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

Студент гр. 8382 _____ Вербин К.М.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

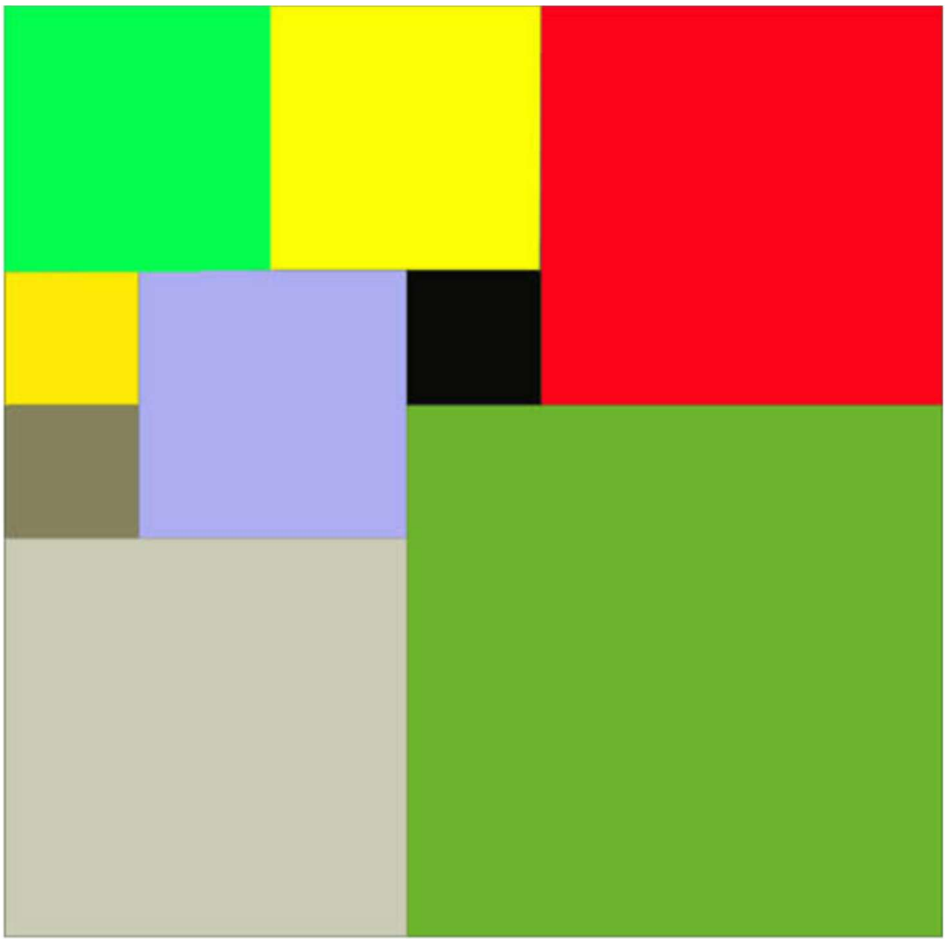
Изучить алгоритм бэктрекинга.

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 ($2 \le N \le 20$).

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 ($1 \le x, y \le N$) 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

Вариант 3и

Итеративный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Ход работы

Описание алгоритма

На вход программе поступает ширина квадрата. Сначала производятся проверки на кратность двум, трем, и пяти. В зависимости от кратности одному из этих чисел, выводятся данные об оптимальном замощении. Иначе применяется итеративный бэктрекинг. Он заключается в полном переборе всех решений.

Сначала в один из углов квадрата ставится первый квадрат с шириной на единицу меньшей, чем ширина исходного квадрата. Затем свободные клетки замощаются квадратами максимально возможной ширины. После

этого, начиная с конца, квадраты с шириной равной единице обнуляются, пока не будет найден квадрат с шириной, большей единицы. Ширина такого квадрата уменьшается на единицу и алгоритм продолжает замощать свободные клетки квадратами максимально возможной ширины пока не обнулятся все квадраты кроме первого. В таком случае рассматриваются различные конфигурации с шириной первого квадрата ещё на единицу меньшей. Это повторяется до тех пор, пока ширина первого квадрата не станет равна нулю.

Сложность алгоритма по операциям: O (e^N)

Сложность алгоритма по памяти: O (N^2)

Описание функций и структур данных

1.

```
struct Square {
    int y;
    int x;
    int width;
    int num;
};
```

Описывает квадрат размера width*width с координатами его верхнего левого угла y и x, шириной width и порядковым номером num.

2.

```
void fill(int width, vector& bestVector)
```

Основная функция бэктрекинга, вызывающаяся в main. В качестве аргуметов принимает ширину исходного квадрата и ссылку на вектор с лучшим решением. Предназначена для заполнения столешницы квадратами.

3.

```
void printField(int** field, int width)
```

Функция принимает двумерный массив клеток исходного квадрата, а также его ширину. Предназначена для печати конфигурации столешницы.

4.

void printVector(vector vector)

Функция принимает вектор и распечатывает его содержимое.

5.

void drawSquare(int** field, int width, int y0, int x0, int num)

Функция принимает двумерный массив клеток исходного квадрата, координаты левого верхнего угла квадрата, длину его стороны и порядковый номер. Рисует квадрат по координатам x0 и y0 с шириной width и порядковым номером num.

6.

int findSize(int** field, int widthOfField, int y0, int x0)

Функция принимает двумерный массив клеток исходного квадрата, его ширину и координаты левого верхнего угла квадрата, которым будет замощена часть столешницы. Предназначена для определения максимально возможной ширины квадрата, рисуемого из клетки с координатами y0 и x0. Функция возвращает найденную ширину.

7.

void addToVector(vector& vector, int y, int x, int width, int num)

Функция принимает ссылку на вектор с частным решением, координаты добавляемого в вектор квадрата, его ширину и номер. Необходима для добавления в вектор данных о новом квадрате на столешнице.

8.

void vectorToBestVector(vector& bestVector, vector vector, int widthOfLargerSquare)

Функция принимает ссылку на вектор с лучшим решением, вектор с частным решением и ширину квадрата в левом верхнем углу. Предназначена для перенесения данных о расположении квадратов на столешнице из вектора с частным решением в вектор с лучшим решением.

9.

void even(int width)

Функция принимает ширину исходного квадрата. Выводит результат для случая с четной стороной квадрата.

10.

void odd3(int width)

Функция принимает ширину исходного квадрата. Выводит результат для случая со стороной квадрата, кратной трём.

11.

void odd5(int width)

Функция принимает ширину исходного квадрата. Выводит результат для случая со стороной квадрата, кратной пяти.

Использованные оптимизации алгоритма

В случае если ширина исходного квадрата кратна двум, трем или пяти, то результат выводится сразу без запуска бэктрекинга. Ответ получается, исходя из пропорции и ширины замощаемого квадрата.

Изначально квадрат замощается тремя квадратами: первый - в правом нижнем углу с шириной width / 2 + 1, второй и третий в нижнем левом и правом верхнем углах соответственно с шириной width / 2. Таким образом площадь, которую необходимо замощать алгоритмом бэктрекинга уменьшается примерно втрое.

Если во время бэктрекинга при добавлении данных об очередном квадрате в вектор с частным решением будет обнаружено, что количество квадратов на столешнице стало равно количеству квадратов при лучшей конфигурации, то дальнейшее замощение для данной конфигурации прервется.

Тестирование

Input	Output	Add_count	Del_count	Com_count	Rel_count	Count
5	8 3 3 3 4 1 2	20	12	4	2	38

	1 4 2 1 1 2 3 1 1 3 2 1 1 3 1 2 3 1					
7	9 4 4 4 5 1 3 1 5 3 1 1 2 3 1 2 1 3 2 3 3 1 4 3 1 3 4 1	58	43	19	4	124
11	11 6 6 6 7 1 5 1 7 5 1 1 4 5 1 2 5 3 2 1 5 2 3 5 2 5 5 1 6 5 1 5 6 1	565	554	404	2	1525
13	11 7 7 7	1141	1122	1081	3	3347

	8 1 6 1 8 6 1 1 3 4 1 4 1 4 2 3 4 1 3 5 3 6 5 2 1 6 2 6 7 1					
17	12 9 9 9 10 1 8 1 10 8 1 1 5 6 1 4 6 5 2 8 5 2 1 6 4 5 6 1 5 7 3 8 7 2 8 9 1	6878	6856	7867	3	21604
19	13 10 10 10 11 1 9 1 11 9 1 1 6 7 1 4 7 5 4	27761	27748	33766	2	89277

	1 7 4 5 7 2 5 9 2 7 9 2 9 9 1 10 9 1 9 10 1					
23	13 12 12 12 13 1 11 1 13 11 1 1 5 6 1 7 1 6 3 4 6 2 4 8 1 5 8 5 10 8 3 1 9 4 10 11 2 12 11 1	95365	95342	129797	3	320507
29	14 15 15 15 16 1 14 1 16 14 1 1 8 9 1 7 9 8 2 11 8 5 1 9 7	783642	783570	1133364	6	2700582

	8 9 1 8 10 3 8 13 3 11 13 3 14 13 2 14 15 1					

Исследование количество операций

На рисунке 1 представлен график зависимости количества операций от длины стороны квадрата.

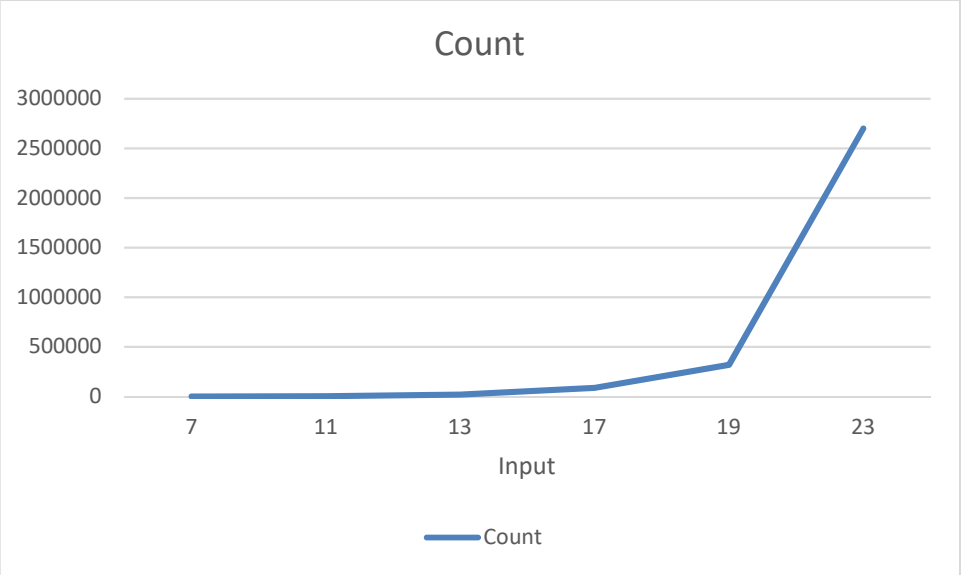


Рисунок 1 Зависимость количества операций от длины стороны квадрата.

Из полученных результатов можно сделать вывод, что сложность данного алгоритма для простых значений схожа с экспоненциальной, но для четных и делящихся на 3 алгоритм работает за константу, если быть точнее, то константа умноженная на 2 в степени N.

Вывод

В ходе написания лабораторной работы были получены умения по использованию бэктрекинга в алгоритмах и применению оптимизаций для ускорения работы поиска с возвратом.

ПРИЛОЖЕНИЕ

КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <time.h>

//#define DEBUG
int add_count = 0;
int del_count = 0;
int rel_count = 0;
int com_count = 0;

using namespace std;

struct Square
{
   int y;
   int x;
   int width;
   int num;
};

void printField(int** field, int width)
{
   for (int i = 0; i < width; i++) {
      for (int j = 0; j < width; j++)
         cout << field[i][j] << " ";
      cout << endl;
   }
   cout << endl;
}

void printVector(vector<Square> vector)
{

   for (int i = 0; i < vector.size(); ++i)
      cout  << vector[i].x + 1 << " " << vector[i].y + 1 << " " <<
vector[i].width  << endl;
}

int findSize(int** field, int widthOfField, int y0, int x0)
{
```

```
        int width;                                                  //
        for (width = 2; width < widthOfField + 1; ++width) {                    //
            for (int y = y0; y < y0 + width; ++y) {                         //
                for (int x = x0; x < x0 + width; ++x) {                     //
Нахождение максимальной ширины
                    if (y >= widthOfField || x >= widthOfField || field[y][x] != 0)  //
квадрата, который можно вставить
                        return width - 1;                                   //  в данной точке
                }                                                       //
            }                                                           //
        }                                                               //
        return width - 1;                                                   //
    }

    void drawSquare(int** field, int width, int y0, int x0, int num)
    {
        for (int y = y0; y < y0 + width; y++)
            for (int x = x0; x < x0 + width; x++)
                field[y][x] = num;
    }

    void addToVector(vector<Square>& vector, int y, int x, int width, int num)
    {
        Square square;
        square.y = y;
        square.x = x;
        square.width = width;
        square.num = num;
        vector.push_back(square);
        add_count++;
    }

    void even(int width)
    {
        cout << "Minimal count of Squares: " << 4 << endl;
        cout << 1 << " Point ( x = " << "1, y = " << "1, width = " << width / 2 <<
")" << endl;
        cout << 2 << " Point ( x = " << "1, y = " << width / 2 + 1 << ", width = "
<< width / 2 << ")" << endl;
        cout << 3 << " Point ( x = " << width / 2 + 1 << ", y = " << "1, width = "
<< width / 2 << ")" << endl;
        cout << 4 << " Point ( x = " << width / 2 + 1 << ", y = " << width / 2 + 1
<< ", width = " << width / 2 << ")" << endl;
    }
```

```
        void odd3(int width) {
            cout << "Minimal count of Squares: " << 6 << endl;
            cout << 1 << " Point ( x = " << "1, y = 1, width = " << 2 * width / 3 << ")"
<< endl;
            cout << 2 << " Point ( x = " << "1, y = " << 2 * width / 3 + 1 << ", width
= " << width / 3 << ")" << endl;
            cout << 3 << " Point ( x = " << width / 3 + 1 << ", y = " << 2 * width / 3 +
1 << ", width = " << width / 3 << ")" << endl;
            cout << 4 << " Point ( x = " << 2 * width / 3 + 1 << ", y = " << 2 * width /
3 + 1 << ", width = " << width / 3 << ")" << endl;
            cout << 5 << " Point ( x = " << 2 * width / 3 + 1 << ", y = " << width / 3 +
1 << ", width = " << width / 3 << ")" << endl;
            cout << 6 << " Point ( x = " << 2 * width / 3 + 1 << ", y = " << 1 << ",
width = " << width / 3 << ")" << endl;
        }

        void odd5(int width) {
            cout << "Minimal count of Squares: " << 8 << endl;
            cout << 1 << " Point ( x = " << "1, y = 1, width = " << 3 * width / 5 << ")"
<< endl;
            cout << 2 << " Point ( x = " << "1, y = " << 3 * width / 5 + 1 << ", width
= " << 2 * width / 5 << ")" << endl;
            cout << 3 << " Point ( x = " << 2 * width / 5 + 1 << ", y = " << 3 * width /
5 + 1 << ", width = " << 2 * width / 5 << ")" << endl;
            cout << 4 << " Point ( x = " << 3 * width / 5 + 1 << ", y = 1, width = " <<
2 * width / 5 << ")" << endl;
            cout << 5 << " Point ( x = " << 3 * width / 5 + 1 << ", y = " << 2 * width /
5 + 1 << ", width = " << width / 5 << ")" << endl;
            cout << 6 << " Point ( x = " << 4 * width / 5 + 1 << ", y = " << 2 * width /
5 + 1 << ", width = " << width / 5 << ")" << endl;
            cout << 7 << " Point ( x = " << 4 * width / 5 + 1 << ", y = " << 3 * width /
5 + 1 << ", width = " << width / 5 << ")" << endl;
            cout << 8 << " Point ( x = " << 4 * width / 5 + 1 << ", y = " << 4 * width /
5 + 1 << ", width = " << width / 5 << ")" << endl;
        }

        void vectorToBestVector(vector<Square>& bestVector, vector<Square>
vector, int widthOfLargerSquare)
        {
            bestVector[3].width = widthOfLargerSquare;
            int i;
            int size = bestVector.size() - 4;
            for (i = 0; i < size; ++i)
                bestVector.pop_back();
            for (i = 0; i < vector.size(); ++i) {
```

```
            addToVector(bestVector, vector[i].y, vector[i].x, vector[i].width,
vector[i].num);
            }
            rel_count++;
        }

        void fill(int width, vector<Square>& bestVector)
        {
            vector<Square> vector;

            int countOfSquares = 4;

            int newWidth = width / 2 + 1;            //  Ширина квадрата в левом
верхнем углу

            int** field = new int* [newWidth];        //
            for (int y = 0; y < newWidth; ++y)        //  Выделение памяти под
двумерный массив
                field[y] = new int[newWidth];        //

            int startWidth = newWidth / 2 + 1;
            //  Добавление четырех начальных квадратов
            addToVector(bestVector, newWidth - 1, newWidth - 1, newWidth, 1);   //
_ _ _  в вектор с лучшей конфигурацией
            addToVector(bestVector, 0, newWidth, width - newWidth, 2);          //
|_|_|_|
            addToVector(bestVector, newWidth, 0, width - newWidth, 3);          //  |_|
|
            addToVector(bestVector, 0, 0, startWidth, 4);                       //  |_|_ _|

            for (int widthOfLargerSquare = startWidth; widthOfLargerSquare >= 1; --
widthOfLargerSquare) {

                bool flagToEnd = false;

        #ifdef DEBUG
                cout << "Draw the first square with a width of one less: " << endl <<
endl;
        #endif

                for (int y = 0; y < newWidth; ++y)                              //
                    for (int x = 0; x < newWidth; ++x) {                        //
                        if (x < widthOfLargerSquare && y < widthOfLargerSquare)  //  В
верхний левый угол квадрата
```

```
                    field[y][x] = 1;                                    //  вставляется квадрат с
определенной
                else                                                    //  шириной, остальные
клетки зануляются
                    field[y][x] = 0;                                    //
                }                                                       //

            field[newWidth - 1][newWidth - 1] = -1;                     //  Крайний
нижний угол замощаемого в процессе
                                                                        //  бэктрэкинга квадрата
устанавливается в -1

        #ifdef DEBUG
            printField(field, newWidth);
            cout << "Square filling: " << endl << endl;
        #endif
            for (int y = 0, num = 2; y < newWidth && !flagToEnd; ++y)       //
Нулевые клетки замещаются квадратами
                for (int x = 0; x < newWidth && !flagToEnd; ++x) {          //
максимально возможной ширины, при этом
                    if (field[y][x] == 0) {                                 //  частное решение
заносится в вектор
                        if (vector.size() + 4 >= countOfSquares &&
widthOfLargerSquare != startWidth) {                //  Если количество
квадратов, необходимых
                            flagToEnd = true;
//  в получаемом частном решении, больше количества
        #ifdef DEBUG                                                        //
квадратов, получаемых при лучшей конфигурации, поднимается
                            cout << "Already found a solution with the same or fewer
squares!" << endl << endl;       //  флаг для прекращения поиска лучшего
решения
        #endif
                            break;
                        }
                        int size = findSize(field, newWidth, y, x);         //
                        drawSquare(field, size, y, x, num);                 //
                        addToVector(vector, y, x, size, num);               //
                        num++;                                              //
        #ifdef DEBUG
                        printField(field, newWidth);
        #endif
                    }
                }
            com_count++;
```

```
            if (countOfSquares == 4 || vector.size() + 4 < countOfSquares) {  // Если это первая итерация или найденное
                countOfSquares = vector.size() + 4;                    //  частное решение лучше найденного ранее, то
                rel_count++;
                vectorToBestVector(bestVector, vector, widthOfLargerSquare); // новое решение становится лучшим
            }


            while (1) {

                flagToEnd = false;
                Square lastSquare;
        #ifdef DEBUG
                cout << "Decreasing the width of the square per unit: " << endl << endl;
        #endif
                for (int i = vector.size(); i != 0; --i) {              //  Начиная с конца вектора, содержащего
                    lastSquare = vector[vector.size() - 1];             //  последнее частное решение, ищем первый
                    if (lastSquare.width == 1) {                        //  квадрат ширины, большей 1, попутно обнуляя
                        field[lastSquare.y][lastSquare.x] = 0;          //  квадраты с шириной 1
                        vector.pop_back();                              //
                        del_count++;
        #ifdef DEBUG
                        printField(field, newWidth);
        #endif
                    }
                    else {
        #ifdef DEBUG
                        cout << "Reducing the width of the last from the end of the square with a width greater than one: " << endl << endl;
        #endif
                        for (int y = lastSquare.y; y < lastSquare.y + lastSquare.width; ++y) {                          //
                            for (int x = lastSquare.x; x < lastSquare.x + lastSquare.width; ++x) {                      //
                                if (y >= lastSquare.y + lastSquare.width - 1 || x >= lastSquare.x + lastSquare.width - 1)    //
                                    field[y][x] = 0;
//  Если ширина очередного квадрата
```

```
                        }                                                                   //
в векторе больше 1, то она
                    }                                                                       //
уменьшается на 1 и даннный цикл
                    vector[vector.size() - 1].width = lastSquare.width - 1;
//  завершает работу
        #ifdef DEBUG                                                                    //
                    printField(field, newWidth);
//
        #endif                                                                          //
                    break;
//
                }                                                                       //
            }                                                                           //
                                                        //  Если на предыдущем шаге
были обнулены все
            if (vector.empty())                             //  квадраты ширины
1, то цикл while завершает
                break;                                  //  поиск оптимальных
конфигураций при данной
                                                        //  ширине квадрата в левом
верхнем углу
        #ifdef DEBUG
            cout << "Square filling: " << endl << endl;
        #endif

            for (int y = 0, num = lastSquare.num + 1; y < newWidth &&
!flagToEnd; ++y)  //  Нулевые клетки замещаются квадратами
                for (int x = 0; x < newWidth && !flagToEnd; ++x) {
//  максимально возможной ширины, при этом
                    if (field[y][x] == 0) {                         //  частное
решение заносится в вектор
                        if (vector.size() + 4 >= countOfSquares) {
//  Если количество квадратов, необходимых
                            flagToEnd = true;
//  в получаемом частном решении, больше количества
        #ifdef DEBUG                                            //  квадратов,
получаемых при лучшей конфигурации, поднимается
                            cout << "Already found a solution with the same or fewer
squares!" << endl << endl;//  флаг для прекращения поиска лучшего решения
        #endif
                            break;
                        }
                        int size = findSize(field, newWidth, y, x);             //
                        drawSquare(field, size, y, x, num);                     //
```

```
                    addToVector(vector, y, x, size, num);                    //
                    num++;                                                   //
#ifdef DEBUG
                    printField(field, newWidth);
#endif
                }
            }
            com_count++;
            if (vector.size() + 4 < countOfSquares) {                        // Если
найденное частное решение лучше
                countOfSquares = vector.size() + 4;                          //
найденного ранее, то новое решение
                vectorToBestVector(bestVector, vector, widthOfLargerSquare);
//  становится лучшим
            }                                                                //
        }
        vector.clear();                 //  Очищается вектор частных решений
    }
    cout << countOfSquares << endl;
}



int main() {
    int width;
    int count;
    vector<Square> bestVector;
    cout << "Enter width of square :";
    cin >> width;

        if (!(width % 2) )
            even(width);
        else if (!(width % 3))
            odd3(width);
        else if (!(width % 5))
            odd5(width);
        else {
            fill(width, bestVector);
            printVector(bestVector);
        }

        cout <<"Adding count:\t" <<  add_count << endl;
        cout <<"Delete count:\t" << del_count << endl;
        cout <<"Compare count:\t" << com_count << endl;
        cout << "Reload count:\t" << rel_count << endl;
```

```
            cout <<"Count:\t\t" << del_count + add_count + com_count +
rel_count << endl;


        }
```