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

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

## Gotchas

Simply running `rand5()` twice, adding the results, and taking a modulus won't give us an equal probability for each possible result.

Not convinced? Count the number of ways to get each possible result from 1..7.

Your function will have worst-case infinite runtime, because sometimes it will need to "try again."

However, at each "try" you only need to make two calls to `rand5()`. If you're making 3 calls, you can do better.

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.

## Breakdown

Because we need a random integer between 1 and 7, we need at least 7 possible outcomes of our calls to `rand5()`. One call to `rand5()` only has 5 possible outcomes. So we must call `rand5()` at least twice.

**Can we get away with calling `rand5()` exactly twice?**

Our first thought might be to simply add two calls to `rand5()`, then take a modulus to convert it to an integer in the range 1..7:

```
int rand7Mod()
{
    return (rand5() + rand5()) % 7 + 1;
}
```

C++

However, this won't give us an equal probability of getting each integer in the range 1..7. Can you see why?

There are at least two ways to show that different results of `rand7Mod()` have different probabilities of occurring:

1. Count the number of outcomes of our two `rand5()` calls which give each possible result of `rand7Mod()`
2. Notice something about the *total* number of outcomes of two calls to `rand5()`

**If we count the number of ways to get each result of `rand7Mod()`:**

| result of `rand7Mod()` | # pairs of `rand5()` results that give that result |
|---|---|
| 1 | 4 |
| 2 | 3 |
| 3 | 3 |
| 4 | 3 |
| 5 | 3 |
| 6 | 4 |
| 7 | 5 |

So we see that, for example, there are five outcomes that give us 7 but only three outcomes that give us 5. We're almost twice as likely to get a 7 as we are to get a 5!

But even without counting the number of ways to get each possible result, we could have **noticed something about the *total* number of outcomes of two calls to `rand5()`** , which is 25 (5*5). If each of our 7 results of `rand7Mod()` were equally probable, we'd need to have the same number of outcomes for each of the 7 integers in the range 1..7. That means *our total number of outcomes would have to be divisible by* 7, and 25 is not.

Okay, so `rand7Mod()` won't work. **How do we get equal probabilities for each integer from 1 to 7?**

Is there some number of calls we can make to `rand5()` to get a number of outcomes that *is* divisible by 7?

When we roll our die $n$ times, we get $5^n$ possible outcomes. **Is there an integer $n$ that will give us a $5^n$ that's evenly divisible by 7?**

No, there isn't.

That might not be obvious to you unless you've studied some number theory. It turns out every integer can be expressed as a product of prime numbers (its prime factorization). It also turns out that every integer has only *one* prime factorization.

Since 5 is already prime, any number that can be expressed as $5^n$ (where $n$ is a positive integer) will have a prime factorization that is all 5s. For example, here are the prime factorizations for $5^2, 5^3, 5^4$:

$$5^2 = 25 = 5 * 5$$

$$5^3 = 125 = 5 * 5 * 5$$

$$5^4 = 625 = 5 * 5 * 5 * 5$$

7 is also prime, so if any power of 5 were divisible by 7, 7 would be in its prime factorization. But 7 can't be in its prime factorization because its prime factorization is all 5s (and it has only *one* prime factorization). So no exponent of 5 is divisible by 7. BAM MATHPROOF.

So no matter how many times we run `rand5()` we won't get a number of outcomes that's evenly divisible by 7. What do we dooooo!?!?

Let's ignore for a second the fact that 25 isn't evenly divisible by 7. We can think of our 25 possible outcomes from 2 calls to rand5 as a set of 25 "slots" in an array:

```
int results[25] = {
    0,0,0,0,0,
    0,0,0,0,0,
    0,0,0,0,0,
    0,0,0,0,0,
    0,0,0,0,0
};
```

Which we could then *try* to evenly distribute our 7 integers across:

```
int results[25] = {
    1,2,3,4,5,
    6,7,1,2,3,
    4,5,6,7,1,
    2,3,4,5,6,
    7,1,2,3,4
};
```

It *almost works*. We could randomly pick an integer from the array, and the chances of getting any integer in 1..7 are *pretty evenly* distributed. Only problem is that extra 1,2,3,4 in the last row.

Any way we can sidestep this issue?

What if we just "throw out" those extraneous results in the last row?

21 is divisible by 7. So if we just "throw out" our last 4 possible outcomes, we have a number of outcomes that are evenly divisble by 7!

But what should we do if we get one of those 4 "throwaway" outcomes?

We can just try the whole process again!

Okay, this'll work. But how do we translate our two calls to `rand5()` into the right result from our array?

What if we made it a *2-dimensional* array?

```C++
int results[5][5] = {
    {1,2,3,4,5},
    {6,7,1,2,3},
    {4,5,6,7,1},
    {2,3,4,5,6},
    {7,1,2,3,4}
};
```

Then we can simply treat our first roll as the row and our second roll as the column. We have an equal chance of choosing any column and any row, and there are never two ways to choose the same cell!

```C++
int rand7Table()
{
    const int results[5][5] = {
        {1,2,3,4,5},
        {6,7,1,2,3},
        {4,5,6,7,1},
        {2,3,4,5,6},
        {7,0,0,0,0}
    };

    // do our die rolls
    int row = rand5() - 1;
    int column = rand5() - 1;

    // case: we hit an extraneous outcome
    // so we need to re-roll
    if (row == 4 && column > 0) {
        return rand7Table();
    }

    // our outcome was fine. return it!
    return results[row][column];
}
```

This'll work. But we can clean things up a bit.

By using recursion we're incurring a space cost in the call stack, and risking stack overflow⮥ . This is especially scary because our function *could* keep rerolling indefinitely (though it's unlikely).

How can we rewrite this iteratively?

Just wrap it in a while loop:

```C++
int rand7Table()
{
    const int results[5][5] = {
        {1,2,3,4,5},
        {6,7,1,2,3},
        {4,5,6,7,1},
        {2,3,4,5,6},
        {7,0,0,0,0}
    };

    while (true) {

        // do our die rolls
        int row = rand5() - 1;
        int column = rand5() - 1;

        // case: we hit an extraneous outcome
        // so we need to re-roll
        if (row == 4 && column > 0) {
            continue;
        }
        // our outcome was fine. return it!
        return results[row][column];
    }
}
```

One more thing: we don't *have* to put our whole 2-d results array in memory. Can you replace it with some arithmetic?

We could start by coming up with a way to translate each possible *outcome* (of our two `rand5()` calls) into a different integer in the range 1..25. Then we simply mod the result by 7 (or throw it out and try again, if it's one of the last 4 "extraneous" outcomes).

How can we use math to turn our two calls to `rand5()` into a unique integer in the range 1..25?

What did we do when we went from a 1-dimensional array to a 2-dimensional one above? We cut our set of outcomes into sequential slices of 5.

How can we use math to make our first roll select which slice of 5 and our second roll select which item within that slice?

We could take *something* like:

```
int outcomeNumber = roll1 * 5 + roll2;
```

C++

But since each roll gives us an integer in the range 1..5 our lowest possible outcome is two 1s, which gives us $5 + 1 = 6$, and our highest possible outcome is two 5s, which gives us $25 + 5 = 30$. So we need to do some adjusting to ensure our outcome numbers are in the range 1..25:

```
int outcomeNumber = ((roll1-1) * 5 + (roll2-1)) + 1;
```

C++

(If you're a math-minded person, you might notice that we're essentially treating each result of `rand5()` as a digit in a two-digit base-5 integer. The first roll is the fives digit, and the second roll is the ones digit.)

Can you adapt our function to use this math-based approach instead of the `results` array?

## Solution

Because `rand5()` has only 5 possible outcomes, and we need 7 possible results for `rand7()`, we need to call `rand5()` at least twice.

When we call `rand5()` twice, we have $5 * 5 = 25$ possible outcomes. If each of our 7 possible results has an equal chance of occuring, we'll need each outcome to occur in our set of possible outcomes *the same number of times*. So our total number of possible outcomes must be divisible by 7.

25 isn't evenly divisible by 7, but 21 is. So when we get one of the last 4 possible outcomes, we throw it out and roll again.

So we roll twice and translate the result into a unique `outcomeNumber` in the range 1..25. If the `outcomeNumber` is greater than 21, we throw it out and re-roll. If not, we mod by 7 (and add 1).

```C++
int rand7()
{
    while (true) {
        // do our die rolls
        int roll1 = rand5();
        int roll2 = rand5();

        int outcomeNumber = (roll1 - 1) * 5 + (roll2 - 1) + 1;

        // if we hit an extraneous
        // outcome we just re-roll
        if (outcomeNumber > 21) {
            continue;
        }

        // our outcome was fine. return it!
        return outcomeNumber % 7 + 1;
    }
}
```

## Complexity

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

## What We Learned

As with the previous question about writing a `rand5()` function (/question/simulate-5-sided-die), the requirement to "return each integer with equal probability" is a real sticking point.

Lots of candidates come up with clever $O(1)$-time solutions that they are *so sure* about. But their solutions *aren't actually uniform* (in other words, they're not *truly random*).

In fact, it's *impossible* to have true randomness *and* non-infinite worst-case runtime.

If you don't understand why, go back over our proof using "prime factorizations," a little ways down in the breakdown section.

---