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 Chapter 1 **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 . . . 11
1.6 Interpreter . . . 14
1.7 Check The Check . . . 16
1.8 Australian Voting . . . 20

2 Chapter 2 **22**
2.1 Jolly Jumper . . . 22
2.2 Hartals . . . 23
2.3 Crypt Kicker . . . 24
2.4 Stack 'em Up . . . 30
2.5 Erdös Numbers . . . 32
2.6 Contest Scoreboard . . . 34
2.7 Yahtzee . . . 37

3 Chapter 3 **46**
3.1 WERTYU . . . 46
3.2 Crypt Kicker II . . . 47
3.3 File Fragmentation . . . 49

4 Chapter 4 **51**
4.1 Bridge . . . 51
4.2 ShellSort . . . 55

5 Chapter 5 **57**
5.1 Primary Arithmetic . . . 57
5.2 Reverse And Add . . . 59
5.3 The Archeologists' Dilemma . . . 60
5.4 Ones . . . 62
5.5 A Multiplication Game . . . 64

[1] If you are going to submit any of these programs to the UVa Online Judge make sure the class name is Main and that it's not in any package; For the class names I use problem names, and I put everything into my package com.rvprg.pc)

5.6 Polynomial Coefficients 66
5.7 The Stern-Brocot Number System 68
5.8 Pairsumonious Numbers 70

6 Chapter 7 **70**
6.1 Light, More Light 70

7 License **73**

1 Chapter 1

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⟩≡
```
package com.rvprg.pc;

⟨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;
```

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⟩≡
```
package com.rvprg.pc;

⟨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;
```

[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.

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⟩≡
```
package com.rvprg.pc;

⟨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⟩≡

```
package com.rvprg.pc;

⟨1.4 Imports 8a⟩

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

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:

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

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.

8c ⟨*1.4 Input/Output* 8c⟩≡
```
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 is to implement `segments`.

9a ⟨*1.4 Convertion* 9a⟩≡
```
private static List<String> segments(final String digits, final int scale) {
    ⟨1.4 Helpers 9c⟩
    ⟨1.4 Process 10e⟩
    ⟨1.4 Return 10g⟩
}
```

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:

9b ⟨*1.4 Imports* 8a⟩+≡
```
import static java.lang.String.join;
import static java.util.Collections.nCopies;
import java.util.function.Function;
```

9c ⟨*1.4 Helpers* 9c⟩≡
```
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:

9d ⟨*1.4 Helpers* 9c⟩+≡
```
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`:

9e ⟨*1.4 Imports* 8a⟩+≡
```
import static java.util.stream.Collectors.joining;
import java.util.function.BiFunction;
```

9f ⟨*1.4 Helpers* 9c⟩+≡
```
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).

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

10b ⟨*1.4 Helpers* 9c⟩+≡
```
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 10c⟩,
  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:

10c ⟨*1.4 Middle Column Construction* 10c⟩≡
```
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:

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

10e ⟨*1.4 Process* 10e⟩≡
```
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:

10f ⟨*1.4 Imports* 8a⟩+≡
```
import static java.util.stream.IntStream.range;
import static java.util.stream.IntStream.rangeClosed;
```

10g ⟨*1.4 Return* 10g⟩≡
```
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.

11 ⟨*Graphical Editor* 11⟩≡

```
package com.rvprg.pc;

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.

14 ⟨*Interpreter* 14⟩≡

```
package com.rvprg.pc;

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.

16 ⟨*Check The Check* 16⟩≡

```
package com.rvprg.pc;

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⟩≡

```
package com.rvprg.pc;

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 Chapter 2

2.1 Jolly Jumper

This task must be a joke.

22b ⟨*Jolly Jumpers* 22b⟩≡

```
package com.rvprg.pc;

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 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.

23 ⟨*Hartals* 23⟩≡

```
package com.rvprg.pc;

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));
        }
    }

}
```

2.3 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.

24 ⟨*Crypt Kicker* 24⟩≡

```
package com.rvprg.pc;

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

⟨2.4 Imports 25a⟩

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

    ⟨2.4 Variables 25f⟩

    ⟨2.4 Constructor 25g⟩

    ⟨2.4 Methods 25b⟩

    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.

25a ⟨*2.4 Imports* 25a⟩≡

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

25b ⟨*2.4 Methods* 25b⟩≡

```
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.)

25c ⟨*2.4 Imports* 25a⟩+≡

```
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;
```

25d ⟨*2.4 Methods* 25b⟩+≡

```
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.

25e ⟨*2.4 Imports* 25a⟩+≡

```
import static java.util.stream.Collectors.groupingBy;
import java.util.ArrayDeque;
```

25f ⟨*2.4 Variables* 25f⟩≡

```
private final Map<Integer, List<String>> dictionary;
private final Map<String, List<Integer>> patterns;
```

25g ⟨*2.4 Constructor* 25g⟩≡

```
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:

26a ⟨*2.4 Imports* 25a⟩+≡
```
import static java.lang.Math.abs;
```

26b ⟨*2.4 Methods* 25b⟩+≡
```
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.

26c ⟨*2.4 Variables* 25f⟩+≡
```
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.

26d ⟨*2.4 Imports* 25a⟩+≡
```
import java.util.HashSet;
import java.util.Set;
```

26e ⟨*2.4 Variables* 25f⟩+≡
```
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.

26f ⟨*2.4 Variables* 25f⟩+≡
```
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.

27a ⟨*2.4 Methods* 25b⟩+≡

```
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.

27b ⟨*2.4 Imports* 25a⟩+≡

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

28a ⟨*2.4 Methods* 25b⟩+≡

```
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.

28b ⟨*2.4 Methods* 25b⟩+≡

```
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:

28c ⟨*2.4 Imports* 25a⟩+≡

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

29a ⟨*2.4 Methods* 25b⟩+≡

```
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:

29b ⟨*2.4 Imports* 25a⟩+≡

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

29c ⟨*2.4 Methods* 25b⟩+≡

```
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();
}
```

2.4 Stack 'em Up

Easy.

30 ⟨*Stack em Up* 30⟩≡

```
package com.rvprg.pc;

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();
            }
        }

    }

}
```

2.5 Erdös Numbers

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

32 ⟨*Erdos Numbers* 32⟩≡

```
package com.rvprg.pc;

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"));
            }
        }
    }
}
```

2.6 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.

34 ⟨*Contest Scoreboard* 34⟩≡

```
package com.rvprg.pc;

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();
            }
        }
    }
```

```
}
```

2.7 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.

37 ⟨ *Yahtzee* 37⟩≡

```
package com.rvprg.pc;

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 38a⟩

    ⟨2.8 Helpers 38c⟩

    ⟨2.8 Constructor 42a⟩

    ⟨2.8 Methods 42b⟩

    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:

38a ⟨*2.8 Constants* 38a⟩≡

```
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:

38b ⟨*2.8 Constants* 38a⟩+≡

```
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.

38c ⟨*2.8 Helpers* 38c⟩≡

```
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 39a⟩

    ⟨2.8 Round Methods 39b⟩
}
```

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

39a ⟨*2.8 Round Constructor* 39a⟩≡
```
public Round(List<Integer> dice) {
    this.allDiceSum = dice.stream().reduce(0, Integer::sum);
    this.dice = dice;
    this.category = new HashSet<>();

    ⟨2.8 Categorize 41a⟩
}
```

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:

39b ⟨*2.8 Round Methods* 39b⟩≡
```
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:

39c ⟨*2.8 Round Methods* 39b⟩+≡
```
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:

40a ⟨*2.8 Round Methods* 39b⟩+≡

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

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

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

40b ⟨*2.8 Round Methods* 39b⟩+≡

```
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:

41a ⟨*2.8 Categorize* 41a⟩≡

```
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:

41b ⟨*2.8 Categorize* 41a⟩+≡

```
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.

41c ⟨*2.8 Categorize* 41a⟩+≡

```
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:

41d ⟨*2.8 Round Methods* 39b⟩+≡

```
@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:

42a ⟨*2.8 Constructor* 42a⟩≡
```
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:

42b ⟨*2.8 Methods* 42b⟩≡
```
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:

42c ⟨*2.8 Methods* 42b⟩+≡
```
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):

43 ⟨*2.8 Methods* 42b⟩+≡

```
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.

44 ⟨*2.8 Methods* 42b⟩+≡

```
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:

45 ⟨*2.8 Methods* 42b⟩+≡

```
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 Chapter 3

3.1 WERTYU

Trivial.

46 ⟨*WERTYU* 46⟩≡

```
package com.rvprg.pc;

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 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.

47 ⟨*Crypt Kicker II* 47⟩≡

```
package com.rvprg.pc;

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();
            }
        }
    }
}
```

3.3 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.

49 ⟨*File Fragmentation* 49⟩≡

```
package com.rvprg.pc;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
import java.util.ArrayList;
import java.util.List;
⟨3.6 Imports 50a⟩

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

    ⟨3.6 Helpers 50e⟩

    private static String restore(List<String> fragments) {
        ⟨3.6 Implementation 50b⟩
    }

    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:

50a ⟨*3.6 Imports* 50a⟩≡
```
import static java.util.Comparator.comparing;
```

50b ⟨*3.6 Implementation* 50b⟩≡
```
fragments.sort(comparing(String::length));
```

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

50c ⟨*3.6 Imports* 50a⟩+≡
```
import static java.util.stream.Collectors.toList;
```

50d ⟨*3.6 Implementation* 50b⟩+≡
```
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.

50e ⟨*3.6 Helpers* 50e⟩≡
```
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.

50f ⟨*3.6 Implementation* 50b⟩+≡
```
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.

4 Chapter 4

4.1 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.

51 ⟨*Bridge* 51⟩≡

```
package com.rvprg.pc;

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 52a⟩
    }

    private static List<List<Integer>> getStrategy(List<Integer> input) {
        ⟨4.3 Get Strategy 53a⟩
    }

    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.

52a ⟨*4.3 Print Result* 52a⟩≡

```
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).

52b ⟨*4.3 Print Result* 52a⟩+≡

```
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.

52c ⟨*4.3 Print Result* 52a⟩+≡

```
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:

53a ⟨*4.3 Get Strategy* 53a⟩≡
```
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:

53b ⟨*4.3 Get Strategy* 53a⟩+≡
```
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:

53c ⟨*4.3 Get Strategy* 53a⟩+≡
```
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.

54 ⟨*4.3 Get Strategy* 53a⟩+≡

```
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.

4.2 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:

55 ⟨*ShellSort* 55⟩≡

```
package com.rvprg.pc;

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 56⟩
    }

    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.

56 ⟨*4.7 Implementation* 56⟩≡

```
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;
```

5 Chapter 5

5.1 Primary Arithmetic

This task is trivial.

57 ⟨*Primary Arithmetic* 57⟩≡

```
package com.rvprg.pc;

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.

59 ⟨*Reverse And Add* 59⟩≡

```
package com.rvprg.pc;

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.

60 ⟨*The Archeologists Dilemma* 60⟩≡

```
package com.rvprg.pc;

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:

62a ⟨*Ones* 62a⟩≡

```
package com.rvprg.pc;

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 62b⟩
    }

    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:

62b ⟨*5.4 Calculation* 62b⟩≡

```
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:

63 ⟨*5.4 Calculation* 62b⟩+≡

```
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:

64 ⟨*A Multiplication Game* 64⟩≡

```
package com.rvprg.pc;

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.

66 ⟨*Polynomial Coefficients* 66⟩≡

```
package com.rvprg.pc;

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.

68 ⟨*The Stern-Brocot Number System* 68⟩≡

```
package com.rvprg.pc;

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.

Now let's have a look at what is the range for our initial value x.

TBD.

6 Chapter 7

6.1 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.

70 ⟨*Light, More Light* 70⟩≡

```
package com.rvprg.pc;

import java.io.BufferedReader;
import java.io.IOException;
import java.io.InputStreamReader;
⟨7.1 Imports 71a⟩

class LightMoreLight {
    private static final BufferedReader reader = new BufferedReader(
            new InputStreamReader(System.in));
    ⟨7.1 Variables 71b⟩
```

```
    LightMoreLight() {
        ⟨7.1 Constructor 72a⟩
    }

    public long calculate(long value) {
        ⟨7.1 Implementation 72b⟩
    }

    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.

71a ⟨*7.1 Imports* 71a⟩≡

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

71b ⟨*7.1 Variables* 71b⟩≡

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

72a ⟨*7.1 Constructor* 72a⟩≡

```
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 $v = p_1^n p_2^m$, its divisors are:

1	p_1	p_1^2	...	p_1^n
p_2	p_1p_2	$p_1^2p_2$	...	$p_1^np_2$
p_2^2	$p_1p_2^2$	$p_1^2p_2^2$	...	$p_1^np_2^2$
...	...	...	...	...
p_2^m	$p_1p_2^m$	...	...	$p_1^np_2^m$

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

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

72b ⟨*7.1 Implementation* 72b⟩≡

```
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 License

References

[1] Donald E. Knuth, Literate Programming, The Computer Journal, 1984, pp 97–111

[2] Skiena, Steven S, Revilla, Miguel A., Programming Challenges, 2003

Definitions

⟨*1.3 Calculation* 6c⟩ 6a, 6c
⟨*5.4 Calculation* 62b⟩ 62a, 62b, 63
⟨*2.8 Categorize* 41a⟩ 39a, 41a, 41b, 41c
⟨*1.2 Constants* 4c⟩ 4a, 4c
⟨*1.4 Constants* 8b⟩ 7d, 8b
⟨*2.8 Constants* 38a⟩ 37, 38a, 38b
⟨*1.1 Constructor* 3a⟩ 1a, 3a
⟨*2.4 Constructor* 25g⟩ 24, 25g
⟨*2.8 Constructor* 42a⟩ 37, 42a
⟨*7.1 Constructor* 72a⟩ 70, 72a
⟨*1.4 Convertion* 9a⟩ 7d, 9a
⟨*1.8 Election loop* 21f⟩ 21d, 21f, 21g, 21h, 21i, 22a
⟨*1.3 Finding the minimum* 6e⟩ 6c, 6e, 7a
⟨*4.3 Get Strategy* 53a⟩ 51, 53a, 53b, 53c, 54
⟨*1.1 Helpers* 2b⟩ 1a, 2b
⟨*1.4 Helpers* 9c⟩ 9a, 9c, 9d, 9f, 10b
⟨*2.8 Helpers* 38c⟩ 37, 38c
⟨*3.6 Helpers* 50e⟩ 49, 50e
⟨*1.8 Implementation* 21a⟩ 20, 21a, 21b, 21c, 21d
⟨*3.6 Implementation* 50b⟩ 49, 50b, 50d, 50f
⟨*4.7 Implementation* 56⟩ 55, 56
⟨*7.1 Implementation* 72b⟩ 70, 72b
⟨*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, 9b, 9e, 10a, 10d, 10f
⟨*1.8 Imports* 21e⟩ 20, 21e
⟨*2.4 Imports* 25a⟩ 24, 25a, 25c, 25e, 26a, 26d, 27b, 28c, 29b
⟨*3.6 Imports* 50a⟩ 49, 50a, 50c
⟨*7.1 Imports* 71a⟩ 70, 71a
⟨*1.1 Input/Output* 3c⟩ 1a, 3c
⟨*1.3 Input/Output* 7c⟩ 6a, 7c
⟨*1.4 Input/Output* 8c⟩ 7d, 8c
⟨*1.2 Main* 5⟩ 4a, 5
⟨*2.4 Methods* 25b⟩ 24, 25b, 25d, 26b, 27a, 28a, 28b, 29a, 29c
⟨*2.8 Methods* 42b⟩ 37, 42b, 42c, 43, 44, 45
⟨*1.4 Middle Column Construction* 10c⟩ 10b, 10c
⟨*4.3 Print Result* 52a⟩ 51, 52a, 52b, 52c
⟨*1.4 Process* 10e⟩ 9a, 10e
⟨*1.4 Return* 10g⟩ 9a, 10g

⟨*2.8 Round Constructor* 39a⟩ 38c, 39a
⟨*2.8 Round Methods* 39b⟩ 38c, 39b, 39c, 40a, 40b, 41d
⟨*2.4 Variables* 25f⟩ 24, 25f, 26c, 26e, 26f
⟨*7.1 Variables* 71b⟩ 70, 71b
⟨*A Multiplication Game* 64⟩ 64
⟨*Australian Voting* 20⟩ 20
⟨*Bridge* 51⟩ 51
⟨*Check The Check* 16⟩ 16
⟨*Contest Scoreboard* 34⟩ 34
⟨*Crypt Kicker* 24⟩ 24
⟨*Crypt Kicker II* 47⟩ 47
⟨*Erdos Numbers* 32⟩ 32
⟨*File Fragmentation* 49⟩ 49
⟨*Graphical Editor* 11⟩ 11
⟨*Hartals* 23⟩ 23
⟨*Interpreter* 14⟩ 14
⟨*Jolly Jumpers* 22b⟩ 22b
⟨*LC Display* 7d⟩ 7d
⟨*Light, More Light* 70⟩ 70
⟨*Minesweeper* 4a⟩ 4a
⟨*Ones* 62a⟩ 62a
⟨*Polynomial Coefficients* 66⟩ 66
⟨*Primary Arithmetic* 57⟩ 57
⟨*Reverse And Add* 59⟩ 59
⟨*ShellSort* 55⟩ 55
⟨*Stack em Up* 30⟩ 30
⟨*The Archeologists Dilemma* 60⟩ 60
⟨*The Stern-Brocot Number System* 68⟩ 68
⟨*The Trip* 6a⟩ 6a
⟨*WERTYU* 46⟩ 46
⟨*Yahtzee* 37⟩ 37
⟨*3n+1* 1a⟩ 1a

Index