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 _____ Пивоев Н. М.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург
2025

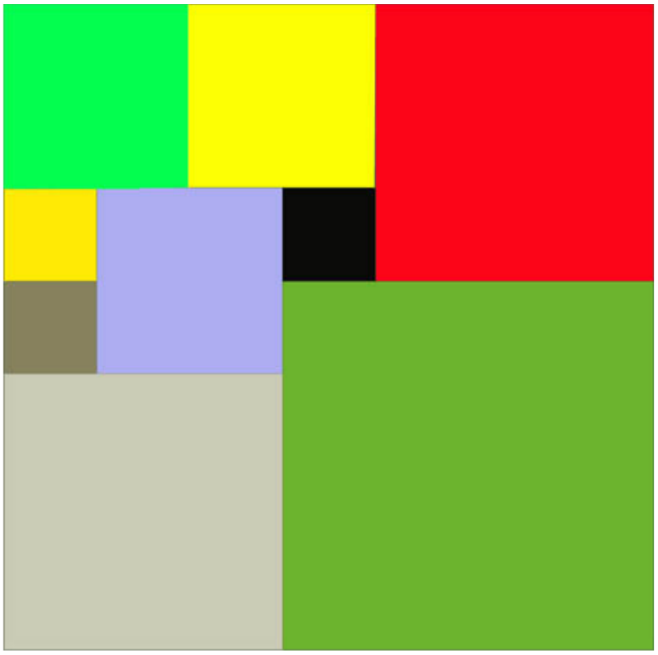
Цель работы

Изучить работу алгоритма поиска с возвратом и написать его итеративную реализацию для задачи 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 её, собрав из уже имеющихся обрезков (квадратов).

Например, столешница размера 7 x 7 может быть построена из 9 обрезков.



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 <= N <= 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 <= x, y <= N) и длину стороны соответствующего обрезка (квадрата).

Вариант 3и. Итеративный бэктрекинг. Исследование количества операций от размера квадрата.

Выполнение работы

Описание алгоритма

Для поиска оптимального размещения используется алгоритм итеративного бэктрекинга, путём перебора всех вариантов расстановки следующего квадрата. Если квадрат данного размера возможно расположить на столе, то данная расстановка добавляется в очередь. Таким образом очередь можно представить, как граф, где каждая вершина – определённая расстановка, а глубина конкретной вершины – число квадратов в данной расстановке. Поскольку задача – найти полную расстановку с наименьшим числом квадратов, то можно воспользоваться поиском в ширину для нахождения первой такой расстановки.

Оценка сложности

Временная сложность алгоритма в худшем случае экспоненциальная – $O(n^k)$, где n – длина стола, а k – число квадратов в оптимальном разбиении. Каждая расстановка порождает n других, увеличивая число квадратов в них на 1. Значит, чтобы дойти до решения потребуется k таких разделений, а так как, мы обрабатываем уровни поочерёдно, то сложность получает такую оценку.

Пространственная сложность алгоритма - $O(n^{2k})$, поскольку нам необходимо в каждой позиции хранить информацию о всём столе – матрице $n * n$.

Оптимизации

1. Для всех чётных размеров стола сразу можно определить расстановку, поделив стол на 4 части.

2. Для всех составных размеров стола, расстановку можно определить, как увеличенную расстановку его наименьшего делителя на дополняющий делитель.

3. Для всех простых размеров стола сразу же можно расположить три квадрата.

4. Все квадраты полые, а на их границах написан их размер, что позволяет пропускать обход внутри квадратов и сокращает время их обработки до $O(n)$.

5. Полностью обработанная расстановка удаляется из очереди.

Код программы содержит реализацию классов: *Square, Table*.

Square является классом, содержащим информацию о квадрате (обрезке). Он имеет следующие поля:

- *Coordinate coordinate* – координаты квадрата (x, y).
- *int length* – размер стороны квадрата.

И следующие методы:

- *Square(Coordinate coordinate, int length)* – конструктор класса, создающий квадрат.
- *Coordinate getCoordinate() const* – возвращает координаты квадрата.
- *int getLength() const* – возвращает длину стороны квадрата.

Класс *Table* отвечает за операции на столе, он хранит информацию о всех квадратах и текущей наполненности поля. Он имеет следующие поля:

- *std::vector<std::vector<int>> canvas* – матрица значений, показывающая, является ли конкретная точка частью квадрата или нет.
- *int n* – размер стороны стола.
- *std::vector<Square> squares* – все квадраты (обрезки), лежащие на столе.

И следующие методы:

- *Table(int n)* – конструктор класса.
- *Coordinate findEmpty()* – находит и возвращает координаты незаполненного места на столе, поскольку квадраты обрисованы только с внешней стороны, пропускает внутреннюю часть.
- *boll isPossibleToInsert(Coordinate coordinate, int currentSize)* – проверяет, возможно ли вставить квадрат заданного размера в данную позицию.
- *void insertSquare(Coordinate coordinate, int currentSize)* – заполняет выбранное пространство стола квадратом.

- *void setEvenPreset()* – расставляет 4 квадрата по заготовке.
- *void setPrimePreset()* – расставляет 3 квадрата по заготовке и запускает бэктрекинг для заполнения оставшегося места.
- *void setCompositePreset()* – находит два делителя размера стола, вычисляет ответ для наименьшего из них и расширяет его.
- *Table backtracking()* – выполняет поиск оптимального решения задачи размещения квадратов на столе.
- *Table scaler(int multiplier)* – расширяет ответ для большего размера, кратного данному.
- *void printIteration(int iteration)* – выводит промежуточную итерацию программы.
- *void printCanvas()* – выводит заполненный квадратами стол.
- *void printSquares()* – выводит информацию о всех квадратах.
- *std::vector<std::vector<int>> getCanvas() const* – возвращает матрицу наполненности стола.
- *int getN() const* – возвращает размер стороны стола.
- *std::vector<Square> getSquares() const* – возвращает все использованные квадраты.

Тестирование

Программа была протестирована на различных входных данных, с учётом ограничения на размер стола ($2 \leq n \leq 20$).

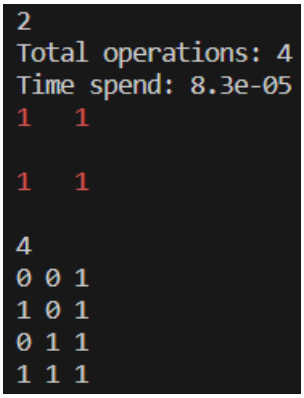


Рисунок 1 – Вывод программы для граничного случая n = 2.

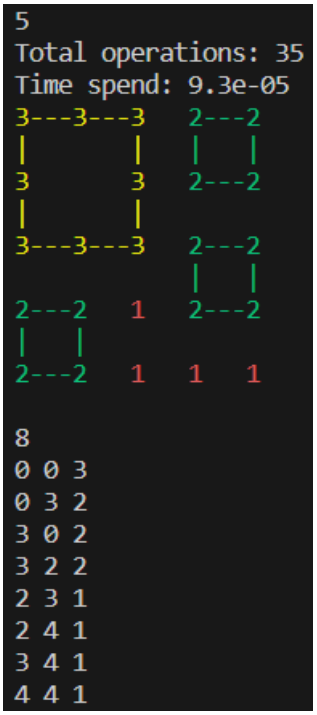


Рисунок 2 – Вывод программы для n = 5.

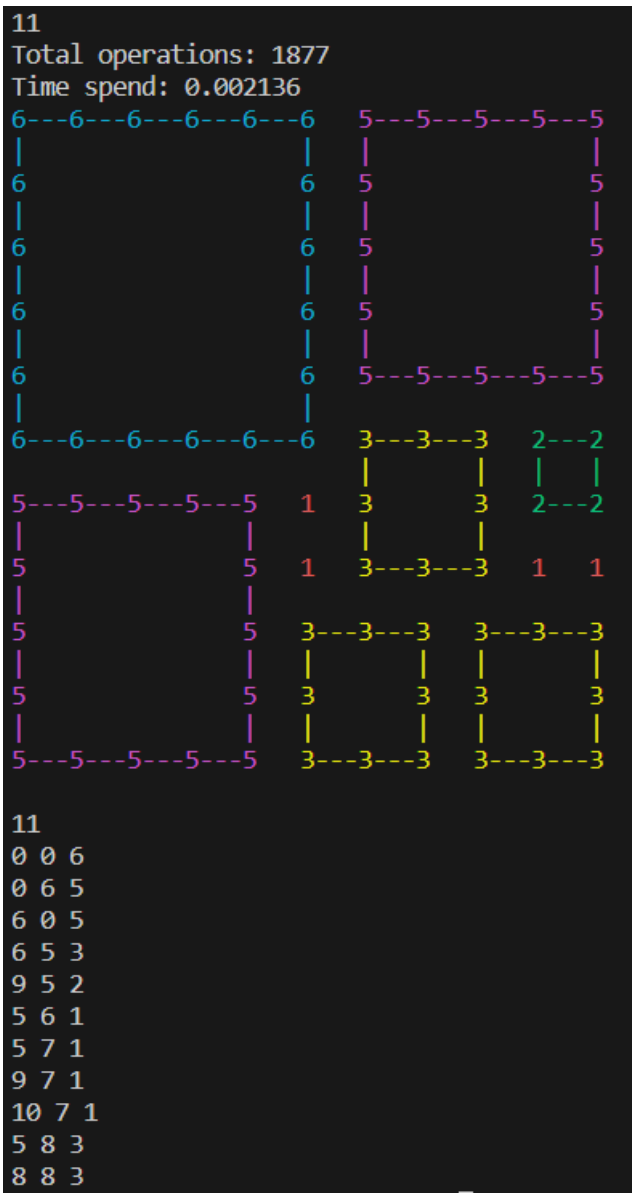


Рисунок 3 – Вывод программы для n = 11.

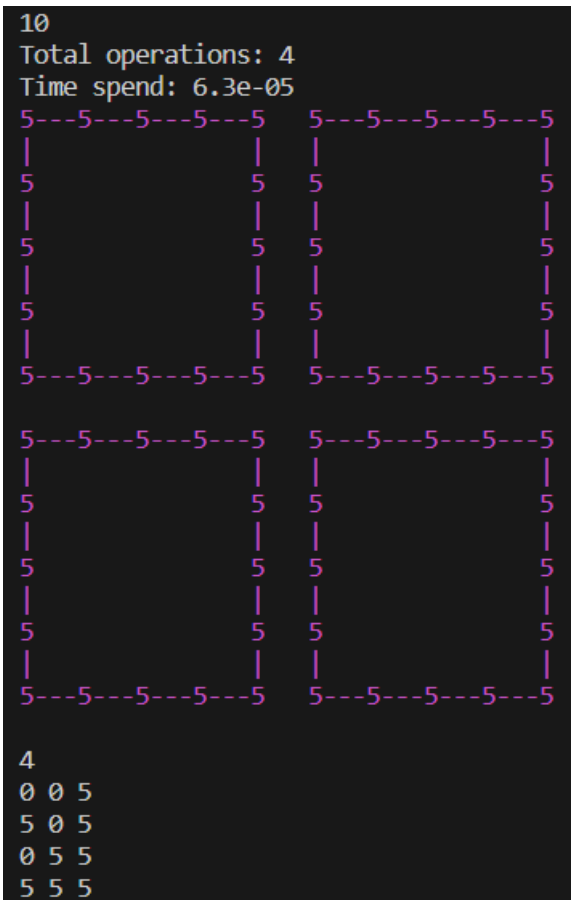


Рисунок 4 – Вывод программы для чётного n.

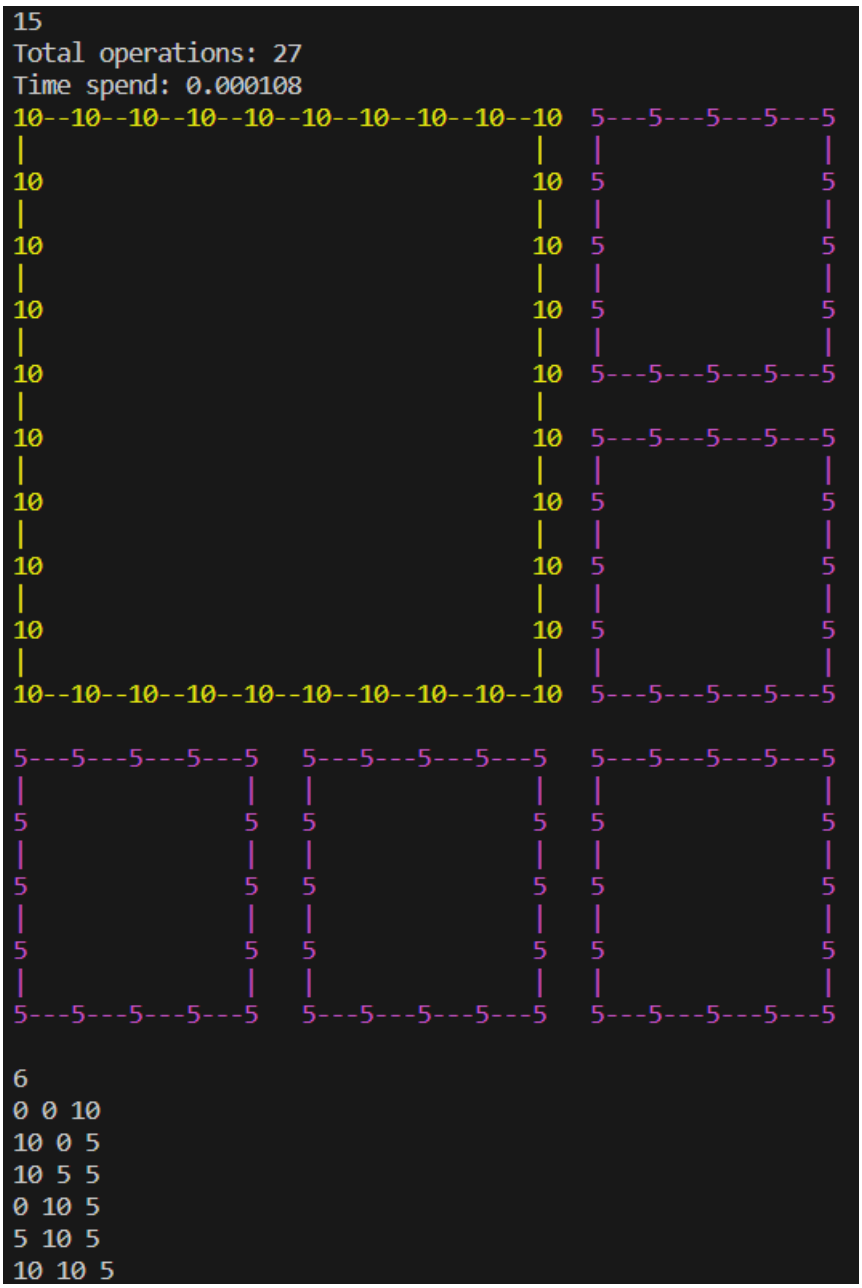


Рисунок 5 – Вывод программы для составного n.

Исследование

Вариант 3и. Итеративный бэктрекинг. Исследование количества операций от размера квадрата.

Для подсчёта количества операций, определим операцию как попытку присоединения квадрата к столу. Для чётных размеров стороны стола ответ всегда 4. Для составных размеров число операций зависит от их делителей: берётся расстановка для наименьшего из них и расширяется с коэффициентом, который является вторым делителем, дополняющим первого. Для простых

размеров разбиение требует меньшего числа операций, поскольку возможно сразу поставить три квадрата.

Таблица 1. Результаты полученных данных.

Размер стороны	Число операций	Затраченное время (с)	Число квадратов
2	4	0.000060	4
3	5	0.0000108	6
4	4	0.000052	4
5	35	0.000165	8
6	4	0.000058	4
7	**136**	0.000345	9
8	4	0.000058	4
9	27	0.000186	6
10	4	0.000095	4
11	**1877**	0.003337	11
12	4	0.000037	4
13	**4366**	0.006952	11
14	4	0.000068	4
15	27	0.000259	6
16	4	0.000032	4
17	**28156**	0.068246	12
18	4	0.000085	4
19	**86631**	0.193334	13
20	4	0.000072	4

Отобразим на графике результаты только для простых чисел, поскольку для остальных применяются значительные оптимизации. Сравним результат со смещённым графиком экспоненты $\boldsymbol{e^x}$:

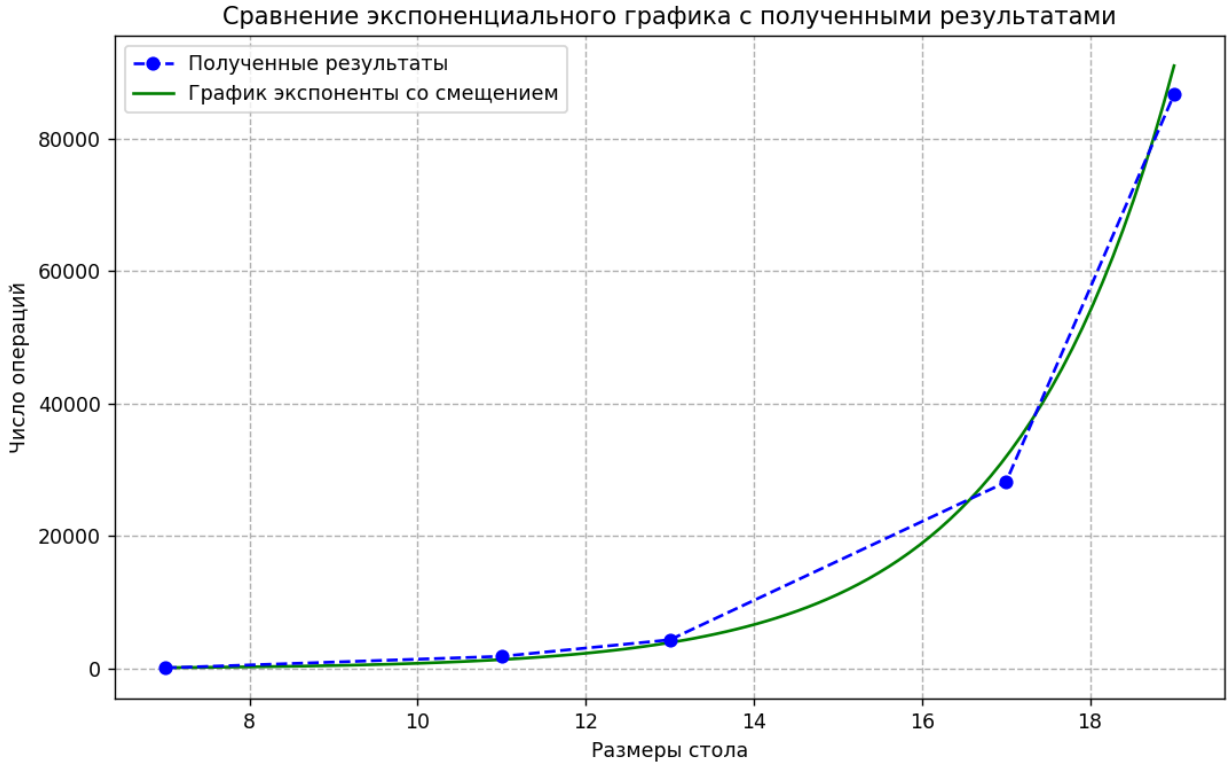


Рисунок 6 – Зависимость числа операций от размера стола.

Можно сделать следующие выводы по исследованию:

1. Число операций действительно растёт экспоненциально, что видно при сравнении с соответствующим графиком.

2. Время, необходимое для вычисления расстановки для простой стороны стола, также растёт крайне быстро.

Выводы

Во время выполнения лабораторной работы, была реализована программа, выполняющая итеративный бэктрекинг для поиска оптимального числа квадратов для покрытия стола. Было выявлено, что число операций зависит от размера поля и растёт экспоненциально.

ПРИЛОЖЕНИЕ
ИСХОДНЫЙ КОД ПРОГРАММЫ

Имя файла: main.cpp

```
#include "../include/Table.hpp"

#include <iostream>

int main() {
    int n;
    std::cin >> n;
    if (n < 2 || n > 20) {
        std::cout << "Invalid input: n should be between 2 and 20." << std::endl;
        return 0;
    }

    Table table(n);
    if (n % 2 == 0) {
        table.setEvenPreset();
    } else if (isPrime(n)) {
        table.setPrimePreset();
    } else {
        table.setCompositePreset();
    }

    table.printCanvas();
    table.printSquares();

    return 0;
}
```

Имя файла: Table.cpp

```
#include "../include/Table.hpp"

Table::Table(int n) : n(n) {
    this->canvas = std::vector<std::vector<int>>();
    for (int i = 0; i < n; ++i) {
        this->canvas.push_back(std::vector<int>(n));
    }
    this->squares = std::vector<Square>();
};

Coordinate Table::findEmpty() {
    for (int y = 0; y < this->n; ++y) {
        for (int x = 0; x < this->n; ++x) {
            if (this->canvas[y][x] == 0) return Coordinate{x, y};
            x += this->canvas[y][x] - 1;
        }
    }
    return Coordinate{-1, -1};
}

bool Table::isPossibleToInsert(Coordinate coordinate, int currentSize) {
    if (coordinate.x + currentSize > this->n || coordinate.y + currentSize > this->n) return false;
```

```
            for (int y = coordinate.y; y < coordinate.y + currentSize; ++y) {
                for (int x = coordinate.x; x < coordinate.x + currentSize; ++x) {
                    if (this->canvas[y][x] != 0) return false;
                }
            }
            return true;
        }

        void Table::insertSquare(Coordinate coordinate, int currentSize) {
            Square square = Square(coordinate, currentSize);
            for (int y = 0; y < currentSize; ++y) {
                this->canvas[coordinate.y + y][coordinate.x] = currentSize;
                this->canvas[coordinate.y + y][coordinate.x + currentSize - 1] =
currentSize;

                this->canvas[coordinate.y][coordinate.x + y] = currentSize;
                this->canvas[coordinate.y + currentSize - 1][coordinate.x + y] =
currentSize;
            }

            this->squares.push_back(square);
        }

        void Table::setEvenPreset() {
            clock_t time = clock();
            insertSquare({0, 0}, this->n/2);
            insertSquare({this->n/2, 0}, this->n/2);
            insertSquare({0, this->n/2}, this->n/2);
            insertSquare({this->n/2, this->n/2}, this->n/2);
            std::cout << "Total operations: 4" << std::endl;
            std::cout << "Time spend: " << (double)(clock() - time)/CLOCKS_PER_SEC
<< std::endl;
        }

        void Table::setPrimePreset() {
            clock_t time = clock();
            insertSquare({0, 0}, (this->n+1)/2);
            insertSquare({0, (this->n+1)/2}, (this->n)/2);
            insertSquare({(this->n+1)/2, 0}, (this->n)/2);
            *this = this->backtracking();
            std::cout << "Time spend: " << (double)(clock() - time)/CLOCKS_PER_SEC
<< std::endl;
        }

        void Table::setCompositePreset() {
            int div1 = getDivs(n);
            int div2 = n / div1;
            clock_t time = clock();
            Table smallerTable = Table(div1).backtracking();
            *this = smallerTable.scaler(div2);
            std::cout << "Time spend: " << (double)(clock() - time)/CLOCKS_PER_SEC
<< std::endl;
        }

        Table Table::backtracking() {
            std::queue<Table> q = std::queue<Table>();
            q.push(*this);
            int counter = 0;

            while (q.front().findEmpty() != Coordinate{-1, -1}) { // проверка на
заполненность
                Table lead = q.front(); // копия первого стола в очереди
                Coordinate coordinate = lead.findEmpty(); // поиск пустого места
```

```
            for (int i = this->n - 1; i > 0; --i) { // перебор размеров квадрата
для расстановки
                if (lead.isPossibleToInsert(coordinate, i)) { // проверка на
возможность вставки
                    Table current = lead;
                    current.insertSquare(coordinate, i); // вставка квадрата

                    if (current.findEmpty() == Coordinate{-1, -1}) { // проверка
на заполненность стола
                        std::cout << "Total operations: " << counter <<
std::endl;
                        return current;
                    }

                    if (this->n < 10) {
                        current.printIteration(counter);        //        вывод
промежуточного результата
                    }
                    q.push(current); // сохранение очередной расстановки
                }
                ++counter;
            }
            q.pop(); // удаление расстановки, которая дала все возможные
разбиения
        }
        std::cout << "Total operations: " << counter << std::endl;
        return q.front();
    }

    Table Table::scaler(int multiplier) {
        Table newTable(this->getN() * multiplier);

        for (const auto& square : this->getSquares()) {
            int x = square.getCoordinate().x * multiplier;
            int y = square.getCoordinate().y * multiplier;
            int length = square.getLength() * multiplier;
            newTable.insertSquare({x, y}, length);
        }
        return newTable;
    }

    void Table::printIteration(int iteration) {
        std::cout << "Iteration " << iteration << std::endl;
        for (int y = 0; y < this->n; ++y) {
            for (int x = 0; x < this->n; ++x) {
                std::cout << this->getCanvas()[y][x] << " ";
            }
            std::cout << std::endl;
        }
        std::cout << std::endl;
    }

    void Table::printCanvas() {
        int counter, size;
        for (int i = 0; i < this->n; ++i) {
            counter = 1;
            for (int j = 0; j < this->n; ++j) {
                size = this->canvas[i][j];

                if (size == 0) {
                    std::cout << "    ";
                } else if (size < 10) {
                    if (this->canvas[i][j+1] == 0 || counter == size) {
```

```
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << size
<< "   ";
                        } else {
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << size
<< "---";
                        }
                    } else {
                        if (this->canvas[i][j+1] == 0 || counter == size) {
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << size
<< "  ";
                        } else {
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << size
<< "--";
                        }
                    }

                    ++counter;
                    if (counter == size + 1)
                        counter = 1;
                }
                std::cout << std::endl;

                for (int j = 0; j < this->n; ++j) {
                    size = this->canvas[i][j];
                    if (size == 0 || size == 1) {
                        std::cout << "    ";
                        continue;
                    }
                    if (i+1 == n || size != canvas[i+1][j] || ((i-size+1) >= 0 &&
(i+size) < n && canvas[i-size+1][j] == size && canvas[i+size][j] == size &&
(canvas[i][j-1] == size || canvas[i][j+1] == size))) {
                        if (size > 2 && i-1 >= 0 && i+1 < n && canvas[i-1][j] ==
size && canvas[i+1][j] == size && ((j-1 >= 0 && canvas[i][j-1] == size &&
canvas[i+1][j-1] == 0) || (j+1 < n && canvas[i][j+1] == size && canvas[i+1][j+1]
== 0))) {
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << "|
";
                        } else {
                            std::cout << "\033[3" << canvas[i][j] % 7 << "m" << "
";
                        }
                    } else {
                        std::cout << "\033[3" << canvas[i][j] % 7 << "m" << "|   ";
                    }
                }
                std::cout << "\033[0m" << std::endl;
            }
        }

        void Table::printSquares() {
            std::cout << this->squares.size() << std::endl;
            for (const auto& square : this->squares) {
                std::cout    <<    square.getCoordinate().x    <<    "    "    <<
square.getCoordinate().y << " " << square.getLength() << std::endl;
            }
        }

        std::vector<std::vector<int>> Table::getCanvas() const {
            return this->canvas;
        }

        int Table::getN() const {
            return this->n;
        }
```

```
    std::vector<Square> Table::getSquares() const {
        return this->squares;
}
```

Имя файла: Square.hpp

```
    #pragma once

    #include "Utils.hpp"

    class Square {
        private:
            Coordinate coordinate;
            int length;

        public:
            Square(Coordinate coordinate, int length) : coordinate(coordinate),
length(length) {};

            Coordinate getCoordinate() const {
                return coordinate;
            }

            int getLength() const {
                return length;
            }
};
```

Имя файла: Table.hpp

```
    #pragma once

    #include "Square.hpp"

    #include <vector>
    #include <queue>

    class Table {
        private:
            std::vector<std::vector<int>> canvas;
            int n;
            std::vector<Square> squares;

        public:
            Table(int n);

            Coordinate findEmpty();
            bool isPossibleToInsert(Coordinate coordinate, int currentSize);
            void insertSquare(Coordinate coordinate, int currentSize);

            void setEvenPreset();
            void setPrimePreset();
            void setCompositePreset();

            Table backtracking();
            Table scaler(int multiplier);

            void printIteration(int iteration);
            void printCanvas();
            void printSquares();
```

```
        std::vector<std::vector<int>> getCanvas() const;
        int getN() const;
        std::vector<Square> getSquares() const;
};
```

Имя файла: Utils.hpp

```
#pragma once

#include <iostream>
#include <math.h>

struct Coordinate {
    int x;
    int y;

    bool operator==(const Coordinate& other) const {
        return x == other.x && y == other.y;
    }

    bool operator!=(const Coordinate& other) const {
        return !(*this == other);
    }
};

inline int isPrime(int n) {
    for (int i = 2; i < int(sqrt(n)) + 1; i++) {
        if (n % i == 0) return 0;
    }
    return 1;
}

inline int getDivs(int n) {
    for (int i = 2; i < int(sqrt(n)) + 1; i++) {
        if (n % i == 0) {
            return i;
        }
    }
    throw std::invalid_argument("Error: Number is prime.");
}
```