Course Notes for

CS 1501
Algorithm Implementation

By
John C. Ramirez
Department of Computer Science
University of Pittsburgh

- These notes are intended for use by students in CS1501 at the University of Pittsburgh and no one else
- These notes are provided free of charge and may not be sold in any shape or form
- These notes are NOT a substitute for material covered during course lectures. If you miss a lecture, you should definitely obtain both these notes and notes written by a student who attended the lecture.
- Material from these notes is obtained from various sources, including, but not limited to, the following:
  - Algorithms in C++ by Robert Sedgewick
  - Algorithms, 4th Edition by Robert Sedgewick and Kevin Wayne
  - Introduction to Algorithms, by Cormen, Leiserson and Rivest
  - Various Java and C++ textbooks
  - Various online resources (see notes for specifics)

- GCD(A, B)
  - Largest integer that evenly divides A and B
    - i.e. there is no remainder from the division
  - Simple, brute-force approach?
    - Start with min(A,B) since that is largest possible answer
    - If that doesn't work, decrement by one until the GCD is found
    - Easy to code using a simple for loop
  - Worst case run-time?
    - We want to count how many mods we need to do
      - Assume time for a mod is similar to a mult ($\sim N^2$)
    - **Theta(min(A, B))** mods are needed – is this good?

- Remember exponentiation?
- A and B are N bits, and the loop is linear in the VALUE of min(A,B)
  - This is exponential in N, the number of bits!
  - We could try up to ~2^N mods in worst case (ex: what would be worst situation for GCD?)

How can we improve?

- Famous algorithm by Euclid:

  GCD(A,B) = GCD(B, A mod B)
- Ok let's try it
  - GCD(30,24) = GCD(24,6) = GCD(6,0) = ?????
- What is missing?
  - The base case: GCD(A,B) = A if B = 0
  - Now GCD(6,0) = 6 and we are done

- Run-time of Euclid's GCD?
  - Let's again count number of mods
  - Tricky to analyze exactly, but in the worst case it has been shown to be linear in N, the number of bits
  - Similar improvement to exponentiation problem
  - Also can be easily implemented iteratively
- Extended GCD
  - It is true that GCD(A,B) = D = AS + BT for some integer coefficients S and T
    - Ex: GCD(30,24) = 6 = (30)(1) + (24)(-1)
    - Ex: GCD(99,78) = 3 = (99)(-11) + (78)(14)

- With a bit of extra logic (same Theta run-time), GCD can also provide the coefficients S and T
- This is called the Extended Greatest Common Divisor algorithm
  - We will see soon that this will be useful in RSA encryption

- Arithmetic summary
  - We have looked at multiplication, exponentiation, and GCD (XGCD)
  - These will all be necessary when we look at public key encryption next

- What is Cryptography?
  - Designing of secret communications systems
    - A **SENDER** wants a **RECEIVER** (and no one else) to understand a **PLAINTEXT** message
    - Sender **ENCRYPTS** the message using an *encryption algorithm* and some *key parameters*, producing **CIPHERTEXT**
    - Receiver has *decryption algorithm* and key parameters, and can restore original plaintext

- Why so much trouble?
  - **CRYPTANALYST** would like to read the message
    - Tends to be very clever
    - Can usually get a copy of the *ciphertext*
      - If this were NOT the case we would not need encryption at all – think about it
    - Usually knows (or can figure out) the *encryption algorithm*
      - These are typically published and well-known algorithms

- So the **key parameters** (and how they affect decryption) are really the only thing preventing the cryptanalyst from decrypting
  - So cryptanalyst would really like to know or figure out these key parameters
  - Alternatively (since cryptanalyst is clever) they may hope to decrypt your message without knowing them
    - There may be some other ways to determine what the message was

Some Encryption Background

- Early encryption schemes were quite simple
  - Ex. Caesar Cipher
    - Algorithm: Simple shift of letters by some integer amount
    - Key parameters: Amount of the shift (integer)

ABCDEFGHIJKLMNO. . . <- original alphabet

ABCDEFGHIJKLMNOPQR. . . <- letters used in code

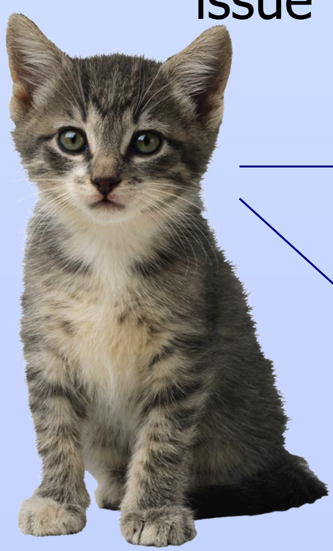
shift = 3

KHOOR <- ciphertext

- Almost trivial to break by brute force
  - Try each shift until right one is found
    - Only 25 possible shifts for letters, 255 for ASCII
  - Try arbitrary substitution instead: Substitution Cipher
    - Algorithm: Permute the alphabet to create encryption alphabet
    - Key Parameters: Specific permutation used
      - Sender and receiver know permutation, but cryptanalyst does not
  - Much more difficult to break by brute force
    - With alphabet of S characters, S! permutations

Some Encryption Background

- But still relatively easy to break today in other ways
  - Frequency tables, sentence structure
  - Play "Wheel of Fortune" with the ciphertext
  - Once we guess some letters, others become easier
    - Not a trivial program to write, by any means
    - But run-time is not that long – that is the important issue



Hey Pat, can I buy a vowel?

These "before and after"s always give me trouble

Some Encryption Background

- Better if "structure" of ciphertext differs from that of plaintext
  - We can try to do this with the alphabet and in the source message
    - Remove vowels, remove punctuation and spaces, etc
  - However, it is better if we can achieve this as part of the encryption process
    - Ex: Instead of substitution, "add" a key value to the plaintext
      - We can add a different key value for each position in the plaintext

- If the keys are pseudorandom, this will allow us to map the same plaintext character to different ciphertext characters, based on the location
- Ex: Assume a simple key of ABCD that we add cyclically to our plaintext

```
ALGORITHMS ARE COOL   <- original message
ABCDABCDABCDABCDABC   <- key sequence
-------------------
BNJSSKWLNUCESGCDPQO   <- ciphertext
```

- With more complex keys we can create a fairly challenging code

- Vigenere Cipher (and other similar ciphers)
  - Now the same ciphertext character may represent more than one plaintext character
  - The longer the key sequence, the better the code
    - If the key sequence is random and is as long as the message, we call it a Vernam Cipher
  - This is effective because it makes the ciphertext appear to be a "random" sequence of characters
    - The more "random", the harder to decrypt

Some Encryption Background

- Vernam Cipher is provably secure for one-time use (as long as key is not discovered)
  - Since a "random" character is added to each character of the message, the ciphertext appears to be completely random to the cryptanalyst
  - However, with repeated use its security diminishes somewhat
  - Used in military applications when absolute security is required
    - See http://en.wikipedia.org/wiki/One-time_pad

      http://www.pro-technix.com/information/crypto/pages/vernam_base.html

- Variations and combinations of this technique are used in some modern encryption algos
    - Ex. 3DES, AES, Blowfish
  - Most of these are called block ciphers, because they process the data in blocks (ex: 128 bits)
  - They are also called symmetric ciphers, because the encryption and decryption keys are either the same or easily derivable from each other
    - See: https://en.wikipedia.org/wiki/Symmetric-key_algorithm
  - An advantage of symmetric ciphers is that they are very fast
    - Typically linear in the size of the message

- Symmetric ciphers can be effective, but have a **KEY DISTRIBUTION** problem
  - How can key pass between sender and receiver securely?
    - Problem is recursive (must encrypt key; must encrypt key to decrypt key, etc).
  - How can we prevent multiple sender/receivers from reading each other's messages?
    - Need a different key for each pair of users
  - The solution is to have an asymmetric cipher, also called public-key cryptography

- In **PUBLIC-KEY** cryptography
  - Key has two parts
    - A **public** part, used for encryption
      - Everyone can know this
    - A **private** part, used for decryption
      - Only receiver should know this
  - Solves distribution problem
    - Each receiver has their own pair of keys
    - Don't care who knows public part
    - Since senders don't know private part, they can't read each other's messages
- **RSA** is most famous of these algorithms

- The idea of an asymmetric cipher is generally attributed to Diffie and Helman
  - See: https://en.wikipedia.org/wiki/Diffie%E2%80%93Hellman_key_exchange
- Rivest, Shamir and Adelman (RSA) from MIT were the first to prominently achieve the goals of an asymmetric cipher
  - Others exist but we will focus on RSA

- How/why does RSA work?

Let E = encryption (public) key operation

Let D = decryption (private) key operation

1) D(E(plaintext)) = plaintext
   - E and D operations are inverses
2) All E, D pairs must be distinct (mod PHI)
3) Knowing E, it must be VERY difficult to determine D (exponential time)
4) E and D can be created and used in a reasonable amount of time (polynomial time)

- Theory?
  - We will be "lite" on theory here, but …
  - Assume plaintext is an integer, M
    - **C** = ciphertext **= M^E mod N**
      - So E simply is a power
      - We may need to convert our message into an integer (or possibly many integers) first, but this is not difficult – we can simply interpret each block of bits as an integer
    - **M = C^D mod N**
      - D is also a power
    - Or **M^{ED} mod N = M**
  - So our public key is (E, N) and private key is (D, N)
    - So how do we determine E, D and N?

- Not trivial by any means
  - This is where we need our extremely large integers – 512, 1024 and more bits
- Process
  - Choose random prime integers X and Y
  - Let N = XY
  - Let PHI = (X-1)(Y-1)
  - Choose another random prime integer (less than PHI and relatively prime to PHI) and call this E
  - Now all that's left is to calculate D
    - We need some number theory to do this
    - We will not worry too much about the theory

- We must calculate D such that ED mod PHI = 1
  - We will not go over theory for this but if D satisfies this equation then our formulas from Slide 22 will work
- Note that this is saying that the remainder of ED divided by PHI is 1 or that

ED = 1 + K(PHI) [for some K, and, rearranging]

1 = ED – K(PHI) [rearranging a bit more]

1 = PHI(-K) + ED

  - Luckily we can solve this using XGCD
  - We know already that GCD(PHI, E) = 1
    - > Why?
  - We also know, from XGCD discussed prev., that

GCD(PHI,E) = 1 = (PHI)S + (E)T for some S and T

Public Key Cryptography

- XGCD calculates values for S and T
  - S will be –K (we don't really care about this)
  - T will be D (this is our decoding key)
- Let's look at a simple example
  - We will use small numbers for this example just to make the math easier
  - For a real key we would use very large numbers

Public Key Cryptography

X = 7, Y = 11 (random primes)

XY = N = 77

(X-1)(Y-1) = PHI = 60

E = 37 (random prime < PHI)

Solve the following for D:

37D mod 60 = 1

Converting to form from prev. slides and solve using XGCD:

GCD(60,37) = 1 = (60)(-8)+(37)(13)

So D = 13

- $C = M^E \bmod N \rightarrow M^{37} \bmod 77$
- $M = C^D \bmod N \rightarrow C^{13} \bmod 77$
  - See RSATest.java

Is RSA Feasible?

- In order for RSA to be useable
  1) The keys must be able to be generated in a reasonable (polynomial) amount of time
  2) The encryption and decryption must take a reasonable (polynomial) amount of time
  3) Breaking the code must take an extremely large (exponential) amount of time
  - Let's look at these in our next lecture