МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра математического обеспечения и применения ЭВМ

ОТЧЕТ

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

Студент гр. 8382 _____ Нечепуренко Н.А.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2020

Цель работы.

Разработать программу с итеративным алгоритмом поиска с возвратом решения задачи о квадрировании квадрата. Проанализировать сложность полученного решения.

Постановка задачи.

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 (см. рис 1).

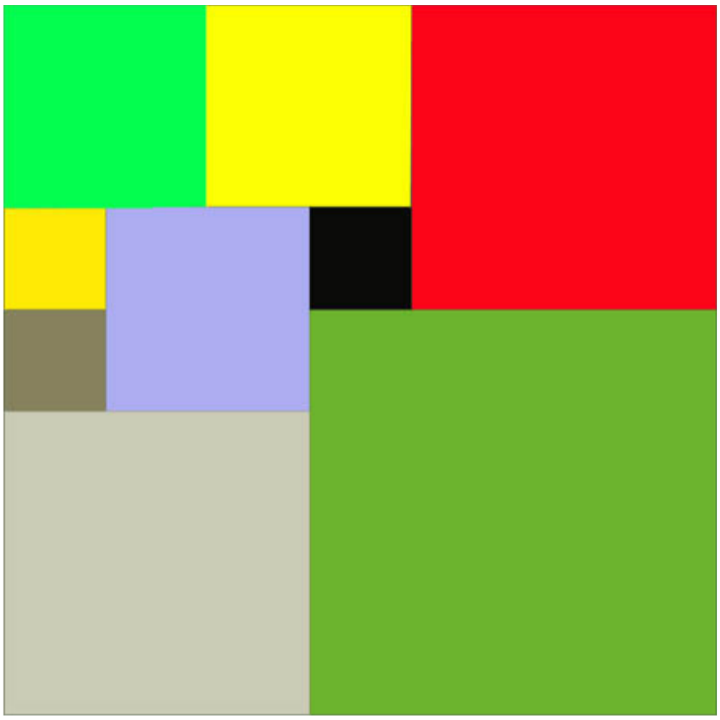

Рисунок 1 – Пример разбиения квадрата со стороной семь.

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≤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≤x, y≤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

Индивидуальное задание: вар. 1и. Итеративный бэктрекинг. Выполнение на Stepik двух заданий в разделе 2.

Описание алгоритма.

Опишем кратко алгоритм. Будем перебирать текущий размер квадрата, который хотим поместить в исходный квадрат, пытаясь найти свободное место (пару индексов матрицы, представляющей исходный квадрат). Под свободным местом понимаем, что квадрат не выходит за пределы исходного квадрата и не пересекается с уже заполненными квадратами. Ставим квадрат и повторяем

снова перебор квадратов по текущей размерности, пытаясь найти свободное место. Когда весь квадрат будет заполнен или будет выполнено одно из условий (см. ниже), либо проверяем, заполнен ли квадрат полностью и обновляем лучшее решение и убираем квадрат, либо просто убираем последний поставленный квадрат.

Можно заметить, что описанный алгоритм имеет рекурсивную природу, но, так как индивидуальный вариант предполагает итеративную реализацию, придется воспользоваться структурой данных стек. Будем использовать встроенный контейнер *std::stack*, реализующий необходимый функционал.

Оценка сложности алгоритма ниже, в соответствующем разделе.

Реализация алгоритма.

Приведем используемые в реализации алгоритма оптимизации.

1. В любом наилучшем решении всегда есть квадрат со стороной n пополам, округленный в большую сторону, и два квадрата n пополам, округленные в меньшую сторону.
2. Если число имеет делители 2, 3, 5, то строим решение для минимального делителя и масштабируем его.
3. Если на определенной итерации алгоритма имеется больше квадратов, чем в наилучшем решении – прекращаем поиск.
4. Для определения того, заполнен исходный квадрат или нет, будем поддерживать числовую переменную площади, и проверять ее на равенство n^2.
5. Определять, пересекаются ли два квадрата, будем за константное время в функции *noIntersections* (см. ниже).

Хранить информацию о квадратах будем в упорядоченной тройке целых чисел с помощью *std::tuple*. Будем называть этот тип *luPoint*:

```
typedef tuple<int, int, int> luPoint;
```

Проверка пересечений квадратов реализована в функции *noIntersections*:

```
bool noIntersections(const luPoint& lu1, const luPoint& lu2)
```

Логическое значение вычисляется по относительному расположению левых верхних углов квадратов с учетом размера с помощью условных конструкций.

С помощью этой функции будем проверять, пересекается ли квадрат кандидат с уже имеющими квадратами, в функции *pass*:

```
bool pass(const vector<luPoint>& v, const luPoint& p)
```

Для хранения текущего решения и ответа воспользуемся контейнером std::vector:

```
vector<luPoint> currentSolution;
```

Добавим в него три первых квадрата, описанных в оптимизации. Начнем перебирать размер квадрата:

```
for (int size = n >> 1; size > 0; size--)
```

Далее будем поддерживать стек состояний итерации в переменной *rStack*:

```
stack<tuple<int,int,int>> rStack;
```

Состояние характеризуется тройкой чисел: текущий размер квадрата-кандидата, был ли он уже поставлен и с какого размера требуется перебирать следующий квадрат. Добавляем начальное состояние *size*, 0, n пополам.

Пока стек не пуст, выполняем алгоритм описанный выше.

Полный исходный код с комментариями находится в Приложении А.

Тестирование программы.

Проведем тестирование программы с фиксацией времени выполнения (см. табл. 1). Соберем программу с флагом O4. Тестировать будем на простых числах, т. к. составные сводятся к ним.

Таблица 1 – Тестирование программы.

Входные данные	**Выходные данные**	**Время работы, мс**
3	6 1 1 2 1 3 1 3 1 1	0

	2 3 1 3 2 1 3 3 1	
5	8 1 1 3 1 4 2 4 1 2 3 4 1 3 5 1 4 3 1 4 4 2 5 3 1	0
7	9 1 1 4 1 5 3 5 1 3 4 5 1 4 6 2 5 4 1 5 5 1 6 4 2 6 6 2	0
11	11 1 1 6 1 7 5 7 1 5 6 7 1 6 8 1 6 9 3 7 6 1 7 7 2 8 6 1 9 6 3 9 9 3	1
13	11 1 1 7 1 8 6 8 1 6 7 8 1 7 9 3 7 12 2 8 7 2 9 12 2 10 7 4	5

	10 11 1 11 11 3	
17	12 1 1 9 1 10 8 10 1 8 9 10 1 9 11 3 9 14 4 10 9 2 12 9 2 12 11 2 12 13 1 13 13 5 14 9 4	53
19	13 1 1 10 1 11 9 11 1 9 14 14 6 16 10 4 10 16 4 10 11 1 10 12 2 10 14 2 11 10 1 11 11 1 12 10 4 12 14 2	1*
23	13 1 1 12 1 13 11 13 1 11 12 13 1 12 14 3 12 17 7 13 12 2 15 12 5 19 17 2 19 19 5 20 12 4 20 16 1 21 16 3	1382
29	14	60*

	1 1 15 1 16 14 16 1 14 22 22 8 23 15 7 15 23 7 15 16 1 15 17 3 15 20 3 16 15 2 18 15 5 18 20 3 21 20 2 21 22 1	
31	15 1 1 16 1 17 15 17 1 15 16 17 1 16 18 1 16 19 4 16 23 3 16 26 6 17 16 3 19 23 3 20 16 6 20 22 1 21 22 1 22 22 10 26 16 6	46273

* – была применена соответствующая оптимизация для простых чисел, оканчивающихся на девять.

Оценка сложности алгоритма.

Оценим затраты по памяти. Текущее решение не может содержать больше, чем 2n квадратов (один квадрат размера n-1 и 2n-1 квадратов размера 1). Стек растет только при добавлении квадрата, т. е. тоже не превышает 2n по памяти. Итого получаем O(n) по памяти. По скорости все гораздо сложнее. Поиск места, для вставки выполняется за n^3, так как сначала ищется пара

индексов, а затем для каждого квадрата (их не более 2n) производится проверка на пересечение. Это все происходит столько раз, сколько мы делаем итераций, т.е. много меньше, чем n^n, точное количество сложно подсчитать теоретически, примем за e^n. Итого получаем $O(n^3e^n)$ по скорости.

Вывод.

В результате выполнения работы была разработана программа, выполняющая квадрирование квадрата за приемлемое время. Была проанализирована скорость выполнения программы на простых числах и дана грубая оценка асимптотике алгоритма.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ.

```
#include <bits/stdc++.h>
using namespace std;

typedef tuple<int, int, int> luPoint; // Тип для хранения верхнего левого угла (
y, x, size)

bool noIntersections(const luPoint& lu1, const luPoint& lu2){
    //Проверяем, пересекаются ли два квадрата, заданные двумя точками.
    //auto [y1, x1, n1] = lu1;
    //auto [y2, x2, n2] = lu2;
     int y1 = std::get<0>(lu1);
     int x1 = std::get<1>(lu1);
     int n1 = std::get<2>(lu1);
     int y2 = std::get<0>(lu2); // for stepik
     int x2 = std::get<1>(lu2);
     int n2 = std::get<2>(lu2);
    if (x1 == x2 && y1 == y2) return false;
    if (x1 == x2) {
        return (y1 > y2) ? \
        abs(y1-y2) >= n2 : abs(y1-y2) >= n1;
    }
    if (y1 == y2) {
        return abs(x1-x2) >= (x1 < x2 ? n1 : n2);
    }
    if (x1 < x2){
        if (y1 < y2){
            return abs(x1-x2) >= n1 || abs(y1-y2) >= n1;
        }
        return abs(x1-x2) >= n1 || abs(y1-y2) >= n2;
    } else {
```

```
        if (y1 < y2){
            return abs(x1-x2) >= n2 || abs(y1-y2) >= n1;
        }
        return abs(x1-x2) >= n2 || abs(y1-y2) >= n2;
    }
}

bool pass(const vector<luPoint>& v, const luPoint& p){
    //Проверяем пересечение точки кандидата с уже имеющимися точками в решении.
    bool res = true;
    for (auto el : v){
        res &= noIntersections(el, p);
    }
    return res;
}

int main(){
    int n = 0;
    cin >> n;
    auto start = std::chrono::steady_clock::now();


    // Находим наименьший делитель числа n
    int currentDivisor = 2;
    int divisor = 1;
    while (currentDivisor*currentDivisor <= n) {
        if (n % currentDivisor == 0){
            divisor = currentDivisor;
            break;
        }
        currentDivisor++;
    }

    int scalar = 1;
    if (divisor != 1) {
        scalar = n / divisor;
        n = divisor;
    }

    // Самый тривиальный случай -- просто выводим ответ
    /*

    _______
    |  |  |
    -------
    |  |  |
    -------

    */
    if (n == 2){
        cout << 4 << endl;
```

```
        cout << 1 << " " << 1 << " " << (scalar*n>>1) << endl;
        cout << 1+(scalar*n>>1) << " " << 1 << " " << (scalar*n>>1) << endl;
        cout << 1 << " " << 1+(scalar*n>>1) << " " << (scalar*n>>1) << endl;
        cout << 1+(scalar*n>>1) << " " << 1+(scalar*n>>1) << " " << (scalar*n>>1
) << endl;
        return 0;
    }

    // Добавляем 3 квадрата, уменьшая общее пространство решений. Эмпирическим п
утем было выяснено,
    // что в наилучшем решении есть эти три квадрата
    vector<luPoint> currentSolution;
    currentSolution.push_back({0, 0, (n >> 1) + 1});
    currentSolution.push_back({0, (n >> 1) + 1, n >> 1});
    currentSolution.push_back({(n >> 1) + 1, 0, n >> 1});

    if (divisor == 1 && n % 10 == 9) {
        // Если число простое и с девяткой на конце, то можно добавить еще три к
вадрата, тем самым ускорив
        // выполнение программы
        int anotherBigSize = ((n - (n >> 1)) >> 1) + 1;
        int smallerSize = n - anotherBigSize - (n>>1);
        currentSolution.push_back({n - anotherBigSize, n -
 anotherBigSize, anotherBigSize});
        currentSolution.push_back({n - smallerSize, n - smallerSize -
 anotherBigSize, smallerSize});
        currentSolution.push_back({n - smallerSize - anotherBigSize, n -
 smallerSize, smallerSize});
    }

    int maxCount = 2*n;
    vector<luPoint> answer;

    int currentSquare = 0;
    for (auto element : currentSolution){
        //auto [y, x, size] = element;
        int y = std::get<0>(element);
        int x = std::get<1>(element); // for stepik
        int size = std::get<2>(element);
        currentSquare += size*size;
    }

    for (int size = n >> 1; size > 0; size--){
        // Перебираем по размеру, пытаясь начать с квадрата размером size
        stack<tuple<int,int,int>> rStack; // эмуляция рекурсивного поведения
        rStack.push({size, 0, n >> 1});
        do {

            tuple<int,int,int> stack_top = rStack.top();
            //auto [cursz, visited, itsz] = stack_top;
```

```
            int cursz = std::get<0>(stack_top);
            int visited = std::get<1>(stack_top); // for stepik
            int itsz = std::get<2>(stack_top);
            if ((int)currentSolution.size() == maxCount && n*n -
 currentSquare > cursz*cursz) {
                // Если неполное решение уже нелучшее
                rStack.pop();
                continue;
            }
            auto flag = false;
            for (int i = n >> 1; i < n && !visited; i++){
                for (int j = n >> 1; j < n && !visited; j++){
                    if (pass(currentSolution, {i, j, 1})){ // Пустая клетка
                        if (i + cursz <= n && j + cursz <= n && pass(currentSolu
tion, {i, j, cursz})){
                            // Квадрат кандидат помещается
                            currentSolution.push_back({i, j, cursz});
                            visited = 1; // Если в этой итерации уже ставили, то
 больше не ставим
                            currentSquare += cursz*cursz;
                            rStack.pop();
                            rStack.push({cursz, visited, itsz});
                        } else {
                            flag = true;
                            break;
                        }
                    }

                }
                if (flag) break;
            }

            if (flag) {
                // Есть пустая клетка, мы в нее не поставили, нужно продолжать п
еребор в родительском вызове.
                rStack.pop();
                continue;
            }

            if (currentSquare == n*n && (int) currentSolution.size() <= maxCount
){
                // Нашли решение
                maxCount = (int) currentSolution.size();
                answer = currentSolution;
                auto point = currentSolution.back();
                auto topPointSize = std::get<2>(point);
                currentSquare -= topPointSize * topPointSize;
                currentSolution.pop_back();
                rStack.pop();
                continue;
```

```
            }

            if (maxCount == (int) currentSolution.size()){
                // Нет необходимой площади, уже нелучшее решение
                auto point = currentSolution.back();
                auto topPointSize = std::get<2>(point);
                currentSquare -= topPointSize * topPointSize;
                currentSolution.pop_back();
                rStack.pop();
                continue;
            }

            auto cont = false;
            for (int i = itsz; i > 0; i--){
                // Вызываем перебор для размеров меньше
                if (n*n - currentSquare >= i*i){
                    rStack.pop();
                    rStack.push({cursz, visited, i-1});
                    rStack.push({i, 0, n >> 1});
                    cont = true;
                    break;
                }
            }
            if (cont) continue;

            // Удаляем поставленный квадрат
            rStack.pop();
            auto p = currentSolution.back();
            auto point = currentSolution.back();
            auto topPointSize = std::get<2>(point);
            currentSquare -= topPointSize * topPointSize;
            currentSolution.pop_back();
        } while (!rStack.empty());
    }

    // Выводим ответ
    cout << (int) answer.size() << endl;
    for (auto element : answer) {
        //auto [y, x, size] = element;
        int y = std::get<0>(element);
        int x = std::get<1>(element); // for stepik
        int size = std::get<2>(element);
        cout << scalar*y + 1 << " " << scalar*x + 1 << " " << scalar*size << end
l;
    }
    auto end = std::chrono::steady_clock::now();
    std::cout <<std::chrono::duration_cast<std::chrono::milliseconds>(end -
 start).count() << "ms";
    return 0;
}
```