

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. 3342

Иванов Д. М.

Преподаватель

Виноградова Е. В.

Санкт-Петербург

2025

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.

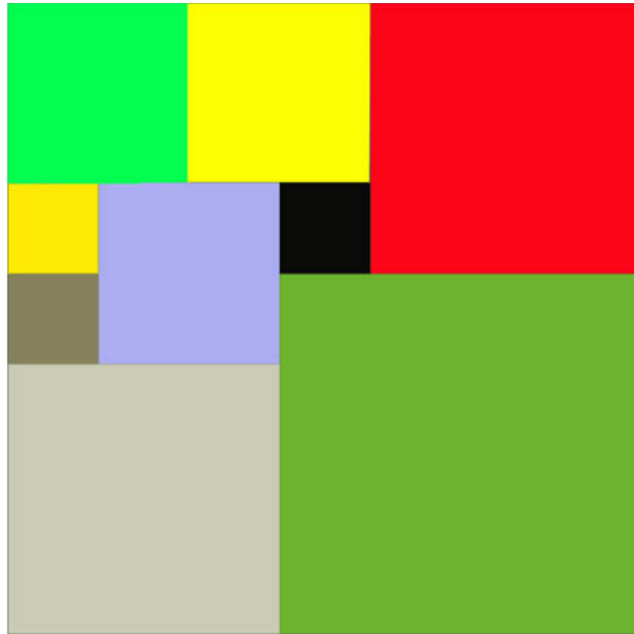


Рисунок 1 – пример расположения квадратов для тестового случая

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 ($2 \leq N \leq 20$).

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

Индивидуальный вариант: 2и. Итеративный бэктрекинг. Исследование времени выполнения от размера квадрата.

Описание алгоритма

Основная идея алгоритма заключается в использовании итеративного бэктрекинга (поиска с возвратом), который позволяет перебирать все возможные варианты заполнения столешницы квадратами, начиная с самых больших и постепенно уменьшая их размер.

В данном переборе используется стек для хранения состояний и их обхода. Каждое состояние представляет из себя вариант заполнения столешницы в виде матрицы клеток. Алгоритм извлекает состояние из стека, обрабатывает его и добавляет в стек новые состояния, которые могут быть получены из текущего. Он находит некоторую пустую клетку и добавляет в нее все возможные квадраты разных размеров. Как только столешница становится заполненной, программа удаляет из стека данное состояние, тем самым делая возврат, и идет рассматривать следующие. Такой обход идет до того момента, пока стек не опустеет. По итогу, программа возвращает решение, которое содержало меньше всего квадратов (во время обхода это хранится в отдельной переменной).

Сложность по времени: $O(n^{(n^2 + 2)})$

Создание сетки столешницы – $O(n^2)$.

Основной цикл – $O(n^{(n^2)})$. В худшем случае каждое состояние может порождать $O(n)$ новых состояний (для каждого возможного размера квадрата). Максимальное количество квадратов (количество шагов) может достигать $O(n^2)$ (если каждый квадрат имеет размер 1×1 и нужно заполнить n^2 клеток).

Вспомогательные функции (нахождение новой клетки, размещение квадрата, проверка возможности размещения квадрата) – $O(n^2)$ каждая.

Общая сложность: для каждого шага основного цикла применяются вспомогательные функции, поэтому получаем: $O(n^{(n^2)} * n^2) = O(n^{(n^2 + 2)})$.

Сложность по памяти: $O(n^{(n^2)})$

Матрица $n \times n$ занимает $O(n^2)$ места. В худшем случае стек может содержать все возможные состояния столешницы. Количество состояний

зависит от количества возможных комбинаций квадратов, которые можно разместить на столешнице. В каждую пустую клетку можно вставить одно из k значений. Так как всего n^2 клеток, то общее количество возможных состояний столешницы равно $k^{(n^2)}$. Итоговая сложность - $O(n^{(n^2)})$.

Сложность получилась очень большая. Однако мы добавили некоторые оптимизации, которые позволяют избежать эти худшие случаи и иметь заполненную столешницу уже на первых шагах алгоритма.

Оптимизация алгоритма

Данный полный перебор будет неэффективным. Есть возможность оптимизировать программу, чтобы она не рассматривала некоторые заведомо невыгодные состояния. Так что были добавлены следующие оптимизации алгоритма:

1) Если текущее состояние во время своего заполнения и обработки имеет больше квадратов чем лучшее решение, оно отбрасывается, и алгоритм переходит к следующему состоянию.

2) Очевидно, что для использования минимального количества квадратов нужно брать квадраты большого размера. Так что изначально можно поставить в левом верхнем углу квадрат размера от $n/2$ до $n-1$, который будет являться самым большим для данной столешницы. Также справа и снизу от него будут некоторые свободные клетки. На них также выгоднее всего расположить квадраты максимально возможного размера. В итоге получим уже заполненную верхнюю левую часть столешницы. Для остальных клеток будет использован описанный выше перебор.

3) Для нахождения следующей пустой клетки используется не построчный обход, одновременно и по строкам, и по столбцам: симметрично относительно диагонали. Таким образом, мы быстрее находим пространство для вставки квадрата большего размера.

Способ хранения частичных решений

По алгоритму бэктрекинга необходимо иметь возможность делать откаты к предыдущим частичным решениям, когда программе не удастся на очередном шаге провести расширение (столешница полностью заполнена). Для этого нужно каким-то образом хранить частичные решения и обрабатывать их. Для итеративного варианта алгоритма лучше всего подойдет использование стека.

Частичные решения хранятся в стеке, который представляет собой вектор элементов типа `StackElement`. Каждый элемент стека (`StackElement`) содержит информацию о текущем состоянии задачи: значение каждой клетки столешницы на данном этапе решения, множество использованных квадратов, следующая пустая клетка в матрице (см. подробнее “Описание функций и структур данных”).

Новые состояния добавляются в конец стека. Программа берет последнее добавленное и начинает работать с ним. Оно сначала удаляется из стека командой `pop`. Если в данной матрице есть свободные клетки, данное состояние расширяется путем добавления в сохраненные координаты столешницы всех возможных квадратов, и новые состояния сохраняются в стек. Если же столешница оказалась полностью заполненной, новые состояния не добавляются, и цикл начинается с начала. То есть мы откинули предыдущий вариант решения и перешли к следующему, хранящемуся в стеке.

Использование стека позволяет эффективно управлять частичными решениями и перебирать их в порядке, который минимизирует количество шагов для нахождения оптимального решения.

Описание функций и структур данных

Прежде всего необходимо создать класс, из объектов которого будет состоять наш список. Класс Node будет иметь следующие поля:

struct StackElement - Структура, описывающая элементы, которые будут храниться в стеке для перебора всех возможных состояний и выполнения отката. Содержит следующие поля: матрица, описывающая состояние каждой клетки столешницы (grid); вектор использованных квадратов (координаты и размер) (squaresUsed); пара чисел, хранящая следующие пустые клетки на столешнице (emptyCell).

void printMatrix(const std::vector<std::vector<int>>& grid) – Функция для построчного вывода состояния клеток переданной матрицы через std::cout. Передается константная ссылка на матрицу столешницы (grid).

bool isValid(int x, int y, int w, int n, const std::vector<std::vector<int>>& grid) - Функция, которая проверяет возможность поставить на координаты (x, y) квадрат размера w. Передаются в качестве аргументов параметры квадрата (x, y, w), размер столешницы (n) и константная ссылка на матрицу столешницы (grid). Через цикл идет проверка, что каждая клетка, на которую будет ставиться квадрат, окажется пустой. В итоге возвращается соответствующее булево значение.

void placeSquare(int x, int y, int w, std::vector<std::vector<int>>& grid, int value) - Функция для вставки на соответствующие координаты столешницы квадрата размера w. Передаются в качестве аргументов параметры квадрата (x, y, w) и ссылка на матрицу столешницы (grid), чтобы можно было изменить состояние ее клеток внутри функции. А также в качестве аргумента выступает номер добавленного квадрата в данной ветке состояний (value). Через двойной цикл идет проход по соответствующим клеткам и их заполнение необходимым значением value.

std::pair<int, int> findEmptyCell(const std::vector<std::vector<int>>& grid, int n) - Функция для нахождения по определенному алгоритму пустой клетки. Передаются матрица столешницы (grid) и ее размер (n). В функции необходимо

прописать оптимизированный поиск клетки. Так, для начала смотрятся границы столешницы ($x = 0$ или $y = 0$). Так как там выгоднее всего расположить квадрат сразу максимально возможного размера. Если там не нашлось места – идет поиск в оставшейся части квадрата двойным циклом. При этом идет обход сразу и по строкам, и по столбцам, как описано было выше. Если координата свободная нашлась – возвращаются значения x, y через `std::pair`. Если вся столешница заполнена – на выходе получается `{-1, -1}`.

```
void newStackForState(int x, int y, int w, const
std::vector<std::vector<int>>& grid, const std::vector<std::tuple<int, int, int>>&
squaresUsed, std::vector<StackElement>& stack, int n) -
```

 Функция для добавления нового состояния в стек. В качестве аргументов передаются параметры нового квадрата (x, y, w), константная ссылка на матрицу столешницы (`grid`), константная ссылка на вектор использованных квадратов (`squaresUsed`), ссылка на стек состояний обхода (`stack`) и размер столешницы (n). В начале копируется в новую переменную состояние столешницы и в нее добавляется новый квадрат под номером `squaresUsed.size() + 1`. По аналогии копируется `squaresUsed` и добавляется туда квадрат в виде его параметров: координаты и размер. После этого находится следующая пустая клетка, и все эти 3 переменные добавляются в стек в виде нового состояния, которое продолжит рассматривать алгоритм на следующем шаге до полного заполнения столешницы.

```
std::vector<std::tuple<int, int, int>> solve(int n) -
```

 Функция с основным алгоритмом решения в виде итеративного бэктрекинга. Передается размер столешницы (n). В начале идет инициализация основных переменных.

```
std::vector<std::vector<int>> grid
```

 – матрица, хранящая состояние каждой клетки столешницы. Изначально заполняется нулями и представляется в виде вложенного вектора размера $n*n$.

```
std::vector<StackElement> stack
```

 – стек, представленный в виде вектора, для добавления в него новых состояний обхода и обработки последних добавленных. Пока в данной ветке состояний можно добавлять элементы,

алгоритм работает с ней. Как только столешница становится заполненной, алгоритм переходит к следующему состоянию.

`std::vector<std::tuple<int, int, int>>` `bestSolution` – вектор, хранящий множество квадратов, из которых наилучшим образом можно заполнить столешницу. Каждый элемент вектора содержит множества, хранящие координаты квадрата и его размер. В процессе алгоритма эта переменная будет изменяться новыми оптимальными вариантами заполнения, ее в итоге алгоритм и будет возвращать.

В начале самого алгоритма в стек добавляются состояния, хранящие максимально возможные квадраты размером от $n/2$ до $n - 1$. Далее запускается цикл, который будет работать до опустошения стека. Берется последнее добавленное состояние.

Идет проверка на завершение работы с данным состоянием: если в нем уже квадратов больше, чем в сохраненном ранее лучшем решении (`bestSolution`) или столешница полностью заполнена (при этом данное решение сравнивается с лучшим, и, в случае необходимости, оно обновляется) – данная ветка перебора обрывается и идет в следующей итерации работа с новым состоянием.

Если прерывания не было, алгоритм идет дальше по данной итерации. Если найденные свободные клетки находятся на границах столешницы ($x == 0$ || $y == 0$), идет добавление квадрата сразу максимально возможного размера в этой части матрицы для оптимизации. Данное состояние добавляется в стек.

В противном случае бэктрекинг переходит к своему обычному перебору всех возможных значений. Перебираются все возможные размеры квадрата (от 1 до $\min(\{n - x, n - y, n - 1\})$), левый верхний угол которого может располагаться на данной пустой клетке. Если данный квадрат можно расположить, он добавляется в матрицу, а новое состояние – в стек.

После этого стек берет последнее добавленное состояние и процесс повторяется. Когда стек опустеет, это будет означать, что мы рассмотрели все возможные варианты. Возвращаем `bestSolution`. Далее в `main` мы

пробегаемся по этому множеству и выводим построчно параметры каждого квадрата.

Тестирование

Результаты тестирования представлены в табл. 1.

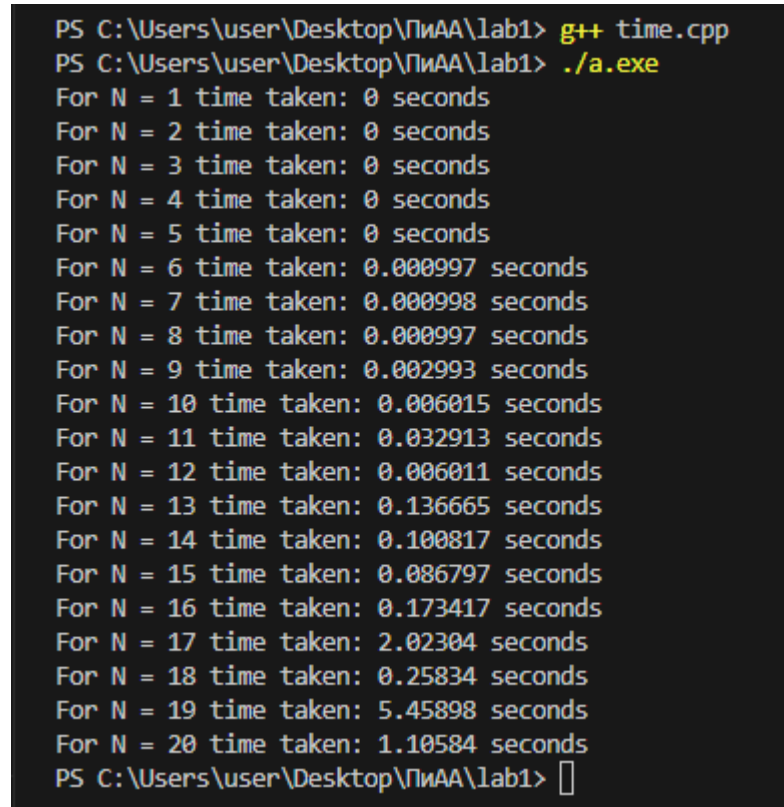
Таблица 1 – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	7	9 1 1 4 5 1 3 1 5 3 4 5 2 5 4 1 6 4 2 4 7 1 5 7 1 6 6 2	Верный вывод
2.	3	6 1 1 2 3 1 1 1 3 1 2 3 1 3 2 1 3 3 1	Верный вывод
3.	12	4 1 1 6 7 1 6 1 7 6	Верный вывод
4.	9	6 1 1 6 7 1 3 1 7 3 4 7 3	Верный вывод

		7 4 3	
		7 7 3	

Исследование времени выполнения от размера квадрата

Для исследования алгоритма по времени выполнения от размера квадрата запустим программу для n от 1 до 20. Также замерим время, через которое наша функция возвращает результат, с помощью модуля `<chrono>`. Получили результаты, представленные в рис 2.



```
PS C:\Users\user\Desktop\ПиАА\lab1> g++ time.cpp
PS C:\Users\user\Desktop\ПиАА\lab1> ./a.exe
For N = 1 time taken: 0 seconds
For N = 2 time taken: 0 seconds
For N = 3 time taken: 0 seconds
For N = 4 time taken: 0 seconds
For N = 5 time taken: 0 seconds
For N = 6 time taken: 0.000997 seconds
For N = 7 time taken: 0.000998 seconds
For N = 8 time taken: 0.000997 seconds
For N = 9 time taken: 0.002993 seconds
For N = 10 time taken: 0.006015 seconds
For N = 11 time taken: 0.032913 seconds
For N = 12 time taken: 0.006011 seconds
For N = 13 time taken: 0.136665 seconds
For N = 14 time taken: 0.100817 seconds
For N = 15 time taken: 0.086797 seconds
For N = 16 time taken: 0.173417 seconds
For N = 17 time taken: 2.02304 seconds
For N = 18 time taken: 0.25834 seconds
For N = 19 time taken: 5.45898 seconds
For N = 20 time taken: 1.10584 seconds
PS C:\Users\user\Desktop\ПиАА\lab1> 
```

Рисунок 2 – результаты времени выполнения алгоритма для различных n

По данным результатам видно, что различаются изменения времени выполнения для четных и нечетных n . То есть по сложности алгоритма должно получаться так, что с увеличением n время выполнения должно увеличиться. Однако, мы видим, что, к примеру, для числа $n=18$, которое стоит между $n=17$ (2.02 сек) и $n=19$ (5.46 сек), время выполнение гораздо меньше по сравнению с соседними нечетными числами.

Данная ситуация происходит по той причине, что для четных чисел алгоритм работает значительно быстрее, так как у четных чисел всегда $k = 4$. Мы просто делим столешницу на 4 равные части и ставим туда квадрат. Наш написанный алгоритм сразу определяет эти 4 квадрата и в дальнейшем будет

отсекать все варианты, в которых количество квадратов уже превышает это число.

Построим отдельно графики для четных и нечетных чисел с помощью Python и библиотеки matplotlib, чтобы подтвердить данную гипотезу (см. рис. 3).

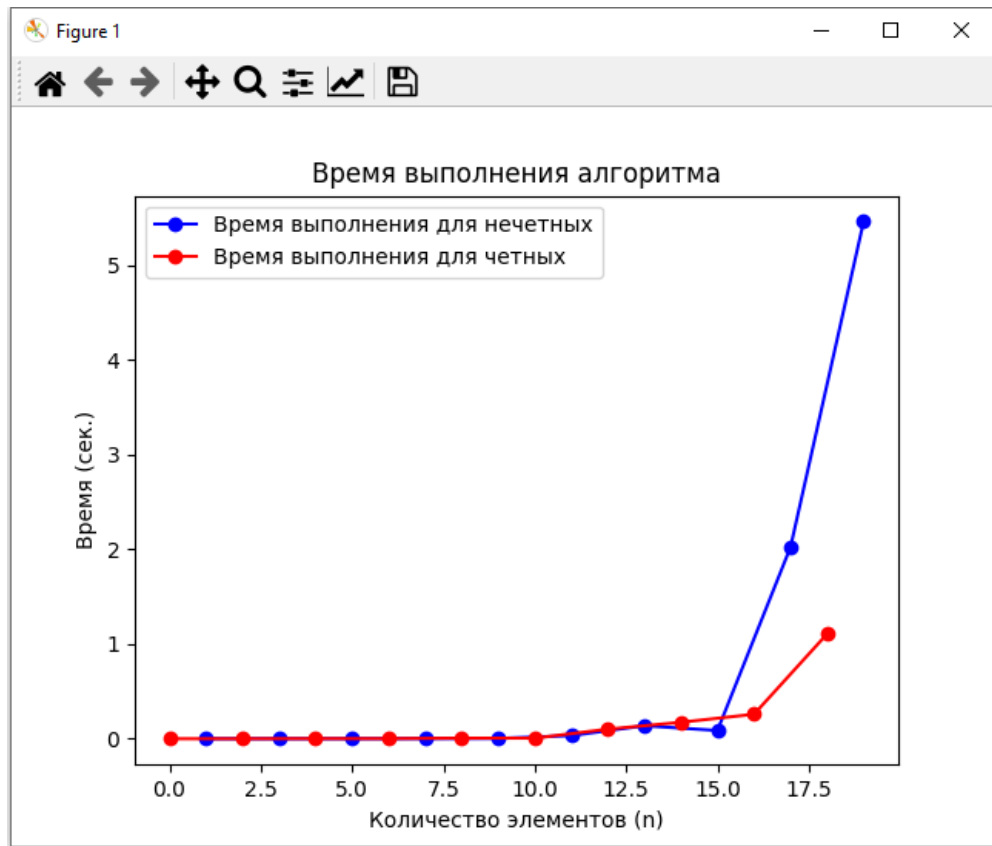


Рисунок 3 – график зависимости времени выполнения алгоритма от n

По данному графику мы заметим, что на маленьких значениях разница не особо заметна. Однако для больших n видно, что время выполнения растет быстрее для нечетных n . Также можно увидеть, что с увеличением n графики растут экспоненциально вверх. С увеличением n время выполнения будет сильно возрастать, что практически аналогично со сложностью, которую мы описали ранее: $O(n^{(n^2+2)})$. Однако этот график в силу своей большой степени будет еще сильнее возрастать вверх, так как рассматривает худшие случаи для нашего алгоритма. В нашем алгоритме таких ситуации практически

не возникает, также мы использовали некоторые оптимизации. Так что наши практические графики по времени будут, очевидно, лежать ниже.

В результате исследования мы получили, что время выполнения алгоритма для нечетных n будет возрастать быстрее, чем для четных. Также мы увидели, что данные графики изменяются примерно с экспоненциальной скоростью, что приближено к сложности, которую мы описали ранее.

ПРИЛОЖЕНИЕ А

ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

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

/* Структура, описывающая элементы, которые будут
храниться в стеке*/
struct StackElement {
    std::vector<std::vector<int>>> grid; // матрица, описывающая
    состояние каждой клетки столешницы
    std::vector<std::tuple<int, int, int>> squaresUsed; // вектор
    использованных квадратов (координаты и размер)
    std::pair<int, int> emptyCell; // пара чисел, хранящая следующие
    пустые клетки на столешнице
};

void printMatrix(const std::vector<std::vector<int>>& grid) {
    // Функция для вывода в консоль текущего состояния матрицы
    for (const auto& row : grid) {
        for (int val : row) {
            std::cout << val << "\t";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

bool isValid(int x, int y, int w, int n, const
std::vector<std::vector<int>>& grid) {
    // Функция, которая проверяет возможность поставить на координаты
    (x, y) квадрат размера w
    if (x + w > n || y + w > n) {
        return false;
    }
    for (int i = x; i < x + w; i++) {
        for (int j = y; j < y + w; j++) {
            if (grid[i][j] != 0) {
                return false;
            }
        }
    }
    return true;
}

void placeSquare(int x, int y, int w, std::vector<std::vector<int>>&
grid, int value) {
    // Функция для вставки на соответствующие координаты столешницы
    квадрата размера w
    for (int i = x; i < x + w; i++) {
        for (int j = y; j < y + w; j++) {
            grid[i][j] = value;
        }
    }
}
```

```

    }
}

std::pair<int, int> findEmptyCell(const std::vector<std::vector<int>>&
grid, int n) {
    // Функция для нахождения по определенному алгоритму пустой клетки
    for (int x = 0; x < n; x++){
        // Сначала идет поиск свободного места на на границах столешницы
        if (grid[x][0] == 0) { // Проверка первого столбца
            return {x, 0};
        }
        if (grid[0][x] == 0) { // Проверка первой строки
            return {0, x};
        }
    }
    for (int i = 1; i < n; i++) { // Обход всех оставшихся клеток
        for (int j = 1; j < n; j++) {
            if (grid[i][j] == 0) {
                return {i, j};
            }
            if (grid[j][i] == 0) { // Оптимизация для симметричной
проверки
                return {j, i};
            }
        }
    }
    return {-1, -1};
}

void newStateForStack(int x, int y, int w, const
std::vector<std::vector<int>>& grid, const std::vector<std::tuple<int,
int, int>>& squaresUsed, std::vector<StackElement>& stack, int n){
    // Функция для добавления нового состояния в стек
    std::vector<std::vector<int>> newGrid = grid;
    placeSquare(x, y, w, newGrid, squaresUsed.size() + 1); //
Копирование предыдущего состояния сетки и добавление нового квадрата

    std::vector<std::tuple<int, int, int>> newSquares = squaresUsed;
    newSquares.push_back({x + 1, y + 1, w}); // Копирование предыдущего
состояния вектора квадратов и добавление нового
    std::pair<int, int> nextEmptyCell = findEmptyCell(newGrid, n);

    stack.push_back({newGrid, newSquares, nextEmptyCell}); //
Добавление обновленных полей
}

std::vector<std::tuple<int, int, int>> solve(int n) {
    // Функция с алгоритмом решения в виде итеративного бэктрекинга
    std::vector<std::vector<int>> grid(n, std::vector<int>(n, 0)); //
Инициализация матрицы столешницы размера n*n нулями
    std::vector<StackElement> stack; // Стек, куда будут добавляться все
возможные варианты и поочередно рассматриваться
    std::vector<std::tuple<int, int, int>> bestSolution; // Вектор
множества квадратов (координаты и размеры), из которых можно заполнить
столешницу

```

```

        for (int i = n / 2; i < n; i++) { // Начальное заполнение матрицы
            максимально большим квадратом
            newStateForStack(0, 0, i, grid, {}, stack, n);
        }

        while (!stack.empty()) { // Цикл для обхода всех состояний столешницы
            StackElement current = stack.back(); // Получение последнего
            состояния
            stack.pop_back();

            if (!bestSolution.empty() && current.squaresUsed.size() >=
bestSolution.size()) {
                // Если в данном состоянии уже есть перебор по количеству
                квадратов по сравнению с последним лучшим решением,
                // происходит выход из этой ветки и переход к следующей
                std::cout<< "Stop check this state: " <<
current.squaresUsed.size() << std::endl;
                printMatrix(current.grid);
                continue;
            }

            if (current.emptyCell.first == -1 && current.emptyCell.second
== -1) {
                if (bestSolution.empty() || current.squaresUsed.size() <
bestSolution.size()) {
                    // Обновление нового лучшего решения, если столешница
                    заполнена и было затрачено меньше квадратов
                    std::cout<< "Add new best solution: " <<
current.squaresUsed.size() << std::endl;
                    printMatrix(current.grid);
                    bestSolution = current.squaresUsed;
                }
                continue;
            }

            int x = current.emptyCell.first;
            int y = current.emptyCell.second;

            if (x == 0 || y == 0){
                // Если есть свободные клетки на границах матрицы, идет
                добавление матрицы максимально возможного размера для оптимизации
                int w = std::min({n - x, n - y});
                if (isValid(x, y, w, n, current.grid)) {
                    newStateForStack(x, y, w, current.grid,
current.squaresUsed, stack, n);
                }
                continue;
            }

            for (int w = 1; w <= std::min({n - x, n - y, n - 1}); w++) {
                // Если на предыдущем шаге не удалось добавить квадрат,
                // для свободной клетки идет перебор всех возможных размеров
                квадрата и добавление этих состояний в стек
                if (isValid(x, y, w, n, current.grid)) {
                    newStateForStack(x, y, w, current.grid,
current.squaresUsed, stack, n);
                }
            }
        }
    }

```

```

    }

    // Выход из цикла означает пустоту стека и рассмотрение всех
    // возможных вариантов. Возвращаем лучшее решение
    return bestSolution;
}

int main() {
    int n;
    std::cin >> n; // Чтение входных данных
    auto solution = solve(n);

    std::cout << solution.size() << std::endl;
    for (const auto& square : solution) { // Обход вектора и вывод всех
        чисел каждого кортежа (x, y, w)
        std::cout << std::get<0>(square) << " " << std::get<1>(square)
        << " " << std::get<2>(square) << std::endl;
    }

    return 0;
}

```

Название файла: time.cpp

```

#include <chrono>
#include <iostream>
#include <vector>
#include <algorithm>
#include <tuple>

/* Структура, описывающая элементы, которые будут
храниться в стеке*/
struct StackElement {
    std::vector<std::vector<int>>> grid; // матрица, описывающая
    состояние каждой клетки столешницы
    std::vector<std::tuple<int, int, int>> squaresUsed; // вектор
    использованных квадратов (координаты и размер)
    std::pair<int, int> emptyCell; // пара чисел, хранящая следующие
    пустые клетки на столешнице
};

void printMatrix(const std::vector<std::vector<int>>& grid) {
    // Функция для вывода в консоль текущего состояния матрицы
    for (const auto& row : grid) {
        for (int val : row) {
            std::cout << val << "\t";
        }
        std::cout << std::endl;
    }
    std::cout << std::endl;
}

bool isValid(int x, int y, int w, int n, const
std::vector<std::vector<int>>& grid) {
    // Функция, которая проверяет возможность поставить на координаты
    (x, y) квадрат размера w
    if (x + w > n || y + w > n) {
        return false;
    }
}

```

```

    }
    for (int i = x; i < x + w; i++) {
        for (int j = y; j < y + w; j++) {
            if (grid[i][j] != 0) {
                return false;
            }
        }
    }
    return true;
}

void placeSquare(int x, int y, int w, std::vector<std::vector<int>>&
grid, int value) {
    // Функция для вставки на соответствующие координаты столешницы
    квадрата размера w
    for (int i = x; i < x + w; i++) {
        for (int j = y; j < y + w; j++) {
            grid[i][j] = value;
        }
    }
}

std::pair<int, int> findEmptyCell(const std::vector<std::vector<int>>&
grid, int n) {
    // Функция для нахождения по определенному алгоритму пустой клетки
    for (int x = 0; x < n; x++) {
        // Сначала идет поиск свободного места на на границах столешницы
        if (grid[x][0] == 0) { // Проверка первого столбца
            return {x, 0};
        }
        if (grid[0][x] == 0) { // Проверка первой строки
            return {0, x};
        }
    }
    for (int i = 1; i < n; i++) { // Обход всех оставшихся клеток
        for (int j = 1; j < n; j++) {
            if (grid[i][j] == 0) {
                return {i, j};
            }
            if (grid[j][i] == 0) { // Оптимизация для симметричной
проверки
                return {j, i};
            }
        }
    }
    return {-1, -1};
}

void newStateForStack(int x, int y, int w, const
std::vector<std::vector<int>>& grid, const std::vector<std::tuple<int,
int, int>>& squaresUsed, std::vector<StackElement>& stack, int n) {
    // Функция для добавления нового состояния в стек
    std::vector<std::vector<int>> newGrid = grid;
    placeSquare(x, y, w, newGrid, squaresUsed.size() + 1); //
Копирование предыдущего состояния сетки и добавление нового квадрата

    std::vector<std::tuple<int, int, int>> newSquares = squaresUsed;

```

```

        newSquares.push_back({x + 1, y + 1, w}); // Копирование предыдущего
состояния вектора квадратов и добавление нового
        std::pair<int, int> nextEmptyCell = findEmptyCell(newGrid, n);

        stack.push_back({newGrid, newSquares, nextEmptyCell}); //
Добавление обновленных полей
    }

std::vector<std::tuple<int, int, int>> solve(int n) {
    // Функция с алгоритмом решения в виде итеративного бэктрекинга
    std::vector<std::vector<int>> grid(n, std::vector<int>(n, 0)); //
Инициализация матрицы столешницы размера n*n нулями
    std::vector<StackElement> stack; // Стек, куда будут добавляться все
возможные варианты и поочередно рассматриваться
    std::vector<std::tuple<int, int, int>> bestSolution; // Вектор
множества квадратов (координаты и размеры), из которых можно заполнить
столешницу

    for (int i = n / 2; i < n; i++) { // Начальное заполнение матрицы
максимально большим квадратом
        newStateForStack(0, 0, i, grid, {}, stack, n);
    }

    while (!stack.empty()) { // Цикл для обхода всех состояний столешницы
        StackElement current = stack.back(); // Получение последнего
состояния
        stack.pop_back();

        if (!bestSolution.empty() && current.squaresUsed.size() >=
bestSolution.size()) {
            // Если в данном состоянии уже есть перебор по количеству
квадратов по сравнению с последним лучшим решением,
            // происходит выход из этой ветки и переход к следующей
            /*std::cout<< "Stop check this state: " <<
current.squaresUsed.size() << std::endl;
            printMatrix(current.grid);*/
            continue;
        }

        if (current.emptyCell.first == -1 && current.emptyCell.second
== -1) {
            if (bestSolution.empty() || current.squaresUsed.size() <
bestSolution.size()) {
                // Обновление нового лучшего решения, если столешница
заполнена и было затрачено меньше квадратов
                /*std::cout<< "Add new best solution: " <<
current.squaresUsed.size() << std::endl;
                printMatrix(current.grid);*/
                bestSolution = current.squaresUsed;
            }
            continue;
        }

        int x = current.emptyCell.first;
        int y = current.emptyCell.second;

        if (x == 0 || y == 0){

```

```

        // Если есть свободные клетки на границах матрицы, идет
добавление матрицы максимально возможного размера для оптимизации
        int w = std::min({n - x, n - y});
        if (isValid(x, y, w, n, current.grid)) {
            newStateForStack(x, y, w, current.grid,
current.squaresUsed, stack, n);
        }
        continue;
    }

    for (int w = 1; w <= std::min({n - x, n - y, n - 1}); w++) {
        // Если на предыдущем шаге не удалось добавить квадрат,
        // для свободной клетки идет перебор всех возможных размеров
квадрата и добавление этих состояний в стек
        if (isValid(x, y, w, n, current.grid)) {
            newStateForStack(x, y, w, current.grid,
current.squaresUsed, stack, n);
        }
    }
}

// Выход из цикла означает пустоту стека и рассмотрение всех
возможных вариантов. Возвращаем лучшее решение
return bestSolution;
}

int main() {
    for (int n = 1; n <= 20; n++){
        auto start = std::chrono::high_resolution_clock::now();
        auto solution = solve(n);
        auto end = std::chrono::high_resolution_clock::now();

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

        std::cout << "For N = " << n << " time taken: " << elapsed.count()
<< " seconds" << std::endl;
    }

    return 0;
}

```