Number theoretic algorithms

Lecture 3

Akshar Varma
6th July, 2023

CS3000 Algorithms and Data

Why Number Theory

Complexity of Basic Arithmetic Operations

(Modular) Arithmetic

Fast exponentiation (binary)

GCD

1. Why Number Theory

"If the theory of numbers could be employed for any practical and obviously honourable purpose, if it could be turned directly to the furtherance of human happiness or the relief of human suffering, as physiology and even chemistry can, then surely neither Gauss nor any other mathematician would have been so foolish as to decry or regret such applications. But science works for evil as well as for good; and both Gauss and lesser mathematicians may be justified in rejoicing that there is one science at any rate, and that their own, whose very remoteness from ordinary human activities should keep it gentle and clean."

– G. H. Hardy, A Mathematician's Apology

Why Number Theory

- Many fields of computer science make extensive use of number theoretic algorithms.
- Modular arithmetic is a hydra that will keep showing up:
  GCD, Diffie-Hellman, Primality testing, Coding theory (checksums, error-detection), cryptography, ...
- Euclid's GCD algorithm is the oldest surviving algorithm and is *the* tool of number theory.
- While this course cannot cover all of this, we will present the foundational concepts.
- Knowing this is essentially knowing half of the steps in all basic number theoretic algorithms.

2. Complexity of Basic Arithmetic Operations

Complexity of Basic Arithmetic Operations

- Addition: Depends on length of numbers.
- We might add 2 digit numbers in our head, but 100 digit numbers?
- Subtraction: Essentially the same as addition if we think of numbers as integers.
- Multiplication: Also depends on length of numbers.
- Grade school multiplication is quadratic.
- Every digit of one number with multiplied with every digit of the other.
- Division: Basically complexity of multiplication.
- Exponentiation: Repeated multiplication? We'll do better.

3. (Modular) Arithmetic

Euclidean division: $m = q \cdot n + r$

- For all $n \in \mathbb{Z}^+$ we can write any $m \in \mathbb{Z}$ uniquely as follows:

$$m = n \cdot q + r \quad q \in \mathbb{Z}, 0 \leq r < n$$

- Here q is the quotient and r the remainder when m is divided by n
- Euclidean division: Given m and n compute the values of q and r.
- Shows up almost everywhere when dealing with integers in mathematics/computer science.
- We will see it in modulo arithmetic and Euclid's GCD Algorithm.

Modular Arithmetic $(\mathrm{mod}\ n)$

- In modular arithmetic, we fix some value of n and then we ignore the quotient q.
- So $m_1 = q_1 n + r$ is the same as $m_2 = q_2 n + r$ in the $(\mathrm{mod}\ n)$ world.
- For example, in the $(\mathrm{mod}\ 12)$ world, 13 is the same as 1. Written as $13 \equiv 1\ (\mathrm{mod}\ 12)$.
- Hence 0100 and 1300 hrs both look the same on an analog clock: one o'clock.
- The above fact is the origin of the alternate term "clock arithmetic".

- Since $m_1 \bmod n = m_2 \bmod n = r$, we will always be using numbers in $\{0, \ldots, n-1\}$

Properties of Modular Arithmetic

- **Basically: Add, multiply, cancel, subtract, etc. as normal.**
  (Optionally, but usually) take remainder at all intermediate points in calculation.
- Equivalence relation:
  1. *Reflexivity:* $a \equiv a \pmod{n}$
  2. *Symmetry:* $a \equiv b \pmod{n}$ if $b \equiv a \pmod{n}$
  3. *Transitivity:* If $a \equiv b \pmod{n}$ and $b \equiv c \pmod{n}$, then $a \equiv c \pmod{n}$
- Let $a \equiv b \pmod{n}$:
  - For any integer k, we have $a + k \equiv b + k \pmod{n}$ and $ak \equiv bk \pmod{n}$
  - For any non-negative integer k, we have $a^k \equiv b^k \pmod{n}$
- Let $a_1 \equiv b_1 \pmod{n}$, $a_2 \equiv b_2 \pmod{n}$, then

  $a_1 + a_2 = b_1 + b_2 \pmod{n}$, $a_1 \cdot a_2 = b_1 \cdot b_2 \pmod{n}$, $a_1 a_2 = b_1 b_2 \pmod{n}$
- Cancellation (be a little careful when dividing):
  - For any integer k, if $a + k \equiv b + k \pmod{n}$ then $a \equiv b \pmod{n}$
  - If k and n are coprime (no common factors), and if $ak \equiv bk \pmod{n}$ then $a \equiv b \pmod{n}$
- For proving any of these, or for other basic properties always resort to Euclidean division.

1. Prove divisibility rules for the numbers: $\{2, 5, 10, 4, 8, 16, 2^n, 5^n, 10^n, 9, 3, 6, 11\}$
2. Consider the digit-root function defined for all non-negative integers $n \in \mathbb{Z}^+$ as:

$$dr(n) = \begin{cases} n & n < 10 \\ dr(\text{sum of digits of } n) & \text{otherwise} \end{cases}$$

   Show that $dr(n) = n \bmod 9$.
3. Suggested reading: “Casting out nines” on Wikipedia.

4. Fast exponentiation (binary)

“Yay binary numbers”

Quickly Computing $a^b \pmod n$

$$a^b = a \cdot a \cdot a \cdots a$$

$$a^b \pmod n = (a \cdot a \pmod n) \cdot a \pmod n \cdots a \pmod n$$

$$\text{Let: } b = \sum_{i=0}^{m} 2^i b_i \qquad \text{binary representation}$$

$$a^b = a^{\sum_{i=0}^{m} 2^i b_i}$$

$$a^b = \prod_{i=0}^{m} a^{2^i b_i}$$

If $b_i = 1$ for all i, then the terms are just repeated squarings. Binary Exponentiation: Keep squaring the base, accumulate to product if $\mathbf{1}\,[]\,b_i$

repeated squaring

input: a, b and optionally n
output: $a^b \pmod{n}$; (if n not provided, use $n = 1$)

```
1  current ← a
2  accumulator ← 1
3  while b > 0
4      if b%2 = 1                          if last bit is one
5          accumulator = accumulator · current   if bit is 1, accumulate
6      current = current · current         do squaring
7      b = b/2                             divide by 2 to reduce length by 1 bit
8  return accumulator
```

5. GCD

”[The Euclidean algorithm is] the granddaddy of all algorithms, because it is the oldest nontrivial algorithm that has survived to the present day.” – Donald Knuth

Greatest Common Divisor (GCD)

- The GCD of two integers a, b is the largest number d such that d divides both a and b.
- Note that the GCD is always positive since 1 always divides any two integers.
- If a, b have no common factors then the GCD is 1 since there are no larger factors.
- Whenever $GCD(a, b) = 1$, we call a and b to be co-prime (prime, relative to each other).
- The oldest known surviving algorithm is from Euclid's Elements for computing $GCD(a, b)$.

GCD Algorithm Key Insight via Invariance

- Let $GCD(a, b) = d$ and let $a = k_1 d, b = k_2 d$ with $k_1 \geq k_2 \geq 1$.
- By definition, $GCD(a, b) = d \implies GCD(k_1, k_2) = 1$ (coprime).
  Otherwise $GCD(a, b) = d \cdot GCD(k_1, k_2) > d$ which contradicts $GCD(a, b) = d$.
- Consider: $a' = (k_1 - k_2)d < a$.
- Insight! $GCD(a, b) = GCD(a', b) = GCD((k_1 - k_2)d, k_2 d)$.
  If $c > 1$, $k_1 - k_2 = x \cdot c$, $k_2 = y \cdot c$, then $k_1 = (x + y) \cdot c \implies GCD(k_1, k_2) = c > 1$.
  Contradiction!
- Subtraction keeps GCD invariant $\implies$ GCD will stay invariant after repeated subtractions.
- $GCD(a, b) = GCD((k_1 - q_1 k_2)d, k_2 d)$ where q_1 is largest number s.t. $(k_1 - q_1 k_2) \geq 0$.
- Note that by definition, $(k_1 - q_1 k_2)d = a \mod b$. So even remainder keeps GCD invariant!
- Euclid's algorithm crux: $GCD(a, b) = GCD(a \mod b, b)$.

Euclid's GCD Algorithm

Input: Integers a, b
Output: The greatest common divisor $GCD(a, d)$ of a and b

```
1 if a < b then                          if b is larger
2       a, b = b, a                      swap to make a larger
3 if b = 0 then                          if the smaller number is 0
4       return max(1, a)                 the larger number must be GCD (unless < 1)
5 return GCD(a mod b, b)                 recurse after invariance preserving modulus operation
```