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. 9383 _____ Камзолов Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

Задание. Вариант – 1и(итеративный бэктрекинг).

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).

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 ($2 \le N \le 40$).

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).

Описание алгоритма.

Для решения задачи был реализован итеративный алгоритм бэктрекинга, который перебирает все возможные заполнения квадрата квадратами меньшей стороны:

1. Проходим циклом по матрице, которая является прототипом столешницы и находим минимальную свободную ячейку, в которую можно вместить квадрат и помещаем туда максимально возможный (квадраты помещаем в стек). Так делаем до тех пор, пока массив не будет полностью заполнен.

2. Если разложение, которое мы получили на первом шаге является минимальным на текущий момент, то запоминаем его (запоминаем стек).
3. Дальше функция, которая имитирует работу бэктрекинга поочерёдно удаляет из стека квадраты с единичной стороной, если ей на пути встречается квадрат со стороной больше единицы, то она уменьшает сторону этого квадрата на один и возвращается к шагу 1. В том случае, если удаление квадратов больше невозможно – алгоритм прекращает работу.

Было подмечено, что у столешниц со стороной равной простому числу (кроме 2) есть совпадения при разложении. У всех них совпадают три квадрата, расположенных по трем смежным углам столешницы. Таким образом в левом верхнем углу расположен квадрат со стороной W / 2 + 1, в левом нижнем углу – W / 2, в правом верхнем – W / 2, где W – сторона столешницы. Таким образом, проставив эти квадраты до начала работы основного алгоритма, мы уменьшили количество переборных операций примерно в 4 раза. Для столешниц со стороной кратной 2 – написан отдельный случай, который разбивает столешницу на 4 квадрата со стороной W / 2. Также было подмечено, что у столешницы со стороной не равной простому числу разложение совпадает с разложением столешницы, у которой сторона является минимальным простым делителем стороны первой (столешницы) в уменьшенном масштабе.

Не смотря на все оптимизации алгоритм все равно имеет экспоненциальную сложность.

Описание функций и структур данных.

Все операции с полем подразумевают работу с матрицей размера W*W. Матрица реализуется с помощью std::vector.

struct Square – структура данных, которая является прототипом квадратных обрезков доски. В ней содержатся координаты левого верхнего угла квадрата, а также длина его стороны.

class Table – класс, являющийся прототипом столешницы. Внутри этого класса содержатся методы, который реализуют алгоритм заполнения квадратами столешницы:

void running() – метод, который в зависимости от введенных данных запускает алгоритм заполнения (здесь рассматриваются особые случаи для квадратов со сторонами, делящимися на 2/3/7).

void calculations() – метод, реализующий работу алгоритма – заполняет столешницу квадратами, проверяет является ли это разложение минимальным (если да, то запоминает его), а затем вызывает бэктрекинг.

void startAlignment() – метод, реализующий оптимизацию с расставлением трех изначальных квадратов.

bool newSquare() – метод, который проверяет можно ли поместить в текущую позицию квадрат с выбранной стороной, если это возможно, то помещает его в это место.

bool clearSquare(int x, int y, int side) – метод, очищающий квадрат по выбранным координатам (x и y), с заданной стороной (side).

bool alignment() – метод, реализующий алгоритм заполнения столешницы квадратами с максимально возможной стороной. Возвращает true в том случае, если ему удалось заполнить квадрат и false, если не удалось.

bool backtracking() – метод, реализующий алгоритм бэктрекинга. Пробегается по стеку и удаляет квадраты из него в том случае, если квадрат имеет единичную сторону или уменьшает сторону квадрата на один, если сторона квадрата была больше единицы. Возвращает true, если дальнейшее заполнение имеет смысл и false, если со столешницы удалены все квадраты, кроме трех изначальных.

Также для функций написано тестирование с помощью библиотеки Catch2.

Для файлов тестирования и основной программы написан Makefile.

```
1 1 7
nikita@DESKTOP-7FRVGJJ:/mnt/c/CodeForces/AlgosLab1$ ./run_tests
===============================================================================
All tests passed (15 assertions in 6 test cases)
```

Рисунок 1 – Демонстрация корректного отрабатывания тестирования.

Демонстрация работы.

```
6
Count of squares: 4
x: 1 y: 1 side: 3
x: 4 y: 1 side: 3
x: 1 y: 4 side: 3
x: 4 y: 4 side: 3
```

Рисунок 2 – Демонстрация работы программы при столешнице со стороной 6.

```
9
Count of squares: 6
x: 7 y: 7 side: 3
x: 4 y: 7 side: 3
x: 7 y: 4 side: 3
x: 1 y: 7 side: 3
x: 7 y: 1 side: 3
x: 1 y: 1 side: 6
```

Рисунок 3 – Демонстрация работы программы при столешнице со стороной 9.

```
13
Count of squares: 11
x: 9 y: 12 side: 2
x: 7 y: 12 side: 2
x: 11 y: 11 side: 3
x: 10 y: 11 side: 1
x: 7 y: 9 side: 3
x: 7 y: 8 side: 1
x: 10 y: 7 side: 4
x: 8 y: 7 side: 2
x: 1 y: 8 side: 6
x: 8 y: 1 side: 6
x: 1 y: 1 side: 7
```

Рисунок 4 – Демонстрация работы программы при столешнице со стороной 13.

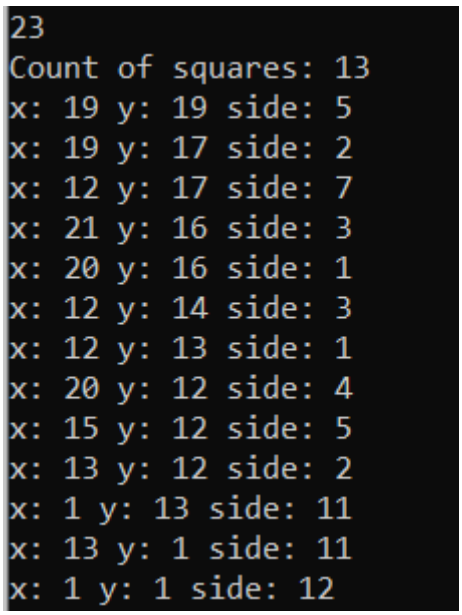

Рисунок 5 – Демонстрация работы программы при столешнице со стороной 23.

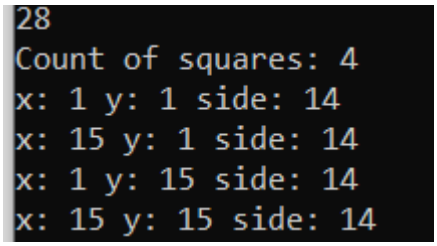

Рисунок 6 – Демонстрация работы программы при столешнице со стороной 28.

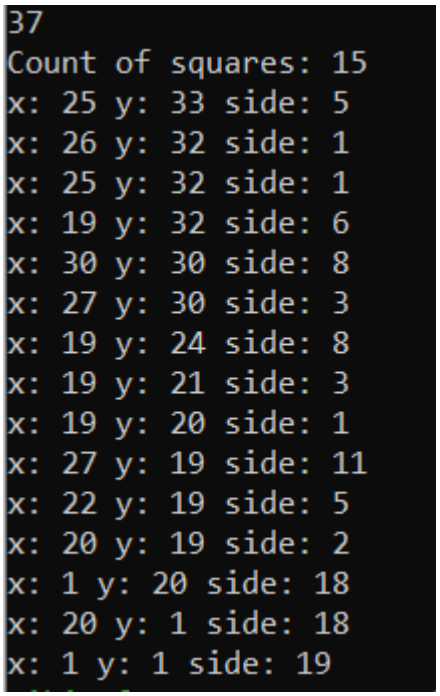

Рисунок 7 – Демонстрация работы программы при столешнице со стороной 37.

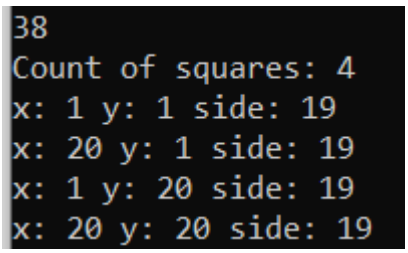

Рисунок 8 – Демонстрация работы программы при столешнице со стороной 38.

Выводы.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов. Придумана оптимизация, позволяющая сократить кол-во переборных операций в 4 раза. Написана программа, реализующая алгоритм заполнения квадрата минимальным кол-вом квадратов меньшей стороны с помощью поиска с возвратом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл main.cpp:

```
#include "Table.hpp"

using namespace std;

int main() {
    int side;
    cin >> side;
    Table a(side);
    a.running();

}
```

Файл Table.hpp:

```
#include <bitset>
#include <iostream>
#include <list>
#include <vector>
#include <stack>


struct Square {
    int x = 0;
    int y = 0;
    int side = 0;
    Square(int x, int y, int side): x(x), y(y), side(side) {}
};

class Table {
    std::vector <std::vector<int>> table;
    std::vector <std::vector<int>> bestTable;

    int tempY = 0;
    int tempX = 0;
    int width;
    int squaresCounter = 0;
    int minSquares = 0;

    int currentSquare;

    std::stack<Square> squares;

public:
    const std::vector <std::vector<int>> getTable() { return table; }
    int getsquaresCounter() { return squaresCounter; }
    void setTempX(int tempX) { this->tempX = tempX; }
    void setTempY(int tempY) { this->tempY = tempY; }
    void setCurrentSquare(int tempSquare) { currentSquare = tempSquare; }
    void calculations();
```

```
    void printAnswer(int pow = 1);
    void startAlignment();
    void print();
    bool isFullAlignment();
    bool newSquare();
    void clearSquare(int x, int y, int side);
    bool backTracking();

    std::stack<Square> bestCase;
    void running();
    Table(int width);
};
```

Файл Table.cpp:

```
#include "Table.hpp"


Table::Table(int width) {
    this->width = width;
    table.resize(width);
    for (int i = 0; i < width; i++) {
        table[i].resize(width);
        for(int j = 0; j < width; j++) {
            table[i][j] = 0;
        }
    }
}

void Table::calculations() {
    startAlignment();
    int min = width * width;
    minSquares = min;
    do {
        if(isFullAlignment()) {
            if(squaresCounter < min) {
                bestCase = squares;
                bestTable = table;
                min = squaresCounter;
                minSquares = min;
            }
        }
        if(!backTracking())
            break;
    }while(true);
}

void Table::running() {
    if (width % 2 == 0) {
        std::cout << "Count of squares: " <<  4 << std::endl;
        std::cout << "x: " << 1 << " y: " << 1 << " side: " << width / 2
<< '\n';
        std::cout << "x: " << width / 2 + 1
                  << " y: " << 1
```

```
                  << " side: " << width / 2
                  << '\n';
        std::cout << "x: " << 1
                  << " y: " << width / 2 + 1
                  << " side: " << width / 2
                  << '\n';
        std::cout << "x: " << width / 2 + 1
                  << " y: " << width / 2 + 1
                  << " side: " << width / 2
                  << '\n';
        return;
    }

    if (width % 3 == 0) {
        int tempW = width;
        width = 3;
        calculations();
        printAnswer(tempW / width);
        return;
    }

    if (width % 5 == 0) {
        int tempW = width;
        width = 5;
        calculations();
        printAnswer(tempW / width);
        return;
    }

    calculations();
    printAnswer();

}

void Table::printAnswer(int pow) {
    std::cout << "Count of squares: "<< minSquares << std::endl;
    for(int i = 0; i < minSquares; i++) {
        Square temp = bestCase.top();
        bestCase.pop();
        std::cout << "x: " <<temp.x * pow + 1
            << " y: " << temp.y * pow + 1
            << " side: "<< temp.side * pow
            << std::endl;
    }
}

void Table::startAlignment() {
    for(int i = 0; i < width; i++) {
        for(int j = 0; j < width; j++) {
            if(i < width/2 + 1 && j < width/2 + 1) {
                table[i][j] = 1;
            }
            else if(j < width/2) {
                table[i][j] = 2;
            }
            else if(i < width/2) {
                table[i][j] = 3;
```

```
            }
        }
    }
    squaresCounter = 3;
    squares.push(Square(0, 0, width/2 + 1));
    squares.push(Square(width/2 + 1, 0, width/2));
    squares.push(Square(0, width/2 + 1, width/2));
    tempX = width/2;
    tempY = width/2;
}

void Table::print() {
    for (int i = 0; i < width; i++) {
        for(int j = 0; j < width; j++) {
            std::cout << bestTable[i][j];
        }
        std::cout << std::endl;
    }
}
bool Table::isFullAlignment() {
    if(currentSquare == 0)
        currentSquare = width / 2;
    for(int i = width/2; i < width; i++) {
        for(int j = width/2; j < width; j++) {
            if(table[i][j] == 0) {
                tempY = i;
                tempX = j;
                if(squaresCounter >= minSquares) {
                    return false;
                }
                while(true) {
                    if (newSquare())
                    {
                        currentSquare = width / 2;
                        break;
                    }
                    else currentSquare--;
                }

            }
        }
    }
    return true;
}

bool Table::newSquare() {
    if(tempX + currentSquare > width
    || tempY + currentSquare > width) return false;
    for (int i = tempY; i < tempY + currentSquare; i++) {
        for (int j = tempX; j < tempX + currentSquare; j++) {
            if(table[i][j]) return false;
        }
    }
    squaresCounter++;
    for (int i = tempY; i < tempY + currentSquare; i++) {
        for (int j = tempX; j < tempX + currentSquare; j++) {
            table[i][j] = squaresCounter;
```

```
        }
    }

    squares.push(Square(tempX, tempY, currentSquare));
    return true;
}

void Table::clearSquare(int x, int y, int side) {
    if(x < 0 || y < 0 || x + side > width
        || y + side > width || side < 0 || squares.empty())
        return;
    for(int i = y; i < y + side; i++) {
        for(int j = x; j < x + side; j++) {
            table[i][j] = 0;
        }
    }
    squares.pop();
    squaresCounter--;
}

bool Table::backTracking() {
    Square temp = squares.top();
    while(squares.size() > 3 && temp.side == 1) {
        clearSquare(squares.top().x, squares.top().y,
squares.top().side);
        temp = squares.top();
    }

    if(squares.size() == 3){
        return false;
    }
    clearSquare(squares.top().x, squares.top().y, squares.top().side);
    currentSquare = temp.side - 1;
    return true;
}
```

Файл AlgosTest.cpp:

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"
#include "../Source/Table.hpp"




TEST_CASE( "Squares are placed when appropriate", "[placing]" ) {
    Table table = Table(10);
    std::vector<std::vector<int>> testTable = {
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
```

```
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};
table.setCurrentSquare(5);
table.setTempX(0);
table.setTempY(0);
table.newSquare();

REQUIRE(table.getTable() == testTable);

table.clearSquare(0, 0, 5);


testTable = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 1, 1, 0, 2, 2, 0, 0 },
            {0, 0, 0, 1, 1, 0, 2, 2, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

table.setTempX(3);
table.setTempY(2);
table.setCurrentSquare(2);
table.newSquare();

table.setTempX(6);
table.newSquare();

REQUIRE(table.getTable() == testTable);

table.clearSquare(3, 2, 2);
table.clearSquare(6, 2, 2);

testTable = {{1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
            {0, 0, 0, 0, 0, 2, 2, 2, 2, 2 },
            {0, 0, 0, 0, 0, 2, 2, 2, 2, 2 },
            {0, 0, 0, 0, 0, 2, 2, 2, 2, 2 },
            {0, 0, 0, 0, 0, 2, 2, 2, 2, 2 },
            {0, 0, 0, 0, 0, 2, 2, 2, 2, 2 }};

table.setCurrentSquare(5);
table.setTempX(0);
table.setTempY(0);
table.newSquare();
```

```
    table.setTempX(5);
    table.setTempY(5);
    table.newSquare();

    REQUIRE(table.getTable() == testTable);

    table.clearSquare(0, 0, 5);
    table.clearSquare(5, 5, 5);
}

TEST_CASE( "Squares are not placed when they are not appropriate", "[not
placing]" ) {
    Table table = Table(10);
    std::vector<std::vector<int>> testTable = {
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};
    table.setCurrentSquare(5);
    table.setTempX(6);
    table.setTempY(6);
    table.newSquare();

    REQUIRE(table.getTable() == testTable);


    testTable = {{0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
                {0, 0, 0, 1, 1, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

    table.setTempX(3);
    table.setTempY(2);
    table.setCurrentSquare(2);
    table.newSquare();

    table.newSquare();

    REQUIRE(table.getTable() == testTable);

    table.clearSquare(3, 2, 2);

    testTable = {{1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
```

```
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

    table.setCurrentSquare(5);
    table.setTempX(0);
    table.setTempY(0);
    table.newSquare();

    table.setTempX(4);
    table.setTempY(4);
    table.newSquare();

    REQUIRE(table.getTable() == testTable);
    table.clearSquare(0, 0, 5);
}

TEST_CASE( "Squares are cleared when it is possible ", "[clearing]" ) {
    Table table = Table(10);
    std::vector<std::vector<int>> testTable = {
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

    table.setCurrentSquare(10);
    table.setTempX(0);
    table.setTempY(0);
    table.newSquare();
    table.clearSquare(0, 0, 10);

    REQUIRE(table.getTable() == testTable);



    testTable = {{1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 0, 0, 0, 0, 0, 0, 0 },
                {1, 1, 1, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};

    table.setCurrentSquare(5);
```

```
    table.setTempX(0);
    table.setTempY(0);
    table.newSquare();

    table.clearSquare(3, 3, 2);

    REQUIRE(table.getTable() == testTable);

    table.setCurrentSquare(2);
    table.setTempX(3);
    table.setTempY(3);
    table.newSquare();
    table.clearSquare(0, 0, 5);

}

TEST_CASE( "Squares are not cleared when it is not possible ", "[no
clearing]" ) {
    Table table = Table(10);
    std::vector<std::vector<int>> testTable = {
                {0, 0, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};
    table.setCurrentSquare(5);
    table.setTempX(0);
    table.setTempY(0);
    table.newSquare();
    table.clearSquare(-1, 0, 2);
    REQUIRE(table.getTable() != testTable);

    testTable = {{1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {1, 1, 1, 1, 1, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 },
                {0, 0, 0, 0, 0, 0, 0, 0, 0, 0 }};
    REQUIRE(table.getTable() == testTable);
    table.clearSquare(0, 0, 5);
}



TEST_CASE( "Start alignment works right ", "[start alignment]" ) {
    Table table3 = Table(3);
    std::vector<std::vector<int>> testTable = {
```

```
                {1, 1, 3},
                {1, 1 , 0},
                {2, 0, 0}};
    table3.startAlignment();
    REQUIRE(table3.getTable() == testTable);

    table3.clearSquare(2, 0, 1);
    table3.clearSquare(0, 2, 1);
    table3.clearSquare(0, 0, 2);

    testTable = {{1, 1, 1, 3, 3},
                {1, 1, 1, 3, 3},
                {1, 1, 1, 0, 0},
                {2, 2, 0, 0, 0},
                {2, 2, 0, 0, 0}};
    Table table5 = Table(5);
    table5.startAlignment();
    REQUIRE(table5.getTable() == testTable);
    table5.clearSquare(3, 0, 2);
    table5.clearSquare(0, 3, 2);
    table5.clearSquare(0, 0, 3);

}

TEST_CASE( "Backtracking works right ", "[alignment]" ) {
    Table table3 = Table(3);
    std::vector<std::vector<int>> testTable = {
                {1, 1, 3},
                {1, 1, 0},
                {2, 0, 0}};

    table3.startAlignment();

    table3.setCurrentSquare(1);
    table3.setTempX(1);
    table3.setTempY(2);
    table3.newSquare();

    table3.setTempX(2);
    table3.setTempY(2);
    table3.newSquare();

    table3.backTracking();

    REQUIRE(table3.getTable() == testTable);

    table3.setCurrentSquare(0);
    table3.isFullAlignment();
    table3.backTracking();
    REQUIRE(table3.getTable() == testTable);

    Table table5 = Table(5);
    testTable = {{1, 1, 1, 3, 3},
                {1, 1, 1, 3, 3},
                {1, 1, 1, 0, 0},
                {2, 2, 0, 0, 0},
                {2, 2, 0, 0, 0}};
```

```
    table5.startAlignment();
    table5.setCurrentSquare(2);
    table5.setTempX(3);
    table5.setTempY(2);
    table5.newSquare();
    table5.backTracking();

    REQUIRE(table5.getTable() == testTable);
}
```