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 _____ Крейсманн К.В.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Познакомиться с алгоритмическим методом «Поиск с возвратом», и применить его для построения алгоритма нахождения минимального заполнения квадрата меньшими квадратами. Проанализировать зависимость времени выполнения работы алгоритма от длины стороны квадрата.

Задание.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 11 до N - 1$N-1$, и у него есть неограниченное число обрезков любого размера. Но ему очень хочется получить большую столешницу - квадрат размера N*N. Он может получить ее, собрав из уже имеющихся обрезков(квадратов).

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

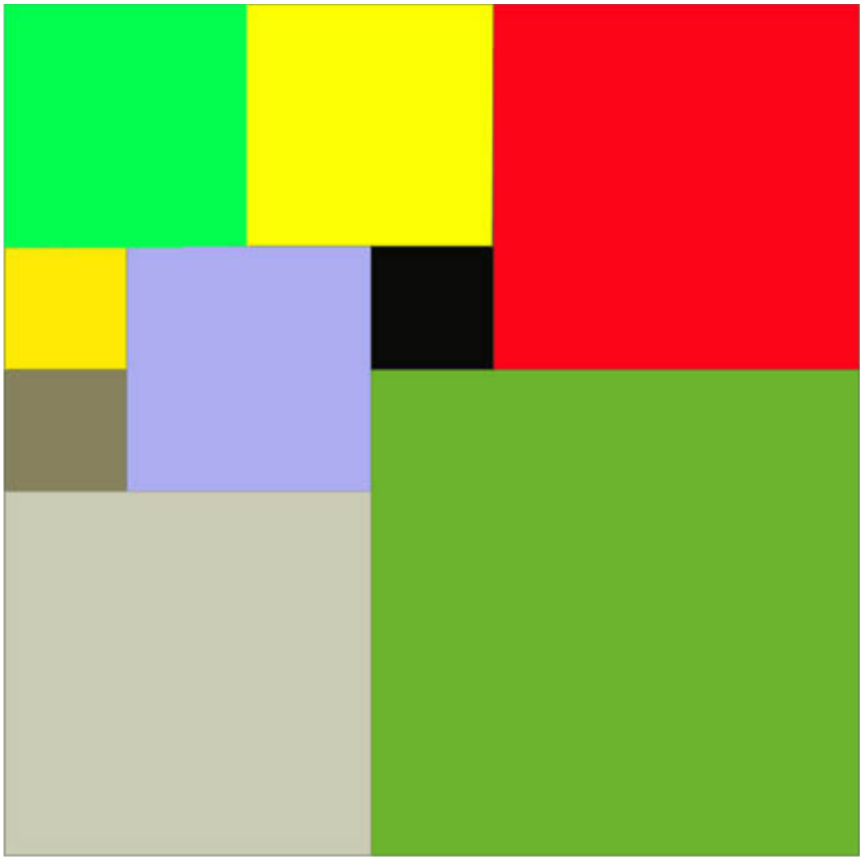

Рисунок 1 - Столешница 7*7

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<=40).

Выходные данные

Одно число K, задающее минимальное количество обрезков, из которых можно построить столешницу заданного размера N. Дале должны идти K строк, каждая из которых должна содержать три целых числа x,y,w, задающие координаты левого верхнего угла и длину стороны соответствующего квадрата.

Вариант 2и.

Итеративный бэктрэкинг. Исследование времени выполнения от размера квадрата.

Теория

Бэктрэкинг – 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. Данный алгоритм позволяет найти все существующие решения. Для ускорения метода стараются организовать вычисления таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

Время нахождения решения может быть очень велико даже при небольших размерностях задачи.

Описание алгоритма

Был реализован итеративный алгоритм поиска с возвратом. Алгоритм перебирает все возможные варианты расстановки квадратов и ищет решение с наименьшим количеством квадратов.

На каждом шаге алгоритма происходит работа с вершиной стека, где указан размер квадрата и координата. Когда стек станет пустым, алгоритм завершит работу.

Если на каком-то шаге количество поставленных на столешницу квадратов равно или больше, чем в минимальном на данный момент найденном решении без единицы, то продолжение этого шага не рассматривается.

Алгоритм с помощью функции nextCoordinate() получает следующую пустую координату (самую верхнюю левую) и начиная с размера 1 пытается поставить на эту координату квадрат.

Если координата найдена (не всё поле заполнено), то в стек кладется информация (координата и размер), которая указывает на какую координату нужно ставить и какого размера квадрат, когда алгоритм вернется на этот шаг (т.е. если на текущем шаге был поставлен на координату (x,y) квадрат размером m, то когда алгоритм «вернется» к этому месту, он должен пытаться на (x,y)

поставить квадрат уже с размером (m+1)). После этого в стек заносится информация для следующего шага, с найденной пустой координатой и начальным размером квадрата (1).

Если координата не найдена, а значит все поле заполнено квадратами, то значит найдено лучшее на текущий момент решение (т.к. если бы это было не лучшее решение, то оно бы отбросилось ранее), оно запоминается. Затем алгоритм возвращается назад.

Когда алгоритм вернется к первому шагу, и размер будет превышать максимальный, из стека удалится последний элемент и алгоритм завершит работу.

Оптимизации алгоритма

1) Любой квадрат со стороной нечетной длины можно представить как минимум N+3 квадратами, поставив квадрат со стороной [N/2] + 1 в верхний левый угол, и три квадрата со сторонами [N/2] в оставшиеся углы, оставшиеся клетки заполнив квадратами со стороной 1.

2) Также было замечено, что любое оптимальное решение для квадратов с нечетной стороной содержит три квадрата размерностями [N/2]+1, [N/2], [N/2], которые стоят по углам.

3) Если длина стороны столешницы составное число, то размер оптимального решения будет таким же, как и для столешницы со стороной равной наименьшему делителю (кроме 1) длины стороны данной столешницы.

В виду оптимизаций, для части чисел можно построить оптимальное решение моментально, но для остальных чисел алгоритм имеет экспоненциальную сложность.

Описание основных функций и структур данных

1) Table – столешница, представляет собой вектор векторов чисел.
2) Coordinate – координата, пара чисел.
3) Solution – решение, стек квадратов.
4) Pair – пара из числа и координаты.

5) StackBacktrack - стек Pair-ов.

6) Square – квадрат, содержит размер и координату.

7) createSquare(Coordinate coordinate ,size_t size) – создает квадрат, возвращает Square с размером size и координатой coordinate.

8) checkPlaceForSquare(Coordinate coord, size_t size, Table table) – проверяет «влезет» ли квадрат с размером size на столешницу, если его верхний левый угол находится на координате coord.

9) addSquare(Square square, Table table, Solution solution) – добавляет квадрат на столешницу и в решение Solution.

10) dellSquare(Square square, Table table, Solution solution) – удаляет квадрат.

11) nextCoordinate(Table table) – получение следующей свободной координаты, возвращает std::optional<Coordinate>, если значение std::nullopt, то столешница заполнена.

12) evenFillTable(Table table) – заполняет столешницу и возвращает решение, если N-четное число.

13) multipleOfThreeFillTable(Table table) – заполняет столешницу и возвращает решение, если N делится на 3.

14) multipleOfFiveFillTable(Table table) – заполняет столешницу и возвращает решение, если N делится на 5.

15) initialFill(Table table) – добавляет на столешницу первые 3 квадрата.

16) Process(Table table, Solution tempSolution, Solution bestSolution) – основная функций, реализующая перебор с возвратом.

Тестирование

Входные данные	Выходные данные
4	4 3 3 2 1 3 2 3 1 2 1 1 2
7	9 6 6 2 4 6 2 5 5 1 4 5 1 6 4 2 5 4 1 1 5 3 5 1 3 1 1 4
9	6 4 7 3 1 7 3 7 7 3 7 4 3 7 1 3 1 1 6
11	11 9 9 3 6 9 3 6 8 1 7 7 2

	6 7 1 9 6 3 8 6 1 7 6 1 1 7 5 7 1 5 1 1 6
13	11 11 11 3 11 10 1 7 10 4 12 9 2 7 8 2 12 7 2 9 7 3 8 7 1 1 8 6 8 1 6 1 1 7
20	4 11 11 10 1 11 10 11 1 10 1 1 10
23	13 16 21 3 16 20 1 12 20 4 19 19 5

	17 19 2 12 15 5 12 13 2 17 12 7 14 12 3 13 12 1 1 13 11 13 1 11 1 1 12
25	8 11 21 5 11 16 5 21 11 5 16 11 5 16 16 10 1 16 10 16 1 10 1 1 15
17	12 9 14 4 13 13 5 13 12 1 11 12 2 9 12 2 9 10 2 14 9 4 11 9 3 10 9 1 1 10 8

	10 1 8 1 1 9

Таблица 1 - Тестирование

Анализ времени работы алгоритма

Т.к. для чисел кратных 2,3,5 решение строится очень быстро и без алгоритма перебора, то анализ произведен для остальных чисел.

Длина стороны квадрата	Время работы алгоритма(сек)
7	0.002
11	0.007
13	0.014
17	0.104
19	0.352
23	0.814
29	11.2

Таблица 2 – Время работы алгоритма

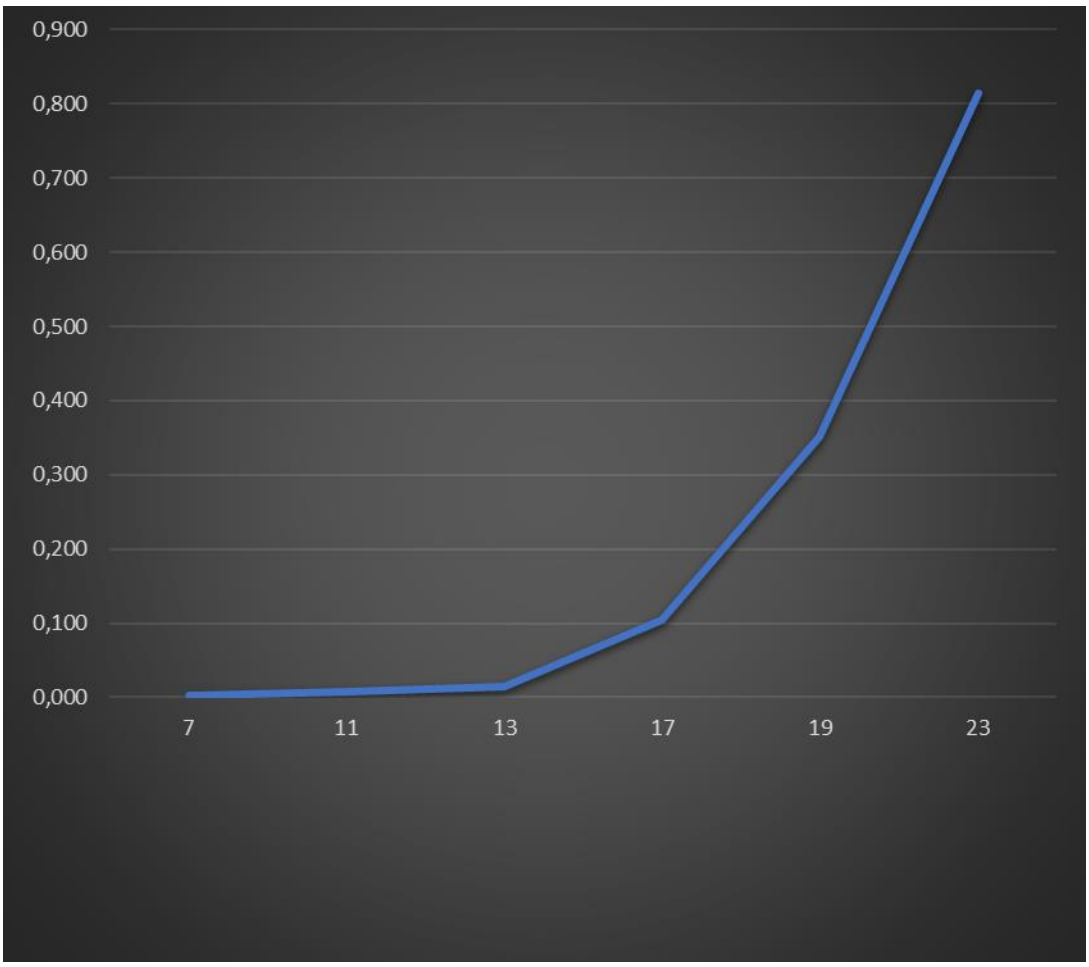

Рисунок 2 - График зависимости времени работы от N

Из рисунка видно, что время работы алгоритма экспоненциально зависит от длины стороны квадрата.

Вывод.

Произведено знакомство с алгоритмическим методом «Поиск с возвратом». С его помощью построен и реализован алгоритм для нахождения минимального количества квадратов со сторонами <N, с помощью которых можно заполнить квадрат со стороной N. Найдена оптимизация для составных чисел.

Исследована зависимость времени работы алгоритма от длины стороны квадрата. Поиск с возвратом работает очень долго даже при не больших значениях входных данных.

Приложение А

main.cpp

```
#include "header.hpp"
Table initTable()
{
   unsigned short N=0;
   while(N<2 || N>40)
   {
      std::cout<<"Input 2<=N<=40: ";
      std::cin >> N;
   }
   return createTableNxN(N);
}



Solution fillTable(Table& table)
{
   if (table.size() % 2 == 0)
   {
      return evenFillTable(table);
   }
   else if (table.size() % 3 == 0)
   {
      return multipleOfThreeFillTable(table);
   }
   else if (table.size() % 5 == 0)
   {
      return multipleOfFiveFillTable(table);
   }
   else
   {
      Solution tempSolution = initialFill(table);//расставляет первые три квадрата
      Solution result;
      auto start = std::chrono::steady_clock().now();
```

```
        Process(table,tempSolution, result);
        auto end = std::chrono::steady_clock().now();
        std::cout << "\nTime:" << std::chrono::duration_cast<std::chrono::milliseconds>(end-
start).count() << '\n';
        return result;
    }
}

void printAnswer(Solution best)
{
    std::cout << best.size() << '\n';
    while (!best.empty())
    {
        std::cout << best.top().x + 1 << ' ' << best.top().y + 1 << ' ' << best.top().size << '\n';
        best.pop();
    }
}


int main()
{
    Table table = initTable();
    printAnswer(fillTable(table));
    return 0;
}
```

header.hpp

```
#pragma once

#include <iostream>
#include <vector>
#include <stack>
#include <algorithm>
#include <ctime>
#include <optional>
```

```
#include <chrono>

struct Square;
using Table = std::vector<std::vector<unsigned short>>;
using Coordinate = std::pair<unsigned short, unsigned short>;
using Solution = std::stack<Square>;
using Pair = std::pair<unsigned short, Coordinate>;
using StackBacktrack = std::stack<Pair>;

struct Square
{
   size_t size;
   int x, y;
};

Square createSquare(const int y, const int x, const size_t size);
bool checkPlaceForSquare(const Coordinate coord, const size_t size, const Table table);
void addSquare(const Square& square, Table& table, Solution& solution);
void dellSquare(Table& table, Solution& solution);
std::optional<Coordinate> nextCoordinate(const Table& table);
Solution evenFillTable(Table& table);
Solution multipleOfThreeFillTable(Table& table);
Solution multipleOfFiveFillTable(Table& table);
Solution initialFill(Table& table);
void Process(Table& table, Solution& tempSolution, Solution& bestSolution);
Table createTableNxN(const unsigned short N);
```

backtracking.cpp

```
#include "header.hpp"
Square createSquare(const int y, const int x, const size_t size)
{
   Square square;
   square.size = size;
   square.x = x;
   square.y = y;
```

```
    return square;
}
/*void printTable(const Table table)
{
    for (int i = 0; i < table.size(); i++)
    {
        for (int j = 0; j < table.size(); j++)
        {
            std::cout << table[i][j] << " ";
        }
        std::cout << '\n';
    }
    std::cout << '\n';
}*/

bool checkPlaceForSquare(const Coordinate coord, const size_t size, const Table table)
{
    try
    {
        if(size==0 || size>=table.size())
        {
            throw(0);
        }
        if (coord.first + size > table.size() || coord.second + size > table.size())
        {
            return false;
        }
        for (int i = coord.first; i < coord.first + size; i++)
        {
            for (int j = coord.second; j < coord.second + size; j++)
            {
                if (table[i][j] == 1)
                {
                    return false;
                }
```

```
            }
        }
        return true;
    }
    catch(int)
    {
        std::cerr<<"invalid size (checkPlaceForSquare())"<<'\n';
        exit(0);
    }
}

void addSquare(const Square& square, Table& table, Solution& solution)
{
    try{
        if(square.size > table.size()-1)
        {
            throw(0);
        }

        for (int i = square.y; i < square.y + square.size; i++)
        {
            for (int j = square.x; j < square.x + square.size; j++)
            {
                if(table[i][j]==1)
                {
                    throw(0);
                }
                table[i][j] = 1;
            }
        }
        solution.push(square);
    }
    catch(int)
    {
        std::cerr<<"invalid square size or coordinate (addSquare())"<<'\n';
```

```
        exit(0);
    }
}

void dellSquare(Table& table, Solution& solution)
{
    Square square = solution.top();
    try
    {
        if(solution.size()==0)
        {
            throw 0;
        }
        if(solution.top().x + solution.top().size > table.size() ||
solution.top().y+solution.top().size>table.size())
        {
            throw 0;
        }
        for (int i = square.y; i < square.y + square.size; i++)
        {
            for (int j = square.x; j < square.x + square.size; j++)
            {
                if(table[i][j]==0)
                {
                    throw 0;
                }
                table[i][j] = 0;
            }
        }
        solution.pop();
    }
    catch(int)
    {
        std::cerr<<"error (dellSquare())"<<'\n';
        exit(0);
```

```
    }


}


std::optional<Coordinate> nextCoordinate(const Table& table)
{
    const size_t N = table.size();
    for (size_t i = N / 2 + 1; i < N; i++)
    {
        if (table[N / 2][i] == 0)
        {
            return Coordinate(N / 2, i);
        }
    }
    for (size_t i = N / 2 + 1; i < N; i++)
    {
        for (size_t j = N / 2; j < N; j++)
        {
            if (table[i][j] == 0)
            {
                return Coordinate(i, j);
            }
        }
    }
    return std::nullopt;
}




void Process(Table& table, Solution& tempSolution, Solution& bestSolution)
{
```

```
    StackBacktrack Stack;
    const int startSize=1;
    Coordinate tempCoord = nextCoordinate(table).value();
    Stack.push(Pair(startSize,tempCoord));
    const unsigned short minSizeBestSolution = table.size() +3;
    while (!Stack.empty())
    {
        if (tempSolution.size() >= bestSolution.size()-1 && bestSolution.size()!=0 ||
tempSolution.size()> minSizeBestSolution-1 && bestSolution.size()==0)
        {
            dellSquare(table, tempSolution);
            Stack.pop();
            continue;
        }
        tempCoord = Stack.top().second;
        int sizeSquare = Stack.top().first;
        if (checkPlaceForSquare(tempCoord, sizeSquare, table))
        {
            addSquare(createSquare(tempCoord.first, tempCoord.second, sizeSquare), table,
tempSolution);
            std::optional<Coordinate> coord = nextCoordinate(table);
            if (coord)
            {
                Stack.pop();
                Stack.push(Pair(sizeSquare + 1, tempCoord));
                Stack.push(Pair(startSize, coord.value()));
            }
            else
            {
                bestSolution = tempSolution;
                dellSquare(table, tempSolution);
                dellSquare(table, tempSolution);
                Stack.pop();
            }
        }
```

```
        else
        {
            dellSquare( table, tempSolution);
            Stack.pop();
            continue;
        }
    }
}

Solution evenFillTable(Table& table)
{
    const size_t N = table.size();
    Solution result;
    addSquare(createSquare(0, 0, N / 2), table, result);
    addSquare(createSquare(0, N / 2 , N / 2), table, result);
    addSquare(createSquare(N / 2, 0, N / 2),table, result);
    addSquare(createSquare(N / 2, N / 2, N / 2), table, result);

    return result;
}
Solution multipleOfThreeFillTable(Table& table)
{
    const size_t N = table.size();
    Solution result;
    addSquare(createSquare(0, 0, N * 2 / 3), table, result);
    addSquare(createSquare(0, N * 2 / 3, N * 1 / 3), table, result);
    addSquare(createSquare(N * 1 / 3, N * 2 / 3, N * 1 / 3), table, result);
    addSquare(createSquare(N * 2 / 3, N * 2 / 3, N * 1 / 3), table, result);
    addSquare(createSquare(N * 2 / 3, 0, N * 1 / 3), table, result);
    addSquare(createSquare(N * 2 / 3, N * 1 / 3, N * 1 / 3), table, result);
    return result;
}
Solution multipleOfFiveFillTable(Table& table)
{
    const size_t N = table.size();
```

```
    Solution result;
    addSquare(createSquare(0, 0, N * 3 / 5), table, result);
    addSquare(createSquare(0, N * 3 / 5, N * 2 / 5), table, result);
    addSquare(createSquare(N * 3 / 5, 0, N * 2 / 5), table, result);
    addSquare(createSquare(N * 3 / 5, N * 3 / 5, N * 2 / 5), table, result);
    addSquare(createSquare(N * 2 / 5, N * 3 / 5, N * 1 / 5), table, result);
    addSquare(createSquare(N * 2 / 5, N * 4 / 5, N * 1 / 5), table, result);
    addSquare(createSquare(N * 3 / 5, N * 2 / 5, N * 1 / 5), table, result);
    addSquare(createSquare(N * 4 / 5, N * 2 / 5, N * 1 / 5), table, result);
    return result;
}
Solution initialFill(Table& table)
{
    const size_t N = table.size();
    Solution result;
    addSquare(createSquare(0, 0, N / 2 + 1), table, result);
    addSquare(createSquare(0, N / 2 + 1, N / 2), table, result);
    addSquare(createSquare(N / 2 + 1, 0, N / 2), table, result);
    return result;
}

Table createTableNxN(const unsigned short N)
{
    try
    {
        if(N<2 || N>40)
        {
            throw(0);
        }
        Table table = Table();
        table.resize(N);
        std::for_each(table.begin(), table.end(), [N](std::vector<unsigned short>& i)
            {
                i.resize(N, 0);
            });
```

```
    return table;
  }
  catch(int)
  {
    std::cerr<<"N must be >=2 && <=40 ( createTableNxN() )"<<'\n';
    exit(0);
  }
}
```

tests.cpp

```
#include "includeCatch.hpp"
#include "../Source/header.hpp"


bool operator==(const Square square1,const Square square2)
{
  return square1.size==square2.size && square1.x==square2.x && square1.y==square2.y;
}

TEST_CASE("Test for function createTableNxN","[Creating a Table]")
{
  Table table1 = createTableNxN(5);
  Table table2 = {{0,0,0,0,0},
          {0,0,0,0,0},
          {0,0,0,0,0},
          {0,0,0,0,0},
          {0,0,0,0,0}};
  REQUIRE(table1==table2);
}

TEST_CASE("Test for function addSquare","[Adding a square]")
{
  Table testTable1_1 = {
    {1,1,1,0,0,0},
    {1,1,1,0,0,0},
```

```
    {1,1,1,0,0,0},
    {0,0,0,0,0,0},
    {0,0,0,0,0,0},
    {0,0,0,0,0,0}
};
Solution solution1_1;
solution1_1.push(createSquare(0,0,3));
Solution solution1_2;
Table testTable1_2 = createTableNxN(6);
addSquare(createSquare(0,0,3),testTable1_2,solution1_2);
REQUIRE(solution1_2==solution1_1);//добавление в пустую столешницу
REQUIRE(testTable1_2==testTable1_1);

Table testTable2_1 ={
    {1,1,1,1,1},
    {1,1,1,1,1},
    {1,1,1,0,0},
    {0,0,0,0,0},
    {0,0,0,0,0}
};
Solution solution2_1;
solution2_1.push(createSquare(0,0,3));
solution2_1.push(createSquare(0,3,2));
Table testTable2_2 = {
    {1,1,1,0,0},
    {1,1,1,0,0},
    {1,1,1,0,0},
    {0,0,0,0,0},
    {0,0,0,0,0}
};
Solution solution2_2;
solution2_2.push(createSquare(0,0,3));
addSquare(createSquare(0,3,2),testTable2_2,solution2_2);
REQUIRE(solution2_1==solution2_2);//добавление не в пустую столешницу
REQUIRE(testTable2_2==testTable2_1);
```

```
    Table testTable3_1 ={
        {1,1,1,1,0},
        {1,1,1,1,0},
        {1,1,1,1,0},
        {1,1,1,1,0},
        {0,0,0,0,0}
    };
    Table testTable3_2 = createTableNxN(5);
    Solution solution3_1;
    solution3_1.push(createSquare(0,0,4));
    Solution solution3_2;
    addSquare(createSquare(0,0,4),testTable3_2,solution3_2);
    REQUIRE(solution3_1==solution3_2);//добавление квадрата максимального размера
    REQUIRE(testTable3_1==testTable3_1);
}


TEST_CASE("Test for function delSquare","[deleting a square]")
{
    Table testTable1 = {{0,0,0},
                        {0,0,0},
                        {0,0,0},};
    Table testTable2 = createTableNxN(3);
    Solution solution2;
    addSquare(createSquare(0,0,2),testTable2,solution2);
    dellSquare(testTable2,solution2);//удаляем единственный квадрат
    REQUIRE(testTable1==testTable2);
    REQUIRE(solution2.size()==0);

    Table testTable3 = createTableNxN(3);
    Solution solution3;
    addSquare(createSquare(0,0,2),testTable3,solution3);

    Table testTable4 = createTableNxN(3);
```

```
    Solution solution4;
    addSquare(createSquare(0,0,2),testTable4,solution4);
    addSquare(createSquare(0,2,1),testTable4,solution4);
    dellSquare(testTable4,solution4);//удаляем не единственный квадрат
    REQUIRE(testTable4==testTable3);
    REQUIRE(solution4==solution3);

}


TEST_CASE("Test for function checkPlaceForSquare","[Checking the location]")
{
    Table testTable = createTableNxN(7);
    REQUIRE(checkPlaceForSquare(Coordinate(0,1),3,testTable)==true);//граница сверху
    REQUIRE(checkPlaceForSquare(Coordinate(1,5),2,testTable)==true);//граница справа
    REQUIRE(checkPlaceForSquare(Coordinate(5,1),2,testTable)==true);//граница снизу
    REQUIRE(checkPlaceForSquare(Coordinate(1,0),3,testTable)==true);//граница слева
    REQUIRE(checkPlaceForSquare(Coordinate(2,1),6,testTable)==false);//не входит в
столешницу снизу
    REQUIRE(checkPlaceForSquare(Coordinate(1,2),6,testTable)==false);//не входит в
столешницу справа

    Solution solution;
    addSquare(createSquare(1,1,3),testTable,solution);
    REQUIRE(checkPlaceForSquare(Coordinate(0,0),3,testTable)==false);//место занято другим
квадратом
    REQUIRE(checkPlaceForSquare(Coordinate(4,1),2,testTable)==true);//граничит с
квадратом
}


TEST_CASE("Test for function nextCoordinate","[finding coordinate]")
{
    Table table = createTableNxN(7);
    Solution solution = initialFill(table);
```

```
    auto coord = nextCoordinate(table);
    REQUIRE(coord.value()==Coordinate(3,4));

    addSquare(createSquare(3,4,2),table,solution);
    coord = nextCoordinate(table);
    REQUIRE(coord.value()==Coordinate(3,6));

    table = {{1,1,1},{1,1,1},{1,1,1}};
    coord = nextCoordinate(table);
    REQUIRE(coord == std::nullopt);
}
```