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

Студентка гр. 1304 _____ Чернякова В.А.

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 ($2 \le N \le 20$).

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 ($1 \le x, y \le N$) 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. M.

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 поиска с возвратом в данном случае основывается на поиске в ширину. Осуществляется перебор вариантов расстановки очередного квадрата. Для каждого частичного решения перебираются все возможные расширения и добавляются в очередь для дальнейших расширений, при этом расширяемое на данном шаге решение удаляется из очереди. Таким образом, первое полученное решение является оптимальным.

Чтобы решить задание лабораторной работы необходима была оптимизация алгоритма. В ходе рассуждений были выявлены следующие оптимизации:

1) Если размер стола не кратен 2 или 3, то заранее можно расставить три квадрата, чтобы существенно сократить перебор. Расставляются следующим образом: в самый верхний левый угол ставится больший квадрат со стороной, вычисляемой как увеличенная на единицу длина стороны стола, деленная на 2. Далее смежные с ними снизу и справа два квадрата будут со стороной длины стороны стола, уменьшенного на единицу и деленного на 2. Такая оптимизация обусловлена тем, что для простых чисел, в том числе 2 и 3, расстановка всегда будет одинаковая. В других же случаях, чтобы уменьшить время работы, квадраты расставляются так, как описано выше.

2) Новый квадрат ставится в самую верхнюю левую клетку. Таким образом сокращается количество расстановок, так как отбрасываются одинаковые расстановки, но с разным порядком размещения квадратов.

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

Структура *Square* содержит поля типа *int*, такие как *size* – длина стороны квадрата, *x_coord* и *y_coord* – местоположение левого верхнего угла квадрата относительно осей x и y соответственно. Далее в теле структуры описан конструктор, который устанавливает значения полей, описанных ранее. Для корректного вывода ответа был перегружен оператор вывода в поток. Использовалось ключевое слово *friend*, так как оператор перегружается внутри

класса, необходимые поля для вывода находятся именно в нем, а сам вывод происходит в другой функции, которая будет описана далее. При перегрузке передается ссылка на объект *std::ostream& os* – поток вывода, а второй параметр оператора – ссылка на константный объекта класса, который надо вывести в поток, а именно *const Square& square*. Осуществляется вывод полей структуры через пробел в следующем порядке: *square.x_coord, square.y_coord, square.size.* Для совместимости с другими операторами переопределяемый оператор возвращает значение параметра *ostream*, в данном случае это переменная *os.*

Частичные решение хранятся в виде объектов класса *Desk* со следующими полями с модификатором доступа *private*: *table* – матрица со значениями, обозначающими занятость клетки (0 – свободно, иное – занято), *len_side* – длина стороны стола, *count_squares* – количество размещенных квадратов*, list_square* – вектор, содержащий указатели на объекты типа *Square.*

Методы класса *Desk* с модификатором доступа public:

1) *Desk(int n):len_side(n).* Конструктор класса, который в качестве аргумента принимает число *n*, обозначающее длину стороны стола. Создается матрица векторов соответственно заданному размеру, значение счетчика размещённых квадратов приравнивается к нулю. Если длина стороны стола не кратна 2 и 3, то применяется метод расстановки квадратов по умолчанию, который будет описан далее. Это необходимо для оптимизации алгоритма.

2) *Desk(*const Desk*& obj):len_side(obj.len_side), count_squares(obj.count_squares).* Конструктор копирования. При реализации алгоритма *backtracking* нельзя скопировать данные объекта, не определив данный конструктор. В качестве аргумента принимает объект данного класса по ссылке и конструирует новый объект, значения полей которого совпадают со значениями полей аргумента.

3) *Desk& operator=(const Desk& obj).* Перегрузка оператора присваивания с копированием. Определяется для того, чтобы присваивать значение экземпляра существующему объекту. Принимает в качестве аргумента параметр

данного класса, передаваемый по константной ссылке, и возвращает ссылку на тип класса.

4) *int get_length_side() const.* Константный метод или же геттер, необходимый для получения доступа к приватному полю класса, а именно *len_side* – длина стороны столешницы. Метод возвращает значение данного поля.

5) *std::vector<Square*> get_square_list () const.* Константный метод или же геттер, необходимый для получения доступа к приватному полю класса, а именно *list_square* – вектор указателей на объекты типа *Square*. Метод возвращает значение данного поля.

6) *int get_count_square() const.* Константный метод или же геттер, необходимый для получения доступа к приватному полю класса, а именно *count_squares* – количество использованных квадратов. Метод возвращает значение данного поля.

7) *void set_default().* Метод, с помощью которого устанавливаются 3 квадрата, если длина стороны стала не кратна 2 и 3. В самый верхний левый угол устанавливается больший квадрат. Его длина равна длине стола, увеличенной на единицу и деленную на 2. Таким образом занимается большая часть стола. Два других устанавливаются соответственно, как смежные с ним справа и снизу, их длина вычисляется как деление на 2 длины сторона стола уменьшенной на единицу. Таким образом, занимается максимально пространство по вертикали и горизонтали, что оптимизирует алгоритм. Добавление квадратов с учетом их месторасположения и размера происходит с помощью метода *add_square(),* который описан в следующем пункте.

8) *void add_square(int size, int x, int y).* Метод размещения квадрата на столешнице. На вход принимается размер квадрата и координаты его левого верхнего угла. В теле метода создается объект структуры *Square* в соответствии с полученными параметрами. С помощью цикла *for* значение матрицы *table[i][j]*, где *i* и *j* – параметры, описывающие в координатах занятое добавленным квадратом пространство, приравнивается к единице, что

демонстрирует занятость клетки квадратом. Добавленный квадрат с помощью метода *push_back()* добавляется в вектор, хранящий использованные квадраты. Счетчик использованных квадратов увеличивается на единицу.

9) *bool is_full()*. Данный метод проверяет, заполнена ли столешница полностью. С помощью цикла *for* просматриваются все элементы матрицы *table*, и, если хотя бы один из них равен нулю, значит, свободное место есть и функция возвращается *false*. Иначе по истечению цикла вернется *true*.

10) *bool can_add(int x, int y, int size)*. Данный метод проверяет, возможно ли добавить на столешницу квадрат с заданными параметрами. В качестве аргументов на вход принимаются координаты левого верхнего угла и размер квадрат. В начале проверяется выход за границы стола, если это происходит, то возвращается *false*. Далее с помощью цикла *for* просматриваются значения матрицы в тех местах, где должен расположиться квадрат, чтобы проверить, не занято ли оно, то есть при выполнении условия *table[i][j] != 0*, вернется *false*, место занято. При успешном завершении цикла вернется *true*.

11) *std::pair<int, int> empty_cell()*. Метод позволяет получить координаты самой верхней левой свободной клетки в виде пары. С помощью цикла *for* просматриваются клетки матрицы и первая клетка, значение которой равно нулю, то есть данная клетка свободна, значение ее координат и возвращается. Если таковых клеток нет, то возвращается пара *std::pair<int, int>{-1, -1}*, что сигнализирует об отсутствии искомой свободной клетки.

12) *Desk backtracking(Desk desk)*. Функция, в которой реализуется алгоритм поиска с возвратом на основе очереди. На вход функции в качестве аргумента принимается объект класса *Desk*. Класс *std::queue* представляет собой контейнерный адаптер, предоставляющий функциональные возможности очереди. Создается переменная *queue* соответствующая описанному в предыдущем предложении классу. В очередь с помощью *push()* помещается переданная в качестве аргумента столешница *desk*. Пока первый элемент очереди не является полностью заполненным, проверка данного параметра осуществляется с помощью метода *is_full()*, работает цикл *while*. В переменную

desk_copy типа класса *Desk* копируется значение первого элемента очереди *queue.front()*. В *desk_copy* теперь хранится частичное решение. Для данного решения с помощью метода *empty_cell()* находится пара-значение самой верхней левой свободной клетки и записывается в переменную *empty_cell* типа *std::pair<int, int>*. Далее запускается цикл *for*, который работает от длины стороны стола, уменьшенного на единицу, до единицы. В теле этого цикла создается новый объект типа класса *Desk* – переменная *current*, в которую копируется *desk_copy*. В *current* будет формироваться новое частичное решение. Прежде чем создать новое частичное решение, для *desk_copy* проверяется, возможно ли добавить с помощью метода *can_add()*, в который передаются координаты найденной *emty_cell*, значение которых получается с помощью обращения к полям first и second соответственно, и размер квадрата. Размер квадрата в данном случае равен значению *i*, которое варьируется, начиная с размера столешницы уменьшенной на единицу, до 1, так как стол надо заполнить минимальным количеством квадратов, значит, значение их размера должно быть максимальным. Если добавление такого квадрата на столешницу возможно, то у переменной *current* взываем метод с параметрами этого квадрата *add_square(i, empty_cell.first, empty_cell.second)* и добавляем полученное решение в очередь. По окончанию цикла *for* из очереди удаляется первый элемент, из которого строились следующие расширенные решения. В итоге функция вернет значение первого элемента очереди, это и будет является оптимальным решением нашей задачи.

13) *int read_size()*. Функция считывания размера столешницы. В теле функции объявляется переменная типа *int – size*, и с помощью *std::cin>>* считывается ее значение с клавиатуры. Введённое с клавиатуры значение размера столешницы возвращается функцией.

14) *void print_result(Desk printing_answer)*. Функция осуществляет форматированный в соответствии с заданием вывод ответа на экран. На вход в качестве аргумента принимается объект типа класса *Desk*. В начале выводится значение количества использованных квадратов, доступ к данному полю

возможен с помощью геттера *get_count_square()*. Далее для каждого элемента из вектора, в котором хранятся указатели на квадраты, которыми заполнена столешница, получаем доступ с помощью геттера *get_square_list()* и осуществляется корректный вывод каждого элемента благодаря перегруженному оператору вывода в поток в структуре *Square*.

15) *void solution()*. Функция запуска решения поставленной задачи. В теле функции переменной типа *int* – *table_size*, присваивается значение размера столешницы, возвращаемое с помощью функции описанной выше *read_size()*. Создается объект *desk* класса *Desk* в соответствии с размером, хранящимся в переменной *table_size*. Далее в переменной *answer* также типа класса *Desk* присваивается ответ на задачу с помощью запуска функции поиска с возвратом – *backtracking(desk)*. Далее вызывается функция *print_result()*, которой в качестве аргумента передается переменная *answer*, для вывода ответа на экран.

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 стороны вмещаемого квадрата на 2 и 3.
2) Размещение квадрата в самый верхний левый свободный угол.

Подробное описание данных оптимизаций представлено в отчете.

Для решения задачи была создана структура *Square*, которая отвечает за параметры размещаемого квадрат. Также был создан класс *Desk*, с его помощью осуществляется визуализация заполняемой обрезками столешницы. Методы класса позволяют, основываясь на оптимизации, по умолчанию заполнить несколькими квадратами стол и проверить наличие свободного места.

Сам алгоритм поиска с возвратом был реализован на основе очереди из столешниц – частичных решений, где каждое следующее значение является расширенным решением предыдущего.

Разработанный программный код для решения поставленной задачи успешно прошел тестирование на онлайн платформе *Stepik* при ограничении параметра N от 2 до 20 включительно.

ПРИЛОЖЕНИЕ А
ИСХОДНЫЙ КОД ПРОГРАММЫ

Название файла: main.cpp

```
#include <iostream>
#include <queue>
#include <vector>

//Структура, описывающая квадрат
//Содержит три поля типа int - размер стороны, координаты левого верхнего угла
struct Square{
    int size;
    int x_coord;
    int y_coord;

    //Конструктор структуры
    Square(int size, int x, int y):size(size), x_coord(x), y_coord(y){}
    //Перегрузка оператора вывода в поток
    friend std::ostream& operator<<(std::ostream& os, const Square& square){
        os << square.x_coord <<' '<< square.y_coord<< ' ' << square.size << std::endl;
        return os;
    }
};

//Класс, описывающий собираемую из квадратов столешниц
/*Поля класса - представление доски в виде матрицы, длина стороны стола,
количество квадратов и вектор размещенных квадратов*/
class Desk{
private:
    std::vector<std::vector<int>> table;
    int len_side;
    int count_squares;
    std::vector<Square*> list_square;
public:
    //Констурктор от длины стороны стола
    Desk(int n):len_side(n){
        this->table = std::vector<std::vector<int>>();
        for(auto i = 0; i < len_side; i++){
            this->table.push_back(std::vector<int>(n));
        }
        this->list_square = std::vector<Square*>();
        this->count_squares = 0;
        //Проверка делимости для оптимизации алгоритма
        if(n % 2 != 0 && n % 3 != 0){
            this->set_default();
        }
    }
    //Констурктор копирования
    //Принимает в качестве аргумента объект класса по ссылке
```

```
        Desk(const Desk& obj):len_side(obj.len_side),
count_squares(obj.count_squares){
            this->table = std::vector<std::vector<int>>();
            for(auto i = 0; i < len_side;i++){
                this->table.push_back(std::vector<int>(this->len_side));
            }
            for(auto i = 0; i < len_side; i++){
                for(auto j = 0; j < len_side; j++){
                    this->table[i][j] = obj.table[i][j];
                }
            }
            this->list_square = std::vector<Square*>(obj.list_square);
        }
        //Оператор присваивания с копированием
        Desk& operator=(const Desk& obj){
            if(this != &obj){
                this->table = std::vector<std::vector<int>>();
                for(auto i = 0; i < len_side;i++){
                    this->table.push_back(std::vector<int>(this-
>len_side));
                }
                for(auto i = 0; i < len_side; i++){
                    for(auto j = 0; j < len_side; j++){
                        this->table[i][j] = obj.table[i][j];
                    }
                }
                len_side = obj.len_side;
                count_squares = obj.count_squares;
                list_square = std::vector<Square*>(obj.list_square);
            }
            return *this;
        }
        //Получение значение приватного поля класса len_side
        int get_length_side() const{
            return len_side;
        }
        //Получение значение приватного поля класса list_square
        std::vector<Square*> get_square_list() const{
            return list_square;
        }
        //Получение значение приватного поля класса count_squares
        int get_count_square() const{
            return count_squares;
        }
        //Установка трех квадратов по умолчанию, если сторона не кратна
2 и 3
        //Примененная оптимизация
        //В левый верхний угол устанавливается больший квадрат
        //Два оставшихся ставятся как смежные с ним справа и снизу
        void set_default(){
            this->add_square((len_side+1)/2, 0, 0);
            this->add_square((len_side-1)/2, 0, (len_side+1)/2);
            this->add_square((len_side-1)/2, (len_side+1)/2, 0);
        }
        //Размещение квадрата на доске, заполнением единицами занятые
клетки матрицы
        //Принимает на вход размер квадрата и координаты его левого
верхнего угла
```

```
        void add_square(int size, int x, int y){
            Square* square = new Square(size, x, y);
            for(auto i = x; i < x + size; i++){
                for(auto j = y; j < y + size; j++){
                    this->table[i][j] = 1;
                }
            }
            this->list_square.push_back(square);
            this->count_squares++;
        }
        //Проверка на заполненность стола квадратами
        bool is_full(){
               for (int i = 0; i < len_side; i++){
                for (int j = 0; j < len_side; j++){
                    if (table[i][j] == 0){
                        return false;
                    }
                }
            }
               return true;
          }
        //Проверка на возможность разместить квадрат
        //Принимает на вход координаты левого верхнего угла квадрата и
его размер
        bool can_add(int x, int y, int size){
            if(y+size > this->len_side || x+size > this->len_side){
                return false;
            }
            for(auto i = x; i < x+size; i++){
                for(auto j = y; j < y+size; j++){
                    if(this->table[i][j] != 0){
                        return false;
                    }
                }
            }
            return true;
        }
        //Получение координаты самой верхней левой свободной клетки
        //Для оптимизации в такие координаты ставится новый квадрат
        std::pair<int, int> empty_cell(){
            for(auto i = 0; i < this->len_side; i++){
                for(auto j = 0; j < this->len_side; j++){
                    if(this->table[i][j] == 0){
                        return std::pair<int, int>{i, j};
                    }
                }
            }
            return std::pair<int, int>{-1, -1};
        }

    };
    //Реализация алгоритма поиска с возвратом на основе очереди
    //Принимает объект типа класса Desk
    //Возвращает минимальное заполнение матрицы квадратами
    Desk backtracking(Desk desk){
        std::queue<Desk> queue = std::queue<Desk>();
        queue.push(desk);
        while(!queue.front().is_full()){
```

```
            Desk desk_copy = queue.front();
            std::pair<int, int> empty_cell = desk_copy.empty_cell();
            for(int i = desk.get_length_side() - 1; i > 0 ; i--){
                Desk current = desk_copy;
                if(desk_copy.can_add(empty_cell.first,
empty_cell.second, i)){
                    current.add_square(i,             empty_cell.first,
empty_cell.second);
                    queue.push(current);
                }
            }
            queue.pop();
        }
        return queue.front();
    }
    //Функция считывания размера столешницы
    //Возвращает считанный размер соответственно
    int read_size(){
        int size;
        std::cin>>size;
        return size;
    }
    //Функция вывода ответа на экран в сооветсвии с требованием задания
    void print_result(Desk printing_answer){
        std::cout << printing_answer.get_count_square() << std::endl;
        for(auto elem:printing_answer.get_square_list()){
            std::cout << *elem;
        }
    }
    //Функция, запускающая решение поставленной задачи
    void solution(){
        int table_size = read_size();
        Desk desk = Desk(table_size);
        Desk answer = backtracking(desk);
        print_result(answer);
    }

    int main(){
        solution();
        return 0;
    }
```