1) (50 pts) It was mentioned in class that there is no fast solution to the Discrete Log problem. The only method discussed to solve it was to loop through all possible exponents x to see if \( a^x \equiv b \pmod{p} \), given integers a and b and a prime number p. As long as multiplications and mods are assumed to take constant time, this algorithm solves the problem in \( O(p) \) time. However, there is a faster algorithm that is relatively simple (understandable within the scope of ideas taught in this class.) that cuts this run-time down to \( O(\sqrt{p}) \). Here is how the algorithm works:

Let \( n = \lceil \sqrt{p} \rceil \). Then, if there is a valid solution \( x \) to the given discrete log problem, then there will exist integers \( c \) (\( 1 \leq c \leq n \)) and \( d \) (\( 0 \leq d \leq n \)) such that:

\[
 a^{nc-d} \equiv b \pmod{p}
\]

Now, multiply this equation through by \( a^d \):

\[
 a^{nc} \equiv ba^d \pmod{p}
\]

Obviously, doing a double for loop through all possible values of \( c \) and \( d \) will have the same run time, \( O(p) \), as the original algorithm.

But, we can do better by storing a table (map in Java, dictionary in Python) which maps each answer of the form \( a^{nc} \pmod{p} \) to the value of \( c \) that achieved it, and store this map in memory.

Then, for each value of \( d \), we can iteratively compute \( ba^d \pmod{p} \). For each of these answers, see if it is a key in the original map. If so, then this value of \( d \) “matches” with the output value \( c \) produced by the map, which means that the answer to the given query is simply \( nc-d \).

Note: if none of the values produced in the second separate for loop produces a hit in the map, then there is no exponent, \( x \), which satisfies the given query.

Let’s look at a quick illustration with \( p = 11 \), \( a = 2 \) and \( b = 6 \)

For this example \( n = \lceil \sqrt{11} \rceil = 4 \).

In our map, we first store \( 5 \rightarrow 1 \), because \( 2^{4(1)} \equiv 5 \pmod{11} \).

For each subsequent value, we can take the previous value and multiply it by 5, since this is equivalent to \( 2^4 \pmod{11} \). So the rest of the values in our map would be:

\( 3 \rightarrow 2 \), because \( 2^{4(2)} \equiv 3 \pmod{11} \).

\( 4 \rightarrow 3 \), because \( 2^{4(3)} \equiv 4 \pmod{11} \), and

\( 9 \rightarrow 4 \), because \( 2^{4(4)} \equiv 9 \pmod{11} \).

Next, we start a variable = 6, the result we want. Since 6 isn’t in the map, we multiply 6 by the base 2, to get 1 mod 11. This indicates that \( 6(2^1) \equiv 1 \pmod{11} \). (Notice that we are trying to match an answer of 5, 3, 4 or 9.) Next, we multiply this again by 2 to get 2, which is also not in the map.
Next, when we multiply this by 2, we get 4, which is in the map. This indicates that $6(2^{3}) \equiv 4 \pmod{11}$, and our map tells us that $2^{4(3)} \equiv 4 \pmod{11}$, which means that it must be the case that $2^{4(3) - 3} = 2^{9} \equiv 6 \pmod{11}$, solving this instance of the discrete log problem.

Write two functions (in Python or Java) with the following prototypes:

**Python**

```python
def slowDiscLog(base, ans, mod)
def sqrtDiscLog(base, ans, mod)
```

**Java**

```java
public static long slowDiscLog(long base, long ans, long mod)
public static long sqrtDiscLog(long base, long ans, long mod)
```

Your functions should return the smallest non-negative integer $x$ such that $base^x \equiv ans \pmod{mod}$. If no such integer exists, it should return -1. The functions will ONLY be tested on cases that work where mod is a prime number less than 2 billion, base is a primitive root of that prime number and ans is in between 2 and mod-2. (So, I will only test them on “regular” cases, so to speak, and no corner cases.)

The first method should run in $O(mod)$ time, just iteratively exponentiating base and checking if the current value is answer.

The second method should run in $O(\sqrt{mod})$ time, using the algorithm described above.

**Test them yourself on these test cases and provide a table of correct answer and run times of both of your methods for that test case.**
Sample Solution Write Up

The Python code is attached in the file disclog.py and the Java code is attached in the file disclog.java.

Here are the results provided by the Python version:

<table>
<thead>
<tr>
<th>Base</th>
<th>Ans</th>
<th>Mod</th>
<th>X</th>
<th>Slow (sec)</th>
<th>Fast (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>123456</td>
<td>1000000007</td>
<td>333597120</td>
<td>77.11</td>
<td>0.01563</td>
</tr>
<tr>
<td>5</td>
<td>87123456</td>
<td>1000000007</td>
<td>990379224</td>
<td>237.49</td>
<td>0.01560</td>
</tr>
<tr>
<td>211523205</td>
<td>1036204576</td>
<td>1999999973</td>
<td>191948022</td>
<td>54.66</td>
<td>0.03125</td>
</tr>
<tr>
<td>75853221</td>
<td>96317213</td>
<td>1450001227</td>
<td>947032818</td>
<td>259.90</td>
<td>0.03125</td>
</tr>
<tr>
<td>1003708272</td>
<td>1820444653</td>
<td>1910003723</td>
<td>1892024220</td>
<td>540.63</td>
<td>0.01562</td>
</tr>
<tr>
<td>1204331962</td>
<td>505493879</td>
<td>1910003723</td>
<td>1812512929</td>
<td>541.00</td>
<td>0.03124</td>
</tr>
</tbody>
</table>

We can clearly see here that the slow version is MUCH slower than the fast one. Secondly, regardless of the answer, so long as the mod is the same, the fast version has a similar run time (with the curious exception of the 5th case). But, the slow version’s run time scales linearly with the value of X (which makes perfect sense). If we carefully analyze the fast version, we see that there’s some start up cost for storing the initial table, and then we have a second loop which could run more of fewer times, so the explanation for the fifth case is that the second loop cut out very early where as in the other cases, it probably ran a good deal.

Here are the results provided by the Java version:

<table>
<thead>
<tr>
<th>Base</th>
<th>Ans</th>
<th>Mod</th>
<th>X</th>
<th>Slow (sec)</th>
<th>Fast (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>123456</td>
<td>1000000007</td>
<td>333597120</td>
<td>4.000</td>
<td>0</td>
</tr>
<tr>
<td>5</td>
<td>87123456</td>
<td>1000000007</td>
<td>990379224</td>
<td>11.782</td>
<td>0</td>
</tr>
<tr>
<td>211523205</td>
<td>1036204576</td>
<td>1999999973</td>
<td>191948022</td>
<td>2.453</td>
<td>0.00359</td>
</tr>
<tr>
<td>75853221</td>
<td>96317213</td>
<td>1450001227</td>
<td>947032818</td>
<td>11.859</td>
<td>0</td>
</tr>
<tr>
<td>1003708272</td>
<td>1820444653</td>
<td>1910003723</td>
<td>1892024220</td>
<td>24.204</td>
<td>0</td>
</tr>
<tr>
<td>1204331962</td>
<td>505493879</td>
<td>1910003723</td>
<td>1812512929</td>
<td>23.375</td>
<td>0</td>
</tr>
</tbody>
</table>

Note: Obviously the Java version didn’t take 0 seconds. But the millisecond tool I used wasn’t precise enough. To try to get better results, I’ll run each call 100 times and divide that answer by 100 to create this chart.

<table>
<thead>
<tr>
<th>Base</th>
<th>Ans</th>
<th>Mod</th>
<th>X</th>
<th>Slow (sec)</th>
<th>Fast (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>123456</td>
<td>1000000007</td>
<td>333597120</td>
<td>4.000</td>
<td>0.00297</td>
</tr>
<tr>
<td>5</td>
<td>87123456</td>
<td>1000000007</td>
<td>990379224</td>
<td>11.782</td>
<td>0.00281</td>
</tr>
<tr>
<td>211523205</td>
<td>1036204576</td>
<td>1999999973</td>
<td>191948022</td>
<td>2.453</td>
<td>0.00359</td>
</tr>
<tr>
<td>75853221</td>
<td>96317213</td>
<td>1450001227</td>
<td>947032818</td>
<td>11.859</td>
<td>0.00391</td>
</tr>
<tr>
<td>1003708272</td>
<td>1820444653</td>
<td>1910003723</td>
<td>1892024220</td>
<td>24.204</td>
<td>0.00297</td>
</tr>
<tr>
<td>1204331962</td>
<td>505493879</td>
<td>1910003723</td>
<td>1812512929</td>
<td>23.375</td>
<td>0.00328</td>
</tr>
</tbody>
</table>

These times are likely more accurate with the key issue being the accuracy of the currentTimeMillis() method. Also, we note just how much faster using a native long type vs. Python’s built in big integer is.
2) (50 pts) Here is a cipher to break. It is encrypted using one of the classical schemes we learned about in class, which was tested in either Quiz 1 or Quiz 2. Good luck!

mrsrlbfmmgmxlweamktcfclosagvgumvnlxcpkhtkxmzageoacmcsstgrcvslgcrnpqcoaqhdhgcemhmhmdhalgdciasotblrsqmgnhlkcsszhrvgkhgmocmnchruahgcoegvbcxhungmtrcmrkagirvogmeuageavqgnxckscimlprpxrtnvpmdetnovulxxgmlurpmgdhavgrycshvhargdmcecxvvpconhcfvzhafhdw

As always, discuss your whole process. What clues did you use to try to figure out what cipher was used. Describe what you did to try to break the cipher, and hopefully, determine both the plaintext and the key! (If you do, please include these in the write up.)

**Sample Solution Write Up**

When we look for digraphs, one of the things we notice is that there are no double letters. In addition, the letter frequencies don’t quite seem to have the same variation as regular English letters. Here are the frequencies as printed out by cryptotool.html on the course website:

- A 5.7% N 3.7%
- B 1.2% O 3.7%
- C 9.8% P 2.8%
- D 3.7% Q 1.2%
- E 3.7% R 6.5%
- F 1.6% S 4.1%
- G 9.8% T 2.8%
- H 8.1% U 2.4%
- I 0.8% V 5.7%
- J 0% W 0.8%
- K 2.8% X 3.7%
- L 4.5% Y 0.4%
- M 9.3% Z 1.2%

While some letters occur more often than others, there seems to be a more flat distribution (several letters in the 4% to 6% range) than the regular English language. There are 246 characters, which is an even number, and 48 digraphs appear more than once. Many of these are clues that the cipher used might be Playfair. (Letter frequencies are not flat enough to be Vigenere and as previously mentioned, they also don’t resemble Substitution. Finally, Hill tends to create more even frequencies, so Playfair seems to be a good first guess.)

Let’s look at the digraph frequencies (created by p2.py):

- 7 GM
- 4 HG
- 4 AG
It’s likely that GM is either TH or HE, with the latter being a much better bet, since it’s quite possible that G and H are on the same row of the square, with neither being in the keyword. So, our first arrangement has the GM \(\rightarrow\) HE. If this is a box, then we have something like:

E M  
G H

Which would insinuate that both E and M are in the keyword while G and H are not.

If we start looking at the digraphs with 4 repetitions (ignoring CS for now), we have:

4 HG  
4 AG  
4 HA

These are 3 frequent digraphs involving the same three letters. So it’s possible that ‘A’ is on the same row as G and H. If A is not in the keyword, then it would have to be to the left, so something like this:

.. E M  
A .. G H

One thing that is strange about this picture is that it would put B, C, D, and F (or many of those letters) in the keyword above.

So an alternate picture might have G and H aligned differently, but A almost definitely shares a row with either G or H. Since HA is more common in English than GA, let’s try HA in the keyword with G below the H:

H A  
G  
E  
M

In this case, E would have to be in the same column so HE \(\rightarrow\) GM. In this arrangement, E and M are not in the keyword, but H and A definitely are, and maybe even G.

Since there are 4 copies of HA in the ciphertext the only way to get HA is for there to be a letter to the left of the H. The most common diagram with H as the second letter is TH, so we have:

T H A  
G  
E  
M

A huge lucky break might be that THA is the start of THANKSGIVING and this question was made very close to that holiday. If we make that crazy guess, we have the following so far:
Then we might expect B, C and D before E but one of those letters is missing. Either our guess of starting with Thanksgiving is wrong OR the keyword is even longer. If we assume that W, X Y and Z aren’t in the keyword, we have:

```
THANK
SGIV
E
M
UWXYZ
```

There is one letter missing between M and U because we don’t have O, P, Q and R in the keyword. This is stronger evidence that maybe the keyword is longer. A natural word to follow Thanksgiving is holiday or break. Holiday doesn’t work because the ‘E’ gets in the way. Let’s try break:

```
THANK
SGIVB
RE
M
UWXYZ
```

Now, if we do the math, the rest of the letters fit in perfectly:

```
THANK
SGIVB
RECDF
LMOPQ
UWXYZ
```

Using this square, we are able to obtain the plaintext (using Crypttool)

```
LET’S SEE HOW MUCH HARDER THIS IS WHEN YOU DON’T KNOW WHICH CIPHER IS
USED. I TRIED TO PICK ONE WHERE WE WENT OVER IN CLASS THE ATTRIBUTES IN
THE CIPHER TEXT WHICH GIVE AWAY THE USE OF THIS CIPHER, WHICH IS PLAYFAIR.
GOOD LUCK AND PERHAPS YOU WILL SOLVE THIS DURING THE TIME PERIOD
INDICATED BY THE KEY.
```

Note: There’s a ton of guesswork in this solution. There are probably better ways with fewer guesses to break this cipher that dig deeper into the characteristics of the digraphs. Also, I apologize for misspelling cipher the first time around in the plaintext! That was not intentional.
3) (50 pts) This is another cipher to break. It doesn’t have a key, per se, but the system used is a “fixed” system, with the key embedded in it. In particular, the character at ciphertext position i (1 based) is shifted by f(i+1) characters, where f(i+1) is some function with an integer output. The function is one that is based on a famous number sequence. Good luck!

qtlrkvdxptjmvoeyiyhszegnrzovxzphplqojwiatpkiduxeswjdupygled
uqgeorzdmnmfgpjxsqzfvbcljixjjiwrldfbfaiadcsvyvvhjemadoiwm
dghrzqvbvtruwegmrriiyftlnqgwaltmriobayesqjjiaqmaqifntbd
shkxgajsovnnvtipwggxrn

As always, discuss your whole process. What clues did you use to try to figure out how each character was shifted. Describe what you did to try to break the cipher, and hopefully, determine both the plaintext and what the function f(x) used for creating the shift for each character is! (If you do, please include these in the write up.)

**Sample Solution Write Up**

Since sequences tend to start with small numbers, we might try to guess the first word by subtracting small (but increasing) numbers from the plaintext.

‘Q’ – 1 = ‘P’  
‘T’ – 2 = ‘R’

This looks pretty promising. The third letter would have to be ‘I’:  

‘L’ – 3 = ‘I’

The sequence starts 1, 2, 3. It would probably be too easy if the sequence was just the positive numbers. (And trying that fails…)

If the 4th term is 4 we have ‘R’ – 4 = ‘N’.  
If the 4th term is 5 we have ‘R’ – 5 = ‘M’

Words that start “PRIN” are “PRINCESS”, “PRINT”. In fact, a quick search of a standard dictionary yields that the only common letters to be the fifth letter are ‘C’ or ‘T’. If ‘N’ is the 4th plaintext letter and ‘C’ the fifth, the fifth shift would be ‘K’ – ‘C’ = 8, so the corresponding sequence would be 1, 2, 3, 4, 8. If T were the fifth letter, ‘K’ – ‘T’ = 17 (under mod), so the corresponding sequence would be 1, 2, 3, 4, 17. Both seem strange.

Let’s try ‘M’ as the fourth letter: “PRIM”. Based on the course, a great guess for the fifth letter would be ‘E’ and ‘K’ – ‘E’ = 6 would be the next shift.

So, in this scenario the shifts are 1, 2, 3, 5, 6.

A potential follow up to prime would be number. So, let’s guess that

**VDXPTJ decrypts to NUMBER**
If this is the case, the next few shifts would be:

V – N = 8  
D – U = 9  
X – M = 11  
P – B = 14  
T – E = 15  
J – R = 18

So, this gives us the decryption sequence for the function f:

1, 2, 3, 5, 6, 8, 9, 11, 14, 15, and 18

This sequence seems pretty dense at first, but none of the numbers themselves look recognizable. The plaintext seems to indicate that prime numbers are involved somehow. But if they were, the shifts might be 2, 3, 5, 7, 11, 13, 17, 19, 23, 29, 31, 37, etc.

Let’s try looking at the list of primes and the sequence numbers:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>11</th>
<th>14</th>
<th>15</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
<td></td>
</tr>
</tbody>
</table>

All of the prime numbers are odd except for the first, and having the 2 in the table seems to make it difficult to find a pattern. Also, notice that the hint said f(i+1) instead of f(i). So maybe the first shift is based on the second prime number instead of the first. Let’s try again:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>11</th>
<th>14</th>
<th>15</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>5</td>
<td>7</td>
<td>11</td>
<td>13</td>
<td>17</td>
<td>19</td>
<td>23</td>
<td>29</td>
<td>31</td>
<td>37</td>
<td></td>
</tr>
</tbody>
</table>

If we look at the columns the bottom number is barely more than twice the top number. In fact, let’s subtract 1 from each bottom number:

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>6</th>
<th>8</th>
<th>9</th>
<th>11</th>
<th>14</th>
<th>15</th>
<th>18</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>4</td>
<td>6</td>
<td>10</td>
<td>12</td>
<td>16</td>
<td>18</td>
<td>22</td>
<td>28</td>
<td>30</td>
<td>36</td>
<td></td>
</tr>
</tbody>
</table>

AHA!!!!

Looks like the shift for the \( i^{th} \) character is \( (p_{i+1} - 1)/2 \), where \( p_i \) denotes the \( i^{th} \) prime number!

We can write a prime sieve to generate primes up to say 10,000 (should be enough) and then use this formula to decrypt:

**Prime numbers are very important to cryptography, so I thought I would use them in my special code. To try to make it harder since all the numbers are odd after two, I decided to divide the numbers by two but if you are reading this, you already know that.**