

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

Студентка гр. 8304

Николаева М. А.

Преподаватель

Размочаева Н. В.

Санкт-Петербург

2020

Цель работы.

Ознакомиться с алгоритмом поиска с возвратом. Научиться применять его для решения задач, а также оценивать временную сложность алгоритма.

Постановка задачи.

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

Входные данные:

Размер столешницы – одно целое число N ($2 \leq N \leq 20$).

Выходные данные:

Одно число K , задающее минимально 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 ($1 \leq x, y \leq N$) и длину стороны соответствующего обрезка (квадрата).

Описание алгоритма.

Для решения поставленной задачи рассмотрим задачу разбиения квадрата размера $N \times N$ на минимальное кол-во квадратов. И рассмотрим соответственно два случая: когда квадрат можно оптимально разбить и когда разбиение происходит посредством бэктрекинга с предварительной подготовкой для дополнительной оптимизации.

Алгоритм разбиения:

- 1) Если квадрат оптимальный (т.е. сторона квадрата делится на 2, 3 или 5), тогда алгоритм за $O(1)$ вычисляет разбиение и завершает работу.
- 2) Если квадрат неоптимальный (т.е. сторона не делится без остатка на 2, 3 или 5), запускается оптимизированный алгоритм разбиения:
 - а. Исходный квадрат делится на 3 полных квадрата и на неполный квадрат.
 - б. Для неполного квадрата запускается бэктрекинг. В результате находим минимальное разбиение.

Разбиение представлено на рисунке 1.



Рисунок 1 – Разбиение квадрата

Анализ алгоритма.

Для оптимальных квадратов алгоритм находит решение за $O(1)$ (т.е. сторона квадрата делится на 2, 3 или 5). Для неоптимальных – в следствие изначальной оптимизации площадь квадрата для бэктрекинга примерно равна $\frac{1}{4}$ исходной площади, что существенно сокращает его время работы. Так как алгоритм рекурсивный и содержит циклы, точную оценку времени работы бэктрекинга дать сложно. Наибольшее время работы будет для квадратов, у которых длина стороны является простым числом. Для квадрата с длиной стороны 37, алгоритм находит решение менее, чем за 3 секунды. Алгоритм использует $O(n^2)$ памяти.

Сложность алгоритма можно проанализировать по времени поиска решения для различных выходных данных. График, отображающий зависимость времени работы от длины стороны квадрата представлен на рисунке 2.

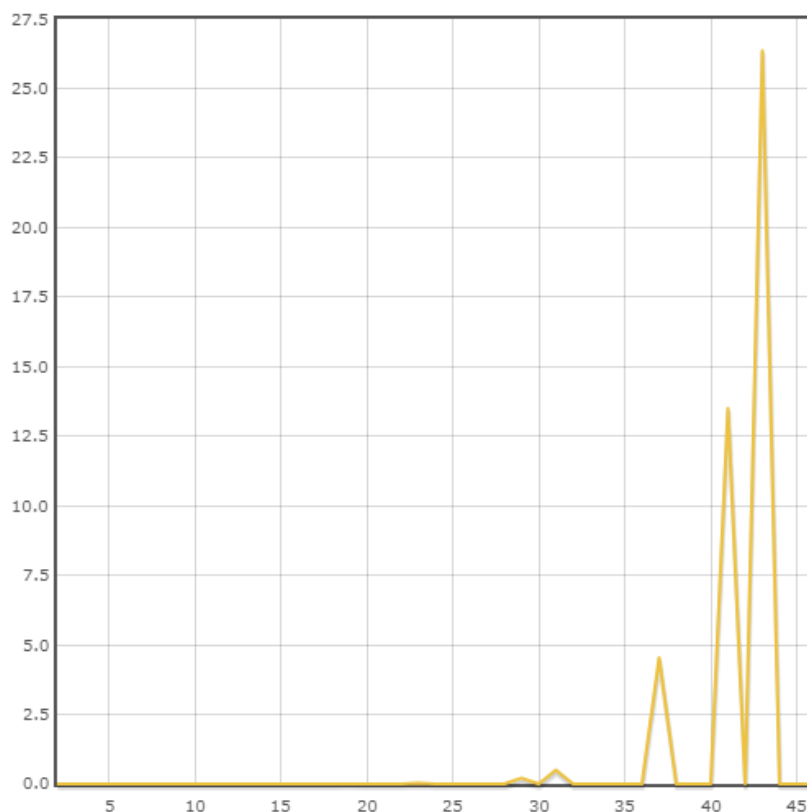


Рисунок 2 – Зависимость времени работы от длины стороны квадрата

Также были найдены значения времени работы для чисел не кратных 2, 3 и 5, так как в этих случаях алгоритм работает за константное время. Для этих значений построен график, а также найдено показательное уравнение регрессии и построен его график вместе с вычисленными значениями времени работы для этих чисел. Данные графики представлены на рисунках 3 и 4 соответственно.

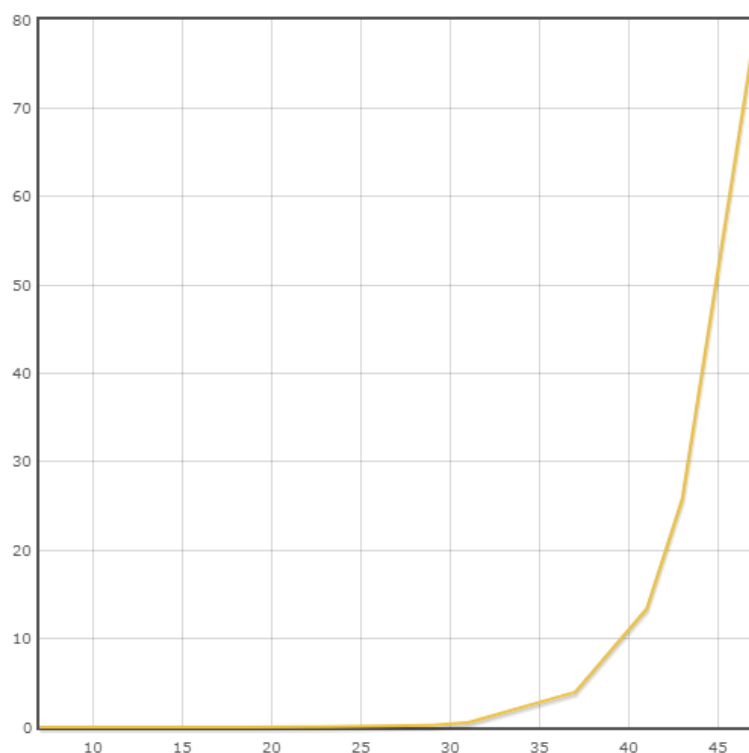


Рисунок 3 – Зависимость времени работы от длины стороны (для разбиения бэктрекигом)

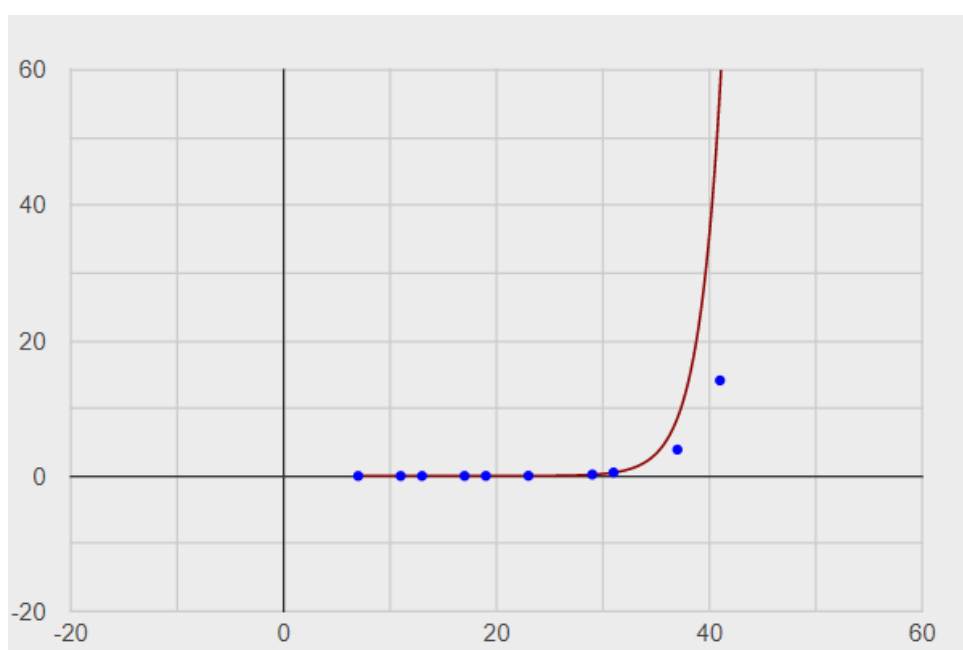


Рисунок 4 – График уравнения регрессии

Уравнение регрессии имеет вид:

$$\hat{y} = 0 \cdot 1.6082^x$$

Можно сделать вывод, что для таких чисел сложность работы алгоритма будет $O(1.6^N)$.

Описание функций и СД.

Для решения поставленной задачи был реализован класс TableTop.

Данный класс содержит метод вывода на экран результата решения (минимальное число квадратов, их координаты и длину стороны), методы, используемые для реализации бэктрекинга:

- `void paintSquare(int x, int y, int length);`

Функция раскраски квадрата. Принимает левый верхний угол квадрата и длину стороны. Возвращаемое значение отсутствует.

- `bool findAvaibleCoordinate(int x, int y, int& savedX, int& savedY);`

Функция поиска доступной координаты для квадрата. Принимает точку начала поиска и две ссылки для координаты. Возвращает true в случае удачного поиска.

- `bool canPaintSquare(int x, int y, int length);`

Функция проверки возможность закрасить квадрат. Принимает координаты левого верхнего угла квадрата и длину стороны. Возвращает значение типа bool.

- `void clearSquare(int x, int y, int len);`

Функция отчистки раскраски квадрата. Принимает левый верхний угол квадрата и длину его стороны. Возвращаемое значение отсутствует.

- `void checkMinSquare();`

Функция для проверки, является ли данное разбиение минимальным. Не принимает значения. Возвращаемое значение отсутствует. Промежуточные решения хранятся в двумерном массиве.

- `void writeRes();`

Функция для вывода результата работы на экран. Не принимает значения. Возвращаемое значение отсутствует.

- `void TableTop::backtracking(int length, int x, int y)`

Функция бэктрекинга принимает на вход длину квадрата, и координаты левого верхнего угла, возвращаемое значение отсутствует. Функция записывает промежуточные данные и результат в поля класса.

Спецификация программы.

Программа предназначена для нахождения минимального способа разбиения квадрата на меньшие квадраты. Программа написана на языке C++. Входными данными является число N (сторона квадрата), выходными – минимальное количество меньших квадратов и K строк, содержащие координаты левого верхнего угла и длину стороны соответствующего квадрата. Результат работы программы представлен в табл. 1.

Таблица 1 – Результаты работы программы

13
1 1 10
11 1 9
1 11 9
11 10 3
14 10 6
10 11 1
10 12 1
10 13 4
14 16 1
15 16 1
16 16 4
10 17 3
13 17 3

Выводы.

В ходе выполнения данной лабораторной работы был реализован алгоритм рекурсивного бэктрекинга, получены навыки решения задач с помощью алгоритма поиска с возвратом и дана оценка времени работы алгоритма.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp.

```
#include <iostream>
#include <Windows.h>

#include "Tabletop.h"

void finishedPartition(int n);

int main() {
    SetConsoleCP(1251);
    SetConsoleOutputCP(1251);

    int n = 0;
    std::cin >> n;

    if (n < 2 || n > 40) {
        std::cout << "\nОшибка ввода";
        exit(1);
    }

    if (n % 2 == 0 || n % 3 == 0 || n % 5 == 0) {
        finishedPartition(n);
    }
    else {
        Tabletop tabletop(n);
        tabletop.startBacktracking();
    }

    std::cout << std::endl;
    return 0;
}

void finishedPartition(int n) {
    if (n % 2 == 0) {
        std::cout << "4\n";
        std::cout << "1 1 " << n / 2 << std::endl;
        std::cout << 1 + n / 2 << " 1 " << n / 2 << std::endl;
        std::cout << "1 " << 1 + n / 2 << " " << n / 2 << std::endl;
        std::cout << 1 + n / 2 << " " << 1 + n / 2 << " " << n / 2 << std::endl;
    }
    else if (n % 3 == 0) {
        std::cout << "6\n";
        std::cout << "1 1 " << 2 * n / 3 << std::endl;
        std::cout << 1 + 2 * n / 3 << " 1 " << n / 3 << std::endl;
        std::cout << "1 " << 1 + 2 * n / 3 << " " << n / 3 << std::endl;
        std::cout << 1 + 2 * n / 3 << " " << 1 + n / 3 << " " << n / 3 << std::endl;
        std::cout << 1 + n / 3 << " " << 1 + 2 * n / 3 << " " << n / 3 << std::endl;
        std::cout << 1 + 2 * n / 3 << " " << 1 + 2 * n / 3 << " " << n / 3 << std::endl;
    }
    else if (n % 5 == 0) {
        std::cout << "8\n";
        std::cout << "1 1 " << 3 * n / 5 << std::endl;
        std::cout << 1 + 3 * n / 5 << " 1 " << 2 * n / 5 << std::endl;
        std::cout << "1 " << 1 + 3 * n / 5 << " " << 2 * n / 5 << std::endl;
    }
}
```



```

        std::cout << 1 + 3 * n / 5 << " " << 1 + 3 * n / 5 << " " << 2 * n / 5 << std::endl;
        std::cout << 1 + 2 * n / 5 << " " << 1 + 3 * n / 5 << " " << n / 5 << std::endl;
        std::cout << 1 + 2 * n / 5 << " " << 1 + 4 * n / 5 << " " << n / 5 << std::endl;
        std::cout << 1 + 3 * n / 5 << " " << 1 + 2 * n / 5 << " " << n / 5 << std::endl;
        std::cout << 1 + 4 * n / 5 << " " << 1 + 2 * n / 5 << " " << n / 5 << std::endl;
    }
}

```

tabletop.cpp

```

#include "Tabletop.h"

Tabletop::Tabletop(int length)
{
    this->length = length;

    buffSquare = new Square * [length];
    minSquare = new Square * [length];

    for (auto i = 0; i < length; ++i) {
        buffSquare[i] = new Square[length];
        minSquare[i] = new Square[length];
    }

    minSquareNum = length * length;
    squareCount = 0;
}

Tabletop::~Tabletop()
{
    for (auto i = 0; i < length; ++i) {
        delete[] buffSquare[i];
        delete[] minSquare[i];
    }

    delete[] buffSquare;
    delete[] minSquare;
}

void Tabletop::startBacktracking()
{
    paintSquare(0, 0, length / 2 + 1);
    paintSquare(length / 2 + 1, 0, length / 2);
    paintSquare(0, length / 2 + 1, length / 2);

    backtracking(length / 2, length / 2, length / 2);
    writeRes();
}

void Tabletop::paintSquare(int x, int y, int length)
{
    ++squareCount;
    for (auto i = y; i < y + length; ++i) {
        for (auto j = x; j < x + length; ++j) {
            buffSquare[i][j].size = length;
            buffSquare[i][j].number = squareCount;

```

```

        }
    }

}

void Tabletop::backtracking(int length, int x, int y)
{
    if (squareCount >= minSquareNum) {
        return;
    }

    int savedX;
    int savedY;

    if (findAvaibleCoordinate(x, y, savedX, savedY)) {
        for (auto len = length; len > 0; --len) {
            if (canPaintSquare(savedX, savedY, len)) {
                backtracking(length, x, savedY);
                clearSquare(savedX, savedY, len);
            }
        }
        return;
    }
    checkMinSquare();
}

bool Tabletop::findAvaibleCoordinate(int x, int y, int& savedX, int& savedY)
{
    for (auto i = y; i < length; ++i) {
        for (auto j = x; j < length; ++j) {
            if (buffSquare[i][j].size == 0) {
                savedX = j;
                savedY = i;
                return true;
            }
        }
    }
    return false;
}

bool Tabletop::canPaintSquare(int x, int y, int length)
{
    if (x + length > this->length || y + length > this->length) {
        return false;
    }

    for (auto i = y; i < y + length; ++i) {
        for (int j = x; j < x + length; ++j) {
            if (buffSquare[i][j].size != 0) {
                return false;
            }
        }
    }

    paintSquare(x, y, length);
}

```

```

        return true;
    }

void Tabletop::clearSquare(int x, int y, int len)
{
    for (auto i = y; i < y + len; ++i) {
        for (int j = x; j < x + len; ++j) {
            buffSquare[i][j].number = 0;
            buffSquare[i][j].size = 0;
        }
    }
    --squareCount;
}

void Tabletop::checkMinSquare()
{
    if (squareCount < minSquareNum) {
        minSquareNum = squareCount;

        for (auto i = 0; i < length; ++i) {
            for (int j = 0; j < length; ++j) {
                minSquare[i][j] = buffSquare[i][j];
            }
        }
    }
}

void Tabletop::writeRes()
{
    std::cout << minSquareNum << std::endl;

    for (auto i = 1; i <= minSquareNum; ++i) {
        for (auto y = 0; y < length; ++y) {

            int len = 0;
            for (auto x = 0; x < length; ++x) {
                if (minSquare[y][x].number == i) {
                    len = minSquare[y][x].size;
                    std::cout << x + 1 << " " << y + 1 << " " << len << std::endl;
                    break;
                }
            }

            if (len) {
                break;
            }
        }
    }
}

square.h
#pragma once

struct Square
{
    Square()

```

```

    {
        size = 0;
        number = 0;
    }

    int size;
    int number;
};

```

tabletop.h

```

#pragma once
#include "Square.h"
#include <iostream>

class Tabletop
{
public:
    Tabletop(int length);
    ~Tabletop();
    void startBacktracking();

private:
    void paintSquare(int x, int y, int length);
    void backtracking(int length, int x, int y);
    bool findAvaibleCoordinate(int x, int y, int& savedX, int& savedY);
    bool canPaintSquare(int x, int y, int length);
    void clearSquare(int x, int y, int len);
    void checkMinSquare();
    void writeRes();

private:
    Square** buffSquare;
    Square** minSquare;
    int length;
    int minSquareNum;
    int squareCount;
};

```

ПРИЛОЖЕНИЕ Б.
UML – диаграмма программы.

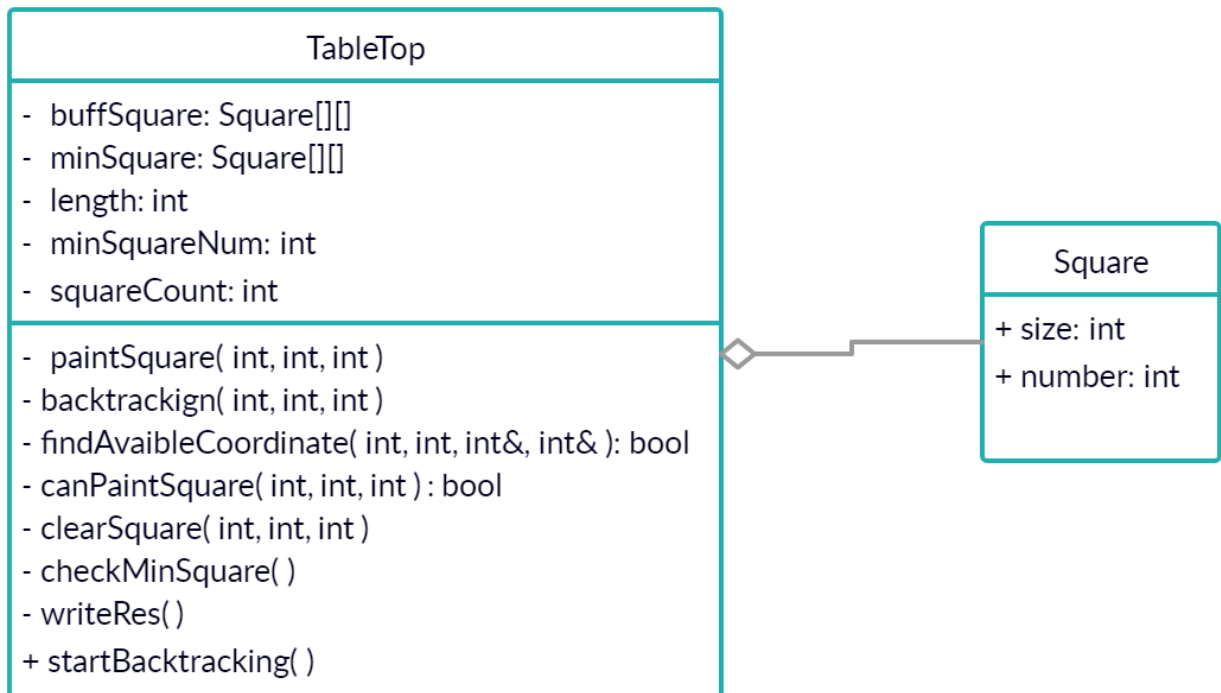


Рисунок 5 – UML диаграмма для иллюстрации реализованного класса