**Отчёт по лабораторным работам**

**Вариант 1**

**Лабораторная работа 2. Комбинаторные алгоритмы решения оптимизационных задач**

**Студент:**

**Адамович Карианна Павловна, 2-3**

**Преподаватель:**

**Бурмакова Анастасия Владимировна**

**г.Минск, 2025**

**ЦЕЛЬ РАБОТЫ:** приобрести навыки разработки генераторов подмножеств, перестановок, сочетаний и размещений на С++; научиться применять разработанные генераторы для решения задач о рюкзаке (упрощенную, коммивояжера, об оптимальной загрузке судна и об оптимальной загрузке судна с центровкой.

**ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:**

**Задание 1.** Разобрать и разработать генератор подмножеств заданного множества.

Для выполнения данного задания нам необходимо было разобрать и разработать генератор подмножеств заданного множества.

1. Изначально необходимо было создать структуру subset, которая будет генерировать подмножества заданного множества.

#pragma once

namespace combi

{

struct subset

{

short n;

short sn;

short\* sset;

unsigned \_\_int64 mask;

subset(short n = 1);

short getfirst();

short getnext();

short ntx(short i);

unsigned \_\_int64 count();

void reset();

};

}

1. Далее определяли поля и методы структуры:

· Поля:

short n: количество элементов исходного множества.

short sn: количество элементов текущего подмножества.

short\* sset: массив индексов текущего подмножества.

unsigned \_\_int64 mask: битовая маска для генерации подмножеств.

· Методы:

Конструктор subset(short n = 1): инициализация структуры.

short getfirst(): формирование массива индексов по битовой маске.

short getnext(): увеличение маски и формирование массива индексов.

short ntx(short i): получение i-го элемента массива индексов.

unsigned \_\_int64 count(): вычисление общего количества подмножеств.

void reset(): сброс генератора, чтобы начать сначала.

1. На данном этапе создаём файл Combi.cpp, который будет содержать реализацию методов структур, используемых для генерации подмножеств. Эти структуры и методы позволят нам генерировать различные комбинации элементов множества и использовать их в алгоритмах. Реализуем структуру:

Конструктор subset::subset(short n)

Конструктор инициализирует структуру subset с количеством элементов n и выделяет память для массива индексов текущего подмножества.

subset::subset(short n)

{

this->n = n;

this->sset = new short[n];

this->reset();

}

Метод reset()

Метод reset сбрасывает состояние структуры, обнуляя количество элементов текущего подмножества и устанавливая битовую маску в 0.

void subset::reset()

{

this->sn = 0;

this->mask = 0;

}

Метод getfirst()

Метод getfirst формирует массив индексов текущего подмножества на основе битовой маски. Он проходит по битам маски и, если бит установлен, добавляет соответствующий индекс в массив.

short subset::getfirst()

{

\_\_int64 buf = this->mask;

this->sn = 0;

for (short i = 0; i < n; i++)

{

if (buf & 0x1) this->sset[this->sn++] = i;

buf >>= 1;

}

return this->sn;

}

Метод getnext()

Метод getnext увеличивает битовую маску и вызывает метод getfirst для формирования нового подмножества. Если маска превышает количество подмножеств, метод возвращает -1.

short subset::getnext()

{

int rc = -1;

this->sn = 0;

if (++this->mask < this->count()) rc = getfirst();

return rc;

}

Метод ntx(short i)

Метод ntx возвращает i-й элемент массива индексов текущего подмножества.

short subset::ntx(short i)

{

return this->sset[i];

}

Метод count()

Метод count возвращает общее количество подмножеств, которое равно 2^n.

unsigned \_\_int64 subset::count()

{

return (unsigned \_\_int64)(1 << this->n);

}

1. Реализуем конечный этап - тестирование генератора подмножеств. Вызовем функции генератора подмножества в основном файле и проверим их работу.

#include <iostream>

#include "Combi.h"

#include "Salesman.h"

#include <iomanip>

#include <ctime>

void TestSubsets()

{

std::cout << std::endl << " - Генератор множества всех подмножеств -";

char AA[][2] = { "A", "B", "C", "D" };

std::cout << std::endl << "Исходное множество: { A, B, C, D }";

combi::subset s1(sizeof(AA) / 2);

int n = s1.getfirst();

while (n >= 0)

{

std::cout << std::endl << "{ ";

for (int i = 0; i < n; i++)

std::cout << AA[s1.ntx(i)] << ((i < n - 1) ? ", " : " ");

std::cout << "}";

n = s1.getnext();

}

std::cout << std::endl << "Всего: " << s1.count() << std::endl;

}

int main()

{

setlocale(LC\_ALL, "rus");

std::cout << "ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:" << std::endl;

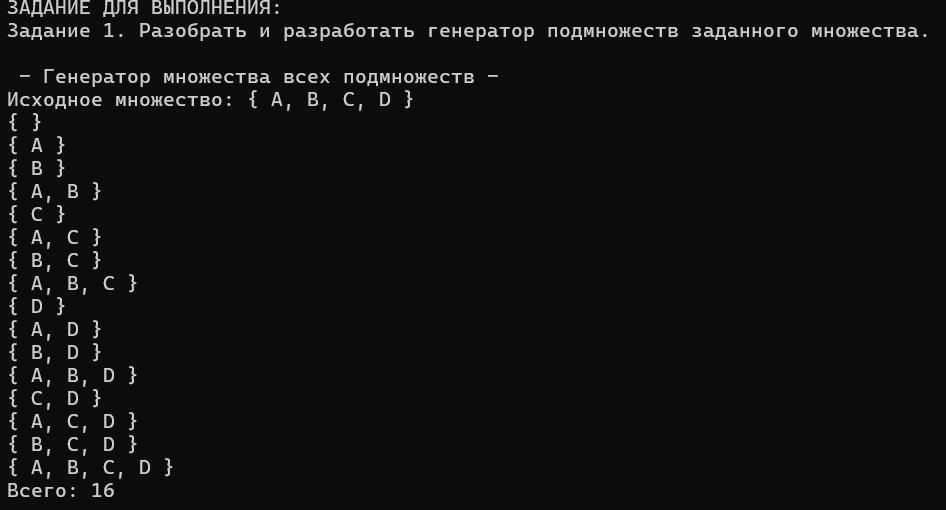
std::cout << "Задание 1. Разобрать и разработать генератор подмножеств заданного множества." << std::endl;

TestSubsets();

return 0;

}

Я создала структуру subset, которая использует битовые маски для генерации всех подмножеств. Включила методы для инициализации, сброса, получения первого и последующих подмножеств, а также вычисления общего количества подмножеств. Протестировала работу генератора с множеством {"A", "B", "C", "D"}.



**Задание 2.** Разобрать и разработать генератор сочетаний.

Для выполнения Задания 2, необходимо было разобрать и разработать генератор сочетаний.

1. Создадим структуру comb:

Определим структуру comb, которая будет генерировать сочетания из заданного множества.

#pragma once

namespace combi

{

struct comb

{

short n;

short k;

short\* sset;

comb(short n, short k);

short getfirst();

short getnext();

short ntx(short i);

unsigned \_\_int64 count();

void reset();

};

}

1. Определим поля и методы структуры:

· Поля:

short n: количество элементов исходного множества.

short k: количество элементов в сочетании.

short\* sset: массив индексов текущего сочетания.

· Методы:

Конструктор comb(short n, short k): инициализация структуры.

short getfirst(): формирование первого сочетания.

short getnext(): генерация следующего сочетания.

short ntx(short i): получение i-го элемента массива индексов.

unsigned \_\_int64 count(): вычисление общего количества сочетаний.

void reset(): сброс генератора, чтобы начать сначала.

1. Реализуем структуру Comb, для генерации сочетаний

Конструктор comb::comb(short n, short k)

Конструктор инициализирует структуру comb с количеством элементов n и длиной сочетания k.

comb::comb(short n, short k)

{

this->n = n;

this->k = k;

this->sset = new short[k];

this->reset();

}

Метод reset()

Метод reset инициализирует массив индексов начальными значениями.

void comb::reset()

{

for (short i = 0; i < k; i++)

this->sset[i] = i;

}

Метод getfirst()

Метод getfirst сбрасывает генератор и возвращает количество элементов в сочетании.

short comb::getfirst()

{

this->reset();

return k;

}

Метод getnext()

Метод getnext генерирует следующее сочетание, увеличивая индексы элементов.

short comb::getnext()

{

int i;

for (i = k - 1; i >= 0; i--)

{

if (sset[i] < n - k + i)

{

sset[i]++;

for (int j = i + 1; j < k; j++)

sset[j] = sset[j - 1] + 1;

return k;

}

}

return -1;

}

Метод ntx(short i)

Метод ntx возвращает i-й элемент массива индексов текущего сочетания.

short comb::ntx(short i)

{

return this->sset[i];

}

Метод count()

Метод count возвращает общее количество сочетаний, которое вычисляется как n! / (k! \* (n-k)!).

unsigned \_\_int64 comb::count()

{

unsigned \_\_int64 result = 1;

for (short i = 1; i <= k; i++)

result = result \* (n - i + 1) / i;

return result;

}

1. В заключение, протестируем генератор сочетаний:

#include <iostream>

#include "Combi.h"

#include "Salesman.h"

#include <iomanip>

#include <ctime>

void TestCombinations()

{

std::cout << std::endl << " - Генератор сочетаний -";

char AA[][2] = { "A", "B", "C", "D" };

std::cout << std::endl << "Исходное множество: { A, B, C, D }";

combi::comb c1(sizeof(AA) / 2, 2);

int n = c1.getfirst();

while (n >= 0)

{

std::cout << std::endl << "{ ";

for (int i = 0; i < n; i++)

std::cout << AA[c1.ntx(i)] << ((i < n - 1) ? ", " : " ");

std::cout << "}";

n = c1.getnext();

}

std::cout << std::endl << "Всего: " << c1.count() << std::endl;

}

int main()

{

setlocale(LC\_ALL, "rus");

std::cout << "ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:" << std::endl;

std::cout << "Задание 1. Разобрать и разработать генератор подмножеств заданного множества." << std::endl;

TestSubsets();

std::cout << std::endl << "Задание 2. Разобрать и разработать генератор сочетаний." << std::endl;

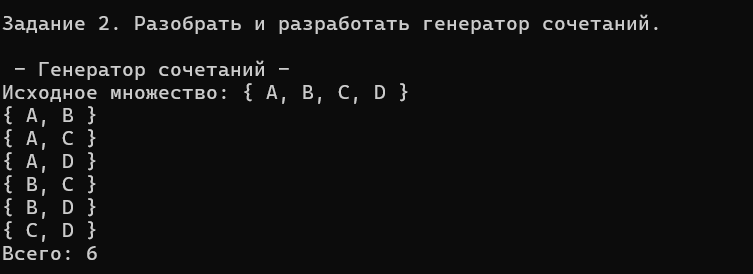
TestCombinations();

return 0;

}

Вызовем функции генератора сочетаний в основном файле и проверим их работу.

Я создала структуру comb, которая используется для генерации всех возможных сочетаний из заданного множества. Включила методы для инициализации, сброса, получения первого и последующих сочетаний, а также вычисления общего количества сочетаний. Протестировала работу генератора с множеством {"A", "B", "C", "D"} для сочетаний длиной 2.



**Задание 3.** Разобрать и разработать генератор перестановок.

Для выполнения Задания 3 необходимым было разобрать и разработать генератор перестановок.

1. Создание структуры perm:

Определите структуру perm, которая будет генерировать перестановки из заданного множества.

#pragma once

namespace combi

{

struct perm

{

short n;

short\* sset;

perm(short n);

short getfirst();

short getnext();

short ntx(short i);

unsigned \_\_int64 count();

void reset();

};

}

1. Определение полей и методов структуры:

· Поля:

short n: количество элементов исходного множества.

short\* sset: массив индексов текущей перестановки.

· Методы:

Конструктор perm(short n): инициализация структуры.

short getfirst(): формирование первой перестановки.

short getnext(): генерация следующей перестановки.

short ntx(short i): получение i-го элемента массива индексов.

unsigned \_\_int64 count(): вычисление общего количества перестановок.

void reset(): сброс генератора, чтобы начать сначала.

1. Реализация структуры perm:

Структура perm используется для генерации перестановок.

Конструктор perm::perm(short n)

Конструктор инициализирует структуру perm с количеством элементов n.

perm::perm(short n)

{

this->n = n;

this->sset = new short[n];

this->reset();

}

Метод reset()

Метод reset инициализирует массив индексов начальными значениями.

void perm::reset()

{

for (short i = 0; i < n; i++)

this->sset[i] = i;

}

Метод getfirst()

Метод getfirst сбрасывает генератор и возвращает количество элементов в перестановке.

short perm::getfirst()

{

this->reset();

return n;

}

Метод getnext()

Метод getnext генерирует следующую перестановку, используя функцию std::next\_permutation.

short perm::getnext()

{

if (std::next\_permutation(sset, sset + n))

return n;

return -1;

}

Метод ntx(short i)

Метод ntx возвращает i-й элемент массива индексов текущей перестановки.

short perm::ntx(short i)

{

return this->sset[i];

}

Метод count()

Метод count возвращает общее количество перестановок, которое вычисляется как n!.

unsigned \_\_int64 perm::count()

{

unsigned \_\_int64 result = 1;

for (short i = 2; i <= n; i++)

result \*= i;

return result;

}

1. Проведём тестирование генератора перестановок, вызовем функции генератора в основном файле и проверим их работу.

#include <iostream>

#include "Combi.h"

#include "Salesman.h"

#include <iomanip>

#include <ctime>

void TestPermutations()

{

std::cout << std::endl << " - Генератор перестановок -";

char AA[][2] = { "A", "B", "C", "D" };

std::cout << std::endl << "Исходное множество: { A, B, C, D }";

combi::perm p1(sizeof(AA) / 2);

int n = p1.getfirst();

while (n >= 0)

{

std::cout << std::endl << "{ ";

for (int i = 0; i < n; i++)

std::cout << AA[p1.ntx(i)] << ((i < n - 1) ? ", " : " ");

std::cout << "}";

n = p1.getnext();

}

std::cout << std::endl << "Всего: " << p1.count() << std::endl;

}

int main()

{

setlocale(LC\_ALL, "rus");

std::cout << "ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:" << std::endl;

std::cout << "Задание 1. Разобрать и разработать генератор подмножеств заданного множества." << std::endl;

TestSubsets();

std::cout << std::endl << "Задание 2. Разобрать и разработать генератор сочетаний." << std::endl;

TestCombinations();

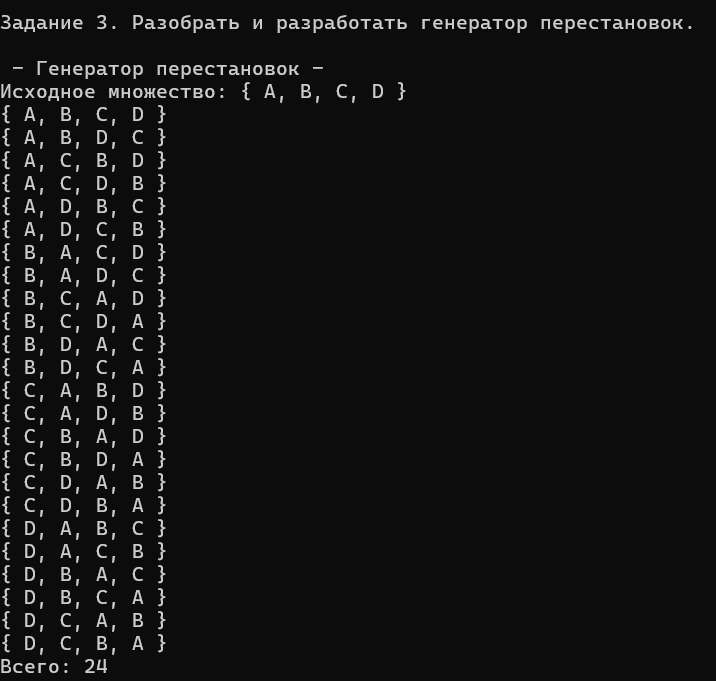
std::cout << std::endl << "Задание 3. Разобрать и разработать генератор перестановок." << std::endl;

TestPermutations();

return 0;

}

Я создала структуру perm, которая используется для генерации всех возможных перестановок заданного множества. Включила методы для инициализации, сброса, получения первой и последующих перестановок, а также вычисления общего количества перестановок. Протестировала работу генератора с множеством {"A", "B", "C", "D"}.



**Задание 4.** Разобрать и разработать генератор размещений.

Для выполнения Задания 4 нам было необходимо разобрать и разработать генератор перестановок.

1. Создание структуры arr:

Определили структуру arr, которая будет генерировать размещения из заданного множества.

1. Определили поля и методы структуры:

· Поля:

short n: количество элементов исходного множества.

short k: количество элементов в размещении.

short\* sset: массив индексов текущего размещения.

· Методы:

Конструктор arr(short n, short k): инициализация структуры.

short getfirst(): формирование первого размещения.

short getnext(): генерация следующего размещения.

short ntx(short i): получение i-го элемента массива индексов.

unsigned \_\_int64 count(): вычисление общего количества размещений.

void reset(): сброс генератора, чтобы начать сначала.

1. Реализуем структуру arr:

Структура arr используется для генерации размещений.

Конструктор arr::arr(short n, short k)

Конструктор инициализирует структуру arr с количеством элементов n и длиной размещения k.

arr::arr(short n, short k)

{

this->n = n;

this->k = k;

this->sset = new short[k];

this->reset();

}

Метод reset()

Метод reset инициализирует массив индексов начальными значениями.

void arr::reset()

{

for (short i = 0; i < k; i++)

this->sset[i] = i;

}

Метод getfirst()

Метод getfirst сбрасывает генератор и возвращает количество элементов в размещении.

short arr::getfirst()

{

this->reset();

return k;

}

Метод getnext() Метод getnext генерирует следующее размещение, увеличивая индексы элементов.

short arr::getnext()

{ for (short i = k - 1; i >= 0; i--)

{

if (sset[i] < n - 1)

{

sset[i]++;

for (short j = i + 1; j < k; j++)

sset[j] = 0;

return k;

}

}

return -1;

}

Метод ntx(short i)

Метод ntx возвращает i-й элемент массива индексов текущего размещения.

short arr::ntx(short i)

{

return this->sset[i];

}

Метод count()

Метод count возвращает общее количество размещений, которое вычисляется как n^k.

unsigned \_\_int64 arr::count()

{

unsigned \_\_int64 result = 1;

for (short i = 0; i < k; i++)

result \*= n;

return result;

}

1. Протестируем генератор размещений, вызвав функции генератора размещений в основном файле и проверим их работу.

#include <iostream>

#include "Combi.h"

#include "Salesman.h"

#include <iomanip>

#include <ctime>

void TestArrangements()

{

std::cout << std::endl << " - Генератор размещений -";

char AA[][2] = { "A", "B", "C", "D" };

std::cout << std::endl << "Исходное множество: { A, B, C, D }";

combi::arr a1(sizeof(AA) / 2, 2);

int n = a1.getfirst();

while (n >= 0)

{

std::cout << std::endl << "{ ";

for (int i = 0; i < n; i++)

std::cout << AA[a1.ntx(i)] << ((i < n - 1) ? ", " : " ");

std::cout << "}";

n = a1.getnext();

}

std::cout << std::endl << "Всего: " << a1.count() << std::endl;

}

int main()

{

setlocale(LC\_ALL, "rus");

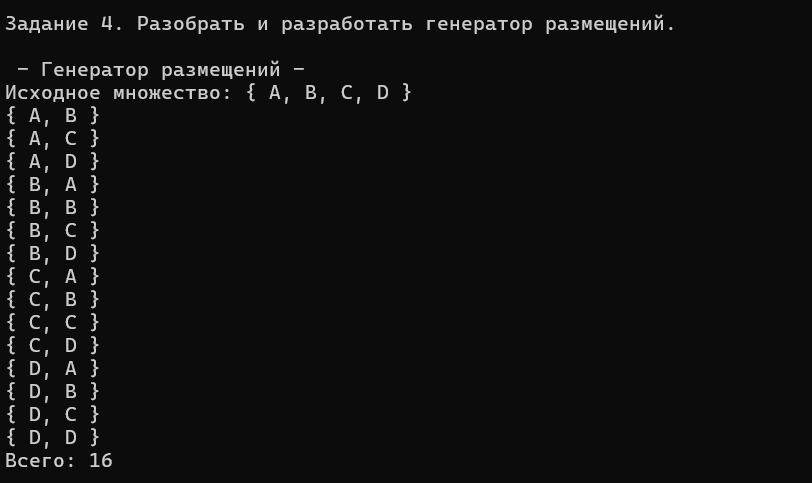
std::cout << std::endl << "Задание 4. Разобрать и разработать генератор размещений." << std::endl;

TestArrangements();

return 0;

}

Я создала структуру arr, которая используется для генерации всех возможных размещений из заданного множества. Включила методы для инициализации, сброса, получения первого и последующих размещений, а также вычисления общего количества размещений. Протестировала работу генератора с множеством {"A", "B", "C", "D"} для размещений длиной 2.



**Задание 5.**  Решить в соответствии с вариантом задачу и результат занести в отчет (Вариант распределяется по списку):

1, 5, 9, 13) коммивояжера (расстояния сгенерировать случайным образом: 10 городов, расстояния 10 – 300 км, 3 расстояния между городами задать бесконечными);

Для решения задачи коммивояжера, необходимо обладать навыками для реализации генератора перестановок. В этом задании мы решали задачу коммивояжера, используя метод перебора вариантов.

1. Создали структуру permutation:Определили структуру permutation, которая генерирует перестановки городов для решения задачи коммивояжера.

#pragma once

namespace combi

{

struct permutation

{

short n;

short\* sset;

permutation(short n);

short getfirst();

short getnext();

short ntx(short i);

unsigned \_\_int64 count();

void reset();

};

}

1. Реализовали функцию salesman:

Определили функцию salesman, которая находит оптимальный маршрут коммивояжера, используя перестановки и вычисляя расстояния между городами.

int salesman(int n, const int\* d, int\* r){

int\* s = source(n), \* b = firstpath(n), rc = INF, dist = 0;

combi::permutation p(n - 1);

int k = p.getfirst();

while (k >= 0)

{

indx(n, b, s, p.sset);

if ((dist = distance(n, b, d)) < rc)

{

rc = dist; copypath(n, r, b);

}

k = p.getnext();

};

return rc;}

1. Сгенерировали матрицы расстояний:

Создали функцию generateDistances, которая случайным образом генерирует расстояния между 10 городами в диапазоне от 10 до 300 км, и задает три расстояния как бесконечные.

void generateDistances(int n, int\* d){

srand(time(0));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j)

d[i \* n + j] = 0;

else

d[i \* n + j] = (rand() % 291) + 10;

}

}

// Задаем три расстояния как бесконечные

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;}

}

1. Протестировали задачи коммивояжера:

Вызвали функции генерации матрицы расстояний и функции поиска оптимального маршрута в основном файле и проверили их работу.

#include <iostream>

#include "Combi.h"

#include "Salesman.h"

#include <iomanip>

#include <ctime>

#define N 10

#define INF 0x7fffffff

void generateDistances(int n, int\* d)

{

srand(time(0));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j)

d[i \* n + j] = 0;

else

d[i \* n + j] = (rand() % 291) + 10;

}

}

// Задаем три расстояния как бесконечные

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

}

void printDistances(int n, int\* d)

{

std::cout << "-- матрица расстояний --" << std::endl;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (d[i \* n + j] == INF)

std::cout << std::setw(4) << "INF";

else

std::cout << std::setw(4) << d[i \* n + j];

}

std::cout << std::endl;

}

}

int main()

{

setlocale(LC\_ALL, "rus");

std::cout << "ЗАДАНИЕ ДЛЯ ВЫПОЛНЕНИЯ:" << std::endl;

std::cout << "Задание 5. Решить в соответствии с вариантом задачу и результат занести в отчет:" << std::endl;

std::cout << "1, 5, 9, 13) коммивояжера (расстояния сгенерировать случайным образом: 10 городов, расстояния 10 – 300 км, 3 расстояния между городами задать бесконечными);" << std::endl;

int d[N \* N];

generateDistances(N, d);

std::cout << "-- Задача коммивояжера -- " << std::endl;

std::cout << "-- количество городов: " << N << std::endl;

printDistances(N, d);

int r[N];

int s = salesman(N, d, r);

std::cout << "-- оптимальный маршрут: ";

for (int i = 0; i < N; i++) std::cout << r[i] << "-->";

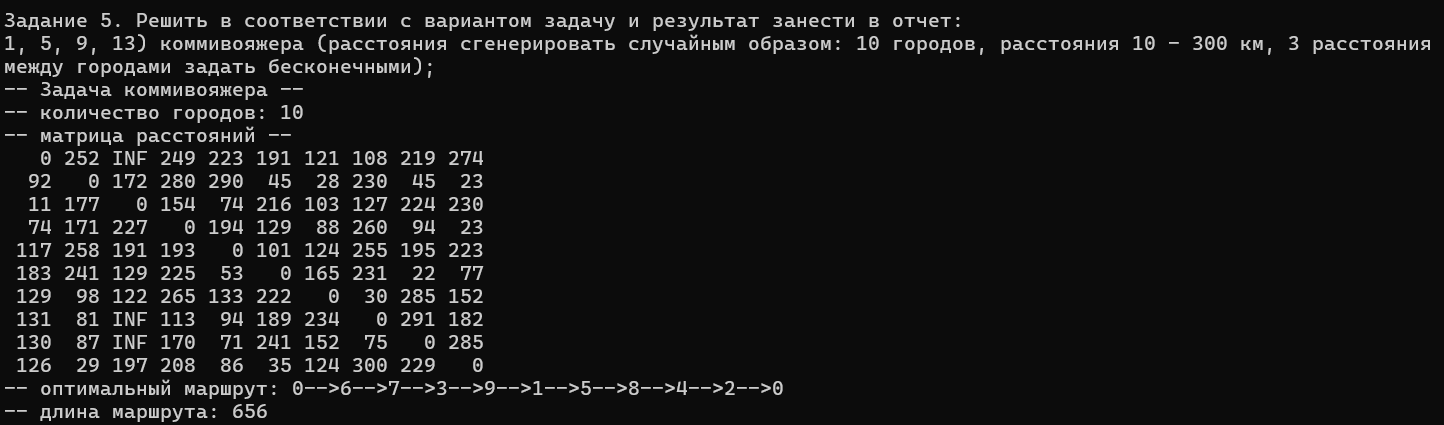
std::cout << 0 << std::endl;

std::cout << "-- длина маршрута: " << s << std::endl;

return 0;

}

Для выполнения задания 5 необходимо было решить задачу коммивояжера, используя перестановки и метод перебора вариантов. Я создала структуру permutation, которая генерирует все возможные перестановки городов. Также реализовала функцию salesman, которая находит оптимальный маршрут коммивояжера, вычисляя расстояния между городами и выбирая минимальный путь. Протестировала работу алгоритма с 10 городами, случайно сгенерированными расстояниями (в диапазоне от 10 до 300 км) и тремя бесконечными расстояниями. Программа находит и выводит оптимальный маршрут и его длину.



**Задание 6.** Исследовать зависимость времени вычисления необходимое для решения задачи (в соответствии с вариантом) от размерности задачи и результат в виде графика с небольшим пояснением занести в отчет:

1, 5, 9, 13) коммивояжера (6 – 12 городов);

В этом задании нам нужно было исследовать, как время выполнения задачи коммивояжера зависит от количества городов. Мы выполнили тестирование задачи коммивояжера для различных количеств городов в диапазоне от 6 до 12, замерили время выполнения при каждом количестве городов и построили график зависимости времени вычисления от количества городов. Ниже приведены шаги и код, который был использован для выполнения этого задания.

1. Изначально мы дополнили код замером времени выполнения задачи:

Добавили код для замера времени выполнения задачи коммивояжера для различных количеств городов. Вывод результатов замеров времени поместили на экран.

#pragma once

namespace combi

{

struct permutation

{

short n;

short\* sset;

permutation(short n);

short getfirst();

short getnext();

short ntx(short i);

unsigned \_\_int64 count();

void reset();

};

}

#include "stdafx.h"

#include "Combi.h"

#include <algorithm>

namespace combi

{

permutation::permutation(short n)

{

this->n = n;

this->sset = new short[n];

this->reset();

}

void permutation::reset()

{

for (short i = 0; i < n; i++)

this->sset[i] = i;

}

short permutation::getfirst()

{

this->reset();

return n;

}

short permutation::getnext()

{

if (std::next\_permutation(sset, sset + n))

return n;

return -1;

}

short permutation::ntx(short i)

{

return this->sset[i];

}

unsigned \_\_int64 permutation::count()

{

unsigned \_\_int64 result = 1;

for (short i = 2; i <= n; i++)

result \*= i;

return result;

}

}

1. Сгенерировали матрицу расстояний:

Использовали функцию generateDistances для генерации случайных расстояний между городами.

void generateDistances(int n, int\* d)

{

srand(time(0));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j)

d[i \* n + j] = 0;

else

d[i \* n + j] = (rand() % 291) + 10;

}

}

// Задаем три расстояния как бесконечные

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

}

void printDistances(int n, int\* d)

{

std::cout << "-- матрица расстояний --" << std::endl;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (d[i \* n + j] == INF)

std::cout << std::setw(4) << "INF";

else

std::cout << std::setw(4) << d[i \* n + j];

}

std::cout << std::endl;

}

}

1. Измерили время выполнения:

Использовали функции clock, startTime, endTime для замера времени выполнения задачи.

void generateDistances(int n, int\* d)

{

srand(time(0));

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (i == j)

d[i \* n + j] = 0;

else

d[i \* n + j] = (rand() % 291) + 10;

}

}

// Задаем три расстояния как бесконечные

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

d[rand() % n \* n + rand() % n] = INF;

}

void printDistances(int n, int\* d)

{

std::cout << "-- матрица расстояний --" << std::endl;

for (int i = 0; i < n; i++)

{

for (int j = 0; j < n; j++)

{

if (d[i \* n + j] == INF)

std::cout << std::setw(4) << "INF";

else

std::cout << std::setw(4) << d[i \* n + j];

}

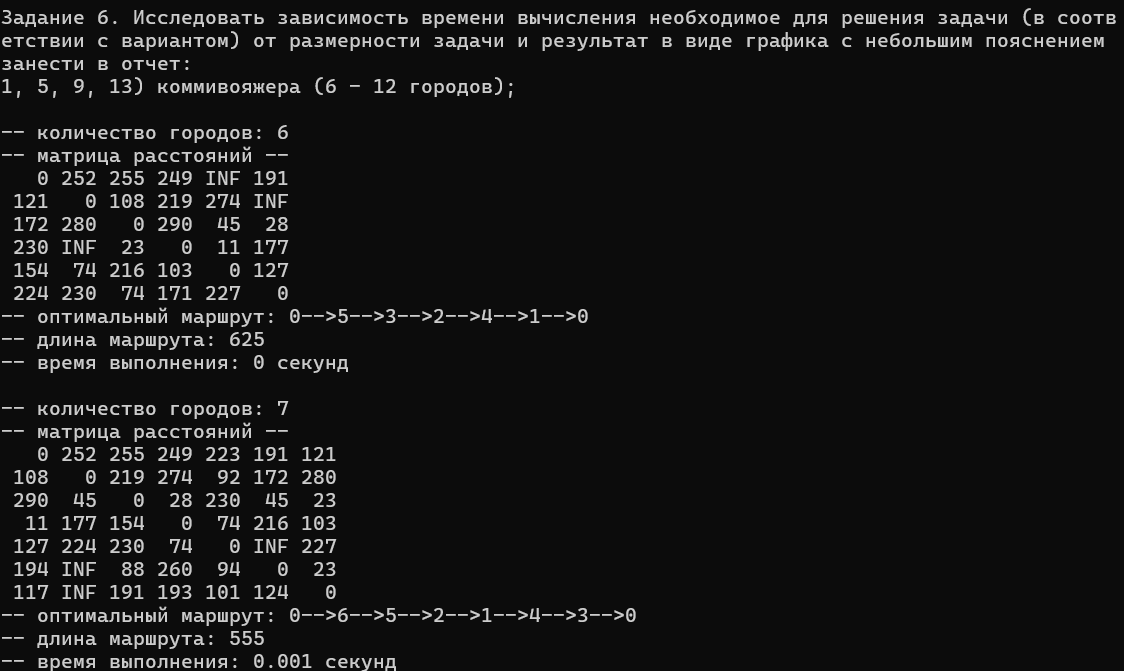
std::cout << std::endl;

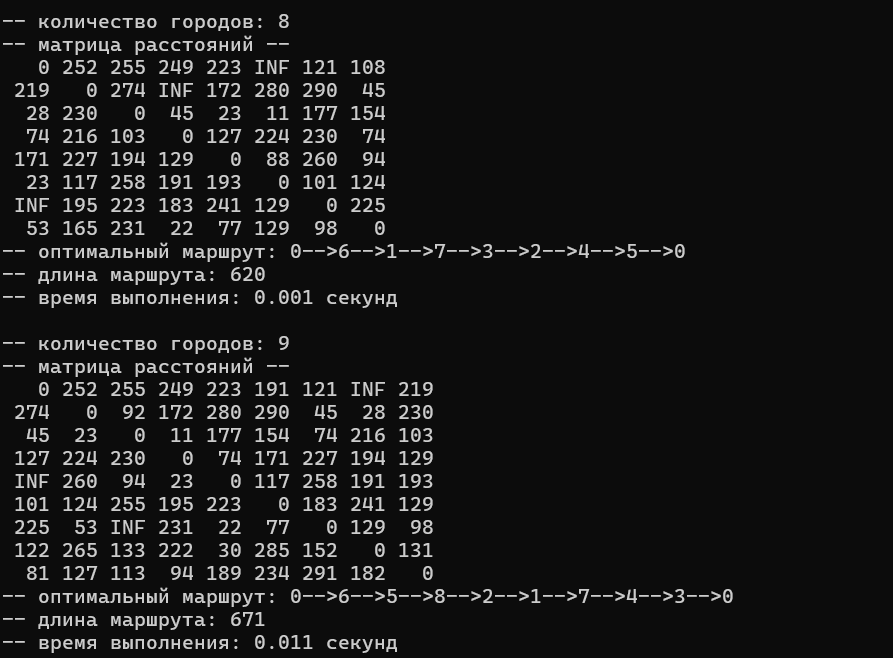
}

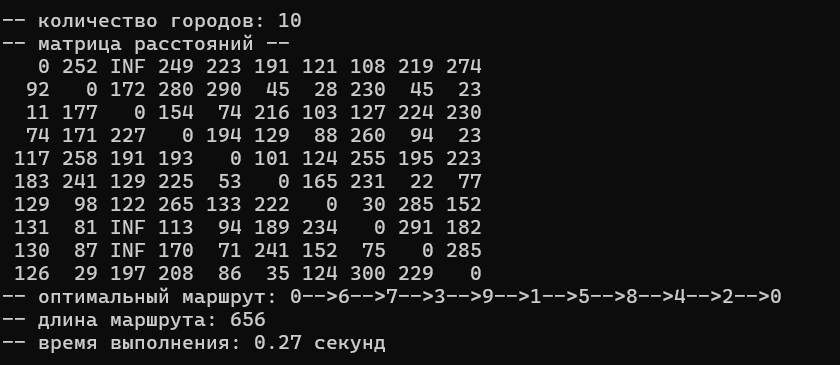
}

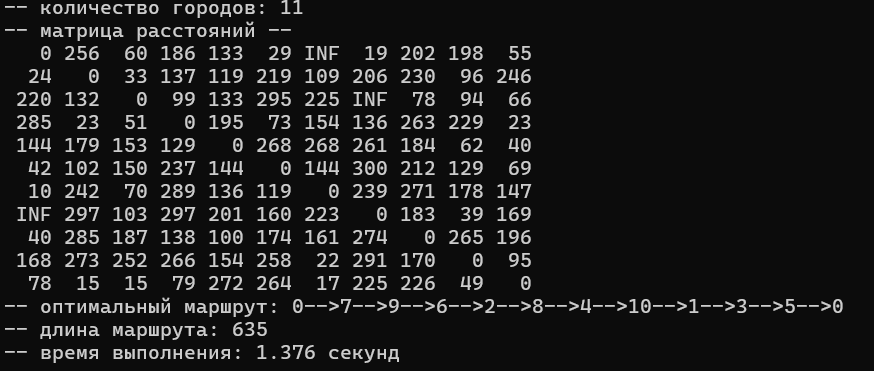
1. Запустили программу и замер времени для каждого количества городов:

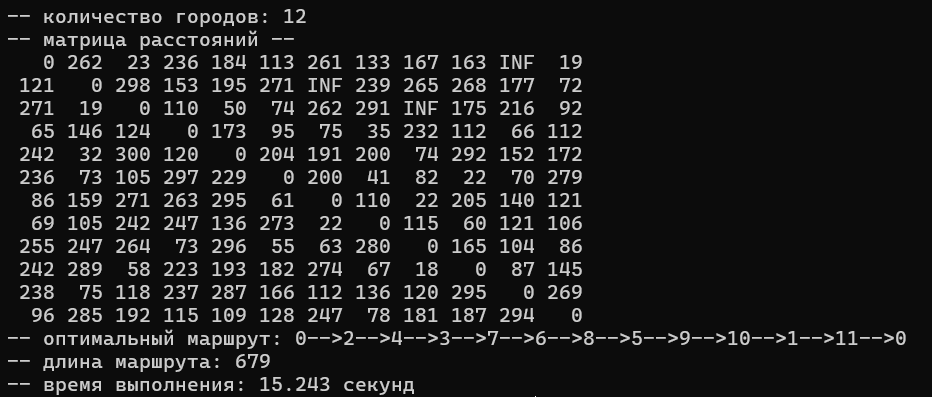
Запускали программу и замеряли время выполнения для каждого количества городов от 6 до 12.







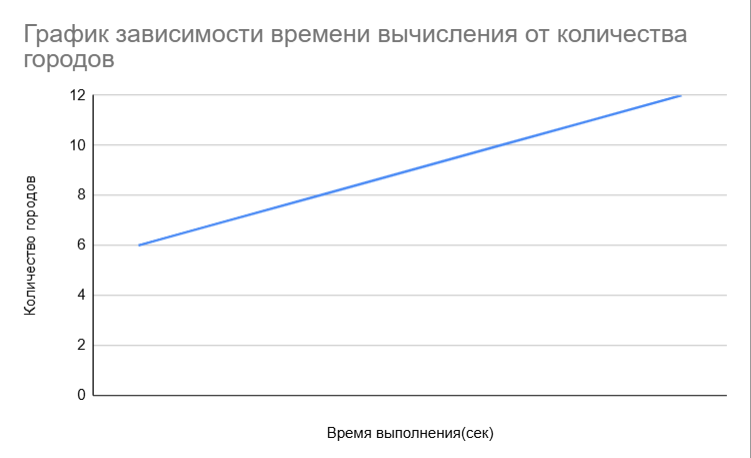




1. Построили график зависимости времени вычисления от количества городов в exel:

Использовали полученные данные для построения графика.





Зависимость наблюдаемая на графике линейная, то бишь по мере увеличение городов увеличивается время выполнение задачи.