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. 1303 _____Коренев Д.А.

Преподаватель _____Фирсов М.А.

Санкт-Петербург

2023

Цель работы.

Изучить принцип работы бэктрекинга, выполнить задание основываясь на этом алгоритме.

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. Представлена на рисунке 1.

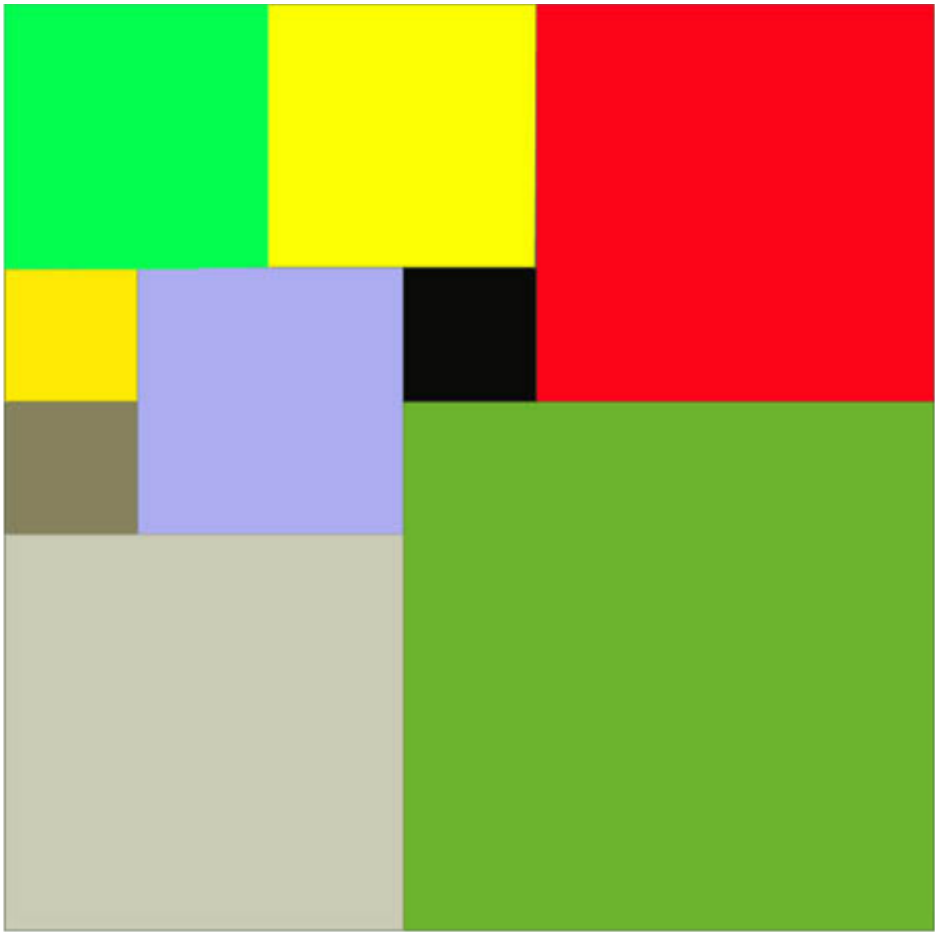

Рисунок 1 - Пример столешницы 7x7

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

Выполняю вариант 3р. Рекурсивный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Выполнение работы.

Весь код программы представлен в приложении А. Для решения задачи был создан класс Table с полям:

- int size – размер столешницы
- int freeArea – свободная площадь столешницы, не заполненная квадратами

- std::vector<std::tuple<int, int, int>> sauarePlaces - кортеж, в котором хранятся тройки координата левого верхнего угла квадрата и его размер

Методы класса Table:

- void insertSquare(int sqSize, int x, int y) – вставляет квадрат со стороной sqSize с координатой левого верхнего угла в точке x:y. Метод не возвращает ничего.
- bool isFree(int x, int y) – рассматривает все квадраты на столешницы и если координата x:y не занята, возвращает true, если занята – false.
- bool isFull() – возвращает true, если столешница заполнена, и false в противном случае.
- void printData() – выводит в консоль количество квадратов внутри столешницы и далее в каждой строке информацию о каждом квадрате, лежащем на столешнице: координаты и размер.
- void print() – выводит в консоль двумерный массив – символьное представление столешницы и квадратов лежащем на ней.
- int getFreeArea() возвращает целочисленное значение - поле freeArea.
- std::vector<std::tuple<int, int, int>> getSquarePlaces() возвращает кортеж трех целочисленных значений поля squarePlaces.

Описание работы алгоритма.

Были созданы глобальные переменные step, boardSize, bestSolutionCountSquare, solution. В функции main считывается значение – размер квадрата в переменную size, вычисляется наибольший делитель size и присваивается переменной boardSize, а результат деления size на

boardSize присваивается в переменную step. Это делается чтобы ускорить решение: достаточно решить задачу для квадрата с меньшей стороной в step раз, и увеличить (масштабировать) результат выполнения программы. Далее создается частичное решение – столешница заполненная тремя квадратами, размерами равными половине стороны столешницы, причем размер одного из них, расположенного в верхнем левом углу, округляется в большую сторону при делении на два. Такая конфигурация была замечена при полном переборе всех решений, и является оптимальной. Пример с нечетной стороной квадрата представлен на рисунке 2, и с четной стороной на рисунке 3.

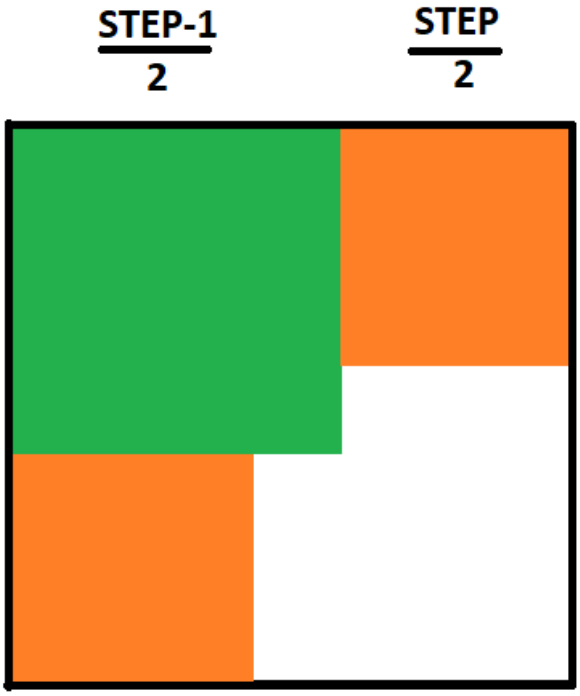

Рисунок 2 частичное решение столешницы с нечетной стороной

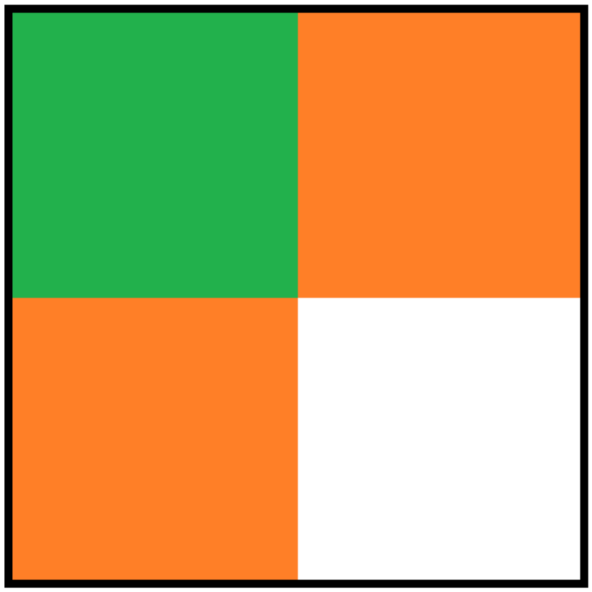

Рисунок 3 частичное решение столешницы с четной стороной

Так как полный перебор всех решений – ресурсозатратный подход, были использованы следующие методы оптимизации:

1. Если количество квадратов частичного решения превышает количество квадратов в имеющемся решении, то процесс нахождения ответа останавливается, так как он заведомо будет хуже.
2. Функция backtracking принимает значения minX и minY, которые ограничивают площадь для поиска свободной координаты.
3. При нахождении свободной координаты высчитывается максимальный размер квадрата, который может поместиться. Таким образом отсеиваются проверки вместимости квадратов большего размера.
4. Переменная freeArea позволяет узнавать свободную площадь столешницы, не высчитывая это значение каждый раз при необходимости

Рассмотрю рекурсивную функцию void backtracking(Table& table, int minX, int minY), где table текущее заполнение столешницы, minX и minY равны значения, меньше которых рассматривать не имеет смысла. Далее в программе используются переменные x, y, maxStepSquare, stepSizeSquare (к ним же относятся и minX, minY), их значение интерпретируется как количество boardSize, как если бы заданная столешница была уменьшена в step раз (см. выше масштабирование).С помощью двух циклов for перебираются значения x от minX до step и значения y от minY до step. Если координата (x*boardSize, y*boardSize) на столешнице не свободна, то переходим к следующей итерации, если же свободна, то просчитывается максимальный размер квадрата, который можно вставить в эту координату.

Сначала в переменную maxStepSquare присваивается минимальное значение из трех: длина максимально возможного квадрата, расстояние до нижнего края столешницы, расстояние до правого края столешницы. Теперь, перебирая все уже вставленные квадраты в столешницы, уменьшается максимальный размер квадрата, если бы он накладывался на соседние квадраты. После этих операций в переменной maxStepSquare храниться максимальный размер квадрата, который точно можно вставить в координат (x:y).

Так как алгоритм backtracking подразумевает перебор всех значений, то и функция backtracking перебирает размер квадрата от максимально возможного (как он находится написано выше), до единицы в цикле for. Чтобы не потерять предыдущее заполнение столешницы, оно копируется в переменную nextTable. В nextTable вставляю квадрат с размером stepSizeSquare (переменная в цикле for) по координате (x*boardSize, y*boardSize). Теперь необходимо проверить, заполнена ли столешница, для этого у объекта класса Table есть метод isFull(). Если он возвращает true, то сравниваются два значения: количество квадратов лучшего решения и

nextTable. Если количество квадратов в nextTable меньше либо равно, то глобальная переменная solution приравнивается nextTable, а переменной solutionCountSquare количество квадратов в новом решении. Для наглядности это решение выводится в консоль. Если же текущая столешница не заполнена и количество квадратов в nextTablt больше чем в лучшем решение, дальнейший перебор останавливается (см. способы оптимизации). Иначе рекурсивно вызывается функция backtracking с аргументами nextTable, x, y + stepSizeSquare. Такие аргументы для x и y выбраны с тем соображением, что по координате x могут быть заполнены не все ячейки столбца (например, при максимально возможном размере квадрата 3, в ходе работы функции был вставлен квадрат размером 2, тогда под этим квадратом пусто). После того как произошел весь перебор размеров квадрата, функция возвращает пустое значение, так как все возможные варианты с конфигурацией, которая поступила на входе, были рассмотрены. После завершения цикла for для y, значение minY делится пополам, так как при смещении вправо (x увеличивается на единицу) столбец может оказаться свободен.

Для теоретической оценки скорости алгоритма, надо проанализировать функцию. Во-первых, идет перебор по всем координатам то есть O(n^2) (где n – размер столешницы). Во-вторых, для каждой координаты рассматриваются возможные размеры квадрата, который можно вставить – выполняется за S операций, где S – количество квадратов в столешнице и, как правило, является небольшим значением. Таким образом получаем оценку скорости работы как O(s^(n^2)), где s – некоторая константа.

Чтобы дать оценку памяти, также проведу анализ. Для оценки памяти следует рассмотреть худший случай: столешница заполнена единичными квадратами. От сюда следует оценка O(n^2), где n – размер столешницы.

Тестирование.

№ п/п	Входные данные	Выходные данные	Комментарий
1	2	4 1 1 1 1 2 1 2 1 1 2 2 1	Верно.
2	3	6 1 1 2 1 3 1 3 1 1 2 3 1 3 2 1 3 3 1	Верно.
3	7	9 1 1 4 1 5 3 5 1 3 4 5 1 4 6 2 5 4 1 5 5 1 6 4 2 6 6 2	Верно.
4	10	4 1 1 5 1 6 5 6 1 5	Верно.

		6 6 5	
5	17	12 1 1 9 1 10 8 10 1 8 9 10 1 9 11 3 9 14 4 10 9 2 12 9 2 12 11 2 12 13 1 13 13 5 14 9 4	Верно.
6	18	4 1 1 9 1 10 9 10 1 9 10 10 9	Верно.
7	20	4 1 1 10 1 11 10 11 1 10 11 11 10	Верно.
8	23	13 1 1 12 1 13 11 13 1 11 12 13 1	Верно.

		12 14 3 12 17 7 13 12 2 15 12 5 19 17 2 19 19 5 20 12 4 20 16 1 21 16 3	
9	37	15 1 1 19 1 20 18 20 1 18 19 20 1 19 21 3 19 24 7 19 31 7 20 19 2 22 19 5 26 24 2 26 26 12 27 19 4 27 23 1 28 23 3 31 19 7	Верно.

Исследование.

Исследование количества операций в зависимости от размера столешницы. За одну операцию было принята вставка квадрата. Так как для всех столешниц четного размера единственное решение: 4 квадрата

одинакового размера, то исследовать стоит квадраты с нечетной длинной стороны, график представлен на рисунке 4.

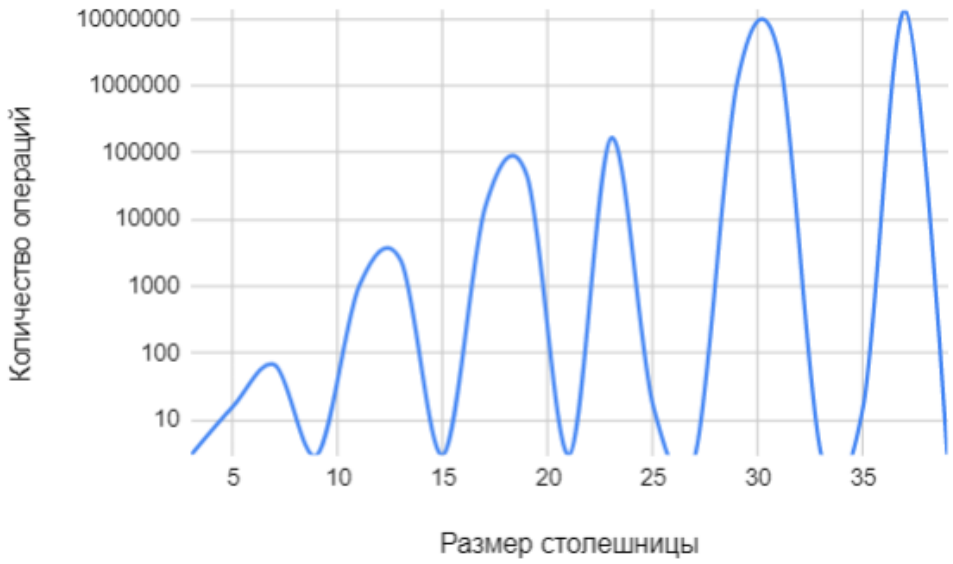

Рисунок 4 зависимость количества вызова функции от размера столешницы

Данные, по которым строился данный график:

Y	3	16	66	3	1010	2378	3	15307	44241	3
X	3	5	7	9	11	13	15	17	19	21
Y	166364	16	3	1160409	2798016	3	16	1344961	3	
X	23	25	27	29	31	33	35	37	39	

Заметны «всплески» количества операций, это происходит, когда длина столешницы – простое число, так как при других значениях решение сводится к более простому решению. Рассмотрим график, в построении которого участвуют только простые числа, представлен на рисунке 5.

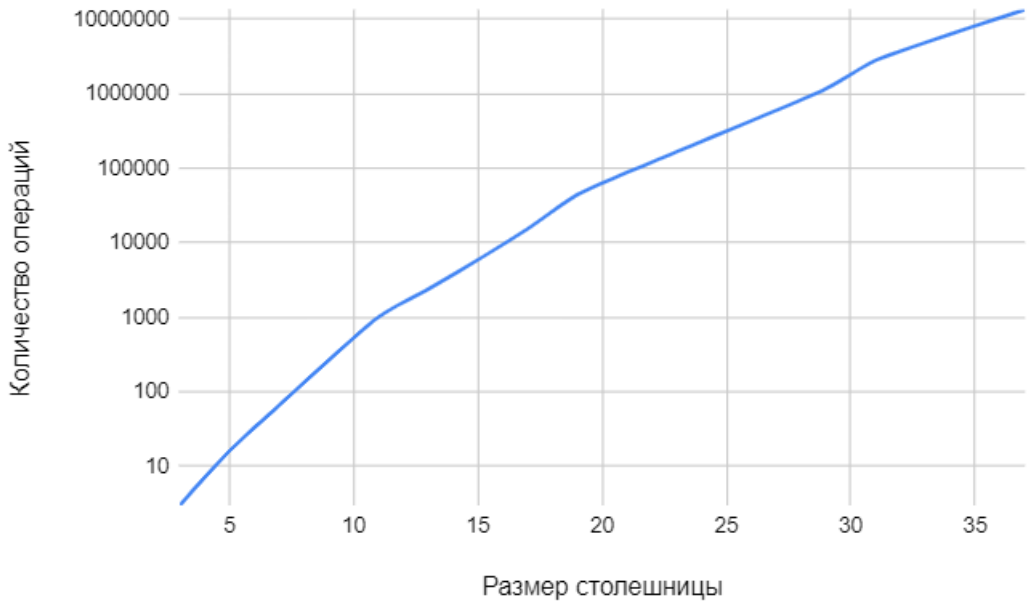

Рисунок 5 зависимость количества вызова функции от размера столешницы (простые числа)

Данные, по которым строился данный график:

Y	3	16	66	1010	12378	15397	44241	166354	1160409	2798016	13443961
X	3	5	7	11	13	17	19	23	29	31	37

Анализируя данные, видно, что рост количества операций экспоненциальный. То есть сложность алгоритма backtracking может быть оценена как O(к^n), где k – константа (например экспонента), n – размер столешницы. Оценка несколько отличается от аналитической оценки алгоритма (напомню, она равна O(k^n^2)), это происходит из-за того что в функции применены методы оптимизации, значительно уменьшающие глубину рекурсии, и, как следствие, ресурс затратность алгоритма.

Вывод.

В ходе выполнения работы был изучен подход к решению задач «Поиск с возвратом» и была решена задача, решение которой возможно лишь применяя данный подход. Была разработана программа, решающая поставленную задачу путём перебора всех возможных вариаций, а также использующая ряд оптимизаций для отсечения заведомо худших решений.

Проведено исследование, направленное на изучение зависимости количества операций от размера стола, был сделан вывод о скорости алгоритма.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <tuple>
#include <valarray>


class Table{
private:
    int size = 0;
    int freeArea = 0;
    //x y w
    std::vector<std::tuple<int, int, int>> squarePlaces;
public:
    //иницилизация полей в констркуторе
    Table(int size): size(size){
        freeArea = size*size;
    }

    //вставка в кортеж нового квадрата
    void insertSquare(int sqSize, int x, int y) {
        freeArea -= sqSize * sqSize;
        squarePlaces.emplace_back( x, y, sqSize);
    }

    //проверка свободна ли координата
    bool isFree(int x, int y){
        if (x < 0 or y < 0 or x > size-1 or y > size-1) return
false;
        for (std::tuple<int, int, int> csq: squarePlaces){
            int sqX = std::get<0>(csq);
            int sqY = std::get<1>(csq);
            int sqW = std::get<2>(csq);

            if ((sqX <= x) and (x < sqX+sqW) and (sqY <= y) and (y <
sqY+sqW)){
                return false;
            }
        }
        return true;
    }

    // вернет true если свободная площадь отсутствует
    bool isFull(){
        return (freeArea == 0);
    }

    //возвращает размер кортежа
    int getAmountSquare(){
        return int(squarePlaces.size());
    }

    //возвращает значение поля freeArea
    int getFreeArea(){
        return freeArea;
```

```
    }

    //возвращает кортеж squarePlaces
    std::vector<std::tuple<int, int, int>> getSquarePlaces(){
        return squarePlaces;
    }

    //выводит данне о квадратах
    void printData(){
        std::cout << squarePlaces.size() << "\n";
        for (auto csq: squarePlaces){
            int sqX = std::get<0>(csq);
            int sqY = std::get<1>(csq);
            int sqW = std::get<2>(csq);

            std::cout << sqX+1 << " " << sqY+1 << " " << sqW <<
"\n";
        }
    }

    //создается двумерный массив - столешница
    //заполняется на основании квадратов в столешнице
    void print(){
        std::vector<std::vector<int>> table;
        for (int row = 0; row < size; ++row){
            table.emplace_back();
            for (int column = 0; column < size; ++column){
                table.at(row).push_back(0);
            }
        }

        for (std::tuple<int, int, int> csq: squarePlaces) {
            int sqX = std::get<0>(csq);
            int sqY = std::get<1>(csq);
            int sqW = std::get<2>(csq);

            for (int x = sqX; x < sqX + sqW; ++x) {
                for (int y = sqY; y < sqY + sqW; ++y) {
                    table.at(y).at(x) = sqW;
                }
            }
        }

        bool flagSize = table.at(0).at(0) > 9;
        for (int y = 0; y < size; ++y){
            for (int x = 0; x < size; ++x){
                int value = table.at(y).at(x);
                if (value < 10 and flagSize) std::cout << " ";
                std::cout << value << " ";
            }
            std::cout << "\n";
        }
    }
};


int step;
```

```
int boardSize;
int solutionCountSquare;
Table solution(0);
int countBacktracking = 0;

void backtracking(Table& table, int minX, int minY){
    ++countBacktracking;
    for (int x = minX; x < step; ++x){
        for (int y = minY; y < step; ++y){
            // X and y измеряются в штуках (чанках) т.е. количестве
boardsize
            //если можно вставить в координату
            if (table.isFree(x*boardSize, y*boardSize)){
                //std::cout << "x = " << x << " y = " << y << "\n";

                //maxStepSquare измеряется в штуках (чанках)
boardsize
                //так как step это максимальное кол-во штук (чанков)
boardsize, то берем на один меньше (нельзя квадрат равный столу)
                //step-x это длина в штуках которая вместится от
координаты x до правого края
                //step-y это длина в штуках которая вместится от
координаты y до низа стола
                //выбираем их этого наименьшее, чтобы вмещалось
                int maxStepSquare = std::min(step - 1, std::min(step
- x, step - y));
                //std::cout << "r first = " << maxStepSquare <<
'\n';

                //для каждого вставленного квадрата рассматриваем
случай
                for (std::tuple<int, int, int> square:
table.getSquarePlaces()){
                    int sqX = std::get<0>(square);
                    int sqY = std::get<1>(square);
                    int sqW = std::get<2>(square);

                    //если нашелся квадрат который ниже по
координате y и "налазит" или превышает по координате x на текущую
координату (x;y)
                    if (sqX+sqW > x*boardSize && sqY > y*boardSize){
                        //Надо уменьшить сторону вставляемого
квадрата до максимально возможного
                        //для этого надо рассмотреть предыдущий
вариант и
                        //т.к. снизу есть квадрат (см. условие по y)
то надо выбрать размер, чтобы новый квадрат не налез на этот нижний
                        maxStepSquare = std::min(maxStepSquare, sqY
/ boardSize - y);
                    }
                }

                //перебрать все размеры квадрата бэктрекингом
                for (int stepSizeSquare = maxStepSquare;
stepSizeSquare > 0; --stepSizeSquare) {
                    //чтобы не менять исходный стол, создать новый
                    Table nextTable = table;
```

```
                    //зная максимальный возможный размер
вставляемого квадрата будем вставлять его в новый стол
                    //причем уменьшать его сторону на один (цикл
for), чтобы сделать перебор значений
                    nextTable.insertSquare(stepSizeSquare *
boardSize, x * boardSize, y * boardSize);

                    //если стол максимально заполнен, надо сделать
проверку на лучший результат
                    if (nextTable.isFull()) {
                        //если в нем количество квадратов меньше чем
в лучшем результате
                        if (nextTable.getAmountSquare() <=
solutionCountSquare) {
                            //сделать замену количества лучшего
результата и решения
                            solutionCountSquare =
nextTable.getAmountSquare();
                            solution = nextTable;
                            std::cout << "New best solution: total
squares " << solutionCountSquare << "\n";
                            solution.print();
                        }
                    } else {
                        // Если стол заполнен не до конца, но в нем
УЖЕ больше квадратов чем в лучшем решении
                        // То заканчиваем перебор, т.к. лучшее
решение из этого не получить
                        if (nextTable.getAmountSquare() <=
solutionCountSquare) {
                            //если же еще есть смысл дополнять,
запускам бектрекинг от полученного стола
                            //с аргументами x (так как "левее" уже
все заполнилось в главном цикле for)
                            //и y + stepSizeSquare так как вставили
квадрат в координату y и размером stepSizeSquare
                            std::cout << "\nInserted square w = "
                                      "" <<
stepSizeSquare*boardSize << ""
                                      " in x = " << x*boardSize
<< ""
                                      "y = " << y*boardSize <<
"\n";
                            nextTable.print();
                            std::cout << "Start backtracking x = "
<< x << " y = " << y+stepSizeSquare << "\n";
                            backtracking(nextTable, x, y +
stepSizeSquare);
                        } else{
                            std::cout << "Backtracking stopped.
Excess squares. Total squares = "
                                      "" <<
nextTable.getAmountSquare() << " but best solution = "
                                      "" << solutionCountSquare
<< '\n';
                            return;
                        }
```

```
                    }
                }
                std::cout << "All version was checked, backtracking
return\n";
                return;
            }
        }
        //так как заполнение идет по столбцам, а уже потом по
рядам(цикл for y внутренний)
        //то после того как мы посмотрели текущий столбец, следующий
следует начинать рассматривать выше
        minY = step/2;
    }
}

int main() {
    int size;
    std::cin >> size;
    // поиск наибольшего делителя size
    for (int i = 1; i < size; ++i){
        if (size%i == 0) boardSize = i;
    }
    step = size/boardSize; // => step*boardSize = size of table
    // начальное приближение это такое приближение,
    // когда весь стол можно заполнить одинаковыми
    // квадратами размера boardsize, по ширине и высоте их по step
штук

    Table table(size);
    table.insertSquare(((step+1)/2)*boardSize, 0, 0);
    table.insertSquare((step/2)*boardSize, 0,
((step+1)/2)*boardSize);
    table.insertSquare((step/2)*boardSize, ((step+1)/2)*boardSize,
0);
    solutionCountSquare = 3 + table.getFreeArea(); // 3 + остальные
единицы

    //backtracking
    backtracking(table, step/2, (step+1)/2);

    //result
    solution.printData();
    std::cout << "\n";
    solution.print();
    std::cout << "\n" << countBacktracking << '\n';

    return 0;
}
```