Interview Cake

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:

```
def rand7_mod():
    return (rand5() + rand5()) % 7 + 1
```

Python 3.6

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 `rand7_mod()` have different probabilities of occurring:

1. Count the number of outcomes of our two `rand5()` calls which give each possible result of `rand7_mod()`
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 `rand7_mod()`:

result of `rand7_mod()`	# 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 `rand7_mod()` 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 `rand7_mod()` 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):

> Every number can be expressed as a product of prime numbers. This is called its **prime factorization**.
>
> For example:
>
> $$8 = 2 * 2 * 2$$
>
> $$15 = 5 * 3$$
>
> $$864 = 2 * 2 * 2 * 2 * 2 * 3 * 3 * 3$$
>
> $$13 = 13$$
>
> **Every positive integer has *only one* prime factorization.** This is called the "fundamental theorem of arithmetic."

). 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 power 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 a list:

```
results = [
    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,
]
```

Python 3.6

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

```
results = [
    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,
]
```

Python 3.6

It *almost works*. We could randomly pick an integer from the list, 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 divisible 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 list?

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

```
results = [
    [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],
]
```

Python 3.6

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!

Python 3.6

```
def rand7_table():
    results = [
        [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
    row = rand5() - 1
    column = rand5() - 1

    # Case: we hit an extraneous outcome
    # so we need to re-roll
    if row == 4 and column > 0:
        return rand7_table()

    # 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.

Overview

The **call stack** is what a program uses to keep track of function calls. The call stack is made up of **stack frames**—one for each function call.

For instance, say we called a function that rolled two dice and printed the sum.

Python 3.6

```
def roll_die():
    return random.randint(1, 6)

def roll_two_and_sum():
    total = 0
    total += roll_die()
    total += roll_die()
    print(total)

roll_two_and_sum()
```

First, our program calls `roll_two_and_sum()`. It goes on the call stack:

```
roll_two_and_sum()
```

That function calls `roll_die()`, which gets pushed on to the top of the call stack:

```
roll_die()
```

```
roll_two_and_sum()
```

Inside of `roll_die()`, we call `random.randint()`. Here's what our call stack looks like then:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

When `random.randint()` finishes, we return back to `roll_die()` by removing ("popping") `random.randint()`'s stack frame.

```
roll_die()
```

```
roll_two_and_sum()
```

Same thing when `roll_die()` returns:

```
roll_two_and_sum()
```

We're not done yet! `roll_two_and_sum()` calls `roll_die()` *again*:

```
roll_die()
```

```
roll_two_and_sum()
```

Which calls `random.randint()` again:

```
random.randint()
```

```
roll_die()
```

```
roll_two_and_sum()
```

`random.randint()` returns, then `roll_die()` returns, putting us back in `roll_two_and_sum()`:

```
roll_two_and_sum()
```

Which calls `print()()`:

```
print()()
```

```
roll_two_and_sum()
```

What's stored in a stack frame?

What *actually* goes in a function's stack frame?

A stack frame usually stores:

- Local variables
- Arguments passed into the function
- Information about the caller's stack frame
- The *return address*—what the program should do after the function returns (i.e.: where it should "return to"). This is usually somewhere in the middle of the caller's code.

> Some of the specifics vary between processor architectures. For instance, AMD64 (64-bit x86) processors pass some arguments in registers and some on the call stack. And, ARM processors (common in phones) store the return address in a special register instead of putting it on the call stack.

The Space Cost of Stack Frames

Each function call creates its own stack frame, taking up space on the call stack. That's important because it can impact the *space complexity* of an algorithm. *Especially* when we use **recursion**.

For example, if we wanted to multiply all the numbers between 1 and n, we could use this recursive approach:

Python 3.6

```
def product_1_to_n(n):
    return 1 if n <= 1 else n * product_1_to_n(n - 1)
```

What would the call stack look like when `n = 10`?

First, `product_1_to_n()` gets called with `n = 10`:

```
product_1_to_n()    n = 10
```

This calls `product_1_to_n()` with `n = 9`.

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Which calls `product_1_to_n()` with `n = 8`.

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

And so on until we get to `n = 1`.

```
product_1_to_n()    n = 1
```

```
product_1_to_n()    n = 2
```

```
product_1_to_n()    n = 3
```

```
product_1_to_n()    n = 4
```

```
product_1_to_n()    n = 5
```

```
product_1_to_n()    n = 6
```

```
product_1_to_n()    n = 7
```

```
product_1_to_n()    n = 8
```

```
product_1_to_n()    n = 9
```

```
product_1_to_n()    n = 10
```

Look at the size of all those stack frames! The entire call stack takes up $O(n)$ space. That's right—we have an $O(n)$ space cost even though our function itself doesn't create any data structures!

What if we'd used an iterative approach instead of a recursive one?

Python 3.6

```
def product_1_to_n(n):
    # We assume n >= 1
    result = 1
    for num in range(1, n + 1):
        result *= num

    return result
```

This version takes a constant amount of space. At the beginning of the loop, the call stack looks like this:

```
product_1_to_n()    n = 10, result = 1, num = 1
```

As we iterate through the loop, the local variables change, but we stay in the same stack frame because we don't call any other functions.

```
product_1_to_n()    n = 10, result = 2, num = 2
```

```
product_1_to_n()    n = 10, result = 6, num = 3
```

```
product_1_to_n()    n = 10, result = 24, num = 4
```

In general, even though the compiler or interpreter will take care of managing the call stack for you, it's important to consider the depth of the call stack when analyzing the space complexity of an algorithm.

Be especially careful with recursive functions! They can end up building huge call stacks.

> What happens if we run out of space? It's a **stack overflow**! In Python 3.6, you'll get a `RecursionError`.

> If the *very last* thing a function does is call another function, then its stack frame might not be needed any more. The function *could* free up its stack frame before doing its final call, saving space.
>
> This is called **tail call optimization** (TCO). If a recursive function is optimized with TCO, then it may not end up with a big call stack.
>
> In general, most languages *don't* provide TCO. Scheme is one of the few languages that guarantee tail call optimization. Some Ruby, C, and Javascript implementations *may* do it. Python and Java decidedly don't.

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:

Python 3.6

```
def rand7_table():
    results = [
        [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
        row = rand5() - 1
        column = rand5() - 1

        # Case: we hit an extraneous outcome
        # so we need to re-roll
        if row == 4 and 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 list 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 list 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:

```
outcome_number = roll1 * 5 + roll2
```

Python 3.6

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 $1..25$:

```
outcome_number = ((roll1-1) * 5 + (roll2-1)) + 1
```

Python 3.6

(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` list?

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 occurring, 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 `outcome_number` in the range $1..25$. If the `outcome_number` is greater than 21, we throw it out and re-roll. If not, we mod by 7 (and add 1).

Python 3.6

```
def rand7():
    while True:
        # Do our die rolls
        roll1 = rand5()
        roll2 = rand5()
        outcome_number = (roll1-1) * 5 + (roll2-1) + 1

        # If we hit an extraneous
        # outcome we just re-roll
        if outcome_number > 21:
            continue

        # Our outcome was fine. return it!
        return outcome_number % 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.

Ready for more?

Check out our full course ➡

Want more coding interview help?

Check out **interviewcake.com** for more advice, guides, and practice questions.