CS 3510 — Algorithms, Fall 2022

Lecture 5: Modular Arithmetic

January 25, 2022

Lecturer: Frederic Faulkner *Scribe: Aditya Diwakar*

Homework 2 (1B) is out and due on Saturday. Further, I also posted an announcement on how to properly format an algorithm for homeworks.

Modular Arithmetic

What is $2 + 2$? Pretty easy, it's obviously $1 \pmod 3$. Modulo simply means the remainder after dividing two numbers.

$$a \pmod b$$

is equal to the remainder of a after dividing it by b. For example, $37 \pmod 5 = 2$ since $37 = (7 \cdot 5) + 2$. A popular example of modulo arithmetic is clocks where the 24-hour time format is $\pmod{24}$.

A more formal definition for this class is talking about a and b in the modulo classes. In notation, we write

$$a \equiv b \pmod n \quad \text{or} \quad a \equiv_N b$$

if a and b are part of the same modulo class. For example:

$$-2 \equiv_3 1 \equiv_3 4 \equiv_3 7 \equiv_3 10 \quad \text{or} \quad -1 \equiv_3 2 \equiv_3 5 \equiv_3 8$$

In the left, all numbers are $1 \pmod 3$ and in the second, they are $2 \pmod 3$

If $x \equiv_N x'$ and $y \equiv_N y'$, then the following are true:

1. $x + y \equiv_N x' + y'$
2. $xy \equiv_N x'y'$
3. $x^c \equiv_N (x')^c$

These operations are useful, as you can compute calculations such as $321 \cdot 165 \pmod{160}$ where you could naively multiply 321 and 165 first and then apply the modulo *or* you could take the modulo first and only need to multiply $321 \pmod{160} \cdot 165 \pmod{160} \equiv 1 \cdot 5 \pmod{160} \equiv 5$

Similarly, you could compute $2^{210} \pmod{31}$ by doing:

$$2^{210} \equiv_{31} \left(2^5\right)^{42} \equiv_{31} (32)^{42} \equiv_{31} 1^{42} \equiv_{31} 1$$

1 Modular Arithmetic Algorithms

What is the time complexity for the `mod` operation? We can write a division algorithm that returns the quotient and remainder written as:

```
Function Div(x, y):
    if x = 0 then
        return 0, 0
    q, r ← Div(⌊x/2⌋, y)
    q, r ← 2q, 2r
    if x is odd then
        r = r + 1 // error when flooring x/2
    /* r > y means remainder overflowed divisor */
    if r > y then
        q = q + 1 // an additional copy of y fits in x
        r = r − y // reduce remainder by overflow amount
    return q, r
```

For this algorithm, the doubling of q, r are $O(1)$ and all other operations are $O(n)$ as we could have an n long carry when adding numbers together ($q = q + 1, r = r - y$, etc). Hence, the non-recursive work is $O(n)$

The recursive call has a size of $n - 1$ since we divide x by 2 which removes a bit meaning the recurrence relation is:

$$T(n) = T(n-1) + O(n) = O(n^2)$$

This is $O(n^2)$ because there are roughly n calls with roughly $O(n)$.

1.1 Addition under Modulo

What is the time complexity for `addmod` (addition under a modulo)? If we assume that we are given inputs x, y that are both already under $\pmod{n}$, then we don't need to worry about modding them again.

Hence, we can simply perform $(x + y) mod N$ which takes $O(n)$ time and since $x < N, y < N$, then $x + y < 2N$ so if $x + y \geq N$, we can subtract N.

In total, $O(n)$ for addition, $O(n)$ for comparision (subtraction and determining sign), gives a total of $O(n)$ runtime for addition with mod.

1.2 Multiplication under Modulo

What is the time complexity for `multmod` (multiplication under a modulo)? Under the same assumptions as above: $xy \pmod{n}$ requires xy multiplication which can take $O(n^{\log_2 3})$ and then computing the modulo takes $O(n^2)$ making multiplication under modulo an $O(n^2)$ operation.

1.3 Exponentiation under Modulo

What is the time complexity for `modexp` (exponentiation under modulo)? A bad implementation for this could be to multiply:

$$\begin{aligned}
&x \pmod{N} \\
&x^2 \pmod{N} \\
&x^3 \pmod{N} \\
&\vdots \\
&x^y \pmod{N}
\end{aligned}$$

and multiply them together which is a total of y multiplications, giving you $O(2^n)$ multiplications (which is very bad). Rather, we need to compute x taken to a power 2^i for multiple values i.

An alternative method can be constructed after realizing that any number x^y can be written as $x^{2^n y_n + \cdots + 2^1 y_1 + 2^0 y_0}$ where y_i are the bits of y. Due to exponentiation rules, this is equivelant to:

$$x^{2^n y_n + \cdots + 2^1 y_1 + 2^0 y_0} = x^{2^n y_n} \cdots x^{2^1 y_1} \cdot x^{2^0 y_0}$$

Since these constructions are equivelant, we only need to compute x^{2^i} for $i \in [0, n-1]$ as any exponentiated number can be reconstructed with these n numbers. How long does this method take? Since we need to multiply x^{2^i} for $i \in [0, n-1]$, we are doing n multiplications which each take $O(n^2)$ giving a runtime of $O(n^3)$ to compute our table of exponentiated x.

The $O(n^2)$ multiplication comes from the fact that when squaring the number, there is a possibility of overflowing under modulo requiring us to call the `mod` algorithm again (with runtime $O(n^2)$).

Then, to reconstruct x^y, y can have as many n bits potentially requiring n total multiplications which also has a runtime of $O(n^3)$. Hence, the `modexp` algorithm is $O(n^3) + O(n^3) = O(n^3)$.

As an example, let's compute $7^{25} \pmod{23}$ and since $25 = (11001)_2$ then, we need to compute $2^{2^0}, 2^{2^1}, 2^{2^2}, 2^{2^3}, 2^{2^4}$ all under $\pmod{23}$:

$$\begin{aligned}
7^{2^0} &= 7^1 = 7 \\
7^{2^1} &= 7^2 = 49 \equiv_{23} 3 \\
7^{2^2} &= 7^2 \cdot 7^2 \equiv_{23} 3 \cdot 3 \equiv_{23} 9 \\
7^{2^3} &= (7^{2^2})^2 \equiv_{23} 9^2 \equiv_{23} 12 \\
7^{2^4} &= (7^{2^3})^2 \equiv_{23} 12^2 \equiv_{23} 6
\end{aligned}$$

Hence, we can combine these together to compute 7^{25} as:

$$7^{25} = 7^{2^4 + 2^3 = 2^0} = 7^{2^4} 7^{2^3} 7^{2^0} \equiv_{23} 6 \cdot 12 \cdot 7 \equiv_{23} 504 \equiv_{23} 21$$

meaning we can reach our final conclusion of $7^{25} \equiv 21 \pmod{23}$.

1.4 Division under Modulo

What is the time complexity of `divmod` (division under modulo)? Since fractions don't exist within modular arithmetic, this becomes slightly complicated. For example, how can we solve $5x \equiv_7 3$? By inspection, we can find $x = 2$ would give $10 \equiv_7 3$ which is true.

More precisely, if we can find some number $a \cdot 5 = 1$, then we say a is the multiplicative inverse of $5 \pmod 7$. In this case, $a = 3$ as $3 \cdot 5 = 15 \equiv_7 1$.

The multiplicative inverse is powerful because:

$$\begin{aligned} 5x &\equiv_7 3 \\ 3(5x) &\equiv_7 3 \cdot 3 \\ 15x &\equiv_7 9 \\ x &\equiv_7 2 \end{aligned}$$

which is a more robust way of finding that $x = 2$. But, how did we find the multiplicative inverse (the value a from above)? It doesn't always exist!

Theorem. *x only has a multiplicative inverse modulo N if $gcd(x, N) = 1$*

In other words, the inverse only exists if x and N are relatively prime.

2 Greatest Common Denominator

$\gcd(a, b)$ is the largest integer that divides both a and b. How do we find it?

One way is to factor both numbers and pick the largest factors:

$$gcd(24, 54) = gcd(3 \cdot 2 \cdot 2 \cdot 2, 3 \cdot 3 \cdot 3 \cdot 2) = 3 \cdot 2 = 6$$

We picked out $3 \cdot 2$ because both numbers had $3 \cdot 2$ as multiplied factors. However, this method relies on factoring which is an extremely slow operation, so rather: we use the Euclidean Algorithm!

2.1 Euclidean Algorithm

```
Function gcd(x, y):
    if y = 0 then
        return x
    return gcd(x, x (mod y))
```

Performing $x \pmod y$ takes $O(n^2)$ time so the non-recursive work is $O(n^2)$ and we make a single recursive call with size $n-1$ because after two calls of this function, the input goes from $(x, y) \to (x, x \pmod y) \to (x \pmod y, \cdots)$ and $x \pmod y$ is bounded by $x/2$ meaning we have reduced the number of bits by 1 giving a final running time recurrence relation of

$$T(n) = T(n-1) + O(n^2) = O(n^3)$$

This is $O(n^3)$ as we are making roughly n total calls with $O(n^2)$ work per call giving a total of $nO(n^2) = O(n^3)$.

An interesting fact regarding GCDs is that we can always write $gcd(x, y)$ as a linear combination of x and y:

$$gcd(x, y) = ax + by$$

But, what is the important of this form? From an above theorem, a multiplicative inverse only exists for $x \pmod N$ if $gcd(x, N) = 1$.

This form allows us to say:$gcd(x, N) = 1$ then $1 = ax + bN$.

$$\begin{aligned} gcd(x, N) = 1 \implies 1 = ax + bN &\implies 1 \pmod n = (ax + bN) \pmod N \\ &\implies ax \equiv 1 \pmod n \end{aligned}$$

Hence, if there exists an algorithm to find values a, b such that $gcd(x, y) = ax + by$ then we can find the multiplicative inverse if x and N are relatively prime.

The algorithm to find these values is known as the Extended Euclidean algorithm:

Function `extgcd(`x, y`)`**:**

    if $y = 0$ **then**

        return $x, 1, 0$

    $d, a', b' \leftarrow$ `extgcd(`$y, x \pmod y$`)`

    $a'', b'' \leftarrow b', a' - \left\lfloor \frac{x}{y} \right\rfloor \cdot b'$

    return d, a'', b''

The runtime of the non-recursive work is $O(n^2)$ as we are computing $x \pmod y$ and are performing a recursion with size $n - 1$ giving a recurrence relation that is the same as the Euclidean algorithm:

$$T(n) = T(n-1) + O(n^2) = O(n^3)$$