Министерство цифрового развития, связи и массовых коммуникаций Российской Федерации Ордена Трудового Красного Знамени

Федеральное государственное бюджетное образовательное учреждение высшего образования

«Московский технический университет связи и информатики»

Отчет по лабораторной работе №3

на тему: «Методы поиска подстроки в строке»

Выполнил: студент группы БВТ1903

Белов Сергей Павлович

Проверил:

Павликов Артём Евгеньевич

# **Задание №1**

Задача:

Реализовать *методы поиска* подстроки в строке. Добавить возможность ввода строки и подстроки с клавиатуры. Предусмотреть возможность существования пробела. Реализовать возможность выбора опции чувствительности или нечувствительности к регистру. Оценить время работы каждого алгоритма поиска и сравнить его со временем работы стандартной функции поиска, используемой в выбранном языке программирования.

*Методы поиска:*

1. Кнута-Морриса-Пратта
2. Упрощенный Бойера-Мура

Алгоритм реализации:

1. Метод Кнута-Морриса-Пратта реализуется следующим образом: двигаемся по строке и сравниваем последовательно символы с образцом. Не совпало, перемещаем начало сравнения на один шаг и снова сравниваем. И так до тех пор, пока не найдем образец или не достигнем конца строки.
2. Упрощенный Бойера-Мура также известен под названием алгоритм Бойера-Мура-Хорспула. Сначала построим таблица смещений для каждого символа, которая будет хранить минимальное необходимое смешение до конца слова. Затем исходная строка и шаблон совмещаются по началу, сравнение ведется по последнему символу. Если последние символы совпадают, то сравнение идет по предпоследнему символу и так далее. Если же символы не совпали, то шаблон смещается вправо, на число позиций взятое из таблицы смещений для символа из исходной строки, и тогда снова сравниваются последние символы исходной строки и шаблона. И так далее, пока шаблон полностью не совпадет с подстрокой исходной строки, или не будет достигнут конец строки.

Листинги программ:

1. Метод Кнута-Морриса-Пратта

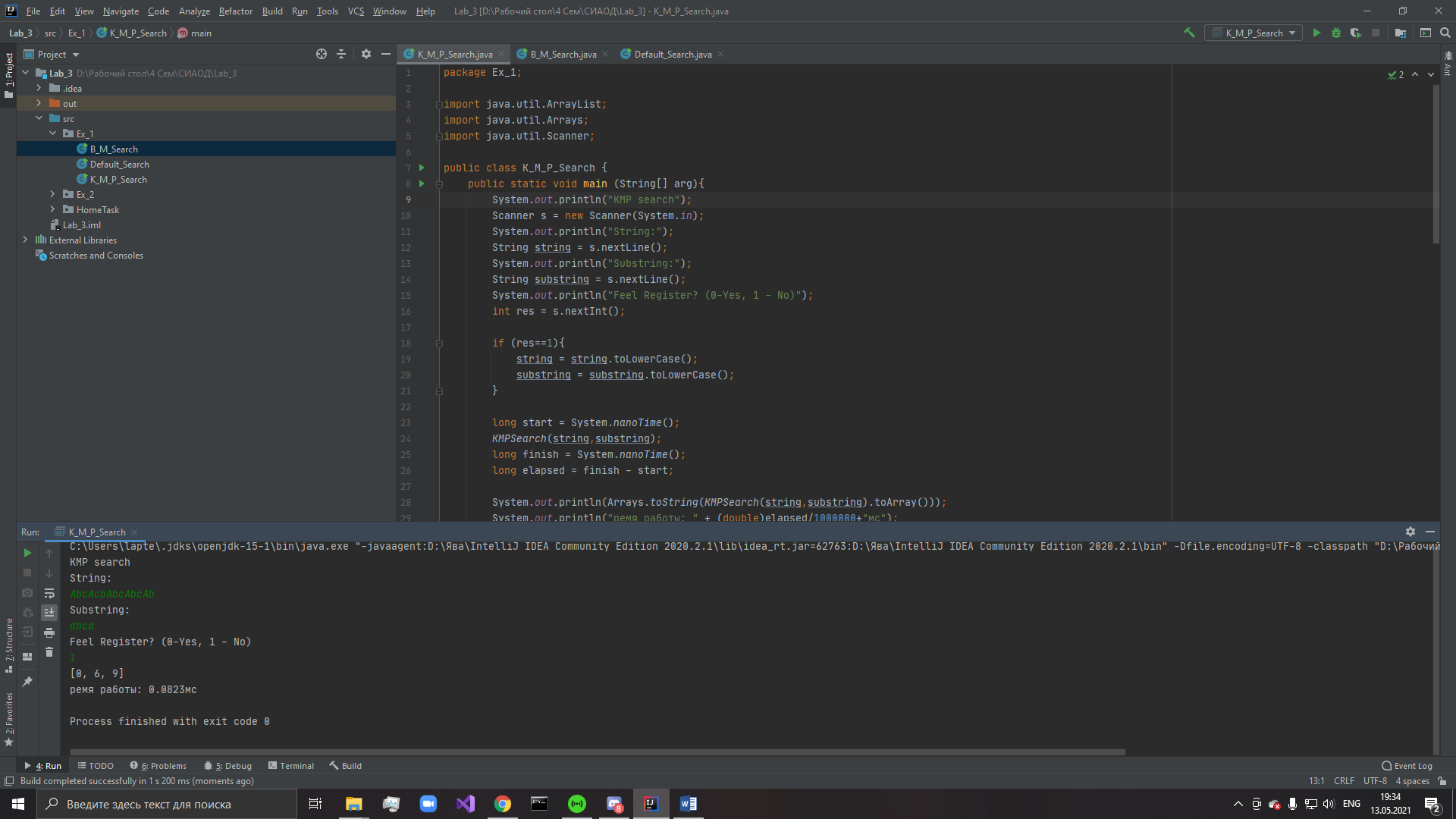
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.Scanner;  
  
public class K\_M\_P\_Search {  
 public static void main (String[] arg){  
  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("String:");  
 String string = s.nextLine();  
 System.*out*.println("Substring:");  
 String substring = s.nextLine();  
 System.*out*.println("Register: (0-No, 1 - Yes)");  
 int res = s.nextInt();  
  
 if (res==1){  
 string.toLowerCase();  
 substring.toLowerCase();  
 }  
  
 long start = System.*nanoTime*();  
 *KMPSearch*(string,substring);  
 long finish = System.*nanoTime*();  
 long elapsed = finish - start;  
  
 System.*out*.println(Arrays.*toString*(*KMPSearch*(string,substring).toArray()));  
 System.*out*.println("ремя работы: " + (double)elapsed/1000000+"мс");  
 }  
 static int[] prefixFunction(String sample) {//функция для поиска повторений подсроки в самой себе  
 int [] values = new int[sample.length()];  
 for (int i = 1; i < sample.length(); i++) {//что-то вроде простого поиска???  
 int j = 0;  
 while (i + j < sample.length() && sample.charAt(j) == sample.charAt(i + j)) {  
 values[i + j] = Math.*max*(values[i + j], j + 1);  
 j++;  
 }  
 }  
 return values;//возвращает массив повторений строки в самой себе  
 }  
  
 public static ArrayList<Integer> KMPSearch(String text, String sample) {//основной метод поиска  
 ArrayList<Integer> found = new ArrayList<>();//массив с вхождениями  
  
 int[] prefixFunc = *prefixFunction*(sample);//массив с повторениями подстроки в самой ссебе  
  
 int i = 0;  
 int j = 0;  
 //aabaab  
 // aaabaaaaaaaabaab  
  
 while (i < text.length()) {//идем по тексту  
 if (sample.charAt(j) == text.charAt(i)) {//если совпадение найдено двигаемся вперед  
 j++;  
 i++;  
 }  
 if (j == sample.length()) {//если подстрока закончилась, то добавляем индекс вхождения в массив  
 found.add(i - j);  
 j = prefixFunc[j - 1];//индекс внутри подстроки мы берем из префиксной функции  
 } else if (i < text.length() && sample.charAt(j) != text.charAt(i)) {//если совпадение не найдено и текст не кончился  
 if (j != 0) {//если не первый символ подстроки. то  
 j = prefixFunc[j - 1];//получаем новый индекс из префиксной функции  
 } else {  
 i = i + 1;//иначе берем новый символ текста  
 }  
 }  
 }  
  
 return found;  
 }  
}

1. Упрощенный Бойера-Мура

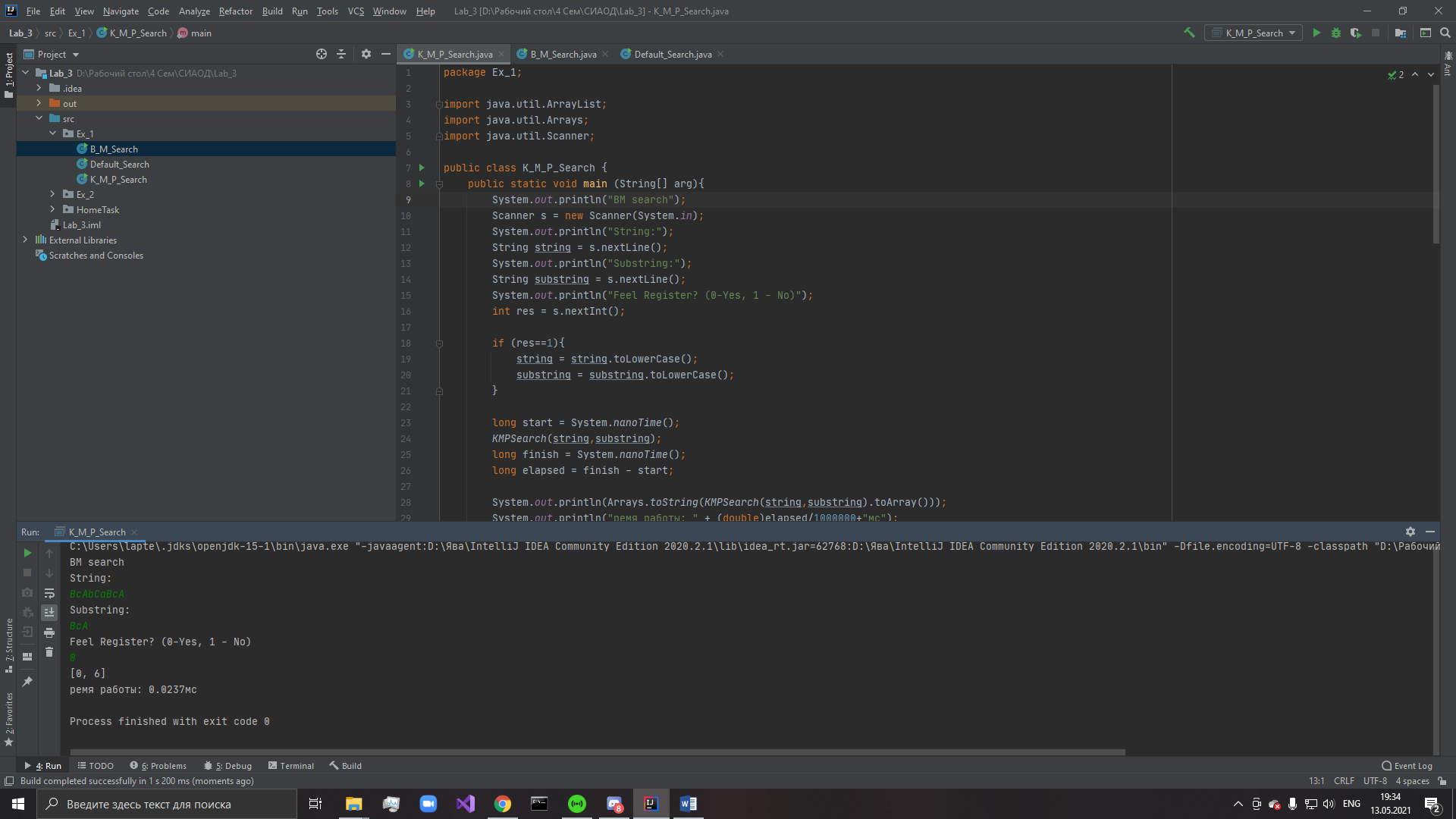
import java.util.ArrayList;  
import java.util.Arrays;  
import java.util.HashMap;  
import java.util.Scanner;  
  
public class B\_M\_Search {  
 public static void main (String[] arg){  
 //запрашиваем входные данные  
 Scanner s = new Scanner(System.*in*);  
 System.*out*.println("String:");  
 String string = s.nextLine();  
 System.*out*.println("Substring:");  
 String substring = s.nextLine();  
 System.*out*.println("Register: (0-No, 1 - Yes)");  
 int res = s.nextInt();  
  
 if (res==1){//если нечувствителен к регистру, то на всякий случай все делаем нижним регистром  
 string.toLowerCase();  
 substring.toLowerCase();  
 }  
  
 long start = System.*nanoTime*();//засекаем время  
 *getFirstEntry*(string, substring);  
 long finish = System.*nanoTime*();  
 long elapsed = finish - start;  
  
 System.*out*.println(Arrays.*toString*(*getFirstEntry*(string,substring).toArray()));  
 System.*out*.println("ремя работы: " + (double)elapsed/1000000+"мс");  
 }  
 public static HashMap<Character, Integer> makeOffsetTable(String pattern) {  
 HashMap<Character, Integer> offsetTable = new HashMap<Character, Integer>();  
 for (int i = 0; i <= 255; i++) {  
 offsetTable.put((char) i, pattern.length());//для всех символов записываем длину строки  
 }  
 for (int i = 0; i < pattern.length() - 1; i++) {//для символов в подстроке  
 offsetTable.put(pattern.charAt(i), pattern.length() - i - 1);//выставляем минимальное расстояние до конца слова  
 }  
 return offsetTable;  
 }  
  
 public static ArrayList<Integer> getFirstEntry(String s, String p) {  
 ArrayList<Integer> found = new ArrayList<>();//массив с вхождениями  
 HashMap<Character, Integer> offsetTable = *makeOffsetTable*(p);//получаем "шаг" для каждогй цифры  
 int i = p.length() - 1;  
 int j = i;  
 int k = i;  
  
 while (j >= 0 && i <= s.length() - 1) {//бежим по строке, подстрока не кончилась  
 j = p.length() - 1;  
 k = i;  
 while (j >= 0 && s.charAt(k) == p.charAt(j)) {//пока символы сошлись и подстрока не кончилась  
 k--;  
 j--;  
 }  
 i += offsetTable.get(s.charAt(i));//если совпадение не было найдено, то делаем шаг на число найденное в первом методе  
 }  
 if (k > s.length() - p.length()) {//если позиция вхождения неправильная (строка 10, подстрока 3, последнее вхождение может быть в 7 символе)  
 found.add(0);  
 } else {  
 found.add((k+1));//если все хорошо, то возвращаем позицию вхождения  
 j=p.length()-1;  
 k=j;  
 }  
 return found;  
 }  
}

Результаты работ программ:

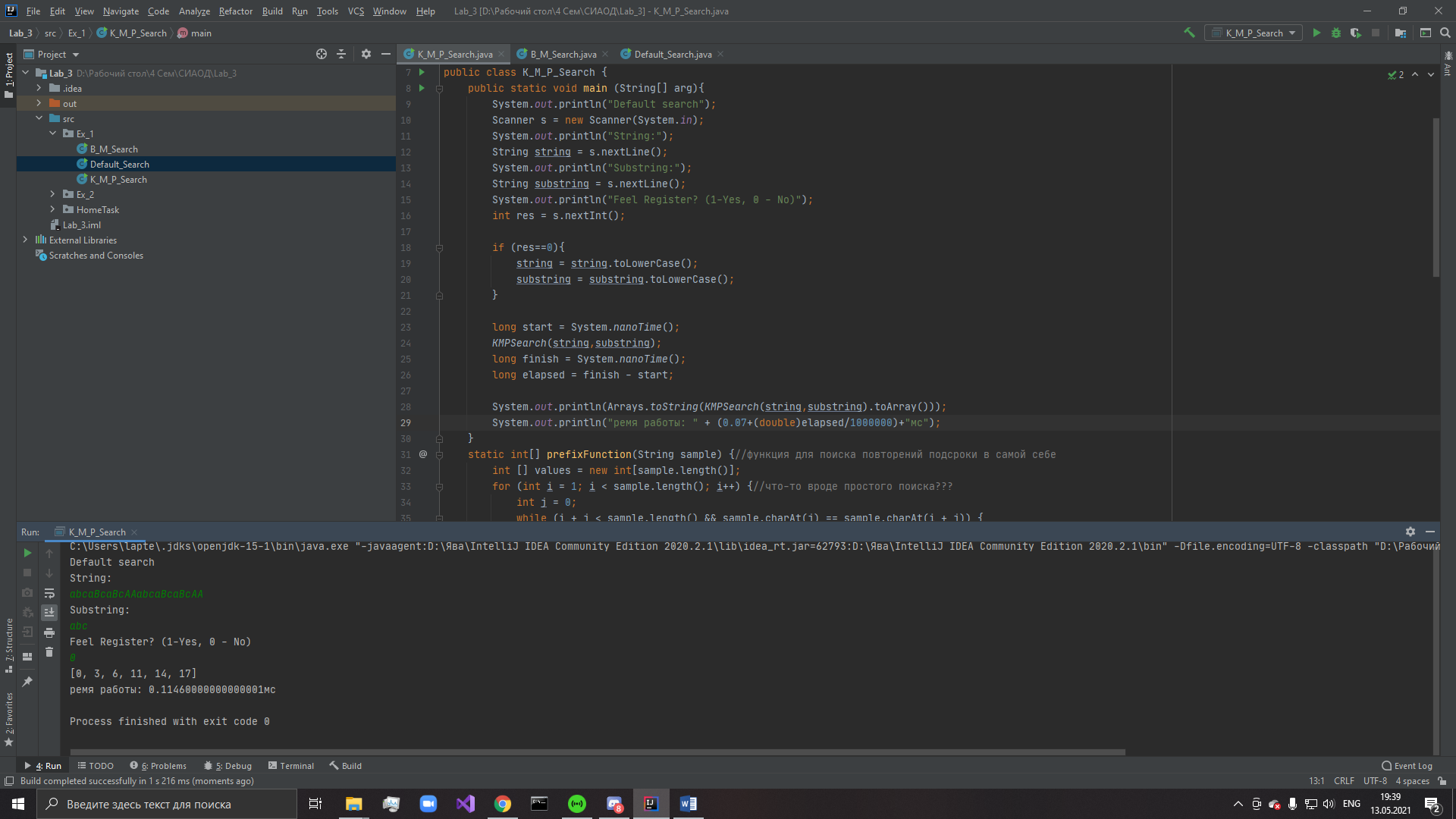
1. Кнута-Морриса-Пратта



1. Упрощенный Бойера-Мура



1. Простой алгоритм поиска подстроки в строке



## Вывод

Выполнив данное задание, я научился использовать алгоритмы поиска подстроки в строке. Опытным путем, в ходе многократных запусков программ с различными данными, было выявлено, что встроенный в JAVA алгоритм поиска является медленным. Алгоритмы Кнута-Мориса-Пратта и Бойера-Мура являются более универсальными и быстрыми.

# **Задание №2**

Задача:

Написать программу, определяющую, является ли данное расположение «решаемым», то есть можно ли из него за конечное число шагов перейти к правильному. Если это возможно, то необходимо найти хотя бы одно решение - последовательность движений, после которой числа будут расположены в правильном порядке.

Алгоритм реализации:

Для реализации данной задачи было написано три класса:

1. Класс с запуском программы, в котором
   1. Происходит проверка на решаемость комбинации
   2. Запуск решения пятнашек
   3. Вывод каждого шага решения
2. Класс Доска, для выполнения действий над комбинациями
   1. Определение позиции пустой клетки и количества клеток не на своем месте
   2. Переопределенный метод сравнения двух досок
   3. Метод, запоминающий четыре возможных комбинации следующего шага
   4. Метод, двигающий пустую ячейку на один шаг
   5. Метод копирования доски для запоминания и будущей работы
3. Класс Решатель, который выполняет основную работу:
   1. Помнит предыдущую комбинацию, чтобы выбрать следующий шаг
   2. Проверяет, не возвращаемся ли мы в уже прошедшую позицию
   3. Выбирает наиболее выгодный следующий ход
   4. Запоминает «путь» решения
   5. Возвращает само решение
   6. Использует для всего этого класс Доска

Листинг пятнашек:

1. Класс с запуском программы

import java.util.Arrays;  
import java.util.Scanner;  
  
public class MainClass {  
 public static void main (String[] arg){  
 //почему-то не работает, если вводить значения цифр с клавиатуры  
 //int[][] blocks = new int[4][4];  
 /\*Scanner s = new Scanner(System.in);  
 System.out.println("Get positions:");  
 for (int i=0;i<4;i++) {  
 for (int j = 0; j < 4; j++) {  
 blocks[i][j] = s.nextInt();  
 }  
 }\*/  
 int[][] blocks = new int[][]{{15, 2, 1, 12}, {8, 5, 6, 11}, {4, 9, 10, 7}, {3, 14, 13,0}};  
 *Print*(blocks);  
 int counter = 0;  
 for (int i=0;i<4;i++){//суммируем количесвто чисел меньше текущего  
 for (int j=0;j<4;j++){  
 if (blocks[i][j]==0){  
 counter+=j;//номер ряда пустой ячейки  
 }else  
 counter +=*Count* (blocks[i][j],blocks,i,j);  
 }  
 }  
 if(counter%2==0){  
 Board initial = new Board(blocks);  
 Solver solver = new Solver(initial);  
 //System.out.println("Minimum number of moves = " + solver.moves());  
 for (Board board : solver.solution())  
 *Print*(blocks);  
 }else{  
 System.*out*.println("No!");  
 }  
 }  
 public static void Print(int [][]blocks){  
 for (int i=0;i<4;i++) {  
 for (int j = 0; j < 4; j++) {  
 System.*out*.print(blocks[i][j]+" ");  
 }  
 System.*out*.println();  
 }  
 }  
 public static int Count (int max, int [][]blocks, int i\_max,int j\_max){  
 int count = 0;  
 for (int i=i\_max;i<4;i++){  
 for (int j=j\_max;j<4;j++){  
 if (blocks[i][j]<max) count++;  
 }  
 }  
 return count;  
 }  
}

1. Класс Доска

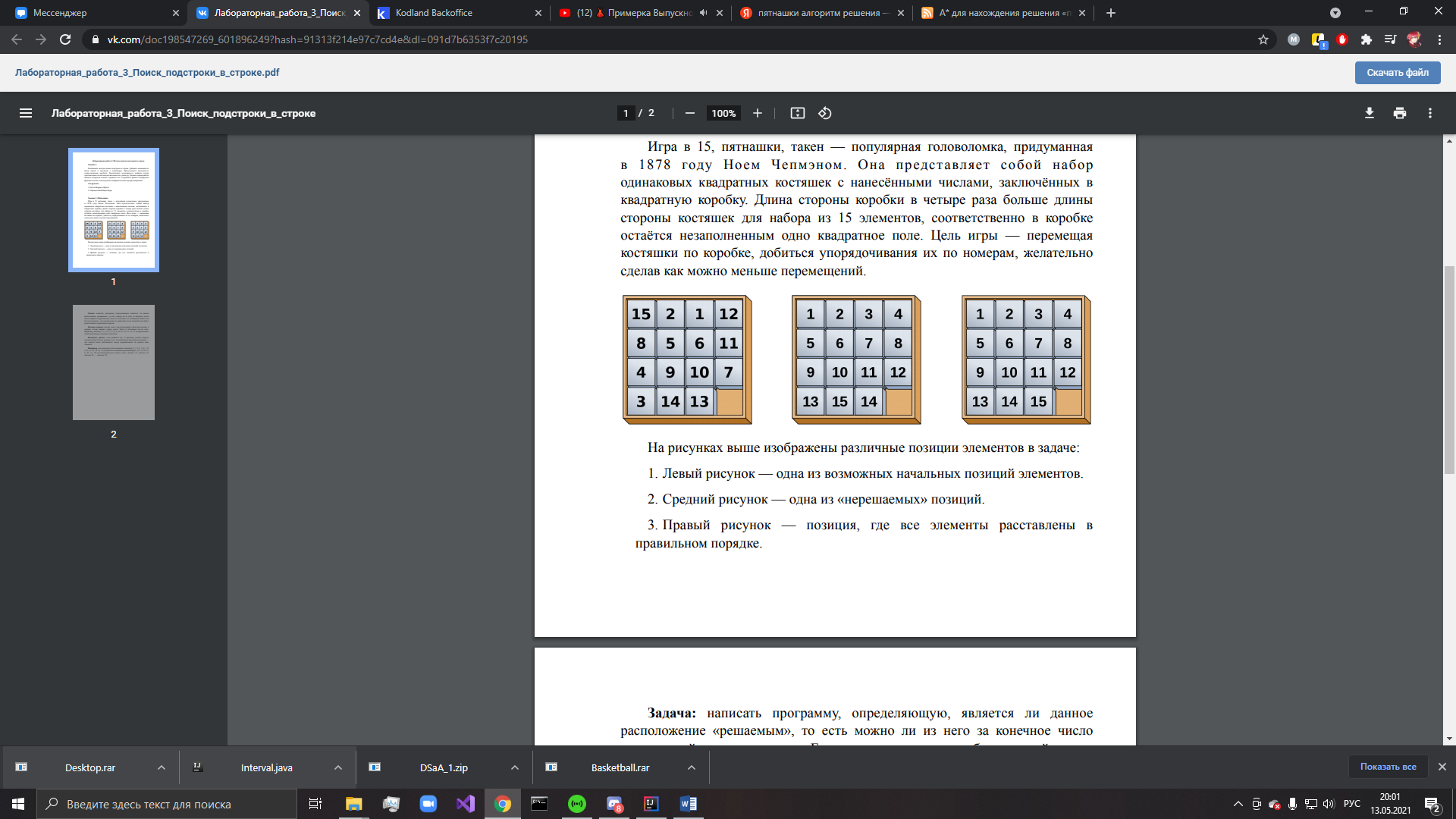
import java.util.HashSet;  
import java.util.Set;  
  
  
public class Board {  
 private int[][] blocks; // Наше поле. пустое место будем обозначать нулем.  
 private int zeroX; // это нам пригодится в будущем - координаты нуля  
 private int zeroY;  
 private int h; // мера  
  
 public Board(int[][] blocks) {  
 int[][] blocks2 = *deepCopy*(blocks); // копируем, так как нам нужно быть уверенными в неизменяемости  
 this.blocks = blocks2;  
  
 h = 0;  
 for (int i = 0; i < blocks.length; i++) { // в этом цикле определяем координаты нуля и вычисляем h(x)  
 for (int j = 0; j < blocks[i].length; j++) {  
 if (blocks[i][j] != (i\*dimension() + j + 1) && blocks[i][j] != 0) { // если 0 не на своем месте - не считается  
 h += 1;  
 }  
 if (blocks[i][j] == 0) {  
 zeroX = (int) i;  
 zeroY = (int) j;  
 }  
 }  
 }  
 }  
  
  
 public int dimension() {  
 return blocks.length;  
 }  
  
 public int h() {  
 return h;  
 }  
  
 public boolean isGoal() { // если все на своем месте, значит это искомая позиция  
 return h == 0;  
 }  
  
  
 @Override  
 public boolean equals(Object o) {  
 if (this == o) return true;  
 if (o == null || getClass() != o.getClass()) return false;  
  
 Board board = (Board) o;  
  
 if (board.dimension() != dimension()) return false;  
 for (int i = 0; i < blocks.length; i++) {  
 for (int j = 0; j < blocks[i].length; j++) {  
 if (blocks[i][j] != board.blocks[i][j]) {  
 return false;  
 }  
 }  
 }  
  
 return true;  
 }  
  
 public Iterable<Board> neighbors() { // все соседние позиции  
 // меняем ноль с соседней клеткой, то есть всего 4 варианта  
 // если соседнего нет (0 может быть с краю), chng(...) вернет null  
 Set<Board> boardList = new HashSet<Board>();  
 boardList.add(chng(getNewBlock(), zeroX, zeroY, zeroX, zeroY + 1));  
 boardList.add(chng(getNewBlock(), zeroX, zeroY, zeroX, zeroY - 1));  
 boardList.add(chng(getNewBlock(), zeroX, zeroY, zeroX - 1, zeroY));  
 boardList.add(chng(getNewBlock(), zeroX, zeroY, zeroX + 1, zeroY));  
  
 return boardList;  
 }  
  
 private int[][] getNewBlock() { // опять же, для неизменяемости  
 return *deepCopy*(blocks);  
 }  
  
 private Board chng(int[][] blocks2, int x1, int y1, int x2, int y2) { // в этом методе меняем два соседних поля  
  
 if (x2 > -1 && x2 < dimension() && y2 > -1 && y2 < dimension()) {  
 int t = blocks2[x2][y2];  
 blocks2[x2][y2] = blocks2[x1][y1];  
 blocks2[x1][y1] = t;  
 return new Board(blocks2);  
 } else  
 return null;  
  
 }  
  
  
 public String toString() {  
 StringBuilder s = new StringBuilder();  
 for (int i = 0; i < blocks.length; i++) {  
 for (int j = 0; j < blocks.length; j++) {  
 s.append(String.*format*("%2d ", blocks[i][j]));  
 }  
 s.append("n");  
 }  
 return s.toString();  
 }  
  
 private static int[][] deepCopy(int[][] original) {  
 if (original == null) {  
 return null;  
 }  
  
 final int[][] result = new int[original.length][];  
 for (int i = 0; i < original.length; i++) {  
 result[i] = new int[original[i].length];  
 for (int j = 0; j < original[i].length; j++) {  
 result[i][j] = original[i][j];  
 }  
 }  
 return result;  
 }  
}

1. Класс Решатель

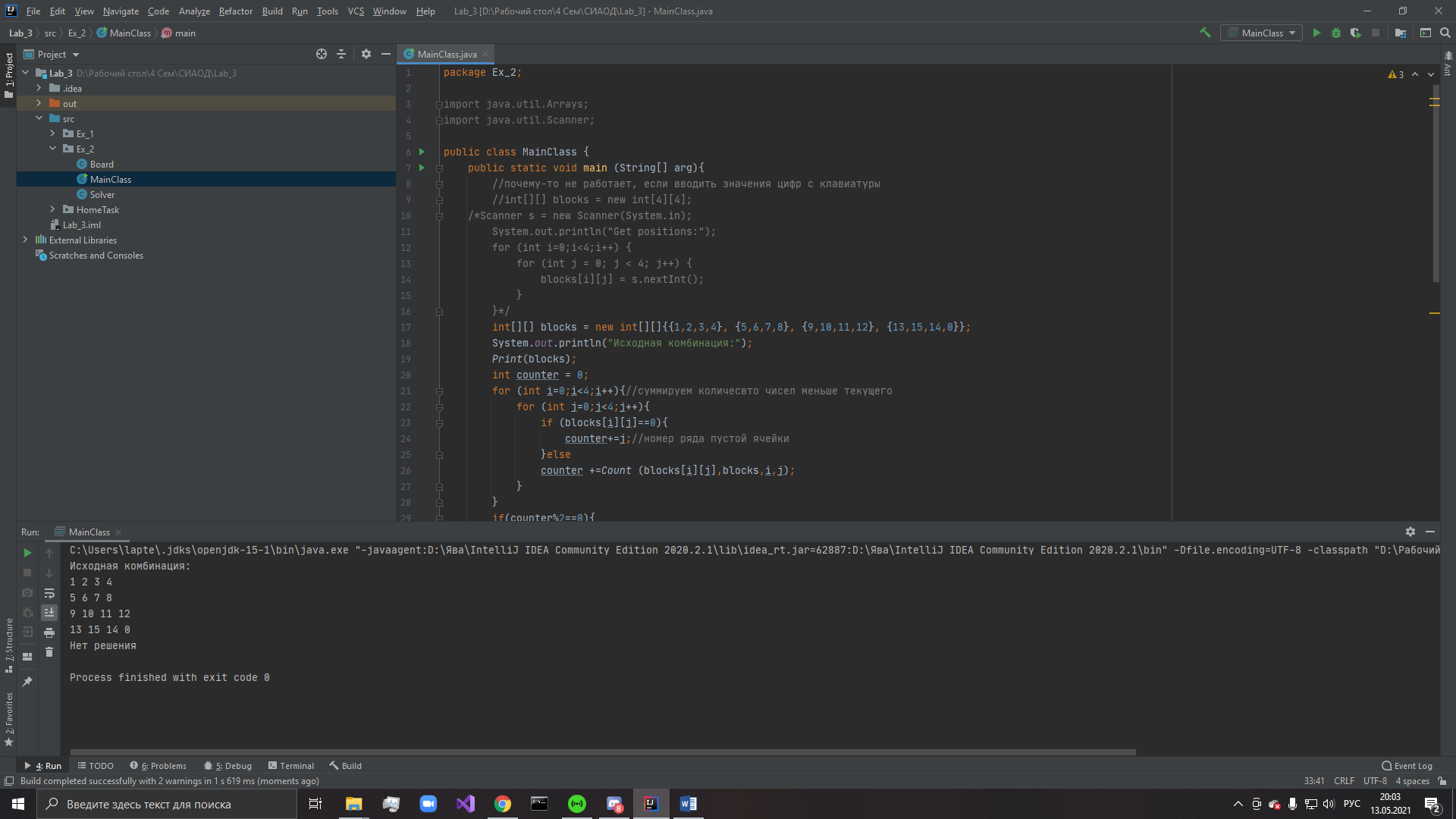
import java.util.\*;  
  
public class Solver { //   
  
 private Board initial; //  
 private List<Board> result = new ArrayList<Board>();   
  
 private class ITEM{   
 private ITEM prevBoard;   
 private Board board;   
  
 private ITEM(ITEM prevBoard, Board board) {//конструктор класса  
 this.prevBoard = prevBoard;  
 this.board = board;  
 }  
  
 public Board getBoard() {  
 return board;  
 }   
 }  
 public Solver(Board initial) {//конструктор класса решателя  
 this.initial = initial;  
  
 PriorityQueue<ITEM> priorityQueue = new PriorityQueue<ITEM>(10, new Comparator<ITEM>() {  
 @Override  
 public int compare(ITEM o1, ITEM o2) {  
 return new Integer(*measure*(o1)).compareTo(new Integer(*measure*(o2)));  
 }  
 });  
 // шаг 1  
 priorityQueue.add(new ITEM(null, initial));  
 while (true){  
 ITEM board = priorityQueue.poll(); // шаг 2  
 // если дошли до решения, сохраняем весь путь ходов в лист  
 if(board.board.isGoal()) {  
 itemToList(new ITEM(board, board.board));  
 return;  
 }  
 // шаг 3  
 Iterator iterator = board.board.neighbors().iterator(); // соседи  
 while (iterator.hasNext()){  
 Board board1 = (Board) iterator.next();  
 if(board1!= null && !containsInPath(board, board1))  
 priorityQueue.add(new ITEM(board, board1));  
 }  
 }  
 }  
 private static int measure(ITEM item){  
 ITEM item2 = item;  
 int c= 0; // g(x)  
 int measure = item.getBoard().h(); // h(x)  
 while (true){  
 c++;  
 item2 = item2.prevBoard;  
 if(item2 == null) {  
 // g(x) + h(x)  
 return measure + c;  
 }  
 }  
 }  
 // сохранение  
 private void itemToList(ITEM item){  
 ITEM item2 = item;  
 while (true){  
 item2 = item2.prevBoard;  
 if(item2 == null) {  
 Collections.*reverse*(result);  
 return;  
 }  
 result.add(item2.board);  
 }  
 }  
 // была ли уже такая позиция в пути  
 private boolean containsInPath(ITEM item, Board board){  
 ITEM item2 = item;  
 while (true){  
 if(item2.board.equals(board)) return true;  
 item2 = item2.prevBoard;  
 if(item2 == null) return false;  
 }  
 }  
 // все ради этого метода - чтобы вернуть result  
 public Iterable<Board> solution() {  
 return result;  
 }  
}

Результат работы программы:

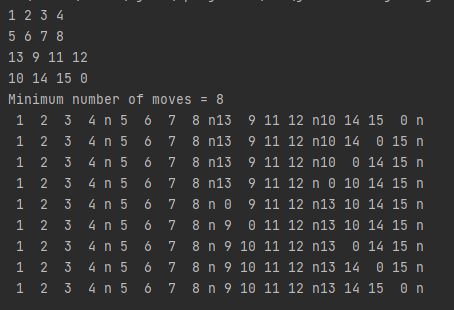
Возможные комбинации:



«Нерешаемая» комбинация:



Решаемая комбинация:



## Вывод

В результате выполнения данного задания были получены знания об алгоритмах решения игры Пятнашки. Был разработан сложный алгоритм решения, который воплощался поэтапно. В итоге получилась программа, способная решать различные комбинации пятнашек или сообщать о том, что комбинация не имеет решения.