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

Студенка гр. 8303 _____ Самойлова А. С.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2020

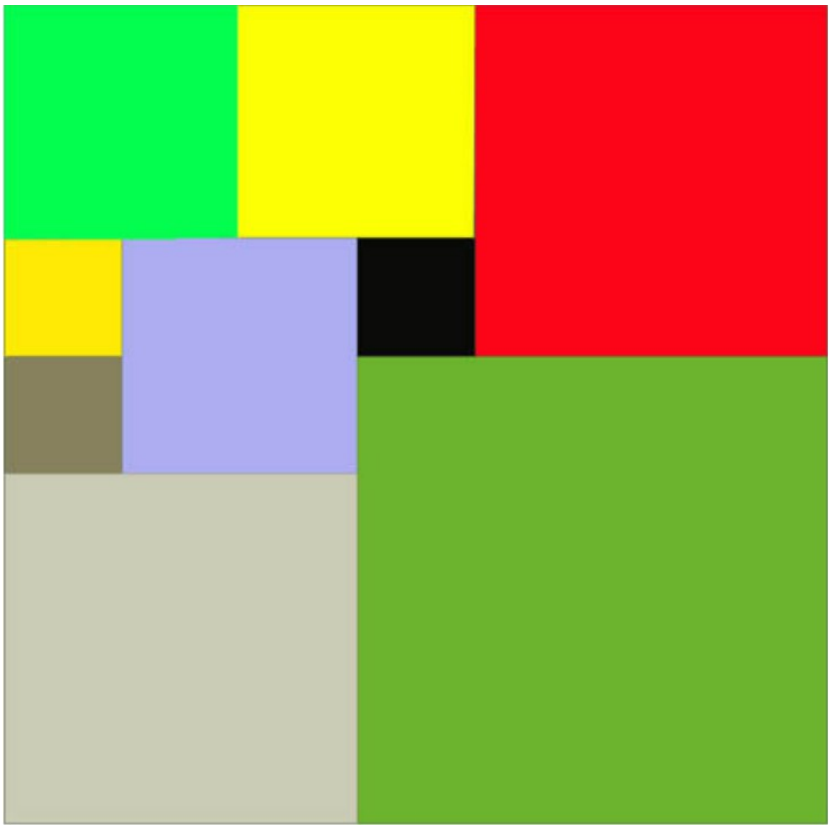
Цель работы

Изучить алгоритм бэктрекинга.

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.



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

Вариант 5р

Рекурсивный бэктрекинг. Возможность задать список квадратов (от 0 до N^2 квадратов в списке), которые обязательно должны быть использованы в покрытии квадрата со стороной N.

Ход выполнения работы

Структуры хранения данных:

Класс **Square** хранит информацию о столешнице:

- **iSize** — размер столешницы
- **iBestConfiguration** — лучшее расположение квадратов на столешнице
- **res** — количество квадратов, используемых для лучшего расположения
- **iArray** — поле для поиска расположения квадратов на столешнице
- **iColors** — счетчик квадратов для поиска их оптимального расположения на столешнице

Структура **SquareData** хранит:

- структуру **Coord** (несущую в себе координаты (**x**, **y**) верхнего левого угла квадрата, расположенного на столешнице)
- **w** — размер этого квадрата.

Структура **SquareList** хранит:

- **count** — количество квадратов, которые необходимо расположить на столешнице
- **arr** — размеры квадратов, которые необходимо расположить на столешнице

Используемые оптимизации:

В качестве оптимизаций используются частные случаи расположения квадратов при определённых размерах столешницы:

- Если сторона столешницы кратна 2, то наиболее оптимальным решением является разделить её на 4 равных квадрата со сторонами равными половине стороны столешнице.

```
void Square::div2(){
        setSquare(0, 0, iSize/2);
        setSquare(iSize/2, 0, iSize/2);
        setSquare(0, iSize/2, iSize/2);
        setSquare(iSize/2, iSize/2, iSize/2);
}
```

- Если сторона столешницы кратна 3, то наиболее оптимальным решением является разделить её на 6 квадратов:
  1. Квадрат со сторонами равными 2/3 стороны столешницы
  2. 5 квадратов со сторонами равными 1/3 стороны столешницы

```
void Square::div3(){
        setSquare(0, 0, iSize/3*2);
        setSquare(iSize/3*2, 0, iSize/3);
        setSquare(iSize/3*2, iSize/3, iSize/3);
        setSquare(0, iSize/3*2, iSize/3);
```

```
        setSquare(iSize/3, iSize/3*2, iSize/3);
        setSquare(iSize/3*2, iSize/3*2, iSize/3);
    }
```

- Если сторона столешницы кратна 5, то наиболее оптимальным решением является разделить её на 8 квадратов:

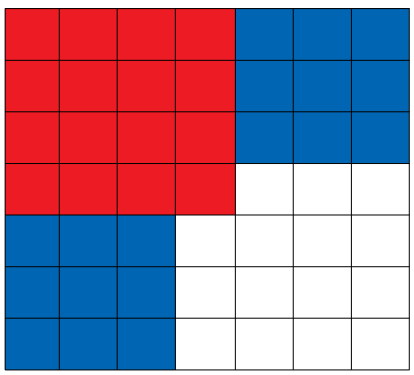
    1. Квадрат со сторонами равными 3/5 стороны столешницы
    2. 3 квадрата со сторонами равными 2/5 стороны столешницы
    3. 4 квадрата со сторонами равными 1/5 стороны столешницы

```
    void Square::div5(){
        setSquare(0, 0, iSize/5*3);
        setSquare(iSize/5*3, 0, iSize/5*2);
        setSquare(iSize/5*3, iSize/5*2, iSize/5*2);
        setSquare(0, iSize/5*3, iSize/5*2);
        setSquare(iSize/5*2, iSize/5*3, iSize/5);
        setSquare(iSize/5*2, iSize/5*4, iSize/5);
        setSquare(iSize/5*3, iSize/5*4, iSize/5);
        setSquare(iSize/5*4, iSize/5*4, iSize/5);
    }
```

Если сторона столешницы не кратна ни 2, ни 3, ни 5 и при этом пользователь не хочет разместить на столешнице квадраты заданных размеров, то используется оптимизация:

- Первые три квадрата функция **rec_backtrack** размещает на столешнице автоматически:

    1. Квадрат со сторонами равными половине (округлённой в большую сторону) стороны столешницы
    2. 2 квадрата со сторонами равными половине (округлённой в меньшую сторону) стороны столешницы, присоединённые к первому квадрату



```
setSquare(0, 0, iSize/2+1);
setSquare(0, iSize/2+1, iSize/2);
setSquare(iSize/2+1, 0, iSize/2);
```

Оптимизации позволяют уменьшить время работы алгоритма и количество памяти необходимой для поиска решения.

Алгоритм программы:

Программа считывает размер столешницы, а затем количество квадратов, которые пользователь хочет обязательно разместить на столешнице, с помощью функции **read_sizes**. Если количество квадратов, которые необходимо разместить на столешнице не равно 0, то функция **read_sizes** считывает размеры этих квадратов в массив, хранящийся в структуре **SquareList**.

Начиная обрабатывать входные данные, программа обращается к функции **proceed**, которая проверяет можно ли найти решения, используя функции для оптимизации алгоритма (описанные в разделе «Используемые оптимизации»). Для выполнения индивидуального задания функция была перегружена для возможности проверять наличие заданных пользователем квадратов в найденном варианте решения.

Если оптимальное решение найти не удалось или оно не удовлетворяет заданным условиям, программа обращается к функции **rec_backtrack**.

Функция **rec_backtrack** принимает количество уже размещённых на столешнице квадратов и не имеет возвращаемого значения.

Для выполнения индивидуального задания функция **rec_backtrack** была перегружена. Перегруженный вариант функции принимает количество уже размещённых на столешнице квадратов и указатель на структуру содержащую информацию о количестве квадратов, которые необходимо разместить и массив размеров этих квадратов. Возвращаемым значением является флаг,

показывающий удалось ли найти такой вариант размещения квадратов на столешнице, который содержит квадраты заданных размеров.

Рекурсивная функция сначала проверяет, что количество уже размещённых на столешнице квадратов не превышает количество квадратов в наилучшем варианте решения (изначально это значение равно квадрату стороны столешницы). Затем рекурсивная функция осуществляет поиск максимально большого пустого квадрата начиная с самой верхней и самой левой пустой точки. Это выполняет функция **findPotentialSquare**, не принимающая аргументов и возвращающая координаты самой верхней и самой левой точки найденного квадрата, а так же размер его стороны.

Если функция **findPotentialSquare** вернула квадрат размером во всю столешницу и пользователь не требует расположить на столешнице определённые квадраты, то функция **rec_backtrack** использует оптимизацию по размещению первых трёх квадратов (описанную в разделе «Используемые оптимизации»).

Если функция **findPotentialSquare** вернула квадрат размером 0, то считается, что программа нашла какое-то расположение квадратов на столешнице, и проверяется что количество квадратов в найденном расположении меньше, чем в **res** (наилучшем на данный момент). Если найденное расположение квадратов является оптимальным на данный момент, то оно сохраняется с помощью функции **setConfiguration**.

Если функция **findPotentialSquare** вернула квадрат размером от 0 до размера столешницы, то программа закрашивает этот квадрат с помощью функции **setSquare**.

Функция **rec_backtrack** вызывается рекурсивно.

При возвращении на описываемый уровень рекурсии закрашиваемый квадрат уменьшается на 1 и снова рекурсивно вызывается функция

rec_backtrack. Эти действия повторяются до тех пор, пока размер закрашиваемого квадрата больше 0.

После завершения работы рекурсивной функции **rec_backtrack**, найденное оптимальное решение выводится на экран в виде числа, показывающего количество используемых квадратов, и строк с числами в формате:

[*координата по оси Ox] [координата по оси Oy] [размер квадрата]*

Если же решение не было найдено (что вероятно при наличие квадратов, которые необходимо вставить в столешницу по желанию пользователя) выводится сообщение об этом.

Хранение промежуточных данных:

Промежуточные данные о текущем расположении квадратов на столешнице хранятся в классе Square. В переменной iColors хранится текущее количество расположенных на поле квадратов, в двумерном массиве iArray хранится информация о клетках размером 1x1 на столешнице (свободна или относится к какому-либо квадрату).

Оценка сложности работы алгоритма

Сложность работы алгоритма по времени оценивается по количеству необходимых операций для поиска решения как n^n в худшем случае (когда не работают оптимизации и осуществляется полный перебор вариантов), где n — размер стороны столешницы.

Так как алгоритм хранит не все варианты размещения квадратов на столешнице, а только лучший вариант и массив для текущих изменений расположений квадратов, то оценка сложности по памяти основывается только на глубине рекурсии и оценивается как $2*n^2 + C*n^2$ в худшем случае, где n — размер стороны столешницы, а C — константа, показывающая сколько

памяти занимает каждый вызов рекурсивной функции и состоящая из размеров передаваемых в функцию значений, инициализируемых в этой функции переменных, а так же памяти, отдаваемой под техническую информацию о вызове этой функции и адресе возврата.

Тестирование:

```
Введите размер столешницы:
11
Введите количество квадратов, которые необходимо разместить:
0
Решение:
11
1 1 6
1 7 5
7 1 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
```

```
Введите размер столешницы:
7
Введите количество квадратов, которые необходимо разместить:
1
Введите размеры квадратов, которые необходимо разместить:
5
Решение:
10
1 1 5
6 1 2
6 3 2
6 5 2
1 6 2
3 6 2
5 6 1
5 7 1
6 7 1
7 7 1
```

```
Введите размер столешницы:
9
Введите количество квадратов, которые необходимо разместить:
3
Введите размеры квадратов, которые необходимо разместить:
4 3 1
Решение:
13
1 1 5
6 1 4
6 5 4
1 6 3
4 6 2
4 8 2
1 9 1
2 9 1
3 9 1
6 9 1
7 9 1
8 9 1
9 9 1
```

Пример работы программы с выводом частичных решений:

```
Введите размер столешницы:
7
Введите количество квадратов, которые необходимо разместить:
0
Вариант лучшего расположения квадратов:
10
1 1 4
1 5 3
5 1 3
5 4 3
4 5 1
4 6 1
4 7 1
5 7 1
6 7 1
7 7 1
Вариант лучшего расположения квадратов:
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
```

```
Решение:
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
```

Выводы

Была разработана программа для нахождения оптимального расположения квадратов, заполняющих столешницу, использующая алгоритм поиска с возвратом, а также имеющая возможность находить решение, содержащее в себе квадраты размеров, заданных пользователем.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

```
#include <iostream>

#include <algorithm>
#include <vector>

struct SquareList
{
    int count = 0;
    int* arr;
};

struct Coord
{
    int x;
    int y;
};

struct SquareData
{
    Coord pos;  // координаты
    int w;   // размер квадрата

    friend std::ostream& operator<<(std::ostream& os, const SquareData& sq){
        os << sq.pos.x + 1 << " " << sq.pos.y + 1 << " " << sq.w << std::endl;
    }
};

class Square
{
private:
    int **iBestConfiguration; // лучшее расположение квадратов на поле
    int res; // количество квадратов в лучшем расположении

    const int iSize; //размер большого квадрата

    int **iArray; // поле для поиска расположения квадратов
    int iColors = 0; //количество квадратов на поле

    int optimize();
    void div2();
```

```
    void div3();
    void div5();
    SquareData findPotentialSquare() const;
    SquareData findSquare_(int color, int** Array);
    void rec_backtrack(int iCurSize);
    int rec_backtrack(int iCurSize, SquareList* list);
    void setConfiguration();

public:
    Square(int iSize):
        iSize(iSize)
    {
        iBestConfiguration = new int*[iSize];
        iArray = new int*[iSize];
        for(int i = 0; i < iSize; i++){
            iBestConfiguration[i] = new int[iSize];
            iArray[i] = new int[iSize];
            for(int j = 0; j < iSize; j++){
                iBestConfiguration[i][j] = 0;
                iArray[i][j] = 0;
            }
        }
    }

    void proceed();
    int proceed(SquareList* list);
    int getSize() const;
    void setSquare(int x, int y, int w);
    void printConfiguration(std::ostream &os);
    void delSquare(int x, int y);

    ~Square(){
        for(int i = 0; i < iSize; i++){
            delete [] iBestConfiguration[i];
            delete [] iArray[i];
        }
        delete [] iBestConfiguration;
        delete [] iArray;
    }
};

void Square::proceed(){
```

```
    if(!optimize()){
        res = iSize*iSize;
        rec_backtrack(0);
        iColors = res;
    }
}

int Square::proceed(SquareList* list){
    int find_flag = 0;
    if(optimize()){
        for(int j = 0; j<list->count; j++){
            int flag = 0;
            for(int i = 1; i <= iColors; i++){
                auto sq = findSquare_(i, iBestConfiguration);
                if(list->arr[j] == sq.w){
                    flag = 1;
                    find_flag = 1;
                    break;
                }
            }
            if(flag == 0){
                res = iSize*iSize;
                iColors = 0;
                for(int i = 0; i < iSize; i++){
                    for(int j = 0; j < iSize; j++){
                        iBestConfiguration[i][j] = 0;
                        iArray[i][j] = 0;
                    }
                }
                find_flag = rec_backtrack(0, list);
                return find_flag;
            }
        }
    }
    else{
        find_flag = rec_backtrack(0, list);
    }
    return find_flag;
}
```

```
//
_____OPTIMIZE_____
___

int Square::optimize(){
    if (iSize % 2 == 0){
        div2();
        setConfiguration();
        return 1;
    }
    else if (iSize % 3 == 0){
        div3();
        setConfiguration();
        return 1;
    }
    else if (iSize % 5 == 0){
        div5();
        setConfiguration();
        return 1;
    }
    return 0;
}

void Square::div2(){
    setSquare(0, 0, iSize/2);
    setSquare(iSize/2, 0, iSize/2);
    setSquare(0, iSize/2, iSize/2);
    setSquare(iSize/2, iSize/2, iSize/2);
}

void Square::div3(){
    setSquare(0, 0, iSize/3*2);
    setSquare(iSize/3*2, 0, iSize/3);
    setSquare(iSize/3*2, iSize/3, iSize/3);
    setSquare(0, iSize/3*2, iSize/3);
    setSquare(iSize/3, iSize/3*2, iSize/3);
    setSquare(iSize/3*2, iSize/3*2, iSize/3);
}

void Square::div5(){
    setSquare(0, 0, iSize/5*3);
    setSquare(iSize/5*3, 0, iSize/5*2);
```

```
    setSquare(iSize/5*3, iSize/5*2, iSize/5*2);
    setSquare(0, iSize/5*3, iSize/5*2);
    setSquare(iSize/5*2, iSize/5*3, iSize/5);
    setSquare(iSize/5*2, iSize/5*4, iSize/5);
    setSquare(iSize/5*3, iSize/5*4, iSize/5);
    setSquare(iSize/5*4, iSize/5*4, iSize/5);
}

//
```

```
SquareData Square::findPotentialSquare() const{ //поск свободного места для квадратика на поле, возвращает позицию и размер
    int x = -1;
    int y = -1;
    for(int i = 0; i < iSize; i++){
        bool b = 0; // флаг для выхода из цикла
        for(int j = 0; j < iSize; j++){
            if(iArray[i][j] == 0){
                y = i;
                x = j;
                b = 1;
                break;
            }
        }
        if(b) break;
    }
    if(x == -1 && y == -1){
        return SquareData{Coord{0, 0}, 0};
    }
    int s;
    for(s = 0; s <= iSize - std::max(x, y); s++){
        bool b = 0; // флаг для выхода из цикла
        for(int i = y; i < y+s; i++){
            if(iArray[i][x+s-1] != 0){
                b = 1;
                break;
            }
        }
        for(int i = x; i < x+s; i++){
            if(iArray[y+s-1][i] != 0){
```

```
            b = 1;
            break;
        }
    }
    if(b) break;
  }
  s--;
  return SquareData{Coord{x, y}, s}; // максимально возможный для
вмещения квадрат
}

SquareData Square::findSquare_(int color, int** Array){
  int x = -1;
  int y = -1;
  for(int i = 0; i < iSize; i++){ //находим верхний левый угол с заданным
цветом
    bool b = 0;
    for(int j = 0; j < iSize; j++){
      if(Array[i][j] == color){
        y = i;
        x = j;
        b = 1;
        break;
      }
    }
    if(b) break;
  }
  if(x == -1 && y == -1)
    return SquareData{Coord{0, 0}, 0};
  int s;
  for(s = x; s < iSize; s++){  // находим размер квадрата
    if(Array[y][s] != color)
      break;
  }
  s -= x;
  return SquareData{Coord{x, y}, s};
}

void Square::rec_backtrack(int iCurSize){
  if(iCurSize >= res)  // если количество уже найденных квадратов больше
чем
                  // лучшая конфигурация =>
```

```
        return;
    auto emptySquare = findPotentialSquare();
    if(emptySquare.w == iSize){
        setSquare(0, 0, iSize/2+1);
        setSquare(0, iSize/2+1, iSize/2);
        setSquare(iSize/2+1, 0, iSize/2);
        rec_backtrack(3);
        return;
    }
    if(emptySquare.w == 0){   // ура!, мы нашли какую-то конфигурацию
        if(iCurSize < res){
            res = iCurSize;
            setConfiguration();
        }
    }
    else{
        int w = emptySquare.w;
        while(w > 0){
            setSquare(emptySquare.pos.x, emptySquare.pos.y, w);  // записываем
цифры в найденный квадрат
            rec_backtrack(iCurSize+1);
            delSquare(emptySquare.pos.x, emptySquare.pos.y); // удаляем квадрат
текущего этапа рекурсии
            w--; // уменьшаем размер вставляемого квадрата
        }
    }
}

int Square::rec_backtrack(int iCurSize, SquareList* list){
    int find_flag = 0;
    if(iCurSize >= res)  // если количество уже найденных квадратов больше
чем
                    // лучшая конфигурация =>
        return find_flag;
    auto emptySquare = findPotentialSquare();
    if(emptySquare.w == 0){   // ура!, мы нашли какую-то конфигурацию
        //проверка подходит ли нам эта конфигурация
        bool f = true;
        for(int j = 0; j<list->count; j++){
            int flag = 0;
            for(int i = 1; i <= iColors; i++){
                auto sq = findSquare_(i, iArray);
```

```
                if(list->arr[j] == sq.w){
                    flag = 1;
                    break;
                }
            }
            if(flag == 0){
                f = false;
                return find_flag;
            }
        }
        if(iCurSize < res && f == true){
            res = iCurSize;
            setConfiguration();
            find_flag = 1;
        }
    }
    else{
        int w = emptySquare.w;
        while(w > 0){
            setSquare(emptySquare.pos.x, emptySquare.pos.y, w);  // записываем
цифры в найденный квадрат
            find_flag = rec_backtrack(iCurSize+1, list);
            if(find_flag == 1)
                return find_flag;
            delSquare(emptySquare.pos.x, emptySquare.pos.y); // удаляем квадрат
текущего этапа рекурсии
            w--; // уменьшаем размер вставляемого квадрата
        }
    }
    return find_flag;
}

int Square::getSize() const {
    return iSize;
}

void Square::setSquare(int x, int y, int w){
    for(int i = y; i < y + w && i < iSize; i++){
        for(int j = x; j < x + w && j < iSize; j++){
            iArray[i][j] = iColors+1;
        }
    }
```

```
    iColors++;
}

void Square::setConfiguration(){  //записываем нынешний вариант, как лучший
    for(int i = 0; i < iSize; i++){
        for(int j = 0; j < iSize; j++){
            iBestConfiguration[i][j] = iArray[i][j];
        }
    }
}

void Square::printConfiguration(std::ostream &os){
    os << iColors << std::endl;
    for(int i = 1; i <= iColors; i++){
        auto sq = findSquare_(i, iBestConfiguration);
        os << sq;
    }
}

void Square::delSquare(int x, int y){
    int color = iArray[y][x];
    for(int i = y; i < iSize; i++){
        for(int j = x; j < iSize; j++){
            if(iArray[i][j] == color)
                iArray[i][j] = 0;
            else break;
        }
    }
    for(int i = 0; i < iSize; i++){
        for(int j = 0; j < iSize; j++){
            if(iArray[i][j] > color)
                iArray[i][j]--;
        }
    }
    iColors--;
}

void read_sizes(int size, SquareList *list){
    std::cout << "Введите количество квадратов, которые необходимо разместить:" << std::endl;
    std::cin >> list->count;
```

```
    if(list->count <= 0)
        return;
    std::cout << "Введите размеры квадратов, которые необходимо
разместить:" << std::endl;
    list->arr = new int[list->count];
    for(int i = 0; i<list->count; i++){
        std::cin >> list->arr[i];
        if(list→arr[i]>=size){
            list->count = -1;
            return;
         }
    }
}

int main(){
    int size, flag;
    SquareList *list = new SquareList;
    std::cout << "Введите размер столешницы:" << std::endl;
    std::cin >> size;
    read_sizes(size, list);
    if(list->count < 0){
        std::cout << "Невозможно заполнить столешницу заданными
квадратами!" << std::endl;
        return 0;
    }
    else if(list->count == 0){
        if(size == 0 || size == 1)
            std::cout << 0 << std::endl;
        else{
            Square square(size);
            square.proceed();
            square.printConfiguration(std::cout);
        }
    }
    else{
        Square square(size);
        flag = square.proceed(list);
        if(flag == 1){
            std::cout << "Решение:" << std::endl;
            square.printConfiguration(std::cout);
         }
        else
```

```
            std::cout << "Невозможно заполнить столешницу заданными квадратами!" << std::endl;
    }
    return 0;
}
```