Fast modular exponentiation, help me find the mistake

Asked 5 years, 9 months ago Modified 5 years, 9 months ago Viewed 4k times

4

2

I am trying to implement a scheme of fast exponentiation. Degree is represented in binary form:

```
def pow_h(base, degree, module):
    degree = bin(degree)[2:]
    r = 1

    for i in range(len(degree) - 1, -1, -1):
        r = (r ** 2) % module
        r = (r * base ** int(degree[i])) % module

    return r
```

But function is not working properly, where is the mistake?

$$x^a \pmod{n} = (\ldots(\ldots x^{a_{k-1}})^2 x^{a_{k-2}})^2 \ldots x^{a_1})^2 x^{a_0} \pmod{n}$$

python algorithm

Share Edit Follow Flag

edited Nov 13, 2016 at 20:38

OneCricketeer
158k 18 118 219

asked Nov 13, 2016 at 20:37

Andrii Matiiash
479 8 17

add `print()` in function to see what values you get - and compare with own calculations on paper. – furas Nov 13, 2016 at 20:46

FWIW, the built-in `pow` function accepts a modulus as an optional 3rd argument. – PM 2Ring Nov 13, 2016 at 21:00

2 Answers

Sorted by:
Trending sort available
Highest score (default)

4

As I said in the comments, the built-in `pow` function already does fast modular exponentiation, but I guess it's a reasonable coding exercise to implement it yourself.

Your algorithm is close, but you're squaring the wrong thing. You need to square `base`, not `r`, and you should do it after the multiplying step.

```
def pow_h(base, degree, module):
    degree = bin(degree)[2:]
    r = 1
    for i in range(len(degree) - 1, -1, -1):
        r = (r * base ** int(degree[i])) % module
        base = (base ** 2) % module
    return r

#test

for i in range(16):
    print(i, 2**i, pow_h(2, i, 100))
```

output

```
0 1 1
1 2 2
2 4 4
3 8 8
4 16 16
5 32 32
6 64 64
7 128 28
8 256 56
9 512 12
10 1024 24
11 2048 48
12 4096 96
13 8192 92
14 16384 84
15 32768 68
```

Using `r * base ** int(degree[i])` is a cute trick, but it's probably more efficient to use a `if` statement than exponentiation. And you can use arithmetic to get the bits of `degree`, rather than using string, although `bin` *is* rather efficient. Anyway, here's my version:

```
def pow_h(base, power, modulus):
    a = 1
    while power:
        power, d = power // 2, power % 2
        if d:
            a = a * base % modulus
        base = base * base % modulus
    return a
```

Share Edit Follow Flag

answered Nov 13, 2016 at 21:06

PM 2Ring
52.8k 5 76 167

I might use `power, d = divmod(power, 2)` instead. – chepner Nov 13, 2016 at 21:26

@chepner: I occasionally use `divmod` for its elegance, but in my `timeit` tests I've found `divmod(a,`

`b)` slightly slower than just using `a // b, a % b` . But that was on Python 2.6 on a rather old machine, so YMMV. – PM 2Ring Nov 13, 2016 at 21:59

Agreed; I tested it a few times first and got wildly varying results, with the `//-%` pair being 1-15x faster. That range, and the fact that `divmod` times remained stable, suggested some caching or a poor test setup, so I went with a more neutral comment. – chepner Nov 13, 2016 at 22:03

2

Such fast exponentiation must act differently if the current exponent is even or odd, but you have no such check in your code. Here are some hints:

To find `x**y` , you need an "accumulator" variable to hold the value calculated so far. Let's use `a` . So you are finding `a*(x**y)` , with your code decreasing `y` and increasing `a` and/or `x` until `y` becomes zero and `a` is your final answer.

If `y` is even, say `y==2*k` , then `a*x**(2*k) == a*(x**2)**k` . This decreased `y` to `y//2` and increased `x` to `x**2` .

If `y` is odd, say `y==2k+1` , then `a*x**(2*k+1) == (a*x)*x**(2*k)` . This decreased `y` to `y-1` and increased `a` to `a*x` .

You should be able to figure the algorithm from here. I did not include using the modulus: that should be easy.

Share Edit Follow Flag

answered Nov 13, 2016 at 20:49

Rory Daulton
20.9k 5 39 48