

**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**

Студентка гр. 9382

Балаева М.О.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2021

### **Цель работы.**

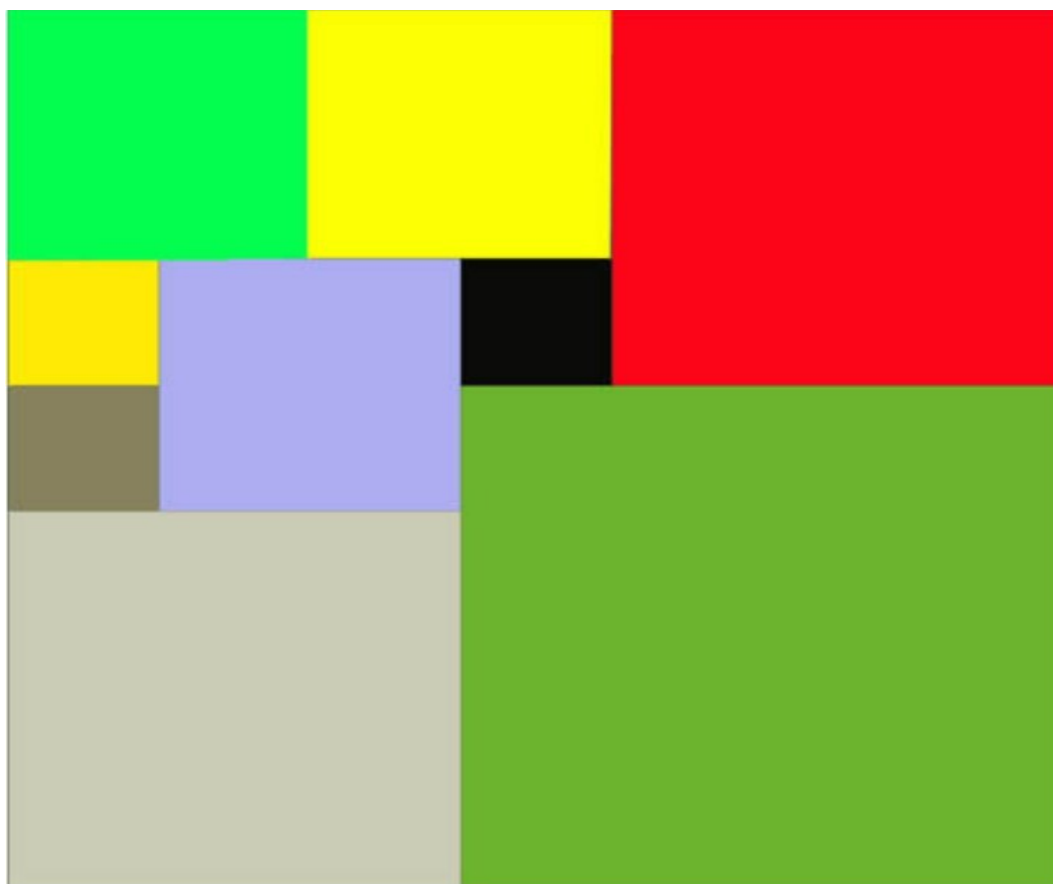
Реализовать программу, основанную на рекурсивном бэктрекинге. Исследовать время выполнения алгоритма от параметра, прописанного в задании., проследить зависимость количества операций для решения поставленной задачи от входных данных.

### **Задание.**

**Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения 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  $7 \times 7$  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

## Описание алгоритма.

В начале проверяются входные данные, в зависимости от их значения, ответ и размещение «вложенных» квадратов сразу же записывается (Использованные оптимизации пункт 2), или размещаются начальные 3 квадрата. Далее с помощью алгоритма Бэктрекинга находится минимальное количество «вложенных» квадратов и их размещение.

Создается матрица  $N$  на  $N$ , в которой натуральными числами будут отмечаться где (то есть верхний левый угол) и какой длины вставлен квадрат. Далее происходит поиск свободного места и определяется максимальная длина квадрата, который будет размещен в найденной области. При достижении условия возврата, то есть если текущее количество вложенных квадратов не является минимальным или было найдено количество меньше, возвращаемся к последнему квадрату, размер которого не равен 1, его длина уменьшается на 1, и строятся последующие комбинации. Если в процессе выполнения алгоритма будет найдено число квадратов меньше текущего минимального, оно заменится на новое. Когда будет найдено минимальное количество «вложенных» квадратов, записывается это число и разбиение для него, алгоритм завершится.

## Использованные оптимизации.

- Матрицу перебора изначально можно заполнить на 75% тремя квадратами размеров  $N//2$ ,  $N//2 - 1$  соответственно, то поиск свободной клетки, куда можно поставить квадрат, можно осуществлять только в оставшихся 25% квадрата.
- Квадрат с четной стороной имеет постоянное решение – 4 квадрата. Для квадратов наименьший простой множитель, которых равен трем, не производится перебор, а сразу выводится ответ 6. Поэтому можно не осуществлять перебор для таких квадратов, а сразу выводить ответ.

- Сжатие квадрата. Квадрат с размером  $N$ , можно сжать до размера значения наименьшего простого делителя числа  $N$ .
- Поскольку 75% квадрата заполнены, то максимальный размер квадрата, который можно поставить в матрицу перебора –  $N // 2$ .

### Описание функций и структур данных.

Класс Table – класс, предназначенный для выполнения поставленной задачи. Поля класса:

1. size – длина стороны квадрата.
2. `std::vector<std::vector<int>>` table – матрица квадрата.
3. Count – переменная, показывающая количество “вложенных” квадратов.

`void constTable()` - Метод класса Table, который в зависимости от длины квадрата вызываются методы `best_` или `insertTable`. `best_` вызывается, если сторона кратна 2 или 3, `insertTable` вызывается в остальных случаях. Если длина квадрата не кратна ни 2, ни 3, то происходит размещение 3х гарантированных квадратов (Использованные оптимизации пункт 1).

`int getnumber()` - метод класса Table, возвращающий количество расположенных “вложенных” квадратов.

`void best_(int k)` — метод класса Table, записывает количество «вложенных» квадратов для случаев, когда сторона кратна 2 или 3.

Аргументы:

`int k` — число принимающее значения 2 или 3 в зависимости от того, которому из этих чисел кратна сторона исходного квадрата ( $k = 2$ , при длине стороны кратной 2 и  $k = 3$ , при длине стороны кратной 3).

`int getSize()` - метод класса Table, возвращающий длину стороны квадрата

`bool isPossible(int i, int j, int n)` – метод класса Table, показывающий можно ли разместить еще один квадрат, это необходимо для выполнения алгоритма.  $i$  и  $j$  являются координатами верхнего левого угла квадрата.

Аргументы:

1.  $i$  – координата по  $y$ .
2.  $j$  – координата по  $x$ .
3.  $n$  – длина рассматриваемого квадрата.

`void insertTable(int i, int j, int n)` – Метод, наносящий квадрат на карту, также считает количество “вложенных” квадратов.  $i$  и  $j$  являются координатами верхнего левого угла квадрата.

Аргументы:

1.  $i$  – координата по  $y$ .
2.  $j$  – координата по  $x$ .
3.  $n$  – длина рассматриваемого квадрата.

`bool checkSpace(int i)` – метод класса `Table`, показывающий есть ли на карте еще свободные места.

Аргументы:

1.  $i$  – координата по  $y$ .

`int findi(int i)` - метод , возвращающий координату по  $y$ .

Аргументы:

1.  $i$  – координата по  $y$ .

`int findj(int i)` - метод , возвращающий координату по  $x$ .

Аргументы:

1.  $i$  – координата по  $y$

`void deleteTable(int i, int j)` — метод , удаляющий( «зануляющий») матрицу.

`void result()` - метод , выводящий результат.

`void shower_table` — метод класса `Table`, заполняющий итоговую карту, в зависимости от значения длины исходного квадрата.

`Table backTracking(Table table, int i, int j, int pr)`— основная рекурсивная функция необходимая для реализации алгоритма , на вход , которой подается экземпляр класса `Table`, хранящий в себе необходимые данные, то есть `size` – размер текущего квадрата, `std::vector<std::vector<int>>` `table` — матрица, в которой будет содержаться заполнение «вложенными квадратами» , `count` – переменная , отвечающая за текущее количество вложенных квадратов. `int i, int j` — координаты верхнего левого угла квадрата. `int pr` — счетчик нужный для корректной печати промежуточных данных. `Table table, int i, int j` — аргументы, передача, которых необходима для корректного и однозначного определения квадрата, участвующего в алгоритме. Функция возвращает экземпляр класса `Table` – `best`, с зафиксированными:

- 1) Наименьшим количеством «вложенных» квадратов - `bestNum`

2) Лучшим заполнением матрицы.

В `main()` производится проверка выделенных случаев( наименьшие делители 2 и 3 соответственно), вызов всех необходимых функций и методов.

### **Оценка сложности алгоритма по времени.**

Поскольку используется довольно большое количество оптимизаций, посчитать точную сложность алгоритма сложно, поэтому произведем оценку алгоритма сверху.

$N$  – длина стороны квадрата. Имеется  $N^2$  свободных клеток, также  $N$  размеров квадрата, которые будем перебирать. Таким образом, получаем, что сложность алгоритма по времени равна  $O((N^2)! * N^N)$ .

### **Оценка сложности алгоритма по памяти.**

Матрица квадрата, хранящаяся в экземпляре класса `Table`, при каждом рекурсивном проходе копируется, поэтому мы возьмем максимальное количество единичных квадратов в матрице, оно равняется  $N*N$  и умножается на количество рекурсивных проходов. В процессе рекурсивного прохода, скопированные экземпляры класса удаляются, поэтому за максимум можно считать проход по матрице —  $N*N$ . Следовательно сложность алгоритма по памяти —  $O(N^4)$ .

## Тестирование.

Таблица 1. Результаты работы программы

№ попытки	Входные данные	Выходные данные без промежуточного вывода
1	3	6 1 1 2 1 3 1 2 3 1 3 1 1 3 2 1 3 3 1
2	5	8 1 1 3 1 4 2 3 4 1 3 5 1 4 1 2 4 3 1 4 4 2 5 3 1
3	2	4 1 1 1 2 1 1 1 2 1 2 2 1
4	9	6 1 1 6 1 7 3 4 7 3 7 1 3 7 4 3 7 7 3
5	11	11 1 1 6 1 7 5



		6 7 1 6 8 1 6 9 3 7 1 5 7 6 1 7 7 2 8 6 1 9 6 3 9 9 3
6	12	4 1 1 6 7 1 6 1 7 6 7 7 6
7	19	13 1 1 10 1 11 9 10 11 1 10 12 1 10 13 2 10 15 5 11 1 9 11 10 2 11 12 1 12 12 3 13 10 2 15 10 5 15 15 5

### **Исследование.**

В данном варианте необходимо исследовать зависимость времени от размера квадрата, чтобы это сделать посчитаем время выполнения алгоритма для каждой длины стороны квадрата(от 2 до 20) .

Результаты времени выполнения алгоритма от размера главного квадрата представлены в Таблице 2.

Таблица 2. Зависимость времени от размера квадрата.

Длина стороны квадрата(N)	Время (с)
2	0.000118
3	0.000331
4	0.000147
5	0.000956
6	0.000151
7	0.002397
8	0.000134
9	0.000234
10	0.000154
11	0.021493
12	0.00013
13	0.045497
14	0.000127
15	0.000268
16	0.000141
17	1.338965
18	0.000124
19	3.03642
20	0.000128



Исходя из графика , можно сделать вывод, что из-за оптимизаций, время выполнения программы сокращается. Время выполнения программы при нечетных значениях растет экспоненциально, что видно из графика.

### **Выводы.**

В ходе работы были изучены методы бэктрекинга, написана программа для поиска минимального количества квадратов для заполнения заданного с помощью рекурсивного бэктрекинга, практически освоены решения по возможным оптимизациям и исследована зависимость времени выполнения алгоритма от размера квадрата.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

```
int bestNum = 0;
```

```
class Table // исходный квадрат
```

```
{
    int size;
    std::vector<std::vector<int>> table;
    int count;
```

```
public:
```

```
    Table(int size): size(size), table(size, std::vector<int>(size, 0)), count(0) {
        if(size != 0) {
            constTable();
        }
    }
```

```
    Table(Table const &other): size(other.size), table(other.size,
std::vector<int>(other.size, 0)), count(other.count) {
        for (int i = 0; i < size; i++)
            for (int j = 0; j < size; j++)
                table[i][j] = other.table[i][j];
    }
```

```
    Table& operator=(Table const &other){
        if(&other != this){
            Table tmp(other);
            count = tmp.count;
            size = tmp.size;
            table.swap(tmp.table);
        }
        return *this;
    }
```

```
    ~Table(){}
    void best_(int k){
```

```
        if( k==2){
            bestNum = 4;
        }
        else if ( k==3)
```

```

    {
        bestNum = 6;
    }
}

```

```

void constTable() {
    int temp = size/2;
    if(size%2==0){
        best_(2);
    }
    else if(size%3==0){
        best_(3);
    } else {
        insertTable(0, 0, temp + 1);
        insertTable(0, temp + 1, temp);
        insertTable(temp + 1, 0, temp);

    }
}

```

```

int getnumber() {
    return count;
}
int getsize() {
    return size;
}

```

```

bool isPossible(int i, int j, int n) {
    if((i + n) > size || (j + n) > size) {
        return false;
    }
    for(int y = i; y < i + n; y++)
        for (int x = j; x < j + n; x++)
            if(table[y][x] != 0) {
                return false;
            }
    return true;
}

```

```

void insertTable(int i, int j, int n) {
    for(int y = i; y < i + n; y++) {
        for (int x = j; x < j + n; x++) {
            table[y][x] = n;
            std::cout<<"The coordinates of the upper-left corner of the inserted
square: ( ";
            std::cout<<y<<" , "<<x<<)"<<std::endl;
            std::cout<<"the current side of the square: "<<n<<std::endl;

```

```

    }
}
++count;
std::cout<<"Current best partition:"<<count<<std::endl;
}
bool checkSpace(int i){
    for(int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if(table[y][x] == 0)
                return true;
        return false;
}
int findi(int i){
    for (int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if (table[y][x] == 0){
                return y;
            }
}
int findj(int i){
    for (int y = i; y < size; y++)
        for (int x = 0; x < size; x++)
            if (table[y][x] == 0){
                return x;
            }
}
void deleteTable(std::vector<std::vector<int>> &arr, int i, int j){
    int val = arr[i][j];
    for (int y = i; y < i + val; y++)
        for (int x = j; x < j + val; x++)
            arr[y][x] = 0;
}
std::vector<std::vector<int>> copyTable(std::vector<std::vector<int>> table) {
    std::vector<std::vector<int>> copy;
    for(int i = 0; i < size; i++){
        std::vector<int> v(table[i]);
        copy.push_back(v);
    }
    return copy;
}
void result(){
    std::vector<std::vector<int>> copy = copyTable(table);
    for(int i = 0; i < size; i++){
        for(int j = 0; j < size; j++){
            if(copy[i][j] != 0){
                std::cout << i + 1 << " " << j + 1 << " " << copy[i][j] << std::endl;
            }
        }
    }
}

```

```

        deleteTable(copy, i, j);
    }
}
}
}
void shower_table(){
    if(size%2==0){
        insertTable(0,0, size/2);
        insertTable(0,size/2, size/2);
        insertTable(size/2,size/2, size/2);
        insertTable(size/2,0, size/2);
    }
    else if(size%3==0){
        insertTable(0,0, (2*size)/3);
        insertTable((2*size)/3, 0 , size/3);
        insertTable(0, (2*size)/3, size/3);
        insertTable((2*size)/3, (2*size)/3, size/3);
        insertTable(size/3, (2*size)/3, size/3);
        insertTable((2*size)/3, size/3, size/3);
    }

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

```

Table best(0);

Table backTracking(Table table, int i, int j, int pr){

```

for(int n = table.getsize() / 2; n > 0; n--){
    for(int l=0; l < pr; l++)
        std::cout << " ";
    std::cout<<"The considered length of the square ="<<n<<std::endl;
    if(table.getnumber() > bestNum){
        for(int l=0; l < pr; l++)
            std::cout << " ";
        std::cout<<"The current option is not minimal"<<std::endl;
        return table;
    }
}

```

```

Table shape = table;
if(shape.isPossible(i, j, n)){
    for(int l=0; l < pr; l++)
        std::cout << " ";
    std::cout<<"established square:"<< n <<std::endl;
    shape.insertTable(i, j, n);
    if(shape.checkSpace(i)){
        for(int l=0; l < pr; l++)
            std::cout << " ";
        std::cout<<"Entering recursion"<<std::endl;
        shape = backTracking(shape, shape.findi(i), shape.findj(i),pr+1);
        shape.shower_table();
        for(int l=0; l < pr; l++)
            std::cout << " ";
        std::cout<<"Exiting recursion"<<std::endl;
    }
    else if(bestNum >= shape.getnumber()){
        best = shape;
        bestNum = shape.getnumber();
    } else{
    }
}
}
return table;
}

void res (int size, int k){

    if(k==2){
        int temp = size/2;
        std::cout << "4" << std::endl;
        std::cout << "1 1 " << temp << std::endl;
        std::cout << temp+1 << " 1 " << temp << std::endl;
        std::cout << "1 " << temp+1 << " " << temp << std::endl;
        std::cout << temp+1 << " " << temp+1 << " " << temp << std::endl;
    }
    else if(k==3){
        int temp = size/3;
        std::cout << "6" << std::endl;
        std::cout << "1" << " 1 " << temp*2 << std::endl;
        std::cout << "1 " << 1+temp*2 << " " << temp << std::endl;
        std::cout << 1+temp << " " << 1+temp*2 << " " << temp << std::endl;
        std::cout << 1+temp*2 << " 1 " << temp << std::endl;
        std::cout << 1+temp*2 << " " << 1+temp << " " << temp << std::endl;
        std::cout << 1+temp*2 << " " << 1+temp*2 << " " << temp << std::endl;
    }
}

```



```
}
```

```
int main(){
    int size = 0;
    std::cin >> size;
    clock_t start = clock();
    bestNum = size * size;
    Table A(size);
    A = backTracking(A, A.findi(0), A.findj(0),0);
    clock_t end = clock();
    std::cout << "Lead time: " << (double )(end - start) / CLOCKS_PER_SEC << "\n";
    if(size%2==0){
        res(size, 2);
    } else if(size%3==0){
        res(size, 3);
    } else {
        std::cout << "Best partition:"<< bestNum << std::endl;
        best.result();
    }
    if(size%2==0 || size%3==0 ){
        A.shower_table();
    }
    else{
        best.shower_table();
    }
    return 0;
}
```