Programming Challenges

Roman Valiušenko
roman.valiusenko@gmail.com

Abstract

This is a collection of literate programs. If you are unfamiliar with the idea of literate programming please refer [1]. These programs are my solutions to the programming tasks from the "Programming Challenges" book[2] which in turn is a collection of problems from the UVa Online Judge hosted by University of Valladolid[1].

Contents

1 Getting Started **1**
1.1 The $3n + 1$ Problem . . . 1
1.2 Minesweeper . . . 4
1.3 The Trip . . . 6
1.4 LC Display . . . 7
1.5 Graphical Editor . . . 12
1.6 Interpreter . . . 15
1.7 Check The Check . . . 17
1.8 Australian Voting . . . 20

2 Data Structures **22**
2.1 Jolly Jumpers . . . 22
2.2 Poker Hands . . . 23
2.3 Hartals . . . 29
2.4 Crypt Kicker . . . 30
2.5 Stack 'em Up . . . 36
2.6 Erdös Numbers . . . 38
2.7 Contest Scoreboard . . . 40
2.8 Yahtzee . . . 42

3 Strings **51**
3.1 WERTYU . . . 51
3.2 Where's Waldorf . . . 52
3.3 Common Permutation . . . 54
3.4 Crypt Kicker II . . . 56
3.5 Automated Judge Script . . . 58
3.6 File Fragmentation . . . 59
3.7 Doublets . . . 61
3.8 Fmt . . . 64

[1] If you are going to submit any of these programs to the UVa Online Judge (which you obviously shouldn't) make sure the class name is Main and that it is not in any package; Note that for the class names I use problem names

4 Sorting **67**
4.1 Vito's Family . . . 67
4.2 Stacks of Flapjacks . . . 68
4.3 Bridge . . . 70
4.4 Longest Nap . . . 74
4.5 Shoemaker's Problem . . . 77
4.6 CDVII . . . 79
4.7 ShellSort . . . 83
4.8 Football (aka Soccer) . . . 85

5 Arithmetic and Algebra **89**
5.1 Primary Arithmetic . . . 89
5.2 Reverse And Add . . . 91
5.3 The Archeologists' Dilemma . . . 92
5.4 Ones . . . 93
5.5 A Multiplication Game . . . 95
5.6 Polynomial Coefficients . . . 97
5.7 The Stern-Brocot Number System . . . 99
5.8 Pairsumonious Numbers . . . 101

6 Combinatorics **105**
6.1 How Many Fibs? . . . 105
6.2 How Many Pieces of Land? . . . 105
6.3 Counting . . . 108
6.4 Expressions . . . 110
6.5 Complete Tree Labeling . . . 112
6.6 The Priest Mathematicians . . . 115
6.7 Self-Describing Sequence . . . 117
6.8 Steps . . . 119

7 Number Theory **121**
7.1 Light, More Light . . . 121
7.2 Carmichael Numbers . . . 124
7.3 Euclid Problem . . . 126
7.4 Factovisors . . . 128
7.5 Summation of Four Primes . . . 130
7.6 Smith Numbers . . . 132
7.7 Marbles . . . 134
7.8 Repackaging . . . 137

8 Backtracking **137**
8.1 Little Bishops . . . 137
8.2 15-Puzzle Problem . . . 141
8.3 Queue . . . 147
8.4 Servicing Stations . . . 150
8.5 Tug Of War . . . 155
8.6 Garden of Eden . . . 157
8.7 Colours Hash . . . 159
8.8 Bigger Square Please... . . . 166

9 Graph Traversal **181**
9.1 Bicoloring . . . 181
9.2 Playing With Wheels . . . 183
9.3 The Tourist Guide . . . 186
9.4 Slash Maze . . . 188
9.5 Edit Step Ladders . . . 191
9.6 Tower of Cubes . . . 193
9.7 From Dusk Till Dawn . . . 196
9.8 Hanoi Tower Troubles Again! . . . 199

10 Graph Algorithms **200**
10.1 Freckles . . . 200
10.2 The Necklace . . . 202
10.3 Fire Stations . . . 205
10.4 Railroads . . . 209
10.5 War . . . 212
10.6 Tourist Guide . . . 215
10.7 The Grand Dinner . . . 217
10.8 The Problem with the Problem Setter . . . 219

11 Dynamic Programming **223**
11.1 Is Bigger Smarter? . . . 223
11.2 Distinct Subsequences . . . 224
11.3 Weights and Measures . . . 227
11.4 Unidirectional TSP . . . 229
11.5 Cutting Sticks . . . 232
11.6 Ferry Loading . . . 234

12 Grids **236**
12.1 Ant on a Chessboard . . . 236

13 Geometry **237**

14 Computational Geometry **237**

15 License **237**

1 Getting Started

1.1 The $3n + 1$ Problem

This task is not difficult if you notice that all the lengths of the sequences can easily be calculated up front. Then all that is needed is to lookup the pre-calculated table to find out the maximum lengths for the given input numbers.

(I noticed though that I could have simply calculated the values on the file without any tricks. The reason why I have done a more sophisticated algorithm is that at first I though the input number may go up to 1M, but in reality, according to the problem statement, they won't exceed 10000. So I solved a more tricky problem.)

So let's start with the definitions of the array that will hold all the `lengths` and the `reader` that will be used to read the input data.

1a ⟨*3n+1* 1a⟩≡
⟨*1.1 Imports* 1b⟩

```
class Collatz {
    private static int MAX = 1000000;
    private int[] lengths = new int[MAX];
    private static final BufferedReader reader =
             new BufferedReader(new InputStreamReader(System.in));
    ⟨1.1 Helpers 2b⟩
    ⟨1.1 Constructor 3a⟩
    ⟨1.1 Input/Output 3c⟩
}
```

We need the necessary imports:

1b ⟨*1.1 Imports* 1b⟩≡

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
```

The idea is to hold the lengths of the sequences in the `lengths`, but because the sequence member can sometimes go over 1M we will need to store them somewhere temporarily. For that a `surplus` hash map will be used. Its contents will be thrown away once the sequence lengths were computed.

So we write two helper methods: `set` and `get`. Both take an `index` and `surplus` hash map and depending on the index value either use the array or the hash map to set or get a value.

2a ⟨*1.1 Imports* 1b⟩+≡

```
import java.util.HashMap;
```

2b ⟨*1.1 Helpers* 2b⟩≡

```
int get(long index, HashMap<Long, Integer> surplus) {
    return (index < MAX) ? lengths[(int) index] :
        (surplus.containsKey(index) ? surplus.get(index) : 0);
}

void set(long index, int value, HashMap<Long, Integer> surplus) {
    if (index < MAX) {
        lengths[(int) index] = value;
    } else {
        surplus.put(index, value);
    }
}
```

Now we can easily pre-calculate all the lengths using the helper methods `set` and `get`, but we must not re-calculate the lengths for the indexes that we have calculated already.

We calculate a member of the sequence at each step using the definition. Each time we calculate a new member of the sequence we push it onto the `stack`. We stop if we notice that we already have the length calculated for that specific value or when we reach 1. Now all the values that are on the stack are potential inputs, that is they are all potential initial `n`s. We use this knowledge to update elements in the `lengths`:

2c ⟨*1.1 Imports* 1b⟩+≡

```
import java.util.ArrayDeque;
import java.util.Deque;
```

3a ⟨*1.1 Constructor* 3a⟩≡

```
Collatz() {
    final HashMap<Long, Integer> surplus = new HashMap<Long, Integer>();
    lengths[1] = 1;
    for (long i = 2; i < MAX; ++i) {
        final Deque<Long> stack = new ArrayDeque<Long>();
        long n = i;
        int len = 2;
        while (n != 1) {
            stack.push(n);
            int prev = get(n, surplus);
            if (prev > 0) {
                len = prev;
                break;
            }
            n = n % 2 == 0 ? n / 2 : n * 3 + 1;
        }
        while (!stack.isEmpty()) {
            set(stack.pop(), len++, surplus);
        }
    }
}
```

Processing the input is easy but cumbersome[2]:

3b ⟨*1.1 Imports* 1b⟩+≡

```
import java.util.stream.IntStream;
```

[2]It turns out that the UVa Judge tends to give some extra spaces here and there in the input, so we need to make sure we account for some sporadic spaces in the input. This was my first submission and it took me seven attempts before I got past that super annoying "Runtime Error", because the judge was giving some extra spaces between the values which my program was not taking into account.

3c ⟨*1.1 Input/Output* 3c⟩≡
```
public static void main(String[] args) {
    Collatz s = new Collatz();
     String input;
     while ((input = reader.readLine()) != null &&
             !input.trim().equalsIgnoreCase("")) {
         List<String> str = Arrays.stream(input.trim().split(" "))
            .filter(x -> !x.equals("")).collect(Collectors.toList());
         int x[] = new int[] { Integer.parseInt(str.get(0)),
                 Integer.parseInt(str.get(1)) };
         System.out.println(x[0] + " " + x[1] + " " +
                 IntStream.rangeClosed(Math.min(x[0], x[1]), Math.max(x[0],
                         x[1])).map(v -> s.lengths[v]).max().getAsInt());
     }
}
```

1.2 Minesweeper

This task is trivial: We simply count the number of mines around each cell. There are eigth cells around each cell that we need to inspect. If our cell is (x, y), then we check $(x-1, y-1)$, $(x, y-1)$ and so on, and count the number of cells that have '*' in them.

Our program structure is simple as usual:

4a ⟨*Minesweeper* 4a⟩≡
```
⟨1.2 Imports 4b⟩

class Minesweeper {
    ⟨1.2 Constants 4c⟩
    ⟨1.2 Main 5⟩
}
```

Of course, we need a reader, so we define it next. Then we need to define the constants. We are going to split the lines by spaces, so let's have it as a constant. We also define an array of the offsets `p` to determine the cells around a given cell.

4b ⟨*1.2 Imports* 4b⟩≡
```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

4c ⟨*1.2 Constants* 4c⟩≡
```
private static final BufferedReader reader =
    new BufferedReader(new InputStreamReader(System.in));
private static final String SPACE = " ";
private static final int[][] p = new int[][] {
        { -1, -1 }, { 0, -1 }, { 1, -1 }, { -1, 0 },
        { 1, 0 }, { -1, 1 }, { 0, 1 }, { 1, 1 }
};
```

Now let's write the main method. I'll delibirately use one-dimentional array instead of the two-dimensional, and I will use a couple of helper lambdas. One, `count`, to count the mines around a cell, and another, `mine`, which returns a cell value for the given coordinates.

4d ⟨*1.2 Imports* 4b⟩+≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.joining;
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;
import java.util.List;
import java.util.function.IntBinaryOperator;
import java.util.function.IntUnaryOperator;
```

5 ⟨*1.2 Main* 5⟩≡

```
public static void main(String[] args) throws IOException {
    int lineNum = 0;
    String currentLine = INPUT_END;
    while ((currentLine = reader.readLine()) != null) {
        if (currentLine.equalsIgnoreCase("")) {
            continue;
        }
        List<Integer> nm = stream(currentLine.split(SPACE))
                .filter(x -> !x.equals("")).map(Integer::parseInt).collect(toList());;
        int n = nm.get(0);
        int m = nm.get(1);
        if (n == 0 && m == 0) {
            break;
        }

        final int[] field = reader.lines().limit(n)
            .collect(joining()).chars().map(x -> x == '*' ? -1 : 0).toArray();

        final IntBinaryOperator mine =
           (x, y) -> (x < 0 || x > (n - 1) || y < 0 || y > (m - 1)) ? 0 : field[x * m + y];

        final IntUnaryOperator count = (i) -> range(0, p.length)
            .map(j -> Math.abs(mine.applyAsInt(i / m + p[j][0], i % m + p[j][1]))).sum();

        int[] result = range(0, field.length)
            .map(x -> field[x] >= 0 ? count.applyAsInt(x) : field[x]).toArray();

        if (lineNum > 0) {
            System.out.println();
        }

        System.out.println("Field #" + (++lineNum) + ":");
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                System.out.print(result[i * m + j] == -1 ? "*" : result[i * m + j]);
            }
            System.out.println();
        }
    }
}
```

1.3 The Trip

This task is much more fun that the previous two. The important thing that we should note for ourselves is that we are not going to use the floating point types to do the calculations.

6a ⟨*The Trip* 6a⟩≡
```
⟨1.3 Imports 6b⟩

class TheTrip {
    ⟨1.3 Calculation 6c⟩
    ⟨1.3 Input/Output 7c⟩
}
```

First thing we need to do is to calculate the average spend, don't we? Because we know that the input is a list of how much each of `n` students spent, let's define a function that takes this list of values and returns the minimum amount of money asked in the problem. Of course, the types will be `long`. And we can immediately cover the degenerate case of a input consisting of one element:

6b ⟨*1.3 Imports* 6b⟩≡
```
import static java.util.Arrays.stream;
```

6c ⟨*1.3 Calculation* 6c⟩≡
```
static long calculate(long[] values) {
    if (values.length == 1)
        return 0;
    long total = stream(values).sum();
    ⟨1.3 Finding the minimum 6e⟩
}
```

Now we need to partition the students into two groups: One group of students that will be giving money (those that spent less than group average) and the ones who will be receiving the money (those that spent more than the group average). But the `total` won't always divide without a reminder. So we divide the `total` by the number of students to get the quotient and the reminder, and we partition only using the quotient; that is group 1 will contain spends x such that $x - quotient \leq 0$, and group 2 will have the others.

6d ⟨*1.3 Imports* 6b⟩+≡
```
import static java.lang.Math.abs;
import static java.util.stream.Collectors.partitioningBy;
import java.util.List;
import java.util.Map;
```

6e ⟨*1.3 Finding the minimum* 6e⟩≡
```
long quotient = total / values.length;
long reminder = total % values.length;
Map<Boolean, List<Long>> diff =
    stream(values).map(x -> x - quotient).boxed().collect(partitioningBy(x -> x > 0));
```

So what do we do with the `reminder`? These are those cents that we need to finally re-distribute among the members of the two groups. Note that the `reminder` will always be less than `n`. We choose the following strategy: We distribute these cents to the group that spent less than or equal to the `quotient`, the remaining cents are finally distributed to group 2. This is captured in the following code:

7a ⟨*1.3 Finding the minimum* 6e⟩+≡

```
long sum = abs(diff.get(false).stream().reduce(Long::sum).get());
long len = diff.get(true).size();
reminder = len <= reminder ? reminder - len : 0;
return sum + reminder;
```

All we need to do now is to write input reading, which is trivial:

7b ⟨*1.3 Imports* 6b⟩+≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;
```

7c ⟨*1.3 Input/Output* 7c⟩≡

```
public static void main(String[] args) throws IOException {
    BufferedReader r = new BufferedReader(new InputStreamReader(System.in));
    int n = 0;
    while ((n = Integer.parseInt(r.readLine().trim())) > 0) {
        long[] values = r.lines().limit(n).map(x -> x.replaceAll("\\.",
                "").trim()).mapToLong(Long::parseLong).toArray();
        System.out.println("$" + BigDecimal.valueOf(calculate(values), 2));
    }
}
```

1.4 LC Display

This task may seem quite involved at first sight, because you may start thinking about two-dimensional patterns and scaling functions. But in reality this task is much easier if you notice that the digits can be constructed not in a top to bottom (or bottom to up) row-by-row manner, but in a columnar manner; at the same time scaling becomes very easy.

7d ⟨*LC Display* 7d⟩≡

```
⟨1.4 Imports 8a⟩

class LCDisplay {
    ⟨1.4 Constants 8b⟩
    ⟨1.4 Convertion 9b⟩
    ⟨1.4 Input/Output 9a⟩
}
```

Digit	Binary	Hex
0	11 101 11	77
1	00 000 11	03
2	10 111 01	5D
3	00 111 11	1F
4	01 010 11	2B
5	01 111 10	3E
6	11 111 10	7E
7	00 001 11	07
8	11 111 11	7F
9	01 111 11	3F

Table 1: LCD segments

Each LCD digit has 7 segments: Two in the first and the third columns and three in the second column. Let's encode our digits as in the table.

Since we know that the input ends in two zeros we define this string constant plus a couple of other string constants.

8a ⟨*1.4 Imports* 8a⟩≡
```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.stream.Stream;
```

8b ⟨*1.4 Constants* 8b⟩≡
```
private static final BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
private static final String INPUT_END = "0 0";
private static final String EMPTY = "";
private static final String SPACE = " ";
private static final byte[] pattern = new byte[] {
        0x77, 0x03, 0x5d, 0x1f, 0x2b, 0x3e, 0x7e, 0x07, 0x7f, 0x3f
};
```

The array `pattern` for the given digit `i` returns bits that correspond to the segments, so for example `digits[5]` would return segments for digit 5. We will be using masks to discover which bits are set and not set.

But let's write input/output first as this is very easy. At the same time, let's assume our method that converts a string into LCD style digits is called `segments`. This method takes two arguments, the digits string and `scale`. Let's assume it returns list of strings which we can simply output to the console.

9a ⟨*1.4 Input/Output* 9a⟩≡

```
public static void main(String[] args) throws IOException {
    String currentLine = INPUT_END;
    while ((currentLine = reader.readLine()) != null &&
            !currentLine.trim().equalsIgnoreCase(INPUT_END)) {
        List<String> input = Arrays.stream(currentLine.trim().split(SPACE))
            .filter(x -> !x.equals("")).collect(Collectors.toList());
        segments(input.get(1), Integer.valueOf(input.get(0))).stream()
            .forEach(System.out::println);
        System.out.println();
    }
}
```

Now all that's left to implement `segments`.

9b ⟨*1.4 Convertion* 9b⟩≡

```
private static List<String> segments(final String digits, final int scale) {
    ⟨1.4 Helpers 9d⟩
    ⟨1.4 Process 11a⟩
    ⟨1.4 Return 11c⟩
}
```

The idea is simple: We check bit 6 and bit 5 of the pattern and construct ASCII representation of the first column, then we check bit 4, 3, and 2 and construct the middle column, finally we check bit 1 and 0 to construct the last column. Of course, we need to take into account the scaling.

So let's have a look at an example. Let's say we need to construct digit 2 with scale 3. First, we get the pattern value `pattern[2]`=5D, or 1011101. Then, we start with the masks 40 and 20 to see which segments are on in the first column (bit 6 and bit 5); so, `40 & 5D = 1` and `20 & 5D = 0`, which means that the first segment is on and the second is off, so we output ␣|‗. Because our scale is 3, we output | and ␣ three times, so we end up with ␣|||‗‗‗; Similarly we construct the third column, but we use different masks: 02 and 01.

OK, let's write some helpers already before we get back to producing the middle column. We will need some function that replicates a specified string n times. There's a Java function `nCopies` that does that, so we will use it. However, it returns a list of strings, therefore we use `join` function to join that into a single string using `EMPTY` as a delimiter. Let's write that:

9c ⟨*1.4 Imports* 8a⟩+≡

```
import static java.lang.String.join;
import static java.util.Collections.nCopies;
import java.util.function.Function;
```

9d ⟨*1.4 Helpers* 9d⟩≡

```
Function<String, String> g = x -> join(EMPTY, nCopies(scale, x));
```

Note that we use the fact that the `scale` is captured in the closure.

OK, we also need a mapping function that checks which bits are on and off in the given value using the list of masks. Depending on whether bits are on or off it returns ASCII character | or ␣. It will be a stream of such characters:

10a ⟨*1.4 Helpers* 9d⟩+≡
```
BiFunction<Stream<Integer>, Byte, Stream<String>> h =
    (m, x) -> m.map(mask -> (x & mask) > 0 ? "|" : SPACE);
```

Here `m` is a stream of masks, and `x` is the value `pattern[i]` for some `i`. Note the function returns a stream as well.

Now we can write a function that constructs a column of our LCD digit. Let's call it `k`:

10b ⟨*1.4 Imports* 8a⟩+≡
```
import static java.util.stream.Collectors.joining;
import java.util.function.BiFunction;
```

10c ⟨*1.4 Helpers* 9d⟩+≡
```
BiFunction<Stream<Integer>, Byte, String> k =
    (m, d) -> SPACE + h.apply(m, d).map(x -> g.apply(x)).collect(joining(SPACE)) + SPACE;
```

Note `SPACE`s around (as per requirement) and that the segments within a column are joined by a space.

So far so good. Basically we can now write function `f` that takes a digit pattern `pattern[i]` and returns a stream of strings (in fact, the columns of our LCD digits).

10d ⟨*1.4 Imports* 8a⟩+≡
```
import static java.util.stream.Collectors.toList;
import static java.util.stream.Stream.of;
import java.util.Arrays;
```

10e ⟨*1.4 Helpers* 9d⟩+≡
```
final int digitHeight = scale * 2 + 3;
Function<Byte, Stream<String>> f = x -> Arrays.asList(
  of(k.apply(of(0x40, 0x20), x)),
  ⟨1.4 Middle Column Construction 10f⟩,
  of(k.apply(of(0x02, 0x01), x)),
  of(join(EMPTY, nCopies(digitHeight, SPACE))))
    .stream().reduce(Stream::concat).get();
```

Also note the last line which adds spaces between consecutive digits.

Finally, let's get back to the middle of the digit. To contruct it, we will re-use exactly the same functions we've already defined. We use `h` to obtain the segments that are on and off. Note though that `h` returns | symbols, not the dashes, which are used to indicate horizontal LCD segments. So we will need to replace all occurances of vertical bars with dashes. It's easy to see that the number of spaces between the horizontal segments will be exactly `scale`, which is already captured in the `g` function implementation. Finally, all we need to do, is to replicate the middle column `scale` times. All this can be very easily implemented like so:

10f ⟨*1.4 Middle Column Construction* 10f⟩≡
```
nCopies(scale, h.apply(of(0x10, 0x08, 0x04), x)
    .collect(joining(g.apply(SPACE))).replace('|', '-')).stream()
```

Now we can map the string of digits using our `f` function:

10g ⟨*1.4 Imports* 8a⟩+≡
```
import java.util.List;
```

11a ⟨*1.4 Process* 11a⟩≡
```
List<String> segments = digits.chars().map(x -> x - '0').boxed()
    .flatMap(x -> f.apply(pattern[x])).collect(toList());
```

But remember, this gives us a list of columns of the LCD digits, not rows, so before returning it we need a little post-processing: For each column we take the last characters, concatenate these last characters into a string, and add to a list; then we take the next to the last characters and do the same, and so on:

11b ⟨*1.4 Imports* 8a⟩+≡
```
import static java.util.stream.IntStream.range;
import static java.util.stream.IntStream.rangeClosed;
```

11c ⟨*1.4 Return* 11c⟩≡
```
return rangeClosed(1, digitHeight).boxed()
    .map(j -> digitHeight - j).map(j -> range(0, segments.size() - 1).boxed()
    .map(i -> Character.toString(segments.get(i).charAt(j))).collect(joining())).collect(toList());
```

And that completes the program.

1.5 Graphical Editor

Very straightforward task. The only difficult part being the `fill` operation, but I leave it without comments either as the code is self-explanatory.

12 ⟨*Graphical Editor* 12⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.List;

public class GraphicalEditor {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private int[][] canvas;
    private int m = 0, n = 0;

    private void clear() {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                canvas[j][i] = 'O';
            }
        }
    }

    private void execute(List<String> command) {
        int x, y1, y2, y, x1, x2, c;
        switch (command.get(0)) {
        case "I":
            m = Integer.parseInt(command.get(1));
            n = Integer.parseInt(command.get(2));
            canvas = new int[m][n];
            clear();
            break;
        case "C":
            clear();
            break;
        case "L":
            x = Integer.parseInt(command.get(1)) - 1;
            y = Integer.parseInt(command.get(2)) - 1;
            canvas[x][y] = command.get(3).charAt(0);
            break;
        case "V":
            x = Integer.parseInt(command.get(1)) - 1;
```

```
            y1 = Integer.parseInt(command.get(2)) - 1;
            y2 = Integer.parseInt(command.get(3)) - 1;
            c = command.get(4).charAt(0);
            for (y = Math.min(y1, y2); y <= Math.max(y1, y2); ++y) {
                canvas[x][y] = c;
            }
            break;
        case "H":
            x1 = Integer.parseInt(command.get(1)) - 1;
            x2 = Integer.parseInt(command.get(2)) - 1;
            y = Integer.parseInt(command.get(3)) - 1;
            c = command.get(4).charAt(0);
            for (x = Math.min(x1, x2); x <= Math.max(x1, x2); ++x) {
                canvas[x][y] = c;
            }
            break;
        case "K":
            x1 = Integer.parseInt(command.get(1)) - 1;
            y1 = Integer.parseInt(command.get(2)) - 1;
            x2 = Integer.parseInt(command.get(3)) - 1;
            y2 = Integer.parseInt(command.get(4)) - 1;
            c = command.get(5).charAt(0);
            for (x = x1; x <= x2; ++x) {
                for (y = y1; y <= y2; ++y) {
                    canvas[x][y] = c;
                }
            }
            break;
        case "F":
            x = Integer.parseInt(command.get(1)) - 1;
            y = Integer.parseInt(command.get(2)) - 1;
            int newColor = command.get(3).charAt(0);
            int oldColor = canvas[x][y];
            fill(new Point(x, y), oldColor, newColor);
            break;
        case "S":
            String name = command.get(1);
            System.out.println(name);
            for (int i = 0; i < n; ++i) {
                for (int j = 0; j < m; ++j) {
                    System.out.print((char) canvas[j][i]);
                }
                System.out.println();
            }
            break;
        default:
            break;
        }
    }

    private static class Point {
        final int x, y;
```

```
        public Point(int x, int y) {
            this.x = x;
            this.y = y;
        }
    }

    private void fill(Point pt, int oldColor, int newColor) {
        if (canvas[pt.x][pt.y] != oldColor || oldColor == newColor) {
            return;
        }
        Deque<Point> q = new ArrayDeque<>();
        q.addLast(pt);
        canvas[pt.x][pt.y] = newColor;
        while (!q.isEmpty()) {
            pt = q.pop();
            if (pt.x + 1 < m && canvas[pt.x + 1][pt.y] == oldColor) {
                canvas[pt.x + 1][pt.y] = newColor;
                q.addLast(new Point(pt.x + 1, pt.y));
            }
            if (pt.x - 1 >= 0 && canvas[pt.x - 1][pt.y] == oldColor) {
                canvas[pt.x - 1][pt.y] = newColor;
                q.addLast(new Point(pt.x - 1, pt.y));
            }
            if (pt.y + 1 < n && canvas[pt.x][pt.y + 1] == oldColor) {
                canvas[pt.x][pt.y + 1] = newColor;
                q.addLast(new Point(pt.x, pt.y + 1));
            }
            if (pt.y - 1 >= 0 && canvas[pt.x][pt.y - 1] == oldColor) {
                canvas[pt.x][pt.y - 1] = newColor;
                q.addLast(new Point(pt.x, pt.y - 1));
            }
        }
    }

    public static void main(String[] args) throws IOException {
        GraphicalEditor editor = new GraphicalEditor();
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<String> command = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).collect(toList());
            if (command.get(0).equalsIgnoreCase("X")) {
                break;
            }
            editor.execute(command);
        }
    }
}
```

1.6 Interpreter

This task is disappointingly straightforward.

15 ⟨*Interpreter* 15⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

class Interpreter {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    private static int interpret(List<Integer> input) {
        int[] reg = new int[10];
        int[] ram = new int[1000];
        for (int i = 0; i < input.size(); ++i) {
            ram[i] = input.get(i);
        }
        int pc = 0;
        int r = 0;
        while (ram[pc] != 100) {
            int op = ram[pc];
            int c = (op / 100) % 10;
            pc = (pc + 1) % 1000;
            r++;
            switch (c) {
            case 2:
                reg[(op / 10) % 10] = op % 10;
                break;
            case 3:
                reg[(op / 10) % 10] = (reg[(op / 10) % 10] + (op % 10)) % 1000;
                break;
            case 4:
                reg[(op / 10) % 10] = (reg[(op / 10) % 10] * (op % 10)) % 1000;
                break;
            case 5:
                reg[(op / 10) % 10] = reg[op % 10];
                break;
            case 6:
                reg[(op / 10) % 10] = (reg[(op / 10) % 10] + reg[op % 10]) % 1000;
                break;
            case 7:
                reg[(op / 10) % 10] = (reg[(op / 10) % 10] * reg[op % 10]) % 1000;
                break;
            case 8:
                reg[(op / 10) % 10] = ram[reg[op % 10]];
                break;
            case 9:
                ram[reg[op % 10]] = reg[(op / 10) % 10];
                break;
            case 0:
```

```
                if (reg[op % 10] != 0) {
                    pc = reg[(op / 10) % 10];
                }
                break;
            }
        }

        return r + 1;
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.valueOf(reader.readLine().trim());
        reader.readLine();
        String currentLine;
        for (int i = 0; i < n; ++i) {
            List<Integer> input = new ArrayList<Integer>();
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().equalsIgnoreCase("")) {
                input.add(Integer.parseInt(currentLine.trim()));
            }
            System.out.println(interpret(input));
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
}
```

1.7 Check The Check

This task is trivial.

17 ⟨*Check The Check* 17⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class CheckTheCheck {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final int BOARD_SIZE = 8;

    private static final int[][] king = new int[][] {
            { -1, -1 }, { 0, -1 }, { 1, -1 }, { -1, 0 }, { 1, 0 }, { -1, 1 },
            { 0, 1 }, { 1, 1 }
    };
    private static final int[][] knight = new int[][] {
            { -2, -1 }, { -1, -2 }, { 1, -2 }, { 2, -1 }, { 2, 1 }, { 1, 2 },
            { -1, 2 }, { -2, 1 }
    };
    private static final int[][] bishop = new int[][] {
            { 1, 1 }, { -1, -1 }, { -1, 1 }, { 1, -1 }
    };
    private static final int[][] rook = new int[][] {
            { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 }
    };
    private static final int[][] queen = new int[][] {
            { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 }, { 1, 1 }, { -1, -1 },
            { -1, 1 }, { 1, -1 }
    };
    private static final int[][] white_pawn = new int[][] {
            { -1, -1 }, { -1, 1 }
    };
    private static final int[][] black_pawn = new int[][] {
            { 1, -1 }, { 1, 1 }
    };

    private static boolean isWithinBounds(int d, int v) {
        if (d == 0) {
            return true;
        }
        return d > 0 ? (v < BOARD_SIZE) : (v >= 0);
    }

    private static void check(int di, int dj, int i, int j, int[][] board,
            int[][] attackBoard) {
        int c = j + dj;
        int r = i + di;
        while (isWithinBounds(dj, c) && isWithinBounds(di, r)) {
            attackBoard[r][c] = 1;
            if (board[r][c] != '.')
                break;
```

```
            r += di;
            c += dj;
        }
    }

    private static void check(int[][] d, int i, int j, int[][] board,
            int[][] attackBoard) {
        if (d == king || d == knight || d == black_pawn || d == white_pawn) {
            for (int k = 0; k < d.length; ++k) {
                if ((i + d[k][0] >= 0 && i + d[k][0] < BOARD_SIZE) &&
                        (j + d[k][1] >= 0 && j + d[k][1] < BOARD_SIZE)) {
                    attackBoard[i + d[k][0]][j + d[k][1]] = 1;
                }
            }
            return;
        }

        for (int k = 0; k < d.length; ++k) {
            check(d[k][0], d[k][1], i, j, board, attackBoard);
        }
    }

    private static int[] locate(int v, int[][] board) {
        for (int i = 0; i < BOARD_SIZE; ++i) {
            for (int j = 0; j < BOARD_SIZE; ++j) {
                if (board[i][j] == v) {
                    return new int[] { i, j };
                }
            }
        }
        return null;
    }

    private static String checkTheCheck(int[][] board) {
        int[][] attackBoardWhites = new int[BOARD_SIZE][BOARD_SIZE];
        int[][] attackBoardBlacks = new int[BOARD_SIZE][BOARD_SIZE];

        for (int i = 0; i < BOARD_SIZE; ++i) {
            for (int j = 0; j < BOARD_SIZE; ++j) {
                if (board[i][j] == 'R' || board[i][j] == 'r') {
                    check(rook, i, j, board, board[i][j] == 'R'
                            ? attackBoardWhites : attackBoardBlacks);
                }
                if (board[i][j] == 'B' || board[i][j] == 'b') {
                    check(bishop, i, j, board, board[i][j] == 'B'
                            ? attackBoardWhites : attackBoardBlacks);
                }
                if (board[i][j] == 'K' || board[i][j] == 'k') {
                    check(king, i, j, board, board[i][j] == 'K'
                            ? attackBoardWhites : attackBoardBlacks);
                }
                if (board[i][j] == 'N' || board[i][j] == 'n') {
                    check(knight, i, j, board, board[i][j] == 'N'
```

```
                            ? attackBoardWhites : attackBoardBlacks);
                }
                if (board[i][j] == 'Q' || board[i][j] == 'q') {
                    check(queen, i, j, board, board[i][j] == 'Q'
                            ? attackBoardWhites : attackBoardBlacks);
                }
                if (board[i][j] == 'P' || board[i][j] == 'p') {
                    boolean isWhite = board[i][j] == 'P';
                    check(isWhite ? white_pawn : black_pawn, i, j, board,
                            isWhite ? attackBoardWhites : attackBoardBlacks);
                }
            }
        }

        int[] wk = locate('K', board);
        int[] bk = locate('k', board);

        boolean bkCheck = (attackBoardWhites[bk[0]][bk[1]] == 1);
        boolean wkCheck = (attackBoardBlacks[wk[0]][wk[1]] == 1);

        if (wkCheck) {
            return "white king is in check.";
        }
        if (bkCheck) {
            return "black king is in check.";
        }

        return "no king is in check.";
    }

    public static void main(String[] args) throws IOException {
        boolean empty = true;
        int game = 1;
        do {
            int[][] board = new int[BOARD_SIZE][BOARD_SIZE];
            empty = true;
            for (int i = 0; i < BOARD_SIZE; ++i) {
                String currentLine = reader.readLine();
                for (int j = 0; j < BOARD_SIZE; ++j) {
                    board[i][j] = currentLine.charAt(j);
                    empty = empty && board[i][j] == '.';
                }
            }
            if (empty) {
                break;
            }
            System.out.println("Game #" + game + ": " + checkTheCheck(board));
            game++;
        } while (reader.readLine().trim().equals(""));
    }
}
```

1.8 Australian Voting

This task is very straightforward.

Let's sort out input/output first as usual. We will assume our function that does the election is called `elect`, and that it takes two arguments, a list of candidates and a list of ballots, and returns a list of those who win the election. Note the `ballots` is a list of deques, that's because we will be checking the next candidate in the ranking, and note that we subtract one from each index in the ballots, this is for easier access to the arrays, as they are indexed from 0.

20 ⟨*Australian Voting* 20⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;
⟨1.8 Imports 21e⟩

class AustralianVoting {
    private static final String EMPTY = "";
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    private static List<String> elect(List<String> candidates,
            List<Deque<Integer>> ballots) {
        ⟨1.8 Implementation 21a⟩
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.valueOf(reader.readLine().trim());
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            int count = Integer.valueOf(reader.readLine().trim());
            List<String> candidates = reader.lines().limit(count).collect(toList());
            List<Deque<Integer>> ballots = new ArrayList<Deque<Integer>>();
            String currentLine = EMPTY;
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.equalsIgnoreCase(EMPTY)) {
                ballots.add(new ArrayDeque<Integer>(stream(currentLine.trim().split(" "))
                        .filter(x -> !x.equals(EMPTY))
                        .map(Integer::parseInt).map(x -> x - 1).collect(toList())));
            }
            elect(candidates, ballots).forEach(System.out::println);
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
```

```
}
```

Now let's implement `elect` function. First we need to figure out the majority. That's easy as that's simply the half of the ballots plus one.

21a ⟨*1.8 Implementation* 21a⟩≡
```
final int majority = ballots.size() / 2 + 1;
```

Because candidates in the ballots are numbered by their indexes in the table, let's have an array of `int`s, which will hold the number of votes.

21b ⟨*1.8 Implementation* 21a⟩+≡
```
final int[] counter = new int[candidates.size()];
```

Now let's count votes for the candidates specified as first in the ballots:

21c ⟨*1.8 Implementation* 21a⟩+≡
```
ballots.stream().map(Deque::peek).forEach(x -> counter[x]++);
```

After this point two things may happen: We will have somebody who got the majority of the votes, in which case we know the winner (or winners), or not, in which case we repeat the procedure described in the problem statement.

21d ⟨*1.8 Implementation* 21a⟩+≡
```
while (true) {
    ⟨1.8 Election loop 21f⟩
}
```

OK, because some candidates may get equal number of votes we need to group them by votes. This is pretty easy:

21e ⟨*1.8 Imports* 21e⟩≡
```
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.IntStream.range;
import java.util.Map;
```

21f ⟨*1.8 Election loop* 21f⟩≡
```
Map<Integer, List<Integer>> result = range(0, candidates.size()).boxed()
    .filter(x -> counter[x] >= 0).collect(groupingBy(i -> counter[i], toList()));
```

Pay attention to the candidates who got zeros votes, because those will need to go through the elimination process too.

Now we need to find out who got the most votes and who got the least:

21g ⟨*1.8 Election loop* 21f⟩+≡
```
int max = result.keySet().stream().max(Integer::compareTo).get();
int min = result.keySet().stream().min(Integer::compareTo).get();
```

It's easy to see that if $max \geq majority$ or $max = min$, then we know the winner:

21h ⟨*1.8 Election loop* 21f⟩+≡
```
if (max >= majority || max == min) {
    return result.get(max).stream().map(x -> candidates.get(x)).collect(toList());
}
```

Otherwise we need to re-distribute the votes. We get the indexes of the candidates who got the least votes and mark them as having -1 votes in the `counter` array so that we never consider them again in our filters.

21i ⟨*1.8 Election loop* 21f⟩+≡
```
List<Integer> eliminated = result.get(min);
eliminated.forEach(x -> counter[x] = -1);
```

Now we need to remove the eliminated candidates from the ballots. However it needs to be done carefully. We make note of who is currently the first in the ballot. If after the elimination process the first in the rank has changed, we need to take that into account. This is captured in the following chunk:

22a ⟨*1.8 Election loop* 21f⟩+≡

```
ballots.forEach(b -> {
    int first = b.peek();
    eliminated.forEach(x -> b.remove(x));
    counter[b.peek()] += (first != b.peek()) ? 1 : 0;
});
```

2 Data Structures

2.1 Jolly Jumpers

This task must be a joke.

22b ⟨*Jolly Jumpers* 22b⟩≡

```
import static java.lang.Math.abs;
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

class JollyJumpers {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> nums = stream(currentLine.trim().split(" ")).filter(x -> !x.equals(""))
                .skip(1).map(Integer::parseInt).collect(toList());
            int[] diffs = range(0, nums.size() - 1)
                .map(i -> abs(nums.get(i) - nums.get(i + 1))).distinct().sorted().toArray();
            boolean isJolly = range(0, diffs.length).boxed()
                .map(i -> diffs[i] == i + 1).reduce(true, (x, y) -> x && y);
            System.out.println(diffs.length == nums.size() - 1 && isJolly ? "Jolly" : "Not jolly");
        }
    }
}
```

2.2 Poker Hands

The task isn't particularly difficult in any way, but requires some lengthy coding. Anyway, here's the whole program without comments as the code is self-explanatory.

23 ⟨*Poker Hands* 23⟩≡

```
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class PokerHands {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static class Card {
        final int value;
        final int suit;
        private final static Map<Character, Integer> map = initialize();

        private static Map<Character, Integer> initialize() {
            Map<Character, Integer> map = new HashMap<Character, Integer>();
            for (char c = '2'; c <= '9'; ++c) {
                map.put(c, c - '0');
            }
            map.put('T', 10);
            map.put('J', 11);
            map.put('Q', 12);
            map.put('K', 13);
            map.put('A', 14);
            map.put('C', 1);
            map.put('D', 2);
            map.put('H', 3);
            map.put('S', 4);
            return map;
        }

        public Card(char value, char suit) {
            this.value = map.get(value);
            this.suit = map.get(suit);
        }
    }

    private static class Hand {
        private List<Card> hand;
        private int category = 0;
        private int rank = 0;
```

```
public Hand(List<Card> hand) {
    this.hand = hand;
    Collections.sort(hand, (x, y) -> Integer.compare(x.value, y.value));
    straightFlush();
    fourOfAKind();
    fullHouse();
    flush();
    straight();
    threeOfAKind();
    twoPairs();
    pair();
    highCard();
}

private void highCard() {
    if (category != 0) {
        return;
    }

    category = 1;
    for (int i = hand.size() - 1; i >= 0; --i) {
        rank = (rank * 100) + hand.get(i).value;
    }
}

private void pair() {
    if (category != 0) {
        return;
    }

    Map<Integer, Integer> count = groups();
    List<Integer> pairs = new ArrayList<Integer>();
    for (Integer c : count.keySet()) {
        if (count.get(c) == 2) {
            pairs.add(c);
        }
    }

    if (pairs.size() == 1) {
        category = 2;
        rank = pairs.get(0);
        for (int i = hand.size() - 1; i >= 0; --i) {
            if (hand.get(i).value != pairs.get(0)) {
                rank = (rank * 100) + hand.get(i).value;
            }
        }
    }
}

private void twoPairs() {
    if (category != 0) {
        return;
```

```
        }

        Map<Integer, Integer> count = groups();
        List<Integer> pairs = new ArrayList<Integer>();
        int singleton = 0;
        for (Integer c : count.keySet()) {
            if (count.get(c) == 2) {
                pairs.add(c);
            } else if (count.get(c) == 1) {
                singleton = c;
            }
        }

        if (pairs.size() == 2) {
            category = 3;
            Collections.sort(pairs);
            for (int i = pairs.size() - 1; i >= 0; --i) {
                rank = (rank * 100) + pairs.get(i);
            }
            rank = (rank * 100) + singleton;
        }
    }

    private void threeOfAKind() {
        if (category != 0) {
            return;
        }

        Map<Integer, Integer> count = groups();
        for (Integer c : count.keySet()) {
            if (count.get(c) == 3) {
                category = 4;
                rank = c;
                for (int i = hand.size() - 1; i >= 0; --i) {
                    if (hand.get(i).value != c) {
                        rank = (rank * 100) + hand.get(i).value;
                    }
                }
                return;
            }
        }
    }

    private void straight() {
        if (category != 0) {
            return;
        }
        int value = hand.get(0).value;
        for (int i = 1; i < hand.size(); ++i) {
            if (hand.get(i).value - value == 1) {
                value = hand.get(i).value;
                continue;
            } else {
```

```
            return;
        }
    }
    category = 5;
    rank = hand.get(hand.size() - 1).value;
}

private void flush() {
    if (category != 0) {
        return;
    }

    int suit = hand.get(0).suit;
    for (int i = 1; i < hand.size(); ++i) {
        if (hand.get(i).suit == suit) {
            continue;
        } else {
            return;
        }
    }

    category = 6;
    for (int i = hand.size() - 1; i >= 0; --i) {
        rank = (rank * 100) + hand.get(i).value;
    }
}

private void fourOfAKind() {
    fourOfAKindFullHouse(4, 8);
}

private void fullHouse() {
    fourOfAKindFullHouse(3, 7);
}

private void fourOfAKindFullHouse(int n, int cat) {
    if (category != 0) {
        return;
    }

    Map<Integer, Integer> count = groups();
    if (count.size() != 2) {
        return;
    }

    List<Integer> keys = new ArrayList<>(count.keySet());
    if (count.get(keys.get(0)) == n) {
        category = cat;
        rank = keys.get(0);
    } else if (count.get(keys.get(1)) == n) {
        category = cat;
        rank = keys.get(1);
    }
```

```
        }

        private void straightFlush() {
            if (category != 0) {
                return;
            }

            int suit = hand.get(0).suit;
            int value = hand.get(0).value;
            for (int i = 1; i < hand.size(); ++i) {
                if (hand.get(i).suit == suit &&
                        hand.get(i).value - value == 1) {
                    value = hand.get(i).value;
                    continue;
                } else {
                    return;
                }
            }

            category = 9;
            rank = hand.get(hand.size() - 1).value;
        }

        private Map<Integer, Integer> groups() {
            Map<Integer, Integer> count = new HashMap<Integer, Integer>();
            for (Card c : hand) {
                count.putIfAbsent(c.value, 0);
                count.put(c.value, count.get(c.value) + 1);
            }
            return count;
        }

    }

    private static int compare(Hand black, Hand white) {
        int compareCategory = Integer.compare(black.category, white.category);
        if (compareCategory == 0) {
            return Integer.compare(black.rank, white.rank);
        }
        return compareCategory;
    }

    private static int compare(String currentLine) {
        List<String> hands = Arrays.stream(currentLine.split(" "))
                .filter(x -> !x.equals("")).collect(toList());
        return compare(getHand(hands.subList(0, 5)),
                getHand(hands.subList(5, 10)));
    }

    private static Hand getHand(List<String> h) {
        return new Hand(h.stream().map(x -> new Card(x.charAt(0), x.charAt(1)))
                .collect(toList()));
    }
```

```
    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            int cmp = compare(currentLine);
            if (cmp == 0) {
                System.out.println("Tie.");
            } else {
                System.out.println(cmp > 0 ? "Black wins." : "White wins.");
            }
        }
    }

}
```

2.3 Hartals

It seems that for this tasks you'd need to use GCDs and LCMs to figure out the overlapping days etc. But in reality this is not needed. If you look at the input numbers you'll notice that they are very small, so a simulation on top of a bitmap will be just fine. This results in a much shorter and less complicated code.

29 ⟨*Hartals* 29⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.BitSet;

public class Hartals {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static void set(int n, BitSet res, int s, int l, boolean v) {
        while (s <= n) {
            res.set(s, v);
            s += l;
        }
    }

    private static int solve(int n, int[] h) {
        BitSet res = new BitSet(n + 1);
        for (int i = 0; i < h.length; ++i) {
            set(n, res, h[i], h[i], true);
        }
        set(n, res, 6, 7, false);
        set(n, res, 7, 7, false);
        int count = 0;
        for (int i = 0; i < res.size(); ++i) {
            count += res.get(i) ? 1 : 0;
        }
        return count;
    }

    public static void main(String[] args) throws IOException {
        int cases = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < cases; ++i) {
            int n = Integer.parseInt(reader.readLine().trim());
            int p = Integer.parseInt(reader.readLine().trim());
            int[] h = new int[p];
            for (int j = 0; j < p; ++j) {
                h[j] = Integer.parseInt(reader.readLine().trim());
            }
            System.out.println(solve(n, h));
        }
    }

}
```

2.4 Crypt Kicker

This task is a lot of fun! To solve it we are going to need a very good bookkeeping discipline.

Let's outline the general strategy. First thing to do is to to group the words by length. Then we need to come up with a method to compare a dictionary word and an encrypted word by looking at their patterns. So we somehow need to tell if the words "abbc" has a similar pattern as "xyyz". But this is very easy: we scan a word from left to right and output an index of the first occurrence of the character, or current index if it's the first occurrence. So for example "abbc" and "xyyz" would both have a pattern 1 2 2 3. Using a pattern and the word length we can find words from the dictionary that could be the potential matches for an encrypted word.

We start with the longest word (if multiple words of the same length, then any words of such length) and we find all the words from the dictionary that have the same length, the same pattern, and agree with the mapping found so far. By the mapping found so far we mean the following: if some previous word has been matched with a candidate, we note the mapping. So if we matched "abbc" with "xyyz" we now know that a maps to x, b maps to y, and c maps to z. This means that if we are now trying to match another word, say "zy", we can eliminate candidates such as "bc", because we now assume that z maps to c, not b. Once we filtered all the potential candidates we try to match the first candidate from the list and move on to the next word. If at any step we fail to find any candidate word, we return one step back, and try another word in the list, if the list is exhausted, we move one step back again. If we exhausted all the lists, then the decryption is impossible. For simplicity of implementation we will implement it as a recursion.

OK, now we just need to write code.

First input/output. The main class will be initialized by a dictionary and will have just one method `decrypt` that will take a string and return either a decrypted text or stars, as per problem statement.

30 ⟨*Crypt Kicker* 30⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

⟨2.4 Imports 31a⟩

class CryptKicker {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    ⟨2.4 Variables 31f⟩

    ⟨2.4 Constructor 31g⟩

    ⟨2.4 Methods 31b⟩

    public static void main(String[] args) throws IOException {
        String currentLine;
        final int size = Integer.parseInt(reader.readLine().trim());
        final List<String> dictionary = reader.lines().limit(size).collect(toList());
        CryptKicker cryptKicker = new CryptKicker(dictionary);
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
```

```
                System.out.println(cryptKicker.decrypt(currentLine));
            }
        }
    }
```

Let's write the method that gives us the pattern of a given word. This method does exactly the thing we've described above.

31a ⟨*2.4 Imports* 31a⟩≡
```
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;
```

31b ⟨*2.4 Methods* 31b⟩≡
```
private static List<Integer> getPattern(String word) {
    return range(0, word.length()).map(i -> word.indexOf(word.charAt(i)))
        .boxed().collect(toList());
}
```

And let's add a helper method that for a given list of words gives a map. (Note that we take distinct words as the words in the input dictionary aren't necessarily unique.)

31c ⟨*2.4 Imports* 31a⟩+≡
```
import static java.util.function.Function.identity;
import static java.util.stream.Collectors.toMap;
import java.util.List;
import java.util.Map;
import java.util.Deque;
```

31d ⟨*2.4 Methods* 31b⟩+≡
```
private static Map<String, List<Integer>> getPatterns(Deque<String> words) {
    return words.stream().distinct().collect(
        toMap(identity(), CryptKicker::getPattern));
}
```

Now we can do the constructor. In the constructor we will group the words by length and get their patterns.

31e ⟨*2.4 Imports* 31a⟩+≡
```
import static java.util.stream.Collectors.groupingBy;
import java.util.ArrayDeque;
```

31f ⟨*2.4 Variables* 31f⟩≡
```
private final Map<Integer, List<String>> dictionary;
private final Map<String, List<Integer>> patterns;
```

31g ⟨*2.4 Constructor* 31g⟩≡
```
public CryptKicker(List<String> inputDictionary) {
    dictionary = inputDictionary.stream()
        .collect(groupingBy(String::length));
    patterns = getPatterns(new ArrayDeque<>(inputDictionary));
}
```

We will also need a function to compare two given patterns. This is easy:

31h ⟨*2.4 Imports* 31a⟩+≡
```
import static java.lang.Math.abs;
```

32a ⟨*2.4 Methods* 31b⟩+≡
```
private static boolean compare(List<Integer> a, List<Integer> b) {
    return a.size() == b.size() &&
        range(0, a.size()).map(i -> abs(a.get(i) - b.get(i))).sum() == 0;
}
```

Let's sort out the variables that we are going to need. We will need a map that will hold patterns of the encrypted words for the given input string.

32b ⟨*2.4 Variables* 31f⟩+≡
```
private Map<String, List<Integer>> encryptedPatterns;
```

This variable could have been passed around via argument to the methods, because this variable's contents depend on each encrypted input line. But I have chosen to just have this as a private member.

We will also need to keep track of the words that have been mapped.

32c ⟨*2.4 Imports* 31a⟩+≡
```
import java.util.HashSet;
import java.util.Set;
```

32d ⟨*2.4 Variables* 31f⟩+≡
```
private final Set<String> mappedWords = new HashSet<>();
```

And we will need the mappings themselves. We will keep both the direct mapping and the reversed mappings in the arrays, where an index is the ASCII character code and the value is another ASCII character code. Because ASCII characters for the lower case letters go from 97 to 122 it should be enough to just create an array of no more than 128 bytes. We could have created a smaller array, but in that case we would need to adjust the indexes which would clutter the code unnecessarily.

The `counter` array will keep track of how many words have used this character mapping so far. This is needed because we will be mapping and unmapping the words multiple times during the search. An empty array will denote unsuccessful mapping.

32e ⟨*2.4 Variables* 31f⟩+≡
```
private static final int MAX_SIZE = 128;
private static final int[] NOT_FOUND = new int[0];
private final int[] dirMapping = new int[MAX_SIZE];
private final int[] revMapping = new int[MAX_SIZE];
private final int[] counter = new int[MAX_SIZE];
```

Let's implement `mapWord` and `unmapWord` methods. Note that they keep track (with help of `counter`) of how many words have used a specific character mapping.

33a ⟨*2.4 Methods* 31b⟩+≡

```
private void mapWord(String e, String c) {
    mappedWords.add(c);
    for (int i = 0; i < e.length(); ++i) {
        dirMapping[e.charAt(i)] = c.charAt(i);
        counter[e.charAt(i)]++;
        revMapping[c.charAt(i)] = e.charAt(i);
    }
}

private void unmapWord(String e, String c) {
    mappedWords.remove(c);
    for (int i = 0; i < e.length(); ++i) {
        counter[e.charAt(i)]--;
        if (counter[e.charAt(i)] == 0) {
            revMapping[dirMapping[e.charAt(i)]] = 0;
            dirMapping[e.charAt(i)] = 0;
        }
    }
}
```

We need to keep track of these mapping and `counter` because of the filtering. For example, if we mapped the word "abc" to "xyz" and the word "ab" to "xy", we now know that a maps to x, b maps to y, and c maps to z. So we can find that the word's "ab" mapping to "xy" is a valid mapping and so we can map that too. If for some reason we unmap the word "abc", our mapping arrays should still keep the mapping of a to x, and b to y, because we haven't unmapped the word "ab".

Now let's implement the filtering. This function take an encrypted word and does the following filtering. First, it gets all the words from the dictionary of the same length. Then it filters out the words that have been mapped already and the words that don't have the same pattern. Next, it checks if this word agrees with the mapping (may be partial) of the mappings found so far. If the word passes all this filtering, it is add to the list, which then returned as the result.

33b ⟨*2.4 Imports* 31a⟩+≡

```
import java.util.ArrayList;
```

34a ⟨*2.4 Methods* 31b⟩+≡

```
private List<String> filter(String encrypted) {
    List<String> matchedWords = new ArrayList<String>();
    for (String word : dictionary.get(encrypted.length())) {
        if (mappedWords.contains(word) ||
            !compare(encryptedPatterns.get(encrypted), patterns.get(word))) {
            continue;
        }

        boolean matched = true;
        for (int i = 0; i < word.length() && matched; ++i) {
            boolean unmapped = dirMapping[encrypted.charAt(i)] == 0;
            boolean mapped = dirMapping[encrypted.charAt(i)] == word.charAt(i);
            boolean unused = revMapping[word.charAt(i)] == 0;
            matched = (unmapped && unused) || mapped;
        }
        if (matched) {
            matchedWords.add(word);
        }
    }
    return matchedWords;
}
```

We can now implement the recursive search method. It takes a deque of encrypted words and then tries to map them to the dictionary. (A deque because it has convenient methods such as `pop` and `push`.) This method assumes that the words in the deque are sorted by length in descending order.

34b ⟨*2.4 Methods* 31b⟩+≡

```
private boolean map(Deque<String> encryptedWords) {
    if (encryptedWords.isEmpty()) {
        return true;
    }
    String encryptedWord = encryptedWords.pop();
    List<String> words = filter(encryptedWord);
    for (String candidate : words) {
        mapWord(encryptedWord, candidate);
        if (map(encryptedWords)) {
            return true;
        }
        unmapWord(encryptedWord, candidate);
    }
    encryptedWords.push(encryptedWord);
    return false;
}
```

Let's add another helper method that will do the clearing up and initialization of the data structures:

34c ⟨*2.4 Imports* 31a⟩+≡

```
import java.util.Arrays;
```

35a ⟨*2.4 Methods* 31b⟩+≡

```
private int[] findMapping(Deque<String> encryptedWords) {
    encryptedPatterns = getPatterns(encryptedWords);
    mappedWords.clear();
    Arrays.fill(dirMapping, 0);
    Arrays.fill(revMapping, 0);
    Arrays.fill(counter, 0);
    return map(encryptedWords) ? dirMapping : NOT_FOUND;
}
```

Finally, we can now implement `decrypt` method:

35b ⟨*2.4 Imports* 31a⟩+≡

```
import static java.util.Comparator.comparing;
```

35c ⟨*2.4 Methods* 31b⟩+≡

```
public String decrypt(String input) {
    StringBuilder result = new StringBuilder();
    int[] mapping = findMapping(
            new ArrayDeque<>(Arrays.stream(input.trim().split(" "))
                .filter(x -> !x.equals("")).distinct()
                .sorted(comparing(String::length).reversed()).collect(toList())));
    input.chars().map(c -> c != ' ' ? (mapping != NOT_FOUND ? mapping[c] : '*') : c)
        .forEachOrdered(x -> result.append((char) x));
    return result.toString();
}
```

2.5 Stack 'em Up

Easy.

36 ⟨*Stack em Up* 36⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Stream;

public class StackEmUp {
    private static final int DECK_SIZE = 52;

    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static final Map<Integer, String> map = initialize();

    private static Map<Integer, String> initialize() {
        Map<Integer, String> map = new HashMap<>();
        int k = 0;
        for (String suit : Stream.of("Clubs", "Diamonds", "Hearts", "Spades")
                .collect(toList())) {
            for (int i = 2; i <= 10; ++i) {
                map.put(Integer.valueOf(k++), i + " of " + suit);
            }
            map.put(Integer.valueOf(k++), "Jack of " + suit);
            map.put(Integer.valueOf(k++), "Queen of " + suit);
            map.put(Integer.valueOf(k++), "King of " + suit);
            map.put(Integer.valueOf(k++), "Ace of " + suit);
        }
        return map;
    }

    private static List<Integer> newDeck() {
        return Stream.iterate(0, i -> i + 1).limit(DECK_SIZE)
                .collect(toList());
    }

    private static List<Integer> apply(List<Integer> deck,
            List<Integer> shuffle) {
        List<Integer> output = newDeck();
        for (int j = 0; j < shuffle.size(); ++j) {
            output.set(j, deck.get(shuffle.get(j)));
        }
```

```
        return output;
    }

    private static List<Integer> shuffle(List<Integer> shuffleIndexes,
            List<List<Integer>> shuffles) {
        List<Integer> deck = newDeck();
        for (Integer i : shuffleIndexes) {
            deck = apply(deck, shuffles.get(i));
        }
        return deck;
    }

    public static void main(String[] args) throws IOException {
        int cases = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        for (int i = 0; i < cases; ++i) {
            int n = Integer.parseInt(reader.readLine().trim());
            List<Integer> shuffles = new ArrayList<>();
            String currentLine;
            while (shuffles.size() < n * DECK_SIZE) {
                currentLine = reader.readLine().trim();
                shuffles.addAll(stream(currentLine.split(" "))
                        .filter(x -> !x.equals(""))
                        .map(Integer::parseInt)
                        .map(x -> x - 1)
                        .collect(toList()));
            }
            List<List<Integer>> shuffleList = new ArrayList<List<Integer>>();
            for (int j = 0; j < n; ++j) {
                shuffleList.add(shuffles.subList(j * DECK_SIZE,
                        j * DECK_SIZE + DECK_SIZE));
            }
            List<Integer> shuffleIndexes = new ArrayList<>();
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().equalsIgnoreCase("")) {
                shuffleIndexes.add(Integer.parseInt(currentLine.trim()) - 1);
            }
            shuffle(shuffleIndexes, shuffleList)
                    .forEach(x -> System.out.println(map.get(x)));
            if (i < cases - 1) {
                System.out.println();
            }
        }

    }

}
```

2.6 Erdös Numbers

To solve this task one just needs to apply breadth-first search algorithm. Very straightforward.

38 ⟨*Erdos Numbers* 38⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class ErdosNumbers {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final Pattern namePattern = Pattern
            .compile("[\\w^.,]+\\s*,\\s*(\\w\\.)+\\s*[,:]");

    private static final String ERDOS = "Erdos, P.";

    private static void add(Map<String, Set<String>> graph,
            List<String> names) {
        for (int i = 0; i < names.size(); ++i) {
            String currName = names.get(i);
            if (!graph.containsKey(names.get(i))) {
                graph.put(currName, new HashSet<String>());
            }
            names.forEach(name -> {
                if (!currName.equalsIgnoreCase(name)) {
                    graph.get(currName).add(name);
                }
            });
        }
    }

    private static List<String> getNames(String input) {
        List<String> names = new ArrayList<>();
        Matcher m = namePattern.matcher(input);
        while (m.find()) {
            names.add(input.substring(m.start(), m.end() - 1).trim());
        }
        return names;
    }
```

```
    private static Map<String, Integer> getAnswer(
            Map<String, Set<String>> graph) {
        Deque<String> q = new ArrayDeque<>();
        Set<String> s = new HashSet<String>();
        Map<String, Integer> r = new HashMap<>();
        q.push(ERDOS);
        r.put(ERDOS, Integer.valueOf(0));

        while (!q.isEmpty()) {
            String n = q.pop();
            int depth = r.get(n);
            for (String x : graph.get(n)) {
                if (!s.contains(x)) {
                    s.add(x);
                    q.addLast(x);
                    r.put(x, Integer.valueOf(depth + 1));
                }
            }
        }

        return r;
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            List<Integer> nm = stream(reader.readLine().trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            Map<String, Set<String>> graph = new HashMap<>();
            for (int j = 0; j < nm.get(0); ++j) {
                add(graph, getNames(reader.readLine().trim()));
            }
            Map<String, Integer> r = getAnswer(graph);
            System.out.println("Scenario " + (i + 1));
            for (int j = 0; j < nm.get(1); ++j) {
                String name = reader.readLine().trim();
                System.out.println(name + " " +
                        (r.containsKey(name) ? r.get(name) : "infinity"));
            }
        }
    }
}
```

2.7 Contest Scoreboard

With this task one must be careful not to add penalties to the tasks that some teams attempted but never solved, that's the only tricky thing that might not be obvious from the problem statement.

40 ⟨*Contest Scoreboard* 40⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class ContestScoreboard {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static class Team implements Comparable<Team> {
        private final int num;
        private int totalTime;
        private Set<Integer> solved = new HashSet<>();
        private final int[] penalties = new int[10];

        @Override
        public String toString() {
            return num + " " + solved.size() + " " + getTotalTime();
        }

        public Team(int num) {
            this.num = num;
        }

        public int getTotalTime() {
            int time = totalTime;
            for (Integer problemId : solved) {
                time += penalties[problemId];
            }
            return time;
        }

        public void update(Integer problem, Integer time, String verdict) {
            switch (verdict) {
            case "C":
                if (solved.add(problem)) {
                    totalTime += time;
                }
                break;
            case "I":
```

```
                if (!solved.contains(problem)) {
                    penalties[problem] += 20;
                }
                break;
            default:
                break;
            }
        }

        @Override
        public int compareTo(Team o) {
            int solvedCmp = Integer.compare(o.solved.size(),
                    this.solved.size());
            if (solvedCmp == 0) {
                int timeCmp = Integer.compare(getTotalTime(), o.getTotalTime());
                if (timeCmp == 0) {
                    return Integer.compare(this.num, o.num);
                }
                return timeCmp;
            }
            return solvedCmp;
        }
    }

    public static void main(String[] args) throws IOException {
        int cases = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        String currentLine = null;
        for (int i = 0; i < cases; ++i) {
            Map<Integer, Team> participants = new HashMap<>();
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().equals("")) {
                List<String> inputLine = stream(currentLine.trim().split(" "))
                        .filter(x -> !x.equals(""))
                        .collect(toList());
                Integer num = Integer.parseInt(inputLine.get(0));
                Integer problem = Integer.parseInt(inputLine.get(1));
                Integer time = Integer.parseInt(inputLine.get(2));
                String verdict = inputLine.get(3);
                if (!participants.containsKey(num)) {
                    participants.put(num, new Team(num));
                }
                participants.get(num).update(problem, time, verdict);
            }
            participants.values().stream().sorted()
                    .forEach(System.out::println);
            if (i < cases - 1) {
                System.out.println();
            }
        }
    }
}
```

2.8 Yahtzee

This task's solution is going to be a bit lengthy due to necessary coding. The solution itself though is not so complicated. Notice that for the five of a kind category, for example, we simply find the round that has the smallest sum of all dice. In a similar way we should find rounds to fit into other three categories: short straight, long straight, and full house. The rest of the categories should be searched exhaustively.

As usual, let's sort out input/output first. Let's assume there's a constructor for our `Yahtzee` class that takes as input a list of lists of integers. These interes are going to be our 13 rounds as defined in the task's description. `getSolutionString` method returns the answer in the format required by the task. Note that dice will be sorted in ascending order. This will be useful later on.

42 ⟨ *Yahtzee* 42⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.BitSet;
import java.util.HashSet;
import java.util.List;
import java.util.Set;
import java.util.stream.Collectors;

public class Yahtzee {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    ⟨2.8 Constants 43a⟩

    ⟨2.8 Helpers 43c⟩

    ⟨2.8 Constructor 47a⟩

    ⟨2.8 Methods 47b⟩

    public static void main(String[] args) throws IOException {
        String currentLine = null;
        List<List<Integer>> input = new ArrayList<>();
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            List<Integer> inputLine = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals(""))
                    .map(Integer::parseInt).sorted()
                    .collect(toList());
            input.add(inputLine);
            if (input.size() == 13) {
                Yahtzee yahtzee = new Yahtzee(input);
                System.out.println(yahtzee.getSolutionString());
```

```
                input.clear();
            }
        }
    }
}
```

Since there are going to be 13 categories, it's convenient to reference them by indexes in an array of 13 elements. Let's define some constants:

43a ⟨*2.8 Constants* 43a⟩≡

```
private final static int fullhouse = 12;
private final static int longstraight = 11;
private final static int shortstraight = 10;
private final static int fiveofakind = 9;
private final static int fourofakind = 8;
private final static int threeofakind = 7;
private final static int chance = 6;
```

And let's define arrays that will hold the best solution and the sum and a bonus of that solution:

43b ⟨*2.8 Constants* 43a⟩+≡

```
private final int[] bestSolutionResult = new int[2];
private final int[] bestSolution = new int[13];
```

Before doing any computations we need to categorize our input data. So let's determine up front whether a given round belongs to a specific category or not. To represent a round we are going to define a class `Round` that will hold information about which categories this round can be used for and also the sum of dice (or points) if this round is chosen to be used in a specific category.

So an instance of this class will have the dice values, the sum of these dice, points depending on categories, and a set of categories this round belongs to.

43c ⟨*2.8 Helpers* 43c⟩≡

```
public static class Round {
    private final List<Integer> dice;
    private final int allDiceSum;
    private final int[] points = new int[13];
    private final Set<Integer> category;

    ⟨2.8 Round Constructor 44a⟩

    ⟨2.8 Round Methods 44b⟩
}
```

The constructor assigns `allDiceSum` and `dice` and determines which categories this round belongs to by filling out `points` and `category`.

44a ⟨*2.8 Round Constructor* 44a⟩≡
```
public Round(List<Integer> dice) {
    this.allDiceSum = dice.stream().reduce(0, Integer::sum);
    this.dice = dice;
    this.category = new HashSet<>();

    ⟨2.8 Categorize 46a⟩
}
```

To determine which categories this specific round belongs to we will need to write some helper methods.

Let's start with the full house. It's pretty self-explanatory:

44b ⟨*2.8 Round Methods* 44b⟩≡
```
private boolean isFullhouse() {
    boolean halfsDifferent = dice.get(0) != dice.get(4);
    boolean twoThree = dice.subList(0, 2).stream().distinct()
            .count() == 1 &&
            dice.subList(2, 5).stream().distinct().count() == 1;
    boolean threeTwo = dice.subList(0, 3).stream().distinct()
            .count() == 1 &&
            dice.subList(3, 5).stream().distinct().count() == 1;
    return halfsDifferent && (twoThree || threeTwo);
}
```

For the long straight and short straights we are going to need to determine the longest sequence, so let's have a helper for that:

44c ⟨*2.8 Round Methods* 44b⟩+≡
```
private int getLongestSequence(List<Integer> list) {
    int longest = 1;
    int currLen = 1;
    for (int i = 0; i < list.size(); ++i) {
        if (i > 0 && list.get(i) - list.get(i - 1) == 1) {
            currLen += 1;
        } else if (i > 0 && list.get(i) == list.get(i - 1)) {
            continue;
        } else {
            longest = Math.max(currLen, longest);
            currLen = 1;
        }
    }
    return Math.max(currLen, longest);
}
```

Now we can write our long and short straights:

45a ⟨*2.8 Round Methods* 44b⟩+≡

```
private boolean isLongStraight() {
    return getLongestSequence(dice) >= 5;
}

private boolean isShortStraight() {
    return getLongestSequence(dice) >= 4;
}
```

Five, four and three of a kind are simple too:

45b ⟨*2.8 Round Methods* 44b⟩+≡

```
private boolean isFiveOfAKind() {
    return (dice.stream().distinct().count() == 1);
}

private boolean isFourOfAKind() {
    List<Integer> v1 = dice.subList(0, 4);
    List<Integer> v2 = dice.subList(1, 5);
    return (v1.stream().distinct().count() == 1 ||
            v2.stream().distinct().count() == 1);
}

private boolean isThreeOfAKind() {
    List<Integer> v1 = dice.subList(0, 3);
    List<Integer> v2 = dice.subList(1, 4);
    List<Integer> v3 = dice.subList(2, 5);
    return (v1.stream().distinct().count() == 1 ||
            v2.stream().distinct().count() == 1 ||
            v3.stream().distinct().count() == 1);
}
```

OK, now we can assign some points depending on whether this round belong to a category or not:

46a ⟨*2.8 Categorize* 46a⟩≡
```
if (isFullhouse()) {
    category.add(fullhouse);
    points[fullhouse] = 40;
}
if (isLongStraight()) {
    category.add(longstraight);
    points[longstraight] = 35;
}
if (isShortStraight()) {
    category.add(shortstraight);
    points[shortstraight] = 25;
}
if (isFiveOfAKind()) {
    category.add(fiveofakind);
    points[fiveofakind] = 50;
}
if (isFourOfAKind()) {
    category.add(fourofakind);
    points[fourofakind] = allDiceSum;
}
if (isThreeOfAKind()) {
    category.add(threeofakind);
    points[threeofakind] = allDiceSum;
}
```

Every round can be used in the chance category:

46b ⟨*2.8 Categorize* 46a⟩+≡
```
category.add(chance);
points[chance] = allDiceSum;
```

First six categories can be determined by simple check if the dice have a specific value (1 to 6) or not. Points are assinged accordingly.

46c ⟨*2.8 Categorize* 46a⟩+≡
```
for (int i = 0; i < 6; ++i) {
    final int v = i + 1;
    if (dice.contains(Integer.valueOf(v))) {
        category.add(i);
        points[i] = (int) (dice.stream().filter(x -> x == v)
                .count() * v);
    }
}
```

An important method that we should implement too is `equals`, let's do that:

46d ⟨*2.8 Round Methods* 44b⟩+≡
```
@Override
public boolean equals(Object obj) {
    return dice.equals(((Round) obj).dice);
}
```

That's it for the `Round` class.

Now let's implement `Yahtzee` constructor. Let's assume it calls `solve` method with a list of `Round`s:

47a ⟨*2.8 Constructor* 47a⟩≡
```
Yahtzee(List<List<Integer>> input) {
    solve(input.stream().map(x -> new Round(x)).collect(toList()));
}
```

Now let's write `solve`. First we create an array `candidateSolution` that will hold points for the categories and we try to fit in the last four categories by finding rounds with the smallest sums of the dice:

47b ⟨*2.8 Methods* 47b⟩≡
```
private void solve(List<Round> input) {
    int[] candidateSolution = new int[13];
    for (int category = 12; category > 8; --category) {
        Round dice = filter(category, input).stream()
                .min((x, y) -> Integer.compare(x.allDiceSum, y.allDiceSum))
                .orElse(null);
        if (dice != null) {
            input.remove(dice);
            candidateSolution[category] = dice.points[category];
        }
    }
    search(8, input, candidateSolution);
}
```

The `filter` method is quite straightforward:

47c ⟨*2.8 Methods* 47b⟩+≡
```
private static List<Round> filter(final int category, List<Round> input) {
    List<Round> res = new ArrayList<>();
    Integer categoryInteger = Integer.valueOf(category);
    for (Round d : input) {
        if (d.category.contains(categoryInteger)) {
            res.add(d);
        }
    }
    return res;
}
```

Let's add one more method that we will need, the method that calculates the sum of all points. It'll return an array where the first element is the bonus (if present) and the second element is the total sum (including the bonus):

48 ⟨*2.8 Methods* 47b⟩+≡

```
private static int[] total(int[] solution) {
    int[] res = new int[2];
    int sixSum = 0;
    for (int i = 0; i < solution.length; ++i) {
        if (i < 6) {
            sixSum += solution[i];
        }
        res[1] += solution[i];
    }
    if (sixSum >= 63) {
        res[1] += 35;
        res[0] = 35;
    }
    return res;
}
```

OK, now let's get to the `search` method. This method is going to be a classic backtracking method.

The first parameter is the position in the array of categories that we are trying. We will work out our way in a methodic way down to the first category. Once we reach that we check what result this categorization gives us, and if it's better than the one we've found so far, we update our found solution to the better one. The second argument is a candidate solution.

So first thing we do in this method is to check if `pos` is -1, which means we have a candidate categorization in the `solution`, and we check if it's any better than the one we've found so far. Otherwise we get all the candidate rounds for the given category and start trying them one by one while recursively calling the `search` method.

49 ⟨*2.8 Methods* 47b⟩+≡

```
private void search(int pos, List<Round> input, int[] solution) {
    if (pos == -1) {
        int[] solutionResult = total(solution);
        if (bestSolutionResult[1] < solutionResult[1]) {
            System.arraycopy(solution, 0, bestSolution, 0,
                    solution.length);
            System.arraycopy(solutionResult, 0, bestSolutionResult, 0,
                    solutionResult.length);
        }
        return;
    }

    List<Round> candidates = filter(pos, input);
    Set<Round> checked = new HashSet<Round>();
    for (Round round : candidates) {
        if (checked.contains(round)) {
            continue;
        }
        solution[pos] = round.points[pos];
        input.remove(round);
        search(pos - 1, input, solution);
        solution[pos] = 0;
        input.add(round);
        checked.add(round);
    }
    if (pos >= 7 || candidates.size() == 0) {
        solution[pos] = 0;
        search(pos - 1, input, solution);
    }
}
```

Finally, all we need to do now is to output the result:

50 ⟨*2.8 Methods* 47b⟩+≡

```
private String getSolutionString() {
    return Arrays.stream(getSolution()).mapToObj(String::valueOf)
            .collect(Collectors.joining(" "));
}

public int[] getSolution() {
    int[] solution = new int[bestSolution.length +
            bestSolutionResult.length];
    System.arraycopy(bestSolution, 0, solution, 0, bestSolution.length);
    System.arraycopy(bestSolutionResult, 0, solution,
            bestSolution.length,
            bestSolutionResult.length);
    return solution;
}
```

This concludes this program.

3 Strings

3.1 WERTYU

Trivial.

51 ⟨*WERTYU* 51⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class WERTYU {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private final static String KEYS = "‘1234567890-=QWERTYUIOP[]\\ASDFGHJKL;’ZXCVBNM,./";
    private final static int[] map = new int[256];

    static {
        for (int i = 0; i < KEYS.length(); ++i) {
            map[KEYS.charAt(i)] = i;
        }
    }

    private static String shift(String currentLine) {
        StringBuilder output = new StringBuilder();
        for (int i = 0; i < currentLine.length(); ++i) {
            output.append(map[currentLine.charAt(i)] != 0
                    ? KEYS.charAt(map[currentLine.charAt(i)] - 1)
                    : currentLine.charAt(i));
        }
        return output.toString();
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            System.out.println(shift(currentLine));
        }
    }

}
```

3.2 Where's Waldorf

Trivial.

52 ⟨*Where is Waldorf* 52⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class WheresWaldorf {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private final int n;
    private final int m;
    private final char[][] table;

    public WheresWaldorf(int n, int m, char[][] table) {
        this.n = n;
        this.m = m;
        this.table = table;
    }

    private static final int[][] dir = new int[][] {
            { 1, 0 }, { -1, 0 }, { 0, 1 }, { 0, -1 }, { 1, 1 }, { -1, -1 },
            { -1, 1 }, { 1, -1 }
    };

    public int[] find(String word) {
        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < m; ++j) {
                for (int k = 0; k < dir.length; ++k) {
                    if (check(dir[k][0], dir[k][1], i, j, word)) {
                        return new int[] { i + 1, j + 1 };
                    }
                }
            }
        }
        return null;
    }

    private boolean check(int di, int dj, int i, int j, String word) {
        if (word.length() == 1) {
            return table[i][j] == word.charAt(0);
        }
        int pos = 0;
        while (i >= 0 && i < n && j >= 0 && j < m && pos < word.length() &&
```

```
                table[i][j] == word.charAt(pos)) {
            j += dj;
            i += di;
            pos++;
        }
        return pos == word.length();
    }

    private static String toString(int[] arr) {
        return Arrays.stream(arr).mapToObj(String::valueOf)
                .collect(Collectors.joining(" "));
    }

    public static void main(String[] args) throws IOException {
        int cases = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        for (int k = 0; k < cases; ++k) {
            List<Integer> nm = stream(reader.readLine().trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            char[][] table = new char[nm.get(0)][nm.get(1)];
            for (int i = 0; i < nm.get(0); ++i) {
                String currentLine = reader.readLine();
                for (int j = 0; j < nm.get(1); ++j) {
                    table[i][j] = currentLine.toLowerCase().charAt(j);
                }
            }
            int wordsCount = Integer.parseInt(reader.readLine().trim());
            List<String> words = new ArrayList<>();
            for (int i = 0; i < wordsCount; ++i) {
                String currentLine = reader.readLine().toLowerCase();
                words.add(currentLine);
            }
            WheresWaldorf ww = new WheresWaldorf(nm.get(0), nm.get(1), table);
            words.forEach(x -> System.out.println(toString(ww.find(x))));
            if (k < cases - 1) {
                System.out.println();
                reader.readLine();
            }
        }
    }
}
```

3.3 Common Permutation

This task is quite simple. First, we sort the input line characters in ascending order. Then, for each character in the input lines we compute its run. A run is a string of consecutive characters that are the same, for example "aaa" is a run of length 3. Next, we take an intersection of distinct characters between both lines, and start compiling the longest string by simple checking character by character while taking into account their runs: We just take the smallest run out of two on each step.

54 ⟨*Common Permutation* 54⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

public class CommonPermutation {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static int[] runs(int[] arr) {
        int[] runs = new int[256];
        for (int i = 0; i < arr.length; ++i) {
            runs[arr[i]] += 1;
        }
        return runs;
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            int[] line1 = currentLine.trim().chars().sorted().toArray();
            int[] line2 = reader.readLine().chars().sorted().toArray();
            int[] run1 = runs(line1);
            int[] run2 = runs(line2);
            List<Integer> distinct = stream(line1).distinct().boxed()
                    .collect(toList());
            distinct.retainAll(stream(line2).distinct().boxed()
                    .collect(toList()));

            StringBuilder longest = new StringBuilder();
            for (int i = 0; i < distinct.size(); ++i) {
                int len = Math.min(run1[distinct.get(i)],
                        run2[distinct.get(i)]);
                for (int j = 0; j < len; ++j) {
                    longest.append((char) distinct.get(i).intValue());
                }
            }
            System.out.println(longest);
        }
    }
```

```
}
```

3.4 Crypt Kicker II

This task is much easier than Crypt Kicker. Here we have a very well known pangram "the quick brown fox jumps over the lazy dog." A pangram is a sentence that uses every letter of the alphabet at least once. So all we need to do is to locate the pangram in the input lines. We will use exactly the same technique as we used while solving the original Crypt Kicker problem.

56 ⟨*Crypt Kicker II* 56⟩≡

```
import static java.lang.Math.abs;
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;

class CryptKickerII {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    private final int[] mapping = new int[128];
    private static final String pangram = "the quick brown fox jumps over the lazy dog";
    private static final String pangramSpaces = pangram.replaceAll("[^ ]", ".");
    private static final List<Integer> pangramPattern = getPattern(pangram);

    private static List<Integer> getPattern(String word) {
        return range(0, word.length()).map(i -> word.indexOf(word.charAt(i)))
                .boxed().collect(toList());
    }

    private static boolean compare(List<Integer> a, List<Integer> b) {
        return a.size() == b.size() && range(0, a.size())
                .map(i -> abs(a.get(i) - b.get(i))).sum() == 0;
    }

    private boolean isPangram(String input) {
        String line = String.join(" ", Arrays.stream(input.trim()
                .split(" ")).filter(x -> !x.equals("")).collect(toList()));
        return compare(pangramPattern, getPattern(line.toString())) &&
                line.replaceAll("[^ ]", ".").equalsIgnoreCase(pangramSpaces);
    }

    public List<String> decrypt(List<String> input) {
        Arrays.fill(mapping, 0);
        List<String> output = new ArrayList<String>();
        String encryptedPangram = input.stream()
                .filter(x -> isPangram(x)).findFirst().orElse("");
        if (encryptedPangram.equalsIgnoreCase("")) {
            output.add("No solution.");
            return output;
```

```
        }

        for (int i = 0; i < encryptedPangram.length(); ++i) {
            mapping[encryptedPangram.charAt(i)] = pangram.charAt(i);
        }

        return input.stream().map(x -> {
            StringBuilder result = new StringBuilder();
            x.chars().map(c -> c != ' ' ? mapping[c] : c)
                    .forEachOrdered(c -> result.append((char) c));
            return result.toString();
        }).collect(toList());
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        final int n = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        CryptKickerII cryptKicker = new CryptKickerII();
        for (int i = 0; i < n; ++i) {
            List<String> input = new ArrayList<String>();
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().equalsIgnoreCase("")) {
                input.add(currentLine);
            }
            cryptKicker.decrypt(input).forEach(System.out::println);
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
}
```

3.5 Automated Judge Script

This task looks so trivial, but it took me a few attempts before the online judge accepted it. The reason was that it was wrong to use the `readLine()` method, because it strips characters; the contract says that "line is considered to be terminated by any one of a line feed ('\n'), a carriage return ('\r'), or a carriage return followed immediately by a line feed." So the input, when read by `readLine()`, would apparently miss some characters. Other than this caveat, the task is trivial.

58 ⟨*Automated Judge Script* 58⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;

public class AutomatedJudgeScript {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static String read(int n) throws IOException {
        StringBuilder input = new StringBuilder();
        int newlines = 0;
        while (newlines < n) {
            int c = reader.read();
            if (c == '\n') {
                newlines++;
            }
            input.append((char) c);
        }
        return input.toString();
    }

    public static void main(String[] args) throws IOException {
        int i = 0;
        while (true) {
            int n = Integer.parseInt(reader.readLine().trim());
            if (n == 0) {
                break;
            }
            String src = read(n);
            String dst = read(Integer.parseInt(reader.readLine().trim()));

            System.out.print("Run #" + (++i) + ": ");
            if (src.equals(dst)) {
                System.out.println("Accepted");
            } else {
                if (Arrays.equals(
                        src.chars().filter(Character::isDigit).toArray(),
                        dst.chars().filter(Character::isDigit).toArray())) {
                    System.out.println("Presentation Error");
                } else {
                    System.out.println("Wrong Answer");
                }
```

```
            }
        }
    }
}
```

3.6 File Fragmentation

Let's sort out input/output assuming that our function `restore` takes a list of strings (i.e. shards) and returns the restored string (i.e. original file). Input is rather straightforward and, unfortunately, due to the format of the input data, isn't very concise.

59 ⟨*File Fragmentation* 59⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
⟨3.6 Imports 60a⟩

class FileFragmentation {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    ⟨3.6 Helpers 60e⟩

    private static String restore(List<String> fragments) {
        ⟨3.6 Implementation 60b⟩
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine());
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            List<String> fragments = new ArrayList<String>();
            do {
                String s = reader.readLine();
                if (s == null || s.equalsIgnoreCase("")) {
                    break;
                }
                fragments.add(s);
            } while (true);
            System.out.println(restore(fragments));
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
}
```

So how do we restore the files? It's easy to see that if we sort the shards by length and then take the largest shard and the shortest one we will end up with a potential original file. But there may be numerous smallest shards and numerous largest shards, so we will need to try them one by one. This is not that bad as it seems at first sight. This is because we only need to try one largest shard with n shortest shards in the worst case, having only two cases: The long shard goes first and the short goes after it or vice versa. Once we got a candidate original file we simply try to fit the rest of the shards. This can be done very easily. We simply partition our candidate file at every point and then check if the list contains these shards, and if it does, we mark that. Once we found every shard in the list in this way we know that the original file was the same as our candidate file. Otherwise we try the next smallest shard. We continue until we fit every shard. This algorithm will always find the original file because of how the problem is formulated.

OK, so first thing we need to do is to sort the shards by length:

60a ⟨*3.6 Imports* 60a⟩≡
```
import static java.util.Comparator.comparing;
```

60b ⟨*3.6 Implementation* 60b⟩≡
```
fragments.sort(comparing(String::length));
```

Then we find the largest (any will do) and get the list of the smallest shards:

60c ⟨*3.6 Imports* 60a⟩+≡
```
import static java.util.stream.Collectors.toList;
```

60d ⟨*3.6 Implementation* 60b⟩+≡
```
String large = fragments.get(fragments.size() - 1);
List<String> smallest = fragments.stream().filter(
    x -> x.length() == fragments.get(0).length()).collect(toList());
```

Let's write `fit` function that takes a list of shards and a candidate and returns true or false depending on whether those shards could be fit with this candidate file or not. This is implemented in accordance to the algorithm described earlier.

60e ⟨*3.6 Helpers* 60e⟩≡
```
private static boolean fit(List<String> fragments, String candidate) {
    List<String> temp = new ArrayList<String>(fragments);
    for (int i = 1; i < candidate.length() && !temp.isEmpty(); ++i) {
        final int j = i;
        temp.removeIf(x -> x.equalsIgnoreCase(candidate.substring(0, j)));
        temp.removeIf(x -> x.equalsIgnoreCase(candidate.substring(j)));
    }
    return temp.isEmpty();
}
```

For the largest and every smallest shard we try to fit the rest of the shards using `fit` function trying both cases: large + small, and small + large.

60f ⟨*3.6 Implementation* 60b⟩+≡
```
for (String small : smallest) {
    if (fit(fragments, large + small)) {
        return large + small;
    } else if (fit(fragments, small + large)) {
        return small + large;
    }
}
return "Impossible";
```

In accordance to the problem statement "Impossible" should never be returned, unless the input is malformed for any reason.

3.7 Doublets

It's a simple task of the shortest path search in a graph. But because this graph is not directed and each edge has length 1, we can use deaph-first search algorithm to find the shortest path.

Very straightforward.

61 ⟨*Doublets* 61⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;
import java.util.stream.Collectors;

public class Doublets {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static List<String> find(Map<String, Set<String>> graph,
            String from,
            String to) {

        if (from.equalsIgnoreCase(to)) {
            List<String> output = new ArrayList<>();
            output.add(from);
            output.add(to);
            return output;
        }

        Deque<String> q = new ArrayDeque<>();
        Set<String> s = new HashSet<String>();
        Map<String, String> r = new HashMap<>();
        q.push(from);
        r.put(from, from);

        while (!q.isEmpty()) {
            String currWord = q.pop();
            for (String adjacentWord : graph.get(currWord)) {
```

```
            if (!s.contains(adjacentWord)) {
                s.add(adjacentWord);
                q.addLast(adjacentWord);
                if (!r.containsKey(adjacentWord)) {
                    r.put(adjacentWord, currWord);
                }
            }
        }
        if (r.containsKey(to)) {
            List<String> output = new ArrayList<>();
            String curr = to;
            while (!curr.equalsIgnoreCase(from)) {
                output.add(0, curr);
                curr = r.get(curr);
            }
            output.add(0, from);
            return output;
        }
    }

    return null;
}

private static boolean adjacent(String a, String b) {
    if (a.length() != b.length()) {
        return false;
    }
    int diffCount = 0;
    for (int i = 0; i < a.length(); ++i) {
        if (a.charAt(i) != b.charAt(i)) {
            diffCount++;
        }
    }
    return diffCount == 1;
}

private static Map<String, Set<String>> getGraph(List<String> dict) {
    Map<String, Set<String>> graph = new HashMap<>();
    dict = dict.stream().distinct()
            .sorted(Comparator.comparing(String::length)).collect(toList());
    Map<Integer, List<String>> grouped = dict.stream()
            .collect(Collectors.groupingBy(String::length));
    for (String word : dict) {
        if (!graph.containsKey(word)) {
            graph.put(word, new HashSet<>());
        }
        List<String> adjacent = grouped.get(word.length()).stream()
                .filter(x -> adjacent(word, x)).collect(toList());
        graph.get(word).addAll(adjacent);
    }
    return graph;
}
```

```
    public static void main(String[] args) throws IOException {
        String currentLine;
        List<String> dict = new ArrayList<>();
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equalsIgnoreCase("")) {
            dict.add(currentLine.trim());
        }
        Map<String, Set<String>> graph = getGraph(dict);
        int line = 0;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equalsIgnoreCase("")) {
            if (line > 0) {
                System.out.println();
            }
            List<String> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals(""))
                    .collect(toList());
            List<String> result = find(graph, input.get(0), input.get(1));
            if (result == null) {
                System.out.println("No solution.");
            } else {
                result.forEach(System.out::println);
            }
            line++;
        }
    }
}
```

3.8 Fmt

This task may look simple to do at first sight, but actually it's quite involved. The task becomes much easier if the whole text is read before trying to format it. We will read the whole text and break it into three type of tokens: words, new line breaks and spaces. Continuous spaces would be represented as one token in this list of tokens. Then, we read this list of tokens and simply follow the rules of formatting outlines in the problem statement.

64 ⟨*Fmt* 64⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class Fmt {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final int LINE_WIDTH = 72;

    private static List<String> tokenize(String input) {
        List<String> output = new ArrayList<>();
        StringBuilder block = new StringBuilder();
        int pos = 0;
        while (pos < input.length()) {
            if (input.charAt(pos) == ' ') {
                while (pos < input.length() && input.charAt(pos) == ' ') {
                    block.append(input.charAt(pos));
                    pos++;
                }
                output.add(block.toString());
                block = new StringBuilder();
            } else if (input.charAt(pos) == '\n') {
                output.add(new String("\n"));
                pos++;
            } else {
                while (pos < input.length() && input.charAt(pos) != ' ' &&
                        input.charAt(pos) != '\n') {
                    block.append(input.charAt(pos));
                    pos++;
                }
                output.add(block.toString());
                block = new StringBuilder();
            }
        }
        if (block.length() > 0) {
            output.add(block.toString());
        }
        return output;
    }

    private static void flush(List<String> line, StringBuilder output,
            boolean newline) {
```

```
        if (line.size() > 1 && line.get(line.size() - 1).startsWith(" ")) {
            line.remove(line.size() - 1);
        }

        StringBuilder lineStr = new StringBuilder();
        for (String x : line) {
            lineStr.append(x);
        }

        output.append(lineStr).append(newline ? "\n" : "");
        line.clear();
    }

    private static String format(List<String> tokens) {
        StringBuilder output = new StringBuilder();
        List<String> line = new ArrayList<>();

        int i = 0;
        while (i < tokens.size()) {
            int currLength = line.stream().map(String::length)
                    .reduce(0, Integer::sum).intValue();
            String token = tokens.get(i);

            if (token.startsWith(" ")) {
                line.add(token);
            } else if (token.equals("\n")) {
                if (i + 1 < tokens.size()) {
                    String next = tokens.get(i + 1);
                    if (next.equals("\n") || next.startsWith(" ") ||
                            line.size() == 0 || (line.size() == 1 &&
                                    line.get(0).startsWith(" "))) {
                        flush(line, output, true);
                    } else if (currLength + next.length() <= LINE_WIDTH) {
                        line.add(" ");
                        line.add(next);
                        ++i;
                    } else {
                        flush(line, output, true);
                    }
                } else {
                    line.add(token);
                }
            } else if (currLength == 0 && token.length() > LINE_WIDTH) {
                line.add(token);
                if (i + 1 < tokens.size()) {
                    flush(line, output, true);
                    i++;
                }
            } else {
                if (currLength + token.length() > LINE_WIDTH) {
                    flush(line, output, true);
                }
                line.add(token);
```

```
            }
            ++i;
        }

        if (line.size() > 0) {
            flush(line, output, false);
        }

        return output.toString();
    }

    public static void main(String[] args) throws IOException {
        StringBuilder input = new StringBuilder();
        while (true) {
            int c = reader.read();
            if (c == -1) {
                break;
            }
            input.append((char) c);
        }
        System.out.print(format(tokenize(input.toString())));
    }
}
```

4 Sorting

4.1 Vito's Family

All we need to do is to find the median and then sum the distances.

67 ⟨*Vitos Family* 67⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Collections;
import java.util.List;

public class VitosFamily {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int solve(List<Integer> input) {
        Collections.sort(input);
        int median = 0;

        if (input.size() % 2 == 0) {
            int p = input.size() / 2 - 1;
            median = (input.get(p) + input.get(p + 1)) / 2;
        } else {
            int p = input.size() / 2;
            median = (input.get(p));
        }

        int sum = 0;
        for (Integer v : input) {
            sum += Math.abs(v - median);
        }
        return sum;
    }

    public static void main(String[] args) throws IOException {
        int cases = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < cases; ++i) {
            List<Integer> input = stream(reader.readLine().trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            System.out.println(solve(input.subList(1, input.size())));
        }
    }

}
```

4.2 Stacks of Flapjacks

To solve this task all we have to do is the following: Find the next largest value in the array that is not already in its correct position and flip it so that it appears at the top of the stack, then do another flip so that it appears next to the previous largest value; continue until the array is sorted.

68 ⟨*Stacks of Flapjacks* 68⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

public class StacksOfFlapjacks {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int max(List<Integer> input, int skip) {
        int index = -1;
        int max = Integer.MIN_VALUE;
        for (int i = skip; i < input.size(); ++i) {
            if (max < input.get(i)) {
                index = i;
                max = input.get(i);
            }
        }
        return index;
    }

    private static List<Integer> solve(List<Integer> input) {
        List<Integer> inputCopy = new ArrayList<>(input);
        Collections.reverse(inputCopy);
        List<Integer> flips = new ArrayList<>();
        int sorted = 0;

        while (sorted < inputCopy.size()) {
            int index = max(inputCopy, sorted);
            if (index != sorted) {
                flips.add(index + 1);
                flips.add(sorted + 1);
                Collections.reverse(inputCopy.subList(index, inputCopy.size()));
                Collections
                        .reverse(inputCopy.subList(sorted, inputCopy.size()));
            }
            sorted++;
        }
        flips.add(0);
        return flips;
```

```
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            System.out.println(currentLine);
            System.out.println(solve(input).stream().map(x -> x.toString())
                    .collect(Collectors.joining(" ")));
        }
    }
}
```

4.3 Bridge

This task is quite tricky. But before trying to solve it, let's just sort out input/output to get it out of the way.

We will assume that the we have a method `getStrategy` that takes a list of integers (crossing times) and returns two lists. The first list holds crossing times going from left to right, and the second list holds crossing times from right to left. (We assume the group of people starts on the left side of the bridge.) Let's assume there's a `printResult` method that takes that output of `getStrategy` and prints it out in the format specified in the problem statement.

70 ⟨*Bridge* 70⟩≡

```
import static java.util.stream.Collectors.toList;
import static java.util.stream.IntStream.range;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.List;
import java.util.PriorityQueue;
import java.util.function.BiConsumer;
import java.util.stream.Stream;


class Bridge {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    private static final int LEFT_RIGHT = 0;
    private static final int RIGHT_LEFT = 1;

    private static void printResult(final List<List<Integer>> result) {
        ⟨4.3 Print Result 71a⟩
    }

    private static List<List<Integer>> getStrategy(List<Integer> input) {
        ⟨4.3 Get Strategy 71d⟩
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.valueOf(reader.readLine().trim());
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            int count = Integer.valueOf(reader.readLine().trim());
            List<Integer> input = reader.lines().map(String::trim)
                .limit(count).map(Integer::parseInt).collect(toList());
            printResult(getStrategy(input));
            if (i < n - 1) {
                reader.readLine();
                System.out.println();
            }
        }
```

```
    }
}
```

Let's implement the `printResult` method. Like we said, the `result` list contains two lists of integers, one list denoting crossing times from left to right, and the other from right to left. The first list will always contain pairs, as people are crossing from left to right (as we agreed).

The simplest case of all is when there's just one person. In that case we simply print the total time, which will equal to the crossing time of this person, and then the same number again, denoting that person crossing the bridge.

71a ⟨*4.3 Print Result* 71a⟩≡

```
List<Integer> lr = result.get(LEFT_RIGHT);
List<Integer> rl = result.get(RIGHT_LEFT);
if (lr.size() == 1) {
    System.out.println(lr.get(0));
    System.out.println(lr.get(0));
    return;
}
```

Otherwise, we need to sum the crossing times. For the list that holds crossing times from right to left is easy, we just sum those numbers. For the list that holds crossing times from left to right we need to sum the second number in each pair (i.e. the slowest person).

71b ⟨*4.3 Print Result* 71a⟩+≡

```
int totalTime = range(0, lr.size()).filter(x -> (x + 1) % 2 == 0)
        .map(x -> lr.get(x)).sum() +
        rl.stream().mapToInt(Integer::intValue).sum();
```

Finally, we just output the `totalTime` and print the strategy.

71c ⟨*4.3 Print Result* 71a⟩+≡

```
System.out.println(totalTime);
Stream.iterate(0, i -> i + 2).limit(lr.size() / 2).forEachOrdered(i -> {
    System.out.println(lr.get(i) + " " + lr.get(i + 1));
    if (i / 2 < rl.size()) {
        System.out.println(rl.get(i / 2));
    }
});
```

OK, now let's figure out the strategy. If there's just one person that's easy:

71d ⟨*4.3 Get Strategy* 71d⟩≡

```
final List<List<Integer>> output = Arrays
                .asList(new ArrayList<Integer>(), new ArrayList<Integer>());

if (input.size() == 1) {
    output.get(LEFT_RIGHT).add(input.get(0));
    return output;
}
```

Obviously the time of crossing the bridge equals to the slowest in a pair. Let's assume we have four people and their crossing speeds are $x_1 <= x_2 <= x_3 <= x_4$. One way to transfer them is this: x_1 and x_2 cross, x_1 returns, then x_3 and x_4 cross, and x_2 returns, finally x_1 and x_2 cross. This amounts to total time $x_1 + 3x_2 + x_4$. Another way to transfer is x_1 and x_2 cross, x_1 returns, then x_1 and x_3 cross, and x_1 returns, finally x_1 and x_4 cross. This amounts to total time $2x_1 + x_2 + x_3 + x_4$. This essentially solves the task, because we simply choose the strategy that leads to the smallest time. That is we simply check if $x_1 + 3x_2 + x_4 \leq 2x_1 + x_2 + x_3 + x_4$, or, equivalently, $2x_2 \leq x_1 + x_3$.

These two strategies still work event if there are more than four people. We assign to x_1 the fastest and to x_4 the slowest, to x_2 the second fastest, and to x_3 the second slowest.

We will be dealing with the fastest and the slowest so having priority queues will be convenient, so let's have them:

72a ⟨*4.3 Get Strategy* 71d⟩+≡

```
final PriorityQueue<Integer> left = new PriorityQueue<>();
final PriorityQueue<Integer> right = new PriorityQueue<>();
```

And we are going to move data from left to right quite a lot, so let's have a helper:

72b ⟨*4.3 Get Strategy* 71d⟩+≡

```
final BiConsumer<PriorityQueue<Integer>, PriorityQueue<Integer>> move = (
        from, to) -> {
    if (!from.isEmpty()) {
        Integer v = from.remove();
        to.add(v);
        output.get(from == left ? LEFT_RIGHT : RIGHT_LEFT).add(v);
    }
};
```

Note that this helper also puts the corresponding values to the `output`.

Now let's implement the main loop. Note that whenever returning from right to left, always the fastest from the group on the right should go. Who goes from left to right will depend on the inequality that we discussed above.

73 ⟨*4.3 Get Strategy* 71d⟩+≡

```
left.addAll(input);

while (!left.isEmpty()) {
    move.accept(right, left);
    move.andThen(move).accept(left, right);
    if (left.isEmpty()) {
        break;
    }

    move.accept(right, left);
    if (left.size() == 2) {
        move.andThen(move).accept(left, right);
        break;
    }

    Integer x1 = left.remove();
    Integer x2 = right.peek();
    Integer x4 = left.stream().max(Integer::compareTo).get();
    left.remove(x4);
    Integer x3 = left.stream().max(Integer::compareTo).get();
    left.remove(x3);
    int[] x = (2 * x2 <= x1 + x3) ? new int[] { x1, x3, x4 }
            : new int[] { x4, x1, x3 };

    left.add(x[0]);
    output.get(LEFT_RIGHT).add(x[1]);
    output.get(LEFT_RIGHT).add(x[2]);
    right.add(x[1]);
    right.add(x[2]);
}

return output;
```

This concludes the program.

4.4 Longest Nap

Here we simply need to read time intervals, then combine the overlapping or connecting intervals into the large ones, finally find the longest gap between these combined intervals.

The program is self-explanatory.

74 ⟨*Longest Nap* 74⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.time.LocalTime;
import java.time.format.DateTimeFormatter;
import java.util.ArrayList;
import java.util.List;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

public class LongestNap {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final int START = 0;
    private static final int END = 1;

    private final static Pattern pattern = Pattern
            .compile("(\\d\\d:\\d\\d)\\s+(\\d\\d:\\d\\d)");

    private final static DateTimeFormatter formatter = DateTimeFormatter
            .ofPattern("HH:mm");

    private static final LocalTime[] parseTime(String line) {
        Matcher matcher = pattern.matcher(line);
        matcher.find();
        return new LocalTime[] {
                LocalTime.parse(matcher.group(1), formatter),
                LocalTime.parse(matcher.group(2), formatter)
        };
    }

    private static final List<LocalTime[]> combine(
            List<LocalTime[]> intervals) {
        List<LocalTime[]> intr = new ArrayList<>(intervals);
        intr.sort((x, y) -> x[START].compareTo(y[START]));
        List<LocalTime[]> res = new ArrayList<>();

        while (!intr.isEmpty()) {
            LocalTime[] curr = intr.remove(0);
            while (!intr.isEmpty()) {
                if (intr.get(0)[START].isBefore(curr[END]) ||
                        intr.get(0)[START].equals(curr[END])) {
                    LocalTime[] next = intr.remove(0);
                    if (curr[END].isAfter(next[START]) ||
```

```
                        curr[END].equals(next[START])) {
                    if (curr[END].isBefore(next[END])) {
                        curr[END] = next[END];
                    }
                }
            } else {
                break;
            }
        }
        res.add(curr);
    }

    return res;
}

private static LocalTime[] findLongest(List<LocalTime[]> intervals) {
    if (intervals.size() == 0) {
        return new LocalTime[] { LocalTime.of(10, 0),
                LocalTime.of(8, 0) };
    }

    LocalTime earliest = intervals.get(0)[START];
    LocalTime latest = intervals.get(intervals.size() - 1)[END];

    long i1 = earliest.toSecondOfDay() -
            LocalTime.of(10, 0).toSecondOfDay();
    long i2 = LocalTime.of(18, 0).toSecondOfDay() - latest.toSecondOfDay();

    LocalTime[] result = (i1 < i2)
            ? new LocalTime[] { latest, LocalTime.ofSecondOfDay(i2) }
            : new LocalTime[] { LocalTime.of(10, 0),
                    LocalTime.ofSecondOfDay(i1) };

    for (int i = 0; i < intervals.size() - 1; ++i) {
        long interval = intervals.get(i + 1)[START].toSecondOfDay() -
                intervals.get(i)[END].toSecondOfDay();
        if (interval >= result[1].toSecondOfDay()) {
            boolean same = interval == result[1].toSecondOfDay();
            result[1] = LocalTime.ofSecondOfDay(interval);
            if (same && result[0].isAfter(intervals.get(i)[END]) || !same) {
                result[0] = intervals.get(i)[END];
            }
        }
    }

    return result;
}

public static void main(String[] args) throws IOException {
    String currentLine;
    int d = 1;
    while ((currentLine = reader.readLine()) != null) {
        int appointments = Integer.parseInt(currentLine.trim());
```

```
            List<LocalTime[]> intervals = new ArrayList<>();
            for (int i = 0; i < appointments; ++i) {
                intervals.add(parseTime(reader.readLine()));
            }
            LocalTime[] result = findLongest(combine(intervals));
            System.out.println("Day #" + d + ": the longest nap starts at " +
                    formatter.format(result[START]) + " and will last for " +
                    ((result[1].getHour() > 0)
                            ? result[1].getHour() + " hours and " : "") +
                    result[1].getMinute() + " minutes.");
            d++;
        }
    }
}
```

4.5 Shoemaker's Problem

Let's assume we have four jobs that take d_1, d_2, d_3, d_4 days and have corresponding fines f_1, f_2, f_3, f_4. Let's also assume they are in the optimal order, that is $S = 0 \cdot f_1 + (d_1) \cdot f_2 + (d_1 + d_2) \cdot f_3 + (d_1 + d_2 + d_3) \cdot f_4$ and S value is minimal. Let's now suppose we swap job three and four, so we get another value $\hat{S} = 0 \cdot f_1 + (d_1) \cdot f_2 + (d_1 + d_2) \cdot f_4 + (d_1 + d_2 + d_3) \cdot f_3$, but since S is optimal it means that $S \leq \hat{S}$, or

$$0 \cdot f_1 + (d_1) \cdot f_2 + (d_1 + d_2) \cdot f_4 + (d_1 + d_2 + d_3) \cdot f_3 - (0 \cdot f_1 + (d_1) \cdot f_2 + (d_1 + d_2) \cdot f_3 + (d_1 + d_2 + d_3) \cdot f_4) \geq 0$$

and so

$$f_3 d_4 - f_4 d_3 \geq 0$$

$$f_3 d_4 \geq f_4 d_3$$

This in turn means that in the optimal order d_3 will go before d_4 if $f_3/d_3 \geq f_4/d_4$. So, generally, d_i will go before d_j if $f_i/d_i \geq f_j/d_j$.

77 ⟨*Shoemakers Problem* 77⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

public class ShoemakersProblem {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static class Job implements Comparable<Job> {
        int start;
        int fine;
        int index;

        public Job(int index, int start, int fine) {
            this.index = index;
            this.start = start;
            this.fine = fine;
        }

        @Override
        public int compareTo(Job o) {
            return Integer.compare(o.fine * start, fine * o.start);
        }
    }

    public static void main(String[] args)
            throws NumberFormatException, IOException {
```

```
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            reader.readLine();
            int count = Integer.parseInt(reader.readLine().trim());
            Job[] jobs = new Job[count];
            for (int j = 0; j < count; ++j) {
                List<Integer> input = stream(
                        reader.readLine().trim().split(" "))
                                .filter(x -> !x.equals(""))
                                .map(Integer::parseInt)
                                .collect(toList());
                jobs[j] = new Job(j, input.get(0), input.get(1));
            }
            Arrays.sort(jobs);
            System.out.println(Arrays.stream(jobs).map(x -> x.index + 1)
                    .map(String::valueOf)
                    .collect(Collectors.joining(" ")));
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
}
```

4.6 CDVII

The problem statement is a bit vague. When I first read it I thought I'd need to figure out the speed of each car and then using that figure out how many kilometers each car went in specific hours. But then I looked at the sample input and realized that it only takes into account the rate at the moment of entry. So the task becomes much easier.

What's the difficulty then? Well, there is not much difficulty in this task. We just need to carefully pair enter and exit times and prepare bills based on these pairs.

Let's code input/output first. We are going to have a helper class `Event` that will encapsulate event information: License number, time of entrance or exit, and location. For parsing the input lines we are going to use a regex. Notice that for `getBills` the return type is `Entry<String, AtomicInteger>`. The key of `Entry` will be a license plate, and the value will be the bill in cents. We are using `AtomicInteger` because it has convenient update semantics.

79 ⟨*CDVII* 79⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;
import java.time.LocalDate;
import java.time.LocalDateTime;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.concurrent.atomic.AtomicInteger;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
import java.util.stream.Collectors;

public class CDVII {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static Pattern pattern = Pattern.compile(
            "([\\d\\w]+)\\s+(\\d\\d):(\\d\\d):(\\d\\d):(\\d\\d)\\s+(\\w+)\\s+(\\d+)");

    private static class Event {
        private final String license;
        private final LocalDateTime timestamp;
        private final String action;
        private final int location;

        public LocalDateTime getTimestamp() {
            return timestamp;
        }
```

```
    public String getAction() {
        return action;
    }

    public String getLicense() {
        return license;
    }

    public Event(String line) {
        Matcher matcher = pattern.matcher(line);
        matcher.find();
        license = matcher.group(1);
        timestamp = LocalDateTime.of(LocalDate.now().getYear(),
                Integer.parseInt(matcher.group(2)),
                Integer.parseInt(matcher.group(3)),
                Integer.parseInt(matcher.group(4)),
                Integer.parseInt(matcher.group(5)));
        action = matcher.group(6).toLowerCase();
        location = Integer.parseInt(matcher.group(7));
    }
}

private static List<Entry<String, AtomicInteger>> getBills(
        List<Integer> rate,
        List<Event> events) {
    ⟨4.6 Get Bills 82a⟩
}

private static Event[] findInterval(List<Event> enters, List<Event> exits) {
    ⟨4.6 Find Interval 81⟩
}

public static void main(String[] args) throws IOException {
    int n = Integer.parseInt(reader.readLine());
    reader.readLine();
    for (int i = 0; i < n; ++i) {
        List<Integer> rate = stream(reader.readLine().trim().split(" "))
                .filter(x -> !x.equals("")).map(Integer::parseInt)
                .collect(toList());
        String currentLine;
        List<Event> events = new ArrayList<>();
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            events.add(new Event(currentLine));
        }
        for (Entry<String, AtomicInteger> e : getBills(rate, events)) {
            System.out.println(e.getKey() + " $" +
                    BigDecimal.valueOf(e.getValue().intValue(), 2));
        }
        if (i < n - 1) {
            System.out.println();
        }
    }
```

```
    }
}
```

OK, now let's have a look at `findInterval`. Obviously the intervals can't overlap. If intervals overlap it means there are two or more cars with the same license plate. So we assume there aren't. This suggests an idea on how to find intervals given the lists of enter and exit times for a license plate.

We assume the lists `enters` and `exits` are sorted in chronological order. We repeat the search loop until any of `enters` or `exits` becomes empty. We get the first entry `enter` from `enters` and try to find an event from the `exits` list such that its timestamp is after the `start` timestamp. If there's not such an entry, we remove `enter` from `enters` and get another one. If there's a candidate in the `exits`, then we must check if there aren't any other events in `starts` that lie in between `start` and `exit`. If there are, we can't use this `start`, because it would mean overlapping. So we skip such `start`. If, however, there is no overlapping, then we've found a candidate interval and we return it. When we can't find an interval, we return `null`.

81 ⟨*4.6 Find Interval* 81⟩≡

```
while (!enters.isEmpty() && !exits.isEmpty()) {
    final Event enter = enters.remove(0);
    final Event exit = exits.stream()
            .filter(x -> x.timestamp.isAfter(enter.timestamp))
            .findFirst()
            .orElse(null);

    if (exit == null) {
        continue;
    }

    boolean overlap = enters.stream()
            .anyMatch(x -> x.timestamp.isAfter(enter.timestamp) &&
                    x.timestamp.isBefore(exit.timestamp));

    if (!overlap) {
        exits.remove(exit);
        return new Event[] { enter, exit };
    }
}
return null;
```

Now let's implement `getBills`. We will keep our billing in a map, with the key being license plate, and the value being the bill. The `group` will group events per license plate. Then for each entry in this group we will prepare a bill. Finally, we return the entries sorted by license plates in alphabetical order.

82a ⟨*4.6 Get Bills* 82a⟩≡
```
Map<String, AtomicInteger> bills = new HashMap<>();
Map<String, List<Event>> groups = events.stream().collect(
        Collectors.groupingBy(Event::getLicense, Collectors.toList()));

for (Entry<String, List<Event>> entry : groups.entrySet()) {
    ⟨4.6 Billing 82b⟩
}

return bills.entrySet().stream()
        .sorted((x, y) -> x.getKey().compareTo(y.getKey()))
        .collect(toList());
```

Bill preparation will be done using these steps: First, we partition the events into to groups, `enters` and `exits`. If either of these arrays is empty, we skip and move onto another license plate.

82b ⟨*4.6 Billing* 82b⟩≡
```
String license = entry.getKey();
Map<String, List<Event>> actions = entry.getValue().stream()
        .collect(Collectors.groupingBy(Event::getAction,
                Collectors.toList()));
List<Event> enters = actions.get("enter");
List<Event> exits = actions.get("exit");
if (enters == null || exits == null) {
    continue;
}
```

Now, let's sort them, because our `findInterval` method expects them in sorted order.

82c ⟨*4.6 Billing* 82b⟩+≡
```
exits.sort(Comparator.comparing(Event::getTimestamp));
enters.sort(Comparator.comparing(Event::getTimestamp));
```

Then, in a loop, we call `findInterval` and calculate the amount to be added to the bill. This is achieved by getting the `enter`'s timestamp and getting the hour value. That will be the index into `rate` array, which holds rates for specific hours. Then we multiply it by the distance to get the total price plus 100, price for each trip.

82d ⟨*4.6 Billing* 82b⟩+≡
```
Event[] interval = findInterval(enters, exits);
while (interval != null) {
    Event enter = interval[0];
    Event exit = interval[1];
    bills.putIfAbsent(license, new AtomicInteger(200));
    int distance = Math.abs(exit.location - enter.location);
    bills.get(license).addAndGet(
            rate.get(enter.timestamp.getHour()) * distance + 100);
    interval = findInterval(enters, exits);
}
```

This concludes the program.

4.7 ShellSort

The key to this problem answer is to note that all the items in the stack above the one that is about to be moved will move down. Therefore we just need to find all such elements, and everything else will need to be moved using the operation described in the problem statement.

Let's start with the input/output assuming that we have `getStrategy` method which takes `input` array and the `target` array and returns an answer, i.e. a list of items that need to be moved to the top:

83 ⟨*ShellSort* 83⟩≡

```
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Collections;
import java.util.List;

class ShellSort {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    private static List<String> getStrategy(List<String> input, List<String> target) {
        ⟨4.7 Implementation 84⟩
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.valueOf(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            int count = Integer.valueOf(reader.readLine().trim());
            List<String> input = reader.lines().limit(count).collect(toList());
            List<String> target = reader.lines().limit(count).collect(toList());
            getStrategy(input, target).forEach(System.out::println);
            System.out.println();
        }
    }
}
```

OK, let's get to the implementation of the method that finds the optimal strategy. We start from the bottom of the lists and work towards the top, comparing the items. The idea is that we move sequentially in the `target` array and move towards the top in the `input` array potentially skipping some elemnts until we hit the start of the array. The index in the `target` array, at which we broke the loop, will be the point that will divide the `target` array into two parts: Elements above it are the elements that will need to be moved, eleemnts below do not need to be moved.

84 ⟨*4.7 Implementation* 84⟩≡

```
int i = input.size() - 1;
int j = target.size() - 1;
while (i >= 0 && j >= 0) {
    while (j >= 0 && !target.get(i).equals(input.get(j))) {
        j--;
    }
    if (j < 0) {
        break;
    }
    i--;
    j--;
}
List<String> output = target.subList(0, i + 1);
Collections.reverse(output);
return output;
```

4.8 Football (aka Soccer)

It took me so many attempts before the online judge accepted my solution (15 times!) so that at some point I thought I'd give up. It turned out that my solution was absolutely correct (of course it was, it's a trivial task!). The only incorrect thing was around input/output encoding. Ludicrous!

In the code below probably Java String's built-in `split` method could have been sufficient, but because I was desperate in figuring out why the online judge didn't like my solution I ended up writing my custom made method. I've left it as is as I don't want to spend any more time on this task.

85 ⟨*Football aka Soccer* 85⟩≡

```
import java.io.BufferedReader;
import java.io.BufferedWriter;
import java.io.IOException;
import java.io.InputStreamReader;
import java.io.OutputStreamWriter;
import java.io.PrintWriter;
import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.Collections;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class FootballAkaSoccer {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in, Charset.forName("ISO-8859-1")));
    private static final PrintWriter output = new PrintWriter(
            new BufferedWriter(
                    new OutputStreamWriter(System.out,
                            Charset.forName("ISO-8859-1"))));

    private static class TeamRank implements Comparable<TeamRank> {
        private final String name;
        private int points;
        private int goalsScored;
        private int goalsAgainst;
        private int gamesPlayed;
        private int wins;
        private int ties;
        private int losses;

        public TeamRank(String name) {
            this.name = name;
        }

        @Override
        public String toString() {
            return name + " " + points + "p, " + gamesPlayed + "g (" + wins +
                    "-" + ties + "-" + losses + "), " +
                    (goalsScored - goalsAgainst) + "gd (" + goalsScored + "-" +
                    goalsAgainst + ")";
```

```
        }

        @Override
        public int compareTo(TeamRank o) {
            int pointsCmp = o.points - points;
            if (pointsCmp != 0) {
                return pointsCmp;
            }

            int winsCmp = o.wins - wins;
            if (winsCmp != 0) {
                return winsCmp;
            }

            int gdCmp = (o.goalsScored - o.goalsAgainst) -
                    (goalsScored - goalsAgainst);
            if (gdCmp != 0) {
                return gdCmp;
            }

            int goalsCmp = o.goalsScored - goalsScored;
            if (goalsCmp != 0) {
                return goalsCmp;
            }

            int gamesPlayedCmp = gamesPlayed - o.gamesPlayed;
            if (gamesPlayedCmp != 0) {
                return gamesPlayedCmp;
            }

            return name.toLowerCase().compareTo(o.name.toLowerCase());
        }
    }

    private static class Game {
        private final String[] teams;
        private final int[] goals;

        private String[] split(String input, String ch) {
            return new String[] {
                    input.substring(0, input.indexOf(ch)),
                    input.substring(input.indexOf(ch) + 1, input.length())
            };
        }

        public Game(String game) {
            String[] parts = split(game, "@");
            goals = new int[] {
                    Integer.parseInt(split(parts[0], "#")[1]),
                    Integer.parseInt(split(parts[1], "#")[0])
            };
            teams = new String[] {
                    split(parts[0], "#")[0],
```

```
                split(parts[1], "#")[1]
        };
    }
}

private static TeamRank getRank(String teamName, List<Game> games) {
    TeamRank rank = new TeamRank(teamName);
    for (Game g : games) {
        boolean t1 = g.teams[0].equals(teamName);
        boolean t2 = g.teams[1].equals(teamName);
        if (t1 || t2) {
            rank.goalsScored += g.goals[t1 ? 0 : 1];
            rank.goalsAgainst += g.goals[t1 ? 1 : 0];
            rank.gamesPlayed++;
            int cmp = Integer.compare(g.goals[t1 ? 0 : 1],
                    g.goals[t1 ? 1 : 0]);
            if (cmp == 0) {
                rank.ties++;
                rank.points++;
            } else if (cmp == -1) {
                rank.losses++;
            } else {
                rank.wins++;
                rank.points += 3;
            }
        }
    }
    return rank;
}

public static void main(String[] args) throws IOException {
    int n = Integer.parseInt(reader.readLine());
    for (int i = 0; i < n; ++i) {
        String tournamentName = reader.readLine();
        int teamCount = Integer.parseInt(reader.readLine().trim());
        Set<String> teamSet = new HashSet<>();
        for (int j = 0; j < teamCount; ++j) {
            teamSet.add(reader.readLine());
        }
        int gamesCount = Integer.parseInt(reader.readLine().trim());
        List<Game> games = new ArrayList<>();
        for (int j = 0; j < gamesCount; ++j) {
            games.add(new Game(reader.readLine()));
        }
        List<String> teams = new ArrayList<>(teamSet);
        List<TeamRank> ranks = new ArrayList<>();
        for (int j = 0; j < teams.size(); ++j) {
            ranks.add(getRank(teams.get(j), games));
        }
        Collections.sort(ranks);
        output.println(tournamentName);
        for (int j = 0; j < ranks.size(); ++j) {
            output.println((j + 1) + ") " + ranks.get(j));
```

```
                output.flush();
            }
            if (i < n - 1) {
                output.println();
            }
        }
        output.close();
    }
}
```

5 Arithmetic and Algebra

5.1 Primary Arithmetic

This task is trivial.

89 ⟨*Primary Arithmetic* 89⟩≡

```
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.Comparator;
import java.util.List;

public class PrimaryArithmetic {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int[] asArray(String input, int pad) {
        int[] a = new int[input.length() + pad];
        for (int i = 0; i < input.length(); ++i) {
            a[i] = input.charAt(input.length() - i - 1) - '0';
        }
        return a;
    }

    public static int count(List<String> input) {
        int[] a = asArray(input.get(0), 0);
        int[] b = asArray(input.get(1),
                input.get(0).length() - input.get(1).length());
        int carry = 0;
        int count = 0;
        for (int i = 0; i < a.length; ++i) {
            int c = a[i] + b[i] + carry;
            if (c >= 10) {
                carry = 1;
                count++;
            } else {
                carry = 0;
            }
        }
        return count;
    }

    public static String toMessage(int count) {
        if (count == 0) {
            return "No carry operation.";
        } else if (count == 1) {
            return "1 carry operation.";
        } else {
            return count + " carry operations.";
        }
```

```
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<String> input = Arrays
                    .stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals(" "))
                    .sorted(Comparator.comparing(String::length).reversed())
                    .collect(toList());
            if (input.get(0).equals("0") && input.get(1).equals("0")) {
                break;
            }
            System.out.println(toMessage(count(input)));
        }
    }
}
```

5.2 Reverse And Add

This task is trivial.

91 ⟨*Reverse And Add* 91⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

class ReverseAndAdd {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static long reverse(long value) {
        long reversed = 0;
        while (value > 9) {
            reversed = reversed * 10 + (value % 10);
            value /= 10;
        }
        reversed = reversed * 10 + value;
        return reversed;
    }

    private static boolean isPalindrome(long value) {
        return value == reverse(value);
    }

    public static long[] calculate(long value) {
        int count = 0;
        do {
            value = value + reverse(value);
            count++;
        } while (!isPalindrome(value));
        return new long[] { count, value };
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            long v = Integer.parseInt(reader.readLine().trim());
            long[] res = calculate(v);
            System.out.println(res[0] + " " + res[1]);
        }
    }
}
```

5.3 The Archeologists' Dilemma

Unlike the previous two tasks, this tasks is quite challenging.

Let's paraphrase this task in mathematical terms. For a given number v find positive integers m and n such that

$$v \cdot 10^n \leq 2^m < (v+1) \cdot 10^n$$

where $n \geq l(v) + 1$, and $l(v)$ is the number of digits in v.

Let's take common logarithms on that inequality

$$log(v \cdot 10^n) \leq log(2^m) < log((v+1) \cdot 10^n)$$

which is the same as

$$log(v) + log(10^n) \leq log(2^m) < log(v+1) + log(10^n)$$

and

$$log(v) + n \cdot log(10) \leq m \cdot log(2) < log(v+1) + n \cdot log(10)$$

which is the same as

$$log(v) + n \leq m \cdot log(2) < log(v+1) + n.$$

This solves the task, because all we need to do now is to iterate on n starting with $n = l(v) + 1$. For a given n we find an initial m by using the left part of the inequality, so

$$m = \lfloor \frac{log(v) + n}{log(2)} \rfloor$$

Then we increment m while $log(v) + n \geq m \cdot log(2)$. Once this loop stops, we check if $m \cdot log(2) < log(v+1) + n$, and if so, m is the answer. Otherwise, we increment n and start everything all over again.

92 ⟨*The Archeologists Dilemma* 92⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigDecimal;

public class TheArcheologistsDilemma {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static long calculate(long v) {
        long n = BigDecimal.valueOf(v).precision() + 1;
        final double left = Math.log10(v);
        final double right = Math.log10(v + 1);
        final double log10_2 = Math.log10(2);
        while (true) {
            long m = (long) Math.floor((left / log10_2) + n / log10_2);
            while (left + n > (log10_2 * m)) {
                m++;
            }
```

```
            if (right + n > (log10_2 * m)) {
                return m;
            }
            n++;
        }
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            System.out.println(calculate(Long.parseLong(currentLine.trim())));
        }
    }
}
```

5.4 Ones

This is a little nice problem but it may take some time to come up with a proper solution. Obviously these "minimum multiples" of `n` can quickly become too large, and so we can't use the standard types of the language to do the calculations. The next natural idea would be to try to use `BigInteger` and repeatedly do $x = x \times 10 + 1$ and then checking `x % n == 0` until it becomes `true`. But this is not a solution, it's too slow.

Another idea would be too come up with some clever "divisibility rules" to see if a given n divides a number that has only 1s in it. But this a dead end too.

Of course, the general idea is to simply test if `x % n == 0` for a given n where x is a number consisting of 1s only.

To do that we can simply do long division and keep appending 1s to the reminder until it doesn't divide without a reminder.

Before we implement the long division, let's write input/output:

93 ⟨*Ones* 93⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

class Ones {
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));

    private static int calculate(int n) {
        ⟨5.4 Calculation 94a⟩
    }

    public static void main(String[] args) throws IOException {
        reader.lines().map(Integer::parseInt)
                      .map(Ones::calculate)
                      .forEach(System.out::println);
    }
}
```

We implement the case when `n` is 1 first:

94a ⟨*5.4 Calculation* 94a⟩≡

```
if (n == 1) {
    return 1;
}
```

Any other number can be calculated using the long division.

Let's workout a small example. Let's say we want to find the minimum multiple for $n = 91$. We start with $s = 11$ and $r = 11$. But clearly because $s < n$ we need to append one more 1, $s = r \times 10 + 1$, so now $s = 111$, and $r = s - (n * \lfloor s/n \rfloor)$, so $r = 20$; and since $r \neq 0$ we continue by extending $s = r \times 10 + 1$ and then repeat the steps until $r = 0$. But note though that $r = s - (n * \lfloor s/n \rfloor)$ is equivalent to `r = s % n`.

OK, now we can capture that in code:

94b ⟨*5.4 Calculation* 94a⟩+≡

```
int l = 0;
int r = 0;
do {
    r = (r * 10 + 1) % n;
    l++;
} while (r > 0);
return l;
```

Brilliant.

5.5 A Multiplication Game

Unfortunately I couldn't come up with anything more clever than a recursive algorithm that tries all the possible multipliers at each step and chooses the one that leads to the win. Because a direct recursive algorithm without any optimization would be awfully slow, we need some memoization. This is possible, because many multipliers would lead to the same value, so we can cache them, we just need to keep track of whose turn it is at this moment of time. For that we will have a list of two maps, one for each player, and the map will map a value to the result.

The program is quite compact:

95 ⟨*A Multiplication Game* 95⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;

class MultiplicationGame {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int solve(long p, long n, int t,
            List<HashMap<Long, Integer>> memo) {
        if (p >= n) {
            return t - 1;
        }

        int s = t % 2;
        for (int i = 9; i >= 2; --i) {
            int result = 0;
            long next = p * i;
            if (memo.get(s).containsKey(next)) {
                result = memo.get(s).get(next);
            } else {
                result = solve(next, n, t + 1, memo);
                memo.get(s).put(next, result);
            }
            if (result % 2 == t % 2) {
                return result;
            }
        }

        return t + 1;
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            long input = Long.parseLong(currentLine.trim());
```

```
            List<HashMap<Long, Integer>> memo = new ArrayList<>();
            memo.add(new HashMap<Long, Integer>());
            memo.add(new HashMap<Long, Integer>());
            System.out.println(
                    solve(1, input, 1, memo) % 2 == 0 ? "Ollie wins."
                            : "Stan wins.");
        }
    }
}
```

5.6 Polynomial Coefficients

This task is very straightforward, we just use the Newton's generalized binomial theorem.

The formula is:

$$\frac{n!}{k_1!k_2!\dots k_m!}$$

We won't calculate it as is, but first simplify the fraction whenever possible.

97 ⟨*Polynomial Coefficients* 97⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

public class PolynomialCoefficients {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static List<Integer> expand(int n) {
        List<Integer> res = new ArrayList<Integer>();
        for (int i = n; i > 0; --i) {
            res.add(i);
        }
        return res;
    }

    private static long calculate(int n, List<Integer> v) {
        List<Integer> numerator = expand(n);
        List<Integer> denominator = new ArrayList<>();
        v.stream().filter(x -> x > 0)
                .forEach(x -> denominator.addAll(expand(x)));
        Iterator<Integer> it = denominator.iterator();
        while (it.hasNext()) {
            if (numerator.remove(it.next())) {
                it.remove();
            }
        }
        return numerator.stream().reduce(1, Math::multiplyExact).intValue() /
                denominator.stream().reduce(1, Math::multiplyExact).intValue();
    }

    private static List<Integer> readList(String input) {
        return stream(input.trim().split(" "))
                .filter(x -> !x.equals("")).map(Integer::parseInt)
                .collect(toList());
    }
```

```
    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> nk = readList(currentLine);
            List<Integer> v = readList(reader.readLine());
            System.out.println(calculate(nk.get(0), v));
        }
    }
}
```

5.7 The Stern-Brocot Number System

This task is just about searching the binary tree, which is trivial.

99 ⟨*The Stern-Brocot Number System* 99⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

public class TheSternBrocotNumberSystem {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int gcd(int a, int b) {
        while (b != 0) {
            int t = b;
            b = a % b;
            a = t;
        }
        return a;
    }

    private static String get(int a, int b) {
        int gcd = gcd(a, b);
        a = a / gcd;
        b = b / gcd;

        int[] l = new int[] { 0, 1 };
        int[] m = new int[] { 1, 1 };
        int[] r = new int[] { 1, 0 };

        StringBuilder result = new StringBuilder();
        while (true) {
            int cmp = Integer.compare(a * m[1], b * m[0]);
            if (cmp == -1) {
                r = new int[] { m[0], m[1] };
                m = new int[] { l[0] + m[0], l[1] + m[1] };
                result.append("L");
            } else if (cmp == 1) {
                l = new int[] { m[0], m[1] };
                m = new int[] { r[0] + m[0], r[1] + m[1] };
                result.append("R");
            } else {
                break;
            }
        }
        return result.toString();
    }

    private static List<Integer> readList(String input) {
```

```
        return stream(input.trim().split(" "))
                .filter(x -> !x.equals("")).map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> ab = readList(currentLine);
            if (ab.get(0) == 1 && ab.get(1) == 1) {
                break;
            }
            System.out.println(get(ab.get(0), ab.get(1)));
        }
    }
}
```

5.8 Pairsumonious Numbers

Let's have a look at a small example. Let's suppose our numbers a_1, a_2, a_3, a_4 are all positive and in ascending order then their sums are $a_1 + a_2, a_1 + a_3, a_1 + a_4, a_2 + a_3, a_2 + a_4, a_3 + a_4$ and are also in ascending order. Let's suppose now we only have $b_1, \ldots, b_6$, where one of the possible assignments for $b_1, \ldots, b_6$ can be, for example, $b_1 = a_1 + a_2, b_2 = a_1 + a_3, b_3 = a_1 + a_4, b_4 = a_2 + a_3, b_5 = a_2 + a_4, b_6 = a_3 + a_4$. How can we restore $a_1, \ldots, a_4$ without knowing which of such assignments was used initially?

We can start with some value x by assuming that $a_1 = x$ (let's suppose any number for now). Then a_2 is determined by one of the values $b_1, \ldots, b_6$. Let's choose b_1, then the second number is obviously $a_2 = b_1 - x$. Similarly, we can work out a_3 and a_4. Of course, there are multiple choices at each step, so we exhaustively try all possible combinations by using a backtracking technique.

But straightforward backtracking won't work. First, we don't know which range to select the initial x from. Second, trying all the combinations without eliminating some dead end combinations will be too slow. So we need to narrow the range for the x, and also not to proceed with some combinations that don't lead to a solution.

We assumed that the values were all positive, however it's easy to see that our backtracking would still work if the number weren't positive.

Let's see how can we eliminate the dead end combinations. Let's suppose we have a_1, a_2 and the other two values are undetermined yet. If $a_1 + a_2$ is a value that is larger than any of the values $b_1, \ldots, b_6$, then we don't need to look for the other two undetermined values. This is because values in $a_1, \ldots, a_4$ and in $b_1, \ldots, b_6$ are in ascending order, and any other combination will lead to even larger values.

Let's have a look at what is the range for our initial value x. Obviously trying the whole range of the integer type is not practical. The upper bound for the range is easy to determine though, it's the largest value in the input. The lower bound can be taken as the minimum of the differences of the pairs made of the input values.

Can you see why?

101 ⟨*Pairsumonious Numbers* 101⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.BitSet;
import java.util.List;
import java.util.stream.Collectors;

public class PairsumoniousNumbers {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private final static int[] IMPOSSIBLE = new int[0];

    private enum Result {
        success, failure, overflow
    };

    private final int n;
    private final int[] v;
```

```
private final BitSet excluded;

public int[] getSolution() {
    int[] partialResult = new int[n];

    int[] bounds = new int[] { Integer.MAX_VALUE, v[v.length - 1] };
    for (int i = 0; i < v.length; ++i) {
        for (int j = i + 1; j < v.length; j++) {
            bounds[0] = Math.min(bounds[0], v[i] - v[j]);
        }
    }

    if (bounds[0] > bounds[1]) {
        int t = bounds[0];
        bounds[0] = bounds[1];
        bounds[1] = t;
    }

    if (bounds[1] < 0) {
        bounds[1] = 0;
    }

    for (int i = bounds[0]; i < bounds[1]; ++i) {
        partialResult = new int[n];
        partialResult[0] = i;
        int[] result = search(partialResult, 0, 0);
        if (result != IMPOSSIBLE) {
            Arrays.sort(result);
            return result;
        }
    }
    return IMPOSSIBLE;
}

PairsumoniousNumbers(List<Integer> input) {
    n = input.get(0);
    v = new int[input.size() - 1];
    for (int i = 1; i < input.size(); ++i) {
        v[i - 1] = input.get(i);
    }
    Arrays.sort(v);
    excluded = new BitSet(n);
}

private Result verify(int upTo, int[] solution) {
    excluded.clear();
    for (int i = 0; i < upTo; ++i) {
        for (int j = i + 1; j < upTo; ++j) {
            int currValue = solution[i] + solution[j];
            if (currValue > v[v.length - 1]) {
                return Result.overflow;
            }
```

```
                int p = Arrays.binarySearch(v, currValue);
                if (p < 0) {
                    return Result.failure;
                }
                while (p > 0 && v[p] == v[p - 1]) {
                    p--;
                }
                while (p < v.length - 1 && excluded.get(p) &&
                        v[p] == v[p + 1]) {
                    p++;
                }
                if (p == v.length || excluded.get(p)) {
                    return Result.failure;
                }
                excluded.set(p);
            }
        }

        for (int i = 0; i < upTo - 1; ++i) {
            if (!excluded.get(i)) {
                return Result.failure;
            }
        }

        return Result.success;
    }

    private int[] search(int[] partialSolution, int last, int pos) {
        for (int i = pos; i < v.length; ++i) {
            partialSolution[last + 1] = v[i] - partialSolution[0];
            Result verificationResult = verify(last + 2, partialSolution);
            if (verificationResult == Result.success) {
                if (last + 1 < n - 1) {
                    int[] solution = search(partialSolution, last + 1, i + 1);
                    if (solution != IMPOSSIBLE) {
                        return solution;
                    }
                } else {
                    return partialSolution;
                }
            } else if (verificationResult == Result.overflow) {
                break;
            }
        }
        return IMPOSSIBLE;
    }

    private static String toString(int[] arr) {
        return Arrays.stream(arr).mapToObj(String::valueOf)
                .collect(Collectors.joining(" "));
    }

    public static void main(String[] args) throws IOException {
```

```
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            int[] solution = new PairsumoniousNumbers(input).getSolution();
            System.out
                    .println(solution == IMPOSSIBLE ? "Impossible"
                            : toString(solution));
        }
    }

}
```

6 Combinatorics

6.1 How Many Fibs?

This task is particularly easy in Java because of `BigInteger` class.

105 ⟨*How Many Fibs* 105⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.util.List;

public class HowManyFibs {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            List<BigInteger> range = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(BigInteger::new)
                    .collect(toList());
            if (range.get(0).equals(BigInteger.ZERO) &&
                    range.get(1).equals(BigInteger.ZERO)) {
                break;
            }
            BigInteger fn2 = BigInteger.ZERO;
            BigInteger fn1 = BigInteger.ONE;
            BigInteger fn = fn2.add(fn1);
            long counter = 0;
            while (fn.compareTo(range.get(1)) <= 0) {
                if (fn.compareTo(range.get(0)) >= 0) {
                    counter++;
                }
                fn2 = fn1;
                fn1 = fn;
                fn = fn1.add(fn2);
            }
            System.out.println(counter);
        }
    }
}
```

6.2 How Many Pieces of Land?

Unlike the previous task this one is much more entertaining.

Let's see if we can consider this task a graph task? Actually yes we can[3]. We can consider

[3]There's a brilliant video that explains exactly the same solution: `https://youtu.be/K8P8uFahAgc`

points on an ellipse and diagonal intersections as vertices of a graph, and edges being the connecting segments. We are asked to find the number of regions of this graph. Note that the task asks for the maximum number of regions, which basically means no more than two segments should cross the same point.

There's an important characteristic in the graph theory called Euler's Formula:

$$V - E + F = 2$$

V is the number of vertices, E is the number of edges, and F is the number of regions bound by the edges (this includes the outer region). This characteristic holds for any finite connected planar graph without intersecting edges.

Using the Euler's Formula we can find the number of faces once we know V and E. That is

$$F = 2 - V + E$$

Let's count. We have n points on the ellipse. How many different chords can we have? Well we can have $\binom{n}{2}$ different chords. Now we need to find at how many points do these chords intersect. Every intersection point is uniquely determined by two lines and every two lines are determined by four different points on the ellipse. Simply put, this means that every intersection point is uniquely determined by four points on the ellipse. How many such different intersection points are there? It's precisely $\binom{n}{4}$. Because we have additional n points on the ellipse the total number of vertices of the graph is $V = n + \binom{n}{4}$.

Now we have two quantities: the number of chords and the number of points at which they intersect. This allows us to calculate the number of edges in our graph. Notice that every intersection point breaks the segment into two peaces. So if we take two lines and they intersect, we have four segments. If we take three lines that intersect at three points, we get nine different segments. Generally, if we have m segments intersecting at p points, then we end up with $m + 2p$ segments. (Note that this is only true if no more than two lines intersect at each point.) This in turn means that our graph will have $E = \binom{n}{2} + 2\binom{n}{4} + n$ edges; additional n because of the ellipse's segments are part of the graph too.

Now we just plug these value into the formula:

$$F = 2 - n - \binom{n}{4} + \binom{n}{2} + 2\binom{n}{4} + n$$

which is the same as:

$$F = 2 + \binom{n}{2} + \binom{n}{4}$$

This formula counts in the outer region too, but in our tasks it's not counted, so we just reduce it by one. The final formula is:

$$F = 1 + \binom{n}{2} + \binom{n}{4}$$

This translates into Java code extremely easily:

106 ⟨*How Many Pieces of Land* 106⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
```

```
import java.math.BigInteger;

public class HowManyPiecesOfLand {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static BigInteger calculate(BigInteger v) {
        BigInteger t1 = v.multiply(v.subtract(BigInteger.ONE))
                .divide(BigInteger.valueOf(2));
        BigInteger t2 = v.multiply(v.subtract(BigInteger.ONE))
                .multiply(v.subtract(BigInteger.valueOf(2)))
                .multiply(v.subtract(BigInteger.valueOf(3)))
                .divide(BigInteger.valueOf(4 * 3 * 2));
        return BigInteger.ONE.add(t2).add(t1);
    }

    public static void main(String[] args)
            throws NumberFormatException, IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            System.out.println(
                    calculate(new BigInteger(reader.readLine().trim())));
        }
    }
}
```

6.3 Counting

There's always one easy case: that's when the number of 1s matches n, so their sum matches n. But now we can start constructing other cases. If we remove three 1s from it and replace by 3 they would still add up to n. Similarly, if we remove two 1s and replace by 2 they would still add up to n. Now, once we removed two or three 1s we end up with strings of ones of length $n-2$ or $n-3$ to which we can do the same operation recursively. This suggests a recursive formula:

$$T(n) = T(n-2) + T(n-3)$$

But 4 and 1 should be counted as 1 according to the problem statement. We can think about this in a similar manner, if we have n 1s we can take out one and replace by 4, so it's an additional $T(n-1)$ term in into the recursive formula above. But since 1 and 4 are the same we also must take into account a symmetric case where the first $n-1$ ones are replaced by $n-1$ fours, so that's a yet additional $T(n-1)$ term. The final formula is:

$$T(n) = T(n-2) + T(n-3) + 2T(n-1)$$

This value grows very quickly, so we will be using `BigInteger` class. We will also implement it in a iterative manner rather than recursive function calls.

108 ⟨*Counting* 108⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;

public class Counting {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static BigInteger count(int n) {
        final BigInteger two = BigInteger.valueOf(2);
        if (n == 1) {
            return two;
        } else if (n == 0) {
            return BigInteger.ONE;
        } else if (n < 0) {
            return BigInteger.ZERO;
        }

        BigInteger[] prev = new BigInteger[] { BigInteger.ZERO, BigInteger.ONE,
                two };

        for (int i = 1; i < n; ++i) {
            BigInteger next = prev[2].multiply(two).add(prev[1]).add(prev[0]);
            prev[0] = prev[1];
            prev[1] = prev[2];
            prev[2] = next;
        }

        return prev[2];
```

```
    }

    public static void main(String[] args)
            throws NumberFormatException, IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null) {
            System.out.println(count(Integer.parseInt(currentLine.trim())));
        }
    }
}
```

6.4 Expressions

It's an interesting task! To solve it let's have a look at the recurrence formula of the Catalan numbers. As the textbook explains it's constructed in this way: The proper string of parentheses starts with a single left parenthesis and is matched by some right parenthesis. This divides the word into two parts both of which are properly constructed. In other words any properly balanced string of parentheses can always be represented as $w = (w_1)w_2$ where both w_1 and w_2 are proper strings of parentheses, possibly of zero length; hence the recursive formula.

Notice that w_1 is in the parenthesis in $w = (w_1)w_2$, so the depth of (w_1) is at least 1. We can keep track of the depth of recursion and stop as long as we reach the required depth:

$$C[n, d] = \sum_{i=0}^{n-1} C[i, d-1]C[n-1-i, d]$$

and

$$C[n, d] = 1 \quad \text{if } n = 0 \text{ or } d = 1$$

Here n is the number of pairs of parentheses, d is the required depth.

The formula above returns all the strings with the depths of up to d. To get all the proper strings of parentheses of depth d we simply need to subtract all the strings of depth up to $d-1$. So the final formula is

$$F[m, d] = C[m/2, d] - C[m/2, d-1]$$

Here m is the length of a string as specified in the problem statement. We divide by two as the length should always be an even number because each opening parenthesis must have a closing parenthesis. If the number is odd, we will return 0.

Of course we need memoization in the implementation which is trivial in this case.

110 ⟨*Expressions* 110⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.List;

public class Expressions {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final BigInteger[][] memo = new BigInteger[151][151];

    public static BigInteger c(int n, int d) {
        if (d == 1 || n == 0) {
            return BigInteger.ONE;
        }

        if (memo[n][d] != null) {
```

```
            return memo[n][d];
        }

        BigInteger v = BigInteger.ZERO;
        for (int i = 0; i <= n - 1; ++i) {
            BigInteger v1 = c(i, d - 1);
            BigInteger v2 = c(n - i - 1, d);
            v = v1.multiply(v2).add(v);
        }

        memo[n][d] = v;
        return v;
    }

    public static BigInteger solve(int n, int d) {
        if (n % 2 != 0) {
            return BigInteger.ZERO;
        }
        for (int i = 0; i < memo.length; ++i) {
            Arrays.fill(memo[i], null);
        }
        return c(n / 2, d).subtract(c(n / 2, d - 1));
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().isEmpty()) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            System.out.println(solve(input.get(0), input.get(1)));
        }
    }
}
```

6.5 Complete Tree Labeling

Let's look at a simple case, a tree of degree k of depth 1. In this case there are $k+1$ nodes: the root and k children. We have a list of labels in ascending order. To preserve the heap structure we need to take the minimum element from the list of labels as the root. Then we have $k!$ ways to arrange the children using the remaining labels. This is going to be our base case for the recursion.

Let's define $f(b, s, k, h)$ as a function that returns the number of ways a k-ary tree can be labeled. Here b is the base value (that is $k!$), s is the number of nodes in the k-ary tree, k is, obviously, the degree of the tree, and h is the height of the tree.

The total number of nodes in a k-ary tree of height h is

$$n(k, h) = \frac{k^{h+1} - 1}{k - 1}$$

Let's say we are at depth 1 in the k-ary tree. At this level there are k subtrees each having $n(k, h-1) = (n(k, h) - 1)/k$ nodes. In how many ways can we choose the labels from the list of labels for this subtree? For the first subtree we can choose labels in

$$\binom{n(k,h) - 1 - 1}{n(k,h-1) - 1}$$

ways. One -1 is because of the parent node, and another -1 is because of the root of the subtree. For the second subtree we can now choose labels in

$$\binom{n(k,h) - 1 - 1 - n(k,h-1)}{n(k,h-1) - 1}$$

ways, and so on for the other subtrees. Remember that all this can be rearranged in $k!$ ways and that each subtree can be labeled in $f(b, n(k, h-1), k, h-1)$ ways. Multiplying all these together gives us the answer.

112 ⟨*Complete Tree Labeling* 112⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.util.ArrayList;
import java.util.Iterator;
import java.util.List;

class CompleteTreeLabeling {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static List<BigInteger> expand(int n, int upto) {
        List<BigInteger> res = new ArrayList<BigInteger>();
        for (int i = n; i > upto; --i) {
            res.add(BigInteger.valueOf(i));
        }
        return res;
```

```
}

private static BigInteger mult(List<BigInteger> l) {
    return l.stream().reduce(BigInteger.ONE, (x, y) -> x.multiply(y));
}

private static BigInteger choose(int n, int k) {
    List<BigInteger> numerator = expand(n, (n - k));
    List<BigInteger> denominator = expand(k, 0);
    Iterator<BigInteger> it = denominator.iterator();
    while (it.hasNext()) {
        if (numerator.remove(it.next())) {
            it.remove();
        }
    }
    return mult(numerator).divide(mult(denominator));
}

private static BigInteger count(BigInteger base, int treeSize, int k,
        int h) {
    if (h == 1) {
        return base;
    }
    int subtreeSize = (treeSize - 1) / k;
    BigInteger subtreeCount = count(base, subtreeSize, k, h - 1);
    BigInteger count = base;
    for (int i = subtreeSize - 1; i <= treeSize - 2; i += subtreeSize) {
        count = count.multiply(subtreeCount)
                .multiply(choose(i, subtreeSize - 1));
    }
    return count;
}

private static BigInteger solve(int k, int h) {
    if (k == 1) {
        return BigInteger.ONE;
    }
    int pow = 1;
    for (int i = 0; i < h + 1; ++i) {
        pow *= k;
    }
    return count(mult(expand(k, 0)), (pow - 1) / (k - 1), k, h);
}

public static void main(String[] args) throws IOException {
    String currentLine;
    while ((currentLine = reader.readLine()) != null) {
        List<Integer> input = stream(currentLine.trim().split(" "))
                .filter(x -> !x.equals("")).map(Integer::parseInt)
                .collect(toList());
        System.out.println(solve(input.get(0), input.get(1)));
    }
}
```

```
}
```

6.6 The Priest Mathematicians

We know that the answer to the classic Hanoi Towers task is $T(n) = 2^n - 1$. However, in this task it's slightly modified where we have an option to use the four pegs instead of three, as described in the problem statement.

It's really easy to come up with a recursive formula:

$$A(n) = min\{2^{n-k} - 1 + 2A(k) : k \in (1, n)\}$$

This is because first we use four pegs to move k discs on to the intermediate peg, then use the classic algorithm using the three pegs, finally use four pegs to move k discs again. We try all such $k \in (1, n)$ and find the minimum value.

This formula coded as is would give the right answer, but it will be very slow. Still, it's very useful to have a look at the first few values:

k	n	A(n)
1	2	3
1	3	5
1	4	9
2	5	13
3	6	17
3	7	25
4	8	33
5	9	41

The first column is the k that lead to the minimal A(n). As you may have noticed it grows monotonically. This suggests how to construct the values iteratively: We simply calculate the value with the current k and $k+1$, and if $k+1$ gives a better answer we accept it and advance k. This way we calculate all the values for each $n \in (1, 10000)$. Then we simply look up the answer in the table.

115 ⟨*The Priest Mathematician* 115⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;

class ThePriestMathematician {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final BigInteger TWO = BigInteger.valueOf(2);
    private static final BigInteger ONE = BigInteger.ONE;
    private static final BigInteger ZERO = BigInteger.ZERO;

    public static BigInteger[] generate() {
        BigInteger[] hanoi = new BigInteger[10001];
        hanoi[0] = ZERO;
        hanoi[1] = ONE;
        for (int i = 2, k = 1; i <= 10000; ++i) {
            BigInteger n1 = TWO.pow(i - k).subtract(ONE)
                    .add(hanoi[k].multiply(TWO));
            BigInteger n2 = hanoi[k + 1] != null ? TWO.pow(i - (k + 1))
                    .subtract(ONE).add(hanoi[k + 1].multiply(TWO))
                    : null;
```

```
            if (n2 != null && n2.compareTo(n1) == -1) {
                k++;
                hanoi[i] = n2;
            } else {
                hanoi[i] = n1;
            }
        }
        return hanoi;
    }

    public static void main(String[] args) throws IOException {
        BigInteger[] hanoi = generate();
        String currentLine = null;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            System.out.println(hanoi[Integer.parseInt(currentLine.trim())]);
        }
    }
}
```

6.7 Self-Describing Sequence

To solve this task we don't even need to come up with a recursive formula. We can simply generate the sequence and then lookup the required value. The trick however is to make the sequence compact. Instead of writing out each member of the sequence we just generate a list of ranges. Indexes of the elements in this list of ranges are the sequence values.

For example, instead of having $1, 2, 2, 3, 3, 4, 4, 4, 5, 5, 5, 6, 6, 6, 6$ etc we represent it as $(1, 1)_1, (2, 3)_2, (4, 5)_3, (6, 8)_4, (9, 11)_5, (12, 15)_6$ etc. Then, to get the value of $a(n)$, we use binary search to find the range into which n falls. Once we know the range, we simply return its index. For instance, $a(5) = 3$, because $5 \in (4, 5)_3$, and therefore $a(5) = 3$.

117 ⟨*Self Describing Sequence* 117⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

class SelfDescribingSequence {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private List<Tuple> seq = new ArrayList<>();
    private static final long limit = 2_000_000_000L;

    SelfDescribingSequence() {
        seq.add(new Tuple(1, 1));
        seq.add(new Tuple(2, 3));
        seq.add(new Tuple(4, 5));
        int tupleIndex = 2;
        boolean cont = true;
        do {
            Tuple currTuple = seq.get(tupleIndex);
            for (long i = currTuple.v1; i <= currTuple.v2; ++i) {
                Tuple lastTuple = seq.get(seq.size() - 1);
                if (lastTuple.v1 >= limit) {
                    cont = false;
                    break;
                }
                seq.add(new Tuple(lastTuple.v2 + 1, lastTuple.v2 + tupleIndex + 1));
            }
            tupleIndex++;
        } while (cont);
    }

    static class Tuple {
        private long v1;
        private long v2;

        public Tuple(long v1, long v2) {
            this.v1 = v1;
            this.v2 = v2;
```

```
        }
    }

    public long get(long n) {
        int i = Collections.binarySearch(seq, new Tuple(n + 1, 0),
                (x, y) -> Long.compare(x.v1, y.v1));
        if (i < 0) {
            i = Math.abs(i + 1);
        }
        return i;
    }

    public static void main(String[] args) throws IOException {
        SelfDescribingSequence g = new SelfDescribingSequence();
        String currentLine;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equalsIgnoreCase("")) {
            long n = Long.parseLong(currentLine.trim());
            if (n == 0) {
                break;
            }
            System.out.println(g.get(n));
        }
    }
}
```

6.8 Steps

The problem statement says that the length of the next step can be either the same or by one bigger or by one smaller as the previous one. Since this sequence always increases by one it's a simple arithmetic series. The sum of such series is $s(n) = \frac{n(1+a_n)}{2}$. We need to find n such that $x + s(n) \leq y$ and $x + s(n+1) > y$, here x and y as described in the problem statement. But we know that the first and the last steps must be of length 1, so we must have another arithmetic series to go down back to step of length 1. Therefore, we need to find n such that $x + 2s(n) \leq y$, and $x + 2s(n+1) > y$. Because $a_n = n$, $s(n) = \frac{n(1+n)}{2}$, so we are looking for an n such that $x + n(1+n) \leq y$ and $x + (n+1)(n+2) > y$, or, equivalently, $n(1+n) \leq y - x$ and $(n+1)(n+2) > y - x$.

Once we found such an n, we know that the number of steps is $2n$. But we need to try two cases to finalize the answer. First, if $n(1+n) < y - x$, that is strictly less, we need to try to add more steps. We know that at some point we reach the step of length n, so we should try inserting additional n and checking if we are still less than $y - x$. If not, keep adding. Adding n is allowed, because a new step can be of the same length as the previous one. If adding another n gets us above $y - x$, we need to try $n - 1$ and so on down to 1. This will give us the optimal number of steps.

We also need to check another case: $(n+1)(n+2) > y - x$. In this case we need to remove steps to get us to $y - x$. We can safely remove steps in pairs. Let's image our steps are as follows $1, 2, 3, 4, 4, 3, 2, 1$. We can remove 4 without breaking the rules. Then we can remove 4 again. Then 3, and then 3 one more time and so on. Basically we remove them in pairs. We keep removing until it gets us at or below $y - x$. Once we are there we are done. If we are below, we can still apply the procedure from the first case to align with $y - x$.

Now from those two cases we take the one that gives the smaller number of steps.

119 ⟨*Steps* 119⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

class Steps {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static long getLength(long limit) {
        long n = 1;
        while (n * (1 + n) <= limit) {
            n++;
        }
        if (n * (1 + n) > limit) {
            n--;
        }
        return n;
    }

    public static long getSteps(long n, long limit) {
        long sum = n * (1 + n);
```

```
        long steps = n * 2;
        long i = n;
        short j = 0;

        while (sum > limit) {
            sum -= i;
            steps--;
            if (j == 1) {
                j = 0;
                i--;
            }
            j++;
        }

        while (sum < limit) {
            while (sum + i <= limit) {
                sum += i;
                steps++;
            }
            if (i > 1) {
                i--;
            }
        }
        return steps;
    }

    public static long solve(long x, long y) {
        long limit = y - x;
        if (limit <= 1) {
            return limit;
        }
        long len = getLength(limit);
        return Math.min(getSteps(len, limit), getSteps(len + 1, limit));
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            List<Integer> input = stream(reader.readLine().trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            System.out.println(solve(input.get(0), input.get(1)));
        }
    }
}
```

7 Number Theory

7.1 Light, More Light

With this tasks we are basically asked to find the number of divisors of the given number. Once we know the number of divisors, we can figure out the last bulb state by checking if the number of the divisors is even or odd.

Let's sort out the input/output first as usual. We assume that we have `calculate` method that returns the number of divisors for a given number.

121a ⟨*Light, More Light* 121a⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
⟨7.1 Imports 121b⟩

class LightMoreLight {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    ⟨7.1 Variables 122a⟩

    LightMoreLight() {
        ⟨7.1 Constructor 122b⟩
    }

    public long calculate(long value) {
        ⟨7.1 Implementation 123⟩
    }

    public static void main(String[] args) throws IOException {
        LightMoreLight l = new LightMoreLight();
        String currentLine;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("0")) {
            long value = Long.parseLong(currentLine.trim());
            System.out.println(l.calculate(value) % 2 == 0 ? "no" : "yes");
        }
    }
}
```

OK, to figure out the number of divisors we will use the fundamental theorem of arithmetic. This theorem states that: Every integer greater than one either is prime itself or is the product of prime numbers, and that this product is unique, up to the order of the factors. To find the prime factorization we can use a straightforward algorithm: simply by dividing a number by the primes less that the number itself, trying them one by one.

Since we need to know the prime numbers, let's pre-calculate them first in the constructor. We won't need primes larger than $\sqrt{2^{32}-1}$, but we'll define a constant `MAX_PRIMES` a bit larger than that. We will use a classic algorithm for finding prime numbers, the sieve of Eratosthenes algorithm (see [3]).

121b ⟨*7.1 Imports* 121b⟩≡

```
import java.util.ArrayList;
import java.util.BitSet;
import java.util.List;
```

1	p_1	p_1^2	...	p_1^n
p_2	$p_1 p_2$	$p_1^2 p_2$	...	$p_1^n p_2$
p_2^2	$p_1 p_2^2$	$p_1^2 p_2^2$	...	$p_1^n p_2^2$
...	...	...	...	...
p_2^m	$p_1 p_2^m$	...	...	$p_1^n p_2^m$

Table 2: Divisors

122a ⟨*7.1 Variables* 122a⟩≡

```
private final List<Long> primes;
private final static int MAX_PRIMES = 70000;
```

122b ⟨*7.1 Constructor* 122b⟩≡

```
BitSet bits = new BitSet(MAX_PRIMES);
for (int i = 2; i < Math.sqrt(MAX_PRIMES); ++i) {
    if (!bits.get(i)) {
        int k = 0;
        int ii = i * i;
        int j = ii + k * i;
        while (j < MAX_PRIMES) {
            bits.set(j);
            k++;
            j = ii + k * i;
        }
    }
}
primes = new ArrayList<Long>();
for (int i = 2; i < bits.length(); ++i) {
    if (!bits.get(i)) {
        primes.add((long) i);
    }
}
```

Now the interesting part: In fact we don't need the prime numbers of the factorization, we only need their exponents to find out the number of divisors.

To see why, consider a number of the form $v = p_1^n$. The divisors of this number are $1, p_1, p_1^2, p_1^3, \ldots, p_1^n$; therefore the number of the divisors is $n + 1$.

Consider a number of the form $v = p_1^n p_2^m$. (See table).

Therefore the number of its divisors is $(n + 1)(m + 1)$.

Generally the number of the divisors for a number $v = p_1^{n_1} p_2^{n_2} \cdots p_k^{n_m}$ is $(n_1 + 1)(n_2 + 1) \cdots (n_m + 1)$.

123 ⟨*7.1 Implementation* 123⟩≡

```
List<Long> factors = new ArrayList<Long>();
for (int i = 0; i < primes.size() && value > 1 &&
        (primes.get(i) * primes.get(i)) <= value; ++i) {
    long p = 0;
    while (value % primes.get(i) == 0) {
        value /= primes.get(i);
        p++;
    }
    if (p > 0) {
        factors.add(p);
    }
}
if (value > 1) {
    factors.add(1L);
}
return factors.stream().map(x -> x + 1).reduce(1L, (a, b) -> a * b);
```

This concludes the program.

7.2 Carmichael Numbers

To solve this task we need two algorithms: the sieve of Eratosthenes (see [3]) and a fast modular exponentiation (specifically, right-to-left binary method, see [5]). BigInteger's modPow would do too, but it will be slower than a handcrafted one. I've tried BigInteger.modPow and the judge accepted my solution with 2.5s execution time. Handcrafted `modPow` though is considerably faster and the judge accepted my program with 560ms execution time.

124 ⟨*Carmichael Numbers* 124⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;
import java.util.Arrays;
import java.util.BitSet;

class CarmichaelNumbers {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private final int[] primes;
    private final static int MAX_PRIMES = 66000;

    CarmichaelNumbers() {
        BitSet bits = new BitSet(MAX_PRIMES);
        for (int i = 2; i < Math.sqrt(MAX_PRIMES); ++i) {
            if (!bits.get(i)) {
                int k = 0;
                int ii = i * i;
                int j = ii + k * i;
                while (j < MAX_PRIMES) {
                    bits.set(j);
                    k++;
                    j = ii + k * i;
                }
            }
        }

        int size = 0;
        for (int i = 2; i < bits.length(); ++i) {
            if (!bits.get(i)) {
                size++;
            }
        }

        int j = 0;
        primes = new int[size];
        for (int i = 2; i < bits.length(); ++i) {
            if (!bits.get(i)) {
                primes[j++] = i;
            }
```

```
        }
    }

    public long modPow(long b, long e, long m) {
        if (m == 1) {
            return 0;
        }

        long result = 1;
        b = b % m;
        while (e > 0) {
            if (e % 2 == 1) {
                result = (result * b) % m;
            }
            e = e >> 1;
            b = (b * b) % m;
        }
        return result;
    }

    public boolean isCarmichael(int n) {
        if (Arrays.binarySearch(primes, n) >= 0) {
            return false;
        }

        for (int i = 2; i < n; ++i) {
            if (modPow(i, n, n) != i) {
                return false;
            }
        }
        return true;
    }

    public static void main(String[] args) throws IOException {
        CarmichaelNumbers n = new CarmichaelNumbers();
        String currentLine = null;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            int v = Integer.parseInt(currentLine.trim());
            if (v == 0) {
                break;
            }
            if (n.isCarmichael(v)) {
                System.out.println(
                        "The number " + v + " is a Carmichael number.");
            } else {
                System.out.println(v + " is normal.");
            }
        }
    }
}
```

7.3 Euclid Problem

We just need to use the Extended Euclid Algorithm. The algorithm below is a direct translation of the pseudocode from [4].

126 ⟨*Euclid Problem* 126⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.Arrays;
import java.util.List;
import java.util.stream.Collectors;

class EuclidProblem {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static long[] euclid(long a, long b) {
        long s = 0;
        long old_s = 1;
        long t = 1;
        long old_t = 0;
        long r = b;
        long old_r = a;
        while (r != 0) {
            long quotient = old_r / r;
            long p = r;
            r = old_r - quotient * r;
            old_r = p;
            p = s;
            s = old_s - quotient * s;
            old_s = p;
            p = t;
            t = old_t - quotient * t;
            old_t = p;
        }
        return new long[] { old_s, old_t, old_r };
    }

    public static String toString(long[] arr) {
        return Arrays.stream(arr).mapToObj(String::valueOf)
                .collect(Collectors.joining(" "));
    }

    public static void main(String[] args) throws IOException {
        String currentLine = null;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
```

```
            System.out.println(toString(euclid(input.get(0), input.get(1))));
        }
    }
}
```

7.4 Factovisors

Obviously any number $m \leq n$ divides $n!$. That's an easy case. If $m > n$, however, we obtain the unique prime factorization of $m = p_1^{k_1} p_2^{k_2} \cdots p_n^{k_n}$. For each $p_i^{k_i}$ we count how many values $v \in [1, n]$ are there such that p_i divides v. There should be at least k_i such values. We check that this holds for every p_i from the unique prime factorization. If it doesn't, then m does not divide $n!$.

128 ⟨*Factovisors* 128⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

class Factovisors {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static boolean check(int n, int p, int k) {
        for (int i = p; i <= n && k > 0; i += p) {
            int m = i;
            while (m % p == 0) {
                m /= p;
                k--;
            }
        }
        return k <= 0;
    }

    public static boolean solve(int n, int m) {
        n = (n == 0) ? 1 : n;
        m = (m == 0) ? 1 : m;
        if (n >= m) {
            return true;
        }
        int k = 0;
        while (m % 2 == 0) {
            m /= 2;
            k++;
        }
        if (!check(n, 2, k)) {
            return false;
        }
        for (int i = 3; i <= Math.sqrt(m); i += 2) {
            k = 0;
            while (m % i == 0) {
                m /= i;
                k++;
            }
            if (!check(n, i, k)) {
```

```
                return false;
            }
        }
        return m <= n;
    }

    public static void main(String[] args) throws IOException {
        String currentLine = null;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().equals("")) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            boolean solution = solve(input.get(0), input.get(1));
            System.out.println(input.get(1) +
                    (solution ? " divides " : " does not divide ") +
                    input.get(0) + "!");
        }
    }
}
```

7.5 Summation of Four Primes

It's an easy task. We simply generate all the primes less than 10000000 using the Sieve of Eratosthenes. Once we have all these primes we can find the required sums by a simple search.

130 ⟨*Summation of Four Primes* 130⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.BitSet;
import java.util.Collections;
import java.util.List;
import java.util.stream.Collectors;

class SummationOfFourPrimes {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private final List<Integer> primes;
    private final static int MAX_PRIMES = 10_000_000;

    SummationOfFourPrimes() {
        BitSet bits = new BitSet(MAX_PRIMES);
        for (int i = 2; i < Math.sqrt(MAX_PRIMES); ++i) {
            if (!bits.get(i)) {
                int k = 0;
                int ii = i * i;
                int j = ii + k * i;
                while (j < MAX_PRIMES) {
                    bits.set(j);
                    k++;
                    j = ii + k * i;
                }
            }
        }
        primes = new ArrayList<>();
        for (int i = 2; i < bits.length(); ++i) {
            if (!bits.get(i)) {
                primes.add(i);
            }
        }
    }

    public int[] find(int a, int c, int[] current) {
        if (a == 0 && c == -1) {
            return current;
        } else if (a < 0 || c == -1) {
            return null;
        }
        int startIndex = Collections.binarySearch(primes, a);
        startIndex = (startIndex < 0) ? Math.abs(startIndex + 1) : startIndex;
        startIndex = primes.size() - 1 >= startIndex ? startIndex
                : primes.size() - 1;
```

```
            for (int i = startIndex; i >= 0; --i) {
                current[c] = primes.get(i);
                int[] result = find(a - primes.get(i), c - 1, current);
                if (result != null) {
                    return result;
                }
            }
            return null;
        }

        public static String toString(int[] arr) {
            return Arrays.stream(arr).mapToObj(String::valueOf)
                    .collect(Collectors.joining(" "));
        }

        public static void main(String[] args) throws IOException {
            SummationOfFourPrimes n = new SummationOfFourPrimes();
            String currentLine = null;
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().equals("")) {
                int[] res = n.find(Integer.parseInt(currentLine.trim()), 3,
                        new int[4]);
                System.out.println(res != null ? toString(res) : "Impossible.");
            }
        }
    }
```

7.6 Smith Numbers

To solve this task it will suffice to do the checks without doing anything very complicated.

132 ⟨*Smith Numbers* 132⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

class SmithNumbers {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int digitsSum(int value) {
        int sum = 0;
        while (value > 9) {
            sum += (value % 10);
            value /= 10;
        }
        return sum + value;
    }

    public static List<Integer> factor(int m) {
        List<Integer> factors = new ArrayList<>();
        if (m <= 2) {
            factors.add(m);
            return factors;
        }
        while (m % 2 == 0) {
            m /= 2;
            factors.add(2);
        }
        for (int i = 3; i <= Math.sqrt(m); i += 2) {
            while (m % i == 0) {
                m /= i;
                factors.add(i);
            }
        }
        if (m > 1) {
            factors.add(m);
        }
        return factors;
    }

    public static int find(int m) {
        int i = m + 1;
        while (true) {
            List<Integer> factors = factor(i);
            if (factors.size() > 1) {
                if (factors.stream().map(SmithNumbers::digitsSum).reduce(Integer::sum)
                        .get() == digitsSum(i)) {
                    return i;
```

```
                }
            }
            ++i;
        }
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            int v = Integer.parseInt(reader.readLine().trim());
            System.out.println(find(v));
        }
    }
}
```

7.7 Marbles

This task is a paraphrased linear Diophantine equation. We need to solve $ax + by = n$, where $n_1 = a$, $n_2 = b$ and n are as stated in the problem. Solving such an equation is easy and a thorough explanation on how to do this can be found in [6]. Once we've found the general solution, we limit ourselves to the positive solutions only by having $x = x_0 + i\frac{b}{d} > 0$ and $y = y_0 - i\frac{a}{d} > 0$, here $d = gcd(a, b)$; equivalently,

$$-\frac{x_0}{b} < i < \frac{y_0}{a}$$

Then, because we have a requirement to find the solution that gives the smallest cost, we check the solutions on both ends of that range and choose the one that is smaller.

134 ⟨*Marbles* 134⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

class Marbles {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static long[] euclid(long a, long b) {
        long s = 0;
        long old_s = 1;
        long t = 1;
        long old_t = 0;
        long r = b;
        long old_r = a;
        while (r != 0) {
            long quotient = old_r / r;
            long p = r;
            r = old_r - quotient * r;
            old_r = p;
            p = s;
            s = old_s - quotient * s;
            old_s = p;
            p = t;
            t = old_t - quotient * t;
            old_t = p;
        }
        return new long[] { old_s, old_t, old_r };
    }

    public static long[] diophant(long a, long b, long c) {
        long[] bezouts = euclid(a, b);
        if (c % bezouts[2] != 0) {
            return null;
        }
```

```
    long e = c / bezouts[2];
    return new long[] { e * bezouts[0], b / bezouts[2], e * bezouts[1],
            -a / bezouts[2] };
}

public static long[] min(long c1, long c2, long[] solution, long start,
        long end) {
    long[] min = new long[] { Long.MAX_VALUE, 0, 0 };
    for (long i = start; i <= end; ++i) {
        long x = solution[0] + i * solution[1];
        long y = solution[2] + i * solution[3];
        if (x < 0 || y < 0) {
            continue;
        }
        long cost = c1 * x + c2 * y;
        if (cost < min[0]) {
            min[0] = cost;
            min[1] = x;
            min[2] = y;
        }
    }
    return min;
}

public static long[] solve(long c1, long n1, long c2, long n2, long n) {
    long[] solution = diophant(n1, n2, n);
    if (solution == null) {
        return null;
    }
    long left = -(solution[0] / solution[1]);
    long right = (solution[2] / -solution[3]);
    long[] minLeft = min(c1, c2, solution, left, left + 1);
    long[] minRight = min(c1, c2, solution, right - 1, right);
    long[] min = minLeft[0] < minRight[0] ? minLeft : minRight;
    return min[0] < Long.MAX_VALUE ? min : null;
}

public static List<Long> parse(String line) {
    return stream(line.trim().split(" "))
            .filter(x -> !x.equals(""))
            .map(Long::parseLong)
            .collect(toList());
}

public static void main(String[] args) throws IOException {
    String currentLine = null;
    while ((currentLine = reader.readLine()) != null) {
        long c = Long.parseLong(currentLine);
        if (c == 0) {
            break;
        }
        List<Long> cn1 = parse(reader.readLine());
        List<Long> cn2 = parse(reader.readLine());
```

```
            long[] solution = solve(cn1.get(0), cn1.get(1), cn2.get(0),
                    cn2.get(1), c);
            System.out
                    .println(solution == null ? "failed"
                            : solution[1] + " " + solution[2]);
        }

    }
}
```

7.8 Repackaging

TBD.

8 Backtracking

8.1 Little Bishops

First, there's a caveat with this task. The program below is absolutely correct and fast. The online judge, however, was rejecting my initial implementation with a timeout. On my machine (a laptop) it ran all possible test cases in less than 100 milliseconds, so I was not sure why the judge didn't like it. Because the variations of the input aren't massive, I decided to make a lookup table with precomputed results. This time the judge accepted the solution. So, apparently, the judge runs some test cases multiple times and if we don't cache the results we will run out of time. Probably it's a good idea to cache the results wherenever possible on these backtracking tasks.

OK, so how to actually solve this? It's very similar to the eight queens problem. We should note that once we placed a bishop on a diagonal, no other bishops should be placed on that diagonal because obviously they will attack each other. This observation would allow us to write a considerably faster program than the naive approach but still would be unacceptably slow. We can speed up things by noting that if a bishop is on a black diagonal it won't attack any bishop on any white diagonal. We can get the number of placements by dividing the bishops into two groups: those that are on the black diagonals and those that are on the white diagonals.

Let's say we have k bishops and i bishops will be used on the black diagonals only, and $k-i$ bishops will be used on the white diagonals. Let's say $T_b(i)$ is the number of placements of i bishops on the black diagonals. Similarly, let's say $T_w(k-i)$ is the number of placements of $k-i$ bishops on the white diagonals. Then the number of combinations of k bishops where exactly i bishops are on the black diagonals and exactly $k-i$ bishops are on the white diagonals and none of them attack each other is $T_b(i) \times T_w(k-i)$. Therefore, to get the number of all possible placements we need to get a sum:

$$T(k) = \sum_{i=1}^{k} (T_w(i)T_b(k-i)) + T_w(k) + T_b(k)$$

$T_w(k)$ and $T_b(k)$ are for the cases when all k bishops are on the black diagonals and all k bishops are on the white diagonals.

In the constructor we will do the calculation as described above. The `getCount` method takes `s`, the start diagonal, which is either 0 or 1, and `k`, which is the number of bishops. The results of this method is the number of placements of k bishops on a board of size n, but only one the squares of the specific color. We can assume 0 means black squares and 1 means white squares (or vice versa, it doesn't matter).

At the same time we do a little optimization here. If it's not possible to place i (out of k) bishops on the black squares, then we don't even try to place the $k-i$ bishops on the white squares.

137 ⟨*Little Bishops* 137⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;
```

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;
import java.util.concurrent.atomic.AtomicInteger;

class LittleBishops {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private final int[][] attack;
    private final int n;
    private final int totalCount;

    public LittleBishops(int n, int k) {
        this.attack = new int[n][n];
        this.n = n;
        if (k > 0) {
            int total = 0;
            for (int i = 1; i < k; ++i) {
                int v = getCount(0, i);
                if (v > 0) {
                    total += v * getCount(1, k - i);
                }
            }
            this.totalCount = total + getCount(0, k) + getCount(1, k);
        } else {
            this.totalCount = 1;
        }
    }

    private int getCount(int s, int k) {
        AtomicInteger counter = new AtomicInteger(0);
        count(s, k, counter);
        return counter.get();
    }

    private void attack(int i, int j, int d) {
        while (i + 1 < n && j + 1 < n) {
            attack[++i][++j] += d;
        }
    }

    public void count(int s, int k, AtomicInteger counter) {
        ⟨8.1 Backtrack 140⟩
    }

    public static void main(String[] args) throws IOException {
        int[][] memo = new int[9][65];
        for (int i = 0; i < 9; ++i) {
            for (int j = 0; j < 65; ++j) {
                memo[i][j] = -1;
            }
```

```
        }

        String currentLine = null;
        while ((currentLine = reader.readLine()) != null) {
            List<Integer> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals("")).map(Integer::parseInt)
                    .collect(toList());
            if (input.get(0) == 0 && input.get(1) == 0) {
                break;
            }
            int i = input.get(0);
            int j = input.get(1);
            if (memo[i][j] == -1) {
                LittleBishops s = new LittleBishops(i, j);
                memo[i][j] = s.totalCount;
            }
            System.out.println(memo[i][j]);
        }
    }
}
```

A n-sized board has $2n - 1$ diagonals. The search will try each diagonal (of the same color) one by one and try to place exactly one bishop on the diagonal in one of the squares of it. We will assume the squares of the given board are enumerated from 0 to $n^2 - 1$, from the upper left to the right and down. For i-th diagonal we need to find the index of the initial square, this is saved in the `ii` variable. When $i < n$, the index of the initial square is simply i. If $i \geq n$ then the initial index of the squares will be $n(i - n + 2) - 1$.

The number of the squares in i-th diagonal is determined by the `squares` variable. If $i < n$ the number of the squares equals $i + 1$ (plus one because we count diagonal from 0). When $i \geq n$ then the number of the squares will be $2n - i - 1$. We iterate on these squares in a diagonal by simply adding $n - 1$ to the current index.

The array `attack` holds the number of attackers for each square. So if a corresponding element in the array is non-zero it means that square is under attack, and therefore we should not try to place any of the remaining bishops on it.

140 ⟨*8.1 Backtrack* 140⟩≡

```
if (k == 0) {
    counter.incrementAndGet();
    return;
}

for (int i = s; i <= 2 * n - 1; i += 2) {
    int ii = i >= n ? n * (i - n + 2) - 1 : i;
    int squares = i >= n ? 2 * n - i - 1 : i + 1;
    for (int j = ii, num = 0; num < squares; j += (n - 1), ++num) {
        int x = j / n;
        int y = j % n;
        if (attack[x][y] > 0) {
            continue;
        }
        attack(x, y, 1);
        count(i + 2, k - 1, counter);
        attack(x, y, -1);
    }
}
```

8.2 15-Puzzle Problem

This task, if it was smaller, could be solved using BFS algorithm. But 15-puzzle is quite troublesome to solve using BFS. However the task can be solved using the iterative deepening A* algorithm. Even though the time limit is 15 seconds, some cases may take much longer even with IDA* algorithm, but it seems that the online judge doesn't have such test cases.

First, let's sort out the unsolvable cases. There is a way to determine unsolvable cases and it is described in [7]: If the square containing the number i appears before n numbers that are less than i, then call it an inversion of order n, and denote it n_i. Let $N = \sum_{i=1}^{15} n_i$. Now let r_0 be the row number containing the empty square (counting from 1), then if $N + r_0$ value is odd, then the puzzle is unsolvable.

The IDA* algorithm requires a heuristic. Various heuristics may work with this task[4], but the simple Manhattan distance is sufficient for the judge's input cases.

We are going to use a 1D array to store the puzzle state, 0 denoting the empty square. We will introduce a class to represent a node in a graph, which will hold puzzle's state and some additional information such as whether this configuration is solvable or not, and the Manhattan distance.

Let's sort out input/output.

141 ⟨*15 Puzzle Problem* 141⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Deque;
import java.util.List;

public class The15PuzzleProblem {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    ⟨8.2. Node Class 143⟩
    ⟨8.2. IDA* Search 145⟩

    private static List<Integer> parseLine(String line) {
        return stream(line.trim().split(" "))
                .filter(x -> !x.equals(""))
                .map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            int[] puzzle = new int[16];
            int l = 0;
```

[4] See for example: `http://www.ic-net.or.jp/home/takaken/e/15pz/index.html`

```
            for (int j = 0; j < 4; ++j) {
                List<Integer> line = parseLine(reader.readLine().trim());
                for (int k = 0; k < 4; ++k) {
                    puzzle[l] = line.get(k);
                    l += 1;
                }
            }
            Node node = new Node(puzzle);
            if (!node.isSolvable()) {
                System.out.println("This puzzle is not solvable.");
            } else {
                System.out.println(solve(node));
            }
        }
    }
}
```

The `Node` class is quite straightforward. Its `isSolvable` method implements the check we discussed earlier. It also overrides `equals` because we will be checking if an instance of `Node` is in the stack or not in our implementation of the IDA* algorithm. The constructor takes a node, two indexes and a string. Two indexes define elements of the squares that need to be swapped, and the string argument is the direction of the move. Obviously, if the Manhattan distance is 0, then this configuration is the solution of the puzzle.

143 ⟨*8.2. Node Class* 143⟩≡

```
static class Node {
    final int[] node;
    final String path;
    final int d;
    final boolean isSolution;

    public Node(Node node, int i, int j, String p) {
        this.node = Arrays.copyOf(node.node, node.node.length);
        int tmp = this.node[i];
        this.node[i] = this.node[j];
        this.node[j] = tmp;
        this.path = node.path + p;
        this.d = distance(this.node);
        this.isSolution = d == 0;
    }

    private boolean isSolvable() {
        int sum = 0;
        int f = indexOfZero(this.node) / 4 + 1;
        for (int i = 0; i < this.node.length; ++i) {
            int c = 0;
            for (int j = i; j < this.node.length; ++j) {
                if (this.node[i] > 0 && this.node[j] > 0 &&
                        this.node[i] > this.node[j]) {
                    c++;
                }
            }
            sum += c;
        }
        return (sum + f) % 2 == 0;
    }

    private int distance(int[] node) {
        int d = 0;
        for (int k = 0; k < node.length; ++k) {
            if (node[k] == 0) {
                continue;
            }
            int r0 = (node[k] - 1) / 4;
            int c0 = (node[k] - 1) % 4;
            int r1 = (k) / 4;
            int c1 = (k) % 4;
            d += Math.abs(r0 - r1) + Math.abs(c0 - c1);
        }
        return d;
```

```
    }

    public Node(int[] puzzle) {
        this.node = puzzle;
        this.path = "";
        this.d = distance(this.node);
        this.isSolution = d == 0;
    }

    @Override
    public int hashCode() {
        final int prime = 31;
        int result = 1;
        result = prime * result + Arrays.hashCode(node);
        return result;
    }

    @Override
    public boolean equals(Object obj) {
        if (this == obj)
            return true;
        if (obj == null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        Node other = (Node) obj;
        if (!Arrays.equals(node, other.node))
            return false;
        return true;
    }
}
```

Now let's implement the search algorithm. This implementation is a direct translation of the pseudocode from [8]. The method `adjacent` simply returns all possible movements for the given `puzzle`.

145 ⟨*8.2. IDA* Search* 145⟩≡

```
private static final int FOUND = Integer.MIN_VALUE;

public static int indexOfZero(int[] puzzle) {
    for (int i = 0; i < puzzle.length; ++i) {
        if (puzzle[i] == 0) {
            return i;
        }
    }
    return 0;
}

public static List<Node> adjacent(Node puzzle) {
    List<Node> nodes = new ArrayList<>();
    int p = indexOfZero(puzzle.node);

    if (p + 1 <= 15 && p != 3 && p != 7 && p != 11 && p != 15) {
        nodes.add(new Node(puzzle, p, p + 1, "R"));
    }

    if (p - 4 >= 0) {
        nodes.add(new Node(puzzle, p, p - 4, "U"));
    }

    if (p - 1 >= 0 && p % 4 != 0) {
        nodes.add(new Node(puzzle, p, p - 1, "L"));
    }

    if (p + 4 <= 15) {
        nodes.add(new Node(puzzle, p, p + 4, "D"));
    }

    return nodes;
}

private static String solve(Node root) {
    int limit = root.d;
    Deque<Node> stack = new ArrayDeque<>();
    stack.push(root);
    while (true) {
        int newBound = search(stack, 0, limit);
        if (newBound == FOUND) {
            return stack.peek().path;
        }
        if (newBound == Integer.MAX_VALUE) {
            return "This puzzle is not solvable.";
        }
        limit = newBound;
    }
```

```
}

private static int search(Deque<Node> stack, int g, int limit) {
    Node node = stack.peek();
    int f = g + node.d;

    if (f > limit) {
        return f;
    }
    if (node.isSolution) {
        return FOUND;
    }
    int min = Integer.MAX_VALUE;
    for (Node adjNode : adjacent(node)) {
        if (!stack.contains(adjNode)) {
            stack.push(adjNode);
            int newBound = search(stack, g + 1, limit);
            if (newBound == FOUND) {
                return FOUND;
            }
            if (newBound < min) {
                min = newBound;
            }
            stack.pop();
        }
    }
    return min;
}
```

8.3 Queue

This task was quite challenging despite the fact it is rated as Level 2 task. It took me two attempts before I came up with a correct solution.

Let's agree that l is the number of people visible while looking from the beginning of the queue, and r is the number of people visible while looing from the end of the queue.

Now, two important observations. Notice that out of n persons there will always be the tallest person and that one will be visible from both ends of the queue. This is because all persons are of different height and if the person visible from both ends is not the tallest, then there will be a person that we will need to fit in between the persons and we won't be able to do that. So in every proper configuration the person visible from both ends is the tallest person. The second observation is that we don't need to arrange people into a queue as described in the problem in order to determine the required queue configurations. All we have to do is to find a configuration where all the people go in ascending order and such that the length of the arrangement is $l + r - 2$; let's call this a queue. (Minus 2 because both l and r parts "contain" the tallest person). Then the number of configurations as it's asked in the problem statement is equal to $\binom{l+r-2}{l-1}$. Now we will have some people that should be distributed in between these people in the queue. For example if we have three people in a queue p_1, p_2, p_3 and $p_1 < p_2 < p_3$ then the other people can be inserted between p_1 and p_2 (let's call it p_1 bucket), and p_2 and p_3 (p_2 bucket), provided that $p_2 - p_1 > 1$ and $p_3 - p_2 > 1$, and that those other people heights are less than p_1 and p_2 and so on. This is where the backtracking technique comes in: We try all possible placements of the people into such buckets. Note that we put the people into buckets in a methodic way and in ascending order. Once we have a placement in a bucket, d people within the bucket can be permuted in $d!$ ways. Since we have multiple buckets we just multiple these factorials to get the final answer. Then we find another distribution in the buckets and repeat the calculation while accumulating the value. Once distributions in the buckets are exhausted for this queue, we find a nother queue and repeat the process.[5]

147 ⟨*Queue* 147⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;
import java.util.stream.LongStream;

public class Queue {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private final int n;
    private final int k;
    private final int[] buckets;
    private final int[] queue;
    private final int[] values;
    private final boolean[] skip;
```

[5] Incidentally, there's a very short and efficient solution using dynamic programming technique. See `https://github.com/morris821028/UVa/blob/master/volume101/10128-Queue.cpp`.

```
private long total;
private long counter;

private static final long[] factorial = LongStream.rangeClosed(0, 13)
        .map(f -> f == 0 ? 1
                : LongStream.rangeClosed(1, f).reduce(1L,
                        (x, y) -> x * y))
        .toArray();

private static final Long[][][] cache = new Long[13 + 1][13 + 1][13 + 1];

public Queue(int n, int l, int r) {
    this.n = n;
    this.k = l - 1;
    buckets = new int[n + 1];
    skip = new boolean[n + 1];
    queue = new int[l + r - 2];
    values = new int[n - (l + r - 2) - 1 < 0 ? 0 : n - (l + r - 2) - 1];
    skip[n] = true;
    if (cache[n][l][r] != null) {
        total = cache[n][l][r];
    } else {
        select(0);
        cache[n][l][r] = total;
    }
}

private void select(int pos) {
    if (pos == queue.length) {
        for (int i = 1, j = 0; i <= n; ++i) {
            if (!skip[i]) {
                values[j++] = i;
            }
        }
        counter = 0;
        count(0);
        int nn = queue.length;
        total += counter * factorial[nn] /
                (factorial[k] * factorial[nn - k]);
        return;
    }

    int start = pos - 1 >= 0 ? queue[pos - 1] + 1 : 1;
    for (int i = start; i < n; ++i) {
        queue[pos] = i;
        skip[i] = true;
        select(pos + 1);
        skip[i] = false;
    }
}

private void count(int pos) {
    if (pos == values.length) {
```

```
            long permutations = 1;
            for (int i = 0; i < buckets.length; ++i) {
                permutations *= factorial[buckets[i]];
            }
            counter += permutations;
            return;
        }

        for (int bucket : queue) {
            if (bucket > values[pos]) {
                buckets[bucket]++;
                count(pos + 1);
                buckets[bucket]--;
            }
        }
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            List<Integer> line = stream(reader.readLine().split(" "))
                    .filter(x -> !x.equals(""))
                    .map(Integer::parseInt)
                    .collect(toList());
            int l = line.get(1);
            int r = line.get(2);
            if (l + r - 1 > 13) {
                System.out.println(0);
            } else {
                Queue q = new Queue(line.get(0), l, r);
                System.out.println(q.total);
            }
        }
    }

}
```

8.4 Servicing Stations

There's no alternative other than just try all possible configurations as this task is known to be an NP problem[6]. So we will have to exhaustively try various configurations until we arrive at one that satisfies the problem requirements. We start with just one node, and if we can't find a solution we try two nodes, then three nodes, and so on, until we find a solution.

One important thing to do is to find the graphs, because some nodes can be disconnected from the others. Once we found these graphs we then find the dominating sets per each graph and sum up the sizes of those dominating sets to get the final result.

We will be using bit arrays per each node. Each such bit array will define which nodes are connected to the node. A bit at index i tells whether node i is connected to this node. At first I chose to use `boolean` arrays but then switched to Java's `BitSet`. However, `BitSet`'s performance wasn't satisfactory, whether `boolean` arrays required a number of ugly cycles to be coded here and there. Therefore I chose to use `long` type to hold my bits and use bit operations to set and unset bits in it.

The template for the program will be as follows.

150 ⟨*Servicing Stations* 150⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.List;

public class ServicingStations {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    ⟨8.4 Bits Class 151⟩
    ⟨8.4 Fields 152b⟩

    public ServicingStations(Bits[] cities) throws Exception {
        ⟨8.4 Constuctor 153a⟩
    }

    ⟨8.4 Graphs 152a⟩
    ⟨8.4 Backtrack 153b⟩

    private static List<Integer> parseLine(String line) {
        return stream(line.split(" "))
                .filter(x -> !x.equals(""))
                .map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws Exception {
        ⟨8.4 Input/Output 154b⟩
```

[6] See "dominating set"

```
    }

}
```

Let's add the `Bits` class first. It's just a wrapper around `long` value that sets and reads bits of it:

151 ⟨*8.4 Bits Class* 151⟩≡

```
static class Bits {
    long v = 0L;

    boolean get(int b) {
        return (this.v & (1L << b)) != 0;
    }

    void set(int b) {
        this.v |= (1L << b);
    }
}
```

Our input graph will be represented as an array of `Bits`, `Bits[] cities`, where an index i in the array `cities` is the node i, and the corresponding array element of type `Bits`, e.g. `cities[i]`, is its adjacency bit array.

The input graph may be actually a set of graphs which are completely isolated from each other. We would like to decompose the input graph to such isolated graphs first.

To do that we are going to use the depth-first search:

152a ⟨*8.4 Graphs* 152a⟩≡

```
private List<Bits> graphs(Bits[] cities) {
    List<Bits> graphs = new ArrayList<>();
    Bits all = new Bits();
    boolean finished = false;
    while (!finished) {
        finished = true;
        for (int i = 0; i < cities.length; ++i) {
            if (!all.get(i)) {
                finished = false;
                Bits graph = bfs(i, cities);
                graphs.add(graph);
                all.v |= graph.v;
                break;
            }
        }
    }
    return graphs;
}

private Bits bfs(int s, Bits[] cities) {
    Deque<Integer> next = new ArrayDeque<>();
    Bits visited = new Bits();
    next.add(s);
    while (!next.isEmpty()) {
        int n = next.pop();
        for (int i = 0; i < cities.length; ++i) {
            if (cities[n].get(i) && !visited.get(i)) {
                next.add(i);
            }
        }
        visited.set(n);
    }
    return visited;
}
```

Before implementing the constructor, let's introduce some fields that we are going to need:

152b ⟨*8.4 Fields* 152b⟩≡

```
private final Bits[] cities;
private final Bits[] stack;
private int count;
private Bits currentGraph;
```

`count` field will hold the minimum number we are seeking; `cities` will be the input array, `stack` and `currentGraph` will be used in our backtracking.

Let's implement the constructor:

153a ⟨*8.4 Constuctor* 153a⟩≡

```
count = 0;
this.stack = new Bits[40];
for (int i = 0; i < this.stack.length; ++i) {
    this.stack[i] = new Bits();
}
this.cities = cities;
for (Bits graph : graphs(this.cities)) {
    currentGraph = graph;
    for (int i = 0; i < cities.length; ++i) {
        ⟨8.4 Invoke Backtrack 154a⟩
    }
}
```

Now let's get to the backtrack. It's going to be a classic backtrack. When we try a new node, we first check if that node is within our current graph, and that adding that node gives us any improvement or not. If it does not, then it moves on to next node. If it does improve, it recurses. Once we've tried the predefined number of nodes and we found a configuration, we check if that configuration covered all the nodes in the graph, and if it did, it means we found a solution. If not, we backtrack. (Note how we are using the bitwise operations on the adjacency bits and the `stack`. We could have done this without the `stack`. But the reason we have the `stack` is to avoid repeated creation of `Bits` objects.)

153b ⟨*8.4 Backtrack* 153b⟩≡

```
private boolean backtrack(int count, int pos, int depth) throws Exception {
    if (count == 0) {
        return ((stack[depth].v &
                currentGraph.v) == currentGraph.v);
    }
    for (int i = pos; i < cities.length; ++i) {
        if (!currentGraph.get(i) && !stack[depth].get(i)) {
            continue;
        }

        stack[depth + 1].v = stack[depth].v | cities[i].v;
        if (stack[depth + 1].v == stack[depth].v) {
            continue;
        }

        if (backtrack(count - 1, i + 1, depth + 1)) {
            return true;
        }
    }
    return false;
}
```

Now we need to invoke this from the constructor:

154a ⟨*8.4 Invoke Backtrack* 154a⟩≡

```
if (backtrack(i, 0, 0)) {
    count += i;
    break;
}
```

Finally we can implement the input/output:

154b ⟨*8.4 Input/Output* 154b⟩≡

```
String currentLine;
while ((currentLine = reader.readLine()) != null) {
    if (currentLine.trim().isEmpty()) {
        continue;
    }
    List<Integer> nm = parseLine(currentLine);
    int n = nm.get(0);
    int m = nm.get(1);
    if (n == 0 && m == 0) {
        break;
    }
    Bits[] cities = new Bits[n];
    for (int i = 0; i < cities.length; ++i) {
        cities[i] = new Bits();
        cities[i].set(i);
    }
    while (m > 0) {
        if ((currentLine = reader.readLine()).trim().isEmpty()) {
            continue;
        }
        List<Integer> fromTo = parseLine(currentLine.trim());
        cities[fromTo.get(0) - 1].set(fromTo.get(1) - 1);
        cities[fromTo.get(1) - 1].set(fromTo.get(0) - 1);
        m--;
    }
    System.out.println(new ServicingStations(cities).count);
}
```

8.5 Tug Of War

I have no idea how to solve this task using backtrack and it seems neither do the authors of the book.

The task can be solved using dynamic programming. Let's say the input array is A of size n. Let S be the sum of the elements of A. Let's also assume we have an array M of boolean values of size $S+1$. The i-th item in this array means whether there's a sum that is equal to i using some elements of A or not. So $M[0] = true$, because we can always have a sum that is equal to 0 by taking none of the elements of A. Let's take the first element from A, $A[0]$. Now we will have $M[0] = true$ and $M[0 + A[0]] = true$, because we can extend our empty set by adding $A[0]$ into it. Now, generally, when we are processing $A[i]$, then at every position j in M where $M[j] = true$ we can extend it by adding the $A[i]$ element by setting $M[j + A[i]]$ to $true$. Once we have finished, we can check if $M[S/2]$ is true and if so, then there's a subset of A that sums to $S/2$.

That answer is absolutely correct, but it doesn't take into account the size of the subsets! We need to somehow modify this algorithm so that it also keeps track of the number of elements of the subsets. Then we could choose the one that matches the requirement, that is such that the size of the subset is the half of the A exactly or differs by no more than 1 and that the sum is as close to the $S/2$ value as possible.

Let's modify our M array to hold integers instead of boolean values (it is of `long` type, because we may have up to 50 elements in our subset and long is 64-bit in Java). Then we shift the value at $M[i]$ to the left by one each time we extend the subset, and use the bitwise OR to combine the values. Once finished, we will have values at $M[i]$ such that bits in this value will define the sizes of the subsets. So the bit at j-th position in the value $M[i]$ means there's a subset of size j that sums to i. To get the required answer, we start with the value at $M[S/2]$ and check if the bit at $n/2$-th position is set or not if n is even, and we check both $n/2$-th and $n/2+1$-th bits if n is odd. If the bit is set, then $S/2$ is the answer. Otherwise we check $M[S/2-1]$, $M[S/2-2]$, and so on until we find a match.

To arrive at this solution a tutorial [12] was of great help. The bit trick was purloined from [13].

155 ⟨*Tug Of War* 155⟩≡

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.Arrays;

public class TugOfWar {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int[] solve(int[] arr) {
        int n = arr.length;
        int sum = Arrays.stream(arr).sum();
        long[] m = new long[sum + 1];
        m[0] = 1;
        for (int i = 0; i < n; ++i) {
            for (int j = sum - arr[i]; j >= 0; --j) {
                m[j + arr[i]] |= m[j] << 1L;
            }
        }
```

```
        for (int i = sum / 2; i >= 0; --i) {
            if (n % 2 == 0) {
                if ((m[i] & 1L << (n / 2)) > 0) {
                    return new int[] { i, sum - i };
                }
            } else {
                if ((m[i] & 1L << (n / 2)) > 0 ||
                        (m[i] & 1L << (n / 2 + 1)) > 0) {
                    return new int[] { i, sum - i };
                }
            }
        }

        return null;
    }

    public static void main(String[] args) throws Exception {
        int n = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            int m = Integer.parseInt(reader.readLine().trim());
            int[] arr = new int[m];
            for (int j = 0; j < m; ++j) {
                arr[j] = Integer.parseInt(reader.readLine().trim());
            }
            int[] s = solve(arr);
            System.out.println(s[0] + " " + s[1]);
            if (i < n - 1) {
                System.out.println();
                reader.readLine();
            }
        }
    }

}
```

8.6 Garden of Eden

Because the given state in the input may be a "Garden of Eden" state, i.e. no other state leads to it using the given automaton, all we need to do is to try to proove such a previous state doesn't exist. If it does, then this state is not "Garden of Eden" state.

To solve the task a simple backtracking without any special tricks will do. We just use the transition table for the given automaton while trying to reconstruct the previous state.

157 ⟨*Garden of Eden* 157⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.List;

public class GardenOfEden {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    static final boolean[][] trans = new boolean[][] {
            { false, false, false },
            { false, false, true },
            { false, true, false },
            { false, true, true },
            { true, false, false },
            { true, false, true },
            { true, true, false },
            { true, true, true }
    };

    final boolean[] auto;
    final boolean[] curr;
    final boolean[] prev;
    boolean reachable;

    GardenOfEden(boolean[] auto, boolean[] curr) {
        this.auto = auto;
        this.curr = curr;
        this.prev = new boolean[curr.length];
        for (int j = 0; j < auto.length; ++j) {
            if (auto[j] == curr[0]) {
                prev[prev.length - 1] = trans[j][0];
                prev[0] = trans[j][1];
                prev[1] = trans[j][2];
                reachable = backtrack(1);
                if (reachable) {
                    break;
                }
            }
        }
    }
```

```
    boolean backtrack(int pos) {
        for (int j = 0; j < auto.length; ++j) {
            if (auto[j] != curr[pos]) {
                continue;
            }
            if (prev[pos - 1] == trans[j][0] &&
                    prev[pos] == trans[j][1]) {
                if (pos == curr.length - 1) {
                    if (prev[0] == trans[j][2]) {
                        return true;
                    }
                } else if (pos + 1 == curr.length - 1) {
                    if (prev[pos + 1] == trans[j][2]) {
                        if (backtrack(pos + 1)) {
                            return true;
                        }
                    }
                } else {
                    prev[pos + 1] = trans[j][2];
                    if (backtrack(pos + 1)) {
                        return true;
                    }
                }
            }
        }
        return false;
    }

    public static void main(String[] args) throws Exception {
        String currentLine;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().isEmpty()) {
            List<String> input = stream(currentLine.trim().split(" "))
                    .filter(x -> !x.equals(""))
                    .collect(toList());
            boolean[] automaton = new boolean[8];
            String id = Integer.toBinaryString(Integer.parseInt(input.get(0)));
            for (int i = id.length() - 1, j = 0; i >= 0; --i, ++j) {
                automaton[j] = id.charAt(i) == '1';
            }
            boolean[] state = new boolean[input.get(2).length()];
            for (int i = 0; i < state.length; ++i) {
                state[i] = input.get(2).charAt(i) == '1';
            }
            System.out
                    .println(new GardenOfEden(automaton, state).reachable ? "REACHABLE"
                            : "GARDEN OF EDEN");
        }
    }

}
```

8.7 Colours Hash

It seems that this can be solved with a breadth-first search. And it can, but it won't be very quick. There's a trick, however, that works well with this task: Using the BFS we can find all the reachable states (and how to get there) starting from the final state up to a certain depth level. Having the depth of 8 is enough for this task. One we have this precomputed set, we start doing the BFS from the input state, but we again limit it to 8 levels of depth. (Note that the task's requirement is to give up searching if the depth is more than 16.) If we can reach any state in the precomputed set from an input state, then we know there's a path, and we can reconstruct it.

159 ⟨*Colours Hash* 159⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class ColoursHash {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    ⟨8.7 Fields 162⟩

    ⟨8.7 State Class 161⟩

    ⟨8.7 Precompute 163b⟩

    ⟨8.7 Find 165⟩

    public static void main(String[] args) throws Exception {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            String line = reader.readLine();
            List<Integer> input = stream(line.trim().split(" "))
                    .filter(x -> !x.equals(""))
                    .map(Integer::parseInt)
                    .collect(toList());
            State state = new State(input);
            if (state.equals(finalState)) {
                System.out.println("PUZZLE ALREADY SOLVED");
            } else {
                String solution = find(state);
                if (solution != null) {
```

```
                    System.out.println(solution);
                } else {
                    System.out.println("NO SOLUTION WAS FOUND IN 16 STEPS");
                }
            }
        }

    }
}
```

The `State` class is going to be fairly simple. It will hold the state as an array of integers. The constructor will allow to create a new state from the given state by rotating the wheels. I have chosen to use `LinkedList` here as it has convenient methods to do the rotations. I could have done this using `System.arraycopy` too, but decided for something that is more readable (that is `LinkedList`). The `equals` and `hashCode` are necessary as we will be using maps and sets while doing the BFS.

161 ⟨*8.7 State Class* 161⟩≡

```
static class State {
    final int[] curr;

    public State(State state, int rotation) {
        this(state.curr, rotation);
    }

    public State(List<Integer> state) {
        curr = new int[21];
        for (int i = 0; i < curr.length; ++i) {
            curr[i] = state.get(i);
        }
    }

    public State(int[] state, int rotation) {
        this.curr = new int[21];

        final LinkedList<Integer> left = new LinkedList<>();
        final LinkedList<Integer> right = new LinkedList<>();
        final LinkedList<Integer> middle = new LinkedList<>();

        for (int i = 0; i < 12; ++i) {
            if (i < 9) {
                left.add(state[i]);
                right.add(state[i + 12]);
            } else {
                middle.add(state[i]);
            }
        }

        if (rotation == 1) {
            middle.addFirst(left.removeLast());
            middle.addFirst(left.removeLast());
            left.addFirst(middle.removeLast());
            left.addFirst(middle.removeLast());
        } else if (rotation == 2) {
            middle.addLast(right.removeFirst());
            middle.addLast(right.removeFirst());
            right.addLast(middle.removeFirst());
            right.addLast(middle.removeFirst());
        } else if (rotation == 3) {
            middle.addLast(left.removeFirst());
            middle.addLast(left.removeFirst());
            left.addLast(middle.removeFirst());
            left.addLast(middle.removeFirst());
```

```
            } else if (rotation == 4) {
                middle.addFirst(right.removeLast());
                middle.addFirst(right.removeLast());
                right.addFirst(middle.removeLast());
                right.addFirst(middle.removeLast());
            }

            for (int i = 0; i < left.size(); ++i) {
                curr[i] = left.get(i);
            }
            for (int i = 0; i < middle.size(); ++i) {
                curr[9 + i] = middle.get(i);
            }
            for (int i = 0; i < right.size(); ++i) {
                curr[12 + i] = right.get(i);
            }
        }

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + Arrays.hashCode(curr);
            return result;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            State other = (State) obj;
            if (!Arrays.equals(curr, other.curr))
                return false;
            return true;
        }
    }
```

Now we can define the initial state:

162 ⟨*8.7 Fields* 162⟩≡

```
    private static State finalState = new State(
            new int[] { 0, 3, 4, 3, 0, 5, 6, 5, 0, 1,
                    2, 1, 0, 7, 8, 7, 0, 9, 10, 9, 0 },
            0);
```

Let's write the initialization of the `precomputed` set, the set of the states that are reachable from the `finalState`. First, let's define `moves` array. Because we are searching from the `finalState` to a reachable state, we will use this array to know the "reversed" moves, because we are looking for a path from a rechable state to the final state and not the other way round.

163a ⟨*8.7 Fields* 162⟩+≡

```
final static int[] moves = new int[] { 3, 4, 1, 2 };
final static Map<State, String> precomputed = init();
```

We will need to keep track of the depths and moves sequences while doing BFS , so let's define a helper class for that:

163b ⟨*8.7 Precompute* 163b⟩≡

```
static class Tuple {
    int depth;
    String solution;

    public Tuple(int depth, String solution) {
        this.depth = depth;
        this.solution = solution;
    }
}
```

Now let's implement the `init()` method that precomputes the reachable states set. The `generate` method is a simple BFS implementation, the only difference is that it terminates once it has reached the depth of 8.

164 ⟨*8.7 Precompute* 163b⟩+≡

```
static Map<State, String> init() {
    Map<State, String> solutions = new HashMap<>();
    generate(solutions, finalState);
    return solutions;
}

private static void generate(Map<State, String> solutions, State node) {
    Deque<State> next = new ArrayDeque<>();
    Map<State, Tuple> depths = new HashMap<>();
    Set<State> visited = new HashSet<>();
    next.addLast(node);
    depths.put(node, new Tuple(0, ""));
    visited.add(node);

    while (!next.isEmpty()) {
        State n = next.pop();
        Tuple tuple = depths.get(n);
        if (tuple.depth > 8) {
            continue;
        }
        String solution = tuple.solution;
        for (int i = 1; i <= 4; ++i) {
            State adj = new State(n, moves[i - 1]);
            if (!visited.contains(adj)) {
                next.add(adj);
                depths.putIfAbsent(adj,
                        new Tuple(tuple.depth + 1, i + solution));
                solutions.putIfAbsent(adj, i + solution);
            }
        }
        visited.add(n);
    }
}
```

Now we can implement the `find` method, that will do a similar BFS, but will check whether we have reached any state in `precomputed` set. If we have, then we have found the solution!

165 ⟨*8.7 Find* 165⟩≡

```
private static String find(State node) {
    Deque<State> next = new ArrayDeque<>();
    Map<State, Tuple> depths = new HashMap<>();
    Set<State> visited = new HashSet<>();
    next.addLast(node);
    depths.put(node, new Tuple(0, ""));
    visited.add(node);

    while (!next.isEmpty()) {
        State n = next.pop();
        Tuple tuple = depths.get(n);
        String solution = tuple.solution;
        if (tuple.depth > 8) {
            continue;
        }
        if (ColoursHash.precomputed.containsKey(n)) {
            return solution + ColoursHash.precomputed.get(n);
        }
        for (int i = 1; i <= 4; ++i) {
            State adj = new State(n, i);
            if (!visited.contains(adj)) {
                next.add(adj);
                depths.putIfAbsent(adj,
                        new Tuple(tuple.depth + 1, solution + i));
            }
        }
        visited.add(n);
    }

    return null;
}
```

8.8 Bigger Square Please...

Well, this task is impossible to solve in under 3 seconds as the judge expects us to solve. So the only sensible way is to write a meta-program that generates a program than then gets accepted by the judge. The generated program got to be a simple lookup type of a program that uses pre-computed table of solutions. This is possible as the range for the input is small.

But even a meta-program will need to be written with some heuristics here and there to allow for better run time. As this meta-program won't be run by the judge we can use all available tools. Specifically, we will use all the cores of the CPU to run computations in parallel as much as possible.

Just a few notes on the problem itself. It's a very well known problem; the sequence of minimal number of smaller integer-sided squares that tile an $n \times n$ square is registered in the On-Line Encyclopedia of Integer Sequences as A018835 sequence[14].

Obviously, any square that has a side of an even length is trivially divisible into four squares of equal size, and you can't do better than that. So at least half of the problem is now solved! Any square that has a side length that is a multiple of 3 (but not 5) is also trivially divisible. So we only need to use backtracking for the rest of the values in $2 \leq p \leq 50$ interval.

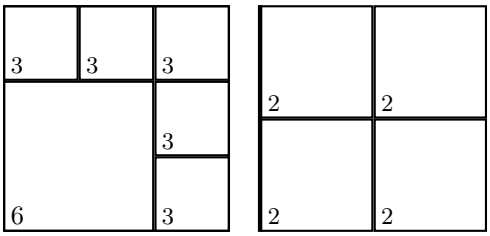

Figure 1: Example for 9×9 and 4×4.

Let's outline the general approach to solve the task. First, it's always very beneficial to try a few small cases. This allows (with some luck) to notice patterns. In our case this may suggest heuristics. There's no guarantee that these heuristics will lead to the optimal solutions, but still worth trying. (Do we have any other choice here?)

Once a few small cases have been computed you will notice that all the solutions for the squares of odd side lengths start with a similar pattern: Namely the biggest square is usually of $n/2 + 1$ and there are two smaller squares of size $n/2$ by its sides; everything else gets packed in the remaining part. This observation will reduce our search space.

Another thing to notice is that for every prime n the number of squares seems to grow monotonically. This also allows for the search space reduction.

OK, once we have put those three squares we have the remaining part to fill out. We know the unfilled area so we generate all the possible square combinations of different sizes that sum up to the unfilled area. Then we try to tile that area with these squares. In both cases we use backtracking, that is generating the candidate square sets, and trying to tile them into the unfilled area.

Now let's start writing code.

166 ⟨*Bigger Square Please Offline* 166⟩≡

```
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
```

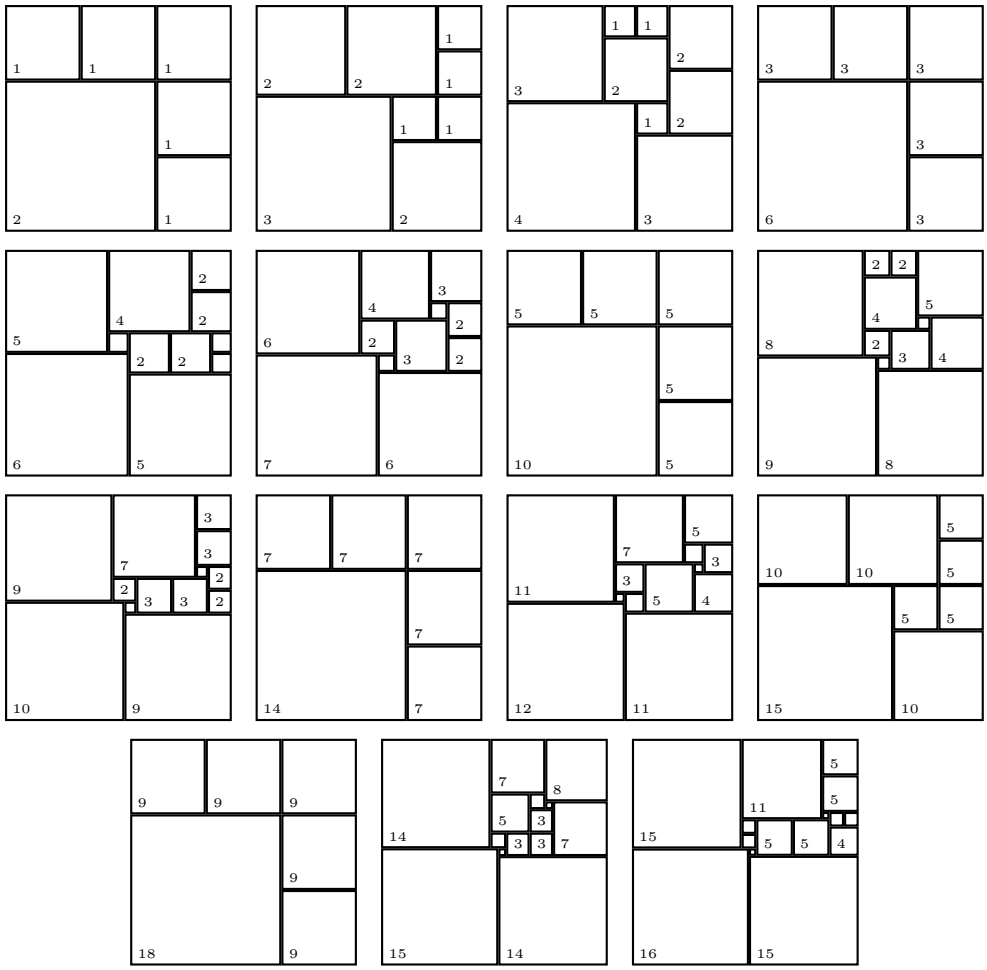

Figure 2: Solutions for odd side lengths $3 \leq n \leq 31$.

```
import java.util.Collections;
import java.util.Deque;
import java.util.List;
import java.util.Map;
import java.util.concurrent.Callable;
import java.util.concurrent.CompletionService;
import java.util.concurrent.ExecutionException;
import java.util.concurrent.ExecutorCompletionService;
import java.util.concurrent.ExecutorService;
import java.util.concurrent.Executors;
import java.util.concurrent.Future;
import java.util.function.Function;
import java.util.stream.Collectors;

public class BiggerSquarePleaseOffline {
    ⟨8.8 Classes 169⟩
    ⟨8.8 Methods 175a⟩
    ⟨8.8 Main 178⟩
}
```

First, we start off with the generator of the square side lengths. It uses backtracking to generate sets (multisets in fact) of the square sizes whose areas sum up to a specified value.

`minSolutionSize` parameter is used to skip the sets that are too small to consider. This is useful, because we assumed earlier that the minimal number of square tiles for the squares whose side lengths are prime values grows monotonically. Also, note `HEURISTIC_DEPTH` constant: We assume that no solution will be bigger than 15 squares. (Well, at least for our task's input range $3 \leq n \leq 50$. We know this by looking at A018835.)

Another heuristic is to filter out those sets that have 4 or more squares of the same size. It seems that the optimal solutions never have more than 4 squares of the same size in them, so we can skip such sets. (Again, this seem to be true only for $n < 50$).

The backtracking itself here is quite straightforward. One the instance of this class is constructed, all the sets will be in `squareSizes` field.

169 ⟨*8.8 Classes* 169⟩≡

```
static class SquareSideLengthsGenerator {
    private static final int HEURISTIC_DEPTH = 15;
    private Deque<Integer> currSquareSizes = new ArrayDeque<>();
    private List<List<Integer>> squareSizes = new ArrayList<>();

    SquareSideLengthsGenerator(int startSquareSize,
            int targetArea,
            int minSolutionSize) {
        search(startSquareSize, 0, targetArea, HEURISTIC_DEPTH);
        squareSizes = squareSizes.stream()
                .filter(SquareSideLengthsGenerator::heuristicFilter)
                .sorted((x, y) -> Integer.compare(x.size(), y.size()))
                .filter(x -> x.size() >= minSolutionSize)
                .collect(Collectors.toList());
    }

    private static boolean heuristicFilter(List<Integer> s) {
        Map<Integer, Long> result = s.stream().collect(
                Collectors.groupingBy(
                        Function.identity(), Collectors.counting()));
        return Collections.max(result.values()) <= 4;
    }

    private void search(int startSquareSize, int currArea, int targetArea,
            int maxLen) {
        if (currArea == targetArea) {
            squareSizes.add(new ArrayList<>(currSquareSizes));
            return;
        }
        for (int i = startSquareSize; i >= 1; --i) {
            if (currArea + i * i <= targetArea &&
                    currSquareSizes.size() <= maxLen) {
                currSquareSizes.addLast(i);
                search(i, currArea + i * i, targetArea, maxLen);
                currSquareSizes.removeLast();
            }
        }
    }
}
```

Now we will write another backtracking code that will be trying to tile the squares generated by the `SquareSideLengthsGenerator`.

It uses a boolean array to keep track of tiling. I could have used some square overlapping conditions instead of an array, but because the input is small, $2 \leq n \leq 50$, I decided to use a boolean array. Placing or removing a square in the bigger square is simply setting corresponding cells to `true` or `false`, and checking if a square can be inserted is as simple as checking if none of the cells inside of that square are set to `true`.

Notice `HEURISTIC_THRESHOLD` constant. It's a heuristic: If we've been trying to tile this set for too long, we halt it and conclude it can't be tiled. It seems that having it set to 25K doesn't affect correctness of the answers, but setting it to some smaller values may give you non-optimal answers.

171 ⟨*8.8 Classes* 169⟩+≡

```
static class SolutionFinder {
    private static final int HEURISTIC_THRESHOLD = 25000;
    private final int n;
    private final boolean[][] square;
    private final List<Integer> candidate;
    private final Deque<int[]> solution = new ArrayDeque<>();
    private final boolean solutionFound;
    private boolean haltSearch;
    private int heuristicCounter;

    SolutionFinder(Deque<int[]> initial,
            List<Integer> candidateSquareSideSizes,
            int n) {
        this.n = n;
        this.candidate = candidateSquareSideSizes;
        this.square = new boolean[n][n];
        initial.forEach(x -> {
            this.solution.add(x);
            set(x, true);
        });
        solutionFound = find(0);
    }

    private boolean isInsertable(int[] s) {
        for (int i = s[0]; i <= s[0] + s[2] - 1; ++i) {
            for (int j = s[1]; j <= s[1] + s[2] - 1; ++j) {
                if (j > n - 1 || i > n - 1 || square[i][j]) {
                    return false;
                }
            }
        }
        return true;
    }

    private void set(int[] s, boolean f) {
        for (int i = s[0]; i <= s[0] + s[2] - 1; ++i) {
            for (int j = s[1]; j <= s[1] + s[2] - 1; ++j) {
                square[i][j] = f;
            }
        }
```

```
    }

    private boolean find(int pos) {
        if (pos == candidate.size()) {
            return true;
        }

        if (++heuristicCounter > HEURISTIC_THRESHOLD) {
            haltSearch = true;
            return false;
        }

        for (int i = 0; i < n; ++i) {
            for (int j = 0; j < n; ++j) {
                if (square[i][j]) {
                    continue;
                }
                int[] s = new int[] { i, j, candidate.get(pos) };
                if (isInsertable(s)) {
                    set(s, true);
                    solution.addLast(s);
                    if (find(pos + 1)) {
                        return true;
                    }
                    if (haltSearch) {
                        return false;
                    }
                    solution.removeLast();
                    set(s, false);
                }
            }
        }

        return false;
    }
}
```

We will be running many tasks in parallel so let's add a few helper classes for that. `SolutionFinderTaskResult` has `index` field. This field denotes its position in the list of sets of square sizes generated by `SquareSideLengthsGenerator`. If we found a solution that has `found` set to `true`, then if all the results that have index less than this index and `found` set to false, then it means this solution is an optimal solution.

173 ⟨*8.8 Classes* 169⟩+≡

```
static class SolutionFinderTaskResult {
    final Deque<int[]> solution;
    final boolean found;
    final int index;

    public SolutionFinderTaskResult(boolean found, Deque<int[]> solution,
            int index) {
        this.solution = solution;
        this.found = found;
        this.index = index;
    }
}

static class SolutionFinderTask
        implements Callable<SolutionFinderTaskResult> {
    final int index;
    final Deque<int[]> initial;
    final List<Integer> sizes;
    final int n;

    SolutionFinderTask(int index, Deque<int[]> initial, List<Integer> sizes,
            int n) {
        this.index = index;
        this.initial = initial;
        this.sizes = sizes;
        this.n = n;
    }

    @Override
    public SolutionFinderTaskResult call() throws Exception {
        SolutionFinder finder = new SolutionFinder(initial, sizes, n);
        return new SolutionFinderTaskResult(finder.solutionFound,
                finder.solution,
                index);
    }
}
```

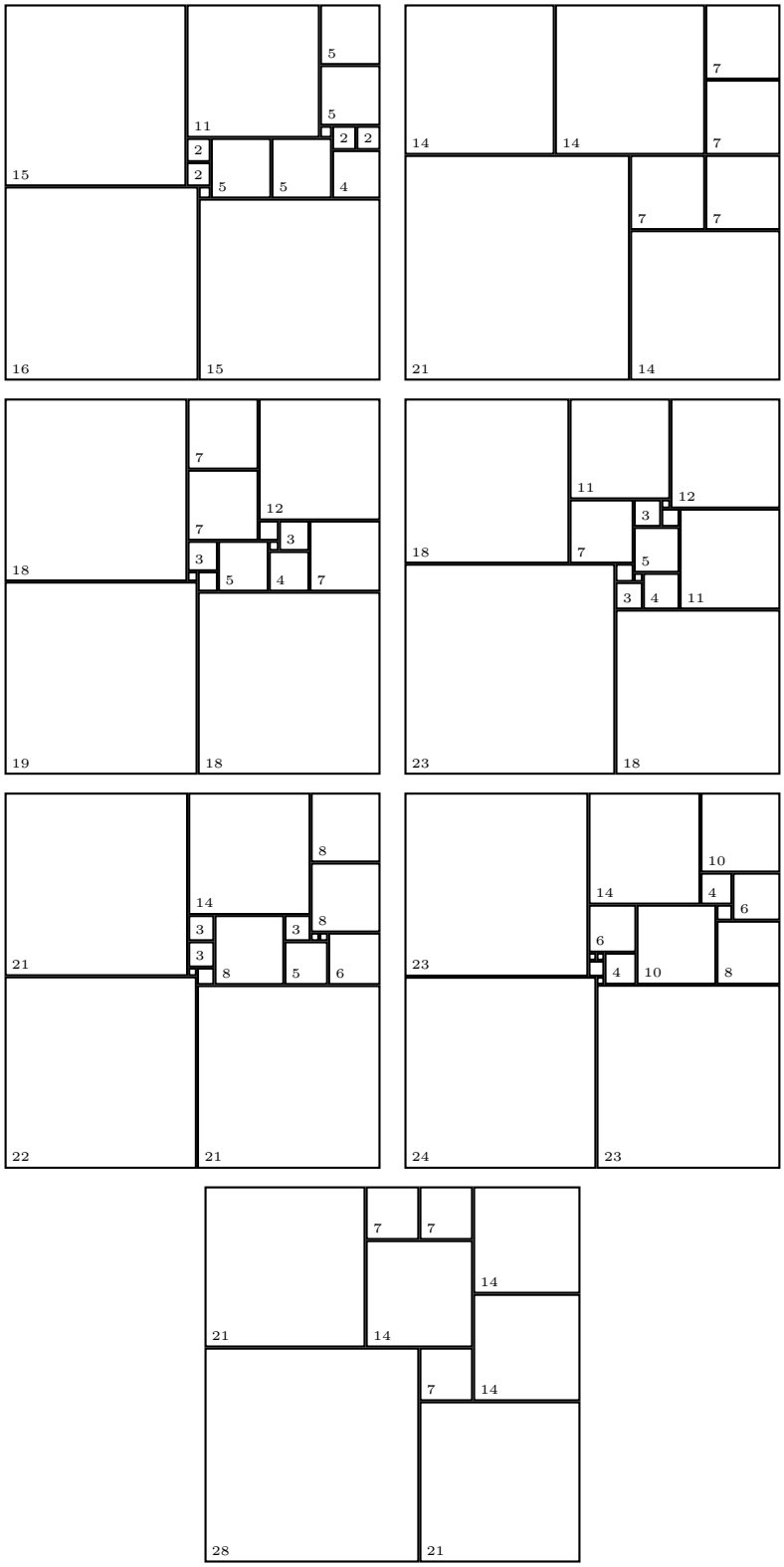

Figure 3: Solutions for prime side lengths ($31 < n < 50$).

Let's now implement the method that solves the trivial cases. It puts one large square and then fills the remaining parts with the samller squares. The parameter `d` can take 2 or 3 as values, and `n` is the square side length. It returns a list of arrays of `int`s, which have three values in the format as outlined in the problem description.

175a ⟨*8.8 Methods* 175a⟩≡

```
private static Deque<int[]> trivial(int n, int d) {
    Deque<int[]> initial = new ArrayDeque<>();
    int h = n / d;
    initial.add(new int[] { 0, 0, (d - 1) * h });
    initial.add(new int[] { (d - 1) * h, (d - 1) * h, h });
    for (int i = 0; i < (d - 1); ++i) {
        initial.add(new int[] { (d - 1) * h, h * i, h });
        initial.add(new int[] { h * i, (d - 1) * h, h });
    }
    return initial;
}
```

Finally, we can start writing the method that will actually try to solve the input. `minSolutionSize` is the minimal size of a solution we are trying to find. This is used for prime values only (except 2) as only for them the solution sizes seem to grow monotonically. `adjust` parameter is used to change the sizes of the initial three squares.

175b ⟨*8.8 Methods* 175a⟩+≡

```
final static int[] primes = new int[] { 3, 5, 7, 11, 13, 17, 19, 23,
        29, 31, 37, 41, 43, 47 };

static Deque<int[]> getSolution(int n, int minSolutionSize, int adjust)
        throws InterruptedException, ExecutionException {
    if (n % 2 == 0) {
        return trivial(n, 2);
    } else if (n % 3 == 0) {
        return trivial(n, 3);
    } else {
        Deque<int[]> initial = new ArrayDeque<>();
        int size = n / 2 + 1;
        initial = new ArrayDeque<>();
        initial.add(new int[] { 0, 0, size + adjust });
        initial.add(new int[] { size + adjust, 0, size - 1 - adjust });
        initial.add(new int[] { 0, size + adjust, size - 1 - adjust });
        if (Arrays.binarySearch(primes, n) >= 0) {
            return getSolution(n, initial, minSolutionSize);
        } else {
            return getSolution(n, initial, 0);
        }
    }
}
```

Now we add a method that repeatedly calls `getSolution` with different `adjust` values and selects the one that gives the minimal solution. It's another heuristic: We increase `adjust` value up to 4, this seems to be enough to give correct answers for $n \leq 50$.

176 ⟨*8.8 Methods* 175a⟩+≡

```
static Deque<int[]> getSolution(int n, int minSolutionSize)
        throws InterruptedException, ExecutionException {
    Deque<int[]> best = getSolution(n, minSolutionSize, 0);
    int size = n / 2 + 1;
    for (int i = 1; size + i < n - 1 && i < 4; ++i) {
        Deque<int[]> curr = getSolution(n, minSolutionSize, i);
        if (curr.size() < best.size()) {
            best = curr;
        }
    }
    return best;
}
```

The following method generates a list of sets of the square sizes and tries to tile them all doing this in parallel.

177 ⟨*8.8 Methods* 175a⟩+≡

```
static Deque<int[]> getSolution(int n, Deque<int[]> initial,
        int minSolutionSize)
        throws InterruptedException, ExecutionException {
    int count = 0;
    for (int[] s : initial) {
        count += s[2] * s[2];
    }
    int maxHeight = n - initial.getFirst()[2];
    SquareSideLengthsGenerator s = new SquareSideLengthsGenerator(
            maxHeight,
            n * n - count,
            minSolutionSize - initial.size());

    final ExecutorService executor =
        Executors.newFixedThreadPool(Runtime.getRuntime().availableProcessors() + 1);
    final CompletionService<SolutionFinderTaskResult> completionService =
        new ExecutorCompletionService<>(executor);

    for (int i = 0; i < s.squareSizes.size(); ++i) {
        completionService.submit(
            new SolutionFinderTask(i, initial, s.squareSizes.get(i), n));
    }

    SolutionFinderTaskResult[] result =
        new SolutionFinderTaskResult[s.squareSizes.size()];

    int lastIndex = 0;
    try {
        while (true) {
            Future<SolutionFinderTaskResult> f = completionService.take();
            result[f.get().index] = f.get();
            for (int i = lastIndex; i < result.length; ++i) {
                if (result[i] == null) {
                    break;
                } else if (!result[i].found) {
                    lastIndex++;
                } else if (result[i].found) {
                    return result[i].solution;
                }
            }
        }
    } finally {
        if (executor != null) {
            executor.shutdownNow();
        }
    }
}
```

And that's all. Now we can write our main method.

178 ⟨*8.8 Main* 178⟩≡

```
public static void main(String[] args) throws Exception {
    int minSolutionSize = 0;
    for (int i = 1; i <= 10; ++i) {
        System.out.println(i);
        if (i < 3) {
            System.out.println("0");
            continue;
        }
        Deque<int[]> solution = getSolution(i, minSolutionSize);
        minSolutionSize = solution.size();
        System.out.println(solution.size());
        for (int[] square : solution) {
            System.out.printf("%d %d %d\n", square[0] + 1,
                    square[1] + 1, square[2]);
        }
    }
}
```

This program is painfuly slow. It took 90 minutes on my laptop to find all the solutions for the input range.

Note though that this program returns answers in the following format: The value on the first line is the size of the square n, the next line has the number of squares s, and the following s lines have the coordinates and sizes of the smaller squares.

To convert this into a program that you can submit to UVa online judge we need another program, which reads this input and produces a program as output.

Of course, we could have written the first program to output a program without this intermediate output, but I used the output to also generate LaTeX code for the images with the solutions.

Here is the program that generates a program to be submitted to the judge.

179 ⟨*Bigger Square Please Meta* 179⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class BiggerSquarePleaseMetaProgram {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    private static final String indend8 = "        ";
    private static final String indend4 = "    ";

    private static void printProgram(Map<Integer, List<List<Integer>>> solutions) {
        System.out.println("import java.io.BufferedReader;");
        System.out.println("import java.io.InputStreamReader;");
        System.out.println();
        System.out.println("public class BiggerSquarePlease {");
        System.out.println("    private static final BufferedReader reader = new BufferedReader(");
        System.out.println("            new InputStreamReader(System.in));");
        System.out.println();
        System.out.println("    private static final int[][][] table = new int[][][] {");

        for (int i = 0; i <= 50; ++i) {
            List<List<Integer>> solution = solutions.get(i);
            if (solution == null || solution.size() == 0) {
                System.out.println(indend8 + "{{}}, //" + i);
                continue;
            }
            System.out.print(indend8 + "{");
            for (List<Integer> s : solution) {
                System.out.printf("{%d, %d, %d},", s.get(0), s.get(1), s.get(2));
            }
            System.out.println("}, //" + i);
        }
        System.out.println(indend4 + "};");
        System.out.println();
```

```
        System.out.println(indend4 + "public static void main(String[] args) throws Exception {");
        System.out.println(indend4 + "    int n = Integer.parseInt(reader.readLine().trim());");
        System.out.println(indend4 + "    for (int i = 0; i < n; ++i) {");
        System.out.println(indend4 + "        int m = Integer.parseInt(reader.readLine().trim());");
        System.out.println(indend4 + "        int[][] solution = table[m];");
        System.out.println(indend4 + "        System.out.println(solution.length);");
        System.out.println(indend4 + "        for (int[] s : solution) {");
        System.out.println(indend4 + "          System.out.println(s[0] + \" \" + s[1] + \" \" + s[2]);");
        System.out.println(indend4 + "        }");
        System.out.println(indend4 + "    }");
        System.out.println(indend4 + "}");
        System.out.println("}");
        System.out.println();
    }

    public static void main(String[] args) throws Exception {
        String currentLine;
        Map<Integer, List<List<Integer>>> solutions = new HashMap<>();
        while ((currentLine = reader.readLine()) != null) {
            int n = Integer.parseInt(currentLine.trim());
            int s = Integer.parseInt(reader.readLine().trim());
            List<List<Integer>> solution = new ArrayList<>();
            for (int i = 0; i < s; ++i) {
                solution.add(stream(reader.readLine().trim().split(" "))
                        .filter(x -> !x.equals(""))
                        .map(Integer::parseInt)
                        .collect(toList()));
            }
            solutions.put(n, solution);
        }
        printProgram(solutions);
    }
}
```

9 Graph Traversal

9.1 Bicoloring

The problem is whether this graph is bipartite or not. It's easy to show if the graph is bipartite or not by using the breadth-first search. Each child of a node is given an opposite color. If a node has been visited, we check if its color is opposite to our current color, and if not, then this graph is not bipartite. If we can complete the breadth-first search without encountering that situation, then the graph is bipartite.

181 ⟨*Bicoloring* 181⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class Bicoloring {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static void add(Map<Integer, Set<Integer>> graph, Integer nodeFrom,
            Integer nodeTo) {
        if (!graph.containsKey(nodeFrom)) {
            graph.put(nodeFrom, new HashSet<Integer>());
        }
        if (!graph.containsKey(nodeTo)) {
            graph.put(nodeTo, new HashSet<Integer>());
        }
        graph.get(nodeFrom).add(nodeTo);
        graph.get(nodeTo).add(nodeFrom);
    }

    private static boolean solve(
            Map<Integer, Set<Integer>> graph) {
        if (graph.size() == 0) {
            return true;
        }

        Deque<Integer> next = new ArrayDeque<>();
        Set<Integer> visited = new HashSet<>();
        Map<Integer, Boolean> colors = new HashMap<>();

        boolean color = true;
        Integer start = graph.keySet().iterator().next();
```

```
            next.push(start);
            colors.put(start, color);

            while (!next.isEmpty()) {
                Integer node = next.pop();
                color = colors.get(node);
                for (Integer adjNode : graph.get(node)) {
                    if (!visited.contains(adjNode)) {
                        visited.add(adjNode);
                        next.addLast(adjNode);
                        colors.put(adjNode, !color);
                    } else if (colors.get(adjNode) == color) {
                        return false;
                    }
                }
            }

            return true;
        }

        public static void main(String[] args) throws IOException {
            while (true) {
                int n = Integer.parseInt(reader.readLine().trim());
                if (n == 0) {
                    break;
                }
                n = Integer.parseInt(reader.readLine().trim());
                Map<Integer, Set<Integer>> graph = new HashMap<>();
                for (int j = 0; j < n; ++j) {
                    List<Integer> toFrom = stream(
                            reader.readLine().trim().split(" "))
                                    .filter(x -> !x.equals(""))
                                    .map(Integer::parseInt)
                                    .collect(toList());
                    add(graph, toFrom.get(0), toFrom.get(1));
                }
                System.out
                        .println(
                                solve(graph) ? "BICOLORABLE." : "NOT BICOLORABLE.");
            }
        }
    }
```

9.2 Playing With Wheels

A simple breadth-first search algorithm solves the task. However the judge has a caveat. Even though the problem statement clearly says there will be a blank line between two consecutive input cases, this is, apparently, not true. You will get a "Runtime error" if you believe there is only one blank line or at least one blank line. Assume there may be multiple blank lines or none at all!

183 ⟨*Playing With Wheels* 183⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.Arrays;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class PlayingWithWheels {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    static class Node {
        int[] node;

        @Override
        public int hashCode() {
            final int prime = 31;
            int result = 1;
            result = prime * result + Arrays.hashCode(node);
            return result;
        }

        @Override
        public boolean equals(Object obj) {
            if (this == obj)
                return true;
            if (obj == null)
                return false;
            if (getClass() != obj.getClass())
                return false;
            Node other = (Node) obj;
            if (!Arrays.equals(node, other.node))
                return false;
            return true;
        }
```

```
        public Node(List<Integer> s) {
            this.node = new int[s.size()];
            for (int i = 0; i < s.size(); ++i) {
                this.node[i] = s.get(i).intValue();
            }
        }

        public Node(Node node, int i, int p) {
            this.node = Arrays.copyOf(node.node, node.node.length);
            this.node[i] += p;
            if (this.node[i] < 0) {
                this.node[i] = 9;
            } else {
                this.node[i] %= 10;
            }
        }
    }

    private static Set<Node> adjacent(Node node) {
        Set<Node> adjacent = new HashSet<>();
        for (int i = 0; i < node.node.length; ++i) {
            adjacent.add(new Node(node, i, 1));
            adjacent.add(new Node(node, i, -1));
        }
        return adjacent;
    }

    private static int solve(Node start, Node end, Set<Node> forbidden) {
        if (start.equals(end)) {
            return 0;
        }

        Deque<Node> next = new ArrayDeque<>();
        Set<Node> visited = new HashSet<>();
        Map<Node, Integer> depths = new HashMap<>();

        int depth = 0;
        next.push(start);
        depths.put(start, depth);

        while (!next.isEmpty()) {
            Node node = next.pop();
            depth = depths.get(node);
            for (Node adjNode : adjacent(node)) {
                if (adjNode.equals(end)) {
                    return forbidden.contains(end) ? -1 : depth + 1;
                }
                if (!forbidden.contains(adjNode) &&
                        !visited.contains(adjNode)) {
                    visited.add(adjNode);
                    next.addLast(adjNode);
                    depths.put(adjNode, depth + 1);
                }
```

```
            }
        }

        return -1;
    }

    private static List<Integer> parseLine(String line) {
        return stream(line.trim().split(" "))
                .filter(x -> !x.equals(""))
                .map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            String currentLine = "";
            while ((currentLine = reader.readLine()).trim().equals(""))
                ;
            Node start = new Node(parseLine(currentLine));
            Node end = new Node(parseLine(reader.readLine()));
            int m = Integer.parseInt(reader.readLine().trim());
            Set<Node> forbidden = new HashSet<>();
            for (int j = 0; j < m; ++j) {
                forbidden.add(new Node(parseLine(reader.readLine())));
            }
            System.out.println(solve(start, end, forbidden));
        }
    }
}
```

9.3 The Tourist Guide

Direct application of maximin using the Floyd-Warshall algorithm (see [9] and [10]). Again and again the judge seems to disregard its own format and inserts empty lines here and there, so watch out with your input parsing!

186 ⟨*The Tourist Guide* 186⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.List;

public class TheTouristGuide {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int solve(int[][] w, int size, int s, int e,
            int t) {
        int[][] d = new int[size][size];
        for (int i = 0; i < size; i++) {
            for (int j = 0; j < size; j++) {
                d[i][j] = w[i][j];
            }
        }
        for (int i = 0; i < size; i++) {
            d[i][i] = 0;
        }
        for (int k = 0; k < size; k++) {
            for (int i = 0; i < size; i++) {
                for (int j = 0; j < size; j++) {
                    d[i][j] = Math.max(d[i][j], Math.min(d[i][k], d[k][j]));
                }
            }
        }
        return d[s][e];
    }

    private static List<Integer> parseLine(String line) {
        return stream(line.trim().split(" "))
                .filter(x -> !x.equals(""))
                .map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        int scenario = 1;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().isEmpty()) {
            List<Integer> nr = parseLine(currentLine);
            if (nr.get(0) == 0 && nr.get(1) == 0) {
```

```
                break;
            }
            int size = nr.get(0) + 1;
            int[][] w = new int[size][size];
            for (int i = 0; i < nr.get(1); ++i) {
                List<Integer> node = parseLine(reader.readLine());
                w[node.get(0)][node.get(1)] = node.get(2);
                w[node.get(1)][node.get(0)] = node.get(2);
            }
            List<Integer> sdt = parseLine(reader.readLine());
            int m = solve(w, size, sdt.get(0), sdt.get(1),
                    sdt.get(2));
            int c = (sdt.get(2)) / (m - 1);
            if (sdt.get(2) % (m - 1) != 0) {
                c++;
            }
            System.out.println("Scenario #" + scenario);
            System.out.println("Minimum Number of Trips = " + c);
            System.out.println();
            scenario++;
        }
    }
}
```

9.4 Slash Maze

We can easily use the depth-first search to find the cycles. The trick is how to read the maze and make it a graph. Well, that's not difficult either: Imagine that each slash is in a two by two square, then imagine there are four triangles in such a square. Each of these triangles is a node in a graph. There will be two pairs of two triangles that will always be connected. Each of the two by two squares will be connected to its neighbours from four sides (provided there are such neighbours). So that's how we can construct a graph. Once we've found a cycle, we need to divide its length by 2, because squares are made of two triangles.

188 ⟨*Slash Maze* 188⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Deque;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class SlashMaze {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static int[] find(Map<Integer, List<Integer>> graph) {
        int count = 0;
        int max = 0;
        Set<Integer> visited = new HashSet<>();
        while (true) {
            int[] m = new int[] { 0, 0 };
            Integer src = graph.keySet().stream()
                    .filter(x -> !visited.contains(x))
                    .findFirst().orElse(Integer.MIN_VALUE);
            if (src == Integer.MIN_VALUE) {
                break;
            }
            find(graph, visited, src, m);
            count += m[0];
            max = Math.max(max, m[1]);
        }
        return new int[] { count, max / 2 };
    }

    private static void find(Map<Integer, List<Integer>> graph,
            Set<Integer> visited, int parent, int m[]) {
        Deque<Integer> stack = new ArrayDeque<>();
        Map<Integer, Integer> depths = new HashMap<>();
```

```
    int src = parent;
    stack.push(src);
    depths.put(src, 0);

    while (!stack.isEmpty()) {
        src = stack.pop();
        int depth = depths.get(src);
        if (!visited.contains(src)) {
            visited.add(src);
            for (int adj : graph.get(src)) {
                if (adj == parent && depth > 2) {
                    m[0] += 1;
                    m[1] = Math.max(m[1], depth + 1);
                }
                stack.push(adj);
                depths.put(adj, depth + 1);
            }
        }
    }
}

private static List<Integer> parseLine(String line) {
    return stream(line.trim().split(" "))
            .filter(x -> !x.equals(""))
            .map(Integer::parseInt)
            .collect(toList());
}

public static void add(Map<Integer, List<Integer>> graph, int src,
        int dst) {
    if (!graph.containsKey(src)) {
        graph.put(src, new ArrayList<>());
    }
    if (!graph.containsKey(dst)) {
        graph.put(dst, new ArrayList<>());
    }
    graph.get(src).add(dst);
    graph.get(dst).add(src);
}

public static void main(String[] args) throws IOException {
    String currentLine;
    int scenario = 1;
    while ((currentLine = reader.readLine()) != null &&
            !currentLine.trim().isEmpty()) {
        List<Integer> wh = parseLine(currentLine);
        if (wh.get(0) == 0 && wh.get(1) == 0) {
            break;
        }
        int node = 0;
        Map<Integer, List<Integer>> graph = new HashMap<>();
        for (int i = 0; i < wh.get(1); ++i) {
            currentLine = reader.readLine().trim();
```

```
                for (char c : currentLine.toCharArray()) {
                    if (c == '/') {
                        add(graph, node, node + 1);
                        add(graph, node + 2, node + 3);
                    } else if (c == '\\') {
                        add(graph, node, node + 3);
                        add(graph, node + 1, node + 2);
                    }
                    add(graph, node, node - 2);
                    int nodeBelow = node + (wh.get(0) * 4) + 1;
                    add(graph, node + 3, nodeBelow);
                    node += 4;
                }
            }
            int[] result = find(graph);
            System.out.println("Maze #" + scenario + ":");
            if (result[0] > 0) {
                System.out.println(
                        result[0] + " Cycles; the longest has length " +
                                result[1] + ".");
            } else {
                System.out.println("There are no cycles.");
            }
            scenario++;
            System.out.println();
        }
    }
}
```

9.5 Edit Step Ladders

This problem can be solved in a few different ways. In this case I'm going to solve it using an obvious way. This implementation isn't particularly fast, but it solves the judge's input under 3 seconds.

OK, the way we solve it the following: We iterate on each input word. For each word we find words that are one edit step away and that are lexicographically preceding. Then we find edit step ladder lengths for each of those preceding words, and choose the maximum. (But notice that we can build a table bottom up. For this to work though words must be sorted in lexicographic order.) Current word ladder length will be the maximum plus one. At the same time we keep track of the maximum ladder length. Then we continue in this manner until we have iterated on the whole list of words. Once finished, we will know the maximum.

191 ⟨*Edit Step Ladders* 191⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Set;

public class EditStepLadders {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static void add(List<String> list, Set<String> dict, String word,
            String newWord) {
        if (newWord.compareTo(word) < 0 && dict.contains(newWord)) {
            list.add(newWord);
        }
    }

    private static List<String> adjacent(Set<String> dict, String word) {
        List<String> words = new ArrayList<>();
        char[] wordArr = word.toCharArray();
        for (int i = 0; i <= word.length(); ++i) {
            for (char c = 'a'; c <= 'z'; ++c) {
                add(words, dict, word, new StringBuilder().append(wordArr, 0, i)
                        .append(c)
                        .append(wordArr, i, wordArr.length - i).toString());
                if (i < word.length()) {
                    add(words, dict, word,
                            new StringBuilder().append(wordArr, 0, i)
                                    .append(c)
                                    .append(wordArr, i + 1,
                                            wordArr.length - i - 1)
                                    .toString());
                }
            }
            if (i < word.length()) {
```

```
                add(words, dict, word,
                        new StringBuilder().append(wordArr, 0, i)
                                .append(wordArr, i + 1, wordArr.length - i - 1)
                                .toString());
            }
        }
        return words;
    }

    private static int solve(Set<String> dict) {
        List<String> sorted = new ArrayList<>(dict);
        sorted.sort((x, y) -> x.compareTo(y));
        Map<String, Integer> lengths = new HashMap<>();
        Integer answer = 1;
        for (String word : sorted) {
            Integer max = 1;
            for (String source : adjacent(dict, word)) {
                Integer length = lengths.get(source);
                max = Math.max(max, length != null ? length + 1 : -1);
            }
            lengths.put(word, max);
            answer = Math.max(max, answer);
        }
        return answer;
    }

    public static void main(String[] args) throws IOException {
        Set<String> dict = new HashSet<>();
        String currentLine;
        while ((currentLine = reader.readLine()) != null &&
                !currentLine.trim().isEmpty()) {
            dict.add(currentLine.trim());
        }
        System.out.println(solve(dict));
    }
}
```

9.6 Tower of Cubes

I'm not sure what would be the best way to solve this task using graph algorithms, but this task can be easily solved using dynamic programming (just like the previous task). We will keep a table to hold the intermediate solutions (that is tower heights). We start with the heaviest cube and place it in every possible way, that is just 6 ways. Then we take the next cube and try to place it on every heavier cube provided the bottom color of this cube matches the top color of the cube below. If we can place it it means we have just improved the height of our tower, and we keep track of this in the table. We keep placing cubes until we've exhausted all the cubes. To get the actual tower we need to keep the references to the cubes below together with the index of the top side.

The method `buildUpTable` builds up the `table` discussed above. And the `buildUpTower` method traverses the stack via `refs` and builds up the actual tower.

193 ⟨*Tower of Cubes* 193⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class TowerOfCubes {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    final static int[] map = new int[] { 1, 0, 3, 2, 5, 4 };
    final static String[] toString = new String[] { "front", "back",
            "left", "right", "top", "bottom" };
    final List<String> tower = new ArrayList<>();
    final int[][] table;
    final int[][][] refs;
    int height = 0;
    int colorIdx = 0;
    int head = 0;

    TowerOfCubes(List<List<Integer>> cubes) {
        table = new int[cubes.size()][6];
        refs = new int[cubes.size()][6][2];
        buildUpTable(cubes);
        buildUpTower();
    }

    private void buildUpTower() {
        int currentColorIdx = colorIdx;
        while (true) {
            tower.add((head + 1) + " " + toString[currentColorIdx]);
            int[] ref = refs[head][currentColorIdx];
            if (ref == null) {
                break;
            }
```

```
            head = ref[0];
            currentColorIdx = ref[1];
        }
    }

    private static int indexOf(int start, int value, List<Integer> array) {
        for (int i = start; i < array.size(); ++i) {
            if (array.get(i) == value) {
                return i;
            }
        }
        return -1;
    }

    private void buildUpTable(List<List<Integer>> cubes) {
        for (int current = cubes.size() - 1; current >= 0; --current) {
            List<Integer> cube = cubes.get(current);
            for (int j = 0; j < 6; ++j) {
                int color = cube.get(map[j]);
                int max = 1;
                refs[current][j] = null;
                for (int i = current + 1; i < cubes.size(); ++i) {
                    int topColorIndex = -1;
                    while ((topColorIndex = indexOf(topColorIndex + 1, color,
                            cubes.get(i))) != -1) {
                        if (max < table[i][topColorIndex] + 1) {
                            max = table[i][topColorIndex] + 1;
                            refs[current][j] = new int[] { i, topColorIndex };
                        }
                    }
                }
                table[current][j] = max;
                if (height < max) {
                    colorIdx = j;
                    height = max;
                    head = current;
                }
            }
        }
    }

    public static void main(String[] args) throws IOException {
        String currentLine;
        int caseNo = 1;
        while ((currentLine = reader.readLine()) != null) {
            while (currentLine.trim().isEmpty()) {
                currentLine = reader.readLine();
            }
            int n = Integer.parseInt(currentLine.trim());
            if (n == 0) {
                break;
            }
            List<List<Integer>> cubes = new ArrayList<>();
```

```
            for (int i = 0; i < n; ++i) {
                cubes.add(stream(
                        reader.readLine().trim().split(" "))
                                .filter(x -> !x.equals(""))
                                .map(Integer::parseInt)
                                .collect(toList()));
            }
            System.out.println("Case #" + (caseNo++));
            TowerOfCubes toc = new TowerOfCubes(cubes);
            System.out.println(toc.height);
            toc.tower.stream().forEach(System.out::println);
            System.out.println();
        }
    }
}
```

9.7 From Dusk Till Dawn

First, we need to filter out all the trains that either depart after 6am or arrive before 6pm or both. For the remaining trains we just do a classic breadth-first search, considering routes $(SRC \to DST)_x$ as a node and connecting them to other nodes $(SRC \to DST)_y$ if $SRC_y = DST_x$ and departure from SRC_y is chronologically after DST_x within the same day. After we have discovered where we can arrive within one day, we check if those destinations do not contain our required destination. If not, it means we need to wait for another day and repeat the DFS again starting with these nodes. We keep doing so until we either arrive at the required station or we declare there's no such route.

196 ⟨*From Dusk Till Dawn* 196⟩≡

```
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayDeque;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Deque;
import java.util.HashSet;
import java.util.List;
import java.util.Set;

public class FromDuskTillDawn {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    static class Train {
        String src;
        int srcTime;
        String dst;
        int dstTime;

        public Train(String src, int srcTime, String dst, int dstTime) {
            this.src = src;
            this.srcTime = srcTime;
            this.dst = dst;
            this.dstTime = dstTime;
        }
    }

    private static List<Train> nextTrains(Train lastTrain,
            List<Train> trains) {
        List<Train> nextTrains = new ArrayList<>();
        for (Train next : trains) {
            if (next.src.equals(lastTrain.dst) &&
                    next.srcTime >= lastTrain.dstTime) {
                nextTrains.add(next);
            }
        }
        return nextTrains;
```

```
}

private static int solve(String src, String dst, List<Train> trains) {
    int count = 0;
    if (src.equals(dst)) {
        return count;
    }
    Set<String> reachable = new HashSet<>();
    Set<String> visited = new HashSet<>();
    reachable.add(src);
    visited.add(src);

    while (!reachable.isEmpty()) {
        Set<String> _reachable = new HashSet<>();
        for (String srcStation : reachable) {
            for (Train trn : trains) {
                if (trn.src.equals(srcStation)) {
                    _reachable.addAll(getReachable(trn, trains));
                }
            }
        }
        _reachable.removeAll(visited);
        visited.addAll(reachable);
        reachable = _reachable;
        if (reachable.contains(dst)) {
            return count;
        }
        count++;
    }

    return -1;
}

private static Set<String> getReachable(Train src,
        List<Train> trains) {
    Deque<Train> stack = new ArrayDeque<>();
    Set<Train> used = new HashSet<>();
    Set<String> reachable = new HashSet<>();
    stack.push(src);
    while (!stack.isEmpty()) {
        Train train = stack.pop();
        reachable.add(train.dst);
        List<Train> nextTrn = nextTrains(train, trains);
        for (Train nextTrain : nextTrn) {
            if (!used.contains(nextTrain)) {
                stack.push(nextTrain);
            }
        }
        used.add(train);
    }
    return reachable;
}
```

```
    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            List<Train> trains = new ArrayList<>();
            int m = Integer.parseInt(reader.readLine().trim());
            for (int j = 0; j < m; ++j) {
                List<String> input = Arrays
                        .stream(reader.readLine().trim().split(" "))
                        .filter(x -> !x.equals(""))
                        .collect(toList());
                int srcTime = Integer.parseInt(input.get(2));
                srcTime = srcTime <= 6 ? srcTime + 24 : srcTime;
                int dstTime = (srcTime + Integer.parseInt(input.get(3)));
                String src = input.get(0);
                String dst = input.get(1);
                if (!(srcTime >= 18 && srcTime <= 29 &&
                        dstTime >= 19 && dstTime <= 30)) {
                    continue;
                }
                trains.add(new Train(src.toLowerCase(), srcTime,
                        dst.toLowerCase(), dstTime));
            }
            List<String> input = Arrays
                    .stream(reader.readLine().trim().split(" "))
                    .filter(x -> !x.equals(""))
                    .collect(toList());
            int count = solve(input.get(0).toLowerCase(),
                    input.get(1).toLowerCase(), trains);
            System.out.println("Test Case " + (i + 1) + ".");
            if (count == -1) {
                System.out.println("There is no route Vladimir can take.");
            } else {
                System.out.println(
                        "Vladimir needs " + count + " litre(s) of blood.");
            }
        }
    }

}
```

9.8 Hanoi Tower Troubles Again!

This task must be a joke because it's trivial. (To much surprise, it's designated as "Level 3" task.)

199 ⟨*Hanoi Tower Troubles Again* 199⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;

public class HanoiTowerTroublesAgain {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static boolean isSquare(int v) {
        int i = (int) Math.sqrt(v);
        return i * i == v;
    }

    public static int solve(int m) {
        int j = 0;
        int i = 0;
        int[] v = new int[m];
        while (i < m) {
            j++;
            for (i = 0; i < m; ++i) {
                if (v[i] == 0 || isSquare(v[i] + j)) {
                    v[i] = j;
                    break;
                }
            }
        }
        return j - 1;
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            int m = Integer.parseInt(reader.readLine().trim());
            System.out.println(solve(m));
        }
    }

}
```

10 Graph Algorithms

10.1 Freckles

This task asks for a direct application of any suitable minimum spanning tree algorithm. I've chosen to use Prim's algorithm as the input of this task doesn't require any sophisticated data structures usage (like Fibonacci heaps etc) and can be implemented in a naive way.

200 ⟨*Freckles* 200⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.text.DecimalFormat;
import java.util.ArrayList;
import java.util.List;

public class Freckles {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    static class Edge {
        final double[] src;
        final double[] dst;
        final double distance;

        public Edge(double[] src, double[] dst) {
            super();
            this.src = src;
            this.dst = dst;
            double xx = Math.abs(src[0] - dst[0]);
            double yy = Math.abs(src[1] - dst[1]);
            this.distance = Math.sqrt(xx * xx + yy * yy);
        }
    }

    private static Edge min(List<double[]> tree, List<double[]> nodes) {
        Edge min = null;
        for (double[] n : nodes) {
            for (double[] t : tree) {
                Edge edge = new Edge(t, n);
                if (min != null && edge.distance < min.distance ||
                        min == null) {
                    min = edge;
                }
            }
        }
        return min;
    }

    private static double mst(List<double[]> nodes) {
        double total = 0;
```

```
        List<double[]> tree = new ArrayList<>();
        tree.add(nodes.remove(0));
        while (!nodes.isEmpty()) {
            Edge edge = min(tree, nodes);
            total += edge.distance;
            tree.add(edge.dst);
            nodes.remove(edge.dst);
        }
        return total;
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        DecimalFormat format = new DecimalFormat("#0.00");
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            int m = Integer.parseInt(reader.readLine().trim());
            List<double[]> nodes = new ArrayList<>();
            for (int j = 0; j < m; ++j) {
                List<Double> tuple = stream(reader.readLine().trim().split(" "))
                        .filter(x -> !x.equals(""))
                        .map(Double::parseDouble)
                        .collect(toList());
                nodes.add(new double[] { tuple.get(0), tuple.get(1) });
            }
            System.out.println(format.format(mst(nodes)));
            if (i < n - 1) {
                System.out.println();
            }
            reader.readLine();
        }
    }

}
```

10.2 The Necklace

It's a bit tricky problem. At first sight this looks like a Hamiltonian path problem, but having 1000 nodes in a graph makes this impossible to solve this way. The problem, as the hint in the book suggests, can be solved if modeled as an Eulerian cycle problem. Each color becomes a node in the graph. Next we find all the simple cycles in the graph using the DFS, and then merge them into an Eulerian cycle. Before doing so we check if all of this graph's nodes have even degrees (as per Euler's theorem), and if they do, we search for the Eulerian cycle; otherwise, we conclude this graph can't have an Eulerian cycle.

202 ⟨*The Necklace* 202⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;

public class TheNecklace {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static final int MAX_COL = 51;
    private final List<List<Integer>> nodes;
    private final List<List<Boolean>> traversed = new ArrayList<>();
    private final List<int[]> path = new ArrayList<>();

    private TheNecklace(List<List<Integer>> nodes) {
        this.nodes = nodes;
        for (int i = 0; i < MAX_COL; ++i) {
            traversed.add(new ArrayList<>());
        }

        boolean eulerian = true;
        int src = -1;
        for (int i = 0; i < nodes.size(); ++i) {
            List<Integer> node = nodes.get(i);
            if (node.size() % 2 != 0) {
                eulerian = false;
                break;
            }
            for (int j = 0; j < node.size(); ++j) {
                traversed.get(i).add(false);
            }
            if (src == -1 && node.size() > 0) {
                src = i;
            }
        }

        if (eulerian) {
            traverse(src, 0);
```

```
    }
}

private void traverse(int src, int pos) {
    for (int i = 0; i < nodes.get(src).size(); ++i) {
        if (traversed.get(src).get(i)) {
            continue;
        }
        traversed.get(src).set(i, true);
        int dst = nodes.get(src).get(i);
        for (int j = 0; j < nodes.get(dst).size(); ++j) {
            if (nodes.get(dst).get(j) == src &&
                    !traversed.get(dst).get(j)) {
                traversed.get(dst).set(j, true);
                break;
            }
        }
        path.add(pos, new int[] { src, dst });
        traverse(dst, pos + 1);
    }
}

public static void main(String[] args) throws IOException {
    int n = Integer.parseInt(reader.readLine().trim());
    for (int i = 0; i < n; ++i) {
        int m = Integer.parseInt(reader.readLine().trim());
        List<List<Integer>> nodes = new ArrayList<>();
        for (int j = 0; j < MAX_COL; ++j) {
            nodes.add(new ArrayList<>());
        }
        for (int j = 0; j < m; ++j) {
            List<Integer> node = stream(
                    reader.readLine().trim().split(" "))
                            .filter(x -> !x.equals(""))
                            .map(Integer::parseInt)
                            .collect(toList());
            nodes.get(node.get(0)).add(node.get(1));
            nodes.get(node.get(1)).add(node.get(0));
        }
        TheNecklace necklace = new TheNecklace(nodes);
        System.out.println("Case #" + (i + 1));
        if (necklace.path.size() != 0) {
            necklace.path
                    .forEach(x -> System.out.println(x[0] + " " + x[1]));
        } else {
            System.out.println("some beads may be lost");
        }
        if (i < n - 1) {
            System.out.println();
        }
    }
}
```

```
}
```

10.3 Fire Stations

Each intersection is a node in the graph. We are asked to minimize the maximum distance to the other nodes in the graph by placing a fire station at one of the nodes in the graph provided we already have a set of fire stations at some of the nodes (or not at all).

OK, because we may have fire stations at some of the intersections, we need to calculate the distances from those fire stations to the other nodes. We keep track of the maximum distances and keep track of the minimum of these maximum distances. Next, at every intersection we calculate the distances to the other intersections using the Dijkstra's algorithm. We keep track of the minimum of the maximum distances for each intersection and select the one that minimizes it, as required by the problem.

Our Dijkstra's algorithm implementation is not efficient as it doesn't use a priority queue. Standard Java doesn't have any proper priority queue. (Well, it does, but it's not a proper priority queue as it doesn't have "decrease priority" semantics to it.) But because the problem has a generous time out of more than 6 seconds, a naive implementation is sufficient.

Let's break it down. First, I/O. We are going to represent the graph as a map, where key is the intersection index (node index) and the value is a list of `AdjNode` edges which are simply tuples with indexes of adjacent nodes and the weights (distances) of the corresponding edges. The `solve` method is the main method that will determine the index of an intersection where a new fire station needs to be built. It takes as input a set of indexes of the existing fire stations and the graph itself.

205 ⟨*Fire Stations* 205⟩≡

```
import static java.util.Arrays.stream;
import static java.util.stream.Collectors.toList;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collection;
import java.util.HashMap;
import java.util.HashSet;
import java.util.List;
import java.util.Map;
import java.util.Map.Entry;
import java.util.Set;

public class FireStations {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    static class AdjNode {
        final int index;
        final long dist;

        public AdjNode(int index, long dist) {
            this.index = index;
            this.dist = dist;
        }
    }
```

⟨*10.3 Methods* 207⟩
⟨*10.3 Solver* 208b⟩

```
    private static List<Integer> parseLine(String line) {
        return stream(line.trim().split(" "))
                .filter(x -> !x.equals(""))
                .map(Integer::parseInt)
                .collect(toList());
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        reader.readLine();
        for (int i = 0; i < n; ++i) {
            List<Integer> fi = parseLine(reader.readLine());
            Set<Integer> stations = new HashSet<>();
            for (int j = 0; j < fi.get(0); ++j) {
                stations.add(Integer.parseInt(reader.readLine().trim()));
            }
            Map<Integer, List<AdjNode>> graph = new HashMap<>();
            String currentLine = "";
            while ((currentLine = reader.readLine()) != null &&
                    !currentLine.trim().isEmpty()) {
                List<Integer> node = parseLine(currentLine);
                graph.putIfAbsent(node.get(0), new ArrayList<>());
                graph.putIfAbsent(node.get(1), new ArrayList<>());
                graph.get(node.get(0))
                        .add(new AdjNode(node.get(1), node.get(2)));
                graph.get(node.get(1))
                        .add(new AdjNode(node.get(0), node.get(2)));
            }

            System.out.println(solve(stations, graph));

            if (i < n - 1) {
                System.out.println();
            }
        }
    }

}
```

Now let's implement the Dijkstra's algorithm. It'll be a naive implementation, but this is sufficient to solve this task in an acceptable time.

The method takes the `graph` and the source `src` node index. It returns an array where each item represents the shortest path cost (in terms of distances) from the `src` to the corresponding item. Array's index corresponds to the intersection number.

207 ⟨*10.3 Methods* 207⟩≡

```
private static long[] dijkstra(
        Map<Integer, List<AdjNode>> graph, int src) {
    long[] distances = new long[501];
    Arrays.fill(distances, Long.MAX_VALUE);
    Set<Integer> next = new HashSet<>();
    for (Entry<Integer, List<AdjNode>> e : graph.entrySet()) {
        next.add(e.getKey());
    }
    distances[src] = 0L;

    while (!next.isEmpty()) {
        long min = -1;
        int node = -1;
        for (Integer nextNode : next) {
            if (distances[nextNode] < min || min == -1) {
                min = distances[nextNode];
                node = nextNode;
            }
        }
        next.remove(node);
        for (AdjNode adj : graph.get(node)) {
            long newMin = min + adj.dist;
            if (newMin < distances[adj.index]) {
                distances[adj.index] = newMin;
            }
        }
    }

    return distances;
}
```

OK, once we've got the array of distances for specific source, we will need to somehow merge it with an array of distances of a different source. For that we are going to use the following method. It'll take the nodes collection and two distance arrays, and produce a new one by retaining the minimal distances.

208a ⟨*10.3 Methods* 207⟩+≡

```
private static long[] merge(Collection<Integer> nodes,
        long[] prevDistances,
        long[] distancesFromStation) {
    long[] merged = Arrays.copyOf(prevDistances,
            prevDistances.length);
    for (Integer j : nodes) {
        merged[j] = Math.min(merged[j],
                distancesFromStation[j]);
    }
    return merged;
}
```

Finally, we can now implement the main `solve` method.

208b ⟨*10.3 Solver* 208b⟩≡

```
private static int solve(Set<Integer> stations,
     Map<Integer, List<AdjNode>> graph) {
    long[] distances = null;
    for (Integer station : stations) {
        long[] distancesFromStation = dijkstra(graph, station);
        distances = distances != null ? merge(graph.keySet(), distances,
                distancesFromStation) : distancesFromStation;
    }

    long minMax = Long.MAX_VALUE;
    int newStationNumber = 1;
    for (Integer src : graph.keySet()) {
        if (stations.contains(src)) {
            continue;
        }
        long[] distancesFromStation = dijkstra(graph, src);
        long[] newDistances = merge(graph.keySet(), distances,
                distancesFromStation);
        long max = 0;
        for (Integer i : graph.keySet()) {
            max = Math.max(newDistances[i], max);
        }
        if (max < minMax) {
            minMax = max;
            newStationNumber = src;
        }
    }

    return newStationNumber;
}
```

10.4 Railroads

Well, I failed to solve this task with the BFS. I thought it would perfectly work, but the judge repeatedly declined my solution with the "Time Limit Exceeded" verdict despide of a number of attempts with various optimizations. It was clear: A different approach was required here. In the end, I have purloined the idea for this task solution from [11].

209 ⟨*Railroads* 209⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;
import java.util.stream.Collectors;

public class Railroads {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    private static final int MAX_TIME = 2360;
    private static final int DEP_TIME = 0;
    private static final int DEP_STATION = 1;
    private static final int ARR_TIME = 2;
    private static final int ARR_STATION = 3;

    private static int[] solve(int m, List<int[]> trains, int source,
            int startTime, int destination) {
        Map<Integer, int[]> times = new HashMap<>();

        for (int j = 0; j < m; ++j) {
            int[] tab = new int[MAX_TIME];
            Arrays.fill(tab, -1);
            times.put(j, tab);
        }

        Collections.sort(trains,
                (x, y) -> x[DEP_TIME] != y[DEP_TIME] ? x[DEP_TIME] - y[DEP_TIME]
                        : x[ARR_TIME] - x[ARR_TIME]);

        for (int[] t : trains) {
            int[] tab = times.get(t[ARR_STATION]);
            if (t[DEP_STATION] == source && t[DEP_TIME] >= startTime) {
                for (int j = t[ARR_TIME]; j < tab.length; ++j) {
                    tab[j] = Math.max(tab[j], t[DEP_TIME]);
                }
            } else if (times.get(t[DEP_STATION])[t[DEP_TIME]] >= 0) {
                for (int j = t[ARR_TIME]; j < tab.length; ++j) {
                    tab[j] = Math.max(tab[j],
                            times.get(t[DEP_STATION])[t[DEP_TIME]]);
                }
```

```
            }
        }

        for (int j = 0; j < MAX_TIME; ++j) {
            if (times.get(destination)[j] > 0) {
                return new int[] { times.get(destination)[j], j };
            }
        }

        return null;
    }

    public static void main(String[] args) throws IOException {
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            int m = Integer.parseInt(reader.readLine().trim());
            Map<String, Integer> names = new HashMap<>();
            for (int j = 0; j < m; ++j) {
                names.put(reader.readLine().trim(), j);
            }

            List<int[]> trains = new ArrayList<>();
            int trainsCount = Integer.parseInt(reader.readLine().trim());
            for (int j = 0; j < trainsCount; ++j) {
                int stops = Integer.parseInt(reader.readLine().trim());
                int[] prev = null;
                for (int k = 0; k < stops; ++k) {
                    List<String> stop = Arrays
                            .stream(reader.readLine().trim().split(" "))
                            .filter(x -> !x.isEmpty())
                            .collect(Collectors.toList());
                    if (prev != null) {
                        trains.add(new int[] {
                                prev[0], prev[1],
                                Integer.parseInt(stop.get(0)),
                                names.get(stop.get(1)) });
                    }
                    prev = new int[] { Integer.parseInt(stop.get(0)),
                            names.get(stop.get(1)) };
                }
            }

            int startTime = Integer.parseInt(reader.readLine().trim());
            String sourceStr = reader.readLine().trim();
            int source = names.get(sourceStr);
            String destinationStr = reader.readLine().trim();
            int destination = names.get(destinationStr);

            int[] result = solve(m, trains, source, startTime, destination);

            System.out.println("Scenario " + (i + 1));
            if (result != null) {
                System.out.printf("Departure %04d %s\n", result[0], sourceStr);
```

```
                System.out.printf("Arrival   %04d %s\n", result[1],
                        destinationStr);
            } else {
                System.out.println("No connection");
            }
            System.out.println();
        }
    }

}
```

10.5 War

I failed to solve this task without help, so the solution below is based on the solution from [13].

The idea is to use a data structure called Disjoin-set[15]. For that we will maintain an array of $2n$ size, first half of which will be maintained for friends and the other half of it will be maintained for enemies; a trick purloined from [13]. And that is key.

Now what we need to do is carefully maintain friends and enemies sets. If i and j are both in the same set, it means they are friends. To check this we check if they are either both in the friends sets, or both are in the enemies sets. Similarly, to test if i and j are enemies, we check if one of them is in friends sets, and the other is in the enemies sets, or vice versa.

Now when we execute `setFriends(i, j)` we first check if i and j aren't enemies. If not, we join i and j, because friends of our friends are our friends too. We also join enemies, because enemies of our friends are our enemies. When we execute `setEnemies(i, j)`, we join i and enemies of j. Similarly we join j and enemies of i together. This is because enemies of our enemies are our friends.

212 ⟨*War* 212⟩≡

```
import java.util.*;

public class War {
    private final int[] friends;
    private final int n;

    public War(int n) {
        this.n = n;
        friends = new int[2 * (n + 1)];
        for (int i = 0; i < 2 * (n + 1); ++i) {
            friends[i] = i;
        }
    }

    public int enemy(int i) {
        return i + n;
    }

    public int find(int i) {
        if (friends[i] != i) {
            friends[i] = find(friends[i]);
        }
        return friends[i];
    }

    public void union(int i, int j) {
        if (find(i) != find(j)) {
            friends[find(j)] = find(i);
        }
    }

    private boolean setFriends(int i, int j) {
        if (areEnemies(i, j)) {
            return false;
        }
```

```
        union(i, j);
        union(enemy(i), enemy(j));
        return true;
    }

    private boolean setEnemies(int i, int j) {
        if (areFriends(i, j)) {
            return false;
        }
        union(i, enemy(j));
        union(j, enemy(i));
        return true;
    }

    private boolean areFriends(int i, int j) {
        return find(i) == find(j) ||
                find(enemy(i)) == find(enemy(j));
    }

    private boolean areEnemies(int i, int j) {
        return find(i) == find(enemy(j)) ||
                find(j) == find(enemy(i));
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        War war = new War(s.nextInt());
        while (s.hasNextInt()) {
            switch (s.nextInt()) {
                case 0:
                    return;
                case 1:
                    if (!war.setFriends(s.nextInt(), s.nextInt())) {
                        System.out.println(-1);
                    }
                    break;
                case 2:
                    if (!war.setEnemies(s.nextInt(), s.nextInt())) {
                        System.out.println(-1);
                    }
                    break;
                case 3:
                    System.out.println(war.areFriends(s.nextInt(), s.nextInt()) ? 1 : 0);
                    break;
                case 4:
                    System.out.println(war.areEnemies(s.nextInt(), s.nextInt()) ? 1 : 0);
                    break;
                default:
                    throw new IllegalArgumentException();
            }
        }
    }
}
```

10.6 Tourist Guide

This task is about finding articulation points in a graph. There are known algorithms to do this in linear time[16]. The only "catch" in this task is that you need to run articulation points search for each connected component, which there may be more than one, although the problem's description may suggest that town road map is a connected component. Apparently, that's not true for this task.

The `findCutPoints` is a direct translaction of the pseudo-code from [16].

215 ⟨*Tourist Guide* 215⟩≡

```
import java.util.*;

public class TouristGuide {
    private final Set<String> cutVertices = new HashSet<>();
    private final Set<String> visited = new HashSet<>();
    private final Map<String, String> parent = new HashMap<>();
    private final Map<String, Integer> depths = new HashMap<>();
    private final Map<String, Integer> low = new HashMap<>();
    private final Map<String, Set<String>> graph;

    public TouristGuide(Map<String, Set<String>> graph) {
        this.graph = graph;
        Set<String> processed = new HashSet<>();
        Set<String> roots = new HashSet<>(graph.keySet());
        while (processed.size() != graph.keySet().size()) {
            findCutVertices(roots.iterator().next(), 0);
            processed.addAll(visited);
            roots.removeAll(visited);
        }
    }

    public Set<String> getCutVertices() {
        return cutVertices;
    }

    private void findCutVertices(String v, int d) {
        visited.add(v);
        depths.put(v, d);
        low.put(v, d);
        int childCount = 0;
        boolean isCutVertix = false;
        for (String n : graph.get(v)) {
            if (!visited.contains(n)) {
                parent.put(n, v);
                findCutVertices(n, d + 1);
                childCount++;
                if (low.get(n) >= depths.get(v)) {
                    isCutVertix = true;
                }
                low.put(v, Math.min(low.get(v), low.get(n)));
            } else if (!n.equals(parent.get(v))) {
                low.put(v, Math.min(low.get(v), depths.get(n)));
            }
```

```
            }
            if ((parent.containsKey(v) && isCutVertix) ||
                    (!parent.containsKey(v) && childCount > 1)) {
                cutVertices.add(v);
            }
        }

        public static void main(String[] args) {
            Scanner s = new Scanner(System.in);
            int n = 0;
            int c = 0;
            while ((n = s.nextInt()) != 0) {
                c++;
                if (c > 1) {
                    System.out.println();
                }
                s.nextLine();
                for (int i = 0; i < n; ++i) {
                    s.nextLine();
                }
                Map<String, Set<String>> graph = new HashMap<>();
                int edgesCount = s.nextInt();
                s.nextLine();
                for (int i = 0; i < edgesCount; ++i) {
                    String[] edge = s.nextLine().replaceAll("\\s+", " ").split(" ");
                    assert (edge.length == 2);
                    graph.putIfAbsent(edge[0], new HashSet<>());
                    graph.putIfAbsent(edge[1], new HashSet<>());
                    graph.get(edge[0]).add(edge[1]);
                    graph.get(edge[1]).add(edge[0]);
                }
                TouristGuide t = new TouristGuide(graph);
                List<String> cutPoints = new ArrayList<>(t.getCutVertices());
                cutPoints.sort(String::compareTo);
                System.out.println(String.format("City map #%d: %d camera(s) found", c, cutPoints.size()));
                cutPoints.forEach(System.out::println);
            }
        }
    }
```

10.7 The Grand Dinner

This task can be modeled as a flow problem. Team sizes would become capacities of the edges coming from the source node. There would be M nodes connecting to the source node, where M is the number of teams. Then there would be another set of nodes corresponding to the tables; there would be N such nodes. Each edge leaving a "table" node and ending in the sink node would have a capacity of the corresponding table. Then each "team" node would be connected to each "table" node with an edge of capacity 1. Then we find a maximum flow in this network. If we can't saturate each edge leaving the source node to their maximum capacity, it means there's no solution to the problem. Otherwise a solution exists and the solution corresponds to the edges between "team" and "table" nodes that have flows through the edges.

I have tried submitting a solution based on Edmonds-Karp algorithm (as it is easy to implement), but the tests have large cases which make this algorithm impractical due to its running time complexity $O(VE^2)$. I got "Time Limit Exceeded" verdicts. (And what I was hoping for?) A solution using push relabel algorithm may pass the tests as it has much better running time complexity of $O(V^2E)$.

However, as the book suggests, there is a greedy algorithm. The strategy is to try to seat the largest team to the tables with the largest capacities first. Keep doing until you either succeed or fail.

217 ⟨*The Grand Dinner* 217⟩≡

```
import java.util.ArrayList;
import java.util.List;
import java.util.Scanner;
import java.util.stream.Collectors;

public class TheGrandDinner {
    static class Tuple {
        final int v1;
        int v2;

        public Tuple(int v1, int v2) {
            this.v1 = v1;
            this.v2 = v2;
        }
    }

    public static List<List<Integer>> solve(List<Tuple> teams, List<Tuple> tables) {
        List<List<Integer>> solution = new ArrayList<>();
        teams.forEach(x -> solution.add(new ArrayList<>()));
        teams.sort((x, y) -> Integer.compare(y.v2, x.v2));
        for (int i = 0; i < teams.size(); ++i) {
            tables.sort((x, y) -> Integer.compare(y.v2, x.v2));
            for (int j = 0; j < tables.size() && teams.get(i).v2 > 0; ++j) {
                solution.get(teams.get(i).v1 - 1).add(tables.get(j).v1);
                tables.get(j).v2--;
                teams.get(i).v2--;
                if (tables.get(j).v2 < 0) {
                    return null;
                }
            }
```

```
            if (teams.get(i).v2 > 0) {
                return null;
            }
        }
        return solution;
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = 0;
        int m = 0;

        while (true) {
            m = s.nextInt();
            n = s.nextInt();
            if (m == 0 && n == 0) {
                break;
            }
            List<Tuple> teams = new ArrayList<>();
            List<Tuple> tables = new ArrayList<>();
            for (int i = 0; i < m; ++i) {
                teams.add(new Tuple(i + 1, s.nextInt()));
            }
            for (int i = 0; i < n; ++i) {
                tables.add(new Tuple(i + 1, s.nextInt()));
            }

            List<List<Integer>> solution = solve(teams, tables);
            if (solution != null) {
                System.out.println(1);
                for (int i = 0; i < m; i++) {
                    System.out.println(solution.get(i)
                            .stream().map(String::valueOf)
                            .collect(Collectors.joining(" ")));
                }
            } else {
                System.out.println(0);
            }
        }
    }

}
```

10.8 The Problem with the Problem Setter

Imagine a graph with four types of nodes: the source, the sink, problem nodes, category nodes. Now, from the source to each problem node there are edges, each with capacity of 1. Then, corresponding to the test case, there are edges from problems to category nodes, again each with capacity of 1. From category nodes there are edges to the sink. These nodes will have capacities corresponding to the number of problems we want to have in this category, as per test case. Now we just need to find a maximum flow through this network. Then, if the nodes coming to the sink nodes are all saturated, we found a solution, otherwise there's no a solution.

Let's first define `Edge` and `Node` classes and define a method to add edges to the graph. When adding an edge, we will add its reversed version to the graph too. This will be needed when calculating a maximum flow.

219 ⟨*The Problem with the Problem Setter* 219⟩≡

```
import java.util.*;

public class TheProblemWithTheProblemSetter {
    private final Map<Node, List<Edge>> graph = new HashMap<>();

    static class Node {
        public static final Node SINK = new Node(0, Type.SINK);
        public static final Node SOURCE = new Node(0, Type.SOURCE);
        final int value;
        final Type type;

        enum Type {
            SINK,
            SOURCE,
            PROBLEM,
            CATEGORY
        }

        public Node(int v, Type type) {
            this.value = v;
            this.type = type;
        }

        @Override
        public boolean equals(Object o) {
            if (this == o) return true;
            if (o == null || getClass() != o.getClass()) return false;
            Node node = (Node) o;
            return value == node.value &&
                    type == node.type;
        }

        @Override
        public int hashCode() {
            return Objects.hash(value, type);
        }
    }
```

```
    static class Edge {
        final Node start;
        final Node end;
        final int capacity;
        int flow;
        int residual;
        Edge reversed;

        public Edge(Node start, Node end, int capacity, int flow, int residual) {
            this.start = start;
            this.end = end;
            this.capacity = capacity;
            this.flow = flow;
            this.residual = residual;
        }
    }

    private void add(Node from, Node to, int capacity) {
        graph.putIfAbsent(from, new ArrayList<>());
        graph.putIfAbsent(to, new ArrayList<>());
        Edge toEdge = new Edge(from, to, capacity, 0, capacity);
        Edge fromEdge = new Edge(to, from, capacity, 0, 0);
        toEdge.reversed = fromEdge;
        fromEdge.reversed = toEdge;
        graph.get(from).add(toEdge);
        graph.get(to).add(fromEdge);
    }

    ⟨10.8 Solver 222⟩

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = 0;
        int m = 0;
        while (true) {
            TheProblemWithTheProblemSetter solver = new TheProblemWithTheProblemSetter();
            m = s.nextInt();
            n = s.nextInt();
            if (m == 0 && n == 0) {
                break;
            }
            int targetFlow = 0;
            ⟨10.8 Input 221a⟩
            ⟨10.8 Output 221b⟩
        }
    }
}
```

Now let's add code to read the input. We will all also calculate a target `maximumFlow`. If our maximum flow is less than this value, then it means there's no solution.

221a ⟨*10.8 Input* 221a⟩≡

```
for (int i = 1; i <= m; ++i) {
    int capacity = s.nextInt();
    targetFlow += capacity;
    solver.add(new Node(i, Node.Type.CATEGORY), Node.SINK, capacity);
}
for (int i = 1; i <= n; ++i) {
    Node node = new Node(i, Node.Type.PROBLEM);
    solver.add(Node.SOURCE, node, 1);
    int neighbors = s.nextInt();
    solver.add(node, Node.SOURCE, 1);
    while (neighbors > 0) {
        Node toNode = new Node(s.nextInt(), Node.Type.CATEGORY);
        solver.add(node, toNode, 1);
        neighbors--;
    }
}
```

Output code is trivial. To read off the solution, we access `graph` of the solver. It will have flow information in it for each edge.

221b ⟨*10.8 Output* 221b⟩≡

```
if (solver.solve() == targetFlow) {
    System.out.println(1);
    for (int i = 1; i <= m; ++i) {
        Node category = new Node(i, Node.Type.CATEGORY);
        List<Edge> edges = solver.graph.get(category);
        boolean first = true;
        for (Edge p : edges) {
            Edge edge = p.reversed;
            if (edge.end.equals(category) && edge.flow == edge.capacity) {
                System.out.print((first ? "" : " ") + edge.start.value);
                first = false;
            }
        }
        System.out.println();
    }
} else {
    System.out.println(0);
}
```

Now the solver part. To find a maximum flow I'm going to use Edmonds-Karp algorithm. The idea is that we keep finding augmenting paths and increase the flow by 1 in the path. We don't need to find the bottlenecks in the augmenting paths because due to how we construct the graph it is always going to be 1. We keep doing it until there are augmenting paths. Once we can't find an augmenting path, we are done.

222 ⟨*10.8 Solver* 222⟩≡

```
private int solve() {
    int flow = 0;
    while (true) {
        Deque<Node> q = new ArrayDeque<>();
        q.push(Node.SOURCE);
        Set<Node> visited = new HashSet<>();
        Map<Node, Edge> path = new HashMap<>();
        while (!q.isEmpty() && !path.containsKey(Node.SINK)) {
            Node n = q.pollFirst();
            visited.add(n);
            for (Edge e : graph.get(n)) {
                if (!visited.contains(e.end) && e.residual > 0 &&
                        !e.end.equals(Node.SOURCE)) {
                    q.addLast(e.end);
                    path.put(e.end, e);
                }
            }
        }
        if (!path.containsKey(Node.SINK)) {
            return flow;
        }
        Edge edge = path.get(Node.SINK);
        while (edge != null) {
            edge.flow++;
            edge.residual--;
            edge.reversed.flow--;
            edge.reversed.residual++;
            edge = path.get(edge.start);
        }
        flow++;
    }
}
```

11 Dynamic Programming

11.1 Is Bigger Smarter?

The task is pretty simple: We iterate on the sorted input data and try to extend the sequence. We keep two arrays, `res`, to keep the lengths of the sequences, and `ref`, to keep the references to the previous elements.

223 ⟨*Is Bigger Smarter* 223⟩≡

```
import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Scanner;

public class IsBiggerSmarter {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));

    public static int[] solve(List<int[]> input) {
        if (input.size() == 1) {
            return new int[] { 0 };
        }
        List<int[]> sorted = new ArrayList<>(input);
        Collections.sort(sorted, (x, y) -> Integer.compare(x[0], y[0]));
        int[] res = new int[sorted.size()];
        int[] ref = new int[sorted.size()];
        int index = 0;

        for (int i = 0; i < sorted.size(); ++i) {
            res[i] = 1;
            ref[0] = -1;
            for (int j = 0; j < i; ++j) {
                if (sorted.get(j)[0] < sorted.get(i)[0] &&
                        sorted.get(j)[1] > sorted.get(i)[1]) {
                    if (res[i] < res[j] + 1) {
                        res[i] = res[j] + 1;
                        ref[i] = j;
                        if (index == -1 || res[i] > res[index]) {
                            index = i;
                        }
                    }
                }
            }
        }

        int[] solution = new int[res[index]];
        int i = solution.length - 1;
        do {
            solution[i] = sorted.get(index)[2];
            index = ref[index];
            i--;
        } while (index != -1 && i >= 0);
```

```
        return solution;
    }

    public static void main(String[] args) throws Exception {
        String currentLine;
        List<int[]> input = new ArrayList<>();
        int i = 0;
        while ((currentLine = reader.readLine()) != null) {
            Scanner s = new Scanner(currentLine);
            input.add(new int[] { s.nextInt(), s.nextInt(), i });
            s.close();
            i++;
        }
        int[] solution = solve(input);
        System.out.println(solution.length);
        for (int idx : solution) {
            System.out.println(idx + 1);
        }
    }
}
```

11.2 Distinct Subsequences

Because the input is very simple, let me do that first.

224 ⟨*Distinct Subsequences* 224⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.math.BigInteger;

public class Main {
    ⟨11.2 Slow Recursive 225⟩
    ⟨11.2 Fast Tabular 226⟩
    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        int n = Integer.parseInt(reader.readLine().trim());
        for (int i = 0; i < n; ++i) {
            String line = reader.readLine().trim();
            String subs = reader.readLine().trim();
            Main solver = new Main();
            //System.out.println(solver.slowRecursive(line, subs));
            System.out.println(solver.fastTabular(line, subs));
        }
    }
}
```

Now, a recursive solution: Sequentially scan for the last character of Z in X. Suppose its index is i. Now recursively call the function with a substring of $X[1, i-1]$ and Z', which is the same as Z except its last character removed. The answer is the sum of the returned values from the recursive calls for each such index i. This solution is slow (even with memoization) and the online judge will not accept it, because of the time limit. But the solution is correct and instructive.

225 ⟨*11.2 Slow Recursive* 225⟩≡

```
private BigInteger slowRecursive(String line, String subs) {
    BigInteger[][] memo = new BigInteger[line.length()+1][subs.length()+1];
    return slowRecursive(line, subs, memo);
}

private BigInteger slowRecursive(String line, String subs, BigInteger[][] memo) {
    BigInteger count = BigInteger.ZERO;

    if (memo[line.length()][subs.length()] != null) {
        return memo[line.length()][subs.length()];
    }

    for (int i = line.length() - 1; i >= 0; --i) {
        if (line.charAt(i) == subs.charAt(subs.length() - 1)) {
            if (subs.length() == 1) {
                count = count.add(BigInteger.ONE);
            } else {
                count = count.add(slowRecursive(line.substring(0, i),
                        subs.substring(0, subs.length() - 1), memo));
            }
        }
    }

    memo[line.length()][subs.length()] = count;
    return count;
}
```

		b	**a**	**b**	**g**	**b**	**a**	**g**
	0	0	0	0	0	0	0	0
b	0	1	1	2	2	3	3	3
a	0	0	1	1	1	1	4	4
g	0	0	0	0	1	1	1	5

A tabular solution. Let's create a table `memo` of $|X|$ by $|Z|$ size. We will fill out the first row and the first column with zeroes and we will assume our indexing starts from 1 because it is easier to maintain the table that way. Then, we start going through Z character by character and fill out the table row by row. If it's the second row, then we just put 1 on the first occurrence of the matching characters and keep increasing it on subsequent matches. The second row answers the question of what if our Z had only the first character. So we just count how many times it occurs in X. The third row (and all subsequent rows) of the table, which corresponds to the case when Z has more than one character, will be constructed in a different way. First, if characters don't match for the given cell, we simply take the previous value to this cell. It basically says that the number of subsequences hasn't changed for this longer X. But if the characters match, we calculate the value by taking $memo[i-1][j-1]+memo[j-1][i]$. Essentially we are saying that the number of subsequences of $Z[1,i]$ in $X[1,j]$ is the same as the number of subsequences of $Z[1,i-1]$ in $X[1,j-1]$ plus the number of subsequences of $Z[1,i]$ in $X[1,j-1]$. Once we have completed filling out the table, it will contain the answers for all pairs of lengths of X and Z.

226 ⟨*11.2 Fast Tabular* 226⟩≡

```
private BigInteger fastTabular(String line, String subs) {
    BigInteger[][] memo = new BigInteger[line.length() + 1][subs.length() + 1];
    for (int i = 0; i <= subs.length(); ++i) {
        memo[0][i] = BigInteger.ZERO;
    }
    for (int i = 0; i <= line.length(); ++i) {
        memo[i][0] = BigInteger.ZERO;
    }
    for (int i = 1; i <= subs.length(); ++i) {
        for (int j = 1; j <= line.length(); ++j) {
            if (line.charAt(j - 1) == subs.charAt(i - 1)) {
                if (i == 1) {
                    memo[j][i] = memo[j - 1][i].add(BigInteger.ONE);
                } else {
                    memo[j][i] = memo[j - 1][i].add(memo[j - 1][i - 1]);
                }
            } else {
                memo[j][i] = memo[j - 1][i];
            }
        }
    }
    return memo[line.length()][subs.length()];
}
```

11.3 Weights and Measures

Let's say we are given $T_1, \ldots, T_n$ turtles, and two functions $weight(T_i)$ and $strength(T_i)$. We are asked to find a largest stack of turtles such that for each turtle in this stack its capacity (i.e. strength minus its own weight) is larger or equal to the weights of all turtles above it.

Let us take a random proper stack of turtles $T_{t_1}, \ldots, T_{t_m}$. Since this is a proper stack, it means that for all $i \leq m$ the following inequality holds:

$$\sum_{j=1}^{i-1} weight(T_{t_j}) \leq strength(T_{t_i}) - weight(T_{t_i})$$

In order to allow for recursive construction we need to define an ordering of turtles in our stacks. Let's assume we have a proper stack of turtles $T_{t_1}, \ldots, T_{t_i}, T_{t_{i+1}}, \ldots, T_{t_m}$, and let's also assume that $strength(T_{t_{i+1}}) < strength(T_{t_i})$. (Remember: $strength(T_i)$ returns turtle's strength which is how much this turtle can carry including its own weight.) But note that $T_{t_{i+1}}$ and T_{t_i} can be swapped and a new stack of turtles would still be a proper stack. That is $T_{t_1}, \ldots, T_{t_{i+1}}, T_{t_i}, \ldots, T_{t_m}$ would be a proper stack. This observation means that we can take any proper stack and change it to a non-decreasing stack with respect to the strengths, which will also be a proper stack. (For a formal proof of this claim see [17].)

This also means that once we have sorted our input by strengths, we can proceed along the sorted list of turtles and obtain proper ordered stacks.

The sub-problem S_j, $j \leq n$ is the following: Among all the stacks of height $k \leq j$ built from turtles $T_1, \ldots, T_j$ find the lightest one. Let's assume we have solved S_{j-1}. If we want to build the lightest stack from $T_1, \ldots, T_j$ of length k, let's call it W_k^j, we look at the lightest stack of lengths k and $k-1$ build from $T_1, \ldots, T_{j-1}$. Let's call them W_k^{j-1} and W_{k-1}^{j-1} correspondingly. If we can extend the stack W_{k-1}^{j-1} with T_j and it is lighter than W_k^{j-1}, then that's our lightest stack built from $T_1, \ldots, T_j$, otherwise we let $W_k^j = W_k^{j-1}$ (see [17]).

227 ⟨*Weights and Measures* 227⟩≡

```
import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.Comparator;
import java.util.List;
import java.util.Scanner;

public class WeightsAndMeasures {
    private static class Turtle {
        int weight;
        int strength;

        public Turtle(int weight, int strength) {
            this.weight = weight;
            this.strength = strength;
        }
    }

    public static int solve(List<Turtle> input) {
        input.sort(Comparator.comparingInt(x -> x.strength));
        int[][] W = new int[2][input.size() + 1];
```

```
        W[0][0] = 0;
        for (int i = 1; i <= input.size(); ++i) {
            W[0][i] = Integer.MAX_VALUE;
            W[1][i] = Integer.MAX_VALUE;
        }

        for (int i = 1; i <= input.size(); ++i) {
            Turtle t = input.get(i - 1);
            for (int k = i; k >= 1; --k) {
                if (W[0][k - 1] <= (t.strength - t.weight)) {
                    W[1][k] = Math.min(W[0][k], W[0][k - 1] + t.weight);
                } else {
                    W[1][k] = W[0][k];
                }
            }
            System.arraycopy(W[1], 0, W[0], 0, W[1].length);
            for (int k = 1; k <= input.size(); ++k) {
                W[1][k] = Integer.MAX_VALUE;
            }
        }
        for (int i = input.size(); i >= 0; --i) {
            if (W[0][i] < Integer.MAX_VALUE) {
                return i;
            }
        }
        return 1;
    }


    public static void main(String[] args) throws IOException {
        BufferedReader reader = new BufferedReader(new InputStreamReader(System.in));
        String line;
        List<Turtle> input = new ArrayList<>();
        while ((line = reader.readLine()) != null) {
            Scanner scanner = new Scanner(line);
            input.add(new Turtle(scanner.nextInt(), scanner.nextInt()));
        }
        System.out.println(solve(input));
    }
}
```

11.4 Unidirectional TSP

The difficulty with this task is how to reconstruct the path in lexicographic order. That tricky thing to realize is that we should start from the end rather than start. For each cell we inspect the next cells with the minimum path and choose the minimum. If there are multiple choices, we prefer the one that has the smaller row number. For each current cell we also store the references to the cell with the minimum path. Keep doing it column by column. Once we at the first column, we identify the cell with the minimum path and start following the `next` references. That resulting path is the answer.

229 ⟨*Unidirectional TSP* 229⟩≡

```
import java.io.IOException;
import java.util.Scanner;

import static java.util.Arrays.stream;
import static java.util.stream.Collectors.joining;

public class UnidirectionalTSP {
    private static class Cell {
        Cell next;
        int row;
        int value;
        int minPath;

        public Cell(int value, int row) {
            this.value = value;
            this.minPath = Integer.MAX_VALUE;
            this.row = row;
        }
    }

    final Cell[][] table;
    final int m;
    final int n;
    int minPath;
    int[] path;

    public UnidirectionalTSP(Cell[][] table, int m, int n) {
        this.table = table;
        this.m = m;
        this.n = n;
        this.path = new int[n];
        this.minPath = Integer.MAX_VALUE;
        solve();
    }

    private Cell get(int row, int col) {
        if (row > m - 1) {
            return table[0][col];
        } else if (row < 0) {
            return table[m - 1][col];
        }
        return table[row][col];
```

```
    }

    private void solve() {
        for (int row = 0; row < m; ++row) {
            Cell cell = get(row, n - 1);
            cell.minPath = cell.value;
            cell.next = null;
        }
        for (int col = n - 2; col >= 0; --col) {
            for (int row = 0; row < m; ++row) {
                Cell cell = get(row, col);
                int min = Integer.MAX_VALUE;
                for (int i = -1; i <= 1; ++i) {
                    Cell next = get(row + i, col + 1);
                    if (min > next.minPath) {
                        min = next.minPath;
                        cell.next = next;
                    } else if (min == next.minPath) {
                        if (cell.next == null || (cell.next != null && cell.next.row > next.row)) {
                            cell.next = next;
                        }
                    }
                }
                cell.minPath = min + cell.value;
            }
        }
        Cell cell = null;
        for (int row = 0; row < m; ++row) {
            Cell curr = get(row, 0);
            if (cell == null || (cell != null && curr.minPath < cell.minPath)) {
                cell = curr;
            }
        }

        this.minPath = cell.minPath;
        for (int col = 0; col < n; ++col) {
            path[col] = cell.row + 1;
            cell = cell.next;
        }
    }

    public static void main(String[] args) throws IOException {
        Scanner s = new Scanner(System.in);
        while (s.hasNextInt()) {
            int m = s.nextInt(); // rows
            int n = s.nextInt(); // cols
            Cell[][] table = new Cell[m][n];
            for (int i = 0; i < m * n; ++i) {
                table[i / n][i % n] = new Cell(s.nextInt(), i / n);
            }
            UnidirectionalTSP main = new UnidirectionalTSP(table, m, n);
            System.out.println(stream(main.path).mapToObj(String::valueOf).collect(joining(" ")));
            System.out.println(main.minPath);
```

```
            }
        }

    }
```

11.5 Cutting Sticks

It's an easy task: At every cut point we recursively calculate the minimum cost for the two remaining sticks, so their sum plus length of the current stick gives us the total cost. Once we have tried every cut point in this way we select the one with the minimum cost. With a simple memoization the program runs within allowed time limit.

232 ⟨*Cutting Sticks* 232⟩≡

```
import java.util.Scanner;

public class CuttingSticks {
    private final int[] cuts;
    private final int length;
    private final Integer[][] memo = new Integer[1000][1000];

    public CuttingSticks(int[] cuts, int length) {
        this.cuts = cuts;
        this.length = length;
    }

    public int solve() {
        return solve(0, length);
    }

    private int solve(int start, int end) {
        if (memo[start][end] != null) {
            return memo[start][end];
        }
        int min = Integer.MAX_VALUE;
        for (int i = 0; i < cuts.length; ++i) {
            if (cuts[i] > start && cuts[i] < end) {
                min = Math.min(min, solve(start, cuts[i]) + solve(cuts[i], end));
            }
        }
        memo[start][end] = min < Integer.MAX_VALUE ? (end - start) + min : 0;
        return memo[start][end];
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        while (true) {
            int length = s.nextInt();
            if (length == 0) {
                break;
            }
            int n = s.nextInt();
            int[] cuts = new int[n];
            for (int i = 0; i < n; ++i) {
                cuts[i] = s.nextInt();
            }
            CuttingSticks solver = new CuttingSticks(cuts, length);
            System.out.println(String.format("The minimum cutting is %d.", solver.solve()));
        }
    }
```

```
}
```

11.6 Ferry Loading

A naive recursive implementation with memoization solves this task. We put the first car in the queue on the left side of the ferry and try recursively to obtain a maximum loading solution using the remaining queue and spaces. We put the same car on the right side of the ferry and try recursively again to obtain a maximum loading solution. Then out of these two solutions we get the maximum one and return it. Note that the total lengths of the cars on both sides of the ferry and the total number of cars on the ferry can be used as a key to the table to lookup the solutions for the remaining spaces and the remaining cars in the queue. And because both sides are equal in sizes, the solutions are symmetric. This means we can order lengths in increasing order while looking up the table or putting a value into it.

234 ⟨*Ferry Loading* 234⟩≡

```
import java.util.*;

public class FerryLoading {
    private final List<Integer> queue;
    private final int ferryLength;
    private final Map<Integer, Map<Integer, Map<Integer, Deque<Boolean>>>> memo = new HashMap<>();

    public FerryLoading(int ferryLength, List<Integer> queue) {
        this.ferryLength = ferryLength * 100;
        this.queue = queue;
    }

    private Deque<Boolean> solve() {
        return solve(0, new ArrayDeque<>(), 0, 0);
    }

    private Deque<Boolean> solve(int i, Deque<Boolean> solution, int leftLength, int rightLength) {
        if (i > queue.size() - 1) {
            return new ArrayDeque<>(solution);
        }
        Integer next = queue.get(i);
        int min = Math.min(leftLength, rightLength);
        int max = Math.max(leftLength, rightLength);

        if (memo.get(min) != null) {
            Map<Integer, Deque<Boolean>> m = memo.get(min).get(max);
            if (m != null && m.containsKey(i)) {
                return m.get(i);
            }
        }

        Deque<Boolean> leftSolution = null;
        if (next + leftLength <= ferryLength) {
            solution.addLast(true);
            leftSolution = solve(i + 1, solution, next + leftLength, rightLength);
            solution.removeLast();
        }

        Deque<Boolean> rightSolution = null;
        if (next + rightLength <= ferryLength) {
```

```
            solution.addLast(false);
            rightSolution = solve(i + 1, solution, leftLength, next + rightLength);
            solution.removeLast();
        }

        Deque<Boolean> finalSolution = null;
        if (leftSolution == null && rightSolution == null) {
            finalSolution = new ArrayDeque<>(solution);
        } else if (leftSolution == null || rightSolution == null) {
            finalSolution = leftSolution == null ? rightSolution : leftSolution;
        } else {
            finalSolution = leftSolution.size() > rightSolution.size() ? leftSolution : rightSolution;
        }

        memo.putIfAbsent(min, new HashMap<>());
        memo.get(min).putIfAbsent(max, new HashMap<>());
        memo.get(min).get(max).putIfAbsent(i, finalSolution);
        return finalSolution;
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        int n = s.nextInt();
        for (int i = 0; i < n; i++) {
            List<Integer> queue = new ArrayList<>();
            int ferry = s.nextInt();
            int k;
            while ((k = s.nextInt()) != 0) {
                queue.add(k);
            }
            FerryLoading solver = new FerryLoading(ferry, queue);
            Deque<Boolean> solution = solver.solve();
            System.out.println(solution.size());
            solution.stream().map(x -> x ? "starboard" : "port").forEach(System.out::println);
            if (i < n - 1) {
                System.out.println();
            }
        }
    }
}
```

5	*25*	24	23	22	$\underline{21}$
4	10	11	12	$\underline{13}$	20
3	*9*	8	$\underline{7}$	14	19
2	2	$\underline{3}$	6	15	18
1	$\underline{1}$	*4*	5	*16*	17
	1	**2**	**3**	**4**	**5**

12 Grids

12.1 Ant on a Chessboard

It's easy to note that each stripe k starts with k^2. This means that we can obtain stripes' number by taking the ceiling of the value of the square root of the input value, that is for any value v we have $k = \lceil \sqrt{v} \rceil$. This tells us one of the coordinates, but whether that's x or y depends on whether k is odd or even.

The values in italics in the table are equal to $m = k^2 - k + 1$. Now once we know k and m we can work out the coordiantes. If v equals m it's the "corner" and so both x and y are the same, that is equal to k. Otherwise, depending on whether k is odd or even, we can work out the coordinates for each case. For example, if k is odd and $v \geq m$, then $x = k^2 - v + 1$. (You can convince yourself by looking at the table.) Similarly we can work out for the rest of the cases, that is when $v < m$ and when k is either odd or even.

236 ⟨*Ant On a Chessboard* 236⟩≡

```
import java.util.Scanner;

public class AntOnAChessboard {
    private static long[] get(long v) {
        long k = (long) Math.ceil(Math.sqrt(v));
        long m = k * k - k + 1;
        boolean isOdd = k % 2 == 1;
        if (v == m) {
            return new long[]{k, k};
        } else if (v >= m) {
            if (isOdd) {
                return new long[]{k * k - v + 1, k};
            }
            return new long[]{k, k * k - v + 1};
        } else {
            if (isOdd) {
                return new long[]{k, k - (m - v)};
            }
            return new long[]{k - (m - v), k};
        }
    }

    public static void main(String[] args) {
        Scanner s = new Scanner(System.in);
        while (true) {
            long v = s.nextLong();
            if (v == 0) {
                break;
```

```
            }
            long[] ans = get(v);
            System.out.println(String.format("%d %d", ans[0], ans[1]));
        }
    }
}
```

13 Geometry

14 Computational Geometry

15 License

References

[1] Donald E. Knuth, Literate Programming, The Computer Journal, 1984

[2] Skiena, Steven S., Revilla, Miguel A., Programming Challenges, 2003

[3] `https://en.wikipedia.org/wiki/Sieve_of_Eratosthenes`

[4] `https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm`

[5] `https://en.wikipedia.org/wiki/Modular_exponentiation`

[6] Gareth A. Jones, Josephine M. Jones, Elementary Number Theory, 1998

[7] Slocum, Jerry and Weisstein, Eric W. "15 Puzzle." From MathWorld–A Wolfram Web Resource. http://mathworld.wolfram.com/15Puzzle.html

[8] `https://en.wikipedia.org/wiki/Iterative_deepening_A*`

[9] `https://en.wikipedia.org/wiki/Floyd-Warshall_algorithm`

[10] `http://masc.cs.gmu.edu/wiki/floydwarshall/`

[11] `https://github.com/alex-lange/contest-programming/blob/master/uva/10039/Main.java`

[12] `http://www.cs.cornell.edu/~wdtseng/icpc/notes/dp3.pdf`

[13] `https://github.com/morris821028`

[14] `https://oeis.org/A018835`

[15] `https://en.wikipedia.org/wiki/Disjoint-set_data_structure`

[16] `https://en.wikipedia.org/wiki/Biconnected_component`

[17] `https://cgi.cse.unsw.edu.au/~cs3121/Lectures/COMP3121_Lecture_Notes_DP.pdf`

Definitions

⟨*8.1 Backtrack* 140⟩ 137, 140
⟨*8.4 Backtrack* 153b⟩ 150, 153b
⟨*4.6 Billing* 82b⟩ 82a, 82b, 82c, 82d
⟨*8.4 Bits Class* 151⟩ 150, 151
⟨*1.3 Calculation* 6c⟩ 6a, 6c
⟨*5.4 Calculation* 94a⟩ 93, 94a, 94b
⟨*2.8 Categorize* 46a⟩ 44a, 46a, 46b, 46c
⟨*8.8 Classes* 169⟩ 166, 169, 171, 173
⟨*1.2 Constants* 4c⟩ 4a, 4c
⟨*1.4 Constants* 8b⟩ 7d, 8b
⟨*2.8 Constants* 43a⟩ 42, 43a, 43b
⟨*1.1 Constructor* 3a⟩ 1a, 3a
⟨*2.4 Constructor* 31g⟩ 30, 31g
⟨*2.8 Constructor* 47a⟩ 42, 47a

⟨*7.1 Constructor* 122b⟩ 121a, 122b
⟨*8.4 Constuctor* 153a⟩ 150, 153a
⟨*1.4 Convertion* 9b⟩ 7d, 9b
⟨*1.8 Election loop* 21f⟩ 21d, 21f, 21g, 21h, 21i, 22a
⟨*11.2 Fast Tabular* 226⟩ 224, 226
⟨*8.4 Fields* 152b⟩ 150, 152b
⟨*8.7 Fields* 162⟩ 159, 162, 163a
⟨*8.7 Find* 165⟩ 159, 165
⟨*4.6 Find Interval* 81⟩ 79, 81
⟨*1.3 Finding the minimum* 6e⟩ 6c, 6e, 7a
⟨*4.6 Get Bills* 82a⟩ 79, 82a
⟨*4.3 Get Strategy* 71d⟩ 70, 71d, 72a, 72b, 73
⟨*8.4 Graphs* 152a⟩ 150, 152a
⟨*1.1 Helpers* 2b⟩ 1a, 2b
⟨*1.4 Helpers* 9d⟩ 9b, 9d, 10a, 10c, 10e
⟨*2.8 Helpers* 43c⟩ 42, 43c
⟨*3.6 Helpers* 60e⟩ 59, 60e
⟨*8.2. IDA* Search* 145⟩ 141, 145
⟨*1.8 Implementation* 21a⟩ 20, 21a, 21b, 21c, 21d
⟨*3.6 Implementation* 60b⟩ 59, 60b, 60d, 60f
⟨*4.7 Implementation* 84⟩ 83, 84
⟨*7.1 Implementation* 123⟩ 121a, 123
⟨*1.1 Imports* 1b⟩ 1a, 1b, 2a, 2c, 3b
⟨*1.2 Imports* 4b⟩ 4a, 4b, 4d
⟨*1.3 Imports* 6b⟩ 6a, 6b, 6d, 7b
⟨*1.4 Imports* 8a⟩ 7d, 8a, 9c, 10b, 10d, 10g, 11b
⟨*1.8 Imports* 21e⟩ 20, 21e
⟨*2.4 Imports* 31a⟩ 30, 31a, 31c, 31e, 31h, 32c, 33b, 34c, 35b
⟨*3.6 Imports* 60a⟩ 59, 60a, 60c
⟨*7.1 Imports* 121b⟩ 121a, 121b
⟨*10.8 Input* 221a⟩ 219, 221a
⟨*1.1 Input/Output* 3c⟩ 1a, 3c
⟨*1.3 Input/Output* 7c⟩ 6a, 7c
⟨*1.4 Input/Output* 9a⟩ 7d, 9a
⟨*8.4 Input/Output* 154b⟩ 150, 154b
⟨*8.4 Invoke Backtrack* 154a⟩ 153a, 154a
⟨*1.2 Main* 5⟩ 4a, 5
⟨*8.8 Main* 178⟩ 166, 178
⟨*10.3 Methods* 207⟩ 205, 207, 208a
⟨*2.4 Methods* 31b⟩ 30, 31b, 31d, 32a, 33a, 34a, 34b, 35a, 35c
⟨*2.8 Methods* 47b⟩ 42, 47b, 47c, 48, 49, 50
⟨*8.8 Methods* 175a⟩ 166, 175a, 175b, 176, 177
⟨*1.4 Middle Column Construction* 10f⟩ 10e, 10f
⟨*8.2. Node Class* 143⟩ 141, 143
⟨*10.8 Output* 221b⟩ 219, 221b
⟨*8.7 Precompute* 163b⟩ 159, 163b, 164
⟨*4.3 Print Result* 71a⟩ 70, 71a, 71b, 71c
⟨*1.4 Process* 11a⟩ 9b, 11a

⟨*15 Puzzle Problem* 141⟩ 141
⟨*1.4 Return* 11c⟩ 9b, 11c
⟨*2.8 Round Constructor* 44a⟩ 43c, 44a
⟨*2.8 Round Methods* 44b⟩ 43c, 44b, 44c, 45a, 45b, 46d
⟨*11.2 Slow Recursive* 225⟩ 224, 225
⟨*10.3 Solver* 208b⟩ 205, 208b
⟨*10.8 Solver* 222⟩ 219, 222
⟨*8.7 State Class* 161⟩ 159, 161
⟨*2.4 Variables* 31f⟩ 30, 31f, 32b, 32d, 32e
⟨*7.1 Variables* 122a⟩ 121a, 122a
⟨*A Multiplication Game* 95⟩ 95
⟨*Ant On a Chessboard* 236⟩ 236
⟨*Australian Voting* 20⟩ 20
⟨*Automated Judge Script* 58⟩ 58
⟨*Bicoloring* 181⟩ 181
⟨*Bigger Square Please Meta* 179⟩ 179
⟨*Bigger Square Please Offline* 166⟩ 166
⟨*Bridge* 70⟩ 70
⟨*CDVII* 79⟩ 79
⟨*Carmichael Numbers* 124⟩ 124
⟨*Check The Check* 17⟩ 17
⟨*Colours Hash* 159⟩ 159
⟨*Common Permutation* 54⟩ 54
⟨*Complete Tree Labeling* 112⟩ 112
⟨*Contest Scoreboard* 40⟩ 40
⟨*Counting* 108⟩ 108
⟨*Crypt Kicker* 30⟩ 30
⟨*Crypt Kicker II* 56⟩ 56
⟨*Cutting Sticks* 232⟩ 232
⟨*Distinct Subsequences* 224⟩ 224
⟨*Doublets* 61⟩ 61
⟨*Edit Step Ladders* 191⟩ 191
⟨*Erdos Numbers* 38⟩ 38
⟨*Euclid Problem* 126⟩ 126
⟨*Expressions* 110⟩ 110
⟨*Factovisors* 128⟩ 128
⟨*Ferry Loading* 234⟩ 234
⟨*File Fragmentation* 59⟩ 59
⟨*Fire Stations* 205⟩ 205
⟨*Fmt* 64⟩ 64
⟨*Football aka Soccer* 85⟩ 85
⟨*Freckles* 200⟩ 200
⟨*From Dusk Till Dawn* 196⟩ 196
⟨*Garden of Eden* 157⟩ 157
⟨*Graphical Editor* 12⟩ 12
⟨*Hanoi Tower Troubles Again* 199⟩ 199
⟨*Hartals* 29⟩ 29
⟨*How Many Fibs* 105⟩ 105

⟨*How Many Pieces of Land* 106⟩ 106
⟨*Interpreter* 15⟩ 15
⟨*Is Bigger Smarter* 223⟩ 223
⟨*Jolly Jumpers* 22b⟩ 22b
⟨*LC Display* 7d⟩ 7d
⟨*Light, More Light* 121a⟩ 121a
⟨*Little Bishops* 137⟩ 137
⟨*Longest Nap* 74⟩ 74
⟨*Marbles* 134⟩ 134
⟨*Minesweeper* 4a⟩ 4a
⟨*Ones* 93⟩ 93
⟨*Pairsumonious Numbers* 101⟩ 101
⟨*Playing With Wheels* 183⟩ 183
⟨*Poker Hands* 23⟩ 23
⟨*Polynomial Coefficients* 97⟩ 97
⟨*Primary Arithmetic* 89⟩ 89
⟨*Queue* 147⟩ 147
⟨*Railroads* 209⟩ 209
⟨*Reverse And Add* 91⟩ 91
⟨*Self Describing Sequence* 117⟩ 117
⟨*Servicing Stations* 150⟩ 150
⟨*ShellSort* 83⟩ 83
⟨*Shoemakers Problem* 77⟩ 77
⟨*Slash Maze* 188⟩ 188
⟨*Smith Numbers* 132⟩ 132
⟨*Stack em Up* 36⟩ 36
⟨*Stacks of Flapjacks* 68⟩ 68
⟨*Steps* 119⟩ 119
⟨*Summation of Four Primes* 130⟩ 130
⟨*The Archeologists Dilemma* 92⟩ 92
⟨*The Grand Dinner* 217⟩ 217
⟨*The Necklace* 202⟩ 202
⟨*The Priest Mathematician* 115⟩ 115
⟨*The Problem with the Problem Setter* 219⟩ 219
⟨*The Stern-Brocot Number System* 99⟩ 99
⟨*The Tourist Guide* 186⟩ 186
⟨*The Trip* 6a⟩ 6a
⟨*Tourist Guide* 215⟩ 215
⟨*Tower of Cubes* 193⟩ 193
⟨*Tug Of War* 155⟩ 155
⟨*Unidirectional TSP* 229⟩ 229
⟨*Vitos Family* 67⟩ 67
⟨*WERTYU* 51⟩ 51
⟨*War* 212⟩ 212
⟨*Weights and Measures* 227⟩ 227
⟨*Where is Waldorf* 52⟩ 52
⟨*Yahtzee* 42⟩ 42
⟨*3n+1* 1a⟩ 1a

Index