Учреждение образования

«Белорусский государственный технологический университет»

**Кафедра информационных систем и технологий**

**Лабораторная работа №4**

**Тема «**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**»**

Вариант 8

Выполнила:

Студентка 2 курса 7 группы ФИТ

Курносенко Софья Андреевна

Проверил:

Барковский Евгений Валерьевич

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

**ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:**

**Задание 1.**

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

Random.h

#pragma once

#include <string>

std::string generRand(int len);

Random.cpp

#include "Random.h"

#include <iostream>

using namespace std;

string generRand(int len) {

int min = 97, max = 122; // в ascii с 97 по 122 символ идут буквы латинского алфавита

string str = "";

int AlphabetLetterNum;

for (int i = 0; i < len; i++) {

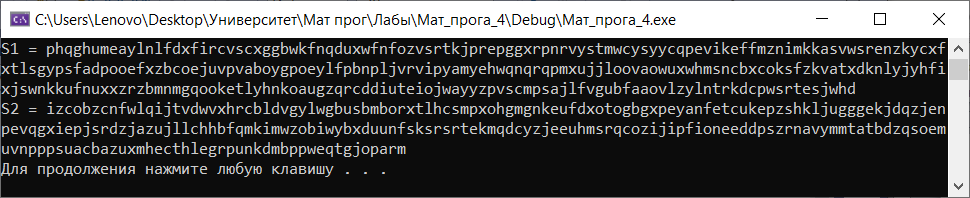
AlphabetLetterNum = (rand() % (max - min + 1) + min);

str.push\_back((char)AlphabetLetterNum);

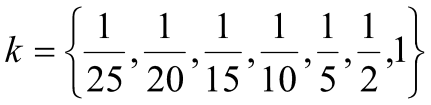
}

return str;

}



**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).

Levenshtein.h

// - Levenshtein.h

// -- дистанция Левенштeйна (динамическое программирование)

int levenshtein(

int lx, // длина слова x

std::string x, // слово x

int ly, // длина слова y

std::string y // слово y

);

// -- дистанция Левенштeйна (рекурсия)

int levenshtein\_r(

int lx, // длина строки x

std::string x, // строка x

int ly, // длина строки y

std::string y // строка y

);

Levenshtein.cpp

// - Levenshtein.cpp

#include <iomanip>

#include <algorithm>

#include "Levenshtein.h"

#include <string>

#define DD(i,j) d[(i)\*(ly+1)+(j)]

//------- Реализация через динамическое программирование

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

{

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

}

int levenshtein(

int lx, std::string x,

int ly, std::string y

)

{

int\* d = new int[(lx + 1) \* (ly + 1)];

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

for (int j = 0; j <= ly; j++) DD(0, j) = j;

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

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

{

DD(i, j) = min3(DD(i - 1, j) + 1, DD(i, j - 1) + 1,

DD(i - 1, j - 1) + (x[i - 1] == y[j - 1] ? 0 : 1));

}

return DD(lx, ly);

}

//------- Реализация через рекурсивное программирование

int levenshtein\_r(

int lx, std::string x,

int ly, std::string 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,

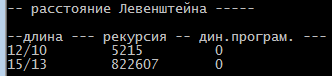
levenshtein\_r(lx - 1, x, ly - 1, y) + (x[lx - 1] == y[ly - 1] ? 0 : 1)

);

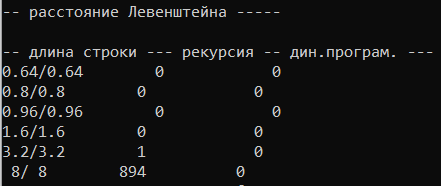
return rc;

};

Со значениями из задания будет медленно производиться расчет:



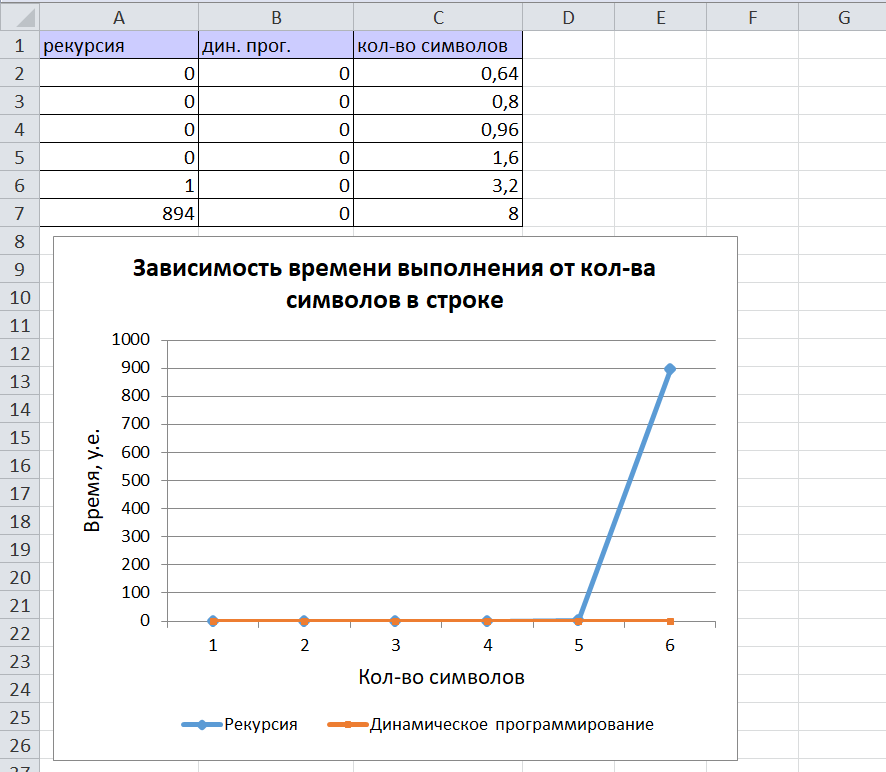
Так что поменяем значения на меньшие:



**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).

На графике, представленном на рисунке ниже, можно заметить, что вычисления, выполненные с помощью динамического алгоритма, 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).

|  |  |  |
| --- | --- | --- |
| Вариант | Задание 4 | |
| 8 | Вар | Баран |

1. L(“Вар”, “Баран”) = min

Убираем последний символ первого слова, во второй строке – второго, в третьей – последние символы обоих слов.

1. L(“Ва”, “Баран”) = min

Последовательно рассматриваем строки из функции min пункта 1. То есть значение L(“Ва”, “Баран”) взято из пункта 1 из min.

1. L() = min
2. L() = min

Все значения из пункта 1 рассмотрены, рассматриваем значения из пункта 2, затем 3, затем 4 и т.д., а *если они уже были рассмотрены ранее* – пропускаем их (например, есть и в пункте 1, и в пункте 2, но все значения первого пункта уже были рассмотрены, таким образом мы его повторно не рассматриваем).

1. L() = min

= 5 – сколько операций вставки нужно провести над строкой “”, чтобы получить строку ”Баран”

= 4

Первое и последнее значение из данного пункта не будет рассматриваться, так как *в них фигурирует пустая строка*. То есть нам не нужно брать эти строки и высчитывать для них min.

1. L() = min

= 4

= 3

1. L() = min
2. L(“Ва”, “Бар”) = min
3. L() = min

= 3

= 2

1. L() = min
2. L() = min
3. L() = min

2

1

1. L() = min

= 3

= 2

1. L() = min
2. L() = min

= 1

= 1

Все значения без пустых строк рассмотрены. Движемся снизу вверх (от 15 пункта к 1).

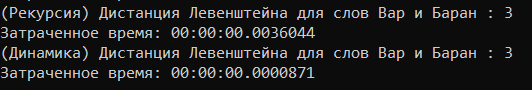
1. L() = min(2, 2, 1) = 1 – числа из пункта 15 суммированные с 1, из них выбираем наименьшее число
2. L() = min(2, 3, 2) = 2

Значение L() взято из пункта 16.

1. L() = min(3, 4, 3) = 3
2. L() = min(3, 2, 2) = 2
3. L() = min(3, 3, 2) = 2
4. L() = min(3, 4, 3) = 3
5. L() = min(4, 3, 3) = 3
6. L(“Ва”, “Бар”) = min(4, 3, 3) = 3
7. L() = min(4, 3, 3) = 3
8. L() = min(4, 4, 3) = 3
9. L() = min(5, 4, 4) = 4
10. L() = min(5, 4 , 4) = 4
11. L() = min(4, 3, 3) = 3
12. L() = min(3, 3, 3) = 3
13. L(“Ва”, “Баран”) = min(4, 3, 4) = 3
14. L(“Вар”, “Баран”) = min(5, 3, 4) = 3

(31 пункт правильный, а в предыдущих пунктах с расчетами напутано, сорян)

Программно:



**Задание 5.**

Выполнить сравнительный анализ времени затраченного на решение задачи об оптимальной расстановке скобок при умножении нескольких матриц для двух методов решения (рекурсивное решение, динамическое программирование). Размерность матриц взять в соответствии с вариантом. Объяснить в отчете принцип расставления скобок по итоговой матрице + код + копии экрана.

|  |  |
| --- | --- |
| Вариант | Задание 5 |
| 8 | 10\*15, 15\*80, 80\*23, 23\*50, 50\*40, 40\*71 |

Matrix.h

// расстановка скобок

#pragma once

// расстановка скобок при умножении матриц

// функции возвращают минимальное количество операций умножения

#define OPTIMALM\_PARM(x) ((int\*)x) // для представления 2мерного массива

// рекурсия

int OptimalM(

int i, // [in] номер первой матрицы

int j, // [in] номер последней матрицы

int n, // [in] количество матриц

const int c[], // [in] массив размерностей

int\* s // [out] результат: позиции скобок

);

// динамическое программирование

int OptimalMD(

int n, // [in] количество матриц

const int c[], // [in] массив размерностей

int\* s // [out] результат: позиции скобок

);

Matrix.cpp

#include "Matrix.h"

// расстановка скобок (рекурсия)

#define INFINITY 0x7fffffff

#define NINFINITY 0x80000000

int OptimalM(int i, int j, int n, const int c[], int\* s) {

#define OPTIMALM\_S(x1,x2) (s[(x1-1)\*n+x2-1])

int o = INFINITY;

int bo = INFINITY;

if (i < j) {

for (int k = i; k < j; k++) {

bo = OptimalM(i, k, n, c, s) + OptimalM(k + 1, j, n, c, s) + c[i - 1] \* c[k] \* c[j];

if (bo < o) {

o = bo;

OPTIMALM\_S(i, j) = k;

}

}

}

else o = 0;

return o;

#undef OPTIMALM\_S

};

// расстановка скобок (динамическое программирование)

int OptimalMD(int n, const int c[], int\* s) {

#define OPTIMALM\_S(x1,x2) (s[(x1-1)\*n+x2-1])

#define OPTIMALM\_M(x1,x2) (M[(x1-1)\*n+x2-1])

int\* M = new int[n \* n], j = 0, q = 0;

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

OPTIMALM\_M(i, i) = 0;

for (int l = 2; l <= n; l++) {

for (int i = 1; i <= n - l + 1; i++) {

j = i + l - 1;

OPTIMALM\_M(i, j) = INFINITY;

for (int k = i; k <= j - 1; k++) {

q = OPTIMALM\_M(i, k) + OPTIMALM\_M(k + 1, j) + c[i - 1] \* c[k] \* c[j];

if (q < OPTIMALM\_M(i, j)) {

OPTIMALM\_M(i, j) = q;

OPTIMALM\_S(i, j) = k;

}

}

}

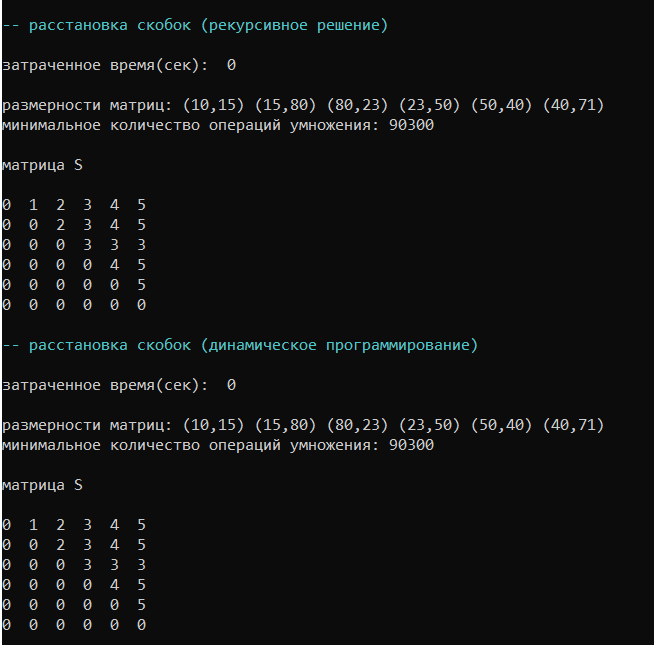
}

return OPTIMALM\_M(1, n);

#undef OPTIMALM\_M

#undef OPTIMALM\_S

};



**Принцип расстановки скобок по итоговой матрице:**

Скобки расставляются по принципу «сначала внешние – затем внутренние». Имеется 6 матриц, вот их размерность:

А1 = 10\*15,

А2 = 15\*80,

А3 = 80\*23,

А4 = 23\*50,

А5 = 50\*40,

А6 = 40\*71.

Расстановку скобок в заданной последовательности матриц можно осуществить с помощью двумерного массива S (матрицы), возвращаемого функцией в конце каждого метода решения.

Матрица S:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 0 | 1 | 2 | 3 | 4 | 5 |
| **2** | 0 | 0 | 2 | 3 | 4 | 5 |
| **3** | 0 | 0 | 0 | 3 | 3 | 3 |
| **4** | 0 | 0 | 0 | 0 | 4 | 5 |
| **5** | 0 | 0 | 0 | 0 | 0 | 5 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 |

Найдем элемент (1,6) в матрице S, он равен 5. Это означает, что точка разрыва между 1-ой и 6-ой матрицей находится после 5-ой матрицы. Что позволяет расставить скобки следующим образом:

(A1\*A2\*A3\*A4\*A5)\*A6

Точку разрыва между 1 и 5 матрицей определяет элемент (1,5). Он равен 4. Следовательно разрыв будет после четвертой матрицы.

((A1\*A2\*A3\*A4)\*A5)\*A6

Далее берем элемент (1,4) и получаем, что он равен 3. Следовательно:

(((A1\*A2\*A3)\*A4)\*A5)\*A6

И на последнем шаге мы возьмем элемент (1,3) и он равен 2:

((((A1\*A2)\*A3)\*A4)\*A5)\*A6

Это выражение и есть конечное.

Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 90300.

**Проверка**

Как известно из высшей математики, умножение матриц ассоциативно, то есть результат перемножения зависит только от порядка матриц и не зависит от расстановки скобок:

*(А\*В)\*С = А\*(В\*С).*

Результат перемножения от расстановки скобок не зависит, зато *трудоемкость* этого перемножения при разных расстановках скобок может отличаться существенно.

Оценим трудоемкость умножения двух матриц *A(p×q)*и *B(q×r)*:

*C(p×r) = A(p×q)\* B(q×r),*

где *C(p×r)* — итоговая матрица.

При этом трудоемкость перемножения двух матриц: *T = p·q·r.*

Рассчитаем трудоемкость от перемножения матриц:

А1 = 10\*15,

А2 = 15\*80,

А3 = 80\*23,

А4 = 23\*50,

А5 = 50\*40,

А6 = 40\*71.

При расставлении скобок:

((((A1\*A2)\*A3)\*A4)\*A5)\*A6

1. A1\*A2 = 10*×*15\*15*×*80 =>

размерность итоговой матрицы = 10*×* 80

трудоемкость = 10\*15\*80 = 12000

1. (A1\*A2)\*A3 = 10×80 \* 80×23 =>

размерность итоговой матрицы = 10*×* 23

трудоемкость = 10\*80\*23 = 18400

1. ((A1\*A2)\*A3)\*A4 = 10*×* 23 \* 23×50 =>

размерность итоговой матрицы = 10*×* 50

трудоемкость = 10\*23\*50 = 11500

1. (((A1\*A2)\*A3)\*A4)\*A5 = 10*×* 50 \* 50×40 =>

размерность итоговой матрицы = 10*×* 40

трудоемкость = 10\*50\*40 = 20000

1. ((((A1\*A2)\*A3)\*A4)\*A5)\*A6 = 10×40 \* 40×71 =>

размерность итоговой матрицы = 10*×* 71

трудоемкость = 10\*40\*71 = 28400

Для получения итогового значения складываем трудоемкости, полученные в этих пяти пунктах:

*Трудоемкость = 12000+18400+11500+20000+28400 = 90300*

Ответ сходиться с тем, что получен программно – значит, скобки были расставлены правильно.

Main.cpp

// --- main

#include <iostream>

#include "Random.h"

#include "Levenshtein.h"

#include "Matrix.h"

#include <iomanip>

using namespace std;

#define N 6

int main()

{

setlocale(LC\_ALL, "rus");

// ------------1

string S1 = generRand(300);

string S2 = generRand(250);

cout << "S1 = " << S1 << endl << "S2 = " << S2 << endl;

// ------------2

clock\_t t1 = 0, t2 = 0, t3 = 0, t4 = 0;

int lx = S1.length();

int ly = S2.length();

// должно быть так, но будет считать слишком медленно

//int x[]{ 12,15,20,30,60,150,300 };

//int y[]{ 10,13,17,25,50,125,250 };

// поэтому берем значения поменьше

float x[]{ 0.64, 0.8, 0.96, 1.6, 3.2, 8 };

float y[]{ 0.64, 0.8, 0.96, 1.6, 3.2, 8 };

cout << "\n\n-- расстояние Левенштейна -----";

cout << "\n\n-- длина строки --- рекурсия -- дин.програм. ---\n";

for (int i = 0; i < min(sizeof(x)/sizeof(float), sizeof(y) / sizeof(float)); i++) {

// рекурсия

t1 = clock();

levenshtein\_r(x[i], S1, y[i], S2);

t2 = clock();

// дин.програм.

t3 = clock();

levenshtein(x[i], S1, y[i], S2);

t4 = clock();

cout << right << setw(2) << x[i] << "/" << setw(2) << y[i]

<< " " << left << setw(10) << (t2 - t1)

<< " " << setw(10) << (t4 - t3) << endl;

}

// ------------5

// N - в define выше

int Mc[N + 1] = { 10, 15, 80, 23, 50, 40, 71 },

Ms[N][N],

r = 0,

rd = 0;

// Ms - массив массивов, заполняем его нулями

memset(Ms, 0, sizeof(int) \* N \* N); // заполнить первые sizeof(int) \* N \* N байт значением 0

t1 = clock();

r = OptimalM(1, N, N, Mc, OPTIMALM\_PARM(Ms));

t2 = clock();

setlocale(LC\_ALL, "rus");

cout << endl;

cout << endl << "-- расстановка скобок (рекурсивное решение) " << endl;

cout << endl << "затраченное время(сек): " << ((double)(t2 - t1)) / ((double)CLOCKS\_PER\_SEC) << endl;

cout << endl << "размерности матриц: ";

for (int i = 1; i <= N; i++) cout << "(" << Mc[i - 1] << "," << Mc[i] << ") ";

cout << endl << "минимальное количество операций умножения: " << r;

cout << endl << endl << "матрица S" << endl;

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

{

cout << endl;

for (int j = 0; j < N; j++) cout << Ms[i][j] << " ";

}

cout << endl;

memset(Ms, 0, sizeof(int) \* N \* N);

t3 = clock();

rd = OptimalMD(N, Mc, OPTIMALM\_PARM(Ms));

t4 = clock();

cout << endl

<< "-- расстановка скобок (динамическое программирование) " << endl;

cout << endl << "затраченное время(сек): " << ((double)(t4 - t3)) / ((double)CLOCKS\_PER\_SEC) << endl;

cout << endl << "размерности матриц: ";

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

cout << "(" << Mc[i - 1] << "," << Mc[i] << ") ";

cout << endl << "минимальное количество операций умножения: "

<< rd;

cout << endl << endl << "матрица S" << endl;

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

{

cout << endl;

for (int j = 0; j < N; j++) cout << Ms[i][j] << " ";

}

cout << endl << endl;

cout << endl;

system("pause");

return 0;

}

**Вывод:** освоила общие принципы решения задач методом динамического программирования, сравнила полученные решения задач с рекурсивным методом.