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
Вариант: 3р

Студент гр. 3388 _____ Дубровин Д.Н.
Преподаватель _____ Жангиров Т.Р.

Санкт-Петербург
2025

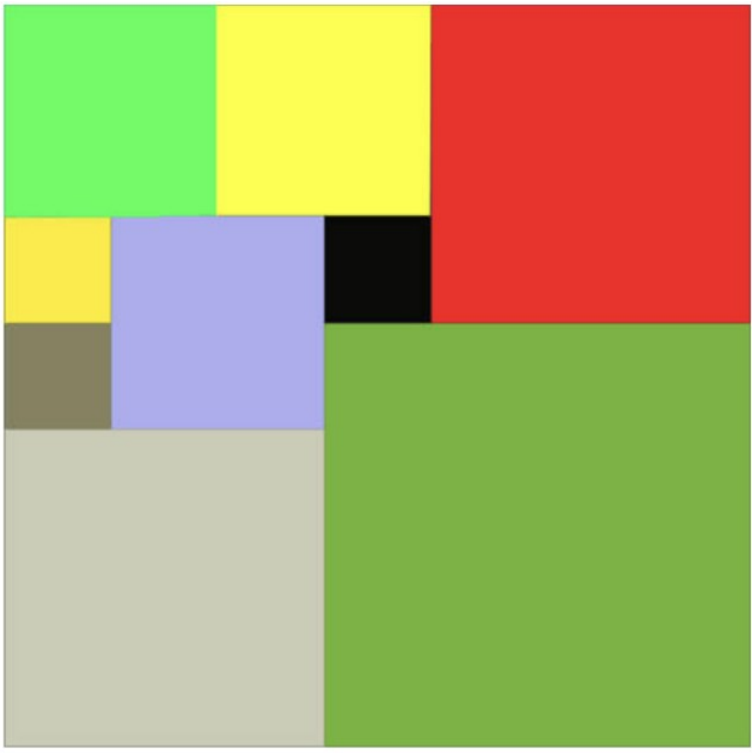
Цель работы:

Изучить теоретические основы алгоритма поиска с возвратом. Решить с его помощью задачу о разбиении квадрата. Провести исследование зависимости количества итераций от стороны квадрата.

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.



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 ($1 \le x,y \le N$) 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

Реализация

Описание алгоритма:

Алгоритм разбиения квадрата на минимальное количество подквадратов основан на методе возврата (backtracking) с отсечениями. Основная идея заключается в том, чтобы разбирать квадрат сеткой и минимизировать количество использованных меньших квадратов, покрывая всю область без наложений.

Основные этапы работы алгоритма:

Масштабирование входного квадрата

- Определяется наибольший возможный делитель squareSize для gridSize, чтобы упростить расчёты.
- В результате формируется новый масштабированный размер сетки newGridSize, который становится основой для поиска разбиения.

Инициализация с исходным расположением квадратов

- Алгоритм стартует с трёх предустановленных квадратов:
  - один большой (mainSquareSize = (gridSize + 1) / 2)
  - два меньших (subSquaresSize = gridSize / 2)

Рекурсивный алгоритм поиска разбиения

- Начинается рекурсивный перебор возможных размещений квадратов, начиная с первой свободной клетки.
- Для каждого нового квадрата оценивается максимальный возможный размер maxSize, чтобы он не выходил за границы уже занятых областей.
- Вставляется квадрат, обновляется площадь занятой области и запускается рекурсивный вызов.

Ограничение перебора (отсечения)

- Если найдено покрытие всей площади, решение фиксируется как лучшее, если количество квадратов минимально.
- Если текущее разбиение использует больше квадратов, чем известное оптимальное, поиск прекращается (backtracking).

Формирование результата

- После завершения рекурсивного перебора наилучший найденный вариант возвращается в виде списка координат квадратов и их размеров.

Описание функций и структур:

Основные структуры данных

- Square — Класс, описывающий квадрат. Поля:
  - X, Y: координаты верхнего левого угла.
  - Size: длина стороны квадрата.
  - Right, Bottom: вычисляемые границы (правый и нижний край).
- SquarePartitioner — Класс, содержащий основную логику алгоритма.

Функции:

- FindOptimalPartition(gridSize, debugMode)
  - Главная функция, запускающая алгоритм.
  - Инициализирует переменные и запускает рекурсивный процесс поиска.
  - Возвращает кортеж (количество квадратов, число операций, список координат и размеров квадратов).
- ScaleSquareSize(gridSize, out squareSize)
  - Определяет наибольший делитель gridSize, упрощая работу алгоритма.
- PlaceInitialSquares(newGridSize, debugMode)
  - Расставляет три стартовых квадрата для оптимизации разбиения.
- Backtrack(

  List<Square> squares,

  List<Square> bestSolution,

  int occupiedArea,

  int currentCount,

  int startX, int startY,

  int gridSize,

  ref int bestCount,

  ref int operationCounter,

bool debugMode

)

- Функция Backtrack() рекурсивно ищет оптимальное разбиение квадрата на наименьшее количество под-квадратов.
  - Для каждой свободной клетки (x, y) на доске пробуются все возможные размеры квадратов, начиная с максимального (CalculateMaxSquareSize() определяет допустимый размер).
  - Если квадрат не выходит за границы и не перекрывает уже размещённые квадраты (IsPositionOccupied), вызывается метод TryPlacingSquares, который рекурсивно вызывает Backtrack() для обновленного списка аргументов.
- IsPositionOccupied(squares, x, y)
  - Проверяет, занята ли клетка (x, y) в сетке.
- CalculateMaxSquareSize(squares, x, y, gridSize)
  - Определяет максимально возможный размер квадрата, который можно разместить в (x, y).
- TryPlacingSquares(

  List<Square> squares,

  List<Square> bestSolution,

  int occupiedArea,

  int currentCount,

  int x, int y,

  int gridSize,

  int maxSize,

  ref int bestCount,

  ref int operationCounter,

  bool debugMode)
  - Пробует вставить квадрат разных размеров и вызывает Backtrack для дальнейшего поиска.

- Если currentCount + 1 уже превышает bestCount, дальнейший перебор прекращается.
- После выхода из рекурсии последний добавленный квадрат удаляется (squares.RemoveAt(squares.Count - 1)) для перебора других вариантов.

- UpdateBestSolution(
  List<Square> squares,
  List<Square> bestSolution,
  int currentCount,
  ref int bestCount,
  bool debugMode)
  - Обновляет наилучший найденный вариант.

Алгоритмы оптимизации:

Инициализация с предустановленными квадратами

- Позволяет уменьшить глубину рекурсии, поскольку часть пространства сразу покрывается.

Оптимизация масштабированием

- Производим масштабирование сетки для сокращения области поиска.

Жадный выбор размера нового квадрата

- Начинаем с максимально возможного квадрата в текущей свободной точке.

Раннее отсечение неэффективных ветвей

- Если текущий вариант уже использует больше квадратов, чем известное оптимальное разбиение, то поиск прекращается.

Оценка сложности алгоритма:

Временная сложность

- В худшем случае имеем n^2 позиций для расстановки квадратов (1x1), сложность перебора $O(n^2)$
- Размер квадрата может быть от 1 до gridSize (n), получаем сложность O(n)

- В худшем случае количество итераций растёт экспоненциально. Сложность O(2^n)
- Итоговая сложность в худшем случае: $O(2^n * n * n^2) = O(2^n * n^3)$

Пространственная сложность

- Худший случай: $O(n^2)$ если предположить что под каждую клетку изначальной сетки придётся выделять структуру Square, то есть квадраты размера (1x1).
- В среднем случае: O(k) где $k < n^2$, k — количество полученных квадратов.

Дополнительные классы:

Также были реализованы следующие классы, которые не связаны с непосредственной работой алгоритма:

- CLI — Реализация интерфейса командной строки с поддержкой флагов:

  -h, --help

  Вывести справку о доступных параметрах и завершить работу.

  -v, --visualize <filename>

  Выполнить визуализацию решения и записать её в файл filename.

  -a, --analyze <filename>

  Запустить анализ производительности (бенчмарк) алгоритма и записать график в файл filename.

  -d, --debug

  Запустить программу в режиме отладки с выводом подробной информации о ходе выполнения.

- Visualizer — Класс для визуализации разбиения сетки на квадраты использующий библиотеку SkiaSharp.
- PerformanceAnalyzer — Класс для анализа и построения графика и таблицы производительности алгоритма в зависимости от gridSize. Использует

библиотеку Spectre.Console для печати таблицы и библиотеку ScottPlot для построения графика.

Визуализация

Для визуализации работы алгоритма была использована библиотека ScottPlot.

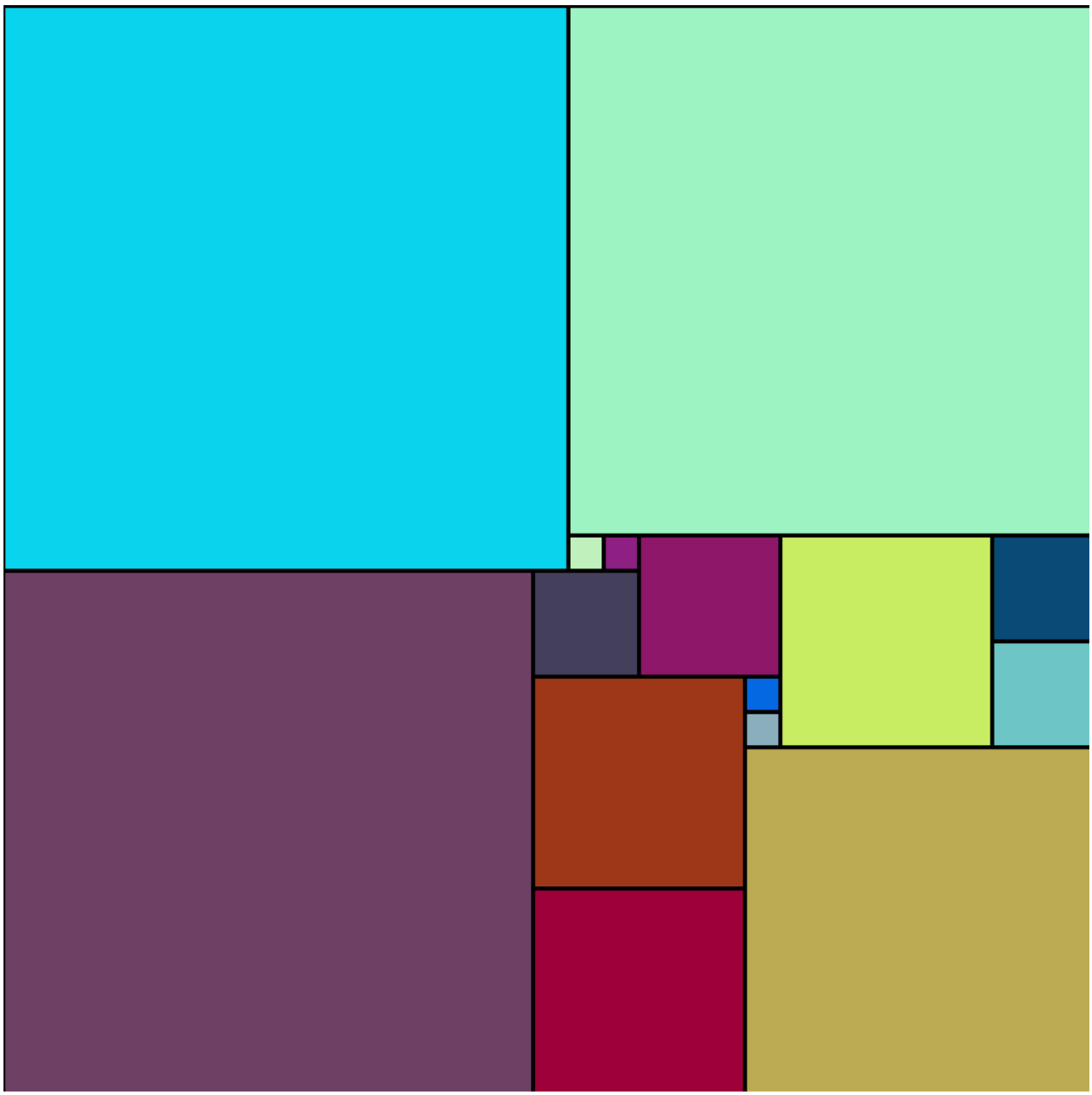


Рис. 1 Визуализация работы алгоритма.

Тестирование

Таблица 1. Тестирование.

Входные данные	*Выходные данные*
4	*4* *1 1 2* *1 3 2* *3 1 2* *3 3 2*
7	9 *1 1 4* *1 5 3* *5 1 3* *4 5 2* *4 7 1* *5 4 1* *5 7 1* *6 4 2* *6 6 2*
11	11 1 1 6 1 7 5 7 1 5 6 7 3 6 10 2 7 6 1 8 6 1 8 10 1 8 11 1 9 6 3 9 9 3
18	*4* *1 1 9* *1 10 9* *10 1 9* *10 10 9*
37	*15* *1 1 19* *1 20 18*

	20 1 18 *19 20 2* *19 22 5* *19 27 11* *20 19 1* *21 19 3* *24 19 8* *30 27 3* *30 30 8* *32 19 6* *32 25 1* *32 26 1* *33 25 5*

Исследование

В ходе лабораторной работы было проведено исследование зависимости количества итераций от стороны квадрата. В ходе исследования получились следующие результаты (рис. 1 и табл. 2).

Таблица 2. Зависимость количества итераций от стороны квадрата.

Сторона квадрата	Количество итераций
2	1
3	3
4	1
5	12
6	1
7	35
8	1
9	3
10	1
11	376
12	1
13	828
14	1
15	3
16	1

17	4621
18	1
19	12237
20	1
21	3
22	1
23	45076
24	1
25	12
26	1
27	3
28	1
29	306178
30	1
31	695875
32	1
33	695875
34	1
35	3
36	1
37	12
38	1
39	3483061
40	1

Построим график зависимости количества итераций от стороны квадрата. Рассматривать будем только простые числа, так как в таком случае будет задействовано минимум оптимизаций бектрекинга и можно будет пронаблюдать зависимость количества итераций от стороны квадрата в худшем случае.

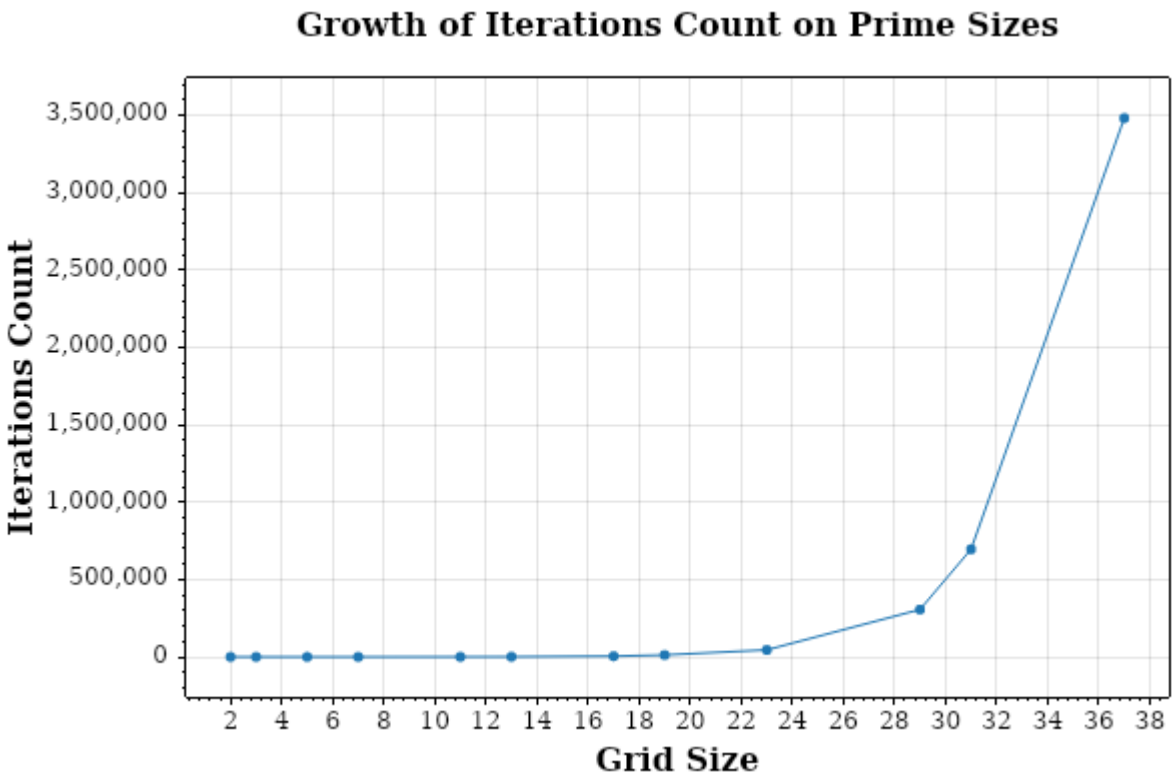


Рис. 2. Зависимость количества итераций от стороны квадрата

Size	Iterations count
2	1
3	3
5	12
7	35
11	376
13	828
17	4621
19	12237
23	45076
29	306178
31	695875
37	3483061

Рис. 3. Вывод количества итераций от стороны квадрата

Вывод

В ходе лабораторной работы была написана программа с использованием алгоритма бэктрекинга. Также было проведено тестирование на различных входных данных. По результатом исследования можно заключить, что зависимость числа операций от размера поля для простых значений экспоненциальна.

Исходный код программы см. в ПРИЛОЖЕНИИ А.

ПРИЛОЖЕНИЕ А.

ИСХОДНЫЙ КОД ПРОГРАММЫ

Algorithm.cpp

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

using namespace std;

struct Square {
    int x, y, h;
};

int cnt = 0;
bool DEBUG=0;

void print_solution_matrix(int n, const vector<tuple<int, int, int>>&
result) {

    vector<vector<int>> matrix(n, vector<int>(n, 0));
    int num = 0;

    for (const auto& s : result) {
        num += 1;
        int x, y, h;
        tie(x, y, h) = s;

        for (int i = x; i < x + h; ++i) {
            for (int j = y; j > y - h; --j) {
                matrix[i][j] = num;
            }
        }
    }

    for (int i = 0; i < n; ++i) {
        for (int j = 0; j < n; ++j) {

            if (matrix[i][j] < 10) {
                cout << matrix[i][j] << "  ";
            } else {
                cout << matrix[i][j] << " ";
            }
        }
        cout << endl;
    }
}

void rec(vector<int>& diagram, vector<int> marks, vector<Square>& ans)
{
    static vector<Square> stack = {};
```

```
    if (DEBUG==1){
        cnt += 1;
        cout << "IT #" << cnt << '\n';

        for (Square s : stack) {
            cout << '\t' << s.x << ' ' << s.y << ' ' << s.h << '\n';
        }
    }

    if (*max_element(diagram.begin(), diagram.end()) == 0) {
        if (ans.empty() || ans.size() > stack.size()) {
            ans = stack;
        }
        return;
    }

    int corners = (diagram.back() != 0);
    for (int i = 0; i < diagram.size() - 1; ++i) {
        corners += (diagram[i] != diagram[i + 1]);
    }

    if (!ans.empty() && stack.size() + corners >= ans.size()) {
        return;
    }

    for (int i = 0; i < diagram.size(); ++i) {
        int j = diagram[i] - 1;
        int max_h = 0;
        while (i - max_h >= 0 && diagram[i - max_h] == diagram[i]) {
            ++max_h;
        }
        if (i == diagram.size() - 1) {
            max_h = min(max_h, diagram[i]);
        } else {
            max_h = min(max_h, diagram[i] - diagram[i + 1]);
        }
        max_h = min(max_h, (int)diagram.size() - 1);
        for (int k = 0; k < max_h; ++k) {
            diagram[i - k] -= max_h;
        }
        for (int h = max_h; h >= 1; --h) {
            if (h > marks[i]) {
                stack.push_back({i + 1 - h, j, h});
                int x = marks[i];
                marks[i] = -1;
                rec(diagram, marks, ans);
                marks[i] = x;
                stack.pop_back();
            }
            diagram[i + 1 - h] += h;
            for (int k = 0; k < h - 1; ++k) {
                ++diagram[i - k];
            }
        }
        marks[i] = max_h;
    }
}
int main() {
```

```
int n;
cin >> n;
vector<Square> ans;
vector<int> hs;

for (int h = (n + 1) / 2; h < min((n + 1) / 2 + 5, n); ++h) {
    hs.push_back(h);
}

if (n > 20) {
    if (n % 2 == 0) {
        hs = {n / 2};
    } else if (n % 3 == 0) {
        hs = {2 * n / 3};
    } else if (n == 25 || n == 27) {
        hs = {(n + 1) / 2 + 2};
    } else if (n == 37) {
        hs = {(n + 1) / 2 + 1};
    } else {
        hs = {(n + 1) / 2 + 1, (n + 1) / 2 + 3};
    }
}

for (int h : hs) {
    vector<int> diagram(n, n);
    vector<Square> cur_ans;

    for (int i = 0; i < h; ++i) {
        diagram[n - 1 - i] -= h;
    }
    for (int i = 0; i < n - h; ++i) {
        diagram[i] -= n - h;
    }
    for (int i = 0; i < n - h; ++i) {
        diagram[n - 1 - i] -= n - h;
    }


    rec(diagram, vector<int>(n, -1), cur_ans);

    cur_ans.push_back({n - h, n - 1, h});
    cur_ans.push_back({0, n - 1, n - h});
    cur_ans.push_back({n - 1 - (n - h) + 1, n - h - 1, n - h});

    if (ans.empty() || ans.size() > cur_ans.size()) {
        ans = cur_ans;
    }
}



cout << ans.size() << endl;
for (Square s : ans) {
    cout << s.x + 1 << ' ' << s.y - s.h + 2 << ' ' << s.h << endl;
}

vector<tuple<int, int, int>> result;
for (const auto& s : ans) {
```

```
            result.push_back(make_tuple(s.x, s.y, s.h));
        }

        if (DEBUG==1){
            cout << "Total iterations: " << cnt << '\n';
            print_solution_matrix(n, result);
        }

        return 0;
    }
```