Белорусский государственный технологический университет

Факультет информационных технологий

Кафедра информационных систем и технологий

Отчёт по лабораторной работе №4

«Динамическое программирование»

по дисциплине

«Математическое программирование»

Выполнил:

студент 2-го курса спец. ДЭиВИ

Глушкова М.Е.

Вариант №2

Проверил:

ассистент кафедры

Барковский Е.В.

**Лабораторная работа 4. Динамическое программирование**

**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 S1 длиной 300 символов и S2 длиной250.

**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]; // 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)];

}

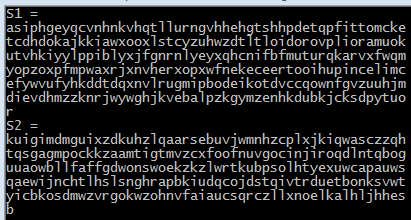


Рис. 1 – Пример генерации строк

**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** \*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);

}

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.

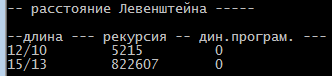


Рис. 2 – Проверка работоспособности решений

**Задание 3.**

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

**Решение:**

Исходные данные для построения графика изображены на рисунке 3:

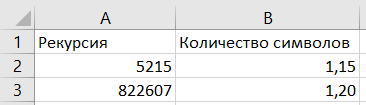


Рис. 3 – Исходные данные

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

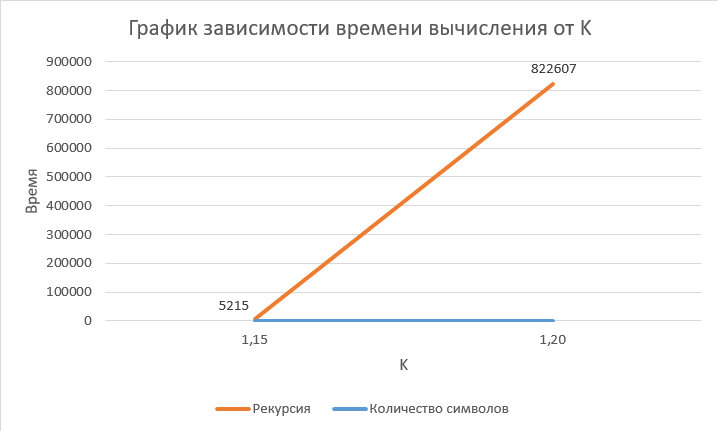


Рис. 4 – График зависимости времени вычисления от К

Исходя из задания номер 2 и графика видно, что при больших значениях К, и, 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).

|  |  |  |  |
| --- | --- | --- | --- |
| Вариант | Задание 4 | | Задание 5 |
| 2 | Том | Исток | 7\*10, 10\*18, 18\*21, 21\*28, 28\*38, 38\*49 |

**Решение:**

Ываыва

**Задание 5.**

**Четные варианты**. 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.

**Решение:**

*Matrix.cpp*

#include "stdafx.h"

#include <cmath>

#include <memory.h>

#include <ctime>

#include <iostream>

#include "MultyMatrix.h" // умножение матриц

#define N 6

int main()

{

clock\_t t1 = 0;

clock\_t t2 = 0;

clock\_t t3 = 0;

clock\_t t4 = 0;

int Mc[N + 1] = { 7,10,18,21,28,38,49 }, Ms[N][N], r = 0, rd = 0;

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

t1 = clock();

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

t2 = clock();

setlocale(LC\_ALL, "rus");

std::cout << std::endl;

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

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

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

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

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

std::cout << std::endl << std::endl << "матрица S" << std::endl;

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

{

std::cout << std::endl;

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

}

std::cout << std::endl;

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

t3 = clock();

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

t4 = clock();

std::cout << std::endl

<< "-- расстановка скобок (динамичеое программирование) " << std::endl;

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

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

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

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

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

<< rd;

std::cout << std::endl << std::endl << "матрица S" << std::endl;

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

{

std::cout << std::endl;

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

}

std::cout << std::endl << std::endl;

system("pause");

return 0;

}

*MultyMatrix.cpp*

#include "stdafx.h"

#include <memory.h>

#include "MultyMatrix.h"

#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

};

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

};

*MultyMatrix.h*

#pragma once

// расстановка скобок при умножении матриц

// функции возвращают минимальное количество операций умножения

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

int OptimalM( // рекурсия

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

);

int OptimalMD( // динамическое программирование

int n, // [in] количество матриц

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

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

);

*stdafx.cpp*

// stdafx.cpp: исходный файл, содержащий только стандартные включаемые модули

// Matrix.pch будет использоваться в качестве предкомпилированного заголовка

// stdafx.obj будет содержать предварительно откомпилированные сведения о типе

#include "stdafx.h"

// TODO: Установите ссылки на любые требующиеся дополнительные заголовки в файле STDAFX.H

// , а не в данном файле

*stdafx.h*

// stdafx.h: включаемый файл для стандартных системных включаемых файлов

// или включаемых файлов для конкретного проекта, которые часто используются, но

// не часто изменяются

//

#pragma once

#include "targetver.h"

#include <stdio.h>

#include <tchar.h>

// TODO: Установите здесь ссылки на дополнительные заголовки, требующиеся для программы

*targetver.h*

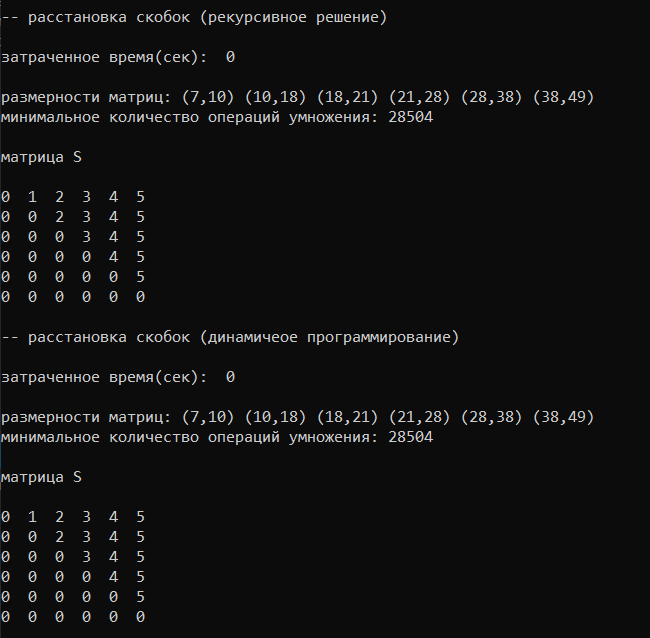
#pragma once

// Включение SDKDDKVer.h обеспечивает определение самой последней доступной платформы Windows.

// Если требуется выполнить сборку приложения для предыдущей версии Windows, включите WinSDKVer.h и

// задайте для макроса \_WIN32\_WINNT значение поддерживаемой платформы перед включением SDKDDKVer.h.

#include <SDKDDKVer.h>



**Принцип расстановки скобок по итоговой матрице:**

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

А1=7\*10,

А2=10\*18,

А3=18\*21,

А 4 =21\*28,

А 5 =28\*38,

А 6 =38\*49.

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 | 4 | 5 |
| **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 первой и пятой 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 получаем:

(((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 28504.

**Вывод:** в ходе работы были освоены общие принципы решения задач методом динамического программирования, проведено сравнение полученных решений задач с рекурсивным методом.