



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 дисциплине "Анализ алгоритмов"

Тема Редакционное расстояние

Студент Варламова Е. А.

Группа ИУ7-51Б

Оценка (баллы) _____

Преподаватель: Волкова Л.Л.

Оглавление

Введение	3
1 Аналитическая часть	4
1.1 Расстояние Левенштейна	4
1.2 Расстояние Дамерау-Левенштейна	4
1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы	5
1.4 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы	5
1.5 Рекурсивный алгоритм нахождения расстояния Левенштейна	5
1.6 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы	5
1.7 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна	5
1.8 Вывод	5
2 Конструкторская часть	6
2.1 Схемы алгоритмов	6
2.2 Вывод	9
3 Технологическая часть	10
3.1 Средства реализации	10
3.2 Реализация алгоритмов	10
3.3 Тестирование	12
3.4 Вывод	12
4 Исследовательская часть	13
4.1 Технические характеристики	13
4.2 Время выполнения реализаций алгоритмов	13
4.3 Оценка затрачиваемой памяти	14
4.3.1 Рекурсивный алгоритм нахождения расстояния Левенштейна	14
4.3.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы	14
4.3.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы	14
4.3.4 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна	14
4.4 Вывод	15
Заключение	16
Литература	17

Введение

Динамическое программирование - это форма вычислений, при которой следующий член вычисляется на основе предыдущего. Простейшим примером применения является вычисление чисел Фибоначчи. Кроме того, динамическое программирование может применяться и в более сложных задачах таких, как преобразование строк из одной в другую. В этом случае задача сводится к вычислению расстояния Левенштейна (редакционного расстояния) - минимального количества операций вставки, удаления символа или замены символа один на другой, необходимых для преобразования одной строки в другую. Расстояние Левенштейна применяется в:

- компьютерной лингвистике для устранения ошибок в набираемом тексте;
- в биоинформатике для сравнения генов.

Поэтому **целью** данной работы является получение навыка динамического программирования на примере реализации алгоритмов редакционного расстояния.

Для достижения поставленной цели необходимо решить следующие **задачи**:

- изучить алгоритмы расчета редакционного расстояния;
- реализовать алгоритмы подсчета редакционного расстояния;
- протестировать реализованные алгоритмы;
- провести сравнительный анализ реализаций алгоритмов по затраченному процессорному времени и памяти.

1 | Аналитическая часть

В данном разделе определяются расстояния Левенштейна и Дамерау-Левенштейна, а также рассматриваются различные алгоритмы вычисления указанных расстояний.

1.1 Расстояние Левенштейна

Для вычисления редакционного расстояния вводятся следующие цены операций:

- замена одного символа на другой - 1;
- вставка символа - 1;
- удаление символа - 1.

С учетом этого вводится рекурсивная формула для вычисления расстояния Левенштейна:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ \min\{ \\ \quad D(s1[1..i], s2[1..j-1]) + 1 \\ \quad D(s1[1..i-1], s2[1..j]) + 1 \\ \quad D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, & s1[i] = s2[j] \\ 1, & \text{иначе} \end{cases} \\ \} \end{cases} \quad i > 0, j > 0 \quad (1.1)$$

1.2 Расстояние Дамерау-Левенштейна

Дамерау дополнил определение расстояния Левенштейна еще одной операцией, а именно операцией перестановки двух букв местами. Расстояние Дамерау-Левенштейна вычисляется по следующей формуле:

$$D(s1[1..i], s2[1..j]) = \begin{cases} 0, & i = 0, j = 0 \\ i, & i > 0, j = 0 \\ j, & i = 0, j > 0 \\ \min\{ \\ \quad D(s1[1..i], s2[1..j-1]) + 1 \\ \quad D(s1[1..i-1], s2[1..j]) + 1 \\ \quad D(s1[1..i-1], s2[1..j-1]) + \begin{cases} 0, & s1[i] = s2[j] \\ 1, & \text{иначе} \end{cases} \\ \quad D(s1[1..i-2], s2[1..j-2]) + 1, & \text{если } i > 1, j > 1, s1[i] = s2[j-1], s1[i-1] = s2[j] \end{cases} \end{cases} \quad (1.2)$$

1.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы

Данный алгоритм использует для решения задачи матрицу размером $(m+1)*(n+1)$, где m и n - длины двух строк, одну из которых необходимо преобразовать к другой. На каждом шаге работы алгоритма заполняется одна клетка матрицы в соответствии с формулой 1.1. По окончании алгоритма результат будет находиться в последней заполненной клетке.

1.4 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы

Данный алгоритм является модификацией предыдущего. Очевидно, что на каждом шаге алгоритма используются значения из текущей и предыдущей строки матрицы, поэтому достаточно хранить только их, а не всю матрицу.

1.5 Рекурсивный алгоритм нахождения расстояния Левенштейна

Данный алгоритм использует для решения формулу 1.1, однако в отличие от предыдущих является рекурсивным, а значит, для хранения промежуточных результатов используется стек. Кроме того, при этом подходе возникает проблема повторных вычислений, так как функция $D(s1[1..i], s2[1..j])$ будет выполняться несколько раз в разных ветвях дерева.

1.6 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы

Данный алгоритм решает проблему повторных вычислений простого рекурсивного алгоритма. При данном подходе вводится матрица размером $(m+1) * (n+1)$, содержащая уже вычисленные промежуточные результаты. Изначально матрица инициализируется значениями, которые заведомо не могут быть получены в результате вычислений, например, -1.

1.7 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Данный алгоритм использует для решения формулу 1.2 и является рекурсивным, а значит, для хранения промежуточных результатов используется стек. Кроме того, при этом подходе возникает проблема повторных вычислений, так как функция $D(s1[1..i], s2[1..j])$ будет выполняться несколько раз в разных ветвях дерева.

1.8 Вывод

В данном разделе были даны определения расстояний Левенштейна и Дамерау-Левенштейна, а также рассмотрены 5 алгоритмов вычисления указанных расстояний.

2 | Конструкторская часть

В данном разделе разрабатываются схемы алгоритмов на основе их описания, приведённого в аналитическом разделе.

2.1 Схемы алгоритмов

На рисунках 2.1, 2.2, 2.3 и 2.4 показаны схемы алгоритма рекурсивного Левенштейна, нерекурсивного алгоритма Левенштейна с кэшем в виде двух строк матрицы, рекурсивного алгоритма Дамерау-Левенштейна и рекурсивного алгоритма с кэшем в виде матрицы соответственно.

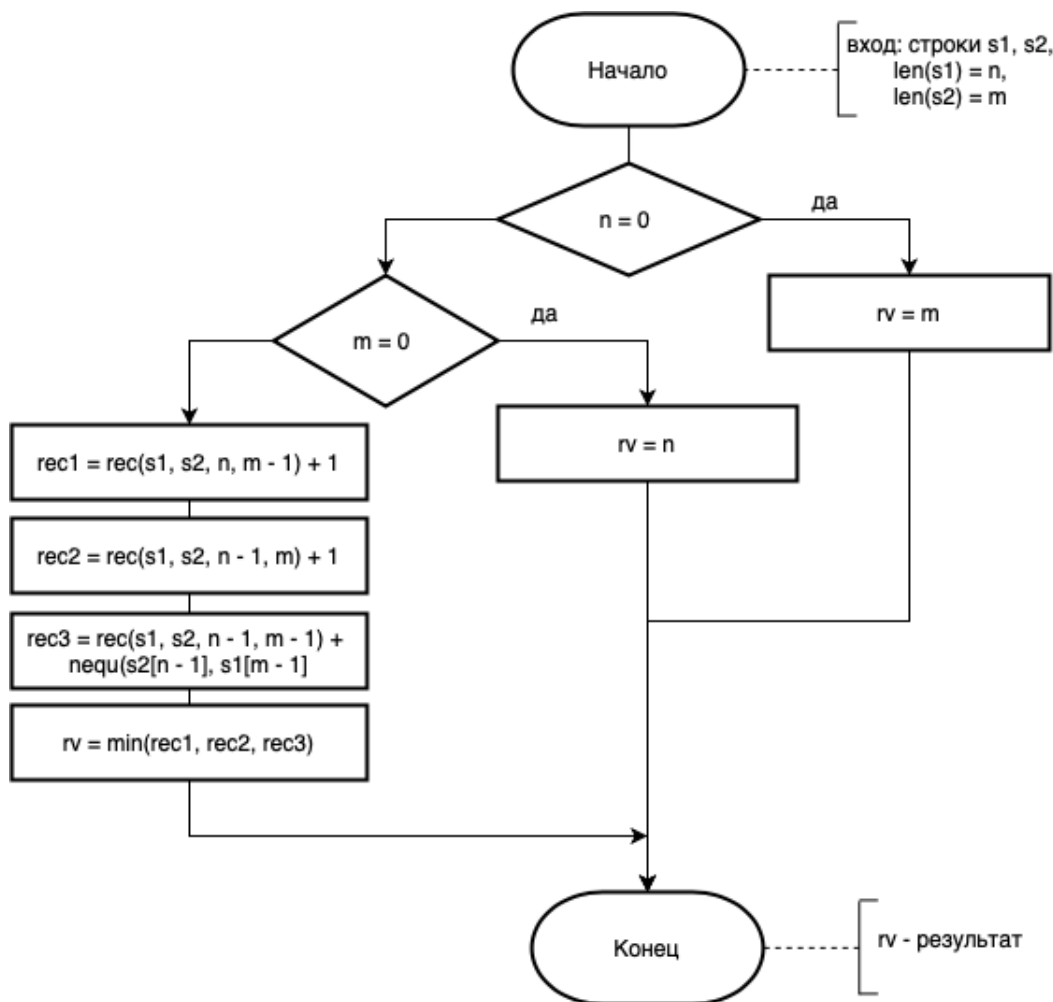


Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

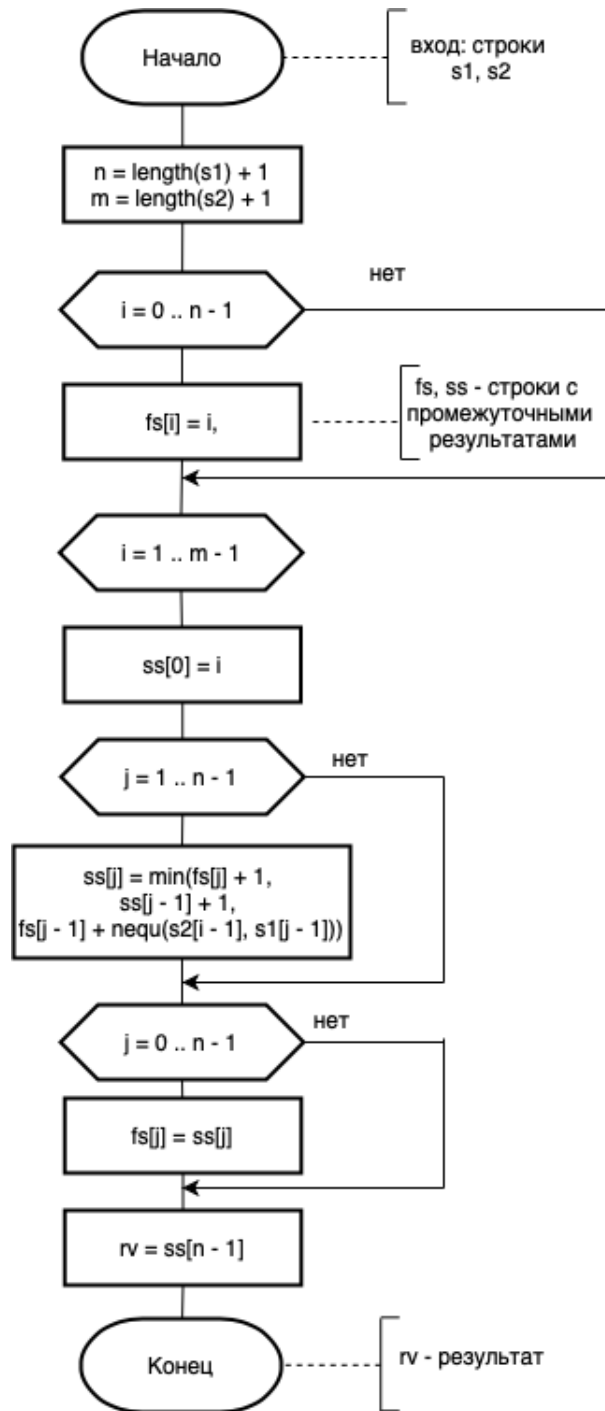


Рис. 2.2: Схема нерекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы

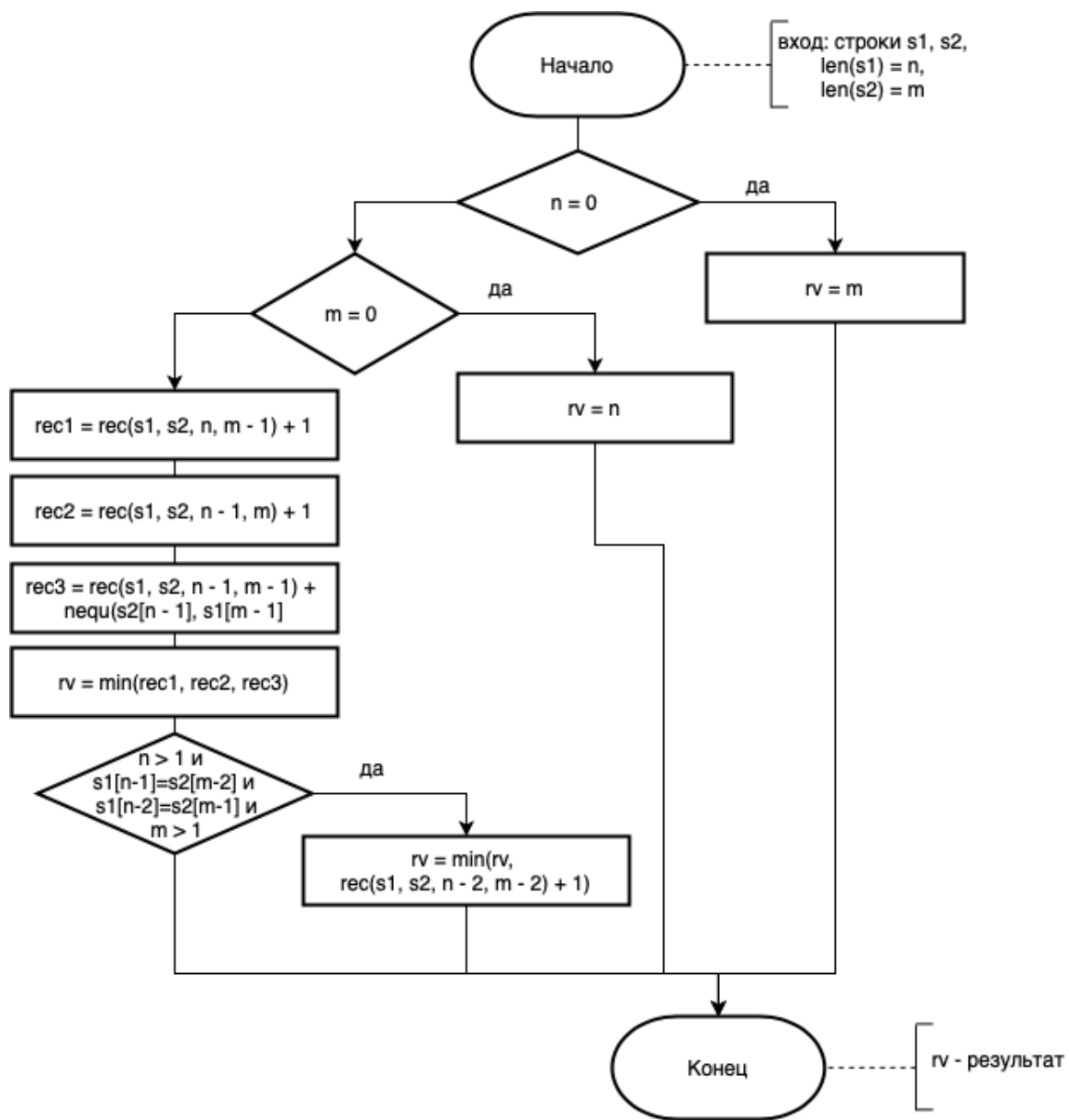


Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

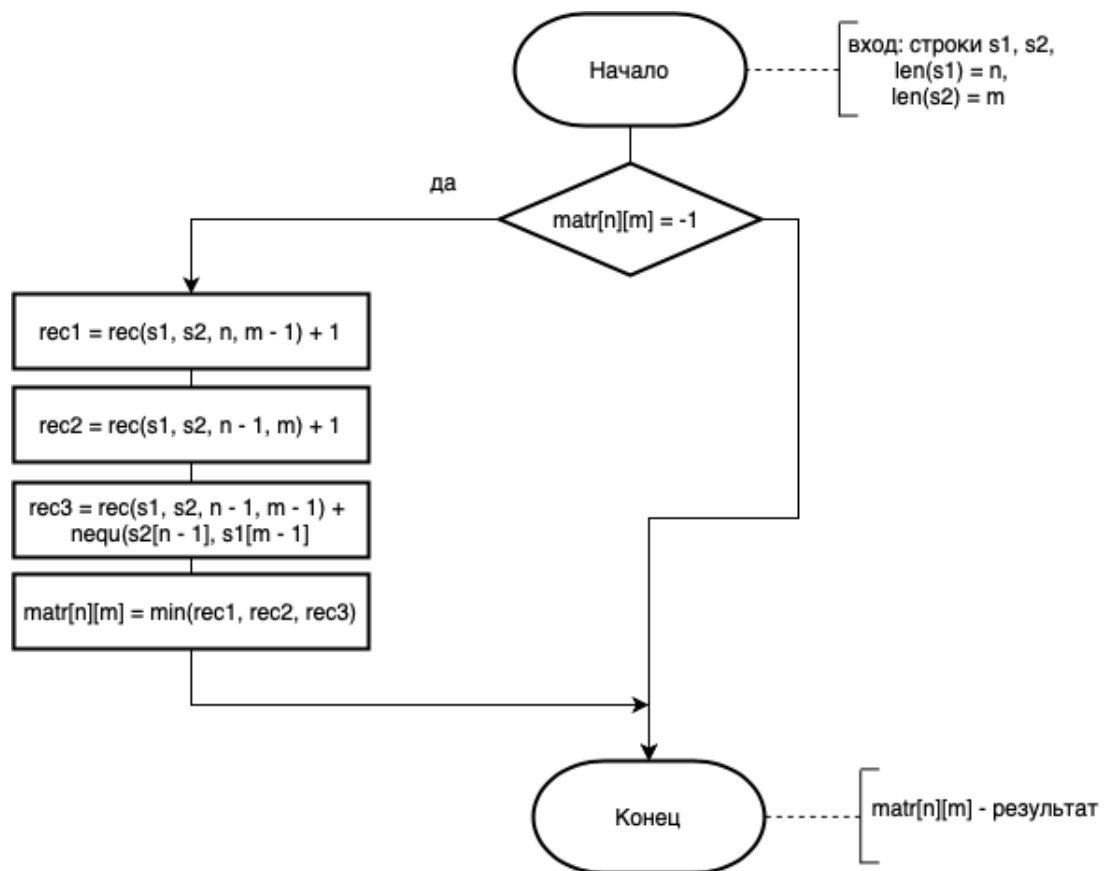


Рис. 2.4: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде матрицы

2.2 Вывод

В данном разделе были построены схемы 4-х алгоритмов нахождения редакционного расстояния на основе их описания, приведённого в аналитической части.

3 | Технологическая часть

В данном разделе приводится реализация алгоритмов, схемы которых были разработаны в конструкторской части. Кроме того, обосновывается выбор технологического стека и проводится тестирование реализованных алгоритмов.

3.1 Средства реализации

В качестве языка программирования был выбран C++ из-за его быстродействия, а среды разработки – CLion. Время работы алгоритмов было замерено с помощью time.h, функции clock, которая измеряет процессорное время [1].

3.2 Реализация алгоритмов

В листингах 3.1 - 3.4 приведена реализации алгоритмов описанных в 2.1.

Листинг 3.1: Функция для рекурсивного нахождения расстояния Левенштейна с кэшем в виде матрицы

```
1 static int _recLevCache(const char * s1, const char * s2, int n, int m, int ** matr)
2 {
3     if (matr[n][m] == -1)
4     {
5         matr[n][m] = minimum(3,
6                               _recLevCache(s1, s2, n, m - 1, matr) + 1,
7                               _recLevCache(s1, s2, n - 1, m, matr) + 1,
8                               _recLevCache(s1, s2, n - 1, m - 1, matr) + (s1[n - 1] != s2[m
9                               - 1]));
10    }
11    return matr[n][m];
12 }
13 int recLevCache(const char * s1, const char * s2)
14 {
15     int n = strlen(s1);
16     int m = strlen(s2);
17
18     int **matr = alloc_matrix(n + 1, m + 1);
19
20     for (int i = 0; i < n + 1; i++)
21         matr[i][0] = i;
22     for (int i = 0; i < m + 1; i++)
23         matr[0][i] = i;
24     for (int i = 1; i < n + 1; i++)
25         for (int j = 1; j < m + 1; j++)
26             matr[i][j] = -1;
27     int rv = _recLevCache(s1, s2, n, m, matr);
28     free_matrix(matr, n, m);
29     return rv;
30 }
```

Листинг 3.2: Функция для нерекурсивного нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы

```

1 int levCache(const char * s1, const char * s2)
2 {
3     size_t n = strlen(s1) + 1;
4     size_t m = strlen(s2) + 1;
5
6     int *fs = (int *)malloc(n * sizeof(int));
7     int *ss = (int *)malloc(n * sizeof(int));
8     for (size_t i = 0; i < n; i++)
9         fs[i] = i;
10
11     for (size_t i = 1; i < m; i++)
12     {
13         ss[0] = i;
14         for (size_t j = 1; j < n; j++)
15             ss[j] = minimum(3,
16                             fs[j] + 1,
17                             ss[j - 1] + 1,
18                             fs[j - 1] + (s2[i - 1] != s1[j - 1]));
19         for (size_t j = 0; j < n; j++) {
20             fs[j] = ss[j];
21         }
22     }
23
24     int rv = ss[n - 1];
25     free(fs);
26     free(ss);
27     return rv;
28 }

```

Листинг 3.3: Функция для рекурсивного нахождения расстояния Дамерау-Левенштейна

```

1 int _recDamLev(const char * s1, const char * s2, int n, int m)
2 {
3     if (n == 0)
4         return m;
5     if (m == 0)
6         return n;
7     int min = minimum(3,
8                     _recDamLev(s1, s2, n, m - 1) + 1,
9                     _recDamLev(s1, s2, n - 1, m) + 1,
10                    _recDamLev(s1, s2, n - 1, m - 1) + (s1[n - 1] != s2[m - 1]));
11
12     if (n > 1 && m > 1 && s1[n - 1] == s2[m - 2] && s1[n - 2] == s2[m - 1])
13         min = minimum(2, min, _recDamLev(s1, s2, n - 2, m - 2) + 1);
14     return min;
15 }
16
17 int recDamLev(const char * s1, const char * s2)
18 {
19     return _recDamLev(s1, s2, strlen(s1), strlen(s2));
20 }

```

Листинг 3.4: Функция для рекурсивного нахождения расстояния Левенштейна

```

1 int _recLev(const char * s1, const char * s2, int n, int m)
2 {
3     if (n == 0)
4         return m;
5     if (m == 0)
6         return n;
7
8     return minimum(3,
9         _recLev(s1, s2, n, m - 1) + 1,
10        _recLev(s1, s2, n - 1, m) + 1,
11        _recLev(s1, s2, n - 1, m - 1) + (s1[n - 1] != s2[m - 1]));
12 }
13
14 int recLev(const char * s1, const char * s2)
15 {
16     return _recLev(s1, s2, strlen(s1), strlen(s2));
17 }

```

3.3 Тестирование

В таблице 3.1 и 3.2 приведены тесты для функций сортировки.

Таблица 3.1: Тестирование алгоритмов нахождения расстояния Левенштейна

Входные строки	Результат	Ожидаемый результат
<i>ckat, kot</i>	2	2
<i>abc, defg</i>	4	4
<i>abcd, abcd</i>	0	0

Таблица 3.2: Тестирование алгоритмов нахождения расстояния Дameraу-Левенштейна

Входные строки	Результат	Ожидаемый результат
<i>ckat, kot</i>	2	2
<i>abc, defg</i>	4	4
<i>abcd, abcd</i>	0	0
<i>abcd, badc</i>	2	2

Все тесты пройдены успешно.

3.4 Вывод

В данном разделе были реализованы 4 алгоритма нахождения редакционного расстояния с помощью выбранных средств разработки. Кроме того, реализованные алгоритмы были протестированы.

4 | Исследовательская часть

В данном разделе проводится сравнительный анализ реализованных алгоритмов по процессорному времени и по затарчиваемой памяти.

4.1 Технические характеристики

Все нижеприведенные замеры времени проведены на процессоре: Intel Core i5, 1,4 GHz, 4-ядерный.

4.2 Время выполнения реализаций алгоритмов

Для сравнительного анализа времени выполнения реализаций алгоритмов проведен эксперимент. Для замеров были сформированы строки, с суммарной длиной, варьирующейся от 6 до 24 включительно с шагом 2.

Время измерялось 10 раз для каждой пары строк, после усреднялось. Время на графиках (рис. 4.1) представлено в миллисекундах.

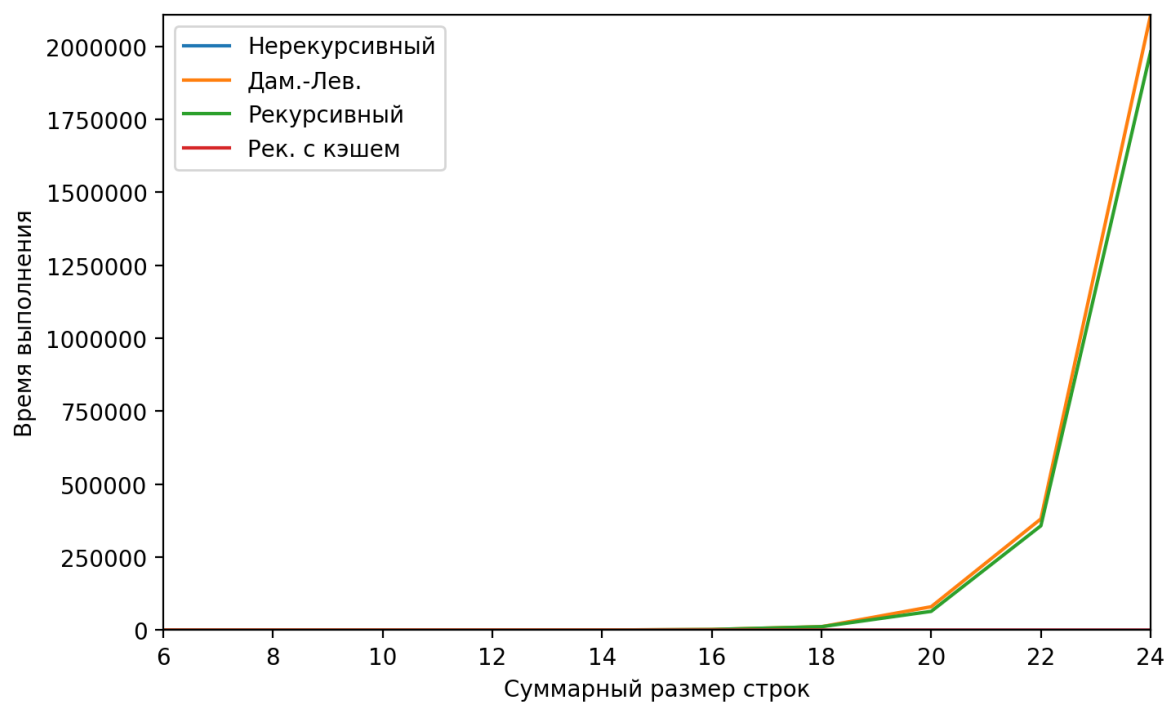


Рис. 4.1: Зависимость времени от суммарной длины строк

Время для всех реализаций представлено в таблице 4.1.

Таблица 4.1: Зависимость затрачиваемого процессорного времени от суммарной длины строк для 4 алгоритмов

Суммарная длина строк	Нерек.	Дам.-Лев.	Рек.	Рек. с кэшем
6	2.70	1.20	1.20	2.20
8	1.00	4.80	3.30	2.20
10	1.30	16.70	15.60	2.20
12	1.10	87.30	86.90	2.80
14	1.50	475.10	451.30	4.80
16	2.00	2652.20	2650.30	4.70
18	3.00	14573.00	16364.90	6.20
20	2.10	73969.10	68309.80	11.30
22	2.60	375357.00	348323.20	5.10
24	4.70	2033420.40	1913548.70	6.00

4.3 Оценка затрачиваемой памяти

В последующих подразделах n - длина первой строки, m - длина второй строки.

4.3.1 Рекурсивный алгоритм нахождения расстояния Левенштейна

Рассмотрим рекурсивный алгоритм нахождения расстояния Левенштейна. Максимальная высота дерева будет равна сумме длин двух строк.

На стеке будет выделено максимум: $40 * (n + m) +$ байт.

На куче память не выделяется.

Итого будет выделено максимум: $40 * (n + m) +$ байт.

4.3.2 Рекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде матрицы

В данном алгоритме дерево будет такой же высоты как и в 4.3.1. Но в данной реализации также выделяется матрица размером $n * m$ на куче.

На стеке будет выделено максимум: $40 * (n + m)$ байт.

На куче будет выделено: $(n * m) * 4$ байт.

Итого будет выделено максимум: $40 * (n + m) + (n * m) * 4$ байт.

4.3.3 Нерекурсивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы

В данной реализации выделяется 2 строки матрицы длиной m . На стеке всегда будет выделено одно и то же количество памяти, так как реализация нерекурсивная.

На куче будет выделено: $m * 2 * 4$ байт.

Итого будет выделено максимум: $8 * m +$ байт.

4.3.4 Рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна

Данная реализация будет занимать столько же памяти сколько и реализация рекурсивного алгоритма нахождения расстояния Левенштейна (4.3.1), так как максимальная высота дерева будет одинаковой в обоих случаях, реализации используют одинаковое количество локальных переменных и переменных, поступающих на стек в виде аргументов.

4.4 Вывод

В результате эксперимента можно сделать вывод, что нерекурсивная реализация алгоритма нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы является самой эффективной по процессорному времени и затрачиваемой памяти. Рекурсивный алгоритм с кэшем в виде матрицы менее эффективен по памяти, чем нерекурсивный, однако значительно быстрее, чем обычный рекурсивный (и рекурсивный Дамерау-Левенштейна).

Заключение

В рамках данной лабораторной работы были решены следующие задачи:

- изучены 4 алгоритма расчета редакционного расстояния;
- реализованы 4 алгоритма расчета редакционного расстояния;
- протестированы реализованные алгоритмы;
- проведён сравнительный анализ алгоритмов по затраченному процессорному времени и памяти.

Поставленная цель, состоящая в получении навыка динамического программирования на примере реализации алгоритмов редакционного расстояния, достигнута.

Литература

1. Стандартная функция `clock()`, измеряющая процессорное время [Электронный ресурс]. Режим доступа: <https://en.cppreference.com/w/cpp/chrono/c/clock>. Дата обращения: 24.09.2021