МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

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

Студент гр.7383 _____ Васильев А.И.

Преподаватель _____ Жангиров Т.Р.

г. Санкт-Петербург
г. 2019

Цель работы.

Ознакомиться с алгоритмом поиска с возвратом. Реализовать программу, используя данный алгоритм.

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).

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).

Описание алгоритма.

В начале программы проверяется делимость на 2, 3, 5. Если обнаружена делимость на 2, то квадрат разбивается на 4 равных квадрата. Если обнаружена делимость на 3 или 5, то исходный квадрат заменяется меньшим квадратом со стороной 3 или 5 соответственно и запускается функция бэктрекинга. Если делимости не обнаружено, то устанавливаются три первых квадрата со сторонами N/2+1, N/2, N/2. Далее для оставшейся части квадрата вызывается

функция бэктрекинга, которая заполняет оставшуюся область квадратами со стороной не более, чем N/2.

Ход работы.

1) Был создан класс Square с полями square(поле для заполнения квадратами), count(количество установленных квадратов), best_square(лучшая расстановка квадратов), best_count(наименьшее число квадратов) и рядом методов для работы этими полями.

2) Был написан метод void insert(int x, int y, int number), который устанавливает квадрат размера number, начиная с клетки с координатами (x,y).

3) Был написан метод bool is_posible(int x, int y, int number), который проверяет можно ли квадрат размера number установить, начиная с клетки с координатами (x,y).

4) Был написан метод bool is_empty(int & x, int & y), который проверяет наличие пустых клеток в заполняемом исходном квадрате.

5) Был написан метод void remove_square(int x, int y, int number, int **arr) для удаления квадрата заданной стороны, верхний левый угол которого расположен в клетке с координатами (x,y).

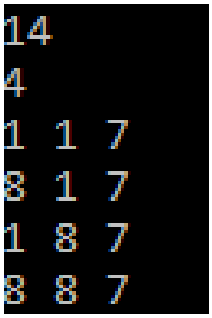
6) Был написан метод void copy_square(), который сохраняет лучший результат.

7) Был написан метод void backtracking(int x, int y), который реализует поиск с возвратом. Метод находит наилучший вариант разложения квадрата на меньшие количества.

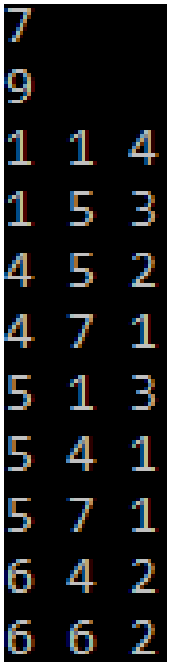
8) Был написан метод void print_result(int k) для вывода результата на экран.

Примеры работы программы.

1) Четная сторона квадрата:



2) Нечетная сторона квадрата:



Выводы.

В ходе выполнения лабораторной работы была написана программа, реализующая алгоритм поиска возврата для заполнения квадрата минимальным количеством меньших квадратов.

Приложение
Исходный код программы

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

using namespace std;

class Square {
	int size;
	int **square;
	int **best_square;
	int count;
	int best_count;
public:
	Square(int size, int count) :size(size), count(count), best_count(size*size) {
		square = new int*[size];
		best_square = new int *[size];
		for (int i = 0; i < size; i++) {
			square[i] = new int[size];
			best_square[i] = new int[size];
			for (int j = 0; j < size; j++) {
				square[i][j] = 0;
				best_square[i][j] = 0;
			}
		}
	}

	~Square() {
		for (int i = 0; i < size; i++) {
			delete[] square[i];
			delete[] best_square[i];
		}
		delete[] square;
		delete[] best_square;
	}

	void start() {
		int tmp = size / 2;
		insert(0, 0, tmp + 1);
		insert(0, tmp + 1, tmp);
		insert(tmp + 1, 0, tmp);
	}

	void insert(int x, int y, int number) {
		for (int i = x; i <x + number; i++) {
			for (int j = y; j <y + number; j++) {
				square[i][j] = number;
			}
		}
		count++;
	}

	bool is_posible(int x, int y, int number) {
		if (x + number > size || y + number > size) {
			return false;
		}
		for (int i = x; i < x + number; i++) {
			for (int j = y; j < y + number; j++) {
				if (square[i][j] != 0) {
					return false;
				}
			}
```

```
		}
		return true;
	}

	bool is_empty(int & x, int & y) {
		while (square[x][y] != 0) {
			if (y == size - 1) {
				if (x == size - 1) {
					return false;
				}
				else {
					x++;
					y = size / 2;
					continue;
				}
			}
			y++;
		}
		return true;
	}

	void remove_square(int x, int y, int number, int **arr) {
		for (int i = x; i <x + number; i++) {
			for (int j = y; j <y + number; j++) {
				arr[i][j] = 0;
			}
		}
	}

	void print_result(int k) {
		cout << best_count << endl;
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				if (best_square[i][j] != 0) {
					cout << i*k + 1 << " " << j*k + 1 << " " <<
best_square[i][j] * k << endl;
					remove_square(i, j, best_square[i][j], best_square);
				}
			}
		}
	}

	void my_print() {
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				cout << square[i][j] << " ";
			}
			cout << endl;
		}
	}

	void my_best_print() {
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				cout << best_square[i][j] << " ";
			}
			cout << endl;
		}
	}

	void copy_square() {
		for (int i = 0; i < size; i++) {
			for (int j = 0; j < size; j++) {
				best_square[i][j] = square[i][j];
```

```
                }
            }
        }

        void backtracking(int x, int y) {
            if (count >= best_count) {
                return;
            }
            for (int n = size / 2; n > 0; n--) {
                if (is_posible(x, y, n)) {
                    insert(x, y, n);
                    int next_x = x;
                    int next_y = y;
                    if (is_empty(next_x, next_y)) {
                        backtracking(next_x, next_y);
                    }
                    else {
                        if (count < best_count) {
                            copy_square();
                            best_count = count;
                        }
                        count--;
                        remove_square(x, y, n, square);
                        return;
                    }
                    count--;
                    remove_square(x, y, n, square);
                }
            }
        }

};

void devided_by_two(int size) {
    cout << 4 << endl;
    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 size = 0, k = 1;
    cin >> size;
    if (size % 2 == 0) {
        devided_by_two(size / 2);
    }
    else {
        if (size % 3 == 0) {
            k = size / 3;
            size = 3;

        }
        else if (size % 5 == 0) {
            k = size / 5;
            size = 5;
        }
        Square A(size, 0);
        A.start();
        A.backtracking(size / 2, size / 2 + 1);
        A.print_result(k);
    }
    system("pause");
    return 0;
}
```