МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 8304 _____ Чебесова И.Д.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Научиться применять алгоритм поиска с возвратом (бэктрекинг) для решения практических задач, реализовав программу заполнения поля наименьшим числом квадратов.

Основные теоретические положения.

Поиск с возвратом, бэктрекинг — это 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.

Вариант 3р. Рекурсивный бэктрекинг. Исследование количества операций от размера квадрата.

У Вовы много квадратных обрезков доски. Их стороны (размер) изменяются от 11 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).

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 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 ($0 \leq x, y \leq N$) и длину стороны соответствующего обрезка(квадрата).

Описание алгоритма.

Для решения поставленной задачи был реализован рекурсивный поиск с возвратом.

На каждом этапе рекурсии находим самый верхний левый угол, образованный сторонами каких-либо квадратов, т.е. пустой левый верхних угол. В этот угол поочерёдно помещаются квадраты допустимых размеров (не выходящих за границу изначального квадрата и не пересекающих уже размещённые квадраты) и для каждого нового квадрата рекурсивно вызывается функция, выполняющая все те же операции. Получаем перебор возможных вариантов расположений.

Если после вставки очередного квадрата всё поле N x N оказалось заполнено, то решение сохраняется при условии, что количество квадратов в данном решении является минимальным из всех полученных ранее.

Используемые оптимизации.

1. Можно дать верхнюю оценку количества необходимых квадратов N+3 для нечетного N=2k+1.

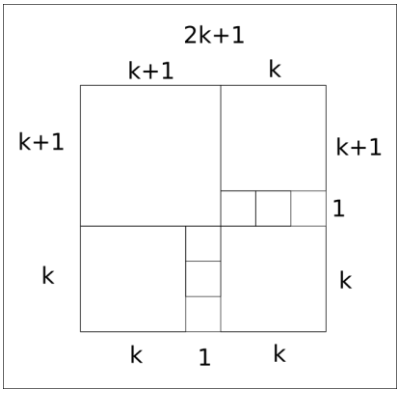

Рисунок 1 — Верхняя оценка

2. Если глубина превосходит размер уже найденного решения (или значения верхней оценки), то ветвь поиска отсекается, ведь рассматривать ее нет смысла.

3. Для квадратов размеров кратных 2, 3, 5 минимальное число квадратов равно 4, 6, 8 соответственно.

4. Если N — простое и нечетное, то три квадрата размерами N/2 + 1, N/2 и N/2 в решении будут размещены в верхнем левом углу (например, как на рисунке 1). Добавление этих квадратов перед началом перебора позволяет уменьшить размер замощаемой площади и количество выполняемых операций примерно в 4 раза.

5. Перебор размеров квадратов начинается с наибольшей возможной стороны.

Так как идет перебор возможных вариантов, то сложность алгоритма будет экспоненциальной и равно $O(2^n)$

Описание функций и структур данных.

Для решения задачи был реализована структура *Square* для хранения квадратов (координаты левого верхнего угла и размера), а также структура *TableTop* для хранения заполненной квадратами столешницы (в ней хранится размер столешницы, поданный во входных данных, массив *filled*, который

отвечает за заполненность каждого конкретного столбца, массив квадратов *current*, рассматриваемый на данный момент, а также минимальный массив квадратов *minSquares* и минимальное количество квадратов *minAmount*).

Функция *getTopColNumber* возвращает номер минимального столбца, в который может быть вставлен квадрат (ищется самый левый верхний незаполненный угол).

Функция *spaceBelow* возвращает количество свободного места внизу, а *spaceRight* – справа.

Функции *insertSquare* и *deleteSquare* отвечают за вставку и удаление квадрата соответственно.

Функция *saveMinSquares* отвечает за сохранения данных, если они минимальны из всех предыдущих расстановок.

Функция *analysisN* является функцией, запускающей алгоритм решения задачи. Она проверяет N на возможность оптимизации алгоритма, выполняет необходимые действия и отсылает нас к самому бэктрекингу.

Функция *backtracking* непосредственно решает задачу, рекурсивно вызывая саму себя.

Тестирование программы.

1.

```
Size of table top: 4
Min number of squares: 4
x coord: 0; y coord: 0; size: 2
x coord: 2; y coord: 0; size: 2
x coord: 0; y coord: 2; size: 2
x coord: 2; y coord: 2; size: 2
Number of operations: 20
```

Рисунок 2 – Пример работы программы при N=4

2.

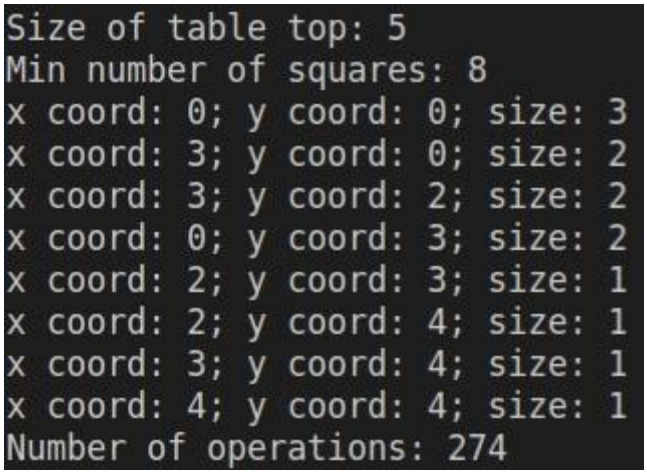

Рисунок 3 – Пример работы программы при N=5

3.

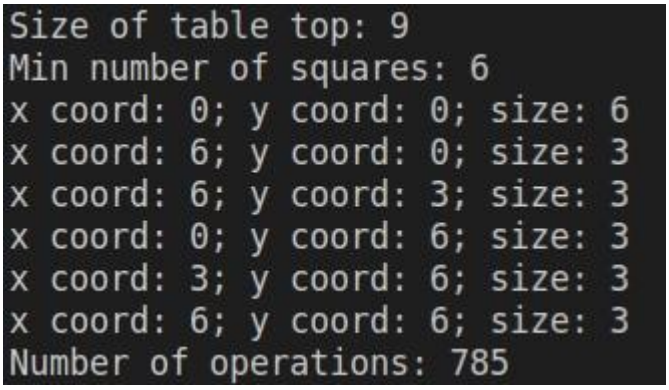

Рисунок 4 – Пример работы программы при N=9

4.

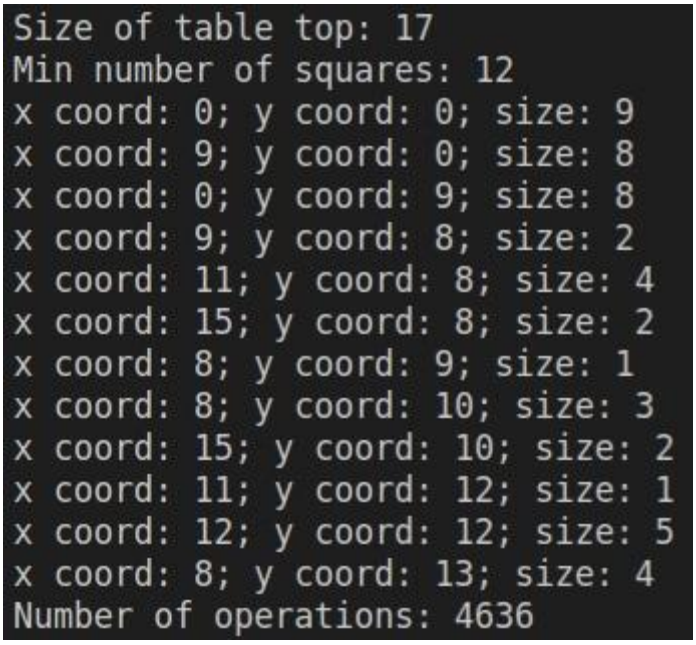

Рисунок 5 – Пример работы программы при N=17

5.

```
Size of table top: 40
Min number of squares: 4
x coord: 0; y coord: 0; size: 20
x coord: 20; y coord: 0; size: 20
x coord: 0; y coord: 20; size: 20
x coord: 20; y coord: 20; size: 20
Number of operations: 17954
```

Рисунок 6 – Пример работы программы при N=40

Исследование количества операций от размера квадрата.

Под операцией подразумевалась вставка квадрата в столешницу. Т.к. случаи со сторонами кратными 2, 3 и 5 являются частью оптимизации – были рассмотрены случае с простыми числами, не кратными этим трем.

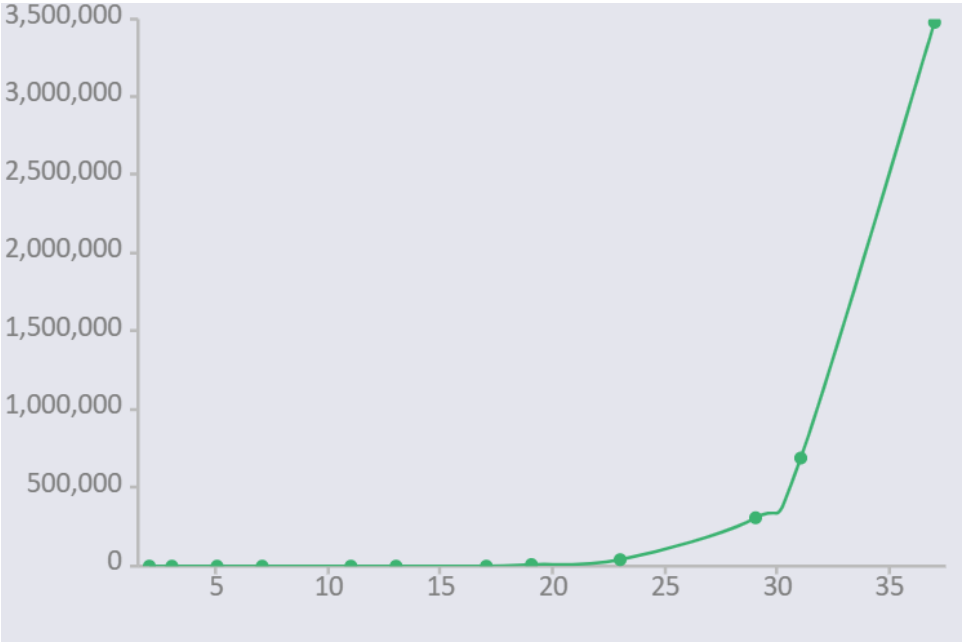

Рисунок 7 – График зависимости числа операций от размера столешницы

Как видно из графика, очевидна экспоненциальный рост количества операций в зависимости от размера стороны столешницы.

Выводы.

В ходе проделанной работы был применён на практике алгоритм поиска с возвратом (бэктрекинг) для решения практической задачи, а именно реализована программа (на языке программирования C++) заполнения поля наименьшим числом квадратов. Было проведено исследование зависимости количества операций от размера квадрата (столешницы), в результате которого был выявлен экспоненциальный рост количества операций в зависимости от размера столешницы. Также были использованы некоторые оптимизации; протестирована корректность работы алгоритма.

ПРИЛОЖЕНИЕ А.
ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл lab1.cpp

```
#include <iostream>
#include <vector>
using namespace std;

int countOperation = 0;  //переменная для подсчета количества операций

struct Square //квадрат
{
    int x, y, size;
};

struct TableTop  //столешница
{
    int N;
    TableTop(int N) : N(N), filled(N, 0) {}
    std::vector<int> filled;   //степень заполненности каждого столбца
построчно
    std::vector<Square> current, minSquares;  //то, с чем работаем
сейчас; минимальный
    int minAmount = 0;  //минимальное количество квадратов

    int getTopColNumber()  //получение координаты левого верхнего столбца
    {
        int minRow = N;
        int minCol = -1;
        for (int i = 0; i < N; i++)
        {
            if (filled[i] < minRow)
            {
                minCol = i;
                minRow = filled[i];
            }
        }
        return minCol;
    }

    int spaceBelow(int col)   //сколько места есть ниже
    {
        return N - filled[col];
    }

    int spaceRight(int col)   //сколько места есть справа
    {
        int endCol = col + 1;
        int row = filled[col];
```

```
        for ( ; endCol < N; endCol++)
        {
            if (filled[endCol] > row)
                break;
        }
        return endCol - col;
    }

    void insertSquare(int topLeftCol, int size)   //вставка квадрата
    {
        countOperation++;
        current.push_back({topLeftCol, filled[topLeftCol], size});
        for (int i = 0; i < size; i++)
        {
            filled[topLeftCol + i] += size;
        }
    }

    void deleteSquare(int topLeftCol, int size)  //удаление квадрата
    {
        for (int i = 0; i < size; i++)
        {
            filled[topLeftCol + i] -= size;
        }
        current.pop_back();
    }

    void saveMinSquares()   //сохранение полученных минимальных значений
    {
        if (current.size() < minAmount)
        {
            minAmount = current.size();
            minSquares = current;
        }
    }
};

void printSquaresData(const std::vector<Square> &current)  //печать
данных по квадратам на экран
{
    cout << "Min number of squares: " << current.size() << '\n';
    for (auto square : current)
    {
        cout << "x coord: " << square.x << "; y coord: " << square.y <<
"; size: " << square.size << '\n';
    }
}

void backtracking(TableTop &table);
```

```
decltype(TableTop::current) analysisN(int N)  //в зависимости от входных
данных выбираем ход действий
{
    TableTop table(N);
    table.minAmount = (N + 3) + 1; // +1 to record squares
    if (N % 2 != 0 && N % 3 != 0 && N % 5 != 0)
    {
        int size = N/2 + 1;
        table.insertSquare(0, size);
        table.insertSquare(size, size - 1);
        table.insertSquare(0, size - 1);
        backtracking(table);
    }
    else
    {
        if (N % 2 == 0) table.minAmount = 4 + 1; // +1 to record squares
        else if (N % 3 == 0) table.minAmount = 6 + 1; // +1 to record
squares
        else if (N % 5 == 0) table.minAmount = 8 + 1; // +1 to record
squares
        backtracking(table);
    }
    return table.minSquares;
}

void backtracking(TableTop &table)   //основаная фукция осуществляющая
рекурсивную вставку
{
    int toppestCol = table.getTopColNumber();
    if (toppestCol == -1)
    {
        table.saveMinSquares();
        return;
    }

    if (table.current.size() >= table.minAmount - 1)
    {
        return;
    }

    int maxSize = std::min(table.spaceBelow(toppestCol),
table.spaceRight(toppestCol));
    maxSize = std::min(maxSize, table.N - 1);

    for (int size = maxSize; size >= 1; size--)
    {
        table.insertSquare(toppestCol, size);
        backtracking(table);
        table.deleteSquare(toppestCol, size);
    }
```

```
}

int main()
{
    int N;
    cout << "Size of table top: ";
    cin >> N;
    auto current = analysisN(N);
    printSquaresData(current);
    cout << "Number of operations: " << countOperation << '\n';
    return 0;
}
```