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 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.)

[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)

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;
```

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

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

1.2 Minesweeper

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

Our program structure is simple as usual:

4a ⟨*Minesweeper* 4a⟩≡
```
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;
```

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 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.

11 ⟨*Australian Voting* 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.ArrayList;
import java.util.Deque;
import java.util.List;
⟨1.8 Imports 12e⟩

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 12a⟩
    }

    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.

12a ⟨*1.8 Implementation* 12a⟩≡
```
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.

12b ⟨*1.8 Implementation* 12a⟩+≡
```
final int[] counter = new int[candidates.size()];
```

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

12c ⟨*1.8 Implementation* 12a⟩+≡
```
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.

12d ⟨*1.8 Implementation* 12a⟩+≡
```
while (true) {
    ⟨1.8 Election loop 12f⟩
}
```

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

12e ⟨*1.8 Imports* 12e⟩≡
```
import static java.util.stream.Collectors.groupingBy;
import static java.util.stream.IntStream.range;
import java.util.Map;
```

12f ⟨*1.8 Election loop* 12f⟩≡
```
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:

12g ⟨*1.8 Election loop* 12f⟩+≡
```
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:

12h ⟨*1.8 Election loop* 12f⟩+≡
```
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.

13a ⟨*1.8 Election loop* 12f⟩+≡

```
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:

13b ⟨*1.8 Election loop* 12f⟩+≡

```
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.

14 ⟨*Jolly Jumpers* 14⟩≡

```
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 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 occurance of the character, or current index if it's the first occurance. 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 initalized 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.

15 ⟨*Crypt Kicker* 15⟩≡

```
package com.rvprg.pc;

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

⟨2.3 Imports 16a⟩

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

    ⟨2.3 Variables 16f⟩

    ⟨2.3 Constructor 16g⟩

    ⟨2.3 Methods 16b⟩

    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.

16a ⟨*2.3 Imports* 16a⟩≡

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

16b ⟨*2.3 Methods* 16b⟩≡

```
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.)

16c ⟨*2.3 Imports* 16a⟩+≡

```
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;
```

16d ⟨*2.3 Methods* 16b⟩+≡

```
private static Map<String, List<Integer>> getPatterns(Deque<String> words) {
    return words.stream().distinct().collect(
        toMap(identity(), CryptKicker::getPattern));
}
```

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

16e ⟨*2.3 Imports* 16a⟩+≡

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

16f ⟨*2.3 Variables* 16f⟩≡

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

16g ⟨*2.3 Constructor* 16g⟩≡

```
public CryptKicker(List<String> inputDictionary) {
    dictionary = inputDictionary.stream()
        .collect(groupingBy(String::length));
    patterns = getPatterns(new ArrayDeque<>(inputDictionary));
}
```

3 Chapter 3

3.1 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.

17 ⟨*File Fragmentation* 17⟩≡

```
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 18a⟩

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

    ⟨3.6 Helpers 18e⟩

    private static String restore(List<String> fragments) {
        ⟨3.6 Implementation 18b⟩
    }

    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:

18a ⟨*3.6 Imports* 18a⟩≡
```
import static java.util.Comparator.comparing;
```

18b ⟨*3.6 Implementation* 18b⟩≡
```
fragments.sort(comparing(String::length));
```

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

18c ⟨*3.6 Imports* 18a⟩+≡
```
import static java.util.stream.Collectors.toList;
```

18d ⟨*3.6 Implementation* 18b⟩+≡
```
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.

18e ⟨*3.6 Helpers* 18e⟩≡
```
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.

18f ⟨*3.6 Implementation* 18b⟩+≡
```
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 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:

19 ⟨*ShellSort* 19⟩≡

```
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 20⟩
    }

    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.

20 ⟨*4.7 Implementation* 20⟩≡

```
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 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:

21a ⟨*Ones* 21a⟩≡

```
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 21b⟩
    }

    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:

21b ⟨*5.4 Calculation* 21b⟩≡

```
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:

22 ⟨*5.4 Calculation* 21b⟩+≡

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

Brilliant.

6 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* 21b⟩ 21a, 21b, 22
⟨*1.2 Constants* 4c⟩ 4a, 4c
⟨*1.4 Constants* 8b⟩ 7d, 8b
⟨*1.1 Constructor* 3a⟩ 1a, 3a
⟨*2.3 Constructor* 16g⟩ 15, 16g
⟨*1.4 Convertion* 9a⟩ 7d, 9a
⟨*1.8 Election loop* 12f⟩ 12d, 12f, 12g, 12h, 13a, 13b
⟨*1.3 Finding the minimum* 6e⟩ 6c, 6e, 7a
⟨*1.1 Helpers* 2b⟩ 1a, 2b
⟨*1.4 Helpers* 9c⟩ 9a, 9c, 9d, 9f, 10b
⟨*3.6 Helpers* 18e⟩ 17, 18e
⟨*1.8 Implementation* 12a⟩ 11, 12a, 12b, 12c, 12d
⟨*3.6 Implementation* 18b⟩ 17, 18b, 18d, 18f
⟨*4.7 Implementation* 20⟩ 19, 20
⟨*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* 12e⟩ 11, 12e
⟨*2.3 Imports* 16a⟩ 15, 16a, 16c, 16e
⟨*3.6 Imports* 18a⟩ 17, 18a, 18c
⟨*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
⟨*2.3 Methods* 16b⟩ 15, 16b, 16d
⟨*1.4 Middle Column Construction* 10c⟩ 10b, 10c
⟨*1.4 Process* 10e⟩ 9a, 10e
⟨*1.4 Return* 10g⟩ 9a, 10g
⟨*2.3 Variables* 16f⟩ 15, 16f
⟨*Australian Voting* 11⟩ 11
⟨*Crypt Kicker* 15⟩ 15
⟨*File Fragmentation* 17⟩ 17
⟨*Jolly Jumpers* 14⟩ 14
⟨*LC Display* 7d⟩ 7d
⟨*Minesweeper* 4a⟩ 4a
⟨*Ones* 21a⟩ 21a
⟨*ShellSort* 19⟩ 19
⟨*The Trip* 6a⟩ 6a

⟨*3n+1* 1a⟩ 1a

Index