Учреждение образования

«Белорусский государственный технологический университет»

**Лабораторная работа №4**

Динамическое программирование

Выполнила:

Студентка 2 курса 8 группы ИТ

Кононенко Игнатий Павлович

2025 г.

**Лабораторная работа №4. Динамическое программирование**

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

**Ход работы**

**Задание 1.**На языке С++ сгенерировать случайным образом строку букв латинского алфавита S1 длиной 300 символов и S2 длиной 200. И это представлено на рисунке 1.



Рисунок 1 Код на C++

А результат этой программы представлен на рисунке 2.

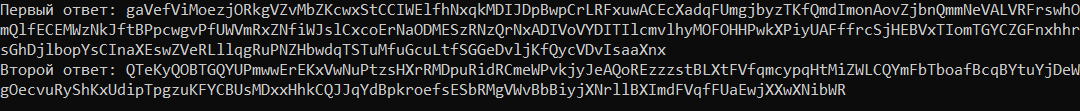
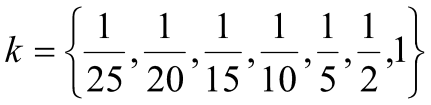


Рисунок 2 Результат программы

Разработанные функции располагаются в файле **Lab1.cpp**

**Задание 2.** Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – дистанцию Левенштейна для , где - длина строки ,  - строка состоящая из первых  символов строки . (копии экрана и код вставить в отчет).

Реализация функций показана в листинге 1, 2.

Листинг 1 – Код файла Levenshtein.cpp

|  |
| --- |
| // - 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 2 – Прототипы функций Levenshtein.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 3 – Код файла Main.cpp

|  |
| --- |
| #include <algorithm>  #include <iostream>  #include <ctime>  #include <iomanip>  #include <cstring>  #include "Levenshtein.h"  #include <Windows.h>  using namespace std;  char\* GenerateRandomString(int size)  {  char\* str = (char\*)malloc(sizeof(char) \* (size + 1));  for (int i = 0; i < size; i++) {  str[i] = rand() % 26 + 'a'; // 26 букв в алфавите  }  str[size] = '\0';  return str;  }  int main()  {  SetConsoleCP(1251);  SetConsoleOutputCP(1251);  const int threeHundred = 300;  const int twoHundred = 200;  char\* s1 = GenerateRandomString(threeHundred);  cout << "S1: " << endl;  for (int i = 0; i < threeHundred; i++) {  if (i % 50 == 0)  {  cout << "\n";  }  cout << s1[i];  }  cout << endl << endl;  srand(time(NULL) + 1);  char\* s2 = GenerateRandomString(twoHundred);  cout << "S2: " << endl;  for (int i = 0; i < twoHundred; 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 = strlen(s1);  int ly = strlen(s2);  int s1\_size[]{ threeHundred / 25, threeHundred / 20, threeHundred / 15, threeHundred / 10, threeHundred / 5, threeHundred / 2, threeHundred };  int s2\_size[]{ twoHundred / 25, twoHundred / 20, twoHundred / 15, twoHundred / 10, twoHundred / 5, twoHundred / 2, twoHundred };  cout << "\n\n-- расстояние Левенштейна -----";  cout << "\n\n--длина --- рекурсия -- дин.програм. ---\n";  for (int i = 0; i < min(sizeof(s1\_size) / sizeof(s1\_size[0]), sizeof(s2\_size) / sizeof(s2\_size[0])); 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;  } |

Вывод экрана представлен на рисунке 3.

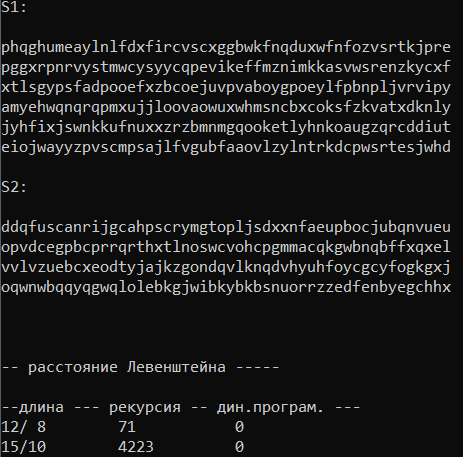


Рисунок 3 Результат программы

Так как мы видим из результата программы, что рекурсия возрастает экспоненциально(и это рассмотрено в третьем задании), то, чтобы дождаться завершение программы, понадобится очень много времени.

**Задание 3.** Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от k. (копии экрана и график вставить в отчет).

Рекурсивная функция – функция, которая вызывает саму себя. И, следовательно, чем больше значений, тем больше времени нужно ждать. Это можно увидеть на моем графике, где представлены результаты выполнения динамического программирования и рекурсии от количества k символов на рисунке 4.

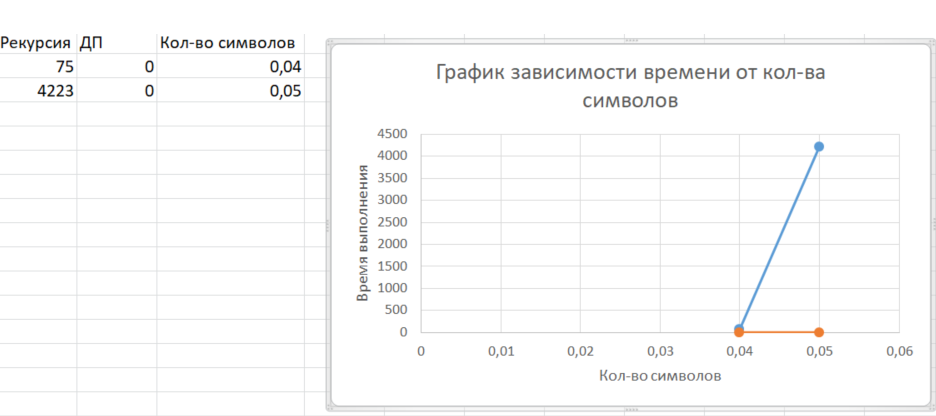


Рисунок 4 Графики рекурсии и ДП

Здесь мы можем убедиться, что лучше использовать динамическое программирование, чем рекурсию. А что такое динамическое программирование?

Динамическое программирование – способ решения сложных задач путем разбиения их на более простые подзадачи. Поэтому здесь будет так мало затраченного времени.

**Задание 4.** Реализовать вручную пример вычисления дистанции Левенштейна при помощи рекурсивного алгоритма (в соответствии с вариантом) (каждый шаг алгоритма по примеру из лекции вставить в отчет).

Для моего варианта нужно было взять два слова: «Сом» и «Домик». А далее я реализовывал алгоритм Левенштейна вручную, и это показано на рисунке 5.

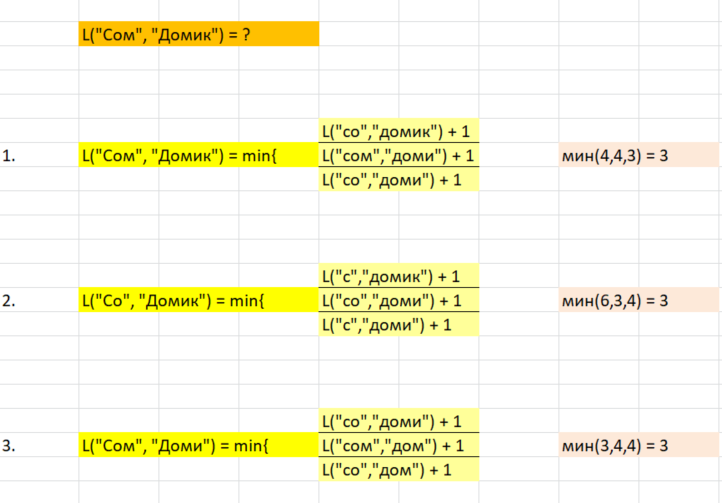


Рисунок 5 Решение задачи

В конечном итоге, у нас получилось число 3. Я решил свериться с программой, которую показывал ранее. В результате на рисунке 6 нам тоже показало число 3.



Рисунок 6 Результат программы

Так что мы сделали правильно.

**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.

Реализация функций показана в листинге 4, 5.

Листинг 4 – Код файла MultiMatrix.cpp

|  |
| --- |
| // --- 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 5 – Прототипы функций в MultiMatrix.h

|  |
| --- |
| // --- 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  ); |

Взяли свои значения по варианту(9\*12, 12\*20, 20\*23, 23\*30, 30\*40, 40\*51).

Результат программы показан на рисунке 7.

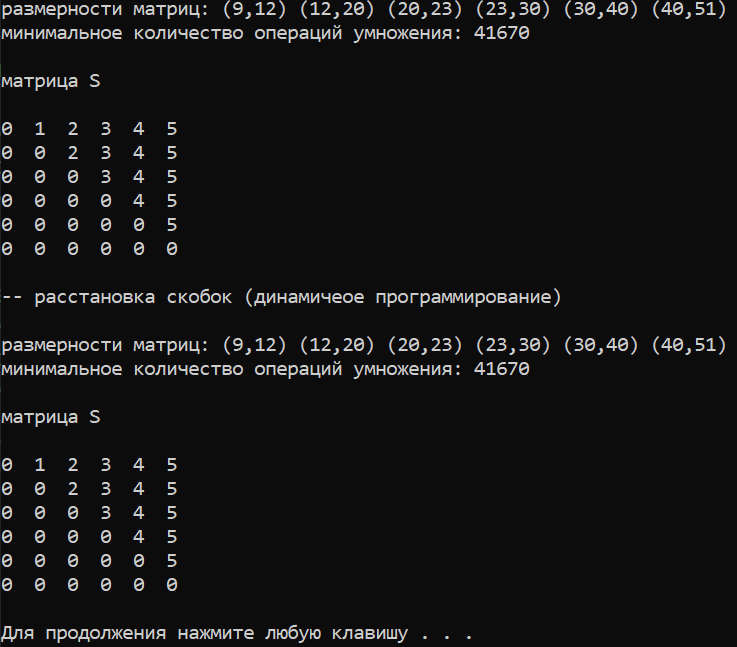


Рисунок 7 Результат программы

В итоге, нам вывело наилучший путь умножения.

**Вывод:** В лабораторной работе №4 были освоены общие принципы решения задач методом динамического программирования, сравнить полученные решения задач с рекурсивным методом.