Integer Factorization

Chinese Remainder Theorem

Modular Exponentiation

Reading: Modular Exponentiation
10 min

Reading: Fast Modular Exponentiation
7 min

Quiz: Fast Modular Exponentiation: Code
2 questions

Reading: Fermat's Little Theorem
10 min

Reading: Euler's Theorem
10 min

Quiz: Modular Exponentiation
4 questions

Modular Exponentiation

Our next results (Fermat's little theorem and its generalization, Euler's totient theorem) deal with modular exponentiation, i.e., with expressions of type $a^b \bmod c$ where a, b, c are integers. But before stating and proving this result, let us discuss the purely algorithmic question: how to compute $a^b \bmod c$ reasonably fast?

Stop and think! How would you compute $314^{271} \bmod 123$ and $314159265358^{2718281828} \bmod 123456789$?

Computing this in python directly gives the following result:

```
1  print((314 ** 271) % 123)
2  print((314159265358 ** 2718281828) % 123456789)
```

Run

Reset

```
Evaluation took more than 5 seconds to complete.
            Please try again with a simpler expression.
```

```
1  38
```

Stop and think! Do you see why python is not able to compute the second value?

The problem is that python starts by computing the number $314159265358^{2718281828}$ that is huge. One can estimate its size as follows: a^n has approximately n times more digits than a, so the power in question has about 2718281828 times the base size (12) digits and hardly fits into computer memory.

Stop and think! How can we do better?

The right way to compute modular exponentiation in python is to use the built-in `pow` method. The following code produces the result in the blink of an eye. We will learn what is going on under the hood below.

```
1  print(pow(314, 271, 123))
2  print(pow(314159265358, 2718281828, 123456789))
```

Run

Reset

```
38
32073907
```

```
1  38
2  32073907
```

To get a hint, you may think about the following problem.

Problem

Compute $8^{100} \bmod 63$.

Of course, starting with computing 8^{100} is a bad idea. Note that $8^2 = 64 \equiv 1 \pmod{63}$, and

$$8^{100} = 8^2 \cdot 8^2 \cdot \ldots \cdot 8^2 \equiv 1 \cdot 1 \cdot \ldots \cdot 1 \pmod{63},$$

hence the answer is 1. Here, we use that the product does not change modulo m, when we replace the factors by equivalent (modulo m) ones. This (and the same property of sums) was discussed in the previous chapter. (When adding some multiple of m to one of the summands or factors, we change the result by a multiple of m: indeed, $(u + km) + v = (u + v) + km$ and $(u + km)v = uv + kmv$.)

Problem

Prove that $2^{1001} + 3^{1001}$ is divisible by 5.

Hint: $3 \equiv (-2) \pmod{5}$.

Problem

Prove that $a^n - b^n$ is divisible by $a - b$ for integers $a > b$.

Hint: $a \equiv b \pmod{d}$ for $d = a - b$.

Problem

Find $2^{1025} \bmod 17$.

Solution

Since $2^{1024} = 16^{256} \equiv (-1)^{256} \equiv 1 \pmod{17}$, the answer is 2.

Stop and think! Now we can compute $a \bmod c, a^2 \bmod c, a^3 \bmod c, \ldots$ sequentially, multiplying each time by a and keeping only the remainder modulo c. In this way we do not need to store large numbers (only numbers smaller than c). Does this approach work for our example $314159265358^{2718281828} \bmod 123456789$?

The size of numbers is no more a problem, but we need to perform 2718281827 multiplications --- too many. Can we do better? This is a general question for computing powers (that makes sense not only for modular computations).

Problem

Let x be some number. Can you compute x^{64} in less than 63 multiplications (that would be needed if we compute $x, x^2, x^3, \ldots, x^{64}$ sequentially)?

For this problem the answer is especially easy to guess, since 64 is a power or 2 and repeated squaring helps:

$x^2 = x \cdot x, x^4 = x^2 \cdot x^2, x^8 = x^4 \cdot x^4, x^{16} = x^8 \cdot x^8, x^{32} = x^{16} \cdot x^{16}, x^{64} = x^{32} \cdot x^{32}$.

In this way the exponents grow faster ($\times 2$ instead or $+1$ at every step), and we use only 6 multiplications.

In general, to compute x^n for $n = 2^k$ we need $k = \log_2 n$ multiplications.

Stop and think! Now we know how to compute x^n fast if n is a power of two. But what if not? How can we compute, say, x^{65}, or x^{68}, or x^{77}?

For x^{65} we may multiply x^{64} by x. Since we have also computed previous powers of 2, we may use that

$x^{68} = x^{64} \cdot x^4$ and $x^{77} = x^{64} \cdot x^8 \cdot x^4 \cdot x$

(note that $77 = 64 + 8 + 4 + 1$). Similar trick can be used in the general case, as the following problem shows:

Problem

Prove that x^n (for some $n > 1$) can be computed in at most $2 \log_2 n$ multiplications.

(If n is not a power of 2, the value of $\log_2 n$ is not an integer, but we may take the closest integer below $\log_2 n$.)

Solution

Indeed, we can compute by squaring all the powers $x^2, x^4, \ldots, x^{2^k}$ where the exponent does not exceed n. There are at most $\log_2 n$ of them (again we may round $\log_2 n$). Then we represent n as sum of powers of 2 (why is it possible? this is what the binary system is about) and get x^n as the product of corresponding powers.

For the practical viewpoint, it may make sense to implement essentially the same algorithm in the other direction, so to say: use formula

$x^n = (x^2)^{n/2}$

that reduces the exponent twice for even n for the price of one multiplication ($x^2 = x \cdot x$), and use

$x^n = x^{n-1} \cdot x$

that uses one multiplication to reduce to the even case (if n is odd, then $n - 1$ is even).

In this case, we reduce the exponent to 1 (or 0, if we do not treat separately the case of x^1) in about $2 \log_2 n$ steps (in two steps we reduce the exponent at least twice).

```
1   def fast_modular_exponentiation(b, e, m):
2       assert m > 0 and e >= 0
3       if e == 0:
4           return 1
5       if e == 1:
6           return b
7       if e % 2 == 0:
8           return fast_modular_exponentiation((b * b) % m, e // 2, m)
9       else:
10          return (fast_modular_exponentiation(b, e - 1, m) * b) % m
11
12
13  print(fast_modular_exponentiation(314159265358, 2718281828, 123456789))
```

Run

Reset

```
32073907
```

```
1  32073907
```

The program implementing this idea computes $314159265358^{2718281828} \bmod 123456789$ in a fraction of a second.

Problem

Prove that the number of multiplications when computing x^n (for $n \geq 1$) both in this program and in the method descibed above (using binary notation) is the same:

$$(\text{number of bits in } n) + (\text{number of ones in } n) - 2$$

where counting the bits and ones in n we use the binary representation of n.

It turns out that this is not exactly the minimal number of multiplications needed to compute x^n. For example, for x^{15} we get 6 multiplications with both our approaches, while one can compute it in 5 multiplications:

$x^2 = x \cdot x; \quad x^4 = x^2 \cdot x^2; \quad x^5 = x^4 \cdot x; \quad x^{10} = x^5 \cdot x^5; \quad x^{15} = x^{10} \cdot x^5.$

See Wikipedia article about addition chains that minimize the number of multiplications.

Completed **Go to next item**

Like **Dislike** **Report an issue**