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.

Москва — 2022 г.

Оглавление

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 проебразование строк из одной в другую. В этом случае задача сводится к вычислению расстояния Левенштейна (редакционного расстояния) - минимального количества опреаций 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 бионформатике 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 учтом этого вводится рекурсивная формула для вычисления расстояннния Левенштейна:

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

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:

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

1.3 Нерекусивный 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 Нерекусивный алгоритм нахождения расстояния Левенштейна с кэшем в виде 2 строк матрицы

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

1.5 Рекусивный алгоритм нахождения расстояния Левенштейна

Данный алгоритм использует для решения формулу 1.1, однако в отличие от прерыдущих 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 Рекусивный 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 Рекусивный 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.

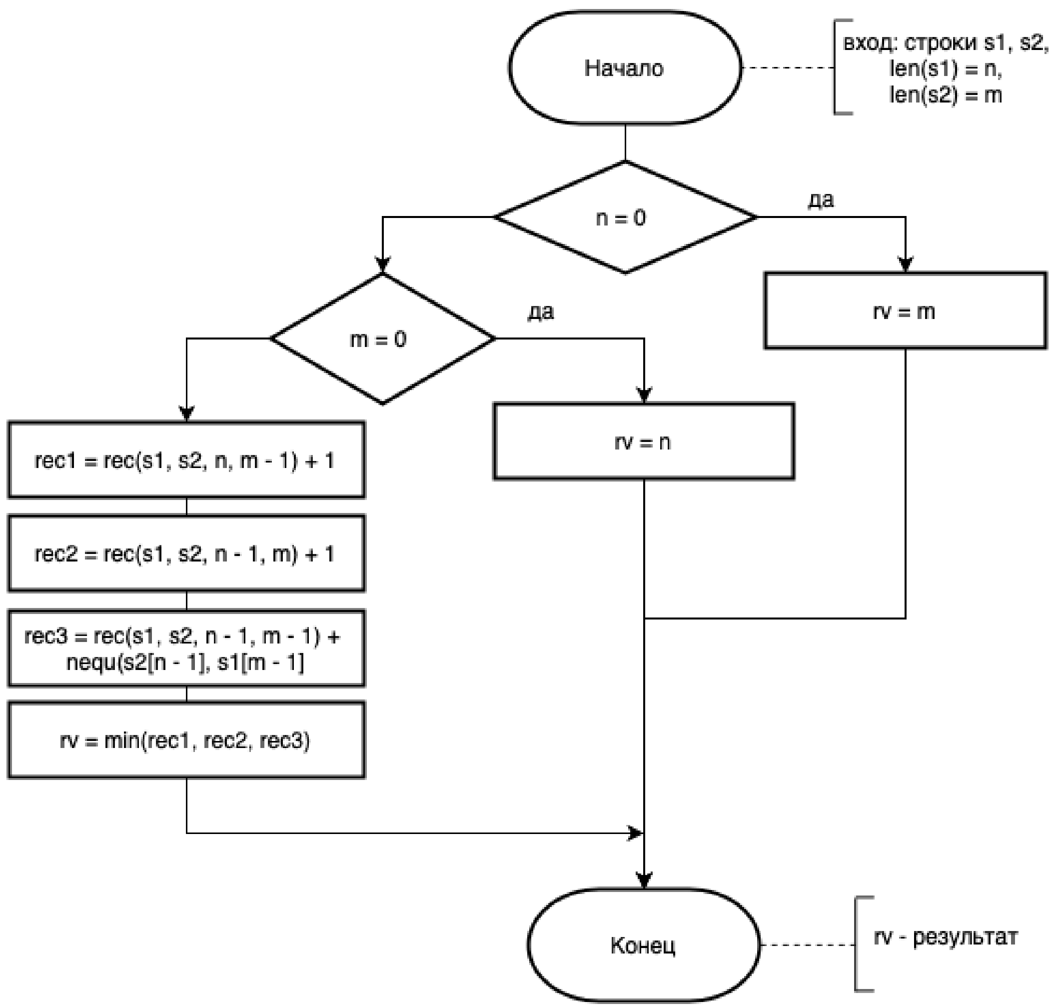

Рис. 2.1: Схема рекурсивного алгоритма нахождения расстояния Левенштейна

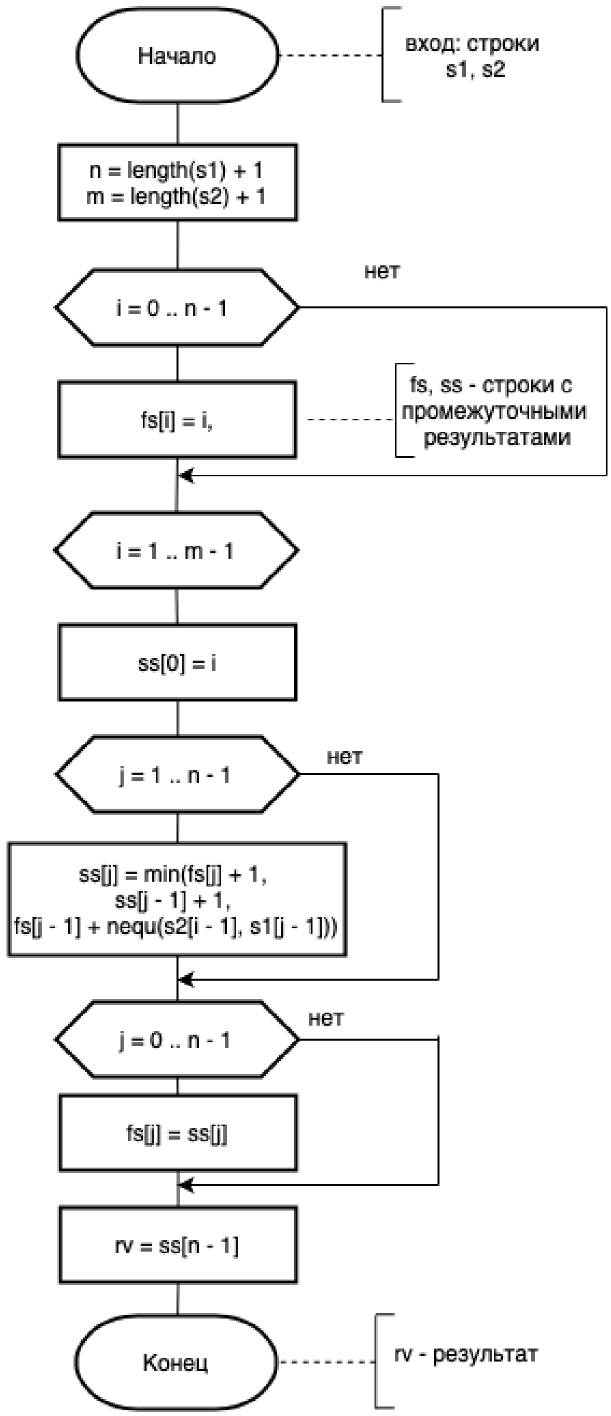

Рис. 2.2: Схема нерекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде двух строк матрицы

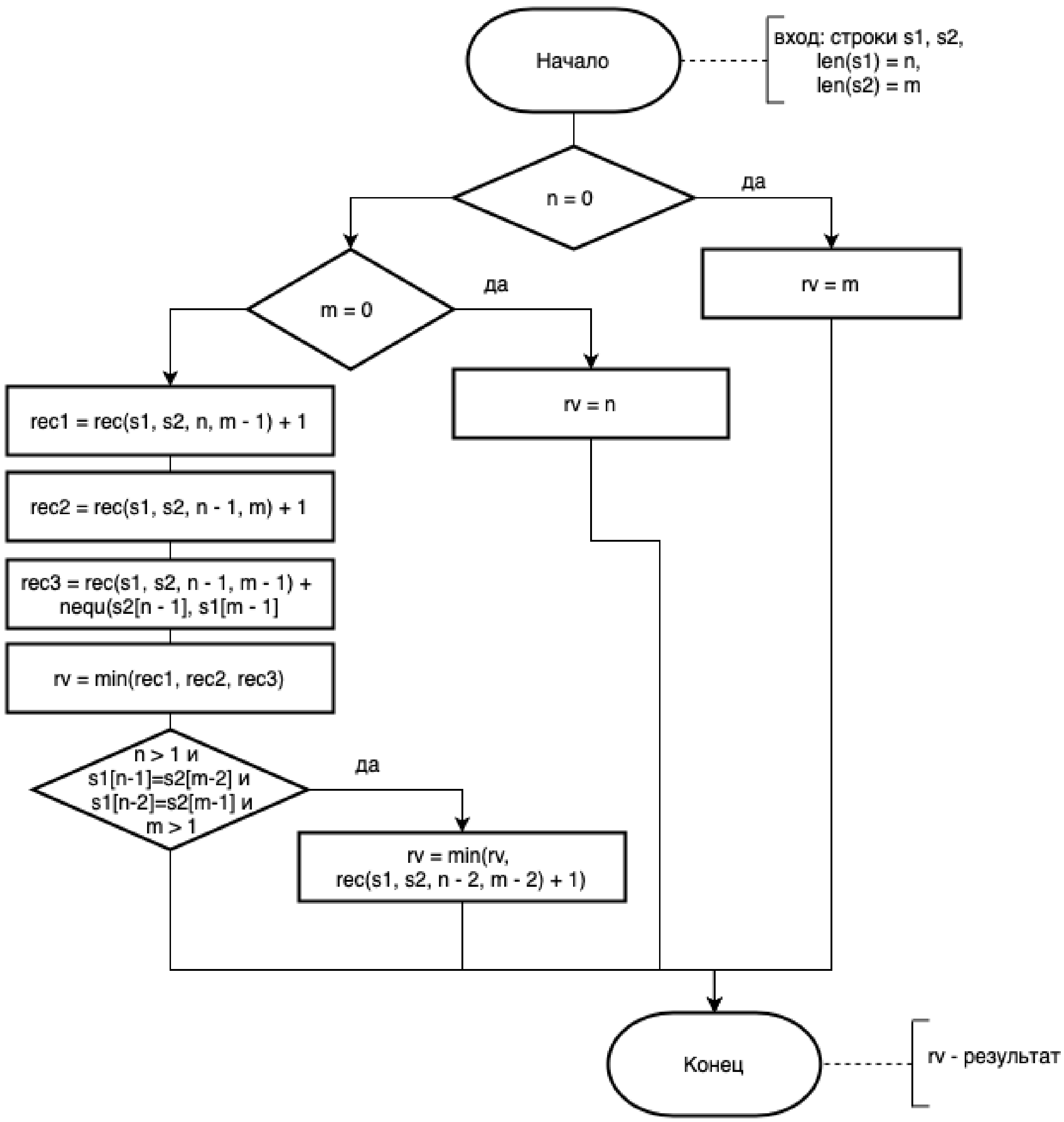

Рис. 2.3: Схема рекурсивного алгоритма нахождения расстояния Дамерау-Левенштейна

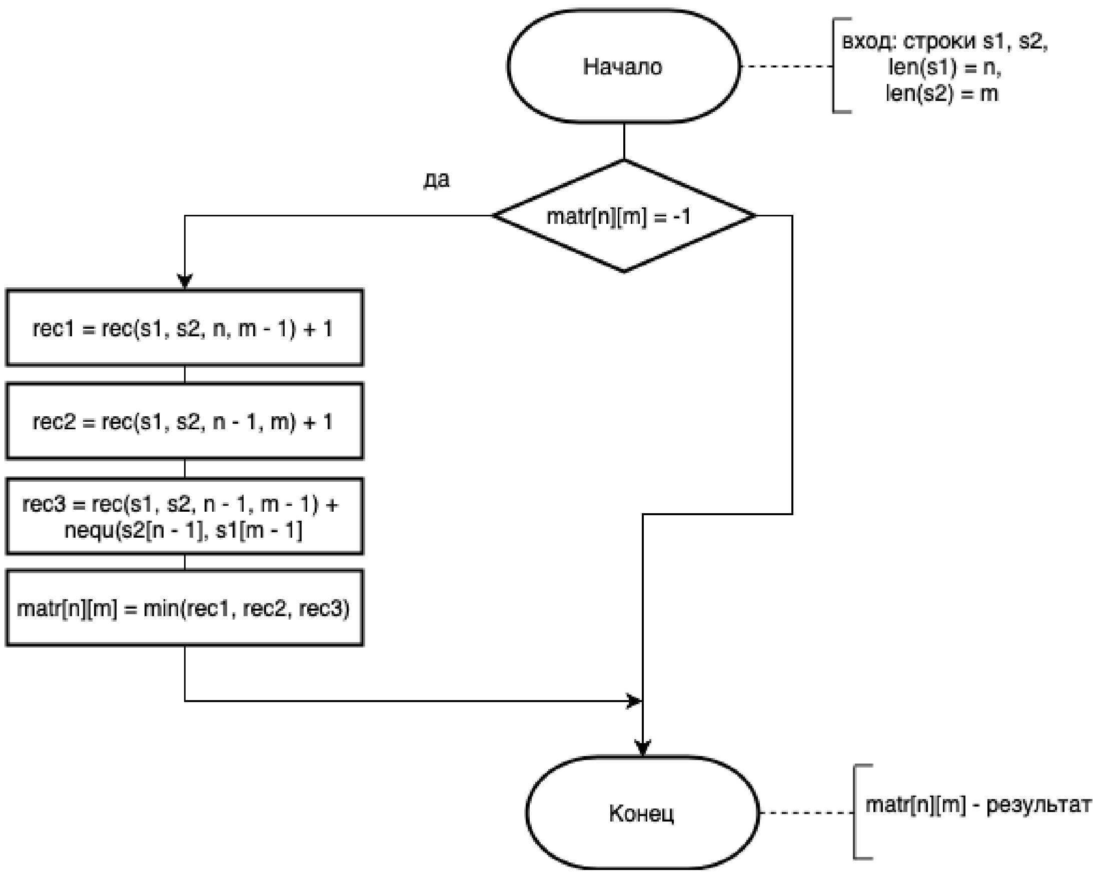

Рис. 2.4: Схема рекурсивного алгоритма нахождения расстояния Левенштейна с кэшем в виде матрицы

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

```
static int _recLevCache(const char * s1, const char * s2, int n, int m, int ** matr)
{
    if (matr[n][m] == -1)
    {
        matr[n][m] = minimum(3,
                             _recLevCache(s1, s2, n, m - 1, matr) + 1,
                             _recLevCache(s1, s2, n - 1, m, matr) + 1,
                             _recLevCache(s1, s2, n - 1, m - 1, matr) + (s1[n - 1] != s2[m
                                 - 1]));
    }
    return matr[n][m];
}

int recLevCache(const char * s1, const char * s2)
{
    int n = strlen(s1);
    int m = strlen(s2);

    int **matr = alloc_matrix(n + 1, m + 1);

    for (int i = 0; i < n + 1; i++)
        matr[i][0] = i;
    for (int i = 0; i < m + 1; i++)
        matr[0][i] = i;
    for (int i = 1; i < n + 1; i++)
        for (int j = 1; j < m + 1; j++)
            matr[i][j] = -1;
    int rv = _recLevCache(s1, s2, n, m, matr);
    free_matrix(matr, n, m);
    return rv;
}
```

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

```
int levCache(const char * s1, const char * s2)
{
    size_t n = strlen(s1) + 1;
    size_t m = strlen(s2) + 1;

    int *fs = (int *)malloc(n * sizeof(int));
    int *ss = (int *)malloc(n * sizeof(int));
    for (size_t i = 0; i < n; i++)
        fs[i] = i;

    for (size_t i = 1; i < m; i++)
    {
        ss[0] = i;
        for (size_t j = 1; j < n; j++)
            ss[j] = minimum(3,
                         fs[j] + 1,
                         ss[j - 1] + 1,
                         fs[j - 1] + (s2[i - 1] != s1[j - 1]) );
        for (size_t j = 0; j < n; j++) {
            fs[j] = ss[j];
        }
    }

    int rv = ss[n - 1];
    free(fs);
    free(ss);
    return rv;
}
```

Листинг 3.3: Функция для рекурсивного нахождения расстояния Дамерау-Левенштейна

```
int _recDamLev(const char * s1, const char * s2, int n, int m)
{
    if (n == 0)
        return m;
    if (m == 0)
        return n;
    int min = minimum(3,
                _recDamLev(s1, s2, n, m - 1) + 1,
                _recDamLev(s1, s2, n - 1, m) + 1,
                _recDamLev(s1, s2, n - 1, m - 1) + (s1[n - 1] != s2[m - 1]));

    if (n > 1 && m > 1 && s1[n - 1] == s2[m - 2] && s1[n - 2] == s2[m - 1])
        min = minimum(2, min, _recDamLev(s1, s2, n - 2, m - 2) + 1);
    return min;
}

int recDamLev(const char * s1, const char * s2)
{
    return _recDamLev(s1, s2, strlen(s1), strlen(s2));
}
```

Листинг 3.4: Функция для рекурсивного нахождения расстояния Левенштейна

```
int _recLev(const char * s1, const char * s2, int n, int m)
{
    if (n == 0)
        return m;
    if (m == 0)
        return n;

    return minimum(3,
                _recLev(s1, s2, n, m - 1) + 1,
                _recLev(s1, s2, n - 1, m) + 1,
                _recLev(s1, s2, n - 1, m - 1) + (s1[n - 1] != s2[m - 1]));
}

int recLev(const char * s1, const char * s2)
{
    return _recLev(s1, s2, strlen(s1), strlen(s2));
}
```

3.3 Тестирование

В таблице 4.1 приведены тесты для функций сортировки.

Таблица 3.1: Тестирование алгоритмов нахождения расстояния Левенштейна

Входные строки	Результат	Ожидаемый результат
$ckat, kot$	2	2
$abc, defg$	4	4
$abcd, abcd$	0	0

Таблица 3.2: Тестирование алгоритмов нахождения расстояния Дамерау-Левенштейна

Входные строки	Результат	Ожидаемый результат
$ckat, kot$	2	2
$abc, defg$	4	4
$abcd, abcd$	0	0
$abcd, badc$	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 эданном разделе проводится сравнительный анализ реализованных алгоритмов по процессорному времени и по затарчиваемой памяти.

4.1 Технические характеристики

Все нижепреведенные 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 милисекундах.

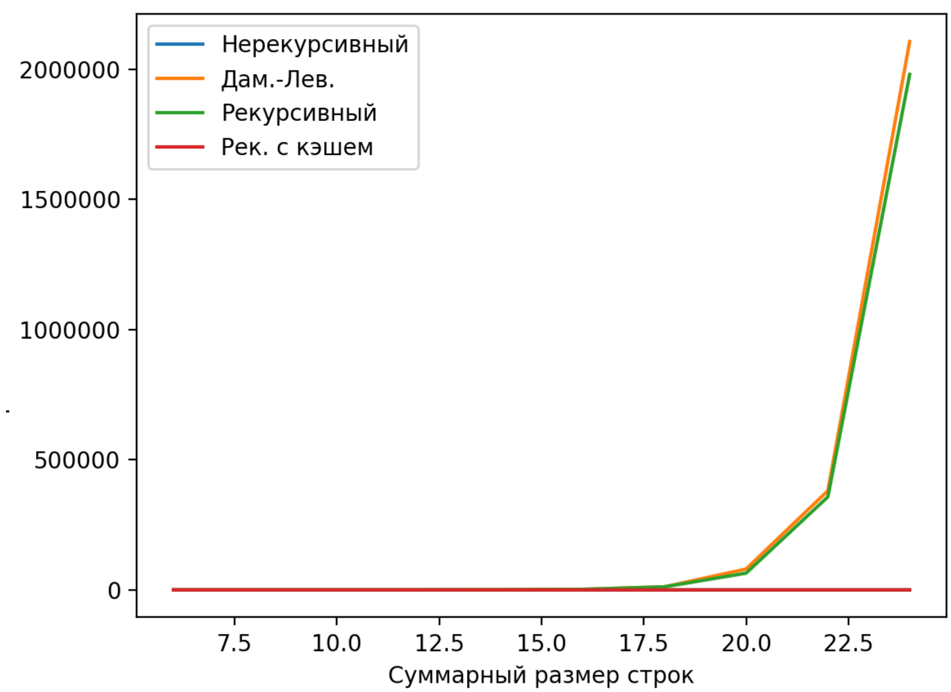

Рис. 4.1: Зависимость времени от суммарной длины строк

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 выделенно одно и то же количество памяти, так так 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 рекусивного 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 кэшэм в виде матрицы менее эффективен по памяти, чем нерекурсивный, однако значительно быстрее, чем обычный рекусивный (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