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: Перебор с возвратом

Студент гр. 9383 _____ Гладких А.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Применить на практике знания о построение алгоритмов поиска с возвратом, реализовать заполнение поля наименьшим числом квадратов, применив алгоритм поиска с возвратом. Исследовать количество операций в зависимости размеров поля.

Основные теоретические положения.

Поиск с возвратом или бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве M.

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.

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. 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.

Необходимо найти число K, задающее минимальное количество обрезков(квадратов), из которых можно построить столешницу(квадрат) заданного размера N и вывести 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).

Вариант 3и - Итеративный бэктрекинг. Исследование количества операций от размера квадрата.

Ход работы:

1. Произведён анализ задания.
2. Был реализован итеративный алгоритма поиска с возвратом:
    1. С помощью циклов *for* осуществляется перебор всех ячеек матрицы, которая является представлением столешницы. Изначально вся матрица состоит из 0.
    2. При нахождении первой свободной (нулевой) ячейки программа ищет максимально допустимый размер квадрата, левый верхний угол которой расположен в данной ячейке.
    3. Найдя размер, программа ставит на данную клетку квадрат — расставляет его номер в матрице, чтобы показать, что эти ячейки заняты, и запоминает квадрат на стеке.
    4. Шаги 2-3 повторяются, пока мы не заполним всю столешницу. Число квадратов, которое потребовалось, чтобы заполнить доску, запоминается в отдельной переменной.
    5. Дальше начинается движение обратно — программа ищет первый квадрат, сторона которого не равна единице, постепенно проходя к корню стека. Когда программа дошла до этого квадрата, она уменьшает его сторону на 1 и пытается заполнить столешницу снова.
    6. Если квадрат последний в стеке и его сторона равна 1, то программа заканчивает работу.
3. В ходе тестирования программы были добавлены следующие улучшения:
    1. Было замечено, что для столешниц, размер которых кратен 2, 3 и 5, число квадратов всегда равно 4, 6 и 8 соответственно, а координаты левых верхних углов квадратов зависят лишь от самого n. Эти случаи были вынесены отдельно.

2. После этого было замечено, что во всех оставшихся ситуациях заполненная столешница имеет в одном из углов три квадрата: один непосредственно в углу с размером стороны *(n/2) + 1*, а два других квадрата со сторонами *n/2* примыкаю к боковым сторонам квадрата и краям столешницы. Пример типичного вида заполненной столешницы представлен на рисунке 1. Ручное выставление этих квадратов заранее сокращает незаполненную площадь, ровно как и количество операций, примерно в 4 раза.

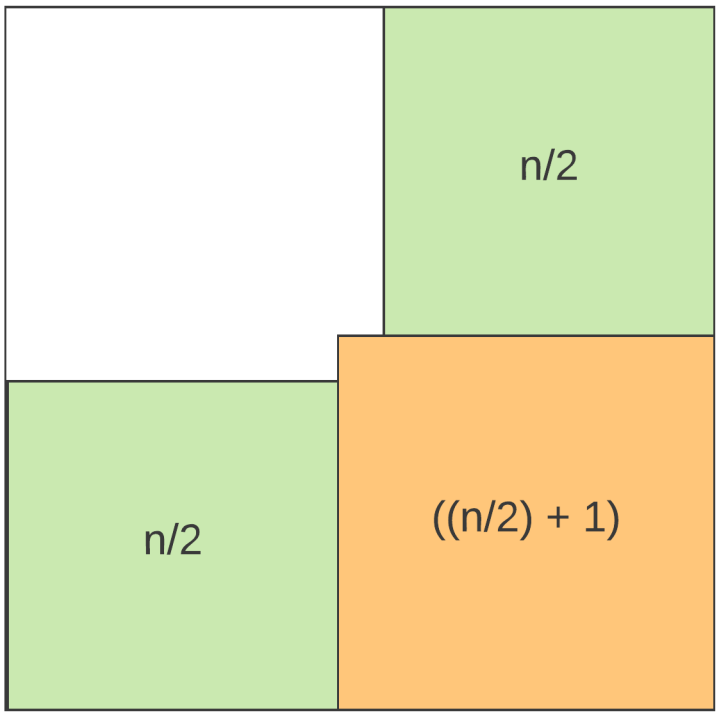

Рисунок 1 - Иллюстрация типичного вида заполненной столешницы, описываемого в пункте 3.2

4. Были написаны тесты с использованием библиотеки Catch2 для функций взаимодействия со столешницей.
5. Код разработанной программы расположен в Приложении А.

Описание функций и структур данных:

1. Структура Square – представление квадрата. Имеет поля *x*,*y* для хранения координат верхнего левого угла квадрата и поле *size* для хранения длины стороны.
2. Класс Field – представление столешницы. Имеет следующие поля — *size_* для хранения размера столешницы, *default_square_size_*для хранения размера выставляемого квадрата по умолчанию, *arr_* - матрица столешницы (представлена с помощью вложенных структур *std::vector* из стандартной библиотеки языка C++), *square_stack_* - стек, в котором хранятся квадраты, *squares_amount_* - количество выставленных квадратов, *cur_square_size_* - размер стороны выставляемого квадрата, *operations_amount_* для хранения числа операций и булева переменная *running_* для обработки остановки работы алгоритма.

   Поле задается с помощью конструктора и метода *Init()*. Методы *Print()* и *PrintStack()* нужны для вывода на экран поля и стека квадратов соответственно. Метод *PlaceSquare()* заполняет матрицу цифрами, которые соответствуют номеру выставляемого квадрата, а метод *DeleteSquare()* очищает область из-под квадрата, выставляя в ней заново 0. Метод *Fill()* - выполняет шаги 2, 3, 4 алгоритма, а метод *Backtrace()* выполняет шаг 5 алгоритма. Метод *CheckEnoughPlace()* проверяет достаточно ли места для выставления на столешницу квадрата заданного размера.
3. Функция main() - функция, в которой происходит инициализация поля и запуск алгоритма.

Примеры работы программы.

Таблица 1 – Пример работы программы

№ п/п	Входные данные	Выходные данные
1.	5	Enter n: 5 8 1 1 3 4 1 2 1 4 2 4 4 2 3 4 1 3 5 1 4 3 1 5 3 1
2.	13	Enter n: 13 Elapsed Time: 0.0005621s 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 2 2 2 2 2 2 1 1 1 1 1 1 1 4 4 5 5 5 5 3 3 3 3 3 3 6 4 4 5 5 5 5

		3 3 3 3 3 3 7 7 7 5 5 5 5 3 3 3 3 3 3 7 7 7 5 5 5 5 3 3 3 3 3 3 7 7 7 8 9 9 9 3 3 3 3 3 3 10 10 11 11 9 9 9 3 3 3 3 3 3 10 10 11 11 9 9 9 11 9 12 2 7 12 2 11 11 3 10 11 1 7 9 3 7 8 1 10 7 4 8 7 2 1 8 6 8 1 6 1 1 7
3.	37	Enter n: 37 Elapsed Time: 7.43436s 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2

		2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 4 4 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 7 4 4 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 8 8 8 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 8 8 8 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 8 8 8 5 5 5 5 5 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9

		9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 10 10 10 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 10 10 10 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 13 14 10 10 10 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11 15 25 33 5 26 32 1

		25 32 1 19 32 6 30 30 8 27 30 3 19 24 8 19 21 3 19 20 1 27 19 11 22 19 5 20 19 2 1 20 18 20 1 18 1 1 19

Иллюстрация работы программы.

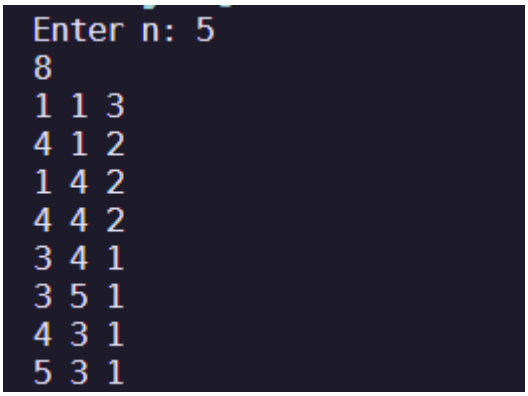

Рисунок 2 - Пример работы программы на входных данных №1

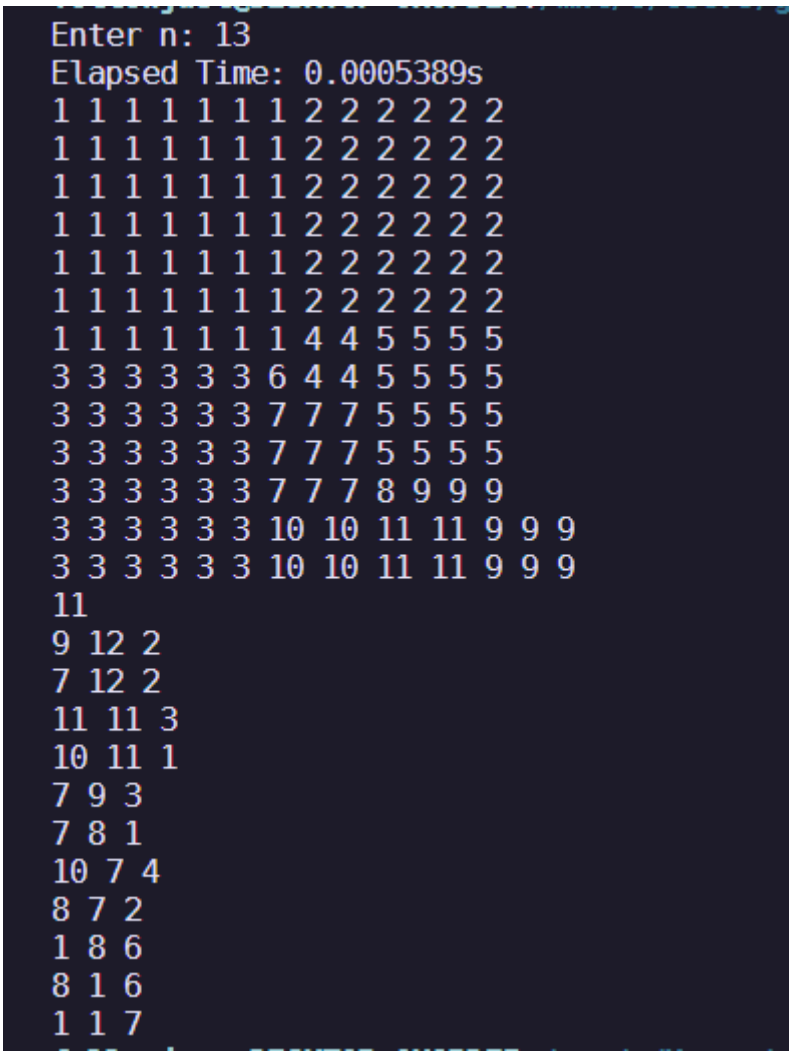

Рисунок 3 - Пример работы программы на входных данных №2

```
Enter n: 37
Elapsed Time: 7.73525s
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 1 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2 2
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 6 6 6 6 6 6 6 6 6 6 6
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 10 10 10 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 9 9 9 9 9 9 9 9 10 10 10 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 13 14 10 10 10 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11
3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 3 12 12 12 12 12 12 15 15 15 15 15 11 11 11 11 11 11 11 11
15
25 33 5
26 32 1
25 32 1
19 32 6
30 30 8
27 30 3
19 24 8
19 21 3
19 20 1
27 19 11
22 19 5
20 19 2
1 20 18
20 1 18
1 1 19
```

Рисунок 4 - Пример работы программы на входных данных №3

Исследование количества операций от размера квадрата.

Было проведено исследование количества операций от размера квадрата. Под операциями подразумевалось обращение к ячейке матрицы, которая представляла столешницу. В силу вышеописанных улучшений алгоритма, измерялось лишь количество операций на столешницах, длина сторон которых была простым числом. График зависимости представлен на рисунке 5.

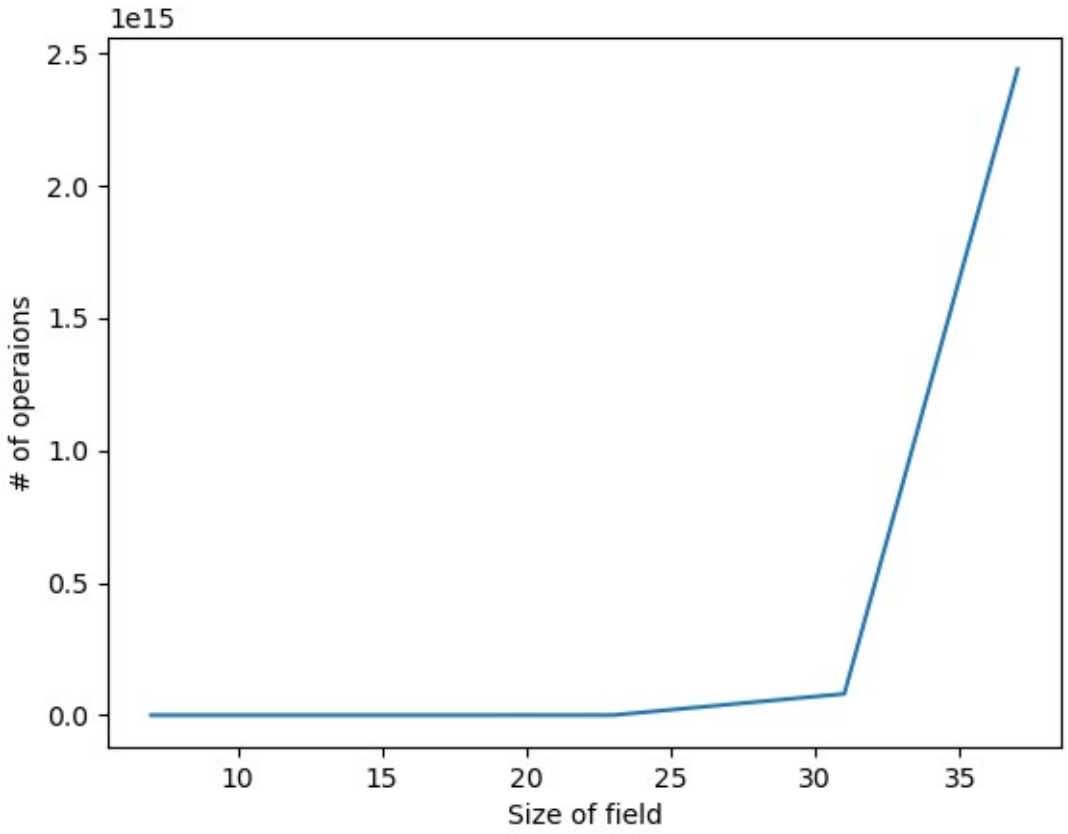

Рисунок 5 - График зависимости числа операций от размеров квадрата

Из результатов исследования видно, что количество операций увеличивается экспоненциально.

Выводы.

Были применены на практике знания о построение алгоритмов поиска с возвратом, на языке программирования C++ было реализовано заполнение поля наименьшим числом квадратов, применив алгоритм поиска с возвратом. Также было проведено исследование количества операций в зависимости размеров поля. По результатам исследования были сделаны выводы об экспоненциальном росте количества операций в зависимости от размеров поля.

ПРИЛОЖЕНИЕ А

Файл main.cpp

```
#include <iostream>

#include <vector>
#include <stack>
#include <chrono>
#include <cmath>

#include "Field.h"
#include "Square.h"


int main() {

    int n;
    std::cout << "Enter n: ";
    std::cin >> n;


    if(n % 2 == 0) {
        std::cout << 4 << "\n";

           std::cout << 1 << " " << 1 << " " << n / 2 << "\n";
           std::cout << 1 + n / 2 << " " << 1 << " " << n / 2 << "\n";
           std::cout << 1 << " " << 1 + n / 2 << " " << n / 2 << "\n";
           std::cout << 1 + n / 2 << " " << 1 + n / 2 << " " << n / 2 <<
"\n";
           return 0;
    }
    else if(n % 3 == 0) {
        std::cout << 6 << "\n";

           std::cout << 1 << " " << 1 << " " << 2 * n / 3 << "\n";

           std::cout << 1 + 2 * n / 3 << " " << 1 << " " << n / 3 << "\
n";
           std::cout << 1 << " " << 1 + 2 * n / 3 << " " << n / 3 << "\
n";
           std::cout << 1 + 2 * n / 3 << " " << 1 + n / 3 << " " << n /
3 << "\n";
           std::cout << 1 + n / 3 << " " << 1 + 2 * n / 3 << " " << n /
3 << "\n";
           std::cout << 1 + 2 * n / 3 << " " << 1 + 2 * n / 3 << " " <<
n / 3 << "\n";
           return 0;
    }
    else if(n % 5 == 0) {
        std::cout << 8 << "\n";
           std::cout << 1 << " " << 1 << " " << 3 * n / 5 << "\n";
           std::cout << 1 + 3 * n / 5 << " " << 1 << " " << 2 * n / 5 <<
"\n";
           std::cout << 1 << " " << 1 + 3 * n / 5 << " " << 2 * n / 5 <<
"\n";
```

```
            std::cout << 1 + 3 * n / 5 << " " << 1 + 3 * n / 5 << " " <<
2 * n / 5 << "\n";
            std::cout << 1 + 2 * n / 5 << " " << 1 + 3 * n / 5 << " " <<
n / 5 << "\n";
            std::cout << 1 + 2 * n / 5 << " " << 1 + 4 * n / 5 << " " <<
n / 5 << "\n";
            std::cout << 1 + 3 * n / 5 << " " << 1 + 2 * n / 5 << " " <<
n / 5 << "\n";
            std::cout << 1 + 4 * n / 5 << " " << 1 + 2 * n / 5 << " " <<
n / 5 << "\n";
            return 0;
    }

    Field field(n);
    Field min_field(n);

    field.set_cur_square_size(std::ceil(double(n) / 2));
    field.PlaceSquare(0, 0);

    field.set_cur_square_size(n / 2);
    field.PlaceSquare(std::ceil(double(n) / 2), 0);

    field.PlaceSquare(0, std::ceil(double(n) / 2));

    auto start = std::chrono::steady_clock::now();


    int min_amount = n * n;

    long long int operations_amount = 0;

    while(field.get_running()) {
        bool filled_succesfully = field.Fill(min_amount);

        operations_amount += field.get_operations_amount();

           if (filled_succesfully && field.get_squares_amount() < min_amount) {
             min_field = field;
                 min_amount = field.get_squares_amount();

           }

        field.Backtrace();

    }

    auto end = std::chrono::steady_clock::now();

    std::chrono::duration<double> elapsed_seconds = end - start;

    std::cout << "Elapsed time: " << elapsed_seconds.count() << "s\n";
    std::cout << "Operaions: " << operations_amount << "\n";
    min_field.Print();
    min_field.PrintStack();
    return 0;
```

```
}
```

Файл Square.h

```
#ifndef SQUARE_H
#define SQUARE_H

struct Square {
     int x;
     int y;
     int size;
};

#endif
```

Файл Field.h

```
#ifndef FIELD_H
#define FIELD_H

#include <vector>
#include <stack>
#include <iostream>

#include "Square.h"

class Field {

public:

     Field(int size);

     void Print();

     void PrintStack();

     void PlaceSquare(int x, int y);
     void DeleteSquare();

     bool Fill(int min_amount);

     void Backtrace();

     bool get_running() { return running_; }

     const std::vector<std::vector<int>> get_arr() { return arr_; }

     int get_squares_amount() { return squares_amount_; }

     long long int get_operations_amount() { return operations_amount_; }

     void set_cur_square_size(int value) { cur_square_size_ = value; }

private:
     int size_;
```

```
    int default_square_size_;
    std::vector<std::vector<int>> arr_;
    std::stack<Square> square_stack_;

    int squares_amount_ = 0;
    int cur_square_size_ = 0;
    long long int operations_amount_ = 0;

    bool running_ = true;

    void Init();

    bool CheckEnoughPlace(int start_x, int start_y);

};

#endif
```

Файл Field.cpp

```
#include "Field.h"


Field::Field(int size) {
    size_ = size;
    default_square_size_ = size_ / 2;
    cur_square_size_ = default_square_size_;
    Init();
}

void Field::Print() {
    for(auto line: arr_) {
        for(auto cell: line) {
            std::cout << cell << " ";
        }
        std::cout << "\n";
    }
}

void Field::PrintStack() {
    //std::cout << "Total: " << square_stack_.size() << " squares\n";
    std::cout << square_stack_.size() << "\n";
    while(!square_stack_.empty()) {
        Square square = square_stack_.top();
        square_stack_.pop();
           std::cout << square.x + 1 << " " << square.y + 1 << " " <<
square.size << "\n";
    }
}

void Field::PlaceSquare(int x, int y) {
    squares_amount_++;

     for (int i = y; i < y + cur_square_size_; i++) {
           for (int j = x; j < x + cur_square_size_; j++) {
             ++operations_amount_;
                 arr_[i][j] = squares_amount_;
```

```
            }
        }

        SQUARE_STACK_.PUSH({ X, Y, CUR_SQUARE_SIZE_ });

    CUR_SQUARE_SIZE_ = DEFAULT_SQUARE_SIZE_;
}

BOOL FIELD::FILL(INT MIN_AMOUNT) {
        FOR (INT Y = SIZE_ / 2; Y < SIZE_; Y++) {
            FOR (INT X = SIZE_ / 2; X < SIZE_; X++) {
              ++OPERATIONS_AMOUNT_;
                IF (!ARR_[Y][X]) {
                    IF (SQUARES_AMOUNT_ >= MIN_AMOUNT) {
                        RETURN FALSE;
                    }
                WHILE(!CHECKENOUGHPLACE(X, Y)) {
                    --CUR_SQUARE_SIZE_;
                }
                //STD::COUT << "X: " << X << " Y: " << Y << "\N";
                //STD::COUT << CUR_SQUARE_SIZE_ << "\N";
                PLACESQUARE(X, Y);
                }
            }
        }
        RETURN TRUE;
}

VOID FIELD::BACKTRACE() {
    SQUARE LAST_SQUARE = SQUARE_STACK_.TOP();
    WHILE(SQUARE_STACK_.SIZE() > 3 && LAST_SQUARE.SIZE == 1) {
        DELETESQUARE();
        LAST_SQUARE = SQUARE_STACK_.TOP();
    }
     IF(SQUARE_STACK_.SIZE() == 3) { // && SQUARE_STACK_.TOP().SIZE == 1 //
SQUARE_STACK_.TOP().SIZE < SIZE_ / 2
        RUNNING_ = FALSE;
        RETURN;
    }

    INT X = LAST_SQUARE.X;
    INT Y = LAST_SQUARE.Y;
    CUR_SQUARE_SIZE_ = LAST_SQUARE.SIZE - 1;

    DELETESQUARE();

    PLACESQUARE(X, Y);
}


VOID FIELD::INIT() {
    ARR_.RESIZE(SIZE_);
    FOR(AUTO& LINE: ARR_) {
        LINE.RESIZE(SIZE_);
        FOR(AUTO& CELL: LINE) {
            CELL = 0;
        }
```

```
    }
}

bool Field::CheckEnoughPlace(int start_x, int start_y) {
      if (start_y + cur_square_size_ > size_ || start_y < 0) { return false; }
      if (start_x + cur_square_size_ > size_ || start_x < 0) { return false; }
    for (int y = start_y; y < start_y + cur_square_size_; y++) {
            for (int x = start_x; x < start_x + cur_square_size_; x++) {
                  if(arr_[y][x]) {
                  return false;
              }
            }
      }
    return true;
}

void Field::DeleteSquare() {
    Square last_square = square_stack_.top();
    square_stack_.pop();
    --squares_amount_;

    for (int y = last_square.y; y < last_square.y + last_square.size; y++) {
            for (int x = last_square.x; x < last_square.x + last_square.size; x+
+) {
              ++operations_amount_;
                  arr_[y][x] = 0;
            }
      }
}
```

Файл Test_Field.cpp

```
#define CATCH_CONFIG_MAIN

#include "../../catch.hpp"
#include "../source/Field.h"

std::vector<std::vector<int>> answer;

TEST_CASE("Field Constructor Test", "[internal Field Test]" ) {
    Field field1(2);
    Field field2(5);
    Field field3(10);

    answer = {
        {0, 0},
        {0, 0}
    };

    REQUIRE(field1.get_arr() == answer);

    answer = {
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0}
```

```
    };

    REQUIRE(field2.get_arr() == answer);

    answer = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0}
    };

    REQUIRE(field3.get_arr() == answer);
}

TEST_CASE("PlaceSquare() and set_cur_square_size() test", "[internal Field
Test]" ) {
    Field field(5);
    field.set_cur_square_size(2);
    field.PlaceSquare(0, 0);

    answer = {
        {1, 1, 0, 0, 0},
        {1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0}
    };

    REQUIRE(field.get_arr() == answer);

    field.set_cur_square_size(1);
    field.PlaceSquare(2, 0);

    answer = {
        {1, 1, 2, 0, 0},
        {1, 1, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0}
    };

    REQUIRE(field.get_arr() == answer);

    field.set_cur_square_size(3);
    field.PlaceSquare(2, 1);

    answer = {
        {1, 1, 2, 0, 0},
        {1, 1, 3, 3, 3},
        {0, 0, 3, 3, 3},
        {0, 0, 3, 3, 3},
```

```
        {0, 0, 0, 0, 0}
    };

    REQUIRE(field.get_arr() == answer);
}

TEST_CASE("DeleteSquare() Test", "[internal Field Test]" ) {
    Field field(5);
    field.set_cur_square_size(2);
    field.PlaceSquare(0, 0);

    field.set_cur_square_size(3);
    field.PlaceSquare(2, 0);

    field.set_cur_square_size(2);
    field.PlaceSquare(0, 2);

    field.PlaceSquare(2, 3);

    answer = {
        {1, 1, 2, 2, 2},
        {1, 1, 2, 2, 2},
        {3, 3, 2, 2, 2},
        {3, 3, 4, 4, 0},
        {0, 0, 4, 4, 0}
    };

    REQUIRE(field.get_arr() == answer);


    field.DeleteSquare();

    answer = {
        {1, 1, 2, 2, 2},
        {1, 1, 2, 2, 2},
        {3, 3, 2, 2, 2},
        {3, 3, 0, 0, 0},
        {0, 0, 0, 0, 0}
    };

    REQUIRE(field.get_arr() == answer);
}


TEST_CASE("Backtrace() Test", "[internal Field Test]" ) {
    Field field(5);
    field.set_cur_square_size(2);
    field.PlaceSquare(0, 0);

    field.set_cur_square_size(2);
    field.PlaceSquare(2, 0);

    field.set_cur_square_size(2);
    field.PlaceSquare(0, 2);

    field.set_cur_square_size(3);
```

```
FIELD.PLACESQUARE(2, 2);

ANSWER = {
     {1, 1, 2, 2, 0},
     {1, 1, 2, 2, 0},
     {3, 3, 4, 4, 4},
     {3, 3, 4, 4, 4},
     {0, 0, 4, 4, 4}
};

REQUIRE(FIELD.GET_ARR() == ANSWER);


FIELD.BACKTRACE();

ANSWER = {
     {1, 1, 2, 2, 0},
     {1, 1, 2, 2, 0},
     {3, 3, 4, 4, 0},
     {3, 3, 4, 4, 0},
     {0, 0, 0, 0, 0}
};

REQUIRE(FIELD.GET_ARR() == ANSWER);

FIELD.BACKTRACE();

ANSWER = {
     {1, 1, 2, 2, 0},
     {1, 1, 2, 2, 0},
     {3, 3, 4, 0, 0},
     {3, 3, 0, 0, 0},
     {0, 0, 0, 0, 0}
};

REQUIRE(FIELD.GET_ARR() == ANSWER);

FIELD.SET_CUR_SQUARE_SIZE(2);
FIELD.PLACESQUARE(3, 3);

ANSWER = {
     {1, 1, 2, 2, 0},
     {1, 1, 2, 2, 0},
     {3, 3, 4, 0, 0},
     {3, 3, 0, 5, 5},
     {0, 0, 0, 5, 5}
};

REQUIRE(FIELD.GET_ARR() == ANSWER);

FIELD.BACKTRACE();

ANSWER = {
     {1, 1, 2, 2, 0},
     {1, 1, 2, 2, 0},
     {3, 3, 4, 0, 0},
```

```
        {3, 3, 0, 5, 0},
        {0, 0, 0, 0, 0}
    };

    REQUIRE(field.get_arr() == answer);
}

TEST_CASE("Fill() Test", "[internal Field Test]" ) {
    Field field(5);

    field.Fill(25);

    answer = {
        {0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0},
        {0, 0, 1, 1, 2},
        {0, 0, 1, 1, 3},
        {0, 0, 4, 5, 6}
    };

    REQUIRE(field.get_arr() == answer);

    Field field1(10);

    field1.set_cur_square_size(1);
    field1.Fill(50);

    answer = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1, 2, 2, 2, 2},
        {0, 0, 0, 0, 0, 3, 2, 2, 2, 2},
        {0, 0, 0, 0, 0, 4, 2, 2, 2, 2},
        {0, 0, 0, 0, 0, 5, 2, 2, 2, 2},
        {0, 0, 0, 0, 0, 6, 7, 8, 9, 10}
    };

    REQUIRE(field1.get_arr() == answer);


    Field field2(10);

    field2.set_cur_square_size(3);
    field2.Fill(4);

    answer = {
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 0, 0, 0, 0, 0},
        {0, 0, 0, 0, 0, 1, 1, 1, 2, 2},
        {0, 0, 0, 0, 0, 1, 1, 1, 2, 2},
        {0, 0, 0, 0, 0, 1, 1, 1, 3, 3},
```

```
        {0, 0, 0, 0, 0, 4, 4, 0, 3, 3},
        {0, 0, 0, 0, 0, 4, 4, 0, 0, 0}
    };

    REQUIRE(field2.get_arr() == answer);

}
```

Файл Graphic.py

```
import matplotlib.pyplot as plt

x = [7, 11, 13, 19, 23, 31, 37]
y = [5229, 2216969, 14819233, 1360878045, 188898355755,
80554248332774, 2440483984772697]

plt.plot(x, y)

#plt.yscale("log")

plt.ylabel("# of operaions")
plt.xlabel("Size of field")
plt.show()
```

Файл makefile

```
FLAGS = -std=c++17 -Wall -Wextra
BUILD = build
SOURCE = source
TEST = test

$(shell  mkdir -p $(BUILD))


all: lab1

lab1: $(BUILD)/main.o $(BUILD)/Field.o
      @echo "To start enter ./lab1"
      @g++ $(BUILD)/main.o $(BUILD)/Field.o -o lab1 $(FLAGS)

run_tests: $(BUILD)/test_field.o $(BUILD)/Field.o
      @g++ $(BUILD)/test_field.o $(BUILD)/Field.o -o run_tests

$(BUILD)/test_field.o: $(TEST)/test_field.cpp
      @g++ -c $(TEST)/test_field.cpp -o $(BUILD)/test_field.o

$(BUILD)/Field.o: ${SOURCE}/Field.cpp
      @g++ -c ${SOURCE}/Field.cpp -o $(BUILD)/Field.o

$(BUILD)/main.o: ${SOURCE}/main.cpp
      @g++ -c $(SOURCE)/main.cpp -o $(BUILD)/main.o

clean:
      @rm -rf $(BUILD)/
```

```
@RM -RF *.O LAB1 RUN_TESTS
```