**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 |  | Калиберов Н. И. |
| Преподаватель |  | Жангиров Т. Р. |

Санкт-Петербург

2025

# Цель работы

Изучить работу алгоритма поиска с возвратом и написать его рекурсивную реализацию для задачи размещения квадратов на столе.

# 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 её, собрав из уже имеющихся обрезков (квадратов).

Например, столешница размера 7 x 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 <= 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).

Вар. 4р. Рекурсивный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

## Выполнение работы

Описание алгоритма

Программа решает задачу оптимального размещения квадратов на столе размером *N*×*N*. Алгоритм использует рекурсивный бэктрекинг для перебора всех возможных вариантов расстановки квадратов. На каждом шаге проверяется, можно ли разместить квадрат определённого размера в текущей позиции. Если это возможно, квадрат размещается, и алгоритм продолжает поиск для оставшейся части стола. Если размещение невозможно, алгоритм откатывается и пробует другие варианты.

Оценка сложности

* **Временная сложность:** В худшем случае временная сложность алгоритма экспоненциальная — *O*(), где *n* — размер стола, а *k* — количество возможных размеров квадратов. Это связано с тем, что на каждом шаге алгоритм перебирает все возможные размеры квадратов и рекурсивно вызывает себя для каждой новой расстановки.
* **Пространственная сложность:** Пространственная сложность составляет *O*(), так как используется двумерный массив board для хранения текущего состояния стола, а также вектор result для хранения текущей расстановки квадратов.

Оптимизации

1. **Пропуск заведомо неоптимальных решений**:
   * Если количество использованных квадратов (*squaresUsed*) превышает текущий минимум (*minSquares*), алгоритм прекращает дальнейший поиск по этой ветке.
   * Для больших *N* (больше 7) введено ограничение на количество квадратов размером 1×1 (переменная *oneCount*). Если их количество превышает *N*/4, алгоритм прекращает поиск по этой ветке.
2. **Ограничение на размер квадрата**:
   * В начальной позиции (0, 0) размер квадрата ограничен значением (*N*−1)/2, чтобы избежать избыточного перебора.
3. **Использование бэктрекинга**:
   * После размещения квадрата алгоритм рекурсивно вызывает себя для следующей позиции. Если дальнейшее размещение невозможно, квадрат удаляется, и алгоритм пробует другой размер.

Код программы

Программа состоит из следующих функций и структур:

**Глобальные переменные**

* *int N* — размер стола.
* *vector<vector<int>> board* — двумерный массив, представляющий стол. Значение 0 означает пустую клетку, 1 — занятую.
* *vector<vector<int>> result* — текущая расстановка квадратов.
* *vector<vector<int>> bestResult* — лучшая найденная расстановка.
* *int minSquares* — минимальное количество квадратов, необходимых для заполнения стола.
* *int oneCount* — количество квадратов размером 1×1.

**Функции**

1. *canPlace(int x, int y, int w):*
   * Проверяет, можно ли разместить квадрат размером *n*×*n* в позиции (x,y)(*x*,*y*).
   * Возвращает true, если это возможно, и false в противном случае.
2. *placeSquare(int x, int y, int w):*
   * Размещает квадрат размером *n*×*n* в позиции (x,y)(*x*,*y*), заполняя соответствующие клетки в массиве *board* значением 1.
3. *removeSquare(int x, int y, int w):*
   * Удаляет квадрат размером *n*×*n* из позиции (x,y)(*x*,*y*), восстанавливая значение 0 в соответствующих клетках массива *board*.
4. *backtrack(int x, int y, int squaresUsed):*
   * Основная функция рекурсивного бэктрекинга.
   * Перебирает все возможные размеры квадратов и пытается разместить их в текущей позиции.
   * Если размещение возможно, рекурсивно вызывает себя для следующей позиции.
   * Если текущая расстановка завершена (все клетки заполнены), обновляет *minSquares* и *bestResult*.
5. *main():*
   * Считывает размер стола *N*.
   * Инициализирует массив *board* и запускает алгоритм бэктрекинга.
   * Выводит минимальное количество квадратов и их координаты.

# Тестирование

Программа была протестирована на различных входных данных, с учётом ограничения на размер стола ().

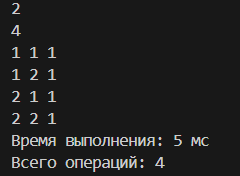


Рисунок 1 – Вывод программы для граничного случая n = 2.

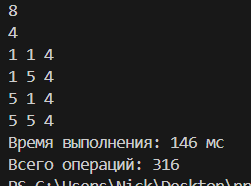


Рисунок 2 – Вывод программы для n = 8.

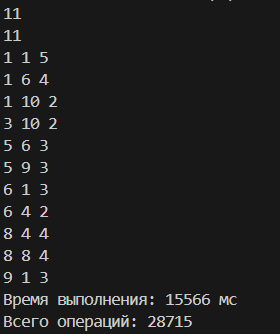


Рисунок 3 – Вывод программы для n = 11.

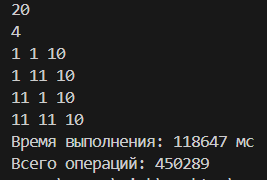


Рисунок 4 – Вывод программы для n = 19.

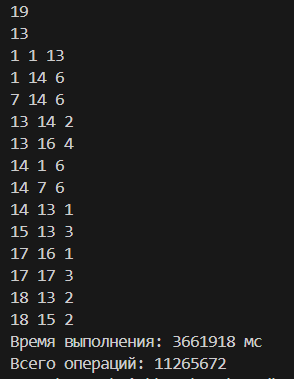


Рисунок 5 – Вывод программы для n = 20.

# Исследование

Вар. 4р. Рекурсивный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

Таблица 1. Результаты полученных данных.

|  |  |  |  |
| --- | --- | --- | --- |
| Размер стороны | Число операций | Затраченное время (мкс) | Число квадратов |
| 2 | 4 | 5 | 4 |
| 3 | 6 | 15 | 6 |
| 4 | 33 | 44 | 4 |
| 5 | 239 | 69 | 8 |
| 6 | 96 | 72 | 4 |
| 7 | 1769 | 467 | 9 |
| 8 | 316 | 105 | 4 |
| 9 | 537 | 219 | 6 |
| 10 | 1458 | 402 | 4 |
| 11 | 28715 | 7858 | 11 |
| 12 | 1233 | 375 | 4 |
| 13 | 143791 | 46975 | 11 |
| 14 | 34367 | 10030 | 4 |
| 15 | 9147 | 3250 | 6 |
| 16 | 46579 | 12511 | 4 |
| 17 | 3024481 | 934376 | 12 |
| 18 | 150147 | 48225 | 4 |
| 19 | 11265672 | 3693579 | 13 |
| 20 | 450289 | 117715 | 4 |

# Отобразим на графике результаты только для простых чисел, поскольку с ними возникает больше всего трудностей.

# 

# Рисунок 6 – Зависимость числа операций от размера стола.

# Можно сделать следующие выводы по исследованию:

1. Число операций растёт экспоненциально.
2. Время, необходимое для вычисления расстановки для простой стороны стола, также растёт крайне быстро.

# Выводы

Во время выполнения лабораторной работы, была реализована программа, выполняющая рекурсивный бэктрекинг для поиска оптимального числа квадратов для покрытия стола. Было выявлено, что число операций зависит от размера поля и растёт экспоненциально.

**ПРИЛОЖЕНИЕ**

**ИСХОДНЫЙ КОД ПРОГРАММЫ**

Имя файла: main.cpp

#include <algorithm>

#include <iostream>

#include <vector>

#include <chrono>

#include <windows.h>

using namespace std;

using namespace std::chrono;

int N;

vector<vector<int>> board;

vector<vector<int>> result;

vector<vector<int>> bestResult;

int minSquares = 400;

int oneCount = 0;

unsigned long long operationsCount = 0;

bool canPlace(int x, int y, int 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 (board[i][j] != 0)

            return false;

        }

    }

    return true;

}

void placeSquare(int x, int y, int w) {

    operationsCount++;

    for (int i = x; i < x + w; ++i) {

        for (int j = y; j < y + w; ++j) {

        board[i][j] = 1;

        }

    }

}

void removeSquare(int x, int y, int w) {

    // operationsCount++;

    for (int i = x; i < x + w; ++i) {

        for (int j = y; j < y + w; ++j) {

        board[i][j] = 0;

        }

    }

}

void backtrack(int x, int y, int squaresUsed) {

    // operationsCount++;

    if (squaresUsed >= minSquares)

        return;

    if (N > 7) {

        if (oneCount > (N / 4)) {

        return;

        }

    }

    if (x == N) {

        minSquares = squaresUsed;

        bestResult = result;

        return;

    }

    if (y == N) {

        backtrack(x + 1, 0, squaresUsed);

        return;

    }

    if (board[x][y] != 0) {

        backtrack(x, y + 1, squaresUsed);

        return;

    }

    for (int w = min(N - x, N - y); w >= 1; --w) {

        if (x == 0 && y == 0 && w < (N - 1) / 2) {

            break;

        }

        if (w == N)

        continue;

        if (canPlace(x, y, w)) {

        placeSquare(x, y, w);

        result.push\_back({x + 1, y + 1, w});

        if (w == 1) {

            oneCount++;

        }

        backtrack(x, y + w, squaresUsed + 1);

        result.pop\_back();

        if (w == 1) {

            oneCount--;

        }

        removeSquare(x, y, w);

        }

    }

}

int main() {

    SetConsoleOutputCP(CP\_UTF8);

    SetConsoleCP(CP\_UTF8);

    cin >> N;

    board.resize(N, vector<int>(N, 0));

    auto start = high\_resolution\_clock::now();

    backtrack(0, 0, 0);

    auto stop = high\_resolution\_clock::now();

    auto duration = duration\_cast<microseconds>(stop - start);

    cout << minSquares << endl;

    for (auto& sq : bestResult) {

        cout << sq[0] << " " << sq[1] << " " << sq[2] << endl;

    }

    cout << "Время выполнения: " << duration.count() << " мс" << endl;

    cout << "Всего операций: " << operationsCount << endl;

    return 0;

}