

**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. 9382

\_\_\_\_\_

Юрьев С.Ю.

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2021

### **Цель работы.**

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным количеством меньших квадратов.

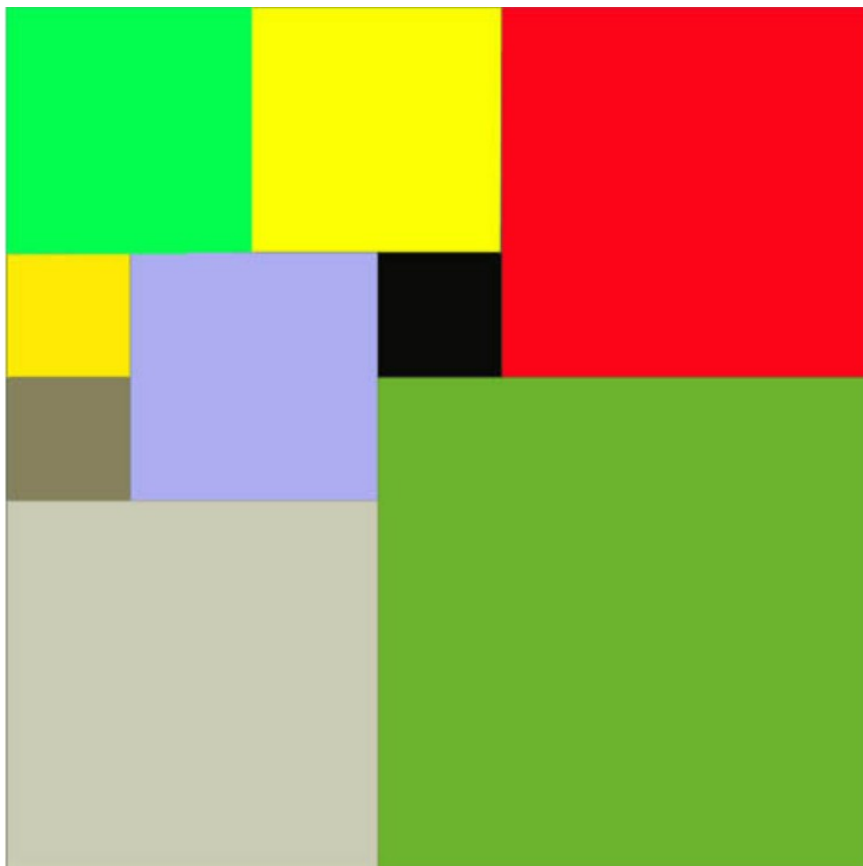
### **Вариант 3р.**

Рекурсивный бэктрекинг. Исследование количества операций от размера квадрата.

### **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  $7 \times 7$  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  $N$  ( $2 \leq N \leq 20$ ).

#### **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 ( $1 \leq x, y \leq N$ ) 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

#### **Теоретические сведения.**

**Бэктрекинг** (поиск с возвратом) – это 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 удастся, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

### **Описание алгоритма.**

Цель алгоритма — найти наименьшее количество квадратов, которыми можно заполнить исходный квадрат со стороной  $N$ .

Сам алгоритм заключается в следующем:

- 1) Создание структур для хранения промежуточной информации и информации о лучшей попытке заполнения начального квадрата.
- 2) Вставка 3 начальных квадратов, заполняющих 75% площади начального квадрата.
- 3) Рекурсивный перебор всех возможных заполнений свободного пространства. Для этого идет циклический поиск незанятого пространства, после чего идет проверка на возможность вставить сюда квадрат определенного размера, после чего продолжается дальнейший поиск свободного места для вставки следующих квадратов.
- 4) При нахождении заполнения свободного пространства меньшим количеством квадратов следует сохранение результата, как наилучшего. При нахождении заполнения таким же количеством квадратов или большим сохранение не производится.

### **Иные оптимизации:**

- Если  $N$  кратно 2, то минимальное разбиение всегда будет состоять из 4 равных частей.

- Если  $N$  кратно 3 или 5, то будет произведено сжатие квадрата для уменьшения количества вычислений.

### **Оценка сложности.**

В алгоритме используется начальный квадрат размера  $N \times N$  и другие переменные, зависящие от  $N$ , но они не дают весомого вклада в увеличение сложности по памяти. Поэтому сложность алгоритма по памяти =  $O(N^2)$ , где  $N$  — размер исходного квадрата.

В исходном квадрате  $N \times N$  свободных клеток, количество размеров квадратов которые будут перебираться  $N$ . Место для первого квадрата можно выбрать  $N^2 \times N$  способами. Для второго  $(N^2 - 1) \times N$  способами. Таким образом, сложность алгоритма по времени =  $O((N^2)! \times N^N)$ , где  $N$  - размер исходного квадрата.

### **Исследование.**

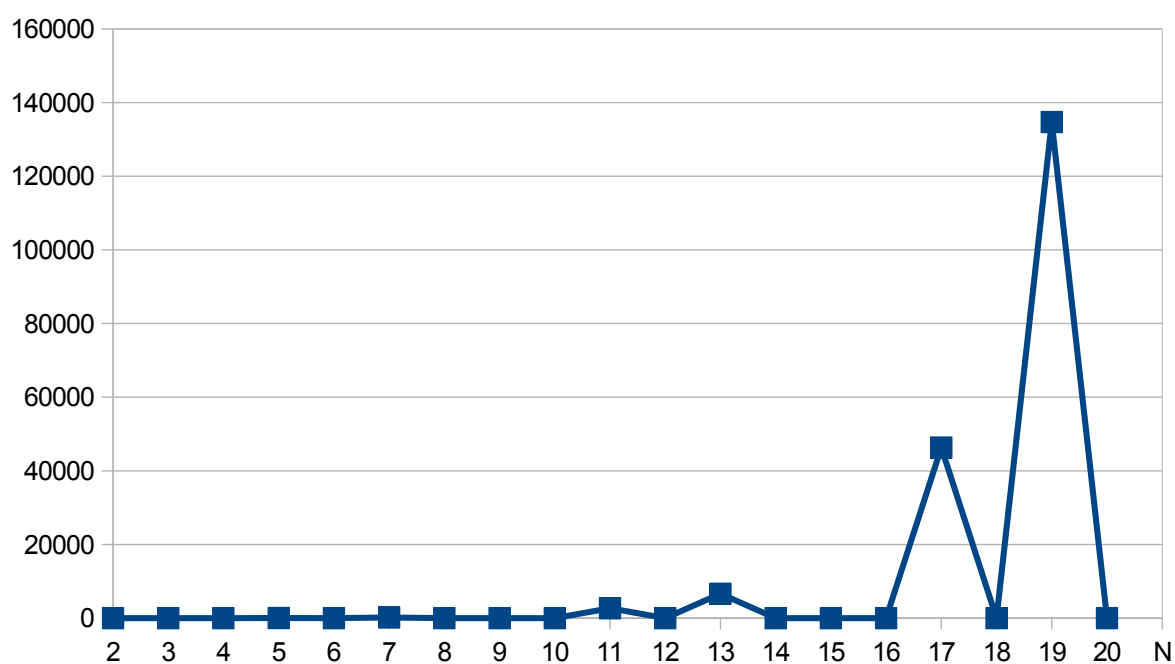
Исследование количества операций от размера квадрата. Результаты измерения количества операций в зависимости от размера квадрата представлены в таблице 1.

Таблица 1.

Сторона квадрата, $N$	Общее количество операций
2	9
3	17
4	9
5	61
6	9
7	202
8	9

9	17
10	9
11	2685
12	9
13	6620
14	9
15	17
16	9
17	46342
18	9
19	134705
20	9

График 1.



По графику 1 можно увидеть, что за исключением случаев, когда используются оптимизации, количество операций растет экспоненциально.

### Тестирование.

Ввод	Вывод
2	4  1 1 1  1 2 1  2 1 1  2 2 1
3	6  1 1 2  1 3 1  3 1 1  3 2 1  2 3 1  3 3 1

7	9 1 1 4 1 5 3 5 1 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2
9	6 1 1 6 1 7 3 7 1 3 7 4 3 4 7 3 7 7 3



20	4 1 1 10 1 11 10 11 1 10 11 11 10
----	---

### **Выводы.**

В ходе выполнения данной лабораторной работы была написана программа, реализующая алгоритм поиска с возвратом рекурсивно.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <ctime>

// строчку ниже можно комментировать/раскомментировать для вклю-
// чения/отключения промежуточного вывода
// #define ADDINFO

// отвечает за подсчет и вывод количества выполненных операций
// #define COUNTINFO

#ifdef COUNTINFO
int operationsCount = 0;
#endif

class Square // класс НЕ "видимых" квадратов
{
public:
    int x;
    int y;
    int size; // длина стороны квадрата
};

bool isPossibleToAddSquare(std::vector<std::vector<int>>& visSqr, int x, int
y, int sizeOfSmallSqr)
{
    #ifdef COUNTINFO
    ++operationsCount;
    #endif

    // проверка на выход за пределы квадрата
    if ((x + sizeOfSmallSqr) > visSqr.size() || (y + sizeOfSmallSqr) >
visSqr.size())
    {
        return false;
    }

    // проверка пустоты выбранной области
    for (int i = y; i < y + sizeOfSmallSqr; i++)
    {
        for (int j = x; j < x + sizeOfSmallSqr; j++)
        {
            if (visSqr[i][j])
            {
                return false;
            }
        }
    }
    return true;
}
```

```
}
```

```
void addToVisibleSquare(std::vector<std::vector<int>>& visSqr, int x, int y,  
int sizeOfSmallSqr) // не делает проверок, сразу "красит"
```

```
{
```

```
    #ifdef COUNTINFO  
    ++operationsCount;  
    #endif
```

```
    for (int i = y; i < y + sizeOfSmallSqr; i++)  
    {  
        for (int j = x; j < x + sizeOfSmallSqr; j++)  
        {  
            visSqr[i][j] = sizeOfSmallSqr;  
        }  
    }  
}
```

```
}
```

```
void printVisibleSquare(std::vector <std::vector <int>>& visSqr, int  
compression, int sideLenOfMainSquare)
```

```
{
```

```
    for (int i = 0; i < sideLenOfMainSquare * compression; i++)  
    {  
        for (int j = 0; j < sideLenOfMainSquare * compression; j++)  
        {  
            std::cout.width(3); // для красивого вывода  
            std::cout << visSqr[i][j];  
        }  
        std::cout << std::endl;
```

```
    }  
}
```

```
}
```

```
void fillVisibleSquareWithZeros(std::vector<std::vector<int>>& visSqr, int  
sideLenOfMainSquare) // заполнен нулями == не поставлено ни одного  
квадрата
```

```
{
```

```
    #ifdef COUNTINFO  
    ++operationsCount;  
    #endif
```

```
    visSqr.resize(sideLenOfMainSquare);  
    for (int i = 0; i < sideLenOfMainSquare; i++)  
    {  
        visSqr[i].resize(sideLenOfMainSquare);  
        for (int j = 0; j < sideLenOfMainSquare; j++)  
        {  
            visSqr[i][j] = 0;  
        }  
    }  
}
```

```
}
```

```

void delLastAddedSquare(std::vector<std::vector<int>>& visSqr,
std::vector<Square>& currentArrOfAddedSquares, int spacesCount)
{
    #ifdef COUNTINFO
        ++operationsCount;
    #endif

    Square removableSquare = currentArrOfAddedSquares.back();
    currentArrOfAddedSquares.pop_back();

    // "обнуление" ранее занятых клеток в видимом квадрате
    for (int i = removableSquare.y; i < removableSquare.y +
removableSquare.size; i++)
    {
        for (int j = removableSquare.x; j < removableSquare.x +
removableSquare.size; j++)
        {
            visSqr[i][j] = 0;
        }
    }

    #ifdef ADDINFO
        for(int l = 0; l < spacesCount; l++)
        {
            std::cout << " ";
        }
        std::cout << "Удаление кв. со стороной " << removableSquare.size <<
" (x = " << removableSquare.x + 1 << ", y = " << removableSquare.y + 1
<< ')' << std::endl;
    #endif
}

```

```

void makeTaskPreparations(int &compression,
std::vector<std::vector<int>>& visSqr, std::vector<Square>&
currentArrOfAddedSquares, int& freeAreaOfMainSquare, int&
sideLenOfMainSquare, int& bestSquaresCount)
{
    #ifdef COUNTINFO
        ++operationsCount;
    #endif

    fillVisibleSquareWithZeros(visSqr, sideLenOfMainSquare); // инициа-
лизация пустого "видимого" квадрата

    // "сжатие" квадрата, чтобы облегчить дальнейшие расчеты
    if (sideLenOfMainSquare % 2 == 0)
    {
        compression = sideLenOfMainSquare / 2;
        sideLenOfMainSquare = 2;
    }
    else if (sideLenOfMainSquare % 3 == 0)
    {

```

```

        compression = sideLenOfMainSquare / 3;
        sideLenOfMainSquare = 3;
    }
    else if (sideLenOfMainSquare % 5 == 0)
    {
        compression = sideLenOfMainSquare / 5;
        sideLenOfMainSquare = 5;
    }
    // так можно продолжать для всех простых чисел

    bestSquaresCount = 2 * sideLenOfMainSquare + 1;

    // выставление первых трех квадратов
    currentArrOfAddedSquares.push_back({ 0, 0, (sideLenOfMainSquare + 1) /
2 });
    currentArrOfAddedSquares.push_back({ 0, (sideLenOfMainSquare + 1) / 2,
sideLenOfMainSquare / 2 });
    currentArrOfAddedSquares.push_back({ (sideLenOfMainSquare + 1) / 2, 0,
sideLenOfMainSquare / 2 });

    addToVisibleSquare(visSqr, 0, 0, (sideLenOfMainSquare + 1) / 2);
    addToVisibleSquare(visSqr, 0, (sideLenOfMainSquare + 1) / 2,
sideLenOfMainSquare / 2);
    addToVisibleSquare(visSqr, (sideLenOfMainSquare + 1) / 2, 0,
sideLenOfMainSquare / 2);

    #ifdef ADDINFO
        std::cout << "Вставка кв. со стороной " << (sideLenOfMainSquare + 1) /
2 << " (x = " << 1 << ", y = " << 1 << ")" << std::endl;
        std::cout << "Вставка кв. со стороной " << sideLenOfMainSquare / 2 <<
" (x = " << 1 << ", y = " << (sideLenOfMainSquare + 1) / 2 + 1 << ")" <<
std::endl;
        std::cout << "Вставка кв. со стороной " << sideLenOfMainSquare / 2 <<
" (x = " << (sideLenOfMainSquare + 1) / 2 + 1 << ", y = " << 1 << ")" <<
std::endl;
    #endif

    // обновление к-ва пустого пространства
    freeAreaOfMainSquare = sideLenOfMainSquare * sideLenOfMainSquare -
((sideLenOfMainSquare + 1) / 2) * ((sideLenOfMainSquare + 1) / 2) - 2 *
(sideLenOfMainSquare / 2) * (sideLenOfMainSquare / 2);
}

void findTaskAnswerWithRecursion(std::vector<std::vector<int>>& visSqr,
int freeAreaOfMainSquare, int currentSizeOfAddedSquare, int
currentCountOfAddedSquares, std::vector<Square>&
currentArrOfAddedSquares, int spacesCount, int sideLenOfMainSquare, int&
bestSquaresCount, std::vector<Square>& bestArrOfAddedSquares)
{
    #ifdef COUNTINFO
        ++operationsCount;
    #endif

```

```

    // даже если квадрат заполнит все пустоты - результат будет не лучше
    // нынешнего
    if ( (currentCountOfAddedSquares == (bestSquaresCount - 1)) &&
        (freeAreaOfMainSquare) )
    {
        #ifdef ADDINFO
        for(int l = 0; l < spacesCount; l++)
        {
            std::cout << " ";
        }
        std::cout << "Разложение не минимально, выход из рекурсии" <<
        std::endl;
        #endif

        return;
    }

    // первый добавленный квадрат (после начальных 3)
    if ( ((currentSizeOfAddedSquare + 1) <= (sideLenOfMainSquare / 2)) &&
        (currentCountOfAddedSquares == 3) )
    {
        // рекурсивный вызов этой же функции
        findTaskAnswerWithRecursion(visSqr, freeAreaOfMainSquare,
        (currentSizeOfAddedSquare+1), currentArrOfAddedSquares.size(),
        currentArrOfAddedSquares, 0, sideLenOfMainSquare, bestSquaresCount,
        bestArrOfAddedSquares);
    }

    bool possibleToAddSuchSquare = false;
    for (int y = 0; y < sideLenOfMainSquare; y++)
    {
        for (int x = 0; x < sideLenOfMainSquare; x++)
        {
            // для каждой пустой клетки происходит попытка вставить квадрат
            // текущего размера
            if (visSqr[y][x] == 0)
            {
                if (isPossibleToAddSquare(visSqr, x, y, currentSizeOfAddedSquare))
                {
                    possibleToAddSuchSquare = true;
                    addToVisibleSquare(visSqr, x, y, currentSizeOfAddedSquare);
                    freeAreaOfMainSquare -= currentSizeOfAddedSquare *
                    currentSizeOfAddedSquare;
                    currentArrOfAddedSquares.push_back({ x, y,
                    currentSizeOfAddedSquare });

                    #ifdef ADDINFO
                    for(int l=0; l < spacesCount; l++)
                    {
                        std::cout << " ";
                    }

```

```

        std::cout << "Вставка кв. со стороной " <<
currentSizeOfAddedSquare<< " (x = " << x + 1 << ", y = " << y + 1 <<
")" << std::endl;
        #endif

        break;
    }
    else
    {
        #ifdef ADDINFO
        for(int l=0; l < spacesCount; l++)
        {
            std::cout << " ";
        }
        std::cout << "Нельзя поставить кв. со стороной " <<
currentSizeOfAddedSquare<< " (x = " << x + 1 << ", y = " << y + 1 <<
")" << std::endl;
        #endif

        return;
    }
}
else
{
    x += (visSqr[y][x] - 1);
}
}

// выход из цикла
if(possibleToAddSuchSquare)
{
    break;
}
}

// нет смысла обновлять показатели лучшего результата при таком
же количестве обрезков
if ( (currentCountOfAddedSquares + 1) == bestSquaresCount)
{
    #ifdef ADDINFO
    for(int l = 0; l < spacesCount; l++)
    {
        std::cout << " ";
    }
    std::cout << "Разложение не минимально, выход из рекурсии" << '\n';
    #endif

    delLastAddedSquare(visSqr, currentArrOfAddedSquares, spacesCount);
    return;
}

// минимальное заполнение

```

```

    if ( ((currentCountOfAddedSquares + 1) < bestSquaresCount) &&
        (freeAreaOfMainSquare == 0))
    {
        bestSquaresCount = currentCountOfAddedSquares + 1;
        bestArrOfAddedSquares.assign(currentArrOfAddedSquares.begin(),
        currentArrOfAddedSquares.end());

        #ifdef ADDINFO
            std::cout << "*Получено новое минимальное к-во кв.: " <<
            bestSquaresCount << std::endl;
        #endif

        delLastAddedSquare(visSqr, currentArrOfAddedSquares, spacesCount);
        return;
    }

    // рекурсивный вызов этой же функции
    for (int i = sideLenOfMainSquare / 2; i > 0; i--)
    {
        if (i * i <= freeAreaOfMainSquare)
        {
            #ifdef ADDINFO
                for (int l=0; l < spacesCount+2; l++)
                {
                    std::cout << " ";
                }
                std::cout << "Вызов рекурсии для кв. со стороной " << i << '\n';
            #endif

            findTaskAnswerWithRecursion(visSqr, freeAreaOfMainSquare, i,
            currentCountOfAddedSquares + 1, currentArrOfAddedSquares,
            spacesCount+2, sideLenOfMainSquare, bestSquaresCount,
            bestArrOfAddedSquares);
        }
    }

    delLastAddedSquare(visSqr, currentArrOfAddedSquares, spacesCount);
}

int main()
{
    setlocale(LC_ALL, "Russian");

    std::vector <std::vector<int>> visibleSquare; // изображение квадрата
    и его занятой площади в виде цифр
    std::vector <Square> currentArrOfAddedSquares;
    std::vector <Square> bestArrOfAddedSquares; // для хранения лучшей
    попытки

    int compression = 1; // коэффициент сжатия квадрата
    int freeAreaOfMainSquare;
    int sideLenOfMainSquare;

```



```
int bestSquaresCount; // минимальное к-во квадратов, которым можно
покрыть основной
```

```
#ifdef ADDINFO
std::cout << "Введите размер стороны квадрата (от 2 до 20):" <<
std::endl;
#endif
std::cin >> sideLenOfMainSquare;
```

```
makeTaskPreparations(compression, visibleSquare,
currentArrOfAddedSquares, freeAreaOfMainSquare, sideLenOfMainSquare,
bestSquaresCount);
```

```
#ifdef ADDINFO
clock_t start = clock();
#endif
```

```
// 1 - это начальный размер добавляемого квадрата, 0 - это глубина
рекурсии для отладочных сообщений
```

```
findTaskAnswerWithRecursion(visibleSquare, freeAreaOfMainSquare, 1,
currentArrOfAddedSquares.size(), currentArrOfAddedSquares, 0,
sideLenOfMainSquare, bestSquaresCount, bestArrOfAddedSquares);
```

```
#ifdef ADDINFO
clock_t end = clock();
```

```
std::cout << "\nВремя выполнения: " << (double) (end - start) /
CLOCKS_PER_SEC << "\n\n";
#endif
std::cout << bestSquaresCount << std::endl;
```

```
#ifdef COUNTINFO
std::cout << "\nКоличество операций = " << ::operationsCount <<
std::endl;
#endif
```

```
// вывод ответа и загрузка лучшего найденного разложения в
"видимый" квадрат
```

```
for (int i = 0; i < bestArrOfAddedSquares.size(); i++)
{
    std::cout << bestArrOfAddedSquares[i].x * compression + 1 << " " <<
bestArrOfAddedSquares[i].y * compression + 1 << " " <<
bestArrOfAddedSquares[i].size * compression << std::endl;
```

```
#ifdef ADDINFO
addToVisibleSquare(visibleSquare, bestArrOfAddedSquares[i].x *
compression, bestArrOfAddedSquares[i].y * compression,
bestArrOfAddedSquares[i].size * compression);
#endif
}
```

```
#ifdef ADDINFO
```

```
std::cout << std::endl;
printVisibleSquare(visibleSquare, compression, sideLenOfMainSquare);
#endif
return 0;
}
```