МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МО ЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и Анализ Алгоритмов»

Тема: Поиск с возвратом

Студент гр. 3343 _____ Малиновский А.А.

Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург

2025

Цель работы.

Изучение алгоритма поиска с возвратом, реализация с его помощью программы, решающей задачу размещения квадратов на столе.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 1 до $N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу – квадрат размера N. Он может получить ее, собрав из уже имеющихся обрезков (квадратов). Например, столешница размера 7×7 может быть построена из 9 обрезков

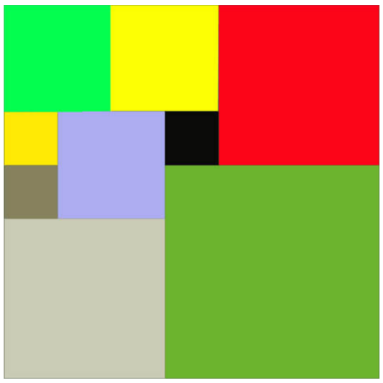


Рисунок 1 – пример размещения квадратов

Внутри столешницы не должно быть пустот, обрезки не должны выходить за пределы столешницы и не должны перекрываться. Кроме того, Вова хочет использовать минимально возможное число обрезков.

Входные данные

Размер столешницы - одно целое число N $(2 \le N \le 20)$.

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x,y и w, задающие координаты левого верхнего угла $(1 \le x,y \le N)$ и длину стороны соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Вар. 3и. Итеративный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Основные теоретические положения.

Поиск с возвратом, backtracking — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют.

Реализация.

Описание алгоритма

Для решения задачи был использован итеративный поиск с возвратом. Поиск осуществляется перебором вариантов расстановки очередного квадрата. Данный алгоритм основывается на поиске в ширину. Для каждого частичного решения перебираются все возможные расширения и добавляются в очередь для дальнейших расширений, при этом расширяемое на данном шаге решение удаляется из очереди. Таким образом, первое полученное полное решение является оптимальным.

Т.к. алгоритм основан на BFS, его сложность по времени можно оценить как O(|V|+|E|), но фактическая сложность стремится к экспоненте, т.к. кол-во вершин и ребер графа сильно возрастает с увеличением длины стороны квадрата. Таким образом реальная сложность алгоритма – $O((n^2)^m)$ – экспоненциальная, где n – длина стороны стола, m – кол-во квадратов. Это подтверждается фактическими измерениями времени работы алгоритма. Затраты по памяти можно оценить как $O((n-1)!*n^2)$, т.к. для любого частичного решения может существовать в худшем случае n расширений, но расширяемое частичное решение при этом удаляется.

Частичные решения хранятся в виде объектов класса Desk, полями которого являются:

матрица map с нулями в свободных клетках и иными значениями в местах, соответствующих квадратам

вектор squareList, содержащий объекты Square

deskSize – длина стороны стола

squareCounter – кол-во размещенных квадратов

Описание методов и структур данных

Для хранения частичных решений использовался stl-контейнер queue.

В реализованном классе Desk определены следующие методы:

Desk(int n) – конструктор класса. В качестве аргумента принимает число n, оно определяет длину стороны конструируемого стола.

void addSquaresForPrimeSizes(): Добавляет три квадрата для столов с простым размером.

void makeAnswerForEvenSize(): Добавляет четыре квадрата для столов с четным размером.

void addSquare(int size, int x, int y, int color) – метод размещения квадрата, в качестве аргументов принимает размер квадрата, координаты верхнего левого угла, число, обозначающее цвет размещаемого квадрата. Данный метод создает объект Square, добавляет его в squareList, увеличивает счетчик квадратов и заполняет матрицу

bool isFull() – данный метод проверяет, есть ли в матрице map нули. Если есть – возвращает false, значит, что стол заполнен не полностью, иначе возвращает true.

bool canAdd(int x, int y, int size) – метод, проверяющий, можно ли расположить на столе квадрат размера size, начиная из точки с координатами (x, y).

std::pair<int, int> emptyCell() – метод, возвращающий пару значений, являющихся координатами самой верхней левой свободной клетки.

Набор getter’ов дающих доступ к приватным полям данного класса.

friend std::ostream& operator<<(std::ostream& os, const Desk& desk) – перегрузка оператора вывода в поток.

Реализация структуры Square:

Поля int size, int x, int y – размер квадрата, координаты левого верхнего угла

Square(int size, int x, int y) – конструктор объекта

friend std::ostream& operator<<(std::ostream& os, const Square& square) – перегрузка оператора вывода в поток.

Функция Desk backtracking(Desk desk) – функция, реализующая итеративный поиск с возвратом. В качестве аргумента принимает объект класса Desk. После того, как решение найдено, возвращает объект этого же класса.

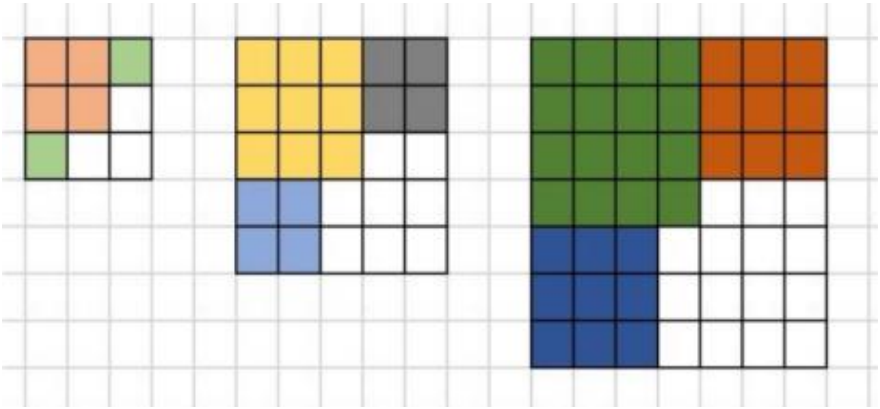
Функция Desk scaleSolution(const Desk& smallDesk, int scaleFactor): Масштабирует решение для стола меньшего размера на стол большего размера, умножая размеры и координаты квадратов на коэффициент масштабирования.

Функция bool isPrime(int n): Проверяет, является ли число n простым, исключая случай.

Функция std::pair<int, int> getFactors(int n): Возвращает пару множителей числа n, если оно составное, иначе возвращает {1, n}.

Примененные оптимизации

1) В случае, если размер стола N - простое число, заранее можно расставить один квадрат размером (N+1)//2 и два смежных ему квадрата N//2. Пример квадратов со сторонами 3,5,7:



2) Новый квадрат всегда устанавливается в максимально верхнюю левую клетку, таким образом сокращается кол-во расстановок, т.к. отбрасываются одинаковые расстановки, но с разным порядком размещения квадратов.

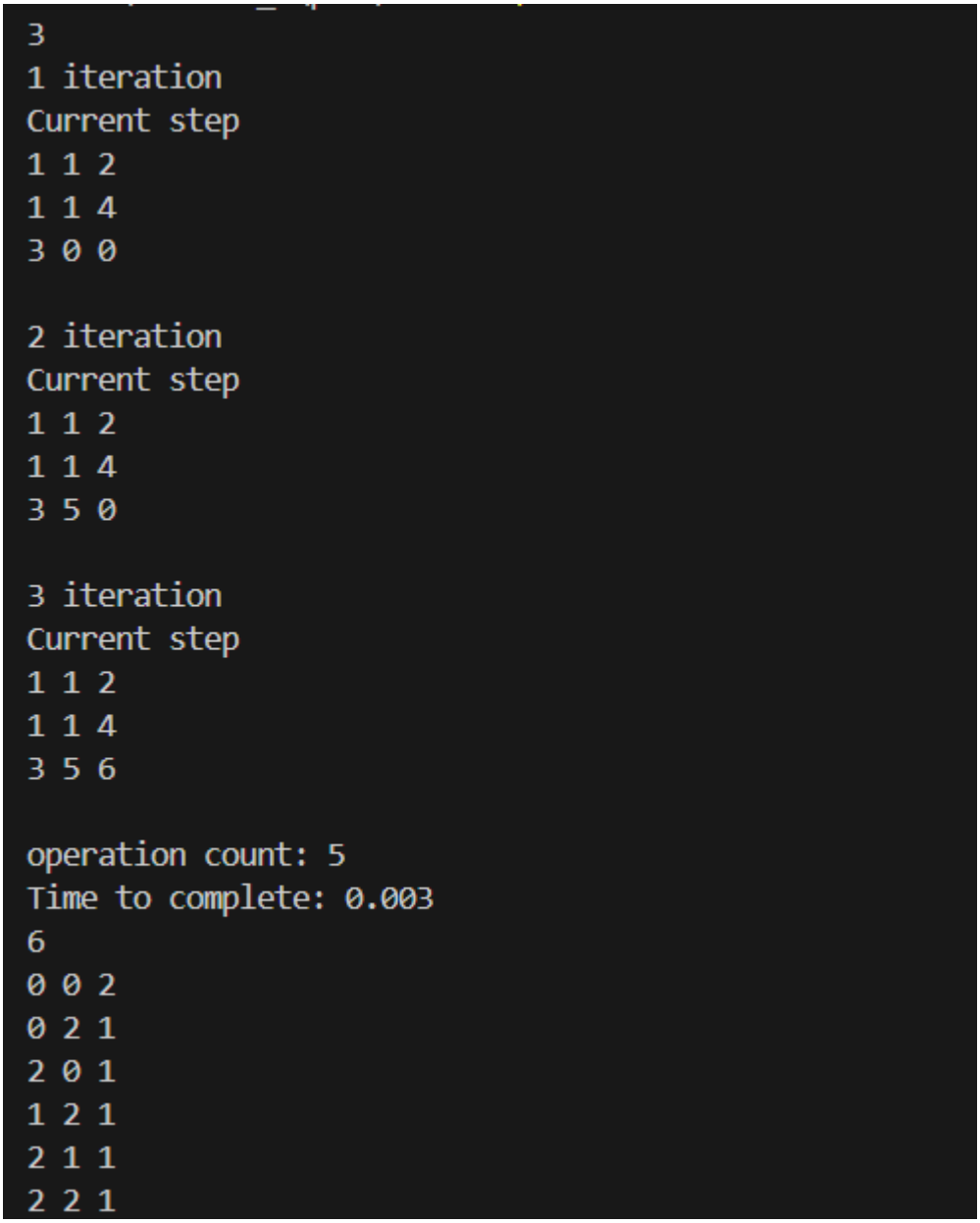
3) Так как алгоритм при нахождении ответа в первую очередь расставляет самые большие квадраты, то расстановка, которую мы примем за итоговую будет найдена первой. Это позволяет нам не просчитывать все возможные варианты, а прекратить выполнение алгоритма при нахождении первой расстановки, полностью заполняющей поле.

4) Если сторона квадрата чётное число, то минимальное разбиение всегда будет равно 4.

5) Если N составное: в таком случае число квадратов в оптимальном разбиении не превосходит аналогичного минимального для какого-либо из множителей числа. Это значит, что нужно найти минимальный делитель числа, применить к нему поиск наименьшего разбиения, затем умножить размер и координаты на оставшиеся делители для получения разбиения N.

Код программы смотреть в приложении А.

Пример работы программы:



Тестирование.

Реализованы тесты для проверки поведения программы в случае ввода неверной стороны стола (n >20 и n < 2).

Также проверена корректность работы алгоритма бэктрекинга для всех возможных размеров из промежутка 2...10.

Ввод	**Вывод**	**Ожидаемый результат**

2	4 0 0 1 0 1 1 1 0 1 1 1 1	Результат верный Оптимизация 4)
3	6 0 0 2 0 2 1 1 2 1 2 0 1 2 1 1 2 2 1	Результат верный Оптимизация 1)
4	4 0 0 2 0 2 2 2 0 2 2 2 2	Результат верный Оптимизация 4)
5	8 0 0 3 0 3 2 3 0 2 2 3 2 3 2 1 4 2 1 4 3 1 4 4 1	Результат верный Оптимизация 1)

6	4 0 0 3 0 3 3 3 0 3 3 3 3	Результат верный Оптимизация 4)
7	9 0 0 4 0 4 3 4 0 3 3 4 2 3 6 1 4 3 1 4 6 1 5 3 2 5 5 2	Результат верный Оптимизация 1)
8	4 0 0 4 0 4 4 4 0 4 4 4 4	Результат верный Оптимизация 4)
9	6 0 0 6 0 6 3 3 6 3 6 0 3 6 3 3 6 6 3	Результат верный Оптимизация 5)
10	4 0 0 5 0 5 5 5 0 5 5 5 5	Результат верный Оптимизация 4)
1	Size must be in range [2;20],	Результат верный

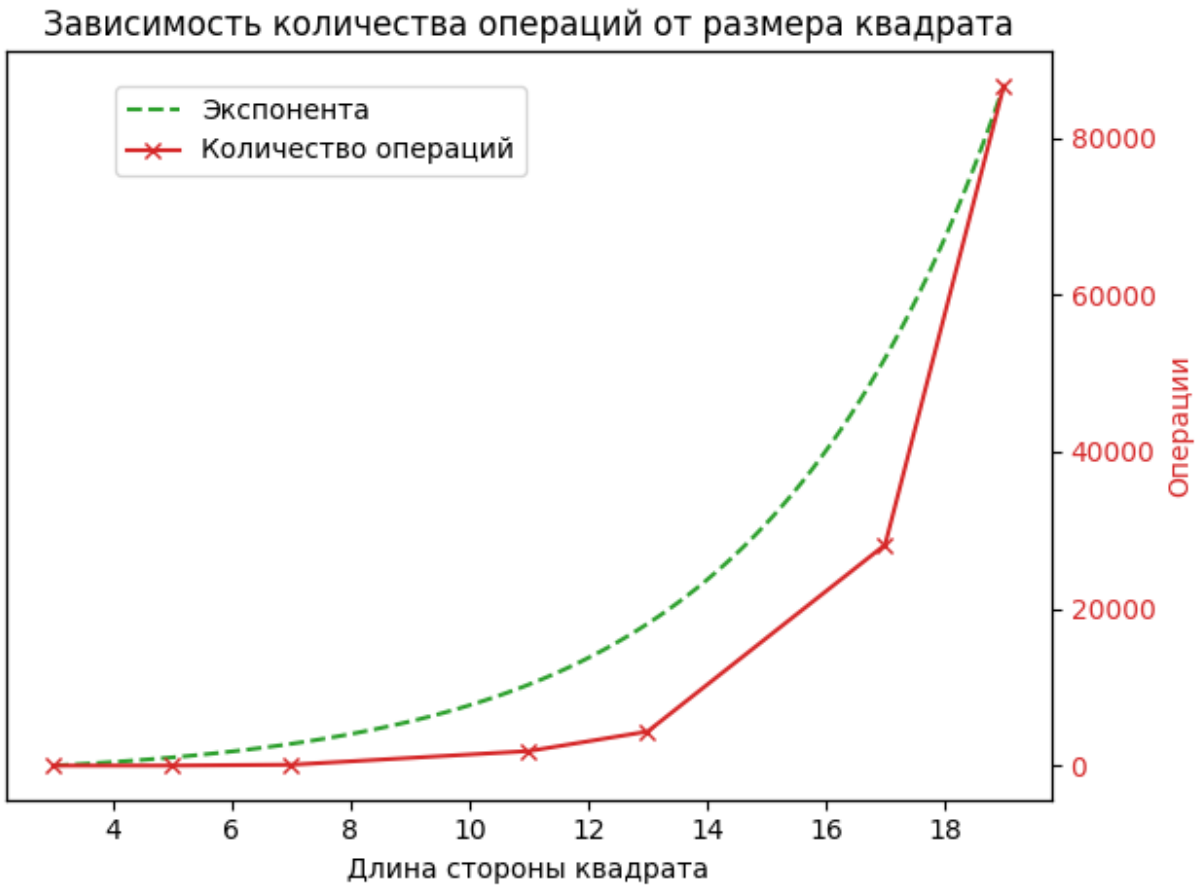
	try again	
21	Size must be in range [2;20], try again	Результат верный

Исследование.

Элементарной операцией является одна попытка постановки квадрата. Не учитываем чётные размеры, т.к для них ответ получается за O(1). Также не рассматриваем составные числа, т.к их разбиение сводится к разбиению их наименьшего делителя. Остаётся рассмотреть простые числа.

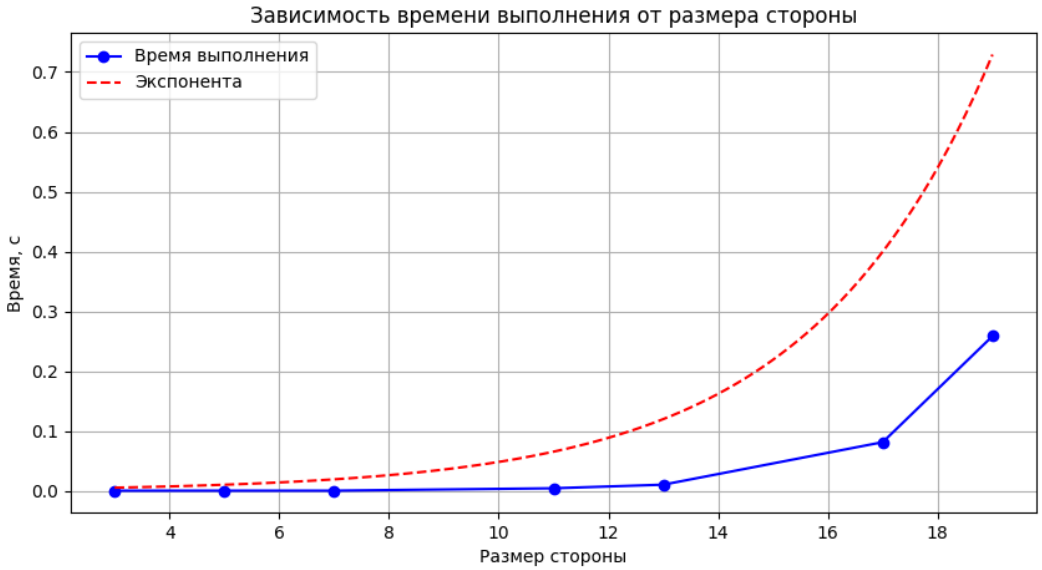
Размер	Время, с	Операции	Результат (кол-во квадратов)
3	0.001	27	6
5	0.001	35	8
7	0.001	136	9
11	0.005	1877	11
13	0.011	4366	11
17	0.082	28156	12
19	0.259	86631	13

Построим график по полученным данным для демонстрации зависимости количества операций от размера квадрата:



\

Построим аналогичный график, показывающий зависимость времени выполнения от размера стороны квадрата:



Пунктирная линия на графиках – экспоненциальная линия тренда, таким образом можем повторно отметить, что реальное время выполнения алгоритма близко к экспоненциально растущему.

Выводы.

В результате работы была написана программа, решающая поставленную задачу с использованием итеративного бэктрекинга. Программа была протестирована, результаты тестов совпали с ожидаемыми. По результатам исследования можем заключить, что зависимости числа операций от размера поля возрастает экспоненциально.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: backtracking.hpp

```
#ifndef BACKTRACKING_HPP
#define BACKTRACKING_HPP

#include <queue>
#include <iostream>
#include "desk.hpp"

bool isPrime(int n);
std::pair<int, int> getFactors(int n);
Desk backtracking(Desk desk);
Desk scaleSolution(const Desk& smallDesk, int scaleFactor);

#endif
```

Название файла: backtracking.cpp

```
  #include "backtracking.hpp"

  Desk backtracking(Desk desk) {
      if(isPrime(desk.getDeskSize())){
          desk.addSquaresForPrimeSizes();
      }
      std::queue<Desk> queue; // кладем исходный стол в очередь
      queue.push(desk);
      int currentIteration = 1, operationCounter = 0; // инициализация
    счетчиков для количества операций и итераций

      while (!queue.front().isFull()) {
          std::cout << currentIteration << " " << "iteration\n";
          Desk s = queue.front(); // создаем копию первого стола из очереди
    для взаимодействия
          std::pair<int, int> emptyCell = s.emptyCell(); // находим свободную
    клетку на столе
          for (int i = desk.getDeskSize() - 1; i > 0; i--) {
           // пытаемся поставить квадрат на место пустой клетки, начинаем
    попытки от наибольшего возможного варианта
              Desk cur = s;
              if (s.canAdd(emptyCell.first, emptyCell.second, i)) { //
    проверка возможности поставить квадрат в стол
                  cur.addSquare(i, emptyCell.first, emptyCell.second,
    cur.getSquareCount());
                  std::cout << "Current partial solution\n" << cur <<
    std::endl;
                  if (cur.isFull()) { // в случае, если мы полностью
    заполнили стол, мы возвращаем полученный итог
                      std::cout << "operation count: " << operationCounter <<
    std::endl;
                      return cur;
                  }
                  queue.push(cur); // если стол был заполнен не до конца,
    полученный этап решения кладем в конец очереди и идем дальше
              }
              operationCounter++;
          }
          queue.pop();// удаляем итерацию стола, над которой мы работали,
```

```
            // т.к. мы либо добавили в очередь все возможные дальнейшие
       расстановки на данном этапе, либо полностью его заполнили
            currentIteration++;
        }
        std::cout << "operation count: " << operationCounter << std::endl;
        return queue.front();
    }

    Desk scaleSolution(const Desk& smallDesk, int scaleFactor) {
        int newDeskSize = smallDesk.getDeskSize() * scaleFactor;
        Desk newDesk(newDeskSize);

        for (const auto& square : smallDesk.getSquareList()) {
            int newSize = square.size * scaleFactor;
            int newX = square.x * scaleFactor;
            int newY = square.y * scaleFactor;
            newDesk.addSquare(newSize, newX, newY, newDesk.getSquareCount());
        }

        return newDesk;
    }

    bool isPrime(int n) {
        if (n == 2) return false;
        for (int i = 2; i <= sqrt(n); i++) {
            if (n % i == 0) return false;
        }
        return true;
    }

    std::pair<int, int> getFactors(int n) {
        for (int i = 2; i <= sqrt(n); i++) {
            if (n % i == 0) {
                return {i, n / i};
            }
        }
        return {1, n}; // Если число простое, возвращаем {1, n}
    }
```

Название файла: desk.hpp

```
#ifndef DESK_HPP
#define DESK_HPP

#include <vector>
#include <queue>
#include <iostream>
#include <cmath>
#include "square.hpp"

class Desk {
private:
    std::vector<std::vector<int>> map; // двумерный массив, отображающий
  расположение квадратов
    int deskSize;    // длина стороны стола
    int squareCounter; // количество квадратов
    std::vector<Square> squareList; // список характеристик квадратов,
  выводимый в конце

public:
    Desk(int n);
```

```
    void addSquaresForPrimeSizes();
    void makeAnswerForEvenSize();
    void addSquare(int size, int x, int y, int color);
    bool isFull() const;
    bool canAdd(int x, int y, int size) const;
    std::pair<int, int> emptyCell() const;
    int getDeskSize() const;
    std::vector<Square> getSquareList() const;
    int getSquareCount() const;
    std::vector<std::vector<int>> getMap() const;
    friend std::ostream& operator<<(std::ostream& os, const Desk& desk);
};

#endif // DESK_HPP
```

Название файла: desk.cpp

```
#include "desk.hpp"

Desk::Desk(int n) : deskSize(n), squareCounter(0) {
    this->map = std::vector<std::vector<int>>();
    for(auto i = 0; i < deskSize; i++){
        this->map.push_back(std::vector<int>(n));
    }
    this->squareList = std::vector<Square>();
    this->squareCounter = 0;
}

void Desk::addSquaresForPrimeSizes() {
    addSquare((deskSize + 1) / 2, 0, 0, squareCounter);
    addSquare((deskSize) / 2, 0, (deskSize + 1) / 2, squareCounter);
    addSquare((deskSize) / 2, (deskSize + 1) / 2, 0, squareCounter);
}

void Desk::makeAnswerForEvenSize() {
    addSquare(deskSize / 2, 0, 0, squareCounter);
    addSquare(deskSize / 2, deskSize / 2, 0, squareCounter);
    addSquare(deskSize / 2, 0, deskSize / 2, squareCounter);
    addSquare(deskSize / 2, deskSize / 2, deskSize / 2, squareCounter);
}

void Desk::addSquare(int size, int x, int y, int color) {
    Square square(size, x, y);
    for (int i = x; i < x + size; i++) {
        for (int j = y; j < y + size; j++) {
            map[i][j] = color + 1;
        }
    }
    squareList.push_back(square);
    squareCounter++;
}

bool Desk::isFull() const {
    for (int i = 0; i < deskSize; i++) {
        for (int j = 0; j < deskSize; j++) {
            if (map[i][j] == 0) return false;
        }
    }
    return true;
}
```

```
bool Desk::canAdd(int x, int y, int size) const {
    if (y + size > deskSize || x + size > deskSize) return false;
    for (int i = x; i < x + size; i++) {
        for (int j = y; j < y + size; j++) {
            if (map[i][j] != 0) return false;
        }
    }
    return true;
}

std::pair<int, int> Desk::emptyCell() const {
    for (int i = 0; i < deskSize; i++) {
        for (int j = 0; j < deskSize; j++) {
            if (map[i][j] == 0) return {i, j};
        }
    }
    return {-1, -1};
}

int Desk::getDeskSize() const {
    return deskSize;
}

std::vector<Square> Desk::getSquareList() const {
    return squareList;
}

int Desk::getSquareCount() const {
    return squareCounter;
}

std::vector<std::vector<int>> Desk::getMap() const {
    return map;
}

std::ostream& operator<<(std::ostream& os, const Desk& desk) {
    for (const auto& elem : desk.getMap()) {
        for (const auto& cell : elem) {
            os << cell << " ";
        }
        os << '\n';
    }
    return os;
}
```

Название файла: square.hpp

```
#ifndef SQUARE_HPP
#define SQUARE_HPP

#include <iostream>

struct Square {
    int size; // длина стороны квадрата
    int x;
    int y;

    Square(int size, int x, int y);
    friend std::ostream& operator<<(std::ostream& os, const Square& square);
};
```

```
#endif // SQUARE_HPP
```

Название файла: square.cpp

```
#include "square.hpp"

Square::Square(int size, int x, int y) : size(size), x(x), y(y) {}

std::ostream& operator<<(std::ostream& os, const Square& square) {
    os << square.x << ' ' << square.y << ' ' << square.size << std::endl;
    return os;
}
```

Название файла: main.cpp

```
#include "desk.hpp"
#include "backtracking.hpp"
#include <iostream>
#include <ctime>
#include <limits>

int main() {
    int n;
    bool inputSuccess = false;
    while (!inputSuccess) {
        if (std::cin >> n) {
            if (n < 2 || n > 20)
                std::cout << "Size must be in range [2;20], try again\n";
            else
                inputSuccess = true;
        } else {
            std::cout << "Invalid input, please enter a number\n";
            std::cin.clear();
            std::cin.ignore(std::numeric_limits<std::streamsize>::max(),
  '\n');
        }
    }

    Desk desk(n);
    auto start = clock();
    if (n % 2 == 0) {
        desk.makeAnswerForEvenSize();
    }
    else if (!isPrime(n)) {
        auto factors = getFactors(n);
        int smallerFactor = factors.first;
        Desk smallDesk(smallerFactor);
        Desk smallAnswer = backtracking(smallDesk);
        desk = scaleSolution(smallAnswer, factors.second);
    } else {
        Desk answer = backtracking(desk);
        desk = answer;
    }

    std::cout << "Time to complete: " << (double)(clock() - start) /
  CLOCKS_PER_SEC << std::endl;
    std::cout << desk.getSquareCount() << std::endl;
    for (auto elem : desk.getSquareList()) {
        std::cout << elem;
    }
```

```
        return 0;
    }
```

Название файла: makefile

```
CC = g++
RM = rm -rf

CFLAGS = -I$(INCDIR)
LIB = -lm

SRCDIR = src
INCDIR = include
OBJDIR = obj

SOURCES = $(wildcard $(SRCDIR)/*.cpp)
OBJECTS = $(patsubst $(SRCDIR)/%.cpp, $(OBJDIR)/%.o, $(SOURCES))
EXECUTABLE = main

all: $(EXECUTABLE)

$(EXECUTABLE): $(OBJECTS)
    $(CC) $(CFLAGS) $(OBJECTS) -o $@ $(LIB)

$(OBJDIR)/%.o: $(SRCDIR)/%.cpp
    @if not exist $(OBJDIR) mkdir $(OBJDIR)
    $(CC) $(CFLAGS) -c $< -o $@

clean:
    $(RM) -r $(OBJDIR) $(EXECUTABLE)
```