**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 |  | Чубан Д.В. |
| --- | --- | --- |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург

2023

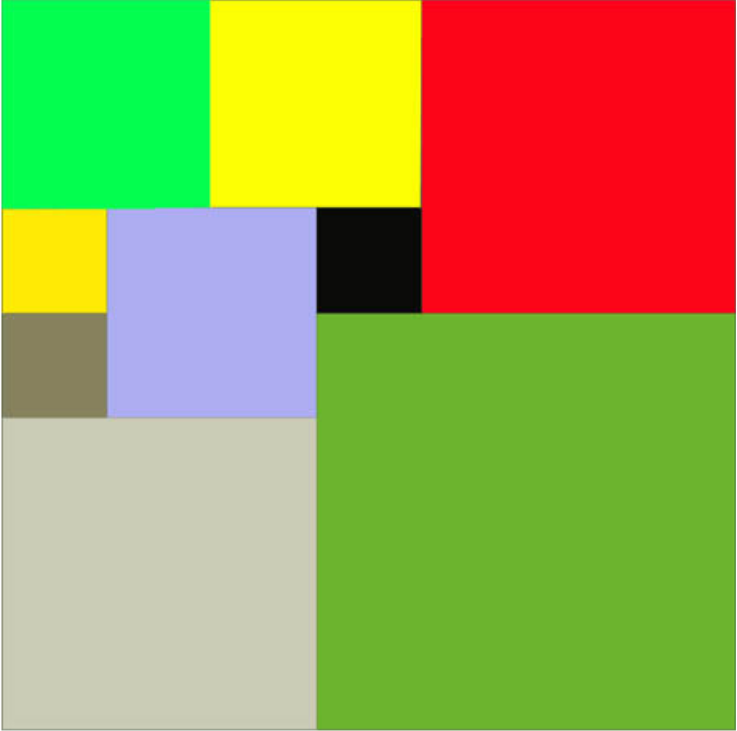
**Цель работы.**

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 *≤ 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).

**﻿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((n2)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(n2), 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 указатели на объекты Square

• sizeLen – 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.

• Desk(const Desk& obj) – конструктор копирования. В качестве аргумента принимает объект данного класса по ссылке и конструирует новый объект, значения полей которого совпадают со значениями полей аргумента.

• Desk& operator=(const Desk& obj) - перегружен оператор присваивания с копированием.

• void setDefault() – метод, необходимый для расстановки квадратов, в случае, если длина стороны не кратна 2 или 3. Примененная оптимизация.

• 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 – соответственно 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 такого же класса.

Примененные оптимизации

В случае, если размер стола не кратен 2 или 3, заранее расставляются 3 квадрата, что существенно сокращает перебор.

Кроме того, новый квадрат устанавливается в максимально верхнюю

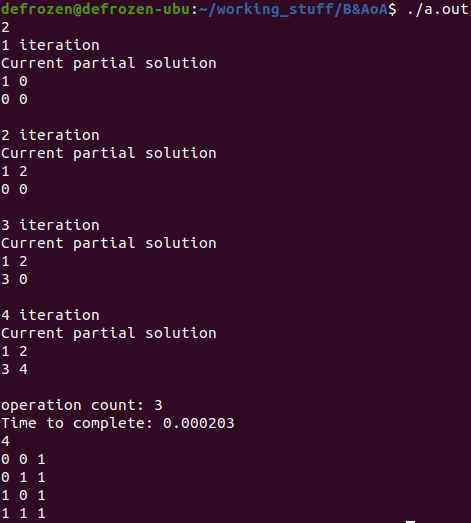
левую клетку, таким образом сокращается кол-во расстановок, т.к.

отбрасываются одинаковые расстановки, но с разным порядком размещения квадратов.

Так как алгоритм при нахождении ответа в первую очередь расставляет самые большие квадраты, то расстановка, которую мы примем за итоговую будет найдена первой. Данное рассуждение позволяет нам не просчитывать все возможные варианты, а прекратить выполнение алгоритма при нахождении первой расстановки, полностью заполняющей поле, что позволяет сократить общее количество выполняемых операций.

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). В случае такого ввода конструктор должен пробрасывать исключение.

Также проверена корректность работы алгоритма бэктрекинга. Для всех возможных размеров из промежутка 2..10 вручную было найдено минимальное необходимое кол-во квадратов, данные значения сравниваются с выводом функции Desk backtracking(Desk desk).

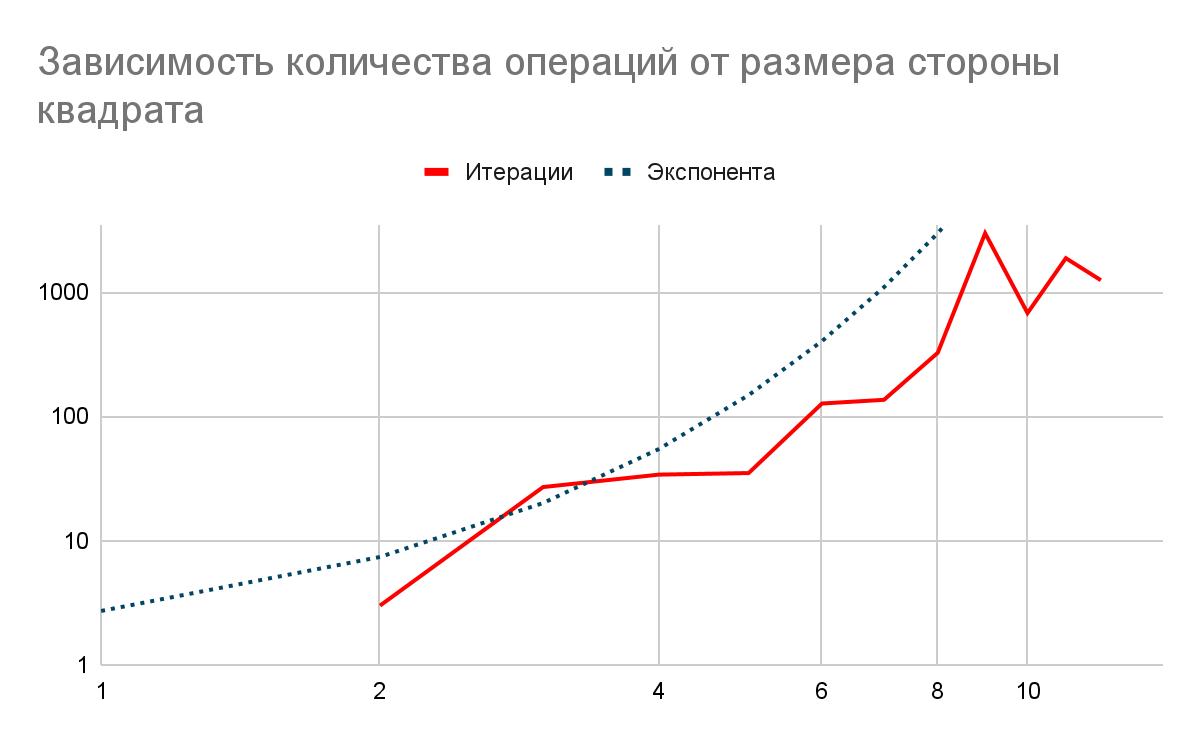
| **Ввод** | **Вывод** | **Ожидаемый результат** |
| --- | --- | --- |
| 2 | 4  0 0 1  0 1 1  1 0 1  1 1 1 | 4  0 0 1  0 1 1  1 0 1  1 1 1 |
| 3 | 6  0 0 2  0 2 1  1 2 1  2 0 1  2 1 1  2 2 1 | 6  0 0 2  0 2 1  1 2 1  2 0 1  2 1 1  2 2 1 |
| 4 | 4  0 0 2  0 2 2  2 0 2  2 2 2 | 4  0 0 2  0 2 2  2 0 2  2 2 2 |
| 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 | 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 |
| 6 | 4  0 0 3  0 3 3  3 0 3  3 3 3 | 4  0 0 3  0 3 3  3 0 3  3 3 3 |
| 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 | 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 |
| 8 | 4  0 0 4  0 4 4  4 0 4  4 4 4 | 4  0 0 4  0 4 4  4 0 4  4 4 4 |
| 9 | 6  0 0 6  0 6 3  3 6 3  6 0 3  6 3 3  6 6 3 | 6  0 0 6  0 6 3  3 6 3  6 0 3  6 3 3  6 6 3 |
| 10 | 4  0 0 5  0 5 5  5 0 5  5 5 5 | 4  0 0 5  0 5 5  5 0 5  5 5 5 |
| 1 | inappropriate size | inappropriate size |
| 21 | inappropriate size | inappropriate size |

**Исследование.**

Элементарной операцией является одна попытка постановки квадрата.

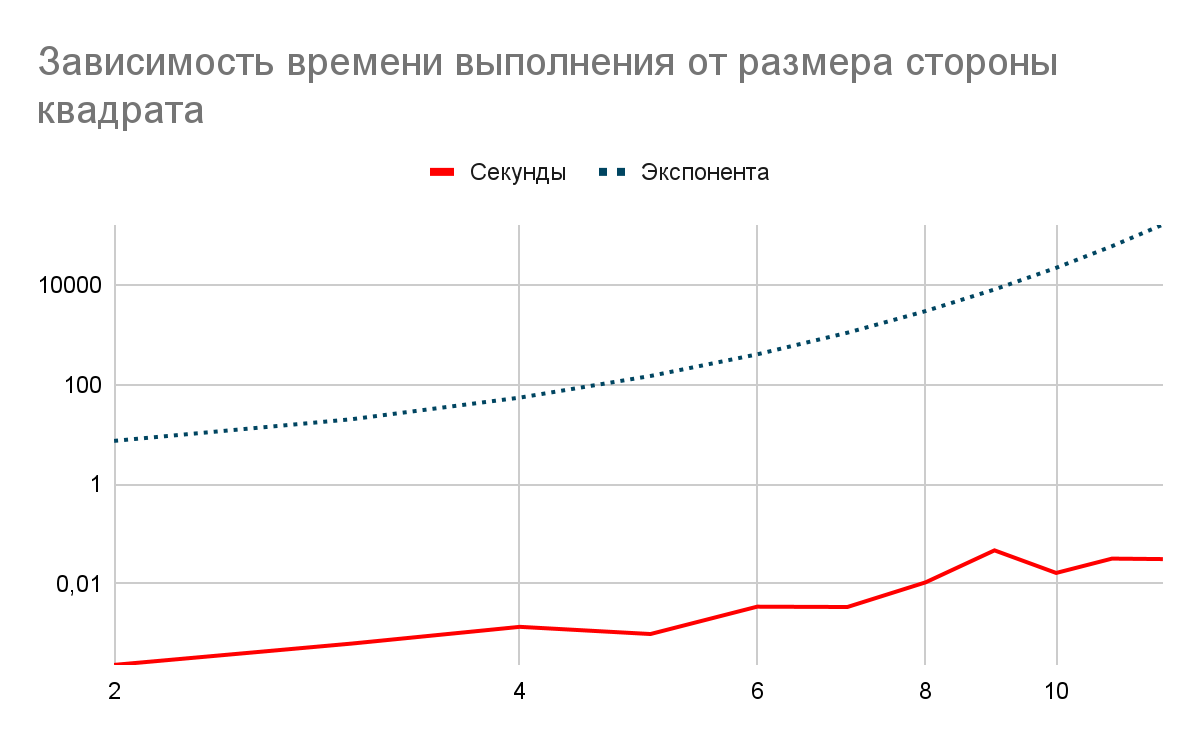
| Размер | Время, с | Операции | Результат (кол-во квадратов) |
| --- | --- | --- | --- |
| 2 | 0.000235 | 3 | 4 |
| 3 | 0.000629 | 27 | 6 |
| 4 | 0.001365 | 34 | 4 |
| 5 | 0.000979 | 35 | 8 |
| 6 | 0.003467 | 127 | 4 |
| 7 | 0.003415 | 136 | 9 |
| 8 | 0.010669 | 325 | 4 |
| 9 | 0.046838 | 2981 | 6 |
| 10 | 0.016486 | 679 | 4 |
| 11 | 0.031971 | 1877 | 11 |
| 12 | 0.031294 | 1248 | 4 |

Построим логарифмический график по полученным данным для количества операций:

**

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.cpp

#include <iostream>

#include <queue>

#include <vector>

#include <time.h>

struct Square{

int size; // длина стороны квадрата

int x; // координаты по x и y

int y;

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

friend std::ostream& operator<<(std::ostream& os, const Square& square){

os << square.x <<' '<< square.y << ' ' << square.size << std::endl;

return os;

}

};

class Desk{

private:

std::vector<std::vector<int>> map; // двумерный массив, "отображающий" расположение квадратов

int sizeLen; // длина стороны стола

int squareCounter; // количество квадратов

std::vector<Square\*> squareList; // список характеристик квадратов, выводимый в конце

public:

Desk(int n):sizeLen(n){ // конструктор класса

if(n < 2 || n > 20) throw std::invalid\_argument("inappropriate size");

this->map = std::vector<std::vector<int>>();

for(auto i = 0; i < sizeLen; i++){

this->map.push\_back(std::vector<int>(n));

}

this->squareList = std::vector<Square\*>();

this->squareCounter = 0;

if(n % 2 != 0 && n % 3 != 0){

this->setDefault();

}

}

Desk(const Desk& obj):sizeLen(obj.sizeLen), squareCounter(obj.squareCounter){ //оператор копирования

this->map = std::vector<std::vector<int>>();

for(auto i = 0; i < sizeLen;i++){

this->map.push\_back(std::vector<int>(this->sizeLen));

}

for(auto i = 0; i < sizeLen; i++){

for(auto j = 0; j < sizeLen; j++){

this->map[i][j] = obj.map[i][j];

}

}

this->squareList = std::vector<Square\*>(obj.squareList);

}

Desk& operator=(const Desk& obj){//оператор присваивания

if(this != &obj){

this->map = std::vector<std::vector<int>>();

for(auto i = 0; i < sizeLen;i++){

this->map.push\_back(std::vector<int>(this->sizeLen));

}

for(auto i = 0; i < sizeLen; i++){

for(auto j = 0; j < sizeLen; j++){

this->map[i][j] = obj.map[i][j];

}

}

sizeLen = obj.sizeLen;

squareCounter = obj.squareCounter;

squareList = std::vector<Square\*>(obj.squareList);

}

return \*this;

}

void setDefault(){ //функция оптимизации для столов со сторонами, кратными 2 или 3, заранее ставит 3 квадрата, уменьшая количество операций

this->addSquare((sizeLen+1)/2, 0, 0, squareCounter);

this->addSquare((sizeLen-1)/2, 0, (sizeLen+1)/2, squareCounter);

this->addSquare((sizeLen-1)/2, (sizeLen+1)/2, 0, squareCounter);

}

void addSquare(int size, int x, int y, int color){ //добавление квадрата на стол

Square\* square = new Square(size, x, y);

for(auto i = x; i < x + size; i++){

for(auto j = y; j < y + size; j++){

this->map[i][j] = color+1;

}

}

this->squareList.push\_back(square);

this->squareCounter++;

}

bool isFull(){ //проверка на полную заполненность стола

for (int i = 0; i < sizeLen; i++)

for (int j = 0; j < sizeLen; j++)

if (map[i][j] == 0)

return false;

return true;

}

bool canAdd(int x, int y, int size){ //проверка на возможность добавления квадрата заданных характеристик на стол

if(y+size > this->sizeLen || x+size > this->sizeLen)

return false;

for(auto i = x; i < x+size; i++){

for(auto j = y; j < y+size; j++){

if(this->map[i][j] != 0)

return false;

}

}

return true;

}

std::pair<int, int> emptyCell(){ // нахождение пустой клетки на столе

for(auto i = 0; i < this->sizeLen; i++){

for(auto j = 0; j < this->sizeLen; j++){

if(this->map[i][j] == 0) return std::pair<int, int>{i, j};

}

}

return std::pair<int, int>{-1, -1};

}

int getSideLen() const{

return sizeLen;

}

std::vector<Square\*> getSquareList() const{

return squareList;

}

int getSquareCount() const{

return squareCounter;

}

std::vector<std::vector<int>> getMap() const{

return this->map;

}

friend std::ostream& operator<<(std::ostream& os, const Desk& desk){ // перегрузка оператора вывода для класса Desk

for(auto& elem: desk.getMap()){

for(auto& cell:elem){

os << cell << " ";

}

os << '\n';

}

return os;

}

};

Desk backtracking(Desk desk){

std::queue<Desk> queue = std::queue<Desk>();

queue.push(desk); // кладем исходный стол в очередь

int k = 1, operationCount = 0; // инициализация счетчиков для количества операций и итераций

while(!queue.front().isFull()){

std::cout << k << " " << "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.getSideLen() - 1; i > 0 ; i--){ // в этом цикле мы 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: " << operationCount << std::endl; //

return cur;

}

queue.push(cur); // если стол был заполнен не до конца, полученный этап решения кладем в конец очереди и идем дальше

}

operationCount++;

}

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

k++;

}

std::cout <<"operation count: " << operationCount << std::endl;

return queue.front();

}

int main(){

int n;

std::cin>>n;

try{

Desk desk = Desk(n);

auto start = clock();

Desk answer = backtracking(desk);

std::cout << "Time to complete: "<< (double)(clock() - start)/CLOCKS\_PER\_SEC << std::endl;

std::cout << answer.getSquareCount() << std::endl;

for(auto elem:answer.getSquareList()){

std::cout << \*elem;

}

}catch(const std::exception& e){

std::cerr<<e.what() << std::endl;

}

return 0;

}