Advanced Algorithms

Floriano Zini

Free University of Bozen-Bolzano
Faculty of Computer Science

Academic Year 2013-2014

Lecture 3 – Algorithms with numbers (cont.)

Modular arithmetic

- For cryptography it is necessary to deal with numbers that are significantly **large** but whose range is **limited**
- Modular arithmetic deals with restricted ranges of integers

Modular exponentiation

- In the **cryptosystem** we are working toward, it is **necessary** to compute $x^y \bmod N$ for values of x, y, and N that are **several hundred bits long**
- The **result** is some number modulo N and is therefore itself a **few hundred bits long**
- The **raw value of x^y** could be **much longer!**

If x and y are 20-bits long, how long is x^y?

$x^y \geq (2^{19})^{(2^{19})} = 2^{19*524288} = 2^{9961472}$

about **10 million bits long**!

Modular exponentiation (cont.)

- We want that the numbers we are dealing with never grow too large
  - We perform all intermediate computations modulo *N*
- **First idea**
  - Calculate $x^y \bmod N$ by repeatedly multiplying by $x \bmod N$
  - The resulting sequence of intermediate products is

    $$x \bmod N \rightarrow x^2 \bmod N \rightarrow x^3 \bmod N \rightarrow \cdots \rightarrow x^y \bmod N$$

  - Each multiplication involves numbers < *N* and does not take to long
  - But what if *y* is 500-bits long?
    - We need to perform $y - 1 \cong 2^{500}$ multiplications!

Modular exponentiation (cont.)

Second idea

- Starting with x and squaring repeatedly modulo N, we get

  $x \bmod N \rightarrow x^2 \bmod N \rightarrow x^4 \bmod N \rightarrow \cdots \rightarrow x^{2*\log(y)} \bmod N$

- Each multiplication takes ***O(log² N)*** time to compute, and there are only ***log y*** multiplications
- To determine $x^y \bmod N$, we simply multiply together a subset of powers, those corresponding to *1*'s in the binary representation of y
  - For instance:
    - $x^{25}=x^{11001} = x^{10000}*x^{1000}*x^{1}=x^{16}*x^{8}*x^{1}$

Modular exponentiation (cont.)

- Another (equivalent) formulation

$$x^y = \begin{cases} \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is even} \\ x \cdot \left(x^{\lfloor y/2 \rfloor}\right)^2 & \text{if } y \text{ is odd.} \end{cases}$$

```
function modexp(x, y, N)
Input: Two n-bit integers x and N, an integer exponent y
Output: x^y mod N

if y = 0: return 1
z = modexp(x, ⌊y/2⌋, N)
if y is even:
   return z^2 mod N
else:
   return x · z^2 mod N
```

- **How long does it take?**
  - The algorithm halts after at most n recursive calls
  - During each call it multiplies (mod N) n-*bit* numbers
  - For a total **running time** of ***O(n³)***

Euclide's algorithm

- Given two integers *a* and *b*, how to find their **greatest common divisor** *gcd(a, b)*?
- **Euclid's rule**:

  if $x, y \in \mathbf{N}$, $x \geq y$, $x \neq 0$, then $gcd(x, y) = gcd(x \bmod y, y)$

- Proof:
  - It is enough to prove the simpler rule *gcd(x,y)=gcd(x-y,y)*
  - Any integer that divides both *x* and *y* must also divide *x-y*, so $gcd(x, y) \leq gcd(x-y, y)$
  - Any integer that divides both *x-y* and *y* must also divide both *x* and *y*, so $gcd(x, y) \geq gcd(x-y, y)$
  - Thus *gcd(x,y)=gcd(x-y,y)*

Euclid of Alexandria
BC 325–265

Euclide's algorithm (cont.)

```
function Euclid(a, b)
Input: Two integers a and b with a ≥ b ≥ 0
Output: gcd(a, b)

if b = 0: return a
return Euclid(b, a mod b)
```

a ≠ 0

How long does it take?

- After any two consecutive rounds of the algorithm, *a* and *b* are at least halved in value (see proof on DPV)
- So the length of *a* and *b* decreases by at least one bit in two rounds
- If *a* and *b* are *n*-bit integers, then the base case is reached within *2*n* recursive calls
- Since each call involves a quadratic-time division, the total time is $\boldsymbol{O(n^3)}$

Extended Euclide's algorithm

- Suppose someone claims that *d=gcd(a,b)*
  - how can we check this?
- It is not enough to verify that ***d* divides *a* and *b***
  - **this only shows *d* to be a common factor**, not necessarily the largest one
- **Lemma**
  - If *d* divides both *a* and *b*, and *d = ax + by* for some integers *x* and *y*, then necessarily *d = gcd(a, b)*

If *d* divides both *a* and *b*, *and d = ax + by* for some integers *x* and *y*, then necessarily *d = gcd(a, b)*

How can we prove it?

- *d* is a common divisor of *a* and *b* then $d \leq gcd(a, b)$
- *gcd(a, b)* is a common divisor of *a* and *b* so it also divides *ax + by = d*, which implies $d \geq gcd(a, b)$
- Thus, *d=gcd(a, b)*

Extended Euclide's algorithm (cont.)

```
function extended-Euclid(a, b)
Input: Two positive integers a and b with a ≥ b ≥ 0   a ≠ 0
Output: Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0: return (1, 0, a)
(x', y', d) = extended-Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

Is the algorithm correct?

- **Lemma**: For any positive integers *a* and *b*, the extended Euclid algorithm returns integers *x*, *y*, and *d* such that *gcd(a,b) = d = ax + by*
- Proof: See DPV

Extended Euclide's algorithm (cont.)

```
function extended-Euclid(a, b)
Input: Two positive integers a and b with a ≥ b ≥ 0   a ≠ 0
Output: Integers x, y, d such that d = gcd(a, b) and ax + by = d

if b = 0: return (1, 0, a)
(x', y', d) = extended-Euclid(b, a mod b)
return (y', x' − ⌊a/b⌋y', d)
```

Example

e-E(25,11) → e-E(11,3) → e-E(3,2) → e-E(2,1) → e-E(1,0)

(1,0,1)

(1,0-1*1,1)=(1,-1,1)

(0,1-2*0,1)=(0,1,1)

(-1,1-3*(-1),1)=(-1,4,1)

(4,-1-2*4,1) = **(4,-9,1)**

x = 4, y = -9, d = 1

25*4 – 11*9 = 100 – 99 = 1

Modular inverse

- We say x is the **multiplicative inverse** of a modulo N if

$$a*x \equiv 1 \bmod N$$

- There can be **at most one such *x* modulo *N***, denoted by a^{-1}
- **Remark**
  - The inverse does not always exist!

If *a* and *N* are both even, then *a* is not invertible modulo *N*

How can we prove it?

a and *N* are both even →

a mod N = *a* – *k***N* (*k* ∈ **Z**) is even →

*a***x mod N* is even for all *x* →

*a***x* ≠ *1 mod N* →

a is not invertible *mod N*

Modular division

- **Modular division theorem**
  - For any *a mod N*, *a* has a multiplicative inverse modulo *N* if and only if *a* is relatively prime to *N*
    - i.e., *gcd(a, N)=1*
  - When the inverse exists, it can be found in time ***O(n³)*** by **running the extended Euclid algorithm**
- **Example**
  - Let's compute the inverse of *11 mod 25*
  - We run the extended Euclid algorithm (see before) and find that *25*4 – 11*9 = 1 = gcd(11,25)*
  - *25*4 – 11*9 = 1* → *-11* 9 ≡ 1 mod 25* →

    -9 ≡ 16 mod 25 = 11^{-1} mod 25
- **When working modulo *N*, we can divide only by numbers relatively prime to *N***
  - To carry out the division, **we multiply by the inverse**
  - **Running time** of division is ***O(n³)***

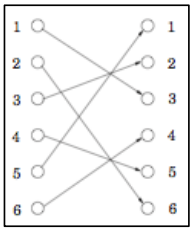
Primality testing

- Telling whether a reasonably large number is a prime is tedious
  - there are **too many candidate factors to try**
- However, there are **tricks**:
  - omit even-valued candidates after you have eliminated the number *2*
  - omit all candidates except those that are themselves primes
  - you can proclaim *N* a prime as soon as you have rejected all factors up to *√N*
    - If *N = x*y* then necessarily *x ≤ √N* or *y ≤ √N*
- Is there an efficient primality test down this road? **Unfortunately not!**
  - We have been trying to tell if a number is a prime by **factoring** it
  - And factoring is a **hard** problem!
- Modern cryptography is about the following important idea:

  factoring is hard and primality is easy
- We cannot factor large numbers, but we can easily test huge numbers for primality!
  - If a number is **composite**, such a **test** will detect this **without finding a factor**

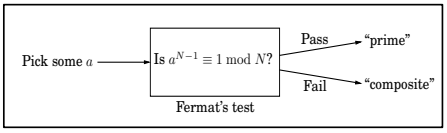
Fermat's little theorem

- Easy primality testing is based on **Fermat's little theorem**
  - If p is prime, then for every $1 \le a < p$, $a^{p-1} \equiv 1 \bmod p$
- **Example**: $a=3$, $p=7$
  - Let's prove that $3^6 \equiv 1 \bmod 7$
  - Let $S = \{1, 2, \ldots, 6\}$ be the set of nonzero integers modulo 7
  - The effect of multiplying the numbers in S by a (*mod p*) is simply to **permute** them
    - $\{1, 2, \ldots, 6\} = \{3*1 \bmod 7, 3*2 \bmod 7, \ldots, 3*6 \bmod 7\}$
  - Let's multiply the numbers on each side
    - $6! = 3^6 * 6! \pmod 7$
  - Let's divide by 6! → $\mathbf{3^6 \equiv 1 \bmod 7}$
- See DPV for the full proof



A first algorithm to test primality

- A "factorless" test for determining if a number N is prime
- **Problem**: the theorem is **not an if-and-only-if condition**
  - e.g., $341 = 11*31$, and $2^{340} \equiv 1 \bmod 341$
- However, for a **composite** N, **most values** of a **fail** the test
- Rather than fixing a in advance, we **choose** it **randomly** from $\{1, \ldots, N-1\}$




```
function primality(N)
Input:  Positive integer N
Output:  yes/no

Pick a positive integer a < N at random
if a^(N-1) ≡ 1 (mod N):
   return yes
else:
   return no
```

Analysis of the algorithm

```
function primality(N)
Input:  Positive integer N
Output:  yes/no

Pick a positive integer a < N at random
if a^(N-1) ≡ 1 (mod N):
   return yes
else:
   return no
```

- **Carmichael numbers** are **composite** but **pass** the test **for all** *a*<*N*. E.g.:
  - ***561*** is the smallest Carmichael number
  - It is **not** a **prime**: *561 = 3 * 11 * 17*
  - it **fools** the Fermat **test**:

    $a^{560} \equiv 1 \pmod{561}$ for all values of *a* relatively prime to *561*
- Carmichael numbers are **infinite** but extremely **rare**
  - They **can be managed**, using a more refined primality test (see DPV)
- In a **Carmichael-free universe**, the algorithm works well
  - If *N* is **prime**, then $a^{N-1} \equiv 1 \bmod N$ for all *a*<*N*
  - If *N* is **not prime**, then $a^{N-1} \equiv 1 \bmod N$ for **at most half the values** of a<*N* (see DPV for proof)
- **Probabilistic behavior**
  - *Pr(primality returns yes when N is prime) = 1*
  - *Pr(primality returns yes when N is not prime) ≤ 1/2*

A second (improved) algorithm

- We can **reduce the error** by repeating the procedure many times

```
function primality2(N)
Input:  Positive integer N
Output:  yes/no

Pick positive integers a_1, a_2, ..., a_k < N at random
if a_i^(N-1) ≡ 1 (mod N) for all i = 1, 2, ..., k:
   return yes
else:
   return no
```

- **Probabilistic behavior**
  - *Pr(primality returns yes when N is prime) = 1*
  - *Pr(primality2 returns yes when N is not prime) ≤ $1/2^k$*
    - The **error drops exponentially fast**, and can be driven arbitrarily low by choosing *k* large enough
    - Testing *k = 100* values of *a* makes the probability of failure at most 2^{-100}, which is **miniscule**

Generating random numbers

- The final step towards cryptography a **fast algorithm for choosing random primes** that are a few hundred bits long
- This task is **easy** as **primes** are **abundant**
  - a random *n*-bit number has roughly a *one-in-n* chance of being prime (actually about $1/(\ln 2^n) \approx 1.44/n$)
- **Lagrange's prime number theorem**
  - Let $\pi(x)$ be the number of primes $\leq x$
  - Then $\pi(x) \approx x/(\ln x)$, or more precisely

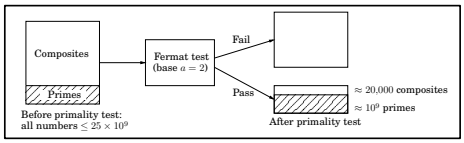
$$\lim_{x\to\infty} \frac{\pi(x)}{(x/\ln x)} = 1.$$

Generating random numbers (cont.)

- **Procedure** to generate a random *n-bit* prime
  1. Pick a random *n-bit* number *N*
  2. Run a primality test on *N*
  3. If *N* passes the test, output *N*; else repeat the process
- **How fast is this algorithm**?
  - If the randomly chosen *N* is truly prime, which happens with probability at least *1/n*, then it will certainly pass the test
  - On each iteration, the algorithm has at least a *1/n* chance of halting
  - On **average** it will **halt within *O(n)* rounds**

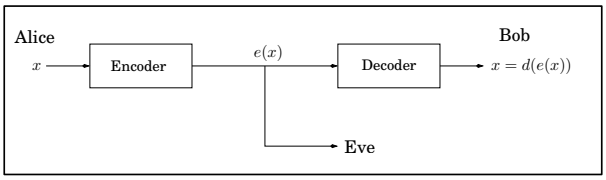
Generating random numbers (cont.)

- **Which primality test?**
  - the numbers we are testing for primality are chosen at random
  - **it is sufficient to perform the Fermat test with base a = 2 (or to be really safe, a = 2,3,5)**
    - for random numbers the Fermat test has a much smaller failure probability than the worst-case 1/2 bound
- **What is the probability that the output is really prime?**
  - Suppose we perform the test with base $a = 2$ for all numbers $N \leq 25*10^9$
  - In this range, there are about 10^9 primes, and about $20{,}000$ composites that pass the test



  - The chance of erroneously outputting a composite is approximately $20{,}000/10^9 = 2*10^{-5}$
  - This chance of error decreases rapidly if we repeat the Fermat test many times with various values for a

Cryptography scenario (again)



- **Alice** and **Bob** wish to **communicate in private**
- **Eve** eager for **finding** out what **Alice** and **Bob** are **saying**
- **Alice** wants to send a specific **message** x, written in **binary**, to **Bob**
  - **Alice encodes** x as $e(x)$, sends it over
  - **Bob** applies his **decryption** function $d(\cdot)$ to decode it: $d(e(x)) = x$
- **Eve intercept** $e(x)$: for instance, she might be a sniffer
- The encryption function $e(\cdot)$ is so that without knowing $d(\cdot)$, Eve cannot do anything with the information she has picked up
  - **knowing *e(x)* tells Eve little or nothing about what *x* might be**

Private-key cryptography

For centuries cryptography was based on what we now call **private-key** *protocols*

- **Alice** and **Bob** meet beforehand and **together choose** a **secret codebook**, with which they encrypt all future correspondence between them

Decrypting watch from the late seventieth ☺

- **Eve's** only **hope** is to **collect** some **encoded messages** and use them to at least partially **figure out the codebook**

Private-key schemes: one-time pad

- **Alice** and **Bob** secretly choose a **binary string** *r* of the same length (***n* bits**) as the message *x* that **Alice** will later send
- **Alice**'s **encryption** function is a **bitwise XOR**

$$e_r(x) = x \oplus r$$

- This function e_r is a **bijection** from *n*-bit strings to *n*-bit strings, as it is its own inverse

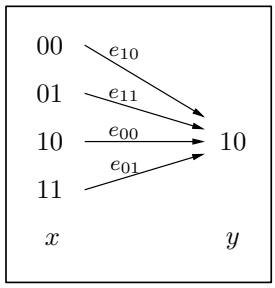
$$e_r(e_r(x)) = (x \oplus r) \oplus r = x \oplus (r \oplus r) = x \oplus 0 = x$$

- **Bob** chooses the **decryption** function

$$d_r(y) = y \oplus r$$

Private-key schemes: one-time pad (cont.)

- How should Alice and Bob choose *r* for this scheme to be secure?
- They should pick *r* at **random**, flipping a (fair) coin for each bit, so that the resulting string is equally likely to be any element of $\{0,1\}^n$
- This ensures that if Eve intercepts the encoded message $y = e_r(x)$, she gets no information about *x*
  - all *r*'s are equally possible, thus all possibilities for *x* are equally likely!



Private-key schemes: one-time pad (cont.)

- **Downside**: the **pad** has to be **discarded after use**, as a **second message** encoded with the same pad would **not** be **secure**
- If **Eve knew $x \oplus r$ and $z \oplus r$** for two messages *x* and *z*, then **she could take the exclusive-or to get $x \oplus z$**, which might be important information
- E.g.:
  1. it **reveals** whether the two **messages begin** or **end the same**
  2. if **one message** contains a **long sequence of zeros** (it is common if the message is an image), then **the corresponding part of the other message will be exposed**

Suppose that Alice uses the same private key *r* to encrypt two messages *x* and *z*, and that Eve intercepts the two encryptions $x \oplus r$ and $z \oplus r$. Then suppose that Eve performs the exclusive or of the two encryptions

Why does she get $x \oplus z$?

$(x \oplus r) \oplus (z \oplus r) = x \oplus r \oplus r \oplus z = x \oplus 0 \oplus z = x \oplus z$

Why $x \oplus z$ can reveal if x and z begin or end the same?

$r = 01101$ $x = 11000$ $z = 11011$

$x \oplus r = 11000 \oplus 01101 = 10101$ $z \oplus r = 11011 \oplus 01101 = 10110$

$(x \oplus r) \oplus (z \oplus r) = x \oplus z = 10101 \oplus 10110 = 00011$

Why if *x* contains a long sequence of zeros, then the corresponding part of *z* is exposed?

$r = 01101$ $x = 10001$ $z = 01011$

$x \oplus r = 10001 \oplus 01101 = 11100$ $z \oplus r = 01011 \oplus 01101 = 00110$

$(x \oplus r) \oplus (z \oplus r) = x \oplus z = 11100 \oplus 00110 = 11010$

Public-key cryptography

Public-key schemes allow **Alice** to send **Bob** a **message without ever having met him before**

- **Why should Bob have any advantage over Eve?**
  - **Bob** is able to implement a **digital lock**, to which only he has the key
  - By making this digital lock **public**, **Bob** gives **Alice** a way to send him a **message that only he can open**
  - This happens, for **example**, when you **send your credit card number to some company over the Internet**
- In the **RSA protocol**
  - **Bob** performs **simple calculations** to implement his **digital lock**
  - **Alice** and **Bob** perform **simple calculations** to **lock** and **unlock** the message
  - To **unlock** the message **without** the **key**, Eve must perform operations like **factoring large numbers**, which are **intractable** for numbers that are just few hundred bits long

RSA cryptosystem

- Rivest-Shamir-Adelman (RSA) cryptosystem derives very **strong** guarantees of **security** by exploiting
  - The **polynomial-time** computability of certain number-theoretic tasks (**modular exponentiation**, **greatest common divisor**, **primality testing**)
  - The **intractability** of others (**factoring**)
- **Deriving strong *e(·)* and *d(·)* is efficient** (can be done in **polynomial** time)
- **Decrypting *e(x)* without *d(·)* is hard** (need **exponential** time)

RSA cryptosystem (cont.)

Let's summarize RSA once again

- **Anybody** can **send** a **message** to **anybody else** using publicly available information
- Each person has a **public key** known to the whole world and a **secret key** known only to him- or herself
- When **Alice** wants to send message *x* to **Bob**, she **encrypt** it using his **public key**
- Bob **decrypts** it using his **secret key**, to retrieve x
- **Eve** is welcome to see as many encrypted messages for **Bob** as she likes but she **will not be able to decode** them, under certain **simple assumptions**

RSA cryptosystem (cont.)

- The **RSA** scheme is **based** heavily upon **number theory**
- Think of **messages** from **Alice** to **Bob** as **numbers modulo *N***
  - messages larger than *N* can be broken into smaller pieces
- The **encryption** function will then be a **bijection on *{0, 1, . . . , N − 1}***, and the **decryption** function will be its **inverse**
- **What values of *N* are appropriate, and what bijection should be used?**

RSA cryptosystem (cont.)

- **Property:** Pick any two primes p and q and let $N = p * q$
  For any e relatively prime to $(p - 1)*(q - 1)$:
  1. The **mapping $x \rightarrow x^e$** *mod N* is a **bijection on *{0, 1 , …, N−1}***
  2. The **inverse mapping** is easily realized: let $d = e^{-1}$ mod $(p-1)*(q-1)$.
     Then for all $x \in \{0, 1, \ldots, N-1\}$
     $$(x^e)^d \equiv x \pmod N$$
- The mapping $x \rightarrow x^e \bmod N$ is reasonable, no information is lost
- If **Bob** publishes ***(N,e)*** as his **public key**
  - Everyone else can use it to send him encrypted messages
- **Bob** should retain ***d*** as his **secret key**
  - He can decode messages by raising them to the dth power modulo *N*

RSA cryptosystem (cont.)

Example

- Let $N = 55 = 5 * 11$ ($p=5$, $q=11$ are prime)
- Choose $e=3$, which is relatively prime to $(p-1)*(q-1)$

  $gcd(e, (p-1)*(q-1)) = gcd(3, 40) = 1$
- The decryption exponent is then $d=3^{-1} \bmod 40 = 27$
- For any message $x \bmod 55$
  - the encryption of x is $y = x^3 \bmod 55$
  - the decryption of y is $x = y^{27} \bmod 55$
- For example, if $x=13$
  - $y = 13^3 = 52 \bmod 55$
  - $x = 13 = 52^{27} \bmod 55$

RSA cryptosystem (cont.)

Proof of the property

- If the mapping $x \rightarrow x^e \bmod N$ is invertible, it must be a bijection: **statement 2 implies statement 1**
- To prove statement 2
  - e is invertible modulo $(p-1)*(q-1)$ because it is relatively prime to this number
  - $(x^e)^d \equiv x \bmod N$ can be proved using Fermat's little theorem (see DPV)

RSA cryptosystem (cont.)

RSA protocol

- **Bob chooses his public and secret keys** $O(n)$
  - **He picks two large (n-bit) random primes *p* and *q***
  - His **public key** is ***(N, e)*** where ***N=p*q*** and ***e*** is a 2n-bit number relatively prime to $(p-1)*(q-1)$ $O(n^2)$
    - A common choice is $e=3$ because it permits fast encoding
  - His **secret key** is ***d***, the inverse of e mod $(p-1)*(q-1)$, computed using the extended Euclid algorithm $O(n^3)$
- **Alice** wishes to send message x to **Bob** $O(n^3)$
  - She looks up his public key (N, e) and sends him $y=x^e \bmod N$ $O(n^3)$
  - He decodes the message by computing $y^d \bmod N$

RSA cryptosystem (cont.)

Security assumption for RSA

- **Given *N, e*, and $y = x^e \bmod N$, it is computationally intractable to determine *x***
- How might Eve try to guess x?
  - She could experiment with all possible values of x, each time checking whether $x^e \equiv y \bmod N$, but this would take exponential time
  - Or she could try to factor N to retrieve p and q, and then figure out d by calculating $e^{-1} \bmod (p-1)*(q-1)$, but factoring would also take exponential time

The RSA calculator and express encryption/decryption

- https://www.cs.drexel.edu/~introcs/Fa12/notes/10.1_Cryptography/RSAWorksheetv4d.html
- https://www.cs.drexel.edu/~introcs/Fa12/notes/10.1_Cryptography/RSA_Express_EncryptDecrypt.html

Real-world applications of cryptography

- **Anonymity networks**
  - **E.g., TOR (The Onion Router)**
    - http://en.wikipedia.org/wiki/Tor_(anonymity_network)
- **Financial cryptography**
  - **E.g., Bitcoin**
    - http://en.wikipedia.org/wiki/Bitcoin
- **E-voting**
  - http://en.wikipedia.org/wiki/Electronic_voting

- **Authentication protocols**
  - **E.g., Kerberos**
    - http://en.wikipedia.org/wiki/Kerberos_(protocol)

TAKE AWAY

- The RSA cryptosystem, derives very **strong** guarantees of **security** by exploiting
  - The **polynomial-time** computability of certain number-theoretic tasks (**modular exponentiation**, **greatest common divisor**, **primality testing**)
  - The **intractability** of others (**factoring**)
- We have developed and analyzed **algorithms for a variety of computational tasks involving numbers**
- These algorithms have been used to **build the RSA cryptosystem step by step**