Lucky Skyscraper

Emanuel Johnson Godin

October 24, 2022

1 Introduction

Let's say you own a skyscraper that gets visitors from all over the world. One day, you decide that you want to make it as accessible as possible to all cultures. Some cultures have numbers that are considered "unlucky" - notable examples are 4 (Asia) and 13 (the West). To make your visitors feel more at ease, you don't want to have unlucky floors - *a floor number that ends on an unlucky number* - showing on signs, in elevators, and such. So if you account for unlucky numbers 4 and 13 and you have 20 floors (starting at 0), your new "lucky" skyscraper would contain floors 0, 1, 2, 3, **5**, 6, 7, 8, 9, 10, 11, 12, **15**, 16, 17, 18, 19, 20, 21, 22 and 22. Now - how do you get the new floor number from an old one, and conversely, how do you get the old floor number from a new one? This is known as the *Lucky Skyscraper* problem.

This document aims to outline different ways of both defining and solving the problem in ascending complexity. Implementations are written in the programming language Java.

Glossary

- an *unlucky* floor: a floor whose number ends with an unlucky number. If the unlucky numbers are for example 4 and 13, 1234 and 2013 are considered unlucky floors.
- a *lucky* floor: a floor that is not *unlucky*.
- a *real* floor: an *unlucky* OR a *lucky* floor.
- a *fake* floor: a floor that can ONLY be *lucky*.

2 Iterative solution

For now, let's say our only unlucky numbers are 4 and 13. Let's also simplify things by starting off with arguably the most intuitive solution: an interative one. To fully solve Lucky Skyscraper, we will need two functions. One to map each real floor to its fake counterpart, one to do the inverse.

2.1 Real to fake

```
public static int realToFake(int real) {
    int fake = -1;
    for (int i = 0; i <= real; i++){
         fake++;
         while (fake % 10 == 4 || fake % 100 == 13) {
             fake++;
         }
    }
    return fake;
}
```

The fake number is iteratively incremented. In each iteration, it is incremented by 1, but if the current fake floor is unlucky, it's moreover incremented by 1 until it isn't.

2.2 Fake to real

```
public static int fakeToReal(int fake) {
    int fakeCounter = 0;
    for (int real = 0; real <= fake; real++) {
        while (fakeCounter % 10 == 4 || fakeCounter % 100 == 13) {
            fakeCounter++;
        }
        if (fakeCounter == fake) {
            return real;
        }
        fakeCounter++;
    }
    return -1; // to make the java compiler happy; should never occur
}
```

This solution is interesting in that it in many ways looks the same as the previous solution, despite being the complete inverse of it - a pattern that will actually re-emerge once we start looking into the arithmetic solution.

Here, the strategy is to try and iteratively "backtrack" to find out what real floor a fake one originates from. In other words: *the amount of iterations needed in order for the fake floor counter to reach the actual fake floor given is equal to the real floor*.

3 Arithmetic solution

The arithmetic solution for going from a fake to a real floor is actually *much easier* to understand than the opposite, so let's start with that instead.

Note: When creating an arithmetic solution, a problem emerges when we define two or more unlucky numbers which both makes the same subset of floors unlucky, for example 4 and 14. All of the coming functions rely on each defined unlucky number actually doing something, and as such, there can only ever be a maximum of 10 different lucky numbers, each ending on their own respective digit (0-9).

3.1 Fake to real

Let's say you are given the fake floor and you need to convert it to a real one - without iteration. How do you begin to approach this problem? First off, we know that any given fake floor must be equal to or bigger than its real counterpart. Knowing this, what does the expression *fake* − *real* signify? *The number of skips the fake floor has done*. Consider the following table, with unlucky numbers 4 and 13:

real:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
fake:	0	1	2	3	5	6	7	8	9	10	11	12	15	16	17	18
skips:	0	0	0	0	1	1	1	1	1	1	1	1	3	3	3	3

Now let's say we hide the entire first column. We can still infer what *real* should be given *fake* and *skips* by simply doing *fake* - *skips*. That's fine and all, but we obviously need to infer the number of skips ourselves given a fake floor. Let's start by looking at our skips in the table above. All through *real* 0-3 we don't skip anything and as such *skips* remain at 0. However, at *real* **4**, *fake* ticks over to **5** and *skips* becomes **1**. We then get that the skips of a fake floor that is skipped due to 4 is given by the function

$$skips_4(fake) = \left\lfloor \frac{10 - 4 + fake}{10} \right\rfloor.$$

10 is the frequency of the skips. More generally, one can define the frequency as the closest upper power of 10; 4 must have a frequency of 10, 13 must have a frequency of 100, and so on. In other words, the frequency of$n$-skips becomes

$$freq(n) = \begin{cases} 10, & \text{if } n \in \{0, 1\} \\ 10n, & \text{if } 10^{\lceil \log_{10} n \rceil} = n \\ 10^{\lceil \log_{10} n \rceil}, & \text{otherwise} \end{cases}$$

The first two edge cases show up as artifacts we need to handle due to the way we calculate frequency; by far the most common outcome of this function will be the 3rd case. Using this skip function, we can express the general skip function $skips_n(fake)$ for unlucky number n with the 4-skip function as a template:

$$skips_n(fake) = \left\lfloor \frac{freq(n) + fake - n}{freq(n)} \right\rfloor = \left\lfloor 1 + \frac{fake - n}{freq(n)} \right\rfloor.$$

With this, we can finally define our fake-to-real function. Let the sequence of unlucky numbers N be defined as $(n_i : n_{i-1} < n_i)$. The corresponding real floor of any fake floor can then be defined as

$$real(fake) = fake - \sum_{i=1}^{|N|} skips_{n_i}(fake)$$

In code, this becomes:

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

import static java.lang.Math.ceil;
import static java.lang.Math.floor;
import static java.lang.Math.pow;
import static java.lang.Math.log10;


public class Main {

    public static int[] unluckies = new int[]{4, 13};

    public static int skips(int n, int fake) {
        return (int) floor(1 + (fake - n) / pow(10, ceil(log10(n))));
    }

    public static int fakeToReal(int fake) {
        return fake - Arrays.stream(unluckies).map(n -> skips(n, fake)).sum();
    }

    public static void main(String[] args) {
        System.out.println(fakeToReal(17)); // 14
    }
}
```

3.2 Real to fake

The reason why converting from a real floor to a fake one is so difficult is because you need to *predict ahead of time how many skips will occur*. This non-trivial task requires non-trivial math; each skip function for each unlucky number won't exist in a vacuum but instead references other skip functions "above" or "below" itself. The general form of a skip function, however, will be reminiscent of what we went through in the previous section. It can help to think of all of the steps here as ways of the skips to "catch up" with the fake number, and to "catch up" properly, each skip function needs knowledge "outside itself".

First, we need to remodel our skip frequency expression. Now, The real-to-fake frequency of 4-skips becomes 9, instead of 10 like we're used to. In fact, *all* of our frequencies will now become (at least!) 1 lower than before, in order to *catch up with the skip we end up doing*. This becomes even more clear when we consider the real-to-fake frequency of 13-skips: 89. All the while we are looking for 13-skips, 10 4-skips will have occured. The 13-skip frequency needs to be aware of this, otherwise the fake floor will "race away" and the 13-skip won't be able to keep up and will trigger way too slow. The "$10-1$" represents the same principle, applied "inward".

Let the sequence of unlucky numbers N be defined as $(n_i : n_i > n_{i-1})$, and the base frequence $Bfreq$ of an unlucky number be defined as our previous fake-to-real $freq$. The frequency of a n-skip with the N-position i then becomes

$$freq(n_i) = Bfreq(n_i) - \sum_{j=1}^{i} \frac{Bfreq(n_i)}{Bfreq(n_j)}.$$

With this, the n-skip with the N-position i then becomes

$$skips_{n_i}(real) = \left\lfloor 1 + \frac{real - fake_{N'=(n_j : n_j,\, j<i)}(real) + \sum_{j=i+1}^{|N|} skips_{n_j}(real)}{freq(n_i)} \right\rfloor.$$

While *technically* only using "old" puzzle pieces, this function definitely requires an explanation in order to parse correctly.

First off, $fake_{N'=(n_j : n_j,\, j<i)}(real)$ uses our old *fake* function. Why? Because our new skip function needs to catch up with the skips already done by smaller unlucky numbers. Imagine we have $N = (4, 13)$ and want to add the unlucky number 101. When we go from 100 to 101, 11 skips have already been done. We somehow need to incorporate this into the 101-skip so that it doesn't trigger 11 floors late. The question then becomes how to get that number. Simple - yield it from $n - fake(real)$! Of course, we can't include the new unlucky number itself when doing this calculation, which is why we use N'; a N with n_i "chopped off". As a bonus, we only need to include $-fake(real)$ in the calculation after simplifying the expression.

In a similar fashion, $\sum_{j=i+1}^{|N|} skips_{n_j}(real)$ is for smaller skips to know how many bigger skips has occured. Obviously this is a dynamic sum that of course changes as *real* changes, while the term from before only really positioned the skip at "t_0" correctly.

Summing it up, both literally and figuratively, we end up with the real-to-fake function

$$fake(real) = real + \sum_{i=1}^{|N|} skips_{n_i}(real)$$

which we can calculate using this code:

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

import static java.util.Arrays.stream;
import static java.util.Arrays.copyOfRange;

import static java.lang.Math.ceil;
import static java.lang.Math.floor;
import static java.lang.Math.floorDiv;

import static java.lang.Math.pow;
import static java.lang.Math.log10;


public class Main {
    public static int bfreq(int n) {
        if (n == 0 || n == 1) return 10;
        if (pow(10, ceil(log10(n))) == n) return n * 10;
        return (int) pow(10, ceil(log10(n)));
    }

    public static int fakeToRealSkips(int n, int fake) {
        return (int) floor(1 + (fake - n) / bfreq(n));
    }

    public static int fakeToReal(int[] ns, int fake) {
        return fake - stream(ns)
                      .map(n -> fakeToRealSkips(n, fake))
                      .sum();
    }

    public static int freq(int[] ns, int n, int i) {
        return bfreq(n) - IntStream.rangeClosed(0, i)
                          .map(j -> bfreq(ns[i]) / bfreq(ns[j]))
                          .sum();
    }

    public static int realToFakeSkips(int[] ns, int n, int i, int real) {
        return (int) floorDiv(
            freq(ns, n, i)
            + real
            - fakeToReal(copyOfRange(ns, 0, i), n)
            + IntStream.range(i + 1, ns.length)
```

```
                .map(j -> realToFakeSkips(ns, ns[j], j, real))
                .sum()
            , freq(ns, n, i));
    }

    public static int realToFake(int[] ns, int real) {
        return real + IntStream.range(0, ns.length)
                    .map(i -> realToFakeSkips(ns, ns[i], i, real))
                    .sum();
    }

    public static int realToFake(int real) {
        return realToFake(new int[]{4, 13}, real);
    }
}
```