Федеральное государственное бюджетное образовательное учреждение высшего профессионального образования

«Московский государственный технический университет имени Н.Э. Баумана (национальный исследовательский институт)»

(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ Информатика и системы управления
КАФЕДРА ИУ7

Отчёт

по лабораторной работе №1

Дисциплина: Анализ алгоритмов

Тема лабораторной работы: Расстояние Левенштейна

Студент гр. ИУ7-51Б _____ **Громова В.П.**
(Подпись, дата) (И.О. Фамилия)

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

Москва, 2019г.

Введение **3**

1. Аналитическая часть **4**

1.1. Расстояние Левенштейна 4

1.2. Расстояние Дамерау-Левенштейна 6

1.3. Практическое применение 7

2. Конструкторская часть **7**

2.1. Схемы алгоритмов 7

2.2. Сравнительный анализ рекурсивной и нерекурсивной реализаций 13

3. Технологическая часть **14**

3.1. Требования к ПО 14

3.2. Средства реализации 14

3.3. Листинги кода 14

4. Экспериментальная часть **17**

4.1. Примеры работы программы 17

4.2. Результаты функционального тестирования 18

4.3. Постановка эксперимента по по замеру времени 18

4.4. Сравнительный анализ на материале экспериментальных данных 19

Заключение **21**

Введение

Цель данной лабораторной работы: ознакомиться с понятиями “расстояние Левенштейна” и “расстояние Дамерау-Левенштейна”, изучить матричные и рекурсивные варианты алгоритмов, позволяющих вычислить эти расстояния, а также получить практические навыки в реализации предложенных алгоритмов и их анализе.

Задачи лабораторной работы:

1. Реализовать матричный вариант алгоритма нахождения расстояния Левенштейна.
2. Реализовать матричный вариант алгоритма нахождения расстояния Дамерау-Левенштейна.
3. Реализовать рекурсивный алгоритм нахождения расстояния Дамерау-Левенштейна.
4. Сравнить реализованные алгоритмы по памяти и по времени.
5. Описать проделанную работу и обосновать получившиеся результаты.

1. Аналитическая часть

1.1. Расстояние Левенштейна

Расстояние Левенштейна (редакционное расстояние) — метрика, позволяющая определить «схожесть» двух строк — минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходимых для конвертации одной строки в другую.

Каждая операция имеет свой вес (штраф). Редакционным предписанием называется последовательность действий, необходимых для получения из первой строки вторую, и минимизирующую суммарную цену. Суммарная цена есть искомое расстояние Левенштейна .

Введем следующее обозначение операций:

D (delete) — удалить = 1

I (англ. insert) — вставить = 1

R (replace) — заменить = 1

M (match) — совпадение = 0

Для двух строк S1 и S2 расстояние Левенштейна можно посчитать по рекуррентной формуле:

$$(1) D(S1[1..i], S2[1..j]) = \begin{cases} 0, если\, i=0, j=0 \\ i, если\, i>0, j=0 \\ j, если\, i=0, j>0 \\ min \begin{cases} 1. D(s1[1..i], s2[1..j-1])+1 \\ 2. D(s1[1..i-1], s2[1..j])+1 \\ 3. D(s1[1..i-1], s2[1..j-1])+((s1[i]=s2[j])\,?\,0:1) \end{cases} \end{cases}$$

Здесь, 1 - вставка символа, 2 - удаление символа, 3 - замена символа, при этом, если S1[i - 1] == S2[j - 1], то вес такой операции будет равен нулю, иначе единице.

Можно решать задачу нахождения расстояния Левенштейна рекурсивно в соответствии с формулами, обрабатывая подстроки, пока не будет

достигнут тривиальный случай (тривиальные случаи описаны в формуле до взятия минимума от трех случаев). Работать с подстроками затратно по объему памяти, также в рекурсивном алгоритме будут обрабатываться одинаковые случаи несколько раз, что неэффективно как по памяти, так и по времени.

Существует другое решение — матричное. Матрица размером (length(S1) + 1)x((length(S2) + 1), где length(S) — операция определения длины строки S. Значение в ячейке [i, j] равно значению D(s1[1...i], s2[1...j]). Первая строка и первый столбец тривиальны. В соответствии с формулой получаем: A[i][j] = min (A[i-1][j] + 1, A[i][j-1] + 1, A[i-1][j-1] + m(s1[i], s2[j])), где m(s1[i], s2[j]) — это индикаторная функция, равная нулю при S1[i - 1] == S2[j - 1] и 1 в противном случае.

Результатом (расстоянием Левенштейна между двумя строками) будет ячейка матрицы с индексами i = length(S1) и j = length(S2) (правый нижний угол).

Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла в левый верхний, на каждом шаге ища минимальное из трёх значений:

• если минимально (D(i-1, j) + цена удаления символа S1[i]), добавляем удаление символа S1[i] и идём в (i-1, j);

• если минимально (D(i, j-1) + цена вставки символа S2[j]), добавляем вставку символа S2[j] и идём в (i, j-1);

• если минимально (D(i-1, j-1) + цена замены символа S1[i] на символ S2[j]), добавляем замену S1[i] на S2[j] (если они не равны; иначе ничего не добавляем), после чего идём в (i-1, j-1) Здесь (i, j) — клетка матрицы, в которой мы находимся на данном шаге. Если минимальны два из трех значений (или равны все три), это означает, что есть 2 или 3 равноценных редакционных предписания.

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: T (transposition) — транспозиция = 1.

Для двух строк S1 и S2 расстояние Дамерау-Левенштейна можно посчитать по рекуррентной формуле:

$$(2) D(S1[1..i], S2[1..j]) = \begin{cases} 0, если\ i=0, j=0 \\ i, если\ i>0, j=0 \\ j, если\ i=0, j>0 \\ min \begin{cases} 1. D(s1[1..i], s2[1..j-1])+1 \\ 2. D(s1[1..i-1], s2[1..j])+1 \\ 3. D(s1[1..i-1], s2[1..j-1])+((s1[i]=s2[j])\ ?\ 0:1) \\ 4. если (i, j>1) u (s1[i-1]=s2[j]) u (s1[i]=s2[j-1]) \\ D(s1[1..i-2], s2[1..j-2])+1 \end{cases} \end{cases}$$

Здесь, 1 - 3 имеют те же значения, что и при определении расстояния Левенштейна, а 4 - перестановка символов только в том случае, если S1[i - 1] = S2[j] и S1[i] = S2[j - 1].

Решения задачи нахождения расстояния Дамерау-Левенштейна аналогичны решениям задачи нахождения расстояния Левенштейна. При рекурсивном решении добавляется еще одна ветвь в дерево рекурсий, когда проверяется условие того, что два заключительных символа очередной подстроки равны двум другим из второй подстроки с учетом транспозиции. Для матричной реализации:

матрица размером (length(S1) + 1)x((length(S2) + 1). Значение в ячейке [i, j] равно значению D(s1[1...i], s2[1...j]). Первая строка и первый столбец тривиальны. В соответствии с формулой получаем значение в других

ячейках таблицы: A[i][j] = min (A[i-1][j] + 1, A[i][j-1] + 1, A[i-1][j-1] + m(s1[i], s2[j]), A[i - 2][j - 2] + 1 если s1[i] = s2[j-1]и s1[i-1] = s2[j] и i > 1 и j > 1). Результат также будет находится в правом нижнем углу.

Для восстановления редакционного предписания требуется вычислить матрицу D, после чего идти из правого нижнего угла в левый верхний, на каждом шаге ища минимальное из трёх значений:

• если минимально (D(i-1, j) + цена удаления символа S1[i]), добавляем удаление символа S1[i] и идём в (i-1, j);

• если минимально (D(i, j-1) + цена вставки символа S2[j]), добавляем вставку символа S2[j] и идём в (i, j-1);

• если минимально (D(i-1, j-1) + цена замены символа S1[i] на символ S2[j]), добавляем замену S1[i] на S2[j] (если они не равны; иначе ничего не добавляем), после чего идём в (i-1, j-1)

• если транспозиция возможна, то в минимум входит также (D(i-2, j-2) + 1)

1.3. Практическое применение

Данные алгоритмы применяются при поиске информации по запросу, с помощью них можно найти наиболее подходящее(имеющее наименьшее расстояние) к нему слово и заменить его в поисковой строке. Метод оценки редакционного расстояния активно применяется в биологии для оценки мутаций.

2. Конструкторская часть

2.1. Схемы алгоритмов

На рисунках 2.1 - 2.5 представлены схемы алгоритмов поиска расстояния Левенштейна и Дамерау-Левенштейна для входных строк s1 и s2.

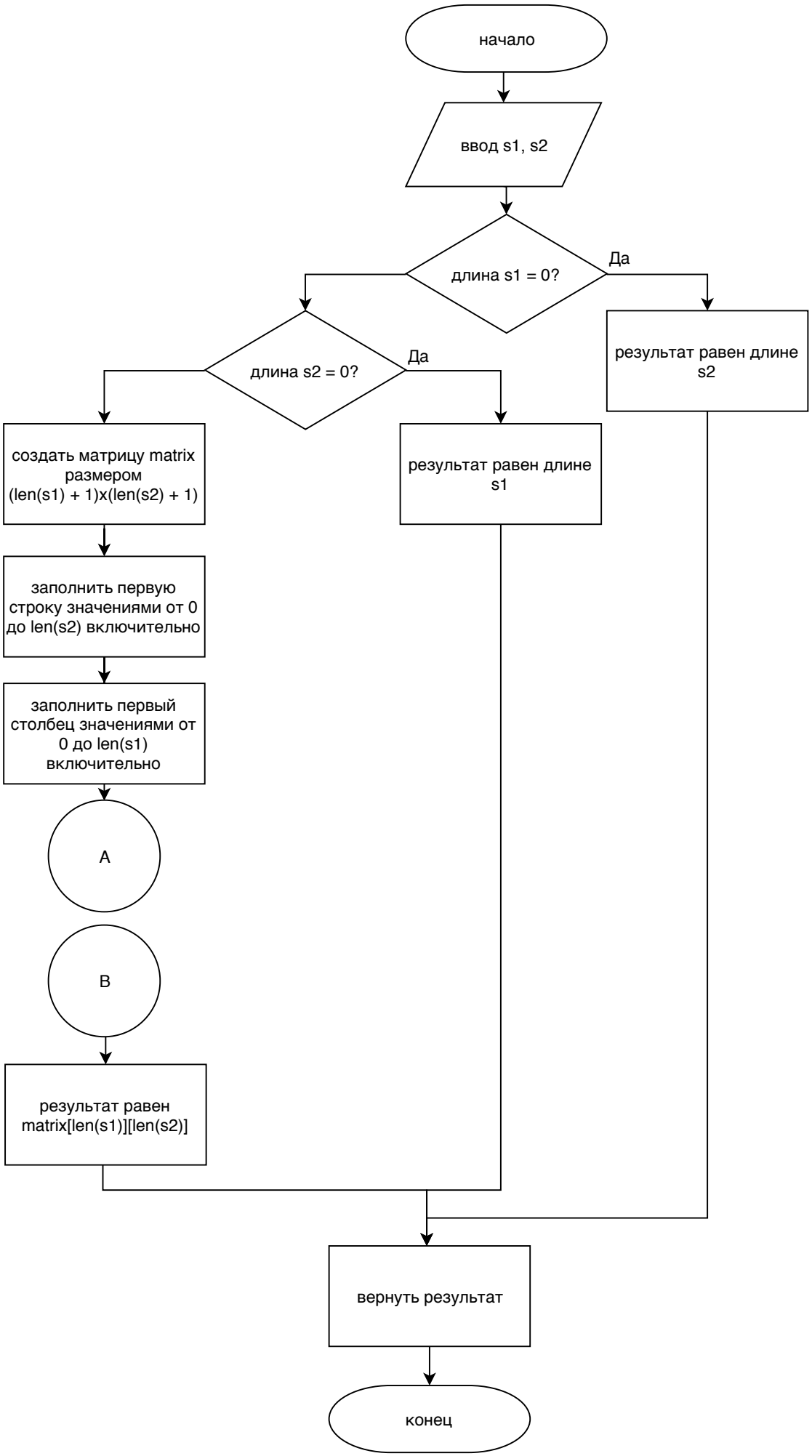


Рис. 2.1 - матричный алгоритм нахождения расстояния Левенштейна

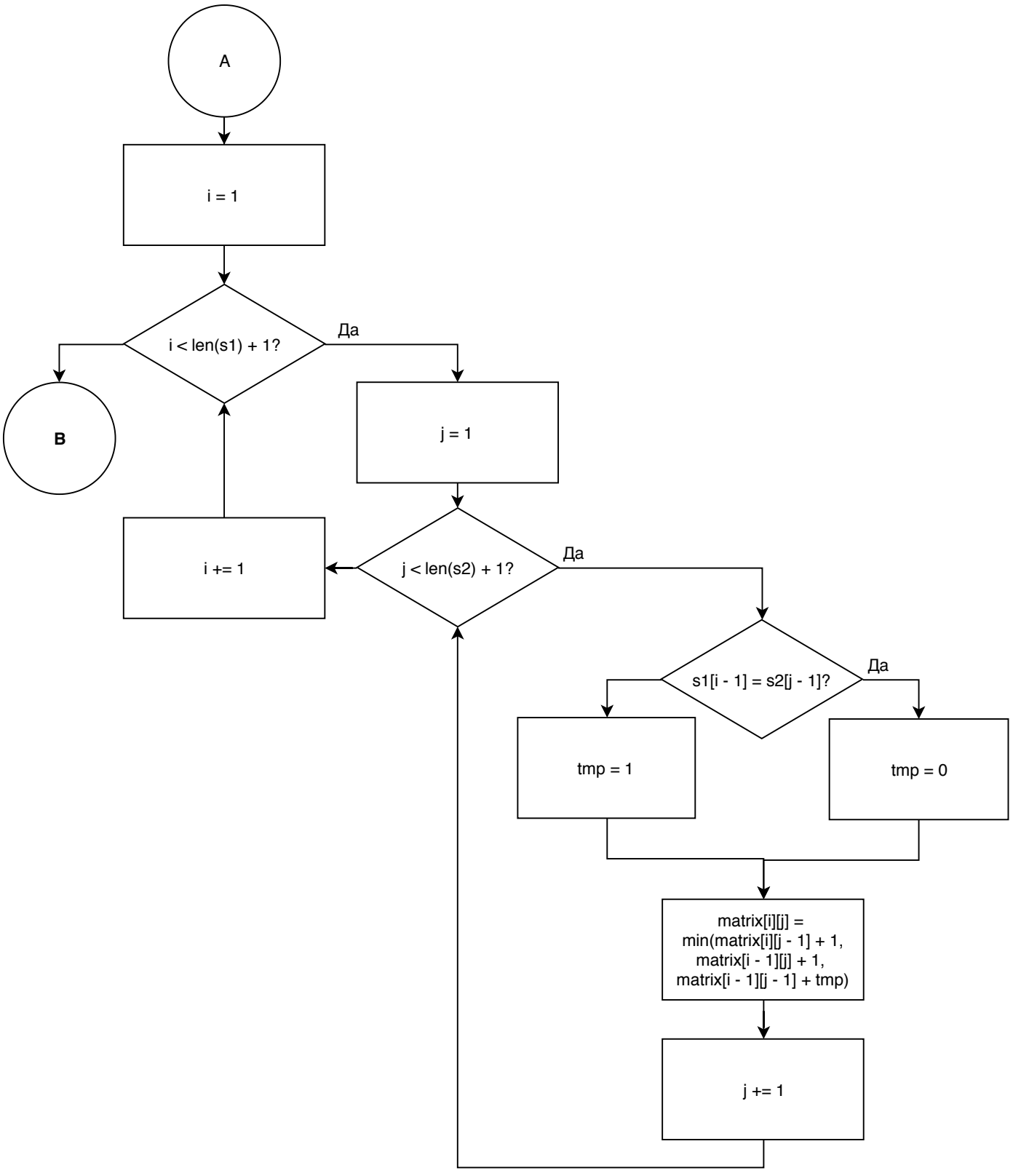


Рис. 2.2 - матричный алгоритм нахождения расстояния Левенштейна (продолжение)

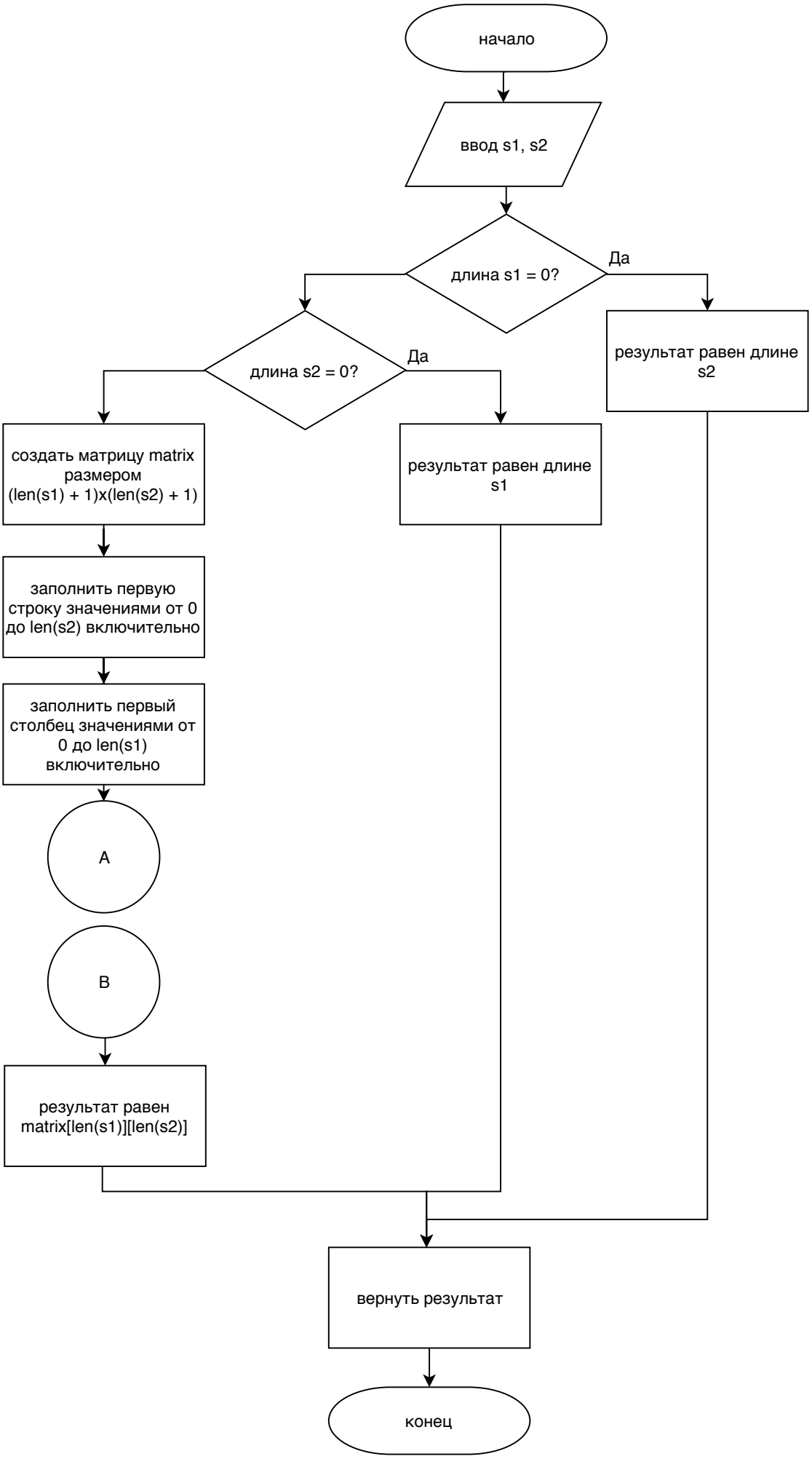


Рис. 2.3 - матричный алгоритм нахождения расстояния Дамерау - Левенштейна

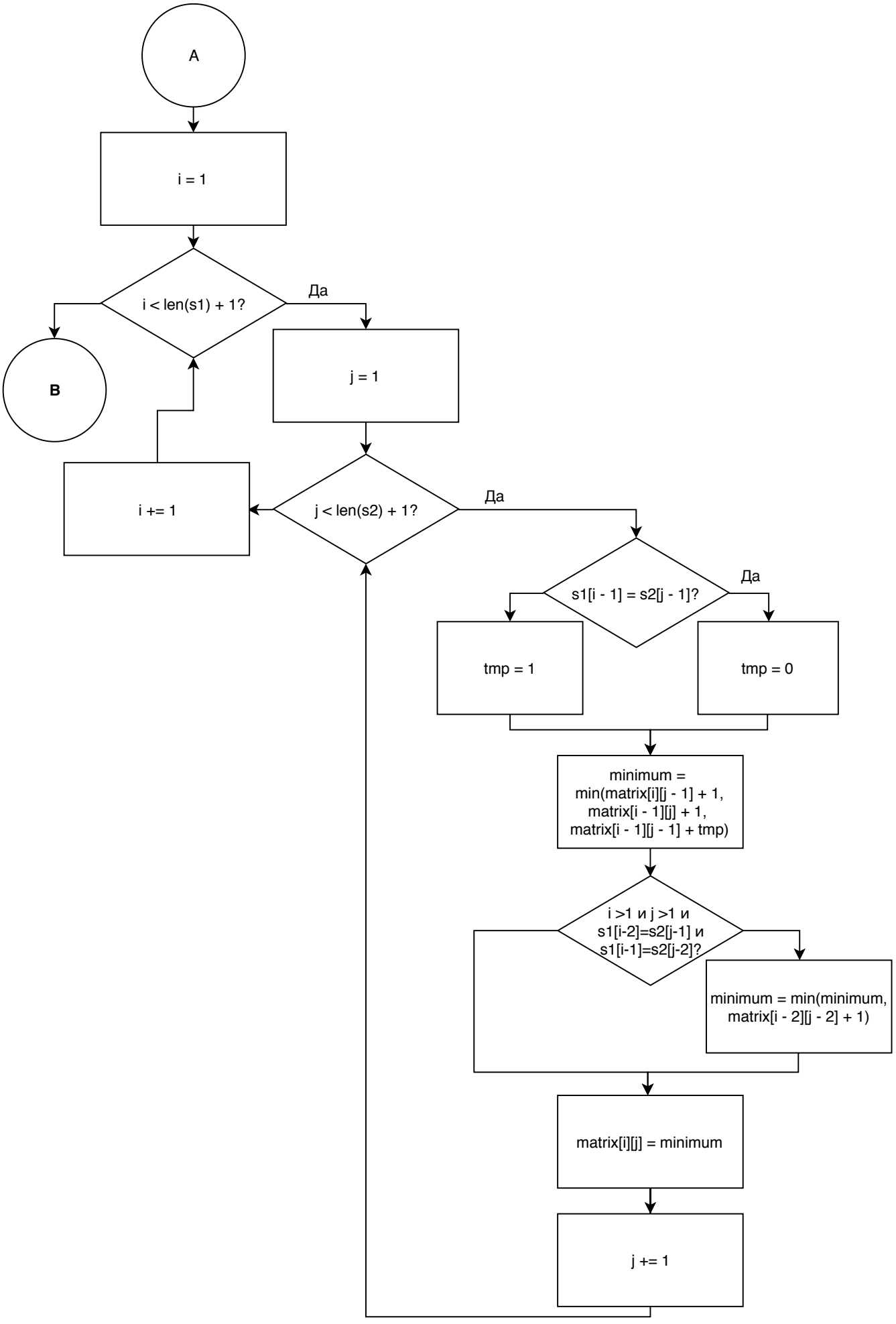


Рис. 2.4 - матричный алгоритм нахождения расстояния Дамерау - Левенштейна (продолжение)

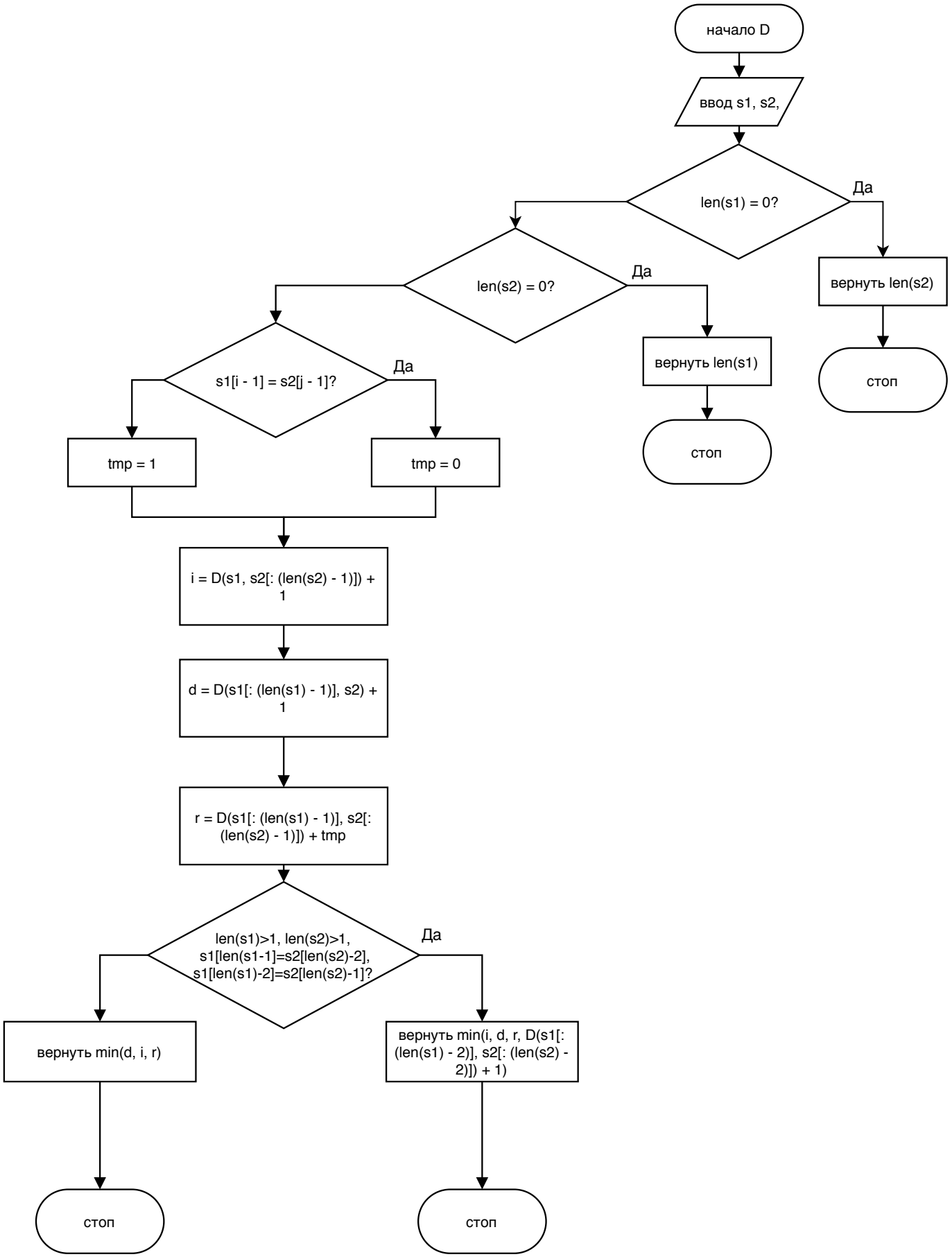


Рис. 2.5 - рекурсивный алгоритм нахождения расстояния Дамерау - Левенштейна

2.2. Сравнительный анализ рекурсивной и нерекурсивной реализаций

Для реализации рекурсивного алгоритма поиска расстояния Дамерау-Левенштейна можно воспользоваться формулой (2) из первого раздела, приняв D за вызов рекурсивной функции. При каждом рекурсивном вызове необходимо передавать в функцию подстроки исходных строк, что явно затратно по памяти. Эту проблему возможно избежать, если занести строки в глобальные переменные и использовать индексы (однако в этом случае могут возникнуть проблемы с глобальными переменными). В то же время вызов функции, которая реализует матричный алгоритм, происходит один раз и в функцию только один раз передаются обе строки. Кроме этого память будет затрачена на хранение матрицы.

Итого затраты памяти при реализации матричных и рекурсивного алгоритма (при входных строках s1 и s2, len(s1) = n, len(s2) = m):

I. матричный алгоритм для расстояния Левенштейна:
   A. хранение строк: (n + m) * sizeof(char)
   B. хранение матрицы: (n + 1) * (m + 1) * sizeof(int)
   C. хранение длин: 2 * sizeof(int)
   D. вспомогательные переменные: 3 * sizeof(int)

II. матричный алгоритм для расстояния Дамерау-Левенштейна:
   A. хранение строк: (n + m) * sizeof(char)
   B. хранение матрицы: (n + 1) * (m + 1) * sizeof(int)
   C. хранение длин: 2 * sizeof(int)
   D. вспомогательные переменные: 6 * sizeof(int)

III. рекурсивный алгоритм Дамерау-Левенштейна (для каждого вызова):
   A. хранение строк: (n + m) * sizeof(char)
   B. хранение длин: 2 * sizeof(int)
   C. вспомогательные переменные: 5 * sizeof(int)
   D. хранение адреса возврата

На каждый вызов рекурсивной функции тратится еще и время. При этом в процессе работы рекурсивный алгоритм обрабатывает одинаковые варианты несколько раз, что влечет за собой повторные вычисления и нерациональную трату ресурсов. Это продемонстрировано на схеме на рис. 2.6. Однако, выполнение самой функции не включает в себя вложенные циклы, как в матричной реализации.

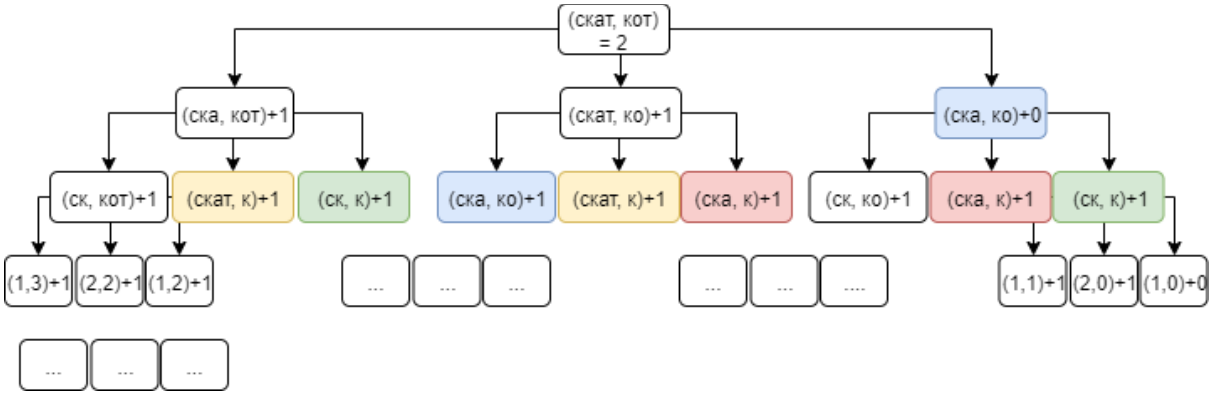


Рис. 2.6 - дерево рекурсивных вызовов

3. Технологическая часть

3.1. Требования к ПО

Программа на вход получает две строки символов. Результат работы программы: число - искомое расстояние. Для матричных реализаций дополнительно выводится получившаяся в результате работы программы матрица.

3.2. Средства реализации

В данной работе использовался язык программирования Python. Для замера времени использовалась дополнительный модуль `time`.

3.3. Листинги кода

На листинге 1 представлена реализация матричной версии алгоритма поиска расстояния Левенштейна

Листинг 1:

```
1. def levenshtein_distance_mtr(s1, s2):
2.     len_s1 = len(s1)
3.     len_s2 = len(s2)
4.
```

```
    mtr = [[0] * (len_s2 + 1) for i in range(len_s1 + 1)]
    mtr[0] = [i for i in range(len_s2 + 1)]
    for i in range (len_s1 + 1):
        mtr[i][0] = i

      for i in range(1, len_s1 + 1):
          for j in range(1, len_s2 + 1):
              tmp = 1
              if (s1[i - 1] == s2[j - 1]):
                  tmp = 0
              mtr[i][j] = min(mtr[i - 1][j] + 1, mtr[i][j - 1] +
1, mtr[i - 1][j - 1] + tmp)

      return mtr, mtr[len_s1][len_s2]
```

На листинге 2 представлена матричная реализация матричного алгоритма поиска расстояния Дамерау-Левенштейна.

Листинг 2:

```
def damerau_levenshtein_distance_mtr(s1, s2):
    len_s1 = len(s1)
    len_s2 = len(s2)
    mtr = [[0] * (len_s2 + 1) for i in range(len_s1 + 1)]
    mtr[0] = [i for i in range(len_s2 + 1)]
    for i in range (len_s1 + 1):
        mtr[i][0] = i

    for i in range(1, len_s1 + 1):
            for j in range(1, len_s2 + 1):
                tmp = 1
                if (s1[i - 1] == s2[j - 1]):
                    tmp = 0
                insert = mtr[i][j - 1] + 1
                delete = mtr[i - 1][j] + 1
                replace = mtr[i - 1][j - 1] + tmp
                mtr[i][j] = min(insert, delete, replace)
                if (i > 1 and j > 1):
                     if ((s1[i - 2] == s2[j - 1]) and (s1[i - 1] ==
s2[j - 2])):
                          mtr[i][j] = min(mtr[i][j], mtr[i - 2][j -
2] + 1)
    return mtr, mtr[len_s1][len_s2]
```

На листинге 3 представлена рекурсивная реализация алгоритма поиска расстояния Дамерау-Левенштейна.

Листинг 3:

```
def damerau_levenshtein_distance_rec(s1, s2):
    len_s1 = len(s1)
    len_s2 = len(s2)

    if (len_s1 == 0):
        return len_s2
    if (len_s2 == 0):
        return len_s1

        insert = damerau_levenshtein_distance_rec(s1, s2[:(len_s2 - 1)]) + 1
         delete = damerau_levenshtein_distance_rec(s1[: (len_s1 - 1)], s2) + 1
        tmp = 0
        if (s1[len_s1 - 1] != s2[len_s2 - 1]):
            tmp = 1
         replace = damerau_levenshtein_distance_rec(s1[: (len_s1 - 1)], s2[: (len_s2 - 1)]) + tmp
        minimum = min(insert, delete, replace)

        if (len_s1 > 1 and len_s2 > 1):
            if ((s1[len_s1 - 2] == s2[len_s2 - 1]) and (s1[len_s1 - 1] == s2[len_s2 - 2])):
                    change = damerau_levenshtein_distance_rec(s1[:(len_s1 - 2)], s2[:(len_s2 - 2)])
                minimum = min(minimum, change + 1)

        return minimum
```

4. Экспериментальная часть

4.1. Примеры работы программы

Ниже представлены примеры работы программы:

```
Введите первую сроку: бот
Введите вторую строку: лот
Результат:
Расстояние Левенштейна (матричный алгоритм): 1
0 1 2 3
1 1 2 3
2 2 1 2
3 3 2 1
Расстояние Дамерау-Левенштейна (матричный алгоритм): 1
0 1 2 3
1 1 2 3
2 2 1 2
3 3 2 1
Расстояние Дамерау-Левенштейна (рекурсивный алгоритм): 1
Введите первую сроку: ночь
Введите вторую строку: день
Результат:
Расстояние Левенштейна (матричный алгоритм): 3
0 1 2 3 4
1 1 2 2 3
2 2 2 3 3
3 3 3 3 4
4 4 4 4 3
Расстояние Дамерау-Левенштейна (матричный алгоритм): 3
0 1 2 3 4
1 1 2 2 3
2 2 2 3 3
3 3 3 3 4
4 4 4 4 3
Расстояние Дамерау-Левенштейна (рекурсивный алгоритм): 3
Введите первую сроку: уклон
Введите вторую строку: кулон
Результат:
Расстояние Левенштейна (матричный алгоритм): 2
0 1 2 3 4 5
1 1 1 2 3 4
2 1 2 2 3 4
3 2 2 2 3 4
4 3 3 3 2 3
5 4 4 4 3 2
Расстояние Дамерау-Левенштейна (матричный алгоритм): 1
0 1 2 3 4 5
1 1 1 2 3 4
2 1 1 2 3 4
3 2 2 1 2 3
4 3 3 2 1 2
5 4 4 3 2 1
Расстояние Дамерау-Левенштейна (рекурсивный алгоритм): 1
```

4.2. Результаты функционального тестирования

В таблице 1 представлены результаты функционального тестирования программы.

Таблица 1

Тестовые случаи (классы эквивалентности)

№	S1	S2	Ожидаемый результат	Полученный результат
1	пустая строка	пустая строка	0, 0, 0	0, 0, 0
2	пустая строка	абв	3, 3, 3	3, 3, 3
3	абв	пустая строка	3, 3, 3	3, 3, 3
4	сон	сон	0, 0, 0	0, 0, 0
5	соня	сон	1, 1, 1	1, 1, 1
6	сон	сен	1, 1, 1	1, 1, 1
7	со	сон	1, 1, 1	1, 1, 1
8	сно	сон	2, 1, 1	2, 1, 1

В таблице выше в колонках “Ожидаемый результат” и “Полученный результат” содержат 3 числа - результаты работы функций:

1) матричный алгоритм поиска расстояния Левенштейна,
2) матричный алгоритм для расстояния Дамерау-Левенштейна,
3) рекурсивный алгоритм поиска расстояния Дамерау-Левенштейна.

Во время тестирования программа корректно сработала при вводе пустых строк и при вводе равных строк; верный результат был получен при необходимости совершения операций удаления (соня/сон), замены (сон/сен), добавления (со/сон), перестановки (сно/сон).

4.3. Постановка эксперимента по по замеру времени

При сравнении быстродействия матричных алгоритмов были использованы строки длинной в диапазоне от 0 до 500 с шагом 100. результат одного эксперимента рассчитывался как средний из результатов проведенных испытаний с одинаковыми входными данными. Количество повторов каждого эксперимента = 10. Результат одного эксперимента

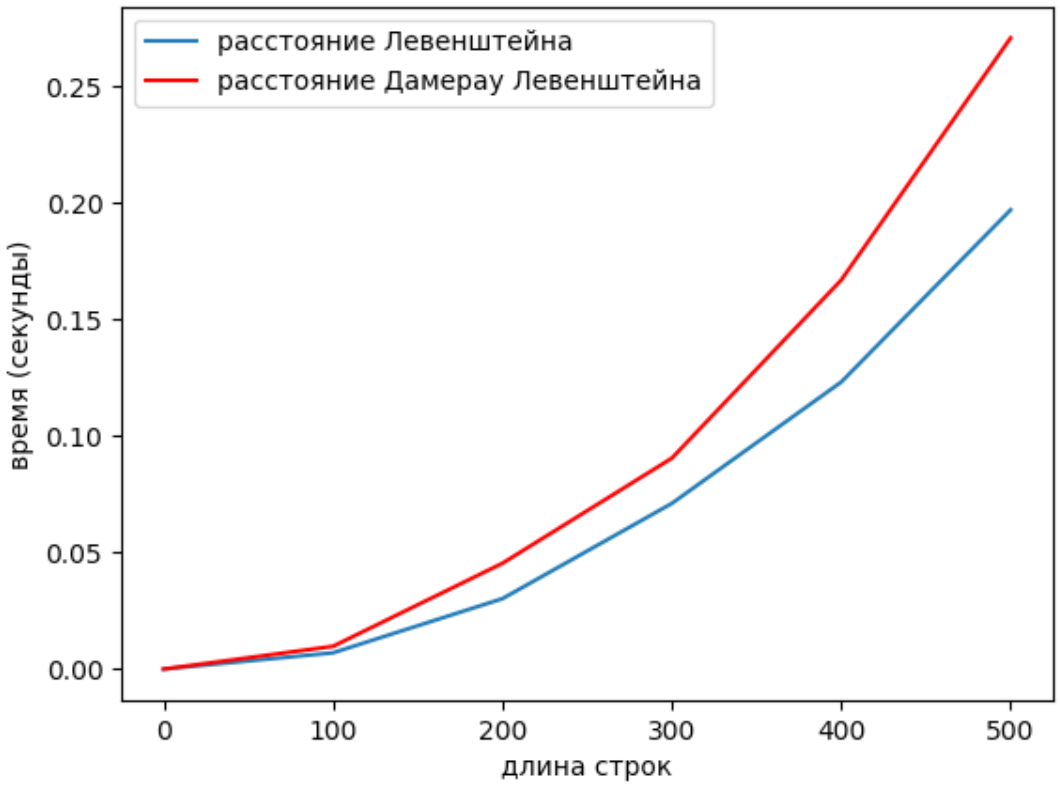
рассчитывается как средний из результатов проведенных испытаний с одинаковыми входными данными. Для сравнения рекурсивной и матричной реализации были использованы строки длинной в диапазоне от 0 до 10 с шагом 1. Каждый эксперимент проводился 10 раз и в результат было записано среднее значение времени.

4.4. Сравнительный анализ на материале экспериментальных данных

Ниже приведены графики зависимости временных затрат работы алгоритмов (в секундах) от длин строк.

График 1

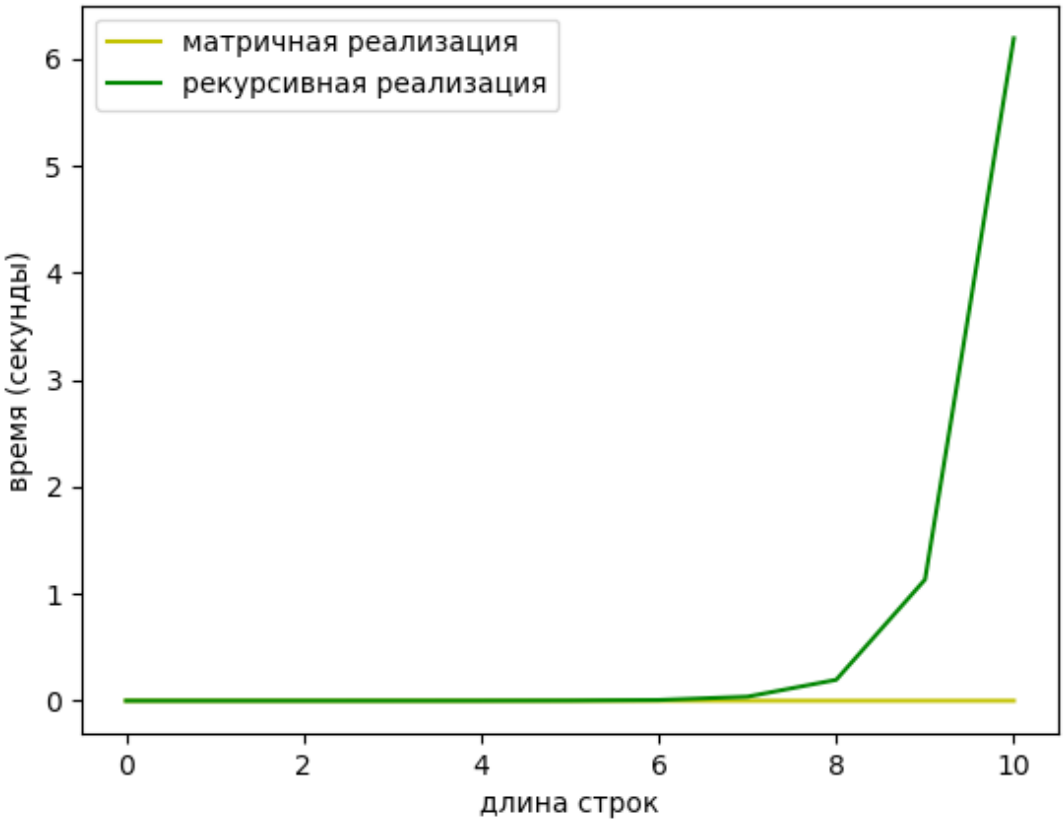
Замеры времени работы программной реализации матричных алгоритмов



На графике видно, что алгоритм для поиска расстояния Дамерау-Левенштейна работает медленнее примерно на десятую долю секунды.

График 2

Замеры времени работы программной реализации итеративного и рекурсивного алгоритмов поиска расстояния Дамерау-Левенштейна



В результате проведенного эксперимента был получен следующий вывод: рекурсивный алгоритм Дамерау-Левенштейна работает гораздо дольше итеративной реализации, начиная с длины строк = 5, время его работы увеличивается в геометрической прогрессии. Итеративный алгоритм значительно превосходит его по эффективности. Алгоритм Дамерау-Левенштейна работает дольше алгоритма Левенштейна, т.к. в нем добавлены дополнительные проверки.

Заключение

Цель данной лабораторной работы была достигнута. В ходе работы были изучены алгоритмы нахождения расстояния Левенштейна и Дамерау-Левенштейна (рекурсивный и итеративный). Выполнено сравнение рекурсивного и итеративного алгоритмов Левенштейна. Изучены зависимости времени выполнения алгоритмов от длин строк. Также были реализованы 3 описанных алгоритма нахождения расстояний Левенштейна и Дамерау-Левенштейна.