Allison Neyer

Homework #3

1. Python translation of your previous posted answers for your past CMSI 282 class on http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/

**Bozosort in Python; bozosort1.py:**

##I modified the following program from http://excode.io/code/27/bozosort/python

import random

#Returns whether list is sorted

def is\_sorted(list):

for i in range(1, len(list)):

if list[i-1] > list[i]:

return False

return True

"""

Does a bozosort and returns the number of swaps required to sort.

"""

def bozosort(list):

swaps = 0

while not is\_sorted(list):

# get random items to swap

index\_1 = random.randint(0, len(list)-1)

index\_2 = random.randint(0, len(list)-1)

if index\_1 != index\_2:

# swap

temp\_holder = list[index\_2]

list[index\_2] = list[index\_1]

list[index\_1] = temp\_holder

swaps = swaps+1

return swaps

"""

Returns a randomly filled in integer list for a given size

"""

def random\_list(size):

list = [random.randint(0, size-1) for i in range(size)]

return list

if \_\_name\_\_ == "\_\_main\_\_":

# try 10 sorts

iterations = 10

print 'Size Average number of swaps'

print '------------------------------'

total\_swaps = 0

for trial\_size in range (1, 15):

total\_swaps = 0

for k in range(0, iterations):

test\_list = random\_list(trial\_size)

random.shuffle(test\_list)

total\_swaps = total\_swaps + bozosort(test\_list)

print '%2d%20.2f\n' %(trial\_size, total\_swaps / float(iterations))

# list = [5, 15, 14, 1, -6, 26, -100, 0, 99]

# print 'Initial:'

# print ' '.join([str(i) for i in list])

# total\_swaps = bozosort(list)

# print 'Total Swaps: ', total\_swaps

# print 'avg swaps', (total\_swaps/2)

# print '\nResult:'

# print ' '.join([str(i) for i in list])

1. Autokey Vigenere cipher in Java:

I found this code online from your previous posted answers for your past CMSI 282 class on <http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/>

/\*\*

\* I found this code online from your previous posted answers for your past CMSI 282 class on http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/

\*\*/

//package edu.lmu.cs.crypto;

/\*\*

\* An auto-key Vigenere cipher. During encryption, the input text is used for

\* the key stream after the initial key is used up. During decryption, we get

\* the key characters from the plaintext we are recovering.

\*/

**public** **class** AutoKeyVigenere {

//Junit no longer exists so this is my test of the AutokEyVigenere class

/\*\* public static void main(String[] args) {

String[] messages = { "Attack **@T** DaWN!", "",

"fjh94h8\u3032\u412b2r98h923h", "%" };

String[] keys = { "......11111111111111111111111111111..",

"\uffff,\uff73" };

for (String message : messages) {

for (String key : keys) {

String cipherText = AutoKeyVigenere.encipher(message, key);

String recoveredText = AutoKeyVigenere

.decipher(cipherText, key);

System.out.println("The original message is: "+message+". The recovered text is: "+ recoveredText);

System.out.println("They should be equal.");

System.out.println("The cipher for this message is: "+cipherText);

}

}

try {

AutoKeyVigenere.encipher("hello", ""); //This should throw an illegal arg exception

}catch (Exception e) {

// try and catch to see if illegal argument exception is thrown. If it's thrown we enter the catch.

System.out.println("\n Illegal Argument Exception is thrown.");

}

} \*/

/\*\*

\* Encrypt the given plain text with the given key.

\*/

**public** **static** String encipher(String plaintext, String key) {

**return** *encipher*(plaintext, key, 1);

}

/\*\*

\* Decrypt the given ciphertext with the given key.

\*/

**public** **static** String decipher(String ciphertext, String key) {

**return** *encipher*(ciphertext, key, -1);

}

**private** **static** String encipher(String text, String key, **int** multiplier) {

**if** ("".equals(key)) {

**throw** **new** IllegalArgumentException("Key cannot be empty");

}

StringBuilder builder = **new** StringBuilder();

**int** keyLength = key.length();

**for** (**int** i = 0, n = text.length(); i < n; i++) {

**char** k = i < keyLength ? key.charAt(i) : multiplier == 1 ? text

.charAt(i - keyLength) : builder.charAt(i - keyLength);

builder.append((**char**) (k \* multiplier + text.charAt(i)));

}

**return** builder.toString();

}

}

1. LETUS CHANG EOURT RADIT IONAL ATTIT UDETO THECO   
   NSTRU CTION OFPRO GRAMS INSTE ADOFI MAGIN INGTH   
   ATOUR MAINT ASKIS TOINS TRUCT ACOMP UTERW HATTO

DOLET USCON CENTR ATERA THERO NEXPL AININ GTOHU

MANBE INGSW HATWE WANTA COMPU TERTO DO

The message is: “Let us change our traditional attitude to the construction of programs instead of imagining that our main task is to instruct a computer what to do Let us concentrate rather on explaining to human beings what we want a computer to do”

I used <http://www.cryptoclub.org/tools/cracksub_topframe.php>.

Also, you mentioned in class there are two ways to break a cipher: find the key via frequency analysis or intercept the key. Like a good spy, I found your key online from your previous posted answers for your past CMSI 282 class on http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/.

1. I worked through the problem and verified the answer with your previous posted answers from your past CMSI 282 class on http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/

Polybius Square:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 1 | 2 | 3 | 4 |
| 0 | D | A | R | N | O |
| 1 | T | H | E | C | Y |
| 2 | P | L | S | I | Q |
| 3 | U | B | F | G | K |
| 4 | M | V | W | X | Z |

Skip the J (or at least one letter is skipped but be consistent.)

Encrypted Message:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| T | W | B | T | L | L | A | E | P | O |
| 10 | 42 | 31 | 10 | 21 | 21 | 01 | 12 | 20 | 04 |
| D | **T** | **U** | **B** | **T** | **W** | **B** | **T** | **L** | **T** |
| 00 | 10 | 30 | 31 | 10 | 42 | 31 | 10 | 21 | 10 |
| D | **L** | **D** | **D** | **V** | **S** | **N** | **N** | **H** | **E** |
| 00 | 21 | 00 | 00 | 41 | 22 | 03 | 03 | 11 | 12 |
| E | **T** | **L** | **S** | **K** | **D** | **D** | **S** | **I** | **F** |
| 12 | 10 | 21 | 22 | 34 | 00 | 00 | 22 | 23 | 32 |
| G | **I** | **I** | **M** | **W** | **L** | **Y** | **D** | **K** | **D** |
| 33 | 23 | 23 | 40 | 42 | 21 | 14 | 00 | 34 | 00 |
| D | **S** | **P** | **H** | **B** | **P** | **Q** | **K** | **O** | **F** |
| 00 | 22 | 20 | 11 | 31 | 20 | 24 | 34 | 04 | 32 |
| H | **M** | **D** | **L** | **S** | **K** | **R** | **S** |  |  |
| 11 | 40 | 00 | 21 | 22 | 34 | 02 | 22 |  |  |

Write letter of the plaintext above the (row, column) coordinates:

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Message** | **CO** | **MP** | **UT** | **ER** | **SC** | **IE** | **NC** | **EI** | **SN** |
| Row 1 | 10 | 42 | 31 | 10 | 21 | 21 | 01 | 12 | 20 |
| Column 1 | 34 | 00 | 00 | 22 | 23 | 32 | 33 | 23 | 23 |
| **Message** | **OM** | **OR** | **EA** | **BO** | **UT** | **CO** | **MP** | **UT** | **ER** |
| Row 2 | 04 | 00 | 10 | 30 | 31 | 10 | 42 | 31 | 10 |
| Column 2 | 40 | 42 | 21 | 14 | 00 | 34 | 00 | 00 | 22 |
| **Message** | **ST** | **HA** | **NA** | **ST** | **RO** | **NO** | **MY** | **IS** | **AB** |
| Row3 | 21 | 10 | 00 | 21 | 00 | 00 | 41 | 22 | 03 |
| Column 3 | 20 | 11 | 31 | 20 | 24 | 34 | 04 | 32 | 11 |
| **Message** | **OU** | **TT** | **EL** | **ES** | **CO** | **PE** | **SS** |  |  |
| Row4 | 03 | 11 | 12 | 12 | 10 | 21 | 22 |  |  |
| Column 4 | 40 | 00 | 21 | 22 | 34 | 02 | 22 |  |  |

**ANSWER**: COMPU TERSC IENCE ISNOM OREAB OUTCO MPUTE RSTHA NASTR ONOMY ISABO UTTEL ESCOP ESS

MESSAGE: COMPUTER SCIENCE IS NO MORE ABOUT COMPUTERS THAN ASTRONOMY IS ABOUT TELESCOPESS.

1. If someone’s RSA public key is (729880581317, 5), then what is her private key?

N = 729880581317 and e = 5 due to the public key notation (n,e) given in the problem. So we know that N = 729880581317 and e = 5.

Step 1) Factor N= p\*q to get p = 822893 and q =886969. After all, 822893 \* 886969 = 729880581317. So, (p-1)(q-1) = 729878871456. So d = 5.modInverse(729878871456) = 583903097165. The private key is (N,d) = (729880581317, 583903097165).

I found this online from your previous posted answers for your past CMSI 282 class on <http://cs.lmu.edu/~ray/classes/a/assignment/2/answers/>

1.45a) We would want digital signatures to verify the identity of message senders and ensure the message is secure and has not been tampered with. This “derives very strong guarantees of security.” P.33 of textbook.

“Public-key schemes such as RSA are significantly more subtle and tricky: they allow Alice to send a message to Bob without their ever having met before. Bob’s encryption function e(\*) is publicly available, and Alice can encrypt her message with this function, thereby digitally locking it. Only Bob knows the key to quickly unlocking this digital lock: the decryption function d(\*). The point is that Alice and Bob need only perform simple calculations to lock and unlock the message respectively- operations that nay pocket computing device could handle. By contrast, to unlock the message without the key, Eve must perform operations like factoring large numbers, which requires more computational power than would be afforded by the world’s most powerful computers combined. This compelling guarantee enables secure Web commerce, such as sending credit card numbers to companies over the Internet. ” p.31 of textbook

1.45b)

**RSA Algorithm**

First, you need to generate a public key and a corresponding private key. Choose two large primes *p* and *q* (probably around 256 bits each). Multiply them together, and call the result *n.* The factors *p* and *q* will remain secret. (You won't tell anybody, and it's practically impossible to factor numbers that large.)

To generate your public key, choose a number *e* that is relatively prime to θ(n). Since you know *p* and *q,* you know θ(n)--it's (p-1)(q-1). Your public key is (e,n).

To generate your private key, find the number *d* that is the multiplicative inverse of *e* mod θ(n). (d, n) is your private key.

To encrypt a message *m* (< n), someone using your public key should compute ciphertext *c* = *me* mod *n.* Only you will be able to decrypt *c,* using your private key to compute *m* = *cd* mod *n.* Also, only you can sign a message *m* (< n) with signature s = *md* mod *n* based on your private key. **Anyone can verify your signature by checking that *m* = *se* mod *n. (answer)* (**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

**Why Does RSA Work?**

RSA does arithmetic mod *n,* where *n* = *pq.* We know that θ(n) = *(p-* 1 )(q-1 ). We've chosen *d* and *e* such that *de =* 1 mod θ(n). Therefore, for any *x, xde* = *x* mod *n.* An RSA encryption consists of taking *x* and raising it *to e.* If we take the result and raise it to the *d* ( i.e., perform RSA decryption), we'll get *(xe)d,* which equals *xed,* which is the same as *x.* So we see that decryption reverses encryption.

In the case of signature generation, x is first raised to the *d* power to get the signature and then the signature is raised to the e power for verification; the result, *xde,* will equal *x.* **(**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

Proof:

If the mapping x🡪 xe mod N is invertible, it must be a bijection; hence statement 2 (the inverse mapping is easily realized: let d be the inverse of e modulo (p-1)(q-1). Then for all x ∈ {0, …, N-1}. (xe)d x mod N) implies statement 1 (The mapping x 🡪 xe mod N is a bijection on {0,1, … , N-1}.). To prove statement 2, we start by observing htat e is invertible modulo (p-1)(q-1) because it is relatively prime to this number. To see that (xe)d x mod N, we examine the exponent: since ed 1 mod (p-1)(q-1), we can write ed in the form 1 + k(p-1)(q-1) for some k. Now we need to show that the difference xed – x = x1+k(p-1)(q-1) –x is always 0 modulo N. The second form of the expression is convenient because it can be simplied using Fermat’s little theorem. It is divisible by p (since xp-1 1 mod p) and likewise by 1. Since p and q are primes, this expression must also be divisible by their product N. Hence xed – x = x1+k(p-1)(q-1) – x 0 (mod N). p.33-34 of text

Anyone who knows the public key can perform verify, i.e. they can check that a signature really was created by the private key because by publishing the public key, everyone can use it to send an encrypted message to the owner of the private key. The owner of the private key can use the private key, d, to decrypt the message by simply raising them to the dth power modulo N. p.33.

**In the case of signature generation, x is the first raised to the d power to get the signature and then the signature is raised to the e power for verification; the result xde, will equal x. p.33 of text book**

Logically, if the a message is used to If this does not come back correctly then the message was not encrypted using the private key.

1.45c)

**RSA Algorithm**

First, you need to generate a public key and a corresponding private key. Choose two large primes *p* and *q* (probably around 256 bits each). Multiply them together, and call the result *n.* The factors *p* and *q* will remain secret. (You won't tell anybody, and it's practically impossible to factor numbers that large.)

To generate your public key, choose a number *e* that is relatively prime to θ(n). Since you know *p* and *q,* you know θ(n)--it's (p-1)(q-1). Your public key is (e,n).

To generate your private key, find the number *d* that is the multiplicative inverse of *e* mod θ(n). (d, n) is your private key.

To encrypt a message *m* (< n), someone using your public key should compute ciphertext *c* = *me* mod *n.* Only you will be able to decrypt *c,* using your private key to compute *m* = *cd* mod *n.* Also, only you can sign a message *m* (< n) with signature s = *md* mod *n* based on your private key. **Anyone can verify your signature by checking that *m* = *se* mod *n.* (**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

my example:

**p** **= 101** and **q** **= 11**; p\*q = (101)(11) = **1111 = N**

θ(n) = 1000. e =3 because 3 is relatively prime to 1000. **the public key is (3, 1111).**

To generate your public key, choose a number *e* that is relatively prime to θ(n). Relatively prime means the only common positive factor of the two numbers is 1 (i.e. if the only positive integer that evenly divides both of them is 1). Since you know *p* and *q,* you know θ(n)--it's (p-1)(q-1). So θ(n) = (101-1)(11-1)= (100)(10) = 1000. So e can be 3 because it’s relatively prime to 1000 because only 1 divides both of them evenly. Your **public key is (e,n).** **(**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

So, **the public key is (3, 1111).**

To generate your private key, find the number *d* that is the multiplicative inverse of *e* mod θ(n). (d, n) is your private key. **(**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

multiplicative inverse of 3 mod 1000 = 3-1 mod 1000 = **667** (wolfram alpha) **= d = private key.**

To encrypt a message *m* (< n), someone using your public key should compute ciphertext *c* = *me* mod *n.* Only you will be able to decrypt *c,* using your private key to compute *m* = *cd* mod *n.* Also, only you can sign a message *m* (< n) with signature s = *md* mod *n* based on your private key **(**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

m = Allison. Allison^(3) mod 1111 = (0112345)^3 mod 1111 = 789**.**

**mapping**: A = 0; L= 1; I = 2; S = 3; O = 4; N= 5

m1 = 0 = A

(m1d) mod N = 0667 mod 1111 = **0**.

(m1d)e = m1 (mod N) => 0667^3  = 0 (mod 1111) => 0667^3  = 0 = 0 (mod 1111) = 0.

1.45d)

NOTE: I’m supposed to remind you that there was a typo in the text instructions.

**The exponent should be raised to Bob’s secret key in order to decrypt.**

public key is in form (N, e). So if the public RSA key is (17, 391) then N = 17 and 3 = 391. So since 17 is a prime number, it can be only factored into two primes, 17 and 1. So p = 17 and q = 1 (note q can be equal to 17 and p can be equal to 1 and nothing would change). So θn = (p-1)(q-1) = (17-1)(1-1) = 0. So 391^-1 mod 0 is not found (according to wolfram alpha). So there is no d that it can be raised to.

OR one would use the multiplicative inverse according to the program below. The multiplicative inverse according to the program below is 18 and it would be raised to 18.

**multiplicative\_inverse.py:**

def multiplicative\_inverse(a, b):

origA = a

X = 0

prevX = 1

Y = 1

prevY = 0

while b != 0:

temp = b

quotient = a/b

b = a%b

a = temp

temp = X

a = prevX - quotient \* X

prevX = temp

temp = Y

Y = prevY - quotient \* Y

prevY = temp

return origA + prevY

print 'modinv: ', multiplicative\_inverse(17,391)

1.46a)

If Bob agrees to sign anything he is asked to, Eve can take advantage of this and decrypt any message sent by Alice to Bob because Eve can take any of Alice’s ciphertext messages (the encrypted message), because it is sent over an unsecured channel, and ask Bob to sign the ciphertext message. When Bob signs the ciphertext message, Bob is decrypting the message for Eve so now Eve has access to the plain text message without stealing Bob’s secret key because Bob handed over the plain text message to Eve by signing the ciphertext message. When Bob signs the ciphertext message, this decrypts the ciphertext message into a plain text message. Therefore, Eve doesn’t need to know Bob’s secret key because Bob is signing anything he is asked to and having Bob sign the ciphertext message decrypts it into a plain text message. Therefore, Eve can take advantage of Bob signing anything he is asked to and decrypt any message sent by Alice to Bob.

### Encrypting messages

Suppose Bob wishes to send a message *m* to Alice. He knows *N* and *e*, which Alice has announced. He turns *m* into a number *n* < *N*, using some previously agreed-upon reversible protocol. For example, each character in a plaintext message could be converted to its ASCII code, and the codes concatenated into a single number. If necessary, he can break *m* into pieces and encrypt each piece separately. He then computes the ciphertext *c*:

 c \equiv n^e\ (\mathrm{mod}\ N) 

This can be done quickly using the method of [exponentiation by squaring](http://en.wikipedia.org/wiki/exponentiating_by_squaring). Bob then transmits *c* to Alice.

### Decrypting messages

Alice receives *c* from Bob, and knows her private key *d*. She can recover *n* from *c* by the following procedure:

 c^d \equiv n\ (\mathrm{mod}\ N) 

Alice can then extract *n*, since *n* < *N*. Given *n*, she can recover the original message *m*.

The decryption procedure works because

 c^d \equiv n^{e \cdot d}\ (\mathrm{mod}\ N)

and *ed* ≡ 1 (mod *p*-1) and *ed* ≡ 1 (mod *q*-1). Fermat's little theorem yields

 n^{e \cdot d} \equiv n\ (\mathrm{mod}\ p)     and      n^{e \cdot d} \equiv n\ (\mathrm{mod}\ q) 

which implies (as *p* and *q* are *different* prime numbers)

 n^{e \cdot d} \equiv n\ (\mathrm{mod}\ pq) 

### Signing Messages

RSA can also be used to sign a message. Suppose Alice wishes to send a signed message to Bob. She produces a hash value of the message, encrypts it with her secret key, and attaches it as a "signature" to the message. This signature can only be decrypted with her public key. When Bob receives the signed message, he decrypts the signature with Alice's public key, and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's secret key, and that the message has not been tampered with since.

## Security

Suppose Eve, an eavesdropper, intercepts the public key *N* and *e*, and the ciphertext *c*. However, she is unable to directly obtain *d*, which Alice keeps secret. The most obvious way for Eve to deduce *n* from *c* is to factor *N* into *p* and *q*, in order to compute (*p*-1)(*q*-1) which allows the determination of *d* from *e*. No polynomial-time method for factoring large integers on a classical computer has yet been found, but it has not been proven that none exists.

http://en.wikibooks.org/wiki/Cryptography/RSA

1.46b)

Like in 1.46a) Eve can take Alice’s ciphertext message from the unsecured channel. Then she can invert it and send the message to Bob to sign it. Then she would take that message and un-invert it to get the plain text message from Alice. The key is getting Bob to sign the messages by making the signatures look random (i.e. when Bob decrypts the ciphertext message sent to him by Eve it still looks random rather than like plain text). So the message needs to be a randomly chosen message – that is a random number in the range {1, … , N-1}. In order to make the message appear randomly chosen to Bob when Eve sends the message to Bob, Eve must do something to the message that is invertible before giving it to Bob. Then when Bob signs it, the message appears to be junk to him and then Bob sends it back to Eve. Eve can then take this message and decrypt it with her own “key” by taking the inverse and so Eve can decrypt any message from Alice.

One way Eve can invert and un-invert the message is: Eve can mod N-1 the message and when Eve gets the message back Eve can raise the message to the Modular inverse of N-1 to get the plain text message.

After all these are the general steps/rule of encrypting and signing messages:

To generate your public key, choose a number *e* that is relatively prime to θ(n). Since you know *p* and *q,* you know θ(n)--it's (p-1)(q-1). Your public key is (e,n).

To generate your private key, find the number *d* that is the multiplicative inverse of *e* mod θ(n). (d, n) is your private key.

To encrypt a message *m* (< n), someone using your public key should compute ciphertext *c* = *me* mod *n.* Only you will be able to decrypt *c,* using your private key to compute *m* = *cd* mod *n.* Also, only you can sign a message *m* (< n) with signature s = *md* mod *n* based on your private key. **Anyone can verify your signature by checking that *m* = *se* mod *n.* (**[http://cs.ucsb.edu/~koc/ns/docs/kaufman/05pkc.pdf](http://cs.ucsb.edu/%7Ekoc/ns/docs/kaufman/05pkc.pdf)**)**

e is part of the public key and accessible to Eve. Therefore, Eve can do e.modInverse(N-1).

m = (md mod N)emod N = mde mod N

Alice’s ciphertext taken by Eve = me mod N. 🡺 Eve mods the message by for example (n-1) = me mod N mod (N-1) = altered ciphertext 🡺Bob signs the seemingly random message (altered ciphertext) 🡺 new ciphertext = mde mod N mod N-1 = ((md mod N)mod (N-1))emod N = 🡺 🡺Bob decrypts ciphertext and sends it back to Eve 🡺 md mod N mod N-1 🡺 Eve takes this message and takes the modular inverse of N-1 ( e.modInverse(N-1) ) to get the plain text back: md mod N mod (n-1) e.modinverse(N-1) to get plain text message back.

Alice’s ciphertext = me mod N

Signed by Bob = md mod N

### Encrypting messages

Suppose Bob wishes to send a message *m* to Alice. He knows *N* and *e*, which Alice has announced. He turns *m* into a number *n* < *N*, using some previously agreed-upon reversible protocol. For example, each character in a plaintext message could be converted to its ASCII code, and the codes concatenated into a single number. If necessary, he can break *m* into pieces and encrypt each piece separately. He then computes the ciphertext *c*:

 c \equiv n^e\ (\mathrm{mod}\ N) 

This can be done quickly using the method of [exponentiation by squaring](http://en.wikipedia.org/wiki/exponentiating_by_squaring). Bob then transmits *c* to Alice.

### Decrypting messages

Alice receives *c* from Bob, and knows her private key *d*. She can recover *n* from *c* by the following procedure:

 c^d \equiv n\ (\mathrm{mod}\ N) 

Alice can then extract *n*, since *n* < *N*. Given *n*, she can recover the original message *m*.

The decryption procedure works because

 c^d \equiv n^{e \cdot d}\ (\mathrm{mod}\ N)

and *ed* ≡ 1 (mod *p*-1) and *ed* ≡ 1 (mod *q*-1). Fermat's little theorem yields

 n^{e \cdot d} \equiv n\ (\mathrm{mod}\ p)     and      n^{e \cdot d} \equiv n\ (\mathrm{mod}\ q) 

which implies (as *p* and *q* are *different* prime numbers)

 n^{e \cdot d} \equiv n\ (\mathrm{mod}\ pq) 

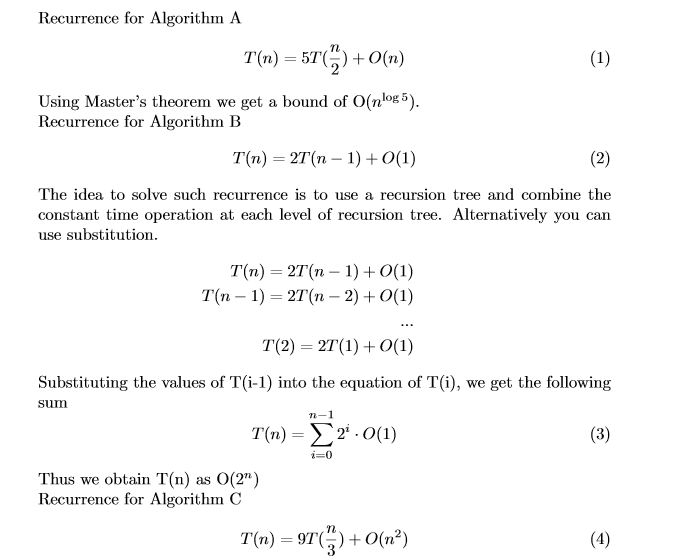
### Signing Messages

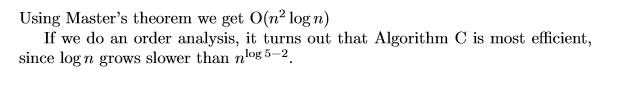
RSA can also be used to sign a message. Suppose Alice wishes to send a signed message to Bob. She produces a hash value of the message, encrypts it with her secret key, and attaches it as a "signature" to the message. This signature can only be decrypted with her public key. When Bob receives the signed message, he decrypts the signature with Alice's public key, and compares the resulting hash value with the message's actual hash value. If the two agree, he knows that the author of the message was in possession of Alice's secret key, and that the message has not been tampered with since.

## Security

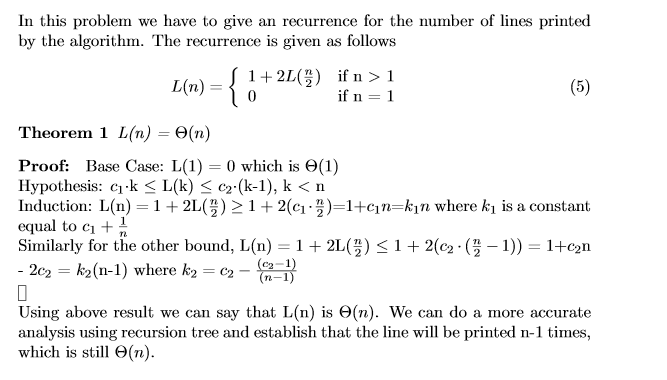
Suppose Eve, an eavesdropper, intercepts the public key *N* and *e*, and the ciphertext *c*. However, she is unable to directly obtain *d*, which Alice keeps secret. The most obvious way for Eve to deduce *n* from *c* is to factor *N* into *p* and *q*, in order to compute (*p*-1)(*q*-1) which allows the determination of *d* from *e*. No polynomial-time method for factoring large integers on a classical computer has yet been found, but it has not been proven that none exists.

http://en.wikibooks.org/wiki/Cryptography/RSA

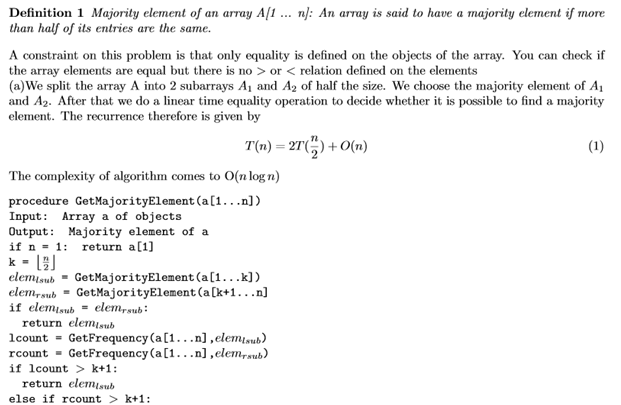
1. 2.4) I found this answer at <http://www.ece.northwestern.edu/~dda902/336/hw3-sol.pdf>. 

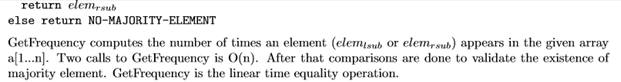


1. 2.12) I found this answer at <http://www.ece.northwestern.edu/~dda902/336/hw3-sol.pdf>.



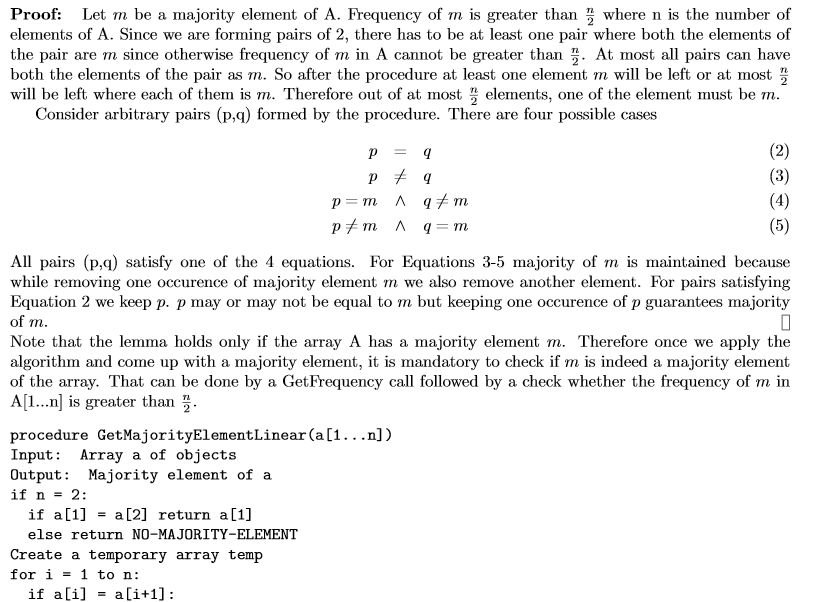
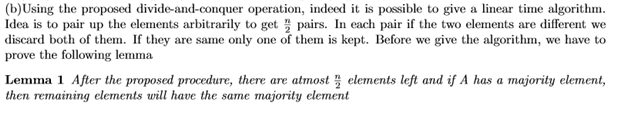
1. I found this answer at <http://www.ece.northwestern.edu/~dda902/336/hw4-sol.pdf>.

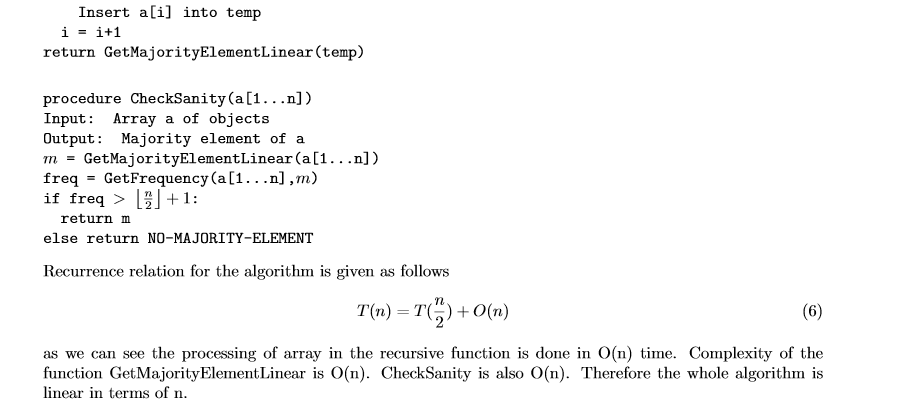
10a) 2.23a) 



10b)

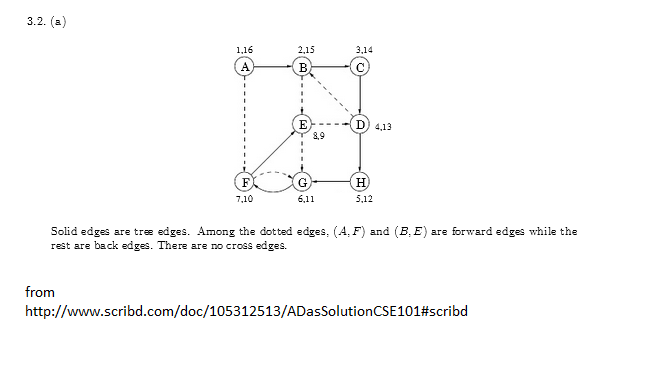
2.23b)







3.2a)





3.8)

