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. 3384 _____ Рудаков А.Л.

Преподаватель _____ Шевелева А.М.

Санкт-Петербург
2025

Цель работы.

Написание алгоритма поиска с возвратом на языке программирования. Анализ созданного алгоритма.

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≤40).

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).

Выполнение работы.

В начале работы вызывается функция *int read(),* которая считывает значение – размер стороны квадрата, после чего возвращает его.

Далее вызывается функция *int find_smallest_divisor(int value),* которая находит минимальный простой делитель поданного числа, после чего возвращает его.

Далее создается объект класса *Table*. Внутри себя он хранит *std::vector<std::vector<int>> table* – матрица, изначально заполненная 0, которая используется для расставления квадратов (занятым клеткам присваиваются значения 1), *std::vector<SquaresInfo> squares* - вектор минимального набора квадратов, заполняющих изначальный квадрат. Структура *SquaresInfo* содержит в себе информацию о квадрате, а если конкретно координаты и размер. Поле - *int size_of_square* – размер изначально поданного квадрата, *int smallest_divisor* – размер минимального простого делителя, *int min_number* – минимальное количество квадратов, требуемых для заполнения, *int space* – число свободных клеток.

При создании объекта класса *table* становиться размером *smallest_divisor * smallest_divisor*, так как если сторона квадрата кратна какому-либо числу, то квадрат со стороной, размер которой равен этому числу разбивается таким же образом, как и изначальный квадрат. *table* заполняется нулями, *min_number* становиться равным *smallest_divisor * smallest_divisor* – количество квадратов со стороной 1, требуемых для заполнения.

Внутри класса содержаться методы:

- *void paint(int x, int y, int size_square)* – закрашивание найденного квадрата значениями 1.
- *void repaint(int x, int y, int size_square)* – перекрашивание найденного квадрата обратно значениями 0.
- *bool check_availability(int x, int y, int size_square)* – проверка возможности поставить квадрат по заданным координатам с заданными размерами. Возвращает *true* если можно, *false* если нельзя.
- *bool check_fullness()* – проверка на заполненность. Если в матрице еще осталось место, возвращается false, иначе true.
- *void cmp_with_other(std::vector<SquaresInfo> current_squares)* – проверка поданного вектора квадратов, заполняющих матрицу, на то,

являются ли они решением (если количество квадратов меньше *min_number*, то меняется *min_number* и *squares*).

- *void add_squares(int x, int y, int size_square, std::vector<SquaresInfo>& current_squares)* – добавление квадрата в цепочку текущих найденных при заполнении. Внутри вызывается метод *paint(…)*, уменьшается space и добавляется квадрат в *current_squares.*
- *void remove_squares(int x, int y, int size_square, std::vector<SquaresInfo>& current_squares)* – удаление квадрата из цепочки текущих найденных при заполнении. Внутри вызывается метод *repaint(…)*, увеличивается space и удаляется квадрат из current_squares.
- *std::pair<int, int> find_place()* – поиск первой пустой самой верхней левой координаты для вставки квадрата. Возвращаются координаты точки.
- *int calculate_max_size (int x, int y)* – вычисление максимального возможного размера квадрата для текущей точки. Размер не может быть больше *smallest_divisor – 1* из условия, так же не может быть больше *smallest_divisor - x + 1* и *smallest_divisor - y + 1*, так как иначе он выйдет за пределы квадрата, а так же нет смысла ставить квадрат, размером больше чем *(smallest_divisor / 2) + 1*.
- *void find_placement(std::vector<SquaresInfo>& current_squares)* – алгоритм *backtracking*`а. Вначале ищутся координаты для вставки текущего квадрата через метод *find_place()*, далее высчитывается максимальный возможный размер квадрата для данной точки через метод *calculate_max_size(…)*, После чего запускается цикл по размерам, от найденного *max_size* до 1 включительно. После этого происходит проверка на возможность поставить квадрат в данную точку через метод *check_availability(…)*, если возможно, то квадрат добавляется благодаря *add_squares(…)*, после чего поле проверяется

на полную занятость *через check_fullness(),* если все закрашено, то вызывается *cmp_with_other(current_squares),* иначе запускается рекурсия для следующего квадрата через *find_placement(current_squares).* После этого квадрат удаляется из текущих значений через *remove_squares(…).*

- *void start_place(std::vector<SquaresInfo>& current_squares)* – создает начальную расстановку из 3х наибольших квадратов, расставленных по своим местам (по сути создано для ускорения вычисления значения для квадрата, с размером 37, в остальных случаях можно обойтись и без него).
- *void update_answer()* запускает начальную расстановку через *start_place(…),* после чего запускает поиск квадратов через *find_placement(…),* вычисляет разницу между размером начального квадрата и минимальным делителем, после чего увеличивает координаты и размеры квадратов в соответствии с найденным соотношением.
- *void print_answer()* – запускает *update_unswer()* и выводит полученные значения.

После создания объекта *Table* с начальными значениями вызывается метод *print_answer().*

Разработанный программный код см. в приложении А.

Результаты тестирования см. в приложении Б.

Дополнительное задание.

Исследовать задачу "Поиск с возвратом" на количество шагов и время выполнения от длины столешницы N. Количество шагов - сколько итераций делает ваша программа для поиска значения. Обязательно подробно описать, как вы проводили исследование, на каких данных, какой получен результат. Продемонстрировать результат на графиках. Для времени результат строить минимум на 10 снятиях для каждого выбранного N, т.е. потом высчитать

среднее значение из этих 10 снятий. А-ля, при N=3 программа выдает время 3, 5, 7 мс, среднее значение 5 мс - оно и будет в графике. Также посчитайте среднеквадратичное отклонение для исследования на время

Исследование.

Для каждого N от 2 до 40 был запущен написанный алгоритм в количестве 10 раз. Во время запуска засекалось время работы алгоритма, после чего высчитывалось среднеквадратичное отклонение. Кроме того были посчитаны количества операций. Для замеров времени использовалась библиотека chrono. Все результаты, полученные в ходе исследования в табл. 1.

Таблица 1 – Результаты замеров

Размер	Среднее время (с)	Среднеквадратичное отклонение (с)	Количество итераций
2	0.0000010	0.0000010	4
3	0.0000034	0.0000010	18
4	0.0000010	0.0000001	4
5	0.0000034	0.0000010	14
6	0.0000010	0.0000001	4
7	0.0000130	0.0000040	64
8	0.0000010	0.0000001	4
9	0.0000030	0.0000010	18
10	0.0000010	0.0000001	4
11	0.0001963	0.0000660	682
12	0.0000011	0.0000001	4
13	0.0004910	0.0001640	1280
14	0.0000011	0.0000001	4
15	0.0000031	0.0000010	18
16	0.0000010	0.0000001	4

Продолжение таблицы 1

Размер	Среднее время (с)	Среднеквадратичное отклонение (с)	Количество итераций
17	0.0040531	0.0013710	5234
18	0.0000012	0.0000001	4
19	0.0124917	0.0042000	16962
20	0.0000010	0.0000001	4
21	0.0000031	0.0000010	18
22	0.0000010	0.0000001	4
23	0.0527474	0.0176180	52743
24	0.0000011	0.0000001	2
25	0.0000032	0.0000010	14
26	0.0000010	0.0000001	4
27	0.0000031	0.0000010	18
28	0.0000010	0.0000001	4
29	0.3090601	0.1031690	212287
30	0.0000011	0.0000001	4
31	0.8447768	0.2817720	517226
32	0.0000011	0.0000001	4
33	0.0000032	0.0000010	18
34	0.0000010	0.0000001	4
35	0.0000031	0.0000010	14
36	0.0000010	0.0000001	4
37	3.0150888	0.6617720	1356686
38	0.0000011	0.0000001	4
39	0.0000035	0.0000010	18
40	0.0000010	0.0000001	4

По результатам замеров были построены график зависимости времени выполнения от размера квадрата для всех точек рис.1, и график зависимости

времени выполнения от размера квадрата для простых чисел рис.2. Кроме того построен график зависимости количества операций от значения размера квадрата для простых чисел рис.3.

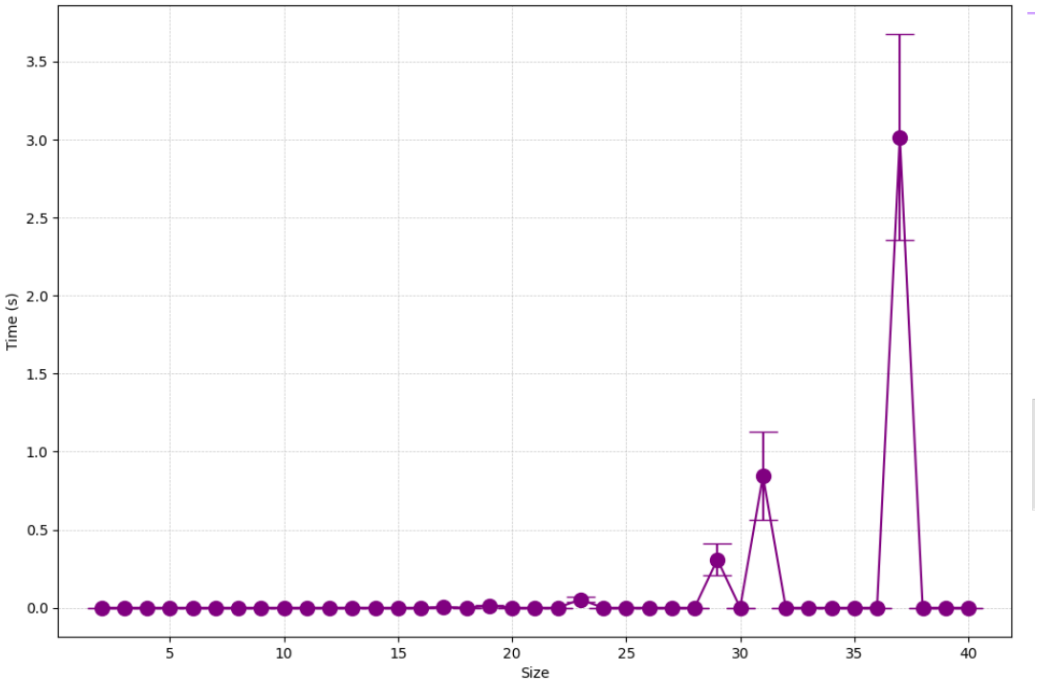

Рисунок 1 – Зависимость времени выполнения от размера

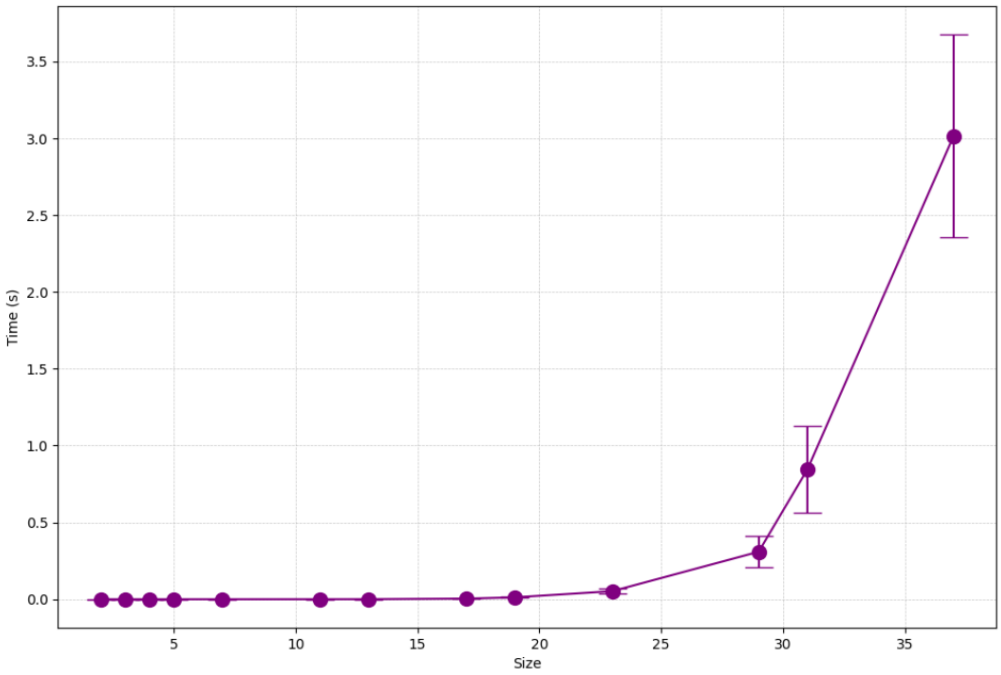

Рисунок 2 – Зависимость времени выполнения от размера (простые числа)

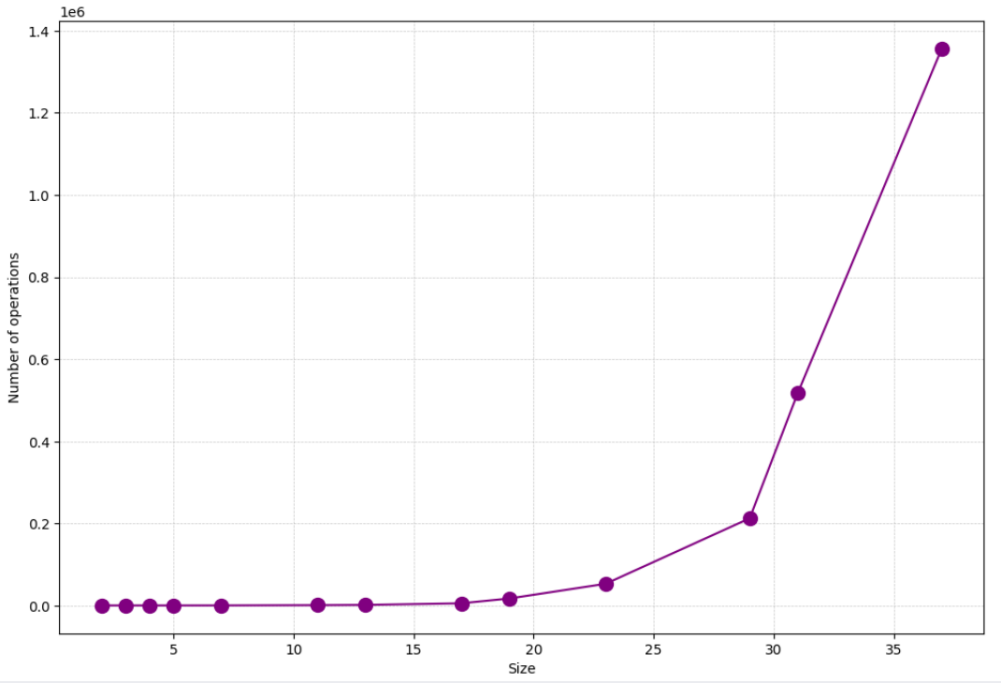

Рисунок 3 – Зависимость количества операций от размера (простые числа)

Исходя из графиков, представленных на рис.2 и рис.3 можно заметить, что время исполнения алгоритма и количество операций растет экспоненциально с увеличением размера квадрата для простых чисел.

Выводы.

В ходе работы был реализован алгоритм поиска с возвратом, позволяющий решать задачу о нахождении разбиения квадрата со стороной N на минимальное число квадратов меньшего размера без пересечений и выходов за пределы квадрата.

Задача была сведена к поиску расстановки квадратов в квадрате со стороной, равной минимальному простому делителю N, так как для квадрата большего размера подходит разбиение меньшего кратного размера, масштабируемого в начальный вид. Алгоритм рекурсивно перебирает все варианты квадратов, не повторяя ни одну расстановку, которая была до этого. Алгоритм расставляет квадраты, находя текущий максимум для размера, учитывая возможность разместить квадрат и количество квадратов, уже поставленных на поле. Если квадратов уже больше минимального полученного ранее числа, алгоритм выйдет из текущей рекурсии.

Алгоритм имеет экспоненциальную сложность по времени и количеству операций. Внутри происходит поиск места для вставки, который в худшем случае занимает O(n^2). Далее идет итерация по размерам, в общем случае занимающая O(n/2 + 1) = O(n/2). Постановка квадрата и удаление квадрата выполняются за O(m^2), где m – размер квадрата, максимальный размер квадрата, который ставится на поле n/2 + 1, поэтому сложность этих операций O((n/2)^2). Глубина рекурсии в общем случае O((n/2)^2). Тогда сложность в худшем случае O(n^2 * (n/2)^2 * ((n/2)^2 * (n/2))) = O(n^2 * (n/2)^5). Но в то же время по графикам видно, что сложность алгоритма экспоненциальная.

Данный алгоритм хорошо работает для небольших чисел, однако для больших время заметно увеличивается.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

int read() {
    int size_of_square;
    std::cin >> size_of_square;
    return size_of_square;
}

int find_smallest_divisor(int value) {
    int returned_value = 1;
    for (int i = 2; i <= value; i++) {
        if (value % i == 0) {
            returned_value = i;
            break;
        }
    }
    return returned_value;
}

struct SquaresInfo {
    int x;
    int y;
    int square_size;
} typedef SquaresInfo;

class Table {
private:
    std::vector<std::vector<int>> table;
    std::vector<SquaresInfo> squares;
    int size_of_square;
    int smallest_divisor;
```

```
        int min_number;
        int space;

        void paint(int x, int y, int size_square) {
            for (int i = x; i < x + size_square; i++) {
                for (int j = y; j < y + size_square; j++) {
                    table[i][j] = 1;
                }
            }
        }

        void repaint(int x, int y, int size_square) {
            for (int i = x; i < x + size_square; i++) {
                for (int j = y; j < y + size_square; j++) {
                    table[i][j] = 0;
                }
            }
        }

        bool check_availability(int x, int y, int size_square) {
            if (x + size_square > smallest_divisor || y + size_square >
smallest_divisor) return false;
            for (int i = x; i < x + size_square; i++) {
                for (int j = y; j < y + size_square; j++) {
                    if (table[i][j] != 0) return false;
                }
            }
            return true;
        }

        bool check_fullness() {
            if (space == 0) return true;
            return false;
        }

        void cmp_with_other(std::vector<SquaresInfo> current_squares) {
            if (current_squares.size() < min_number || min_number ==
smallest_divisor * smallest_divisor) {
                min_number = current_squares.size();
```

```
                squares = current_squares;
            }
        }

        void add_squares(int x, int y, int size_square,
std::vector<SquaresInfo>& current_squares) {
            paint(x, y, size_square);
            current_squares.push_back(SquaresInfo{ x, y, size_square });
            space -= size_square * size_square;
        }

        void remove_squares(int x, int y, int size_square,
std::vector<SquaresInfo>& current_squares) {
            repaint(x, y, size_square);
            current_squares.pop_back();
            space += size_square * size_square;
        }

        std::pair<int, int> find_place() {
            for (int i = 0; i < smallest_divisor; i++) {
                for (int j = 0; j < smallest_divisor; j++) {
                    if (table[i][j] == 0) {
                        return std::make_pair(i, j);
                    }
                }
            }
        }

        int calculate_max_size (int x, int y){
            int max_size = std::min(smallest_divisor - x + 1,
smallest_divisor - y + 1);
            max_size = std::min(max_size, smallest_divisor - 1);
            max_size = std::min(max_size, (smallest_divisor / 2) + 1);
            return max_size;
        }

        void find_placement(std::vector<SquaresInfo>& current_squares)
{
            std::pair<int, int> coordinate = find_place();
```

```
            int x = coordinate.first;
            int y = coordinate.second;
            int max_size = calculate_max_size(x, y);

            if (current_squares.size() + space / max_size > min_number)
{
                return;
            }

            for (int current_size = max_size; current_size > 0;
current_size--) {
                if (check_availability(x, y, current_size)) {
                    add_squares(x, y, current_size, current_squares);
                    if (check_fullness()) {
                        cmp_with_other(current_squares);
                    }
                    else {
                        find_placement(current_squares);
                    }
                    remove_squares(x, y, current_size, current_squares);
                }
            }
        }

        void start_place(std::vector<SquaresInfo>& current_squares) {
            if (smallest_divisor > 3) {
                int size_square = smallest_divisor / 2 + 1;
                add_squares(size_square - 1, size_square - 1,
size_square, current_squares);
                add_squares(0, size_square, size_square - 1,
current_squares);
                add_squares(size_square, 0, size_square - 1,
current_squares);
            }
        }

    public:
        Table(int size_of_square, int smallest_divisor) :
size_of_square(size_of_square),
```

```
            smallest_divisor(smallest_divisor),
min_number(smallest_divisor*  smallest_divisor),  space(smallest_divisor*
smallest_divisor) {
            table.resize(smallest_divisor);
            for (int i = 0; i < smallest_divisor; i++) {
                table[i].resize(smallest_divisor);
                for (int j = 0; j < smallest_divisor; j++) {
                    table[i][j] = 0;
                }
            }
        }

        void update_answer() {
            std::vector<SquaresInfo> current_squares;
            start_place(current_squares);
            find_placement(current_squares);
            int mul = size_of_square / smallest_divisor;
            for (int i = 0; i < min_number; i++) {
                squares[i].x = (squares[i].x + 1) * mul - (mul - 1);
                squares[i].y = (squares[i].y + 1) * mul - (mul - 1);
                squares[i].square_size *= mul;
            }
        }

        void print_answer() {
            update_answer();
            std::cout << min_number << '\n';
            for (int i = 0; i < min_number; i++) {
                std::cout << squares[i].x << ' ' << squares[i].y << ' '
<< squares[i].square_size << '\n';
            }
        }

        ~Table() {
            for (int i = 0; i < smallest_divisor; i++) {
                table[i].clear();
            }
            table.clear();
            squares.clear();
```

```
    }
};

int main() {
    int size_of_squares = read();
    int smallest_divisor = find_smallest_divisor(size_of_squares);
    Table table = Table(size_of_squares, smallest_divisor);
    table.print_answer();
}
```

ПРИЛОЖЕНИЕ Б
ТЕСТИРОВАНИЕ

Таблица Б.1 - Примеры тестовых случаев

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2	4 1 1 1 1 2 1 2 1 1 2 2 1	Ok
2.	40	4 1 1 20 1 21 20 21 1 20 21 21 20	Ok
3.	5	8 3 3 3 1 4 2 4 1 2 1 1 2 1 3 1 2 3 1 3 1 1 3 2 1	Ok

Продолжение таблицы Б.1

№ п/п	Входные данные	Выходные данные	Комментарии
4.	35	8 15 15 21 1 22 14 22 1 14 1 1 14 1 15 7 8 15 7 15 1 7 15 8 7	Ok
5.	37	15 19 19 19 1 20 18 20 1 18 1 1 12 1 13 7 8 13 7 13 1 7 13 8 3 13 11 2 15 11 1 15 12 5 15 17 3 16 8 4 18 17 2 18 19 1	Ok