Implementing RSA Encryption in Java

RSA algorithm

- Select two large prime numbers p, q
- Compute
  n = p × q
  v = (p-1) × (q-1)
- Select small odd integer k relatively prime to v
  gcd(k, v) = 1
- Compute d such that
  (d × k)%v = (k × d)%v = 1
- Public key is (k, n)
- Private key is (d, n)

- example
  p = 11
  q = 29
  n = 319
  v = 280
  k = 3
  d = 187
- public key
  (3, 319)
- private key
  (187, 319)

Encryption and decryption

- Alice and Bob would like to communicate in private
- Alice uses RSA algorithm to generate her public and private keys
  - Alice makes key (k, n) publicly available to Bob and anyone else wanting to send her private messages
- Bob uses Alice's public key (k, n) to encrypt message M:
  - compute E(M) =(M^k)%n
  - Bob sends encrypted message E(M) to Alice
- Alice receives E(M) and uses private key (d, n) to decrypt it:
  - compute D(M) = ($E(M)^d$)%n
  - decrypted message D(M) is original message M

Outline of implementation

- RSA algorithm for key generation
  - select two prime numbers p, q
  - compute n = p × q
    v = (p-1) × (q-1)
  - select small odd integer k such that
    gcd(k, v) = 1
  - compute d such that
    (d × k)%v = 1
- RSA algorithm for encryption/decryption
  - encryption: compute E(M) = (M^k)%n
  - decryption: compute D(M) = ($E(M)^d$)%n

RSA algorithm for key generation

- Input: none
- Computation:
  - select two prime integers p, q
  - compute integers n = p × q
    v = (p-1) × (q-1)
  - select small odd integer k such that gcd(k, v) = 1
  - compute integer d such that (d × k)%v = 1
- Output: n, k, and d

RSA algorithm for encryption

- Input: integers k, n, M
  - M is integer representation of plaintext message
- Computation:
  - let C be integer representation of ciphertext
    C = (M^k)%n
- Output: integer C
  - ciphertext or encrypted message

RSA algorithm for decryption

- Input: integers d, n, C
  - C is integer representation of ciphertext message
- Computation:
  - let D be integer representation of decrypted ciphertext

    $D = (C^d)\%n$
- Output: integer D
  - decrypted message

This seems hard …

- How to find big primes?
- How to find mod inverse?
- How to compute greatest common divisor?
- How to translate text input to numeric values?
- Most importantly: RSA manipulates **big** numbers
  - Java integers are of limited size
  - how can we handle this?
- Two key items make the implementation easier
  - understanding the math
  - Java's **`BigInteger`** class

What is a `BigInteger`?

- Java class to represent and perform operations on integers of arbitrary precision
- Provides analogues to Java's primitive integer operations, e.g.
  - addition and subtraction
  - multiplication and division
- Along with operations for
  - modular arithmetic
  - gcd calculation
  - generation of primes
- **`http://java.sun.com/j2se/1.4.2/docs/api/`**

Using `BigInteger`

- If we understand what mathematical computations are involved in the RSA algorithm, we can use Java's **`BigInteger`** methods to perform them
- To declare a **`BigInteger`** named B

```
BigInteger B;
```

- Predefined constants

```
BigInteger.ZERO
BigInteger.ONE
```

Randomly generated primes

```
BigInteger probablePrime(int b, Random rng)
```

- Returns random positive **`BigInteger`** of bit length **`b`** that is "probably" prime
  - probability that **`BigInteger`** is not prime < 2^{-100}
- **`Random`** is Java's class for random number generation
- The following statement

  ```
  Random rng = new Random();
  ```

  creates a new random number generator named **`rng`**

`probablePrime`

- Example: randomly generate two **`BigInteger`** primes named **`p`** and **`q`** of bit length **`32`** :

```
/* create a random number generator */
Random rng = new Random();

/* declare p and q as type BigInteger */
BigInteger p, q;

/* assign values to p and q as required */
p = BigInteger.probablePrime(32, rng);
q = BigInteger.probablePrime(32, rng);
```

Integer operations

- Suppose have declared and assigned values for **p** and **q** and now want to perform integer operations on them
  - use methods **add**, **subtract**, **multiply**, **divide**
  - result of **BigInteger** operations is a **BigInteger**
- Examples:

```
BigInteger w = p.add(q);
BigInteger x = p.subtract(q);
BigInteger y = p.multiply(q);
BigInteger z = p.divide(q);
```

Greatest common divisor

- The greatest common divisor of two numbers x and y is the largest number that divides both x and y
  - this is usually written as gcd(x,y)
- Example: gcd(20,30) = 10
  - 20 is divided by 1,2,4,5,10,20
  - 30 is divided by 1,2,3,5,6,10,15,30
- Example: gcd(13,15) = 1
  - 13 is divided by 1,13
  - 15 is divided by 1,3,5,15
- When the gcd of two numbers is one, these numbers are said to be relatively prime

Euler's Phi Function

- For a positive integer n, $\phi(n)$ is the number of positive integers less than n and relatively prime to n
- Examples:
  - $\phi(3) = 2$ 1,2
  - $\phi(4) = 2$ 1,2,3 (but 2 is not relatively prime to 4)
  - $\phi(5) = 4$ 1,2,3,4
- For any prime number p,

$$\phi(p) = p-1$$

- For any integer n that is the product of two distinct primes p and q,

$$\phi(n) = \phi(p)\phi(q) = (p-1)(q-1)$$

Relative primes

- Suppose we have an integer x and want to find an odd integer z such that
  - $1 < z < x$, and
  - z is relatively prime to x
- We know that **x** and **z** are relatively prime if their greatest common divisor is one
  - randomly generate prime values for z until gcd(x,z)=1
  - if x is a product of distinct primes, there is a value of z satisfying this equality

Relative BigInteger primes

- Suppose we have declared a **BigInteger x** and assigned it a value
- Declare a **BigInteger z**
- Assign a prime value to **z** using the **probablePrime** method
  - specifying an input bit length smaller than that of **x** gives a value **z<x**
- The expression

```
(x.gcd(z)).equals(BigInteger.ONE)
```

  returns true if gcd(x,z)=1 and false otherwise
- While the above expression evaluates to false, assign a new random to **z**

Multiplicative identities and inverses

- The multiplicative identity is the element e such that

$$e * x = x * e = x$$

  for all elements $x \in X$
- The multiplicative inverse of x is the element x^{-1} such that

$$x * x^{-1} = x^{-1} * x = 1$$

- The multiplicative inverse of x mod n is the element x^{-1} such that

$$(x * x^{-1}) \bmod n = (x^{-1} * x) \bmod n = 1$$

  - x and x^{-1} are inverses only in multiplication mod n

`modInverse`

- Suppose we have declared **`BigInteger`** variables **`x`**, **`y`** and assigned values to them
- We want to find a **`BigInteger z`** such that

  `(x*z)%y =(z*x)%y = 1`

  that is, we want to find the inverse of **`x`** mod **`y`** and assign its value to **`z`**

- This is accomplished by the following statement:

```
BigInteger z = x.modInverse(y);
```

Implementing RSA key generation

- We know have everything we need to implement the RSA key generation algorithm in Java, so let's get started …