**Chinese Reminder Theorem** is used to solve set of congruent equations with one variable but different modulus, which are relatively prime

x ≡ a1 mod m1

x ≡ a2 mod m2

x ≡ a3 mod m3

......

x ≡ ak mod mk

The Chinese Reminder Theorem states that the above equations have unique solution if

the moduli are relatively prime. Below are the steps needed to follow to solve set of

congruent equations using Chinese Reminder Theorem

Step I: Find M = m1 x m2 x m3...mk where M is common modulus

Step II: Find M1 = M/m1, M2 = M/m2 and so on

Step III: Find multiplicative inverses for M1, M2 and so on

Step IV: Put the values in the below equation to solve for X

X = (a1 x M1 x M1-1 + a2 x M2 x M2-1 + a3 x M3 x M3-1) mod M

Let’s take an example,

x ≡ 2(mod 3)

x ≡ 3(mod 5)

x ≡ 2(mod 7)

**Note:** x ≡ a(mod b) means that when we divide x by b then the remainder is a.

From the above example,

m1 = 3, m2 = 5, m3 = 7 ; a1 = 2, a2 = 3, a3 = 2

Step 1: Calculate the value of **m** where **m = m1\*m2\*m3**.

m = 3\*5\*7 = 105

Step 2: Find **M1**, **M2**, **M3** where **M1 = m/m1**, **M2 = m/m2**, **M3 = m/m3**.

M1 = 105/3 = 35

M2 = 105/5 = 21

M3 = 105/7 = 15

Step 3: Find **y1**, **y2**, **y3** which are the modular multiplicative inverses of **M1**, **M2**, **M3** respectively using the congruence relations give below.

**M1\*y1 ≡ 1(mod m1)**

**M2\*y2 ≡ 1(mod m2)**

**M3\*y3 ≡ 1(mod m3)**

We can find the inverse either by using inspection or the Bézout theorem. Since m1, m2 and m3 in the given example are small values we can go with the inspection method.

M1\*y1 ≡ 1(mod m1)  
35\*y1 ≡ 1(mod 3)  
Find a value for y1 such that 0≤y1<3 (m1 = 3) and satisfies the above congruence relation.  
2 seems to satisfy the congruence.  
35\*2 ≡ 1(mod 3) **⇒** 70 mod 3 = 1  
∴ y1 = 2

Similarly,

M2\*y2 ≡ 1(mod m2)  
21\*y2 ≡ 1(mod 5)  
Find a value for y2 such that 0≤y2<5 (m2 = 5). 1 satisfies the above congruence.  
∴ y2 = 1

M3\*y2 ≡ 1(mod m3)  
15\*y3 ≡ 1(mod 7)  
Find a value for y3 such that 0≤y3<7 (m3 = 7). 1 satisfies the above congruence.  
∴ y3 = 1

Step 4: Calculate **x = (a1\*M1\*y1 + a2\*M2\*y2 + a3\*M3\*y3) mod m**

x = (2\*35\*2 + 3\*21\*1 + 2\*15\*1) mod 105 = 233 mod 105 = 23

∴ 23 is the smallest positive integer that satisfies x ≡ 2(mod 3), x ≡ 3(mod 5) and

x ≡ 2(mod 7) simultaneously.

**ALGORITHM:**

We are given two arrays num[0..k-1] and rem[0..k-1]. In num[0..k-1], every pair is coprime (gcd for every pair is 1). We need to find minimum positive number x such that:

x % num[0] = rem[0],

x % num[1] = rem[1],

.......................

x % num[k-1] = rem[k-1]

Basically, we are given k numbers which are pairwise coprime, and given remainders of these numbers when an unknown number x is divided by them. We need to find the minimum possible value of x that produces given remainders.

**Examples :**

Input: num[] = {5, 7}, rem[] = {1, 3}

Output: 31

Explanation:

31 is the smallest number such that:

(1) When we divide it by 5, we get remainder 1.

(2) When we divide it by 7, we get remainder 3.

Input: num[] = {3, 4, 5}, rem[] = {2, 3, 1}

Output: 11

Explanation:

11 is the smallest number such that:

(1) When we divide it by 3, we get remainder 2.

(2) When we divide it by 4, we get remainder 3.

(3) When we divide it by 5, we get remainder 1.

The **extended Euclidean algorithm** is an extension to the Euclidean algorithm. Besides

finding the greatest common divisor of integers a and b, as the Euclidean algorithm

does, it also finds integers x and y (one of which is typically negative).

ax + by = gcd (a, b) or sa + tb = gcd (a, b)

The extended Euclidean algorithm is particularly useful when a and b are coprime, since

x is the multiplicative inverse of a modulo b, and y is the multiplicative inverse of b

modulo a.

**ALGORITHM:**

INPUT: Two non-negative integers a and b with a ≥ b.

OUTPUT: d = gcd(a, b) and integers x and y satifying ax + by = d.

If b = 0 then set d = a, x = 1, y = 0, and return(d, x, y).

Set x2 = 1, x1 = 0, y2 = 0, y1 = 1

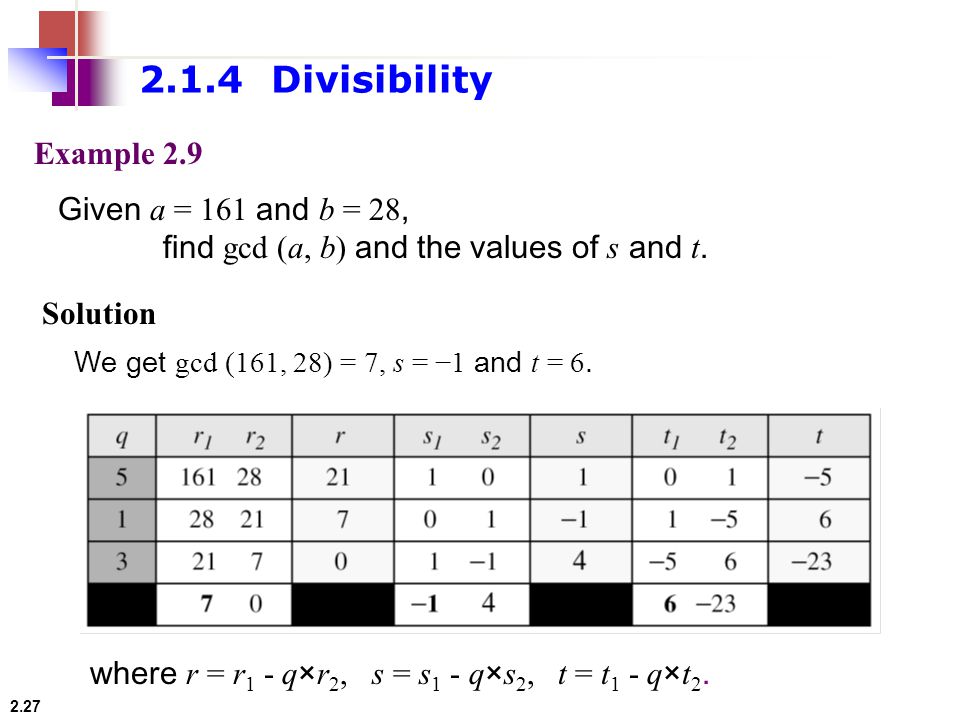
While b > 0, do

q = floor(a/b), r = a - qb, x = x2 - qx1, y = y2 - q y1.

a = b, b = r, x2 = x1, x1 = x, y2 = y1, y1 = y.

Set d = a, x = x2, y = y2, and return(d, x, y).

**Example**



Here the GCD value we are getting is 7. Value of s is -1 and the value of t is 6. If we put

these values into equation,

ax + by = gcd (a, b)

161 \* (-1) + 28 \* (6) = 7

This satisfies the equation for Extended Euclidean Algorithm

**The RSA Algorithm:**

The scheme developed by Rivest, Shamir and Adleman makes use of an expression

with exponentials. Plaintext is encrypted in blocks, with each block having a binary

value less than some number n. That is the block size must be less than or equal to log2

(n); in practice the block size is I bits, where 2i<n<=2i+1. Encryption and decryption are

of the following form, for some plaintext block M and ciphertext block C:

C = M^e mod n

M = C^d mod n

Both sender and receiver must know the value of n. The sender knows the value of e,

and only the receiver knows the value of d. Thus, this is a public-key encryption

algorithm with a public key of PU = {e, n} and a private key of PR = {d, n}. For this

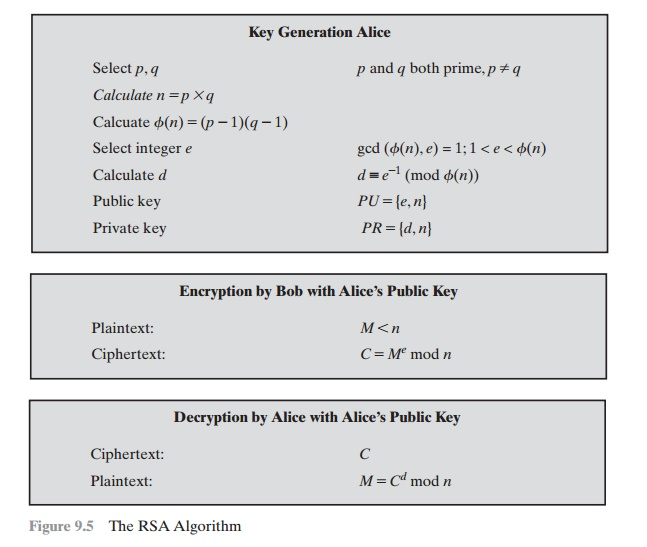
algorithm to be satisfactory for public key encryption, the following requirements must

meet:

1. It is possible to find values of e, d, n.

2. It is relatively easy to calculate Me mod n and Cd mod n for all values of M<n.

3. It is feasible to determine d given e and n.



Example 1:

1. Select two prime numbers, p = 17 and q = 11.

2. Calculate n = pq = 17\*11 = 187.

3. Calculate Ø(n) = (p-1)(q-1) = 16\*10 = 160.

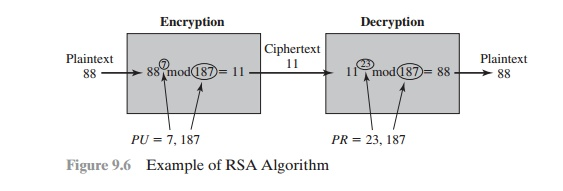
4. Select e such that relatively prime to Ø(n)=160 & less than Ø(n); we choose e = 7.

5. Determine d such that de ≡ 1 (mod 160) and d < 160. The correct value is d = 23;

d can be calculated using the extended Euclid’s algorithm.

The resulting keys are public key PU = {7, 187} and private key PR = {23, 187}. The

example shows the use of these keys for plaintext input of M=88.



**ALGORITHM:**

1. Start

2. Input two prime numbers p and q.

3. Calculate n = pq.

4. Calculate Ø(n) = (p-1)(q-1).

5. Input value of e.

6. Determine d.

7. Determine PU and PR.

8. Take input plaintext.

9. Encrypt the plaintext and show the output.

10. Stop.

**Diffie-Hellman key exchange (D-H)** is a specific method of exchanging keys. It is one of the earliest practical examples of Key exchange implemented within the field of cryptography. The Diffie Hellman key exchange method allows two parties that have no prior knowledge of each other to jointly establish a shared secret key over an insecure communications channel. This key can then be used to encrypt subsequent communications using a symmetric key cipher.

Diffie-Hellman establishes a shared secret that can be used for secret communications by exchanging data over a public network.

**Diffie-Hellman algorithm**

The Diffie-Hellman algorithm is being used to establish a shared secret that can be used for secret communications while exchanging data over a public network using the elliptic curve to generate points and get the secret key using the parameters.

For the sake of simplicity and practical implementation of the algorithm, we will consider only 4 variables, one prime P and G (a primitive root of P) and two private values a and b.

P and G are both publicly available numbers. Users (say Alice and Bob) pick private values a and b and they generate a key and exchange it publicly. The opposite person receives the key and that generates a secret key, after which they have the same secret key to encrypt.

**ALGORITHM:**

**Step by Step Explanation**

| Alice | Bob |
| --- | --- |
| Public Keys available = P, G | Public Keys available = P, G |
| Private Key Selected = a | Private Key Selected = b |
| Key generated = | Key generated = |
| Exchange of generated keys takes place | |
| Key received = y | key received = x |
| Generated Secret Key = | Generated Secret Key = |
| Algebraically, it can be shown that | |
| Users now have a symmetric secret key to encrypt | |

**Example:**

Step 1: Alice and Bob get public numbers P = 23, G = 9

Step 2: Alice selected a private key a = 4 and

Bob selected a private key b = 3

Step 3: Alice and Bob compute public values

Alice: x =(9^4 mod 23) = (6561 mod 23) = 6

Bob: y = (9^3 mod 23) = (729 mod 23) = 16

Step 4: Alice and Bob exchange public numbers

Step 5: Alice receives public key y =16 and

Bob receives public key x = 6

Step 6: Alice and Bob compute symmetric keys

Alice: ka = y^a mod p = 65536 mod 23 = 9

Bob: kb = x^b mod p = 216 mod 23 = 9

Step 7: 9 is the shared secret.

**SHA-1 or Secure Hash Algorithm 1** is a cryptographic hash function which takes an input and produces a 160-bit (20-byte) hash value. This hash value is known as a message digest. This message digest is usually then rendered as a hexadecimal number which is 40 digits long. It is a U.S. Federal Information Processing Standard and was designed by the United States National Security Agency. SHA-1 is now considered insecure since 2005. Major tech giants browsers like Microsoft, Google, Apple and Mozilla have stopped accepting SHA-1 SSL certificates by 2017. To calculate cryptographic hashing value in Java, **MessageDigest Class** is used, under the package **java.security**. MessageDigest Class provides following cryptographic hash function to find hash value of a text as follows:

MD2

MD5

SHA-1

SHA-224

SHA-256

SHA-384

SHA-512

These algorithms are initialized in static method called **getInstance()**. After selecting the algorithm the message digest value is calculated and the results are returned as a byte array. BigInteger class is used, to convert the resultant byte array into its signum representation. This representation is then converted into a hexadecimal format to get the expected MessageDigest.

**ALGORITHM:**

→Step 1: Append Padding Bits:-Message is “padded” with a 1 and as many 0’s as necessary to bring the message length to 64 bits fewer than an even multiple of 512.

→Step 2: Append Length.:-64 bits are appended to the end of the padded message. These bits hold the binary format of 64 bits indicating the length of the original message.

→Step 3: Prepare Processing Functions….

SHA1 requires 80 processing functions defined as:

f(t;B,C,D) = (B AND C) OR ((NOT B) AND D) ( 0 <= t <= 19)

f(t;B,C,D) = B XOR C XOR D (20 <= t <= 39)

f(t;B,C,D) = (B AND C) OR (B AND D) OR (C AND D) (40 <= t <=59)

f(t;B,C,D) = B XOR C XOR D (60 <= t <= 79)

→Step 4: Prepare Processing Constants....

SHA1 requires 80 processing constant words defined as:

K(t) = 0x5A827999 ( 0 <= t <= 19)

K(t) = 0x6ED9EBA1 (20 <= t <= 39)

K(t) = 0x8F1BBCDC (40 <= t <= 59)

K(t) = 0xCA62C1D6 (60 <= t <= 79)

→Step 5: Initialize Buffers….

SHA1 requires 160 bits or 5 buffers of words (32 bits):

H0 = 0x67452301

H1 = 0xEFCDAB89

H2 = 0x98BADCFE

H3 = 0x10325476

H4 = 0xC3D2E1F0

→Step 6: Processing Message in 512-bit blocks (L blocks in total message)…. This is the main task of SHA1 algorithm which loops through the padded and appended message in 512-bit blocks. Input and predefined functions M[1, 2, ..., L]: Blocks of the padded and appended message f(0;B,C,D), f(1,B,C,D), ..., f(79,B,C,D): 80 Processing Functions K(0), K(1), ..., K(79): 80 Processing Constant Word H0, H1, H2, H3, H4, H5: 5 Word buffers with initial values