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

ОТЧЕТ

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

Студентка гр. 8382 _____ Кулачкова М.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

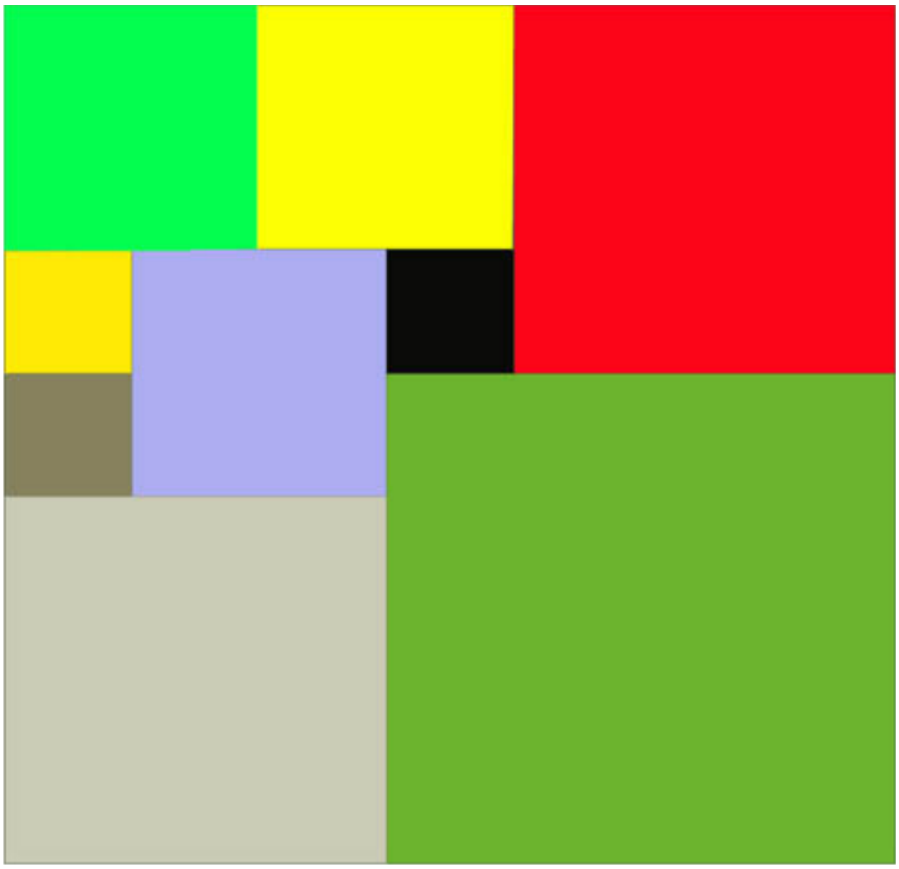
Цель работы.

Построить итеративный алгоритм поиска с возвратом для решения поставленной задачи. Определить сложность полученного алгоритма по операциям и по памяти.

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

1 5 3

3 4 1

Вариант дополнительного задания.

4и: итеративный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

Описание алгоритма.

Итеративный алгоритм поиска с возвратом реализован методом *void Rectangle::covering(int tileSize, Tiles& curr, Tiles& best)* структуры *Rectangle* (см. разделы *Описание структур данных* и *Описание функций и методов*). В

бесконечном цикле обходится матрица поля, представляющая покрываемый квадрат. При нахождении свободной клетки с помощью перебора всех длин сторон в порядке уменьшения на поле добавляется квадрат максимально возможного размера, верхний левый угол которого расположен в этой клетке.

Для хранения промежуточных решений реализована структура *Tiles* (см. раздел *Описание структур данных*), которая содержит массивы координат и длин сторон квадратов разбиения, а также число квадратов в разбиении.

Если в ходе алгоритма число квадратов в текущем разбиении при непустом поле становится больше или равным числу квадратов в минимальном разбиении или если поле оказывается заполнено, происходит удаление с поля «хвоста» разбиения: поле освобождается от размещенных последними единичных квадратов, а квадрат, добавленный на поле перед ними или последним, если в конце разбиения нет единичных квадратов, заменяется квадратом, начинающимся в той же точке поля, со стороной на 1 меньше. При этом дальнейший обход матрицы начинается с правого верхнего угла последнего квадрата в новом разбиении, т. е. оттуда, откуда он продолжился бы, если бы квадрат был добавлен при обычном обходе поля.

Удаление последних квадратов разбиения также служит критерием выхода из цикла. Если при удалении «хвоста» был удален последний квадрат разбиения, т. е. больше нет нерассмотренных вариантов разбиения, работа алгоритма завершается.

В случае, когда после добавления нового квадрата поле оказывается заполнено, проверяется число квадратов в текущем разбиении. Если оно меньше числа квадратов в минимальном разбиении, минимальное разбиение заменяется текущим, а счетчику вариантов минимального разбиения присваивается значение 1. Если число квадратов в текущем разбиении совпадает с числом квадратов в минимальном разбиении, происходит приращение счетчика вариантов минимального разбиения. Так как поле обходится в строго определенном порядке – слева направо и сверху вниз, – и при уменьшении квадратов они остаются на своем изначальном месте,

рассмотрение двух аналогичных разбиений, различающихся только порядком добавления одинаковых квадратов в одну точку поля, невозможно.

Использованные оптимизации.

Квадраты со сторонами, кратными 2, 3 и 5, обрабатываются отдельно, т. к. имеют особые разбиения, которые легко определить вручную. Также реализованы разные начальные условия для применения алгоритма к квадратам и прямоугольникам: если n – сторона покрываемого квадрата или меньшая из сторон прямоугольника, то в качестве максимальной стороны квадрата разбиения в первом случае задается значение $(n + 1)/2$, т.к. известно, что минимальное разбиение квадрата с простой стороной содержит такой квадрат, а во втором случае – $(n – 1)$.

Для проверки программы системой Stepik она была упрощена с целью увеличить скорость выполнения задачи без индивидуализации: на поле со стороной n предварительно размещаются один квадрат со стороной $(n + 1)/2$ и два квадрата со сторонами $(n – 1)/2$ справа и снизу от него. Оставшаяся часть квадрата обрабатывается приведенным выше алгоритмом, при этом уже размещенные квадраты не могут быть удалены с поля.

Оценка сложности алгоритма.

Без учета оптимизаций сложность алгоритма по операциям составила бы $O(n^n)$ для квадрата и $O((w \times h)^n)$ для прямоугольника – w и h – ширина и длина прямоугольника соответственно, – если бы осуществлялся полный перебор вариантов без ограничения на минимальность решения. Ограничение на минимальность приводит к тому, что часть циклов прерывается до прохождения всех возможных значений, поэтому сложность алгоритма по операциям можно оценить величиной $O(e^n)$.

Т.к. хранение текущего и лучшего решения осуществляется с помощью статических массивов (см. раздел *Описание структур данных*), алгоритм занимает фиксированный объем памяти на стеке для любого обрабатываемого

прямоугольника. Матрица прямоугольника хранится в динамическом массиве, поэтому объем памяти, занимаемой ей в куче, можно оценить как $O(w \times h)$.

Описание структур данных.

Для хранения промежуточных решений и конечного (лучшего) решения реализована структура *Tiles* (см. Приложение А):

```
struct Tiles {
int x[MAX_N * MAX_N]; //координаты x квадратов разбиения
int y[MAX_N * MAX_N]; //координаты y квадратов разбиения
int w[MAX_N * MAX_N]; //длины сторон квадратов разбиения
int n; //число квадратов в разбиении
Tiles() { n = 0; }
void printList(); //печать решения
void addTile(int x, int y, int w); //добавление квадрата в решение
void removeTile(int ind = -1); //удаление квадрата из решения
};
```

Она состоит из статических массивов размера MAX_N^2 координат *x*, *y* и длин сторон *w* квадратов разбиения и целого числа *n*, содержащего текущее число квадратов разбиения. При создании объекта структуры переменной *n* конструктором присваивается значение 0. Размер массивов задается так, чтобы они могли вместить наибольшее возможное разбиение, т. е. разбиение квадрата с максимально допустимой по заданию стороной единичными квадратами.

Для работы с прямоугольником реализована структура *Rectangle* (см. Приложение А):

```
struct Rectangle {
int width; //ширина поля
int height; //высота поля
int freeCells; //число пустых клеток
int ** table; //матрица поля
Rectangle(int width, int height);
~Rectangle();
bool isFull(); //проверка заполненности поля
void print(); //вывод матрицы поля на экран
void clear(Tiles tl); //удаление последнего квадрата с поля
bool place(Tiles tl); //размещение последнего квадрата на поле
void tryToFit(Tiles& curr, int tileSize, int x, int y); //поиск
наибольшего квадрата, который можно разместить на поле
bool removeTail(Tiles& curr); //удаление «хвоста» разбиения
void covering(int tileSize, Tiles& curr, Tiles& best); //поиск
минимального разбиения
```

```
};
```

Структура содержит поля *height* и *width*, соответствующие высоте и ширине прямоугольного поля, задаваемого пользователем, поле *freeCells*, хранящее число пустых клеток на поле, и двойной массив целых чисел *table* – матрица поля, на которой отмечаются квадраты разбиения.

Описание функций и методов.

Были реализованы следующие методы структуры *Tiles*, осуществляющие взаимодействие с промежуточными решениями:

- `void Tiles::printList();` — этот метод выводит информацию о квадратах разбиения в формате «y x w».
- `void Tiles::addTile(int x, int y, int w);` — этот метод добавляет в массивы координаты *x*, *y* и длину стороны *w* нового квадрата разбиения и увеличивает счетчик квадратов *n*.
- `void Tiles::removeTile(int ind = -1);` — этот метод удаляет из массивов информацию о квадрате с заданным индексом *ind* и уменьшает счетчик квадратов. Если индекс не задан, удаляется последний добавленный квадрат.

В число методов структуры *Rectangle* входят методы, осуществляющие взаимодействие с матрицей поля, среди которых метод, реализующий основной алгоритм, и вспомогательные методы:

- `bool Rectangle::isFull();` — метод возвращает *true*, если число свободных клеток на поле равно нулю, и *false* в противном случае.
- `void Rectangle::print();` — этот метод выводит на экран матрицу поля.
- `void Rectangle::clear(Tiles tl);` — этот метод удаляет с матрицы поля последний квадрат, добавленный в структуру *tl*, содержащую промежуточное решение.
- `bool Rectangle::place(Tiles tl);` — этот метод пытается разместить в матрице поля последний квадрат, добавленный в структуру *tl*, содержащую промежуточное решение. Метод возвращает *true*, если квадрат был размещен удачно, и *false*, если разместить квадрат невозможно, т. е. при попытке размещения квадрат пересекал уже размещенные на поле квадраты или выходил за границы поля.
- `void Rectangle::tryToFit(Tiles& curr, int tileSize, int x, int y);` — этот метод перебирает все возможные длины сторон квадратов, начиная с *tileSize* и заканчивая 1, и пытается разместить квадраты с такими сторонами по заданным координатам *x* и *y*. Таким

образом по указанным координатам размещается максимально большой квадрат. Размещенный квадрат добавляется в структуру *curr*, содержащую промежуточное решение.

- `bool Rectangle::removeTail(Tiles& curr);` — в этом методе происходит удаление «хвоста» текущего промежуточного решения *curr*. Если решение заканчивается единичными квадратами, они удаляются. Квадрат со стороной больше 1, размещенный перед единичными или в конце массива, заменяется квадратом со стороной на 1 меньше, расположенным по тем же координатам. Метод возвращает *false*, если при попытке удаления «хвоста» из массива был удален последний квадрат, и *true* в противном случае.
- `void Rectangle::covering(int tileSize, Tiles& curr, Tiles& best);` — этот метод осуществляет основной алгоритм разбиения, описанный в предыдущем пункте данного отчета. Здесь *tileSize* – максимально возможная длина квадрата разбиения для данного поля, *curr* – текущее разбиение, *best* – текущее минимальное разбиение.

Тестирование.

1. *Ввод*: 13 13
   Вывод:
   11
   1 1 7
   1 8 6
   7 8 2
   7 10 4
   8 1 6
   8 7 1
   9 7 3
   11 10 1
   11 11 3
   12 7 2
   12 9 2
   Number of minimum fragmentation varieties: 8

Разбиение:

```
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  2  2  2  2  2  2
1  1  1  1  1  1  1  3  3  4  4  4  4
5  5  5  5  5  5  6  3  3  4  4  4  4
5  5  5  5  5  5  7  7  7  4  4  4  4
5  5  5  5  5  5  7  7  7  4  4  4  4
5  5  5  5  5  5  7  7  7  8  9  9  9
5  5  5  5  5  5  10 10 11 11 9  9  9
5  5  5  5  5  5  10 10 11 11 9  9  9
```

2. *Ввод*: 11 9
   Вывод:
   9
   1 1 6
   1 7 3
   4 7 3
   7 1 5
   7 6 4
   11 6 1
   11 7 1
   11 8 1
   11 9 1
   Number of minimum fragmentation varieties: 32

   Разбиение:

```
1  1  1  1  1  1  2  2  2
1  1  1  1  1  1  2  2  2
1  1  1  1  1  1  2  2  2
1  1  1  1  1  1  3  3  3
1  1  1  1  1  1  3  3  3
1  1  1  1  1  1  3  3  3
4  4  4  4  4  5  5  5  5
4  4  4  4  4  5  5  5  5
4  4  4  4  4  5  5  5  5
4  4  4  4  4  5  5  5  5
4  4  4  4  4  6  7  8  9
```

3. *Ввод*: 14 14

Вывод:

4

1 1 7

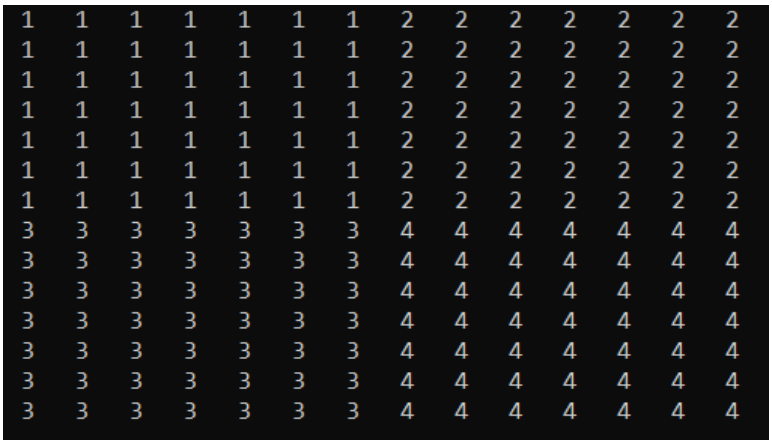
1 8 7

8 1 7

8 8 7

Number of minimum fragmentation varieties: 1

Разбиение:



4. *Ввод*: 7 18

Вывод:

12

1 1 4

1 5 4

1 9 4

1 13 4

1 17 2

3 17 2

5 1 3

5 4 3

5 7 3

5 10 3

5 13 3

5 16 3

Number of minimum fragmentation varieties: 18

Разбиение:

```
1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4  5  5
1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4  5  5
1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4  6  6
1  1  1  1  2  2  2  2  3  3  3  3  4  4  4  4  6  6
7  7  7  8  8  8  9  9  9  10 10 10 11 11 11 12 12 12
7  7  7  8  8  8  9  9  9  10 10 10 11 11 11 12 12 12
7  7  7  8  8  8  9  9  9  10 10 10 11 11 11 12 12 12
```

5. *Ввод*: 9 9

   Вывод:

   6

   1 1 6

   7 1 3

   1 7 3

   7 4 3

   4 7 3

   7 7 3

   Number of minimum fragmentation varieties: 4

   Разбиение:

```
1  1  1  1  1  1  3  3  3
1  1  1  1  1  1  3  3  3
1  1  1  1  1  1  3  3  3
1  1  1  1  1  1  5  5  5
1  1  1  1  1  1  5  5  5
1  1  1  1  1  1  5  5  5
2  2  2  4  4  4  6  6  6
2  2  2  4  4  4  6  6  6
2  2  2  4  4  4  6  6  6
```

Выводы.

Была реализована программа, находящая разбиение прямоугольного поля минимально возможным количеством квадратов и подсчитывающая количество вариантов минимального разбиения. Для решения поставленной задачи был построен и проанализирован итеративный алгоритм поиска с возвратом. Полученный алгоритм обладает экспоненциальной сложностью по операциям и квадратичной сложностью по памяти.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ
Файл main.cpp

```
#include <iostream>

#define MAX_N 20

using namespace std;
int minK, varCounter;

struct Tiles {
	int x[MAX_N * MAX_N]; //координаты x квадратов разбиения
	int y[MAX_N * MAX_N]; //координаты y квадратов разбиения
	int w[MAX_N * MAX_N]; //длины сторон квадратов разбиения
	int n; //число квадратов в разбиении
	Tiles() { n = 0; }
	void printList(); //печать решения
	void addTile(int x, int y, int w); //добавление квадрата в решение
	void removeTile(int ind = -1); //удаление квадрата из решения
};


void Tiles::printList() {
	for (int i = 0; i < n; i++) {
		cout << y[i] + 1 << " " << x[i] + 1 << " " << w[i] << endl;
	}
}

void Tiles::addTile(int x, int y, int w) {
	this->x[n] = x;
	this->y[n] = y;
	this->w[n] = w;
	n++;
}

void Tiles::removeTile(int ind) {
	if (ind == -1) ind = n - 1;
	for (int i = ind; i < n - 1; i++) {
		x[i] = x[i + 1];
		y[i] = y[i + 1];
		w[i] = w[i + 1];
	}
	n--;
}
```

```
struct Rectangle {
	int width; //ширина поля
	int height; //высота поля
	int freeCells; //число пустых клеток
	int ** table; //матрица поля
	Rectangle(int width, int height);
	~Rectangle();
	bool isFull(); //проверка заполненности поля
	void print(); //вывод матрицы поля на экран
	void clear(Tiles tl); //удаление последнего квадрата с поля
	bool place(Tiles tl); //размещение последнего квадрата на поле
	void tryToFit(Tiles& curr, int tileSize, int x, int y); //поиск
наибольшего квадрата, который можно разместить на поле
	bool removeTail(Tiles& curr); //удаление «хвоста» разбиения
	void covering(int tileSize, Tiles& curr, Tiles& best); //поиск
минимального разбиения
};


Rectangle::Rectangle(int width, int height) : width(width), height(height) {
	freeCells = width * height;
	table = new int*[height];
	for (int i = 0; i < height; i++) {
		table[i] = new int[width];
		for (int j = 0; j < width; j++)
			table[i][j] = 0;
	}
}

Rectangle::~Rectangle() {
	for (int i = 0; i < height; i++) {
		delete[] table[i];
	}
	delete[] table;
}

bool Rectangle::isFull() {
	return (freeCells == 0);
}

void Rectangle::print() {
	streamsize w = cout.width();
	for (int i = 0; i < height; i++) {
		for (int j = 0; j < width; j++) {
			cout.width(3);
			cout << table[i][j] << " ";
		}
		cout << endl;
```

```
	}
	cout << endl;
	cout.width(w);
}

void Rectangle::clear(Tiles tl) {
	for (int i = tl.y[tl.n - 1]; i < tl.y[tl.n - 1] + tl.w[tl.n - 1] && i < height; i++) {
		for (int j = tl.x[tl.n - 1]; j < tl.x[tl.n - 1] + tl.w[tl.n - 1] && j < width; j++) {
			if (table[i][j] == tl.n) {
				table[i][j] = 0;
				freeCells++;
			}
		}
	}
}

bool Rectangle::place(Tiles tl) {
	for (int i = tl.y[tl.n - 1]; i < tl.y[tl.n - 1] + tl.w[tl.n - 1]; i++) {
		for (int j = tl.x[tl.n - 1]; j < tl.x[tl.n - 1] + tl.w[tl.n - 1]; j++) {
			//если при размещении квадрата произошел выход за пределы поля или пересечение с другим квадратом,
			//часть квадрата, уже размещенная на поле, удаляется
			if (i >= height || j >= width || table[i][j] != 0) {
				clear(tl);
				tl.x[tl.n - 1] = j;
				return false;
			}
			table[i][j] = tl.n;
			freeCells--;
		}
	}
	return true;
}

void Rectangle::tryToFit(Tiles& curr, int tileSize, int x, int y) {
	//для каждой возможной длины стороны квадрата осуществляется попытка разместить его по заданным координатам
	for (int w = tileSize; w >= 1; w--) {
		if (freeCells < w*w) continue;
		curr.addTile(x, y, w);
		if (place(curr)) {
			return;
		}
		curr.removeTile();
	}
```

```
}

bool Rectangle::removeTail(Tiles& curr) {
	//удаляется конец массива, состоящий из единичных квадратов
	if (curr.w[curr.n - 1] == 1) {
		for (int k = curr.n - 1; k >= 0; k--) {
			if (curr.w[k] == 1) {
				clear(curr);
				curr.removeTile(k);
			}
			else {
				break;
			}
		}
	}

	/*if (width == height) {
		if (curr.n == 3) { return false; }
	}*/

	//если удалены все квадраты, возвращаем false, чтобы завершить алгоритм
	if (curr.n == 0) { return false; }
	//иначе последний квадрат перед единичными заменяется квадратом со стороной на 1 меньше
	clear(curr);
	curr.addTile(curr.x[curr.n - 1], curr.y[curr.n - 1], curr.w[curr.n - 1] - 1);
	curr.removeTile(curr.n - 2);
	place(curr);
	return true;
}

void Rectangle::covering(int tileSize, Tiles& curr, Tiles& best) {
	while (1) {
		//осуществляется обход матрицы поля
		for (int i = 0; i < height; i++) {
			for (int j = 0; j < width; j++) {
				//добавление наибольшего возможного квадрата
				tryToFit(curr, tileSize, j, i);
				//print();
				//если число элементов в текущем неполном решении не меньше лучшего решения, необходимо удалить квадраты с поля
				if (curr.n >= minK && !isFull()) {
					//удаление "хвоста": при возвращении false алгоритм завершает работу
					if (!removeTail(curr)) return;
					//совершается прыжок по индексам правого верхнего угла последнего квадрата
```

```
                              i = curr.y[curr.n - 1];
                              j = curr.x[curr.n - 1] + curr.w[curr.n - 1] - 1;
                        }
                        else if (isFull()) { //поле заполнено
                              //если текущее решение лучше лучшего, лучшее
заменяется текущим
                              if (curr.n < minK) {
                                    print();
                                    minK = curr.n;
                                    best = curr;
                                    varCounter = 1;
                              }
                              else if (curr.n == minK) {
                                    //если текущее решение равно лучшему,
увеличивается счетчик минимальных разбиений
                                    //print();
                                    varCounter++;
                              }
                              //с поля удаляются последние квадраты, чтобы
рассмотреть другие варианты разбиения
                              if (!removeTail(curr)) return;
                              //print();
                              //прыжок (см. строчки 147-148)
                              i = curr.y[curr.n - 1];
                              j = curr.x[curr.n - 1] + curr.w[curr.n - 1] - 1;
                        }
                  }
            }
      }
}


int main() {
      int height = 0, width = 0;
      while (height < 2 || height > MAX_N || width < 2 || width > MAX_N){
            cout << "Enter the height and the width of a rectangle (2 <= N <=
20): ";
            cin >> height >> width;
      }

      /*while (width < 2 || width > MAX_N) {
            cin >> width;
      }
      height = width;*/

      Rectangle rec(width, height);
      minK = width * height; //наименьшее разбиение; первоначально указано
наибольшее возможное число квадратов
```

```
	varCounter = 1; //счетчик вариантов минимального разбиения - для индивидуализации
	int maxTileSide; //наибольший возможный размер квадрата
	Tiles tiles; //массив для хранения информации о разбиении
	Tiles best;

	if (width == height) {
		if (width % 2 == 0) {
			minK = 4;
			for (int i = 0; i < width; i += width / 2) {
				for (int j = 0; j < width; j += width / 2) {
					best.addTile(j, i, width / 2);
					rec.place(best);
				}
			}
			rec.print();
		}
		else if (width % 3 == 0) {
			minK = 6;
			varCounter = 4;
			int bigSide = 2 * width / 3;
			best.addTile(0, 0, bigSide);
			rec.place(best);
			for (int i = 0; i < bigSide; i += width / 3) {
				best.addTile(i, bigSide, width / 3);
				rec.place(best);
				best.addTile(bigSide, i, width / 3);
				rec.place(best);
			}
			best.addTile(bigSide, bigSide, width / 3);
			rec.place(best);
			rec.print();
		}
		else if (width % 5 == 0) {
			minK = 8;
			varCounter = 32;
			int oneFifth = width / 5;
			best.addTile(0, 0, oneFifth * 3);
			rec.place(best);
			best.addTile(oneFifth * 3, 0, oneFifth * 2);
			rec.place(best);
			best.addTile(0, oneFifth * 3, oneFifth * 2);
			rec.place(best);
			best.addTile(oneFifth * 3, oneFifth * 3, oneFifth * 2);
			rec.place(best);
			best.addTile(oneFifth * 3, oneFifth * 2, oneFifth);
			rec.place(best);
			best.addTile(oneFifth * 4, oneFifth * 2, oneFifth);
```

```
                rec.place(best);
                best.addTile(oneFifth * 2, oneFifth * 3, oneFifth);
                rec.place(best);
                best.addTile(oneFifth * 2, oneFifth * 4, oneFifth);
                rec.place(best);
                rec.print();
            }
            else {

                /*tiles.addTile(0, 0, (width + 1) / 2);
                rec.place(tiles);
                tiles.addTile((width + 1) / 2, 0, (width - 1) / 2);
                rec.place(tiles);
                tiles.addTile(0, (width + 1) / 2, (width - 1) / 2);
                rec.place(tiles);
                maxTileSide = (width - 1)/ 2;*/

                maxTileSide = (width + 1) / 2;
                rec.covering(maxTileSide, tiles, best);
            }
        }
        else {
            maxTileSide = (width <= height ? width : height) - 1; //наибольший
возможный размер квадрата
            rec.covering(maxTileSide, tiles, best);
        }
        cout << minK << endl;
        best.printList();
        cout << "Number of minimum fragmentation varieties: " << varCounter <<
endl;
        return 0;
}
```