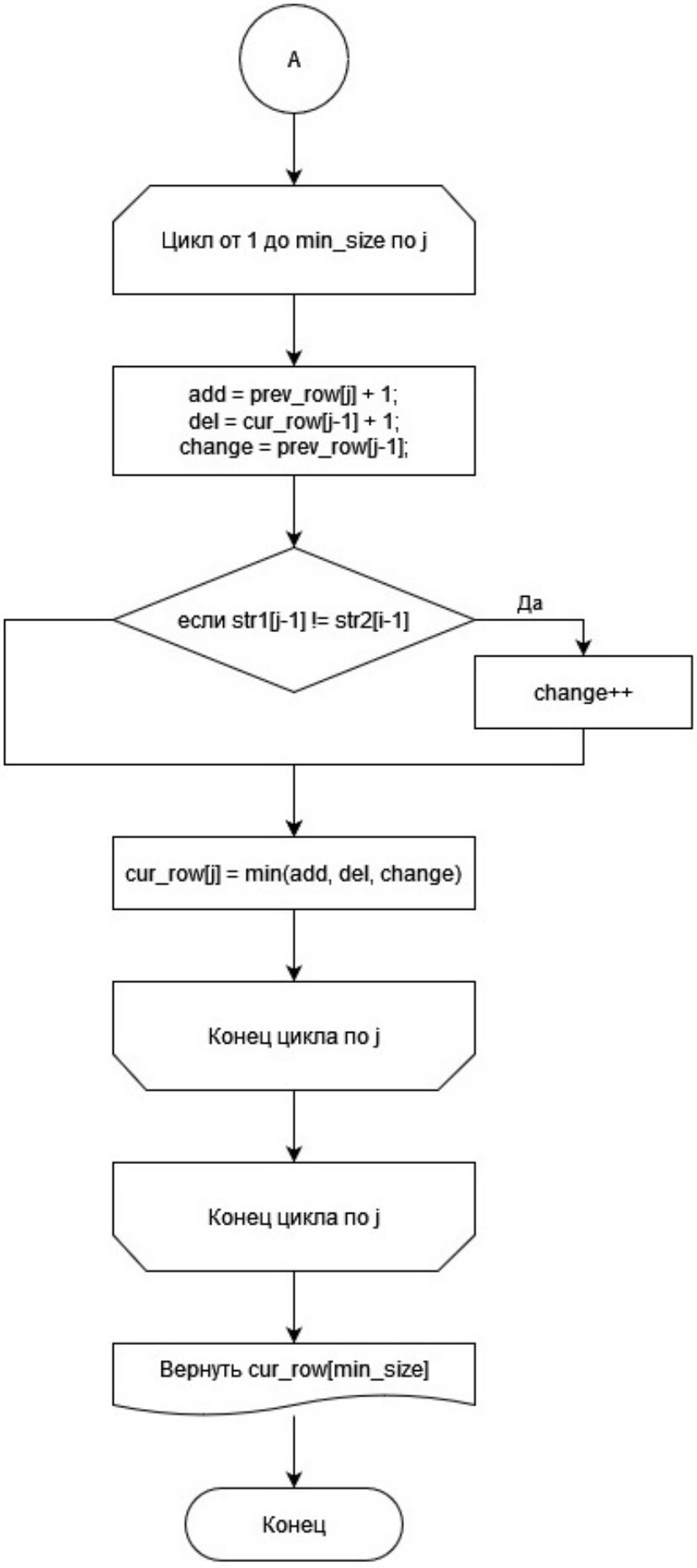


Рис. 2.2: Схема алгоритма Левенштейна, часть 2

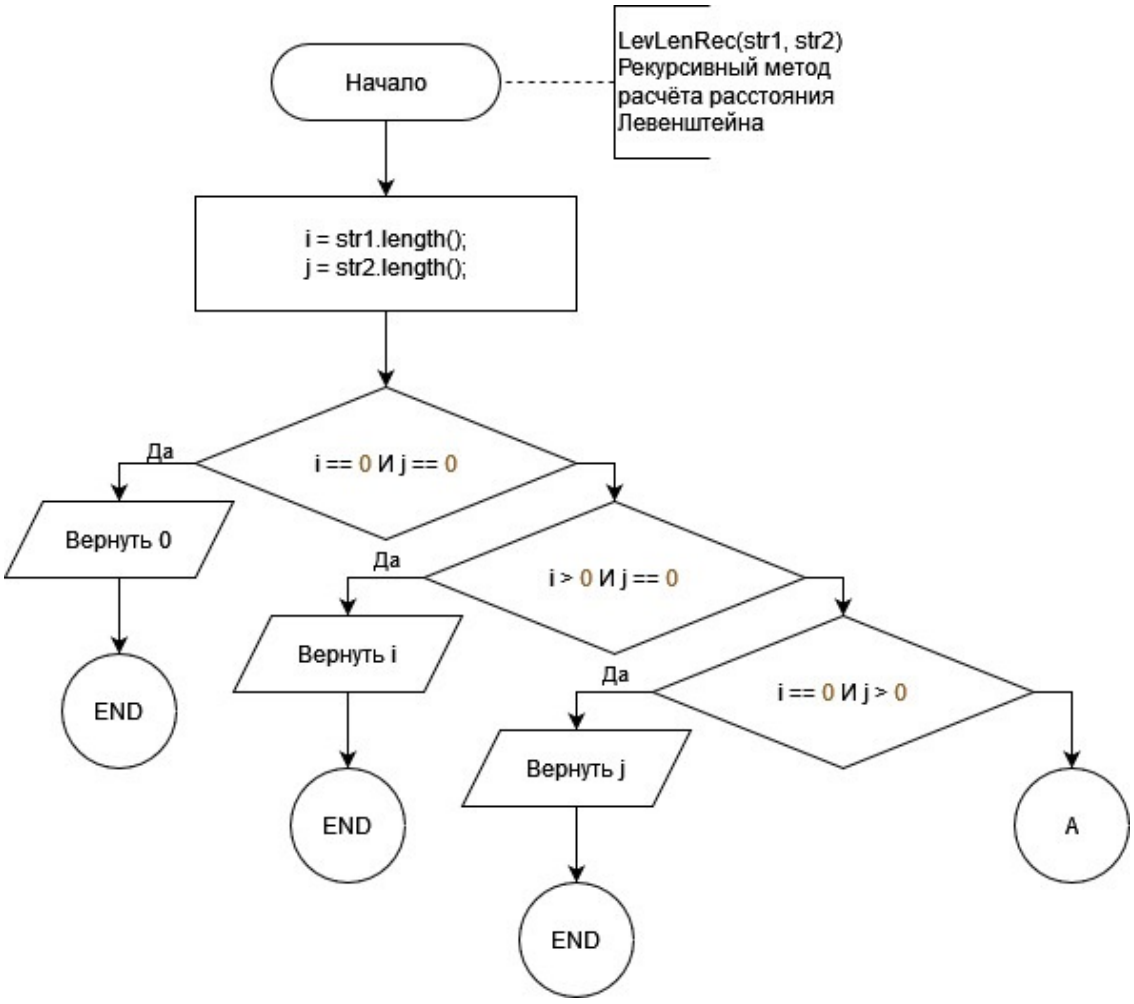


Рис. 2.3: Схема алгоритма Левенштейна с рекурсией, часть 1

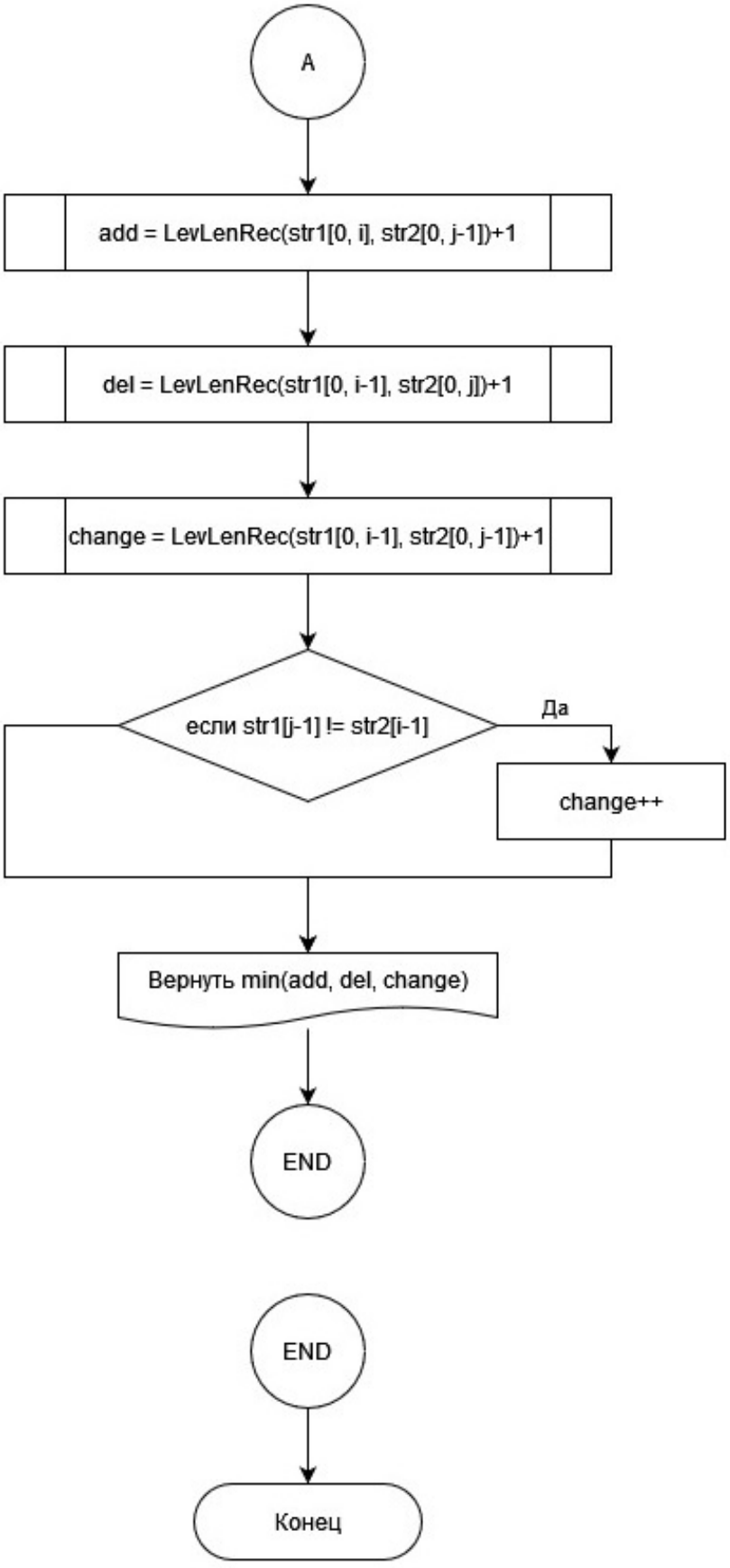


Рис. 2.4: Схема алгоритма Левенштейна с рекурсией, часть 2

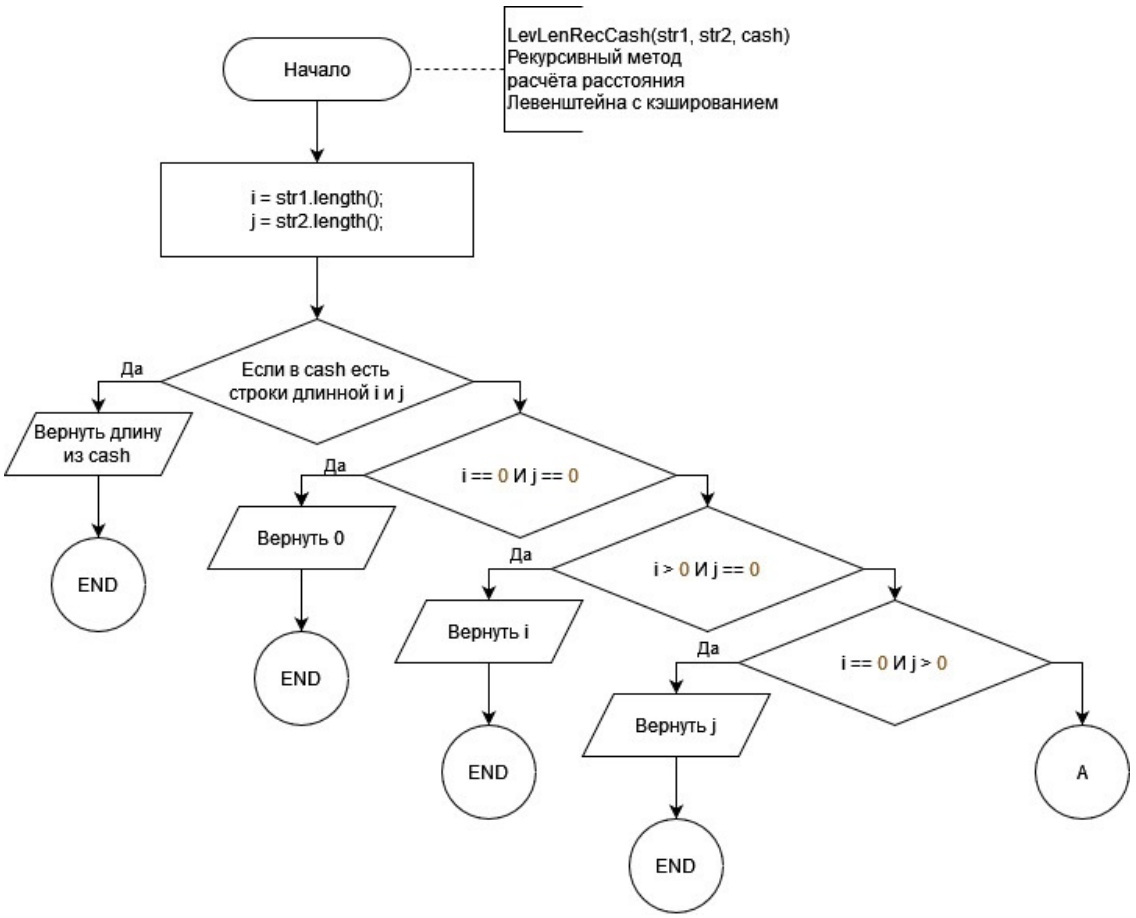


Рис. 2.5: Схема алгоритма Левенштейна с рекурсией и кэшированием, часть 1

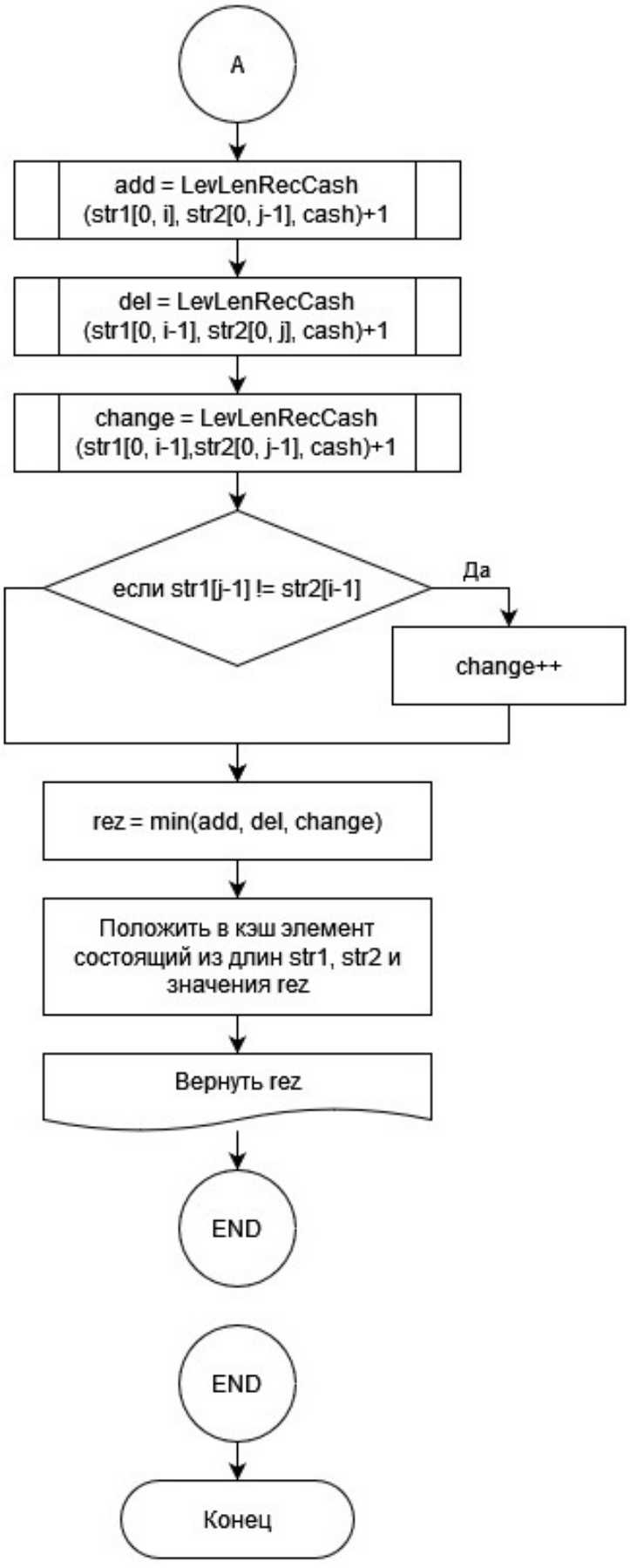


Рис. 2.6: Схема алгоритма Левенштейна с рекурсией и кэшированием, часть 2

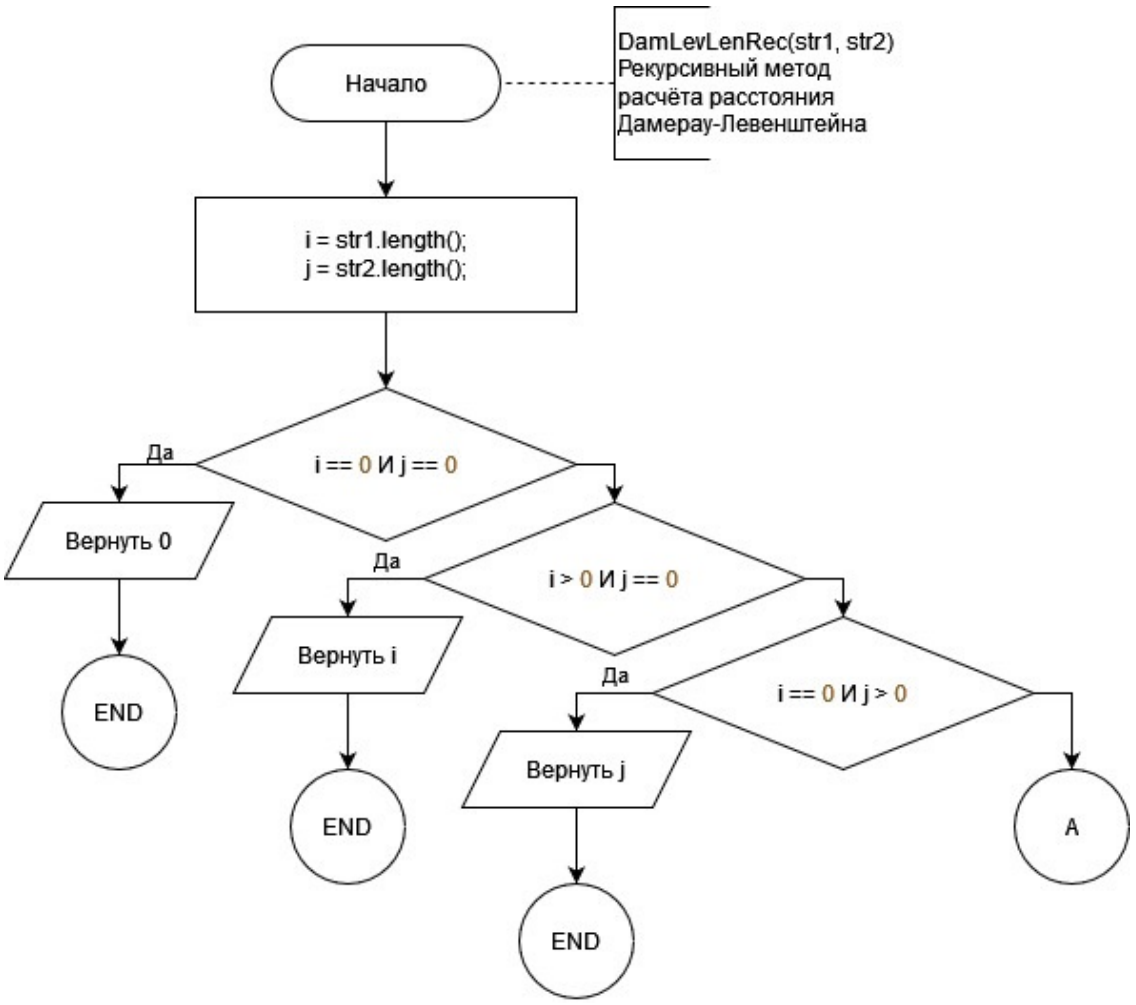


Рис. 2.7: Схема алгоритма Дамерау-Левенштейна с рекурсией, часть 1

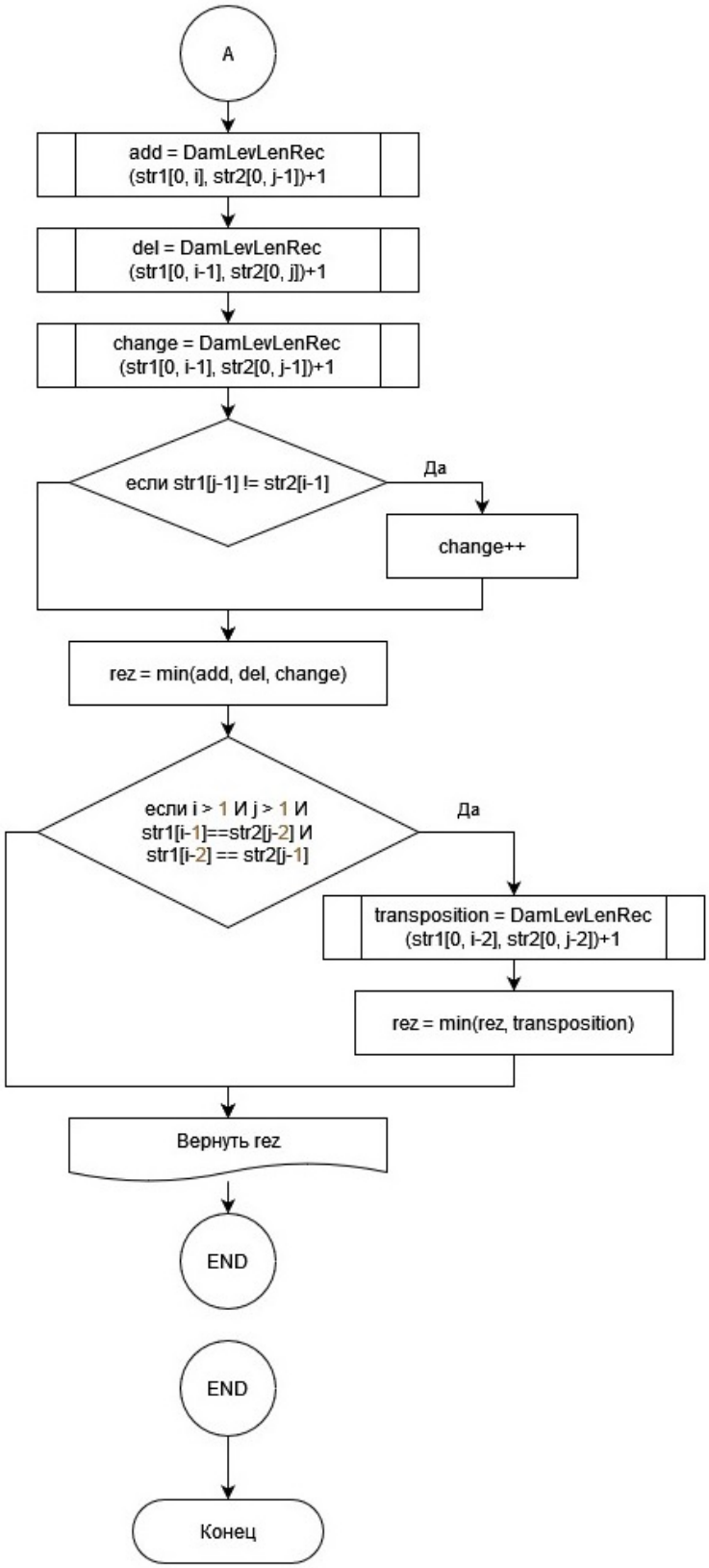


Рис. 2.8: Схема алгоритма Дамерау-Левенштейна с рекурсией, часть 2

3 Технологический раздел

3.1 Выбор инструментов

Язык программирования (c++), среда разработки (QtCreator), библиотеки (windows.h?), valgrind

1. Язык программирования C++

C++ – язык программирования общего назначения с уклоном в сторону системного программирования. [4]

Данный язык, достаточно популярен и широко распространён, кроме этого он имеет ряд следующих плюсов:

1) Высокая производительность: Язык спроектирован так, чтобы дать программисту максимальный контроль над всеми аспектами структуры и порядка исполнения программы. Один из базовых принципов C++ «не платишь за то, что не используешь» то есть ни одна из языковых возможностей, приводящая к дополнительным накладным расходам, не является обязательной для использования. Имеется возможность работы с памятью на низком уровне.

2) Кроссплатформенность: стандарт языка C++ накладывает минимальные требования на ЭВМ для запуска скомпилированных программ.

3) Поддержка различных стилей программирования: традиционное императивное программирование (структурное, объектно-ориентированное), обобщённое программирование, функциональное программирование, порождающее метапрограммирование. Исходя из вышеперечисленных плюсов, очевиден выбор данного языка программирования для реализации поставленной задачи.

Исходя из вышеперечисленных плюсов, очевиден выбор данного языка программирования для реализации поставленной задачи.

Для замера процессорного времени выполнения программы исполь-

зуется функция GetProcessTimes() т.к. я пишу код под Windows. [5]

Для оценки памяти, используемой программой используется valgrind. Для его использования была установлена wsl (Windows Subsystem for Linux) и виртуальная машина с Ubuntu. Для замера максимального кол-ва памяти используемой алгоритмами созданы отдельные файлы, которые выполняют отдельно взятый алгоритм со строками определённой длины. Затем их компилирует с помощью g++ и получают информацию об используем памяти с помощью valgrind –tool=massif. [6]

3.2 Реализация алгоритмов

На листингах , представлены реализации алгоритмов...

Листинг 3.1: Реализация алгоритма Левенштейна обычным способом

```
size_t LevLen(string str1, string str2)
{
        if (str2.length() < str1.length())
        {
                return LevLen(str2, str1);
        }

        const size_t min_size = str1.length();
        const size_t max_size = str2.length();

        vector<size_t> cur_row(min_size+1);
        for (size_t i = 1; i <= min_size; i++)
        {
                cur_row[i] = i;
        }

        for (size_t i = 1; i <= max_size; i++)
        {
                vector<size_t> prev_row = cur_row;
                cur_row = vector<size_t>(min_size+1);
                cur_row[0] = i;
                for (size_t j = 1; j <= min_size; j++)
                {
                        size_t add = prev_row[j] + 1;
                        size_t del = cur_row[j-1] + 1;
                        size_t change = prev_row[j-1];
                        if (str1[j-1] != str2[i-1])
                        change++;
                        cur_row[j] = min(add, min(del, change));
```

```
30                  }
31          }
32
33          return cur_row[min_size];
34  }
```

Листинг 3.2: Реализация алгоритма Левенштейна рекусрсивным способом

```
1   size_t LevLenRec(string str1, string str2)
2   {
3           size_t i = str1.length();
4           size_t j = str2.length();
5           if (i == 0 && j == 0)
6           {
7                   return 0;
8           }
9           else if (i > 0 && j == 0)
10          {
11                  return i;
12          }
13          else if (i == 0 && j > 0)
14          {
15                  return j;
16          }
17          else
18          {
19                  size_t add = LevLenRec(str1.substr(0, i),
                        str2.substr(0, j-1)) + 1;
20                  size_t del = LevLenRec(str1.substr(0, i-1),
                        str2.substr(0, j)) + 1;
21                  size_t change = LevLenRec(str1.substr(0, i-1),
                        str2.substr(0, j-1));
22                  if (str1[i-1] != str2[j-1])
23                  change++;
24                  return  min(add, min(del, change));
25          }
26  }
```

Листинг 3.3: Реализация алгоритма Левенштейна с рекурсией и кэшированием

```
1   bool checkCash(string str1, string str2, const cash_t &cash, size_t&
        rez)
2   {
3           for (auto &el : cash)
4           {
5                   if (el.first.first == str1 && el.first.second == str2)
6                   {
7                           rez = el.second;
```

```
8                       return true;
9               }
10      }
11      return false;
12  }
13
14  size_t LevLenRecCash(string str1, string str2, cash_t &cash)
15  {
16      size_t len = INT_MAX;
17      if (checkCash(str1, str2, cash, len))
18      {
19              return len;
20      }
21      //cout << str1 << " " << str2 << endl;
22      size_t i = str1.length();
23      size_t j = str2.length();
24      if (i == 0 && j == 0)
25      {
26              return 0;
27      }
28      else if (i > 0 && j == 0)
29      {
30              return i;
31      }
32      else if (i == 0 && j > 0)
33      {
34              return j;
35      }
36      else
37      {
38              size_t add = LevLenRecCash(str1.substr(0, i),
                    str2.substr(0, j-1), cash) + 1;
39              size_t del = LevLenRecCash(str1.substr(0, i-1),
                    str2.substr(0, j), cash) + 1;
40              size_t change = LevLenRecCash(str1.substr(0, i-1),
                    str2.substr(0, j-1), cash);
41              if (str1[i-1] != str2[j-1])
42              change++;
43
44              size_t rez = min(add, min(del, change));
45              cash.push_back(make_pair(make_pair(str1, str2), rez));
46              return  rez;
47      }
48  }
49
50  size_t LevLenRecCash(string str1, string str2)
51  {
52      cash_t cash = cash_t();
```

```
53          return LevLenRecCash(str1, str2, cash);
54  }
```

Листинг 3.4: Реализация алгоритма Дамеру-Левенштейна рекурсивным способом

```
size_t DamLevLenRec(string str1, string str2)
{
        size_t i = str1.length();
        size_t j = str2.length();
        if (i == 0 && j == 0)
        {
                return 0;
        }
        else if (i > 0 && j == 0)
        {
                return i;
        }
        else if (i == 0 && j > 0)
        {
                return j;
        }
        else
        {
                size_t add = DamLevLenRec(str1.substr(0, i),
                   str2.substr(0, j-1)) + 1;
                size_t del = DamLevLenRec(str1.substr(0, i-1),
                   str2.substr(0, j)) + 1;
                size_t change = DamLevLenRec(str1.substr(0, i-1),
                   str2.substr(0, j-1));
                if (str1[i-1] != str2[j-1])
                change++;
                size_t rez = min(add, min(del, change));
                if (i > 1 && j > 1 && str1[i-1]==str2[j-2] && str1[i-2]
                   == str2[j-1])
                {
                        size_t transposition =
                           DamLevLenRec(str1.substr(0, i-2),
                           str2.substr(0, j-2)) + 1;
                        rez = min(rez, transposition);
                }
                return  rez;
        }
}
```

4 Исследовательский раздел

4.1 Тесты

Были подготовлены тесты для вычисления расстояний между следующими строками:

- ”skat” и ”kot”;
- ”skat” и ””;
- ”” и ”kot”;
- ”” и ””;
- ”kot” и ”kot”;
- ”ab” и ”ba”.

Также на рисунках 4.1 - 4.6 представлены результаты работы программы.

```
LevLen() between "skat" and "kot" = 2
LevLenRec() between "skat" and "kot" = 2
LevLenRecCash() between "skat" and "kot" = 2
DamLevLenRec() between "skat" and "kot" = 2
length between "skat" and "kot" = 2
```

Рис. 4.1: Получившиеся расстояния между строками ”skat” и ”kot”

```
LevLen() between "skat" and "" = 4
LevLenRec() between "skat" and "" = 4
LevLenRecCash() between "skat" and "" = 4
DamLevLenRec() between "skat" and "" = 4
```

Рис. 4.2: Получившиеся расстояния между строками ”skat” и ””

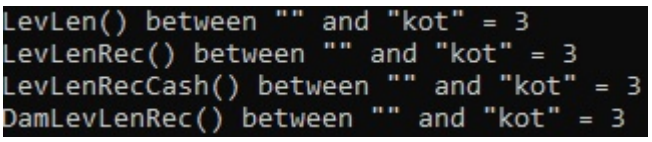


Рис. 4.3: Получившиеся расстояния между строками "" и "kot"

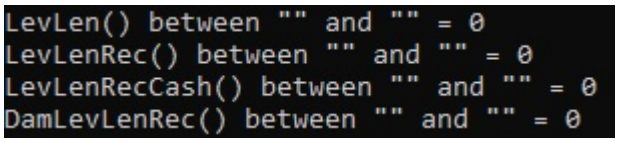


Рис. 4.4: Получившиеся расстояния между строками "" и ""

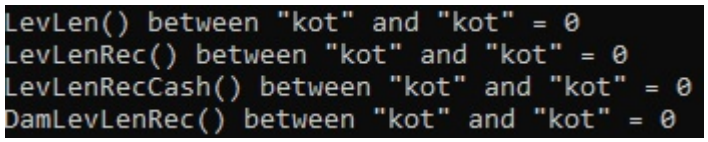


Рис. 4.5: Получившиеся расстояния между строками "kot" и "kot"

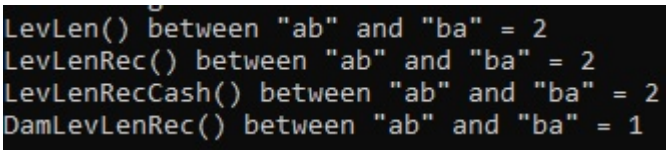


Рис. 4.6: Получившиеся расстояния между строками "ab" и "ba"

Как видно по рисункам, ответы сошлись с ожидаемым результатом, следовательно программа работает верно.

4.2 Сравнительный анализ времени выполнения алгоритмов

На графиках представленных ниже (рисунки 4.7-4.9), изображены зависимости времени выполнения алгоримов от длин слов, между которыми считались расстояния:

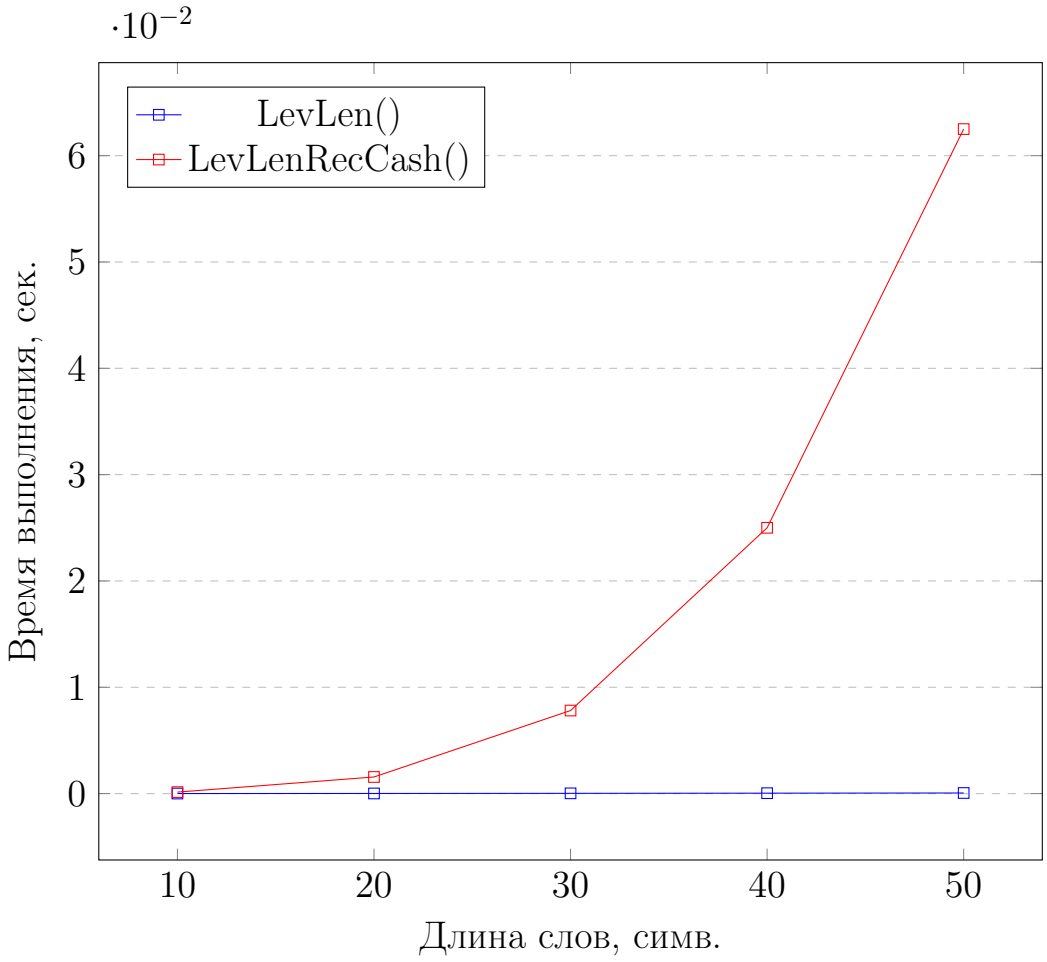


Рис. 4.7: Зависимость времени выполнения алгоритмов LenLev() и LenLevRecCash() от длины слов

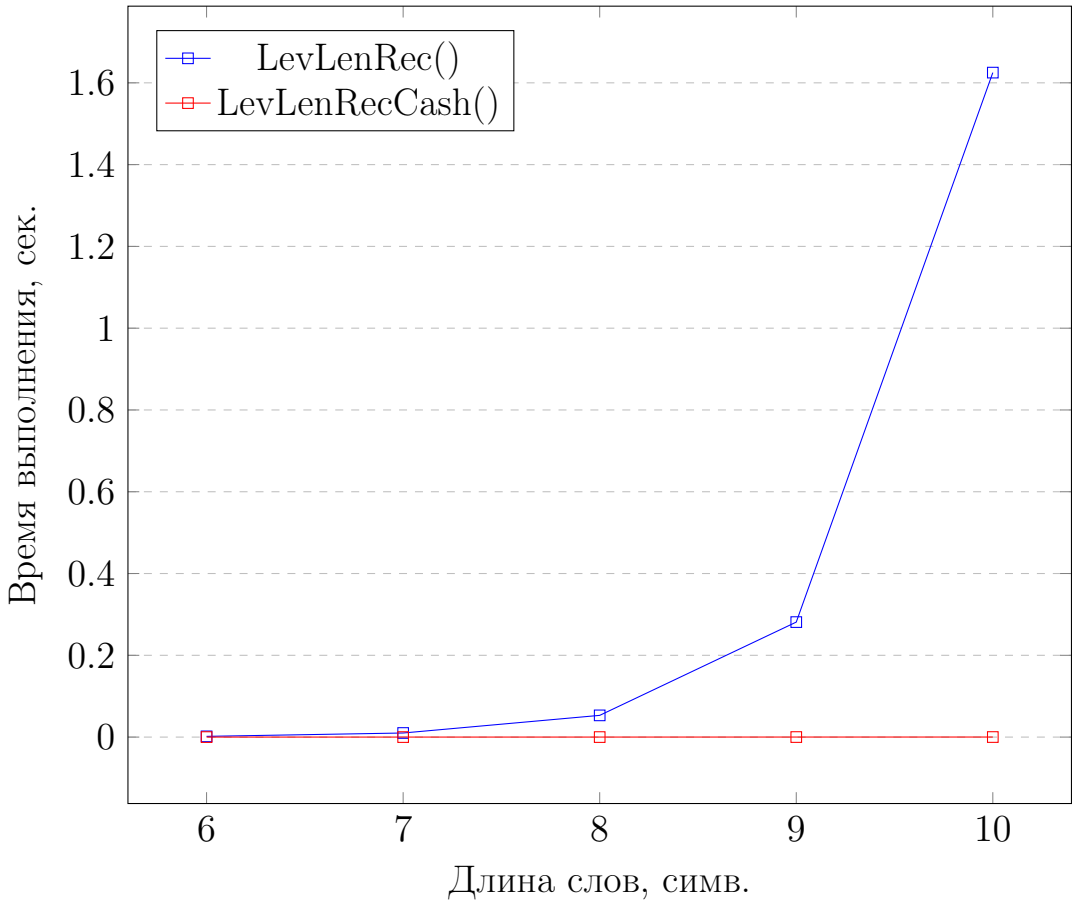


Рис. 4.8: Зависимость времени выполнения алгоритмов LenLevRec() и LenLevRecCash() от длины слов

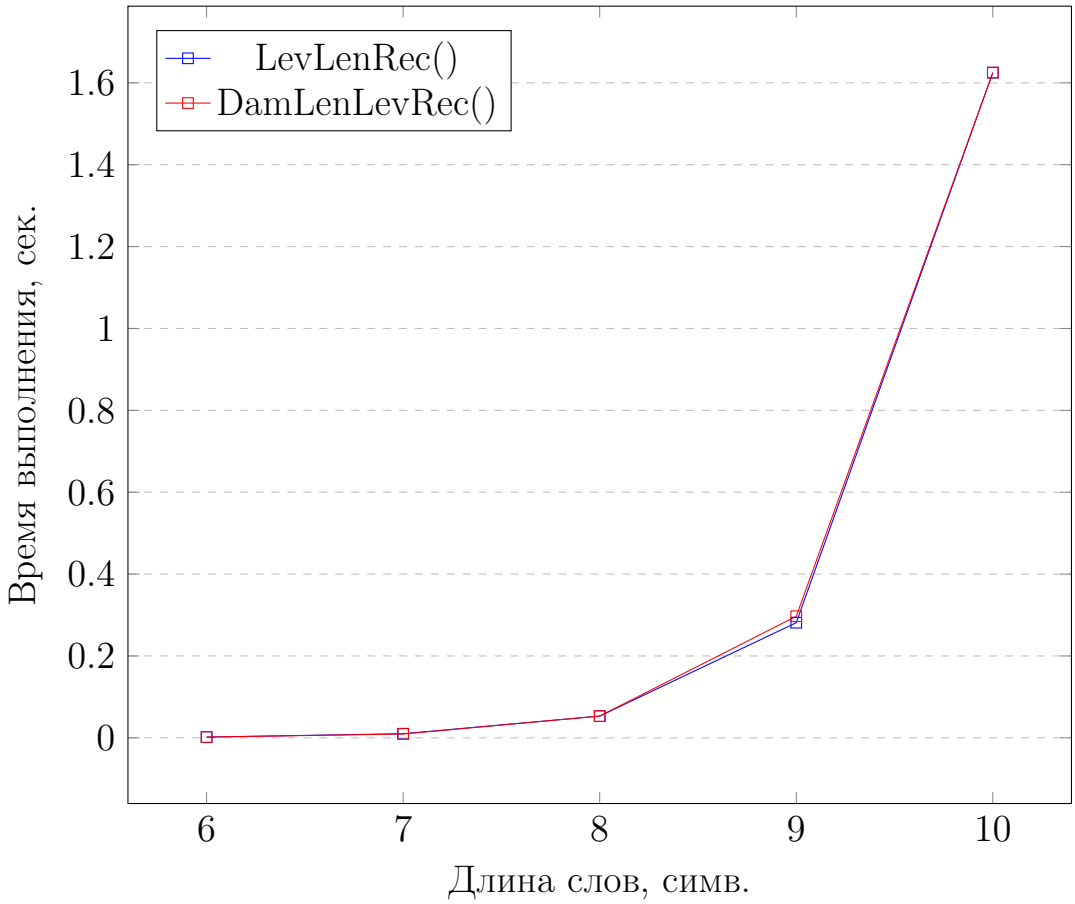
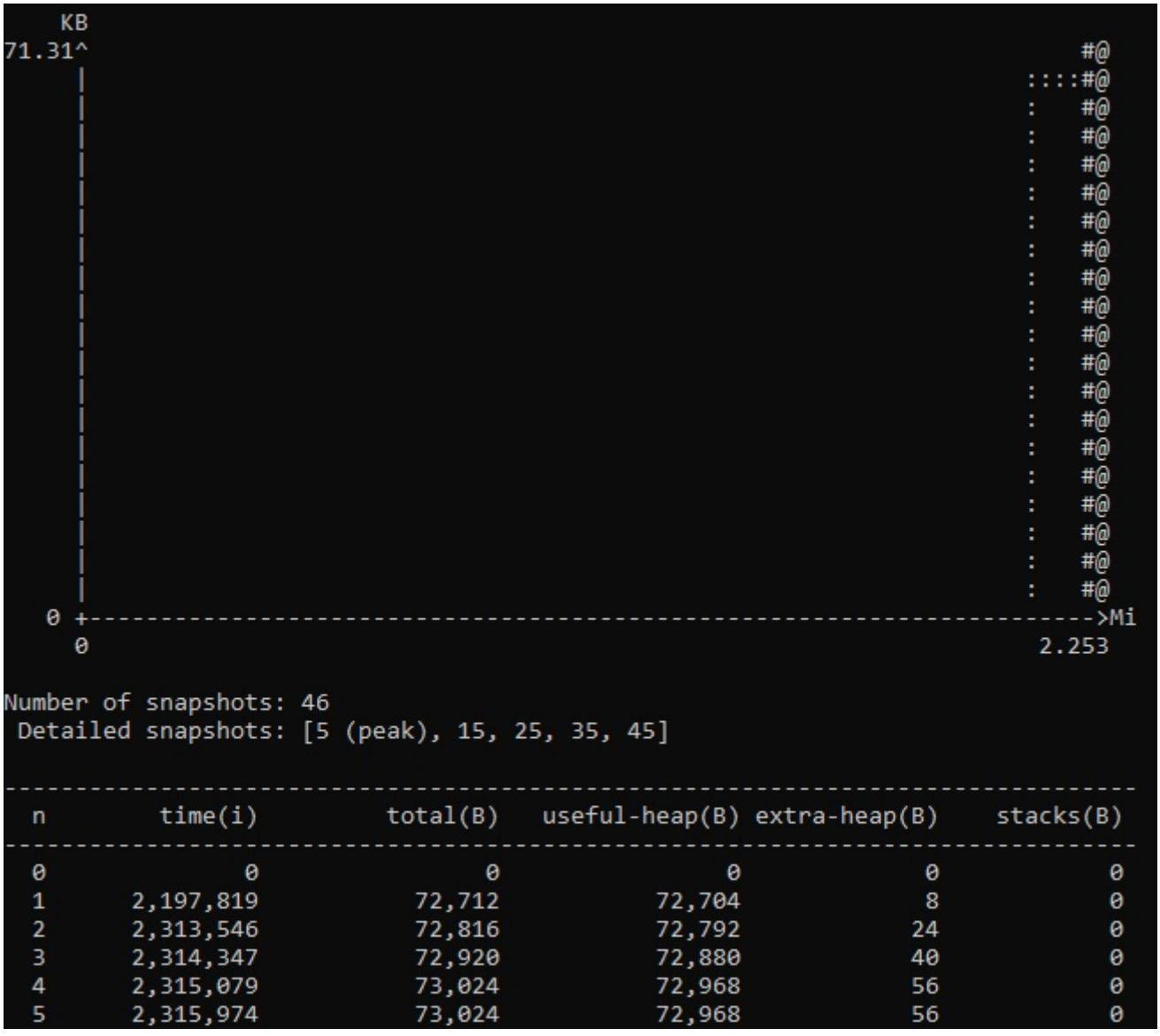


Рис. 4.9: Зависимость времени выполнения алгоритмов LenLevRec() и DamLenLevRec() от длины слов

Проанализировав графики, приходим к выводу, что самым быстрым алгоритмом является обычный (матричный) Левенштейн. А самыми медленными рекурсивные алгоритмы без кэширования.

4.3 Сравнительный анализ времени выполнения алгоритмов

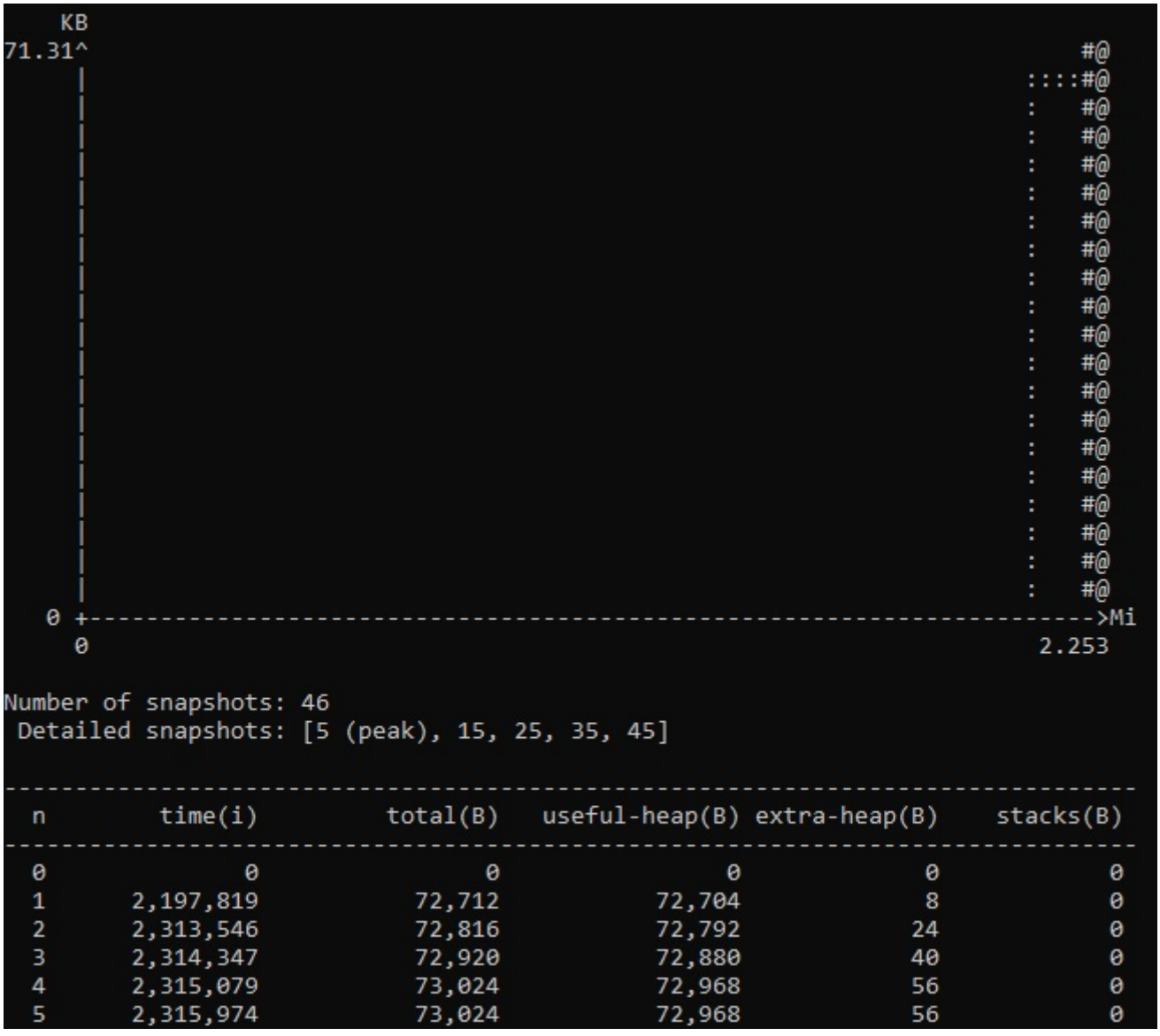
На рисунках 4.10 - 4.13 предсатвлены результаты оценки памяти алгоритмов, с помощью valgrind.



Number of snapshots: 46
Detailed snapshots: [5 (peak), 15, 25, 35, 45]

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,197,819	72,712	72,704	8	0
2	2,313,546	72,816	72,792	24	0
3	2,314,347	72,920	72,880	40	0
4	2,315,079	73,024	72,968	56	0
5	2,315,974	73,024	72,968	56	0

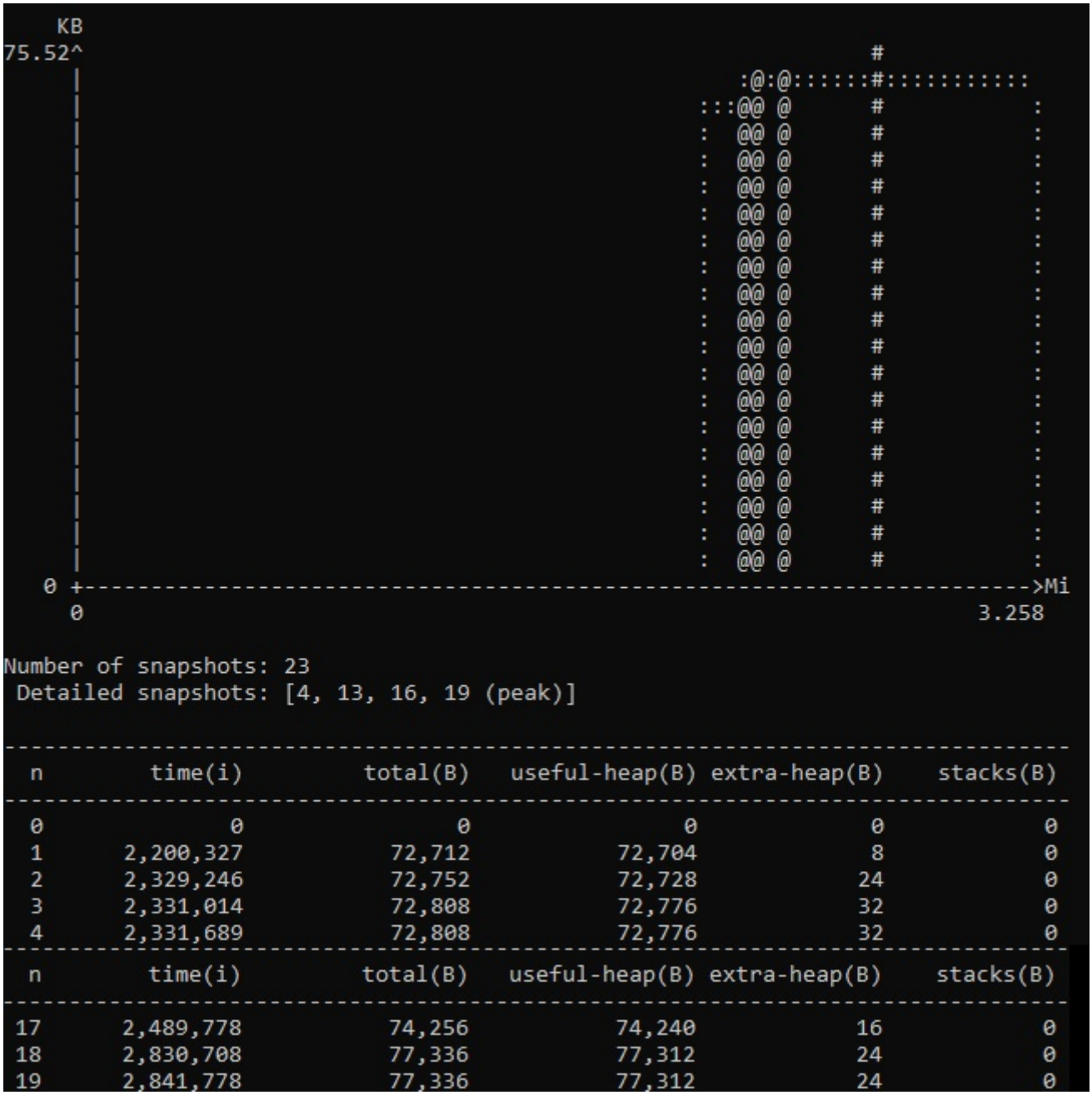
Рис. 4.10: Оценка памяти алгоритма LevLen()



Number of snapshots: 46
Detailed snapshots: [5 (peak), 15, 25, 35, 45]

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,197,819	72,712	72,704	8	0
2	2,313,546	72,816	72,792	24	0
3	2,314,347	72,920	72,880	40	0
4	2,315,079	73,024	72,968	56	0
5	2,315,974	73,024	72,968	56	0

Рис. 4.11: Оценка памяти алгоритма LevLenRec()



n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
0	0	0	0	0	0
1	2,200,327	72,712	72,704	8	0
2	2,329,246	72,752	72,728	24	0
3	2,331,014	72,808	72,776	32	0
4	2,331,689	72,808	72,776	32	0

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)	stacks(B)
17	2,489,778	74,256	74,240	16	0
18	2,830,708	77,336	77,312	24	0
19	2,841,778	77,336	77,312	24	0

Рис. 4.12: Оценка памяти алгоритма LevLenRecCash()

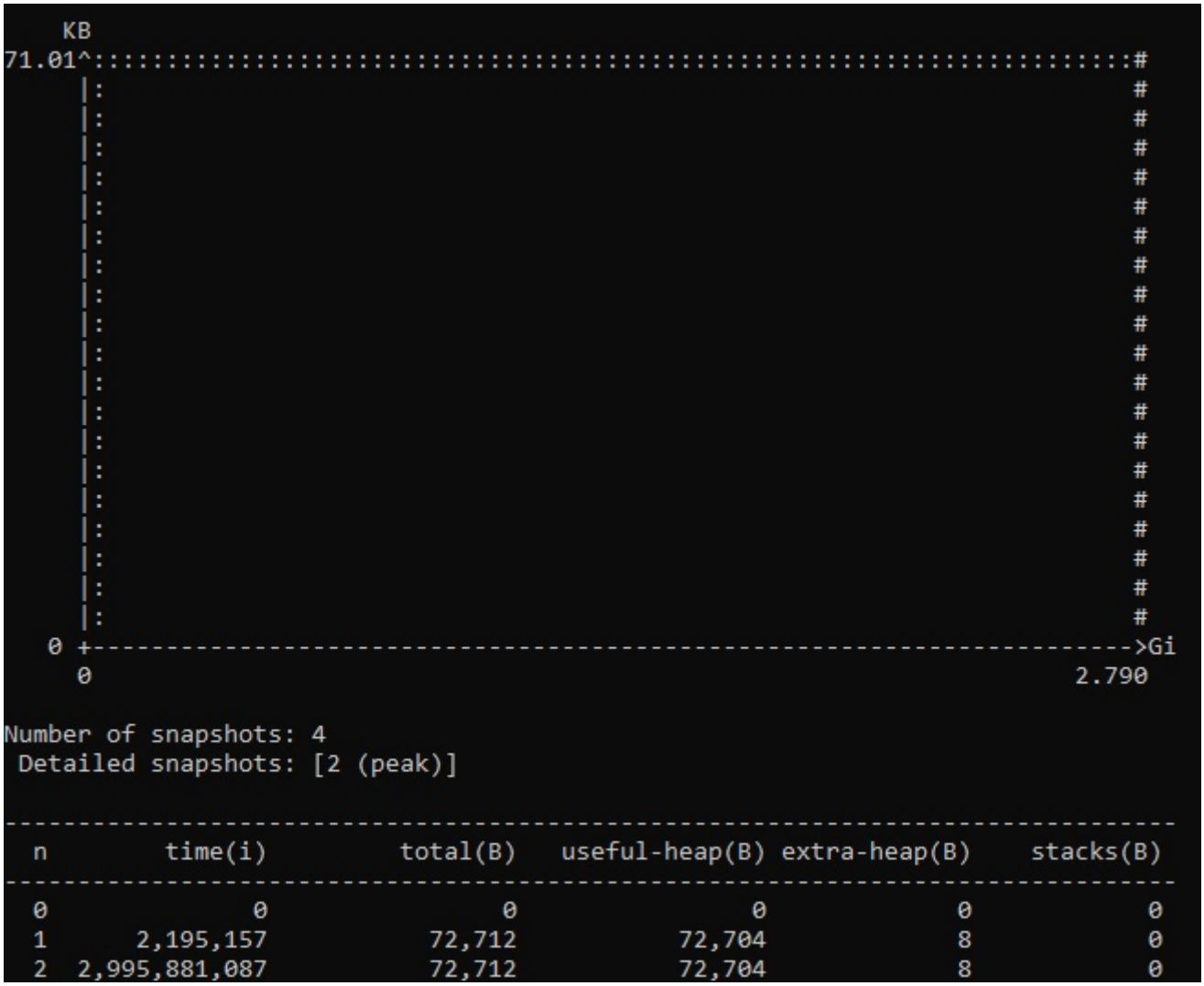


Рис. 4.13: Оценка памяти алгоритма DamLevLenRec()

Судя по результатам меньше всего памяти нужно рекусривным алгоритмам, а больше всего алгоритму с кэшированием.

4.4 Вывод

По итогу иследования выяснилось, что разработанная программа работает верно. Кроме этого, смотря на время выполнения и используемую память каждого алгоритма, логично сделать вывод, что наиболее выгодный по использованию данных ресурсов, является обычный (матричный) Левенштейн.

Заключение

По итогу проделанной работы была достигнута цель - получить навык динамического программирования.

Также были решены все поставленные задачи, а именно:

- изучено расстояние Левенштейна;
- изучено расстояние Дамерау-Левенштейна;
- разработан алгоритм вычисления расстояния Левенштейна обычным способом (матричным);
- разработан алгоритм вычисления расстояния Левенштейна рекурсивным способом;
- разработан алгоритм вычисления расстояния Левенштейна рекурсивным способом с кэшированием;
- разработан алгоритм вычисления расстояния Дамерау-Левенштейна обычным способом (матричным);
- реализованы все алгоритмы;
- проведён сравнительный анализ процессорного времени выполнения реализации алгоритмов;
- проведён анализ пикового значения затрачиваемой памяти в программе.

Список использованных источников

[1] В.И. Левенштейн. *Двоичные коды с исправлением выпадений, вставок и замещений символов.* Доклады АН СССР, 1965.

[2] Сметанин Н. Нечёткий поиск в тексте и словаре [Электронный ресурс]. // URL: `https://habr.com/ru/post/114997/`.

[3] Как рассчитать расстояние Левенштейна? [Электронный ресурс]. // URL: `https://ru.minecraftfullmod.com/1072-how-to-calculate-levenshtein-distance-in-java`.

[4] Бьерн Страуструп. *Язык программирования C++ - специальное издание.* Москва: Бином, 2010. 1136 с.

[5] Getprocesstimes function (processthreadsapi.h) [Электронный ресурс]. // URL: `https://docs.microsoft.com/en-us/windows/win32/api/processthreadsapi/nf-processthreadsapi-getprocesstimes#syntax`.

[6] Massif: a heap profiler [Электронный ресурс]. // URL: `https://valgrind.org/docs/manual/ms-manual.html`.