modpow_one

Let’s consider the function modpow_one(a, b, c) which computes \(a^b \mod c\). This function has many practical applications, including being a key part of the RSA cryptography algorithm.

```c
1 int modpow_one(int a, int b, int c)
2 { //@requires a >= 0 && b >= 0 && c > 0;
3     //@requires c - 1 <= int_max()/max(a, c - 1);
4     //@ensures 0 <= \result && \result < c;
5     { 6         int res = 1 % c;
7             while (b > 0)
8                 { //@loop_invariant 0 <= res && res < c;
9                     res *= a;
10                     res = res % c;
11                     b--; }
12             return res;
13         }
14 }
```

Prove that this function satisfies its postcondition.

**Solution:**

**Precondition and initial lines of code imply loop invariant.** By the precondition on line 2, we know that \(c > 0\). In addition, we set \(res\) equal to \(1 \mod c\) (which must be at least 0 and less than \(c\) since \(0 < c\) and \(0 <= 1\)) on line 6. So, since \(0 <= (1 \mod c) && (1 \mod c) < c\), we know the loop invariant holds initially.

**Preservation of the loop invariant.** Assume that at the start of some iteration of the loop,
\[0 <= res && res < c\).

We know \(res' = (a \times res) \mod c\) (this doesn’t overflow since \(res <= c - 1\) and \(c - 1 <= int_max() / a\), and doesn’t cause division errors since \(c > 0\)).

Since \(res \times a\) doesn’t overflow and both \(res\) and \(a\) are non-negative, \(res \times a\) is non-negative. Further, \(c\) is positive, so by the definition of the modulo operator \(0 <= (res \times a) \mod c < c\). Hence, \(0 <= res' < c\) and so the loop invariant is preserved.

**Loop invariant and negated loop guard imply postcondition** In this case, we don’t need the negated loop guard. By the loop invariant, \(0 <= res && res < c\).

We return \(res\), so \(0 <= \result && \result < c\).

**Termination** When we start, \(b >= 0\). Each iteration of the loop, we decrement \(b\), so \(b\) will eventually be 0 and we’ll break out of the loop.
modpow_two

Now we’ll look at a different implementation, modpow_two.

```c
1 int modpow_two(int a, int b, int c)
2 //@requires a >= 0 && b >= 0 && c > 0;
3 //@requires (c - 1) <= int_max()/max(a, c - 1);
4 //@ensures \result == modpow_one(a, b, c);
5 {
6     int res = 1 % c;
7     int pow = 0;
8     while (pow < b)
9         //-------------------------------------------------------------------------
10         //-------------------------------------------------------------------------
11         // 12         // 13         // 14         if (0 < pow && pow <= b/2) {
15             res *= res;
16             res = res % c;
17             pow *= 2;
18         }
19         else {
20             res *= a;
21             res = res % c;
22             pow++;
23         }
24     }
25     return res;
26 }
```

Is this function asymptotically faster than, slower than, or the same speed as modpow_one? Explain.

**Solution:** This is asymptotically the same speed as modpow_one. This is because once `pow > b/2` we must run at worst `b/2` steps. \( \frac{b}{2} \leq \frac{1}{2} \times b \) for all `b`, so modpow_one is \( O(b) \), just as modpow_one is.

(In practice, modpow_two is faster than modpow_one, since the part of the loop where `pow <= b/2` is much much faster than the first half of the modpow_one loop, but asymptotically they are the same speed.)

Write loop invariants for modpow_two.

**Solution:** From looking at the body of the loop, we can see that `pow` keeps track of the current power we’ve raised `a` to.

At the end of the function, we want to return `modpow_one(a, b, c)`. We return `res`, so it’d be helpful if our loop invariant told us something about that. Since `pow` is the current power, a relevant loop invariant is `//@loop_invariant res == modpow_one(a, pow, c);`.

But just that alone isn’t strong enough. We also need some way of making sure that `pow == b` at the end—otherwise, we won’t be able to prove our postcondition.

So, we can have a loop invariant `//@loop_invariant 0 <= pow && pow <= b;`

So, our loop invariants are:
Now, prove that if the preconditions to modpow_two are satisfied, it satisfies its postcondition.

If it helps, you can assume that $0^0 = 0$, even though it’s actually indeterminate. You can also assume that modpow_one obeys the properties that

\[(modpow\_one(a, b, c) \times a) \mod c = modpow\_one(a, b + 1, c)\] and
\[(modpow\_one(a, b, c) \times modpow\_one(a, b, c)) \mod c = modpow\_one(a, 2 \times b, c)\]

**Solution:**

** Preconditions and initial lines of code imply loop invariant** We set pow to 0 on line 7 and we know $b \geq 0$ by the precondition, so $0 \leq pow \&\& pow \leq b$.

We’ve set res to $1 \mod c$ (on line 6), and pow is 0. modpow_one(a, 0, c) is equivalent to $1 \mod c$, since $a^0 = 1$ for any $a$. So, res $\equiv$ modpow_one(a, pow, c).

Thus, the loop invariants hold before the first iteration of the loop.

**Preservation of loop invariants** Assume $0 \leq pow \&\& pow \leq b$ and res $\equiv$ modpow_one(a, pow, c).

We split into cases.

---

If $0 < pow$ and $pow \leq b/2$, then: res’ $\equiv$ (res * res) $\mod c$ and pow’ $\equiv$ pow $\times 2$.

By the loop invariant, this means that res’ $\equiv$ (modpow_one(a, pow, c) * modpow_one(a, pow, c)) $\mod c$.

But, by our assumption above, this is equal to modpow_one(a, 2 * pow, c).

Since pow’ $\equiv$ 2 * pow, this means that res’ $\equiv$ modpow_one(a, pow’, c). Thus, the second loop invariant holds.

The first invariant holds since pow $\leq b/2$ and pow’ $\geq 2 \times pow$. That means that pow’ $\leq b$ (division rounds down, so this can’t possibly be greater than b). We know $0 \leq pow$ since we increased pow and there was no overflow.

---

In the second case, res’ $\equiv$ (res * a) $\mod c$ and pow’ $\equiv$ pow + 1.

The first loop invariant is preserved since pow $< b$ (by the loop guard), so pow’ $\leq b$. We know pow’ $\geq pow$ and pow $\geq 0$ by the loop invariant, so pow’ $\geq 0$. So, the first invariant is preserved in this case.

res’ $\equiv$ (modpow_one(a, pow, c) * a) $\mod c$, which by our assumption is equal to modpow_one(a, pow + 1, c).

Since pow’ $\equiv$ pow + 1, this means res $\equiv$ modpow_one(a, pow’, c). Thus, the second loop invariant is preserved in this case.

Thus, both loop invariants are preserved.
Loop invariants and negated loop guard imply postcondition  The negated loop guard is \( pow \geq b \). The first loop invariant tells us that \( pow \leq b \). Thus, \( pow = b \).

By the second loop invariant, \( res = modpow_one(a, pow, c) \). But since \( pow = b \), this means that \( res = modpow_one(a, b, c) \).

We return res, so our postcondition is satisfied.

Termination  pow starts out at 0 and is strictly increasing, so it will eventually be as large as b. At that point, the loop terminates. (pow won’t overflow since b is a positive int)

Thus, pow\_fast returns the same result as pow\_slow.