**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 200 символов и *S2*длиной 300. Листинг кода представлен в листинге 1.1. Результат выполнения программы представлен на рисунке 1.1.

|  |
| --- |
| clock\_t t1 = 0, t2 = 0, t3, t4;  //-----------Task1  srand(time(NULL));  char\* str1 = new char[300];  char\* str2 = new char[250];  int i = 0;  for (i; i < 250; i++)  {  str2[i] = (char)(rand() % 25 + 65);  str1[i] = (char)(rand() % 25 + 65);  }  str2[i] = 0x00;  for (i; i < 300; i++) str1[i] = (char)(rand() % 25 + 65);  str1[i] = 0x00;  //-------------Task2  int k[] = { 100, 50, 25, 20, 10, 5, 4, 2, 1 };  int res[sizeof(k) / sizeof(int)];  std::cout << "Исходные строки: " << std::endl;  std::cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~первая строка~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl;  std::cout << str1 << std::endl;  std::cout << "~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~вторая строка~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~~" << std::endl;  std::cout << str2 << std::endl;  for (int i = 0; i < sizeof(k) / sizeof(int); i++)  {  t1 = clock();  res[i] = levenshtein((int)(sz2 / k[i]), str2, (int)(sz1 / k[i]), str1);  t2 = clock();  std::cout << "Для размеров строк : " << sz2 / k[i] << " и " << sz1 / k[i] << "-----" << res[i] << std::endl;  std::cout << "Время :" << (double)(t2 - t1) / (double)(CLOCKS\_PER\_SEC) << std::endl;  } |

Листинг 1.1 – Генерация строк

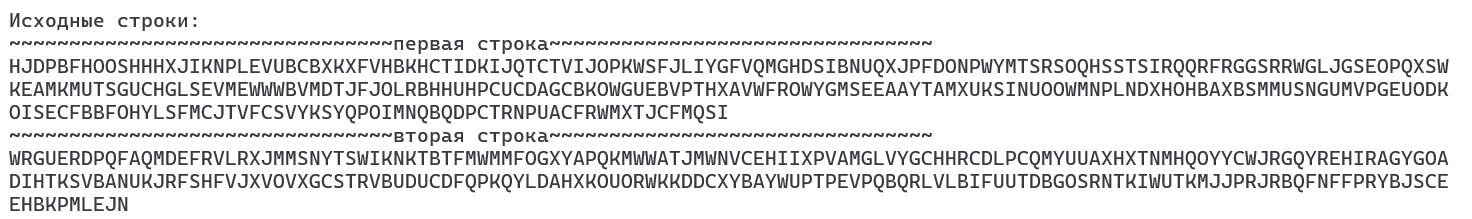


Рисунок 1.1 – Результат работы программы

**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 в листинге 1.2. Результат выполнения программы представлен на рисунке 1.2

|  |
| --- |
| int min3(int x1, int x2, int x3) // выбрать минимум из 3х  {  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;  };  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);  } |

Листинг 1.2 – Вычисление расстояния Левенштейна

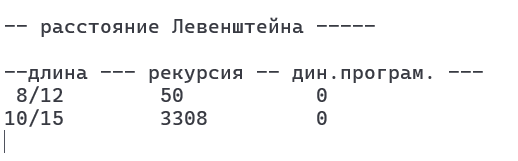


Рисунок 1.2 – Результат работы программы

**Задание 3.**

Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от . На рисунке 1.3 представлены графики зависимости времени вычисления от *k*.

На графике, который изображён на рисунке 1.3, нетрудно заметить, что использование динамического алгоритма во много раз эффективнее по затраченному времени, нежели рекурсивное выполнение.

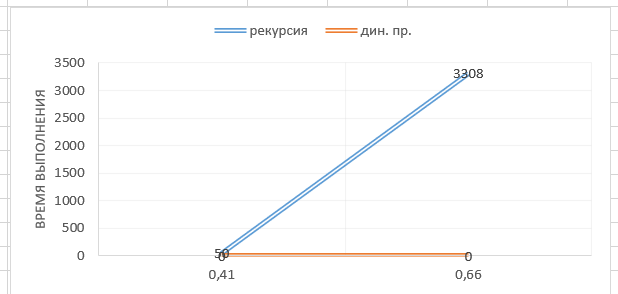


Рисунок 1.3 – График зависимости выполнения

**Задание 4.**

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

|  |  |
| --- | --- |
| Задание 4 | |
| Эхо | Хорек |



= 5.

= 4.

Нетрудно убедиться, что для превращения слова «» в слово «Хорек» достаточно вставить 5 букв.

Нетрудно убедиться, что для превращения слова «» в слово «Хоре» достаточно вставить 4 буквы.



= 4.

= 3.

Нетрудно убедиться, что для превращения слова «» в слово «Хоре» достаточно вставить 4 буквы.

Нетрудно убедиться, что для превращения слова «» в слово «Хор» достаточно вставить 3 буквы.



= 3.

= 2.

Нетрудно убедиться, что для превращения слова «» в слово «Хор» достаточно вставить 3 буквы.

Нетрудно убедиться, что для превращения слова «» в слово «Хо» достаточно вставить 2 буквы.



L(“Э”, “Х”) = 0

Нетрудно убедиться, что для превращения слова «Э» в слово «Э» достаточно вставить 0 букв.



= 2.

= 1.

L(“Э”, “Х”) = 0

Нетрудно убедиться, что для превращения слова «Э» в слово «Э» достаточно вставить 0 букв.

Нетрудно убедиться, что для превращения слова «» в слово «Хо» достаточно вставить 2 буквы.

Нетрудно убедиться, что для превращения слова «» в слово «Х» достаточно вставить 1 букву.



= 3.

= 2.

Нетрудно убедиться, что для превращения слова «Эхо» в слово «» достаточно вставить 3 буквы.

Нетрудно убедиться, что для превращения слова «Эх» в слово «» достаточно вставить 2 буквы.



= 2.

= 1.

L(“Э”, “X”) = 0

Нетрудно убедиться, что для превращения слова «Э» в слово «Э» достаточно вставить 0 букв.

Нетрудно убедиться, что для превращения слова «Эх» в слово «» достаточно вставить 2 буквы.

Нетрудно убедиться, что для превращения слова «Э» в слово «» достаточно вставить 1 букву.

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

Таким образом самая короткая дистанция Левенштейна составляет 4 шага.

**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).

|  |
| --- |
| #pragma once  #include "pch.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, 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 1.4 – Решение задачи о расстановке скобок

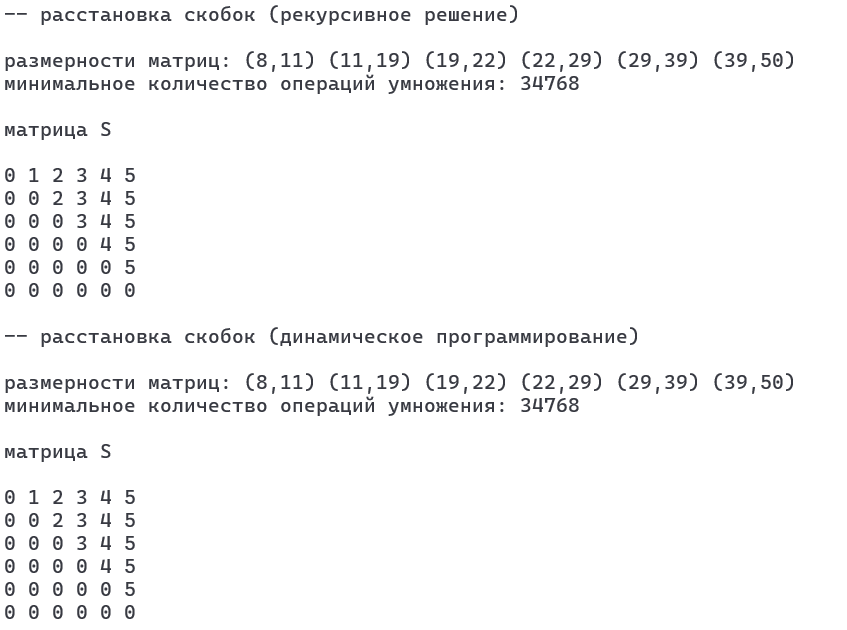


Рисунок 1.5 – Результат работы программы

Принцип расстановки скобок по итоговой матрице:

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

А1=8\*11,

А2=11\*19,

А3=19\*22,

А4 =22\*29,

А5 =29\*39,

А6 =39\*50.

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 5. Следовательно разрыв будет после 4-ой 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 34768.

**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.