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. 9382 _____ Кузьмин Д. И.

Преподаватель _____ Фирсов М. А.

Санкт-Петербург

2021

Цель работы.

Изучить принципы работы алгоритма поиска с возвратом. Освоить навыки написания программ, реализующих и оптимизирующих его.

Основные теоретические положения.

Поиск с возвратом, бэктрекинг — общий метод нахождения решений задачи, в которой требуется полный перебор всех возможных вариантов в некотором множестве M. Как правило, позволяет решать задачи, в которых ставятся вопросы типа: «Перечислите все возможные варианты …», «Сколько существует способов …», «Есть ли способ …», «Существует ли объект…» и т. п.

Задание.

Вариант 2и.

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 $(2 \leq \mathrm{N} \leq 20)$.

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).

Описание функций и структур данных.

1) Для описания квадрата использовался класс Square, имеющий поля:

Int size – размер

Int x – x координата левого верхнего угла.

Int y – y координата левого верхнего угла.

2) Для описания заполненности использовался класс Grid, содержащий двумерный массив bool значений, каждое из которых отвечает за клетку на поле.

3) Для хранения частичных решений использовался вспомогательный класс SquareContainer, содержащий стэк(std::stack) из объектов типа Square и объекта класса Grid, отвечающий за текущую заполненность столешницы.

4) В этом классе описаны функции void push(Square a) – добавление квадрата в стэк и обновление матрицы. Square a – добавляемый квадрат.

5) void pop() – удаление последнего квадрата из стэка и обновление матрицы.

6) bool isFull() – проверка, заполнена ли матрица. Возвращаемое значение – true, если да, false, если нет

7) Для расширения частичного решения использовались следующие функции:

8) Square generateSquare(SquareContainer* a, int maxsize) – генерация нового квадрата максимального размера с координатами левого верхнего угла равными координатам первой незанятой клетки. a – указатель на объект класса SquareContainer, т.е. текущего решения, int maxsize – это максимальный размер создаваемого квадрата. Возвращаемое значение – сгенерированный квадрат, и квадрат с отрицательным размером, если сгенерировать квадрат не удается.

9) bool isPossibleToPlace(Square a, std::vector<Square> used) – проверка, можно ли поставить квадрат a. Square a – это проверяемый квадрат, std::vector<Square> used – это вектор, содержащий уже использованные квадраты

на данном шаге. Возвращаемое значение – true, если квадрат можно поставить, false – если нельзя.

Описание алгоритма.

1) На каждый из возможных размеров (минимальный из таких размеров, как и размер максимального квадрата, определяется зависимости от размера исходного квадрата) производится квадрирование. В случае, если оно найдено, производится выход из цикла, если нет, то размер увеличивается на 1.

1) Первым шагом алгоритма идет постановка квадрата(максмального размера, на первой незанятой клетке)

2) Затем проверка, является ли текущее частичное решение полным решеним.

3) Если не является то, следует проверка размера текущего решения и в случае, если оно меньше требуемого размера, действия повторяются. Если размера текущего решения(кол. - во квадратов) равен заданному размеру, то производится откат на два шага назад (на один не имеет смысла, т.к. ставится всегда максимальный по размеру квадрат).

4) Во время первого шага алгоритма также обрабатывается случай, если на данном шаге поставить квадрат не удается.

5) В этом случае, если размер частичного решения равен нулю, то есть были перебраны уже все варианты, то на данной итерации генерация квадратов завершается и производится переход к поиску решения для следующего размера.

6) Если размер > 0, то производится откат на 1 шаг назад.

7) Использованные оптимизации: рассмотрение случаев, когда размер кратен 2 или 3 и построение разбиения соответственно на 4 и 6; постановка 3х начальных квадратов для сокращения перебора.

8) Сложность по времени можно оценить $O\left(\left(\frac{n}{2}\right)! \cdot \left(\frac{n}{2}\right)^2\right)$, по памяти – $O(n^2)$

Исходный код см. в приложении А.

Тестирование.

Результаты тестирование представлены в табл. 1.

Таблица 1 — результаты тестирования.

№ п/п	Входные данные	Выходные данные	Комментарий
1	5	8 1 1 3 1 4 2 4 1 2 4 3 2 3 4 1 3 5 1 4 5 1 5 5 1	5 – простое число, поэтому квадрирование заранее не известно
2	7	9 1 1 4 1 5 3 5 1 3 5 4 2 7 4 1 4 5 1 7 5 1 4 6 2 6 6 2	7 – простое число, поэтому квадрирование заранее неизвестно
3	10	4 1 1 5 1 6 5 6 1 5 6 6 5	Квадрат с четным размером всегда можно разделить на 4 равных
4	-2	Введите размер >= 2	Некорректный ввод
5	13	11 1 1 7 1 8 6 8 1 6	13 – простое число, поэтому квадрирование заранее неизвестно

		8 7 2 10 7 4 7 8 1 7 9 3 10 11 1 11 11 3 7 12 2 9 12 2	

Исследование

Результаты исследования представлены в таблице 2

Таблица 2 – результаты исследования.

Размер N	Время выполнения с.	Кол. – во итераций	Результат
2	1.7323e-05	1	4
3	4.6036e-05	3	6
4	1.7548e-05	1	4
5	0.000829876	46	8
6	2.3011e-05	1	4
7	0.00110234	132	9
8	9.886e-06	1	4
9	3.1566e-05	3	6
10	1.2868e-05	1	4
11	0.011592	954	11
12	1.3436e-05	1	4
13	0.0189082	1644	11
14	1.4425e-05	1	4
15	4.2796e-05	3	6
16	1.7706e-05	1	4
17	0.0983589	7549	12
18	1.6235e-05	1	4
19	0.320981	22038	13
20	1.9664e-05	1	4
21	5.579e-05	3	6
22	2.3043e-05	1	4
23	1.03883	56604	13
24	2.6131e-05	1	4
25	0.0249961	823	8
26	3.1084e-05	1	4

27	8.5877e-05	3	6
28	3.6265e-05	1	4
29	9.9033	390432	14
30	3.991e-05	1	4

Можно сделать вывод о том, что если n – не простое число, то решение находится быстро. В случае, если n – простое число, решение находится долго.

Выводы.

Был изучен принцип алгоритма поиска с возвратом и его оптимизации. Получены навыки разработки программ, реализующих его.

ПРИЛОЖЕНИЕ А. ИСХОДНЫЙ КОД ПРОГРАММЫ

```
#include <iostream>
#include <vector>
#include <string>
#include <algorithm>
#include <stack>
#include <chrono>
#include <cmath>
#define debugdetails 0
int min(int a , int b);
class Grid {

public:
    std::vector<std::vector<bool>> m;
     int n;

    Grid(int n) :n(n){
        for (int i = 0; i < n; i++) {
            std::vector<bool> a;
            for (int j = 0; j < n; j++)
                a.push_back(0);
            m.push_back(a);
        }

    }
    bool operator()(int i, int j) {
        return *((m.begin() + i)->begin() + j);
    }
    void set(int i, int j, int value) {
        *((m.begin() + i)->begin() + j) = value;
    }
    void print() {
        for (auto it : m) {
            for (auto it2 : it)
                std::cout << it2 << " ";
            std::cout << "\n";
        }

    }
    Grid operator=(Grid one){
     for (int i = 0 ; i < n; i++)
           for (int j = 0 ; j < n; j++)
                set(i,j, one(i,j));

    }
};
class Square {

public:
    int x, y;
    int size;
    Square(int y, int x, int size){

     this->size = size;
     this->x = x;
```

```
     this->y = y;

    }
    Square() {}
    void print() {
        std::cout << x + 1 << " " << y + 1 << " " << size << "\n";
    }
    friend bool operator==(Square s1, Square s2) {
        if (s1.x == s2.x && s1.y == s2.y && s1.size == s2.size) return
true;
        else return false;
    }
    std::vector<Square> squaresUsedOnCurrentStep;

};
class SquareContainer {
public:
    Grid* grid;
    int n;
    int sum;
    std::stack<Square>squares;
    SquareContainer(int n) {
        this->n = n;
        grid = new Grid(n);
        sum = 0;
    }
    void push(Square a) { //функция для вставки в стэк квадрата и
изменения матрицы заполненности
        for (int i = a.y; i < a.y + a.size; i++)
            for (int j = a.x; j < a.x + a.size; j++){
                grid->set(i, j, 1);

                }
          if (squares.size() >
1)squares.top().squaresUsedOnCurrentStep.push_back(a);
        squares.push(a);
        #ifdef debugdetails
              std::cout<<"Поставлен квадрат: ";
              squares.top().print();
        #endif
        sum += a.size * a.size;
    }
    void pop() { //функция для удаления последнего квадрата из стэка и
изменения матрицы заполненности

     for (int i = squares.top().y; i < squares.top().y +
squares.top().size; i++)
            for (int j = squares.top().x; j < squares.top().x +
squares.top().size; j++){
                grid->set(i, j, 0);
            }
             #ifdef debugdetails
                std::cout<<"Удален квадрат ";
                squares.top().print();
            #endif
            sum -= squares.top().size * squares.top().size;
            squares.pop();
    }
```

```
    bool isFull() {
        if (sum == n*n) return true;
        else return false;

    }
    bool isPossibleToPlace(Square s, std::vector<Square> used){
     int i = s.y;
     int j = s.x;
     int size = s.size;
     int intersection = false;

     if (size <= min(n - i, n - j)){    //пересечение и подходящий
размер
            for (int h = i; h < i + size; h++){
                for (int b = j; b < j + size; b++)
                    if ((*grid)(h, b)) {
                        intersection = true;
                        return false;
                    }
            }

            for (auto it : used){
                                        //использован ранее
                if (it == s)
                    return false;
            }
            return true;
        }
        else return false;
     }
};
int min(int a, int b) {
    if (a < b) return a;
    else return b;
}
Square generateNewSquare(SquareContainer* a,  std::vector<Square>& used,
int maxsize) { //расширение частичного ршения
    int n = a->n;
    int size = maxsize;
    int row = 0;
    int column = 0;
    bool breakflag = false;
    for (int i = 0; i < n; i++){
          //поиск первой незанятой клетки
        for (int j = 0; j < n; j++){
                if (!(*a->grid)(i,j)){
                     row = i;
                     column = j;
                     breakflag = true;
                     break;
                }
        }
        if(breakflag)break;
    }

                int minsize = 1;
                while (size >= minsize){
                     Square tmp(row,column,size);
```

```
                        if (a->isPossibleToPlace(tmp, used)){
                              return tmp;
                        }
                        else{
                              if (size == minsize){
                                #ifdef debugdetails
                                      std::cout<<"На данном шаге были
просмотрены все варианты. Не удается найти новых\n";
                                      //std::cout<<"Откат на один шаг\n";
                                  #endif
                                 return Square(-1, -1, -1);
                           }

                              else size -= 1;
                     }
                }
            return Square(-1, -1, -1);
    }
bool isprime(int n) {
    int i = 2;
    while (i <= sqrt(n)) {
        if (n % i == 0) return false;
        i++;
    }
    return true;
}
bool issquare(int n){
     int i = 0;
     for (i; i <= sqrt(n); i++)
          if (n == i * i)return true;
     return false;
}
void doSquaring(int  n){
     int min = sqrt(n);
    int size = n - 1;
    int index = 1;
    bool flg = false;
    if (n % 2 == 0){ //оптимизация дле четных n
        size = n / 2;
        min = 4;
    }
    else if (n % 3 == 0){ //оптимизация для n делящихся на 3
        size = (n / 3) * 2;
        min = 6;
    }
    else if (!isprime(n)) {
        //n = sqrt(n);
        int maxdivide = n;
        int i = n - 1;
        while (i > 0) {
            if (n % i == 0) {
                maxdivide = i;
                break;
            }
            i--;
        }
        size = ceil((double)maxdivide / 2) * maxdivide;
        min = sqrt(n);
```

```
    }
    else if (n > 3) {
        min = sqrt(n) + 2;
        size = n - n / 2;
        index = 2;
        flg = true;

    }
    int count = 0;
    SquareContainer* a = new SquareContainer(n);
    #ifdef debugdetails
        std::cout<<"Постановка 3х начальных квадратов\n";
    #endif
    a->push(Square(0,0,size));
     a->push(Square(size, 0, n - size));
    a->push(Square(0, size, n - size));
    if (flg) size = size/2 + 1;
    auto start = std::chrono::system_clock::now();
    for (min; ;min += 1){
        #ifdef debugdetails
                std::cout<<"\nПостроение решения для размера
"<<min<<std::endl;
        #endif
        while (1) {
            if (!a->isFull()){
                count++;
                if (a->squares.size() == min){ //если размер уже
максимальный
                    #ifdef debugdetails
                    std::cout<<"Достигнут максимальный размер(" << min <<
"), откат назад для поиска лучшего решения\n";
                    #endif
                if (min > 4)a->pop();
                 a->pop();
                }
                else {
                    Square squareGenerationResult;
                    squareGenerationResult = generateNewSquare(a, a-
>squares.top().squaresUsedOnCurrentStep, size); //генерация квадрата

                    if (squareGenerationResult.size != -1){
                        a->push(squareGenerationResult);
                    }
                    else{
                      if (a->squares.size() == 3){
                          #ifdef debugdetails
                                  std::cout<<"Не удалось найти решение
для заданного размера - " << min <<std::endl;
                          #endif

                            break;
                      }
                      else{
                        #ifdef debugdetails
                              std::cout<<"Откат на один шаг\n";
                        #endif
                        a->pop();
```

```
                    }

                    }
                }
            }
             else break;
        }
        if (a->squares.size() != 3){ //нашлось квадрирование
            #ifdef debugdetails
            std::cout<<"Удалось найти решение для размера " << min
<<std::endl;
            #endif
           break;
        }

        else                                //обновляем список использованных
на первом шаге
          a->squares.top().squaresUsedOnCurrentStep =
std::vector<Square>();


    }
    auto end = std::chrono::system_clock::now();

    std::chrono::duration<double> elapsed_seconds = end - start;
    //std::cout<<"Исходный размер: " << n <<"\nВремя: " <<
elapsed_seconds.count()<<" операций: " << count << " Результат: "<<a-
>squares.size()<< "\n";
   // std::cout<<n<<" "<<elapsed_seconds.count()<<" " <<count<<" "<<a-
>squares.size()<< "\n";
    std::cout<<"Кол-во квадратов: " <<a->squares.size()<< "\n";

    while(!a->squares.empty()){
        a->squares.top().print();
        a->squares.pop();
    }

}
int main ()
{

     int n = 2;
    setlocale(LC_ALL, "Russian");
    std::cout<<"Введите размер: ";
    std::cin>>n;

    while (n < 2){
        std::cout<<"Введите размер >= 2\n";
        std::cin>>n;
    }
    doSquaring(n);

    return 0;
}
```