**Лабораторная работа 4.**

Динамическое программирование. Решение задач методом динамического программирования.

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

**Задание 1**

На языке С++ сгенерировать случайным образом строку букв латинского алфавита *S1* длиной 300 символов и *S2*длиной 250.

|  |
| --- |
| char\* s1 = Task1(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;  srand(time(NULL) + 1);  char\* s2 = Task1(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 – Листинг кода генерации строк

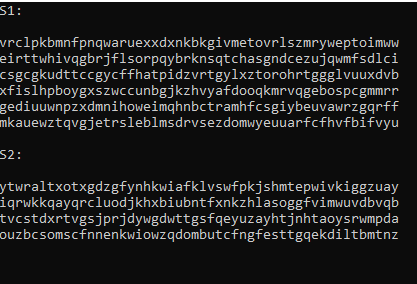


Рисунок 1 – Результат выполнения

**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).

|  |
| --- |
| #include "stdafx.h"  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\*\* matr;  int w, left, top, left\_top;  matr = new int\* [lx];  for (int i = 0; i < lx; i++)  matr[i] = new int[ly];  matr[0][0] = 0;  for (int i = 1; i < lx; i++)  matr[i][0] = i;  for (int j = 1; j < ly; j++)  matr[0][j] = j;  for (int i = 1; i < lx; i++)  for (int j = 1; j < ly; j++)  {  w = x[i - 1] == y[j - 1] ? 0 : 1;  top = matr[i - 1][j];  left = matr[i][j - 1];  left\_top = matr[i - 1][j - 1];  matr[i][j] = std::min(left\_top + w, std::min(top + 1, left + 1));  }  return matr[lx - 1][ly - 1];  }  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 2 – Levenshtein.cpp

|  |
| --- |
| clock\_t t1 = 0, t2 = 0, t3 = 0, t4 = 0;  int lx = sizeof(s1);  int ly = sizeof(s2);  int s1\_size[]{ FIRST\_LEN / 25, FIRST\_LEN / 20, FIRST\_LEN / 15, FIRST\_LEN / 10, FIRST\_LEN / 5, FIRST\_LEN / 2, FIRST\_LEN };  int s2\_size[]{ SECOND\_LEN / 25, SECOND\_LEN / 20, SECOND\_LEN / 15, SECOND\_LEN / 10, SECOND\_LEN / 5, SECOND\_LEN / 2, SECOND\_LEN };  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 3 – LevenshteinMain.cpp

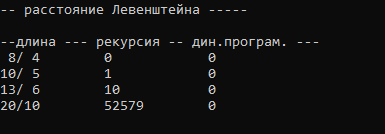


Рисунок 2 – Результат выполнения программы

**Задание 3**

Выполнить сравнительный анализ времени, затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от k. На рисунке 1.5 представлены графики зависимости времени вычисления от k. В консоль выводится k-ая часть строки.

Рисунок 3 – График зависимости выполнения

**Задание 4**

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

|  |  |  |  |
| --- | --- | --- | --- |
| 4. | Акр | Якорь | 20\*15, 15\*30, 30\*53, 53\*10, 10\*20, 20\*11 |

1. (3)
2. (4)
3. (2)
4. (3)
5. (5)

= 5.

= 4.

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

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

1. (4)

= 4.

= 3.

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

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

1. (2)
2. (2)
3. (3)

= 3.

= 2.

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

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

1. (2)
2. (1)
3. (2)

= 2.

= 1.

L(“А”, “Я”) = 1

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

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

Нетрудно убедиться, что для превращения слова «А» в слово «Я» достаточно заменить 1 букву.

1. (3)

= 3.

= 2.

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

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

1. (2)

= 2.

= 1.

L(“А”, “Я”) = 1

1. L(“Ак”, “Я”) = min(3,2,2) = 2
2. L(“Акр”, “Я”) = min(4,3,3) = 3
3. L(“А”, “Як”) = min(3,2,2) = 2
4. L(“Ак”, “Як”) = min(3,3,1) = 1
5. L(“Акр”, “Як”) = min(5,5,2) = 2
6. L(“А”, “Яко”) = min(4,4,3) = 3
7. L(“Ак”, “Яко”) = min(2,3,2) = 2
8. L(“Акр”, “Яко”) = min(4,3,2) = 2
9. L(“А”, “Якор”) = min(5,4,4) = 4
10. L(“Ак”, “Якор”) = min(4,3,3) = 3
11. L(“Акр”, “Якор”) = min(2,4,3) = 2
12. L(“А”, “Якорь”) = min(5,5,5) = 5
13. L(“Ак”, “Якорь”) = min(5,4,4) = 4
14. L(“Акр”, “Якорь”) = min(5,3,4) = 3

Таким образом самая короткая дистанция Левенштейна составляет 3 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).

|  |
| --- |
| #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] = { 100,15,20,43,70,40,71 }, 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;  } |

Листинг 5 – Matrix.cpp

|  |
| --- |
| #include "stdafx.h"  #include <memory.h>  // 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;  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  };  // 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 5 - MultiMatrix.cpp

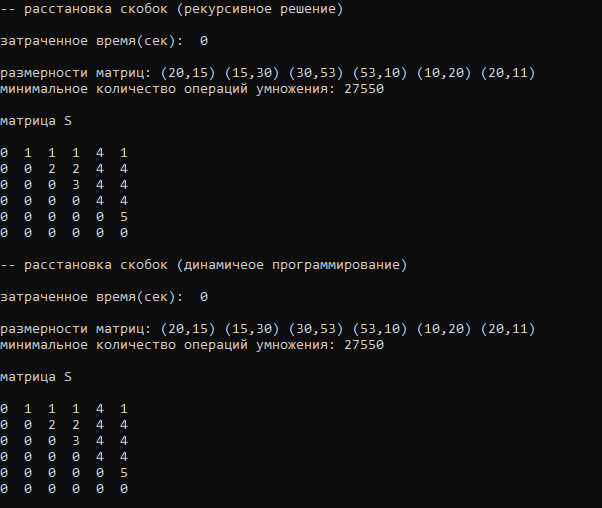


Рисунок 4 – выполнение программы

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

А1=20\*15,

А2=15\*30,

А3=30\*53,

А4 =53\*10,

А5 =10\*20,

А6 =20\*11.

Матрица S:

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
|  | **1** | **2** | **3** | **4** | **5** | **6** |
| **1** | 0 | 1 | 1 | 1 | 4 | 1 |
| **2** | 0 | 0 | 2 | 2 | 4 | 4 |
| **3** | 0 | 0 | 0 | 3 | 4 | 4 |
| **4** | 0 | 0 | 0 | 0 | 4 | 4 |
| **5** | 0 | 0 | 0 | 0 | 0 | 5 |
| **6** | 0 | 0 | 0 | 0 | 0 | 0 |

Найдем элемент (1,6) в матрице S, он равен 1. Это означает, что точка разрыва между 1-ой и 6-ой матрицей находится после 1-ой матрицы. Что позволяет расставить скобки следующим образом:

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

Точку разрыва между второй и шестой матрицей определяет элемент (2,6). Он равен 4. Следовательно разрыв будет после 4-ой матрицы.

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

Далее берем элемент (2,4) и получаем, что он равен 2. Следовательно получаем:

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

Это выражение и есть конечное.

Полученная расстановка скобок позволяет получить минимальное количество операций умножения, равное 27550.

**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.