**Лабораторная работа №4**

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

**Практическая часть**

**Задание 1.**

На языке С++ сгенерировать случайным образом строку букв латинского алфавита *S1* длиной 300 символов и *S2*длиной 250. Листинг кода представлен на рисунке 1.1. Результат выполнения программы представлен на рисунке 1.2.

|  |
| --- |
| #define FIRST\_LEN 300  #define SECOND\_LEN 200  char\* first\_Task(int size)  {  char\* str = new char[size];  for (int i = 0; i < size; i++)  str[i] = rand() % 26 + 'a';  return str;  }  int main()  {  srand(time(NULL));  setlocale(LC\_ALL, "rus");  cout << "First string: \n";  char\* s1 = first\_Task(FIRST\_LEN);  for (int i = 0; i < FIRST\_LEN; i++)  cout << s1[i];  cout << endl;  cout << "\nSecond string: \n";  char\* s2 = first\_Task(SECOND\_LEN);  for (int i = 0; i < SECOND\_LEN; i++)  cout << s2[i];  cout << endl << endl;  return 0;  } |

Рисунок 1.1 – Генерация строк

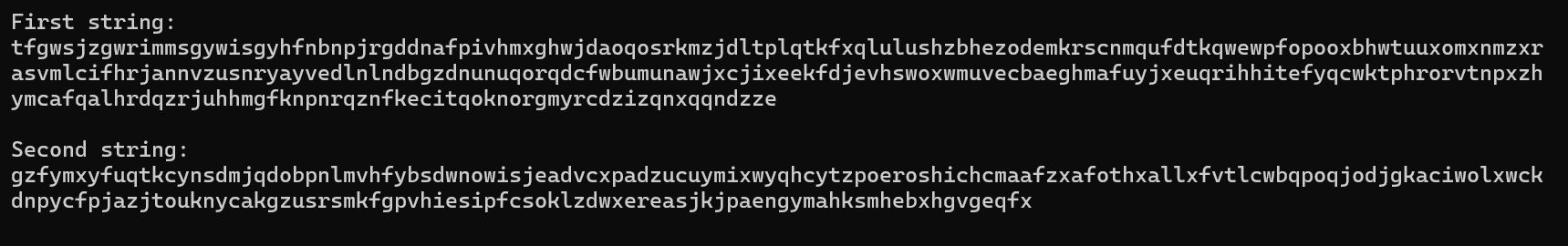


Рисунок 1.2 – Результат работы программы

**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). Листинг кода представлен на рисунке 1.3. Результат выполнения программы представлен на рисунке 1.4.

|  |
| --- |
| //Levenshtain.h  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  );  //Levenshtain.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;  };  //main  void second\_Task(char\* s1, char\* s2)  {  clock\_t t1 = 0, t2 = 0, t3 = 0, t4 = 0;  int lx = sizeof(s1) - 1, ly = sizeof(s2) - 1;  cout << endl;  cout << endl << "-- расстояние Левенштейна -----" << endl;  cout << endl << "--длина --- рекурсия -- дин.програм. ---" << endl;  int s1\_Dist[]{ FIRST\_LEN / 25, FIRST\_LEN / 20, FIRST\_LEN / 15, FIRST\_LEN / 10, FIRST\_LEN / 5, FIRST\_LEN / 2, FIRST\_LEN };  int s2\_Dist[]{ SECOND\_LEN / 25, SECOND\_LEN / 20, SECOND\_LEN / 15, SECOND\_LEN / 10, SECOND\_LEN / 5, SECOND\_LEN / 2, SECOND\_LEN };  for (int i = 0; i < min(lx, ly); i++)  {  t1 = clock();  levenshtein\_r(s1\_Dist[i], s1, s2\_Dist[i], s2);  t2 = clock();  t3 = clock();  levenshtein(s1\_Dist[i], s1, s2\_Dist[i], s2);  t4 = clock();  cout << right << setw(2) << s2\_Dist[i] << "/" << setw(2) << s1\_Dist[i]  << "\t\t" << left << setw(10) << (t2 - t1)  << "\t" << setw(10) << (t4 - t3) << endl;  }  system("pause");  return;  }  int main()  {  srand(time(NULL));  setlocale(LC\_ALL, "rus");  cout << "First string: \n";  char\* s1 = first\_Task(FIRST\_LEN);  for (int i = 0; i < FIRST\_LEN; i++)  cout << s1[i];  cout << endl;  cout << "\nSecond string: \n";  char\* s2 = first\_Task(SECOND\_LEN);  for (int i = 0; i < SECOND\_LEN; i++)  cout << s2[i];  cout << endl << endl;  second\_Task(s1, s2);  //fifth\_Task();  system("pause");  return 0;  } |

Рисунок 1.3 – Вычисление расстояния Левенштейна

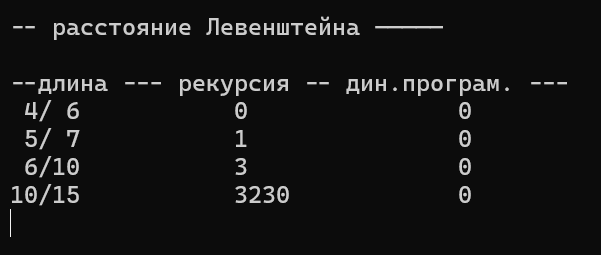


Рисунок 1.4 – Результат работы программы

**Задание 3.**

Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от . На рисунке 1.5 представлены графики зависимости времени вычисления от *k*.

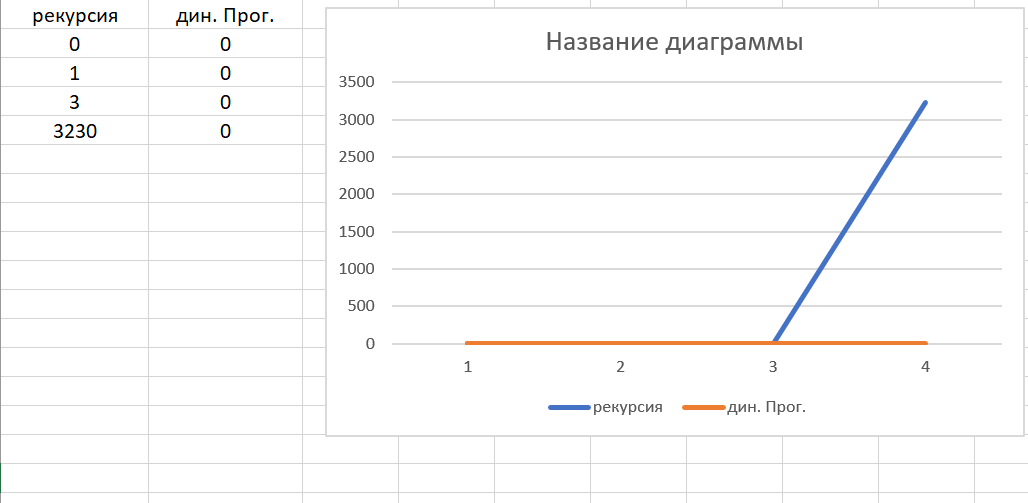
****

Рисунок 1.5 – Графики зависимости

**Задание 4.**

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

Вычисление дистанции Левинштейна

Пусть Х и Y– две символьные строки, тогда для вычисления дистанции Левенштейна L(Х,Y) между ними может быть использовано следующее рекуррентное соотношение:



* + - *len(X)* – количество символов в заданной строке. Например, *len(“привет”) =*6
    - *cut*(*X*) – заданная строка без последнего символа. Например, *cut*(*“привет”*) = *“приве”*
    - *last(X)* – последний символ который находится в заданной строке. Например, *last(“привет”)=“т”*

|  |  |
| --- | --- |
| Задание 4 | |
| Сом | Домик |

Дистанция Левенштейна для «сом» и «домик» равно 3.

**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).

|  |
| --- |
| // MUltiMatrix.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  );  #include <memory.h>  #include "MultiMatrix.h"  // --- MultiMatrix.cpp  // 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, 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  void fifth\_Task()  {  setlocale(LC\_ALL, "rus");  clock\_t t1 = 0, t2 = 0, t3 = 0, t4 = 0;  /\*Mc[N + 1] = { 5,10,15,20,25,30,35 }\*/  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);  t1 = clock();  r = OptimalM(1, N, N, Mc, OPTIMALM\_PARM(Ms));  t2 = clock();  cout << endl;  cout << endl << "-- расстановка скобок (рекурсивное решение) " << 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 << "\n\nИтоговое время выполнения Рекурсивного решения: " << (t2 - t1) << endl;  cout << endl;  memset(Ms, 0, sizeof(int) \* N \* N);  t3 = clock();  rd = OptimalMD(N, Mc, OPTIMALM\_PARM(Ms));  t4 = clock();  cout << endl << "\n-- расстановка скобок (динамичеое программирование) " << 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 << "\n\nИтоговое время выполнения Динамического программирования: " << (t3 - t4) << endl;  cout << endl << endl;  system("pause");  return;  } |

Рисунок 1.6 – Решение задачи о расстановке скобок

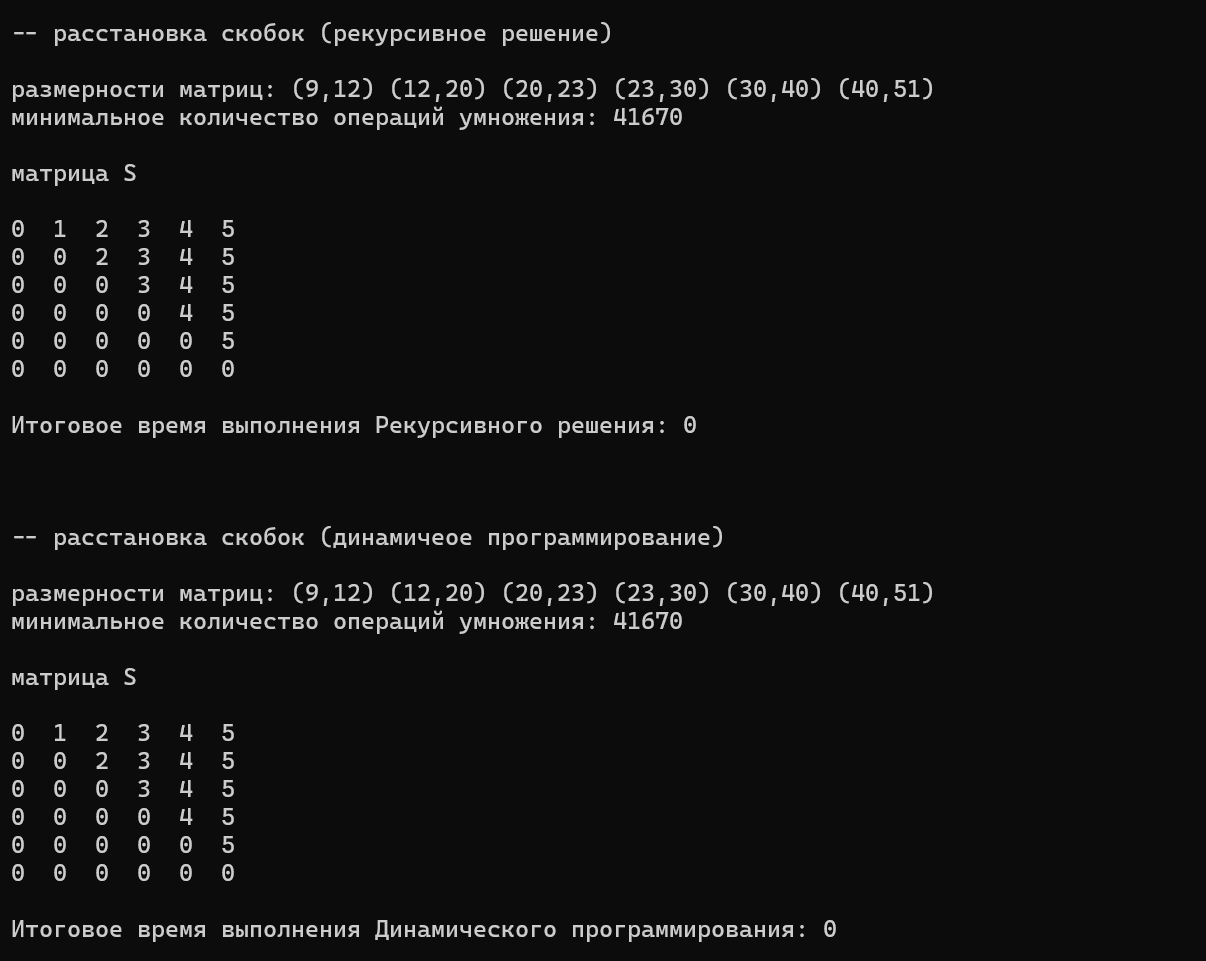


Рисунок 1.7 – Результат работы программы

Принцип расстановки скобок по итоговой матрице:

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

Имеется 6 матриц, вот их размерность:

А1 = 9\*12,

А2 = 12\*20,

А3 = 20\*23,

А4 = 23\*30,

А5 = 30\*40,

А6 = 40\*51.

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 внешние 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 4-ой матрицы.

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

Далее берем элемент (1,3) и получаем, что он равен 3 - Следовательно разрыв будет после 3-ой матрицы:

(((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.

**Вывод:** в результате выполнения лабораторной работы были освоены общие принципы решения задач методом динамического программирования. Были изучены его основные этапы и принципы работы алгоритмов. Были рассмотрены примеры решения задач методом динамического программирования и сравнены с рекурсивным методом.