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. 9383 _____ Моисейченко К.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Изучить и применить на практике алгоритм поиска с возвратом для решения поставленной задачи.

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. 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 <= N <= 40).

Необходимо найти число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N и вывести K строк, каждая из которых должна содержать три целых числа x, y, и w, задающие координаты левого верхнего угла и длину стороны соответствующего обрезка (квадрата).

Вариант 5р. Возможность задать список квадратов (от 0 до N^2 квадратов в списке), которые обязательно должны быть использованы в покрытии квадрата со стороной N.

Выполнение работы:

1. Была создана структура Square, хранящая координаты левого верхнего угла квадрата и длину его стороны.
2. Был разработан класс Field, представляющий собой “столешницу” - квадрат для разбиения. Хранит в себе квадратное поле, состоящее из чисел,

использованные квадраты, размер столешницы и площадь, не покрытую квадратами.

3. Был разработан рекурсивный бэктрекинг-алгоритм:

Поиск с возвратом (англ. backtracking) — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве M.

1) Функция Backtracking принимает в качестве аргументов текущие координаты x, y, а также текущее и минимальное поле по ссылкам.

2) Координаты сдвигаются на ближайшую свободную клетку. Если свободных клеток нет, функция завершается.

3) В цикле длины от N-1 до 1, проверяется на пересечения или выход за границы квадрат с итерируемой длиной стороны и вставляется наибольший доступный квадрат в текущие координаты.

4) Если после вставки поле полностью заполнено, сравнивается количество квадратов текущего и минимального поля, если у текущего поля квадратов меньше, минимальному полю присваивается текущее, а если больше, последний вставленный квадрат удаляется и завершается цикл длины.

5) Если поле не заполнено, функция Backtracking запускается рекурсивно с координатами x+1, y.

6) Удаляется последний вставленный квадрат и цикл длины итерируется дальше.

Таким образом, функция проверяет все возможные способы квадрирования столешницы.

4. Для оптимизации алгоритма были отдельно рассмотрены частные случаи: если длина стороны столешницы четная, то минимальное количество квадратов в разбиении = 4, поле делится на 4 равных квадрата.

Если длина стороны столешницы кратна 3, то минимальное количество квадратов в разбиении = 6.

Если длина стороны столешницы кратна 5, то минимальное количество квадратов в разбиении = 8.

В остальных случаях (т.е. когда длина стороны - простое число, т.к. N<=40) точно содержатся 3 квадрата с длинами сторон N/2 + 1, N/2, N/2 (т.к. N - нечетное, это самый оптимальный способ покрыть бóльшую часть столешницы). Таким образом с помощью поиска с возвратом остается обработать лишь четверть квадрата. Код решения задания без варианта приведен в приложении А (файл main0.cpp).

5. Была разработана программа, которая учитывает квадраты, которые обязательно должны участвовать в разбиении. В этом варианте описанная выше оптимизация работает не во всех случаях, поэтому функцией бэктрекинга обрабатывается вся столешница. Код решения задания варианта 5р приведен в приложении А (файл main1.cpp).

Так как перебираются все подмножества множества клеток столешницы, сложность разработанного алгоритма - $O(2^N)$.

Поля и методы класса Field:

-int freeArea - площадь, не покрытая квадратами.

-int n - размер столешницы.

-vector<vector<int>> field - поле столешницы, состоящее из чисел.

-vector<Square> squares - массив квадратов, используемых в разбиении.

+int SquaresAmount() - возвращает количество используемых квадратов.

+int GetN() - возвращает значение поля n.

+int GetArea() - возвращает значение поля freeArea.

+void PrintSquares() - выводит использованные квадраты на экран.

+void PrintField() - выводит поле столешницы на экран.

+void AddSquare(Square& square) - добавляет квадрат в разбиение, уменьшает площадь freeArea.

+bool CheckIntersection(Square& square) - проверяет, нет ли пересечений у квадрата с добавленными ранее квадратами.

+void RemoveLastSquare() - удаляет последний добавленный квадрат, увеличивает обратно площадь freeArea.

Примеры работы программы

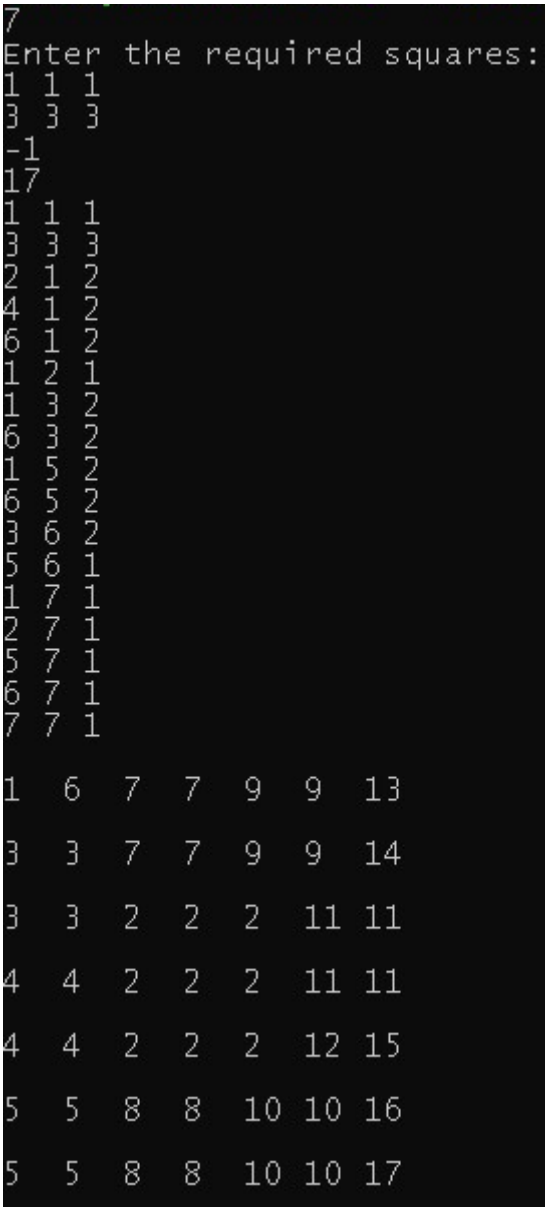

Рисунок 1 - Пример работы программы 1.

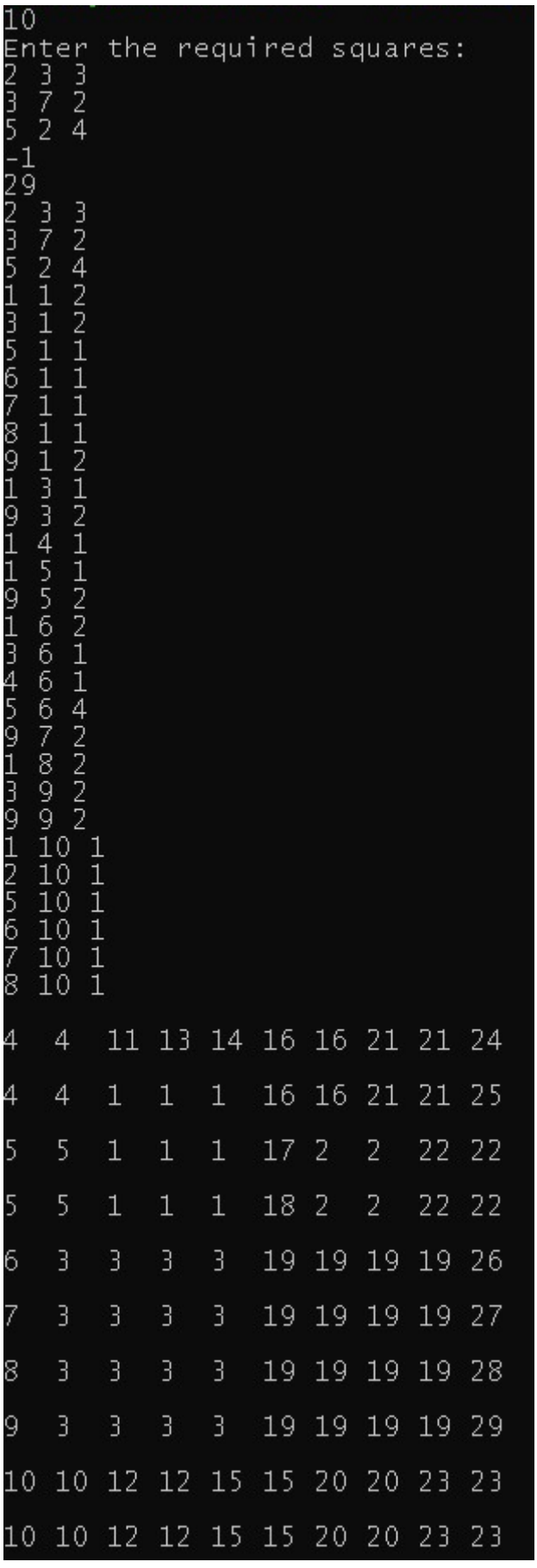

Рисунок 2 - Пример работы программы 2.

Выводы.

В выполненной лабораторной работе был освоен и применен на практике алгоритм поиска с возвратом. Был найден способ оптимизации алгоритма для конкретной задачи, а также была разработана программа, которая решает расширение задачи.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: field.h

```
#pragma once

#include <iostream>
#include <vector>

struct Square {
int x, y;
int size;
Square(int x, int y, int size) : x(x), y(y), size(size) {}
};

class Field {
private:
int freeArea;
int n;
std::vector<std::vector<int>> field;
std::vector<Square> squares;

public:
Field(int n);

std::vector<int>& operator[](int index);

int SquaresAmount();
int GetN();
int GetArea();
void PrintSquares();
void PrintField();

void AddSquare(Square& square);
bool CheckIntersection(Square& square);
void RemoveLastSquare();
};
```

Название файла: field.cpp

```
#include "field.h"

Field::Field(int n) {
this->n = n;
freeArea = n * n;
field.resize(n, std::vector<int>(n, 0));
squares.reserve(n * n);
}

std::vector<int>& Field::operator[](int index) {
return field[index];
}

int Field::SquaresAmount() {
return squares.size();
```

```
}

int Field::GetN() {
return n;
}

int Field::GetArea() {
return freeArea;
}
void Field::PrintSquares() {
for (const auto& square : squares) {
std::cout << square.x << " " << square.y << " " << square.size <<
std::endl;
}
}

void Field::PrintField() {
for (int i = 0; i < n; i++) {
for (int j = 0; j < n; j++) {
std::cout << field[j][i];
                        if(field[j][i] < 10) {
                            std::cout << ' ';
                        }
                        std::cout << ' ';
}
std::cout << "\n\n";
}
}

void Field::AddSquare(Square& square) {
if (square.x + square.size > n + 1 || square.y + square.size > n +
1)
return;

for (int i = square.x; i < square.x + square.size; i++) {
for (int j = square.y; j < square.y + square.size; j++) {
field[j - 1][i - 1] = squares.size() + 1;
}
}

squares.push_back(square);
freeArea -= square.size * square.size;
}

bool Field::CheckIntersection(Square& square) {
for (int i = square.x; i < square.x + square.size; ++i) {
for (int j = square.y; j < square.y + square.size; ++j) {
if (field[j - 1][i - 1])
return true;
}
}

return false;
}

void Field::RemoveLastSquare() {
if (squares.empty()) {
```

```
    return;
    }

    Square& square = squares.back();
    for (int i = square.x; i < square.x + square.size; ++i) {
    for (int j = square.y; j < square.y + square.size; ++j) {
    field[j - 1][i - 1] = 0;
    }
    }
    freeArea += square.size * square.size;
    squares.pop_back();
    }
```

Название файла: main0.cpp

```
    #include <iostream>
    #include "field.h"

    void Backtracking(int x, int y, Field& currentField, Field& minField)
{
        int n = currentField.GetN();
        while (x <= n && y <= n && currentField[y - 1][x - 1]) {
            x++;
            if (x == n + 1) {
                x = 1;
                y++;
            }
        }
        if (y == n + 1)
            return;

        Square square(x, y, n - 1);
        for (int size = n - 1; size >= 1; size--) {
            square = { x, y, size };
            if (x + size > n + 1 || y + size > n + 1 ||
currentField.CheckIntersection(square))
                continue;

            currentField.AddSquare(square);

            if (currentField.GetArea() == 0 &&
currentField.SquaresAmount() < minField.SquaresAmount()) {
                minField = currentField;
            }
            else if (currentField.SquaresAmount() >=
minField.SquaresAmount()) {
                currentField.RemoveLastSquare();
                break;
            }
            else {
                Backtracking(x + 1, y, currentField, minField);
            }

            currentField.RemoveLastSquare();
        }
    }
```

```
int main() {
    int n;
    std::cin >> n;

    Field minField(n);
    Field currentField(n);

    if (n % 2 == 0) {
        Square square(1, 1, n / 2);
        minField.AddSquare(square);
        square = { n / 2 + 1, 1, n / 2 };
        minField.AddSquare(square);
        square = { 1, n / 2 + 1, n / 2 };
        minField.AddSquare(square);
        square = { n / 2 + 1, n / 2 + 1, n / 2 };
        minField.AddSquare(square);
    }
    else if (n % 3 == 0) {
        Square square(1, 1, 2 * n / 3);
        minField.AddSquare(square);
        square = { 1 + 2 * n / 3, 1, n / 3 };
        minField.AddSquare(square);
        square = { 1, 2 * n / 3 + 1, n / 3 };
        minField.AddSquare(square);
        square = { 2 * n / 3 + 1, n / 3 + 1, n / 3 };
        minField.AddSquare(square);
        square = { n / 3 + 1, 2 * n / 3 + 1, n / 3 };
        minField.AddSquare(square);
        square = { 2 * n / 3 + 1, 2 * n / 3 + 1, n / 3 };
        minField.AddSquare(square);
    }
    else if (n % 5 == 0) {
        Square square(1, 1, 3 * n / 5);
        minField.AddSquare(square);
        square = { 3 * n / 5 + 1, 1, 2 * n / 5 };
        minField.AddSquare(square);
        square = { 1, 3 * n / 5 + 1, 2 * n / 5 };
        minField.AddSquare(square);
        square = { 3 * n / 5 + 1, 3 * n / 5 + 1, 2 * n / 5 };
        minField.AddSquare(square);
        square = { 2 * n / 5 + 1, 3 * n / 5 + 1, n / 5 };
        minField.AddSquare(square);
        square = { 2 * n / 5 + 1, 4 * n / 5 + 1, n / 5 };
        minField.AddSquare(square);
        square = { 3 * n / 5 + 1, 2 * n / 5 + 1, n / 5 };
        minField.AddSquare(square);
        square = { 4 * n / 5 + 1, 2 * n / 5 + 1, n / 5 };
        minField.AddSquare(square);
    }
    else {
        Square square(1, 1, (n + 1) / 2);
        currentField.AddSquare(square);
        square = { (n + 3) / 2, 1, n / 2 };
        currentField.AddSquare(square);
        square = { 1, (n + 3) / 2, n / 2 };
```

```
            currentField.AddSquare(square);

            for (int x = 1; x <= n; ++x) {
                for (int y = 1; y <= n; ++y) {
                    square = { x, y, 1 };
                    minField.AddSquare(square);
                }
            }

            Backtracking(1, 1, currentField, minField);
        }

        std::cout << minField.SquaresAmount() << std::endl;
        minField.PrintSquares();

        return 0;
    }
```

Название файла: main1.cpp

```
    #include <iostream>
    #include "field.h"


    void Backtracking(int x, int y, Field& currentField, Field& minField)
{
        int n = currentField.GetN();
        while (x <= n && y <= n && currentField[y - 1][x - 1]) {
            x++;
            if (x == n + 1) {
                x = 1;
                y++;
            }
        }
        if (y == n + 1)
            return;

        Square square(x, y, n - 1);
        for (int size = n - 1; size >= 1; size--) {
            square = { x, y, size };
            if (x + size > n + 1 || y + size > n + 1 ||
currentField.CheckIntersection(square))
                continue;

            currentField.AddSquare(square);

            if (currentField.GetArea() == 0 &&
currentField.SquaresAmount() < minField.SquaresAmount()) {
                minField = currentField;
            }
            else if (currentField.SquaresAmount() >=
minField.SquaresAmount()) {
                currentField.RemoveLastSquare();
                break;
            }
            else {
```

```
            Backtracking(x + 1, y, currentField, minField);
        }

        currentField.RemoveLastSquare();
    }
}


int main() {
    int n;
    std::cin >> n;

    Field minField(n);
    Field currentField(n);

    Square square(1, 1, 1);
    for (int x = 1; x <= n; ++x) {
        for (int y = 1; y <= n; ++y) {
            square = { x, y, 1 };
            minField.AddSquare(square);
        }
    }

    std::cout << "Enter the required squares:" << std::endl;
    for (int i = 0; i < n * n; ++i) {
        int x;
        std::cin >> x;
        if (x == -1)
            break;
        int y, size;
        std::cin >> y >> size;

        square = { x, y, size };
        currentField.AddSquare(square);
    }

    Backtracking(1, 1, currentField, minField);

    std::cout << minField.SquaresAmount() << std::endl;
    minField.PrintSquares();
    std::cout << '\n';
    minField.PrintField();

    return 0;
}
```