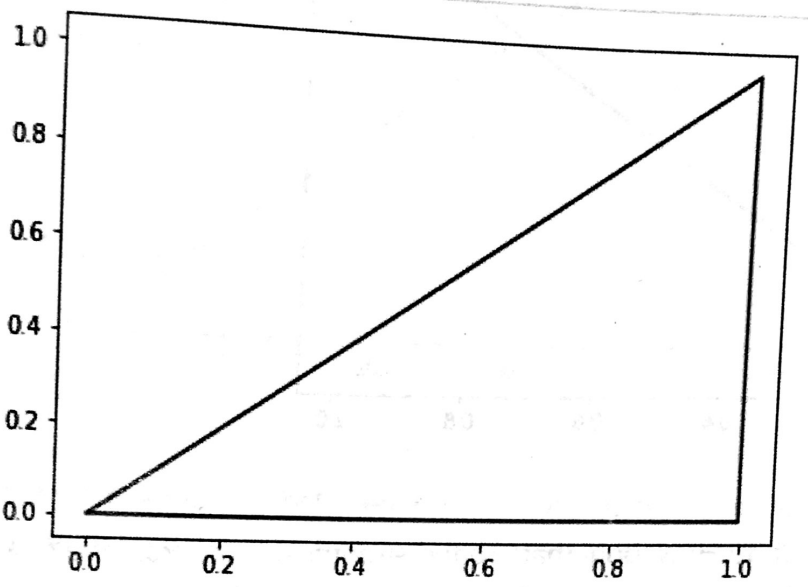
971]:
```
x = np.linspace(0.,1.,1001);
original = x
n = 100
for n in range(n):
    x = np.sqrt(x)
for n in range(n):
    x = x**2

plt.plot(original,original,"-r") #Show the 1:1 line
plt.plot(original,x) #Show how the values have changed.
plt.show()
```

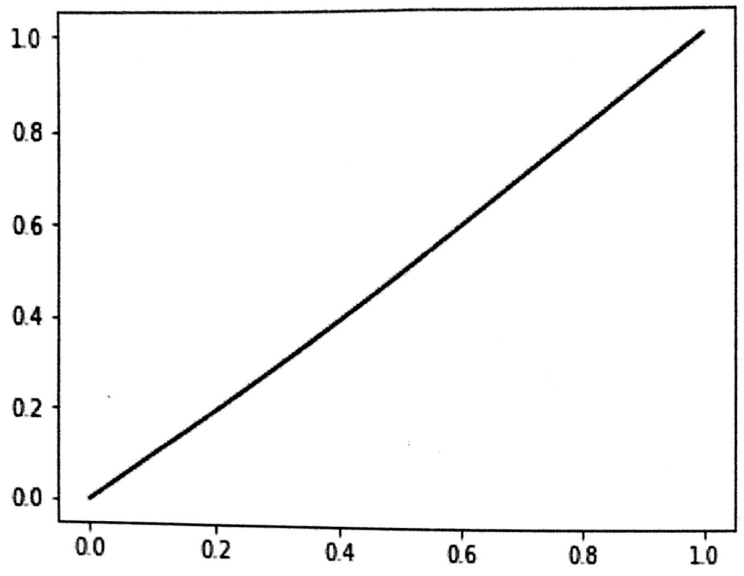


(When using the lower n, we get more steps.)

Changing the number of iterations can alter the above effect. Doing more iterations produces fewer steps with larger error, while doing fewer produces more steps with less error. So it seems to be an issue with compounding error from square-rooting. Now, what would instead happen if we were to not do this iteratively, and just raise x to the appropriate powers?

In [139]:
```
x = np.linspace(0.,1.,1001);
original = x
x = (x**(1./(104.)))**104.

plt.plot(original,original,"-r") #Show the 1:1 line
plt.plot(original,x) #Show how the values have changed.
plt.show()
```



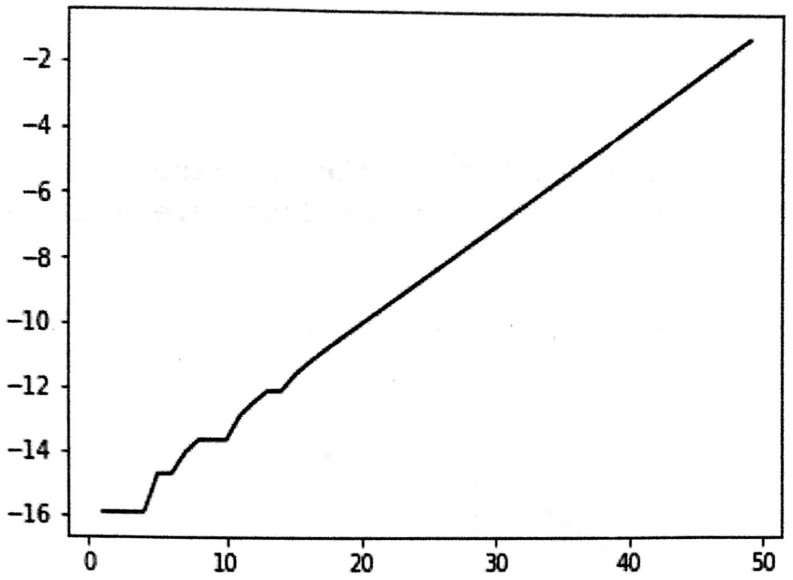
When we do this, there is no error at all! How 'bout that. So the error must come from the fact that we repeatedly took a square root, not from the process of taking x to some small power. What would happen to a value that we perturb by a certain amount? Where is the error creeping in? For convenience, let's just define a function, oops(x).

In [141]:
```
def oops(x,n=52):
    for i in range(n):
        x=np.sqrt(x)
    for i in range(n):
        x=x**2
    return x
```

In [255]:
```
plot = [0.999999999999-oops(0.999999999999,n=i) for i in range(50)]
```

```
plt.plot(np.log10(plot))
plt.show()
```

```
/anaconda3/lib/python3.6/site-packages/ipykernel_launcher.py:1: Runt
imeWarning: divide by zero encountered in log10
  """Entry point for launching an IPython kernel.
```



It is clear now that the discrepancy between a number and its oops() value grows exponentially with the number of iterations in the function. It begins near the machine eps, and then grows to appreciable sizes. What I suspect is that we are losing significant digits in our answer in each step, and the number continues to grow each time. This means that the error should be growing as 2^n. Let's check.

In [ ]:
```
x = np.arange(50.)
y = 2.**(x)*10**(-15.)
plt.plot(x,np.log10(y))
plt.plot(np.log10(plot))
plt.show()
```

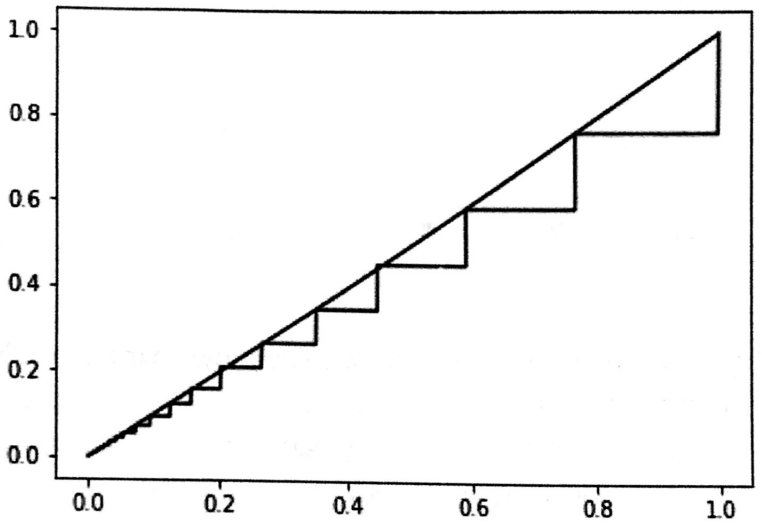
(This was a plot that just showed a line for 2^n following the previous plot.)

Sure enough, it seems to be growing exactly as 2^n. This explanation seems to suffice for how the error grows then. But what's actually happening? Let's see if we can produce the same effect by merely rounding to say, 4 significant figures. In order to do this, we'll multiply by 10^4 and then round to the nearest integer, then divide by 10^4.

In [298]:
```
def rnd(x,sigfigs=3):
    return np.rint(x*10.**sigfigs)*10.**(-sigfigs)
```

In [313]:
```
x = np.linspace(0.,1.,1001);
original = x
n = 8
for i in range(n):
    x = rnd(np.sqrt(x))
for i in range(n):
    x = rnd(x*x)


plt.plot(original,original,"-r") #Show the 1:1 line
plt.plot(original,x,"-g") #Show how the values have changed.
plt.show()
```



Sure enough, we recover the behavior, and for a much smaller number of iterations. As we successively take square roots, rounding the result each time, values are mapped to something closer and closer to 1. The closer they are to 1 to begin with, the closer they get to 1. The first "step" below 1 is a bin where all of the values in that bin were mapped to 1 within roundoff error. The second step is those that were within two roundoff errors of 1. So within a step, many values were mapped to the same value, then mapped back to the same value. The difference between them was beyond what the machine could discern, and so they become degenerate with one another. Why does this become more of a problem at larger values of x, but not at smaller values? At smaller values, they are put into relatively more "bins", so the steps become smaller and smaller. Interestingly, if we reverse our order of operations so that we first take n squares of x and then we take n squareroots of x, the small values are what are bad, while the large values are more accurate. This is because successively squaring a small number yields zero to within machine precision quite quickly, and then the square root of zero is still zero.