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. 1304 Байков Е.С.

Преподаватель Шевелева А.М.

Санкт-Петербург
2023

Цель работы.

Изучение алгоритма поиска с возвратом и реализация с его помощи программы, которая решает задачу 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 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 ($2 \le N \le 40$).

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).

Основные теоретические положения.

Поиск с возвратом — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве. М.

Решение задачи методом поиска с возвратом сводится к последовательному расширению частичного решения. Если на очередном шаге такое расширение провести не удается, то возвращаются к более короткому частичному решению и продолжают поиск дальше.

Выполнение работы.

Алгоритм с возвратом в данном случае основывается на поиске в глубину. Осуществляется заполнение столешницы до того момента, пока нельзя будет вставить квадрат текущего размера. Как только вставка фрагмента заданного размера не возможна, то осуществляется откат на предыдущий шаг и происходит попытка вставки квадрата размером меньше. Как только столешница становится заполненной, происходит сравнение со значением полученном на предыдущем шаге, если оно меньше чем уже имеющееся, то ответ перезаписывается.

Оптимизация

При решении задания лабораторной работы была необходима оптимизация алгоритма. В ходе рассуждений были выявлены следующие оптимизации:

1) Разбиение столешницы со стороной равной составному числу такое же как и разбиение столешницы со стороной равной минимальному простому делителю исходной. Для примера можно взять столешницу со стороной 15, для которой минимальное разбиение равно 6 квадратам, как и для ее минимального делителя – 3.

2) Из первого утверждения следует, что для всех столешниц со стороной равной четному числу минимальным разбиением будет разбиение на четыре равных квадрата, поэтому можно даже не рассматривать их в поиске, а лишь выводить разбиение для столешницы 2x2 увеличенное в половину длины исходной столешницы.

3) Было выявлено что для всех простых чисел, не превышающих 40 разбиение, содержит в себе один квадрат равный *(length + 1) / 2* и два квадрата со стороной *(length - 1) / 2*, где *length* – длина стороны столешницы. Поэтому их можно добавлять по умолчанию.

4) Пользуясь методом ветвей и границ, было выявлено, что можно не рассматривать разбиения, где количество квадратов превышает уже имеющегося числа квадратов.

Структура

Далее описаны структуры и методы, используемые в коде программы для решения поставленной задачи.

1) Структура *Square* необходима для удобного написания кода, в ней содержится информация о квадрате, который был помещен в столешницу, а именно координаты его верхнего левого угла *(x, y)* и размер квадрата – *size*.

2) Класс *Solution*, содержащий в себе все поля и методы необходимые для решения задачи. Все поля имеют модификатор доступа *private*, а все методы *public*.

Поля класса *Solution*:

1) *answer_count* – равный по умолчанию максимальному значению тепа *int*. Хранит в себе число равное минимальному разбиению столешницы.

2) *current_answer_count* – равный по умолчанию 0. Параметр который будет меняться в процессе поиска минимального разбиения, хранит текущее количество квадратов в разбиении.

3) *length_side* сторона квадрата, для которого осуществляется поиск разбиения.

4) *kMaxCount* – максимально возможное разбиение для всех чисел меньше 40.

5) *current_answer* – массив с квадратами текущего разбиения.

6) *answer* – массив с квадратами конечного разбиения, хранящий ответ на задачу.

7) *scheme* – схема столешницы в виде двумерного массива *bool*. Значение *true* означает что данная точка занята, каким-либо квадратом.

8) scale – переменная хранящая в себе значение для скалирования размеров разбиения для столешницы со стороной равной составному числу.

Методы класса *Solution*:

1) *Solution* – конструктор инициализирующий поля класса *length_side*, *scale*. Принимает на вход соответственно *length* и *scale*. Второй параметр по умолчанию равен 1.

2) *solve* – основной метод, реализующий решение задачи. Первым делом в методе происходит проверка длины входной столешницы на четность. Если длина четна, то вызывается метод *print_even_partition*, которому передается длина столешницы и далее происходит вывод и программа завершается. Если же длина столешницы нечетна, то происходит поиск минимального делителя (метод *find_minimal_divisor*) стороны столешницы, значение которого записывается в *divisor*. Если значение *divisor* равно единице, то считается что длина столешницы является простым числом. При ином исходе, создается новый объект *Solution*, в конструктор которого передается размер, равный значению *divisor* и соответственно *scale*, равный максимальному делителю длины столешницы. Если же длиной столешницы является простое число, тогда происходит создание схемы *scheme* размера *length_side*. Далее выделяется память под текущее разбиение *current_answer* и под конечное разбиение *answer* размером *kMaxCount*. Затем происходит заполнение первыми тремя квадратами (см. Оптимизация п.3). После вызывается метод *backtrack* в который передается длина максимального квадрата в разбиении минус один. Процесс заполнения происходит по принципу высоты, т.е. заполнение начинается с самой верхней пустой точки. После отработки метода *backtrack* программа вызывает метод *print_odd_partition()*, выводящий ответ на задачу в необходимом формате, а затем происходит отчистка всех выделенных участков памяти.

3) *print_even_partition* – метод выводит количество, расположение и размер квадратов для столешницы с четной стороной.

4) *print_odd_partition* – метод выводящий разбиение для нечетного числа. Сначала происходит вывод *answer_count* с помощью функции *printf*, а затем поочередно выводит значения квадратов из массива *answer* в порядке: координата *x* координата *y* размер.

5) *find_minimal_divisor* – метод осуществляет поиск минимального делителя числа. С помощью цикла *for* проверяется делимость входного числа на числа от 2 до корня числа включая последний. Метод возвращает первый найденный делитель, либо 1, если число оказалось простым.

6) *add_square* – метод реализует добавление квадрата в схему с помощью циклов *for*, а также добавление структуры в *current_answer*. В точках, которые занимает квадрат, устанавливается значение true.

7) *backtracking* – принимает на вход «высоту» с которой начнется заполнение. Сначала происходит проверка на выход высоты за пределы размеров столешницы. При выходе за пределы сравниваются значения *current_answer_count* и *answer_count* и минимальное сохраняется в *answer_count*, а также происходит сохранение разбиения в *answer*. И происходит завершение метода. В случае с высотой не превосходящей пределы столешницы, происходит первой свободной точки на данной высоте, т.е. первой свободной координаты x, с помощью метода *find_free_x*. Если значение переменной *free_x_coordinate* равно -1 тогда происходит переход на следующую «высоту», т.е. *backtracking(height + 1)*. Затем происходит проверка текущего значения количества квадратов (*current_answer_count*) с максимально возможным количеством (*kMaxCount*) и текущим количеством, подаваемым для ответа (*answer_count*). При превышении или равенстве, происходит выход из метода. Затем в переменную *size* записывается максимальный размер квадрата, который можно поместить в разбиение и от него начинается просмотр в цикле *for*. В цикле проверяется возможность вставки квадрата размера *size* в точку (*free_x_coordinate*, *height)*. Если есть такая возможность происходит вставка и вновь вызывается метод backtracking от текущей высоты. После него происходит отчистка значений только что добавленного квадрата (метод *clear_square*), что позволяет программе возвращаться на шаг назад, для просмотра всех возможных значений.

8) *find_free_x* – метод принимает на вход текущую «высоту» и осуществляет поиск первого значения схемы равного *false* на данной «высоте». Возвращает данное значение, если такое существует, иначе -1.

9) *can_place* – метод проверяет возможность вставки квадрата размером *size* в точку с координатами (*x_coordinate, y_coordinate)*. Если хотя бы одно

значение внутри области от (*x_coordinate, y_coordinate)* до (*x_coordinate + size, y_coordinate + size)* является true то метод возвращает *false* иначе возвращает *true*.

10) *clear_square* - производит удаление квадрата из области описанной в пункте 9, а также уменьшает количество квадратов *current_answer_count*.

Разработанный программный код смотреть в приложении А.

Выводы.

Был изучен алгоритм поиска с возвратом и осуществлена его реализация на языке программирования C++ для решения задачи построения столешницы с помощью минимального числа квадратных обрезков.

Для ускорения работы программы были использованы следующие оптимизации:

1) Проверка делимости числа на 2.
2) Поиск меньшей столешницы с тем же разбиением для составного числа.
3) Использование метода ветвей и границ.

Подробное описание данных оптимизаций представлено в отчете.

Для решения задачи была создана структура *Square,* которая отвечает за параметры размещаемого квадрат. Также был создан класс *Solution*, с его помощью происходит полное решение задачи. Метод *solve* осуществляет основную функцию выделения памяти и оптимизации решения. Полный перебор осуществляется с помощью метода *backtracking*.

Сам алгоритм поиска с возвратом был реализован на основе поиска в глубину. При этом использовалась проверка на превышение значения решения, найденного ранее, что позволило ускорить поиск, не перебирая заведомо неверный результат.

Разработанный программный код для решения поставленной задачи успешно прошел тестирование на онлайн платформе *Stepik* при ограничении параметра *N* от 2 до 40 включительно.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <climits>
#include <iostream>
#include <algorithm>
#include <cmath>
#include <cstring>

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

class Solution {
public:
    Solution(int length, int scale = 1) : length_side(length),
scale(scale) {}
        void solve() {
            if (length_side % 2 == 0) {
                print_even_partition(length_side);
                return;
            }

            int divisor = find_minimal_divisor(length_side);
            if (divisor != 1) {
                Solution inner_solution{ divisor, length_side / divisor };
                inner_solution.solve();
                return;
            }

            scheme = new bool* [length_side];
            for (int i = 0; i < length_side; ++i) {
                scheme[i] = new bool[length_side] {};
            }
            current_answer = new Square[kMaxSize]{};
            answer = new Square[kMaxSize]{};
            int first_square_size = (length_side + 1) / 2;
            add_square(0, 0, first_square_size);
            add_square(0, first_square_size, first_square_size - 1);
            add_square(first_square_size, 0, first_square_size - 1);

            backtrack(first_square_size - 1);
            print_odd_partition();
            for (int i = 0; i < length_side; ++i) {
                delete[] scheme[i];
            }
            delete[] scheme;
            delete[] answer;
            delete[] current_answer;
        }

        void print_even_partition(int length) {
```

```
            printf("%d\n", 4);
            printf("%d %d %d\n", 1, 1, length / 2);
            printf("%d %d %d\n", 1, 1 + length / 2, length / 2);
            printf("%d %d %d\n", 1 + length / 2, 1, length / 2);
            printf("%d %d %d\n", 1 + length / 2, 1 + length / 2, length
/ 2);
        }

        void print_odd_partition() {
            printf("%d\n", answer_count);
            for (int i = 0; i < answer_count; ++i) {
                auto& s = answer[i];
                printf("%d %d %d\n", s.x * scale + 1, s.y * scale + 1,
s.size * scale);
            }
        }

        int find_minimal_divisor(int divisible) {
            for (int i = 2; i <= int(std::sqrt(divisible)); ++i) {
                if (divisible % i == 0) {
                    return i;
                }
            }
            return 1;
        }

        void add_square(int x_coordinate, int y_coordinate, int size) {
            for (int i = y_coordinate; i < y_coordinate + size; ++i) {
                for (int j = x_coordinate; j < x_coordinate + size; ++j)
{
                    scheme[i][j] = true;
                }
            }
            current_answer[current_answer_count++]  =  {  x_coordinate,
y_coordinate, size };
        }

        void backtrack(int height) {
            if (height >= length_side) {
                if (current_answer_count < answer_count) {
                    answer_count = current_answer_count;
                    std::memcpy(answer, current_answer, sizeof(Square) *
answer_count);
                }
                return;
            }
            int free_x_coordinate = find_free_x(height);
            if (free_x_coordinate == -1) {
                backtrack(height + 1);
                return;
            }
            if     (current_answer_count     >=     answer_count     ||
current_answer_count >= kMaxSize) {
                return;
            }
            int  size  =  std::min(length_side  -  free_x_coordinate,
length_side - height);
            for (; size > 0; --size) {
```

```
                if (can_place(free_x_coordinate, height, size)) {
                    add_square(free_x_coordinate, height, size);
                    backtrack(height);
                    clear_square(free_x_coordinate, height, size);
                }
            }
        }

        int find_free_x(int current_height) {
            for (int i = 0; i < length_side; i++) {
                if (scheme[current_height][i] == 0) {
                    return i;
                }
            }
            return -1;
        }

        bool can_place(int x_coordinate, int y_coordinate, int size) {
            for (int i = y_coordinate; i < y_coordinate + size; ++i) {
                for (int j = x_coordinate; j < x_coordinate+ size; ++j)
{
                    if (scheme[i][j]) {
                        return false;
                    }
                }
            }
            return true;
        }
        void clear_square(int x_coordinate, int y_coordinate, int size)
{
            for (int i = y_coordinate; i < y_coordinate + size; ++i) {
                for (int j = x_coordinate; j < x_coordinate + size; ++j)
{
                    scheme[i][j] = false;
                }
            }
            --current_answer_count;
        }
    private:
        int answer_count = INT_MAX;
        int current_answer_count = 0;
        int length_side;
        static constexpr int kMaxSize = 15;
        Square* current_answer;
        Square* answer;
        bool** scheme;
        int scale;
    };

    int main() {
        int square_side;
        scanf("%d", &square_side);
        Solution solution{ square_side };
        solution.solve();
        return 0;
    }
```