Учреждение образования

«БЕЛОРУССКИЙ ГОСУДАРСТВЕННЫЙ ТЕХНОЛОГИЧЕСКИЙ УНИВЕРСИТЕТ»

Математическое программирование

Студент: Смолик В. А.

ФИТ 2 курс 4 группа

Минск 2023

**Лабораторная работа №1. Вспомогательные функции**

**Цель работы:** приобретение навыков составления и отладки программ с использованием пользовательских функций для замера продолжительности процесса вычисления.

***Задание 1.*** Разработайте три функции (start, dget и iget), используя следующие спецификации

|  |
| --- |
| #include "stdafx.h"  #include "Auxil.h"  #include <ctime>  namespace auxil  {  void start() // старт генератора сл. чисел  {  srand((unsigned)time(NULL));  };  double dget(double rmin, double rmax) // получить случайное число  {  return ((double)rand() / (double)RAND\_MAX) \* (rmax - rmin) + rmin;  };  int iget(int rmin, int rmax) // получить случайное число  {  return (int)dget((double)rmin, (double)rmax);  };  } |

Листинг 1.1 — Файл Auxil.cpp

|  |
| --- |
| #pragma once  #include <cstdlib>  namespace auxil  {  void start(); // старт генератора сл. чисел  double dget(double rmin, double rmax); // получить случайное число  int iget(int rmin, int rmax); // получить случайное число  }; |

Листинг 1.2 — Файл Auxil.h

***Задание 2***

1. Реализовать пример 2.
2. Для проверки работоспособности разработанных функций и приобретения навыков замера продолжительности процесса вычисления реализуйте программу, приведенную в примере 2.

|  |
| --- |
| #include "stdafx.h"  #include "Auxil.h" // вспомогательные функции  #include "Fibonachi.h"  #include <iostream>  #include <ctime>  #include <locale>  #define CYCLE 1000000 // количество циклов  int main(int argc, char\* argv[])  {  double av1 = 0, av2 = 0;  clock\_t t1 = 0, t2 = 0;  setlocale(LC\_ALL, "rus");  auxil::start(); // старт генерации  t1 = clock(); // фиксация времени  for (int i = 0; i < CYCLE; i++)  {  av1 += (double)auxil::iget(-100, 100); // сумма случайных чисел  av2 += auxil::dget(-100, 100); // сумма случайных чисел  }  t2 = clock(); // фиксация времени  cout << "\nколичество циклов: " << CYCLE;  cout << "\nсреднее значение (int): " << av1 / CYCLE;  cout << "\nсреднее значение (double): " << av2 / CYCLE;  cout << "\nпродолжительность (у.е): " << (t2 - t1);  cout << "\n (сек): "  <<((double)(t2 - t1)) / ((double)CLOCKS\_PER\_SEC);  cout << endl; q  system("pause");  return 0;  } |

Листинг 1.3 — Пример 2

**Задание 3**

Проведите необходимые эксперименты и постройте график зависимости (Excel) продолжительности процесса вычисления от количества циклов в примере 2. Проанализируйте характер зависимости. Проведите исследование любого другого рекурсивного алгоритма, например, вычисления факториала или генератора чисел Фибоначчи (прим. – например вычислите каким будет 100-е, 200-е, 300-е и т.д число), и включите в отчет график.

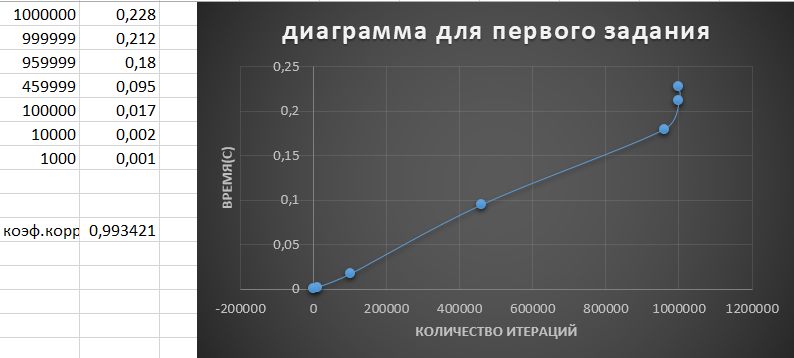


Рисунок 1.1 — График зависимости продолжительности процесса вычисления от количества циклов

Найдем коэффициент корреляции для данной зависимости:



Рисунок 1.2 — Коэффициент корреляции

Так как коэффициент приблизительно равен единице, можем сделать вывод о том, что зависимость продолжительности процесса вычисления от количества цикла имеет линейный вид.

**Числа Фибоначчи:**

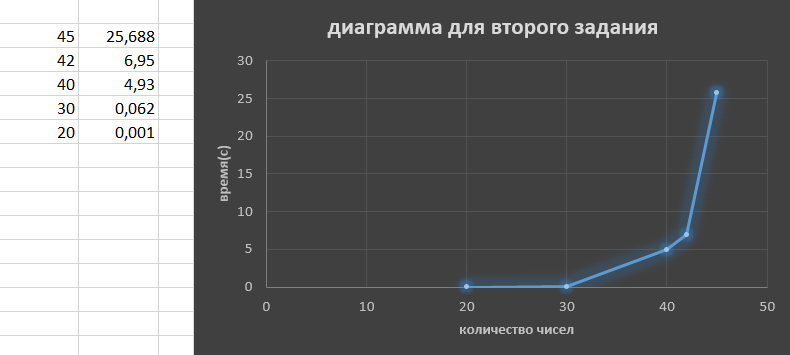
Последовательность чисел Фибоначчи определяется формулой Fn = Fn-1 + Fn-2. То есть, следующее число получается как сумма двух предыдущих.

Первые два числа равны 1, затем 2(1+1), затем 3(1+2), 5(2+3) и так далее: 1, 1, 2, 3, 5, 8, 13, 21....

Реализацию данного алгоритма и график зависимости времени выполнения от заданного числа можно посмотреть в листинге 1.4 и на рисунке 1.3., соответственно.

|  |
| --- |
| //-- Fibonachi.cpp  #include "Fibonachi.h"  #include <iostream>  // Посчитать N-ое число Фибоначчи  int fibonachi(int end) {  if (end == 0)  return 0;  if (end == 1)  return 1;  else {  return fibonachi(end - 1)+fibonachi(end-2);  }  }//-- Lab1.cpp  const int a = 45;  clock\_t t3 = 0, t4 = 0;  t3 = clock();    for (int number = 0; number < a; number++)  {  cout << fibonachi(number)<<"\t";  }  t4 = clock();  cout << "\n\nНомер числа, которое необходимо было вычислить : " << a;  cout << "\nпродолжительность (у.е): " << (t4 - t3);  cout << "\n (сек): " << ((double)(t4 - t3)) / ((double)CLOCKS\_PER\_SEC);  cout << endl;  system("pause"); |

Листинг 1.4 — Реализация алгоритма чисел Фибоначчи и тестирование

Рисунок 1.3 — График зависимости продолжительности процесса вычисления от количества циклов

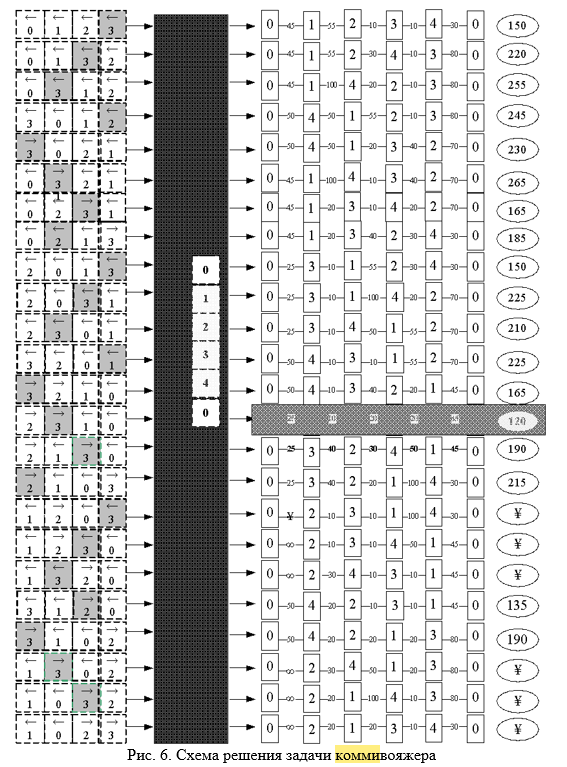
Вывод: скорость выполнения программы линейно зависит от количества итераций цикла. Скорость нахождения N-го числа Фибоначчи имеет вид показательной зависимости.

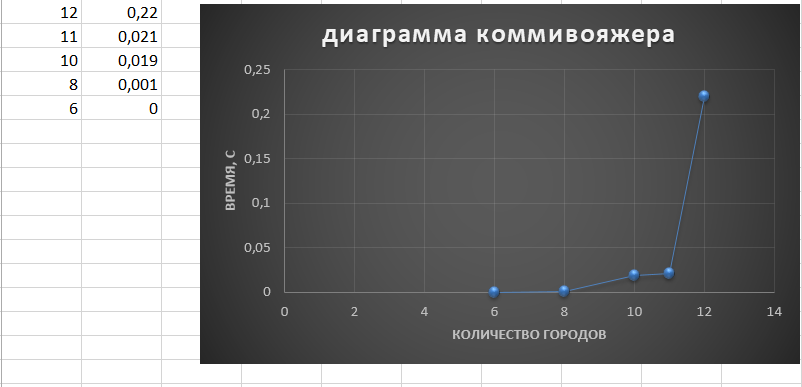
**Лабораторной работе 2. Комбинаторные алгоритмы решения оптимизационных задач**

**Вариант 13**

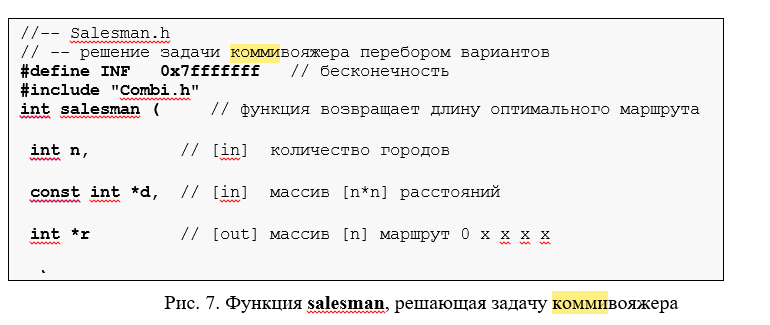
**Задание 5,6**

Задача коммивояжера является одной из известных задач комбинаторной оптимизации и решается с помощью генератора перестановок.





Заголовочный файл salesman.h содержит прототип функции salesman , со входными параметрами : ”n—количество городов, массив d – массив расстояний; r – содержит маршрут из 0-го рода в n-ый” .



Salesman.cpp содержит внутри себя реализацию функции salesman , а также вспомогательные функции, такие как sum, firstpath, source, copypath, distance, indx.

Функция sum,в зависимости от значений, возвращает сумму двух переменных, переданных из функции salesman.

Функция firstpath принимает в качестве параметра количество городов, и формирует 1-ый маршрут. Возвращает массив, содержащий 1-ый переход от 1-о города до всех остальных , возвращаясь в первый город.

Функция source принимает количество городов, как и предыдущая функция, однако формирует исходный массив.

Функция copypath заполняет значениями первый массив(на первой итерации это пустой массив), значениями из второго массива, который был сформирован функцией indx.

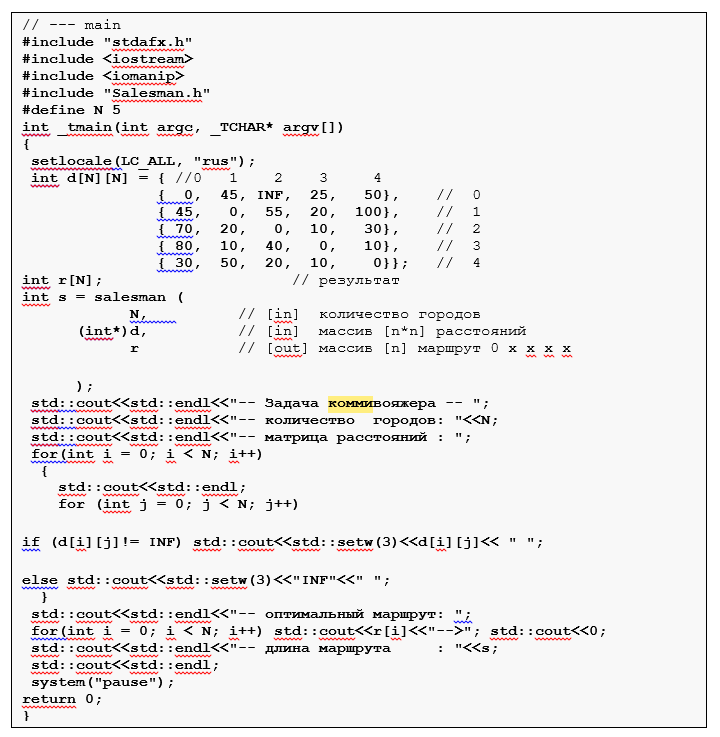
Функция distance возвращает сумму длины дуг (длину маршрута из 0 города во все и снова в 0 город), путем прохождения по каждой дуге и сохраняя результат в переменную rc.

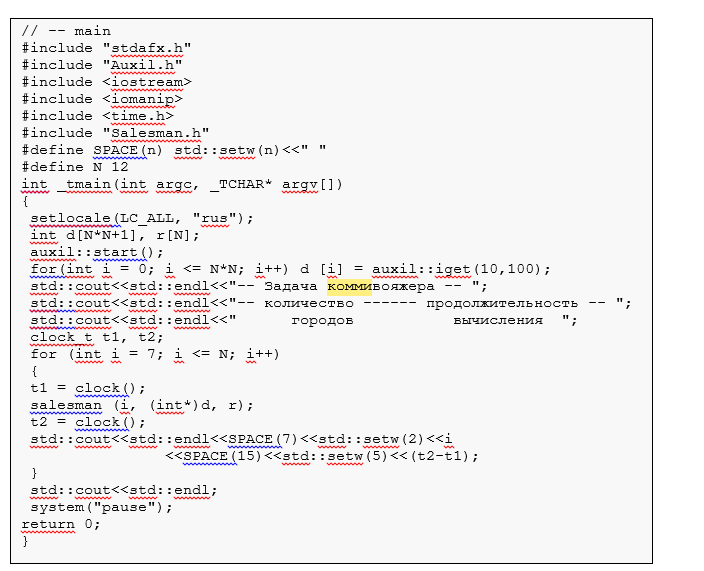
Функция indx перестраивает маршрут(так как передан по значению) , путем применения генератора перестановок.

Функция Salesman возвращает минимальное расстояние от 0 города ко всем остальным. Параметры функции описаны в Salesman.h. Фукнция внутри себя вызывает функции sourse(n),firstpath(n), тем самым инициализируя 2 массива, содержащие внутри себя маршруты, при помощи конструктора структуры перестановок(permutation), создает объект этой структуры. При помощи функции getfirst инициализируется переменная k(присвается 0 , так как начинается с 0-го города).Далее в цикле , на каждой итерации создается новый маршрут, который в последствии передаётся в функцию distance и при помощи if сравнивается с минимальным, до этой итерации , маршрутом. Если минимальный до этой итарации маршрут больше , то ему присваивается значение функции distance и новый путь копируется в массив пути r. Далее переменной k присваивается новая вершина.

|  |
| --- |
| #include "stdafx.h"  #include "Salesman.h"  int sum(int x1, int x2) // суммирование с учетом бесконечности  {  return (x1 == INF || x2 == INF ) ? INF : (x1 + x2);  };  int\* firstpath(int n) // формирование 1го маршрута 0,1,2,..., n-1, 0  {  int\* rc = new int[n + 1];  rc[n] = 0;  for (int i = 0; i < n; i++)  rc[i] = i;  return rc;  };  int\* source(int n) // формирование исходного массива 1,2,..., n-1  {  int\* rc = new int[n - 1];  for (int i = 1; i < n; i++)  rc[i - 1] = i;  return rc;  };  void copypath(int n, int\* r1, const int\* r2) // копировать маршрут  {  for (int i = 0; i < n; i++)  r1[i] = r2[i];  };  int distance(int n, int\* r, const int\* d) // длина маршрута  {  int rc = 0;  for (int i = 0; i < n - 1; i++) {  rc = sum(rc, d[r[i] \* n + r[i + 1]]);  if (rc == 0)  return INF;  }    return sum(rc, d[r[n - 1] \* n + 0]); //+ последняя дуга (n-1,0)  };  void indx(int n, int\* r, const int\* s, const short\* ntx)  {  for (int i = 1; i < n; i++)  r[i] = s[ntx[i - 1]];  }  int salesman(  int n, // [in] количество городов  const int\* d, // [in] массив [n\*n] расстояний  int\* r // [out] массив [n] маршрут 0 x x x x  )  {  int\* s = source(n),  \* b = firstpath(n),  rc = INF,  dist = 0;  combi::permutation p(n - 1);  int k = p.getfirst();  while (k >= 0) // цикл генерации перестановок  {  indx(n, b, s, p.sset); // новый маршрут  if ((dist = distance(n, b, d)) < rc)  {  rc = dist;  copypath(n, r, b);  }  k = p.getnext();  };  return rc;  } |

Инициализация массива расстояний , стандартный вывод.





**Лабораторная работа №3**

**МЕТОД ВЕТВЕЙ И ГРАНИЦ**

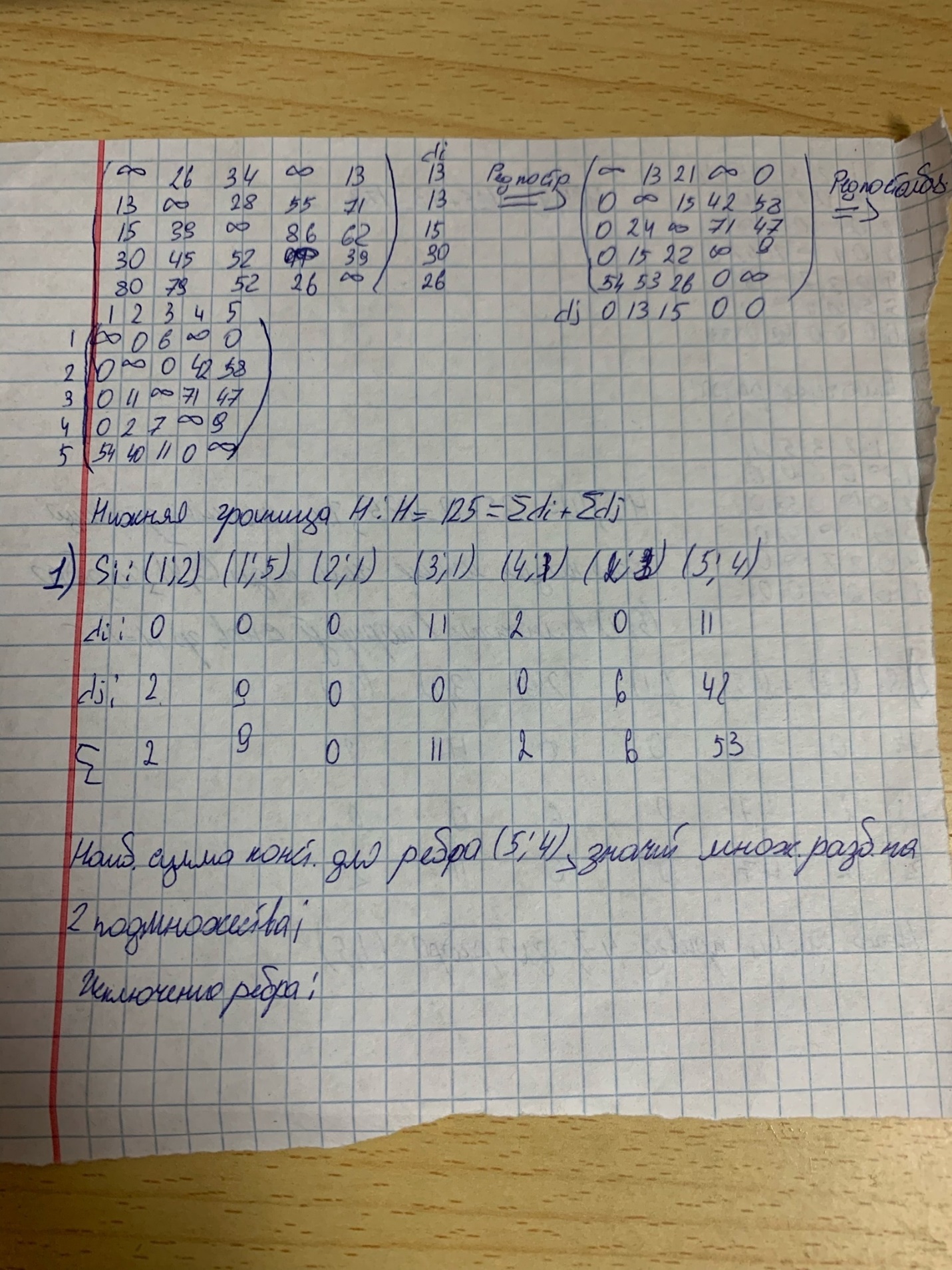
**ЦЕЛЬ РАБОТЫ:** освоить общие принципы решения задач методом ветвей и границ, решить задачу о коммивояжере данным методом, сравнить полученное решение задачи с комбинаторным методом перестановок.

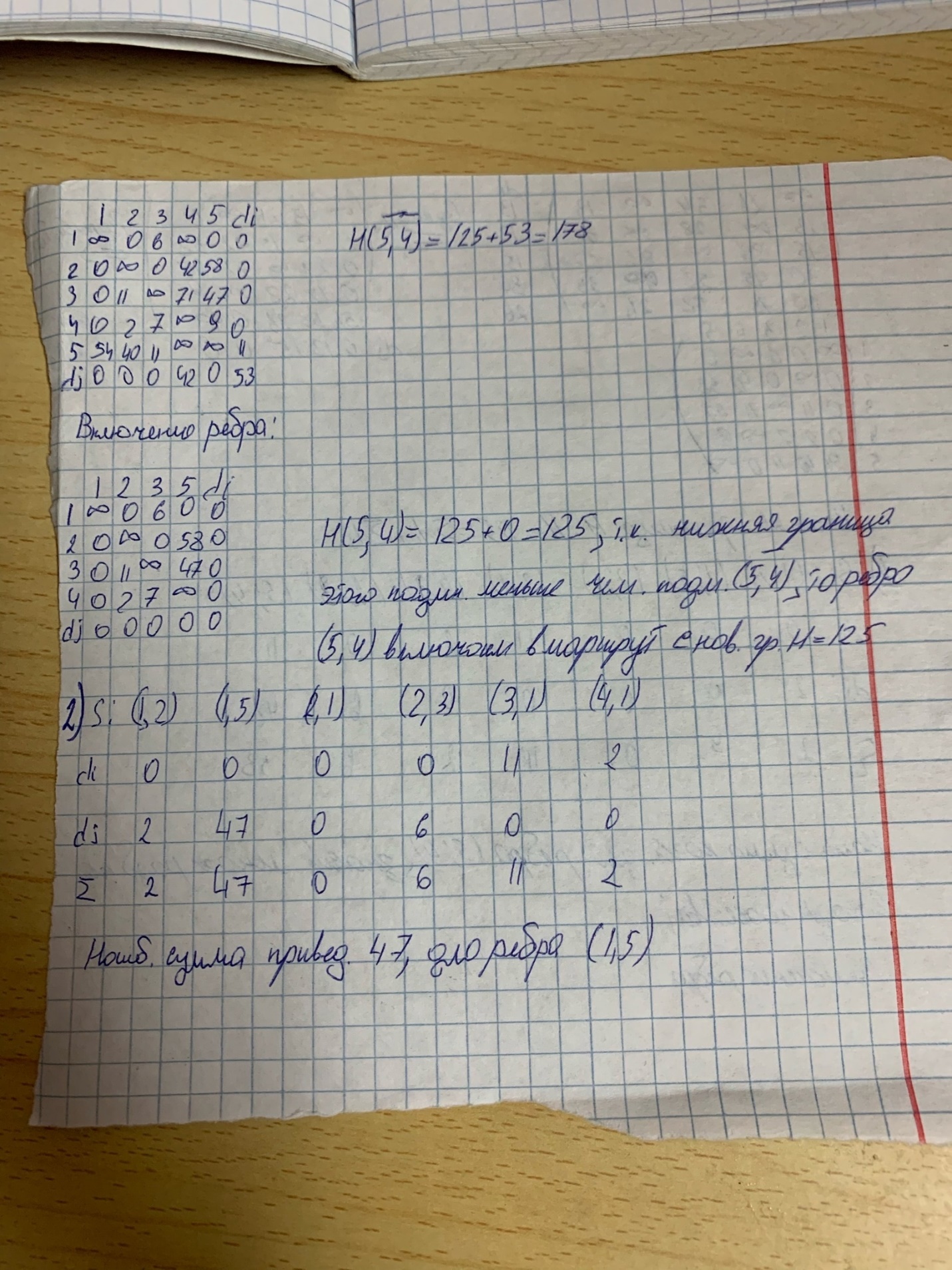
**Задание 1.** Сформулировать условие задачи коммивояжера с параметром.

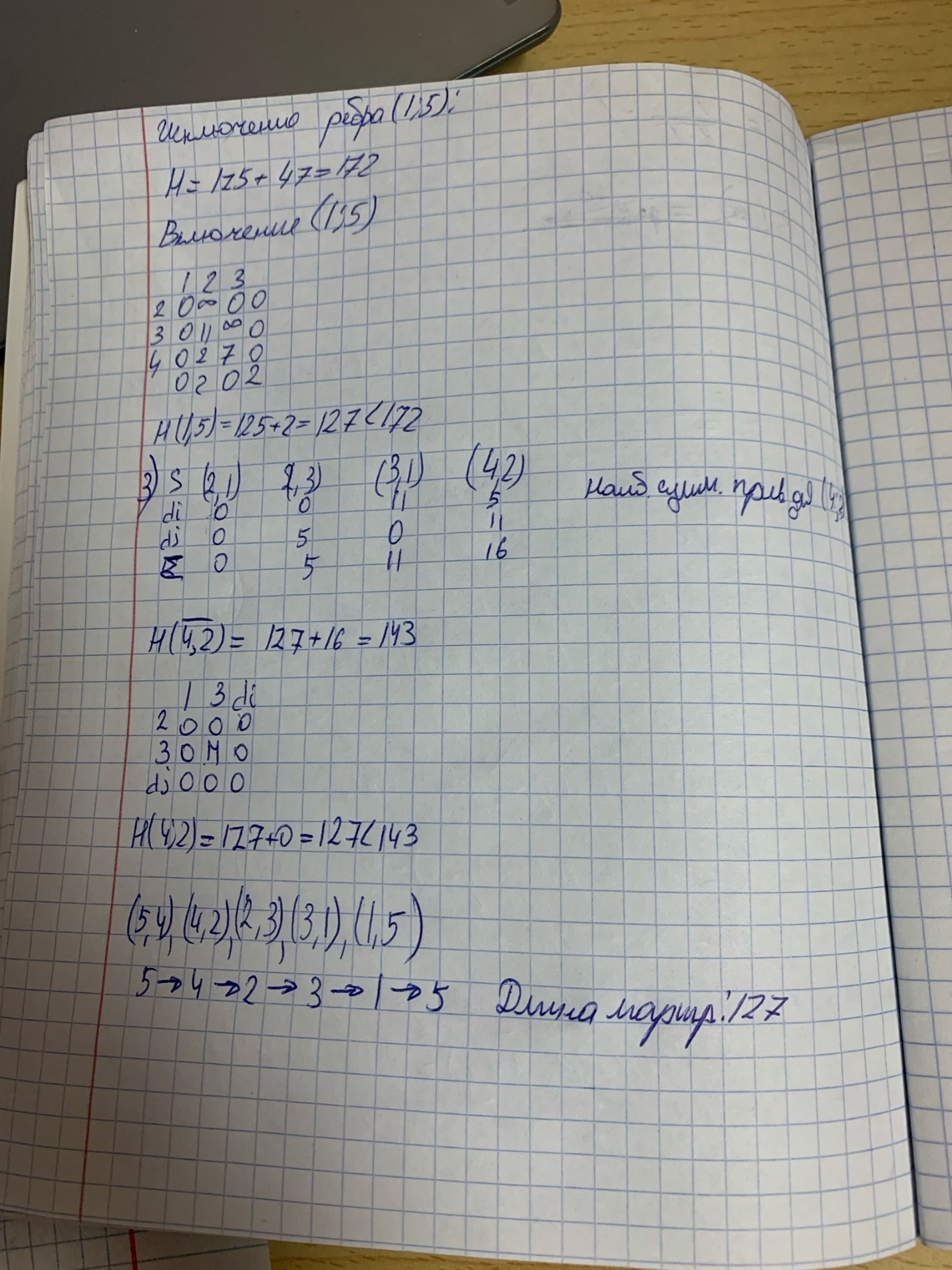
|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Город** | **1** | **2** | **3** | **4** | **5** |
| **1** | INF | 26 | 34 | INF | 13 |
| **2** | 13 | INF | 28 | 55 | 71 |
| **3** | 15 | 39 | INF | 86 | 62 |
| **4** | 30 | 45 | 52 | INF | 39 |
| **5** | 80 | 79 | 52 | 26 | INF |

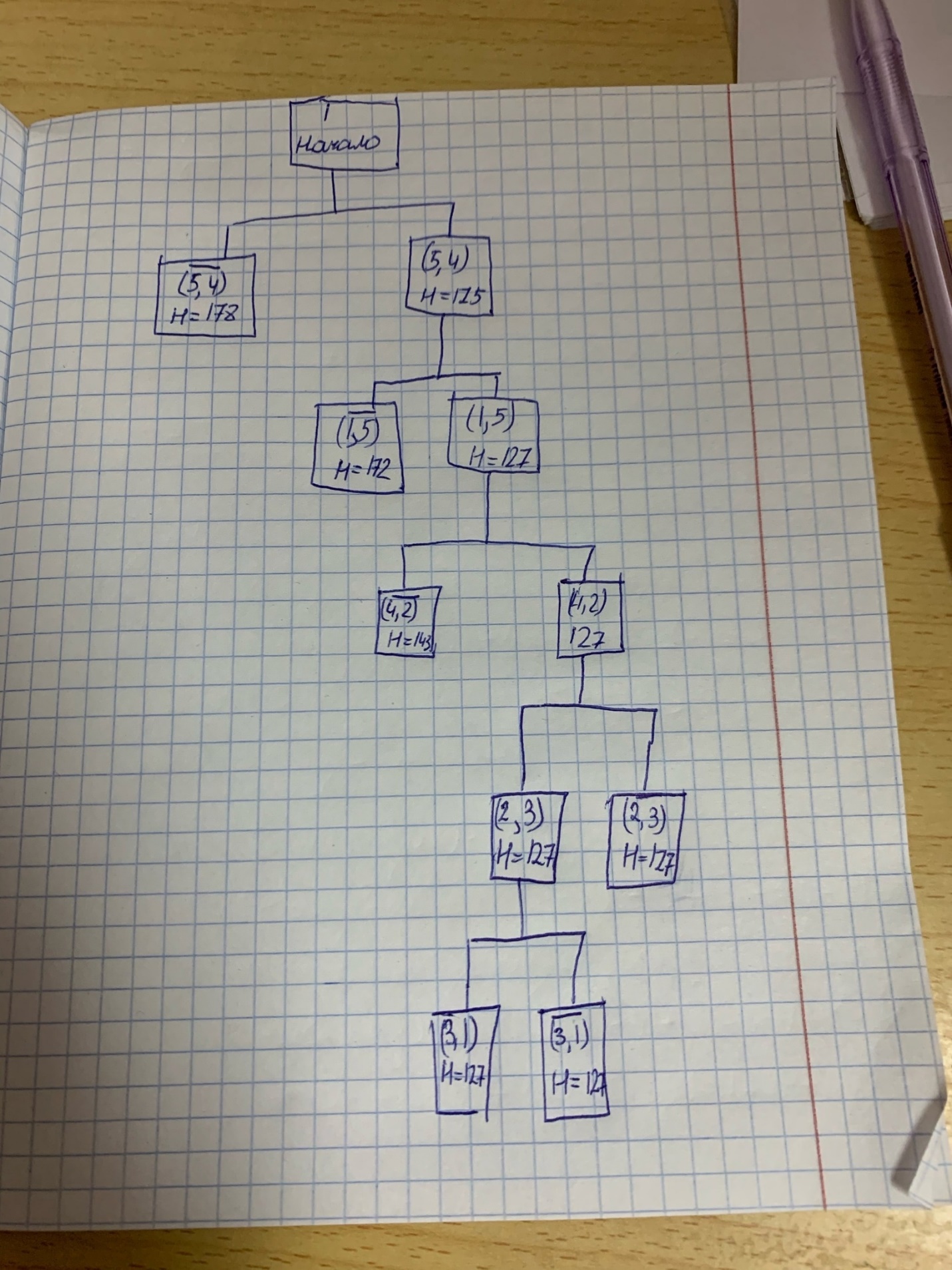
где *n* – номер варианта или номер по журналу (13);

**Задание 2.** Решить сформулированную задачу методом ветвей и границ.

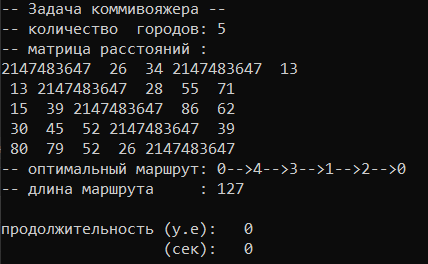








**Задание 3.** Проверить полученное решение при помощи генератора перестановок (см. лаб. 2, задание 5.1.) и включить копию экрана с решением в отчет.



**Вывод:** В этой лабораторной работе были освоены общие принципы решения задач методом ветвей и границ, решена задача о коммивояжере данным методом, а также сравнено полученное решение задачи с комбинаторным методом перестановок.

**Лабораторная работа 4**

**ДИНАМИЧЕСКОЕ ПРОГРАММИРОВАНИЕ**

**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 200.

|  |
| --- |
| char\* generateString(int len) {  char \*arr = new char[len];  char symbols[27] = "abcdefghijklmnopqrstuvwxyz";  int random;  for (size\_t i = 0; i < len; i++)  {  random = rand() % 26;  arr[i] = symbols[random];  }  std::cout << "Generated string:";  for (size\_t i = 0; i < len; i++)  {  if (i % 50 == 0) {  std::cout << std::endl;  }  std::cout << arr[i];  }  std::cout << std::endl<<std::endl;  return arr;  } |

Листинг 1 — gen.cpp

***Задание 2.***

Вычислить двумя способами (рекурсивно и с помощью динамического программирования)  – дистанцию Левенштейна для , где - длина строки ,  - строка состоящая из первых  символов строки .

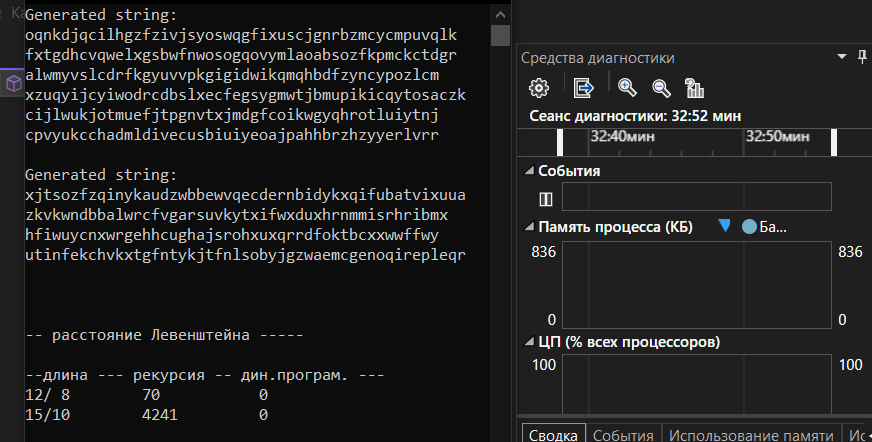


Рисунок 1 – Результат нахождения дистанции Левенштейна

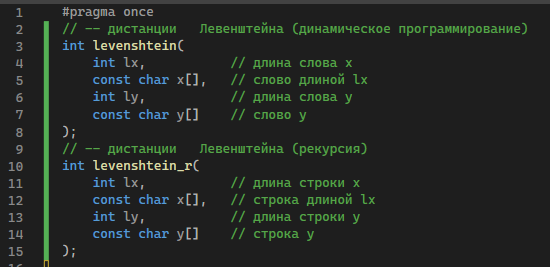


Рисунок 2 – Заголовочный файл с прототипами функций

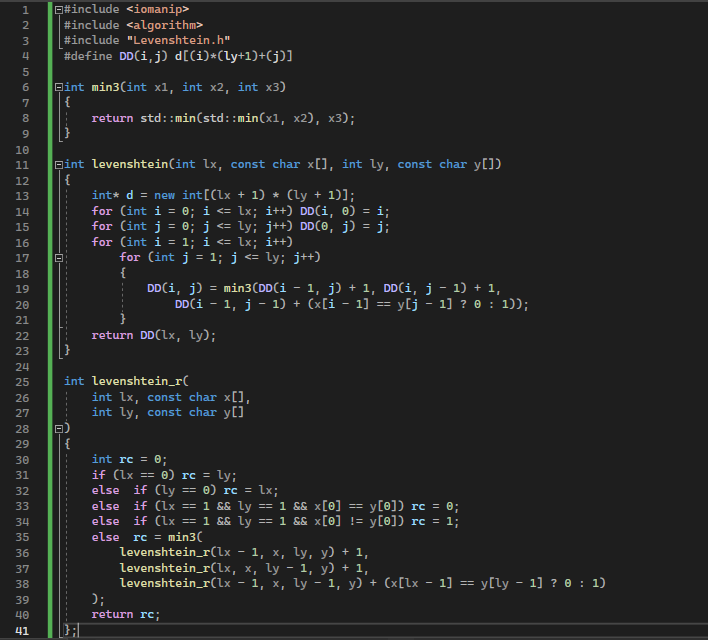


Рисунок 3 – Реализация функций

|  |
| --- |
| //main  #include<iostream>  #include"Levenshtein.h"  #include<iomanip>  using namespace std;  void main() {  setlocale(LC\_ALL, "rus");    char \*S1 = Leven::generateString(300);  char \*S2 = Leven::generateString(200);  clock\_t t1 = 0, t2 = 0, t3, t4;  char x[] = "abcdefghklmnoxm", y[] = "xyabcdefghomnkm";  int lx = sizeof(x) - 1, ly = sizeof(y) - 1;  std::cout << std::endl;  std::cout << std::endl << "-- расстояние Левенштейна -----" << std::endl;  std::cout << std::endl << "--длина --- рекурсия -- дин.програм. ---"  << std::endl;  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 };  for (int i = 0; i < min(lx, ly); i++)  {  t1 = clock();  Leven::levenshtein\_r(s1\_size[i], S1, s2\_size[i], S2);  t2 = clock();  t3 = clock();  Leven::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");  } |

Листинг 2 — main.cpp

***Задание 3.***

Выполнить сравнительный анализ времени затраченного на вычисление дистанции Левенштейна для двух методов решения. Построить графики зависимости времени вычисления от .

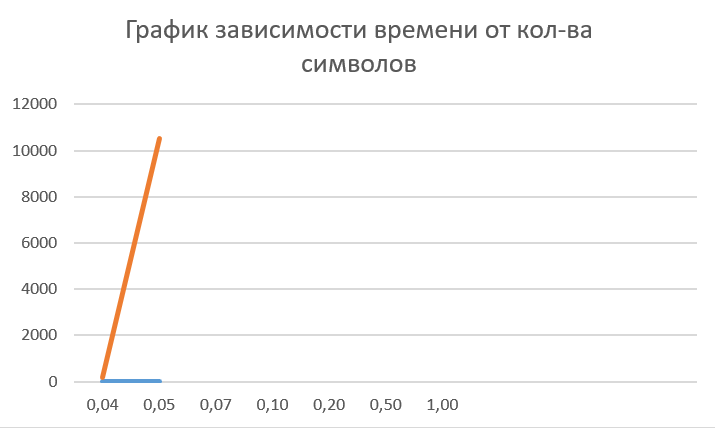


Рисунок 4 – График зависимости выполнения

Из результатов работы программы можно заметить, что подход динамического программирования в разы эффективней рекурсии.

***Задание 4.***

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



Рисунок 5 – Вариант для вычисления дистанции Левенштейна

**Рекурсивное погружение:**



= 5.

= 4.



= 3.

= 2.



= 4.

= 3.



= 2.

= 1.



= 1.

= 1.

= 0.



= 2.

= 1.



= 3.

= 2.

**Восходящее вычисление:**

1. L(“Р”, “Б”) = min(2,2,1) = 1
2. L(“Ра”, “Б”) = min(2,3,2) = 2
3. L(“Р”, “Ба”) = min(3,2,2) = 2
4. L(“Ра”, “Ба”) = min(3,3,1) = 1
5. L(“Раб”, “Б”) = min(3,4,2) = 2
6. L(“Раб”, “Ба”) = min(2,3,3) = 2
7. L(“Р”, “Бар”) = min(4,3,2) = 2
8. L(“Р”, “Барк”) = min(5,3,4) = 3
9. L(“Ра”, “Бар”) = min(4,2,3) = 2
10. L(“Ра”, “Барк”) = min(5,3,4) = 3
11. L(“Раб”, “Бар”) = min(3,3,2) = 2
12. L(“Р”, “Барка”) = min(6,4,5) = 4
13. L(“Ра”, “Барк”) = min(4,3,3) = 3
14. L(“Раб”, “Барк”) = min(4,3,3) = 3
15. L(“Ра”, “Барка”) = min(5,3,3) = 3
16. L(“Раб”, “Барка”) = min(4,4,4) = 4

***Задание 5.***

**Нечетные варианты**. Выполнить сравнительный анализ времени затраченного на решение задачи о наибольшей общей подпоследовательности для двух методов решения (рекурсивное решение, динамическое программирование). Две последовательности взять в соответствии с вариантом.

Листинги кода :

|  |
| --- |
| #pragma once  //- LCH.h динамически  int lcsd(  const char x[], // последовательность X  const char y[], // последовательность Y  char z[] // наибольшая общая подпоследовательность  ); |

Листинг 3 — LCH.h

|  |
| --- |
| #pragma once  //рекурсивно  int lcs(  int lenx, // длина последовательности X  const char x[], // последовательность X  int leny, // длина последовательности Y  const char y[] // последовательность Y  ); |

Листинг 4 — LCS.h

|  |
| --- |
| //динамически  #include <cstring>  #include "LCH.h"  #define LCS\_C(x1,x2) (C[(x1)\*(leny+1)+(x2)])  #define LCS\_B(x1,x2) (B[(x1)\*(leny+1)+(x2)])  #define LCS\_X(i) (x[(i)-1])  #define LCS\_Y(i) (y[(i)-1])  #define LCS\_Z(i) (z[(i)-1])  enum Dart { TOP, LEFT, LEFTTOP };  void getLCScontent(int lenx, int leny, const char x[],  const Dart\* B,  int n, int i, int j, char z[])  {  if ((i > 0 && j > 0 && n > 0))  {  if (LCS\_B(i, j) == LEFTTOP)  {  getLCScontent(lenx, leny, x, B, n - 1, i - 1, j - 1, z);  LCS\_Z(n) = LCS\_X(i);  LCS\_Z(n + 1) = 0;  }  else if (LCS\_B(i, j) == TOP)  getLCScontent(lenx, leny, x, B, n, i - 1, j, z);  else getLCScontent(lenx, leny, x, B, n, i, j - 1, z);  }  };  int lcsd(const char x[], const char y[], char z[])  {  int n;  int lenx = strlen(x), leny = strlen(x),  \* C = new int[(lenx + 1) \* (leny + 1)];  Dart\* B = new Dart[(lenx + 1) \* (leny + 1)];  memset(C, 0, sizeof(int) \* (lenx + 1) \* (leny + 1));  for (int i = 1; i <= lenx; i++)  for (int j = 1; j <= leny; j++)  if (LCS\_X(i) == LCS\_Y(j))  {  LCS\_C(i, j) = LCS\_C(i - 1, j - 1) + 1;  LCS\_B(i, j) = LEFTTOP;  }  else if (LCS\_C(i - 1, j) >= LCS\_C(i, j - 1))  {  LCS\_C(i, j) = LCS\_C(i - 1, j);  LCS\_B(i, j) = TOP;  }  else  {  LCS\_C(i, j) = LCS\_C(i, j - 1);  LCS\_B(i, j) = LEFT;  }  getLCScontent(lenx, leny, x, B, LCS\_C(lenx, leny), lenx, leny, z);  return LCS\_C(lenx, leny);  }  #undef LCS\_Z  #undef LCS\_C  #undef LCS\_B  #undef LCS\_X  #undef LCS\_Y |

Листинг 5 — LCH.cpp

|  |
| --- |
| #include <algorithm>  #include "LCS.h"  //рекурсивно  int lcs(int lenx, const char x[],  int leny, const char y[])  {  int rc = 0;  if (lenx > 0 && leny > 0)  {  if (x[lenx - 1] == y[leny - 1]) rc = 1 + lcs(lenx - 1, x, leny - 1, y);  else rc = std::max(lcs(lenx, x, leny - 1, y), lcs(lenx - 1, x, leny, y));  }  return rc; //длина LCS  } |

Листинг 6 —LCS.cpp

|  |
| --- |
| #include<iostream>  #include"Levenshtein.h"  #include<iomanip>  #include"LCH.h"  #include"LCS.h"  using namespace std;  void main() {  setlocale(LC\_ALL, "rus");    /\*char S1[300] = { 0 };  char \*S2 = Leven::generateString(200);\*/  clock\_t t1 = 0, t2 = 0, t3, t4;  char x[] = "abcdefghklmnoxm", y[] = "xyabcdefghomnkm";  int lx = sizeof(x) - 1, ly = sizeof(y) - 1;  std::cout << std::endl;  std::cout << std::endl << "-- расстояние Левенштейна -----" << std::endl;  std::cout << std::endl << "--длина --- рекурсия -- дин.програм. ---"  << std::endl;  for (int i = 8; i < std::min(lx, ly); i++)  {  t1 = clock();  Leven::levenshtein\_r(i, x, i - 2, y);  t2 = clock();  t3 = clock();  Leven::levenshtein(i, x, i - 2, y);  t4 = clock();  std::cout << std::right << std::setw(2) << i - 2 << "/" << std::setw(2) << i  << " " << std::left << std::setw(10) << (t2 - t1)  << " " << std::setw(10) << (t4 - t3) << std::endl;  }  system("pause");  ///////////////////////////////////////////////////////////////////////////////////////////  char z[100] = "";  char x1[] = "BXWAFRE",  y1[] = "XCDUFR";  clock\_t start1 = clock();  int l = lcsd(x1, y1, z);  clock\_t end1 = clock();  std::cout << std::endl  << "-- наибольшая общая подпоследовательость - LCS(динамическое"  << "программирование)" << std::endl;  std::cout << std::endl << "последовательость X: " << x1;  std::cout << std::endl << "последовательость Y: " << y1;  std::cout << std::endl << " LCS: " << z;  std::cout << std::endl << " длина LCS: " << l;  std::cout << std::endl;  system("pause");  ///////////////////////////////////////////////////////////////////////////////////////////  char X[] = "BXWAFRE", Y[] = "XCDUFR";  std::cout << std::endl << "-- вычисление длины LCS для X и Y(рекурсия)";  std::cout << std::endl << "-- последовательность X: " << X;  std::cout << std::endl << "-- последовательность Y: " << Y;  clock\_t start2 = clock();  int s = lcs(  sizeof(X) - 1, // длина последовательности X  "BXWAFRE", // последовательность X  sizeof(Y) - 1, // длина последовательности Y  "XCDUFR" // последовательность Y  );  clock\_t end2 = clock();  std::cout << std::endl << "-- длина LCS: " << s << std::endl;  cout << "Время затраченное на нахождение обшей подпоследовательности(ДП): " << ((float)start1 - (float)end1) / CLOCKS\_PER\_SEC<<" сек.\n";  cout << "Время затраченное на нахождение обшей подпоследовательности(рекурсия): " << ((float)start2 - (float)end2) / CLOCKS\_PER\_SEC << " сек.\n";  system("pause");  } |

Листинг 7 — main.cpp

Выполнив сравнительный анализ времени затраченного на решение задачи о наибольшей общей подпоследовательности 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 5:

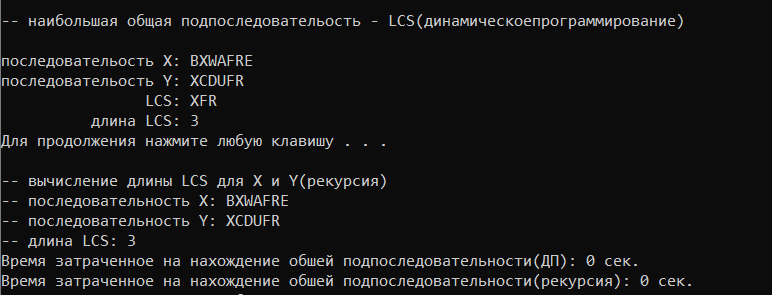


Рисунок 5—Результаты функций нахождения общей подпоследовательности

На рисунке 6 представлен график зависимости от суммарной длинны двух подстрок и необходимом времени выполнения при помощи динамического алгоритма:



Рисунок 6— график зависимости

**Алгоритм прохождения:**

1. Создаем двумерный массив размером (len(BXWAFRE) + 1) на (len(BXWAFRE) + 1), где каждый элемент равен 0.

2. Проходим по каждой строке и столбцу массива, начиная с первого и до последнего.

3. Если значение строки или столбца равно 0, то элемент массива на пересечении этой строки и столбца остается равным 0.

4. Если символ строки и столбца равны, то элемент массива на пересечении этой строки и столбца равен значению элемента на предыдущей диагонали плюс 1, т.е. с\_(𝑛,𝑝) = c\_(𝑛−1,𝑝−1) + 1.

5. Если символ строки и столбца не равны, то элемент массива на пересечении этой строки и столбца равен максимуму между значением элемента на предыдущей строке и значением элемента на предыдущем столбце, т.е. c\_(𝑛,𝑝) = max( с\_(𝑛,𝑝"−1" ), c\_(𝑛−1,𝑝)).

6. После прохода по всем элементам массива, наибольшая общая подпоследовательность будет равна значению элемента на пересечении последней строки и последнего столбца.

**Вывод:** динамический подход к решению задач позволяет выполнять их значительно быстрее, чем рекурсивный, особенно это будет заметно при решении задач с большим объёмом информации, вызвано это тем, что данные кэшируются , а не вычисляются каждый раз заново. Так же, в результате лабораторной работы 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.