Белорусский государственный технологический университет

Кафедра Информационных Систем и Технологий

**Лабораторная работа №4**

**Динамическое программирование**

**Выполнил:**

 студент 2 курса 2 группы

Талатынник Артём Дмитриевич

Минск 2021

**Условие:**

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

Часть кода главной функции:

#include "stdafx.h"

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

{

setlocale(LC\_ALL,"rus");

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

Auxil \*Rand = new Auxil();

unsigned int lenS1=300, lenS2=250;

float k[7] = { 0.04F, 0.05F, 0.067F, 0.1F, 0.2F, 0.5F, 1.0F };

char \*S1 = new char[lenS1],

\*S2 = new char[lenS2];

for(unsigned int i=0; i<(lenS1>lenS2?lenS1:lenS2); i++)

{ if(i<lenS1)

S1[i]=(int)Rand->get\_int('a','z');

if(i<lenS2)

S2[i]=(int)Rand->get\_int('a','z');

}

std::cout<<S1<<std::endl<<S2<<std::endl;

std::cout<<std::endl<< "-- расстояние Левенштейна -----"<< std::endl;

std::cout<<std::endl<< "--длина --- рекурсия -- дин.програм. ---"<<std::endl;

for (int i = 0; i < sizeof(k)/sizeof(float); i++)

{ t1 = clock();

levenshtein\_r((int)(k[i]\*lenS1),S1,(int)(k[i]\*lenS2 + 1),S2);

t2 = clock();

t3 = clock();

levenshtein((int)(k[i]\*lenS1),S1,(int)(k[i]\*lenS2 + 1),S2);

t4 = clock();

std::cout<<std::right<<std::setw(2)<<(int)(k[i]\*lenS2)<<"/"<<std::setw(2)<<(int)(k[i]\*lenS1 + 1)

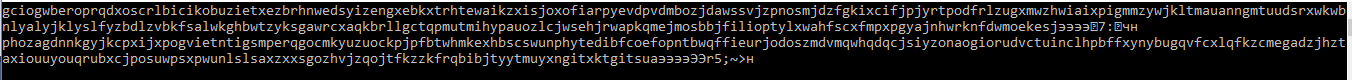
<< " "<<std::left<<std::setw(10)<<(t2-t1)

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

}

return 0;

}

Выполнение программы:

**Задание 2.** 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).

Код программы:

// - Levenshtein.h

// -- дистанции Левенштeйна (динамическое программирование)

int levenshtein(

int lx, // длина слова x

const char x[], // слово длиной lx

int ly, // длина слова y

const char y[] // слово y

);

// -- дистанции Левенштeйна (рекурсия)

int levenshtein\_r(

int lx, // длина строки x

const char x[], // строка длиной lx

int ly, // длина строки y

const char y[] // строка y

);

// - Levenshtein.cpp

#include "stdafx.h"

#include <iomanip>

#include <algorithm>

#include "Levenshtein.h"

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

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);

}

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;

};

// --- main

// вычисление дистанции (расстояния) Левенштейна

#include "stdafx.h"

#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

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

{

setlocale(LC\_ALL, "rus");

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

char x[] = "abcdefghklmnoxm", y[] = "xyabcdefghomnkm";

int lx = sizeof(x)-1, ly = sizeof(y)-1;

std::cout<<std::endl;

std::cout<<std::endl<< "-- расстояние Левенштейна -----"<< std::endl;

//std::cout << std::endl << x << "->" << y << " = " << levenshtein\_r(sizeof(x)-1, x, sizeof(y)-1, y) << std::endl;

std::cout<<std::endl<< "--длина --- рекурсия -- дин.програм. ---"<<std::endl;

for (int i = 8; i < std::min(lx,ly); i++)

{

t1 = clock();

levenshtein\_r(i,x,i-2, y);

t2 = clock();

t3 = clock();

levenshtein(i,x,i-2, y);

t4 = clock();

std::cout<<std::right<<std::setw(2)<<i-2<<"/"<<std::setw(2)<<i

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

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

}

system("pause");

return 0;

}

Выполнение программы для сравнения времени выполнения:

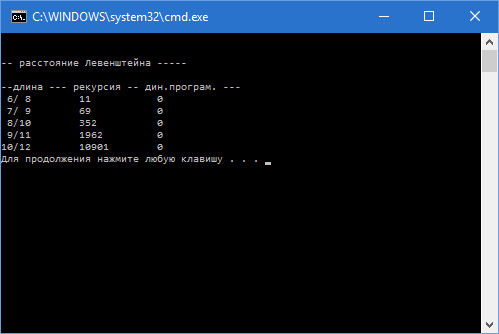


График зависимости:

**Задание 3.** Реализовать вручную пример вычисления дистанции Левенштейна при помощи рекурсивного алгоритма (в соответствии с вариантом)

1. **Вычисление дистанции Левенштейна**



 – количество символов в заданной строке. Например, 

 – заданная строка без последнего символа. Например, 

 – последний символ заданной строки. Например, 

Поясним принцип применения этого рекуррентного соотношения на следующем примере.

1. Пусть необходимо вычислить  Тогда имеем следующую последовательность шагов:
2.  
3.  
4.  
5.  
6.  

1.  

1.  
2.  
3.  
4.  
5.  
6.  
7.  

1.  

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

Таким образом дистанция Левенштейна для данных строк равна 3.

**Задание 4.** 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.

|  |  |  |  |
| --- | --- | --- | --- |
| 6 | Сом | Домик | 9\*12, 12\*20, 20\*23, 23\*30, 30\*40, 40\*51 |

Код программы:

// --- MultyMatrix.h

// расстановка скобок

// 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

);

// --- MultiMatrix.cpp

// расстановка скобок (рекурсия)

#include"stdafx.h"

#include<memory.h>

#include"MultiMatrix.h"

#define INFINITY 0x7fffffff

#define NINFINITY 0x80000000

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

{

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

int o =INFINITY, 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

};

// --- MultyMatrix.cpp (продолжение)

// 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

};

// --- main

// расстановка скобок

#include"stdafx.h"

#include<cmath>

#include<memory.h>

#include<iostream>

#include"MultiMatrix.h"// умножение матриц

#define N 6

int main()

{

int Mc[N+1] = {9, 12, 20, 23, 30, 40, 51}, Ms[N][N], r = 0, rd = 0;

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

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

setlocale(LC\_ALL, "rus");

std::cout<<std::endl;

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

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

std::cout<<std::endl

<<"-- расстановка скобок (динамичеое программирование) "<< 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]<<" " ;

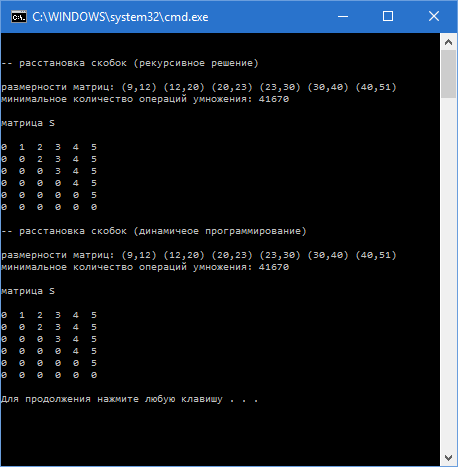
}

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

system("pause");

return 0;

}

Результат выполнения:  


Принцип расстановки скобок по итоговой матрице:

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

А1=9,

А2=12,

А3=20,

А 4 =23,

А 5 =30,

А 6 =40.

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 41670.