МИНИСТЕРСТВО ОБРАЗОВАНИЯ РЕСПУБЛИКИ БЕЛАРУСЬ

Учреждение образования «БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ

ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Факультет Программное обеспечение информационных технологий

Дисциплина Математическое программирование

**Отчет по лабораторным работам по дисциплине**

**“Математическое программирование”**

Выполнил: студент 2курса 5 группы специальности “ПОИТ” Хатченок Д.Н.

(Ф.И.О)

Минск 2024

**Лабораторная работа 4.**

Динамическое программирование. Решение задач методом динамического программирования.

**Задание 1**

На языке С++ сгенерировать случайным образом строку букв латинского алфавита S1 длиной 300 символов и S2 длиной 200.

Листинг кода генерации строк:

|  |
| --- |
| #include <iostream>  #include <ctime>  #include <Windows.h>  using namespace std;  #define FIRST\_LEN 300  #define SECOND\_LEN 250  char\* GenerateRandomString(int size)  {  char\* str = (char\*)malloc(sizeof(char) \* size);  for (int i = 0; i < size; i++) {  str[i] = rand() % 26 + 'a'; // 26 букв в алфавите  }  return str;  }  int main()  {  SetConsoleCP(1251);  SetConsoleOutputCP(1251);  srand(time(NULL));  char\* s1 = GenerateRandomString(FIRST\_LEN);  cout << "S1: " << endl;  for (int i = 0; i < FIRST\_LEN; i++) {  if (i % 50 == 0)  {  cout << "\n";  }  cout << s1[i];  }  cout << endl << endl;  char\* s2 = GenerateRandomString(SECOND\_LEN);  cout << "S2: " << endl;  for (int i = 0; i < SECOND\_LEN; i++) {  if (i % 50 == 0)  {  cout << "\n";  }  cout << s2[i];  }  cout << endl << endl;  } |

Листинг 1 – генерация строк

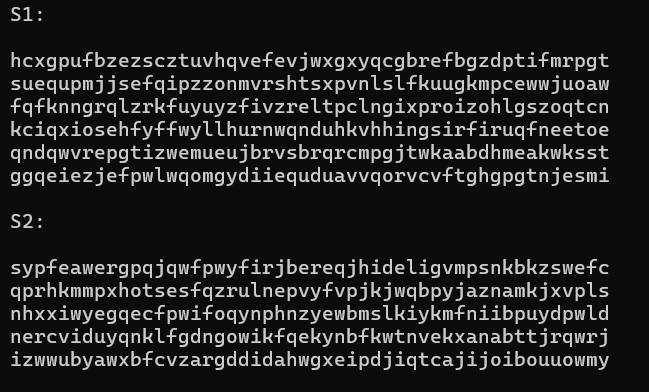
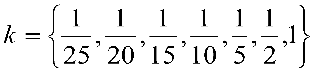


Рисунок 1 – результат работы генератора

**Задание 2**

Необходимо при помощи двух способов вычислить дистанцию Левенштейна для . Код выполнение программы представлен на рисунках ниже.

|  |
| --- |
| #pragma once  #include <tchar.h>  // - 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  ); |

Листинг 2 – Заголовочный файл с прототипами функций

// - Levenshtein.cpp

#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;

};

Листинг 3 – Реализация функций

// --- main

// вычисление дистанции (расстояния) Левенштейна

#include <algorithm>

#include <iostream>

#include <ctime>

#include <iomanip>

#include "Levenshtein.h"

#include <Windows.h>

using namespace std;

char\* GenerateRandomString(int size)

{

char\* str = (char\*)malloc(sizeof(char) \* size);

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

str[i] = rand() % 26 + 'a'; // 26 букв в алфавите

}

return str;

}

int main()

{

SetConsoleCP(1251);

SetConsoleOutputCP(1251);

char\* s1 = GenerateRandomString(300);

cout << "S1: " << endl;

for (int i = 0; i < 300; i++) {

if (i % 50 == 0)

{

cout << "\n";

}

cout << s1[i];

}

cout << endl << endl;

srand(time(NULL) + 1);

char\* s2 = GenerateRandomString(200);

cout << "S2: " << endl;

for (int i = 0; i < 200; i++) {

if (i % 50 == 0)

{

cout << "\n";

}

cout << s2[i];

}

cout << endl << endl;

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

int lx = sizeof(s1);

int ly = sizeof(s2);

int s1\_size[]{ 300 / 25, 300 / 20, 300 / 15, 300 / 10, 300 / 5, 300 / 2, 300 };

int s2\_size[]{ 200 / 25, 200 / 20, 200 / 15, 200 / 10, 200 / 5, 200 / 2, 200 };

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

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

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

{

t1 = clock();

levenshtein\_r(s1\_size[i], s1, s2\_size[i], s2);

t2 = clock();

t3 = clock();

levenshtein(s1\_size[i], s1, s2\_size[i], s2);

t4 = clock();

cout << right << setw(2) << s1\_size[i] << "/" << setw(2) << s2\_size[i]

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

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

}

system("pause");

return 0;

}

Листинг 4 – функция main

Результат выполнения программы представлен на рисунке 4.

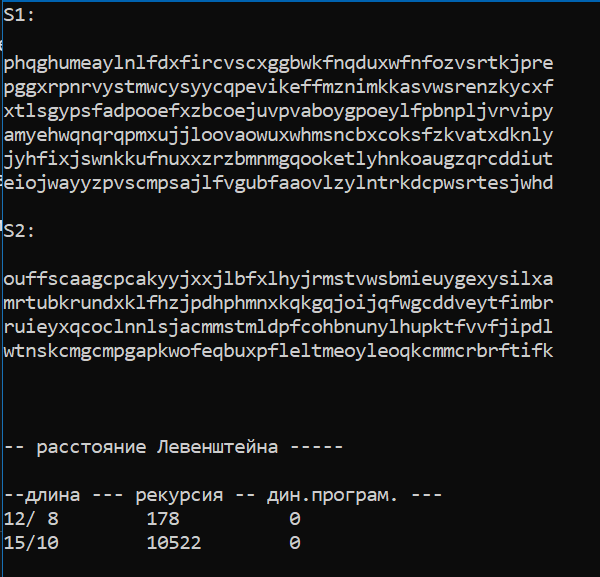


Рисунок 2 – Результат выполнения программы

**Задание 3**

Необходимо провести сравнительный анализ по затраченному на выполнение программы времени. На графике, который изображён на рисунке 5, нетрудно заметить, что использование динамического алгоритма в несколько раз эффективнее по затраченному времени, нежели рекурсивное выполнение.

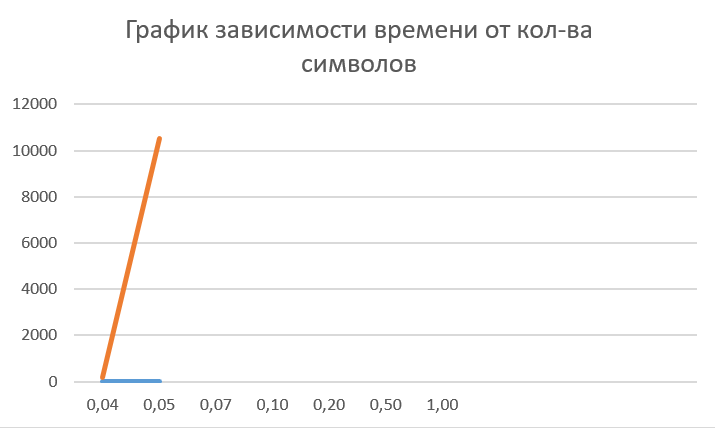


Рисунок 3 – График зависимости выполнения

**Задание 4**

Реализовать в соответствии с вариантом (рисунок 6) пример вычисления дистанции Левенштейна вручную, при помощи рекурсивного алгоритма. Ниже представлен алгоритм решения последовательно:



Рисунок 4 – Вариант для вычисления дистанции Левенштейна



= 5.

= 4.



= 4.

= 3.



= 3.

= 2.



= 2.

= 1.



= 3.

= 2.



= 2.

= 1.



= 1.

= 1.

= 0.

1. L(“Э”, “Х”) = min(2,2,1) = 1
2. L(“Эх”, “Х”) = min(2,3,2) = 2
3. L(“Эхо”, “Х”) = min(3,4,3) = 3
4. L(“Э”, “Хо”) = min(3,2,2) = 2
5. L(“Эх”, “Хо”) = min(3,3,2) = 2
6. L(“Эхо”, “Хо”) = min(3,4,2) = 2
7. L(“Э”, “Хор”) = min(4,3,3) = 3
8. L(“Эх”, “Хор”) = min(4,3,3) = 3
9. L(“Эхо”, “Хор”) = min(4,3,3) = 3
10. L(“Э”, “Хоре”) = min(5,4,4) = 4
11. L(“Э”, “Хорек”) = min(6,5,5) = 5
12. L(“Эх”, “Хоре”) = min(5,4,4) = 4
13. L(“Эхо”, “Хоре”) = min(5,5,4) = 4
14. L(“Эх”, “Хорек”) = min(6,5,5) = 5
15. L(“Эхо”, “Хорек”) = min(6,5,5) = 5

**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.

Листинги кода представлены ниже:

// --- MultyMatrix.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

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

);

Листинг 5 – MultiMatrix.h

// --- MultiMatrix.cpp

// расстановка скобок (рекурсия)

#include <memory.h>

#include "MultiMatrix.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, 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

};

Листинг 6 - MultiMatrix.cpp

// --- main

// расстановка скобок

#include <cmath>

#include <memory.h>

#include <chrono>

#include <iostream>

#include "MultiMatrix.h" // умножение матриц

#define N 6

int main()

{

int Mc[N + 1] = { 8,11,19,22,29,39,50 }, Ms[N][N], r = 0, rd = 0;

auto t1 = std::chrono::system\_clock::now();

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

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

auto t2 = std::chrono::system\_clock::now();

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;

std::chrono::duration<double> elapsed = t2 - t1;

std::cout << "Время выполнения: " << elapsed.count() \* 1000000 << "мкс." << std::endl;

t1 = std::chrono::system\_clock::now();

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

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

t2 = std::chrono::system\_clock::now();

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;

elapsed = t2 - t1;

std::cout << "Время выполнения: " << elapsed.count() \* 1000000 << "мкс." << std::endl;

system("pause");

return 0;

}

Листинг 7 – main.cpp

Выполнив сравнительный анализ времени затраченного на решение задачи об оптимальной расстановке скобок при умножении нескольких матриц для двух методов решения, можно заметить, что динамический алгоритм работает быстрее, однако, в данном ситуации оба метода затрачивают для выполнения минимально времени. Результаты представлены на рисунке 7:

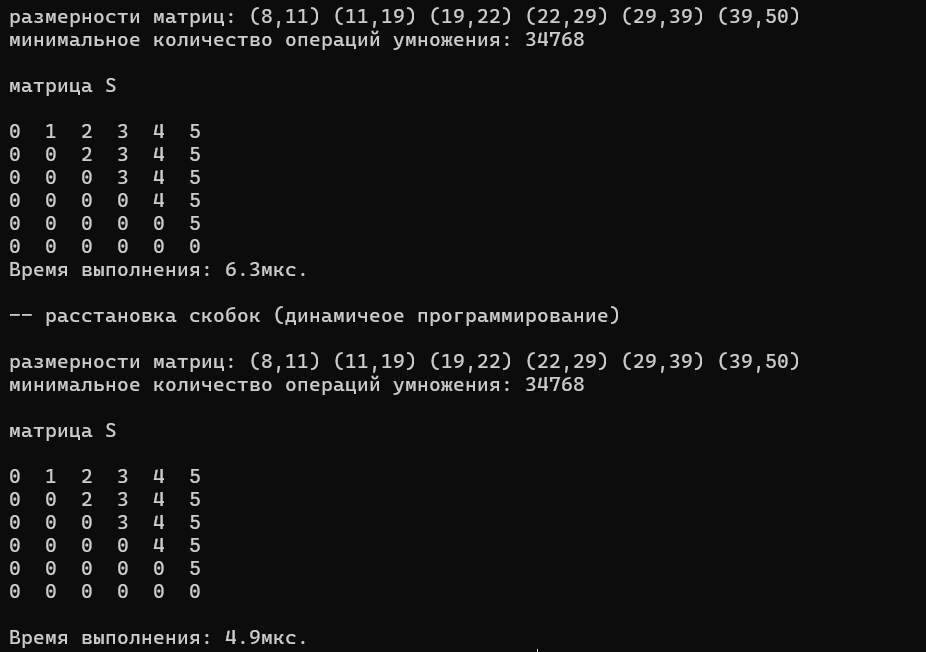


Рисунок 5 – выполнение программы по оптимальной расстановке скобок при умножении нескольких матриц

По данной матрице мы можем оптимально расставить скобки. Для умножения шести входных матриц достаточно выполнить m[1,6]. Оптимальная последовательность умножений имеет вид:

A16 = (s[1,6] = 5) = A15 \* A6 =

(s[1,5] = 4) = (A13 \* A45)\*A66 =

(s[1,3] = 2, s[4,5] = 4) = (A12 \* A33 \* A44 \* A55)\*A66 =

((A1 \* A2 )\* A3 \* A4 \* A5) \* A6

Данная расстановка скобок никак не влияет на последовательность перемножения матриц, значит матрицы уже были расставлены оптимально для наименьшего количества операций умножения.

**Вывод:** динамический подход к решению задач позволяет выполнять их значительно быстрее, чем рекурсивный, особенно это будет заметно при решении задач с большим объёмом информации. Так же, я сделал следующие выводы:

1. Динамическое программирование - это мощный метод решения задач, позволяющий эффективно решать широкий спектр задач, которые не могут быть решены простыми алгоритмами.

2. Метод динамического программирования заключается в разбиении сложной задачи на более простые подзадачи, решение которых затем комбинируется в общее решение задачи.

3. Решение задач методом динамического программирования отличается высокой скоростью выполнения благодаря использованию кэширования вычислений и быстрой обработке данных.

4. В ходе выполнения лабораторной работы был исследован один из наиболее популярных методов динамического программирования - расстояние Левенштейна. Результаты экспериментов показали, что данная техника может быть очень эффективна для решения задач, связанных с обработкой текстовых данных.

5. Решение задач методом динамического программирования может быть осуществлено не только с помощью программирования на языке C++, но и на других языках, таких как Python или Java.