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»

Отчёт

по лабораторной работе №1

Название: Расстояния Левенштейна и Дамерау-Левенштейна

Дисциплина: Анализ алгоритмов

Студент ИУ7-55Б (Группа) _____ (Подпись, дата) А.К. Клименко (И.О. Фамилия)

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

Москва, 2021

Содержание

Введение

Данная работа направлена на изучение и применение алгоритмов нахождения редакционного расстояния на примере алгоритмов Левенштейна и Дамерау-Левенштейна.

Цель работы: провести сравнительный анализ реализаций четырёх алгоритмов нахождения редакционного расстояния по двум критериям: затраченному процессорному времени и по использованию памяти.

Задачи.

1) Реализовать 4 алгоритма поиска редакционного расстояния: алгоритмы Левенштейна (с использованием матрицы и рекурсивных вычислений) и алгоритмы Дамерау-Левенштейна.
2) Замерить процессорное время работы каждого алгоритма.
3) Замерить (либо теоретически рассчитать) пиковое значение памяти, используемое каждым алгоритмом.

1 Аналитический раздел

Редакционное расстояние – это метрика, определяющая модуль разности между последовательностями символов. Оно определяется как минимальное число элементарных операций, применяя которые к одной строке, можно получить другую.

Разные алгоритмы используют разные наборы элементарных операций.

1.1 Расстояние Левенштейна

При нахождении расстояния Левенштейна используется следующий набор операций:

— вставка одного символа в строку;
— удаление одного символа из строки;
— замена одного символа на другой.

Расстояние Левенштейна можно подсчитать по рекуррентной формуле (**??**):

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

где $s[1..k]$ - подстрока длиной k и $M(a,b) = \begin{cases} 0, & a = b \\ 1, & a \neq b \end{cases}$

1.2 Расстояние Дамерау-Левенштейна

Алгоритм Дамерау-Левенштейна использует расширенный набор операций алгоритма Левенштейна. Добавляется операция транспонирования - перестановки двух соседних символов.

Расстояние Дамерау-Левенштейна можно подсчитать по рекуррентной формуле (**??**):

$$D(s_1[1..i],s_2[1..j]) = \begin{cases} max(i,j), & min(i,j) = 0 \\ min \begin{cases} D(s_1[1..i], s_2[1..j-1]) + 1 \\ D(s_1[1..i-1], s_2[1..j]) + 1 \\ D(s_1[1..i-2], s_2[1..j-2]) + 1 \\ D(s_1[1..i-1],s_2[1..j-1]) + M(s[i], s[j]) \end{cases} & \begin{cases} i,j > 1 \\ s_1[i-1] = s_2[j] \\ s_1[i] = s_2[j-1] \end{cases} \\ min \begin{cases} D(s_1[1..i], s_2[1..j-1]) + 1 \\ D(s_1[1..i-1], s_2[1..j]) + 1 \\ D(s_1[1..i-1],s_2[1..j-1]) + M(s[i], s[j]) \end{cases} & \text{иначе.} \end{cases} \tag{1.2}$$

2 Конструкторский раздел

В данном разделе будут рассмотрены схемы алгоритмов, требования к функциональности ПО, и опредены способы тестирования.

2.1 Схемы алгоритмов

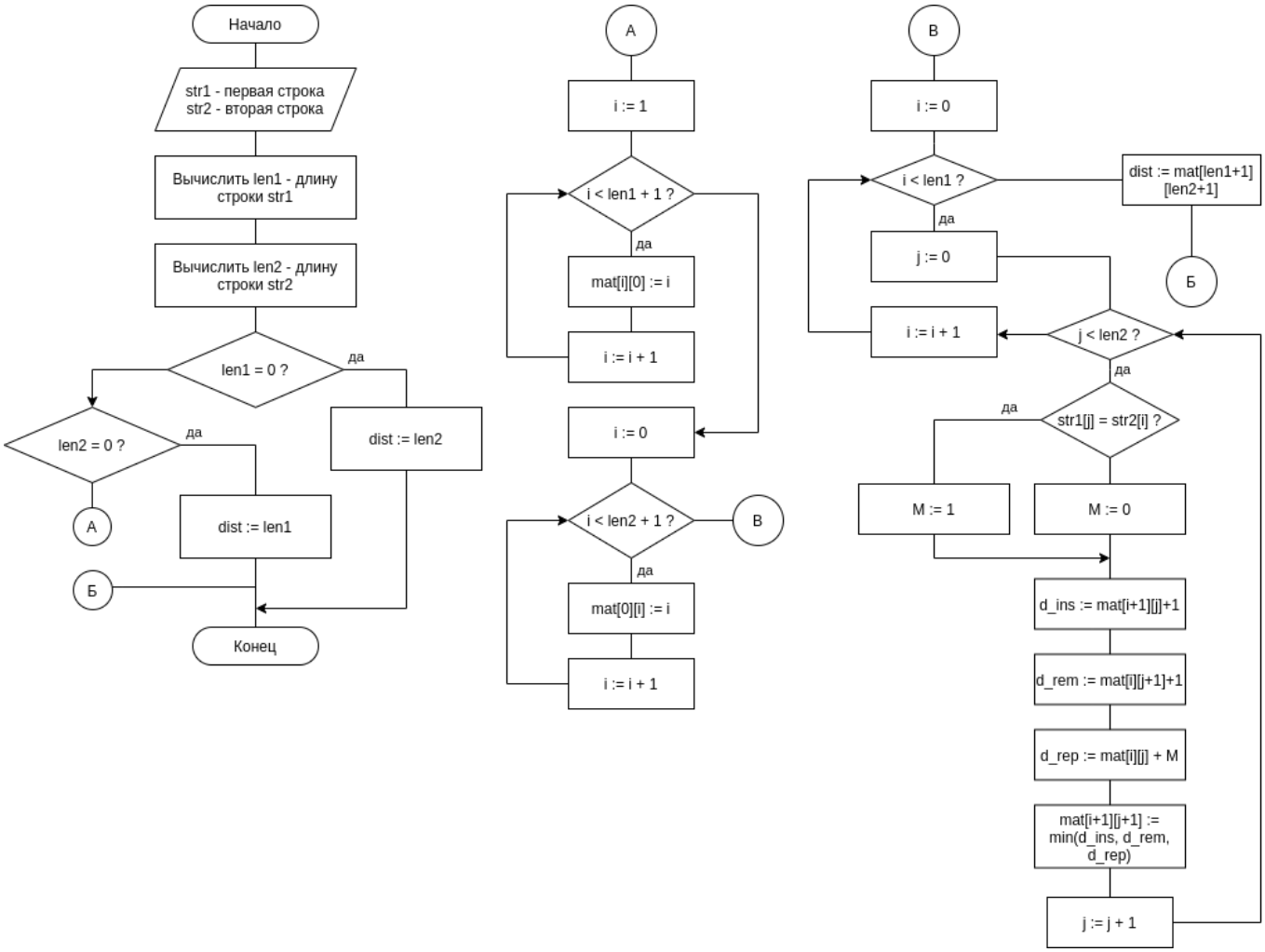

Рисунок 2.1 — Схема матричного алгоритма нахождения расстояния Левенштейна

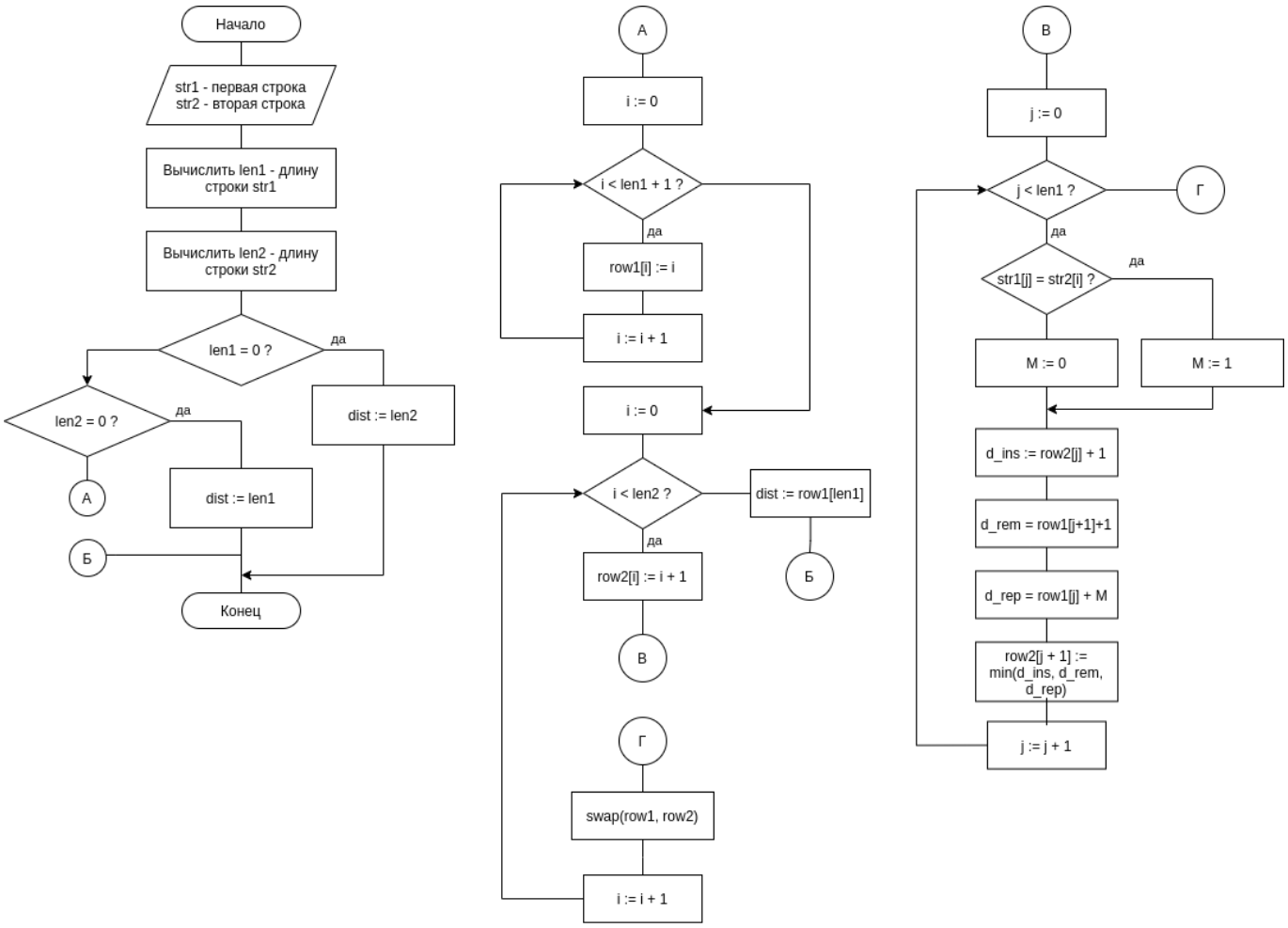

Рисунок 2.2 — Схема матричного алгоритма нахождения расстояния Левенштейна с кешированием двух строк

3 Технологический раздел

В данном разделе будут выбраны средства реализации ПО, представлен листинг кода и проведён теоритический анализ максимальной затрачиваемой памяти.

3.1 Требования к ПО

1) Штуки

3.2 Средства реализации

Для реализации программы нахождения расстояния Левенштейна был выбран язык программирования C++ [**?**]. Выбор данного языка программирования обусловлен имеющимся опытом работы с ним, а также его быстродействием.

3.3 Реализации алгоритмов

```
#include <iostream>
#include <fstream>
#include <string>
#include <tuple>
#include "levenshtein.hpp"
#include "command.hpp"
#include "utils.hpp"

#define MIN_LENGTH 0
#define MAX_LENGTH 100
#define LENGTH_STEP 20

#define MAX_ITERATIONS 1000


void test_func(std::ostream& stream, const char* func_name, ed_dist_fn_t func)
{
    for (unsigned int length = MIN_LENGTH; length <= MAX_LENGTH; length +=
        LENGTH_STEP)
    {
        std::string str1 = random_string(length);
        std::string str2 = random_string(length);

        auto time = time_it(func, str1.c_str(), str2.c_str(), MAX_ITERATIONS);
        stream << func_name << ',' << length << ',' << time << std::endl;
    }
}

std::tuple<std::string, std::string> input_strings()
{
```

```
30      std::string str1, str2;
31
32      std::cout << "Введите первую строку: ";
33      std::getline(std::cin, str1);
34
35      std::cout << "Введите вторую строку: ";
36      std::getline(std::cin, str2);
37
38      return { str1, str2 };
39  }
40
41  void do_user_input_command()
42  {
43      auto strs = input_strings();
44      auto str1 = std::get<0>(strs);
45      auto str2 = std::get<1>(strs);
46
47      std::cout << "lev_rec: " << levenshtein_row(str1.c_str(), str2.c_str()) <<
            std::endl;
48      std::cout << "lev_row: " << levenshtein_row(str1.c_str(), str2.c_str()) <<
            std::endl;
49      std::cout << "lev_mat: " << levenshtein_mat(str1.c_str(), str2.c_str()) <<
            std::endl;
50      std::cout << "dam_lev: " << damerau_levenshtein_mat(str1.c_str(),
            str2.c_str()) << std::endl;
51  }
52
53  void auto_test(std::ostream& stream)
54  {
55      test_func(stream, "row", levenshtein_row);
56      test_func(stream, "mat", levenshtein_mat);
57      test_func(stream, "recur", levenshtein_recur);
58      test_func(stream, "damerau", damerau_levenshtein_mat);
59  }
60
61  void do_auto_test_command()
62  {
63      std::cout << "Введите имя выходного файла: ";
64
65      std::string filename;
66      std::getline(std::cin, filename);
67
68      if (filename.empty())
69          auto_test(std::cout);
70      else
71      {
72          std::ofstream stream(filename);
```

```
73          auto_test(stream);
74      }
75  }
76
77
78  int main(int argc, const char* argv[])
79  {
80      while (true)
81      {
82          Command cmd = input_command();
83
84          switch (cmd)
85          {
86          case Command::USER_INPUT:
87              do_user_input_command();
88              break;
89
90          case Command::AUTO_TEST:
91              do_auto_test_command();
92              break;
93
94          default:
95              return 0;
96              break;
97          }
98      }
99
100     return 0;
101 }
```

Листинг 3.1 — Функция реализующая алгоритм Левенштейна с мемоизацией

```
1  unsigned int levenshtein_row(const char* str1, const char* str2)
2  {
3      unsigned int len1 = strlen(str1);
4      unsigned int len2 = strlen(str2);
5
6      if (len1 == 0)
7      return len2;
8
9      if (len2 == 0)
10     return len1;
11
12     unsigned int dist = std::numeric_limits<unsigned int>::max();
13     unsigned int* row1 = static_cast<unsigned int*>(malloc((len1 + 1) *
           sizeof(unsigned int)));
14     if (row1)
```

```
15      {
16          unsigned int* row2 = static_cast<unsigned int*>(malloc((len1 + 1) *
                sizeof(unsigned int)));
17          if (row2)
18          {
19              for (unsigned int i = 0; i < len1 + 1; i++)
20              row1[i] = i;
21
22              for (unsigned int i = 0; i < len2; i++)
23              {
24                  row2[0] = i + 1;
25                  for (unsigned int j = 0; j < len1; j++)
26                  {
27                      bool eq = str1[j] == str2[i];
28
29                      unsigned int dis_ins = row2[j] + 1;
30                      unsigned int dis_rem = row1[j + 1] + 1;
31                      unsigned int dis_rep = eq ? row1[j] : row1[j] + 1;
32
33                      row2[j + 1] = MIN3(dis_ins, dis_rem, dis_rep);
34                  }
35
36                  std::swap(row1, row2);
37              }
38
39              dist = row1[len1];
40              free(row2);
41          }
42
43          free(row1);
44      }
45
46      return dist;
47  }
```

Листинг 3.2 — Функция реализующая матричный алгоритм Левенштейна

```
1   unsigned int levenshtein_mat(const char* str1, const char* str2)
2   {
3       unsigned int len1 = strlen(str1);
4       unsigned int len2 = strlen(str2);
5
6       if (len1 == 0)
7       return len2;
8
9       if (len2 == 0)
10      return len1;
```

```
11
12      unsigned int dist = std::numeric_limits<unsigned int>::max();
13      unsigned int* mat = static_cast<unsigned int*>(malloc((len1 + 1) * (len2 + 1) *
            sizeof(unsigned int)));
14      if (mat)
15      {
16          for (unsigned int i = 1; i < len1 + 1; i++)
17          mat[i * (len2 + 1)] = i;
18          for (unsigned int i = 0; i < len2 + 1; i++)
19          mat[i] = i;
20
21          for (unsigned int i = 0; i < len1; i++)
22          {
23              for (unsigned int j = 0; j < len2; j++)
24              {
25                  bool eq = str1[i] == str2[j];
26
27                  unsigned int dis_ins = mat[(i + 1) * (len2 + 1) + j] + 1;
28                  unsigned int dis_rem = mat[i * (len2 + 1) + j + 1] + 1;
29                  unsigned int dis_rep = mat[i * (len2 + 1) + j] + (eq ? 0 : 1);
30
31                  mat[(i + 1) * (len2 + 1) + (j + 1)] = MIN3(dis_ins, dis_rem,
                        dis_rep);
32              }
33          }
34
35          dist = mat[(len1 + 1) * (len2 + 1) - 1];
36          free(mat);
37      }
38
39      return dist;
40  }
```

Листинг 3.3 — Функция реализующая рекурсивный алгоритм Левенштейна

```
1   static const char* __lev_rec_str1;
2   static const char* __lev_rec_str2;
3   static unsigned int __lev_rec_proc(unsigned int len1, unsigned int len2)
4   {
5       if (len1 == 0)
6       return len2;
7       else if (len2 == 0)
8       return len1;
9
10      bool eq = __lev_rec_str1[len1 - 1] == __lev_rec_str2[len2 - 1];
11
12      unsigned int dis_ins = __lev_rec_proc(len1 - 1, len2) + 1;
```

```
13      unsigned int dis_rem = __lev_rec_proc(len1, len2 - 1) + 1;
14      unsigned int dis_rep = __lev_rec_proc(len1 - 1, len2 - 1) + (eq ? 0 : 1);
15
16      return MIN3(dis_ins, dis_rem, dis_rep);
17  }
18
19  unsigned int levenshtein_recur(const char* str1, const char* str2)
20  {
21      unsigned int len1 = strlen(str1);
22      unsigned int len2 = strlen(str2);
23
24      __lev_rec_str1 = str1;
25      __lev_rec_str2 = str2;
26
27      return __lev_rec_proc(len1, len2);
28  }
```

Листинг 3.4 — Функция реализующая алгоритм Дамерау-Левенштейна

```
1   unsigned int damerau_levenshtein_mat(const char* str1, const char* str2)
2   {
3       unsigned int len1 = strlen(str1);
4       unsigned int len2 = strlen(str2);
5       unsigned int dist = std::numeric_limits<unsigned int>::max();
6
7       unsigned int* mat = static_cast<unsigned int*>(malloc((len1 + 1) * (len2 + 1) *
            sizeof(unsigned int)));
8       if (mat)
9       {
10          for (unsigned int i = 1; i < len1 + 1; i++)
11          mat[i * (len2 + 1)] = i;
12          for (unsigned int i = 0; i < len2 + 1; i++)
13          mat[i] = i;
14
15          for (unsigned int i = 0; i < len1; i++)
16          {
17              for (unsigned int j = 0; j < len2; j++)
18              {
19                  bool eq = str1[i] == str2[j];
20
21                  unsigned int dis_ins = mat[(i + 1) * (len2 + 1) + j] + 1;
22                  unsigned int dis_rem = mat[i * (len2 + 1) + j + 1] + 1;
23                  unsigned int dis_rep = mat[i * (len2 + 1) + j] + (eq ? 0 : 1);
24
25                  unsigned int dis = MIN3(dis_ins, dis_rem, dis_rep);
26
27                  if (i > 0 && j > 0)
```

```
28                  {
29                      int rev_index = (i - 1) * (len2 + 1) + j - 1;
30                      dis = MIN2(dis, mat[rev_index]);
31                  }
32
33                  mat[(i + 1) * (len2 + 1) + (j + 1)] = dis;
34              }
35          }
36
37          dist = mat[(len1 + 1) * (len2 + 1) - 1];
38          free(mat);
39      }
40
41      return dist;
42  }
```

3.4 Тестовые данные

Таблица 3.1 — Таблица с тестовыми данными

Входные данные	**Выходные данные**
.	.

4 Экспериментальный раздел

В данном разделе будут проведены эксперименты для проведения сравнительного анализа алгоритмов по затрачиваемому процессорному времени[**?**] и максимальной используемой памяти.

4.1 Графики

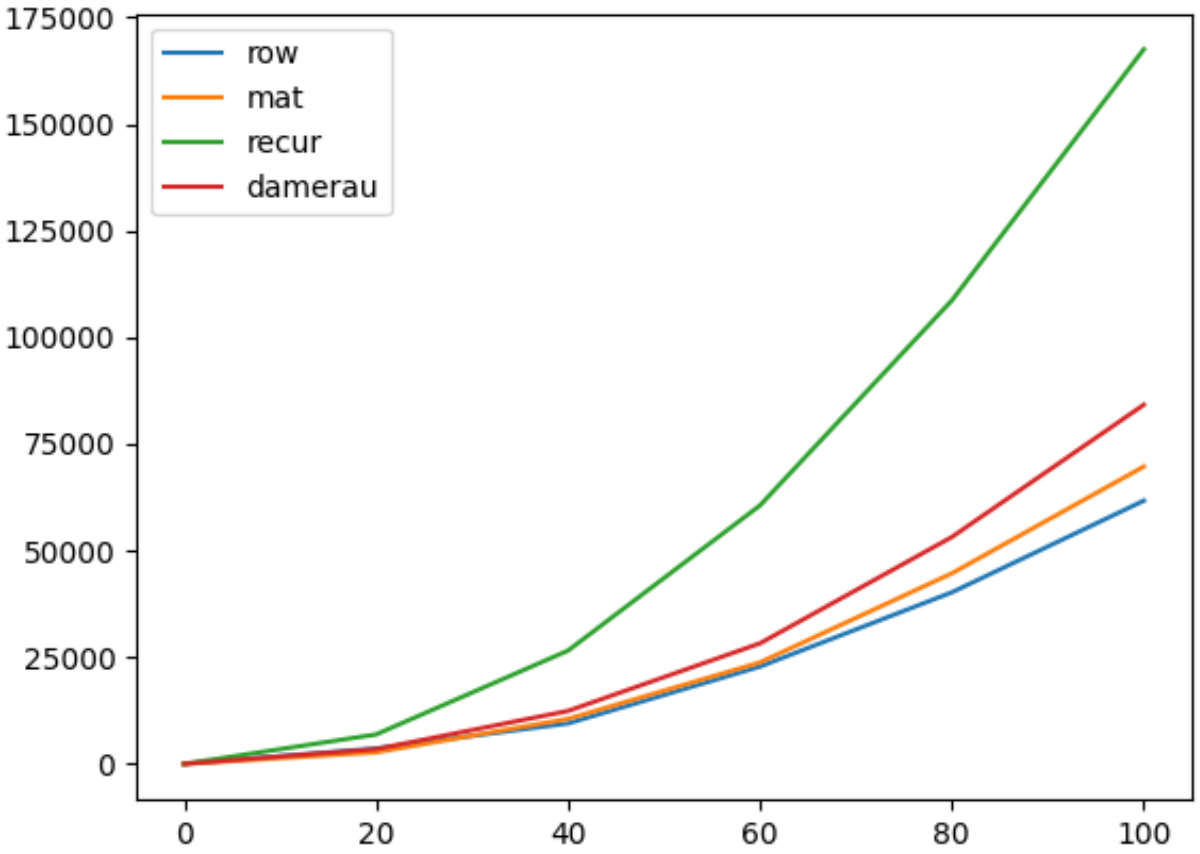

Рисунок 4.1 — Графики зависимостей времени

4.2 Вывод

В данном разделе были поставлены эксперименты ...

Заключение

В ходе работы