Министерство образование Республики Беларусь

Учреждение образования «Белорусский государственный технологический университет»

Факультет информационных технологий

Кафедра информационных систем и технологий

**Курс «Математическое программирование»**

**Отчет по лабораторной работе №4**

**Динамическое программирование**

**Вариант 10**

Выполнила: студентка факультета ИТ

2 курс 4 группа

Прихач А. А.

Проверил: Бракович А. И.

Минск 2017

**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 .

//GenerationString.h

#pragma once

#include <iostream>

#include <vector>

#include <string>

class GenerationString

{

public:

std::string Generation(int size);

void Start();

private:

std::string temp;

};

//GenerationString.cpp

#include "GenerationString.h"

#include <random>

#include <ctime>

std::string GenerationString::Generation(int size)

{

this->temp.clear();

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

{

this->temp.push\_back(rand() % 26 + 97);

}

return this->temp;

}

void GenerationString::Start()

{

srand(time(NULL));

}

**Задание 2.** Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – Левенштейна для , где - длина строки ,  - строка, состоящая из первых  символов строки . (копии экрана и код вставить в отчет).

//DistationLeven.h

#include <iostream>

#include <string>

class DistationLeven

{

public:

int GetDistationRecursion(std::string str\_1, int size\_1, std::string str\_2, int size\_2);

int GetDistationDinamic(std::string str\_1, int size\_1, std::string str\_2, int size\_2);

private:

int ReplaceCoast(char first, char second);

int Min(int value\_1, int value\_2, int value\_3);

};

Рекурсивный алгоритм:

int DistationLeven::GetDistationRecursion(std::string str\_1, int size\_1, std::string str\_2, int size\_2)

{

int result = 0;

if (size\_1 == NULL)

{

result = size\_2;

}

else if (size\_2 == NULL)

{

result = size\_1;

}

else if ((size\_1 == 1) && (size\_2 == 1) &&

(str\_1[0] == str\_2[0]))

{

result = NULL;

}

else if ((size\_1 == 1) && (size\_2 == 1) &&

(str\_1[0] != str\_2[0]))

{

result = 1;

}

else

{

result = this->Min(

GetDistationRecursion(str\_1.substr(0, size\_1 - 1), str\_1.substr(0, size\_1 - 1).length(), str\_2, str\_2.length()) + 1,

GetDistationRecursion(str\_1, str\_1.length(), str\_2.substr(0, size\_2 - 1), str\_2.substr(0, size\_2 - 1).length()) + 1,

GetDistationRecursion(str\_1.substr(0, size\_1 - 1), str\_1.substr(0, size\_1 - 1).length(),

str\_2.substr(0, size\_2 - 1), str\_2.substr(0, size\_2 - 1).length()) + this->ReplaceCoast(str\_1[str\_1.length() - 1], str\_2[str\_2.length() - 1]));

}

return result;

}

Динамическое программирование:

int DistationLeven::GetDistationDinamic(std::string str\_1, int size\_1, std::string str\_2, int size\_2)

{

std::vector<std::vector<int>> temp(size\_1 + 1, std::vector<int>(size\_2 + 1));

temp[0][0] = NULL;

for (int i = 0; i < size\_2; i++)

{

temp[0][i + 1] = temp[0][i] + 1;

}

for (int i = 0; i < size\_1; i++)

{

temp[i + 1][0] = temp[i][0] + 1;

for (int j = 0; j < size\_2; j++)

{

temp[i + 1][j + 1] = this->Min(

temp[i][j + 1] + 1,

temp[i + 1][j] + 1,

temp[i][j] + this->ReplaceCoast(str\_1[i], str\_2[j])

);

}

}

return temp[size\_1][size\_2];

}

Вспомогательные функции:

int DistationLeven::ReplaceCoast(char first, char second)

{

if (first == second)

{

return 0;

}

else

{

return 1;

}

}

int DistationLeven::Min(int value\_1, int value\_2, int value\_3)

{

if ((value\_1 <= value\_2) && (value\_1 <= value\_3))

return value\_1;

else if ((value\_2 <= value\_1) && (value\_2 <= value\_3))

return value\_2;

else

return value\_3;

}

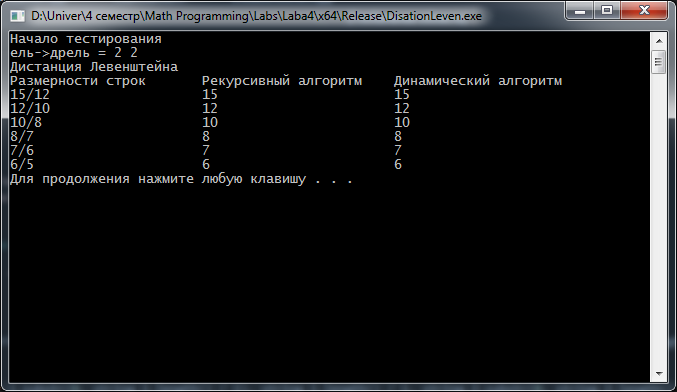


Рисунок 1 – Нахождение дистанции Левенштейна

**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).

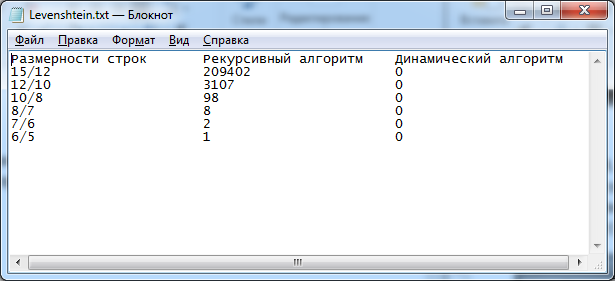


Рисунок 2 – Зависимость времени выполнения

Максимальная длина подстроки равна 15-ти символам, т.к. время вычисления дистанции Левенштейна для большей длины строк очень велико.

График зависимости времени вычисления дистанции Левенштейна от длины подстроки отображено на графике. Время вычисления дистанции с помощью динамического программирования значительно меньше, чем с помощью рекурсии.

Рисунок 3 – График зависимости

**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).

1.  
2.  
3.  
4.  
5.  
6.  
7.  
8.  
9.  
10.  
11.  
12.  
13.  
14.  
15. 
16. 
17. 
18. 
19. 
20. 
21. 
22. 
23. 
24. 
25. 
26. 
27. 

**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.

//MatricaMul.h

#pragma once

#include <iostream>

#include <vector>

class MatricaMul

{

public:

MatricaMul(int count, std::vector<int> size);

int GenerationRecursion(int start, int end);

int GenerationDinamic();

void Reset(int count, std::vector<int> size);

void ShowResultMatrica();

int GetIndex(int i, int j);

private:

int \*\*resultMatrica;

std::vector<int> resultForDinamic;

std::vector<int> size;

int count = 0;

};

Рекурсивный алгоритм:

int MatricaMul::GenerationRecursion(int start, int end)

{

int minMult = INT32\_MAX;

int resultTemp = INT32\_MAX;

if (start < end)

{

for (int k = start; k < end; k++)

{

minMult = this->GenerationRecursion(start, k) +

this->GenerationRecursion(k + 1, end) +

this->size[start - 1] \* this->size[k] \* this->size[end];

if (minMult < resultTemp)

{

this->resultMatrica[start - 1][end - 1] = k;

resultTemp = minMult;

}

}

}

else

{

return 0;

}

return resultTemp;

}

Динамическое программирование:

int MatricaMul::GenerationDinamic()

{

int minMult = INT32\_MAX;

for (int i = 0; i < this->count \* this->count; i++)

{

this->resultForDinamic.push\_back(-1);

}

for (int i = 1; i < this->count + 1; i++)

{

this->resultForDinamic[this->GetIndex(i, i)] = 0;

}

for (int l = 0; l < this->count - 1; l++)

{

for (int i = 1; i < this->count - l; i++)

{

int j = i + l + 1;

this->resultForDinamic[this->GetIndex(i, j)] = INT32\_MAX;

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

{

minMult = this->resultForDinamic[this->GetIndex(i, k)] +

this->resultForDinamic[this->GetIndex(k + 1, j)] +

this->size[i - 1] \* this->size[k] \* this->size[j];

if (minMult < this->resultForDinamic[this->GetIndex(i, j)])

{

this->resultForDinamic[this->GetIndex(i, j)] = minMult;

this->resultMatrica[i - 1][j - 1] = k;

}

}

}

}

return this->resultForDinamic[this->GetIndex(1, this->count)];

}

Вспомогательная функция:

int MatricaMul::GetIndex(int i, int j)

{

return (i - 1) \* this->count + (j - 1);

}

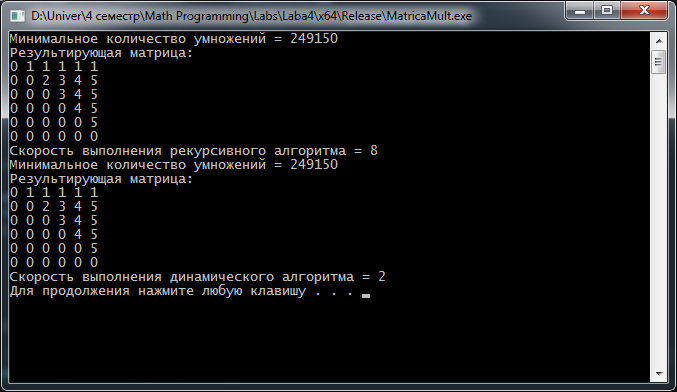


Рисунок 4 – Результат выполнения

Можно заметить, что, используя динамическое программирование, решение задачи об оптимальной расстановке скобок при умножении матриц решается в несколько раз быстрее, чем с помощью рекурсивного алгоритма.

Скобки расставляются по принципу «сначала внешние – затем внутренние».

Сначала берем элемент матрицы (1, 6), он равен 1. Это означает, что точка разрыва между первой и шестой матрицей находится после первой матрицы, что позволяет расставить скобки следующим образом: 1\*(2\*3\*4\*5\*6).

Далее берем элемент матрицы (2, 6), он равен 5, значит точка разрыва между второй и шестой матрицей находится после пятой матрицы. Получаем: 1\*((2\*3\*4\*5)\*6).

Следующий шаг – это элемент (2, 5), равный 4. Точка разрыва между второй и пятой матрицей находится после четвертой матрицы. Получаем: 1\*(((2\*3\*4)\*5)\*6).

Следующий шаг – это элемент (2, 4), равный 3, разрыв между второй и четвертой матрицей находится после третьей матрицы. Окончательный ответ:   
1\*((((2\*3)\*4)\*5)\*6).

**Вывод**

В ходе выполнения данной лабораторной работы была решена задача о дистанции Левенштейна с помощью рекурсивного алгоритма, а также используя динамическое программирование. Задача об оптимальной расстановке скобок при умножении нескольких матриц была решена, используя рекурсивный алгоритм и динамическое программирование. Скорость решения этих задач с помощью динамического программирования оказалась в разы быстрее, чем решение с помощью рекурсивного алгоритма.