Белорусский государственный технологический университет

Кафедра Программной инженерии

**“Математическое программирование”**

**Отчет по лабораторной работе №4**

**Динамическое программирование**

**Вариант 15**

Выполнила: Ковалев А.А.

ФИТ 2 курс, 4 группа

Минск 2021

**Лабораторная работа 4**

**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

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

**Ход выполнения работы**

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

**Решение:**

#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]; // 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)];

}

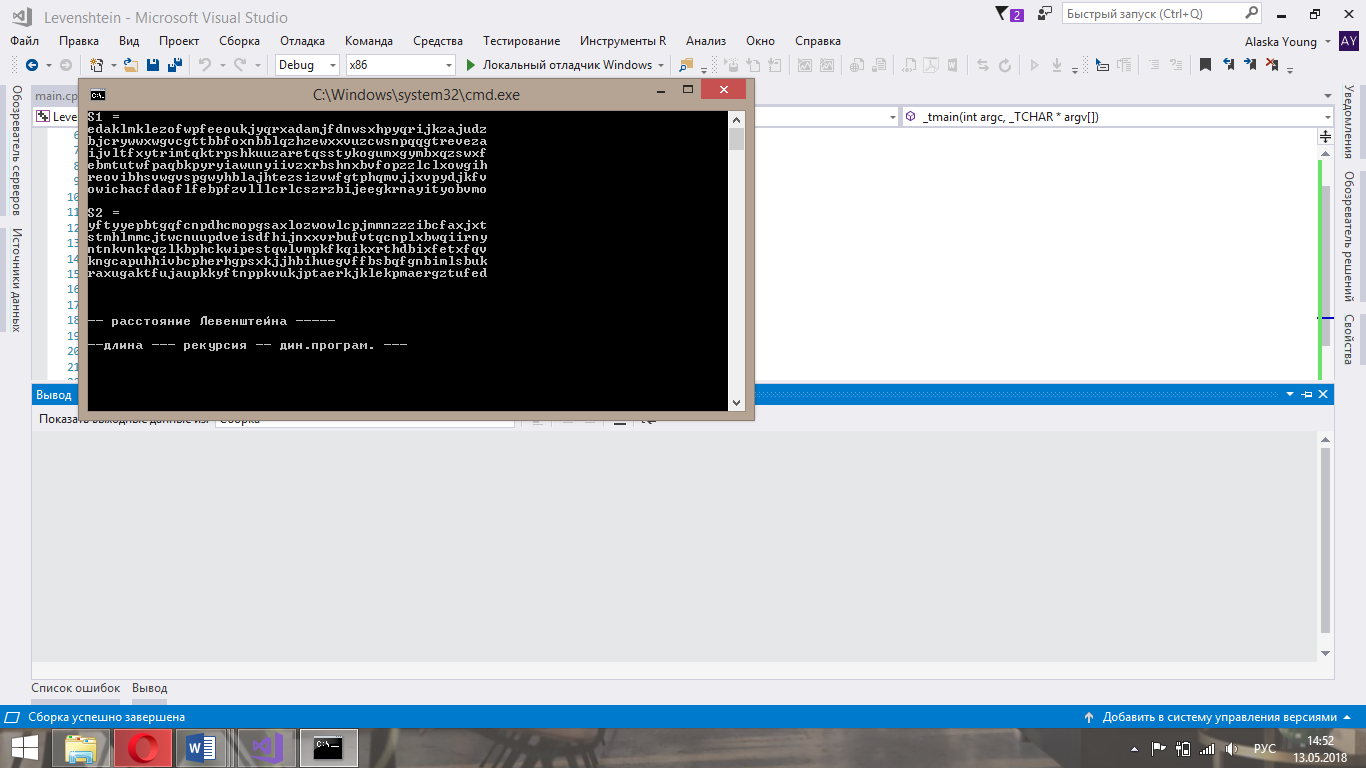


Рис.1 – пример генерации строк

**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).

**Решение:**

Ниже приведены варианты реализации нахождения дистанции Левенштейна при помощи динамического программирования и при помощи рекурсивного алгоритма.

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 5 представлены дистанции Левенштейна вычисленные при помощи метода динамического программирования, а также рекурсивным алгоритмом.

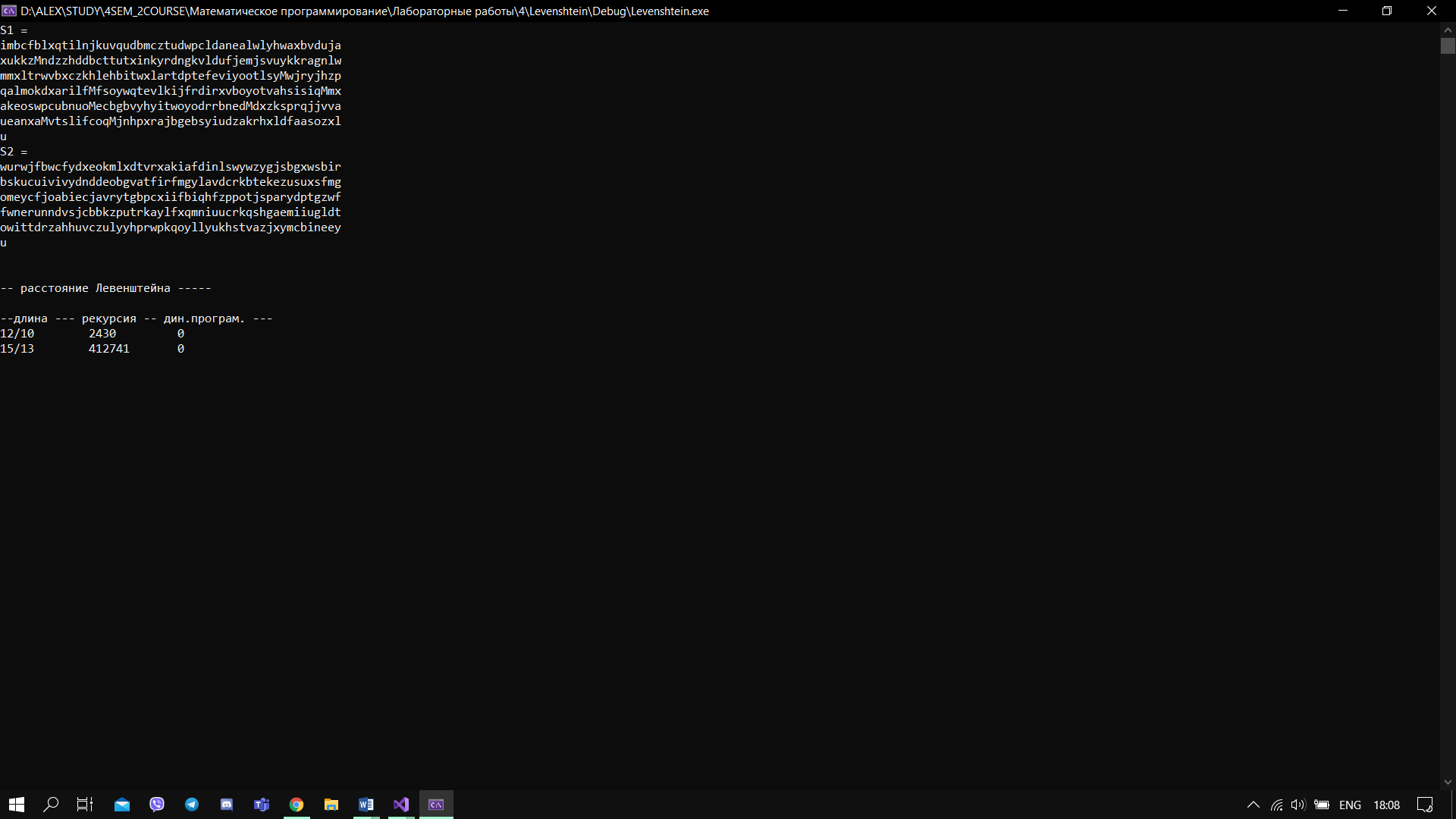


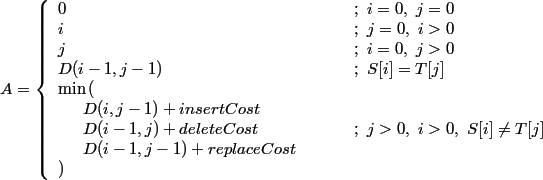
Рис. 2 – проверка работоспособности решений

**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).

**Решение:**

На графике можно заметить, что выполнение с помощью динамического алгоритма, вычисления производятся в разы быстрее, чем с помощью рекурсивного алгоритма.

**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).



|  |  |
| --- | --- |
| Задание 4 | |
| Сын | Сонар |

**Решение:**

1. L(«Сын», «Сонар») = min

2. L(«Сы», «Сонар») = min

3. L(«Сын», «Сона») = min

4. L(«Сы», «Сона») = min

5. L(«С», «Сонар») = min

L(«», «Сонар») = 5,

L(«», «Сона») = 4

6. L(«С», «Сона») = min

L(«», «Сона») = 4,

L(«», «») = 3

7. L(«Сын», «Сон») = min

8. L(«Сы», «Сон») = min

9. L(«Сын», «Со») = min

10. L(«Сын», «С») = min

L(«Сын», «») = 3,

L(«Сы», «») = 2,

11. L(«Сы», «Со») = min

12. L(«С», «Со») = min

L(«», «Со») = 2,

L(«», «С») = 1,

13. L(«Сы», «С») = min

L(«Сы», «») = 2,

L(«С», «») = 1

14. L(«С», «Сон») = min

L(«», «Сон») = 3,

L(«», «Со») = 2,

15. L(«С», «С») = min

L(«», «С») = 1,

L(«С», «») = 1,

16. L(«Сы», «С») = min (2, 3, 2) = 2

17. L(«С», «Со») = min (3, 2, 2) = 2

18. L(«Сы», «Со») = min (3, 3, 2) = 2

19. L(«С», «Сон») = min (4, 3, 3) = 3

20. L(«Сын», «С») = min (3, 4, 3) = 3

21. L(«Сын», «Со») = min (3, 4, 3) = 3

22. L(«Сы», «Сон») = min (4, 3, 3) = 3

23. L(«Сын», «Сон») = min (4, 4, 2) = 2

24. L(«С», «Сона») = min (5, 4, 4) = 4

25. L(«С», «Сонар») = min (6, 5, 5) = 5

26. L(«Сы», «Сона») = min (5, 4, 4) = 4

27. L(«Сын», «Сона») = min (5, 3, 4) = 3

28. L(«Сы», «Сонар») = min (6, 5, 5) = 5

29. L(«Сын», «Сонар») = min (6, 4, 3) = 3

Дистанция Левенштейна для слов «Сын» и «Сонар»: 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.

Дано:

|  |  |
| --- | --- |
| MIOPLKJ | GUIOLW |

**Решение:**

Программный код:

#include <iostream>

#include "LCS.h"

#include "LCH.h"

#include <ctime>

int main()

{

clock\_t t1 = 0;

clock\_t t2 = 0;

clock\_t t3 = 0;

clock\_t t4 = 0;

setlocale(LC\_ALL, "rus");

char X[] = "MIOPLKJ", Y[] = "GUIOLW";

std::cout << std::endl << "-- вычисление длины LCS для X и Y(рекурсия)";

std::cout << std::endl << "-- последовательность X: " << X;

std::cout << std::endl << "-- последовательность Y: " << Y;

t1 = clock();

int s = lcs(

sizeof(X) - 1, // длина последовательности X

"MIOPLKJ", // последовательность X

sizeof(Y) - 1, // длина последовательности Y

"GUIOLW" // последовательность Y

);

t2 = clock();

std::cout << std::endl << "-- длина LCS: " << s << std::endl;

std::cout << std::endl << "затраченное время(сек): " << ((double)(t2 - t1)) / ((double)(clock\_t)1000) << std::endl;

char z[100] = "";

t3 = clock();

int l = lcsd(X, Y, z);

t4 = clock();

std::cout << std::endl

<< "-- наибольшая общая подпоследовательость - LCS(динамическое"

<< "программирование)" << std::endl;

std::cout << std::endl << "последовательость X: " << X;

std::cout << std::endl << "последовательость Y: " << Y;

std::cout << std::endl << " LCS: " << z;

std::cout << std::endl << " длина LCS: " << l;

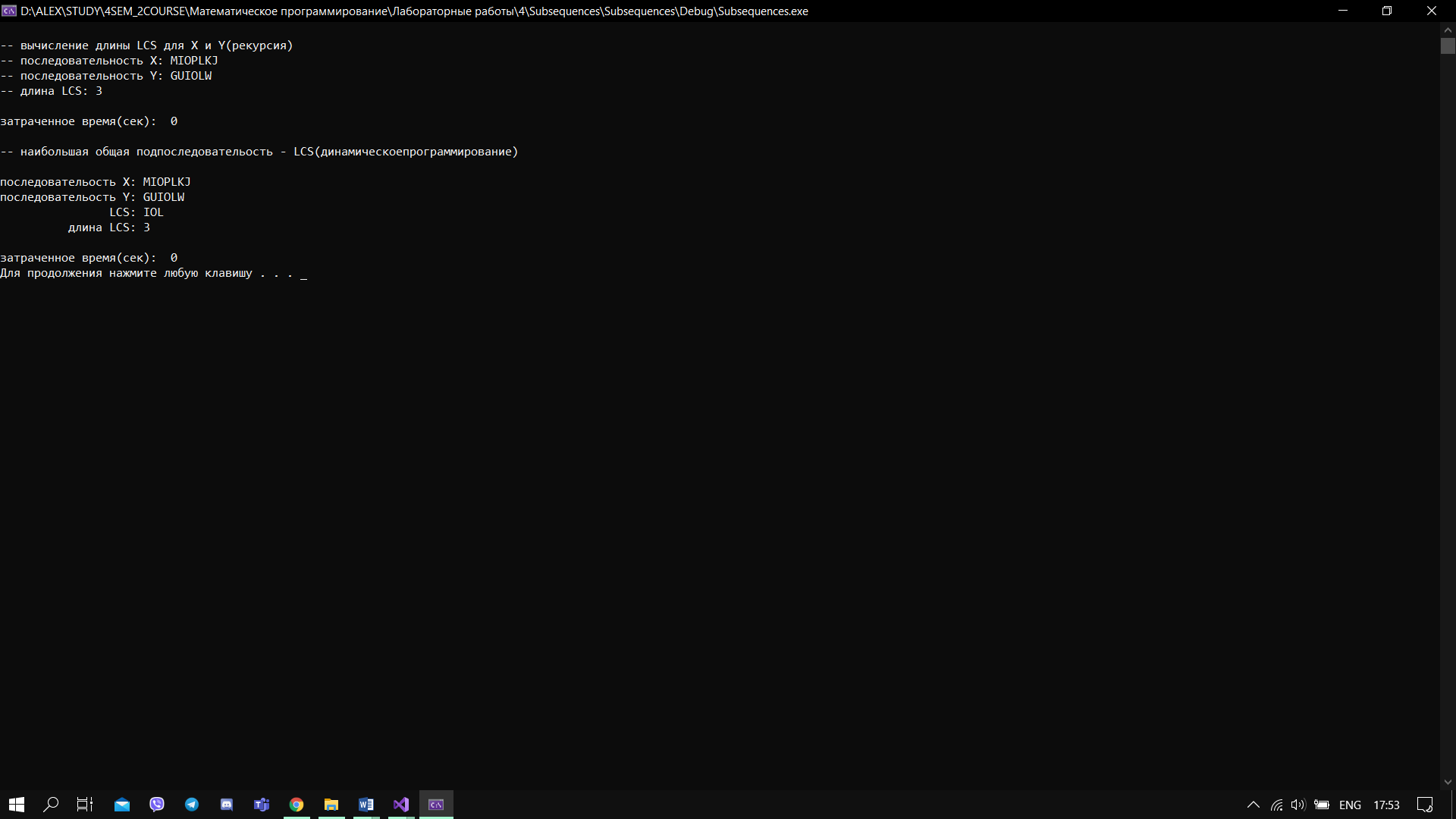
std::cout << std::endl;

std::cout << std::endl << "затраченное время(сек): " << ((double)(t4 - t3)) / ((double)(clock\_t)1000) << std::endl;

system("pause");

return 0;

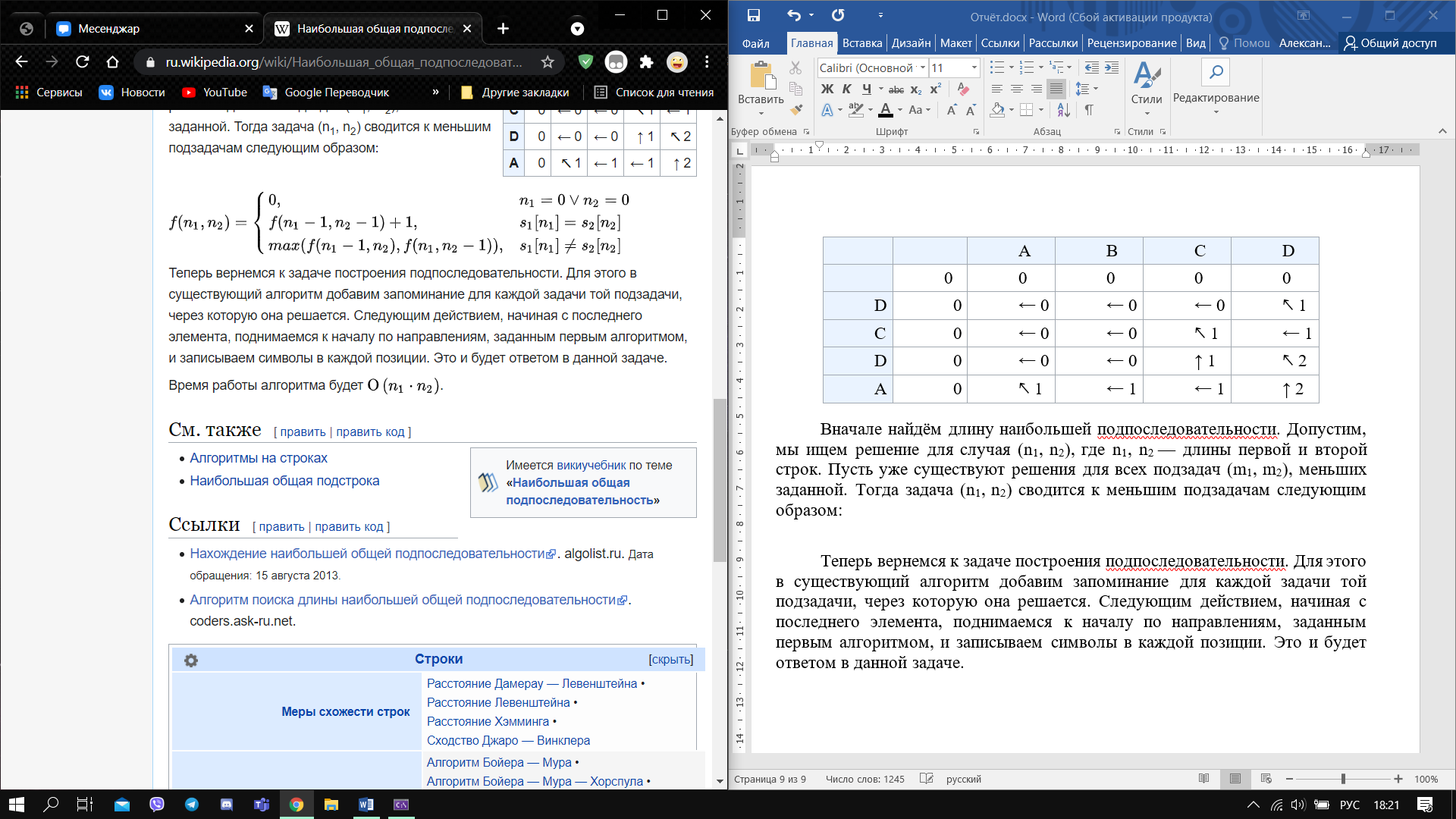
}



Решение для случая k строк

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  |  | A | B | C | D |
|  | 0 | 0 | 0 | 0 | 0 |
| D | 0 | ← 0 | ← 0 | ← 0 | ↖ 1 |
| C | 0 | ← 0 | ← 0 | ↖ 1 | ← 1 |
| D | 0 | ← 0 | ← 0 | ↑ 1 | ↖ 2 |
| A | 0 | ↖ 1 | ← 1 | ← 1 | ↑ 2 |

Вначале найдём длину наибольшей подпоследовательности. Допустим, мы ищем решение для случая (n1, n2), где n1, n2 — длины первой и второй строк. Пусть уже существуют решения для всех подзадач (m1, m2), меньших заданной. Тогда задача (n1, n2) сводится к меньшим подзадачам следующим образом:

{\displaystyle f(n\_{1},n\_{2})=\left\{{\begin{array}{ll}0,&n\_{1}=0\lor n\_{2}=0\\f(n\_{1}-1,n\_{2}-1)+1,&s\_{1}[n\_{1}]=s\_{2}[n\_{2}]\\max(f(n\_{1}-1,n\_{2}),f(n\_{1},n\_{2}-1)),&s\_{1}[n\_{1}]\neq s\_{2}[n\_{2}]\end{array}}\right.} 

Теперь вернемся к задаче построения подпоследовательности. Для этого в существующий алгоритм добавим запоминание для каждой задачи той подзадачи, через которую она решается. Следующим действием, начиная с последнего элемента, поднимаемся к началу по направлениям, заданным первым алгоритмом, и записываем символы в каждой позиции. Это и будет ответом в данной задаче.