SAGE CRYPTOLOGY LABS

CHRISTOPHER DAVIS
UC IRVINE

CONTENTS

INTRODUCTION

In these labs, we will be working with Sage. Sage is computer algebra software developed by William Stein at the University of Washington. It is a free alternative to Mathematica, Maple, Matlab, and Magma. It is also *open source*, which means that if you want to see the code behind a function, you can, and if you want to edit that code, you can. You can also submit your own code in the hopes that it will become part of the standard distribution. The programming is done using the Python language.

Date: March 8, 2013.

Getting started. You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

`http://bduc-claw9.oit.uci.edu/`

From off-campus, proceed to

`http://uci.sagenb.org/`

or else proceed to the first site above using VPN here:

`https://vpn.nacs.uci.edu/`

Alternatively, if you already have Sage installed on your own machine, you can open it up and type notebook() at the prompt.

We will need some special functions defined in an external file, and getting Sage to recognize these functions is a three step process.

a. Go to `http://www.math.uci.edu/~davis` and download "Fall2012Code.sage", keeping the name the same, including the suffix ".sage". Don't save it as a text file.

b. Back in the Sage worksheet, find the "Data" menu at the top, and choose "Upload or create file". On the next screen, upload the file you just downloaded from our website. I always get an error message, but if you get back to the worksheet screen (and perhaps refresh), it seems to work.

c. Finally, we need to tell the worksheet we want to use that data. Use the command

```
load DATA+"Fall2012Code.sage"
```

Acknowledgments. Thanks to Laurie Andress-Delaney, Adam Brenner, Dennis Eichhorn, Sarah Eichhorn, Kiran Kedlaya, Harry Mangalam, Tommy Occhipinti, Tristan Occhipinti, William Stein, Heng Su and the UC Irvine Math 173 students for many forms of assistance.

1. WARM-UP EXERCISES

Before we begin with cryptology, we make some calculations using the Sage notebook. You should be able to copy and paste directly from this document into Sage. If extra indentations (whitespace at the beginning of a line) appear, they can be removed by highlighting the passage and pressing the tab key while holding down shift.

1.1. To perform a computation, you hit enter while holding down the shift key. Some possible computations:

```
3+10
factor(36)
3/10
3/10.
```

1.2. Here's a more impressive computation:

```
factorial(10^6)
```

The only problem is that the big output is annoying. You can hide it, by clicking to the left of the output (click there again to hide more), or you can delete the whole thing, by deleting the contents of the cell, and then clicking delete again. (If the cell is huge, it might be helpful to "select all" text in the cell, by hitting control+a.)

1.3. As we saw in class, gcd computations are easy:

```
gcd(10^163000+2,10^15700+5)
```

Wow, how long did that take?

```
time gcd(10^163000+2,10^15700+5)
```

You can also perform the extended Euclidean algorithm:

```
xgcd(100,71)
```

What did that just do? For more info, type:

```
help(xgcd)
```

1.4. You can also get help with the homework. For example

```
(0+1+2+3+4+5+6)%7
```

or

```
total = 0
for i in range(0,7):
    total = total + i
print total%7
```

Note that range(0,7) might not be what you think. Try:

```
for i in range(0,7):
    print i
```

1.5. Especially in the early labs, it will be important to work with *strings*. For example, try

```
X = "hello there"
for ch in X:
    print ch
```

or

```
X[1]
```

Note that this isn't the first letter of X, but the second, because we start counting from 0.

```
X[1:4]
```

Note that this returns only three letters, because the final number in the range isn't counted.

2. SHIFT CIPHER

2.1. To check that things are working as they should, try

```
shiftencrypt("hello there!")
```

The phrase should have had all spaces and punctuation removed, and then each letter should have been shifted by a random amount. Sage always reads what's between the quotation marks as a string. Another way to do the same thing would be:

```
X = "hello there!"
shiftencrypt(X)
```

Note that the X should not be in quotation marks, because it is not a string, it is instead a variable name. Note also you'll get a different answer each time you do this encryption, because the shift amount is random. Also, note, as will be a theme, the ciphertext is in all capital letters, to help distinguish it from plaintext.

2.2. Of course, it's not very fun to try to decipher ciphertext if you already know the plaintext. It's much more difficult if you try it with a random passage:

```
randomshift()
```

This returns a string, but copying and pasting that string will get cumbersome. So instead we should name the string something, like:

```
X = randomshift()
```

This will be a new random passage. If you want to see what it looks like, evaluate the single letter: X (or whatever you named the string).

2.3. Shift ciphers are easy to break, provided the text is long enough. Of course, if the text is too short, it may be impossible. For example, ciphertext "RS" might correspond to the plaintext "cd", or the plaintext "hi", or to the plaintext "mn", the postal code for the great state of Minnesota. If the text is long enough, though, we can use frequency analysis. To find out how often each character occurs, try:

```
countsubstrings(X,1)
```

or, if you'd rather count bigram frequencies:

```
countsubstrings(X,2)
```

If we're lucky enough, the most commonly occurring letter will correspond to "e", and the most commonly occurring bigram will correspond to "th".

2.4. As it is, though, it's already unpleasant to figure out what the most commonly occurring letter is. For this purpose, we'd rather have the output returned, not in alphabetical order, but in order of most commonly occurring strings:

```
ranksubstrings(X,1)
```

By default, this only returns the first 20 substrings. To return, say, the 35 most commonly occurring substrings of length 2, we could evaluate

```
ranksubstrings(X,2,35)
```

Say the most commonly occurring ciphertext letter is "B". We might then hope that this corresponds to plaintext "e". To check it, try

```
shiftdecrypt(X,3)
```

If we instead thought that "K" corresponded to "e", we would try

```
shiftdecrypt(X,-6)
```

2.5. That wasn't too hard, but we can make our lives even easier. Get a new random ciphertext from the shift cipher, and try to decrypt it using the breakshiftcipher command:

```
X = randomshift()
breakshiftcipher(X)
```

This will display two frequency charts, one for the average frequency in English, and one for the average frequency in our ciphertext. Choose the shift amount that makes the two distributions look the closest. It's impressive that even in a relatively short random English passage, the frequencies match up so closely.

2.6. An example is shown in Figure 1. If we guess that the most commonly occurring letter is "e", then we should shift each letter down by 4; the resulting letter frequencies are shown in Figure 2. This letter distribution does not look much like the typical English frequencies. For example, "Z" occurs much more often than "A" in this distribution. The shift amount that looks most like English is shown in Figure 3.

2.7. The point of this particular example is that we should consider more than just the most commonly occurring letter. The relative frequencies of other letters should also be considered.

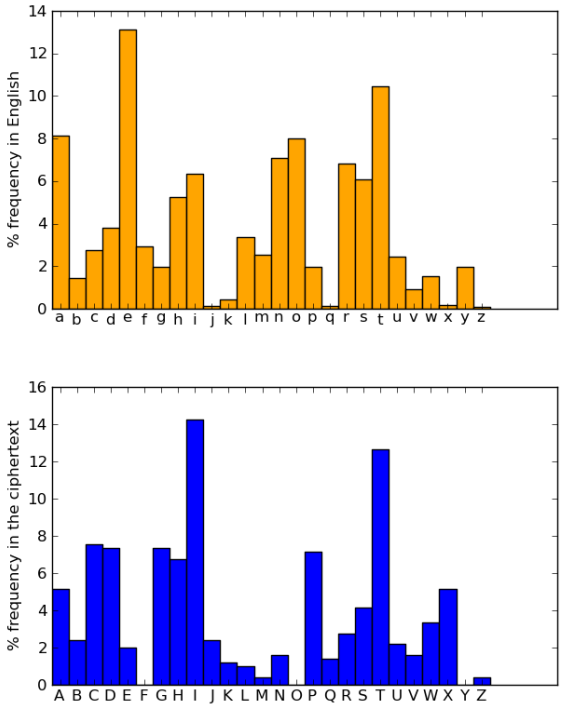


FIGURE 1. Example of the output of the function breakshiftcipher.

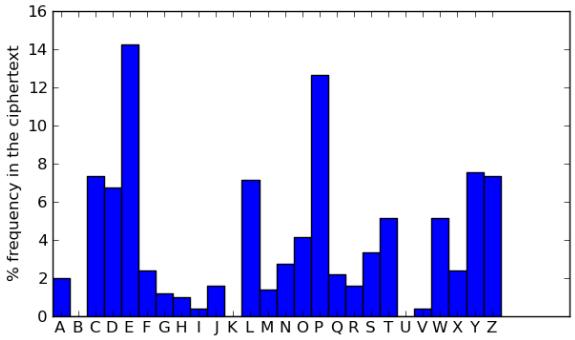


FIGURE 2. The letter frequencies if we assume that “e” occurs most often.

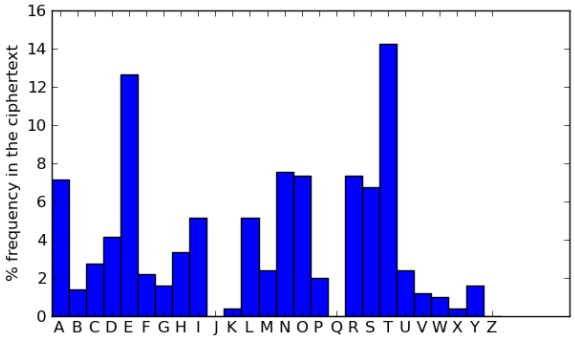


FIGURE 3. The letter frequencies if we assume that “t” occurs most often.

3. VIGENÈRE CIPHER

3.1. In this project, we decipher a random piece of Vigenère ciphertext.

3.2. Now let's try some of the functions designed to work with the Vigenère cipher. For example, the following will reproduce the example from p. 199 in the textbook

```
Y = "The rain in Spain stays mainly in the plain"
vigenereencrypt(Y,"flamingo")
```

If you'd rather not specify the shift amounts, a random key (of random length 5-8) will be chosen:

```
vigenereencrypt("the rain in Spain stays mainly in the plain")
```

(In this case the key will probably be just a sequence of shift amounts, not a meaningful english word.)

3.3. Now get a random piece of Vigenère ciphertext:

```
X = randomvigenere()
```

How should we attempt to decipher it? The first step is to try to determine the keylength. The idea is to look for pieces of ciphertext that occur multiple times. For example, assume the keyword length is 5, and the word "the" appears twice in the text, 30 characters apart. Because 30 is divisible by the keyword length 5, the same shift amount will be used for both t's, and another shift amount will be used for both h's, etc. This will correspond to a piece of ciphertext of length 3 that occurs twice, separated by 30 characters. The following command

```
findgaplengths(X,3)
```

will find all pieces of ciphertext of length 3 that occur multiple times, and it will report the length of the gaps between them. Most of the time, this gap length will be divisible by the keyword length. (Sometimes, the same ciphertext might appear several times just out of coincidence.) If there are too many numbers, you can ask for repetitions of longer pieces of ciphertext:

```
findgaplengths(X,5)
```

Remember that the keyword length for these examples will be one of 5,6,7,8.

3.4. Once you think you know the length of the key, your next task is to determine the exact shifts. Frequency analysis is not immediately useful, because the 0-th character is encrypted using one shift, the 1-st character is encrypted using another shift, etc. Say you think the key length is 7. Then the 7-th character will be encrypted using the same shift as the 0-th character. So will the 14-th character. So we can hope to use frequency analysis on the 0-th, 7-th, 14-th, etc. characters. The command

```
Y = separate(X,0,7)
```

will create a string Y which consists exactly of the characters of X in positions congruent to $0 \bmod 7$. We can now hope to use frequency analysis on this string, because each character in it was enciphered using the same shift.

3.5. Assuming Y is the string consisting of the $0 \bmod 7$ characters of X, we can use the command

```
comparetoenglish(Y)
```

to find the corresponding shift amount.

3.6. Once you have the 0-th shift amount, we need to find 6 more (since our keyword has length 7). Simply repeat the above steps with

```
Y = separate(X,1,7)
```

and so on. For example, if your key length was 6, then the final step will use the command

```
Y = separate(X,5,6)
```

Note that the command Y = separate(X,6,6) would have the same result as Y = separate(X,0,6).

3.7. If at the end we think the shift amounts were 13, -10, 3, 6, 8, 2, -8, in that order, then we check this by running

```
deciphervigenere(X,[13,-10,3,6,8,2,-8])
```

(Note that we move back to the original string X, not the extracted strings Y.) If the result looks like English, then we're done!

4. KNOWN PLAINTEXT ATTACK ON THE SUBSTITUTION CIPHER

(Remember to begin work in a new worksheet! You will need to upload the Week2Code.sage file again.) In this project, we show that the substitution cipher is vulnerable to known plaintext attacks, even if the known plaintext is quite short. (In fact, the substitution cipher is vulnerable without any known plaintext, but the plaintext will make our job easier.)

4.1. There are 26 possible shift keys. If we force our keylength to be 5,6,7, or 8, then there are $26^5 + 26^6 + 26^7 + 26^8$ possible Vigenére cipher keys. (Do you see why?) For the substitution cipher, there are $26!$ possible cipher keys. That corresponds to $403291461126605418404328096$ more keys (computed using Sage!) On the other hand, since plaintext "e" is always encrypted the same way, frequency analysis is more immediately useful than it is with the Vigenère cipher.

4.2. Get a piece of random substitution ciphertext using the command

```
X = randomsubknown()
```

Imagine this is an intercepted message from Bob to Alice, and it is known that this plaintext ends with the signature "your parrot". (This is not so unrealistic. Cryptosystems used by the Nazis in World War II were vulnerable because it became known that the plaintext often ended with the phrase "Heil Hitler".) Playing the role of Eve, we wish to decipher the message.

4.3. We could start by using some of the functions described above:

```
ranksubstrings(X,2)
```

and comparing these to the table on p. 7 of our textbook. There is actually a single function that already does much of this:

```
breaksubcipher(X)
```

This shows the following: All 26 ciphertext characters, listed in order from most common to least common. Then listed again, together with all 26 English letters, listed in decreasing average order of frequency in English. Then the ciphertext is written twice (one of these will update, one will stay fixed). Then are listed the most common bigrams in the ciphertext, along with the most common bigrams in English. Then the same for trigrams. (The trigram "sth" might look strange for a common English trigram, but remember that spaces and punctuation have been removed.)

4.4. Frequency analysis is the main tool which will be used. A lot of relevant data is packaged together in the window created by

```
breaksubcipher(X)
```

This shows the following: All 26 ciphertext characters, listed in order from most common to least common. Then listed again, together with all 26 English letters, listed in decreasing average order of frequency in English. Then the ciphertext is written twice (one of these will update, one will stay fixed). Then are listed the most common bigrams in the ciphertext, along with the most common bigrams in English. Then the same for trigrams. (The trigrams "edt" and "sth" might look strange for common English trigrams, but remember that spaces and punctuation have been removed.)

```
G (G,13.)    J (J,4.1)    N (N,0.82)       (e,13.1) (l,3.4)  (v,0.9)
t (Z,8.8)    p (Y,3.9)    S (S,0.82)       (t,10.5) (f,2.9)  (k,0.4)
o (K,8.2)    O (O,3.3)    M (M,0.75)       (a,8.2)  (c,2.8)  (x,0.2)
r (V,7.8)    H (H,2.7)    E (E,0.068)      (o,8.0)  (m,2.5)  (j,0.1)
B (B,7.7)    I (I,2.1)    A (A,0.00)       (n,7.1)  (u,2.5)  (q,0.1)
D (D,6.9)    Q (Q,1.9)    C (C,0.00)       (r,6.8)  (g,2.0)  (z,0.1)
a (X,6.5)    W (W,1.9)                     (i,6.4)  (y,2.0)
P (P,5.5)    L (L,1.6)                     (s,6.1)  (p,2.0)
A (A,5.3)    U (U,1.3)                     (h,5.3)  (w,1.5)
C (C,4.2)    R (R,1.0)                     (d,3.8)  (b,1.4)

QDtBJIoOGrDtBIGPArHptoLrapCHrGWGrrGOaJIoPtGNAJQPBSGJHtoGDArHptBoDRCBACBPtCGproAGPPoWAoDSGrtBDLorOBD
arHBDWorIatBoDAaJJGOpJaBDtGNtBDtoQDBDtGJJBLBUJGLBUUGrBPCAaJJGOABpCGrtGNtOGArHptBoDBPtCGrGSGrPGBDotC
GrRorOPIoSBDLWroItCGQDBDtGJJBLBUJGABpCGrtGNtUaAMtopJaBDtGNtaABpCGrorAHpCGrBPapaBroWaJLorBtCIPtCatAr
GatGtCGGDArHptBoDaDOtCGrGSGrPBDLOGArHptBoDtCGOGtaBJGOopGratBoDoWaABpCGrBPAoDtroJJGOUotCUHtCGaJLorBt
CIaDOBDGaACBDPtaDAGUHaMGHtCBPBPaPGArGtparaIGtGrBOGaJJHMDoRDoDJHtotCGAoIIQDBAaDtPWoraPpGABWBAIGPPaLG
GNACaDLGAoDtGNtaArHptoPHPtGIBPtCGorOGrGOJBPtoWGJGIGDtPoWWBDBtGpoPPBUJGpJaBDtGNtPWBDBtGpoPPBUJGAHpCG
rtGNtPWBDBtGpoPPBUJGMGHPaDOtCGGDArHptBoDaDOOGArHptBoDaJLorBtCIPRCBACAorrGPpoDOtoGaACMGHMGHParGBIpor
taDtaPABpCGrPRBtCoQtSarBaUJGMGHPAaDUGtrBSBaJJHUroMGDRBtCoDJHtCGMDoRJGOLGoWtCGABpCGrQPGOaDOarGtCGrGW
orGQPGJGPPorGSGDAoQDtGrproOQAtBSGWorIoPtpQrpoPGPCBPtorBAaJJHABpCGrPRGrGoWtGDQPGOOBrGAtJHWorGDArHptB
oDorOGArHptBoDRBtCoQtaOOBtBoDaJproAGOQrGPPQACaPaQtCGDtBAatBoDorBDtGLrBtHACGAMPBDAoJJoEQBaJQPGtCGtGr
IAoOGBPoWtGDQPGOtoIGaDaDHIGtCoOoWGDArHptBoDorAoDAGaJIGDtoWIGaDBDLCoRGSGrBDArHptoLrapCHAoOGCaPaIorGP
pGABWBAIGaDBDLBtIGaDPtCGrGpJaAGIGDtoWaQDBtoWpJaBDtGNtBGaIGaDBDLWQJRorOorpCraPGRBtCaAoOGRorOWorGNaIp
JGRaJJaUHrGpJaAGPattaAMatOaRDAoOGParGDoJoDLGrQPGOBDPGrBoQPArHptoLrapCHGNAGptBDABOGDtaJJHWorPQACtCBD
LPaPQDBtOGPBLDatBoDPGLUroDAoWJBLCtoropGratBoDoSGrJorOPBDAGpropGrJHACoPGDABpCGrParGUotCIorGpraAtBAaJ
aDOIorGPGAQrGtCaDGSGDtCGUGPtAoOGPaDOaJPoarGUGttGraOaptGOtoAoIpQtGrPHoQrparrot
```

FIGURE 4. This shows the letter frequencies in the ciphertext (listed in decreasing order), the letter frequencies in average English text, and the partially decrypted ciphertext. Ciphertext letters are written in black capital letters, while plaintext characters are written in blue lowercase letters. As more guesses are made for letters, more of the text will appear in blue.

4.5. As you determine the plaintext corresponding to ciphertext letters, make the appropriate changes using the dropdown menus. The first block of ciphertext will update, showing the plaintext in lowercase blue letters. This is shown in Figure 4.

4.6. The first step should certainly be to fill in the letters for "your parrot", which we know ends the plaintext.

4.7. In my experience, it is usually easy to determine which trigram corresponds to "the". As a reality check, the "t" and "e" characters should occur very often, but the "h" character should occur less often.

4.8. At this point you should know the plaintext characters for 9 of the 26 ciphertext characters. Determining the rest is trickier. I typically use a combination of digram frequencies, single letter frequencies, and looking at the partially deciphered text for recognizable words. I have found trigrams are not so useful after finding "the", at least in text of the size we are using. A single thematic word seems to be enough to greatly skew the trigram distributions. (For example, in the essay on mathematics, I believe "mat" is the second most common trigram; note that it appears not once but twice in "mathematics".)

4.9. For more of a challenge, we can attempt to decrypt a message enciphered with the substitution cipher without assuming we already know a piece of the message:

```
X = randomsub()
```

The same techniques should work, but of course it will be more of a challenge.

4.10. Do you want quick access to all of the functions which produce random ciphertext? Type in “random" and hit the tab button. This will give a list of the functions Sage knows beginning with "random". You’ll see our 5 cryptology functions among them.

5. EXHAUSTIVE ATTACK ON THE VIGENÈRE CIPHER

Remember to begin work in a new worksheet! You will need to upload the Week2Code.sage file again. In this project, we will attempt an *exhaustive attack* on the Vigenère cipher. We have already seen how to break the Vigenère cipher, but now we will see what happens when we attempt the naïve approach of simply checking every possible key.

5.1. This project is about limits of what can be computed in a reasonable amount of time. We will see that if the key length is short enough, the naïve approach will work, but if the key length is too long, it is hopeless. To get a sense for this, let's try a computation that is hopeless. The following will create a list called "possiblekeys" which contains all possible Vigenère keys of length 5. We will then print all of these keys.

```
possiblekeys = list(itertools.product(range(0,26),repeat=5))
for key in possiblekeys:
   print key
```

(Make sure you have the indentation. Copying and pasting from this document may not work.) This has begun the hopeless task of printing out all $26^5 = 11881376$ possible keys. This will take at least the majority of classtime to finish, so you should go to the top of the worksheet and select "interrupt" from the Action menu. (You may get a red box saying the interrupt has not succeeded, but it should eventually succeed.) You probably want to click to the left of the output twice to hide it.

5.2. Let's start using the naïve approach in the case when the Vigenère key length is only 2. To make this work, we need a piece of Vigenère ciphertext with appropriate key length:

```
X = randomvigenere(2)
```

If you want, you can print X to make sure it looks reasonable:

```
X
```

Next we will create a list containing all possible keys of length 2:

```
possiblekeys = list(itertools.product(range(0,26),repeat=2))
```

5.3. Recall that to decipher a piece of Vigenère ciphertext using a specific key, say $[10, 21]$, we use the command deciphervigenere(X,[10,21]). So, if we want to try deciphering with *every* possible key, we might try

```
for key in possiblekeys:
   deciphervigenere(X,key)
```

But this is disastrous. It is simply printing each proposed "decryption". Perhaps you could look through 676 possibilities and find the one which is English, but we really need a way to automate this process, especially for the longer keys. (Again, you probably want to hide the output, by clicking to the left of the output two times.)

5.4. One method for determining which "decryptions" are actually English would be to use some sort of English dictionary. This would not be easy to automate, however. For one thing, we don't know where the spaces between the words are. For another, the first word could easily be a proper noun which would not appear in a dictionary. Yet another problem is that the first two letters will very often form an English word just by coincidence. About 15% of random two letter strings are words in English (at least according to the Scrabble dictionary). Instead, we use letter frequency to determine if a string is English.

5.5. Look at the following list:

`englishfreq`

It indicates how often each letter in English appears, on average, in a random block of English text. For example, we expect about 8.2% of letters to be 'a', but only about $.1\%$ of letters to be 'z' in a random block of English. The first bar chart shown by the

`comparetoenglish(X)`

command is a graphical illustration of these percentages. (The second bar chart is not so helpful, because X was encrypted using the Vigenère cipher, not the shift cipher.)

5.6. Now what about

`breakshiftcipher(deciphervigenere(X,[10,21]))`

If $[10, 21]$ is really the key, then the two bar charts should look similar, and the text at the top should be English. How can we check this automatically? We already decided it would be difficult to determine automatically if the "deciphered" text was English. So instead we'll have to work with the two bar charts. How can we determine automatically if they look similar?

5.7. For us, it is easier to tell if the two bar charts look similar, but for the computer, it is easier to tell if the two lists of numbers look similar. Say

`Y = deciphervigenere(X,[10,21])`

Then the command

`getfreq(Y)`

will return a list of numbers indicating letter frequencies, and we can hope that it will be similar to the list in englishfreq. It turns out the best way to determine if the two lists are similar is with the dot product. The larger the dot product, the more similar the lists. Here is the command for finding the dot product.

`numpy.dot(getfreq(Y),englishfreq)`

5.8. We are now ready to attempt to use an exhaustive attack on a Vigenère ciphertext with a key length of 2. I assume that X and possiblekeys are defined as above. Then we can run:

```
for key in possiblekeys:
   Y = deciphervigenere(X,key)
   if numpy.dot(getfreq(Y),englishfreq) > .04:
       print key
```

This will print all decryption keys for which the dot product is bigger than .04. It turns out .04 is too small of a cutoff. Experiment to find a more appropriate cutoff.

5.9. Once you have just a few choices of key, try

`deciphervigenere(X,(a,b))`

where (a,b) is one of the possible keys.

5.10. Did you find the actual key? Great! You are now ready to begin the assignment! Repeat the above steps with larger key lengths. (The key steps are getting the ciphertext, then creating possiblekeys, then looking for the biggest dot products.) For which key lengths does this method work to find the actual key? We should have just decided that length 2 was short enough. In a cell at the bottom, report the largest key length that worked. What went wrong in the next key length? Which part of the computation took too long? To make it easy to grade, please have this report in the very bottom cell. (You can simply write out your answer in English; it shouldn't be anything that Sage will understand. The grader will then look over what you did in the previous cells.) When you are finished, choose "Save & quit" from the top.

6. KNOWN PLAINTEXT ATTACK ON THE AFFINE CIPHER

This project is more numerical and less English-based. It is the only project in which letter frequency is not important! We will consider a special case of the affine cipher described on p. 43 of the textbook. Let p be a prime, and let k be some number in the range $0 < k < p$. The message space and the ciphertext space are both $\mathbb{Z}/p\mathbb{Z}$. The encryption function is simply $m \mapsto k \cdot m \bmod p$. In this project, we will perform a known plaintext attack on this cipher.

6.1. Evaluate the following in Sage.

```
[known,ciphertext] = randomlinear()
```

This defines two variables: "known", which you will work with extensively, and "ciphertext", which you will only use at the very end to check your answer.

6.2. The variable "known" is a list which holds pairs [m,c], encrypted using the linear cipher. In other words, each pair [m,c] satisfies $k \cdot m \equiv c \bmod p$. To look at the contents of "known", evaluate

```
known
```

or, to get a version that's easier to read, look evaluate

```
for pair in known:
   print pair
```

We will use these known [plaintext, ciphertext] pairs to determine the values of k and p.

6.3. For example, consider the following possible contents of "known":

```
[5, 12]
[10, 24]
[13, 25]
[18,6]
```

This tells us immediately that
$5k \equiv 12 \bmod p$
$10k \equiv 24 \bmod p$
$13k \equiv 25 \bmod p$
$18k \equiv 6 \bmod p$
Our immediate goal is to find the values of k and p. You are used to thinking that typically only two equations are needed to solve for two unknowns, but the difficulty here is that these aren't equations, they are *congruences*. For example, the first line tells us only that p divides $5k - 12$.

6.4. Nonetheless, we do have hope for finding k and p exactly. For example, consider the second and third lines and note that $4 \cdot 10 - 3 \cdot 13 = 1$. So

$$k = 4 \cdot 10k - 3 \cdot 13k \equiv 4 \cdot 24 - 3 \cdot 25 = 21 \bmod p.$$

Also we have

$$k = 2 \cdot 18k - 7 \cdot 5k \equiv 2 \cdot 6 - 7 \cdot 12 = -72 \bmod p.$$

On one hand, $k \equiv 21 \mod p$, and on the other hand, $k \equiv -72 \mod p$. So $21 \equiv -72 \mod p$, so $93 \equiv 0 \mod p$. This means that p divides 93. In this example, the actual values of k and p were 21 and 31. In your case, the values will be significantly bigger, and you have to make these sorts of calculations several times before you can determine the exact values of k and p.

6.5. Follow this procedure to find the values of k and p for your specific pairs of values stored in the "known" variable. You will want to use the xgcd command. For example,

```
xgcd(10,13)
```

returns (1,4,-3), which I used above to write $k = 4 \cdot 10k - 3 \cdot 13k$. If the numbers are not relatively prime, this method won't be as helpful.

6.6. In my experience, it is easiest to find p before k. For example, above we knew that p divided 93. If we could also show that p divided say 62, then we'd know that p divided $gcd(93, 62) = 31$. We can make sure this is prime using

```
is_prime(31)
```

6.7. Once you know p, it is easy to find k. In the example above, we knew $k \equiv -72 \mod p$. So the command

```
-72%31
```

would immediately tell us that $k = 21$.

6.8. Once you are confident you know the values of both k and p, run the command

```
lineardecrypt(ciphertext,k,p)
```

If the output looks like English, leave that output at the bottom of the worksheet, and choose "Save & quit" from the top.

7. AUTOMATED DECRYPTION OF VIGENÈRE CIPHERTEXT

7.1. Combine the first and third projects to completely automate the decryption process for a piece of Vigenère ciphertext that was encrypted using a keyword of length 30. (Yes, 30!) Your code should look approximately like:

```
X = randomvigenere(30)
key = []
for i in range(0,30):
   Y = separate(X,i,30)
   # we determine the i-th shift amount, called shiftchoice
   # which we set to have initial value 0
   shiftchoice = 0
   # the best shift choice is the one with the biggest
   # corresponding dot product, called dotprod.
   # We again choose the initial value of 0.
   dotprod = 0
   for j in range(0,26):
      [your code here to determine which of the 26 shifts]
      [produces the biggest dot product]
   key.append(shiftchoice)
print deciphervigenere(X,key)
```

Your code will probably need to use the functions numpy.dot() and getfreq() which were used in Project 3. It will also need the command shift(), which luckily is a very simple command. For example,

```
shift("hello",5)
```

returns the output "MJQQT". Once you have code that works, place your code in the final cell. Then evaluate it. Then choose "Save & quit" from the top.

8. PRIVATE KEY CHALLENGE PROBLEMS

8.1. Decipher a random piece of homophonic ciphertext, attained using the command

```
X = randomhomophonic()
```

(No credit for finding my list of all possible plaintexts and checking to see which is correct!) To receive credit, in the second-to-last cell evaluate the string X and in the last cell put the decrypted string. Don't forget to choose "Save & quit" from the top.

8.2. Decipher a randomly chosen piece of ciphertext, encrypted using the autokey cipher. See Exercise 4.19 in the textbook for information about this cipher. Just like for the Vigenère cipher, our keyword has randomly chosen length 5,6,7, or 8. To get the ciphertext, use

```
X = randomautokey()
```

To reproduce the example in the textbook, use

```
Y = "the autokey cipher is cool"
X = autokeyencrypt(Y, "random")
```

8.3. Write a program to automatically decipher a suitably long piece of ciphertext enciphered using the substitution cipher. With the functions used above, it's easy to determine a guess for every character. The difficult part is mimicking the part where we "look for recognizable English words in the partially deciphered text".

9. DIFFIE-HELLMAN KEY EXCHANGE

In this project, you will work with a partner to perform a Diffie-Hellman key exchange. Warning: a few of the following commands are designed to produce errors. In each case, the error message should be understandable.

9.1. Try to evaluate each of the following, one at a time.

```
(2^(10))%13
(2^(100000000))%13
(2^(2^100))%13
```

and then evaluate the same functions using the "pow" function. If you aren't sure how that function works, evaluate

```
help(pow)
```

9.2. Say we want a primitive root in $\mathbb{Z}/31\mathbb{Z}$. The naïve approach is to try computing $2^1, 2^2, 2^3, \ldots, 2^{30}$ modulo 31 and see if each element $1, 2, \ldots, 30$ appears exactly once in the list. Equivalently, we could see if the element 1 occurs only once in the list. To compute all these powers, evaluate

```
for i in range(1,31):
  print pow(2,i,31)
```

Note that, because of Python conventions, range(1,31) includes values $1, 2, \ldots, 30$, but not 31.

9.3. If 2 is not a primitive root, try 3, then 5, then 6, and so on until you find a primitive root. (Do you see why we skipped 4?)

9.4. Say now we want to find the discrete logarithm $\log_2(16)$. We can evaluate

```
16.log(2)
```

Now try

```
23.log(2)
```

or, to get something more readable,

```
float(23.log(2))
```

But there is a problem, decimal exponents don't make sense in modular arithmetic. There is also another problem. How would Sage know we were working in $\mathbb{Z}/31\mathbb{Z}$?

9.5. Try

```
Integers(31)(23).log(2)
```

and

```
Integers(31)(23).log(3)
```

The part Integers(31)(23) tells Sage that we are thinking of 23 as an element of $\mathbb{Z}/31\mathbb{Z}$.

9.6. Use the "pow" command to verify that the discrete logarithm $\log_3(23)$ in $\mathbb{Z}/31\mathbb{Z}$ which we just computed really has the property it should.

9.7. The above process for finding primitive roots is clearly hopeless if p is huge. But luckily, we can use group theory to find a faster method, that works as long as the factorization of $p-1$ is known.

```
p = 31
factorlist = factor(p-1)
isprimroot(p,factorlist,2)
```

This will tell us if 2 is a primitive root for $(\mathbb{Z}/31\mathbb{Z})^*$. If it is, we stop; if it's not, we try again for 3, then 5, then 6, and so on. (This is the first function that won't work if you forgot to load the Week3Code.sage file. Make sure there are no new indentations after pasting this into Sage.)

9.8. We're now ready to perform a Diffie-Hellman key exchange. Find another student to work with. It doesn't really matter if they are sitting near you or not. A group of three is okay, but groups of two are better. Choose a team name.

9.9. Together you should find a huge prime number. For example, to find one with 70 digits, we could use

```
p = next_prime(random.randint(10^70,10^71))
```

Then print p to make sure it looks reasonable (is it the right length, does it end in $1, 3, 7,$ or 9?)

9.10. Now attempt to factor $p-1$. Sometimes this works, and sometimes it is too difficult. If nothing happens after about 10 seconds, choose "Interrupt" from the Action menu at the top, and find a new prime to try.

```
factorlist = factor(p-1)
```

9.11. Once you have a pair of a prime p and a factorization of $p-1$, you can use the command

```
isprimroot(p,factorlist,g)
```

with various values of g until you have found a primitive root.

9.12. Whichever member of group found p (let's call her "Alice") should now choose a (big) exponent between 1 and p; call it a. Compute $A = g^a \bmod p$ using the "pow" command. Make sure a can't be computed using the discrete log. In other words,

```
Integers(p)(A).log(g)
```

should not finish running in a short time (say under a minute).

9.13. Once you are confident your discrete log problem is difficult to solve, go to the messageboard for our class in EEE and publish: your team name, the prime p, the primitive root g, and the power $A = g^a \bmod p$. Make sure not to reveal the exponent a.

9.14. Your teammate (or teammates), let's call him "Bob", should now find his own $B = g^b \bmod p$ for his own secret exponent b. He should also make sure his exponent b cannot be computed using the discrete log command. Once he is confident, he should publish your team name, together with his key B on the messageboard.

9.15. Now both Bob and Alice can both compute their shared secret key $g^{ab} \bmod p$. Let's call it k.

9.16. Choose a secret message:

```
X = "[your secret message here]"
```

Encrypt it using

```
ciphervigenere(X,Integer(k))
```

Post the ciphertext on the messageboard, together with your team name.

9.17. To decipher your classmate's message, use

```
deciphervigenere(X,Integer(k))
```

If it looks like English, then you're done!

10. COMPUTATIONAL COMPLEXITY INVESTIGATIONS

10.1. Our eventual goal is to gather some data that we can plot. To get a sense for what this data should look like, try the following.

```
pts = Graphics()
pts = pts + point([1,5])
pts = pts + point([2,7])
pts = pts + point([3,9])
show(pts)
```

(Make sure nothing gets indented when you paste this into Sage. To unindent, you can highlight it and press shift+tab.)

10.2. There is also a shorter version which does the same thing:

```
pts = Graphics()
pts = pts + point([[1,5],[2,7],[3,9]])
show(pts)
```

In this case, the data is [[1,5],[2,7],[3,9]], which is a list of points that are to be plotted. Note that these points all lie on the line $y = 2x + 3$. We can create such a list all at once:

```
list_of_points = [[x,2*x + 3] for x in range(1,4)]
pts = Graphics()
pts = pts + point(list_of_points)
show(pts)
```

(Remember that in Python, range(a,b) includes a but does not include b.) This method of plotting is the best one yet, because it generalizes very easily:

```
list_of_points = [[x,2*x + 3] for x in range(-40,40)]
pts = Graphics()
pts = pts + point(list_of_points)
show(pts)
```

10.3. Now let's plot something more interesting. Let's plot the powers $2^x \bmod 131$ for x ranging from 0 to 129. Use the pow command.

```
list_of_points = [[x,???] for x in range(0,130)]
pts = Graphics()
pts = pts + point(list_of_points)
show(pts)
```

This will look random, and it should. The fact that it looks random is a big part of the reason that discrete log (which is the inverse of the function you just graphed) is difficult to compute. (Can you recognize the usual exponential function $y = 2^x$ (with no mod) at the very left side of your plot? Do you see why 131 was a good prime to choose, if I wanted this 2^x beginning to be visible?)

10.4. We will not do much more plotting in this lab, but we will be collecting data, and it's useful to know where that data will be used. The data will consist of pairs [x,y], and later we will plot this data. The x term will represent the size of some input, and the y term will represent how long it takes Sage to perform a computation with that input.

10.5. Let's start by getting a sense for how long it takes Sage to factor numbers. Get a random integer with 10 digits and try to factor it (this won't work if you forgot to load Week4Code.sage):

```
n = randomint(10)
factor(n)
```

Probably it seems like Sage did this very quickly. This means that Sage can factor random 10 digit numbers rather easily. But how long exactly did it take? To find out, use

```
n = randomint(10)
time factor(n)
```

Make a note of the input size and the CPU time. For example, if the output is

```
Time: CPU 0.00 s, Wall: 0.00 s
```

make a note of the data point [10, 0.00]. Once you have more data, you will be posting this on the messageboard. This records the fact that to factor a random 10 digit number, it took Sage 0.00 seconds. Keep this data written down somewhere.

10.6. Repeat this with other sizes of random integers. If a computation hangs, choose "interrupt" from Action menu at the top. Choose your integer sizes so that between you and your partner, you have a variety of about 5-10 computation times (say up to 60 seconds or so). While you're doing this, keep in mind that factoring integers is in general considered *slow*. One advantage of working with a partner is that occasionally one of you can let a computation run for a few minutes to see if it eventually finishes. Lastly, keep in mind that if you are asking Sage to factor some randomint(1000), you're not asking Sage to factor a number like 1000, you're instead asking Sage to factor a number with one thousand *digits*.

10.7. Then go to the class messageboard, find the "factor" thread, and post your points. Type "factor" at the top of your post, to convince me you're posting the data in the correct place. For example, I might post

```
factor
[10,0.00]
[50,1.41]
[50,.15]
[70,91.89]
[100,4.48]
```

Please post in **exactly** this format (enclosed in square brackets, the two entries separated by a comma). This will make collecting the data easier to automate. Some data is shown in Figure 6.

10.8. Note that these points are by no means inputs/outputs of a well-defined function. In my example above, we see that there is one 50 digit random number that Sage took 1.41 seconds to factor, and another 50 digit random number that Sage only took .15 seconds to factor. A *function* cannot have two outputs for the same input. When we are considering complexity, we are mostly interested in the larger computation times. There will always be some big numbers which are easy to factor, like a googol, $1000\ldots00000$ with 100 zeros, factors as $2^{100}5^{100}$.

10.9. Integer factorization is considered a *slow* computation. A good example of a *fast* computation is the Euclidean algorithm for finding the gcd of two numbers. For this, we'll need two big random numbers. Use the following code.

```
d = 10
n1 = randomint(d)
n2 = randomint(d)
time gcd(n1,n2)
```

By varying the number of digits d, find about 5 points of data for the gcd function (with varying times, I don't want to see all zeros!) Go and record them on the class messageboard in the "gcd" thread. Start your post with "gcd" at the top, so I know you're posting in the correct place. Use the same format as for the factor thread. Some data is shown in Figure 5.

10.10. Another operation that is considered *fast* is modular exponentiation. Gather data recording how long it takes to compute $g^{10{,}000} \bmod m$ for random values of g and m. Remember to use the pow command. Instead of ending with

```
time pow(g,10000,m)
```

I found it was better to end with

```
time x = pow(g,10000,m)
```

What does this do? It simply stops Sage from printing the output. Record your data in the "modular exponentiation" thread, and start your post with "pow".

10.11. Now let's time some functions related to discrete logarithms. Let's first try the naïve algorithm, which simply computes powers of g until it hits the right one. The syntax is

```
naive_dlog(g,A,p)
```

which will return $\text{dlog}_g(A)$ in $(\mathbb{Z}/p\mathbb{Z})^*$. Time this with varying values of p, and with g a primitive root $\bmod p$. To find a primitive root $\bmod p$, use the command

```
primitive_root(p)
```

which I didn't tell you about last week because I wanted you to see the difficult part of finding a primitive root, namely, factoring $p-1$. The value of A isn't as important, but make sure it's not an obvious power of g. For example, if $g = 2$, don't take $A = 2, 4, 8, 16, \ldots$. Your code should look like

```
d = 2
p = next_prime(randomint(d))
g = primitive_root(p)
A = ?your choice?
time naive_dlog(g,A,p)
```

Record your data in "naive dlog" thread. Start your post with "naive".

10.12. Our last functions are related to the Collision Algorithm. (This is secretly finding the discrete log of 19, but that won't matter today.) Our first step is to make the two lists, one of baby steps and one of giant steps. This requires a prime p, a primitive root g, and the size of each of the lists n, which is $\lfloor \sqrt{p} \rfloor + 1$.

```
p = next_prime(randomint(d))
g = primitive_root(p)
n = floor(sqrt(p))+1
list1 = baby_step(p,g,n)
time list2 = giant_step(p,g,n)
```

Time how long the giant-step list creation takes (in my experience, it takes slightly longer than the baby step list creation). With the lists you just found, use the command

```
time collision(list1,list2)
```

to find entries in the two lists which are the same. For example, if this command returns $(32, 106)$, then that means that list1[32] == list2[106]. Record the timing data for the giant-step part in the thread "list creation" with "list" at the top of your post, and record the timing data for the collision part in the thread "collision", with "collision" at the top of your post. Repeat both the list-creation step and the collision step with a few different values of d, chosen so you get different times. Some data is shown in Figure 7.

10.13. This previous step should have given you an idea of what size of primes will be vulnerable to the collision algorithm. Let d be about 10 more digits. So for example, if you think 35 digit primes are at the boundary of being vulnerable to the collision algorithm, you'll set $d = 45$. This is a big gap, it means you're taking the biggest prime that is vulnerable, and instead choosing one that is 10 billion times larger. You'll now encode a secret message using a Diffie-Hellman key with a prime of d digits. Your secret exponents a and b can be anything that isn't too small; they should have at least 3 digits.

```
p = next_prime(randomint(d))
X = "[your secret message here]"
a = private key 1
b = private key 2
A = pow(2,a,p)
B = pow(2,b,p)
k = pow(2,a*b,p)
Y = enciphervigenere(X,Integer(k))
```

Go to the "Secret messages for Lab 5" thread and post the data p,A,B, and Y. The formatting isn't as important as for the previous posts, but make it clear which number is which parameter. We'll see next week that some, but probably not all, of these are vulnerable to an attack that is related to the collision algorithm.

Computational Complexity Results

x-axis: number of digits of the input
y-axis: time (in seconds) required for the computation

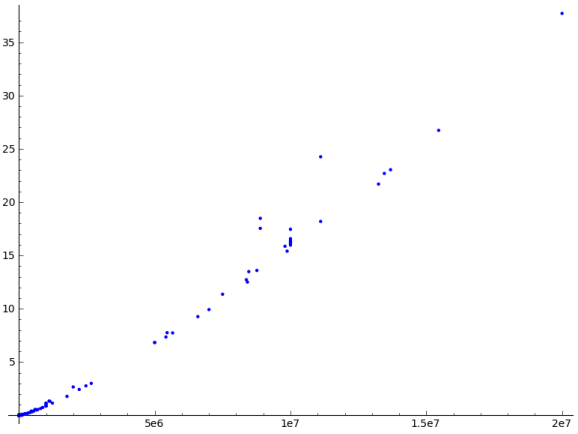


FIGURE 5. GCD

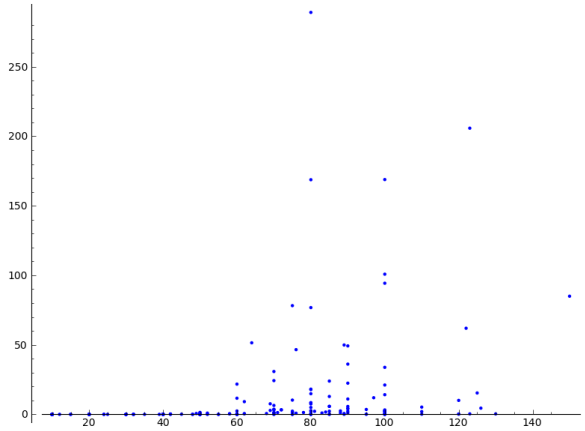


FIGURE 6. Factorization

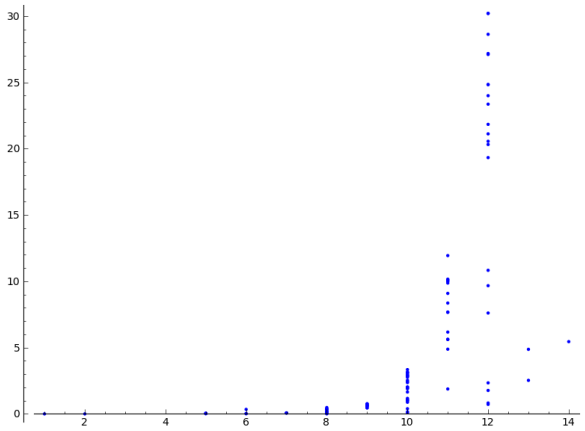


FIGURE 7. Collision Algorithm: list generation step

11. THE POHLIG-HELLMAN ALGORITHM

11.1. Last week, we decided that it was difficult to use the collision algorithm to compute a discrete logarithm in $(\mathbb{Z}/p\mathbb{Z})^\times$ when p has about 15 digits. In particular, the list generation step (i.e., producing the baby step list and the giant step list) was pretty slow when p was this size.

11.2. Today, you will be deciphering a message that was encrypted last week by a classmate. Recall how Diffie-Hellman works. You start with a prime p. (Typically the next step is to find a primitive root g, but last week we took a shortcut and simply used 2 for our base.) Then two secret exponents are chosen a, b, and the powers $A := 2^a \bmod p$ and $B := 2^b \bmod p$ are computed. The numbers A and B are public, but a and b are private. Anyone who knows either a or b can compute the secret key $k :\equiv 2^{ab} \equiv A^b \equiv B^a \bmod p$. A secret message X was chosen last week, and it was enciphered using

```
Y = enciphervigenere(X,Integer(k))
```

This week you will be given the data [p,A,B,Y], and your job is to find k so that you can decipher Y. This is easy once you find a or b.

11.3. Start by getting a piece of data:

```
get_pohlig_hellman()
```

This returns a 4-tuple [p,A,B,Y]. Actually, it will be a mess to be writing down those numbers directly, so get a new piece of data, and this time label everything:

```
[p,A,B,Y] = get_pohlig_hellman()
```

11.4. Look at p by evaluating it on its own line:

```
p
```

This number is too large for us to attempt the collision algorithm directly. We will instead use the Pohlig-Hellman algorithm, which in turn uses the Chinese Remainder Theorem. The Chinese Remainder Theorem concerns determining a number modulo $m_1 m_2 \dots m_n$ by instead determining it modulo m_1 and modulo m_2 and so on individually. This works as long as $\gcd(m_i, m_j) = 1$ for all i, j. For example, the only number in $\mathbb{Z}/99\mathbb{Z}$ which is congruent to $4 \bmod 9$ and is congruent to $7 \bmod 11$ is 40. (But we can't factor the 9 any further. We need to stick with prime powers.) How does this help? To use the Chinese Remainder Theorem, we need to factor something. The number p is prime, so we can't factor that any further. What is it we want to factor?

11.5. Think about the naïve algorithm for discrete logarithms. Say we wanted to compute $a := \text{dlog}_2(A)$. We would compute the powers

$$2^0,\ 2^1,\ 2^2,\ \ldots \bmod p$$

and look for the exponent a such that $2^a \equiv A \bmod p$. Eventually for some big exponent N we have $2^N \equiv 1 \bmod p$. We then have

$$2^N \equiv 1,\ 2^{N+1} \equiv 2,\ 2^{N+2} \equiv 4,\ \ldots \bmod p$$

In other words, the numbers start repeating, so if we were going to find A somewhere in the list, we would find it before N. For example, here are powers of 2 modulo the prime 151:

$$\underbrace{1, 2, 4, 8, 16, 32, 64, 128, 105, 59, 118, 85, 19, 38, 76}_{\text{length 15}}, 1, 2, 4, 8, 16, 32, 64, 128, 105, 59, \ldots.$$

Any number which is a power of 2 modulo 151 occurs among those first 15 numbers, so for example there is no solution to $\text{dlog}_2(50)$. The number 15 is the (multiplicative) order of 2 in $\mathbb{Z}/151\mathbb{Z}$. In this case, N is 15. That number N is what we're going to apply the Chinese Remainder Theorem to. We thus hope that N has prime power factors which aren't too big. (Meaning we can perform the collision algorithm for the individual factors.)

11.6. Determine what N is in your case:

```
N = Integers(p)(2).multiplicative_order()
```

Now factor this number, not into primes (which doesn't work for the Chinese Remainder Theorem), but into prime powers:

```
get_prime_powers(N)
```

11.7. You will be running the collision algorithm on each of these prime powers, so you should be hoping for two things. You should be hoping:

- That the prime powers are small enough that you can run the collision algorithm on them.
- And that there aren't too many prime power factors.

If you've gotten unlucky, you should run [p,A,B,Y] = get_pohlig_hellman() again and hope you get better numbers.

11.8. Now choose one of your prime power factors. (Eventually you'll choose them all.) Call it q:

```
q = paste your factor here
```

Then make the lists for the collision algorithm.

```
n = floor(sqrt(q))+1
list1 = baby_step_pohlig(p,q,2,A,n)
list2 = giant_step_pohlig(p,q,2,A,n)
```

Then find the coordinates of the collision:

```
collision(list1,list2)
```

If you call the output (i, j), then this tells us something about the secret exponent a:

$$a \equiv i + nj \bmod q$$

Make sure you write down both $i + nj$ and q. You will need them later.

11.9. Repeat this for the rest of the prime power factors of N. You will then have a collection of data:

$$a \equiv c_1 \bmod q_1$$
$$a \equiv c_2 \bmod q_2$$
$$a \equiv c_3 \bmod q_3$$
$$\dots$$

To find a exactly using the Chinese Remainder Theorem, evaluate

```
CRT_list([c1,c2,c3,...],[q1,q2,q3,...])
```

where the q's get replaced by your prime power factors, and the c's get replaced by the $i + jn$'s you just found using the collision algorithm. The resulting a found by the Chinese Remainder Theorem is the secret exponent, i.e., it is the discrete logarithm $\text{dlog}_2(A)$ in $(\mathbb{Z}/p\mathbb{Z})^\times$.

11.10. You're almost done. Now use the secret exponent a to find the shared private Diffie-Hellman key k. Once you have it, decipher the message:

```
deciphervigenere(Y,Integer(k))
```

Post your deciphered message, together with the prime p, on the Lab 5 messageboard.

12. RSA

12.1. In this lab, you will share a secret message with a partner using the RSA public key cryptosystem. Unlike with Diffie-Hellman, the two roles in the exchage are rather different. We will use the names "Alice" and "Bob" to distinguish them. Alice will be the one creating the public key, and ultimately receiving Bob's message.

12.2. Alice chooses two large primes p and q and define n to be pq.

12.3. Alice computes $\phi(n)$.

12.4. Alice chooses $e > 1$ with the property that $\gcd(e, \phi(n)) = 1$.

12.5. Alice posts her public key on our message board. Her post should have the form

$$n = ?????,\ e = ??????$$

12.6. Bob picks a secret value b. Alice will recover this as an element of $\mathbb{Z}/n\mathbb{Z}$, so it is important that $0 \leq b < n$. It is also important that b^e be greater than n. (Why?)

12.7. Bob uses the value b as a key for a Vigenère cipher. In particular, he picks a

```
X = "Bob's secret message here"
Y = encipherviginere(X,Integer(b))
```

12.8. Bob computes the value $B := b^e \bmod n$, and posts this value and his Vigenère ciphertext on the messageboard. His post should look like

$$B = ??????,\ Y = ??????$$

12.9. After getting Bob's data from the messageboard, Alice now computes her decryption exponent d, and uses this to recover Bob's plaintext message X.

13. POLLARD'S $p-1$ FACTORIZATION ALGORITHM

13.1. We now describe a method for finding a prime factor of an integer n. We will assume n has the form $p \cdot q$ where p and q are two large primes.

13.2. One of the main reasons that n is difficult to factor relative to its size is that both of its prime factors are "big". So for example it will be very difficult to find a prime factor of n using trial division.

13.3. The opposite sort of number (for which all its prime factors are "small") is called a *smooth* number. Smooth numbers are in general easier to work with, but how can we get a smooth number out of n? The idea is that maybe $p-1$ is smooth, and we can try to access that number. Clearly if we can find $p-1$, then we can simply add one to find p. (And of course, this works just as well if it's q that is smooth.)

13.4. If $p-1$ is smooth, then $p-1$ will divide $M!$ for some M that is not too large. For example, $140 = 2^2 \cdot 5 \cdot 7$ divides 7!.

13.5. We know that for a relatively prime to n (which is a very weak assumption since the factors of n are big):

$$a^{\phi(n)} \equiv 1 \bmod n.$$

By the Chinese Remainder Theorem, this is the same as the statement

$$\begin{aligned} a^{\phi(n)} &\equiv 1 \bmod p \\ a^{\phi(n)} &\equiv 1 \bmod q. \end{aligned}$$

Phrased in terms of Fermat's little theorem:

$$\begin{aligned} a^{p-1} &\equiv 1 \bmod p \\ a^{q-1} &\equiv 1 \bmod q. \end{aligned}$$

Now assume that $p-1$ divides $M!$ for some M. We also assume that $q-1$ does not divide $M!$. So we are assuming that $p-1$ is smooth but $q-1$ is not smooth. We can't take M to be too big, otherwise $M!$ will be divisible by both $p-1$ and $q-1$. We have

$$\begin{aligned} a^{M!} &\equiv 1 \bmod p \\ a^{M!} &\equiv ? \bmod q. \end{aligned}$$

The first line means exactly that p divides $a^{M!}-1$. Unless we have gotten unlucky with our choice of a, we will have that q *does not* divide $a^{M!}-1$. This means that $\gcd(a^{M!}-1, n) = p$.

13.6. The number $a^{M!}$ might be huge, even relative to n. But we can reduce it modulo n without changing $\gcd(a^{M!}-1, n)$. We could try

```
gcd((a^(factorial(M))-1)%n,n)
```

But modular exponentiation, where we reduce modulo n after each step, is much more efficient. The pow command incorporates this:

```
gcd(pow(a,factorial(M),n)-1,n)
```

Actually, this last command confuses Sage, because it thinks of

```
pow(a,factorial(M),n)
```

as an element of $\mathbb{Z}/n\mathbb{Z}$, and then it decides it should make n into an element of $\mathbb{Z}/n\mathbb{Z}$ as well, namely, the element 0. Taking gcd of a number with zero is not interesting. To stop Sage from treating the pow result as an element of $\mathbb{Z}/n\mathbb{Z}$, we do the following.

```
gcd(Integer(pow(a,factorial(M),n))-1,n)
```

13.7. To find a factor of $n = pq$, choose values of a and M and compute

```
gcd(Integer(pow(a,factorial(M),n))-1,n)
```

If the result is 1, the number M is too small: $M!$ is missing some prime factor of both $p-1$ and $q-1$. If the result is n, then either M is too big or a should be changed. The only divisors of n are $1, p, q, n$, so if the gcd is anything other than 1 or n, we have found a prime divisor!

13.8. **Warning!** Prime factorization is difficult. There are certainly some values of n for which Pollard's $p-1$ algorithm will not work. If you have made M so big that that you cannot make the pow computation, and you have only gotten gcd values of 1, then probably this n cannot be factored using this method.

14. Factoring $n = p \cdot q$ given the RSA decryption exponent

14.1. Recall that RSA decryption requires finding d such that $d \equiv e^{-1} \mod \phi(n)$. Because e and n are publicly known in RSA, and because finding inverses modulo m is easy (using the extended Euclidean algorithm), it may seem like anyone can find d. The difficulty is that, without factoring $n = p \cdot q$, it is difficult to compute $\phi(n)$.

14.2. We saw in class that computing $\phi(n)$ was just as difficult as factoring $n = p \cdot q$. This means that if we can factor n, we can easily find $\phi(n)$, and if we can find $\phi(n)$, we can easily factor n. (The last part used the quadratic equation.) The goal of this lab is to show something similar, that if we can find the decryption exponent d, then we can (relatively) easily factor n.

14.3. The idea will be to find numbers x which have the property $x^2 \equiv 1 \mod n$. Let's see why this is useful. The congruence $x^2 \equiv 1 \mod n$ is equivalent, by the Chinese Remainder Theorem, to the pair of congruences

$$\begin{aligned} x^2 &\equiv 1 \mod p \\ x^2 &\equiv 1 \mod q. \end{aligned}$$

It is a fact from Field Theory that this pair of congruences is equivalent to

$$\begin{aligned} x &\equiv \pm 1 \mod p \\ x &\equiv \pm 1 \mod q. \end{aligned}$$

Modulo $n = p \cdot q$, this gives four total possibilities for x: $x \equiv 1$, $x \equiv -1 \equiv n - 1$, and two mystery values. For example, there is the mystery value where $x \equiv 1 \mod p$ and $x \equiv -1 \mod q$. Our goal is to find one of these mystery values.

14.4. Say $x^2 \equiv 1 \mod n$, but $x \not\equiv 1 \mod n$ and $x \not\equiv n - 1 \mod n$. What does this tell us? We have

$$\begin{aligned} x^2 &\equiv 1 \mod n \\ x^2 - 1 &\equiv 0 \mod n \\ (x+1)(x-1) &\equiv 0 \mod n \end{aligned}$$

while at the same time

$$x + 1 \not\equiv 0 \mod n, \text{ and } x - 1 \not\equiv 0 \mod n.$$

Recalling the definition of congruence in terms of division, this tells us that pq divides $(x + 1)(x - 1)$, while pq does not divide $(x + 1)$ and pq does not divide $(x - 1)$.

14.5. Thus exactly one of p and q divides $(x + 1)$ and the other divides $(x - 1)$. So, to find a prime factor of $n = p \cdot q$, we can compute $\gcd(x + 1, n)$.

14.6. So we care about finding a value x such that $x^2 \equiv 1 \mod n$, but such that $x \not\equiv 1 \mod n$ and $x \not\equiv n - 1 \mod n$. In general, this is not possible, but if we know the "secret" decryption exponent d, then it is possible to find such an x.

14.7. Pick a random value of a; for example, $a = 2$ is fine. Assuming $\gcd(a, n) = 1$, which will be true for any small a, since $n = pq$ with p, q both big primes, we have by Euler's Theorem

$$a^{\phi(n)} \equiv 1 \bmod n.$$

The statement that $d \equiv e^{-1} \bmod \phi(n)$ means exactly that $ed - 1$ is divisible by $\phi(n)$. Thus

$$a^{ed-1} \equiv 1 \bmod n.$$

We illustrate how this helps with an example.

14.8. Let $n = 1073$. Assume we know that

$$2^{1008} \equiv 1 \bmod 1073.$$

This tells us that

$$(2^{504})^2 \equiv 1 \bmod 1073,$$

or, in other words, $2^{504} \bmod 1073$ is a square root of 1 modulo 1073. We have

$$2^{504} \equiv 1 \bmod 1073.$$

Combining these two results, we have that 1 is a square root of 1, modulo 1073. This is not very interesting, but we can continue dividing the exponent by 2:

$$2^{252} \equiv 1 \bmod 1073$$
$$2^{126} \equiv 1072 \bmod 1073.$$

This unfortunately yields $1072 \equiv -1 \bmod 1073$ is a square root of 1 modulo 1073, which is still not interesting.

14.9. If we change the 2 to a 5, however, we get something useful.

$$5^{1008} \equiv 1 \bmod 1073$$
$$5^{504} \equiv 1 \bmod 1073$$
$$5^{252} \equiv 1 \bmod 1073$$
$$5^{126} \equiv 813 \bmod 1073.$$

So we have learned that 813 is a square root of 1 modulo 1073. So we have learned that

$$813^2 - 1 \equiv 0 \bmod 1073.$$

Thus

$$(813 + 1)(813 - 1) \equiv 0 \bmod 1073.$$

To find a prime factor of 1073, we compute

$$\gcd(813 + 1, 1073) = 37.$$

This is one prime factor. The other prime factor is $1073/37 = 29$.

14.10. Now return to our setup, where there is $n = p \cdot q$ that you want to factor, and we know e (which the public always knows) and $d := e^{-1} \bmod \phi(n)$ (which the public is not supposed to know). Use this information and the above procedure to factor n. You might want to adapt the following piece of code to help you:

```
m = 1008
for i in range(0,10):
    print m, pow(2,m,1073)
    m = m/2
```

(This computes $2^{1008} \bmod 1073$, then it divides 1008 by 2 and computes $2^{512} \bmod 1073$, etc. You will get an error message as soon as m stops being an integer. If you get nothing useful, change the base 2 into 3 or 5 or ...)

14.11. **Warning.** If things seem not to be working, compute $e \cdot d - 1$. This number should be divisible by $\phi(n)$ (a value we don't yet know), but it should not be equal to zero. Some students were getting that $e \cdot d - 1 = 0$. This seems to be because Sage is viewing e and d as elements of $\mathbb{Z}/\phi(n)\mathbb{Z}$, and so it thinks their product should also be an element of $\mathbb{Z}/\phi(n)\mathbb{Z}$. To prevent this, use

```
Integer(e)*Integer(d)-1
```

The word "Integer" tells Sage that it should view e and d as integers, not as elements of $\mathbb{Z}/\phi(n)\mathbb{Z}$.

15. Introduction to the quadratic sieve

15.1. The goal of this lab is to introduce the quadratic sieve. The numbers we work with here will be small enough that we could compute the factors directly. The point of this lab is to illustrate how the main steps in the quadratic sieve algorithm work.

15.2. Recall from the last lab that we tried to factor a number $n = p \cdot q$ by finding a value $m^2 - 1^2 = (m+1)(m-1)$ which is divisible by n. We then hoped that $\gcd(n, m+1) = p$ or q. In general, it is difficult to find interesting values of m for which this works. We were successful because we were making the assumption that we had found a secret decryption exponent d.

15.3. In this lab, we will introduce a method that works more generally (with no assumption that we have a secret decryption exponent d). Instead of considering values $x^2 - 1$ which are divisible by n, we will consider values $x^2 - y^2$ which are divisible by n. In other words, we want values x and y for which $x^2 \equiv y^2 \bmod n$.

15.4. We want

$$x^2 \equiv y^2 \bmod n.$$

If we have $0 \leq x^2, y^2 < n$, then $x^2 = y^2$, and this will never yield an interesting factorization of n. So we need at least one of x^2 and y^2 to be bigger than n. Let's assume it's x^2 that is bigger than n. What does that tell us about x? We must have $x > \sqrt{n}$. Also, we are only interested in integers. The easiest way to find integers x such that $x > \sqrt{n}$ is with the floor function:

$$x \geq \lfloor \sqrt{n} \rfloor + 1.$$

15.5. Here is an example. Say we want to factor $n = 33$. In this case $\lfloor \sqrt{33} \rfloor = 5$, so we can consider $x = 6, 7, 8, \ldots$. We want

$$x^2 - y^2 \equiv 0 \bmod 33$$

so if we try $x = 6$, we want

$$36 - y^2 \equiv 0 \bmod 33.$$

If we reduce 36 modulo 33, we want

$$3 - y^2 \equiv 0 \bmod 33.$$

This doesn't yield any useful factorizations, but if we instead try $x = 7$,

$$49 - y^2 \equiv 0 \bmod 33,$$

and reducing 49 modulo 33 this time yields

$$16 - y^2 \equiv 0 \bmod 33.$$

Taking $y = 4$ solves this last equation, but what does that tell us? It tells us that $7^2 - 4^2 \equiv 0 \bmod 33$. Factoring the left-hand side, we have $(7+4)(7-4) \equiv 0 \bmod 33$, and the left-hand side is the factorization $11 \cdot 3 = 33$.

15.6. In general we won't get the exact factorization, and will have to compute $\gcd(n, x+y)$ to find an interesting prime factor. Of course, in the above example if we compute $\gcd(33, 7+4)$ we get the prime factor 11.

15.7. We have now explained how to choose the x values in our desired equation $x^2 \equiv y^2 \bmod n$: namely, we choose $x = \lfloor\sqrt{n}\rfloor+1, \lfloor\sqrt{n}\rfloor+2, \ldots$. These x values should be thought of as being "big" (meaning big enough that x^2 is bigger than n, and so reducing modulo n produces a new number). The y-values, on the other hand will be "small". More precisely, they will have small prime factors.

15.8. Here is a more sophisticated example. We want to factor $n = 8033$. In this case, $\sqrt{8033} \approx 89.627$, so we start with $x = 90, 91, 92, \ldots$ and compute the following prime factorizations

$$\begin{aligned} 90^2 &\equiv 67 \bmod 8033 \\ &= 67 \\ 91^2 &\equiv 248 \bmod 8033 \\ &= 2^3 \cdot 31 \\ 92^2 &\equiv 431 \bmod 8033 \\ &= 431 \\ 93^2 &\equiv 616 \bmod 8033 \\ &= 2^3 \cdot 7 \cdot 11 \\ 94^2 &\equiv 803 \bmod 8033 \\ &= 11 \cdot 73. \end{aligned}$$

15.9. The x^2 should be thought of as coming from the left side of such a list, and the y^2 should be thought of as coming from the right side. A positive integer is a square if and only if each of its prime factors occurs an even number of times, and so we are disappointed that the right sides are not squares. We could try more values of x, but for this specific case of $n = 8033$, even if we try 100 values, we will not find a reduction which is a square. This is very bad! For example, if we tried the naive algorithm for prime factorization, we would expect it to take at most $\lfloor\sqrt{8033}\rfloor = 89$ steps. Clearly we don't want to do all this work to use an algorithm that is worse than the naive algorithm!

15.10. Let's see the correct way to factor $n = 8033$ using this method. Assume we have found that

$$\begin{aligned} 96^2 &\equiv 1183 \bmod 8033 \\ &= 7 \cdot 13^2 \\ 127^2 &\equiv 63 \bmod 8033 \\ &= 3^2 \cdot 7. \end{aligned}$$

This may not look so useful, but if we multiply both of these together, we get

$$(96 \cdot 127)^2 \equiv (3 \cdot 7 \cdot 13)^2 \bmod 8033.$$

15.11. The previous step tells us that

$$8033 \text{ divides } (96 \cdot 127 + 3 \cdot 7 \cdot 13) \cdot (96 \cdot 127 - 3 \cdot 7 \cdot 13).$$

We hope that $\gcd(8033, 97 \cdot 127 + 3 \cdot 7 \cdot 13)$ is a prime factor of 8033. We compute

$$\gcd(8033, 97 \cdot 127 + 3 \cdot 7 \cdot 13) = 277,$$

which is indeed a prime factor of 8033.

15.12. We are now ready to start making computations in Sage. Get a five digit $n = pq$ that we will attempt to factor:

```
n = get_sieve_mod(5)
```

To see the value of n, evaluate

```
n
```

on its own line.

15.13. Now run the command

```
interactive_sieve(n)
```

This presents us with two drop-down menus of options: the first is how many squares we should compute, beginning at $\lfloor \sqrt{n} \rfloor + 1$. For example, above we computed $90^2, 91^2, \ldots$ all modulo 8033.

15.14. The second drop-down menu tells how big of prime factors we will look for. It would be cheating to simply factor the reductions of all of these squares, because because factoring is slow. Thus we use a process called *sieving*. We illustrate by factoring our largest integer yet.

15.15. Assume we wish to factor the integer

```
n = 57997
```

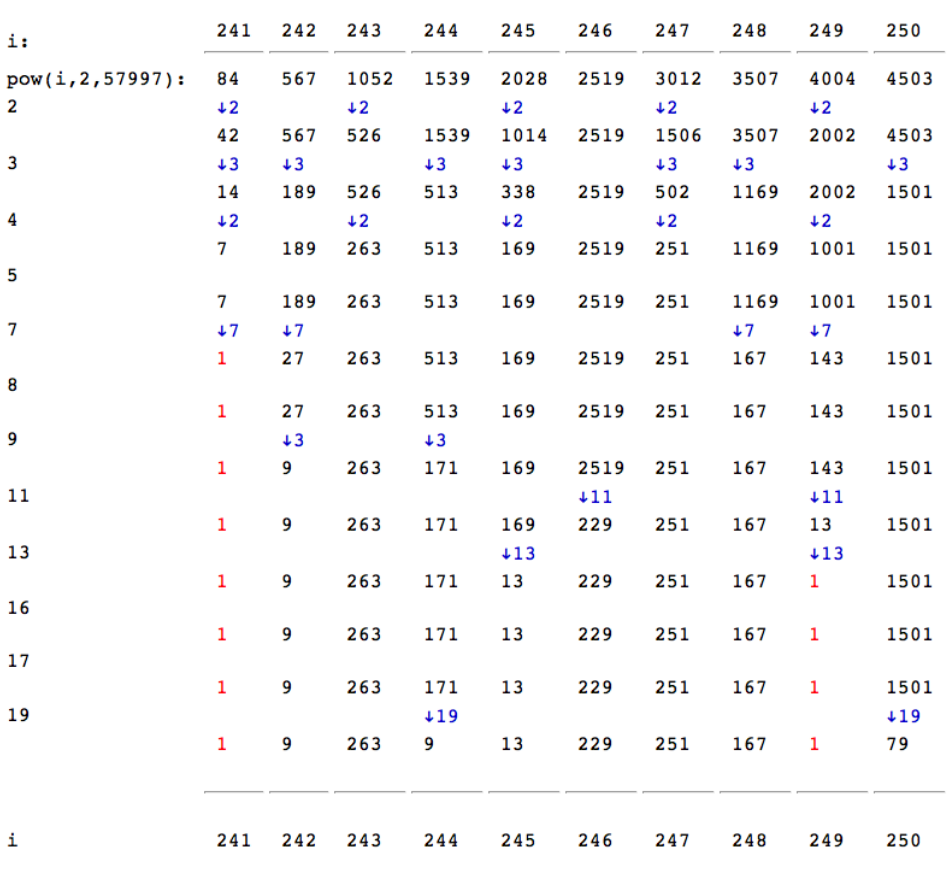
Try running

```
interactive_sieve(n)
```

and choosing 10 for the number of squares and 19 for the factor base bound. The result is shown in Figure 8.

15.16. Because $\lfloor \sqrt{n} \rfloor + 1 = 241$, this attempts to factor 10 reductions starting with $241^2 \bmod 57997$, $242^2 \bmod 57997$, etc. It attempts to perform the factorization by dividing out prime powers up to 19. After a number has been factored completely, the remaining 1 appears in red.

15.17. We look for smooth numbers: numbers whose prime factors are not too big. If we have not found enough, we can either choose more squares or take a larger factor base bound. In our specific case, the reductions modulo 57997 of 241^2, 242^2, 244^2, 245^2, and 249^2 have no prime factors bigger than 19. Note that we count 242^2 and 244^2 even though their numbers don't appear in red in our table. Is there any combination of these reductions that yields a square?



i:	241	242	243	244	245	246	247	248	249	250
pow(i,2,57997):	84	567	1052	1539	2028	2519	3012	3507	4004	4503
2	↓2		↓2		↓2		↓2		↓2	
	42	567	526	1539	1014	2519	1506	3507	2002	4503
3	↓3	↓3		↓3	↓3		↓3	↓3		↓3
	14	189	526	513	338	2519	502	1169	2002	1501
4	↓2		↓2		↓2		↓2		↓2	
	7	189	263	513	169	2519	251	1169	1001	1501
5										
	7	189	263	513	169	2519	251	1169	1001	1501
7	↓7	↓7						↓7	↓7	
	1	27	263	513	169	2519	251	167	143	1501
8										
	1	27	263	513	169	2519	251	167	143	1501
9		↓3		↓3						
	1	9	263	171	169	2519	251	167	143	1501
11						↓11			↓11	
	1	9	263	171	169	229	251	167	13	1501
13					↓13				↓13	
	1	9	263	171	13	229	251	167	1	1501
16										
	1	9	263	171	13	229	251	167	1	1501
17										
	1	9	263	171	13	229	251	167	1	1501
19				↓19						↓19
	1	9	263	9	13	229	251	167	1	79
i	241	242	243	244	245	246	247	248	249	250

FIGURE 8. Result of applying interactive_sieve(57997).

15.18. There is a helper function to look for squares here. The result of applying

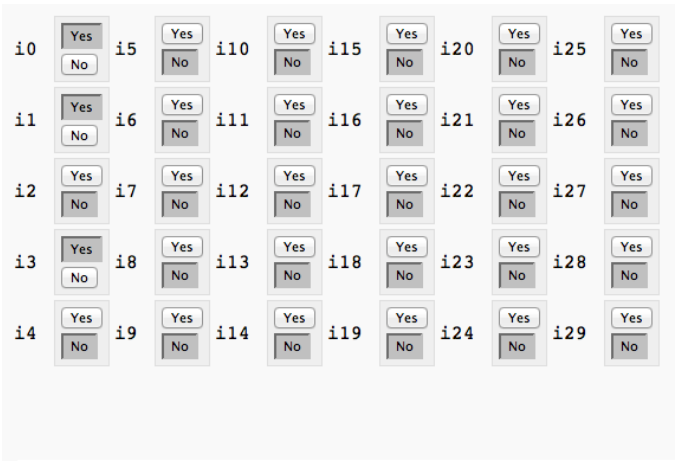
```
sieve_matrix(57997,[241,242,244,245,249])
```

is shown in Figure 9. The Yes/No buttons are used to indicate whether we want to count a factor or not, and then at the bottom in bold is the total number of each prime factor. Our goal is for each total to be an *even* number. So for example in Figure 9, we see that $241^2 \cdot 242^2 \cdot 245^2 \equiv 2^4 \cdot 3^6 \cdot 7^2 \cdot 13^2 \bmod 57997$.

15.19. We compute

```
gcd(57997, 241*242*245 - 2^2*3^3*7*13)
```

The result is 59, which is a prime factor of 57997.



prime:	2	3	7	11	13	19
i0=241	2	1	1	0	0	0
i1=242	0	4	1	0	0	0
i2=244	0	4	0	0	0	1
i3=245	2	1	0	0	2	0
i4=249	2	0	1	1	1	0
Total:	**4**	**6**	**2**	**0**	**2**	**0**

FIGURE 9. Result of applying sieve_matrix(57997,[241,242,244,245,249]).

16. More on the quadratic sieve

16.1. The methods of the preceding section will be too cumbersome if the number n which we want to factor is huge (say ≈ 100 digits). Factoring even a 7 digit number is not so easy using the explicit steps in the previous section. In this section, we automate as much as possible from the previous section.

16.2. Get a 50-digit integer $n = p \cdot q$ as in RSA:

```
n = get_RSA_mod(50)
```

16.3. The two initial steps in the last section were to determine how many squares to compute, and to determine how big the factor base should be. Define a number z which will correspond to the number of squares we will compute and a number L which will determine a bound for the factor base. For example

```
z = 10
L = 31
```

could be used if we wanted to compute 10 squares (which is certainly too few) and wanted to factor out prime powers up to 31.

16.4. Now compute the factor base, which will correspond to the prime powers we will try to sieve out:

```
fb_list = factor_base(L)
```

and similarly compute the list of squares:

```
sq_list = square_list(n,z)
```

16.5. Recall from the previous lab that the function interactive_sieve computed a fixed number of squares modulo n and used the factor base to see which of those reductions were smooth. There is a function which automates this:

```
fac = sieve_out(n,sq_list,fb_list)
```

The parameter fac is an (ordered) dictionary. The keys in this dictionary are the numbers i such that $i^2 - n$ is smooth (meaning it can be factored completely using elements in our factor base). To each of those keys is an associated value, which records the exponents of each prime (not prime power) in the factor base.

16.6. For example, consider the following sequence of commands:

```
n = 221
sq_list = square_list(n,20)
fb_list = factor_base(9)
```

Note that $\lfloor \sqrt{221} \rfloor + 1 = 15$. Then sq_list is the list

$$[15^2 - 221, 16^2 - 221, \ldots, 34^2 - 221].$$

```
fac = sieve_out(n,sq_list,fb_list); fac
```

evaluate

```
OrderedDict([(15, [2, 0, 0, 0]), (16, [0, 0, 1, 1]), (19, [2, 0, 1,
1])])
```

FIGURE 10. Result of applying the quadratic sieve in an attempt to factor 221.

16.7. What do we do with this data? For each of the 20 squares in sq_list, we check to see if it's smooth. Because we chose our factor base to consist of all prime powers that were less than or equal to 9, in this case smooth means divisible by no prime powers bigger than $8, 9, 5$, or 7. See Figure 10. This figure indicates that only three of the squares qualified as smooth. For example, $15^2 - 221 = 4$ which is 2^2. That is indicated by the dictionary entry $[15, [2, 0, 0, 0]]$. Similarly, $19^2 - 221 = 140 = 2^2 \cdot 5 \cdot 7$. This is indicated by the dictionary entry $[19, [2, 0, 1, 1]]$.

16.8. Those lists only show the exponents, not the primes. To see the primes, we can use the function

```
primes_in_list(fb_list)
```

In the above case, that gives as output $[2, 3, 5, 7]$. It will be useful to have access to these primes. Evaluate

```
pr_list = primes_in_list(fb_list)
```

16.9. In the last lab, we wanted to choose combinations of squares so that each total exponent was even. In other words, we want each total exponent to be zero modulo 2. This can be phrased very concisely in terms of matrix multiplication over $\mathbb{Z}/2\mathbb{Z}$; to emphasize that $\mathbb{Z}/2\mathbb{Z}$ is a *field*, we will call it GF(2) here. The "GF" stands for "Galois Field". We need to know the the length and width of the matrix.

16.10. Referring back to Figure 9, we see that the number of rows should be the number of smooth squares. This is the same as the number of entries in the dictionary, which we can access by

```
numrows = len(fac)
```

The number of columns should be the number of primes. This can be accessed by

```
numcols = len(primes_in_list(fb_list))
```

16.11. Combining the previous few steps, we want to create a matrix with the following parameters

```
MS = MatrixSpace(GF(2),numrows,numcols,sparse=True)
```

The "sparse=True" part tells Sage that we expect many of the entries of these matrices to be zero. For such sparse matrices, special algorithms are used.

16.12. Now actually making the matrix is easy. (For Python experts: it is important that we are using an ordered dictionary, not an unordered dictionary.) We use the command

```
exp_matrix = []
for value in fac.values():
    exp_matrix.append(value)
mod_matrix = MS(exp_matrix)
```

16.13. We now want to know what linear combinations of rows add to give a zero vector (where elements are viewed modulo 2). This can be phrased very naturally in terms of a (left) null space:

```
mod_matrix.kernel()
```

This null space is a vector space, and the above Sage command will give you the following basis for it: $[1, 0, 0]$ and $[0, 1, 1]$. Recall that the rows were $15, 16, 19$. Then the previous vectors being in the nullspace tells us that 15^2-221 is square and that $(16^2-221)\cdot(19^2-221)$ is square. These correspond to

$$221 \mid (15^2 - 2^2) \text{ and } 221 \mid \left((16 \cdot 19)^2 - (2 \cdot 5 \cdot 7)^2\right).$$

Although there is no guarantee we won't find trivial factors, in this case, both $\gcd(221, 15+2)$ and $\gcd(221, 16*19+2*5*7)$ yield non-trivial factors of 221. Thus in this case, the quadratic sieve has worked successfully.

16.14. We are now ready to complete the automation of the quadratic sieve algorithm. Assume we have completed the above steps, and we have access to the null-space via the variable V defined by

```
V = mod_matrix.kernel()
```

Recall that our goal is to find A, B such that $n \mid A^2 - B^2$. If $nullvec$ is a vector in the null space, then to find A we can take

```
A = 1
for i in range(0,len(nullvec)):
    if nullvec[i] == 1:
        A *= fac.keys()[i]
```

16.15. To find B, we can take

```
B = 1
for i in range(0,len(nullvec)):
    if nullvec[i] == 1:
        holder = fac.values()[i]
        for j in range(0,len(holder)):
            B *= pr_list[j]^holder[j]
B = sqrt(B)
```

16.16. We can now compute

```
gcd(n,A+B)
```

in the hopes that it is a non-trivial factor of n.

16.17. **Hint.** It is not too difficult to automate the previous few steps further. Your code could begin

```
for nullvec in mod_matrix.kernel():
    A = 1
    B = 1
    your code here
```

Your code could end with the following, which checks to see if you have found a non-trivial factor of n, meaning a factor different from 1 and n:

```
    holder = gcd(n,A+B)
    if (holder != 1) and (holder != n):
        print holder
```

16.18. **Unknown to me.** In the previous step, it was suggested that you check each vector in the null space of the matrix to see if it produces a non-trivial factor of n. I don't know if checking each element in a basis for the null space is enough. Can you either prove that it's enough, or produce a counter-example?

17. PRIMALITY TESTING

You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

`http://bduc-claw9.oit.uci.edu/`

Even if you used Sage last quarter, you will need to sign up for a new account. This is an option under the login. From off-campus, proceed to

`http://uci.sagenb.org/`

For this site, your account from last quarter should still work. If you need an account, contact Professor Davis to have one set up for you. Lastly, you can also proceed to the first site above using VPN here:

`https://vpn.nacs.uci.edu/`

Alternatively, if you already have Sage installed on your own machine, you can open it up and type notebook() at the prompt.

17.1. The goal of this lab is to re-familiarize ourselves with how Sage works and to perform some primality tests. Pay special attention to how long various computations take. For example, notice that the computation

```
factor(n)
```

in general takes much longer than

```
is_prime(n)
```

17.2. Reminder! If a computation seems to be running for ever, you can "interrupt" it by choosing "interrupt" from the "action" menu at the top.

17.3. Sage has a built-in function to test for primality:

```
is_prime(n)
```

where n is some number which you want to determine if it's prime.

17.4. You can also factor using Sage, although in general this is much more difficult than checking primality:

```
factor(319137)
```

17.5. To determine how long it takes for Sage to perform a computation, type the word "time" before it. So for example:

```
time is_prime(n)
```

17.6. You can also write code to count the number of primes in a certain range. For example

```
holder = 0
for i in range(10,20):
    if is_prime(i):
        holder = holder + 1
print holder
```

displays the number of primes p in the range $10 \leq p < 20$. As always, make sure you use the correct indentations (spacing at the beginning of the lines). If you need to indent a block of text, highlight it and then hit the "tab" key. If you need to un-indent a block of text, highlight it and hit "tab" while holding down the "shift" key.

17.7. Another fast method for proving compositeness of a number n is to use $\gcd(a, n)$ where the prime factors of a are all smaller than n. The result of $\gcd(a, n)$ will be a divisor of n, and if that divisor is different from 1, then we've proven that n is composite.

17.8. Here is some code which finds 10 numbers n, starting at 10^7, which satisfy

$$\gcd(n, (10^3)!) = 1.$$

```
holder = 0
n = 10^7
while holder < 10:
    if gcd(n,factorial(10^3)) == 1:
        print n, "is prime:", is_prime(n)
        holder = holder+1
    n = n+1
```

17.9. For each of these 10 numbers, it also indicates if the result is actually prime. For example, here is the output of the above code:

```
10000019 is prime: True
10000043 is prime: False
10000079 is prime: True
10000103 is prime: True
10000117 is prime: False
10000121 is prime: True
10000139 is prime: True
10000141 is prime: True
10000153 is prime: False
10000163 is prime: False
```

17.10. Once you are comfortable with the material from this lab, complete the accompanying worksheet.

173B Lab 1: Worksheet

1. Find a number n for which it takes Sage 5-10 seconds to determine if it's prime.
2. Find a number n for which it takes Sage 5-10 seconds to compute $2^n \bmod n$.
3. Find a number n for which it takes Sage 5-10 seconds to compute $\gcd(n, (10^6)!)$.
4. Find a number n for which it takes Sage 5-10 seconds to factor.
5. Rank your entries in the previous four steps in order from smallest to largest. At the same time as you write down these numbers, indicate which computation that number corresponded to. (The goal of these problems is to get some intuition for which computations are faster and which computations are slower.)
6. Find an interval $[a, b]$ in which there are exactly 100 numbers n such that
$$2^n \equiv 2 \bmod n.$$
How many primes are there in this same interval?
7. Repeat the previous question with numbers a, b which are much larger or much smaller than in your previous answer.
8. Find a composite number n such that $\gcd(n, (10^6)!) = 1$.
9. Some sample code was provided which starts at 10^7 and finds the next 10 numbers all of whose prime factors are greater than 1000. One might naively guess that if all the prime factors of a number are greater than 1000, then that number is probably prime. Can you find a different starting point, where the next 10 numbers found are all composite, despite having no small prime factors? What does this indicate about the effectiveness of this $\gcd$ method as a primality test?
10. An integer n is called a *base 2 Fermat pseudoprime* if n is composite and $2^n \equiv 2 \bmod n$. Find such a base 2 Fermat pseudoprime n satisfying $n > 100,000$. Is 2 a Miller-Rabin witness for n?

18. Elliptic curves over $\mathbb{F}_p$

You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

`http://bduc-claw9.oit.uci.edu/`

From off-campus, proceed to

`http://uci.sagenb.org/`

If you need an account, contact Professor Davis to have one set up for you. Lastly, you can also proceed to the first site above using VPN here:

`https://vpn.nacs.uci.edu/`

Alternatively, if you already have Sage installed on your own machine, you can open it up and type notebook() at the prompt.

18.1. This lab has two goals. First, to write a program that will count the number of points on an elliptic curve over the field $\mathbb{F}_p$. And second, to compare those results to the expected number of points. There is an accompanying worksheet that should be turned in together with Homework 3.

18.2. Here is some (inefficient) sample code for counting the number of primes between x and y. (Including x but not including y. We choose this strange convention because that's how the Python command range(x,y) works.)

```
def count_primes(x,y):
    total = 0
    for i in range(x,y):
        is_prime = True
        for j in range(2,i):
            if i%j == 0:
                is_prime = False
        if is_prime:
            total = total+1
    return total
```

18.3. The above code defines a function count_primes which takes two inputs and returns the number of primes between them. It does this in the simplest possible way. For each number i from x to y, it starts out assuming it's prime. Then for each number j between 2 and i (including 2 but not including i), it checks to see if i is divisible by j. If it is, then it decides i was not prime. If at the end of this process it still thinks i is prime, it adds 1 to the prime total. Then it returns the value of the total.

18.4. You're encouraged to try pasting this into Sage (making sure indentations are preserved; use shift+tab if you need to un-indent). Once you execute this code, while you remain on the Sage worksheet, any time you type count_primes(2,10), for instance, it will execute your program from above. Notice that count_primes(2,11) will give you the same answer as count_primes(2,10), because the upperbound is not counted.

18.5. Our count_primes function is imperfect in many ways. To make it more efficient, we should use sieving. And it gives the wrong answer (!!) for any range that starts $x = 1$ or below. The goal was just to illustrate what some simple Sage programming would look like.

18.6. Say we want to compare our prime counts to the prediction from the prime number theorem.

```
x = var("x")
bound = 50
g = plot(x/log(x),(x,2,bound))
for i in range(2,bound):
    g = g + point((i,count_primes(2,i)))
show(g)
```

The result is shown in Figure 11.

18.7. The above code shows two different types of data that can be plotted: graphs of functions, and points. The line

```
g = plot(x/log(x),(x,2,bound))
```

tells Sage to define a variable g which holds the graph $x/\log(x)$ on the interval from 2 to 50 (because above we set bound = 50).

18.8. The line

```
g = g + point((i,count_primes(2,i)))
```

tells Sage to keep all the data as before (that's what the g on the right side means) and to also include the single point with x-coordinate i and with y-coordinate the number of primes between 2 and i.

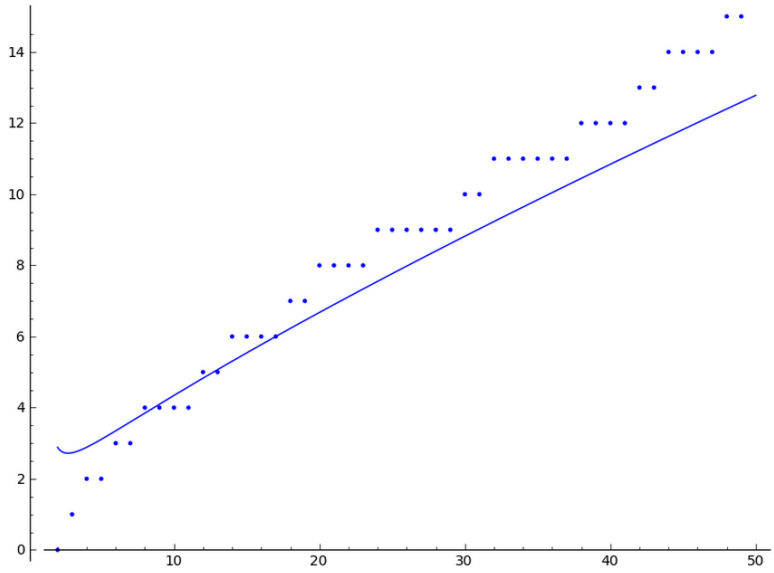


FIGURE 11. Note the two different types of elements shown. There are the collections of points, and the graph of the function $x/\log(x)$.

18.9. Now the work begins. Write a function

`countpts(a,b,p)`

which counts the number of points on the elliptic curve $y^2 = x^3 + a \cdot x + b$ over the finite field $\mathbb{F}_p$. Don't forget to include the point at infinity. (That doesn't mean anything fancy... just add one to your point total!) For now, just try all values of x and y.

18.10. To test your function, you should find that $y^2 = x^3 + 3x + 8$ has nine points over $\mathbb{F}_{13}$ and that $y^2 = x^3 - 3x + 5$ has 28 points over $\mathbb{F}_{19}$.

18.11. From now on, fix values of a and b. This corresponds to fixing an elliptic curve. Don't take a and b to both be zero.

18.12. Using the time command, find a prime value p for which your function takes about $5 - 10$ seconds to count the points. (Hint. next_prime is a good command to use to find a prime number.)

18.13. Now make a new, more efficient function countpts2 by using the command kronecker_symbol. You should not have to check any values of y now.

18.14. Using the time command, find a prime value p for which your improved function takes about $5 - 10$ seconds to count the points. (Hint. next_prime is a good command to use to find a prime number.)

18.15. Alter the following code to simultaneously graph

[point count] - [the expected point count of $p + 1$],

together with a common function. Our provided code shows how to do this for 10 primes and the function $\log(x)$, but you should do it for as many primes as can be completed in a reasonable amount of time, and also for the functions $\sqrt{x}$, x, x^2. (You must replace "???" in the code before it will work.)

```
p = 2
h = point((0,0))
for i in range(0,100):
    p = next_prime(p)
    h = h+point((p,???))
x = var("x")
h = h+plot(log(x),(x,0,p))
show(h)
```

18.16. The h = point((0,0)) line at the beginning should be ignored. It is just there to initialize h.

173B Lab 2: Worksheet

Copy and paste from your Sage worksheet directly in a text document with your answers. Turn in your answers with Homework 3 on Tuesday.

1. What was the code defining your function countpts(a,b,p)?

2. What was the code defining your function countpts2(a,b,p)?

3. What values of a and b did you use?

4. For what value of p did your countpts function take about 5-10 seconds?

5. For what value of p did your countpts2 function take about 5-10 seconds?

6. Which of the functions you plotted seemed to match the data the best? Try pasting the image into your answers file, by right-clicking the image and choosing "copy".

19. ELLIPTIC DIFFIE-HELLMAN KEY EXCHANGE

• You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

```
http://bduc-claw9.oit.uci.edu/
```

From off-campus, proceed to

```
http://uci.sagenb.org/
```

• We will need some special functions defined in an external file, and getting Sage to recognize these functions is a three step process.

a. Go to our class website and download the "Week7Code.sage" file, keeping the name the same, including the suffix ".sage". Don't save it as a text file.

b. Back in the Sage worksheet, find the "Data" menu at the top, and choose "Upload or create file". On the next screen, upload the file you just downloaded from our website. I always get an error message, but if you get back to the worksheet screen (and perhaps refresh), it seems to work.

c. Finally, we need to tell the worksheet we want to use that data. Use the command

```
load DATA+"Week7Code.sage"
```

19.1. To make sure those functions loaded correctly, try evaluating

```
encipherviqenere("hello there",Integer(145))
```

19.2. Find a partner, with whom you will perform an elliptic Diffie-Hellman key exchange.

19.3. For this lab, the following commands will be useful.

19.4. This command creates an elliptic curve over the finite field $\mathbb{F}_p$ defined by

$$y^2 = x^3 + ax + b.$$

```
E = EllipticCurve(GF(p),[0,0,0,a,b])
```

Make sure to replace a, b and p with specific numbers, possibly by first using something like

```
p = 13
a = 1
b = 2
```

19.5. This command finds the next prime occurring after the number n:

```
next_prime(n)
```

19.6. This command finds a random point on the elliptic curve E:

```
E.random_element()
```

Ignore the third coordinate. That coordinate is 0 if it's the point at infinity, but otherwise it is always 1.

19.7. If x and y are two coordinates of a point on an elliptic curve E, then that point can be defined using

```
P = E(x,y)
```

(This tells Sage that P should be thought of as a point on an elliptic curve, so it will know what to do if you ask a question like "What is the order of P?")

19.8. If you just want a random point, and don't care about specifying its coordinates, you can use

```
P = E.random_element()
```

19.9. This command finds the order of a point P on an elliptic curve:

```
P.order()
```

19.10. This command computes $P + P + P$:

```
3*P
```

19.11. This command retrieves the zero-th item in a list called exlist (computer scientists often begin counting elements at zero):

```
exlist[0]
```

19.12. Similar to the previous command, if P is a point on an elliptic curve, the following command finds its x-coordinate:

```
P[0]
```

19.13. Working together with your partner, find the following data.

- Some prime $p \equiv 3 \bmod 4$
- Some elliptic curve E defined over $\mathbb{F}_p$
- A point P on E with order between one million and ten million.

Post the prime p, the elliptic curve E, and the point P on our class messageboard.

19.14. Now one member of your team, call her Alice, should choose a secret coefficient n_A and post the x-coordinate of $n_A P$ on the class messageboard.

19.15. The other member of your team, call him Bob, should choose a secret coefficient n_B and post the x-coordinate of $n_B P$ on the class messageboard.

19.16. Next Bob should find the y-coordinate corresponding to Alice's x-coordinate. (Use the fact that

$$c^{\frac{p+1}{4}} \bmod p$$

is a square root of c. This is where it is important that we chose $p \equiv 3 \bmod 4$.) Then create a point Q having these (x, y) coordinates and compute $n_B Q$. The x-coordinate of this point will be Alice and Bob's shared secret key.

19.17. To keep that x-coordinate accessible, save it using the format

```
x = 3487384738
```

19.18. Now Bob should choose a secret message M, and encipher it using the x-coordinate that was just found:

```
M = "some message goes here"
enciphervigenere(M,Integer(x))
```

(This is the first part that will not work if you forgot to load the customized commands at the beginning of the lab.)

19.19. Bob should post the enciphered message on the message board. Make sure not to reveal either the secret message M or the encryption key x.

19.20. Alice should find the x-coordinate of the shared secret point, and then decipher the message using

```
M = "the encrypted message goes here"
deciphervigenere(M,Integer(x))
```

19.21. Do not post the decrypted message on the messageboard.

20. RSA DIGITAL SIGNATURE

• You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

`http://bduc-claw9.oit.uci.edu/`

From off-campus, proceed to

`http://uci.sagenb.org/`

• We will need some special functions defined in an external file, and getting Sage to recognize these functions is a three step process.

a. Go to our class website and download the "Week9Code.sage" file, keeping the name the same, including the suffix ".sage". Don't save it as a text file.

b. Back in the Sage worksheet, find the "Data" menu at the top, and choose "Upload or create file". On the next screen, upload the file you just downloaded from our website. I always get an error message, but if you get back to the worksheet screen (and perhaps refresh), it seems to work.

c. Finally, we need to tell the worksheet we want to use that data. Use the command

```
load DATA+"Week9Code.sage"
```

20.1. To make sure those functions loaded correctly, try evaluating

```
enciphervigenere("hello there",Integer(145))
```

20.2. For this lab, it is important that you choose a partner, although the actual interaction will be minimal. Your goal is to post a message from yourself on the EEE messageboard, and also a forged message from your partner on the same messageboard. The key to determining which is genuine and which is forged will be the use of a digital signature.

20.3. Both messages will have the form: an animal, a dessert, and a European city. Although the messages will not be encrypted, they will be transformed into numbers, and so it is important that you post the exact same message on the messageboard as you use in Sage.

20.4. Choose two messages, in the form

```
M1 = "penguin, pumpkin pie, Copenhagen" #choose different ones
M2 = something else in the same form
```

20.5. Turn your messages into numbers using

```
X1 = stringtoint(M1)
X2 = stringtoint(M2)
```

(Note that this will not work if you forgot to load the custom functions at the beginning.)

20.6. Sign your message using your personalized RSA key. In other words, compute

$$X_1^d \bmod n,$$

where $d = e^{-1} \bmod \phi(n)$.

20.7. Anonymously post both your message (in plain English) and your digital signature on the messageboard, together with your name. **Make sure you choose to post anonymously by clicking the appropriate check box in EEE.** In a separate post, either before or after you post for yourself, post the second message, with the same digital signature, and your partner's name.

20.8. Now choose someone else in the class (**not you or your partner**), and determine which of their messages is genuine and which is forged. You will need to use that student's public digital signature key, as available in a separate forum on the EEE messageboard. (This is how digital signatures are used in real life. You find the key from some trusted authority; in this case, that trusted authority is the course instructor.)

20.9. Turn in your answer (the student and which was his/her genuine message) with Homework 8.

21. ELLIPTIC COLLISION ALGORITHM

• You are discouraged from using Internet Explorer. In our experience, Firefox, Chrome and Safari all seem to work better for these labs. From the UC Irvine campus, proceed to

```
http://bduc-claw9.oit.uci.edu/
```

From off-campus, proceed to

```
http://uci.sagenb.org/
```

• We will need some special functions defined in an external file, and getting Sage to recognize these functions is a three step process.

a. Go to our class website and download the "Week10Code.sage" file, keeping the name the same, including the suffix ".sage". Don't save it as a text file.

b. Back in the Sage worksheet, find the "Data" menu at the top, and choose "Upload or create file". On the next screen, upload the file you just downloaded from our website. I always get an error message, but if you get back to the worksheet screen (and perhaps refresh), it seems to work.

c. Finally, we need to tell the worksheet we want to use that data. Use the command

```
load DATA+"Week10Code.sage"
```

21.1. To make sure those functions loaded correctly, try evaluating

```
enciphervigenere("hello there",Integer(145))
```

21.2. The goal of this lab is to use a collision algorithm to solve the elliptic curve discrete log problem, and thus decipher the message exchanged by two of your classmates in Lab 3. The moral of this lab will be that a point with order of approximately one million is not large enough for the elliptic curve Diffie-Hellman key exchange to be secure.

21.3. You may work with or without a partner in this lab, it is your choice.

21.4. Choose an encrypted message from the Lab 3 message forum (not your own!) that you would like to decipher.

21.5. Get all the relevant data from that post, using the format:

```
p = 103
a = 17
b = 2
E = EllipticCurve(GF(p),[0,0,0,a,b])
P = E(14,93)
```

21.6. Recall that the reason to use

```
P = E(14,93)
```

instead of

```
P = (14,93)
```

is to let Sage know this is a point on the specific elliptic curve E. This way, when you write $17 \cdot P$, it will know you really mean: "add P to itself on the elliptic curve 17 times".

21.7. Accessing the points Q_A and Q_B is a little more challenging, because you only have their x-coordinates. Recall that we have

$$y^2 \equiv x^3 + ax + b \bmod p$$

and so, using our square root formula modulo primes which are congruent to 3 modulo 4, one valid y-value is

$$y \equiv (x^3 + ax + b)^{\frac{p+1}{4}} \bmod p.$$

21.8. For example, we can create a point Q_A with x-coordinate 44 using the formula

```
x = 44
QA = E(x,pow(x^3 + a*x + b, (p+1)/4, p))
```

21.9. Similarly create the point Q_B.

21.10. Recall that the Elliptic Curve Diffie-Hellman algorithm is vulnerable if either coefficient n_A satisfying $Q_A = n_A P$ or n_B satisfying $Q_B = n_B P$ becomes known. Thus, you only have to find one of these two coefficients. Then the secret key of Alice and Bob is the x-coordinate of $n_A Q_B = n_B Q_A$.

21.11. Here is sample code performing the collision algorithm, not on the elliptic curve E, but in the multiplicative group $(\mathbb{Z}/p\mathbb{Z})^\times$. Here the goal is to find a secret exponent z such that $g^z \equiv w \bmod p$, where p, g, and w are all known .

```
def mult_collision(p,g,w):
    baby_step = []
    giant_step = []
    # baby step and giant step are the two lists
    # which we will seek a collision between.
    N = Integers(p)(g).multiplicative_order()
    r = floor(sqrt(N))
    # our two lists will both have length r+1
    for i in range(0,r+1):
        baby_step.append(pow(g,i,p))
    # for the baby step list, we simply take powers of g
    for i in range(0,r+1):
        giant_step.append(w*pow(g,-r*i,p))
    # remember that the giant step list is more complicated
    collision_index = collision(baby_step,giant_step)
    # collision_index looks like (13,21), and it indicates the
    # location in the lists where the collision occurs.
    # so this says the 13th item in the babystep list
    # is the same as the 21st item in the giantstep list.
    return collision_index[0] + r*collision_index[1]
```

21.12. Adapt the preceding code to solve the elliptic curve discrete log problem. Here the goal is to find a secret coefficient n_A such that $n_A P = Q_A$. Your function should look basically like the following.

```
def elliptic_collision(p,P,QA):
    baby_step = []
    giant_step = []
    N = P.order()
    r = floor(sqrt(N))
    ...
    return collision_index[0] + r*collision_index[1]
```

21.13. Hint. You might wonder why our inputs to the elliptic_collision function don't include the elliptic curve E (or the coefficients a and b). The point is that Sage knows P and Q_A are points on E, so Sage knows automatically which elliptic curve you are using.

21.14. Hint. We are only looking for a collision between the x-coordinates. If you want to find the x-coordinate of $17P$, use

```
(17*P)[0]
```

21.15. Note that

```
17*P[0]
```

would not work. This is multiplying the x-coordinate of P by 17.

21.16. If you want to test your function using the elliptic curve and points from above, you should find that

```
elliptic_collision(p,P,QA)
```

returns 51. In other words, this is telling you that $51 \cdot P = Q_A$.

21.17. Once you have solved this discrete log problem, the rest will be easy. Find the secret key:

```
x = (nA*QB)[0]
```

21.18. Using that x-coordinate, the message can be deciphered using

```
M = "the encrypted message goes here"
decipherviqenere(M,Integer(x))
```

21.19. Turn in the deciphered message with Homework 8. (Don't post it.) Also turn in the name of your partner, if you worked with a partner.