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. 0382 _____ Кривенцова Л.С.

Преподаватель _____ Шевская Н.В.

Санкт-Петербург
2022

Цель работы.

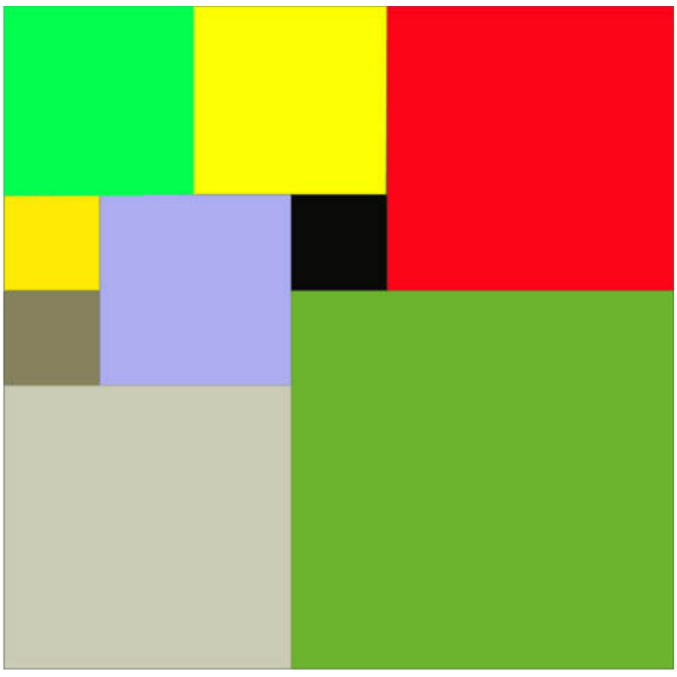
Изучить принцип работы бэктрекинга - алгоритма поиска с возвратом.

Основные теоретические положения.

Поиск с возвратом, бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше. Данный алгоритм позволяет найти все решения поставленной задачи, если они существуют. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

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 ($2 \le N \le 20$).

Выходные данные

Одно число KK, 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 ($1 \le x, y \le N$) 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

Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения от размера квадрата.

Выполнение работы.

Решение задачи реализовано с помощью алгоритма рекурсивного бэктрекинга.

Для каждого нового квадрата сохраняются значения его нижнего левого угла и длины стороны. Функция *Backtracking* находит самую верхнюю из точек, лежащих левее всех, которая при этом не лежит ни в каком из уже найденных квадратов. Для быстрого поиска этой точки, функции передается координата x, левее которой столешница уже заполнена и координата y, выше которой все уже заполнено. Ищется максимальный радиус квадрата, нижний левый угол которого лежит в найденной точке, и функция *Backtracking* вызывается для каждого такого возможного квадрата, радиус которого меньше или равен найденному. Потом суммарная площадь квадратов сравнивается с площадью столешницы: если значения равны, а найденное количество квадратов меньше, чем у уже найденного минимального решения, то количество квадратов записывается в минимум (лучшее – минимальное количество квадратов хранится в переменной *bestcount*). Функция *Backtracking* не вызывается, если длина массива на текущем шаге не меньше минимального решения, полученного на этот момент.

Решение (в квадратных досках) хранится в векторе *vector<board> bestarray* в виде набора квадратов, которые заполнены на текущий момент. Для каждого такого квадрата сохраняются значения его нижнего левого угла и длины стороны.

Для оптимизации в решении участвуют только квадраты с размером, кратным наибольшему делителю n. Также на столешнице сразу располагаются квадратные доски со стороной (n+1)/2 в точку {0, 0} и два квадрата размером n/2 в точки {0, (n+1)/2} и {(n+1)/2, 0}.

struct board {

int x;

int y;

int w;

}; - структура для хранения данных об одной квадратной доске.

bool Collision(std::vector<board>& one, int x, int y) - функция, проверяющая точку (x, y) на принадлежность её другим квадратам столешницы.

int summ; — суммарная площадь размещенных досок.

int boardlen; — длина *vector<board> bestarray.*

int minx, miny; — координаты, левее/выше которой столешница уже заполнена.

Исследование сложности.

Сложность по времени одного витка функции *Backtracking* O(n). Количество вызовов этой функции можно оценить сверху как n*n, так как для каждого нового вызова существует в худшем случае n вариантов поставить новый квадрат. Глубина рекурсии не превосходит n. Для каждого вызова функции хранится массив квадратов длины не больше n. Следовательно, сложность по памяти $O(n^2)$.

Тестирование.

Номер	Входные данные	Выходные данные
1	2	4 0 0 1 0 1 1 1 0 1 1 1 1
2	7	9 0 0 4 0 4 3 4 0 3 3 4 2 3 6 1 4 3 1 4 6 1 5 3 2 5 5 2
3	9	6 0 0 6 0 6 3 6 0 3 3 6 3 6 3 3 6 6 3
4	5	8 0 0 3 0 3 2 3 0 2 2 3 2 3 2 1 4 2 1 4 3 1 4 4 1

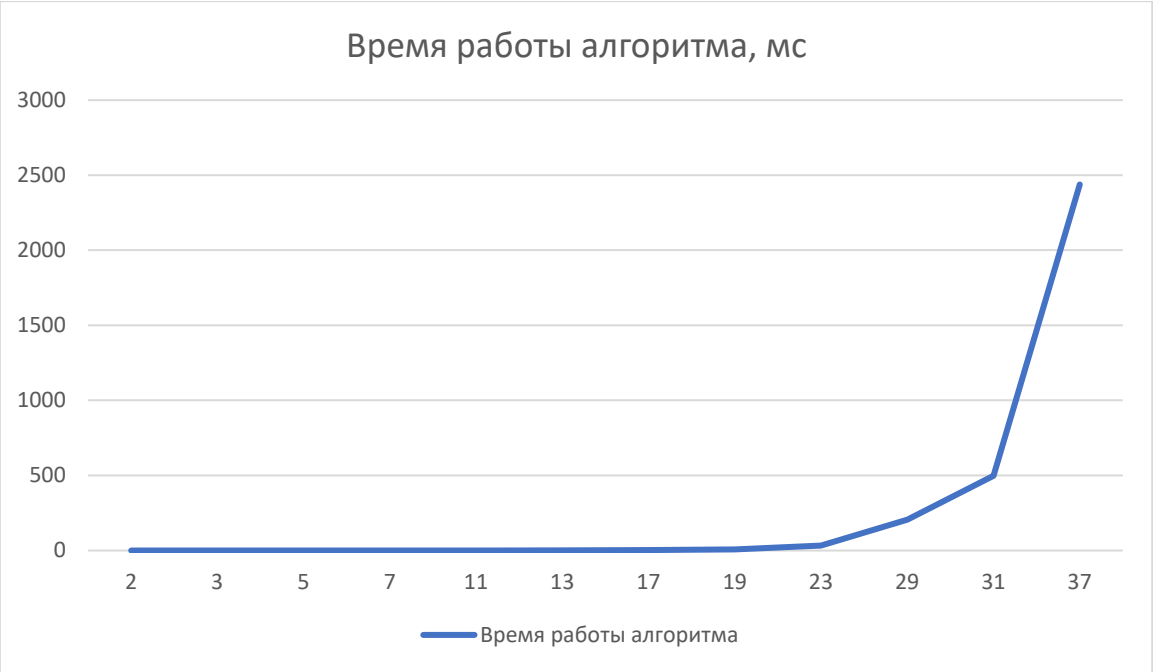
5	17	12 0 0 9 0 9 8 9 0 8 8 9 2 8 11 4 8 15 2 9 8 1 10 8 3 10 15 2 12 11 1 12 12 5 13 8 4

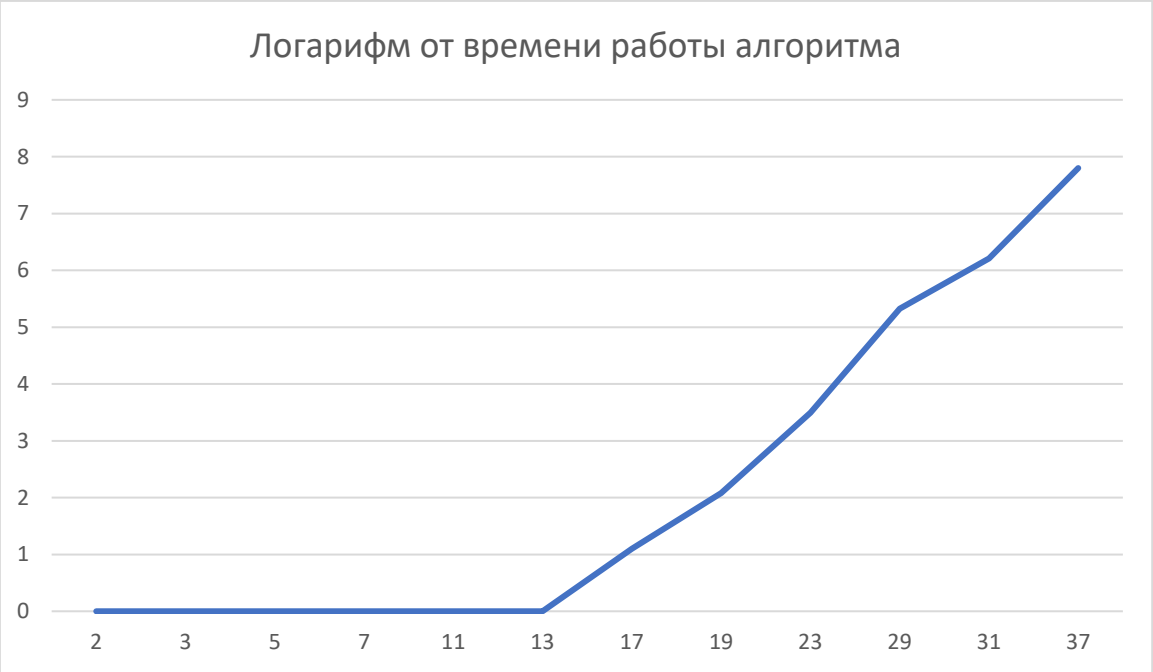
Исследование времени выполнения от размера квадрата.

Измерим время работы алгоритма для простых чисел.

Размер	Время, мс
2	0
3	0
5	0
7	0
11	0
13	1
17	3
19	8
23	33
29	205
31	498
37	2438

Графики время работы алгоритма от размера поля:





Логарифмический график схож с прямой, что значит – зависимость времени выполнения от размера квадрата близится к экспоненциальной.

Выводы.

В результате работы была написана программа, решающая поставленную задачу при помощи рекурсивного бэктрекинга. По результатам исследования зависимость времени работы алгоритма от размера квадрата экспоненциальна.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <chrono>
struct board {
    int x;
    int y;
    int w;
};

int bestcount;
std::vector<board> bestarray;
int n;
int boardsize;

bool Collision(std::vector<board>& one, int x, int y) {
    for (board square:one) {
        if (x >= square.x && x < square.x + square.w &&
            y >= square.y && y < square.y + square.w) {
            return true;
        }
    }
    return false;
}
void Backtracking(std::vector<board>& one, int summ, int boardlen, int
minx, int miny) {
    for (int x = minx; x < n; x++) {
        for (int y = miny; y < n; y++) {
            // проверка текущей клетки на пересечение с уже поставленными
квадратами
            if (!Collision(one,x,y)) {
                // максимально возможный радиус
                int right = std::min(n-1, std::min(n-x, n-y));
                for (board square:one) {
                    if (square.x+square.w > x && square.y > y) {
                        right = std::min(right, square.y-y);
                    }
                }
                // перебор всех возможных радиусов
                for (int r = right; r > 0; r--) {
                    board square({x, y, r});
```

```
                    std::vector<board> b = one;
                    b.push_back(square);
                    // заполено ли поле
                    if (summ + square.w * square.w == n * n) {
                        // обновление ответа
                        if (boardlen + 1 < bestcount) {
                            bestcount = boardlen + 1;
                            bestarray = b;
                        }
                    }
                    else {
                        // отсечение решений, которые заведомо хуже
текущего лучшего
                        if (boardlen + 1 < bestcount)
                            Backtracking(b, summ + square.w * square.w,
boardlen + 1, x, y+r);
                        else return;
                    }
                }
                return;
            }
        }
        miny = n/2;
    }
}
int main() {
    std::cin >> n;
    // поиск наибольшего делителя n
    for (int i = 1; i < n; i++) {
        if (n%i == 0) boardsize = i;
    }
    n = n / boardsize;
    // начальное приближение
    bestcount = 2 * n + 1;
    std::vector<board> table;
    table.push_back(board({0,0,(n + 1)/2}));
    table.push_back({0,(n + 1)/2,n/2});
    table.push_back({(n + 1)/2,0,n/2});
    auto begin = std::chrono::steady_clock::now();
    Backtracking(table, ((n + 1)/2)*((n + 1)/2) + 2 *(n/2)*(n/2), 3,
                 n/2, (n + 1)/2);
    auto end = std::chrono::steady_clock::now();
```

```
    auto elapsed_ms =
std::chrono::duration_cast<std::chrono::milliseconds>(end - begin);
    std::cout << "The time: " << elapsed_ms.count() << " ms\n";
    std::cout << bestcount << '\n';
    for (auto i: bestarray) {
        std::cout << i.x * boardsize << ' ' << i.y * boardsize
                  << ' ' << i.w * boardsize << '\n';
    }
}
```