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. 1384 _____ Усачева Д.В.

Преподаватель _____ Шевелева А. М.

Санкт-Петербург
2023

Цель работы.

Изучить принцип работы алгоритмов поиска кратчайшего пути. Написать две программы, которая решает задачу построения пути в ориентированном графе при помощи жадного алгоритма и алгоритма A*.

Задание.

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 (см. Рисунок 1 - Пример столешницы 7×7).

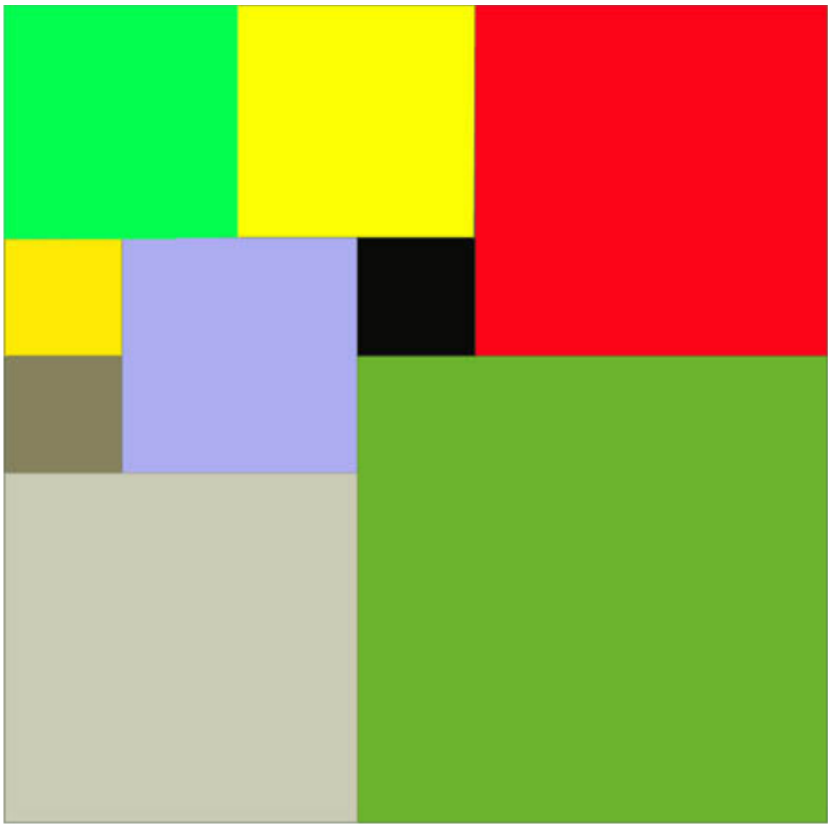

Рисунок 1 - Пример столешницы 7×7

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.

Входные данные

Размер столешницы - одно целое число N (2≤N≤20).

Выходные данные

Одно число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N*N. Далее должны идти K строк, каждая из которых должна содержать три целых числа x, y и w, задающие координаты левого верхнего угла и длину стороны соответствующего обрезка(квадрата), соответствующего обрезка(квадрата).

Пример входных данных

7

Соответствующие выходные данные

9

1 1 2

1 3 2

3 1 1

4 1 1

3 2 2

5 1 3

4 4 4

1 5 3

3 4 1

Выполнение работы.

Для решения поставленной задачи была реализована программа на языке программирования C++.

Определена структура Square, в которой хранятся координаты квадрата и его размер.

Функция find_coefficient необходима для нахождения наименьших простых делителей числа (от 2 до 5, так как задача решается для квадратов

размера меньшего или равного 40). Эта оптимизация значительно упрощает решение задачи, благодаря тому, что расположение квадратов одинаково для самого числа и его наименьшего множителя.

Функция original_size возвращает квадрат к исходному размеру, если была использована оптимизация, описанная выше.

Функция is_inside определяет, находится ли точка с координатами x y внутри какого-либо из квадратов.

Функция print выводит результат в заданном формате.

Функция assembly рекурсивно собирает большой квадрат из множества квадратов меньшего размера (реализация поиска с возвратом). Она принимает следующие аргументы:

- squares: вектор маленьких квадратов, на основе которых собирается большой;
- S: суммарная площадь маленьких квадратов;
- xmin, ymin: минимальные координаты по x и y, для ускорения поиска следующей точки;
- k: размер большого квадрата;
- record: текущий рекорд - минимальное число квадратов, необходимых для сборки большого квадрата;
- record_list: вектор квадратов, соответствующий текущему рекорду.

Функция перебирает каждую точку в большом квадрате, проверяет, свободна ли она, и если да, то перебирает возможные размеры квадратов, которые можно разместить в этой точке (исходя из уже имеющихся квадратов). Затем она добавляет этот квадрат в вектор маленьких квадратов и рекурсивно вызывает сама себя для поиска следующего квадрата. Если суммарная площадь маленьких квадратов совпадает с площадью большого квадрата, рекорд обновляется, и текущие квадраты сохраняются в record_list. Если стол не является заполненным и количество квадратов в нём равняется количеству

квадратов в прошлом решении, уменьшенном на единицу, то функция прерывается и возвращает результат.

В функции main сначала считывается входное число N, затем находится наименьший простой делитель coefficient числа N. Затем создаются три 3 квадрата. Таким образом, нужно заполнять не квадрат со стороной n, а квадрат со стороной n/2, что значительно уменьшает количество итераций. Если coefficient не равен N, то размер квадратов в record_list изменяется обратно на исходный размер. Наконец, выводится record и record_list в заданном формате.

Тестирование.

Результаты тестирования представлены в таблице 1.

Таблица 1 — Результаты тестирования

№	Входные данные	Выходные данные	Комментарии
1.	33	33 6 1 1 22 1 23 11 23 1 11 12 23 11 23 12 11 23 23 11	Правильный ответ.
2.	3	3 6 1 1 2 1 3 1 3 1 1 2 3 1 3 2 1 3 3 1	Правильный ответ. Заметим, что предыдущий ответ идентичен этому умноженному на 11. Значит представленная в решении оптимизация верна.

Выводы.

Был изучен поиск с возвратом. Реализована программа собирающая исходный квадрат из множества меньших квадратов, без перекрытия друг друга и без выхода за пределы исходного квадрата. Алгоритм позволил найти все решения поставленной задачи, если они существуют.

Для ускорения вычисления были добавлены две оптимизации, организованные таким образом, чтобы как можно раньше выявлять подходящие варианты. Это позволило значительно уменьшить время нахождения решения.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл: lb1.cpp

```
#include <iostream>
#include <vector>
#include <cstdio>
#include <algorithm>

// структура для хранения квадратов
// x и y координаты левого верхнего угла квадрата
// size размер квадрата
struct Square {
    int x;
    int y;
    int size;
} typedef Square;

// функция для поиска наименьших простых делителей числа (от 2 до 5)
int find_coefficient(int N) {
    for (int i = 2; i < 6; i++) {
        if (N % i == 0)
            return i;
    }
    return N;
}

// функция для возвращение исходного размера квадрата
void original_size(std::vector<Square> &squares, int coefficient) {
    for (int i = 0; i < squares.size(); i++) {
        squares[i].x *= coefficient;
        squares[i].y *= coefficient;
        squares[i].size *= coefficient;
    }
}

// функция для проверки находится ли точка внутри квадрата или нет
bool is_inside(std::vector<Square> squares, int x, int y) {
    for (int i = 0; i < squares.size(); i++) {
        if (x >= squares[i].x && x < squares[i].x + squares[i].size &&
y >= squares[i].y &&
            y < squares[i].y + squares[i].size)
            return true;
    }
    return false;
}

// функция, обеспечивающая корректный вывод
void print(std::vector<Square> record_list) {
    for (int i = 0; i < record_list.size(); i++) {
        std::cout << record_list[i].x + 1 << " " << record_list[i].y + 1
<< " " << record_list[i].size << "\n";
    }
}

// фунция для сборки большого квадрата из множества квадратов меньшего
размера
void assembly(std::vector<Square> &squares, int S, int xmin, int ymin,
int &k, int &record,
              std::vector<Square> &record_list) {
    for (int x = xmin; x < k; x++) {
```

```
            for (int y = ymin; y < k; y++) {
                if (!(is_inside(squares, x, y))) {
                    int length = std::min(k - x, k - y);
                    for (auto each: squares) {
                        if (each.x + each.size > x && each.y > y) {
                            length = std::min(length, each.y - y);
                        }
                    }
                    for (int i = length; i > 0; i--) {
                        Square s;
                        s.x = x;
                        s.y = y;
                        s.size = i;
                        std::vector<Square>    tmp    =    {squares.begin(),
squares.end()};
                        tmp.push_back(s);
                        if (S + s.size * s.size == k * k) {
                            if (tmp.size() < record) {
                                record = tmp.size();
                                record_list = {tmp.begin(), tmp.end()};
                            }
                        } else {
                            if (tmp.size() < record + 1)
                                assembly(tmp, S + s.size * s.size, x, y + i,
k, record, record_list);
                            else {
                                return;
                            }
                        }
                    }
                    return;
                }
            }
            ymin = int(k / 2);
        }
    }

    int main() {
        int N;
        std::cin >> N;
        int coefficient = find_coefficient(N);
        int record = 2 * N + 1;
        std::vector<Square> record_list;
        std::vector<Square> squares;
        Square tmp = {0, 0, int((coefficient + 1) / 2)};
        squares.push_back(tmp);
        tmp = {0, int((coefficient + 1) / 2), int(coefficient / 2)};
        squares.push_back(tmp);
        tmp = {int((coefficient + 1) / 2), 0, int(coefficient / 2)};
        squares.push_back(tmp);
        assembly(squares, int((coefficient + 1) / 2) * int((coefficient + 1)
/ 2) +
                          2 * (int(coefficient / 2) * int(coefficient / 2)),
int(coefficient / 2),
                 int((coefficient   +   1)   /   2),   coefficient,   record,
record_list);
        if (coefficient != N)
            original_size(record_list, int(N / coefficient));
        printf("%d\n", record);
        print(record_list);
    }
```