Genetic Algorithms and Substitution Ciphers

Eckel, TJHSST AI2, Spring 2021

Background and Explanation

In the story of AI, we're about to make a huge leap. So far, we've been coding a plan, algorithm, or strategy ourselves, and then letting the computer implement it (albeit much faster than we ever could). This assignment marks a change: now, we're giving the AI the tools to evaluate itself so it can *find it's own best plan, algorithm, or strategy*.

To start, we're going to look at a classic problem from Cryptography, and write a naïve "solution" that almost, but doesn't quite, work. Then we will learn how a technique called a genetic algorithm solves it for real.

Our Goal: Cracking Substitution Ciphers

A substitution cipher is a way of encoding text that has been used for centuries, though no one would actually use it to protect sensitive information now (as this lab will make obvious!) It works like this. Take the alphabet and scramble it, rewriting it underneath the original alphabet. This produces a source alphabet and cipher alphabet, like so:

```
ABCDEFGHIJKLMNOPQRSTUVWXYZ
XRPHIWGSONFQDZEYVJKMATUCLB
```

To encode a message, simply replace each letter with the corresponding letter in the cipher alphabet. Using this particular cipher, `"HELLO"` would be encrypted as `"SIQQE"`.

So: what if you have a message encoded by a substitution cipher, and you want to crack it, recovering the original message without knowing the cipher alphabet? There are 26! = 403,291,461,126,605,635,584,000,000 possible substitution ciphers. Given an encoded block of text, a brute force solution is obviously not feasible!

Traditionally, a substitution cipher is cracked using frequency analysis (for instance, the most common letter is likely to be "E") and then judgment/knowledge as you, the human decoder, look for fragments that look like words and try possibilities until the message becomes clear. In the cryptography class I taught at TJ for a couple years, I taught this method, and most students (working in pairs) could do 5 messages in a 95-minute class period. That's clearly too much work; let's find a way for a Python script to do this for us almost instantly instead!

Specifically, here's our goal in this assignment: we want to **start with an encoded message and have our script figure out the decoded message without any intervention on our part**. Enter encoded message, click "run", decoded message pops out.

First: Encode and Decode

Before cracking, we first need to implement the cipher in the first place. So: your first task is to write functions to encode and decode a given block of text using a given cipher alphabet. (You can assume the source alphabet is always in A-Z order, so a cipher can be uniquely defined by a single 26-character string containing the scrambled cipher alphabet.) When implementing this, you should make all letters uppercase, and you should ignore any non-letter characters. So, for example, using the cipher above, the string `"Hello, students!"` would be converted to the string `"HELLO, STUDENTS!"` and be encoded as `"SIQQE, KMAHIZMK!"`, preserving the comma, space, and exclamation point but encoding each letter.

Test this thoroughly before you move on; encode and decode a variety of messages with different alphabets, ensuring you return to the message you started with each time.

Next: A Fitness Function for Substitution Ciphers

In order to decode a message automatically, we need to give the computer a way to judge its own results so that it can improve upon them. This is called a *fitness function* – a function that examines the results of an algorithm or process and decides how good those results are, without the user needing to intervene.

Now, it's not so hard to make a program that determines if text *literally is English* or not; just check the words against a list of English words and see how many of them show up! A much harder question to answer is this: what makes text *English-like*? In other words, if I start from an encoded message and "decode" it using a guess at the decoding cipher alphabet, how can I give a score to how *close to English* that "decoded" message is?

This is a fun problem to think about; you may wish to take a moment to ponder before you read on!

Using n-gram Fitness

Define an n-gram to be a consecutive sequence of n letters inside a word. So, for example, the word "example" has four different 4-grams: "exam", "xamp", "ampl", and "mple". The word "example" has five different 3-grams: "exa", "xam", "amp", "mpl", and "ple". Etc, etc.

Obviously, for any value of n, some n-grams will occur more than others. As it turns out, measuring the relative frequency of occurrence of n-grams in some text and comparing it to the frequency of those n-grams in real English makes an outstanding way to measure how English-like a string of text is!

So: here's how we'll measure the fitness of our "decoded" message. We will decide on a value for n and then take our string of "decoded" text and find all the n-grams in it. We can then score our "decoded" text by finding the frequency of each of those n-grams in English and adding them together. The closer we get to English, the more frequently high-likelihood English n-grams will also occur in our "decoded" text, and the higher the sum should be.

Or, well, almost; it does turn out that there's a problem with this approach, which is that some n-grams in English occur *way, way* more than others, and so a wrong answer that happens to create one or two common n-grams by mistake can overwhelm a correct answer with many more n-grams that are less frequent. This problem can be solved by summing the *log* of each frequency number instead of the frequency numbers themselves, as this shrinks the gap between most frequent and less frequent n-gram scores. Of course, in addition to choosing the value for n, we will have to pick the base we want to use for the log. My best results usually came from using log base 2 and counting 4-grams, though when a ciphertext contains a lot of short words, sometimes 3-grams work better. So we'll want to give ourselves options.

To summarize:

- **The fitness function should take a value of n, an encoded text block, and a candidate cipher alphabet**
- **It should "decode" the text using the substitution key defined by the cipher alphabet**
- **It should find each n-gram in the "decoded" text, find the real English frequency of that n-gram, and take the log of that number**
- **It should return the sum of all of those calculations**

Here are some key details for making this work:

- Obviously, you'll need information on real n-gram frequencies in English! On the course website, you'll find ngrams.txt. For values of n from 1 to 5, this contains every n-gram that appeared at least 100000 times in a multi-million word corpus of English. Each line pairs one n-gram with number of times that n-gram occurs (its frequency). So, for instance, you can see that "E" is the most common letter, "TH" is the most common 2-gram, etc. Use this as the source of your frequency numbers.
- To take logs in python: `from math import log`. The imported "log" function takes two arguments – the number and the base, in that order. (ie, log(8, 2) = 3.)

- The text that we use will contain spaces and grammatical characters. The best way of dealing with this is to loop over your string and take every n-character substring, but only use the ones that are actually n-grams (ie, comprised only of letters). So, as you consider each n-character substring, check to make sure the substring contains only alphabetic characters; if not, simply discard it. This means that words of length less than n will be totally ignored, and contractions like "don't" will similarly cause characters to be skipped. This feels like it should be a problem but will turn out not to be; don't find some kind of clever solution, you won't need it.
- Along those same lines, note that ngrams.txt does not contain every possible ngram; if your ngram does not appear in the list, skip it. This won't be completely accurate, but again, it turns out not to matter!
- Here's a test case to make sure you've got this right. Assuming the following text is a "decoded" block (ie, no need to run it through a deciphering process first), the fitness score of the following text should be 874.2333188683015 using 3-grams or 121.44907724101577 using 4-grams. (It's not surprising that the score is lower for 4-grams; there are more 4-grams than 3-grams, so 4-gram frequencies are lower in the first place.)

```
XMTP CGPQR BWEKNJB GQ OTGRB EL BEQX BWEKNJB, G RFGLI.  GR GQ BEQX
ABSETQB RFGQ QBLRBLSB TQBQ EJJ RBL KMQR SMKKML VMPYQ GL BLDJGQF:
'G FEUB RM AB E DMMY QRTYBLR GL RFER SJEQQ GL RFB PMMK MC RFER
RBESFBP.'
```

A Naïve Solution: Hill Climbing

In general, given any partially successful strategy (specifically in this case, partially successful cipher alphabet), it's often possible to improve the strategy by making random changes, testing the result of those changes, and keeping only the better outcomes. This is called **hill climbing**. This is relatively simple to code, but doesn't work in every situation – it can often get caught in a local maximum, a strategy that is not ideal but from which any single change is not an improvement on its own.

Let's implement hill climbing for this problem. You'll get to watch this phenomenon happen!

Write a function that will:

- Take an encoded message
- Start with a random permutation of the alphabet as a candidate cipher alphabet
- "Decode" the message using that cipher alphabet
- Score the results, using the fitness function described above
- Then: loop infinitely. Randomly swap a single pair of letters in the cipher alphabet, and repeat the decoding/scoring process, keeping the change if it results in a higher score and ignoring it if the score does not improve. Each time, print the "decoded" text out to the console; you can watch the hill climbing process unfold.

In other words, we'll start at a random one of the 403,291,461,126,605,635,584,000,000 possible substitution ciphers, and then "climb" from there, making one random change at a time and seeing how close to English we get.

Try applying this to the encoded message earlier on the page. If your code is like mine, you'll get things that seem closer to English, but also clearly not decoded properly. On five runs of this, I let the code run until the answer seemed "stable" (that is, unchanging for a while) and then halted the process. These were the five results I got:

- `POLG JUGST EBACHME US FLUTE AN EASP EBACHME, U TRUNK.  UT US EASP VEDALSE TRUS SENTENDE LSES AMM TEN COST DOCCON ZOGIS UN ENYMUSR:  'U RAWE TO VE A YOOI STLIENT UN TRAT DMASS UN TRE GOOC OJ TRAT TEADREG.'`
- `BAGC FOCUT EWISHME OU QGOTE IL EIUB EWISHME, O TROLD.  OT OU EIUB PENIGUE TROU UELTELNE GUEU IMM TEL SAUT NASSAL JACKU OL ELYMOUR:  'O RIVE TA PE I YAAK UTGKELT OL TRIT NMIUU OL TRE CAAS AF TRIT TEINREC.'`

- DLHU VAUGT EYNOMIE AG FHATE NS ENGD EYNOMIE, A TRASB. AT AG ENGD XECNHGE TRAG GESTESCE HGEG NII TES OLGT CLOOLS KLUWG AS ESPIAGR: 'A RNZE TL XE N PLLW GTHWEST AS TRNT CINGG AS TRE ULLO LV TRNT TENCREU.'
- MONC DYCAT EGULFIE YA WNYTE US EUAM EGULFIE, Y TRYSV. YT YA EUAM ZEBUNAE TRYA AESTESBE NAEA UII TES LOAT BOLLOS POCKA YS ESHIYAR: 'Y RUXE TO ZE U HOOK ATNKEST YS TRUT BIUAA YS TRE COOL OD TRUT TEUBREC.'
- SIBU WOULD EXAMPHE OL YBODE AN EALS EXAMPHE, O DRONZ. OD OL EALS FETABLE DROL LENDENTE BLEL AHH DEN MILD TIMMIN KIUGL ON ENCHOLR: 'O RAVE DI FE A CIIG LDBGEND ON DRAD THALL ON DRE UIIM IW DRAD DEATREU.'

You should see that this is indeed more English-like than it started; occasionally, an actual word even appears, like TRUNK or WOULD. But we quickly get to a place where the result stabilizes and no further progress is made. This is what I meant earlier by local maximum – we arrive at a place where swapping any two specific characters results in a string that is *less* English-like, even though clearly we need to swap many characters before a correct answer arrives!

Hill climbing isn't an entirely terrible idea, though. In fact, hill climbing will even totally work… *sometimes*. Take this block of text, which is much longer, and try it in your hill climbing algorithm; I get much better results with this one:

```
PF HACYHTTRQ VF N PBYYRPGVBA BS SERR YRNEAVAT NPGVIVGVRF GUNG
GRNPU PBZCHGRE FPVRAPR GUEBHTU RATNTVAT TNZRF NAQ CHMMYRF GUNG
HFR PNEQF, FGEVAT, PENLBAF NAQ YBGF BS EHAAVAT NEBHAQ. JR
BEVTVANYYL QRIRYBCRQ GUVF FB GUNG LBHAT FGHQRAGF PBHYQ QVIR URNQ-
SVEFG VAGB PBZCHGRE FPVRAPR, RKCREVRAPVAT GUR XVAQF BS DHRFGVBAF
NAQ PUNYYRATRF GUNG PBZCHGRE FPVRAGVFGF RKCREVRAPR, OHG JVGUBHG
UNIVAT GB YRNEA CEBTENZZVAT SVEFG.  GUR PBYYRPGVBA JNF BEVTVANYYL
VAGRAQRQ NF N ERFBHEPR SBE BHGERNPU NAQ RKGRAFVBA, OHG JVGU GUR
NQBCGVBA BS PBZCHGVAT NAQ PBZCHGNGVBANY GUVAXVAT VAGB ZNAL
PYNFFEBBZF NEBHAQ GUR JBEYQ, VG VF ABJ JVQRYL HFRQ SBE GRNPUVAT.
GUR ZNGREVNY UNF ORRA HFRQ VA ZNAL PBAGRKGF BHGFVQR GUR PYNFFEBBZ
NF JRYY, VAPYHQVAT FPVRAPR FUBJF, GNYXF SBE FRAVBE PVGVMRAF, NAQ
FCRPVNY RIRAGF.  GUNAXF GB TRAREBHF FCBAFBEFUVCF JR UNIR ORRA
NOYR GB PERNGR NFFBPVNGRQ ERFBHEPRF FHPU NF GUR IVQRBF, JUVPU NER
VAGRAQRQ GB URYC GRNPUREF FRR UBJ GUR NPGVIVGVRF JBEX (CYRNFR
QBA'G FUBJ GURZ GB LBHE PYNFFRF – YRG GURZ RKCREVRAPR GUR
NPGVIVGVRF GURZFRYIRF!). NYY BS GUR NPGVIVGVRF GUNG JR CEBIVQR
NER BCRA FBHEPR – GURL NER ERYRNFRQ HAQRE N PERNGVIR PBZZBAF
NGGEVOHGVBA-FUNERNYVXR YVPRAPR, FB LBH PNA PBCL, FUNER NAQ ZBQVSL
GUR ZNGREVNY.  SBE NA RKCYNANGVBA BA GUR PBAARPGVBAF ORGJRRA PF
HACYHTTRQ NAQ PBZCHGNGVBANY GUVAXVAT FXVYYF, FRR BHE
PBZCHGNGVBANY GUVAXVAT NAQ PF HACYHTTRQ CNTR.  GB IVRJ GUR GRNZ
BS PBAGEVOHGBEF JUB JBEX BA GUVF CEBWRPG, FRR BHE CRBCYR CNTR.
SBE QRGNVYF BA UBJ GB PBAGNPG HF, FRR BHE PBAGNPG HF CNTR.  SBE
ZBER VASBEZNGVBA NOBHG GUR CEVAPVCYRF ORUVAQ PF HACYHTTRQ, FRR
BHE CEVAPVCYRF CNTR.
```

The decoded text begins "CS UNPLUGGED". If your experience is like mine, if you try a few times your code will probably get it.

So: this hill climbing thing isn't a complete dead end, there's potential in this idea, but it's not reliable for this problem. We can absolutely do better. And that's where genetic algorithms come in.

Before you continue, please **send me a message on Mattermost** where you tell me your hill climbing results. Were you able to get the first example, even though I wasn't? Were you able to get CS UNPLUGGED? I'm curious!

Genetic Algorithms

Genetic algorithms are a powerful tool for improving on the idea of hill climbing.

Before continuing, please **watch Video 05-01: Genetic Algorithm Overview on the course website to learn how this type of algorithm works and why it addresses the local maximum problem.** The rest of this section summarizes the conclusions of the video, but does not explain the logic and is not a replacement for watching.

To review, the components of a genetic algorithm are as follows:

1. A **population** of strategies. Instead of improving one strategy, genetic algorithms create a large set of strategies and recombine them in each generation.
2. A **fitness function** that evaluates the success of any particular strategy.
3. A **selection method** that makes better strategies (ie, strategies that return more desirable values from the fitness function) more likely to breed. To increase genetic diversity, it shouldn't be true that the very best strategies are the *only* ones that breed, but better strategies should be more *likely* to.
4. A **breeding process** that selects characteristics from two different strategies to form a new child strategy.
5. A small chance of a **mutation**, an unpredictable change in one of the strategy's variables, as each new child is created.

The algorithm successively creates new **generations** of strategies. We'll have a target population count, and then we'll successively pick pairs of parents from the current generation using the selection method. Each pair will use the breeding process to produce a child, which then may or may not mutate, and which is then added to the next generation. (We won't add the same child twice.) We keep doing that until the next generation is of the desired size.

IMPORTANT NOTE: A common source of confusion comes from selecting parents. **The same strategy may end up being a parent many, many times in the same generation** – we are not "using up" strategies in our current generation when we select them to be parents for the next generation. Out of our whole current population, we use our selection method to select two parents, then they breed. Then we choose again **out of our whole current population, including the parents that were just chosen**.

This process has the effect of simulating something much like natural selection (survival of the fittest) in nature, allowing a diversity of strategies that helps avoid getting stuck in a local maximum while still generally improving. Genetic algorithms are remarkably effective at some problems that seem untenable otherwise!

Using a Genetic Algorithm on Substitution Ciphers

There are a lot of decisions to make when implementing a genetic algorithm. As this is our first exercise, I'll give a set of decisions that definitely works along with a list of parameters you can vary to produce better results.

If you would like to try a different implementation, feel free! I can guarantee that my implementation below will work, but there are certainly other valid solutions. If you can find something better I'd love to hear about it!

In any case, here's one implementation that works. Let's go step by step.

1 – Population

This is straightforward – just **generate as many random permutations of the alphabet as you need**, discarding duplicates. (The odds of generating duplicates randomly is hilariously low, but that's a general principle that I want to emphasize throughout the process – any single generation should never contain duplicates.)

We will need to choose our `POPULATION_SIZE`.

2 – Fitness Function

Use the fitness function you've already coded! We've got this part already.

3 – Selection Method

We want better strategies to be selected more often, but we also want a healthy element of randomness. It's a bad idea to just pick the best two strategies to breed all the children themselves; our population will quickly converge and get stuck at what is most likely a local maximum but not an ideal solution, just like hill climbing often did.

The video describes the clones / tournaments breeding process and the motivation behind it; to summarize again:

- Create a list that will represent the next generation; begin with that list empty.
- Rank all the strategies in the current generation, and copy some number of the very highest-ranked ones directly into the new generation. Do not remove them from the current generation; they can still be parents (in fact, the purpose of cloning is so they can be parents *more*, in the next generation as well as this one.) We'll need to decide how many clones we copy each time, a parameter we'll call `NUM_CLONES`.
- We'll need to decide on a size for each tournament, a parameter `TOURNAMENT_SIZE.`
- Then, to generate each remaining child in the new generation:
    - Randomly select two **distinct** subsets of the current generation, two "tournaments", each one the given size. (Or, in other words, randomly select 2 * TOURNAMENT_SIZE different members of the current population and split them into two separate tournaments.) This is totally random; every member of the current generation has equal likelihood of being chosen.
    - Within each tournament, rank the chosen strategies by fitness score. (**IMPORTANT: calling your fitness function again on any strategy each time a strategy appears in a tournament will slow your code down tremendously.** Avoid repeated work by scoring each strategy ONCE per generation and storing that score somewhere you can look up instead of recalculating.)
    - With probability given by TOURNAMENT_WIN_PROBABILITY, choose the most fit strategy. If the strategy isn't chosen, discard it; move on to the next best strategy, and choose it with the same probability. If it isn't chosen, discard it; etc, etc.

For example, if our probability is .75, we will call random.random() and if we get a number less than .75, we will select the actual winner (ie, the strategy in this tournament with the highest fitness function score). If we **don't** select the winner, that is if the random.random() call is greater than the chosen probability, we'll discard that strategy, moving to the next strategy down the list. Then we'll repeat the process – we'll call random.random() again and if **that** number is less than .75, we'll select the second place strategy. If not, we'll move on to the third… etc, etc.

4 – Breeding Process

Once we select the two parents to breed, we need to generate a child. This process should *also* have a healthy element of randomness – even if it occurs that the same exact pair of parents is selected twice, they should still generate a *different* child the second time. We do, however, want to ensure that the same letter doesn't appear in our decoding alphabet twice, so we can't just take some random number of letters from one parent and the rest from the other.

So, what we will do is **choose a certain number of locations and copy the letter in each of those locations directly from the first parent to the same location in the child**. Then, we will loop over the *other* parent's cipher alphabet, in order, and each time we encounter a letter that isn't already in the child, we will place that letter in the next available open space. We'll end up with each letter appearing once in the child.

If this is confusing, there is another video on the course website where I work through an example!

We will need to choose the number of `CROSSOVER_LOCATIONS`.

5 – Mutation

After each child is generated, there should be a certain probability of a random mutation. In our case, a random mutation would simply be **two randomly chosen letters swapping locations**.

Of course, we will need to choose the `MUTATION_RATE`, the probability that any particular child mutates before being added to the population.

REMINDER: After generating a child, **make sure it is unique in the new population** before adding it. No duplicates!

Final List of Parameters

That leaves us with the following list of parameters that we must set. **Make these GLOBAL CONSTANTS that you set at the top of your code so that they can be modified easily. Do NOT hardcode any of these values inside any method.**

`POPULATION_SIZE` – the number of strategies in each generation
`NUM_CLONES` – the number of precisely cloned strategies retained from each generation to the next
`TOURNAMENT_SIZE` – how many strategies selected for each tournament
`TOURNAMENT_WIN_PROBABILITY` – the probability with which the best strategy in a tournament is selected
`CROSSOVER_LOCATIONS` – how many exact letters from parent 1 are copied to the child in the same locations
`MUTATION_RATE` – the chance that a child experiences a mutation after being generated

My Values

Through a fair amount of entirely unscientific experimentation, these are the values that seemed to produce the best results for me, though I haven't rigorously tested and am sure this is not ideal. Almost everything I've found online suggests a much lower mutation rate, for instance; .8 is incredibly high. Most often, I see numbers like .05 or .1. But it seemed to help my code… In any case, feel free to modify and see what happens! (It's particularly fun to watch what happens when you have zero clones.)

```
POPULATION_SIZE = 500
NUM_CLONES = 1
TOURNAMENT_SIZE = 20
TOURNAMENT_WIN_PROBABILITY = .75
CROSSOVER_LOCATIONS = 5
MUTATION_RATE = .8
```

Helpful Python Methods

The following methods from `random` are essential:

`if random.random() < prob:`	This will trigger a random event with probability `prob`.
`things = random.sample(iter_obj, n)`	This will randomly select *n* **DISTINCT** items from any iterable object.

It might also be a good idea to re-familiarize yourself with how to use functions as keys in a `sort` call.

Required Task

After you code your genetic algorithm, you have two objectives:

- Use it to decode 9 of the messages given at the end of the document. Feel free to tweak parameters as you go; some may work more easily with 3-grams than 4-grams, etc. Keep all of your decoded messages in a document.
- Set the values in your algorithm to whatever the overall most effective strategy you've found seems to be, and then submit your code to take **one command-line argument**, a string containing an encoded message, and run a genetic algorithm that is **cut off after 500 generations or less** to attempt to decode the message. This doesn't need to work every time, but should have a high chance of succeeding on any Medium-difficulty message.
- 500 generations should take about a minute, give or take. If you find your code taking much, much longer than this, the likely culprit is that **you call your fitness function too often**. As noted in the subsection "3 – Selection Method", you should only score each strategy **once per generation**, and then use the **stored** values to sort/rank.

Specification

Submit your **document and code** to the link on the website.

This assignment is **complete** if:

- The "Name" field on the Dropbox submission form contains your **class period**, then your **last name**, then your **first name**, in that order.
- You submit a **document** containing 9 decoded messages.
- Your code accepts **a single command line argument** – a long string of text. (I'll use quotes so that spaces are part of the input.)
- Your code runs a **500 generation or less** genetic process, printing the best result **after each generation**. I do not expect this to work every time, but it should have **a high chance of succeeding**. The highest difficulty will be "Medium"; if it took you like 30 tries to decode each Medium message, you need to tweak your parameters!

For **resubmission**:

- Complete the specification correctly.

Specification for Outstanding Work

The basic idea: find a way to make your decodes work 100% of the time. Be creative; feel free to tweak the genetic algorithm in any way you can think of. No rules, except that you can't use any external libraries or new sources of information about English – whatever you use, you have to get it from ngrams.txt and/or code it yourself. (So, for example, no using NLTK.) Otherwise, the same specification as above.

Submit your **document and code** to the link on the website.

This assignment is **complete** if:

- You meet all the above requirements, **and:**
- In your document, you have decoded **all 11 of the Easy, Medium, and Hard messages**.
- Your code **solves the given input correctly every single time in 500 generations or less**. (I will run your code on 10 different messages; all must be correct. The highest difficulty will be "Medium".)

For **resubmission**:

- Complete the specification correctly.

Decoding problems

In the document you submit, you must include any 9 of these decoded correctly. They're listed in what seems to me, based on the way my code has performed, to be an approximate order of difficulty, but who knows really.

Approximately Easy:

1. PF HACYHTTRQ VF N PBYYRPGVBA BS SERR YRNEAVAT NPGVIVGVRF GUNG GRNPU PBZCHGRE FPVRAPR GUEBHTU RATNTVAT TNZRF NAQ CHMMYRF GUNG HFR PNEQF, FGEVAT, PENLBAF NAQ YBGF BS EHAAVAT NEBHAQ. JR BEVTVANYYL QRIRYBCRQ GUVF FB GUNG LBHAT FGHQRAGF PBHYQ QVIR URNQ-SVEFG VAGB PBZCHGRE FPVRAPR, RKCREVRAPVAT GUR XVAQF BS DHRFGVBAF NAQ PUNYYRATRF GUNG PBZCHGRE FPVRAGVFGF RKCREVRAPR, OHG JVGUBHG UNIVAT GB YRNEA CEBTENZZVAT SVEFG. GUR PBYYRPGVBA JNF BEVTVANYYL VAGRAQRQ NF N ERFBHEPR SBE BHGERNPU NAQ RKGRAFVBA, OHG JVGU GUR NQBCGVBA BS PBZCHGVAT NAQ PBZCHGNGVBANY GUVAXVAT VAGB ZNAL PYNFFEBBZF NEBHAQ GUR JBEYQ, VG VF ABJ JVQRYL HFRQ SBE GRNPUVAT. GUR ZNGREVNY UNF ORRA HFRQ VA ZNAL PBAGRKGF BHGFVQR GUR PYNFFEBBZ NF JRYY, VAPYHQVAT FPVRAPR FUBJF, GNYXF SBE FRAVBE PVGVMRAF, NAQ FCRPVNY RIRAGF. GUNAXF GB TRAREBHF FCBAFBEFUVCF JR UNIR ORRA NOYR GB PERNGR NFFBPVNGRQ ERFBHEPRF FHPU NF GUR IVQRBF, JUVPU NER VAGRAQRQ GB URYC GRNPUREF FRR UBJ GUR NPGVIVGVRF JBEX (CYRNFR QBA'G FUBJ GURZ GB LBHE PYNFFRF – YRG GURZ RKCREVRAPR GUR NPGVIVGVRF GURZFRYIRF!). NYY BS GUR NPGVIVGVRF GUNG JR CEBIVQR NER BCRA FBHEPR – GURL NER ERYRNFRQ HAQRE N PERNGVIR PBZZBAF NGGEVOHGVBA-FUNERNYVXR YVPRAPR, FB LBH PNA PBCL, FUNER NAQ ZBQVSL GUR ZNGREVNY. SBE NA RKCYNANGVBA BA GUR PBAARPGVBAF ORGJRRA PF HACYHTTRQ NAQ PBZCHGNGVBANY GUVAXVAT FXVYYF, FRR BHE PBZCHGNGVBANY GUVAXVAT NAQ PF HACYHTTRQ CNTR. GB IVRJ GUR GRNZ BS PBAGEVOHGBEF JUB JBEX BA GUVF CEBWRPG, FRR BHE CRBCYR CNTR. SBE QRGNVYF BA UBJ GB PBAGNPG HF, FRR BHE PBAGNPG HF CNTR. SBE ZBER VASBEZNGVBA NOBHG GUR CEVAPVCYRF ORUVAQ PF HACYHTTRQ, FRR BHE CEVAPVCYRF CNTR.
2. LTQCXT LRJJ HJRDECD, EZT CDJP SXTFRYTDE EC ZNKT LTTD RASTNHZTY VNF NDYXTV WCZDFCD. ZT VNF NHUBREETY LP N FRDGJT KCET VZTD N LXNKT FTDNECX QXCA ONDFNF XTQBFTY EC PRTJY QXCA SXTFFBXT EC HCDKRHE EZT SXTFRYTDE. ZNY WCZDFCD LTTD HCDKRHETY, EZT FSTNOTX CQ EZT ZCBFT VCBJY ZNKT LTHCAT SXTFRYTDE FRDHT WCZDFCD ZNY DC KRHTSXTFRYTDE. RDHXTYRLJP, RE VNF EZRF FNAT FSTNOTX VZC JTY EZT RASTNHZATDE RD EZT ZCBFT CQ XTSXTFTDENERKTF. EZBF, ZNY EZT FTDNET HCDKRHETY EZT SXTFRYTDE, EZRF VCBJY ZNKT NACBDETY EC N SCJRERHNJ HCBS.
3. ZRTGO Y JPEYPGZA, RP'J IKPGO HIJJRMWG PI RSHEITG PUG JPEYPGZA MA SYDROZ EYOBIS XUYOZGJ, PGJPROZ PUG EGJLWP IK PUIJG XUYOZGJ, YOB DGGHROZ IOWA PUG MGPPGE ILPXISGJ. PURJ RJ XYWWGB URWW XWRSMROZ. PURJ RJ EGWYPRTGWA JRSHWG PI XIBG, MLP BIGJO'P CIED RO GTGEA JRPLYPRIO - RP XYO IKPGO ZGP XYLZUP RO Y WIXYW SYFRSLS, Y JPEYPGZA PUYP RJ OIP RBGYW MLP KEIS CURXU YOA JROZWG XUYOZG RJ OIP YO RSHEITGSGOP IO RPJ ICO. ZGOGPRX YWZIERPUSJ YEG Y HICGEKLW PIIW KIE RSHEITROZ IO PUG RBGY IK URWW XWRSMROZ PI IHPRSRVG Y JIWLPRIO RO JRPLYPRIOJ CUGEG YWW IK PUG KIWWICROZ YEG PELG: Y JPEYPGZA XYO MG HEGXRJGWA QLYOPRKRGB MA Y JHGXRKRX JGP IK TYERYMWGJ ZRTGO XGEPYRO OLSGERX TYWLGJ. PUG ILPXISG IK PUG JPEYPGZA XYO YWJI MG HEGXRJGWA QLYOPRKRGB. PUGEG YEG ROPGEYXPRIOJ MGPCGGO PUG TYERYMWGJ PUYP SYDG JRSHWG URWW XWRSMROZ ROGKKRXRGOP IE LOWRDGWA PI JLXXGGB.
4. CWQ KHTTQKC TFAZJAB HS FGG HS CWQ ECFT YFTE PHRJQE TQGQFEQM EH SFT JE CWQ QPXJTQ ECTJZQE VFKZ, F AQY WHXQ, CWQ GFEC OQMJ, TQCLTA HS CWQ OQMJ, THBLQ HAQ, EHGH, TQRQABQ HS CWQ EJCW, CWQ SHTKQ FYFZQAE, TJEQ HS CWQ EZNYFGZQT, CWQ XWFACHP PQAFKQ, FCCFKZ HS CWQ KGHAQE. CWQ KHTTQKC TFAZJAB HS CWQ CWTQQ JAMJFAF OHAQE PHRJQE JE CWQ GFEC KTLEFMQ, TFJMQTE HS CWQ GHEC FTZ, CQPXGQ HS MHHP. CWQTQ JE AH SHLTCW JAMJFAF OHAQE PHRJQ, FAM FANHAQ YWH CQGGE NHL HCWQTYJEQ JE F GJFT. OLEC CQGG CWQP CH CLTA FTHLAM FAM YFGZ FYFN VQSHTQ CWQN KFA VGQEE NHL YJCW FAN HCWQT JAKHTTQKC HXJAJHAE. FANYFN, EH EFNQCW PN STJQAM VJGG, YWH WFXXQAQM CH VQ HAGJAQ YWJGQ J YFE PFZJAB CWJE FEEJBAPQAC, YWQA J FEZQM WJP 'YWFC YHLGM VQ F BHHM EQKTQC PQEEFBQ SHT PN ECLMQACE CH MQKHMQ?' XGQFEQ CFZQ LX FAN KHPXGFJACE YJCW WJP.

Approximately Medium:

5. XMTP CGPQR BWEKNJB GQ OTGRB EL BEQX BWEKNJB, G RFGLI. GR GQ BEQX ABSETQB RFGQ QBLRBLSB TQBQ EJJ RBL KMQR SMKKML VMPYQ GL BLDJGQF: 'G FEUB RM AB E DMMY QRTYBLR GL RFER SJEQQ GL RFB PMMK MC RFER RBESFBP.' (If, after decoding this problem, you're confused as to why it's in Medium difficulty – I originally made this for my Cryptography class, where this was an easy challenge for the *traditional* method of decoding, but is very short and so would appear to be a bit harder for these programs to get right.)
6. XTV B CHDQCL BHF GCVIVDGDHWPN ABVF ZABPPLHWL, ZTHGDFLV MBJDHW B PTHW BHF XCPPN VLBFBYPL GLVDLG TX UTVFG HLRLV CGDHW B GDHWPL LEBMIPL TX TCV ULPP-PTRLF LHWPDGA WPNIA UADZA TZZCVG GLZTHF IPBZL DH TRLVBPP XVLQCLHZN. DX D BM WLHCDHL, D UDPP GBN MBHN, MBHN GLZTHFG ABRL IBGGLF UADPL D ABRL YLLH ALVL ITHFLVDHW MBJDHW GCZA B UTVJ. FDGZTRLVDHW NTC ZVBZJLF MN YVBDHZADPF, ALVL, DH B GMBPPLV HCMYLV TX GLZTHFG UTCPF WDRL ML HT GCVIVDGL.
7. NU XTZEIMYTNEVZ INUHU YM, ZML SPYVI NXILNFFZ XNFF IVPU N API VNTD. NU PI ILTWU MLI, P XNW YM N FMWY JNZ JPIVMLI LUPWY NWZ MC IVNI YFZEV IVNI ITNDPIPMWNFFZ CMFFMJU 'D' NI NFF. PUW'I IVNI ULTETPUPWY? P CMLWD IVPU ULTETPUPWY, NWZJNZ! NW NLIVMT JVM NFUM CMLWD IVPU ULTETPUPWY, FMWY NYM, NXILNFFZ FMUI SNWZ SMWIVU JTPIPWY N AMMH - N CLFF CPXIPMWNF UIMTZ - JPIVMLI IVNI YFZEV NI NFF. NSNRPWY, TPYVI?
8. RHNJJCBXVCXJYQJNEJNDYDCELTHNBFTVTHNJJREFCLBEECANOTREFDNEBXTHJTNXTXECPCBAPZNSSPXTNYTXFVZCNXTSXRKRJTGTYECJ RKTRDFSNHTRANGRDTNKNFEFZTTECQSNSTXCDVZRHZFEXNRGZEJRDTFEXRNDGJTFFUBNXTFSTDENGCDFZTINGCDFNDYCEZTXQRGBXTFRD FETNYCQXTANRDRDGQRITYRDEZTRXSJNHTFACKTQXTTJPNLCBECDCXRDEZTFBXQNHTLBEVREZCBEEZTSCVTXCQXRFRDGNLCKTCXFRDORD GLTJCVREKTXPABHZJROTFZNYCVFCDJPZNXYVREZJBARDCBFTYGTFNDYPCBVRJJEZTDZNKTNSXTEEPHCXXTHEDCERCDCQAPHCBDEXPNDY HCBDEXPATDNJNFNQTVPTNXFNGCRFZCBJYZNKTFNRYAPBDRKTXFTLBEDCVAPARDYZNFLTTDCSTDTYECZRGZTXKRTVFCQEZRDGF

Approximately Hard:

A note with these; if you find that the decoded message is correct except for one or two obvious letter swaps that occur rarely (like “KUST” instead of “JUST” but everything else is fine, for instance) it’s ok to make edits by hand after the decode is finished.

9. W CTZV VYQXDVD MCWJ IVJJTHV, TYD VYQXDVD WM BVAA, FXK WM QXYMTWYJ MCV JVQKVM XF MCV PYWZVKJV! YX KVTAAS, WM DXVJ! SXP DXY'M NVAWVZV IV? BCS BXPAD SXP YXM NVAWVZV MCTM MCWJ RVKFVQMAS QKXIPAVYM JVQKVM MVGM QXYMTWYJ MCV NV TAA, VYD TAA, HKTYDVJM JVQKVM XF TAA MCV QXJIXJ? YXB W FVVA DWJKVJRVQMVD! CTZV SXP DWJQXZVKVD SXPK XBY NVMMVK PAMWITMV MKPMC XF VZVKSMCWYH? W DWDY'M MCWYL JX. JX BCS TKV SXP HVMMWYH TAA PRRWMS TM IV? CXYVJMAS. YX XYV CTJ TYS ITYYVKJ MCVJV DTSJ. ...BCTM'J MCTM? SXP BTYM IV MX MVAA SXP MCV JVQKVM? YXM TFMVK MCWJ LWYD XF DWJKVJRVQM! HXXDYVJJ HKTQWXPJ IV. NTQL BCVY W BTJ T SXPMC W BTJ YXM JX QTAAXPJ. BCVY JXIVXYV BVAA KVJRVQMVD TYD WIRXKMTYM MXAD IV MCTM MCVS CTD JXIVMCWYH BXKMC MVAAWYH IV, W OPJM AWJMVYVD! W DWDY'M DXPNM MCVI! JX KPDV, CXYVJMAS. OPJM PYTQQVRMTNAV.
10. ZFNNANWJWYBZLKEHBZTNSKDDGJWYLWSBFNSSJWYFNKBGLKOCNKSJEBDWZFNGKLJKJNQFJPFJBXHBZTNRDKNZFNPDEJWYDRPDEGCNZNWJ YFZZFLZTCNBBNBZFNNLKZFSLKONWBLCCKJANKBPHGBZFNGNLOBLWSRDCSBZFNRJWLCBFDKNJWLWSWDTDSUWDTDSUOWDQBQFLZBYDJWYZ DFLGGNWZDLWUTDSUTNBJSNBZFNRDKCDKWKLYBDRYKDQJWYDCSJZFJWODRSNLWEDKJLKZUJNANWZFJWODRDCSSNLWEDKJLKZUZFNRLZFN KQNWNANKRDHWSJZFJWODRSNLWEDKJLKZU
11. FBYSNRBIYVNIJRJZSRSRJZNQCQNIJXCGTNEBSJNYKCUXCGTNONNIRNBUMZSIVSIJZNYBUAXCGURENBJRCBASIVJZUCGVZJZNKFCCUBIY OGUSNYSIXCGUOCINRJZNUNRBIBMZNJZBJXCGMBIJSVICUNJBASIVXCGUOUNBJZRJNBFSIVXCGUQSIYBIYBFFJZBJEBRUNBFSRFNKJONZ SIYYCIJKSVZJSJSJRMCQSIVKCUXCGUGIISIVBJXCGSJRCIFXJZSRQCQNIJYCIJMBUNEZBJMCQNRBKJNUXCGUKNTNUYUNBQMBIJXCGRNN SJVNJJSIVMFCRNUPGRJRGUUNIYNUMBGRNXCGKNNFJZNKNNFSIVJBASIVCTNUSJRKSUNSJRKUNNYCQSJRKFCCYSIVCLNISJRJZNLUNBMZ NUSIJZNLGFLSJBIYXCGUOFSIYYNTCJSCIJZNUNRRCQNJZSIVOUNBASIVBJJZNOUSMACKNTNUXEBFFSJRZCFYSIVBFFJZBJXCGAICERCJ NFFQNYCXCGEBIIBVCEZNUNSJRMCTNUNYSIBFFJZNMCFCUNYFSVZJREZNUNJZNUGIBEBXRBUNUGIISIVJZNISVZJSQLCRRSOFNMCQNRJU GNSJRJBASIVCTNUXCGCZJZSRSRJZNVUNBJNRJRZCEENFSVZJSJGLENECIJMCQNYCEIBIYJZNRGIMBIJRJCLGRICEEBJMZSIVSJMCQNJU GNSJRJBASIVCTNUXCGCZJZSRSRJZNVUNBJNRJRZCE

Just for fun, problems that my code actually can’t solve correctly.

Good luck?

12. KDBULISCWSCTLISJLXTNJULIUJBTALISNYULDEIDJCKTCGLISKLAUKLIMBFUGTCGLIMCGUJUGLIJUTLUCSCWLIULIJUUGLIDMWILKDEB TLLIUQLIULIMWTALIDMWILIUTLJSNTAAXSLQTKDCAXLIULISJLUUCLIDMKTCGLISKLAUKTCGLIDJCKLIJDMWILIUMCGUJCUTLIDEISKL ISWILITLLIULISJLXXUTJDAGLIMWLIDMWILDELITLBDJCSCW
13. ANUYJKHNFL JLNBL, NBENJK YNK KHNKIONS: 'JNYNHYJ - SNJKONS, INHKONS, JNHDSNJ ONBRNJ!'
14. MBLBJJBV, HUBBSKDBM SBIMBJK BO BUS JBIBIB KIDBBUK BO LBJBIIB, BXOBJS, IBUBLHB KBV CBCIBJA-IBUBJ LBJLBA HUBKKBL CBSK.
15. STHTGHTCTITGHTUSKSTHHTOBTGUKTETIWSTBITSQTWKTESTKWSSTYTGUTGETHHTHHTCTETETWHSTESTUSBTUSKHTGSTHWBTGTWTWTGUT GHTIMOTYTGTHITUTKHTGHTIMTBBTWTKTFTHJFTUTIYTRTCTWGTW