

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

Студентка гр. 3388

Титкова С.Д.

Преподаватель

Жангиров Т.Р.

Санкт-Петербург

2025

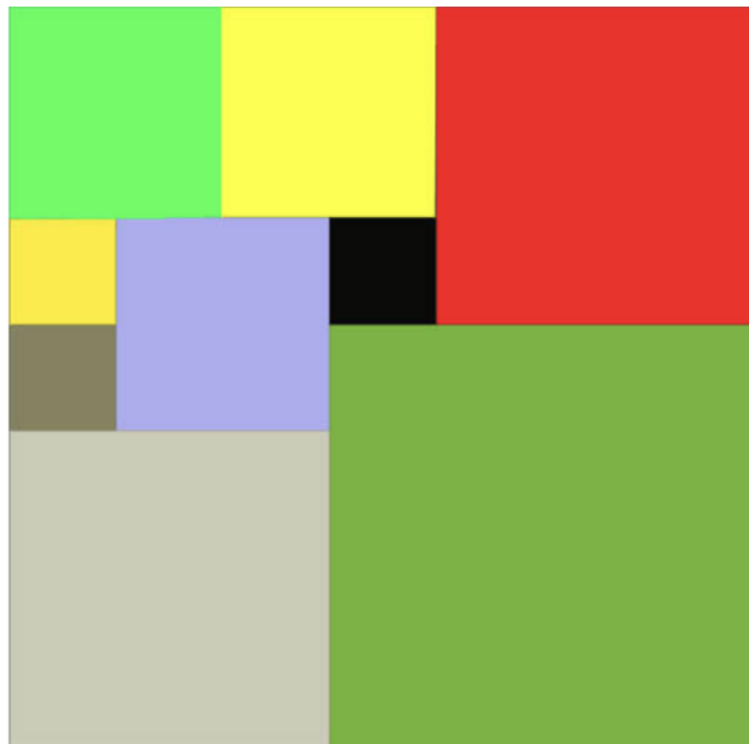
Цель работы:

Изучить принцип работы алгоритма поиска с возвратом. Решить с его помощью задачу. Также 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 ($2 \leq N \leq 20$).

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 | | |
| 112 | | |
| 132 | | |
| 311 | | |
| 411 | | |
| 322 | | |
| 513 | | |
| 444 | | |
| 153 | | |
| 341 | | |

Реализация

Описание алгоритма:

Для решения поставленной задачи был использован рекурсивный бэктрекинг (рекурсивный поиск с возвратом). Так, после ввода стороны генерируется набор возможных высот для начального размещения крупных квадратов. Этот набор зависит от размера входного значения. Для каждой возможной высоты из набора выполняет следующие действия:

1. Инициализирует начальную диаграмму высот прямоугольника с учётом размещения крупных начальных блоков.
2. Вызывает рекурсивную функцию поиска оптимального решения.
3. Обновляет лучшее решение, если найденное решение лучше текущего лучшего решения.

Описание функций и структур:

- *Square* - структура для представления квадрата с координатами *x*, *y* и высотой *h*;
- *vector<tuple<int, int, int>> result* - вектор кортежей для хранения результатов (координаты и размеры квадратов);
- *bool DEBUG* – это переменная, которая является переключателем. При значении 1 будет выводиться отладочная информация(ставить в самом коде).
- *void print_solution_matrix(int n, const vector<tuple<int, int, int>> & result)* - функция выводит матрицу с расположением найденных квадратов;
- *void rec(vector<int> & diagram, vector<int> marks, vector<Square> & ans):*

1. Аргументы:

- *Diagram* - вектор целых чисел, представляющий высоты оставшихся областей прямоугольника;
- *Marks* - вектор целых чисел, хранящий информацию о максимально возможных высотах квадратов на каждой позиции;
- *Ans* - вектор структур *Square*, хранящий лучшее найденное решение.

2. *Возвращаемое значение:* функция модифицирует содержимое `ans`, обновляя его при нахождении более оптимального решения;

3. *Алгоритм:*

Если все области диаграммы равны нулю, то текущее решение сохраняется в `ans`, если оно лучше предыдущего.

Для каждой позиции на диаграмме вычисляется максимально возможная высота нового квадрата (`max_h`) и удаляется из диаграммы соответствующая область.

Затем вызывается сама функция с обновлённой диаграммой и новым состоянием стека частичных решений.

После каждого рекурсивного вызова состояние восстанавливается: добавленные элементы удаляются из стека частичных решений, а изменения в диаграмме отменяются.

- *Main():*

Принимает на вход число. Генерирует набор возможных высот для начального размещения крупных квадратов. Находит 3 стороны для заполнения наибольшей площади за одну итерацию, затем ставит их в диаграмму. Затем вызывается рекурсивная функция для поиска оптимального решения. В конце выводится количество квадратов, а также строки, в которых отражаются координаты левых верхних вершин квадратов и соответствующая им высота. Количество строк равно количеству найденных квадратов.

Способ хранения частичных решений:

Частичные решения хранятся в виде:

1. *static vector<Square> stack* - это статический вектор, который хранит текущее частичное решение во время рекурсивного поиска. Каждый элемент `stack` представляет собой квадрат с координатами `x`, `y` и высотой `h`.

2. *vector<Square> ans* - это вектор, который хранит лучшее найденное решение на данный момент. Он обновляется каждый раз, когда находит более оптимальное решение.

Способ хранения:

- Когда добавляется новый квадрат к текущему решению, он помещается в конец *stack*.
- После того как все возможные ветви для данного квадрата были рассмотрены (т.е., после рекурсивного вызова), последний добавленный квадрат удаляется из конца *stack*. Это позволяет восстановить предыдущее состояние решения.
- Если текущее частичное решение становится полным (все области покрыты) и оно лучше предыдущего лучшего решения, то содержимое *stack* копируется в *ans*.

Алгоритмы оптимизации:

- *Ограничение рекурсии* - используется условие остановки рекурсии при достижении ситуации, когда текущее решение не может быть лучше найденного ранее.
- *Генерация начальных крупных блоков* - генерируются только те начальные блоки, которые имеют шанс дать лучшее решение, что сокращает количество ненужных рассмотрений. Сторона наибольшего из возможных квадратов вычисляется, как $(N+1)/2$, это верно для большинства простых чисел. Данная оптимизация значительно сокращает количество рекурсивных вызовов.
- *Отсечение по количеству углов* - вычисляет количество углов в текущей диаграмме. Если количество квадратов в текущем наборе плюс количество углов больше или равно размеру лучшего найденного решения, то текущая ветвь поиска отсекается, так как она не может привести к лучшему решению.

Оценка сложности алгоритма:

Основной идеей алгоритма является рекурсия, соответственно количество возможных переборов будет расти, как степенная функция. Ввиду использования оптимизаций, сложность алгоритма уменьшается на некоторую константу, но в худших всё ещё приближается к экспоненциальной сложности ($O(e^n)$), где n — сторона квадрата.

Относительно памяти.

Основные структуры данных:

1. **diagram:** вектор из n элементов типа `int`. Память: $O(n)$.
2. **marks:** вектор из n элементов типа `int`. Память: $O(n)$.
3. **stack:** статический вектор `Square` (x, y, h — 3 `int`). В худшем случае может содержать до n^2 квадратов (если каждый квадрат размером 1×1), но обычно гораздо меньше благодаря обрезке. Память: $O(n^2)$ в худшем случае.
4. **ans:** вектор `Square`, хранит лучшее решение. Размер зависит от оптимального числа квадратов, обычно $O(n)$.
5. **matrix в print_solution_matrix:** Матрица $n \times n$ типа `int`. Память: $O(n^2)$.
6. **hs:** Вектор высот, обычно фиксированного размера (до 5 элементов), память: $O(1)$.

Рекурсия:

- Глубина рекурсии в `res` зависит от числа квадратов, которые можно разместить. В худшем случае это $O(n^2)$, но обрезка ветвей (`stack.size() + corners >= ans.size()`) сильно сокращает глубину. Практически глубина ближе к $O(n)$.
- Каждый вызов `res` создает копии `diagram` и `marks` (по $O(n)$), что добавляет память на стек вызовов. Итог: $O(n)$ на уровень рекурсии, умноженное на глубину.

Итоговая сложность по памяти:

- **Без учета рекурсии:** $O(n^2)$ из-за `matrix` в `print_solution_matrix`.

- **С учетом рекурсии:** В худшем случае $O(n^2)$ (глубина $O(n) \times O(n)$ на уровень), но практически $O(n)$ благодаря оптимизациям.
- Если $DEBUG=0$, то `matrix` не используется, и память снижается до $O(n)$.

Тестирование

Таблица 1. Тестирование.

| <i>Входные данные</i> | <i>Выходные данные</i> |
|-----------------------|---|
| 2 | 4 1 1 1 2 2 1 1 2 1 2 1 1 |
| 3 | 6 1 2 1 2 1 1 1 1 1 2 2 2 1 3 1 3 1 1 |
| 11 | 11 5 6 1 4 6 1 4 3 3 1 4 3 5 1 2 4 2 1 4 1 1 1 1 3 6 6 6 1 7 5 7 1 5 |
| 15 | 6 1 6 5 6 1 5 1 1 5 6 6 10 |

| | |
|----|--|
| | <i>1 11 5</i> <i>11 1 5</i> |
| 19 | <i>13</i> <i>9 10 1</i> <i>8 10 1</i> <i>8 7 3</i> <i>4 7 4</i> <i>1 8 3</i> <i>5 1 6</i> <i>4 6 1</i> <i>4 5 1</i> <i>1 5 3</i> <i>1 1 4</i> <i>10 10 10</i> <i>1 11 9</i> <i>11 1 9</i> |

Исследование

Также 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 |
|------------------|---------------------|
| 3 | 5 |
| 4 | 3 |
| 5 | 31 |
| 6 | 39 |
| 7 | 188 |
| 9 | 686 |
| 11 | 4582 |
| 12 | 1864 |
| 13 | 16201 |
| 15 | 14276 |
| 17 | 97188 |
| 23 | 2148501 |

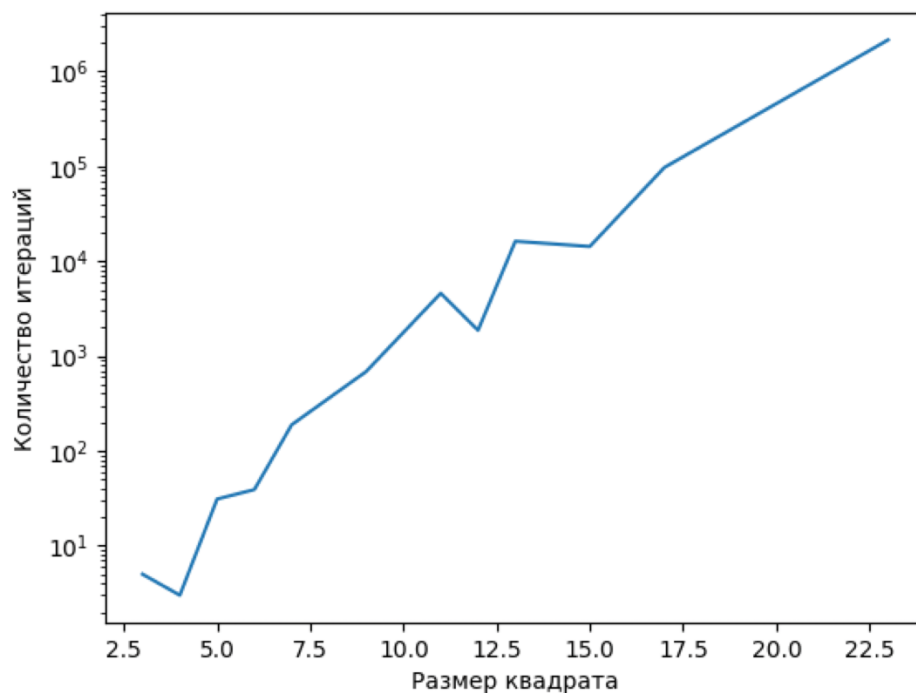


Рис. 1. Зависимость количества итераций от стороны квадрата

Построим логарифмический график зависимости количества итераций от стороны квадрата. Не сложно заметить, что значения в простых числах образуют прямую, что свидетельствует о экспоненциальной зависимости.

Вывод

В ходе лабораторной работы была написана программа с использованием алгоритма бэктрекинга. Также было проведено тестирование на различных входных данных. По результатам исследования можно заключить, что зависимость числа операций от размера поля 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=1;

void print_solution_matrix(int n, const vector<tuple<int, int, int>>&
result) {
    if (DEBUG != 1) return;

    cout << "\n=== Итоговая матрица решения ===\n";
    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;
    }
    cout << "=====\n";
}

void rec(vector<int>& diagram, vector<int> marks, vector<Square>& ans)
{
    static vector<Square> stack = {};
    if (DEBUG == 1) {
        cnt += 1;
        cout << "\nИтерация #" << cnt << ":\n";
    }
}
```

```

        cout << "Текущая диаграмма: ";
        for (int val : diagram) cout << val << " ";
        cout << "\nТекущий стек (" << stack.size() << "
квадратов):\n";
        for (Square s : stack) {
            cout << "\tКвадрат: (" << s.x + 1 << ", " << s.y - s.h + 2
<< ") с размером " << s.h << "\n";
        }
    }

    if (*max_element(diagram.begin(), diagram.end()) == 0) {
        if (ans.empty() || ans.size() > stack.size()) {
            ans = stack;
            if (DEBUG == 1) {
                cout << "\tНайден лучший результат: " << stack.size()
<< " квадратов\n";
            }
        }
        return;
    }

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

    if (DEBUG == 1) {
        cout << "\tКоличество углов: " << corners << "\n";
    }

    if (!ans.empty() && stack.size() + corners >= ans.size()) {
        if (DEBUG == 1) cout << "\tОбрезка ветви: стек + углы >=
текущего лучшего результата\n";
        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]) {
                // Добавляем квадрат
                stack.push_back({i + 1 - h, j, h});
                if (DEBUG == 1) {
                    cout << "\tДобавляем квадрат: (" << i + 1 - h + 1
<< ", " << j - h + 2

```

```

        << ") с размером " << h << "\n";
    }

    int x = marks[i];
    marks[i] = -1;
    rec(diagram, marks, ans);
    marks[i] = x;

    // Убираем квадрат, если он не подошел
    if (DEBUG == 1) {
        cout << "\tУбираем квадрат: (" << stack.back().x +
1 << ", "
        << stack.back().y - stack.back().h + 2 << ")
с размером "
        << stack.back().h << " (не подошел)\n";
    }
    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;
}

```