**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. 3341 |  | Ягудин Д.Р. |
| Преподаватель |  | Фирсов М.А. |

Санкт-Петербург 2025

# Цель работы.

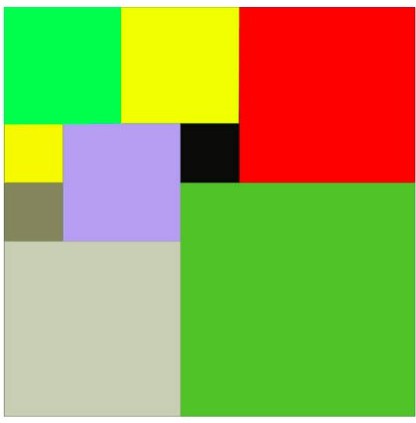
Применить на практике алгоритм поиска с возвратом для заполнения исходного квадрата минимальным количеством квадратов.

# 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.



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*. Далее должны идти *K* строк, каждая из которых должна содержать три целых числа *x*,*y* и *w*, задающие координаты левого верхнего угла (1≤*x*,*y*≤*N*) и длину стороны 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

# Вариант 2и.

Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

# Выполнение работы:

В программе использован класс:

Class Field – класс хранящий в себе двумерный вектор представляющий собой столешницу, длина одной стороны, стек в котором находятся координаты и размер расположенных квадратов, а так же размер стека.

Методы класса:

void print() – выводит столешницу в терминал.

bool place\_square(int x, int y, int z) – размещает новый квадрат, аргументы x и y отвечают за положение нового квадрата, z за длину его стороны.

bool can\_place(int x, int y, int z) – проверяет возможно ли разместить квадрат, аргументы x и y отвечают за положение проверяемого квадрата, z за длину его стороны.

tuple<int, int, int> get\_back() – удаляет последний добавленный квадрат в стек.

void remove\_square(int x, int y, int z) – удаляет квадрат, аргументы x и y отвечают за положение удаляемого квадрата, z за длину его стороны.

tuple<int, int> is\_full() – проверяет заполнена ли столешница, если нет, то возвращает координаты первой незаполненной клетки.

vector<tuple<int, int, int>> get\_stack() – возвращает стек квадратов.

В программе были реализованы следующие функции:

Vector<tuple<int, int, int>> for\_2(int n) - функция для четных чисел (оптимизация 1), аргумент n отвечает за длину стороны заполняемой столешницы;

Vector<tuple<int, int, int>> for\_3(int n) - функция для рассмотрения чисел, кратных 3 (оптимизация 2) ), аргумент n отвечает за длину стороны заполняемой столешницы;

Vector <tuple<int, int, int>> backtracking(int n, bool flag) – функция итеративного бэктрекинга, где n размер столешницы, flag отвечает за отображение промежуточной информации во время выполнения программы.

Описание алгоритма работы программы:

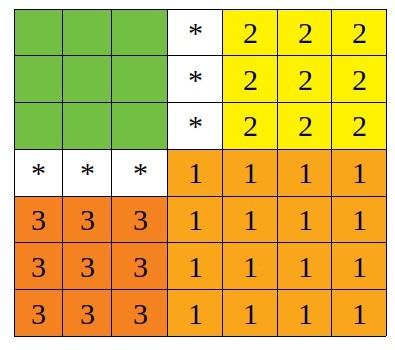
Программа на вход получает число n — сторона исходного квадрата, измеряем время до начала работы программы, результат записываем в переменную start. Заводим переменную result\_flag, в которой будем хранить минимальное число квадратов, изначально оценим результат как 3+2\*(n/2)+1 = 3+n+1, так как для простых чисел мы изначально фиксируем 3 квадрата (помечены как 1, 2 и 3), а остальное пространство заполняем как на рисунке 2 (пример для числа 7) — то есть 2 стороны по n/2 (помечены \* на рисунке) и остается еще 1 квадрат (зеленый)

Рисунок 2 — пример выбора начального значения для квадрата со стороной 7 Затем, в зависимости он значения n выполняем действия:

1. если число кратно 2 — четное, то обращаемся к функции for\_2
2. если число кратно 3, то обращаемся к for\_3
3. иначе (остались только простые числа (до n <=20)) — обращаемся к backtracking, в которой фиксируется 3 квадрата, а после перебираем варианты заполнения оставшегося пространства и выбирается оптимальный.

В программе были использованы следующие оптимизации:

1. Было замечено, что для чисел, кратных 2, минимальное количество квадратов равно 4. Для этого реализована функция for\_2.
2. А для чисел, кратных 3, минимальное количество квадратов равно 6. Для этого реализована функция for\_3.
3. Если число простое, то заметим, что для первых 3 трех располагаемых квадратов есть положения и размеры, оптимальные для решения.
4. Во время выполнения программы сохраняются оптимальные решения, что позволяет исключить рассмотрение вариантов, содержащих большее число квадратов, чем минимально возможное.

Анализ времени работы программы

Благодаря вышеописанным оптимизациям программа для чисел кратных 2 и 3 будет работать быстро, поэтому рассмотрим время работы программы для простых чисел:

|  |  |
| --- | --- |
| Длина исходного квадрата | Время работы |
| 5 | 0.000269 |
| 7 | 0.000694 |
| 11 | 0.009430 |
| 13 | 0.023886 |
| 17 | 0.161188 |
| 19 | 0.68288 |

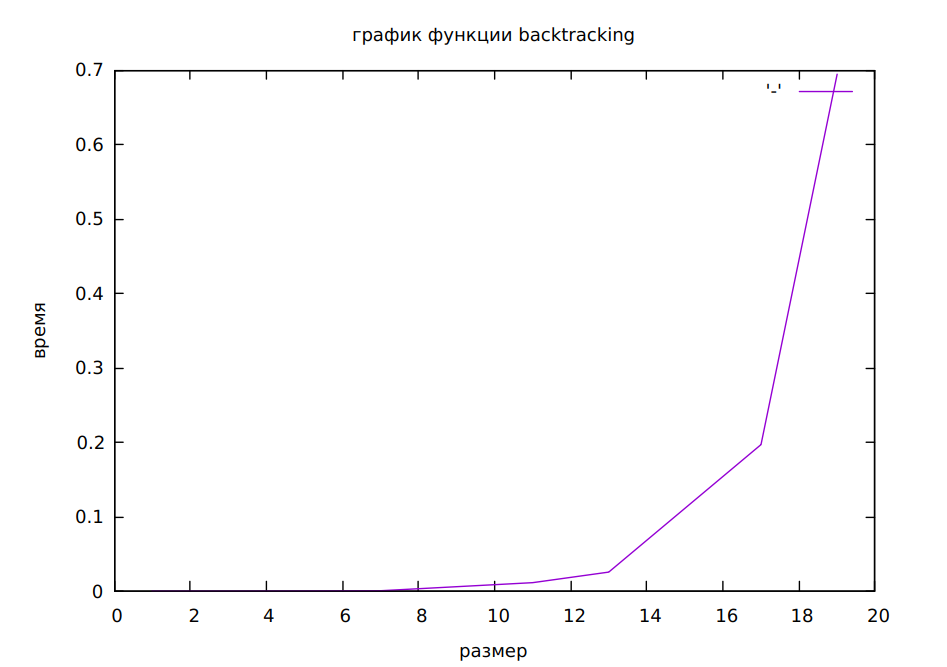
На рисунке 3 приведен график зависимости времени работы программы в секундах в зависимости от длины стороны исходного квадрата (n <= 20): 

Рисунок 3 — График зависимости работы программы от длины стороны квадрата

Проанализировав полученный график, можно сделать вывод, что сложность алгоритма О(n ^ (n ^ 2)).

# Тестирование

|  |  |
| --- | --- |
| Входные данные | Выходные данные |
| 5 | time: 0.000216  8  3 3 3  4 1 2  1 4 2  1 1 2  1 3 1  2 3 1  3 1 1  3 2 1 |
| 11 | time: 0.010684  11  6 6 6  7 1 5  1 7 5  1 1 4  1 5 2  3 5 2  5 1 2  5 3 2  5 5 1  5 6 1  6 5 1 |
| 8 | time: 1.3e-05  4  1 1 4  1 5 4  5 1 4  5 5 4 |
| 9 | time: 7e-06  6  1 1 3  1 4 3  1 7 3  4 1 3  7 1 3  4 4 6 |

# Вывод.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным количеством квадратов. Придумана оптимизация, для чисел, кратных 2 и 3.

# Приложение А Исходный код программы

Название файла: main.cpp

#include <iostream>

#include <vector>

#include <tuple>

#include <ctime>

#include <string>

#include<gnuplot-iostream.h>

using namespace std;

class Field {

private:

    vector<vector<int>> map;

    int size;

    vector<tuple<int, int, int>> stack;

    int stack\_size;

public:

    Field(int size) : size(size), stack\_size(0) {

        map.resize(size, vector<int>(size, 0));

    }

    void print() {//вывести поле в термниал

        for (const auto &row : map) {

            for (int cell : row) {

                cout << cell << " ";

            }

            cout << '\n';

        }

    }

    bool place\_square(int x, int y, int z) {//разместить новый квадрат

        if (can\_place(x, y, z)) {

            stack.emplace\_back(x, y, z);

            stack\_size++;

            for (int i = 0; i < z; ++i) {

                for (int j = 0; j < z; ++j) {

                    map[y + i][x + j] = 1;

                }

            }

            return true;

        }

        return false;

    }

    bool can\_place(int x, int y, int z) {//проверка на возможность разместить квадрат

        if (x + z > size || y + z > size) return false;

        for (int i = 0; i < z; ++i) {

            for (int j = 0; j < z; ++j) {

                if (map[y + i][x + j] != 0) return false;

            }

        }

        return true;

    }

    tuple<int, int, int> get\_back() {//удаляет последний добавленный квадрат

        auto tmp = stack.back();

        stack.pop\_back();

        remove\_square(get<0>(tmp), get<1>(tmp), get<2>(tmp));

        stack\_size--;

        return tmp;

    }

    void remove\_square(int x, int y, int z) {//удаляет область

        for (int i = 0; i < z; ++i) {

            for (int j = 0; j < z; ++j) {

                map[y + i][x + j] = 0;

            }

        }

    }

    tuple<int, int> is\_full() {//проверяет заполнен ли квадрат

        for (int x = 0; x < size; ++x) {

            for (int y = 0; y < size; ++y) {

                if (map[y][x] == 0) return {x, y};

            }

        }

        return {-1, -1};

    }

    vector<tuple<int, int, int>> get\_stack() { return stack; }

};

vector<tuple<int, int, int>> for\_2(int n, bool flag) { // оптимизация для квадратов со стороной четной 2

    if (flag == true) {

        cout << "using optimization for % 2\n";

    }

    vector<tuple<int, int, int>> answer = {

        {0, 0, n / 2}, {0, n / 2, n / 2}, {n / 2, 0, n / 2}, {n / 2, n / 2, n / 2}};

    return answer;

}

vector<tuple<int, int, int>> for\_3(int n, bool flag) {// оптимизация для квадратов со стороной четной 3

    if (flag == true) {

        cout << "using optimization for % 3\n";

    }

    vector<tuple<int, int, int>> answer = {

        {0, 0, n / 3}, {0, n / 3, n / 3}, {0, 2 \* (n / 3), n / 3},

        {n / 3, 0, n / 3}, {2 \* (n / 3), 0, n / 3}, {n / 3, n / 3, 2 \* (n / 3)}};

    return answer;

}

vector<tuple<int, int, int>> backtracking(int n, bool flag) {

    if (n % 2 == 0) {

        return for\_2(n, flag);

    } else if (n % 3 == 0) {

        return for\_3(n, flag);

    }else{

        Field field(n);

        if (flag == true) {

            cout << "using optimization for simple number\n";

        }

        //оптимизация для простых чисел

        field.place\_square(n / 2, n / 2, n / 2 + 1);

        field.place\_square(n - n / 2, 0, n / 2);

        field.place\_square(0, n - n / 2, n / 2);

        //создаем стек состояний столешниц

        vector<vector<tuple<int, int, int>>> stack\_of\_fields = {field.get\_stack()};

        //создаем вектор содержащий лучшее решение и флаг макссимального кол-во кусков столешницы

        vector<tuple<int, int, int>> result;

        int result\_flag = 3 + n + 1;

        while (!stack\_of\_fields.empty()) {

            //берем из стека последнее состояние столешницы

            Field main\_field(n);

            vector<tuple<int, int, int>> last\_stack = stack\_of\_fields.back();

            stack\_of\_fields.pop\_back();

            for (const auto &s : last\_stack) {

                main\_field.place\_square(get<0>(s), get<1>(s), get<2>(s));

            }

            if (flag == true) {

                cout << "main field\n";

                main\_field.print();

            }

            //проверяем заполнена ли столешница

            auto [x, y] = main\_field.is\_full();

            if (x == -1 && y == -1) {

                if (flag == true) {

                    cout << "save optimal solution\n";

                }

                result = main\_field.get\_stack();

                result\_flag = result.size();

            }

            if (x != -1 && y != -1 && flag == true) {

                cout << "empty place: " << x + 1 << " " << y + 1 << "\n";

            }

            //оптимизация пропуска неоптимальных решений

            if (main\_field.get\_stack().size() + 1 >= result\_flag) {

                if (flag == true){

                    cout << "skip, not optimal solution\n";

                    cout << "----------------------------------------\n";

                }

                continue;

            }

            for (int i = 1; i < n - 1; ++i) {

                if (main\_field.can\_place(x, y, i)) {

                    //создаем копию  основного поля

                    Field tmp\_field(n);

                    for (const auto &s : main\_field.get\_stack()) {

                        tmp\_field.place\_square(get<0>(s), get<1>(s), get<2>(s));

                    }

                    //помещаем на нее новый квадрат

                    tmp\_field.place\_square(x, y, i);

                    if (flag == true) {

                        cout << "place new square\n";

                        cout << "x: " << x + 1 << ", y: " << y + 1 << ", i: " << i << ", count squares:" << tmp\_field.get\_stack().size() << '\n';

                        tmp\_field.print();

                    }

                    //сохраняем его в стек

                    stack\_of\_fields.push\_back(tmp\_field.get\_stack());

                } else {

                    break;

                }

            }

            cout << "----------------------------------------\n";

        }

        return result;

    }

}

void test() {

    std:: vector<int> vectN = {3, 5, 6, 7, 9, 10, 11};

    std:: vector<int> expectedMinCountSquares = {6, 8, 4, 9, 6, 4, 11};

    for (int i = 0; i < vectN.size(); i++) {

        vector<tuple<int, int, int>> answer = backtracking(vectN[i], false);

        assert(answer.size() == expectedMinCountSquares[i]);

    }

    std:: cout << "Success!" << std:: endl;

}

void graphic() {

    Gnuplot gp;

    gp << "set title 'график функции backtracking'" << std::endl;

    gp << "set xlabel 'размер'" << std::endl;

    gp << "set ylabel 'время'" << std::endl;

    gp << "plot '-' with lines" << std::endl;

    vector<int> simple\_numbers = {1,2,3,5,7,11,13,17,19};

    for (int i = 0; i < simple\_numbers.size(); i++) {

        clock\_t start = clock();

        backtracking(simple\_numbers[i], false);

        clock\_t end = clock();

        gp << simple\_numbers[i] << " " << (double)(end - start) / CLOCKS\_PER\_SEC << std::endl;

    }

    gp << "e" << std::endl;

}

void check() {

    int n;

    cout << "input size of table\n";

    cin >> n;

    bool flag = false;

    cout << "vision?\n";

    string input;

    cin >> input;

    if (input == "yes") {

        flag = true;

    } else {

        flag = false;

    }

    clock\_t start = clock();

    vector<tuple<int, int, int>> answer = backtracking(n, flag);

    clock\_t end = clock();

    cout << "time: " << (end - start) / (double) CLOCKS\_PER\_SEC << '\n';

    cout << answer.size() << '\n';

    for (auto i : answer) {

        cout << get<0>(i) + 1 << " " << get<1>(i) + 1 << " " << get<2>(i) << '\n';

    }

}

int main() {

    int command;

    cout << "what do you wanna see? 1 - standart usage, 2 - graphic, 3 - test" << endl;

    cin >> command;

    if (command == 1) {

        check();

    } else if (command == 2) {

        graphic();

    } else {

        test();

    }

}