Государственное образовательное учреждение
высшего профессионального образования
«Московский государственный технический
университет имени Н. Э. Баумана»
(МГТУ им. Н.Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

ЛАБОРАТОРНАЯ РАБОТА №1

на тему

Расстояние Левенштейна

Студент ИУ7-54: Морозов И. А

Преподаватель: Погорелов. Д. А.

Москва 2018

Введение

Целью данной лабораторной работы является изучение и применение методов динамического программирования на материале алгоритмов Левенштейна и Дамерау-Левенштейна, а также получить практические навыки реализации указанных алгоритмов. Расстояние Левенштейна и его обощение активно применяются:

1. Для исправления ошибок в слове(в поисковых системах, базах данных, при вводе текста, при автоматическом распознавании отсканированного текста или речи);
2. В биоинформатике для сравнения генов, хромосом и белков

1. Аналитическая часть

В данном разделе будут рассмотрены алгоритмы Левенштейна(как матричный так и рекурсивный варианты) и Дамерау-Левенштейна, представлены описания этих алгоритмов, их блок-схемы.

1.1. Описание алгоритмов

1.1.1 Алгоритм Левенштейна

Расстояние Левенштейна между двумя строками - это минимальное количество операций вставки одного символа, удаления одного символа и замены одного символа на другой, необходисых для превращения строки в другую

Обозначания операций:

$w(a, b)$ - цена замены символа a на символ b;

$w(\varepsilon, b)$ - цена вставки символа b;

$w(a, \varepsilon)$ - цена удаления символа;

Каждая операция имеет свою цену:

$w(a, a) = 0$;

$w(a, b) = 1$ при $a \neq b$;

$w(\varepsilon, b) = 1$;

$w(a, \varepsilon) = 1$;

1.1.2 Алгоритм 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.

Обозначение операций:

$w(a, b)$ - цена замены символа a на символ b;

$w(\varepsilon, b)$ - цена вставки символа b;

$w(a, \varepsilon)$ - цена удаления символа;

Каждая операция имеет свою цену:

$w(a, a) = 0$;

$w(a, b) = 1$ при $a \neq b$;

$w(\varepsilon, b) = 1$;

$w(a, \varepsilon) = 1$;

1.1.3 Область применения алгоритмов

Данные алгоритмы применяются:

1. Для поиска на сайтах;

2. В формах заполнения информации на сайтах;

3. Для распознавания рукописных символов;

4. В базах данных;

2. Конструкторская часть

Для реализации алгоритмов будут использованы два подхода: непосредственно рекурсивный и матричный. Под матричным подразумевается заполнение матрицы $m + 1$ на $n + 1$ содержащее в себе значение функции $d(S_1, S_2)$ для всех возможных промежуточных значений. Это необходимо для того, чтобы минимизировать количество промежуточных значений, которые возникают при рекурсивной реализации алгоритма. Для реализации алгоритма Дамерау-Левенштейна будет использован только матричный подход.

2.1 Разработка алгоритмов

2.1.1 Схема алгоритма Левенштейна (Матричный подход)

Входные данные:

substitionCost - цена замены;

S - первая строка

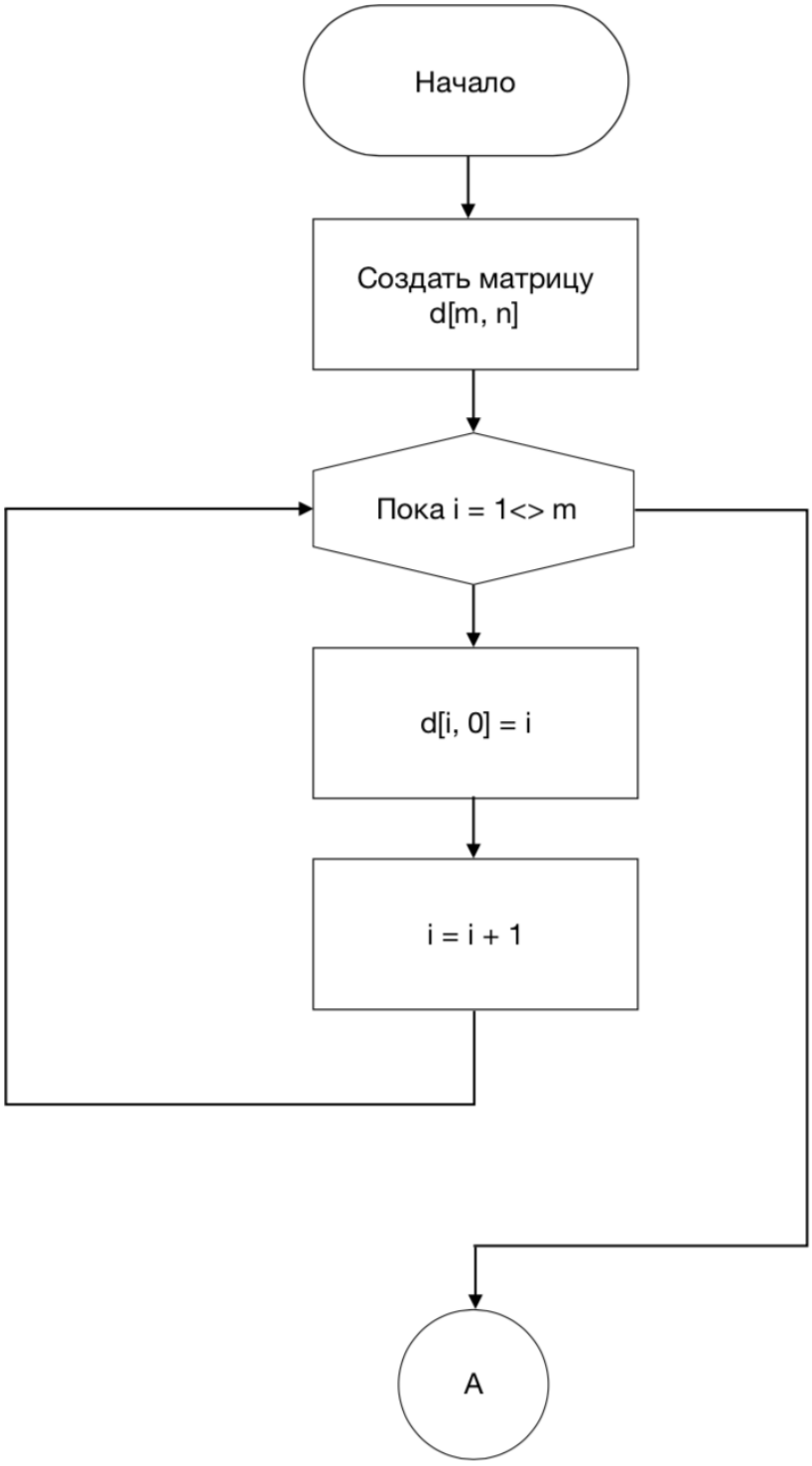
P - вторая строка

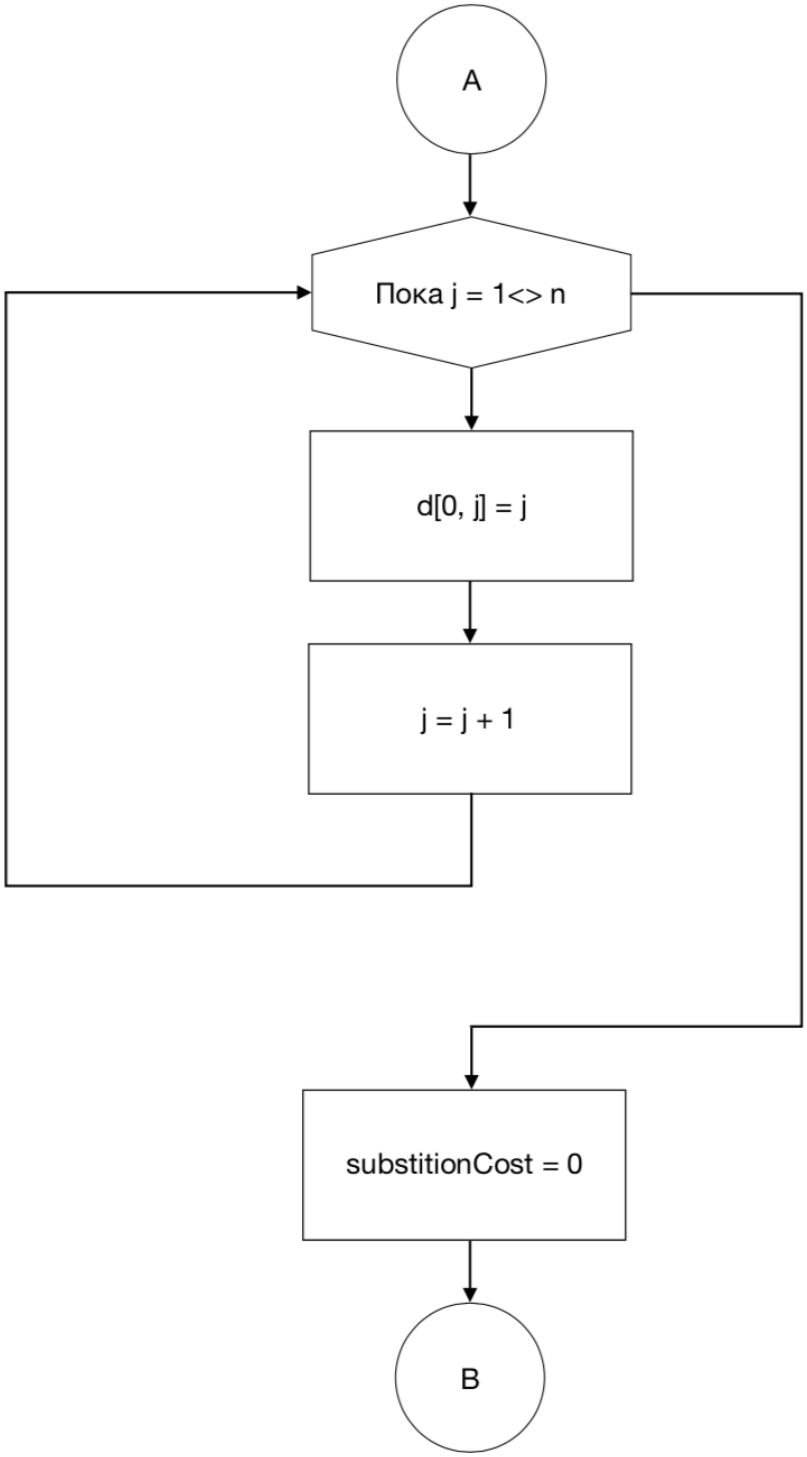
N - длина первой строки

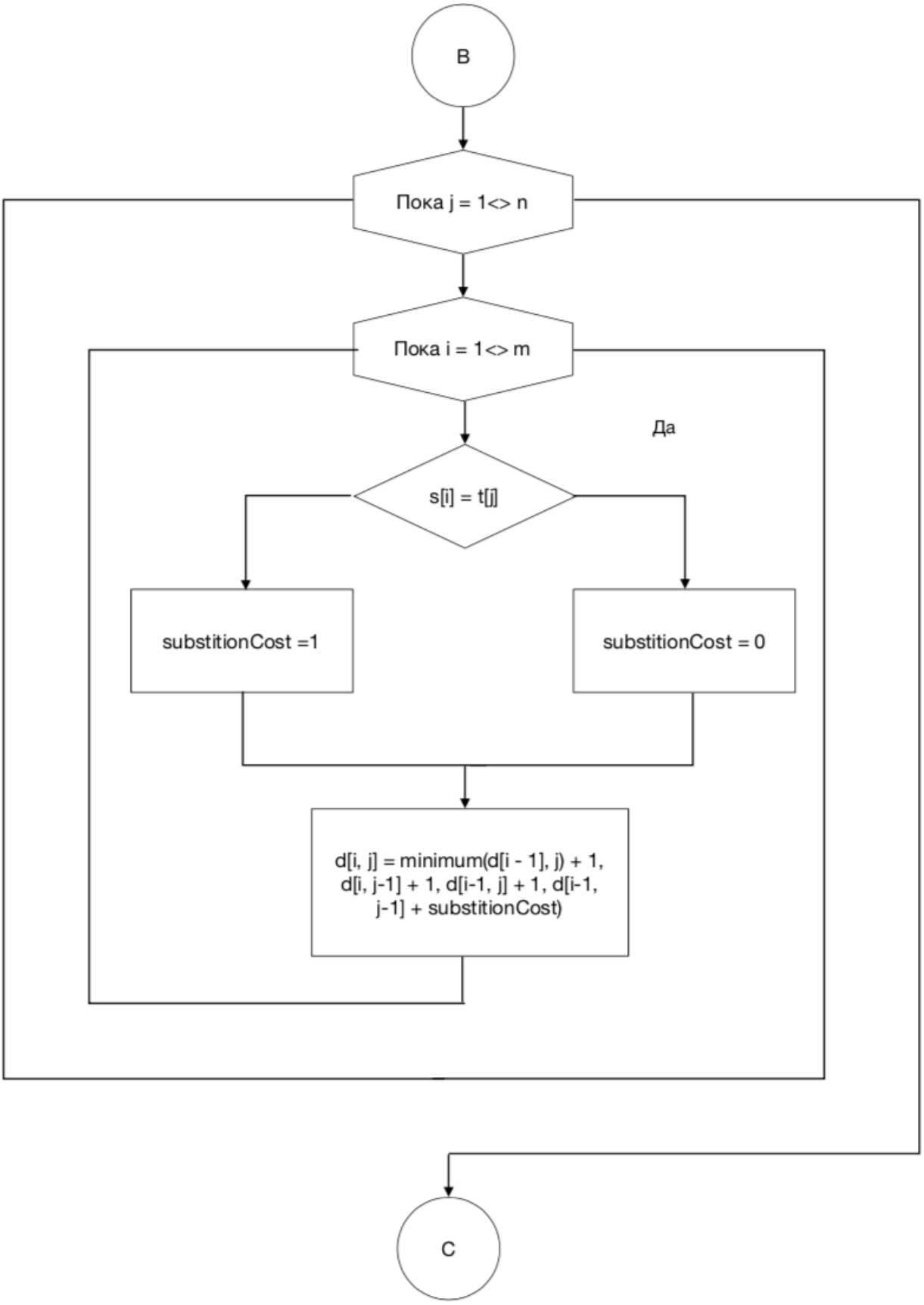
M - длина второй строки

Выходные данные:

d[m][n] - нижний правый элемент таблицы, соответствующий расстоянию между s и t




A
Пока j = 1<> n
d[0, j] = j
j = j + 1
substitionCost = 0
B


B
Пока j = 1<> n
Пока i = 1<> m
Да
s[i] = t[j]
substitionCost =1
substitionCost = 0
d[i, j] = minimum(d[i - 1], j) + 1,
d[i, j-1] + 1, d[i-1, j] + 1, d[i-1,
j-1] + substitionCost)
C

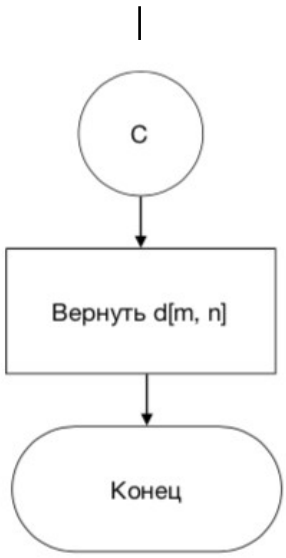


Рисунок 1 - Схема алгоритма Левенштейна

2.1.2 Схема алгоритма Левенштейна (рекурсивная)

Входные данные:

S - первая строка;

len_s - длина первой строки;

T - вторая строка;

len_t - длина второй строки;

Cost - цена замены;

Функция Levenshtein(s, len_s, len_t)

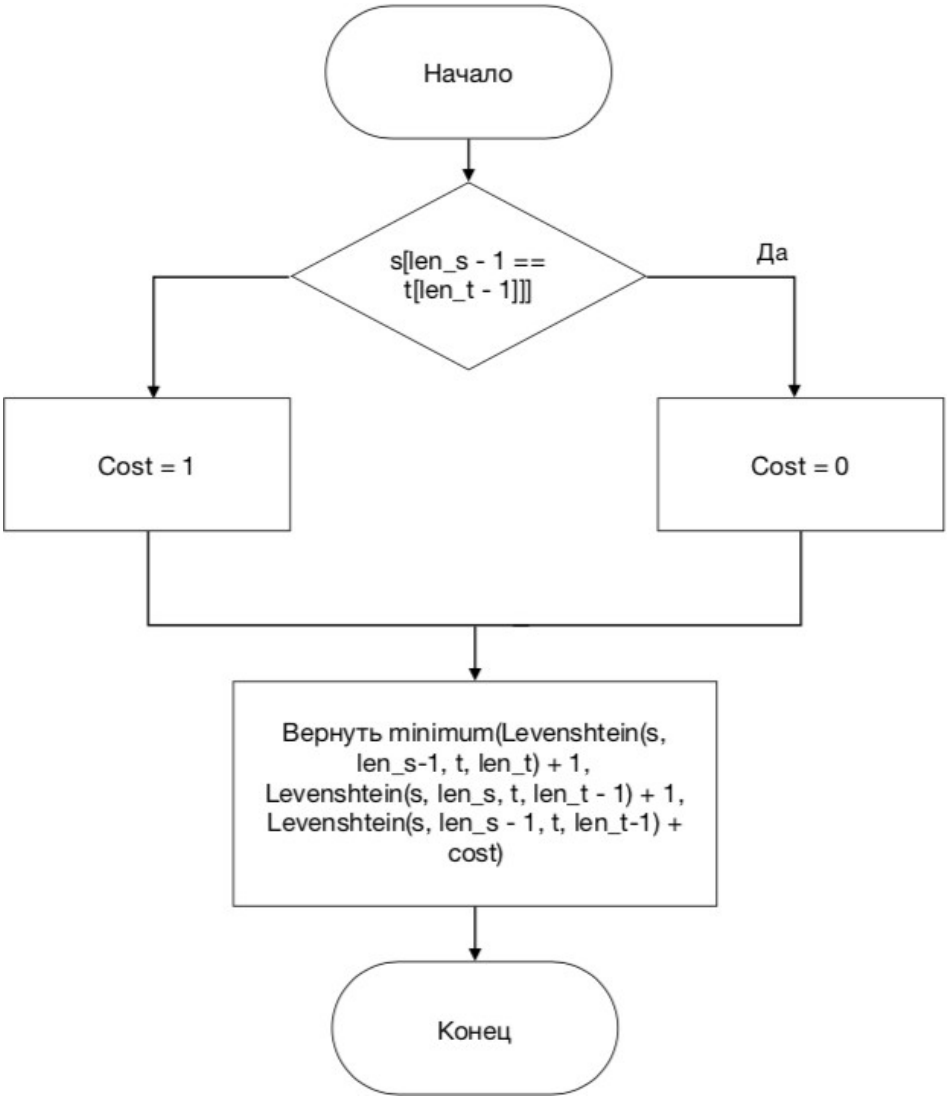


Рисунок 2 - Схема алгоритма Левенштейна (Рекурсивная)

2.1.3 Схема алгоритма Дамерау-Левенштейна

Схема алгоритма Дамерау-Левенштейна отличается от блок-схемы алгоритма Левенштейна только содерджанием бока «Вычиление значения d[i][j] в зависимости от ранее полученных значений». Тоесть меняется только цикл.

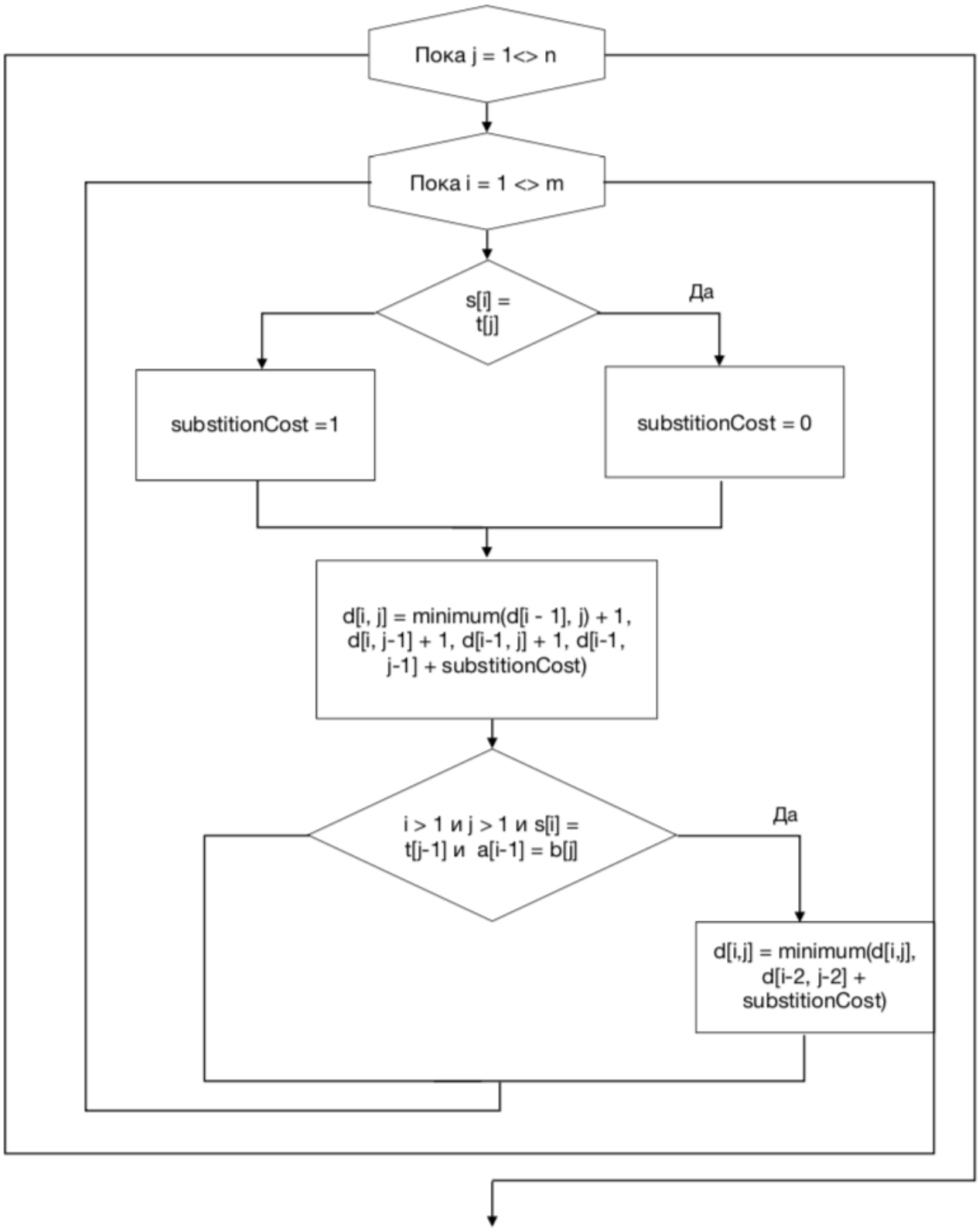


Рисунок 3 - Содержание блока «Вычисление значения d[i][j] в зависимости от ранее полученных значений» для алгоритма Дамерау-Левенштейна

2.2 Сравнительный анализ рекурсивной и нерекурсивной реализаций

Проблема рекусривной реализации алгоритма Левенштейна заключается в том, что значения для некоторых промежуточных строк будут высчитываться несколько раз. Так, например вычисление $d(S_1[1..i], S_2[1..j])$ в худшем случае произойдет при подсчете $d(S_1[1..i], S_2[1..j+1])$ (для учета операции вставки), при подсчете $d(S_1[1..i+1], S_2[1..j])$ (для учета операции удаления) и при подсчете $d(S_1[1..i+1], S_2[1..j+1])$ (для учета совпадения и замены). Каждый из этих промежуточных значений, в свою очередь, также может высчитываться для трех других значений для учета операций вставки, удаления, совпадения и замены. Таким образом, если требуется подсчитать $d(S_1[1..i+k], S_2[1..j+k])$, то формула $d(S_1[1..i], S_2[1..j])$ будет вычислена $k-1$ раз. Очевидно, что при росте длин строк количества пересчетов значений формулы при малых i и j будет резко увеличиваться, что не способствует высокой производительности программы. Также, рекурсивный метод может оказаться и очень невыгодным с точки зрения занимаемой памяти в ОЗУ, так как каждый раз при рекурсивном вызове функции в стек записывается адрес возврата и локальные переменные, которые не убираются из стека, пока не будут выполнены все вложенные функции. Матричный же способ очевидно требует оперативную память под матрицу промежуточных значений, и для малых длин строк, может потребовать больших затрат ресурсов. Однако матрицы позволяют уйти от множественного пересчета одного и того же промежуточного значения, высчитывая все промежуточные значения лишь один раз.

3. Технологическая часть

В данном разделе будет представлено описание используемого языка программирование, а также будет показан листинг кода функций, работающих согласно указанным выше алгоритмам.

3.1 Требования к программному обеспечению

Данная программа была реализована на языке C++ , компилятор для которого поддерживается многими операционными системами.

Компилятор: g++

3.2 Средства реализации

Программа была реализована на операционной система MacOS в среде разработки Xcode.

3.3 Листинг кода

3.3.1 Листинг алгоритма Левенштейн (итеративный)

s1 - первая строка;
s2 - вторая строка;

Листинг 1. Алгоритм Левенштейна.

```
size_t myLevenshteinDistance(const std::string &s1, const std::string
&s2)
{
size_t m
=
s1.size();
```

```
size_t n
=
s2.size();

// Проверка на кириллические символы
bool is_cyrillic = 0;
std::regex txt_regex("[a-zA-Z]");
if(!regex_search(s1, txt_regex)) { m = m / 2;
is_cyrillic = 1; } if(!regex_search(s2, txt_regex)) {
n = n / 2; is_cyrillic = 1; }

if( m==
0 )
return n;
if( n==0
) return
m;
vector<vector<size_t> >
matrix(n+1); for(size_t i =
0; i < (n+1); i++)
{
matrix[i].resize(m+1);
for(size_t j = 0; j <
(m+1); j++)
{
matrix[i][j] = 0;
}
}
for(size_t i = 0; i < (m+1); i++)
matrix[0][i] = i; for(size_t i = 0; i <
```

```
(n+1); i++) matrix[i][0] = i;

// Функция min_three находит минимальный из трех поданных
элементов
for(size_t i = 1; i < (n+1); i++)
{
for(size_t j = 1; j < (m+1); j++)
{
matrix[i][j] = min_three(matrix[i][j-1] + 1, matrix[i-1][j] + 1, matrix[i-1]
[j-1]
+ (s1[cyrillic(j - 1, is_cyrillic)] == s2[cyrillic(i - 1, is_cyrillic)] ? 0 : 1));
}
}

cout << "D matrix:
" << endl;
for(size_t i = 0; i <
(n+1); i++)
{
for(size_t j = 0; j < (m+1); j++)
{
std::cout << matrix[i][j] << " ";
}
std::cout << endl;
}

cout << endl;

return matrix[n][m];
}
```

3.3.2 Листинг алгоритма Левенштейн (рекурсивный)

s1 - первая строка;

s2 - вторая строка;

N - размер первой строки;

M - размер второй строки;

is_cyrillic - флаг проверки на кириллицу(проверяется в другой части кода)

Листинг 2. Алгоритм Левенштейна (рекурсивный)

```
size_t D(const string &s1, const string &s2, size_t n,size_t m, bool
is_cyrillic)
{
    if(n==0)
        return m;
    else
        if(m==0)
            return n;
        else
            return min_three(D(s1, s2, n-1, m, is_cyrillic) + 1, D(s1, s2, n,
m-1, is_cyrillic) + 1, D(s1, s2, n-1, m-1, is_cyrillic)+(s1[cyrillic(m - 1,
is_cyrillic)] == s2[cyrillic(n - 1, is_cyrillic)] ? 0 : 1));
}
```

3.3.3 Листинг алгоритма Домерау- Левенштейна

s1 - первая строка

s2 - вторая строка

Листинг 3. Алгоритм Домерау-Левенштейна

```
size_t DomerauLevenshteinDistance(const std::string &s1, const
std::string &s2, double *startTime, double *endTime)
{
   size_t m =
   s1.size();
   size_t n =
   s2.size();
   size_t cost
   = 0;

   // Проверка на кириллицу
   bool is_cyrillic = 0;
   std::regex txt_regex("[a-zA-Z]");
   if(!regex_search(s1, txt_regex)) { m = m / 2;
   is_cyrillic = 1; } if(!regex_search(s2, txt_regex)) { n
   = n / 2; is_cyrillic = 1; }

   if( m==0 )
   return n;
   if( n==0 )
   return m;

   *startTime = getCPUTime( );
   std::vector<std::vector<size_t
   >> d(m+1);

   for(size_t i = 0; i <= m; i++)
```

```
    {
        d[i].resize(n+1);
        for(size_t j = 0; j <= n; j++)
        {
            d[i][j] = 0;
        }
    }

    for(int i = 0; i <= m; i++)
    d[i][0] = i; for(int j = 0; j
    <= n; j++) d[0][j] = j;
    for(int i = 1; i <= m; ++i)
    {
        for(int j = 1; j <= n; ++j)
        {
            if(s1[cyrillic(i, is_cyrillic)] == s2[cyrillic(j,
                is_cyrillic)]) cost = 0;
            else
                cost = 1;

            d[i][j] = min_three(d[i-1][j] + 1, d[i][j-1] + 1, d[i-1][j-1]
            + cost); if(i > 1 && j > 1 && (s1[cyrillic(i, is_cyrillic)]
            == s2[cyrillic(j-1,
is_cyrillic)]) && (s1[cyrillic(i-1, is_cyrillic)] == s2[cyrillic(j,
                is_cyrillic)])) d[i][j] = min_two(d[i][j], d[i-2][j-2] +
                cost);
        }
    }
    *endTime = getCPUTime();

    cout <<
```

```
    "Матрица:" <<
    endl; for(size_t i =
    0; i <= m; i++)
    {
        for(size_t j = 0; j <= m; j++)
        {
            std::cout << d[i][j] << " ";
        }
        std::cout << endl;
    }



    return d[m][n];
}
```

4. Экспериментальная часть

В данном разделе будет представлены результаты выполнения лабораторной работы, примеры работы написанного ПО и временные характеристики алгоритмов.

Результат работы и интерфейс программы для пары слов МГТУ-МГУ:

```
Введите перую строку: МГУ
Введите вторую строку: МГТУ

Матрица:
0 1 2 3
1 0 1 2
2 1 0 1
3 2 1 1
4 3 2 1
Ливенштейн: 1

Домерау-Ливенштейн: 1
Матрица:
0 1 2 3
1 0 1 2
2 1 0 1
3 2 1 1
4 3 2 1


Ливенштейн(рекурсия): 1
```

Специальный тест на случай, когда Дамерау-Левенштейн имеет преимущество:

```
Введите перую строку: опечатка
Введите вторую строку: оепчатка

Матрица:
0 1 2 3 4 5 6 7 8
1 0 1 2 3 4 5 6 7
2 1 1 1 2 3 4 5 6
3 2 1 2 2 3 4 5 6
4 3 2 2 2 3 4 5 6
5 4 3 3 3 2 3 4 5
6 5 4 4 4 3 2 3 4
7 6 5 5 5 4 3 2 3
8 7 6 6 6 5 4 3 2
Левенштейн: 2

Матрица:
0 1 2 3 4 5 6 7 8
1 1 1 2 3 4 5 6 7
2 1 1 2 3 4 5 6 7
3 2 2 1 2 3 4 5 6
4 3 3 2 1 2 3 4 5
5 4 4 3 2 1 2 3 4
6 5 5 4 3 2 1 2 3
7 6 6 5 4 3 2 1 2
8 7 7 6 5 4 3 2 1
Домерау-Левенштейн: 1


Левенштейн(рекурсия): 2
```

4.1 Постановка эксперимента

Первый эксперимент (с учетом рекурсивной реализации алгоритма Левентейна) производится на строках с размерами от 2 до 12 шагов в 2 символа, так как рекурсивный метод при строках размером уже 20 символов считается слишком долгий промежуток времени. Каждый эксперимент, тем не менее был повторен 100 раз для усреднения значений. Так, например, для строк длинной 12 символов мне уже пришлось прождать около 5 минуу.

Второй эксперимент уже без рекурсивной реализации алгоритма Левенштейна, что позволяет провести указанный в требованиях экперимент со строками длинной от 100 до 1000 символов с шагом в 100 символов. Замер производится 100 раз, затем делится на 100 для выявления среднего значения. Для обоих экспериментов использовались автоматическая генерация строк из заглавных латинских букв и цифр.

4.3 Сравнительный анализ на материале экспериментальных данных

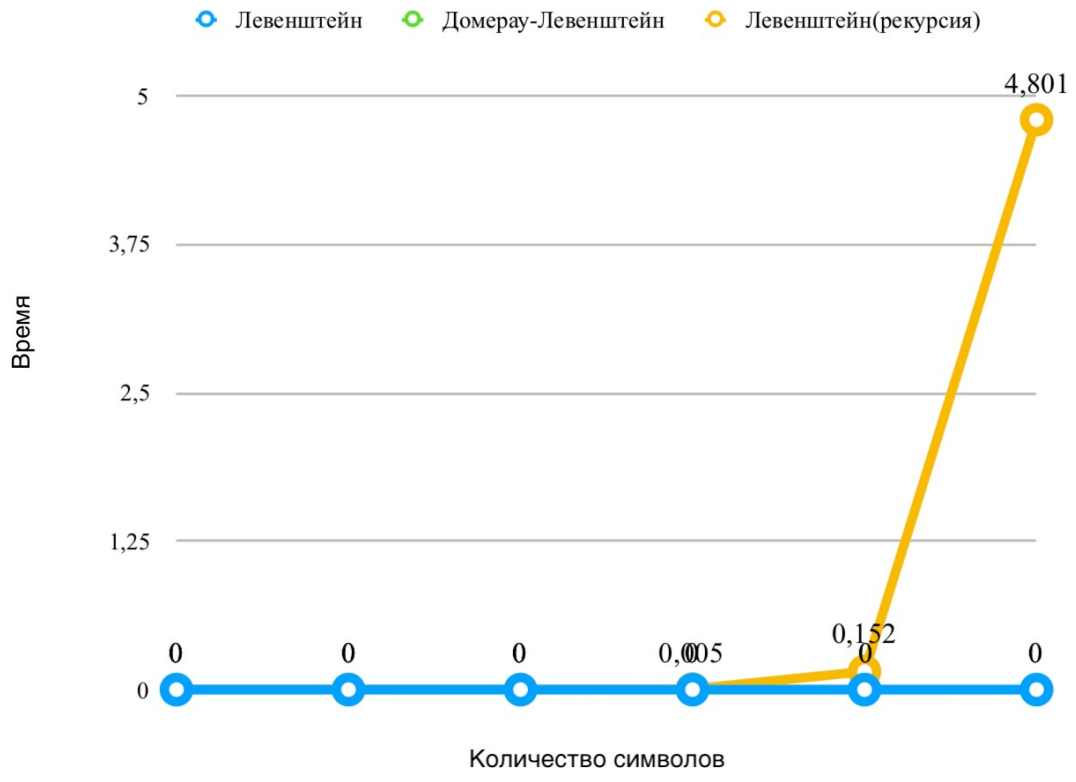
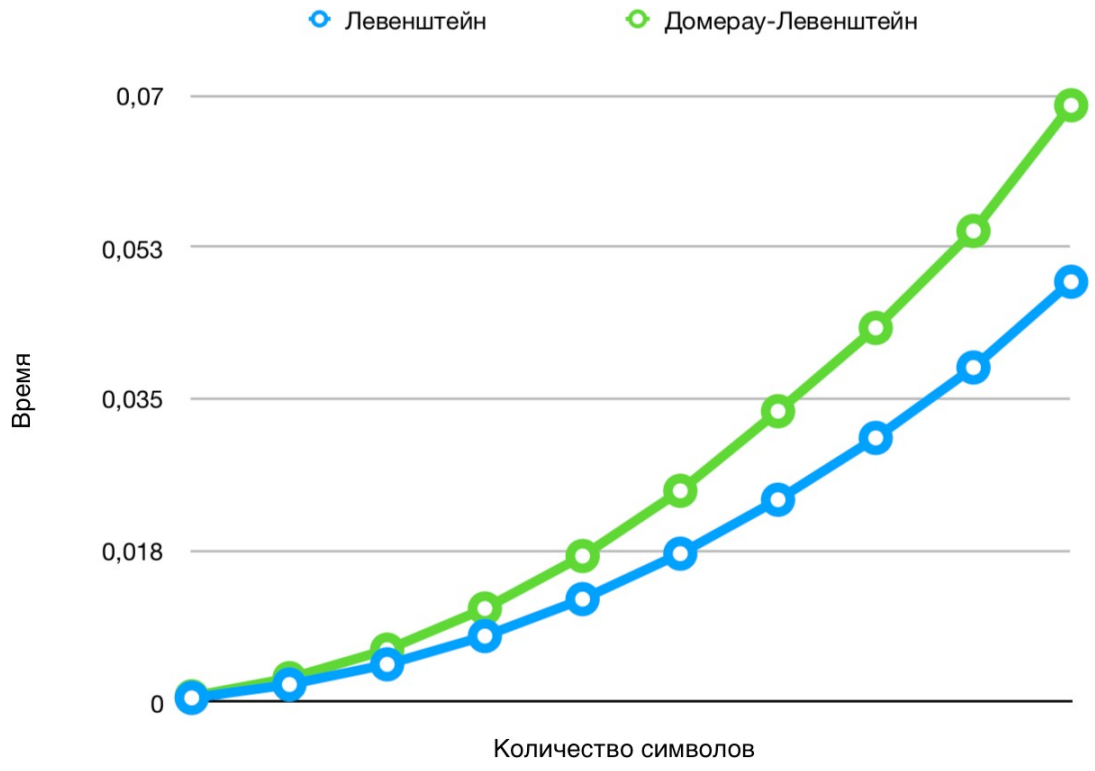


График зависимости от количества символов для разных алгоритмов при большом количестве символов в строках.

Как видно из результатов поставленного эксперимента эффективность рекурсивного алгоритма резко падает с ростом количества символов в строках, однако для строк с двумя символами показывает наилучший результат из всех использованных алгоритмов.

В случае с длинными строками, Дамерау-Левенштейн показывает несколько менее эффективную работоспособность (примерно в 2 раза ниже) нежели простой Левенштейн. Очевидно, это связано с дополнительной проверкой на транспонирование.



Заключение

В ходе данной лабораторной работы были изучены и реализованы алгоритмы нахождения расстояния между двух строк, а именно: алгоритм Левенштейна и алгоритм Дамерау-Левенштейна.

Был проведен анализ двух методов реализации этих алгоритмов - матричного и рекурсивного. Рекурсивный показал себя как наименнее эффективный для большинства обрабатываемых строк, что сходится с ожидаемым результатом.

Несмотря на то, что Дамерау-Левенштейн показывает меньшую эффективность на огромных строках, часто при решении задачи автоматического исправления ошибок, он будет показывать более высокую эффективность, чем Левенштейн. Это связано с тем, что наиболее распространенной ошибкой человека, работающего ща клавиатурой, является случайная перемена метами символов.