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. 3343 ___ Какира Умар

Преподаватель ___ Жангиров Т.Р

Санкт-Петербург

2025

Цель работы.

Применить на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов.

Задание. Вариант – 1р (рекурсивный бэктрекинг).

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).

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 ($2 \le N \le 40$).

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 ($1 \le x,y \le N$) и длину стороны соответствующего обрезка(квадрата).

Теоретические материалы.

Бэктрекинг (поиск с возвратом) – это общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. 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. Для ускорения метода стараются вычисления организовать таким образом, чтобы как можно раньше выявлять заведомо неподходящие варианты. Зачастую это позволяет значительно уменьшить время нахождения решения.

Описание алгоритма.

Функция **backtrack**() рекурсивно находит оптимальное решение для задачи расстановки квадратов на прямоугольной доске. Она перебирает все возможные варианты размещения квадратов на оставшейся части доски. Для каждого варианта она проверяет, не перекрывается ли он с уже размещенными квадратами на доске, используя функцию **isOverlap** (). Если квадрат не перекрывается, он добавляется к списку размещенных квадратов, и функция вызывается рекурсивно с обновленным списком квадратов и доски.

Функция повторяет этот процесс, пока все квадраты не будут размещены на доске или пока не будет найдено лучшее решение. Если найдено решение с меньшим количеством квадратов, чем было найдено ранее, то список лучших квадратов обновляется.

Используются дополнительные параметры для отслеживания количества рекурсивных вызовов и ускорения перебор.

Описание функций и структур данных.

Для решения задачи использован алгоритм рекурсивного бэктрекинга.

Class Square: используется для хранения информации о каждом квадрате. Он содержит:

- Координаты верхнего левого угла *(x,y)*.
- Длину стороны квадрата *size*.
- Вычисляемые свойства *trailing* и *bottom*, которые определяют правую и нижнюю границы квадрата.

Class Table: используется для заполнения сетки и вывода результата Содержит:

- Коэффициент соотношения координат для правильного вывода **squareSize**
- Размер сетки ***gridSize****;*
- Количество квадратов в наилучшем заполнении ***bestCount****;*
- Лучшее заполнение ***bestSolution****;*

Method backtrack: Если вся поверхность заполнена ***occupiedArea*** = *gridSize* × *gridSize*

проверяется, является ли текущее решение лучше найденного ранее. Для каждой точки *(x,y)* проверяется, можно ли разместить там квадрат. Для каждой точки вычисляется максимальный размер квадрата, который можно разместить без перекрытия с уже размещенными квадратами. Рекурсивно вызывается функция *backtrack* для каждого возможного размера квадрата. Если текущее количество квадратов превышает лучшее найденное решение, ветка отсекается.

Для **оптимизации** в решении участвуют только квадраты с размером, кратным наибольшему делителю n. Также на столешнице сразу располагаются квадратные доски со стороной *(n+1)/2* в точку *{0, 0}* и два квадрата размером *n/2* в точки *{0, (n+1)/2} и {(n+1)/2, 0}*.

Method isOverlap: Функция проверяет, перекрывает ли точка (x,y) уже размещенные квадраты. Она работает следующим образом:

- Итерируется по массиву квадратов и проверяет, попадает ли точка в границы какого-либо квадрата.
- Возвращает true, если перекрытие обнаружено, и false в противном случае.

Method findMaxSizeSquare: Функция определяет максимально возможный размер квадрата, который можно разместить в сетке, начиная с заданной точки, при этом не пересекаясь с существующими квадратами.

Аргументы: const vector<Square> squares (current squares) int x (x-axis) int y (y-axis)

Тип возвращаемого значения: max possible size of a square (**int**)

Method simplifyGrid: Функция пытается упростить размер сетки, приводя его к наименьшему возможному целому числу

Method placeSquares: Функция инициализирует начальное состояние поверхности, размещая три квадрата:

- Один квадрат в левом верхнем углу размером (n+1)/2(*n*+1)/2.
- Два квадрата размером n/2*n*/2 в правом верхнем и левом нижнем углах.

Запускает бектрекинг.

Method printResult: выводит результат бектрекинга в необходимом формате

Исследование Сложность.

Этот алгоритм использует поиск с возвратом (backtracking) для разбиения квадратного поля (**gridSize × gridSize**) на минимальное количество квадратов.

Анализ временной сложности

1- Количество возможных размещений квадратов:

- В худшем случае каждый квадрат может начинаться в любой клетке сетки gridSize × gridSize.

- Следовательно, начальных позиций порядка $O(N^2)$, где N = gridSize.

2- Размер квадратов

В худшем случае размер квадрата может быть от 1 до gridSize, что даёт O(N) возможных размеров.

3- Рекурсивная глубина:

- Максимальное количество квадратов, которыми можно покрыть поле, соответствует разбиению на наименьшие возможные квадраты (например, разбиение N × N на квадраты 1 × 1).

- Это даёт потенциально экспоненциальную глубину рекурсии: $O(N^2)$.

4- Фильтрация путей:

- Используются эвристики для обрезки невыгодных веток (например, minSquaresNeeded, проверка bestCount).

- Это снижает среднее время работы, но в худшем случае ветки всё равно исследуются.

Итоговая сложность

Если рассматривать худший случай, то алгоритм может генерировать экспоненциальное количество решений, что даёт сложность O((N²)^N) в наивном случае. Однако, благодаря отсечению неэффективных путей (эвристикам), реальная сложность намного меньше, но всё равно экспоненциальная в худшем случае.

Итог:

- В среднем случае: O(2^N) (благодаря отсечениям).
- В худшем случае: O((N²)^N) (если эвристики не помогут).

Тестирование.

Ввод	Вывод
7	9 1 1 4 1 5 3 5 1 3 4 5 1 4 6 2 5 4 2 6 6 2 7 4 1 7 5 1
10	4 1 1 5 1 6 5 6 1 5 6 6 5

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
39	6 1 1 26 1 27 13 27 1 13 14 27 13 27 14 13 27 27 13

25	8 1 1 15 16 1 10 1 16 10 11 16 10 16 11 5 21 11 5 21 16 5 21 21 5

Выводы.

Применен на практике алгоритм поиска с возвратом для заполнения квадрата минимальным кол-вом меньших квадратов. В результате работы было придумано несколько оптимизаций, которые позволили уменьшить основание в экспоненциальной сложности, сократив время работы алгоритма.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

File main.cpp:

```
#include <iostream>
#include "Table.cpp"

using namespace std;

int main() {
    int gridSize;
    cin >> gridSize;

// Create a new Table
with a size of gridSize

    Table table(gridSize);
    table.placeSquares();
    table.printResult();

    return 0;

}
```

File Table.cpp:

```
#include <vector>
#include "Square.cpp"
#include <iostream>

using namespace std;

class Table {
private:
    int squareSize;
    int gridSize;
    int bestCount;
    vector<Square> bestSolution;

public:
    // Place squares in the grid
    void placeSquares() {
        simplifyGrid();
        //cout << "SquareSize: " << squareSize << endl;
        //cout << "GridSize: " << gridSize << endl;
        int startX = gridSize / 2;
        int startY = (gridSize + 1) / 2;


        vector<Square> squares = { Square(0, 0, startY), Square(0, startY, startX),
                                   Square(startY, 0, startX) };

        int occupiedArea = pow(startY, 2) + 2 * pow(startX, 2);

        backtrack(squares, occupiedArea, 3, startX, startY);
    }

    // Print the result (best solution)
    void printResult() {
        cout << bestCount << endl;
        for (const auto& square : bestSolution) {
            cout << square.x * squareSize + 1 << " " << square.y * squareSize + 1 << " "
                 << square.size * squareSize << endl;
        }
    }

    Table(int gridSize) : gridSize(gridSize), bestCount(gridSize * gridSize + 1) {}

private:

    // Explore All other possible position and deteremin the best solution
    void backtrack(vector<Square> currentSquares, int occupiedArea, int currentCount, int startX, int startY) {

        // Base case: If the entire grid is occupied, we check if this solution is better than the current best.
        if (occupiedArea == gridSize * gridSize) {
            if (currentCount < bestCount) {
                // Update the best solution if the current solution uses fewer squares.
                bestCount = currentCount;
                bestSolution = currentSquares;
            }
```

```
        return; // Stop further exploration for this branch.
    }

    // Iterate over all possible positions (x, y) in the grid.
    for (int x = startX; x < gridSize; x++) {
        for (int y = startY; y < gridSize; y++) {
            // Skip this position if placing a square here would overlap with existing squares.
            if (isOverlap(currentSquares, x, y))
                continue;

            // Find the maximum possible size of a square that can be placed at (x, y).
            int maxSizeOfSquare = findMaxSizeSquare(currentSquares, x, y);

            // If no square can be placed at (x, y), skip this position.
            if (maxSizeOfSquare <= 0)
                continue;

            // Try placing squares of all possible sizes, starting from the largest.
            for (int size = maxSizeOfSquare; size >= 1; size--) {
                // Create a new square at (x, y) with the current size.
                Square newSquare(x, y, size);

                // Calculate the new occupied area after placing this square.
                int newOccupiedArea = occupiedArea + size * size;

                // Calculate the remaining area that still needs to be covered.
                int remainingArea = gridSize * gridSize - newOccupiedArea;

                // If there is remaining area, estimate the minimum number of squares needed to cover it.
                if (remainingArea > 0) {
                    // The maximum possible size of a square that can fit in the remaining area.
                    int maxPossibleSize = min(gridSize - x, gridSize - y);

                    // Estimate the minimum number of squares needed to cover the remaining area.
                    int minSquaresNeeded =
                        (remainingArea + (maxPossibleSize * maxPossibleSize) - 1) /
                        (maxPossibleSize * maxPossibleSize);

                    // If the current solution cannot be better than the best solution, skip this branch.
                    if (currentCount + 1 + minSquaresNeeded >= bestCount) {
                        continue;
                    }
                }

                // Add the new square to the current solution.
                currentSquares.push_back(newSquare);

                // If the grid is fully occupied, check if this solution is better than the best solution.
                if (newOccupiedArea == gridSize * gridSize) {
                    if (currentCount + 1 < bestCount) {
                        // Update the best solution.
                        bestCount = currentCount + 1;
                        bestSolution = currentSquares;
                    }
                    // Remove the last square and continue exploring other possibilities.
                    currentSquares.pop_back();
                    continue;
```

```
            }

            // If the current solution can still be improved, recursively explore further.
            if (currentCount + 1 < bestCount) {
                backtrack(currentSquares, newOccupiedArea, currentCount + 1, x, y);
            }

            // Remove the last square to backtrack and try other possibilities.
            currentSquares.pop_back();
        }

        // After trying all sizes at (x, y), move to the next position.
        return;
    }

    // Reset startY to 0 after the first row to ensure all positions are explored.
    startY = 0;
  }
}

// Check if the position (x,y) overlap with existing squares
bool isOverlap(const vector<Square>& squares, int x, int y) {
  for (const auto& square : squares) {
    if (x >= square.x && x < square.trailing && y >= square.y && y < square.bottom) {
      return true;
    }
  }
  return false;
}

// Find the possible max size for a square
int findMaxSizeSquare(const vector<Square>& squares, int x, int y) {
  // Initialize the maximum possible size of a square that can be placed at (x, y).
  // This is constrained by the grid boundaries: the square cannot extend beyond the grid.
  int maxSizeOfSquare = min(gridSize - x, gridSize - y);

  // Iterate through all existing squares to check if they restrict the size of the new square.
  for (const auto& square : squares) {
    // Check if the existing square is to the right and below the position (x, y).
    // If so, the new square's size is restricted by the distance between y and the existing square's top edge.
    if (square.trailing > x && square.y > y) {
      maxSizeOfSquare = min(maxSizeOfSquare, square.y - y);
    }
    // Check if the existing square is below and to the right of the position (x, y).
    // If so, the new square's size is restricted by the distance between x and the existing square's left edge.
    else if (square.bottom > y && square.x > x) {
      maxSizeOfSquare = min(maxSizeOfSquare, square.x - x);
    }
  }

  // Return the maximum possible size of the new square that can be placed at (x, y)
  // without overlapping any existing squares or exceeding the grid boundaries.
  return maxSizeOfSquare;
}

void simplifyGrid() {
  int maxDivisor = 1;
```

```
        for (int i = gridSize / 2; i >= 1; --i) {
            if (gridSize % i == 0) {
                maxDivisor = i;
                break;
            }
        }

        squareSize = maxDivisor;
        gridSize = gridSize / maxDivisor;
    }

};
```

File Square.cpp:

```
struct Square {
public:
    const int trailing; // right boundary (edge)
    const int bottom; // bottom boundary (edge)
    int x, y, size; // x-axis / y-axis positions / length of a square

    // Constructor
    Square(int x, int y, int size) : x(x), y(y), size(size), trailing(x + size), bottom(y + size) {}

    // A copy constructor
    Square(const Square& other)
        : x(other.x), y(other.y), size(other.size), trailing(other.trailing), bottom(other.bottom) {
    }

    Square& operator=(const Square& other) {
        if (this != &other) {
            x = other.x;
            y = other.y;
            size = other.size;
        }
        return *this;
    }
};
```