МИНОБРНАУКИ РОССИИ
САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ
ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ
«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)
Кафедра МОЭВМ

ОТЧЕТ

по лабораторной работе №1
по дисциплине «Построение и анализ алгоритмов»
Тема: Поиск с возвратом

Студент гр. 9383 _____ Мосин К.К.

Преподаватель _____ Фирсов М.А.

Санкт-Петербург
2021

Цель работы.

Применить и проанализировать алгоритм поиска с возвратом.

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.

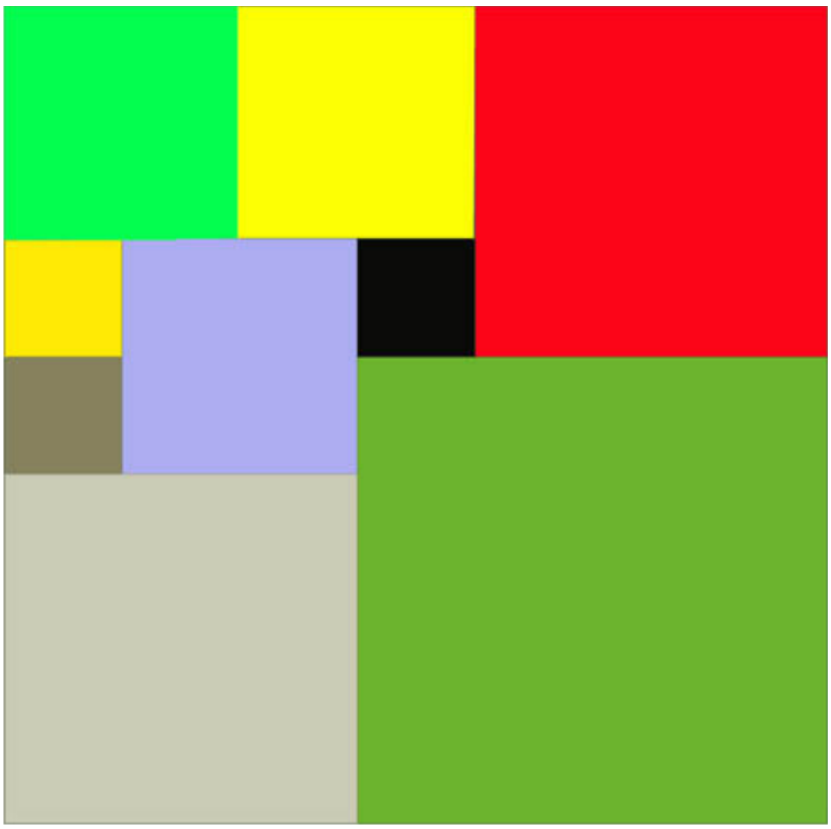

Рис 1. - Пример

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.

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

Вариант 4и. Итеративный бэктрекинг. Расширение задачи на прямоугольные поля, рёбра квадратов меньше рёбер поля. Подсчёт количества вариантов покрытия минимальным числом квадратов.

Выполнение работы.

1. Объявляется макрос N и 4 глобальные переменные:
   - N = 20 - максимальное ребро поля
   - int* mutation[N+N] - текущие квадраты, находящиеся на поле
   - int* answer[N+N] - копия удачной мутации
   - int table[N*N] - поле
   - bool rectangle - переменная, отслеживающая введенную фигуру (квадрат или прямоугольник)

2. Создается массив из трех чисел, символизирующий координаты квадрата и его длину.

3. После вставки квадрата вычисляются координаты для следующего квадрата

4. Если таковых нет и текущее количество квадратов в поле меньше переменной, хранящая минимальное количество возможно вставляемых квадратов, то минимум обновляется, ответ сохраняется.

5. Если текущее количество квадратов больше минимально возможного, выполняется поиск с возвратом.

6. Если поиск с возвратом не удался (означает, что перебор закончен и ответ получен), алгоритм заканчивается.

Улучшения

- Вместо двумерного массива поля используется одномерный.
- Перебор массивов осуществляется не по индексам, а по указателям.

Описание функций

1. bool restore(int*, int, int, int&) - осуществляет поиск с возвратом

2. void fill(const int*, int, int) - заполнение поля

3. bool update(int*, int, int) - поиск координат для вставки

4. int find(int, int, int, int) - поиск ширины для вставки

5. void clean(const int*, int, int) - удаление из поля

6. void copy(int*&, int*, bool) - примитивная функция копии массива A в массив B

7. void answer_delete(int) - функция для контроля памяти.

8. void answer_print(int, int) - печать результатов на экран.

Тестирование.

Результаты тестирования представлены в табл. 1. Основные функции, манипулирующие полем и текущим количеством квадратов - restore и update. Были написаны тесты с использованием библиотеки catch2. Исходный код тестов представлены в приложении В.

Табл. 1 - результаты тестирования

Входные данные	Минимальное число покрытия квадратами	Число покрытий
2 2	4	1
5 7	7	4
15 19	7	16
19 19	13	892

Анализ алгоритма.

Массив, хранящий квадраты, является одномерным и поиск ближайшей свободной координаты оценивается в O(n), где n количество квадратов между последним вставленным и свободной клеткой. Вставка, как и удаление квадрата, оценивается как $O(n^2)$, где n сторона квадрата. Функция поиска с возвратом оценивается как $O(mn^2)$, где m количество удаленных квадратов и n сторона квадрата. Число перестановок квадратов зависит от стороны поля, исключая поля, являющиеся квадратами и имеющие четную сторону (число перестановок всегда равно единицы). Из тестируемых данных видно, что число перестановок растет экспоненциально от стороны поля.

Вывод

В ходе выполнения лабораторной работы был использован алгоритм с возвратом.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: API.h

```
#pragma once

#include <iostream>
#include <ctime>

#define N 20 //maximum edge length

bool restore(int*, int, int, int&); //backtracking current branch solution
void fill(const int*, int, int); //insert square into figure
bool update(int*, int, int); //search coordinates to insert square
int find(int, int, int, int); //search square size to insert
void clean(const int*, int, int); //delete square from figure
void copy(int*&, int*, bool); //primitive copy array A to array B (if boolean param is true array A allocate)
void answer_delete(int); //need to control allocated memory
void answer_print(int, int); //print in console
```

Название файла: API.cpp

```
#include "API.h"

int* mutation[N+N]; //current branch solution
int* answer[N+N]; //need to copy mutation for save the best solution
int table[N*N]; //figure projection
bool rectangle; //boolean to switch moving in an array

bool restore(int* square, int width, int height, int& count) {
```

```
    int w = std::min(width, height);

    while (count) {
        copy(square, mutation[count - 1], false);
        delete [] mutation[count - 1];
        mutation[count - 1] = nullptr;
        count--;
        clean(square, width, height);
        if (!count && square[2] < 2) {
            return false;
        }

        if (square[2] > 1) {
            square[2]--;
            return true;
        }

    }
    return false;
}

void fill(const int* square, int width, int height) {
    for (int *i = &table[square[0] + square[1] * width]; i != &table[square[0] + square[1]
* width + square[2]]; ++i) {
        for (int j = 0; j < square[2]; ++j) {
            *(i + j * width) = square[2];
        }
    }
}

bool update(int* square, int width, int height) {
```

```
    int index = square[0] + square[1] * width;

    while (index < height * width) {
        if (!table[index]) {
            square[0] = index % width;
            square[1] = index / width;
            square[2] = find(square[0], square[1], width, height);
            return true;
        }

        index += table[index];
    }
    return false;
}

int find(int x, int y, int width, int height) {
    int w = std::min(width, height) - 1;
    int min = 1;
    while (min <= w) {

        if (x + min > width || y + min > height) {
            return min - 1;
        }

        if (rectangle) {
            for (int *i = &table[x + y * width]; i != &table[x + y * width + min]; ++i) {
                for (int j = 0; j < min; ++j) {
                    if (*(i + j * width)) {
                        return min - 1;
                    }
                }
```

```
            }
        }

        else if (table[x + y * width + min - 1] || table[x + y * width + (min - 1) * width] ||
table[x + y * width + min - 1 + (min - 1) * width]) {
            return min - 1;
        }

        min++;
    }

    return min - 1;
}

void clean(const int* square, int width, int height) {
    for (int *i = &table[square[0] + square[1] * width]; i != &table[square[0] + square[1]
* width + square[2]]; ++i) {
        for (int j = 0; j < square[2]; ++j) {
            *(i + j * width) = 0;
        }
    }
}

void copy(int*& a, int* b, bool memory) {
    if (memory) {
        if (a) {
            delete [] a;
        }
        a = new int[3];
    }
    for (int i = 0; i < 3; ++i) {
```

```
        a[i] = b[i];
    }
}

void answer_delete(int count) {
    if(!answer[0]) {
        return;
    }

    for (int i = 0; i < count; ++i) {
        delete [] answer[i];
        answer[i] = nullptr;
    }
}

void answer_print(int square_count, int solve_count) {
    std::cout << "minimal square count: " << square_count << std::endl;
    std::cout << "{x,y,w}" << std::endl;
    for (int i = 0; i < square_count; ++i) {
        std::cout << "{" << answer[i][0] + 1 << "," << answer[i][1] + 1 << "," <<
answer[i][2] << "}" << std::endl;
        delete [] answer[i];
    }

    std::cout << "variation count: " << solve_count << std::endl;
    std::cout << "time = " << clock()/1000000.0 << " sec" << std::endl;
}
```

Название файла: main.cpp

```
#include "API.h"

extern int* mutation[N+N];
extern int* answer[N+N];
extern int table[N*N];
extern bool rectangle;

int main(int argc, char *argv[]) {
    int width, height;
    std::cin >> width >> height;
    if (width < 2 || width > 20 || height < 2 || height > 20) {
        std::cout << "BAD CONFIGURATION: 1 < width,height < 21" << std::endl;
        std::exit(1);
    }

    int min = width * height;
    int solve_count = 0;

    if (width < height) {
        std::swap(width, height);
    }

    width == height ? rectangle = false, min = 17 : rectangle = true;

    int square[3] = {0, 0, std::min(width, height) - 1};
    /*  square[0] = x
        square[1] = y
        square[2] = width */

    int count = 0;
```

```
    for(;;) {
        if (count >= min && !restore(square, width, height, count)) {
            break;
        }
        fill(square, width, height);
        copy(mutation[count++], square, true);
        if (!update(square, width, height) && count <= min) {
            count < min ? answer_delete(min), min = count, solve_count = 1 :
solve_count++;
            for (int i = 0; i < min; ++i) {
                copy(answer[i], mutation[i], true);
            }
        }
    }
    answer_print(min, solve_count);

    return 0;
}
```

ПРИЛОЖЕНИЕ В
ТЕСТЫ

Название файла: test.cpp

```
#define CATCH_CONFIG_MAIN
#include "catch.hpp"

#include "API.h"

extern int* mutation[N+N];
extern int* answer[N+N];
extern int table[N*N];
extern bool rectangle;

TEST_CASE("Restore branch solution function") {
    int M = 3;
    int square[3] = {0,0,0};
    int count = 0;
    SECTION("return false - stop squaring") {
        square[2] = 1;
        fill(square, M, M);
        copy(mutation[count++], square, true);
        REQUIRE(restore(square, M, M, count) == false);
    }
    SECTION("return true - restart squaring with less square at upper left corner") {
        square[2] = 2;
        fill(square, M, M);
        copy(mutation[count++], square, true);
        REQUIRE(restore(square, M, M, count) == true);
    }
}
```

```
TEST_CASE("Search new coordinates to insert square") {
    int M = 3;
    int square[3] = {0,0,0};
    SECTION("return false - no there to insert") {
        square[2] = 3;
        fill(square, M, M);
        REQUIRE(update(square, M, M) == false);
        clean(square, M, M);
    }
    SECTION("return true - coordinates was found") {
        square[2] = 2;
        fill(square, M, M);
        REQUIRE(update(square, M, M) == true);
        clean(square, M, M);
    }
}
```