$e \geq 0$:
$b^e * x^y$ } LI

When loop is exited :

① $e > 1$ becomes false

This means that $e \leq 1$

② LI are true.

$e \geq 0$

$b^e * x^y$

At L^* we have

$e \leq 1$ & $e \geq 0$

1 either $e = 0$

2 or $e = 1$

```
1  int POW(int x, int y)
2  //@requires y >= 0;
3  {
4      if (y == 0) {
5          return 1;
6      }
7      else {
8          return POW(x, y-1) * x;
9      }
10 }
11
12 int f(int x, int y)
13 //@requires y >= 0;
14 //@ensures \result == POW(x,y);
15 {
16     int b = x;
17     int e = y;
18     int r = 1;
19
20     while (e > 1)          <- loop guard
21     //@loop_invariant e >= 0;
22     //@loop_invariant POW(b, e)*r == POW(x, y);
23     {
24         if (e % 2 == 1) {
25             r = b * r;
26         }
27         b = b * b;
28         e = e / 2;
29     }
30     return r * b;          <- L*
31 }
```

Case I: $e = 0$:

$x^y = b^e * r$ L22.

$x^y = b^0 * r = r$

Case II: $e = 1$:

$x^y = b^e * r = b * r$

Bug occurs when $e = 0$ at L30.

at least once

if loop is entered, then $e = 1$ at L30.

if loop is never entered then $e = 0$ {input}.

Correcting the code :-

① put a special case saying when $y = 0$ return 1. ⟷ insert at L16

② Change the precondition to $y \geq 0$

```
1  int POW(int x, int y)
2  //@requires y >= 0;
3  {
4      if (y == 0) {
5          return 1;
6      }
7      else {
8          return POW(x, y-1) * x;
9      }
10 }
11
12 int f(int x, int y)
13 //@requires y >= 0;
14 //@ensures \result == POW(x,y);
15 {
16     int b = x;
17     int e = y;
18     int r = 1;
19
20     while (e > 1)
21     //@loop_invariant e >= 0;
22     //@loop_invariant POW(b, e)*r == POW(x, y);
23     {
24         if (e % 2 == 1) {
25             r = b * r;
26         }
27         b = b * b;
28         e = e / 2;
29     }
30     return r * b;
31 }
```

③ Why not use POW instead of f ? What is the advantage of using f instead of POW.

Ex:- trace the code on values $x = 2$, $y = 8$ for both POW & f.

f requires 4 steps while POW require 8+ steps.

So; f is way-way faster.

④ Change LG to $e \geq 0$.

when we exit the loop: (I is true & LG is false.

$e \geq 0$ ⟷ $e \leq 0$

$e = 0$

On changing the loop guard to e>0, we must return x as $x^y = z$ and the value of e at L30 is 0.

// insert

//@ assert e==0;

after L29 & before line L30.

```
12 int f(int x, int y)
13 //@requires y >= 0;
14 //@ensures \result == POW(x,y);
15 {
16     int b = x;
17     int e = y;
18     int r = 1;
19
20     while (e > 0)
21     //@loop_invariant e >= 0;
22     //@loop_invariant POW(b, e)*r == POW(x, y);
23     {
24         if (e % 2 == 1) {
25             r = b * r;
26         }
27         b = b * b;
28         e = e / 2;
29     }
30     //@assert e == 0;
31     return r;
32 }
```

e ← e changes

L G

Shown till now :-

If our program reaches L31 then we would output the correct answer.

But does it ever reach there?

while (e > 0){

//.....

e = e/2;

}

Do we ever exit the loop?

* look for loop guard variables - where are they getting changed?

1 $e = 0$ never enters the loop.

2 $e \geq 1$ loop is entered but we will break out of the loop because e keeps on decreasing.

$e = 7$

$e = e/2 = 3$

$er_2 = 1$

$er_2 = 0$

If $e \geq 1$ then e' (the value of e at the end of the loop) is strictly less than e.

① $e \geq 0$ L.I.

② $e' = e/2$

③ $e' < e$

④ $e' \geq 0$

← This proves that our loop terminates

8, 7, 6 0

Remember:- If a seq of ints is strictly decreasing and lower bounded then it is finite.

similarly for [decreasing ↔ increasing
lower ↔ upper

Proving correctness when we have one loop:-

pre condition, post conditions.

① → INIT ← before the loop LI
→ PRES ← loop body preserves LI

② EXIT: LI & negation of LG gives us post condition.

③ TERM: exiting out of loop.
↳ loop guard fails at some point.