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. 8304 _____ Матросов Д. В.

Преподаватель _____ Размочаева Н.В.

Санкт-Петербург
2020

Цель работы.

Ознакомиться с алгоритмом поиска с возвратом, научиться оценивать временную сложность алгоритма и применять его для решения задач.

Постановка задачи.

Вариант 1и. Итеративный бэктрекинг. Поиск решения за разумное время (меньше минуты) для $2 \leq N \leq 30$.

Входные данные:

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

Выходные данные:

Одно число 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).

Анализ алгоритма.

Для квадратов, сторона которых не является простым числом алгоритм работает примерно за одно и то же время, что и для квадрата со стороной равной минимальному простому делителю числа. Сложность алгоритма по времени возрастает по экспоненте. Сложность по памяти $O(N^2)$

Описание СД.

1. Класс Square задает квадрат на столе по его координатам верхнего левого угла и длины стороны.
2. Класс Table определяет стол, размер которого задает пользователь.

Описание функций и методов.

1. Метод Table& shareTable() отвечает за разбиение стола на квадраты.
2. Метод Table& rewrite(Square p) отвечает за вписывание заданного квадрата на столешницу.
3. Метод Table& rewriteShare() отвечает за копирование «рабочей» столешницы в «итоговую» столешницу.
4. Метод Table& remove(Square point) отвечает за удаление квадрата с «рабочей» столешницы.
5. Метод Table& printTable() отвечает за печать столешницы на экран. (не используется)
6. Метод void print_result() отвечает за вывод результата работы алгоритма.
7. Метод bool can_insert(int x, int y, int lenght) проверяет, можно ли заданный квадрат положить на столешницу.
8. Метод bool chek() проверяет столешницу на заполненность.
9. Метод size_t primal_size(size_t size) проверяет заданный пользователем размер на простоту. Если он является простым числом – задача сводится к частному случаю, иначе к частному.

Спецификация программы.

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.

Выводы.

В ходе лабораторной работы был разработан алгоритм разбивающий столешницу заданного размера на минимальное число квадратов.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <ctime>
#include <vector>
#include <math.h>
using std::vector;

class Square
{
public:
    size_t x;
    size_t y;
    size_t length;

    Square();
    Square(const Square& copy);
    ~Square();

    void print();
    void print(size_t mlp);
};

Square::Square() {
    x = 0;
    y = 0;
    length = 0;
}

Square::Square(const Square& copy) {
    x = copy.x;
    y = copy.y;
    length = copy.length;
}

Square::~Square() {

}

void Square::print() {
    std::cout << x << " " << y << " " << length <<
std::endl;
}

void Square::print(size_t mlp) {
    std::cout << x * mlp << " " << y * mlp << " " <<
length * mlp << std::endl;
}
```

```
class Table {
public:
    Table(size_t size_);
    Table(Table& copy);
    ~Table();

    Table& operator=(const Table& copy);

    Table& shareTable();

    Table& rewrite(Square p);
    Table& rewriteShare();
    Table& remove(Square point);
    Table& printTable();

    void print_result();

    bool can_insert(int x, int y, int lenght);
    bool chek();

    size_t primal_size(size_t size);

private:
    int** table;
    int** result_share;
    int mlp;

    size_t size;
    size_t count;
    size_t result_count;
    vector <Square> current_share;
    vector <Square> result;
};



Table::Table(size_t size_) {
    size = primal_size(size_);
    mlp = size_ / size;
    table = new int* [size];
    result_share = new int* [size];
    for (int i = 0; i < size; i++)
    {
        table[i] = new int[size];
        result_share[i] = new int[size];
        for (int j = 0; j < size; j++)
        {
            table[i][j] = 0;
```

```
            result_share[i][j] = 0;
        }
    }
    count = 0;
    result_count = size*size;
}

Table::Table(Table& copy) {
    size = copy.size;
    table = new int* [size];
    result_share = new int* [size];
    mlp = copy.mlp;
    for (int i = 0; i < size; i++) {
        table[i] = new int[size];
        result_share[i] = new int [size];
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            table[i][j] = table[i][j];
            result_share[i][j] = copy.result_share[i][j];
        }
    }
    count = copy.count;
    result_count = copy.result_count;
}

Table::~Table() {
    for (int i = 0; i < size; i++) {
        delete table[i];
        delete result_share[i];
    }
    delete table;
    delete result_share;
}

Table& Table::operator=(const Table& copy) {
    size = copy.size;
    table = new int* [size];
    result_share = new int* [size];
    for (int i = 0; i < size; i++) {
        table[i] = new int[size];
        result_share[i] = new int[size];
    }
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            table[i][j] = table[i][j];
            result_share[i][j] = copy.result_share[i][j];
        }
    }
    count = copy.count;
    result_count = copy.result_count;
```

```
    return *this;
}

Table& Table::rewrite(Square p) {
    for (int i = p.y; i < p.y + p.length; i++)
    {
        for (int j = p.x; j < p.x + p.length; j++)
        {
            table[i][j] = count + 1;
        }
    }
    count++;
    return *this;
}

Table& Table::rewriteShare() {
    for (int i = 0; i < size; i++)
    {
        for (int j = 0; j < size; j++)
        {
            result_share[i][j] = table[i][j];
        }
    }
    return *this;
}

Table& Table::printTable() {
    for (size_t i = 0; i < size; i++) {
        for (size_t j = 0; j < size; j++) {
            std::cout << table[i][j] << " ";
        }
        std::cout << std::endl;
    }
    return *this;
}


bool Table::chek() {
    size_t count = 0;
    for (size_t i = 0; i < size; i++) {
        for (size_t j = 0; j < size; j++) {
            if (table[i][j] == 0) return false;
        }
    }
    return true;
}

Table& Table::remove(Square point) {
    for (int i = point.y; i < point.y + point.length;
i++)
```

```
    {
        for (int j = point.x; j < point.x + point.length;
j++)
        {
            table[i][j] = 0;
        }
    }
    count--;
    return *this;
}

bool Table::can_insert(int x, int y, int lenght) {
    if (x >= size || y >= size || x + lenght > size || y
+ lenght > size)
        return false;
    for (int i = y; i < y + lenght; i++)
    {
        for (int j = x; j < x + lenght; j++)
        {
            if (table[i][j] != 0)
            {
                return false;
            }
        }
    }
    return true;
}

size_t Table::primal_size(size_t size_){
    for (size_t i = 2; i <= sqrt(size_); i++)
    {
        if (size_ % i == 0)
            return i;
    }
    return size_;
}

Table& Table::shareTable()
{

    Square point;
    point.length = size / 2 + size % 2;
    current_share.push_back(point);
    rewrite(point);

    point.length = size / 2;
    point.x = size / 2 + size % 2;
    current_share.push_back(point);
    rewrite(point);

    point.x = 0;
```

```
    point.y = size / 2 + size % 2;
    current_share.push_back(point);
    rewrite(point);

    do
    {
        while (count < result_count && !chek())
        {
            for (size_t i = 0; i < size; i++)
            {
                for (size_t j = 0; j < size; j++)
                {
                    if (table[i][j] == 0)
                    {
                        for (int len = size - 1; len >
0; len--)
                        {
                            if (can_insert(j, i, len))
                            {
                                point.x = j;
                                point.y = i;
                                point.length = len;
                                break;
                            }

                        }
                        rewrite(point);
                        current_share.push_back(point);
                    }
                }
            }
        }

        if (result_count > count || result_count == 4)
        {
            result_count = count;
            rewriteShare();
            result = current_share;
        }
        while (!current_share.empty() && cur-
rent_share[current_share.size() - 1].length == 1)
        {
            remove(current_share[current_share.size() -
1]);
            current_share.pop_back();
        }
        if (!(current_share.empty()))
        {
            point = current_share[current_share.size() -
1];
            current_share.pop_back();
```

```
                remove(point);
                point.length -= 1;
                rewrite(point);
                current_share.push_back(point);
            }


    } while (count < result_count * 3 && !(cur-
rent_share.empty()));

    return *this;
}

void Table::print_result()
{
    std::cout << result_count <<  std::endl;
    for(int i = 0; i < result.size(); i++)
    {
        result[i].print(mlp);
    }
}

int main() {
    size_t size;

    std::cout << "Please enter size of square size: ";
    std::cin >> size;

    Table t(size);

    t.shareTable().print_result();

    return 0;
}
```