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: Бэктрекинг

Студентка гр. 9382 _____ Пя С.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург

2021

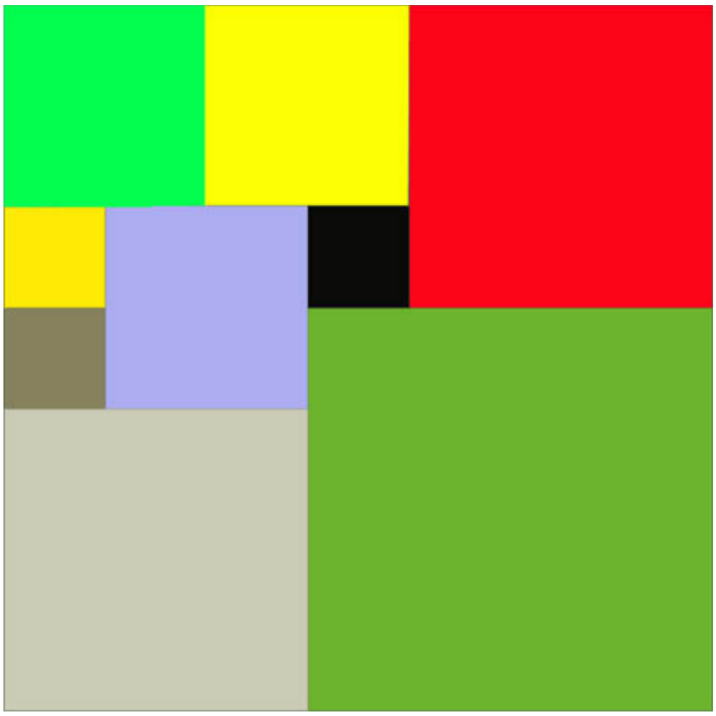
Цель работы.

Реализовать программу, основанную на рекурсивом бэктрекинге. Исследовать время выполнения алгоритма от параметра, прописанного в задании.

Задание.

Вар. 2р. Рекурсивный бэктрекинг. Исследование времени выполнения 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). 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 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 ($1 \leq x, y \leq N$) 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

Алгоритм.

Алгоритм заключается в поиске наименьшей возможной комбинации квадратов разного или одного размера, составляющих квадрат с заданным размером N. Перебираются все возможные варианты размещения квадратов. Для удобства создаются два массива для хранения текущего заполнения квадратами и минимально возможного. Начиная с большего размера размещаем квадраты, находя пустые места, пока это возможно, положив их в массив, переходим к меньшему размеру до единичного. Если заданный квадрат оказывается полностью заполненным, массив квадратов сравнивается с минимальным, если первый меньше, минимальному массиву присваивается текущий. Далее из текущего массива убираются квадраты для перебора всех вариантов.

1. Находим самый большой делитель для числа N, не включая само число, уменьшаем на него число N.

2. Определяем, четное ли число N, для четного разбиение будет на 4 равных квадрата. Для нечетных в массив кладем три квадрата длиной N / 2 + 1, N / 2, N / 2.
3. Проверяем, превысило ли количество квадратов в текущем массиве количество квадратов в минимально возможном, при превышении возвращаемся, если не превысило, продолжаем выполнение.
4. Далее ищем пустое место для размещения квадрата, постепенно уменьшая размер фигуры. При нахождении заполненного минимальным способом массива квадратов он сохраняется как возможно верный и сравнивается с остальными найденными в процессе массивами квадратов.
5. Затем удаляем квадраты и продолжаем поиск, пока длина положительна.
6. В ответе все выводимые значения умножаются на наибольший делитель.

Способ хранения частичных решений.

Для хранения квадратов была использована структура square, содержащая в себе координаты левого верхнего угла квадрата и длину. Для хранения заполнения главного квадрата был реализован массив структур, один, содержащий результат, другой – содержащий временное заполнение квадратами, которые в рекурсивной функции удаляются и прибавляются. При нахождении заполнения меньшего, чем в минимальном массиве, итоговый массив сохраняет текущий массив.

Методы оптимизации.

- Упростили вычисления для квадратов, размеры которых не являются простыми числами.
- Решение делится на два типа: с четными и нечетными размерами. Для четных минимальным количеством содержания квадратов будет 4, для нечетных три квадрата размерами N / 2 + 1, N / 2, N / 2 и остальное количество, вычисляемое программой.

- Сразу отбрасываем варианты, в которых количество квадратов превышают минимальное.

Оценка сложности по памяти и времени.

Сложность алгоритма по памяти – O(N)

N –длина квадрата, максимальное количество данных в массиве N*2 (квадрат размером N – 1 и квадраты размером 1), массива два: N*2*2, также храним другие переменные, не увеличивающие сложность по памяти.

Сложность алгоритма по времени – O((N^2)! * N^N)

Так как в этом алгоритме используются оптимизации, оценку сложности по времени вычислить довольно сложно. N – размер квадрата, всего квадратов может быть в массиве N^2. Рассмотрим расстановку квадратов по очереди, первый можно расставить N^2*N, так как квадраты разного размера, второй можно расставить (N^2 – 1)*N способами, таким образом получаем O((N^2)! * N^N).

Функции и структуры данных.

1. Структура square содержит координаты x, y и размер квадрата length.
2. list<square> minArr используется для хранения решения задачи. list<square> curArr используется для поиска решения задачи. В них используются квадраты, которые составляют заданный.

Функции:

3. showArea

Сигнатура: void showArea(int& N, bool isMin)

Назначение: выводит содержимое массива текущего или минимального, визуализируя его в виде значения размера квадратов.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить, isMin : bool – выбор вывода результата или текущего массива.

Возвращаемое значение: -

4. showAnswer

Сигнатура: void showAnswer(int& N, int& k)

Назначение: выводит количество квадратов и их координаты с длиной.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить, k : int – наибольший делитель числа N.

Возвращаемое значение: -

5. checkPossibilityToPlace

Сигнатура: bool checkPossibilityToPlace(int& N, int& x, int& y, int& length)

Назначение: проверяет возможность размещения квадрата с заданными параметрами.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить, x : int, y : int – координаты левого верхнего угла квадрата, length : int – размер квадрата.

Возвращаемое значение: bool: true – если квадрат можно разместить, false – если нет.

6. defineBase

Сигнатура: void defineBase(int& N)

Назначение: оптимизирует поиск, определив начальное заполнение квадрата.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить.

Возвращаемое значение: -

7. findEmptyPlace

Сигнатура: bool findEmptyPlace(int& x, int& y, int& N)

Назначение: находит пустое место, меняя координаты на найденные.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить, x : int, y : int – координаты левого верхнего угла квадрата.

Возвращаемое значение: bool – true, если место найдено, false – в противном случае.

8. calculateOptimization

Сигнатура: void calculateOptimization(int& N, int x, int y)

Назначение: находит минимальное возможное количество квадратов, заполняющих собой заданный.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить, x : int, y : int – координаты левого верхнего угла квадрата.

Возвращаемое значение: -

9. findMin

Сигнатура: void findMin(int& N)

Назначение: определяет наибольший общий делитель числа N, вызывает функции поиска минимального массива и выводит его.

Описание аргументов: N : int – размер главного квадрата, который нужно заполнить.

Возвращаемое значение: -

Исследование.

Необходимо исследовать зависимость времени от размера квадрата. Для этого посчитаем время выполнения алгоритма для каждого размера квадрата в пределах заданных границ.

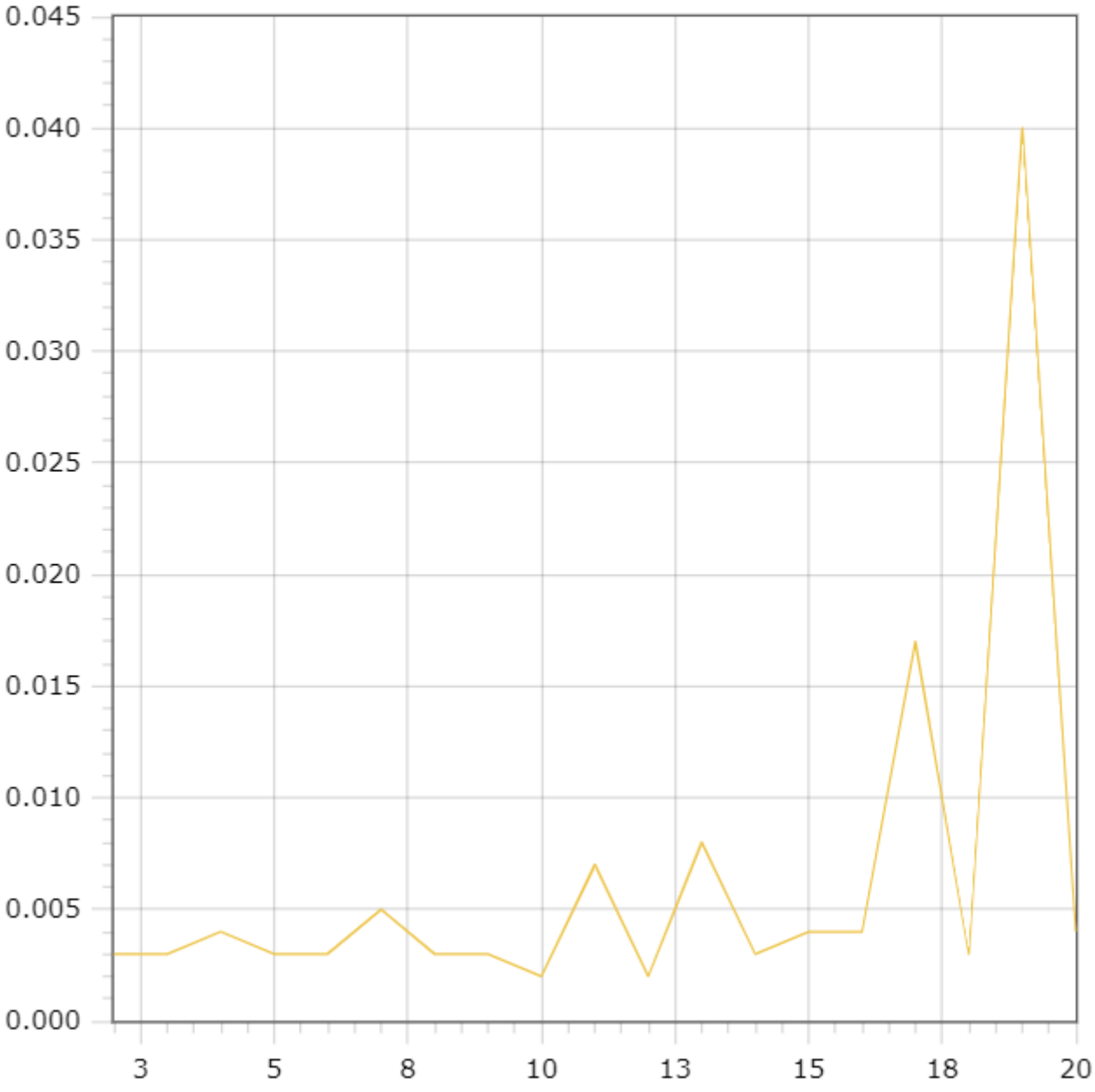
Результаты времени выполнения алгоритма от размера главного квадрата представлены в Таблице 2.

Таблица 2. Зависимость времени от размера квадрата

Размер квадрата N	**Время выполнения алгоритма**
2	0.003
3	0.003
4	0.004
5	0.003
6	0.003
7	0.005
8	0.003
9	0.003
10	0.002
11	0.007
12	0.002

13	0.008
14	0.003
15	0.004
16	0.004
17	0.017
18	0.003
19	0.04
20	0.004

График 1. Зависимость времени выполнения алгоритма от размера квадрата



Из графика можно заметить, что при четных значениях график опускается резко вниз, это происходит из-за оптимизаций. При нечетных значениях график растет экспонентциально.

Выводы.

В ходе работы были изучены методы бэктрекинга, написана программа для поиска минимального количества квадратов для заполнения заданного с помощью рекурсивного бэктрекинга, практически освоены решения по возможным оптимизациям и исследована зависимость времени выполнения алгоритма от размера квадрата.

Тестирование.

Протестировано с помощью тестировочной системы на Stepik. Также результаты можно посмотреть в Таблице 1.

Таблица 1. Результаты работы программы

№ п/п	Входные данные	Выходные данные
1	2	4 1 1 1 1 2 1 2 1 1 2 2 1
2	3	6 1 1 2 1 3 1 3 1 1 2 3 1 3 2 1 3 3 1
3	8	4 1 1 4 1 5 4 5 1 4 5 5 4
4	9	6 1 1 6 1 7 3 7 1 3 4 7 3 7 4 3 7 7 3
5	5	8

		1 1 3 1 4 2 4 1 2 3 4 2 4 3 1 5 3 1 5 4 1 5 5 1
6	19	13 1 1 10 1 11 9 11 1 9 10 11 3 10 14 6 11 10 1 12 10 1 13 10 4 16 14 1 16 15 1 16 16 4 17 10 3 17 13 3
7	20	4 1 1 10 1 11 10 11 1 10 11 11 10

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <list>
#include <ctime>

struct square {
    int x, y, length;
    square(int x, int y, int length) {
        this->x = x;
        this->y = y;
        this->length = length;
    }
};

std::list <square> minArr;
std::list <square> curArr;

void showArea(int& N, bool isMin) {
    std::list<square> &viewArr = (isMin) ? minArr : curArr;
    int area[N][N];
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            area[i][j] = 0;
        }
    }
    int n = 1;
    for (auto it : viewArr) {
        for (int j = it.y; j < it.y + it.length; j++) {
            for (int i = it.x; i < it.x + it.length; i++) {
                area[j][i] = n;
            }
        }
        n++;
    }
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            std::cout << area[i][j] << " ";
        }
        std::cout << "\n";
    }
}

void showAnswer(int& N, int& k) {
    std::cout << minArr.size() << "\n";
    for (auto i : minArr) {
        std::cout << i.x * k + 1 << " " << i.y * k + 1 << " " << i.length * k <<
"\n";
    }
}

bool checkPossibilityToPlace(int& N, int& x, int& y, int& length) {
    if (x + length > N || y + length > N)
        return false;
    for (auto it : curArr) {
        if ((it.x < (x + length) && (it.x + it.length) > x) && (it.y < (y +
length) && (it.y + it.length) > y)) {
            return false;
        }
```

```
    }
    return true;
}

void defineBase(int& N) {
    int halfN = N / 2;
    if (N % 2 == 0) {
        curArr.emplace_back(0, 0, halfN);
        curArr.emplace_back(0, halfN, halfN);
        curArr.emplace_back(halfN, 0, halfN);
        curArr.emplace_back(halfN, halfN, halfN);
        minArr = curArr;
    } else {
        curArr.emplace_back(0, 0, halfN + 1);
        curArr.emplace_back(0, halfN + 1, halfN);
        curArr.emplace_back(halfN + 1, 0, halfN);
    }
}

bool findEmptyPlace(int& x, int& y, int& N) {
    int l = 1;
    while (!checkPossibilityToPlace(N, x,y, l)) {
        if (y == N - 1) {
            if (x == N - 1) {
                return false;
            }
            else {
                x++;
                y = N / 2;
                continue;
            }
        }
        y++;
    }
    return true;
}

void calculateOptimization(int& N, int x, int y) {
    if (curArr.size() >= minArr.size() && !minArr.empty())
        return;
    for (int length = N / 2; length > 0; --length) {
        if (checkPossibilityToPlace(N, x,y, length)) {
            curArr.emplace_back(x, y, length);
            int tX = x, tY = y;
            if (findEmptyPlace(tX, tY, N)) {
                calculateOptimization(N, tX, tY);
            } else {
                if (curArr.size() < minArr.size() || minArr.empty()) {
                    minArr = curArr;
//                    showArea(N, true);
                }
                curArr.pop_back();
                return;
            }
            curArr.pop_back();
        }
    }
}

void findMin(int& N) {
    int size = N, k = 0;
    for (auto i = size; i > 0; --i)
    {
```

```
        if (N % i == 0 && N != i)
        {
            N /= i;
            k = i;
            break;
        }
    }
    defineBase(N);
    calculateOptimization(N,N /2,N / 2 + 1);
    showAnswer(N, k);
}

int main() {
    int N = 0;
    std::cin >> N;
    clock_t start = clock();
    findMin(N);
    clock_t end = clock();
    std::cout << (double)(end - start) / CLOCKS_PER_SEC;
    return 0;
}
```