Written Solutions to MIT PRIMES' 2017 Computer Science Problems

Problem 1

Part 1:

How many secret keys total would that organization need?

You could use combinations. Since there are 2 people share 1 key and there are 100 people, we would do $C(100,2) = \frac{100!}{2!(100-2)!} = 4950$. **The organization of 100 people would need 4950 secret keys in total.**

What about an organization of 1000 people?

$C(1000,2) = \frac{1000!}{2!(100-2)!} = 499500$. **The organization of 1000 people would need 499500 secret keys in total**

Come up with the formula for an organization of N people and prove it.

Let *p* be the amount of secret keys

Let *N* be the number of people in the organization

$$p = \frac{N!}{2!(N-2)!}$$

Proof:

Every two people have a secret key shared between them. Since the order of each person doesn't matter (the key between person A and person B is the same key between person B and person A), we can use the combination formula: C(n, r) = $\frac{n!}{r!(n-r)!}$. *n* would be the number of people in the organization because that is the amount of things to choose from. *r* would be 2 because we are choosing 1 secret key between 2 people.

Part 2:

Assuming that we also need a separate secret key for any group of $3 \leq k \leq N$ participants, how many keys total would we need?

Let *p* be the amount of secret keys

Let *N* be the number of people in the organization

$$\sum_{k=2}^{N} p_k = \frac{N!}{2!(N-2)!} + \frac{N!}{3!(N-3)!} + \frac{N!}{4!(N-4)!} + \ldots + \frac{N!}{(N-1)!(N-(N-1))!} + \frac{N!}{N!(N-N)!}$$

p_k **is the number of keys given *k* people.**

So, for 100 people there would need to be 1267650600228229401496703205275 keys.

Part 3:

Public key cryptography also allows a clear way of identifying the sender. If someone encrypts his or her message with their private key, the receiver can tell who it came from because the receiver can decrypt it with his or her public key. This allows for a clear system to verify a genuine message.

Problem 2:

Function is written in the "CSProblem2.java" file.

Yes, my algorithm can handle a larger prime than any of the prime numbers given. An example of a prime you can test with my program is 10093100991010310111 .

Problem 3:

1) The best case time complexity is $O(1)$ and the worst case time complexity is $O(sqrt(n))$ because if the input was 0,1, or 2, my algorithm would immediately output the right answer. The worst case would be $O(sqrt(n))$ because my algorithm checks the squared of the current iteration to save time.
2) The algorithm runs the longest if I am testing a prime number or a number with factor(s) close to half the number. If it is testing a prime, my for loop would have to finish which would be the longest time possible.
3) **For every digit increase, the worst-case time approximately increases polynomially** because the worst case time complexity is $O(sqrt(n))$. If someone increases the number of digits for low digit numbers, the time complexity would slowly increase. If someone increased the number of digit for higher digit numbers, the time complexity would increase much faster. Also, mathematically speaking the inverse of the square root function is an polynomial function.

Problem 4:

Function located in "CSProblem4.java" file.

1. 98496421
2. 1

Problem 5:

Function located in "CSProblem5.java" file.

1. 7
2. 6658951232733
3. 5879176124998

The worst-case running time is $O(n) = logn$, given that the number of digits m. The best-case running time is $O(n) = 1$. So, as the the number of digits *m* increases by 1, the running time increases exponentially (10^{d} *for every additional digit*).

Problem 6:

Function located in "CSProblem6.java" file.

1. 8173443850111
2. 2784574781170
3. 1737895540235

For every extra decimal digit of m, the program would take 10^{2} times longer than the previous length of m because modular multiplication and exponentiation runs generally in $O(n^{2})$

Problem 7:

Part 1: Function in CSProblem7.java

1. Public key is n = 126555186554398568341 and e = 23. Private key is d = 110047988288481331367
2. Public key is n = 126555186554398568341 and e = 65537. Private key is d = 124295858697977929025

Part 2:

Choosing *e* as a prime may not be sufficient because the *e* has to simultaneously coprime to p-1 and q-1, not small, and have padding. For example, if *e* was 3, then an attacker can use the Chinese Remainder Theorem to decrypt the encrypted messages, using the public key(s). Since an attacker can derive statistical information of a ciphertext, padding is necessary to make RSA not deterministic any more (by adding randomness).

Problem 8:

1. The parts of the key generation that are fast for long numbers is the Euclidean Algorithm and Extended Euclidean Algorithm because it runs in logarithmic time complexity. It is slower when it is generating large primes since that runs in exponential time complexity.
2. Probabilistic primality testing is used instead of the method I used in Problem 2 because they are generally much faster than deterministic testing.

Problem 9:

1. Encrypted Message: **62022011753771218891**
2. e: **65537** , Decrypted Message: **42**
3. e: **17** , Decrypted Message: **77777777777**

Problem 10:

It's fast to encrypt a message *m* using *n* and *e* because *n* and *e* are already calculated thus reducing the main computation to a modulus power computation: $c = m^e \mod n$. There is no need to calculate the coprime of the euler's totient function of n $(\varphi(n) = (p-1)(q-1))$. It's also fast to compute *m* from its encryption *c* and private key *d* because there requires only one main computation: $m^{ed} = m \mod n$ Without the knowledge *c* (the message), there would be no way of producing the deciphered message. Without the *d*, one would have to brute force the modular multiplicative inverse of e (mod φ(n)) to get *d*.

Problem 11:

Yes, Oscar can use the encrypted text info he received from Alice when she bought the mug to reverse engineer the account number of Alice. Since the equation for encrypting the account number is simply $a^e \mod n$, he can find the account number by using this equation (assume d is encrypted message): $a = d^e \mod n$. Oscar could then transfer a large amount of money to his account using Alice's account number to trick Bob to deposit a large amount of money to his (Oscar) account.

Problem 12:

Yes, Oscar can still decrypt the concatenated string, assuming he has access to the public key *n*, *e* . After decrypting the concatenated string using (assume d is encrypted message and m is decrypted message) $m = d^e \mod n$, he can split the message into chunks based on length (1 digit for instruction, 10-digit number for account, and the rest for the amount). Thus, with the knowledge of Alice's account number, he can transfer money from Alice to Bob to his own account using "2" for transfer.

I would prevent this by providing a signature in each message to ensure that each message is authentic. Using a hash function, each signature has to be produced with the private key, thus ensuring the message came from only the person with the private key. Thus, since only the public key is available in Oscar's case, one can check the message's authenticity by placing a signature policy.

Problem 13:

1. Signature is verified
2. Signature is not verified

Problem 14:

The verification part of a digital signature is very fast due to short public keys *e* and fast acceleration operations. It is fast to sign a message knowing the private key and to verify a signature on a message *m* knowing only the public key because $s = x^d$ and $x' = s^e$. It's prohibitively slow to forge a digital signature on a desired message *m* knowing only the public key because one would have to brute force what *p* and *q* was to get the totient function and thus find the multiplicative inverse of *e modulo φ(n)* to get the private key *d* which would then help to finally calculate the signature: $s = x^d$

It is fast to obtain a valid pair of a message *m* and its digital signature *c* knowing only the signer's public key if it doesn't matter to the forger what *m* is because only the public key is needed to verify if the message is valid. No, you would need the private key *d* to forge a signature. One could brute force the value *d* but that would be nearly impossible.

Problem 15:

See function in CSProblem15.java

Step by step demonstration of Alice obtaining a blind signature from Bob (also written in CSProblem15.java [type s to run function]):

1. Acquire given information: 33333333333 is the message, 694420975411 is p, 12583867141 is q, 7272195257079707781473 is d
2. Compute public key
    a. n = p*q
    b. To get e, calculate totient function of p and q then find the multiplicative inverse of d and phiN
3. You can then obtain a blind signature by:
    a. Choose a random number r (it is called the blinding factor).
    b. Compute $m' = mr^e \bmod n$
    c. Compute $s' = (m')^d \bmod n$
    d. Compute s = $s' r^{-1} \bmod n$
4. To Verify:
    a. Compute x' = $s^e \bmod n$
    b. If x' is equal to $m \bmod n$ then it is verified

Results:

- Signature s on m: 2714515962155244094473 and it is verified!

Problem 16:

No, this digital signature scheme is not completely safe from double spending. Vendors can cheat this system by creating a brute force system of faking signatures and generating random numbers that aren't checked to gain credit. This would enable the vendors to withdraw

as much cash as they want since the bank cannot tell who paid the vendor. Nevertheless, Alice cannot easily duplicate cash without a check from various steps such as at the beginning at the bank and at vendors. There are various checks to make sure the digital cash is distinct and from Alice, such as random numbers and

Problem 17:

An attacker can give a blinded version of the victim's message encrypted with his/her public key to sign. Since the victim signs the message directly during a blind signature, the attacker will see that the message will be clearly revealed. The signing and decrypting function is the same with the same public key, thus it is best to use different public/private keys for signing and decrypting

Sources:

- *Understanding Cryptography* by Christof Paar and Jan Pelzi
- https://www.khanacademy.org/computing/computer-science/cryptography/modarithmetic
- http://www.mathaware.org/mam/06/Kaliski.pdf
- https://engineering.purdue.edu/kak/compsec/NewLectures/Lecture12.pdf