

Programming Challenges

Roman Valiusenko
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.

Contents

1	Chapter 1	1
1.1	The $3n + 1$ Problem	1
1.2	Minesweeper	3
1.3	The Trip	5
1.4	LC Display	6
1.5	Australian Voting	10
2	Chapter 3	13
2.1	File Fragmentation	13
3	Chapter 5	15
3.1	Ones	15
4	License	16

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.

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 2a>
        <1.1 Constructor 2c>
        <1.1 Input/Output 3a>
    }

```

We need 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 a `surplus` hash map and depending on the index value either use the array or the hash map to set or get a value.

```
1c  <1.1 Imports 1b>+≡
    import java.util.HashMap;

2a  <1.1 Helpers 2a>≡
    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 calculated 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 ns. We use this knowledge to update elements in the `lengths`:

```
2b  <1.1 Imports 1b>+≡
    import java.util.ArrayDeque;
    import java.util.Deque;
```

```

2c  <1.1 Constructor 2c>≡
    Collatz() {
        final HashMap<Long, Integer> surplus = new HashMap<Long, Integer>();
        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:

```

2d  <1.1 Imports 1b>+≡
    import java.util.stream.IntStream;

3a  <1.1 Input/Output 3a>≡
    public static void main(String[] args) {
        Collatz s = new Collatz();
        reader.lines()
            .map(x -> x.split(" "))
            .map(x -> new int[] { Integer.parseInt(x[0]), Integer.parseInt(x[1]) })
            .map(x -> new int[] { x[0], x[1], Math.min(x[0], x[1]), Math.max(x[0], x[1]) })
            .forEach(x -> {
                System.out.println(x[0] + " " + x[1] + " " +
                    IntStream.range(x[2], x[3]).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 eight 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:

```
3b  <Minesweeper 3b>≡
    package com.rvprg.pc;

    <1.2 Imports 3c>

    class Minesweeper {
        <1.2 Constants 3d>
        <1.2 Main 4b>
    }
```

Of course, we need a reader, so we define it next. Then we need to define the constants. We know that the input end is two zeroes, so we define such a constant. We are going to split the lines by spaces, so let's have that too. Finally, we define an array of the offsets **p** to determine the cells around a given cell.

```
3c  <1.2 Imports 3c>≡
    import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;

3d  <1.2 Constants 3d>≡
    private static final BufferedReader reader =
        new BufferedReader(new InputStreamReader(System.in));
    private static final String INPUT_END = "0 0";
    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 deliberately use one-dimensional 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.

```
4a  <1.2 Imports 3c>+≡
    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;
```

```

4b  <1.2 Main 4b>≡
    public static void main(String[] args) throws IOException {
        int lineNum = 0;
        String currentLine = INPUT_END;
        while (!(currentLine = reader.readLine().trim()).equalsIgnoreCase(INPUT_END)) {
            List<Integer> nm = stream(currentLine.split(SPACE)).map(Integer::parseInt).collect(toList());
            int n = nm.get(0);
            int m = nm.get(1);

            final int[] field = reader.lines().limit(m)
                .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();

            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 than the previous two. The first thing that we should note for ourselves is that we are not going to use the floating point types to do the calculations.

```

5a  <The Trip 5a>≡
    package com.rvprg.pc;

    <1.3 Imports 5b>

    class TheTrip {
        <1.3 Calculation 5c>
        <1.3 Input/Output 6c>
    }

```

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:

```

5b  <1.3 Imports 5b>≡
    import static java.util.Arrays.stream;

```

```

5c  <1.3 Calculation 5c>≡
    static long calculate(long[] values) {
        if (values.length == 1)
            return 0;
        long total = stream(values).sum();
        <1.3 Finding the minimum 5e>
    }

```

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 remainder. So we divide the **total** by the number of students to get the quotient and the remainder, and we partition only using the quotient; that is group 1 will contain spends x such that $x - \text{quotient} \leq 0$, and group 2 will have the others.

```

5d  <1.3 Imports 5b>+≡
    import static java.lang.Math.abs;
    import static java.util.stream.Collectors.partitioningBy;
    import java.util.List;
    import java.util.Map;

5e  <1.3 Finding the minimum 5e>≡
    long quotient = total / values.length;
    long remainder = 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 **remainder**? These are those cents that we need to finally re-distribute among the members of the two groups. Note that the **remainder** 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:

```

6a  <1.3 Finding the minimum 5e>+≡
    long sum = abs(diff.get(false).stream().reduce(Long::sum).get());
    long len = diff.get(true).size();
    remainder = len <= remainder ? remainder - len : 0;
    return sum + remainder;

```

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

```

6b  <1.3 Imports 5b>+≡
    import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;
    import java.math.BigDecimal;

```

```

6c  <1.3 Input/Output 6c>≡
    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.

```

6d  <LC Display 6d>≡
    package com.rvprg.pc;

    <1.4 Imports 7a>

    class LCDisplay {
        <1.4 Constants 7b>
        <1.4 Conversion 8a>
        <1.4 Input/Output 7c>
    }

```

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.

```

7a  <1.4 Imports 7a>≡
    import java.io.BufferedReader;
    import java.io.IOException;
    import java.io.InputStreamReader;
    import java.util.stream.Stream;

```

7b `<1.4 Constants 7b>≡`

```
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.

7c `<1.4 Input/Output 7c>≡`

```
public static void main(String[] args) throws IOException {
    String currentLine = INPUT_END;
    while (!(currentLine = reader.readLine().trim()).equalsIgnoreCase(INPUT_END)) {
        String[] input = currentLine.split(SPACE);
        segments(input[1], Integer.valueOf(input[0])).stream().forEach(System.out::println);
        System.out.println();
    }
}
```

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

8a `<1.4 Conversion 8a>≡`

```
private static List<String> segments(final String digits, final int scale) {
    <1.4 Helpers 8c>
    <1.4 Process 9e>
    <1.4 Return 9g>
}
```


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:

```
8b  <1.4 Imports 7a>+≡
    import static java.lang.String.join;
    import static java.util.Collections.nCopies;
    import java.util.function.Function;

8c  <1.4 Helpers 8c>≡
    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:

```
8d  <1.4 Helpers 8c>+≡
    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`:

```
8e  <1.4 Imports 7a>+≡
    import static java.util.stream.Collectors.joining;
    import java.util.function.BiFunction;

8f  <1.4 Helpers 8c>+≡
    BiFunction<Stream<Integer>, Byte, String> k =
        (m, d) -> SPACE + h.apply(m, d).map(x -> g.apply(x)).collect(joining(SPACE)) + SPACE;
```

Note `SPACES` 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).

```
9a  <1.4 Imports 7a>+≡
    import static java.util.stream.Collectors.toList;
    import static java.util.stream.Stream.of;
    import java.util.Arrays;
```

```

9b  <1.4 Helpers 8c>+≡
    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 9c>,
        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 construct 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 occurrences 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:

```

9c  <1.4 Middle Column Construction 9c>≡
    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:

```

9d  <1.4 Imports 7a>+≡
    import java.util.List;

9e  <1.4 Process 9e>≡
    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:

```

9f  <1.4 Imports 7a>+≡
    import static java.util.stream.IntStream.range;
    import static java.util.stream.IntStream.rangeClosed;

9g  <1.4 Return 9g>≡
    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 dequeues, 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.

```
10 <Australian Voting 10>≡
    package com.rvpgrg.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 11e>

    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 11a>
        }

        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()).equalsIgnoreCase(EMPTY) && currentLine != null) {
                    ballots.add(new ArrayDeque<Integer>(stream(currentLine.trim().split(" "))
                        .map(Integer::parseInt).map(x -> x - 1).collect(toList())));
                }
                elect(candidates, ballots).forEach(System.out::println);
                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.

```
11a <1.8 Implementation 11a>≡
    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 `ints`, which will hold the number of votes.

```
11b <1.8 Implementation 11a>+≡
    final int[] counter = new int[candidates.size()];
```

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

```
11c <1.8 Implementation 11a>+≡
    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.

```
11d <1.8 Implementation 11a>+≡
    while (true) {
        <1.8 Election loop 11f>
    }
```

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

```
11e <1.8 Imports 11e>≡
    import static java.util.stream.Collectors.groupingBy;
    import static java.util.stream.IntStream.range;
    import java.util.Map;
```

```
11f <1.8 Election loop 11f>≡
    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:

```
11g <1.8 Election loop 11f>+≡
    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 $max \leq majority$ or $max = min$, then we know the winner:

```
11h <1.8 Election loop 11f>+≡
    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.

```
11i <1.8 Election loop 11f>+≡
    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 elimination process the first in the rank has changed, we need to take that into account. This is captured in the following chunk:

```
12  <1.8 Election loop 11f>+≡  
    ballots.forEach(b -> {  
        int first = b.peek();  
        eliminated.forEach(x -> b.remove(x));  
        counter[b.peek()] += (first != b.peek()) ? 1 : 0;  
    });
```

2 Chapter 3

2.1 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.

```

13  <File Fragmentation 13>≡
    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 14a>

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

        <3.6 Helpers 14e>

        private static String restore(List<String> fragments) {
            <3.6 Implementation 14b>
        }

        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));
                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:

```
14a  <3.6 Imports 14a>≡
      import static java.util.Comparator.comparing;
```

```
14b  <3.6 Implementation 14b>≡
      fragments.sort(comparing(String::length));
```

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

```
14c  <3.6 Imports 14a>+≡
      import static java.util.stream.Collectors.toList;
```

```
14d  <3.6 Implementation 14b>+≡
      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.

```
14e  <3.6 Helpers 14e>≡
      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`.

```
14f  <3.6 Implementation 14b>+≡
      for (String small : smallest) {
          if (fit(fragments, large + small)) {
              return large + small;
          } else if (fit(fragments, small + large)) {
              return small + large;
          }
      }
      return "Impossible";
```

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

3 Chapter 5

3.1 Ones

This is a little cute 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 to come up with some clever "divisibility rules" to see if a given n divides a number that has only 1s in it. But this is 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 remainder until it doesn't divide without a remainder.

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

```
15a  <Ones 15a>≡
      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 15b>
          }

          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:

```
15b  <5.4 Calculation 15b>≡
      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 $r = 0$.

```
16  <5.4 Calculation 15b>+≡
    int l = 2;
    int s = 11;
    int r = 11;

    if (n != s) {
        do {
            s = r * 10 + 1;
            int d = s / n;
            r = s - (n * d);
            l++;
        } while (r > 0);
    }

    return l;

    Brilliant.
```

4 License

Copyright©2017 Roman Valiushenko

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

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 5c> 5a, [5c](#)
 <5.4 Calculation 15b> 15a, [15b](#), [16](#)
 <1.2 Constants 3d> 3b, [3d](#)
 <1.4 Constants 7b> 6d, [7b](#)
 <1.1 Constructor 2c> 1a, [2c](#)
 <1.4 Conversion 8a> 6d, [8a](#)
 <1.8 Election loop 11f> 11d, [11f](#), [11g](#), [11h](#), [11i](#), [12](#)
 <1.3 Finding the minimum 5e> 5c, [5e](#), [6a](#)
 <1.1 Helpers 2a> 1a, [2a](#)
 <1.4 Helpers 8c> 8a, [8c](#), [8d](#), [8f](#), [9b](#)
 <3.6 Helpers 14e> 13, [14e](#)
 <1.8 Implementation 11a> 10, [11a](#), [11b](#), [11c](#), [11d](#)
 <3.6 Implementation 14b> 13, [14b](#), [14d](#), [14f](#)
 <1.1 Imports 1b> 1a, [1b](#), [1c](#), [2b](#), [2d](#)
 <1.2 Imports 3c> 3b, [3c](#), [4a](#)
 <1.3 Imports 5b> 5a, [5b](#), [5d](#), [6b](#)
 <1.4 Imports 7a> 6d, [7a](#), [8b](#), [8e](#), [9a](#), [9d](#), [9f](#)
 <1.8 Imports 11e> 10, [11e](#)
 <3.6 Imports 14a> 13, [14a](#), [14c](#)
 <1.1 Input/Output 3a> 1a, [3a](#)
 <1.3 Input/Output 6c> 5a, [6c](#)
 <1.4 Input/Output 7c> 6d, [7c](#)
 <1.2 Main 4b> 3b, [4b](#)
 <1.4 Middle Column Construction 9c> 9b, [9c](#)
 <1.4 Process 9e> 8a, [9e](#)
 <1.4 Return 9g> 8a, [9g](#)
 <Australian Voting 10> [10](#)
 <File Fragmentation 13> [13](#)
 <LC Display 6d> [6d](#)
 <Minesweeper 3b> [3b](#)
 <Ones 15a> [15a](#)
 <The Trip 5a> [5a](#)
 <3n+1 1a> [1a](#)

Index