Alex Hortin
PA1

1) Cyptographic Checksum of your Source Code (5 pts)
Compute an md5 or SHA1 checksum of a tar file (or similar archive) of your source code and
executable image. Include the name of this file, it's size, and it's checksum in your write-up.

To compute the MD5 sum of my Work, I used the 'md5sum' command line utility in ubuntu on my source code file.

Name	Size	MD5 Checksum
cpts425p1Hortin.tar.gz	1933b	179c03b878eb6e4298959baadc7bc385

2) Abbreviated Software Design Document (10 pts)
Briefly describe the data structures and algorithms used to implement the three required tools. (1-2 page limit)

This first thing I had to do when considering this assignment was choose a language that I was not only comfortable with, but also one that had functionality that would be useful for me in the assignment. There were three languages I considered; C, Python, and Perl. I decided not to use C, because string processing in C often involves implementing a lot of functionality yourself that Python and Perl include as defaults. I ended up choosing Python since I have taken a few classes that allowed me to use it, and I am very comfortable with built in functionality that would be useful toward the completion of this assignment.

The main data structures I used in python were Lists, Strings, and Tuples, Files, and the sys.argv field. I used the argv field in all the programs to determine what the arguments were for eah file, and than open the proper files for read and write, as well as set user specified options for depending on each program. In the first program *caesarCipher* I was able to get through it using just characters. It was fairly trivial to read in a file character by character, find its ascii value with the ord() cast, shift it with the key, than write it out to a cipher file. There were really no complicated algorithms here that I had to implement because Python had a lot of the needed functionality. I did use modulo to properly shift the ascii values. It was also necessary to subtract 32 due to the offset in the ascii table, and than re-add 32 after the shift was applied before recasting as an char.

Lists were very useful for storing tuples of information based on the data retrieved by *charFreqGen*. I stored a list of tuples that stored (Frequency of Character,Character). For example the entry for character 'A' would look like [.....,(.07,A),.....]. Since the information needed to be sorted, I was able to use the default sort method (sorted()) to sort all the entries by their first entry in the tuple. The string processing included in python was useful because I was able to use the built in count methods to generate the frequency for each character, as well as using the lower() method to make the frequency case insensitive before I calculated their frequency.

Out of all the applications *caesarAttack* had the most interesting code challenges. I had to read and process data that was contained within another output, and parsing it was a challenge due to the printable characters represented within. I decided to use the a split(" ") on each line, which left a problem when the actual " " char was supposed to be added to the dictionary. Luckily I hacked a solution, and was able to code in a special case. Than I

generated f(c) by iterating through the given ciphered file and finding the occurrences and entering them into a dictionary with the char as the key. Lists worked fine for the frequency generator but in *caesarAttack* I wanted to look up data fast without iterating through the array. I read the data into dictionaries so I could look up key values quickly to calculate phi. It was a matter of just inputting the proper key frequency when calculating phi for f(c), and than for p(c - i) I needed to apply the proper shift. To calculate phi for each p(i), I used a simple for loop that iterated through all possible values 0-95, and ran them through the ciphered code calculating each chars component (f(c)p(c-i)) and adding it to a persistent variable to generate the total phi. The dictionary made it very easy to look up these values with just their character value. After I had a list of all Phi's, I sorted it, and output the list to a file. Than using the best one I output the data in the input cipher using the most likely (highest phi) i as a shift to the cipher.

I was very pleased with my decision to use Python for this assignment, because using a C like language would have created much more work. I believe that the algorithms that I designed are very simple, and benefited from the extensive libraries that Python has

3) Software Test Document (10 pts)
Describe your software test plan and methodology that you used to verify that you implemented the three required software tools correctly. List any know deficiencies. (1-2 page limit)

The test plan I used for this assignment was, at it's simplest level, testing each separate component, than moving onto the next after I was satisfied it was correct to the standards of the assignment. At a deeper level I structured working on each program on a tiered basis so I would be able to test 1 part of the program than move onto the next part, confident that I had eliminated most of the large bugs that might hinder my code later.

At the individual level for the *caesarCipher* application I started off by implementing my command line parsing to ensure that later on I would have all files available to me that the program required. I implemented it in such a way that the arguments could be given in any order, and if one was missing the program would output a usage message to the user. Once this was fully tested I wrote the rest of my program and than tested my encode and decode. During testing encode I used examples that were in the book as tests to make sure it was encoding right. For decode I simply used my encode test files and decoded them. Than I used diff -s to see if they were identical. Once they were identical I moved on to the next program

The next program that I implemented was the *charFreqGen* program. This was one that was fairly easy to test, since I had already tested out the command line switch code, and needed to make a few switches to adapt it for use here. After that I implemented the code, which turned out to be very little on my part due to the count string method in python. After I had all the file output working I simply hand counted a few files, than ran it through my program, and checked to see if the frequencies generated were the same. I ran into a few problems that actually led me to test out my *caesarCipher* program and find some errors their that I needed to fix. After I found the frequencies matching, I concluded that this program met the specifications.

Finally I was ready to work on the bulk of the *caesarAttack* program, and was fairly confident that the other pieces of the assignment which input into this program were complete. To begin I tested the input of the frequency file into a dictionary data structure, as well as the cipher file into a data structure. Once these were both in their respected data structures I was able to test the functionality of generating phi. This was the most difficult to test because I had to shift the number multiple times to get a correct i offset that would

generate proper values. I ended up going through a few iterations of writing this formula before I finally got it. Than to test whether or not Phi was correct I checked if the highest phi(i) was the match for i = the key. Once this was correct I output the cipher text while applying the proper value for i to each letter to shift it into it's proper place.

After I was sure that each file was tested well, I ran the demo.sh script to make sure it ran, and was pleased when it worked the very first time. I than independently tested the program a few more times to make sure using my own files as tests. I ran into no problems with this.

4) Lessons Learned / Project Evaluation (10 pts)
Discuss the lessons learned from this project. Are there improvements that should be made in a future release? During testing, did your real-world input data files generate similar character frequencies? Where you able to always predict the correct key on your first try? Why/Why not. Other possible questions to address: Was this assignment as easy as you anticipated? Did implementing this cipher inspire you to become a cryptanalyst? Etc.

The biggest lesson I think I took away from this is how vulnerable encryption can be with simple math. I also think that the case sensitive issues that we discussed in class were interesting, because it never occurred to me that something like that would be important. I had never thought about using large pieces of test to generate a frequency chart indicative of a whole set of characters than comparing a ciphered file. It makes sense why RSA (and others) were deemed necessary.

As far as improvements, I think that the caesarAttack has a lot of room for improvement. There could be the ability to check matches with given keys to a dictionary of common words to generate another correlation rate. Words like (the, and, my, date,...,etc) could be searched for after a decryption to automate the process.

The key could not always be predicted on the first try, unless you were using the same file to generate the freq file and the cipher file. If you don't have this the whole assignment becomes based on probability of unrelated files. For example during my real world testing I tested a German document against the dissertation.txt and it was not certain there was a heavy correlation at all. A document with statistical data would also be hard to break using this. Say that an intercepted documents was all numbers (hex would be worse) with meaning...this kind of statistical attack would be pretty hard to figure out the order of those numbers.

I thought that the first two part of this assignment was pretty trivial, but the third part I actually though a lot about. I had to think through the data structures that I was going to use very carefully to make sure that I didn't start with something that would make more work for me later. That is why I rewrote it at one point using dictionaries instead of lists. It turned out to save me a lot of time.

I also learned what md5 sums were used for and how to find them. I had a vague idea of what they were used for, but I never had ever checked the sums for a file, and had no idea how to do it. Next time I download something off of source forge I will probably check the md5 sums just to be sure.

This assignment was interesting because I have not done much cryptography work, but I would never want to become a cryptanalyst. I enjoy programming, but the heavy statistics background needed to be successful, and the current state of encryption seems overwhelming. I have read the Neal Stephenson book Cryptonimicon which is about

cryptography, and after that decided it was not the field for me.

I also think that Python is an extremely powerful language, and I would love to continue learning more about it. The one problem I had were inconsistencies within the lab computers and my own, so I had to modify my code to adher to the version that they had. This assignment let me refine my knowledge of how to use some of the more difficult data structures. It was a great refresher on python and I hope to use it for future assignments in this class.