Tutorial for Just a simple sum

REVISIONS VIEW

Table of Contents [hide]

Warning - mathematical content ahead!

If you've participated in programming contests before, you'll know that most of the time, the simpler the problem statement, the tougher the problem seems to be. This one looks like it follows the trend, but as long as you are grounded with the correct mathematical knowledge, each step in the solution to this problem was reasonably straightforward - the trick was not getting sidetracked into alternate methods that were never going to work.

Let's start by looking at a couple of key algorithms in problems like these.

Two initial algorithms

Calculating powers

Calculating a large power of a number in modular arithmetic is a common problem, and can be done very quickly. To calculate a^b mod m, we calculate a, a^2, a^4, a^8, a^{16} ... by repeated squaring the previous value. We then multiply together the right terms to reach whatever power of b we want.

We can combine both these steps into one function as follows:

```
result = 1;
power = a;
while (b>0) {
if (b%2==1) result = (result*power)%m;
power = (power*power)%m;
b/=2;
}
```

Make sure you understand how this works - we are basically multiplying in the correct power of *a* every time we see a '1' bit in binary in *b*.

Calculating inverses

If we are working mod *m* and want to divide two numbers (ie find *a*/*b*), we can do so when *b* and *m* share no common factors. In that case, we find the *inverse* of *b* mod *m*, and then multiply this by *a*.

How do we calculate an inverse? By using The Extended Euclidean Algorithm. This isn't a complicated algorithm to learn, and you should store this away for future use so you don't have to rewrite it every time. (Which, as it happens, I always end up doing anyway, because I keep losing it :)) Wikipedia covers this algorithm in detail, so I won't go over it here.

Initial thoughts

We should always start by looking at the constraints. *n* can be huge in this problem - up to 10^{18}. This means we are going to need to look for an algorithm whose speed doesn't really depend on *n* at all - at the worst, with time O(log *n*). But *m* is going to need to be the key value in our algorithm.

We should also see straight away 64 bit integers will be required - while we can reduce calculations mod 200000 at each stage, we could end up with a calculation of 199999*199999 which will not fit inside an int.

The last thing we should see immediately is that as we are doing calculations mod *m*, we can immediately reduce all of the base numbers in each power modulo *m*.

Combining the bases

In order to do things in a faster way than looping through each of the *n* terms, we're going to have to group things together in some way.

After we reduce all of the bases mod *m*, there are going to be lots of powers of 1, lots of powers of 2, all the way up to powers of *m*-1. There will be some powers of 0 as well, but that's not going to affect the sum at all. Perhaps we can work out how to add all the powers with the same base at the same time.

The sum we are looking at is of the form $i^j + i^{j+m} + i^{j+2m} + \ldots + i^{j+(k-1)m}$ for some values of *i* and *k* (we can work out *k* in terms of *i* and *n*). Each term in the sum can be found from the previous term by multiplying by i^m at each stage.

This tells you the sum is a *geometric series*. If we forget about the fact we need to calculate the value mod *m*, there is a formula for calculating the exact sum of such a series:

$a * (r^n - 1) / (r - 1)$

where *a* = the first term, *r* = the fixed ratio between adjacent terms, and *n* is the number of terms we are adding. In this case $a = i^j$, $r = i^m$, and $n = k$.

Now, if *r*-1 and *m* share no common factors, then we can calculate this division mod *m* by multiplying by the inverse of *r*-1 mod *m* instead of the division. But what if *r*-1 and *m* do share common factors? There doesn't seem to be any easy way forward.

Another approach