

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. 6382

Сергеев А.Д.

Преподаватель

Размочаева Н.А.

Санкт-Петербург

2019

Цель работы.

Научиться использовать алгоритм поиска с возвратом на примере задачи о мощности квадрата. Дополнительная задача: исследование зависимости количества операций от размера квадрата.

Основные теоретические положения.

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.

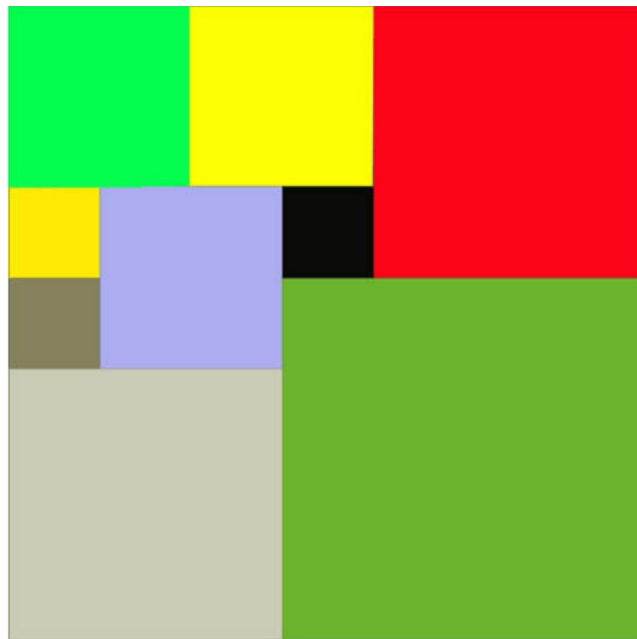


Рисунок 1 — Столешница размера $7 * 7$

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.

Поиск с возвратом — это метод перебора всех возможных конфигураций поискового пространства. На каждом шаге перебора с возвратом предпринимается попытка расширить некоторое заданное частичное решение путем добавления в него еще одного элемента. После этого необходимо проверить, является ли полученное множество решением и, если да, завершить работу алгоритма, а если нет, проверить, возможно ли расширить полученное множество до решения. Если возможно — надо сделать еще один шаг алгоритма, приняв полученное множество за частичное решение, а если нет, удалить последний добавленный элемент множества и подставить на его место следующий возможный элемент, после чего проверить заново.

Порядок выполнения работы.

Написание работы производилось на базе операционной системы Windows 10 на языке программирования java в среде программирования IntelliJ IDEA.

Для упрощения работы алгоритма было принято решение о том, что квадраты с кратными сторонами будут обрабатываться одинаково, для этого был написан метод масштабирования квадратов (в конструкторе класса *PseudoTree*) в самом начале поиска решения. Сразу после масштабирования алгоритм обрабатывает квадраты с четными сторонами, так как очевидно, что их оптимально можно разделить на четыре части.

Далее было замечено, что для квадрата с нечетной стороной N оптимальному разбиению всегда принадлежит квадрат со стороной $(N / 2) + (N \% 2)$ — он находится в одном из углов, а два соседних угла занимают два квадрата со стороной $(N / 2) - (N \% 2)$. Таким образом можно упростить задачу, подвергнув разбиению каждый раз не большой квадрат, а получившуюся после подстановки указанных выше квадратов область — квадрат со стороной $(N / 2) + (N \% 2)$, один из углов которого будет «выколот» — то есть в одном из углов которого будет одна

занятая клетка. В том же конструкторе класса *PseudoTree* к решению добавляются указанные 3 квадрата и, при необходимости, создается объект класса *TableCoverage* для указанного квадрата с выколотым углом.

Класс *PseudoTree* указанным образом упрощает и обрабатывает задачу, он же содержит функцию вычисления идеального мощения — *buildAndParseTree*, в качестве аргумента в которую передается флаг, при положительном значении которого используется метод построения дерева всевозможных решений, а при отрицательном — метод поиска с возвратом. Также этот класс содержит метод *checkList*, проверяющий решение и выводящий в консоль результат мощения.

Класс *TableCoverage* представляет собой область, которая подвергается мощению. Он содержит в себе поле *table*, двойной массив нулей и единиц, изображающий занятые и незанятые клетки, а также поле *size* — максимально возможную сторону вложенного квадрата. Заполнение свободной области производится по рядам. Метод *findSquare* находит первую свободную клетку и проверяет, квадрат с какой максимальной стороной в нее можно вставить. Сторона может быть ограничена границей самой области, уже занятым участком или полем *size*. Методы *cover* и *uncover* помечают указанные области занятыми или свободными соответственно. Следует обратить внимание, что в начале работы алгоритма в целях оптимизации поле *size* принимает значение $(N / 2) + (N \% 2)$ для области со стороной N , так как было замечено, что квадраты с большей стороной оптимальному решению не принадлежат.

Класс *MonoBitArray* представляет из себя реализацию двойного массива нулей и единиц на базе одинарного массива целочисленных переменных. Это позволяет экономить память во время вычисления решения тогда, когда это необходимо.

Класс *Square* изображает квадрат с координатами вершины, стороной и методы работы с ним.

Саму операцию поиска с возвратом выполняет метод класса *PseudoTree* *backtrackRows*. Поиск с возвратом выполняется по описанным в теоретических положениях правилам.

Описание классов в UML-виде приложено к отчету в файле UML.png.

Кроме самого метода поиска минимального замощения был произведен расчет зависимости количества операций от размера квадрата. За единичную операцию был принят процесс нахождения следующего подходящего для замощения квадрата. В следствии предпринятых оптимизаций рассматривать имеет смысл только квадраты, у которых сторона является простым числом, так как решение кратных им квадратов будет сведено до решения их. Также было замечено, что начиная с квадрата со стороной в 53 клетки на поиск решения тратится слишком много времени, поэтому рассмотрим простые числа до 53. Полученная зависимость приведена на рисунке 2.

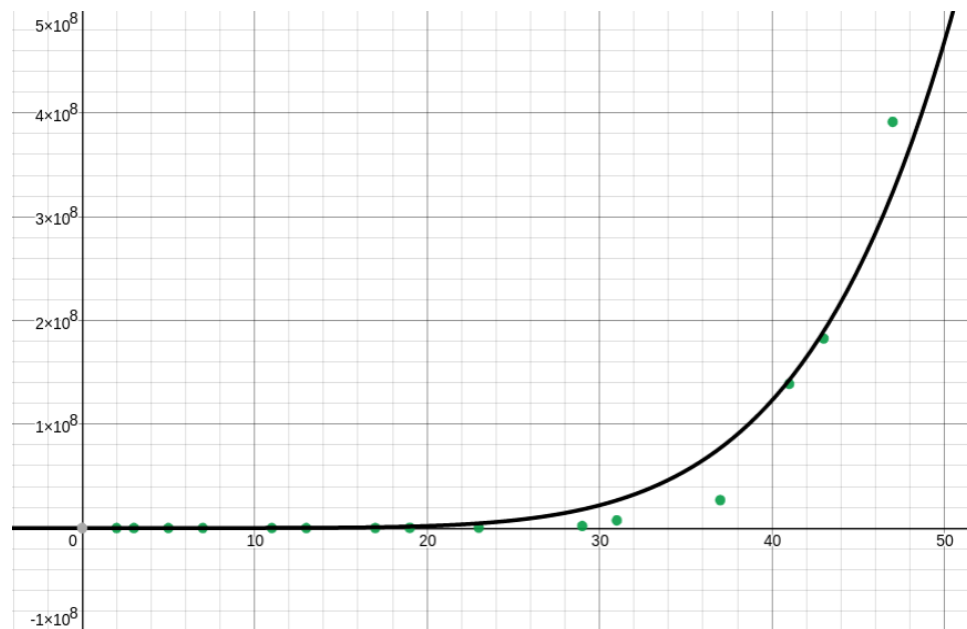


Рисунок 2 — Зависимость количества операций от размера квадрата

Очевидно, существует степенная зависимость количества операций от размера квадрата. Четная аппроксимирующая линия — график функции x^6 .

Вывод.

В результате лабораторной работы были получены знания об алгоритме поиска с возвратом, а также случаях его применения.

ПРИЛОЖЕНИЯ

Приложение А

Исходный код программы, файл Main.java

```
import java.util.Arrays;
import java.util.LinkedList;
import java.util.Scanner;

public class Main {

    public static void main(String[] args) {
        for (int i = 2; i <= 100; i++) {
            PseudoTree tree = new PseudoTree(i);

            int finalI = i;
            if ((i < 50) &&
(Arrays.stream(PseudoTree.smp).anyMatch(j -> j == finalI))) {
                LinkedList<Square> tros =
tree.buildAndParseTree(false);
                System.out.println("For " + i + "*" + i + "
square it took " + tree.leavesNumber + " operations. Number of
squares: " + tros.size());
            }
        }

        /*Scanner sc = new Scanner(System.in);
        int size = sc.nextInt();

        PseudoTree tree = new PseudoTree(size);
        LinkedList<Square> ans = tree.buildAndParseTree();
```

```
        System.out.println(ans.size());
        for (Square sq : ans) {
            System.out.println(sq.toString());
        }

        System.out.println();
        System.out.println(tree.leavesNumber);
        System.out.println();
        System.out.println(PseudoTree.checkList(ans, size));*/
    }
}
```


Приложение Б

Исходный код программы, файл PseudoTree.java

```
import java.util.Arrays;
import java.util.Collections;
import java.util.LinkedList;
import java.util.Stack;

public class PseudoTree {
    public static final int [] smp = new int [] {97, 89, 83, 79,
73, 71, 67, 61, 59, 53, 47, 43, 41, 37, 31, 29, 23, 19, 17, 13, 11,
7, 5, 3, 2};

    private int multiplier = 1;
    private LinkedList<Square> head;
    private TableCoverage root;
    private boolean hasTail;
    private int tailOffset;
    private int halfSize;

    public int leavesNumber = 0;

    public PseudoTree(int sz) {
        for (int i = 0; i < smp.length; i++) {
            if ((sz % smp[i] == 0) && (sz / smp[i] != 1)) {
                sz /= smp[i];
                multiplier *= smp[i];
                i--;
            }
        }

        head = new LinkedList<>();
```

```

halfSize = sz / 2 + (sz == 2 ? 0 : 1);

head.add(new Square(0, 0, halfSize));
head.add(new Square(halfSize, 0, sz / 2));
head.add(new Square(0, halfSize, sz / 2));

if (sz == 2) {
    head.add(new Square(halfSize, halfSize, halfSize));
    hasTail = false;
} else {
    tailOffset = halfSize;
    root = new TableCoverage(halfSize);
    hasTail = true;
}
}

```

```

public LinkedList<Square> buildAndParseTree(boolean useTree)
{
    LinkedList<Square> answer = new LinkedList<>();
    for (Square sq : head) answer.push(sq);

    if (hasTail) {
        Stack<Square> tail;
        if (useTree) tail = getIteration();
        else tail = backtrackRows();

        for (Square sq : tail) {
            sq.setX(2*tailOffset - sq.getSize() - sq.getX()
- 1);

```

```

        sq.setY(2*tailOffset - sq.getSize() - sq.getY()
- 1);

    }

    answer.addAll(tail);
}

for (Square sq : answer) {
    sq.setX(1 + sq.getX() * multiplier);
    sq.setY(1 + sq.getY() * multiplier);
    sq.setSize(sq.getSize() * multiplier);
}

return answer;
}

private Stack<Square> backtrackRows() {
    Stack<Square> filling = new Stack<>();
    int maxSize = halfSize * halfSize;
    Stack<Square> idealFilling = new Stack<>();

    while (maxSize > 1) {
        Square novus = root.addSquare();
        root.setSize(halfSize);

        while (novus != null) {
            filling.push(novus);
            root.cover(novus);
            if ((!idealFilling.isEmpty()) && (filling.size()
> idealFilling.size())) break;
            novus = root.addSquare();
            leavesNumber++;

```

```

    }

    if ((filling.size() < idealFilling.size()) ||
(idealFilling.isEmpty())) {
        idealFilling.clear();
        idealFilling.addAll(filling);
    }

    Square top;
    maxSize = 1;
    while (!filling.isEmpty()) {
        top = filling.pop();
        maxSize = top.getSize();
        root.uncover(top);
        if (maxSize > 1) {
            root.setSize(maxSize - 1);
            break;
        }
    }
}

return idealFilling;
}

private Stack<Square> getIteration() {
    TableCoverage complete = iterateRowsUntilSuccess();
    Stack<Square> tail = new Stack<>();

    while (complete.getParent() != null) {
        tail.push(complete.getPayload());
        complete = complete.getParent();
    }
}

```

```

        return tail;
    }

    private TableCoverage iterateRowsUntilSuccess() {
        LinkedList<TableCoverage> currentRow = new
LinkedList<>(Collections.singletonList(root));
        LinkedList<TableCoverage> newRow = new LinkedList<>();
        TableCoverage finalContainer = null;

        while (finalContainer == null) {
            for (TableCoverage leaf : currentRow) {
                LinkedList<Square> children = leaf.addSquares();
                if (children.isEmpty()) {
                    finalContainer = leaf;
                    break;
                } else leavesNumber += children.size();
                for (Square square : children) newRow.add(new
TableCoverage(leaf, square));
            }
            currentRow = newRow;
            newRow = new LinkedList<>();
        }

        return finalContainer;
    }

    public static String checkList(LinkedList<Square> squares,
int size) {
        char [][] form = new char [size][size];

```

```

        for (int k = 0; k < squares.size(); k++) {
            for (int i = squares.get(k).getY() - 1; i <
squares.get(k).getY() - 1 + squares.get(k).getSize(); i++) {
                for (int j = squares.get(k).getX() - 1; j <
squares.get(k).getX() - 1 + squares.get(k).getSize(); j++) {
                    form[i][j] = (char) (k + 'A');
                }
            }
        }

        StringBuilder sb = new StringBuilder();
        for (char [] chars : form) {
            sb.append(Arrays.toString(chars)).append("\n");
        }
        return sb.toString();
    }
}

```

Приложение В

Исходный код программы, файл TableCoverage.java

```
import java.util.LinkedList;

public class TableCoverage {
    private TableCoverage parent;

    private MonoBitArray table;
    private Square lastAdded;
    private int size;

    public TableCoverage(int sz) {
        parent = null;

        table = new MonoBitArray(sz);
        table.addLine(sz - 1, 1);
        size = sz / 2 + 1;
    }

    public TableCoverage(TableCoverage parental, Square
additional) {
        parent = parental;

        table = new MonoBitArray(parental.table.size());
        for (int i = 0; i < parental.table.size(); i++) {
            table.addLine(i, parental.table.getLine(i));
        }
        lastAdded = additional;
        size = parental.size;
    }
}
```

```

        for (int i = additional.getY(); i < additional.getY() +
additional.getSize(); i++) {
            for (int j = additional.getX(); j <
additional.getX() + additional.getSize(); j++) {
                table.addLine(i, 1 << (table.size() - 1 - j));
            }
        }
    }
}

```

```

    public void cover(Square additional) {
        for (int i = additional.getY(); i < additional.getY() +
additional.getSize(); i++) {
            for (int j = additional.getX(); j <
additional.getX() + additional.getSize(); j++) {
                table.addLine(i, 1 << (table.size() - 1 - j));
            }
        }
    }
}

```

```

    public void uncover(Square additional) {
        for (int i = additional.getY(); i < additional.getY() +
additional.getSize(); i++) {
            for (int j = additional.getX(); j <
additional.getX() + additional.getSize(); j++) {
                table.deleteLine(i, 1 << (table.size() - 1 -
j));
            }
        }
    }
}

```

```

    public void setSize(int size) {
        this.size = size;
    }
}

```



```

}

public int getSize() {
    return size;
}

public TableCoverage getParent() {
    return parent;
}

public Square getPayload() {
    return lastAdded;
}

public LinkedList<Square> addSquares() {
    LinkedList<Square> tp = new LinkedList<>();

    Coords pos = table.findFirstEmpty();
    if (pos == null) return tp;

    int maxY = Math.min(table.size(), pos.getY() + size);
    int maxX = Math.min(table.size(), pos.getX() + size);
    int occupiedX = maxX, occupiedY = maxY, topSize;

    for (int k = pos.getY() + 1; k < maxY; k++) {
        if (table.isOccupied(new Coords(pos.getX(), k))) {
            occupiedY = k;
            break;
        }
    }
}

```

```

        for (int l = pos.getX() + 1; l < maxX; l++) {
            if (table.isOccupied(new Coords(l, pos.getY()))) {
                occupiedX = l;
                break;
            }
        }
        topSize = Math.min(occupiedX - pos.getX(), occupiedY -
pos.getY());
        if (table.isOccupied(new Coords(pos.getX() + topSize -
1, pos.getY() + topSize - 1))) topSize--;

        for (int k = 1; k <= topSize; k++) tp.push(new
Square(pos.getX(), pos.getY(), k));

        return tp;
    }

    public Square addSquare() {
        LinkedList<Square> added = addSquares();
        return added.isEmpty() ? null : added.pop();
    }

    @Override
    public String toString() {
        return table.toString();
    }
}

```

Приложение Г

Исходный код программы, файл Square.java

```
public class Square extends Coords {
    private int size;

    public Square(int x, int y, int size) {
        super(x, y);
        this.size = size;
    }

    public int getSize() {
        return size;
    }

    public void setSize(int size) {
        this.size = size;
    }

    @Override
    public String toString() {
        return getX() + " " + getY() + " " + size;
    }
}
```

Приложение Д

Исходный код программы, файл Coords.java

```
public class Coords {  
    private int x, y;  
  
    public Coords(int x, int y) {  
        this.x = x;  
        this.y = y;  
    }  
  
    public int getX() {  
        return x;  
    }  
  
    public int getY() {  
        return y;  
    }  
  
    public void setX(int x) {  
        this.x = x;  
    }  
  
    public void setY(int y) {  
        this.y = y;  
    }  
}
```

Приложение Е

Исходный код программы, файл MonoBitArray.java

```
public class MonoBitArray {
    private int [] data;

    public MonoBitArray(int size) {
        data = new int [size];
    }

    public Coords findFirstEmpty() {
        int full = Integer.MAX_VALUE % (1 << data.length);

        for (int i = 0; i < data.length; i++) {
            int partial = data[i] % (1 << data.length);
            if (partial < full) {
                int counter = 0, diff = full - partial;
                while (diff > 1) {
                    diff >>= 1;
                    counter++;
                }
                return new Coords(data.length - 1 - counter, i);
            }
        }

        return null;
    }

    public boolean isOccupied(Coords coords) {
        int k = data[coords.getY()];
        int l = (data.length - 1 - coords.getX());
        return (k >> l) % 2 != 0;
    }
}
```

```

    }

    public int getLine(int num) {
        return data[num];
    }

    public void addLine(int num, int line) {
        data[num] += line;
    }

    public void deleteLine(int num, int line) {
        data[num] -= line;
    }

    int size() {
        return data.length;
    }

    @Override
    public String toString() {
        StringBuilder output = new StringBuilder();
        for (int line : data) {
            output.append(String.format("%" + data.length + "s",
Integer.toBinaryString(line)).replace(' ', '0'));
            output.append('\n');
        }
        return output.toString();
    }
}

```