

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. 8303

Пушпышев А.И.

Преподаватель

Фирсов М.А.

Санкт-Петербург

2020

Цель работы.

Научиться использовать алгоритм бэктрекинга на практическом примере по замещению квадратной столешницы заданной длины наименьшим числом квадратов.

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.

Алгоритм принимает длину стороны столешницы. Далее, если длина кратна двум, трём или пяти, то она заполняется уже заранее просчитанными значениями, вычисляется коэффициент кратности, затем заготовленный

результат (минимальное количество квадратов для столешниц со сторонами 2, 3 или 5) умножается на этот коэффициент. Если число простое, то используется итеративный алгоритм поиска с возвратом. Число лучших квадратов изначально равняется квадрату заданной стороны. В столешницу помещается квадрат, который сразу же помещается на стек(точнее координаты его левого верхнего угла и длина стороны) со стороной $n/2+1$ (n - сторона столешницы), если длина стороны столешницы не кратна трём или пяти, то на поле вставляются ещё два квадрата со сторонами $n/2$ снизу и справа от первого квадрата, для того чтобы сразу отсеять неоптимальные варианты с большим количеством квадратов. После этого в столешнице ищется максимальная пустая область, которая заполняется квадратом. Данный процесс происходит до тех пор, пока столешница не заполнится, то есть на ней не будет пустого места. Запоминается текущая конфигурация столешницы (координаты верхних углов квадратов, их стороны, и количество квадратов), количество квадратов на данном шаге сравнивается с лучшим количеством квадратов, если текущее меньше лучшего, то лучшее приравнивается к текущему. Неизвестно, является ли данное решение оптимальным, поэтому со стека снимается верхний квадрат, его сторона уменьшается на 1, затем он снова кладётся на стек, если со стека снялся квадрат со стороной 1, то он удаляется, пока не встретится квадрат со стороной более 1. Затем снова запускается цикл, который заполняет столешницу, и так происходит пока стек не опустеет или текущее количество квадратов превысит лучшее.

В консоли выводятся промежуточные значения (после вставки каждого нового квадрата в столешницу, и после нахождения каждой новой лучшей конфигурации при длине стороны столешницы, являющейся простым числом) в виде раскраски каждого квадрата в столешнице его порядковым номером вставки.

Сложность алгоритма по операциям:

$O(e^N)$

Сложность алгоритма по памяти:

$O(N^2)$

Описание функций и структур данных.

1.

```
struct Point
{
    int x;
    int y;
    int length;
    Point()
    {
        x = 0;
        y = 0;
        length = 0;
    }
};
```

Представляет собой квадрат на столешнице размера $length*length$, в полях x , y хранится координата верхнего левого угла данного квадрата.

2.

```
class Table
{
private:
    int length;
    int** table;
    int** best_table;
    int count;
    int best_count;
    vector <Point> vec_curr;
    vector <Point> vec_best;
    unsigned int count_of_operations;
}
```

Описывает столешницу размера $length*length$, с текущим замощением в массиве `table`, лучшим замощением в массиве `best_table`, текущим числом квадратов на столешнице `count`, наименьшим числом квадратов `best_count` в лучшей конфигурации, вектор для текущей конфигурации `vec_curr`, и для лучшей конфигурации `vec_best`, `count_of_operations` - счётчик количества операций.

3. `Table(int length, int count) :length(length), count(0), best_count(length*length)`

Конструктор класса столешницы. Предназначен для инициализации полей экземпляра. Принимает размер столешницы, записывая его в поле `length`. Текущее число вставленных квадратов `count` равно нулю, в лучшей конфигурации число квадратов `best_count` условно принимается как квадрат стороны заданной столешницы, для двумерного массива `table` выделяется память и он инициализируется нулями, для `best_table` тоже выделяется память и он тоже инициализируется нулями.

4. `void insert_square(Point dot)`

Метод класса, который вставляет квадрат в столешницу, в качестве аргумента – структура квадрата, из полей структуры берутся координаты верхнего левого угла, а также длину стороны квадрата, по ним в цикле, который пробегается по двумерному массиву `table` и ищет, квадрат вносится в `table`. Счётчик текущих квадратов увеличивается на один.

5. `void remove_square(Point dot)`

Метод класса, который убирает со столешницы квадрат, в качестве аргумента – структура квадрата, из полей структуры берутся координаты верхнего левого угла, а также длину стороны квадрата, по ним квадрат убирается из двумерного массива `table`. Счётчик текущих квадратов уменьшается на один.

6. `void print_table()`

Метод класса, который выводит текущую конфигурацию столешницы в виде двумерной матрицы с помощью двойного цикла.

7. void print_best_table()

Метод класса, который выводит лучшую минимальную конфигурацию столешницы в виде двумерной матрицы с помощью двойного цикла.

8. void copy_square()

Метод класса, который копирует текущую комбинацию в лучшую комбинацию. Т.е. в двойном цикле i -тый j -тый элемент `best_table` приравнивается к i -тому j -тому элементу `table`.

9. bool fill_table()

Метод класса, который проверяет с помощью двойного цикла, сравнивая i -тый j -тый элемент `table` с нулем, заполнена ли столешница, возвращает `false`, если в столешнице есть пустое место, иначе возвращает `true`.

10. bool check_zero(int x, int y, int m)

Метод класса, который проверяет возможность вставки в столешницу квадрата заданной длины. В качестве аргументов принимает x и y - координаты верхнего левого угла квадрата, и m - длину стороны квадрата. После чего происходит проверка не превосходят ли x , y и $x + m$, $y + m$ длину столешницы. Если превосходят, то функция возвращает `false`. Затем в двойном цикле происходит проверка, свободно ли место для квадрата с заданной длиной стороны путем сравнения i -того j -того элемента `table` с нулем. Если свободного места для квадрата нет, то функция возвращает `true`.

11. void backtracking()

Основной метод класса, который итеративно находит минимальное количество квадратов, необходимое для заполнения столешницы. Вначале создается квадрат с координатами верхнего левого угла $(0; 0)$ и длиной стороны, равной сумме длины столешницы, деленной пополам, и единицы.

Квадрат складывается на вершину стэка. Квадрат вставляется в двумерный массив `table` с помощью функции `insert_square()`. Если длина стороны столешницы не кратна трём или пяти, то вставляются на столешницу справа и снизу от первого квадрата и складываются на стек ещё два квадрата со сторонами равными половине длины столешницы. Затем запускается цикл `do while`, который работает пока стек не опустеет. В него вложен цикл `while`, который работает пока столешница не будет заполнена, или текущее количество квадратов на столешнице не превысит лучшее количество квадратов. В `while` ищется свободная ячейка, затем в цикле `for` с помощью функции `check_zero()` ищется максимальный возможный квадрат, который можно вставить. При нахождении такого квадрата он вставляется в `table` с помощью функции `insert_square()` и помещается на стек. Это происходит до тех пор, пока `table` не заполнится. Проверка заполненности `table` осуществляется при помощи функции `fill_table()`. Как только `table` заполнится, происходит выход из цикла `while`. Затем с помощью условного оператора `if` проверяется условие, что текущее количество квадратов меньше лучшего количества квадратов, которое изначально равняется квадрату стороны столешницы. Если условие выполнено, то лучшее количество квадратов приравнивается к текущему количеству квадратов, и данная конфигурация столешницы запоминается с помощью функции `copy_square()`. Далее, с вершины стэка снимается квадрат. Если длина стороны такого квадрата равняется единице, то он удаляется, и происходит переход к следующему квадрату, пока не встретится квадрат с неединичной стороной. Когда встречается квадрат с неединичной стороной, его сторона уменьшается на один, и он снова складывается на стек, после чего снова запускается цикл `while`.

12. void print_result(int mul)

Метод класса, который выводит результат на экран в следующем виде: сначала идет количество минимальных квадратов, которыми можно заполнить столешницу, затем идут строки, состоящие из трех цифр: первые две —

координаты верхнего левого угла квадрата, формирующего столешницу, третья – длина его стороны. В качестве аргумента принимается коэффициент, на который будет умножаться результат, в случае если заданная пользователем длина столешницы будет кратна двум, трем или пяти. Коэффициент может быть равен единице, в случае простого числа. А если длина столешницы кратна двум, трем или пяти, то коэффициент равен степени числа, которому он кратен.

13. void divided_by_two(int size)

Метод класса, который применяется в случае, если заданная пользователем длина столешницы кратна двум. В качестве аргумента принимается число, равное половине длины стороны столешницы. Соответственно, минимальное количество квадратов для чисел, кратных двум, будет всегда четыре. Нет необходимости просчитывать координаты углов квадратов с помощью итеративной функции `backtracking()`, так как координаты углов можно вычислить зная длину стороны. Координаты угла верхнего левого квадрата (1; 1), верхнего правого (`size + 1; 1`), левого нижнего (1; `size + 1`) и правого нижнего (`size + 1; size + 1`). Длина стороны каждого из квадратов будет равняться `size`.

14. int main()

Считывается с консоли длина столешницы, введенная пользователем. Создается объект класса `Table` с длиной, считанной ранее. Затем происходит проверка на возможность оптимизации работы программы путем проверки кратности длины столешницы двойке, тройке или пятерке. В случае кратности двойке, запускается функция `divided_by_two()`, в которую в качестве аргумента подается половина длины столешницы. В случае кратности тройке или пятерке, запоминается степень кратности. Потом запускается итеративная функция `backtracking()`, которая ищет минимальное количество квадратов, необходимое для заполнения столешницы. После этого выполняется функция `print_result()`, в

которую в качестве аргумента подается степень кратности или единица в случае простого числа. Осуществляется вывод результатов на экран.

Способ хранения частичных решений.

Частичные решения, т.е. конфигурации замощения столешницы, которые ещё не покрывают её всю, хранятся в двумерном массиве `table` типа `int**`.

Использованные оптимизации алгоритма.

Если на данном шаге поиска с возвратом в столешнице квадратов больше или равно числу квадратов в лучшей конфигурации, то такая конфигурация не рассматривается.

```
while (count > best_count) {  
  
}
```

Исходная столешница изначально заполняется квадратом со стороной $N/2+1$ в левом верхнем углу и квадратами со стороной $N/2$ в левом нижнем и правом верхнем углах. В итоге функция поиска вызывается для квадрата размера $(N/2)^2$ в правом нижнем углу.

Также введены эвристики для случаев, когда сторона квадрата не простое число (кратна двум, трём или пяти). В этом случае, столешница замащивается по шаблонам с учётом пропорций.

Исследование.

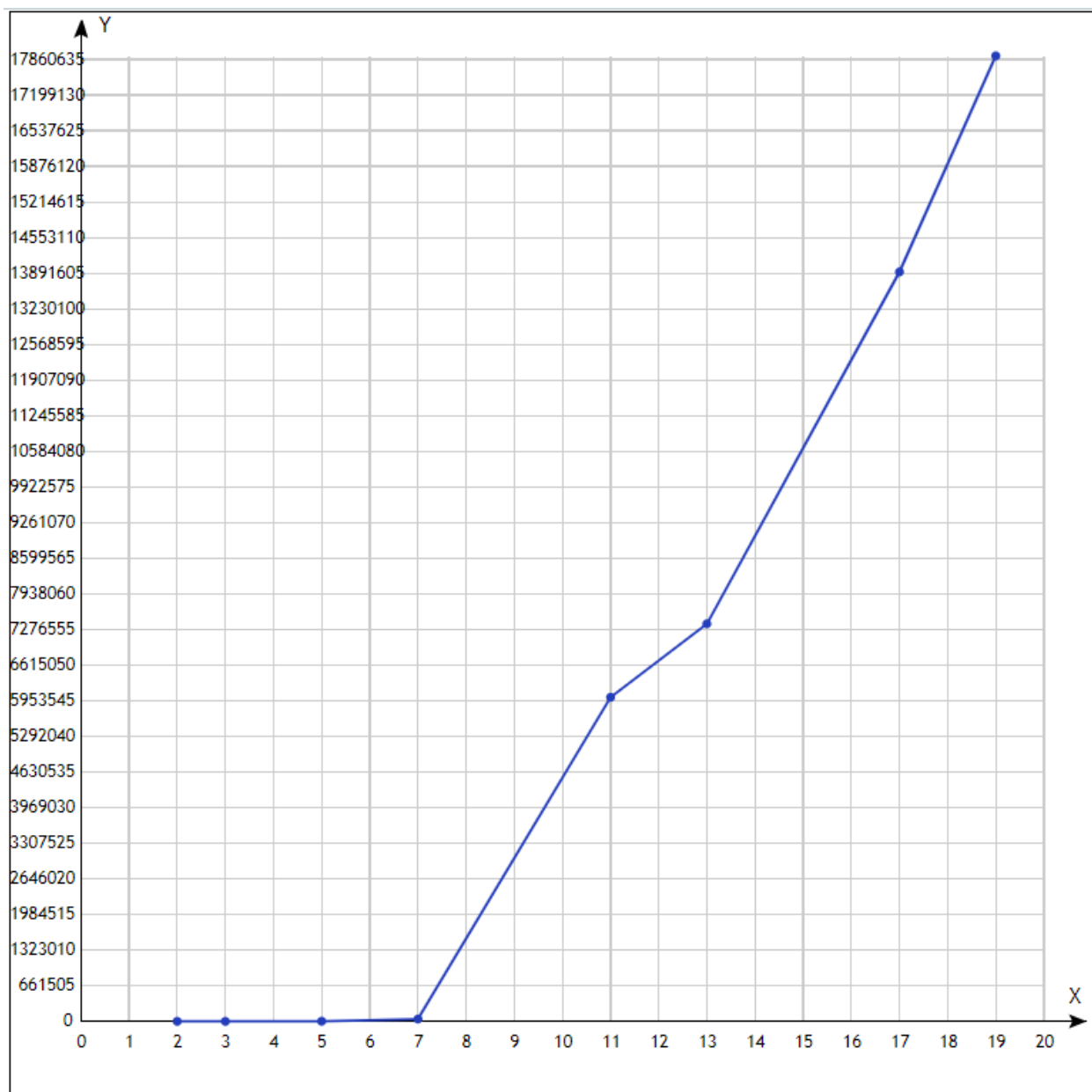
В том случае, когда длина стороны столешницы введённая пользователем кратна двум, то есть чётное число, количество операций равно нулю, потому что минимальное количество квадратов не высчитывается за счёт оптимизации, вывод результата происходит по заготовленному шаблону.

В случае, когда длина кратна трём, количество операций – 33, так как решение сводится заполнению квадрата 3×3 и при выводе результата домножается на степень кратности

В случае, когда длина кратна пяти, количество операций – 2132, , так как решение сводится заполнению квадрата 5x5 и при выводе результата домножается на степень кратности

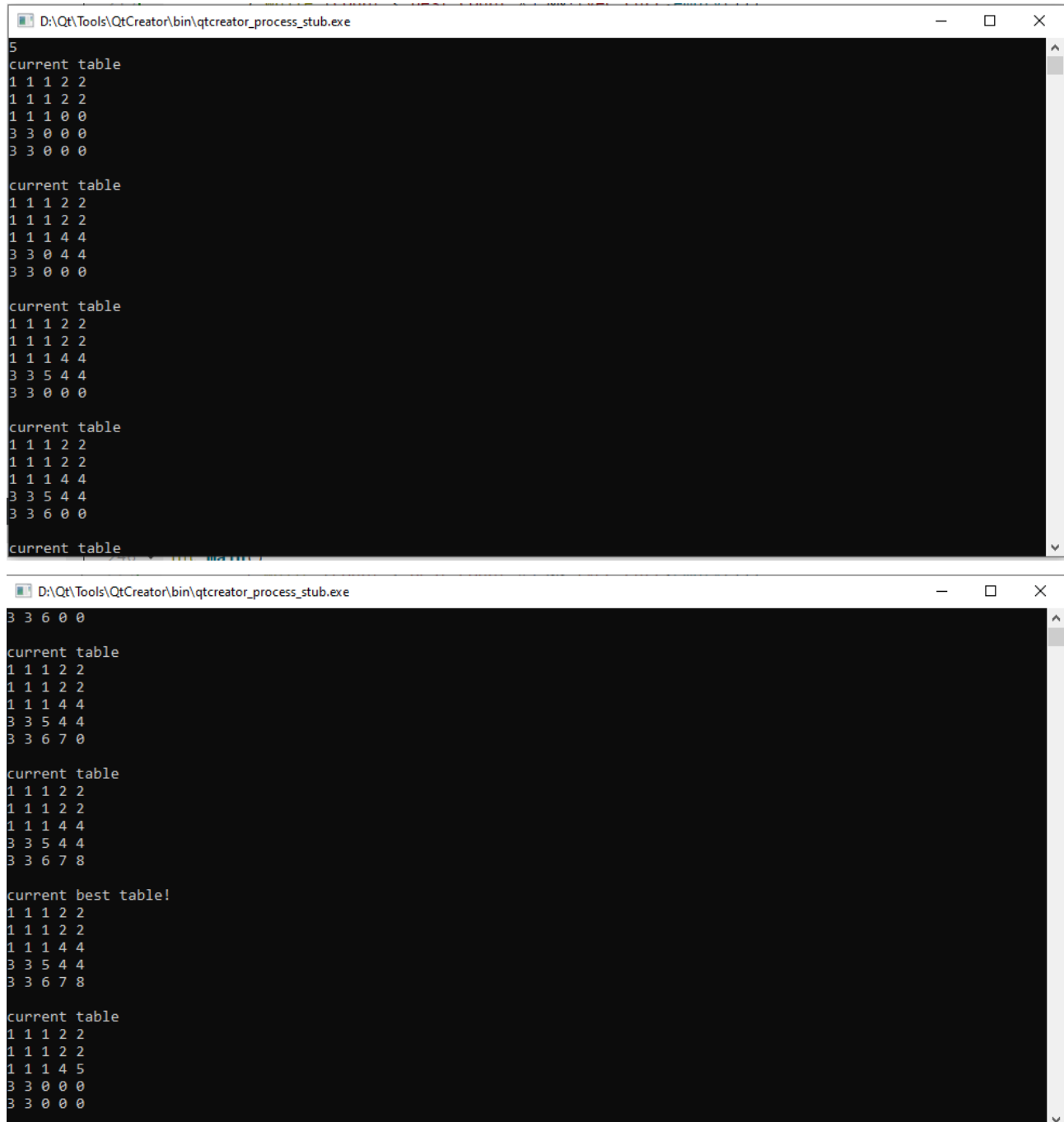
Случаи с простыми числами необходимо рассматривать отдельно. Ниже приведена таблица количества операций при каждой длине столешницы.

| Длина стороны столешницы | Количество операций |
|--------------------------|---------------------|
| 2 | 0 |
| 3 | 33 |
| 4 | 0 |
| 5 | 2132 |
| 6 | 0 |
| 7 | 42119 |
| 8 | 0 |
| 9 | 33 |
| 10 | 0 |
| 11 | 6016786 |
| 12 | 0 |
| 13 | 7381635 |
| 14 | 0 |
| 15 | 33 |
| 16 | 0 |
| 17 | 13914301 |
| 18 | 0 |
| 19 | 17926783 |
| 20 | 0 |



Тестирование.

1.



```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
5
current table
1 1 1 2 2
1 1 1 2 2
1 1 1 0 0
3 3 0 0 0
3 3 0 0 0

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 0 4 4
3 3 0 0 0

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 5 4 4
3 3 0 0 0

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 5 4 4
3 3 6 0 0

current table
3 3 6 0 0

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 5 4 4
3 3 6 7 0

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 5 4 4
3 3 6 7 8

current best table!
1 1 1 2 2
1 1 1 2 2
1 1 1 4 4
3 3 5 4 4
3 3 6 7 8

current table
1 1 1 2 2
1 1 1 2 2
1 1 1 4 5
3 3 0 0 0
3 3 0 0 0
```

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe

current table
1 2 3 4 5
6 7 8 8 8
9 9 8 8 8
9 9 8 8 8
10 11 12 13 0

current table
1 2 3 4 5
6 7 8 8 8
9 9 8 8 8
9 9 8 8 8
10 11 12 13 14

Minimum number of squares: 8
5 5 1
4 5 1
3 5 1
3 4 1
4 3 2
1 4 2
4 1 2
1 1 3
runtime = 4.403
```

2.

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe

Please, input length of table's side
16
Minimum number of squares:4
1 1 8
9 1 8
1 9 8
9 9 8
runtime = 2.752
```

3.

D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe

Please, input length of table's side

15

current table

1 1 2

1 1 0

3 0 0

current table

1 1 2

1 1 4

3 0 0

current table

1 1 2

1 1 4

3 5 0

current table

1 1 2

1 1 4

3 5 6

current best table!

1 1 2

1 1 4

3 5 6

current table

1 2 2

0 2 2

0 0 0

current table

1 2 2

3 2 2

0 0 0

current table

1 2 2

3 2 2

4 0 0

current table

1 2 2

3 2 2

4 5 0

current table

```
D:\Qt\Tools\QtCreator\bin\qtcreator_process_stub.exe
4 5 0
current table
1 2 2
3 2 2
4 5 6
current table
1 2 3
0 0 0
0 0 0
current table
1 2 3
4 4 0
4 4 0
current table
1 2 3
4 4 5
4 4 0
current table
1 2 3
4 4 5
4 4 6
current table
1 2 3
4 5 5
0 5 5
current table
1 2 3
4 5 5
6 5 5
current table
1 2 3
4 5 6
0 0 0
current table
1 2 3
4 5 6
7 0 0
current table
7 0 0
current table
1 2 3
4 5 6
7 8 0
current table
1 2 3
4 5 6
7 8 9
Minimum number of squares: 6
11 11 1
6 11 1
11 6 1
1 11 1
11 1 1
1 1 2
runtime = 1.76
```

Выводы.

Были получены умения по использованию бэктрекинга в алгоритмах. Написана программа по замещению квадратной столешницы заданной длины наименьшим числом квадратов с помощью итеративной функции.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД

•

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

using namespace std;

struct Point// квадрат столешницы
{
    int x;
    int y;
    int length;
    Point()
    {
        x = 0;
        y = 0;
        length = 0;
    }
};

class Table// столешница
{
private:
    int length;
    Point coord;
    int** table;
    int** best_table;
    int count;
    int best_count;
    vector <Point> vec_curr;
    vector <Point> vec_best;
    unsigned int count_of_operations;
public:
```

```
Table(int length, int count) :length(length), count(0), best_count(length*length),count_of_operations(0)
```

```
{
    table = new int* [length];
    best_table = new int* [length];
    for (int i = 0; i < length; i++)
    {
        table[i] = new int[length];
        best_table[i] = new int[length];
        for (int j = 0; j < length; j++)
        {
            table[i][j] = 0;
            best_table[i][j] = 0;
        }
    }
}
```

```
void insert_square(Point dot)//вставка квадрата заданного размера
```

```
{
    for (int i = dot.y; i < dot.y + dot.length; i++)
    {
        for (int j = dot.x; j < dot.x + dot.length; j++)
        {
            table[i][j] = count + 1;
        }
    }
    count++;
}
```

```
void remove_square(Point dot)//удаление квадрата заданного размера
```

```
{
    for (int i = dot.y; i < dot.y + dot.length; i++)
    {
        for (int j = dot.x; j < dot.x + dot.length; j++)
        {
            table[i][j] = 0;
        }
    }
    count--;
}
```

```
void print_table()//ВЫВОД ВСЕГО ПОЛЯ
```

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

```

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

void print_best_table()//вывод всего лучшего поля
{
    std::cout << "current best table!" << "\n";
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            std::cout << best_table[i][j] << " ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}

void copy_square()//сохранение лучшей комбинации для стола
{
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            best_table[i][j] = table[i][j];
        }
    }
}

bool fill_table()// проверка на незаполненный элемент
{
    for (int i = length - 1; i >= 0; --i)
        for (int j = length - 1; j >= 0; --j)
            if (table[i][j] == 0)
                return false;
    return true;
}

bool check_zero(int x, int y, int m)//проверка на возможность вставки квадрата со стороной
m
{
    if (x >= length || y >= length)
        return false;
    if (x + m > length || y + m > length)
    {
        return false;
    }
    for (int i = y; i < y + m; i++)
    {
        for (int j = x; j < x + m; j++)
        {
            if (table[i][j] != 0)
            {

```

```

        return false;
    }
}
}
return true;
}

```

void **backtracking()**// разобраться с тройкой и пятёркой

```

{
    Point dot;
    dot.x = 0;
    dot.y = 0;
    dot.length = length / 2 + 1;
    vec_curr.push_back(dot);
    insert_square(dot);
    if(length % 3 != 0 && length % 5 != 0)
    {
        dot.length = length / 2;
        dot.x = length / 2 + 1;
        vec_curr.push_back(dot);
        insert_square(dot);
        dot.x = 0;
        dot.y = length/2+1;
        vec_curr.push_back(dot);
        insert_square(dot);
    }
    print_table();
    do
    {
        while (count < best_count && !fill_table() )//пока стол не будет заполнен
        {

            for (int y = 0; y < length; y++)
            {
                for (int x = 0; x < length; x++)
                {
                    if (table[y][x] == 0)
                    {
                        for (int m = length - 1; m > 0; m--)
                        {
                            count_of_operations++;
                            if (check_zero(x, y, m))//проверка на вместимость квадрата
                            {
                                dot.x = x;
                                dot.y = y;
                                dot.length = m;
                                break;
                            }
                        }
                    }

                    insert_square(dot);
                }
            }
        }
    }
}

```

```

        vec_curr.push_back(dot);
        print_table();//вывод столешницы при добавлении квадрата
    }
}
}

if (best_count > count)
{
    best_count = count;
    copy_square();
    print_best_table();//вывод промежуточного лучшего результата
    vec_best = vec_curr;
}
while (!vec_curr.empty() && vec_curr[vec_curr.size() - 1].length == 1)//удаление
квадратов со стороной 1
{
    remove_square(vec_curr[vec_curr.size() - 1]);
    vec_curr.pop_back();
}
if (!(vec_curr.empty()))//уменьшение стороны квадрата на 1
{
    dot = vec_curr[vec_curr.size() - 1];
    vec_curr.pop_back();
    remove_square(dot);
    dot.length -= 1;
    insert_square(dot);
    vec_curr.push_back(dot);
}

} while (count < best_count * 3 && !(vec_curr.empty()));
}

void print_result(int mul)//вывод результата работы программы
{
    Point dot;
    std::cout<< "Minimum number of squares: " << best_count << "\n";
    //1std::cout<< "Count of operations: " << count_of_operations << "\n";
    while (!(vec_best.empty()))
    {
        dot = vec_best[vec_best.size() - 1];
        vec_best.pop_back();
        std::cout << dot.x*mul + 1 << " " << dot.y*mul + 1 << " " << dot.length << endl;
    }
}

};

void devided_by_two(int size)// оптимизация в случае когда входные данные кратны двойке
{
    cout << "Minimum number of squares:"<< 4 << endl;
    //cout<< "Count of operations:" << 0 << "\n";

```

```

    cout << "1 1 " << size << endl;
    cout << 1 + size << " 1 "<< size << endl;
    cout << "1 " << 1 + size << " " << size << endl;
    cout << 1 + size << " " << 1 + size << " " << size << endl;
}

int main()
{
    int squareSideLength;
    std::cout << "Please, input length of table's side\n";
    std::cin >> squareSideLength;
    Table table(squareSideLength, 0);
    int mul = 1;
    srand(time(0));
    if (squareSideLength % 2 == 0)// проверка возможности оптимизации
    {
        devided_by_two(squareSideLength / 2);
    }
    else
    {
        if (squareSideLength % 3 == 0)
        {
            mul = squareSideLength / 3;
            squareSideLength = 3;

        }
        else if (squareSideLength % 5 == 0)
        {
            mul = squareSideLength / 5;
            squareSideLength = 5;
        }
        Table table(squareSideLength, 0);
        table.backtracking();
        table.print_result(mul);
    }
    cout << "runtime = " << clock() / 1000.0 << endl; // время работы программы
}

```