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 3и.

Студентка гр. 3343 _____ Ермолаева В. А.

Преподаватель _____ Жангиров Т. Р.

Санкт-Петербург

2025

Цель работы

Реализовать алгоритм поиска с возвратом для решения задачи о размещении квадратов на столешнице так, чтобы их количество было минимально возможным.

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.

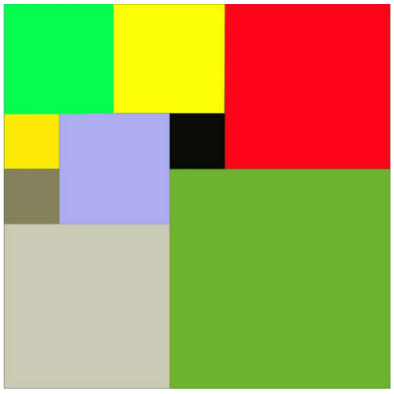

Рис. 1 - Пример

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.

Вариант 3и. Итеративный бэктрекинг. Исследование кол-ва операций от размера квадрата.

Выполнение работы

Для решения задачи был применен алгоритм поиска с возвратом (итеративный) для перебора всех возможных вариантов размещения квадратов и нахождения среди них решения. Алгоритм выбирает из массива первый хранящийся в ней вариант заполнения, ищет на нем первую пустую клетку, пробует разместить в ней квадраты разных размеров, начиная с наибольшего возможного, и, если решение не найдено, добавляет новые варианты заполнений в массив, а текущий удаляет.

Хранение промежуточных решений осуществляется путем использования массива, хранящего текущие варианты заполнения столешницы и работающего по принципу FIFO. Каждое отдельное состояние доски хранится в структуре Board. Описание структуры представлено ниже.

Временная сложность алгоритма зависит от количества возможных размещений квадратов, а также примененных оптимизаций. В худшем случае перебираются все варианты решения и сложность будет экспоненциальной. В лучшем же, как например когда размер столешницы четное число, сложность будет O(1).

Пространственная сложность алгоритма также зависит от того, сколько вариантов различных размещений перебирается, т. к. все они хранятся в массиве. Таким образом, затраты по памяти также будут в худшем экспоненциальны, а в лучшем - $O(n^2)$ для хранения единственного решения.

Описание реализованных функций и структур:

- struct Square: структура для хранения квадрата.
  - int x, y, size, value: координаты, размер стороны квадрата, значение (цвет).
- struct Board: структура для хранения столешницы.
  - int boardSize: размер стороны столешницы
  - int squareCount: количество квадратов
  - int** board: столешница

    - Square squares[40]: квадраты
- Board* createBoard(int size): создание новой доски с выделением под нее памяти.
- void freeBoard(Board* board): освобождение памяти, занимаемой одной доской.
- Board* createBoardCopy(Board* origBoard): возвращает новую доску, являющуюся копией переданной в качестве аргумента.
- void copyBoardValue(Board* origBoard, Board* destBoard): копирует все значения одной доски в другую.
- bool checkFilled(Board* board): проверяет, заполнена ли доска полностью.
- void findEmptyCell(Board* board, int* x, int* y): передает в аргументы координаты первой свободной ячейки доски, начиная с левого верхнего угла.
- bool checkSpace(Board* board, Square square): проверяет, можно ли поместить квадрат на доску.
- void placeSquare(Board* board, Square square): размещает квадрат на доске.
- void printBoard(Board* board): выводит доску в консоль.
- void printSquares(Board* board): выводит в консоль квадраты, заполняющие доску.
- void placeSquaresForPrime(Board* board): размещает квадраты для столешницы, размер стороны которой простое число.
- void placeSquaresForEven(Board* board): размещает квадраты для столешницы, размер стороны которой четное число.
- void placeSquaresForThree(Board* board): размещает квадраты для столешницы, размер стороны которой число, делящееся на 3.
- void backtracking(Board* board): поиск с возвратом.

Описание оптимизаций алгоритма:

1) Если размер столешницы четное число, то заранее известно, что оптимальным будет заполнение четырьмя квадратами со сторонами boardSize/2.

2) Если размер столешницы - число, которое делится на 3, то оптимальное заполнение включает в себя 2 маленьких квадрата в правом верхнем и левом нижнем углах, размер которых - boardSize / 3, и один большой в левом верхнем углу размера boardSize - boardSize / 3.

3) Если размер столешницы не четное число и не делится на 3, то это простое число и в таком случае оптимальное заполнение - 1 большой квадрат в левом верхнем углу размером (boardSize + 1) / 2 и 2 маленьких в правый верхний, левый нижние углы размера (boardSize - 1) / 2.

4) Расстановка квадратов начиная с наибольшего размера. Позволяет найти оптимальный алгоритм быстрее, не требуется перебирать все варианты.

5) Расстановка квадратов начинается с левой верхней клетки, сокращая количество расстановок.

Исходный код программы смотреть в приложении А.

Исследование

Исследуем скорость работы алгоритма на разных размерах столешницы, а также количество операций для определения того, как изменение размера столешницы влияет на производительность алгоритма. Было выбрано 10 размеров столешницы от 2 до 20. Результаты исследования представлены в табл. 1.

Табл. 1. Исследование работы алгоритма.

№	Размер столешницы	Время, с	Операции, шт.
1	2	0.000001	4
2	5	0.000281	36
3	7	0.000776	137
4	9	0.000760	38
5	12	0.000002	4

6	13	0.011470	4367
7	15	0.000825	94
8	18	0.000004	4
9	19	0.363216	86632
10	20	0.000005	4

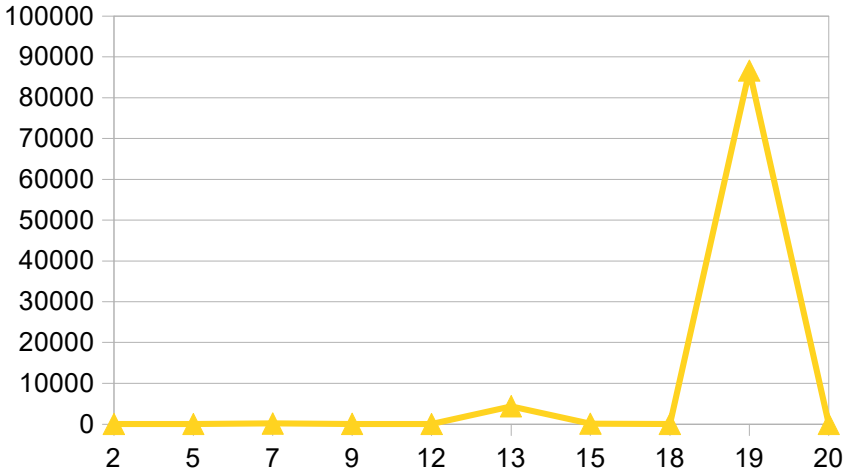

Рис. 1 - Зависимость количества операций от размера квадрата.

Из графика становится понятно, что благодаря примененным оптимизациям алгоритма его количество операций остается на большем количестве размеров весьма низким. Однако несмотря на это оно сильно растет на простых числах даже с использованием оптимизаций, показывая худшую производительность на размерах 13 и 19.

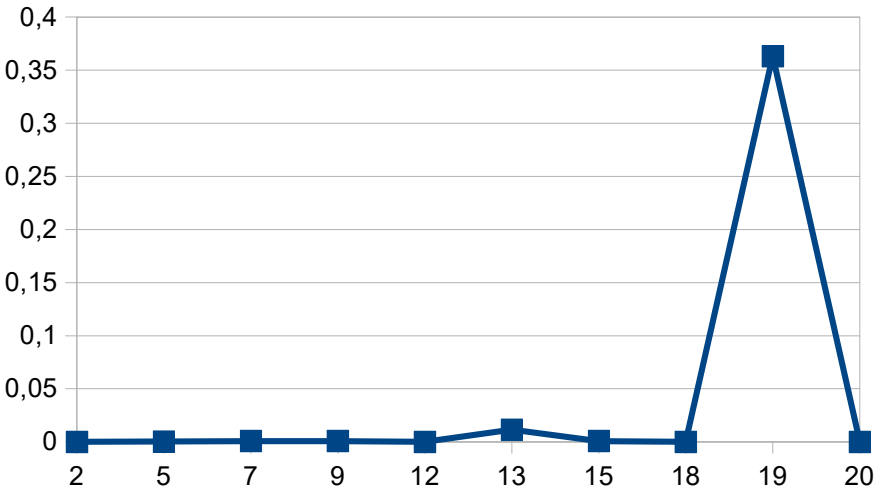

Рис. 2 - Зависимость времени заполнения от размера квадрата.

График времени работы алгоритма повторяет график зависимости количества операций от размера квадрата. Быстрее всего алгоритм работает на четных числах, после которых по скорости идут числа, делящиеся на 3.

Тестирование

Результаты тестирования представлены в таблице 1.

Табл. 1. – Результаты тестирования

№ п/п	Входные данные	Выходные данные	Комментарии
1.	2	4 0 0 1 0 1 1 1 0 1 1 1 1	Результат соответствует ожиданиям.
2.	20	4 0 0 10 0 10 10 10 0 10 10 10 10	Результат соответствует ожиданиям.
3.	11	11 0 0 6 0 6 5 6 0 5 6 5 3 9 5 2 5 6 1 5 7 1 9 7 1 10 7 1 5 8 3 8 8 3	Результат соответствует ожиданиям.
4.	-5		Алгоритм применен не будет, т. к. размер доски не соответствует требованиям.

Выводы

В ходе выполнения лабораторной работы был написан алгоритм заполнения столешницы минимальным числом квадратов с использованием поиска с возвратом. Были применены оптимизации для повышения производительности.

ПРИЛОЖЕНИЕ А

```
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>
#include <sys/time.h>

#define MAX_SIZE 250

typedef struct {
    int x, y, size, value;
} Square;

typedef struct {
    int boardSize;
    int squareCount;
    int** board;
    Square squares[40];
} Board;

Board* createBoard(int size) {
    Board* newBoard = (Board*) malloc(sizeof(Board));
    newBoard->boardSize = size;
    newBoard->squareCount = 0;

    newBoard->board = (int**) malloc(size * sizeof(int*));
    for (int i = 0; i < size; i++) {
        newBoard->board[i] = (int*) calloc(size, sizeof(int));
    }

    return newBoard;
}

void freeBoard(Board* board) {
    for (int i = 0; i < board->boardSize; i++) {
        free(board->board[i]);
    }

    free(board->board);
    free(board);
}

Board* createBoardCopy(Board* origBoard) {
    Board* newBoard = createBoard(origBoard->boardSize);
    newBoard->squareCount = origBoard->squareCount;

    for (int y = 0; y < origBoard->boardSize; y++) {
        for (int x = 0; x < origBoard->boardSize; x++) {
            newBoard->board[y][x] = origBoard->board[y][x];
        }
    }

    for (int i = 0; i < origBoard->squareCount; i++) {
        newBoard->squares[i] = origBoard->squares[i];
    }

    return newBoard;
}

void copyBoardValue(Board* origBoard, Board* destBoard) {
    destBoard->squareCount = origBoard->squareCount;

    for (int y = 0; y < origBoard->boardSize; y++) {
```

```
            for (int x = 0; x < origBoard->boardSize; x++) {
                destBoard->board[y][x] = origBoard->board[y][x];
            }
        }

        for (int i = 0; i < origBoard->squareCount; i++) {
            destBoard->squares[i] = origBoard->squares[i];
        }
    }

    bool checkFilled(Board* board) {
        for (int y = 0; y < board->boardSize; y++) {
            for (int x = 0; x < board->boardSize; x++) {
                if (board->board[y][x] == 0) {
                    return false;
                }
            }
        }
        return true;
    }

    void findEmptyCell(Board* board, int* x, int* y) {
        for (int Y = 0; Y < board->boardSize; Y++) {
            for (int X = 0; X < board->boardSize; X++) {
                if (board->board[Y][X] == 0) {
                    *x = X; *y = Y;
                    return;
                }
            }
        }

        *x = -1; *y = -1;
    }

    bool checkSpace(Board* board, Square square) {
        if (square.x + square.size > board->boardSize || square.y +
square.size > board->boardSize) {
            return false;
        }

        for (int y = 0; y < square.size; y++) {
            for (int x = 0; x < square.size; x++) {
                if (board->board[y + square.y][x + square.x] > 0) {
                    return false;
                }
            }
        }

        return true;
    }

    void placeSquare(Board* board, Square square) {
        for (int y = 0; y < square.size; y++) {
            for (int x = 0; x < square.size; x++) {
                board->board[y + square.y][x + square.x] = board->squareCount
+ 1;
            }
        }

        board->squares[board->squareCount++] = square;
    }

    void printBoard(Board* board) {
```

```
        for (int y = 0; y < board->boardSize; y++) {
            for (int x = 0; x < board->boardSize; x++) {
                printf("%d ", board->board[y][x]);
            }
            printf("\n");
        }
    }

    void printSquares(Board* board) {
        printf("%d\n", board->squareCount);
        for (int i = 0; i < board->squareCount; i++) {
            printf("%d %d %d\n", board->squares[i].x, board->squares[i].y,
board->squares[i].size);
        }
    }

    void placeSquaresForPrime(Board* board) {
        Square s1 = {0, 0, (board->boardSize + 1) / 2};
        Square s2 = {0, (board->boardSize + 1) / 2, (board->boardSize - 1) /
2};
        Square s3 = {(board->boardSize + 1) / 2, 0, (board->boardSize - 1) /
2};

        placeSquare(board, s1);
        placeSquare(board, s2);
        placeSquare(board, s3);
    }

    void placeSquaresForEven(Board* board) {
        Square s1 = {0, 0, board->boardSize / 2};
        Square s3 = {0, board->boardSize / 2, board->boardSize / 2};
        Square s2 = {board->boardSize / 2, 0, board->boardSize / 2};
        Square s4 = {board->boardSize / 2, board->boardSize / 2, board-
>boardSize/ 2};

        placeSquare(board, s1);
        placeSquare(board, s2);
        placeSquare(board, s3);
        placeSquare(board, s4);
    }

    void placeSquaresForThree(Board* board) {
        int size = board->boardSize - board->boardSize / 3;
        Square s1 = {0, 0, size};
        Square s3 = {0, size, board->boardSize / 3};
        Square s2 = {size, 0, board->boardSize / 3};

        placeSquare(board, s1);
        placeSquare(board, s2);
        placeSquare(board, s3);
    }

    void backtracking(Board* board) {
        Board* allBoards[MAX_SIZE];
        int x, y;
        int allBoardsCount = 0;
        int operationCount = 0;
        allBoards[allBoardsCount++] = board;

        while (allBoardsCount > 0) {
            findEmptyCell(allBoards[0], &x, &y);
            Board* currentBoard = createBoardCopy(allBoards[0]);
```

```
            printf("Going through options for the following board:\n");
            printBoard(currentBoard);

            for (int size = board->boardSize - 1; size > 0; size--) {
                Square square = {x, y, size, currentBoard->squareCount};
                Board* newBoard = createBoardCopy(currentBoard);
                operationCount++;

                if (checkSpace(newBoard, square)) {
                    placeSquare(newBoard, square);

                    printf("Possible variant:\n");
                    printBoard(newBoard);

                    if (allBoardsCount < MAX_SIZE) {
                        allBoards[allBoardsCount++] =
createBoardCopy(newBoard);
                    }

                    if (checkFilled(newBoard)) {
                        printf("Amount of operations required to fill a square
of size [%d] is [%d].\n", board->boardSize, operationCount);
                        copyBoardValue(newBoard, board);
                        for (int i = 0; i < allBoardsCount; i++) {
                            freeBoard(allBoards[i]);
                        }
                        freeBoard(newBoard);
                        freeBoard(currentBoard);
                        return;
                    }
                }

                freeBoard(newBoard);
            }

            freeBoard(currentBoard);
            for (int i = 0; i < allBoardsCount - 1; i++) {
                allBoards[i] = allBoards[i + 1];
            }
            allBoardsCount--;
        }
    }

    int main() {
        int size;
        scanf("%d", &size);
        if (size >= 2 && size <= 40) {
            Board* board = createBoard(size);
            struct timeval stop, start;

            gettimeofday(&start, NULL);

            if (size % 2 == 0) {
                placeSquaresForEven(board);
            }

            else if (size % 3 == 0) {
                placeSquaresForThree(board);
                backtracking(board);
            }

            else {
                placeSquaresForPrime(board);
```

```
                backtracking(board);
            }

            gettimeofday(&stop, NULL);

            double seconds = (stop.tv_sec - start.tv_sec) + (stop.tv_usec -
start.tv_usec) / 1000000.0;
            printf("Filling the square of size [%d] took [%.6f] seconds.\n",
board->boardSize, seconds);

            printf("The board was filled with the following squares:\n");
            printBoard(board);
            freeBoard(board);
        }
        return 0;
    }
```