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 _____ Масалыкин Д.Р.

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 $2 \le N \le 30$.

Входные данные:

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

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 ($1 \le x, y \le N$) и длину стороны соответствующего обрезка (квадрата).

Описание алгоритма.

Алгоритм разбиения:

Сначала проверяется является ли длина стороны простым числом. Если нет – задача сводится к частному случаю. Если число четное разбивается на 4 квадрата со стороной N/4, если делится на 3 – на 6 квадратов, если делится на 5 – на 8 квадратов и т.д. Координаты и длина в данном случае пропорциональны. Если число не попадает ни в одну из вышеперечисленных групп, то при помощи бэктрекинга(полного перебора) ищется наилучшее разбиение.

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 функций и СД.

Для решения задачи был реализован класс Matrix.

Класс содержит методы вывода на экран промежуточных решений, минимального числа квадратов, результата решения.

Промежуточные решения хранятся в двумерном массиве.

Метод бэктрекинга:

void backtracking()

Ничего не принимает т.к. использует поля класса, возвращаемое значение отсутствует. Функция записывает промежуточные данные и результат в поля класса.

void insert_square(Point point)
Метод добавляет маленький квадрат на столешницу.

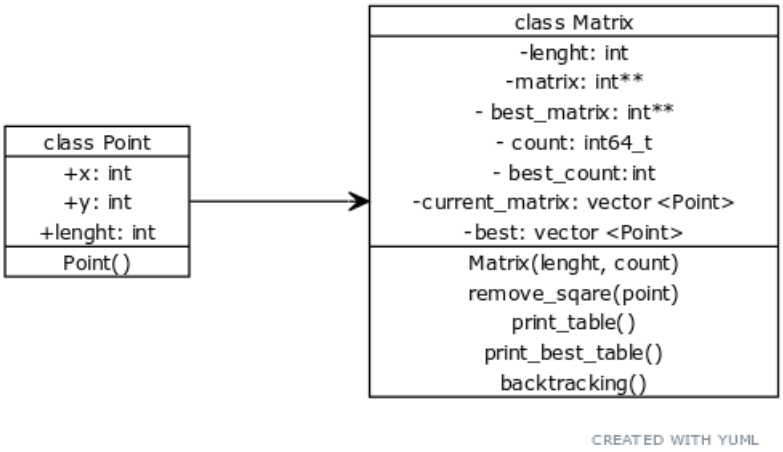
void remove_square(Point point)
Метод удаляет маленький квадрат со столешницы.

bool is_filled()
Метод проверяет закрашена ли вся матрица.

bool is_possible(int x, int y, int m)
Метод проверки на возможность закрасить квадрат.

Метод для вывода информации на экран
void print_result(int multiply)

UML-диаграмма



Тестирование.

Входные данные	Выходные данные	Ожидаемый результат
2	4	4
3	6	6
10	4	4
11	11	11
25	8	8
29	14	14

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.

Выводы.

В ходе выполнения лабораторной работы был реализован алгоритм итеративного бэктрекинга, дана оценка времени работы алгоритма, а также были получены навыки решения задач с помощью поиска с возвратом.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

main.cpp.

```
#include <cmath>
#include <iostream>
#include <vector>
#include <ctime>

using namespace std;

class Point
{
public:
    int x;
    int y;
    int length;
    Point()
    {
        x = 0;
        y = 0;
        length = 0;
    }
};

class Matrix
{
private:
    int length;
    int** matrix;
    int** best_matrix;
    int64_t count;
    int best_count;
    vector <Point> current_matrix;
    vector <Point> best;
    unsigned int count_of_operations;
public:
                    Matrix(int     length,     int     count)     :length(length),     count(0),
best_count(length*length),count_of_operations(0)
    {
        matrix = new int* [length];
        best_matrix = new int* [length];
        for (int i = 0; i < length; i++)
        {
            matrix[i] = new int[length];
            best_matrix[i] = new int[length];
            for (int j = 0; j < length; j++)
            {
                matrix[i][j] = 0;
                best_matrix[i][j] = 0;
            }
        }
    }

    void insert_square(Point point)
    {
        for (int i = point.y; i < point.y + point.length; i++)
```

```
    {
        for (int j = point.x; j < point.x + point.length; j++)
        {
            matrix[i][j] = count + 1;
        }
    }
    //std::cout<<count+1<<"\n";
    count++;
}

void remove_square(Point point)
{
    for (int i = point.y; i < point.y + point.length; i++)
    {
        for (int j = point.x; j < point.x + point.length; j++)
        {
            matrix[i][j] = 0;
        }
    }
    count--;
}

void print_table()
{
    std::cout << "current table" << "\n";
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            std::cout << matrix[i][j] << " ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}

void print_best_table()
{
    std::cout << "current best table!" << "\n";
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            std::cout << best_matrix[i][j] << " ";
        }
        std::cout << "\n";
    }
    std::cout << "\n";
}

void copy_square()
{
    for (int i = 0; i < length; i++)
    {
        for (int j = 0; j < length; j++)
        {
            best_matrix[i][j] = matrix[i][j];
        }
    }
```

```
}

bool is_filled()
{
    for (int i = length - 1; i >= 0; --i)
        for (int j = length - 1; j >= 0; --j)
            if (matrix[i][j] == 0)
                return false;
    return true;
}

bool is_possible(int x, int y, int m)
{
    if (x >= length || y >= length)
        return false;
    if (x + m > length || y + m > length)
    {
        return false;
    }
    for (int i = y; i < y + m; i++)
    {
        for (int j = x; j < x + m; j++)
        {
            if (matrix[i][j] != 0)
            {
                return false;
            }
        }
    }
    return true;
}

void backtracking()
{
    Point point;
    point.x = 0;
    point.y = 0;
    point.length = ceil(length / 2);
    current_matrix.push_back(point);
    insert_square(point);

    point.length = length / 2;
    point.x = ceil(length / 2);
    current_matrix.push_back(point);
    insert_square(point);
    point.x = 0;
    point.y = ceil(length / 2);
    current_matrix.push_back(point);
    insert_square(point);
    print_table();
    do
    {
        while (count < best_count && !is_filled() )
        {
            for (int y = 0; y < length; y++)
            {
                for (int x = 0; x < length; x++)
                {
                    if (matrix[y][x] == 0)
```

```
                    {
                        for (int m = length - 1; m > 0; m--)
                        {
                            count_of_operations++;
                            if (is_possible(x, y, m))
                            {
                                point.x = x;
                                point.y = y;
                                point.length = m;
                                break;
                            }

                        }
                        insert_square(point);
                        current_matrix.push_back(point);
                        //print_table();
                    }
                }
            }
        }

        if (best_count > count || best_count == 4)
        {
            best_count = count;
            copy_square();
            print_best_table();//вывод промежуточного лучшего результата
            best = current_matrix;
        }
         while (!current_matrix.empty() && current_matrix[current_matrix.size() - 1].length ==
1)//удаление квадратов со стороной 1
        {
            remove_square(current_matrix[current_matrix.size() - 1]);
            current_matrix.pop_back();
        }
        if (!(current_matrix.empty()))//уменьшение стороны квадрата на 1
        {
            point = current_matrix[current_matrix.size() - 1];
            current_matrix.pop_back();
            remove_square(point);
            point.length -= 1;
            insert_square(point);
            current_matrix.push_back(point);
        }


    } while (count < best_count * 3 && !(current_matrix.empty()));
}

void print_result(int multiply)//вывод результата работы программы
{
    Point point;
    std::cout<< "Minimum number of squares: " << best_count << "\n";
    std::cout<< "Count of operations: " << count_of_operations << "\n";
    while (!(best.empty()))
    {
        point = best[best.size() - 1];
        best.pop_back();
            std::cout << point.x * multiply + 1<< " " << point.y * multiply + 1 << " " <<
point.length*multiply;
```

```
            if(!(best.empty()))
                std::cout << std::endl;
        }
    }
};


int min_primal_size(int size)
{
    int primal_size = size;
    for (int i = 2; i <= sqrt(size); i++)
    {
        if (size % i == 0)
            return i;
    }
    return  primal_size;
}

int main()
{
    int square_side;
    std::cout << "Please, input length of sub_matrix's side\n";
    std::cin >> square_side;
    int primal_size = min_primal_size(square_side);
    Matrix sub_matrix(primal_size, 0);
    std::cout<<primal_size<<std::endl;
    int multiply = square_side/primal_size;
    srand(time(0));
    sub_matrix.backtracking();
    sub_matrix.print_result(multiply);
    cout << std::endl <<"runtime = " << clock() / 1000.0 << endl; // время работы программы
}
```