CS F364
Design & Analysis of Algorithms

Problem Domain – number Theory

Modular Arithmetic:

- Groups Z_n, Z^*_n
- Size of Z^*_n
- Computing the size of Z_n^*

Congruence Arithmetic

- "congruence modulo n":
  - if $a \bmod n = b \bmod n$
    - then a and b are congruent modulo n
  - This is an equivalence relation.
    - Why?
  - This is often denoted as
    - $a \equiv b \pmod{n}$

Congruence Arithmetic - Z_N

- (Z_n = { 0, 1, … n-1 }, $+_n$) is a group
    - where $+_n$ refers to addition modulo n.
- **Exercise:** *Verify the following properties*:
    - Closure:
    - Associativity:
    - Existence of Identity :
        - 0
    - Existence of Inverse:
        - (a^{-1} is n-a)

Congruence Arithmetic: Z^*_N

- $(Z^*_n = \{ x \mid 1 <= x <= n \text{ and } \gcd(x,n) = 1\}, *_n)$ is a group.
  - **Exercise**:
    - *Verify Closure and Associativity*
  - Identity Element exists:
    - $(a * 1 = a)$
  - Inverse?

Congruence Arithmetic: Z^*_N [contd.]

- $(Z^*_n = \{ x \mid 1 <= x <= n \text{ and } \gcd(x,n) = 1\}, *_n)$ is a group.

  - Existence of Inverse:
    - Is there an **x** such that **a*x = 1 (mod n)**, for **a** in $\mathbf{Z^*_n}$?
      - i.e. Is there an **x** such that **a*x = 1 + b*n** for some **+ve** integer **b**?
    - *<u>The answer is yes</u>*, by extended Euclid's Theorem since **gcd(a,n)=1**
      - Furthermore, by <u>Aryabhatia's algorithm</u>:
        - *the inverse of any element in (Z^*_n, *n) can be computed in polynomial time i.e.*
          - *time that is polynomial in log(n)*

Congruence Arithmetic – Size of Z^*_N

- What is the size of $\mathbf{Z^*_n}$?
  - Let ϕ**(n),** known as ***Euler's phi function,*** denote the size of $\mathbf{Z^*_n}$
- Properties of ϕ**(n)**
  - $\phi(p) = p-1$ for *prime* **p**
    - Proof**:** for any m < p, **gcd(m,p) = 1** for *prime* p.
  - $\phi(p^m) = p^m - p^{m-1}$ for *prime* **p**
    - Proof:
      - Only multiples of **p** have common factors with $\mathbf{p^m}$
      - Multiples of p (less than p^m) are:
        - $\mathbf{p, 2*p, 3*p, \ldots, (p^{m-1} - 1) * p}$
      - So, $\phi(p^m) = (p^m - 1) - (p^{m-1} - 1)$ for prime **p**.

CONGRUENCE ARITHMETIC - PROPERTIES OF ϕ(N) [*CONTD*]

- **$\phi(p*q) = (p-1)(q-1)$** for primes **p** and **q**

Proof:

- Only multiples of **p** or **q** or both have common factors with **p*q**
  - i.e. **p, 2*p, …, q*p,** *and* **q, 2*q, …,p*q**
- And they are all distinct except for **p*q**
- So **$\phi(p*q) = (p*q) – p – q +1$**

- ϕ is multiplicative i.e.
  - **$\phi(m*n) = \phi(m)* \phi(n)$** *if* **gcd(m,n) = 1**
    - Proof: *Left as an exercise.*
    - **Note:** We only need **$\phi(p^{k1} * q^{k2}) = \phi(p^{k1}) * \phi(q^{k2})$** for primes **p** and **q**. **End of Note.**

Congruence Arithmetic – Computing ϕ(n)

- Value of **ϕ(n)**
  - If $\mathbf{n = p_1^{k1} * p_2^{k2} * \ldots * p_m^{km}}$ for primes p_i and +ve integers **ki** then $\boldsymbol{\phi(n) = \Pi_i (p_i^{ki} - p_i^{ki-1})}$
- Computing **ϕ(n)**
  - If the prime factors of n are known then **ϕ(n)** can be computed in polynomial time
    - But <u>*computing factors*</u> is known to be "***difficult***".
    - In particular, there is no known polynomial time algorithm to compute factors of a given integer.
  - Is there an alternative?
    - i.e can we compute **ϕ(n)** – efficiently – without computing factors of **n**?

Congruence Arithmetic – Computing ϕ(n) – Special Case

- Can we compute **ϕ(n)** – efficiently – without computing factors of **n**?
- Consider **n = p * q**
  - Given **n** and **ϕ(n)**, one can compute **p** and **q** in polynomial time !
    - How?
  - i.e. *Computing **ϕ(n)** is <u>at least as "difficult" as</u> factoring **n***.

ASIDE: REDUCTION AND LOWER BOUNDING

- We argued that one problem (say, <u>computing **ϕ(n)**</u>) is ***at least as difficult to solve as*** another problem (say, <u>factoring **n**</u>):
  - but we argued this without solving – i.e. <u>*without providing an algorithm for*</u> – either of these problems independently!!

This construction is referred to as **reduction** i.e. *we reduce π_2 to π_1*

- In the abstract, we argued that:
  - given an algorithm **f** for problem π_1,
  - if we can construct an algorithm **g • f** for problem π_2
  - such that **g** costs no more than **f**
    - then we can conclude that π_1 is ***at least as difficult as*** π_2

This is referred to as ***lower-bounding*** (<u>*the cost / complexity of a problem*</u>)