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. 8304 _____ Мешков М.А.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург

2020

Цель работы.

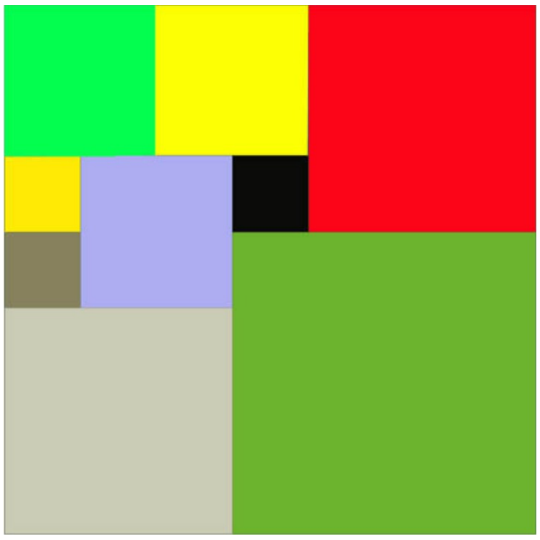
Научиться применять алгоритм поиска с возвратом (бэктрекинг) для решения задач и оценивать его сложность.

Основные теоретические положения.

Вариант 1р. Рекурсивный бэктрекинг. Поиск решения за разумное время (меньше 2 минут) для $2 \le N \le 40$.

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 ($0 \le x,y < N$) и длину стороны соответствующего обрезка (квадрата).

Описание функций и структур данных.

Для решения задачи был реализована структура Square для хранения квадратов, класс Area для хранения заполненной квадратами области.

Частичные решения хранятся в массиве квадртаов minSquares класса Area.

Была создана функция printSquares для вывода частичных решений и решений на экран.

Функция main принимает размер квадрата и выводит ответ. Для вывода частичных решений программе при запуске нужно передать опцию -v.

Функция findMinSquares является функцией запускающей алгоритм решения задачи. Она принимает размер квадрата и возвращает минимальное разбиение.

Функция findMinSquaresRecursively непосредственно решает задачу, рекурсивно вызывая саму себя. Она принимает класс Area и в него же сохраняет частичные решения.

Описание алгоритма.

Для решения задачи был использован рекурсивный поиск с возвратом.

На каждом этапе рекурсии находим самый верхний левый угол, образованный «стенками» квадрата, который нужно замостить, и/или уже помещёнными квадратами. Очевидно, что в этом углу должен быть расположен квадрат, поэтому в этот угол поочерёдно располагаются квадраты допустимых размеров (не выходящих за границу изначального квадрата и не пересекающих

уже размещённые квадраты) и для каждого нового квадрата рекурсивно вызывается функция, выполняющая все те же операции.

Если после вставки очередного квадрата всё поле N x N оказалось заполнено, то решение сохраняется при условии, что количество квадратов в нём не превосходит количества квадратов в уже найденном.

Используемые оптимизации.

1. Можно дать верхнюю оценку количества необходимых квадратов N+3 для нечетного N=2k+1 (см. рис 1).

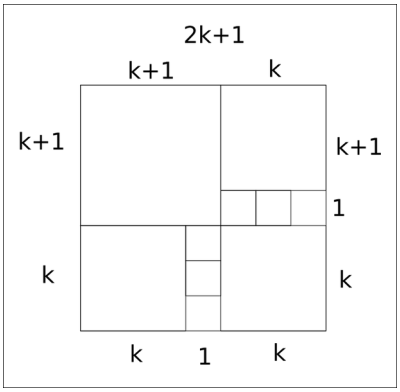


Рисунок 1 — Верхняя оценка

2. Ветвь поиска отсекается, если глубина превосходит размер уже найденного решения (или значения верхней оценки).

3. Для квадратов размеров кратных 2, 3, 5 можно дать верхние оценки равные 4, 6, 8 (очевидные разбиения).

4. Если N — простое и нечетное, то три квадрата размерами N / 2 + 1, N / 2 и N / 2 в решении будут размещены в верхнем левом углу (как на рисунке 1). Добавление этих квадратов перед началом перебора позволяет уменьшить размер замощаемой площади в ~4 раза.

5. Перебор размеров квадратов начинается с наибольшей возможной стороны.

Оценка сложности алгоритма.

Как указано выше количество размещаемых квадратов не превосходит N+3. Максимальный размер каждого квадрата N-1. После перемножения всех возможных размеров для N+3 квадратов получаем оценку сложности по времени O(N^(N+3)). Сложность по памяти пропорциональна максимальной глубине рекурсии, которая равна максимальному количеству квадратов (N+3), и месту, выделяемому для хранения промежуточных решений и ответа (тоже линейно зависит от N). Получаем сложность по памяти O(N).

Тестирование.

```
11
0 0 6
6 0 5
0 6 5
6 5 3
9 5 2
5 6 1
5 7 1
9 7 1
10 7 1
5 8 3
8 8 3
```

N = 11

```
4
0 0 7
7 0 7
0 7 7
7 7 7
```

N = 14

```
13
0 0 12
12 0 11
0 12 11
12 11 2
14 11 5
19 11 4
11 12 1
11 13 3
19 15 1
20 15 3
11 16 7
18 16 2
18 18 5
```

N = 23

Оценка сложности алгоритма.

Для всех N <= 40 алгоритм работает значительно быстрее требуемых 2-х минут.

Выводы.

В ходе работы была написана программа, решающая поставленную задачу с использованием алгоритма поиска с возвратом. В алгоритме использовались оптимизации, была проанализирована сложность составленного алгоритма. Была протестирована корректность работы алгоритма и выполнение условий на скорость его работы.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ НА ЯЗЫКЕ C++

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

bool verboseMode = false;

struct Square {
    int x, y, size;
};

class Area;
void printArea(const Area &area);

struct Area {
    int n;
    Area(int n) : n(n), filled(n, 0) {}
    std::vector<int> filled; // key is column, value is rows from top
    std::vector<Square> squares, minSquares;
    int minSquaresAmount = 0;

    int getToppestCol() {
        int min = n;
        int minI = -1;
        for (int i = 0; i < n; i++) {
            if (filled[i] < min) {
                minI = i;
                min = filled[i];
            }
        }
        return minI;
    }
    int toBelow(int col) {
        return n - filled[col];
    }
    int toRight(int col) {
        int endCol = col + 1;
        int row = filled[col];
        for ( ; endCol < n; endCol++) {
            if (filled[endCol] > row)
                break;
        }
        return endCol - col;
    }
    void addSquare(int topLeftCol, int size) {
```

```
        if (verboseMode) {
            std::cout << "Adding square " << topLeftCol << ' ' <<
filled[topLeftCol] << ' ' << size << std::endl;
        }
        squares.push_back({topLeftCol, filled[topLeftCol], size});
        for (int i = 0; i < size; i++) {
            filled[topLeftCol + i] += size;
        }
        if (verboseMode) {
            printArea(*this);
        }
    }
    void removeSquare(int topLeftCol, int size) {
        if (verboseMode) {
            std::cout << "Removing square " << topLeftCol << ' ' <<
filled[topLeftCol] << ' ' << size << std::endl;
        }
        for (int i = 0; i < size; i++) {
            filled[topLeftCol + i] -= size;
        }
        squares.pop_back();
        if (verboseMode) {
            printArea(*this);
        }
    }
    void saveMinSquares() {
        if (squares.size() < minSquaresAmount) {
            minSquaresAmount = squares.size();
            minSquares = squares;
            if (verboseMode) {
                std::cout << "Saving current squares as minimal (" <<
squares.size() << " squares)." << std::endl;
            }
        }
    }
};

void printArea(const Area &area) {
    std::vector<std::vector<unsigned char>> toPrint(area.n,
std::vector<unsigned char>(area.n, '.'));
    for (auto i = 0; i < area.squares.size(); i++) {
        auto square = area.squares[i];
        for (int row = square.y; row < square.y + square.size; row++) {
            for (int col = square.x; col < square.x + square.size; col++)
{
                toPrint[row][col] = (i < 26 ? 'a' + i : 'A' + (i - 26));
            }
```

```
            }
        }
        for (int row = 0; row < area.n; row++) {
            for (int col = 0; col < area.n; col++) {
                std::cout << toPrint[row][col] << ' ';
            }
            std::cout << std::endl;
        }
    }

    void printSquares(const std::vector<Square> &squares) {
        std::cout << squares.size() << std::endl;
        for (auto square : squares) {
            std::cout << square.x << " " << square.y << " " << square.size <<
    std::endl;
        }
    }

    decltype(Area::squares) findMinSquares(int n);

    int main(int argc, char *argv[]) {
        if (argc != 1) {
            verboseMode = true;
        }
        int n;
        std::cin >> n;
        auto squares = findMinSquares(n);
        if (verboseMode) {
            std::cout << "Solution: " << std::endl;
        }
        printSquares(squares);
        return 0;
    }

    void findMinSquaresRecursively(Area &area);

    decltype(Area::squares) findMinSquares(int n) {
        Area area(n);
        area.minSquaresAmount = (n + 3) + 1; // +1 to record squares
        if (n % 2 != 0 && n % 3 != 0 && n % 5 != 0) {
            int size = n / 2 + 1;
            area.addSquare(0, size);
            area.addSquare(size, size - 1);
            area.addSquare(0, size - 1);
            findMinSquaresRecursively(area);
        }
```

```
    else {
        if (n % 2 == 0) area.minSquaresAmount = 4 + 1; // +1 to record
squares
        else if (n % 3 == 0) area.minSquaresAmount = 6 + 1; // +1 to
record squares
        else if (n % 5 == 0) area.minSquaresAmount = 8 + 1; // +1 to
record squares
        findMinSquaresRecursively(area);
    }
    return area.minSquares;
}

void findMinSquaresRecursively(Area &area) {
    int toppestCol = area.getToppestCol();
    if (toppestCol == -1) {
        area.saveMinSquares();
        return;
    }

    if (area.squares.size() >= area.minSquaresAmount - 1)
        return;

    int maxSize = std::min(area.toBelow(toppestCol),
area.toRight(toppestCol));
    maxSize = std::min(maxSize, area.n - 1);

    for (int size = maxSize; size >= 1; size--) {
        area.addSquare(toppestCol, size);
        findMinSquaresRecursively(area);
        area.removeSquare(toppestCol, size);
    }
}
```