МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1

по дисциплине «Построение и анализ алгоритмов»

Тема: Поиск с возвратом

Студент гр. 9382 _____ Иерусалимов.Н

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

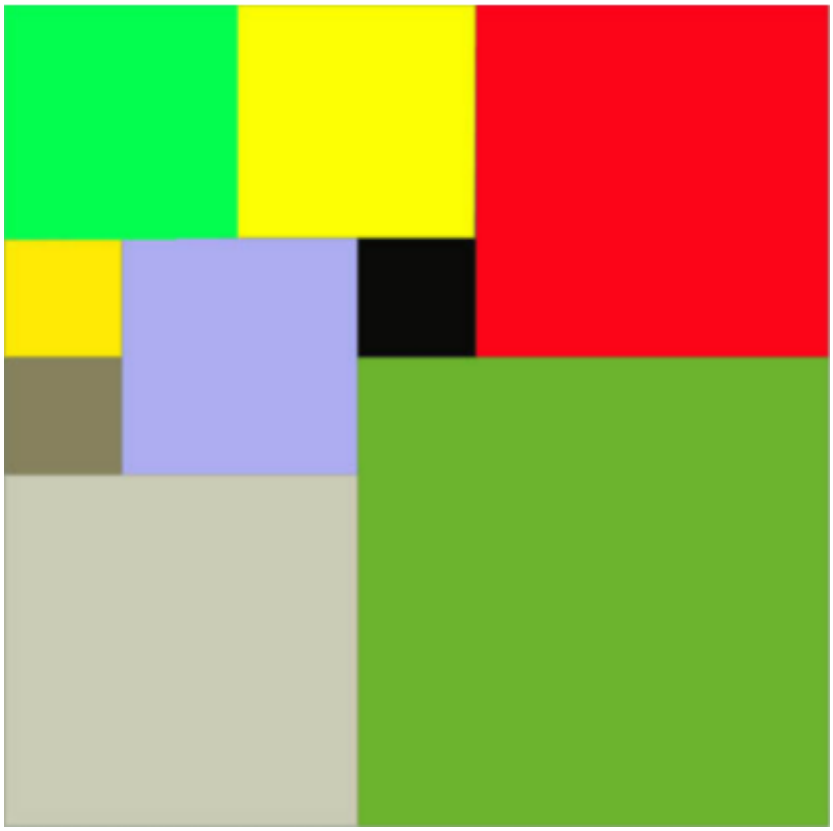
Познакомиться с алгоритмом поиск с возвратом. Получить навыки решения задач на этот алгоритм. Произвести исследование на количество операций.

Задание.

Вар. 3и. Итеративный бэктрекинг. Исследование кол-ва операций от размера квадрата.

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.



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

Описание алгоритма.

Первоначально алгоритм ищет пустое место на карте, путем прохода по каждой клетке.

Тут может быть два варианта:

1)Если такое место найдено тогда проверяется, какой максимальный квадрат туда можно положить с учетом доступного места, после чего ставит на это место квадрат с посчитанной длинной.

2)Если же такое место не найдено, тогда естественно квадрат заполнен и алгоритм выходит из цикла заполнения . Проверяется минимальное количество с лучшим прошлым заполнением. Если оно лучше тогда перезаписываем лучшее, если нет, тогда идем дальше. После проверки начинаем удалять из стека единичные квадраты и следуемый за ними не единичный квадрат. После чего меняем допустимый размер для постановки новых квадратов, он должен быть на единицу меньше последнего удаленного квадрата. После чего алгоритм запускается заново. Он завершает свою работу когда удаляется третий поставленный квадрат в стек.

Оптимизация алгоритма.

Есть 3 варианта когда пользователь вводит размер большого квадрата:

- Число четное .
- Число нечетное и не простое .
- Число простое .

1)Если размер четный тогда довольно очевидно можно поставить только 4 квадрата и это и будет нашим отчетом.(рис.1)

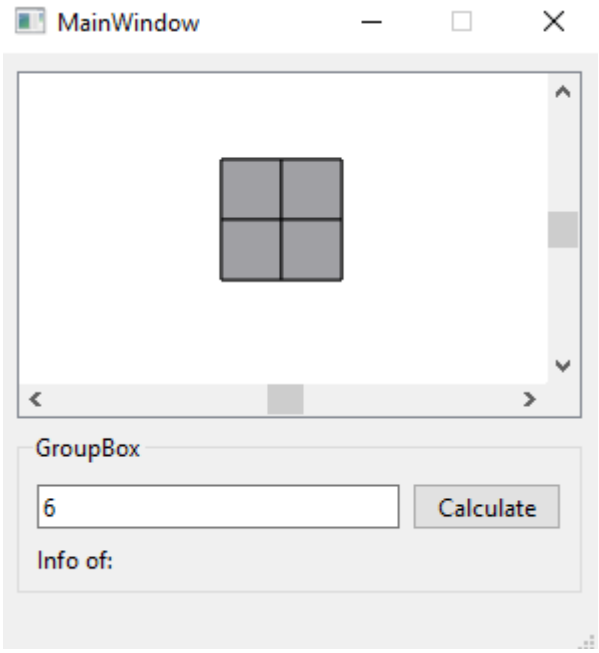


Рис.1<Четная сторона>

2)Размер нечетный и не простой. Это число можно сжать до минимального квадрата и там уже запускать алгоритм. После чего до множить ответы на сжатие. Пример, у нас квадрат стороной 9 число не простое и нечетное, это число можно уменьшить до 3, запустить алгоритм и потом умножить результаты на 3.(рис.2)

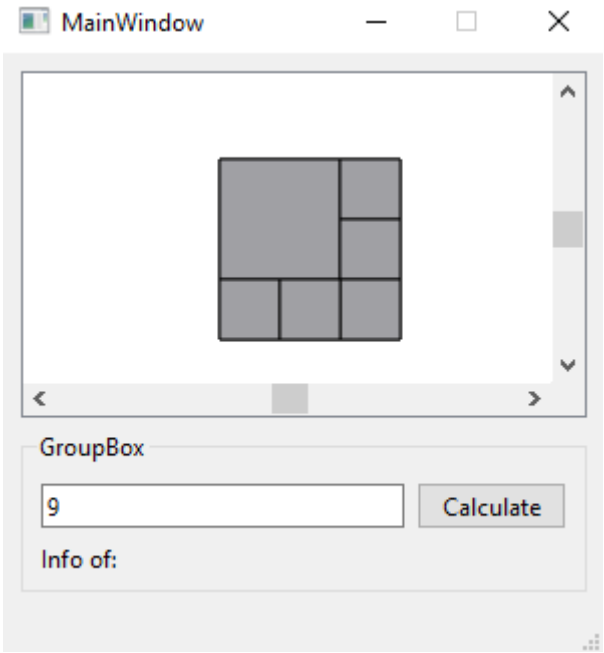


Рис.2<Нечетная сторона и не простая>

3)Число простое. В таком квадрате можно сразу вставить 3 квадрата ибо они точно будут входить в решение. Первый квадрат будет размером (size+1)/2, а остальные size - (size+1)/2.(рис.3)

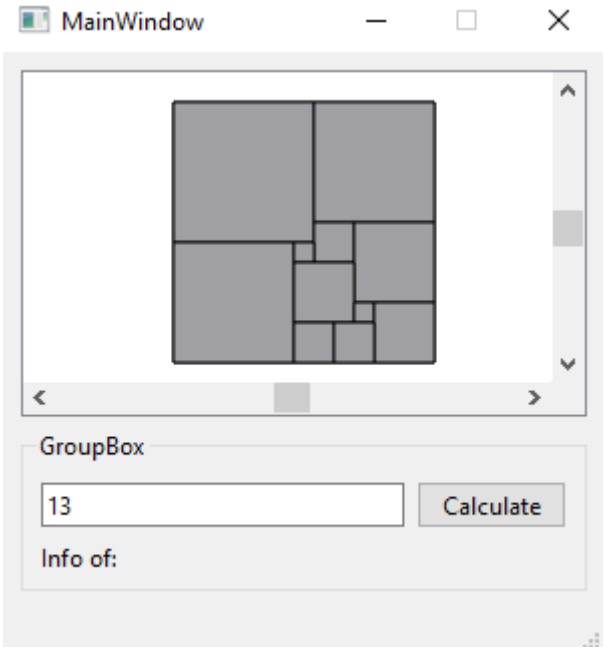


Рис.3<Сторона простая>

Сложность алгоритма.

В программе присутствует матрица в которой храниться положение квадратов, вектор с координатами данных квадратов . Вектор не превысит O(n^2) так как в матрицу можно максимуму вставить n^2 единичных квадратиков. На практике оно еще меньше из-за оптимизаций. Матрица у нас n на n так что сложно по памяти будет O(n^2).

С оценкой скорости сложнее. Так как оптимизация была довольно значительная, сложно определить точную скорость, так что оценка будет грубой. Для прохода по матрице и поиска нового свободного места надо O(n^2), для каждой новой скорости понадобиться еще O(n^2) и тд. Т.е O(n^n). Удаление квадрата из матрицы занимает O((n-1)^2).

Тогда сложность по времени выходит O(n^n * (n-1)^2).

Описание функции:

struct Cell – Структура клетки в матрице

x – координата по столбцу.

Y – координата по строке.

Size – размер квадрата.

class Square – класс квадрата.

Square(int size) – конструктор. Size – размер стороны квадрата

~Square() – Деструктор. Освобождает память.

void setSquare(int x, int y, int size) – Ставит квадрат определенного размера на определенные координатам.

Int x - координата по столбцу.

Int y – координата по строке.

Int size – размер квадрата

int canSetSquareinRange(Cell range) – Какой самый большой квадрат можно поставить на эти координаты.

Cell range – клетка, в которой содержаться координаты откуда надо считать и какой максимальный размер может быть у квадрата.

Cell removeSquare(int x, int y, int size) – Удаляет квадрат и возвращает данные удаленного квадрата.

Int x - координата по столбцу.

Int y – координата по строке.

Int size – размер квадрата

Cell whereEmpty(int x, int y) – ищет пустую клетку, и возвращает ее.

Int x - координата по столбцу.

Int y – координата по строке.

*Square *copy()* – Копирует текущий класс и возвращает его.

class Solve – класс решение. Тут прописан сам алгоритм и дополнения к нему.

Solve(int size)- конструктор. Инициализирует, переменные класса.

Int size – сторона квадрата

int compression(int size, int &comp) – Сжимает размер, ищет делитель и делит размер. Возвращает новый размер квадрата и по ссылке меняет comp на делитель. Чтобы потом можно было вернуть размер.

Int size – сторона квадрата

int &comp – ссылка на переменную компресс

void bestSquare() – инициализирует лучший вариант для нечетных и не простых чисел.

void currentMap() - ставит первые три “основных” квадраты.

void solveEven() – Решение для четной стороны.

void solve() – Запускает основной алгоритм поиска с возвратом.

void fillSquare() – подготавливает матрицы ”карты” и в зависимости от размера вызывает нужное решение.

Тестирование:

№	Входные данные	Выходные данные
1	4	Count squares: 4 Cords-> x y size 1 1 2 1 3 2 3 1 2 3 3 2 count operation: 17

2	5	Count squares: 8 Cords-> x y size 1 1 3 4 1 2 1 4 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1 count operation: 344
3	7	Count squares: 9 Cords-> x y size 1 1 4 5 1 3 1 5 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2 count operation: 1538
4	9	Count squares: 6 Cords-> x y size 1 1 6 7 1 3 7 4 3 7 7 3 1 7 3 4 7 3 count operation: 112
5	11	Count squares: 11

		Cords-> x y size 1 1 6 7 1 5 1 7 5 7 6 3 10 6 2 6 7 1 6 8 1 10 8 1 11 8 1 6 9 3 9 9 3 count operation: 24021
6	13	Count squares: 11 Cords-> x y size 1 1 7 8 1 6 1 8 6 8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2 count operation: 57479
7	14	Count squares: 4 Cords-> x y size 1 1 7 1 8 7 8 1 7 8 8 7

		count operation: 17
8	18	Count squares: 4 Cords-> x y size 1 1 9 1 10 9 10 1 9 10 10 9 count operation: 17
9	19	Count squares: 13 Cords-> x y size 1 1 10 11 1 9 1 11 9 11 10 3 14 10 6 10 11 1 10 12 1 10 13 4 14 16 1 15 16 1 16 16 4 10 17 3 13 17 3 count operation: 1162246

Исследование

Для подсчета количества операций был введен счетчик “countOp”….

Его увеличивали когда происходило присваивание, любое взаимодействие с массивом(за исключением сложения вычета и тд.), вызов функции/метода, на каждом проходе цикла и на любых булевых операциях. Не считались операции которые отвечали за вывод промежуточной информации в программе. Уже так мы можем увидеть тенденцию

увеличение операций на простых числах. Снизу приведен график и таблица по полученным данным.

Длина квадрата	Количество операций
2	17
3	112
4	17
5	344
6	17
7	1538
8	17
9	112
10	17
11	24021
12	17
13	57479
14	17
15	118
16	17
17	390971
18	17
19	1162246
20	17
21	124
22	17
23	4673828

Таблица 1<Количество операций>

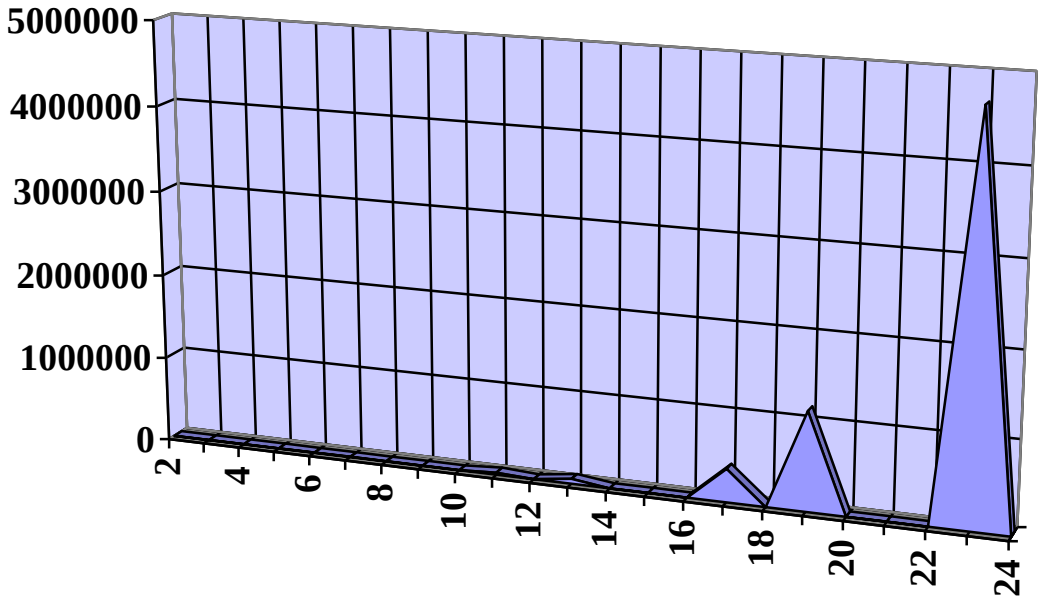


Диаграмма 1<Тенденция возрастания операций>

Чем дальше, тем больше операций требуется для проверки всех вариантов. На диаграмме заметны изменения начиная с размера 11. Причем с помощью оптимизации прирост видно только на простых числах. На других изменений почти нет ибо там мы все сводим к одинаковым случаям и получаем маленькое изменение количества операций. Операции там могут расти только за счет функции сжатия

Выводы.

Был исследован часто используемый на практике алгоритм - поиск с возвратом. Также были получены навыки решения задач на этот алгоритм. Произвели исследование по количеству операций относительно введенных данных. Смогли заметно оптимизировать процесс.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <array>

int choise = 1;

struct Cell {
    int size;
    int x, y;
};

class Square {
public:

    Square(int size,int comp) {
        this->size = size;
        this->comp = comp;
        count = 0;
        ++countOp;
    }

    ~Square() {
    }

    void setSquare(int x, int y, int size) {
        ++(this->count);
        for (int i = y; i < y + size; ++i) {
            for (int j = x; j < x + size; ++j) {
                this->map[i][j] = size;
            }
        }
        Cell newSquare;
        newSquare.x = x;
        newSquare.y = y;
        newSquare.size = size;
        squareStack.push_back(newSquare);
        countOp += 4 + (size - 1) * (size - 1);
    }

    int canSetSquareinRange(Cell range) {
        --range.size;
        int sizeSmallSquare = 1;
        while (sizeSmallSquare < range.size && (range.x +
sizeSmallSquare) < size &&
                (range.y + sizeSmallSquare) < size &&
                !(this->map[range.y][range.x + sizeSmallSquare])) {
            ++sizeSmallSquare;
            countOp += 7;
        }

        return sizeSmallSquare;
```

```
    }

    Cell removeSquare(int x, int y, int size) {
        --(this->count);
        for (int i = y; i < y + size; ++i) {
            for (int j = x; j < x + size; ++j) {
                this->map[i][j] = 0;
            }
        }
        Cell temp = squareStack.back();
        squareStack.pop_back();
        countOp += 3 + (size - 1) * (size - 1);
        return temp;
    }

    Cell whereEmpty(int x, int y) {
        Cell arr;
        while (this->map[y][x]) {
            if (x == this->size - 1) {
                if (y == this->size - 1) {
                    return arr = {-1, -1, -1};
                }
                x = this->size / 2;
                ++y;
                countOp += 4;
            } else {
                ++x;
            }

        }
        arr = {0, x, y};
        countOp += 3;
        return arr;
    }

    Square *copy() {
        Square *newSq = new Square(this->size,this->comp);
        for (int i = 0; i < this->size; ++i) {
            for (int j = 0; j < this->size; ++j) {
                newSq->map[i][j] = this->map[i][j];
            }
        }
        newSq->count = this->count;
        newSq->squareStack = this->squareStack;
        countOp += 3 + ((size - 1) * (size - 1));
        return newSq;
    }

    std::array<std::array<int,30>,30> map;
    std::vector<Cell> squareStack; //стек наших кубиков.
    int size;                      //размер полатна
    int count;                     // счетчик квадратов
    int comp;
    int countOp = 0;
};
```

```
//template<typename Type>
void printMessage(const char* message){
    if (choise == 1) {
        std::cout<<message;
    }
}

void printSquare(Square *some, bool flagIsMap) {
    //bool is_same_v = std::is_same < Type, Square > :: value ;
    if (choise == 1) {
      if ( flagIsMap) {

            for (int i = 0; i < some->size; ++i) {
                for (int j = 0; j < some->size; ++j) {
                    std::cout << some->map[i][j] << " ";
                }
                std::cout << "\n";
            }

        } else {
            for (int i = 0; i < some->squareStack.size(); ++i) {
                std::cout << some->squareStack[i].x * some->comp + 1
<< " "
                          << some->squareStack[i].y * some->comp + 1
<< " "
                          << some->squareStack[i].size * some->comp <<
"\n";
            }
        }
    }

}


class Solve {
public:
    Solve(int size) {
        comp = 1;
        this->size = compression(size, comp);
        curMap = new Square(size,comp);
        easyMap = new Square(size,comp);
    }

    ~Solve() {
        delete curMap;
        delete easyMap;
    }

    int compression(int size, int &comp) {
        int compSize = size;
        for (int i = size - 1; i > 1; --i) {
            if (!(size % i)) {
                comp = i;
                compSize = size / i;
                countop += 2;
                break;
```

```
            }
        }
        countop += 1 + (size - 2);
        return compSize;
    }

    void bestSquare() {
        easyMap->setSquare(0, 0, size - 1);
        for (int y = 0; y < size; ++y) {
            easyMap->setSquare(size - 1, y, 1);
        }
        for (int x = 0; x < size - 1; ++x) {
            easyMap->setSquare(x, size - 1, 1);

        }
    }

    void currentMap() {
        curMap->countOp += 3;
        curMap->setSquare(0, 0, size / 2 + 1);
        curMap->setSquare(size / 2 + 1, 0, size / 2);
        curMap->setSquare(0, size / 2 + 1, size / 2);
    }


    void solve() {
        currentMap();
        printMessage("Add 3 square with this parameters:\nx y
size|\n");
        printSquare(easyMap,0);
        printMessage("This is the most optimal
way.\n\nVisualization:\n");
        printSquare(curMap,1);
        int x = size / 2 + 1;
        int y = size / 2;
        curMap->countOp += 2;

        Cell tempS;

        printMessage("\nTry find free space...\n");
        tempS = curMap->whereEmpty(x, y);
        tempS.size = size - 1;
        int sizeOfCanSet;
        curMap->countOp += 3;
        printMessage("Found an empty space on: ");

        if (choise) {
            std::cout << "{ " << tempS.x + 1 << ", " << tempS.y + 1 <<
" }\n";
        }

        while (true) {
            curMap->countOp += 2;
            printMessage("The big cycle has begun...\n");
            if (curMap->count == 2) {
```

```
                printMessage("Removed the 3-d square. \nIt makes no
sense to continue "
                        "further, all the best options have already been
found.\n");
                break;
            }

            bool full = false;

            while (!full) {

                //Выбираем самый большой квадрат который можно
вставить в эту область.
                printMessage("\n\tThe cycle of filling has
begun...\n");
                sizeOfCanSet = curMap->canSetSquareinRange(tempS);
                if (choise) {
                    printMessage("\tWe are looking for the largest
square that can be put on: ");
                    std::cout << "{ " << tempS.x + 1 << ", " <<
tempS.y + 1
                              << " } in range " << tempS.size << ", ";
                    printMessage("this is it: ->");

                    std::cout << " " << sizeOfCanSet << "\n";
                }

                // insert square

                curMap->setSquare(tempS.x, tempS.y, sizeOfCanSet);
                printMessage("\tSet a square to these
coordinates:\n\tx y size\n");
                if (choise) {
                    std::cout << "\t" << curMap->squareStack.back().x
* comp + 1 << " "
                              << curMap->squareStack.back().y * comp +
1 << " "
                              << curMap->squareStack.back().size *
comp
                              << "\n\n\tVisualization:\n";
                }
                printSquare(curMap,1);

                //Если кол-во квадратов в текущем больше или равно
лучшего варианта. Не
                //продолжать. Смысла нет.

                if (curMap->count >= easyMap->count) {
                    printMessage("\tThe number of squares in the new
case is more than in the "
                                 "best.\nExit the fill cycle\n");
                    break;
                }


                printMessage("\n\tTry find free space...\n");
```

```
                tempS = curMap->whereEmpty(x, y);

                if (tempS.size == -1) {
                    full = true;
                    ++curMap->countOp;
                    printMessage("\tNot found an empty space.\nExit
the fill cycle\n");

                } else {

                    printMessage("\tFound an empty space on: ");
                    if (choise) {
                        std::cout << "{ " << tempS.x + 1 << ", " <<
tempS.y + 1 << " }\n";
                    }
                }
                tempS.size = size - (size + 1) / 2;
                curMap->countOp += 6;
            }

            if (full && (curMap->count < easyMap->count)) {

                if (choise) {
                    std::cout << "Best map count  " << easyMap->count
<< " >  "
                              << curMap->count << " new map count\n";
                }
                ++curMap->countOp;
                easyMap = curMap->copy();

                printMessage("Copied the new version to the best one.
Since there are fewer "
                     "squares\n\nVisualization new Best Map:\n");
                printSquare(curMap,1);
            }

            printMessage("\n_____________________________Remove to
try new "
                  "Map_____________________________\nCycle removed
start!\n");
            do {
                curMap->countOp += 6;
                printMessage("\nRemove square at this pos:\nx y
size\n");
                if (choise) {
                    std::cout << curMap->squareStack.back().x * comp +
1 << " "
                              << curMap->squareStack.back().y * comp +
1 << " "
                              << curMap->squareStack.back().size *
comp
                              << "\n\nVisualization:\n";
                }
                tempS = curMap->removeSquare(curMap-
>squareStack.back().x,
```

```
                                                curMap-
>squareStack.back().y,
                                                curMap-
>squareStack.back().size);
                printSquare(curMap,1);

printMessage("\n||||||||||||||||||||||||||||||||||||||||||||||||||||||||||
||||||||"
                        "||||||||||||||||||\n");
            } while (curMap->count > 2 && tempS.size == 1);
            curMap->countOp += 5;
            printMessage("\nCycle removed
end.\n_____________________________________End "
                  "Removing_____________________________________\n");
        }
    }

    void solveEven() {
        curMap->countOp += 4;
        easyMap->setSquare(0, 0, 1);
        easyMap->setSquare(0, 1, 1);
        easyMap->setSquare(1, 0, 1);
        easyMap->setSquare(1, 1, 1);
    }

    void fillSquare() {

        printMessage("Getting started filling the "
              "square!\n_____________________________________\n");
        if (!(this->size % 2)) {
            curMap->countOp += 1;
            printMessage("The side is even!\n When the side is even,
the square is divided "
                  "into 4 equal squares. This will be the answer.\n");
            solveEven();

        } else {
            bestSquare();
            solve();
            curMap->countOp += 2;
            easyMap->countOp = curMap->countOp + countop;
        }

        return;
    }

    Square *easyMap; // лучший вариант
    Square *curMap;  // текущий вариант
    int comp; //сжатие для четной стороны квадрата. И для нечетной не
простой
    //стороны
    int size;
    int countop = 0;

private:
};
```

```
int main() {
    int SizeOfSide;

//    std::cout << "Logs?\n0 - No || 1 - Yes\n";
//    std::cin >> choise;
//
//    while (choise < 0 || choise > 1) {
//        std::cout << "Wrong Enter, try again.\n";
//        std::cin.clear();
//        std::cin >> choise;
//    }

    std::cout << "Enter the length of the square: \n";
    std::cin >> SizeOfSide;
    Solve ml(SizeOfSide);

    std::cout << "Created a square with a side " << SizeOfSide <<
".\n";
    ml.fillSquare();

    std::cout << "\nResult!" <<
"\n_____________________________________________________\n";
    printSquare(ml.easyMap,0);
    std::cout << "\n count operation: " << ml.easyMap->countOp <<
"\n";
    std::cout << "\nBest map is:\n";

    printSquare(ml.easyMap,1);
    return 0;
}
```