Final Project

Isaac Schaal - CS 144

Question 1

Can you conclude that $A = B$ *if* $A, B,$ *and* C *are sets such that:*

a. $A \cup C = B \cup C$?
b. $A \cap C = B \cap C$?
c. $A \cup C = B \cup C$ *and* $A \cap C = B \cap C$?

a)

Given $A \cup C = B \cup C$, we can't conclude that $A = B$. We will show this by providing a counterexample. Let $A = \{1, 2\}$, $B = \{1\}$, and $C = \{2, 3\}$. $A \cup C = \{1, 2, 3\} = B \cup C$. However, $A \neq B$.

b)

Given $A \cap C = B \cap C$, we can't conclude that $A = B$. We will show this by providing a counterexample. Let $A = \{1, 2, 3\}$, $B = \{3, 4, 5\}$, and $C = \{3, 6, 7\}$. $A \cap C = \{3\} = B \cap C$. However, $A \neq B$.

c)

Given $A \cup C = B \cup C$ and $A \cap C = B \cap C$, we can conclude that $A = B$. In order to prove that $A = B$, we must prove that $A \subseteq B$ and $B \subseteq A$. We will first prove that $A \subseteq B$. For any element $x \in A$, there are two cases, either $x \in C$ or $x \notin C$. If $x \in C$, then $x \in A \cap C = B \cap C$, so $x \in B$. Now consider $x \notin C$. Since $x \in A$, we have $x \in A \cup C = B \cup C$. Since $x \notin C$ and $x \in B \cup C$, we have $x \in B$. Thus, for any $x \in A$, we have $x \in B$, which means that $A \subseteq B$. The proof that $B \subseteq A$ procedes in exactly the same way.

Extension

This question is an example of adding to our toolbox of set theory. If we are trying to prove set equality, we previously had to show that $A \subseteq B$ and $B \subseteq A$. However, we have just proved that if $A \cup C = B \cup C$ and $A \cap C = B \cap C$, we can conclude that $A = B$. If we are given two sets, A and B, and we know that $A \cup C = B \cup C$, we only have to prove that $A \cap C = B \cap C$ in order to prove set equality (and vice versa). Other similar problems in set theory give us similar tools, where we can use “shortcuts” to solve further problems.

In order to make this question more difficult, we can pose the following generalization.

Can you conclude that $A_1 = A_2 = \ ... \ = A_n$ if $A_1, A_2, ... A_n$ and C are sets such that:

d. $A_1 \cup C = A_2 \cup C = ... = A_n \cup C$?
e. $A_1 \cap C = A_2 \cap C = ... = A_n \cap C$?
f. $A_1 \cup C = A_2 \cup C = ... = A_n \cup C$ *and* $A_1 \cap C = A_2 \cap C = ... = A_n \cap C$?

d)

We can not conclude that $A_1 = A_2 = \ ... \ = A_n$ given $A_1 \cup C = A_2 \cup C = ... = A_n \cup C$. Let C be the universe. No matter what $A_1, A_2, ... A_n$ are, $A_1 \cup C = A_2 \cup C = ... = A_n \cup C$ will be equal, as C is the universe and any $A_i \cup C = C$. Thus, we can learn nothing about the equality of $A_1, A_2, ... A_n$.

e)

We can not conclude that $A_1 = A_2 = \ ... \ = A_n$ given $A_1 \cap C = A_2 \cap C = ... = A_n \cap C$. Let C be the empty set. No matter what $A_1, A_2, ... A_n$ are, $A_1 \cap C = A_2 \cap C = ... = A_n \cap C$ will be equal, as C is the empty set and any $A_i \cap C = C$. Thus, we can learn nothing about the equality of $A_1, A_2, ... A_n$.

f)

By applying our proof from **Question 1c)** multiple times, we can prove that $A_1 = A_2 = \ ... \ = A_n$ given that $A_1 \cup C = A_2 \cup C = ... = A_n \cup C$ *and* $A_1 \cap C = A_2 \cap C = ... = A_n \cap C$. For all possible pairs of A_i and A_j, we know that $A_i \cup C = A_j \cup C$ and $A_i \cap C = A_j \cap C$, and thus that $A_i = A_j$. This leads us to conclude that $A_1 = A_2 = \ ... \ = A_n$.

Question 2

*Data are transmitted over the Internet in datagrams, which are structured blocks of bits. Each datagram contains header information organized into a maximum of **14** different fields (specifying many things, including the source and destination addresses) and a data area that contains the actual data that are transmitted. One of the 14 header fields is the header length field (denoted by **HLEN**), which is specified by the protocol to be 4 bits long and that specifies the header length in terms of the 32 bit blocks of bits. For example, if HLEN = 0110, the header is made up of six 32-bit blocks. Another of the 14 header fields is the 16-bit long **total length field **(denoted by **TOTAL LENGTH)**, which specifies the length in bits of the entire datagram, including both the header fields and the data area. The length of the data area is the total length of the datagram minus the length of the header.*

*a.** **The largest possible value of TOTAL LENGTH (which is 16 bits long) determines the maximum total length in octets (block of 8 bits) of an Internet datagram. What is this value?*

*b. ** **The largest possible value of HLEN (which is 4 bits long) determines the maximum total header length in 32-bit blocks. What is this value? What is the maximum total header length in octets?*

*c.** **The minimum (and most common) header length is 20 octets. What is the maximum total length in octets of the data area of an Internet datagram?*

*d.** **How many different strings of octets in the data area can be transmitted if the header length is 20 octets and the total length is as long as possible?*

a)

TOTAL LENGTH is 16 bits long. The largest possible value would be given by the 16 bit string of all 1s, which has a value of $2^0 + 2^1 + 2^2 + \dots + 2^{15} = 65535.$ This means that the longest total length of an Internet datagram is 65535 octets.

b)

HLEN is 4 bits long. The largest possible value would be given by the 4 bit string of all 1s, which has a value of $2^0 + 2^1 + 2^2 + 2^3 = 15.$ This means that the maximum total header length is 15 32-bit blocks. Accordingly, the maximum total header length in octets is $(15 \times 32)/8 = 60.$

c)

The maximum total length of the data area is equal to the largest possible value of TOTAL LENGTH - the minimum possible header length. Thus, the maximum total length is $65535 - 20 = 65515$ octets.

d)

The maximum total length of the data area is 65515 octets. There are 2 choices for each of the 8 bits in the octet, so the total number of octets is $2^8 = 256$. Thus, for each of the 65515 octets, there are 256 options. Thus, there are $256^{65515} = 6.94 \times 10^{157775}$ possible strings of octets that can be transmitted in the data area.

Question 3

a. *Suppose that* $\lim_{x\to c} f(x)$ *exists. Prove that there exists a constant* M *and a* $\delta > 0$ *such that* $|f(x)| < M$ *for* $0 < |x - c| < \delta$.
b. *Give an explicit example that illustrates the above statement. You should give an actual* f, c, M, *and* δ. *(Also include a well-labeled illustrative diagram.)*

a)

We suppose that $\lim_{x\to c} f(x)$ exists. Assume that $\lim_{x\to c} f(x) = L$. This means that for every $\varepsilon > 0$, there exists a δ such that if $0 < |x - c| < \delta$, then $f(x) \in (L - \varepsilon, L + \varepsilon)$. Let $\varepsilon = 1$. Then, we know that there exists a δ such that if $0 < |x - c| < \delta$, we know that $f(x) \in (L - 1, L + 1)$. Let $M = |L| + 1$. Thus, $L + 1 \le M$ and $L - 1 \ge -M$. Thus, we know that $f(x) \in (-M, M)$ which means that $|f(x)| < M$.

b)

Let $f(x) = 2x + 1$ and $c = 1$. The $\lim_{x\to 1} f(x) = 3 = L$. We let $M = |L| + 1 = 4$. Let $\varepsilon = 1$ and $\delta = 1/2$. If $0 < |x - 1| < 1/2$, then $f(x) \in (L - \varepsilon, L + \varepsilon) = (3 - 1, 3 + 1) = (2, 4)$. So, if $0 < |x - 1| < 1/2$, then $|f(x)| < 4 = M$.

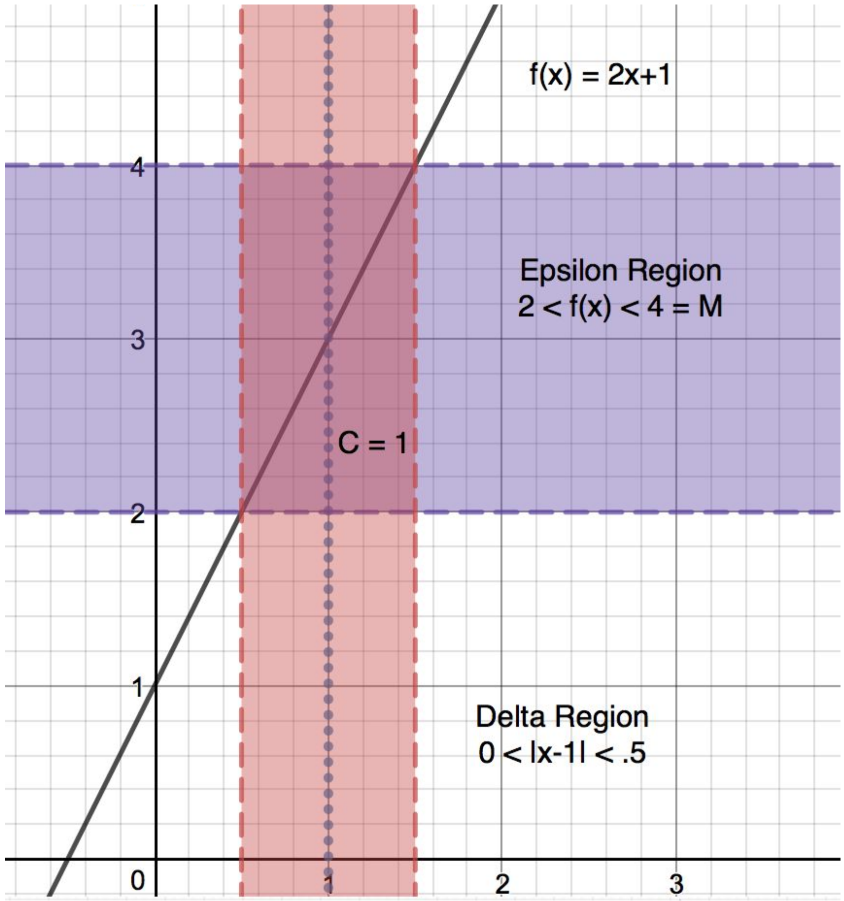

Figure 1: Epsilon and Delta regions

Question 4

(#algebra) Classify ALL groups of order 6 using the tools we have learned in this class. It is up to you to define what "classify" means, but it does NOT mean listing just the examples that we have seen in class. Hint: Break this problem up into cases. You must justify all your conclusions. You may use Sage if you think it is helpful. Note also that if you try to do background reading from other sources, you may encounter a lot of unfamiliar terminology! This is not necessary to do a thorough job on this task.

There are 2 groups of order 6. There can be multiple ways to represent them, but all representations can be classified into two cases. There is an abelian group of order 6, of which all representations are isomorphisms of the cyclic group Z_6, and there is a non abelian group of order 6, of which all representations are isomorphisms of the symmetric group S_3. In order to understand this, we can examine the cayley table for both groups. We can use the following code in Sage to create the below tables.

```
A = CyclicPermutationGroup(6)
print A.cayley_table()
B = SymmetricGroup(3)
print B.cayley_table()
```

Cayley Table for Z_6

*	a	b	c	d	e	f
a	a	b	c	d	e	f
b	b	c	d	e	f	a
c	c	d	e	f	a	b
d	d	e	f	a	b	c
e	e	f	a	b	c	d
f	f	a	b	c	d	e

Cayley Table for S_3

*	a	b	c	d	e	f
a	a	b	c	d	e	f
b	b	a	d	c	f	e
c	c	f	e	b	a	d
d	d	e	f	a	b	c
e	e	d	a	f	c	b
f	f	c	b	e	d	a

Figure 2: Cayley Tables of groups of order 6

We can first see that for any two elements α, β in the Cayley Table for Z_6, $\alpha * \beta = \beta * \alpha$, and thus the group is abelian. For the Cayley Table for S_3, on the other hand, we can see that $b * c = f \neq e = c * b$, and thus the group is non abelian.

As an example, we can choose a group of order 6 that is not represented by Z_6 or S_3 and show that it essentially the same as one of them. For two groups to be essentially the same, there is a bijection between the elements of the two groups such that both representations would have the same Cayley Table.

From **Corollary 6.11** in the Judson text, we know that the order of any element in a group of 6 elements must be 1,2,3 or 6, as they are the divisors of 6. If an element has order 6, it can be mapped to Z_6, and if there are no elements of order 6 in the group, it can be mapped to S_3.

We will choose the dihedral group D_3. This group is the symmetries of a triangle, where the operators are a flip f and a rotation to the right r. There are 6 elements, the identity i, one flip f, one rotation r, a flip and a rotation f, two rotations r^2, and a flip and two rotations, fr^2. Note that three rotations or two flips is the same as the identity.

We can observe that this group is non abelian, as $f * r = fr \neq fr^2 = r * f$. This means that a rotation followed by a flip is not the same as a flip followed by a rotation.

By running the following Sage code,

```
C = DihedralGroup(3)
print C.cayley_table(["i", "f", "r", "fr", "r^2", "fr^2"])
```

We get the Cayley Table:

```
    *      i     f     r    fr   r^2  fr^2
     +------------------------------------
    i|     i     f     r    fr   r^2  fr^2
    f|     f     i    fr     r  fr^2   r^2
    r|     r  fr^2   r^2     f     i    fr
   fr|    fr   r^2  fr^2     i     f     r
  r^2|   r^2    fr     i  fr^2     r     f
 fr^2|  fr^2     r     f   r^2    fr     i
```

If we use the map $i \leftrightarrow a$, $f \leftrightarrow b$, $r \leftrightarrow c$, $fr \leftrightarrow d$, $r^2 \leftrightarrow e$, $fr^2 \leftrightarrow f$, we see that the Cayley Tables are the same.

We could do this same process for all other representations of groups of order 6, and could create a bijection between the representation and either Z_6 or S_3.

Question 5

In this task, you demonstrate how to encode and decode your first and last name in the RSA cryptosystem by following the instructions in Judson, section 7.7. In particular, complete Judson 7.7: 1-4, except instead of sending the message “Math,” send your first and last name. Be sure to explain how you deal with blocks of characters. For 4, you can use your name again. (Special note: Instead of using primes as specified in Judson, use two 200 digit primes that you find yourself. You must show how you found them, e.g., show any code you construct to find them.)

This problem is detailed in the following 10 pages of code, which was converted from a Jupyter Notebook running Sage.

In [1]:

```
## CONSTRUCTING A KEYPAIR FOR ALICE
# We use the random_prime function to create a random prime
# with 200 digits.
# The random_prime generates a prime between cutoff values
# which I chose specifically to generate a 200 digit number.
p_a = random_prime(10^200-1,False,10^199)

# We check that it is prime and that it has 200 digits
if is_prime(p_a) and len(str(p_a)) == 200:
    print"p_a is a prime number of 200 digits, it is: \n", p_a

q_a = random_prime(10^200-1,False,10^199)

# We check that it is prime, that it has 200 digits, and that
# it is not the same as p_a
if is_prime(q_a) and len(str(q_a)) == 200 and q_a != p_a:
    print"\np_a is a prime number of 200 digits, it is: \n", q_a



n_a = p_a * q_a
#We can't use m_a = euler_phi(n_a) because our primes are to large
#and as such, more secure. Instead, we use the below formula.

m_a = (p_a -1)*(q_a-1)

print "\nm_a is : \n", m_a
```

```
p_a is a prime number of 200 digits, it is:
38092642388946735939879406752157474086626160180141790937970175342361855061566934228749845832299275016063752664803987444314097957986619505834237351512085135112198721704117498693487188021198066063134271

p_a is a prime number of 200 digits, it is:
260280396892316373785924638998604796130912616544907579428314887458991904145609378945494689597855618700966103345414053928237178011612735436559348945178671363723020640049485848117826955708940476084100061

m_a is :
991476807967213097776819710379447353183689037716621776995074068209407027999484394055397917847073151658181995382771717328026206715830589346746773470316892643937864565058510336888942754644722081198566205315034104971371292471698233644100810932926747094015690860935524181912214731849772616384164274055329958937048304452460346178498954019321468051477705979970277071675375119005244022127550046174998756200
```

In [2]:

```
#It is not possible to factor m_a, and thus
# we must find E_a by randomally choosing numbers
# and checking if the gcd is = 1. Note that E is
# not neccesarily (and probably not) the lowest possible E.
E_a = 0
while gcd(E_a, m_a) != 1:
    E_a = ZZ.random_element(m_a)
```

In [3]:

```
D_a = inverse_mod(E_a, m_a)
```

In [4]:

```
print "These are the values for Alice \n"
print " n = ", n_a
print " E = ", E_a
print " D = ", D_a
```

```
These are the values for Alice

 n =  9914768079672130977768197103794473531836890377166217769950740 6
820940702799948439405539791784707315165818199538277171732802620671 58
305893467467734703168926439378645650585103368889427546447220811985 66
269435716183149744610943568885662054510650348581726564571662599612 44
295769085972189591569895635889221611930004764984529748399425810191 23
7095822372373593224176157246108418508874929201114213828867030053 1
 E =  545931369018071867073047198359368863581208142720905994126909 45
736132282625885103678176736539237234858883365778510641178904382542 04
257583596381121790523733247926960519065102674171975435679633359917 24
513760838310612611700079615098815455534685540615499743998932354236 18
029845809620686593016678295635707754269827843406603261502449610116 47
3946791855509917729999564301461440527200455854344479557510115400 7
 D =  566294748703939700110295901603217122680365772767012980526784 94
223359528861720513351725630893099092894353666782970914960402715242 39
766458571841607633076418249016576309527446120826517648979193962063 58
186351609394473796376465783744047967885231113050813045833442736707 54
014982258170215002860837817510114427827432585885097367012555068286 94
3179913750267340462728656583958617964080563679646506997440699914 3
```

In [5]:

```
print """We can assert that the encryption and decryption keys are multiplicativ
e inverses..."""
if inverse_mod(E_a, m_a) == D_a and inverse_mod(D_a, m_a) == E_a:
    print "They are multiplicative inverses"
```

```
We can assert that the encryption and decryption keys are multiplica
tive inverses...
They are multiplicative inverses
```

In [6]:

```
## NOW FOR BOB
p_b = random_prime(10^200-1,False,10^199)

if is_prime(p_b) and len(str(p_b)) == 200:
    print"p_b is a prime number of 200 digits, it is: \n", p_b

q_b = random_prime(10^200-1,False,10^199)

if is_prime(q_b) and len(str(q_b)) == 200 and q_b != p_b:
    print"\np_b is a prime number of 200 digits, it is: \n", q_b



n_b = p_b * q_b
m_b = (p_b -1)*(q_b-1)
E_b = 0
while gcd(E_b, m_b) != 1:
    E_b = ZZ.random_element(m_b)
D_b = inverse_mod(E_b, m_b)
print "\n These are the values for Bob \n"
print " n = ", n_b
print " E = ", E_b
print " D = ", D_b
print """\nWe can assert that the encryption and decryption keys are multiplicat
ive inverses..."""
if inverse_mod(E_b, m_b) == D_b and inverse_mod(D_b, m_b) == E_b:
    print "They are multiplicative inverses"
```

```
p_b is a prime number of 200 digits, it is:
48300024253958051679091274870598542327015093633381379473315222422414
33273921719077552238101930883346212194345031900067864404935969145963
1609607201616769388870124453108533815515886384216748693300734819

p_b is a prime number of 200 digits, it is:
20547157568939371780290655252793604200664164763754120123115895297750
13912308054424170157659135080672576605115372794626085257809807481324
7168573055631244405587374828387036171059603796088461190579548621

 These are the values for Bob

 n =  99242820892966941528409779863077841097029738263330163534249854
19445555599260174946501533967046574331244042600490855371483778472305
25077140900774008171932565432617437869594270385420196322801374437990
97083423883092914922204156831378016887648304785656320517847804367203
41105054355925064036133888383476532497172682772590791733407354876013
63240379865911350812085344306297612778707593824269167412638134599
 E =  81281740855071145590232591513700767207565638549445012177148478
69769130178607820751601255889017928164699193663951289143936383091250
32019534700020742707159998325054674600517041273422798579760679322483
43489908735128686090213165829522890175177985011391707774756870970347
23576524342172399256484732430752187124870399594476829035325526140764
88049036971217050108634401370104443606735515454920512321090854613
 D =  26209001717422089132303442523752315053049630583401150954590294
57107513078254715243153857652774702884890571208485204793720090745185
70847843532056274596261293145297138428033105847625732980161013163151
61313867018118284399004684971268709884662253720785124456780365066956
43764398237756468309681664088723695222950672869496454888407072123621
8406260546125186023349198419146084177171278544823131976935958157

We can assert that the encryption and decryption keys are multiplica
tive inverses...
They are multiplicative inverses
```

In [7]:

```
### ENCODING MY FIRST AND LAST NAME
# I now encode "Isaac Schaal"...
# however, I know I need to send strings that
# are divisible by 4. I thus will encode and send
# the string "IsaacxSchaal", with the "x" representing
# a space. I will then break this string into 3 strings
# of length 4.

def split_by_len(text, chunksize):
    return [text[i*chunksize:(i*chunksize+chunksize)] for i in range(len(text)/c
hunksize)]
string = "IsaacxSchaal"
string = split_by_len(string, 4)
print string
```

```
['Isaa', 'cxSc', 'haal']
```

In [8]:

```
# We now must encrypt the message
digits_list = []
messages = []
for word in string:
    digits_list.append([ord(letter) for letter in word])
for digits in digits_list:
    messages.append(ZZ(digits, 128))
print "The message is :", messages
```

```
The message is : [205027785, 208993379, 228094184]
```

In [9]:

```
signed_list = []
for message in messages:
    signed_list.append(power_mod( message, D_a, n_a))
print "The signed message is : ", signed_list
```

```
The signed message is :  [254164326502664313598874651600280660381579
22194211070839961729895792864409699002015416580841519456565600599793
93612715104625515791470952269676047158579189228307913858599527924777
72925872727837100434345157878025258843808093072140162691277758533449
41365731521877791330952422685819779099683015129261070441220048572568
71119074984076957512963068361424731625698420264601552205926969517749
61702043069714825, 7556741169340472409842847444709190329052688910565
81713296269843936827734798687325927792555989207389803895891878350281
24778000746427622135420326600721917116903378692599261361215559480134
24478613058180577392185195301112651878800619638997840421506390574598
84006513105632024786321758375335174805280477178253919823940897454929
24480411417488593655385997615568956268639618572238526310652993622781
0032224443, 45083820833899571128426451969546685057129008985711515893
20535283005725514625109353754520146150791554867604775995974698515941
90319029670142152867757019540281450892262397271081680470925673916838
24372612459120960284314567986340235015063042876430332758499750545169
40870667628973354650391034574750615582778496225803332489198708168760
88619298725790695222596159490394562330116731666661051426925465166270
707]
```

In [10]:

```
encrypted_list = []
for signed in signed_list:
    encrypted_list.append(power_mod(signed, E_b, n_b))
print "The encrypted message is : ", encrypted_list
```

```
The encrypted message is :  [401925809566064050975869074026045151585
71247373816008087902749957313912351672416212774923470251100789867741
26191994305512944509959424819398019102004143510668352798947269690328
82496948452039090524414195689662078577705066613738191386755150665051
23141742759585808482987268260308066408993008902631729802014138399698
95554829732676084874935174770449759100784099367392127318380750099802
71352418905441508646, 2279108591140747084730939355803877498618107901
65432137759405460604085920203347777420077491377621512983144556750801
37191914108361566971582597881059905794281055181212640068710148090972
49068367311628093761492043774696339724514675172965788986463657008999
85683287647661240034654559269266517283866465170078918969970326320798
33117681907924279670521556129897793063660715080170060169735412042552
7681121210277, 94711587906195612238276766839171629325828761250654544
32029216499517218527178759727199490504567555335334436653549039832454
03134086475287473362471219812178490377358376276846489086131819205774
59980103858796806299961301662685245624409301060610054144856490844945
03580939696425132188853356405088734949455233050469813789789390721170
39347751815107505577113226369155693884463075572309864541114056550123
75055]
```

In [11]:

```
### DECODING
# The message is sent to Bob. He decodes it.
decrypted_list = []
for encrypted in encrypted_list:
    decrypted_list.append(power_mod(encrypted, D_b, n_b))
print "The decrypted message is : ",decrypted_list
received_list = []
for decrypted in decrypted_list:
    received_list.append(power_mod(decrypted, E_a, n_a))
print "The unsigned message (from Alice) is : ", received_list
digits_list = []
for received in received_list:
    digits_list.append(received.digits(base=128))
letters_list = []
for digits in digits_list:
    letters_list.append([chr(ascii) for ascii in digits])
lis = []
for letter in letters_list:
    lis.append("".join(letter))
answer = ''.join(lis)
print "The final answer is : ", answer
```

```
The decrypted message is :  [254164326502664313598874651600280660381
57922194211070839961729895792864409699002015416580841519456565600599
79393612715104625515791470952269676047158579189228307913858599527924
77772925872727837100434345157878025258843808093072140162691277758533
44941365731521877791330952422685819779099683015129261070441220048572
56871119074984076957512963068361424731625698420264601552205926969517
74961702043069714825, 7556741169340472409842847444709190329052688910
56581713296269843936827734798687325927792555989207389803895891878350
28124778000746427622135420326600721917116903378692599261361215559480
13424478613058180577392185195301112651878800619638997840421506390574
59884006513105632024786321758375335174805280477178253919823940897454
92924480411417488593655385997615568956268639618572238526310652993622
7810032224443, 45083820833899571128426451969546685057129008985711515
89320535283005725514625109353754520146150791554867604775995974698515
94190319029670142152867757019540281450892262397271081680470925673916
83824372612459120960284314567986340235015063042876430332758499750545
16940870667628973354650391034574750615582778496225803332489198708168
76088619298725790695222596159490394562330116731666661051426925465166
270707]
The unsigned message (from Alice) is :  [205027785, 208993379, 22809
4184]
The final answer is :  IsaacxSchaal
```

In [12]:

```
### TAMPERED MESSAGE
# We first display the sent message again.
print "The encrypted message is : ", encrypted_list
```

```
The encrypted message is :  [401925809566064050975869074026045151585712473738160080879027499573139123516724162127749234702511007898677412619199430551294450995942481939801910200414351066835279894726969032882496948452039090524414195689662078577705066613738191386755150665051231417427595858084829872682603080664089930089026317298020141383996989555482973267608487493517477044975910078409936739212731838075009980271352418905441508646, 227910859114074708473093935580387749861810790165432137759405460604085920203347777420077491377621512983144556750801371919141083615669715825978810599057942810551812126400687101480909724906836731162809376149204377469633972451467517296578898646365700899985683287647661240034654559269266517283866465170078918969970326320798331176819079242796705215561298977930636607150801700601697354120425527681121210277, 94711587906195612238276766839171629325828761250654544320292164995172185271787597271994905045675553353344366535490398324540313408647528747336247121981217849037735837627684648908613181920577459980103858796806299961301662685245624409301060610054144856490844945035809396964251321888533564050887349494552330504698137897893907211703934775181510750557711322636915569388446307557230986454111405655012375055]
```

In [13]:

```
#We then add 1 to each segment of the encrypted message
# to emulate it being tampered with.
tampered_list = []
for i in range(len(encrypted_list)):
    tampered_list.append(encrypted_list[i]+1)
print "The tampered message is : ", tampered_list
```

```
The tampered message is :  [40192580956606405097586907402604515158571
247373816008087902749957313912351672416212774923470251100789867741261
919943055129445099594248193980191020041435106683527989472696903288
249694845203909052441419568966207857770506661373819138675515066505123
1417427595858084829872682603080664089930089026317298020141383996989
5554829732676084874935174770449759100784099367392127318380750099802713
52418905441508647, 22791085911407470847309393558038774986181079016
54321377594054606040859202033477774200774913776215129831445567508013
71919141083615669715825978810599057942810551812126400687101480909724
90683673116280937614920437746963397245146751729657889864636570089998
56832876476612400346545592692665172838664651700789189699703263207983
31176819079242796705215561298977930636607150801700601697354120425527
681121210278, 94711587906195612238276766839171629325828761250654544320
29216499517218527178759727199490504567555335334436653549039832454031
34086475287473362471219812178490377358376276846489086131819205774599
80103858796806299961301662685245624409301060610054144856490844945035
809396964251321888533564050887349494552330504698137897893907211703934
77518151075055771132263691556938844630755723098645411140565501237
5056]
```

In [14]:

```
# We now see what happens if Bob trys to decrypt the message
decrypted_list = []
for tampered in tampered_list:
    decrypted_list.append(power_mod(tampered, D_b, n_b))
received_list = []
for decrypted in decrypted_list:
    received_list.append(power_mod(decrypted, E_a, n_a))
digits_list = []
for received in received_list:
    digits_list.append(received.digits(base=128))
letters_list = []
for digits in digits_list:
    letters_list.append([chr(ascii) for ascii in digits])
lis = []
for letter in letters_list:
    lis.append("".join(letter))
answer = ''.join(lis)
print "The final answer is : ", answer
```

```
The final answer is :  h_h"VuxeJ0{;|$[ngp<(m?7_]?P{JF{_bAq3?i0I<gZRm
sOSEe{#B[>'- 5
{3Ruw?##2W!b`$)6kC^AF
IMnR9K<hwnQrD(v{B1%5*qIpKPk*Trp\tw.{gz"J[<iV$1}`MZw
uyPCPskYudO7C0jPe*N"xm1p@5B=Bf|,I/=_w)4CTgD,r:
;Bq%Q} -Lw-TSi ,b?xlGW0}4s,=&D\HyXV-,b
.HNwRT
i6TD<PPhxJV"bl#tpArjou2=::oUDymji lm(@
Q<i%iw+j;v^WU<L-~Qt%sAWC:iGlL|=(Z
@wrfE_~u<,E(]2E48Y
```

In [1]:

```
#This is cleary gibberish, and shows us how sensitive
#RSA is to small amounts of tampering with the message
```