Code Jam 2013 - Qualification Round

Analysis: Fair and Square

The first thing to do in this problem (as in many other problems) is to make sure you read it carefully. Many contestants thought that 676 should be a fair and square number - after all, it is a square **and** a palindrome. It is *not*, however, a square **of** a palindrome, and this example is actually mentioned specifically in the problem statement!

The small input

Once you realize this, you can approach the small testcase by iterating over all the numbers Little John considers, and checking each one of them. You have to check for each number **X** whether it is a palindrome, and whether it is the square of a palindrome.

To check whether **X** is a palindrome, you can simply convert it to string format (the details here depend on the programming language you are using) and compare the first character to the last, the second to the second last, and so on.

To check whether **X** is the square of a palindrome, there are multiple options. One is to calculate the square root, and if the square root is an integer, check if it is a palindrome as described above. Another is simply to iterate over all numbers up to **X**, and for each palindrome, square it and see if the square is **X**. That's a perfectly good solution for the small input, but it will be too slow for the larger ones.

The first large input

For the first large input, we need to deal with numbers up to 10^{14}, and also with 10,000 test cases. A linear search of all numbers up to 10^{14} is not going to be fast enough, so we have to be smarter - we can't afford to check each number in the interval individually.

We don't really need to go all the way up to 10^{14} though! We are interested in numbers whose *squares* are Fair and Square and between **A** and **B** - and that means we have to check up to the *square root* of **B** only. That's only 10^7 numbers to check in the worst case.

We are not done though. While 10^7 numbers can be processed within the time limit, processing 10,000 cases like this is somewhat risky. There are two tricks you can notice to make your solution faster.

One trick is that we are interested not in all the numbers up to 10^7, but only in palindromes. We can generate all the palindromes much faster. Start by taking all the numbers up to 10^4, and then taking their mirror reflections (either duplicating the last number or not) to generate all palindromes of length up to 8 (and then square each and check whether it is a Fair and Square number in the interesting interval). This will cause us to evaluate only around 10,000 numbers for each test case, which is small enough that even a slow machine can deal with all the test cases in four minutes. You would need to use a reasonably efficient language however.

An alternative is to simply generate all the fair and square numbers up to 10^{14} *before* processing the test cases. There are relatively few of them — it turns out only 39. Thus, if you find all of them (in any fashion) before downloading the input file, you can easily give the correct answers to all the input cases.

Note that if you do this, you have to include the code you used to generate Fair and Square numbers - not just the code that includes the full list!

The second large data set

Now we come to the largest data set. Even combining both the tricks above is not enough - we need to go over 10^{25} palindromes to precompute everything. This will take a very long time in any language on any computer! A good idea here is to generate the first few Fair and Square numbers (and their square roots) to try and get an idea of what they look like. There are two things you can notice:

- All the Fair and Square numbers have an odd number of digits
- All the digits are rather small. In particular, with one exception, every square root of a Fair and Square number consists only of digits 0, 1 and 2.

Let's try to understand why these things would be true.

Let's begin with the "odd number of digits". A square of a number with **N** digits will have either 2**N** - 1 or 2**N** digits, depending on whether there is a carry on the last position. Let's try to prove a carry never happens. Let **X** be Fair and Square, and let its square root be **Y**. Let the first digit of **Y** be **c** - then the first two digits of **X** are between $\mathbf{c}^2$ and $(\mathbf{c}+1)^2$. In particular:

- If the first digit of **Y** is 1, the first digit of **X** is between 1 and 4 - and thus no carry.
- If the first digit of **Y** is 2, the first digit of **X** is between 4 and 9 - and thus no carry.
- If the first digit of **Y** is 3, the first digits of **X** are between 9 and 16, so the first digit is 9 or 1. As **Y** is a palindrome, the last digit of **Y** is 3 as well, and thus the last digit of **X** is 9 - meaning the first digit of **X** is 9 as well, meaning no carry.
- If the first and last digit of **Y** is 4, the last digit of **X** is 6, while the first is either 1 or 2 - so **X** can't be Fair and Square.
- Similarly, if the first and last digit of **Y** is 5 (last digit of **X** is 5, first is 2 or 3), 6 (last digit of **X** is 6, first is 3 or 4), 7 (last digit of **X** is 9, first is 4, 5 or 6), 8 (last digit of **X** is 4, first is 6, 7 or 8) and 9 (last digit of **X** is 1, first is 8 or 9), then **X** also can't be Fair and Square.

This means there is no carry in the first digit.

Now since all the digits seem so small, maybe this means there is no carry at all? Note that if you take a palindrome and square it, and there's no carry, the result is a palindrome as well - so that would give us a nice characterization of Fair and Square numbers. Indeed, it turns out to be the case, and the proof follows.

Let **Y** have digits $(a_d)(a_{d-1})...(a_0)$. Let $b_k = a_0 * a_k + a_1 * a_{k-1} + ... + a_k * a_0$. Note that b_i is exactly the *ith* digit of $\mathbf{X} = \mathbf{Y}^2$ when performing long multiplication, before carries are performed. Since $a_j = a_{d-j}$, we also have $b_i = b_{2d-i}$.
Now suppose there's a carry in the long multiplication (meaning some b_j is greater than 9), and that we take a carry into digit i but no larger digits. We know digits i and 2d-i in **X** are equal, and are equal to b_i plus whatever we carried into digit *i*. Since we carry nothing to digit i+1, b_i is no larger than 9.
Now we will see that digit 2d-i of **X** has to equal b_{2d-i} (which is equal to b_i, and thus no larger than 9). For it to be different, we would have to carry something into digit 2d-i - but this would mean that b_j is larger than 9 for some j < 2d-i, and hence b_{2d-j} is also greater than 9 and we would have a carry after digit i.
Since **X** is a palindrome, this tells us that digit i of **X** is equal to b_i, which means that no carry entered digit i, and we have a contradiction.
We conclude that no carries were performed in the long multiplication at all.

Thus, the Fair and Square numbers are exactly the palindromes with no carries inside. In particular, the middle digit of **X** is the sum of squares of all the digits of **Y**, so this sum has to be no larger than nine. We conclude that only 0, 1, 2, and 3 can appear in **Y**.

To find all Fair and Square numbers, it therefore suffices to consider only palindromes consisting of these four digits, with the sum of squares of digits at most 9. It turns out this is a small enough set that it can be directly searched, allowing us to generate the full list of all Fair and Square numbers up to 10^{100} in a few seconds - and thus allowing us to solve the largest dataset.

Lessons learned

There are a few things we want to remind you of in the context of this problem:

- It is really important to read the problem statement carefully.
- We can sometimes have problems in which we don't have the standard combination of one small and one large input. The rules for dealing with small and large inputs are still the same (unless explicitly stated otherwise in the problem statement).
- We can also sometimes give problems that require very large integers. We did give Fair Warning about this some time ago, but it's always worth reminding.
- Finally, if you use precomputation in your solution, remember that you are required to provide us not only with the code that you actually used to solve the problem (containing the precomputed values), but also the code that you used for precomputation.