Белорусский государственный технологический университет

Кафедра Программной инженерии

**“Математическое программирование”**

**Отчет по лабораторной работе №4**

**Динамическое программирование**

**Вариант 7**

Выполнила: Керезь Е.В.

ФИТ 2 курс, 4 группа

Проверила Харланович А. В.

Минск 2020

**Цель работы:** освоить общие принципы решения задач методом динамического программирования, сравнить полученные решения задач с рекурсивным методом.

Ход выполнения

**Задание 1.** На языке С++ сгенерировать случайным образом строку букв латинского алфавита  длиной  символов и длиной .

#define \_rand(min, max) ( rand() % ((max) - (min) + 1) + (min) )

int \_tmain(int argc, \_TCHAR\* argv[])

{

setlocale(LC\_ALL, "rus");

srand(time(NULL));

char abc[25]; // наш алфавит

char s1[300];

char s2[250];

// заполняем массив

for (int i = 97, n = 0; i <= 122; ++i, ++n)

{

abc[n] = (char)i;

}

for (int i = 0; i < 300; i++)

{

s1[i] = abc[\_rand(0, 25)];

}

for (int i = 0; i < 250; i++)

{

s2[i] = abc[\_rand(0, 25)];

}

Результат:

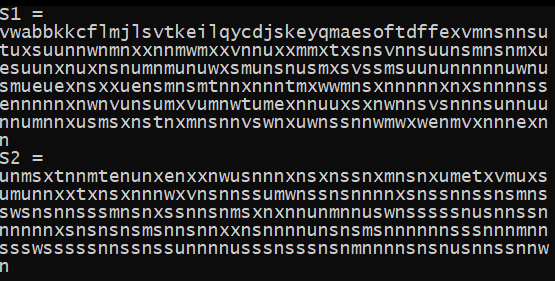


Рис. 1

**Задание 2**

Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – дистанцию Левенштейна для , где - длина строки ,  - строка состоящая из первых  символов строки . (копии экрана и код вставить в отчет).

Исходный код реализации через динамическое программирование:

int min3(int x1, int x2, int x3)

{ return std::min(std::min(x1,x2),x3); }

int levenshtein(int lx, const char x[],int ly, const char y[])

{

int \*\*matr;

int w, left, top, left\_top;

matr = new int\*[lx];

for (int i = 0; i < lx; i++)

matr[i] = new int[ly];

matr[0][0] = 0;

for (int i = 1; i < lx; i++)

matr[i][0] = i;

for (int j = 1; j < ly; j++)

matr[0][j] = j;

for (int i = 1; i < lx; i++)

for (int j = 1; j < ly; j++){

w = x[i - 1] == y[j - 1] ? 0 : 1;

top = matr[i - 1][j];

left = matr[i][j - 1];

left\_top = matr[i - 1][j - 1];

matr[i][j] = std::min(left\_top + w, std::min(top + 1, left + 1));

}

return matr[lx-1][ly-1];

}

Пример реализации рекурсивным методом:

int min3(int x1, int x2, int x3)

{ return std::min(std::min(x1,x2),x3); }

int levenshtein\_r(int lx, const char x[],

int ly, const char y[])

{

int rc = 0;

if (lx == 0) rc = ly;

else if (ly == 0) rc = lx;

else if (lx == 1 && ly == 1 && x[0] == y[0]) rc = 0;

else if (lx == 1 && ly == 1 && x[0] != y[0]) rc = 1;

else rc = min3(

levenshtein\_r(lx-1, x, ly, y)+1,

levenshtein\_r(lx, x, ly-1, y)+1,

levenshtein\_r(lx-1, x, ly-1, y)+(x[lx-1] == y[ly-1]?0:1)

);

return rc;

};

На рисунке 2 представлены дистанции Левенштейна вычисленные при помощи метода динамического программирования, а также рекурсивным алгоритмом.

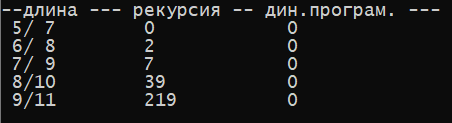
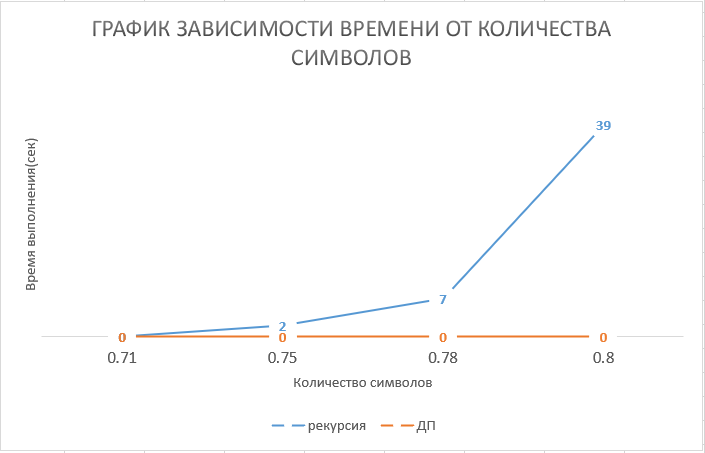


Рис. 2 – проверка работоспособности решений

**Задание 3.** Выполнить сравнительный анализ времени, затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от . (копии экрана и график вставить в отчет).

Метод динамического программирования значительно эффективнее рекурсивного метода, т.к. выполняется намного быстрее.

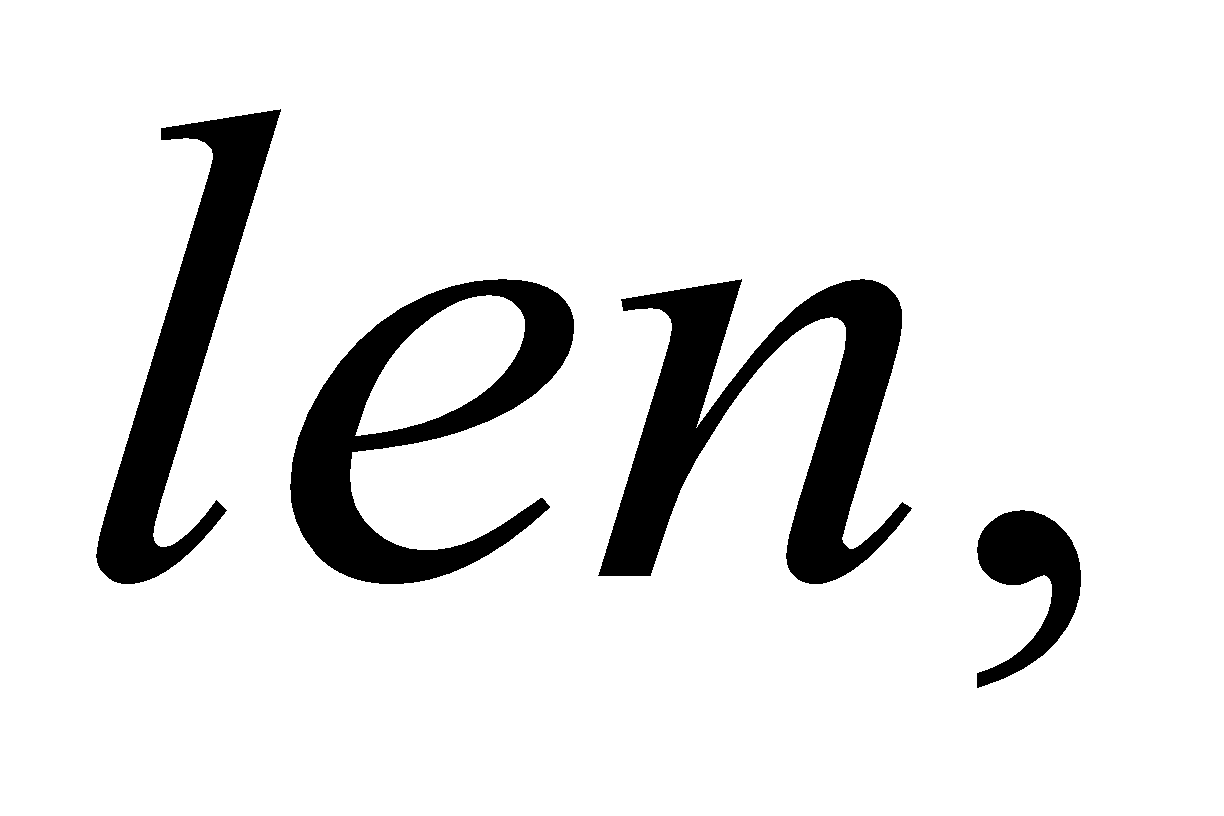
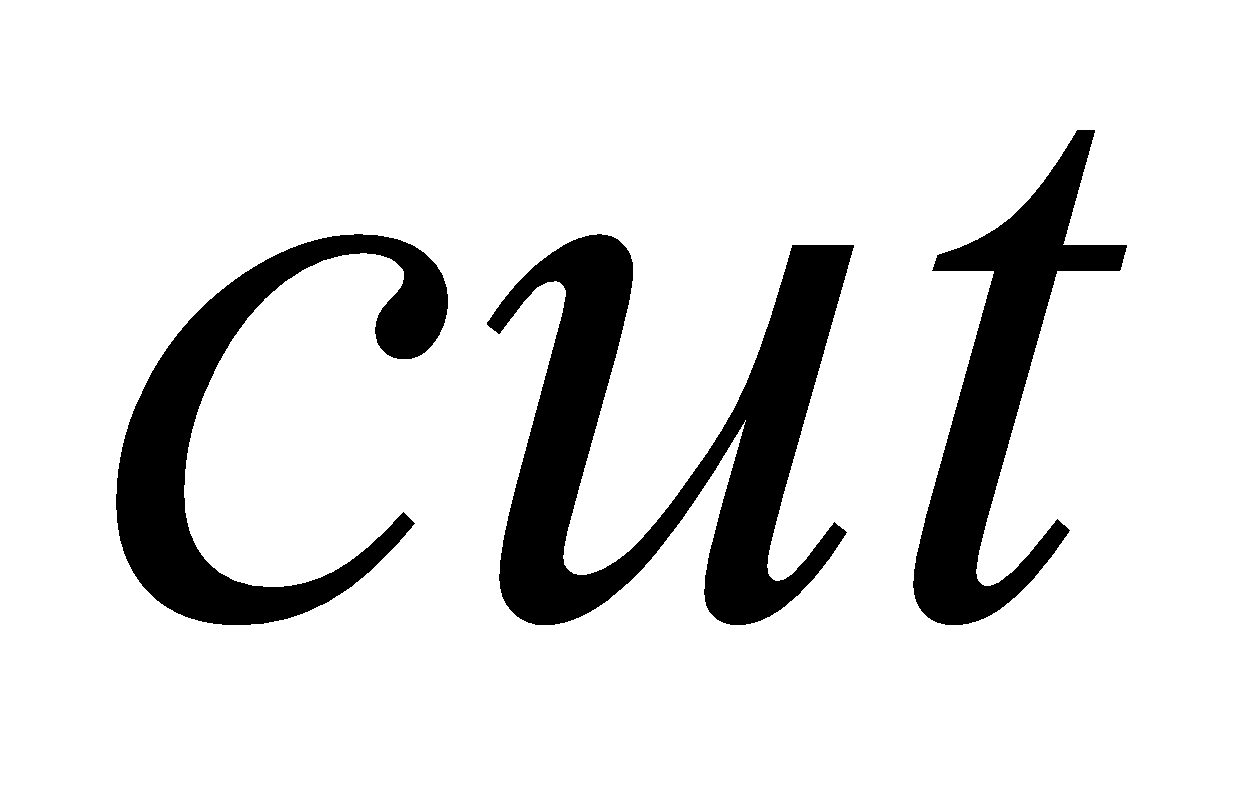
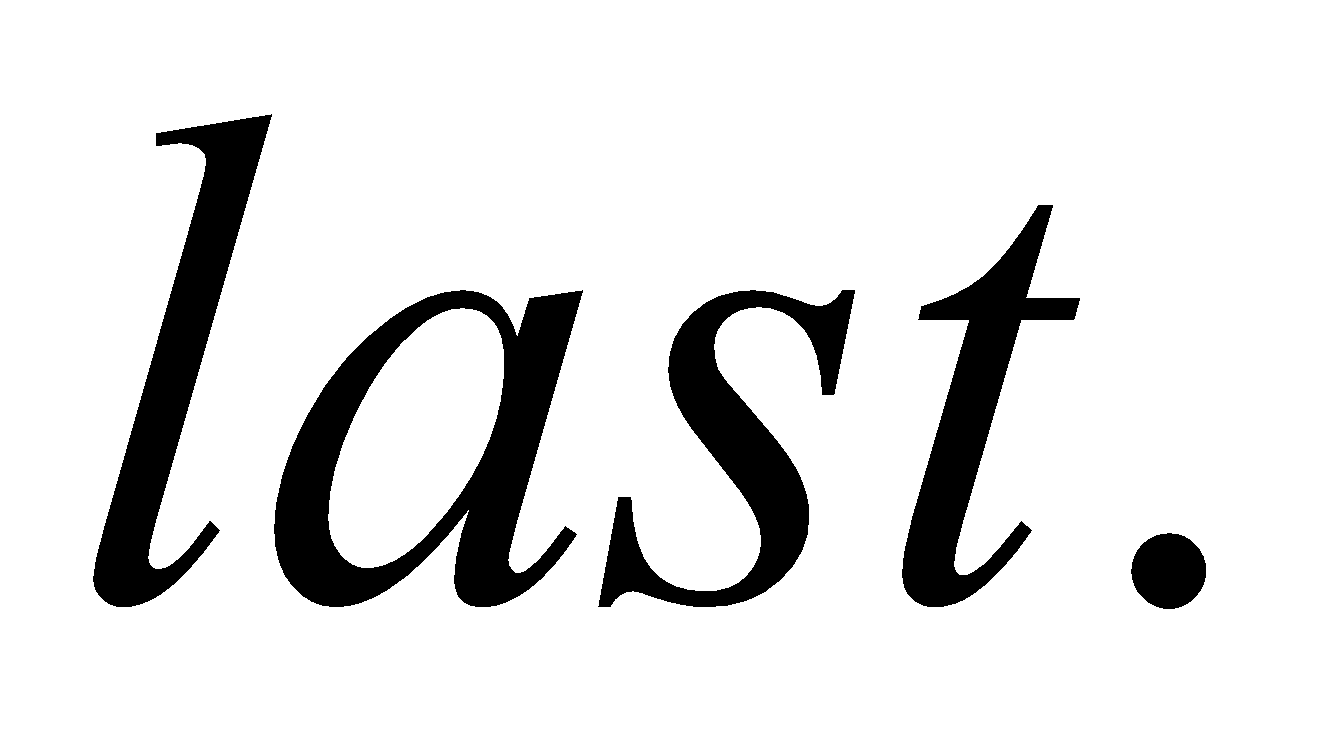


Как видно из задания номер 2 и графиков, при больших значениях k, а соответственно, при небольшой длине строк, метод динамического программирования является выигрышным вариантом по сравнению с методом рекурсии. Это происходит по той причине, что в методе ДП мы должны рассмотреть полиноминальное количество вариантов, пока не найдем решение, а в методе рекурсии перебор является экспоненциальным.

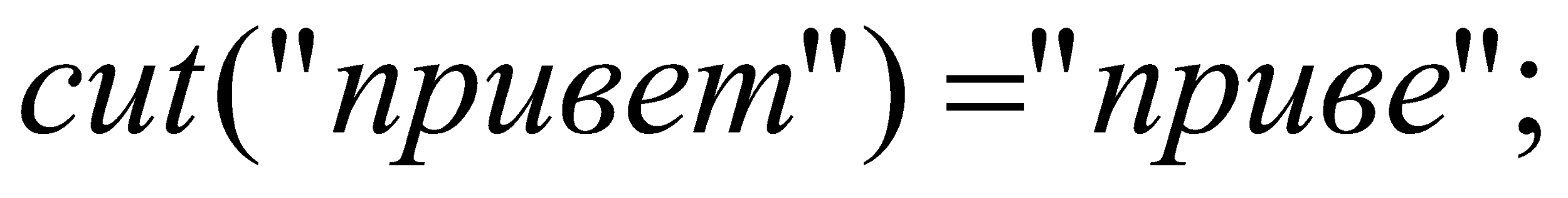
**Задание 4**

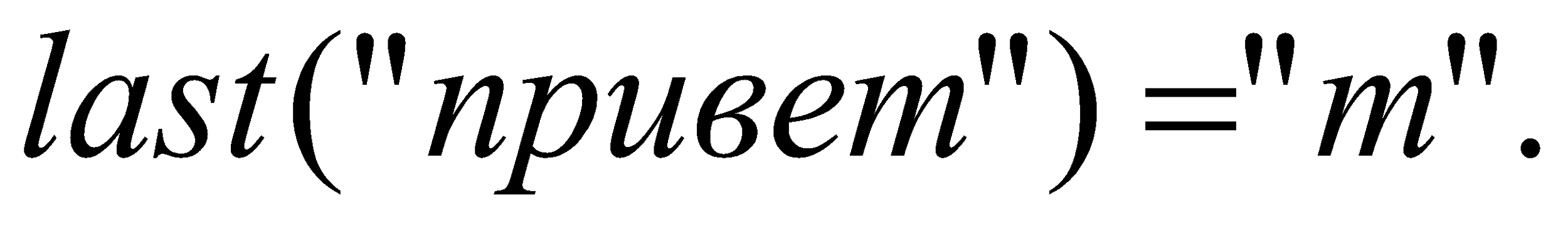
Реализовать вручную пример вычисления дистанции Левенштейна при помощи рекурсивного алгоритма (в соответствии с вариантом) (каждый шаг алгоритма по примеру из лекции вставить в отчет).



В предыдущем выражении используются символы   и  Разъясним их смысл:

 – количество символов в заданной строке. Например, 

 – заданная строка без последнего символа. Например, 

 – последний символ заданной строки. Например, 

Найти расстояние Левенштейна между словами «Кол» и «Столб».

1.  
2.  
3.  
4.  
5.  

 **= 5.**

 **= 4.**

1.  

 **= 4.**

 **= 3.**

1.  
2.  
3.  
4.  

 **= 3.**

 **= 2.**

1.  

 **= 2.**

1.  
2.  

 **= 1**

1.  

 **= 2.**

 **= 1.**

 **= 1.**

1. 
2. 
3. 
4. 
5. 
6. 
7. 
8. 
9. 
10. 
11. 
12. 
13. 
14. 

Дистанция Левенштейна для слов «Кол» и «Столб»: 3.

**Задание 5**

Выполнить сравнительный анализ времени затраченного на решение задачи о наибольшей общей подпоследовательности для двух методов решения (рекурсивное решение, динамическое программирование). Две последовательности взять в соответствии с вариантом. Построить графики зависимости времени вычисления от . **Отобразить ход решения в отчете**(по примеру из лекции) + код и копии экрана.



Ниже представлена функция **lcsd**, реализующая алгоритм динамического программирования для решения задачи о наибольшей общей подпоследовательности (LCS).

**//- LCH.h**

**int lcsd(**

**const char x[],** // последовательность X

**const char y[],** // последовательность Y

**char z[]** // наибольшая общая подпоследовательность

**);**

//- LCH.cpp

**#include "stdafx.h"**

**#include "LCS.h"**

**#define LCS\_C(x1,x2) (C[(x1)\*(leny+1)+(x2)])**

**#define LCS\_B(x1,x2) (B[(x1)\*(leny+1)+(x2)])**

**#define LCS\_X(i) (x[(i)-1])**

**#define LCS\_Y(i) (y[(i)-1])**

**#define LCS\_Z(i) (z[(i)-1])**

**enum Dart{TOP,LEFT,LEFTTOP};**

**void getLCScontent( int lenx, int leny, const char x[],**

**const Dart\* B,**

**int n, int i, int j, char z[])**

**{**

**if ((i > 0 && j > 0 && n > 0 ))**

**{**

**if (LCS\_B(i,j) == LEFTTOP)**

**{**

**getLCScontent(lenx, leny,x, B, n-1, i-1, j-1, z);**

**LCS\_Z(n) = LCS\_X(i);**

**LCS\_Z(n+1) = 0;**

**}**

**else if (LCS\_B(i,j)== TOP)**

**getLCScontent(lenx, leny,x, B, n, i-1, j, z);**

**else getLCScontent(lenx, leny,x, B, n, i, j-1, z);**

**}**

**};**

**int lcsd(const char x[], const char y[], char z[])**

**{**

**int n;**

**int lenx = strlen(x), leny = strlen(x),**

**\*C = new int[(lenx+1)\*(leny+1)];**

**Dart\* B = new Dart[(lenx+1)\*(leny+1)];**

**memset(C,0,sizeof(int)\*(lenx+1)\*(leny+1));**

**for (int i = 1; i <= lenx; i++)**

**for(int j = 1; j <= leny; j++)**

**if (LCS\_X(i) == LCS\_Y(j))**

**{LCS\_C(i,j) = LCS\_C(i-1,j-1)+1;**

**LCS\_B(i,j) = LEFTTOP;}**

**else if (LCS\_C(i-1,j) >= LCS\_C(i, j-1))**

**{**

**LCS\_C(i,j) = LCS\_C(i-1, j);**

**LCS\_B(i,j) = TOP;**

**}**

**else**

**{**

**LCS\_C(i,j) = LCS\_C(i, j-1);**

**LCS\_B(i,j) = LEFT;**

**}**

**getLCScontent(lenx, leny, x, B, LCS\_C(lenx,leny), lenx, leny, z);**

**return LCS\_C(lenx,leny);**

**}**

**#undef LCS\_Z**

**#undef LCS\_C**

**#undef LCS\_B**

**#undef LCS\_X**

**#undef LCS\_Y**

Функция **lcsd** имеет три параметра: **x** (символьная строка, интерпретируемая как первая заданная последовательность), **y** (символьная строка, интерпретируемая как вторая заданная последовательность) и возвращаемый параметр **z** (символьная строка, интерпретируемая как LCS двух последовательностей, заданных первыми двумя параметрами). Функция возвращает длину LCS.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **R** | **Q** | **T** | **W** | **Y** | **K** |
|  | **0** | **0** | **0** | **0** | **0** | **0** | **0** |
| **Q** | **0** | **0** | **1** | **1** | **1** | **1** | **1** |
| **V** | **0** | **0** | **1** | **1** | **1** | **1** | **1** |
| **T** | **0** | **0** | **1** | **2** | **2** | **2** | **2** |
| **W** | **0** | **0** | **1** | **2** | **3** | **3** | **3** |
| **N** | **0** | **0** | **1** | **2** | **3** | **3** | **3** |
| **H** | **0** | 0 | **1** | **2** | **3** | **3** | **3** |
| **O** | **0** | **0** | **1** | **2** | **3** | **3** | **3** |

Порядок заполнения таблицы:

– первая строка и первый столбец таблицы заполняются нулями;

– каждый элемент последовательно заполняется по формуле для c[i, j]:

если символы для позиции i, j совпадают, то в неё записывается значение c[i−1, j −1]+1, иначе вычисляется максимум от соседей слева и сверху.

Элемент в правом нижнем углу показывает длину наибольшей общей

подпоследовательности.

Вторая матрица заполняется следующим образом: все ячейки кроме технических строк заполняются стрелками, направленными вверх.

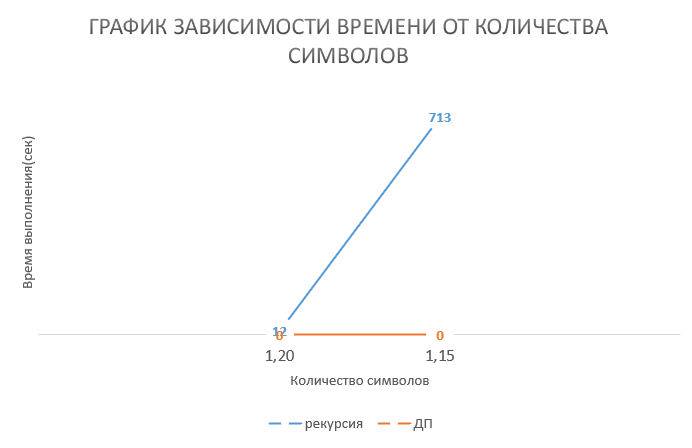
Если символы для позиции i, j совпадают, то стрелка меняется на лево-вверх.

Если числовое значение от соседа слева больше, чем от соседа сверху, то стрелка меняется на лево.

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  |  | **R** | **Q** | **T** | **W** | **Y** | **K** |
|  |  |  |  |  |  |  |  |
| **Q** |  |  |  |  |  |  |  |
| **V** |  |  |  |  |  |  |  |
| **T** |  |  |  |  |  |  |  |
| **W** |  |  |  |  |  |  |  |
| **N** |  |  |  |  |  |  |  |
| **H** |  |  |  |  |  |  |  |
| **O** |  |  |  |  |  |  |  |

### Длина наибольшей общей подпоследовательности – 3.

### 



Вывод: освоила общие принципы решения задач методом динамического программирования.