

**МИНОБРНАУКИ РОССИИ**  
**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**  
**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**  
**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**  
**Кафедра математического обеспечения и применения ЭВМ**

**ОТЧЕТ**  
**по практической работе №1**  
**по дисциплине «Построение и анализ алгоритмов»**  
**Тема: Поиск с возвратом**

Студентка гр. 8382

\_\_\_\_\_

Ефимова М.А

Преподаватель

\_\_\_\_\_

Фирсов М.А.

Санкт-Петербург

2020

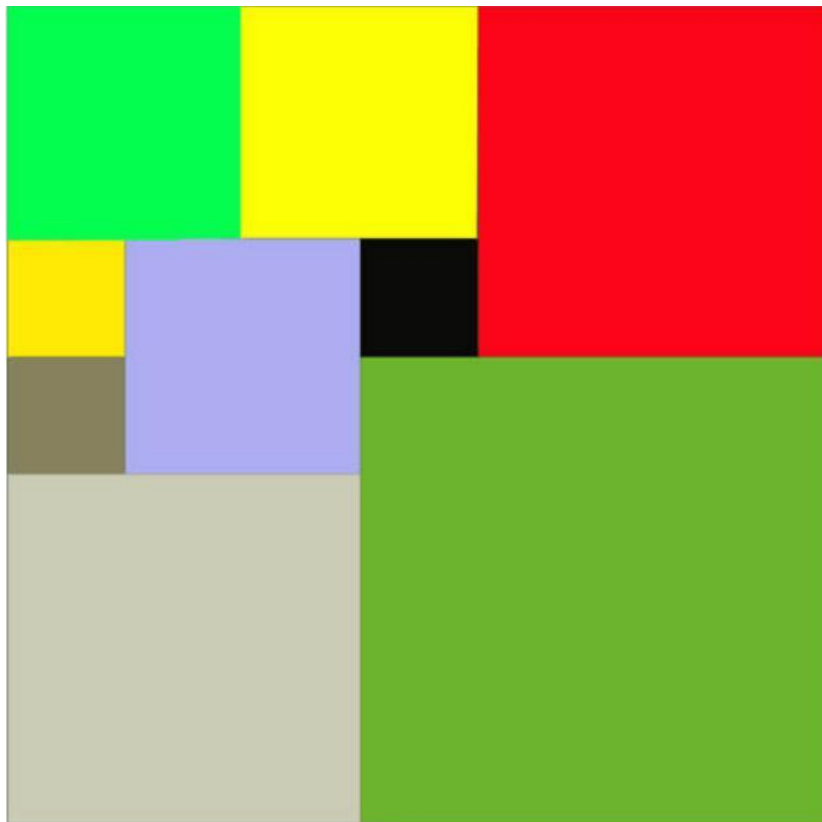
### **Цель работы.**

Ознакомиться с алгоритмом перебора с возвратом и научиться применять его на практике. Написать программу реализовывающую поиск с возвратом.

### **Постановка задачи.**

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  $N \times N$ . 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  $NN(2 \leq N \leq 20, 2 \leq N \leq 20)$ .

### **Выходные данные**

Одно число  $KK$ , задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу (квадрат) заданного размера  $NN$ . Далее должны идти  $KK$  строк, каждая из которых должна содержать три целых числа  $x, y, w$ , задающие координаты левого верхнего угла ( $1 \leq x, y \leq N, 1 \leq x, y \leq N$ ) 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

#### **Индивидуальное задание.**

Вар. 2р

Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата

## Анализ сложности алгоритма.

Временная сложность алгоритма определяется количеством входных данных. В данном алгоритме ключевым моментом является перебор делителей вводимого числа  $n$ .

Для упрощения предположим, что введено значение  $n$ , представив его в виде простого числа и составного:

$$n = mk.$$

Рассматриваем алгоритм без

- Если  $m = 0$ , мы называем его один раз, но функция никогда не вызывает себя рекурсивно.
- Если  $m = 1$ , функция вызывается один раз, а затем мы назовем ее рекурсивно  $k$  раз (из-за цикла `int j`), но каждый раз, когда мы ее вызываем, она будет вызываться в подсписке размера 0.
- Если  $m = 2$ , функция вызывается один раз, а затем функция вызывает себя рекурсивно  $k$  раз.
- Если  $m = 3$ , он вызывает себя рекурсивно  $k$  раз с подслоем размера 2, опять же с подписанием размера 1, снова с подпиской размера 0. Таким образом, мы можем видеть шаблон.

Если рассматривать алгоритм с оптимизацией, то при  $m::2$ ,  $m::3$ , то сложность отдельно данных случаев будет  $O(1)$ , так как в программе вызывается константа и выводится на экран.

При больших  $m$  получаем:

Если обозначить  $f(m)$  как общее число раз, когда начальный размер равен  $m$ , то:

$$f(0) = 1$$

$$f(m) = 1 + k * f(m-1) + k * f(m-2) + \dots + k * f(0)$$

который

$$f(0) = 1$$

$$f(1) = 1 + k$$

$$f(2) = 1 + 2k + k^2 = \text{const}$$

$$f(3) = 1 + 3k + 3k^2 + k^3 = \text{const}$$

или

$$f(m) = (k + 1)^m$$

Получаем формулу :  $O((k + 1)^m)$ . Если будем рассматривать один элемент, так что  $k = 1$  и  $n = m$ , то число рекурсивных вызовов равно  $O(2^n)$ .

### **Сложность алгоритма.**

Сложность алгоритма  $O(2^N)$ . Объяснить это можно тем, что для каждого маленького квадрата существует 2 варианта размещения : либо он ставиться в большой квадрат, либо нет.

Если рассмотреть разбиение всех квадратов длины от 2 до 40, то можно вывести следующее утверждение: минимальному разбиению (Напр.: 6, 21, 33) будет соответствовать разбиение квадрата с длиной стороны равной наименьшему целочисленному делителю числа  $N$  не равному единице.

### **Описание алгоритма.**

Сам квадрат представлен в виде двумерного массива `square`.

С помощью функции `Intialization` мы инициализируем значениями простых чисел. Если  $N$  - простое число, то берем значения  $N/2$ ,  $N/2 + 1$ . Далее мы инициализируем память и с помощью функции `void search_first_elem(int** square, int &x_new, int &y_new, int N)` находим свободное место для квадрата с координатами  $x$  и  $y$ . Заполняем квадрат и находим новое ближайшее место.

Далее мы рассматриваем отдельные случаи, представляя наше число ввода, как простое число или непростое число. Если остаток от деления нашего числа на два равно нулю, то результатом будет являться число 4. Если же остаток от деления нашего числа на три равно нулю, то результатом будет являться число 6.

Поиск свободного места для квадрата (по координатам) мы находим с помощью функции `void search_first_elem(int** square, int &x_new, int &y_new, int N)`. Мы проходим по всему нашему двумерному массиву и находим первое свободное места, начиная от левого верхнего угла.

Вывод результатов на основе заполнения матрицы мы производим с помощью функции `void setPosition`.

В самой рекурсии происходит следующее:

Проверка на достижение предельной глубины рекурсии. Если это предел, то возвращается -1. Осуществляется поиск пустой клетки в массиве arr. Если таковой не оказалось, то возвращается 0.

Далее происходит нахождение максимальной длины квадрата, который можно поместить, начиная с данной точки левого верхнего угла. Пробуются разные длины квадрата.

Возвращенное значение количества квадратов проверяется на минимальное и, если это так, запоминаем длину текущего квадрата. Далее очищаем квадрат. И переходим к следующей длине квадрата.

### **Оптимизация.**

Оптимизация алгоритма заключается в использовании двух функций:

- 1) Когда вводимое число кратно двум
- 2) Когда вводимое число кратно трем

В первом случае решением нашей задачи будет число 4. Во втором случае решением нашей задачи будет число 6.

Оптимизация заключается в том, что при делении на наше вводимое число мы можем каждый раз делить на  $2 \cdot n$  квадратов.

### **Описание функций.**

**void Intialization(int\*\* square,int N)** – инициализация значениями простых чисел. Если N - простое число, то берем значения  $N/2$ ,  $N/2 + 1$

**void Initialize(int\*\* square,int N,int number1,int number2)** – проход по квадратам

**void copy(int\*\*square,int\*\* best\_square,int N)** – сохранение наилучшего расположения квадрата

**void printBoard(int\*\* best\_square,int N)** – вывод текущего положения квадрата на экран

**void clear(int\*\* square,int\*\* best\_square,int N)** – освобождение памяти после вывода квадратов на экран

**void remove\_numbers(int\*\* square,int N,int curr\_square,int x\_start,int y\_start)**

– когда мы получили разделение, то мы возвращаемся к тому моменту, когда квадрат был меньше. Рекурсивно возвращаемся к изначальному номеру квадрата, обнуляю его и все последующие квадраты.

**void search\_first\_elem(int\*\* square,int &x\_new,int &y\_new,int N)** – поиск свободного места для квадрата с координатами x и y. Заполнили квадрат и надо найти новое ближайшее место.

**void memoryInitialization(int\*\*square,int\*\*best\_square,intN)**–

инициализация памяти

**void printMultipleTwo (int N)** – когда кратно двум, то ответ всегда будет 4

**void printMultipleThree(int N)** - когда кратно трем, то ответ всегда будет

6 Далее высчитываем координаты по формуле

**void printResult(int\*\* best\_sq,int N,int min)** – получаем массив чисел, по массиву чисел находим координаты. Из левого верхнего угла находим первый квадрат и тд.

**void setPosition(int\*\* square,int\*\* best\_square\_version,int N,int x,int y,int number\_of\_squares,int &min,deque<pair<int,int>> &components\_of\_square, bool &forward)** - вывод результатов на основе заполнения матрицы. Строим максимальный квадрат, далее уменьшаем каждый раз квадрат на единицу и меняем остальные. Координаты хранят x и y.

### Тестирование.

Таблица 3 – Тестирование

| № теста | Тест | Результат                             |
|---------|------|---------------------------------------|
| 1       | 2    | 4<br>1 1 1<br>1 2 1<br>2 1 1<br>2 2 1 |
| 2       | 6    | 4                                     |

|   |    |   |
|---|----|---|
|   |    | 1 1 3<br>1 4 3<br>4 1 3<br>4 4 3  |
| 3 | 13 | 11<br>1 1 7<br>1 8 6<br>8 1 6<br>7 8 1<br>7 9 3<br>7122<br>8 7 2<br>9122<br>1074<br>10111<br>11113              |
| 5 | 37 | 37<br>15<br>1119<br>12018<br>20118<br>19 20 11<br>9213<br>19247<br>19317<br>20192<br>22195<br>26242<br>26 26 12 |



|   |    |   |
|---|----|---|
| 6 | 25 | 8<br>1 1 5<br>1 6 5<br>1115<br>11610<br>6110<br>6115<br>11 11 15<br>16110 |
| 7 | 33 | 6<br>1111<br>11211<br>12311<br>12111<br>12 12 22<br>23111                 |
|   |    |   |

### Исследование времени.

| Ввод | Вывод (сек) |
|------|-------------|
| 1    | 0           |
| 6    | 0           |
| 13   | 0           |
| 29   | 0.33        |

|    |      |
|----|------|
| 31 | 0.69 |
| 37 | 0.76 |
| 45 | 0    |

График исследования памяти от времени.

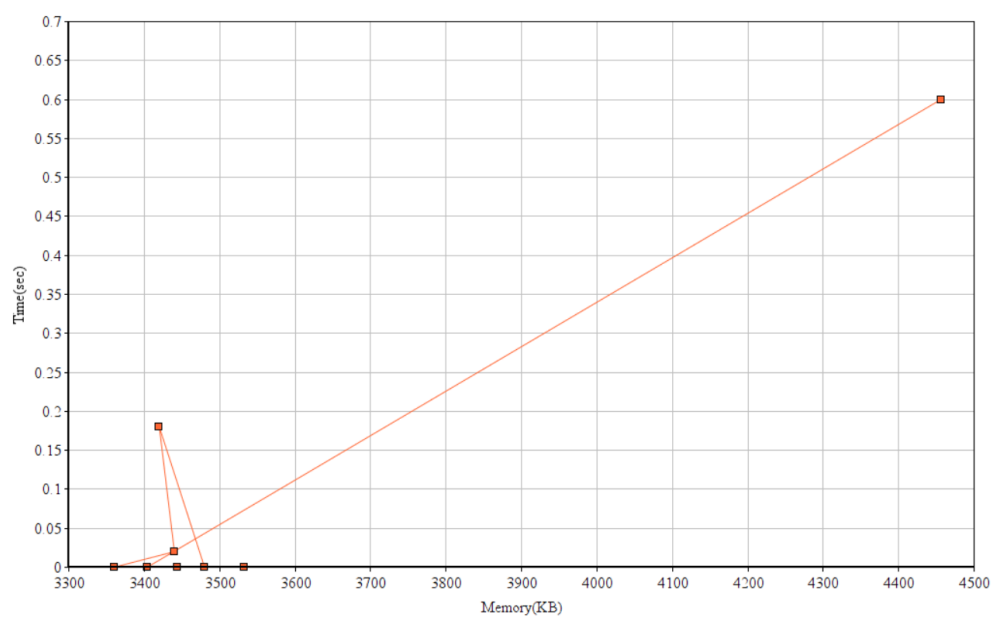
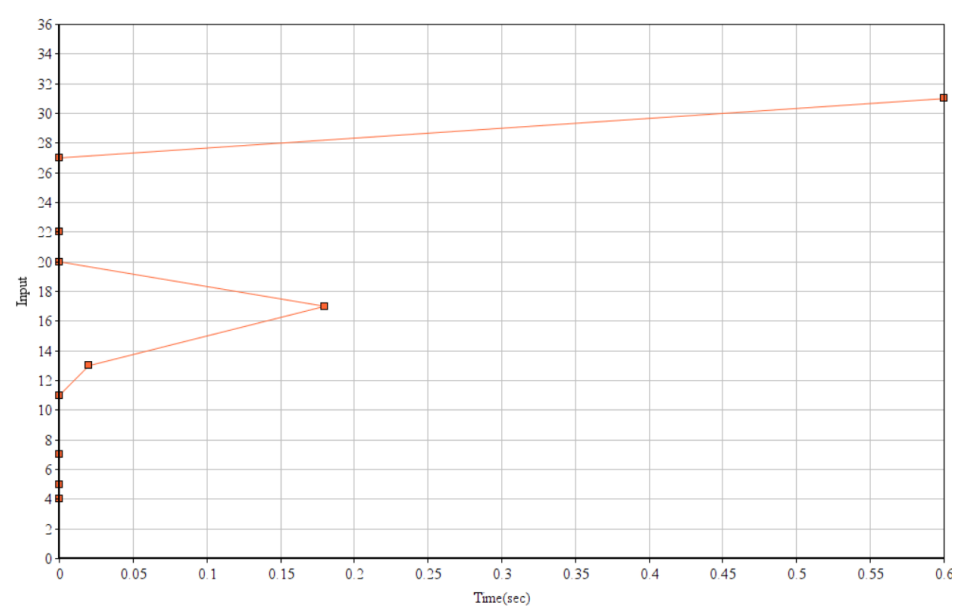
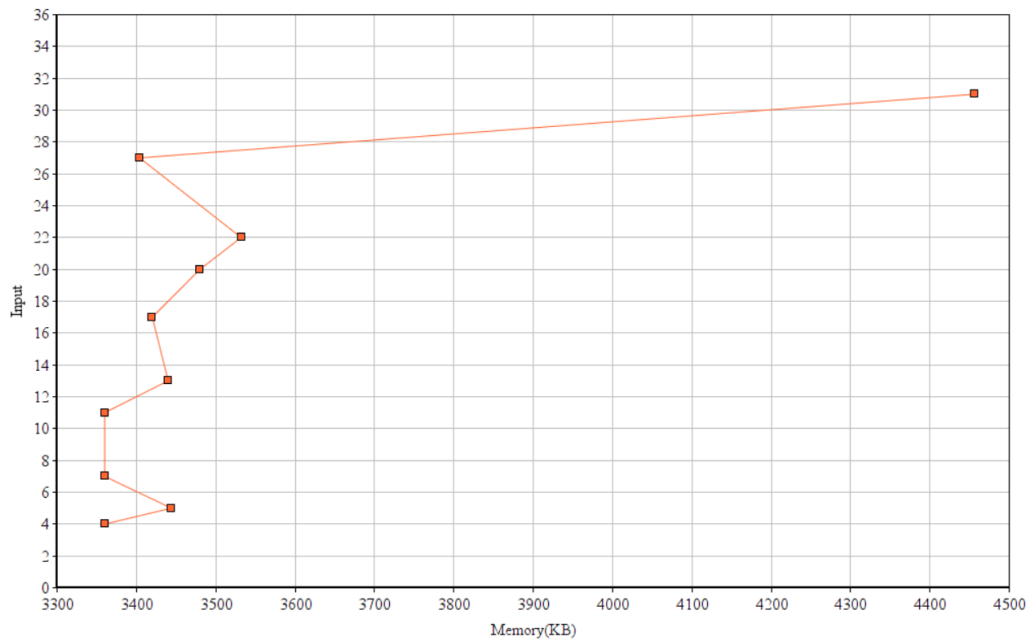


График исследования времени от данных.



## График исследования памяти от данных.



Пример вывода программы по времени и памяти:

```
Result
CPU Time: 6.64 sec(s), Memory: 4456 kilobyte(s)

15
1 1 16
1 17 15
17 1 15
17 16 3
16 17 1
16 18 1
16 19 7
16 26 6
20 16 3
22 26 3
22 29 3
23 16 9
23 25 1
24 25 1
25 25 7
Time: 6653sec
```

## Частичные решения.

Решения храним в стеке и сравниваем с минимальным разбиением квадратов. В рекурсивную функцию подаем массив квадрата, массив наилучшего квадрата, координаты, сторону квадрата, и переменную, которая отвечает за направление прогона по нашему квадрату.

### **Вывод.**

В ходе выполнения лабораторной работы был изучен и применен на практике алгоритм перебора с возвратом. Была реализована программа, находящая разбиение прямоугольного поля минимально возможным количеством квадратов и подсчитывающая количество вариантов минимального разбиения. Полученный алгоритм обладает экспоненциальной сложностью по операциям.

## ПРИЛОЖЕНИЕ А

### Исходный код программы

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

using namespace std;

void Initialization(int** square,int N){//инициализация нулями изначальных
квдратов
for(int i = 0;i < N / 2 + 1;i++)
for(int j = 0;j < N / 2 + 1;j++)
square[i][j] = 1;
for(int i = 0;i < N / 2;i++)
for(int j = N / 2 + 1;j < N;j++)
square[i][j] = 2;
for(int i = N / 2 + 1;i < N;i++)
for(int j = 0;j < N / 2;j++)
square[i][j] = 3;
square[N / 2 + 1][N / 2] = 4;
}
void Initialize(int** square,int N,int number1,int number2){
for(int i = 0;i < number1;i++)
for(int j = 0;j < number1;j++)
square[i][j] = 1;
for(int i = 0;i < number2;i++)
for(int j = number1;j < N;j++)
square[i][j] = 2;
for(int i = number2;i < number2 * 2;i++)
for(int j = number1;j < N;j++)
square[i][j] = 3;
for(int i = number1;i < N;i++)
for(int j = 0;j < number2;j++)
square[i][j] = 4;
square[number1][number2] = 5;
//printBoard(square,N);
}
void copy(int**square,int** best_square,int N){//сохранение наилучшего
расположения квадратов
for (int i = 0; i < N ; i++) {
for(int j = 0;j < N;j++){
best_square[i][j] = square[i][j];
}
}
}

void printBoard(int** best_square,int N){//вывод текущего положения квадратов на
экран
cout << "----- " << "\n";
```

```

for(int i = 0;i < N;i++){
for(int j = 0;j < N;j++)
cout << best_square[i][j] << " ";
printf("\n");
}
}

```

```

void clear(int** square,int** best_square,int N){//освобождение памяти по
завершению работы алгоритма
for(int i = 0;i < N;i++){
delete square[i];
delete best_square[i];
}
}

```

```

void remove_numbers(int** square,int N,int curr_square,int x_start,int y_start){
for(int i = 0;i < N;i++){
for(int j = 0;j < N;j++){
if(square[i][j] > curr_square || (square[i][j] == curr_square && (i > x_start || j >
y_start)))
square[i][j] = 0;
}
}
}

```

```

void search_first_elem(int** square,int &x_new,int &y_new,int N){//функция поиска
свободного места для квадрата
x_new = -1;
y_new = -1;
for (int i = 0; i < N; i++) {
for (int j = 0; j < N; j++) {
if (!square[i][j]) {
x_new = i;
y_new = j;
break;
}
}
}
if (x_new != -1)
break;
}
}
84.

```

```

void memoryInitialization(int** square,int** best_square,int N){
for(int i = 0;i < N;i++) {
square[i] = new int[N];
best_square[i] = new int[N];
}
}
void printMultipleTwo(int N){
cout << 4 << "\n";
cout << 1 << " " << 1 << " " << N / 2 << "\n";
}

```

```

cout << 1 << " " << N / 2 + 1 << " " << N / 2 << "\n";
cout << N / 2 + 1 << " " << 1 << " " << N / 2 << "\n";
cout << N / 2 + 1 << " " << N / 2 + 1 << " " << N / 2 << "\n";
}
void printMultipleThree(int N){
cout << 6 << "\n";
cout << 1 << " " << 1 << " " << N - N/3 << "\n";
cout << 1 << " " << N - N/3 + 1 << " " << N / 3 << "\n";
cout << N / 3 + 1 << " " << N - N/3 + 1 << " " << N / 3 << "\n";
cout << N - N/3 + 1 << " " << 1 << " " << N / 3 << "\n";
cout << N - N/3 + 1 << " " << N/3 + 1 << " " << N / 3 << "\n";
cout << N - N/3 + 1 << " " << N - N/3 + 1 << " " << N / 3 << "\n";
}
void printResult(int** best_sq,int N,int min){//вывод результатов на основе
заполнения матрицы
int x,y,len;
bool find;
for(int k= 1;k <= min;k++){
x = 0;
len = 0;
y = 0;
find = false;
for(int i = 0;i < N;i++){
for(int j = 0;j < N;j++){
if(find){
if(i + 1 >= N || j + 1 >= N || best_sq[i + 1][j + 1] != k) {
len = i - x + 2;
break;
} else
i++;
}
if(!find && best_sq[i][j] == k){
find = true;
x = i + 1;
y = j + 1;
if(i + 1 >= N || j + 1 >= N || best_sq[i + 1][j + 1] != k){
len = 1;
break;
}
}
i++;
}
}
if(len)
break;
}
cout << x << " " << y << " " << len << "\n";
}
}

```

```

void setPosition(int** square,int** best_square_version,int N,int x,int y,int
number_of_squares,int &min,deque<pair<int,int>> &components_of_square, bool
&forward) {
int x_new = -1;
int y_new = -1;
unsigned long size_of_queue = components_of_square.size();
bool check = true;
pair<int, int> coordinates;
vector<pair<int, int>> curr_components;
for (int i = 0; i < size_of_queue; i++) { //проверка на то, можно ли на
свободных позициях увеличить квадрат
coordinates = components_of_square.front();
components_of_square.pop_front();
x_new = coordinates.first;
y_new = coordinates.second;
if (((x_new + 1 < N && y_new + 1 < N && number_of_squares != 1) ||
(x_new + 1 < N - 1 && y_new + 1 < N - 1)) &&
(!square[x_new + 1][y_new + 1] || square[x_new + 1][y_new + 1] ==
number_of_squares) &&
(!square[x_new + 1][y_new] || square[x_new + 1][y_new] ==
number_of_squares)
&& (!square[x_new][y_new + 1] || square[x_new][y_new + 1] ==
number_of_squares)) {
if (!square[x_new + 1][y_new]) {
components_of_square.emplace_back(make_pair(x_new + 1, y_new));
curr_components.emplace_back(make_pair(x_new + 1, y_new));
}
if (!square[x_new][y_new + 1]) {
components_of_square.emplace_back(make_pair(x_new, y_new + 1));
curr_components.emplace_back(make_pair(x_new, y_new + 1));
}
if (!square[x_new + 1][y_new + 1]) {
components_of_square.emplace_back(make_pair(x_new + 1, y_new + 1));
curr_components.emplace_back(make_pair(x_new + 1, y_new + 1));
}
} else
check = false;
}
if (check) { //если можно. рекурсивно запускаем процесс для следующей вершины
for (int i = 0; i < curr_components.size(); i++) {
coordinates = curr_components[i];
x_new = coordinates.first;
y_new = coordinates.second;
square[x_new][y_new] = number_of_squares;
}
//printBoard(square,N);
setPosition(square, best_square_version, N, x + 1, y + 1, number_of_squares, min,
components_of_square,
forward);
} else {

```



```

search_first_elem(square, x_new, y_new, N); //если нельзя, ищем свободную позицию
для построения следующего квадрата
}
if (x_new != -1) {
if (!forward) { //вернувшись до ближайшего не единичного квадрата, изменяем
его размер и продолжаем построение
if(number_of_squares + 1 >= min)
return;
remove_numbers(square, N, number_of_squares, x, y);
search_first_elem(square, x_new, y_new, N);
forward = true;
}
forward = true;
components_of_square.resize(0);
components_of_square.emplace_back(make_pair(x_new, y_new));
square[x_new][y_new] = number_of_squares + 1;
if(number_of_squares + 1 < min) {
setPosition(square, best_square_version, N, x_new, y_new, number_of_squares
1, min, components_of_square,
forward);
}
else {
forward = false;
}
} else {
if (number_of_squares < min) { //если все клетки заняты, возвращаемся до
ближайшего не единичного квадрата
min = number_of_squares;
copy(square, best_square_version, N); //если к-ство квадратов при данной
конфигурации минимально, сохраняем текущее расположение
// cout << "Печать наилучшего на данное время решения" << "\n";
//printBoard(best_square_version,N);
}
forward = false;
}
}
}

void menu(int N,int** square,int** best_square_version){
bool forward = true;
deque<pair<int,int>> components_of_square;
int min = (N * N) - (N - 1) * (N - 1) + 1;
memoryInitialization(square,best_square_version,N);
if((N % 2 && N % 5 && N % 3) || (N == 3) || (N == 5)) {
components_of_square.emplace_back(make_pair(N / 2 + 1,N / 2));
Initialization(square,N);
setPosition(square, best_square_version, N, N / 2 + 1, N / 2, 4, min,
components_of_square, forward);
}
else
if(!(N % 2))
printMultipleTwo(N);

```

```

else
if(!(N % 3))
printMultipleThree(N);
else
if(!(N % 5)){
int coord1 = N - N / 5 - N / 5;
int coord2 = N - coord1;
Initialize(square, N, coord1, coord2);
components_of_square.emplace_back(make_pair(coord1, coord2));
setPosition(square, best_square_version, N, coord1, coord2, 5, min,
components_of_square, forward);
cout << min << "\n";
printResult(best_square_version, N, min);
}if((N % 2 && N % 5 && N % 3) || (N == 3) || (N == 5)) {
cout << min << "\n";
printResult(best_square_version, N, min);
}
}

int main() {
srand(time(0));
time_t start;
time(&start);
start = clock();
deque<pair<int,int>> components_of_square;
int N;
cin >> N;
int** square = new int*[N];//массив для текущей конфигурации
int** best_square_version = new int*[N];//массив для наилучшей
конфигурации.
menu(N,square,best_square_version);
clear(square,best_square_version,N);
time_t end;
end = clock();
cout << "Time : "
std::cout << (end - start)/1000 << "sec\n";
return 0;
}

```

### Ссылки на тестирование

1. <https://ideone.com/jJC9E>
2. <https://ideone.com/XfsLDM>
3. <https://ideone.com/SN0fOV>