

## You have a method `rand7()` that generates a random integer from 1 to 7. Use it to write a method `rand5()` that generates a random integer from 1 to 5.

`rand7()` returns each integer with equal probability. `rand5()` must also return each integer with equal probability.

### Gotchas

Your first thought might be to simply take the result of `rand7()` and take a modulus:

```
public static int rand5() {  
    return rand7() % 5 + 1;  
}
```

Java ▼

**However, this won't give an equal probability for each possible result.** We can write out each possible result from `rand7()` (each of which is equally probable, per the problem statement) and see that some results for `rand5()` are more likely because they are caused by more results from `rand7()`:

<code>rand7()</code>	<code>rand5()</code>
1	2
2	3
3	4
4	5
5	1
6	2
7	3

So we see that there are two ways to get 2 and 3, but only one way to get 1, 4, or 5. This makes 2 and 3 twice as likely as the others.

What about calling `rand7()` five times, summing up the result, and then taking the modulus?

This is *really close* to uniform, but not quite. Since we're calling `rand7()` five times, there are  $7^5 = 16,807$  possible results. That's not divisible by five, so some outcomes must be more likely than others. (If you're curious, 1 is the result 3,357 times; 2 and 5 are the result 3,360 times each; and 3 and 4 are the result 3,365 times.)

In fact, no matter how many times we run `rand7()`, we'll never get a number of outcomes that's divisible by five.

The answer takes worst-case infinite time. However, we can get away with worst-case  $O(1)$  space. **Does your answer have a non-constant space cost?** If you're using recursion (and your language doesn't have tail-call optimization), you're potentially incurring a worst-case infinite space cost in the call stack. But replacing your recursion with a loop avoids this.

## Breakdown

`rand5()` must return each integer with equal probability, but we don't need to make any guarantees about its runtime...

In fact, the solution has a small possibility of *never* returning...

## Solution

We simply "re-roll" whenever we get a number greater than 5.

```
public static int rand5() {  
    int result = 7; // arbitrarily large  
    while (result > 5) {  
        result = rand7();  
    }  
    return result;  
}
```

Java ▼

So each integer 1,2,3,4, or 5 has a probability  $\frac{1}{7}$  of appearing at each roll.

## Complexity

Worst-case  $O(\infty)$  time (we might keep re-rolling forever) and  $O(1)$  space.

Note that if we weren't worried about the potential space cost (nor the potential stack overflow) of recursion, we could use an arguably-more-readable recursive approach with  $O(\infty)$  space cost:

```
public static int rand5() {  
    int result = rand7();  
    return (result <= 5) ? result : rand5();  
}
```

Java ▼

## Bonus

This kind of math is generally outside the scope of a coding interview, but: if you know a bit of number theory you can *prove* that there exists no solution which is guaranteed to terminate.

Hint: it follows from the fundamental theorem of arithmetic.

## What We Learned

Making sure each possible result has *the same probability* is a big part of what makes this one tricky.

If you're ever struggling with the math to figure something like that out, don't be afraid to *just count*. As in, write out all the possible results from `rand7()`, and label each one with its final outcome for `rand5()`. Then count up how many ways there are to make each final outcome. If the counts aren't the same, the method isn't uniformly random.

## Ready for more?

Check out our full course →

Want more coding interview help?

Check out **[interviewcake.com](https://www.interviewcake.com)** for more advice, guides, and practice questions.