Project 1: Cryptographic Voting

This project has two due dates: **Friday, February 19** and **Friday, February 26**, both at **6 p.m.**. Late submissions are not accepted. If you have a conflict due to travel, interviews, etc., please plan accordingly and turn in your homework early.

This is a group project; you will work in **teams of two** and submit one project per team. Please find a partner as soon as possible. If have trouble forming a team, post to the relevant Piazza thread. The final exam will cover project material, so you and your partner should collaborate on each part.

The code and other answers your group submits must be entirely the work of your group, and you are bound by the Honor Code. You may consult with other students about the conceptualization of the project and the meaning of the questions, but you may not look at any part of someone else's solution or collaborate with anyone outside your group. You may consult published references, provided that you appropriately cite them (e.g., with program comments), as you would in an academic paper. *Even though this project is based on code subject to an open source license, you may not redistribute your work, because that would constitute a violation of the Honor Code.*

Once you have your partner, then you will use GitHub Classroom, via this clone link. One of you goes first, and GitHub will ask for a name for your "team". Please use your two Rice NetIDs with a hyphen between them, e.g., `abc12-def34`, as your team name. **Do not visit the clone link until both you and your partner have agreed to work together.** GitHub makes it difficult to rearrange partnered repositories, so we'd like to avoid this unless absolutely necessary.

Before you do anything else, be sure to edit the `README.md` file to have your names, email addresses, and so forth, then commit the file. Just to make sure you're not having any weird GitHub issues, do this with separate commits: one from each partner editing their name. That way, if there are weird GitHub problems, we'll know about them as soon as possible.

https://classroom.github.com/a/XXXXXXXX

Contents

1 Introduction **2**
1.1 Objectives 3

2 Setting up your computer **3**

3 Cloning from GitHub, setting up your environment **4**

4 Cryptographic basics **4**

5 ElGamal encryption and related proofs **5**
5.1 Homomorphic accumulation 6
5.2 Key sharing 6
5.3 Chaum-Pedersen proofs 7
5.4 Nonces and nondeterminism 9

6 The ElectionGuard library **9**
6.1 Testing 10
6.2 Property-based testing 10

7 Submission Strategy & Engineering **11**

8 Grading **12**

9 Optional Part 3: Performance **13**

1 Introduction

This project gives you a sophisticated toolbox of primitives that are useful for building "end to end verifiable" cryptographic voting systems, including ElGamal cryptography (additively homomorphic) and Chaum-Pedersen proofs. It's based on an open-source project from Microsoft called ElectionGuard, which has multiple implementations for different purposes. This assignment is based on the Python "reference implementation"[1] but has substantial changes.

The "real" ElectionGuard code has a number of features that are useful for "real" elections, like understanding different "ballot styles" (i.e., not every candidate appears on every ballot, but you still want to be able to add ballots together). These features add a huge amount of complexity that might get in your way of understanding and using the code without a lot of study.

Consequently, we've stripped all the fancy stuff out and kept all the cryptographic primitives. You'll be implementing "simple" elections, where every ballot has exactly one contest on it, with the same candidates on every ballot, and each voter gets to pick k of the n total candidates. These are simply added up, and whoever gets the most wins. (This is a modest generalization of the usual "vote for one" scheme, sometimes called "first past the post"[2].)

This project has two parts. You'll first encrypt, decrypt, and validate "selections", which is to say, the individual choices that a voter might make. For the second part, you'll move up one level of abstraction and implement encryption, decryption, and validation for whole ballots, as well as for the election tabulations.

[1] https://github.com/microsoft/electionguard-python
[2] https://en.wikipedia.org/wiki/First-past-the-post_voting

For both parts, we've written extensive unit tests, leveraging the Python Hypothesis property-based testing library[3] , which will exercise your code with both positive examples (i.e., where validation should succeed) and negative examples (i.e., where validation should fail). (See Section 6.2 for more details on property-based testing.) We're also going to ask you to write some of your own unit tests that validate specific properties. Your grade will be primarily based on which unit tests you pass and which unit tests you fail. Of course, you're welcome to add any additional tests you want, if you find that helpful, but *if you make any modifications to existing tests, we will grade them as if they failed*.

1.1 Objectives

- Gain exposure to modern, real-world cryptographic techniques.
- Learn to implement cryptographic code and validate its correctness.

2 Setting up your computer

This assignment generally requires you to set up your computer as one would do for a modern Python3 development environment. There are a number of good Python IDEs, like PyCharm, that you're welcome to use, although you can run all the tests from the command-line, and that's how we'll describe everything here.

Once your computer is ready, you should install:

- Python 3.8 (https://www.python.org/downloads/)
- (Mac only) Install the Apple command-line developer tools. Run `xcode-select --install` and click to accept the license. Alternately, you can install the full XCode from the app store.
- (Mac only) Install Homebrew (https://brew.sh/)
- (Windows only) Install Git (https://gitforwindows.org/) – this includes the "Git Bash Shell", where you can type the commands we'll show you. You'll also need to install `make`[4], which you put in the `mingw64/bin` directory.
- (Windows only) A fancier way to install Git is through Chocolatey, which also lets you easily install other commands you might want like `make`, which you'll need for this project. (Simple instructions: https://jcutrer.com/windows/install-chocolatey-choco-windows10, more complex instructions: https://chocolatey.org/install)

Earlier versions of Python won't work. This assignment hasn't been tested against Python 3.9, so it might work or it might not.

[3] https://hypothesis.readthedocs.io/en/latest/

[4] http://gnuwin32.sourceforge.net/packages/make.htm

3 Cloning from GitHub, setting up your environment

If you're using a tool like PyCharm, it will *try* to do all of this automatically. You give it the GitHub URL, and it will clone it locally. PyCharm might offer to set up a Python "virtual environment" for you, but you should instead let `make` do it for you and then tell PyCharm to use the virtual environment you just made.

Launch your favorite shell and go to the directory after the `git clone` process is complete. Run `make`. This will run the appropriate commands for Linux, Mac, or Windows, to download and install all of the necessary dependencies. **You need to do this, exactly once, even if you've already used PyCharm or another IDE to clone the repository.**

One way that you'll know you've succeeded is if that `make` process gets all the way through to running the unit tests. Most of the unit tests should succeed, with others that will fail because you haven't implemented the assignment yet.

Similarly, you can compare what you see on your own computer with what happens every time you push some commits to GitHub, where all the unit tests will be executed in a virtual environment by GitHub Actions. You'll see a red X or green checkmark, telling you whether the tests have failed or passed. You can click on that and you'll then see a transcript of the unit tests running.

4 Cryptographic basics

There are several cryptographic things going on in ElectionGuard that we didn't have time to cover in class, but that you really need to understand.

When we introduced the digital signature algorithm, we mentioned that it has two separate primes that it uses, p and q, but we didn't spend much time explaining why, except to say that we want our exponents to be smaller, if we can get away with it without giving up any cryptographic strength. (The text below was written by Josh Benaloh, a principal senior cryptographer at Microsoft Research, who also designed the cryptography for ElectionGuard.)

The arithmetic is performed using a large 4096-bit prime p which provides a substantial security margin beyond what can be attacked with today's computers. We need to work in a multiplicative subgroup of $\mathbb{Z}_p = \{0,1,2,\ldots p-1\}$ – the integers modulo p. The largest multiplicative group within $\mathbb{Z}_p$ simply excludes zero and is denoted as $\mathbb{Z}_p^* = \{1,2,\ldots p-1\}$. However, to avoid having a large number of elements with special properties which distinguish them from others, we want our subgroup to be of *prime order*, and $\mathbb{Z}_p^*$ has $p-1$ elements – which will not be a prime number (except in the trivial case that $p=3$). Group theory tells us that the order of a subgroup must divide the order of the group, so the order of our subgroup must be a prime q that is a divisor of $p-1$.

We can improve efficiency without significantly impacting security by choosing a 256-bit prime q. A q just below 2^{256} matches well with outputs from SHA-256, so we use the largest 256-bit prime $q = 2^{256} - 189$. The elements of our subgroup will be $\left\{g^0, g^1, \ldots g^{q-1}\right\}$ (all computed modulo p) where g is a so-called *generator*.

To make this possible, q needs to be a factor of $p-1$. We can then find a suitable generator by computing any $g = x^{(p-1)/q} \mod p$ that does not produce $g = 1$. We just choose $g = 2^{(p-1)/q} \mod p$. An important property of our subgroup is that for all elements y in the subgroup, y^q

mod $p = 1$. Be very careful to distinguish the two primes; while all computations in the base are modulo p, any computations done in the exponent should be modulo q.

One might then choose p to be the largest 4096-bit prime such that $p-1$ is a multiple of q (which is what I did initially). However, having p be close to a power of 2 (or otherwise representable as a value of a polynomial with "small" coefficients) weakens its resistance to discrete log attacks via the special number-field sieve[5]. It is desirable to have parameters generated deterministically to avoid concerns that there may be some hidden back door. I tried to find a clean way to do this, but every attempt that I made wound up being representable as a polynomial with small coefficients. I therefore did what everyone else does and used an unrelated mathematical constant to deterministically produce p. Both π and e are overused for this purpose, and the description is slightly cleaner if the constant is less than one and greater than one half, so I wound up with the Euler-Mascheroni constant[6]. For computational efficiency, there is some benefit to having all of the extreme high-order bits and low-order bits be one, so this drove the selection of a suitable p.

5 ElGamal encryption and related proofs

ElGamal encryption, invented in 1985 by Tahir ElGamal[7], is very straightforward. If you have a secret key a in $[0,q)$, the corresponding public key is $g^a \mod p$, in $[0,p)$. (We'll stop writing the p, q, and mod parts because they just overcomplicate the equations.) It's safe to share g^a with anybody because it's (believed to be) very hard to recover a. This is called the "discrete log problem" (see also, the not entirely identical "decisional Diffie-Hellman problem"[8], which is the actual hard problem we're counting on for ElGamal encryption).

Because we're interested in having an *additive homomorphism*, where we can "add" together two encryptions and get another encryption, we're going to use a variant called "exponential ElGamal". With this, we put the message m we're encrypting into an exponent, *encoding* our message m as g^m. For voting, these messages are mostly going to be 0-or-1, so the values we're encrypting are, for the most part, just g^0 and g^1. Later on, when we multiply these together, their exponents will add.

One of the essential features of ElGamal is that it's a *nondeterministic* cryptosystem, which means that even when we're encrypting pretty much the same values over and over again, we'll get a different ciphertext every time. This is accomplished by introducing a random number r, called a *nonce*, into the encryption as follows:

[5] https://en.wikipedia.org/wiki/Special_number_field_sieve
[6] https://en.wikipedia.org/wiki/Euler-Mascheroni_constant
[7] https://ieeexplore.ieee.org/document/1057074
[8] https://en.wikipedia.org/wiki/Decisional_Diffie%E2%80%93Hellman_assumption

$$
\begin{aligned}
c &= \text{Encrypt}(m, r, g^a) = \langle g^r, g^{ar} g^m \rangle \\
p &= \text{Decrypt}(c, a) \\
&= \text{Decrypt}(\langle g^r, g^{ar} g^m \rangle, a) \qquad\qquad \text{note} : (g^r)^a = (g^a)^r = g^{ar} \\
&= \text{DLog}_g \left(g^{ar} g^m * (g^{ar})^{-1} \right) \\
&= \text{DLog}_g g^m \\
&= m
\end{aligned}
$$

Note that $\text{DLog}_g g^m$ is only efficiently computable for relatively small m, since our implementation uses *memoization* to build a table $\left[1, g, g^2, g^3, ...\right]$.[9] Multiplicative inverses, however, are always efficient to find, for any value in the group. In particular, we know that $\forall x : x^q \mod p = 1$, by virtue of how we've set up p and q, so therefore an easy way to find x^{-1} is just x^{q-1} because $x * x^{q-1} = x^q = 1$. Alternatively, a efficient way to compute inverses would be to use the extended Euclidean algorithm[10].

5.1 Homomorphic accumulation

With this definition of ElGamal encryption, we can define an accumulation operation using piecewise multiplication:

$$
\begin{aligned}
c_1 &= \text{Encrypt}(m_1, r_1, g^a) = \langle g^{r_1}, g^{ar_1} g^{m_1} \rangle \\
c_2 &= \text{Encrypt}(m_2, r_2, g^a) = \langle g^{r_2}, g^{ar_2} g^{m_2} \rangle \\
c_1 \oplus c_2 &= \langle g^{r_1}, g^{ar_1} g^{m_1} \rangle \oplus \langle g^{r_2}, g^{ar_2} g^{m_2} \rangle \\
&= \langle (g^{r_1})(g^{r_2}), (g^{ar_1} g^{m_1})(g^{ar_2} g^{m_2}) \rangle \\
&= \left\langle g^{r_1 + r_2}, g^{a(r_1 + r_2)} g^{m_1 + m_2} \right\rangle \\
&= \text{Encrypt}(m_1 + m_2, r_1 + r_2, g^a)
\end{aligned}
$$

Any intermediary can compute the encrypted sum ($\oplus$) of these encryptions without needing to decrypt, and without needing to know the secret key. This is called an *additive homomorphism*.

5.2 Key sharing

The real ElectionGuard library supports a sophisticated "threshold cryptography" system which allows multiple trustees to each have a "share" of the election's secret key material. These shares are combined into a single public key, that all the voting machines use. For decryption operations,

[9] We could trade space for time, using an algorithm called "baby-step, giant-step" by Daniel Shanks. https://en.wikipedia.org/wiki/Baby-step_giant-step; see also rainbow tables. https://en.wikipedia.org/wiki/Rainbow_table

[10] https://en.wikipedia.org/wiki/Extended_Euclidean_algorithm#Computing_multiplicative_inverses_in_modular_structures

it takes a (predefined) k of the n trustees to collaborate together in order to do any decryption operation. This "key ceremony" is a bit complicated.

As a warmup, you'll be implementing a simplified version, where all of the key shareholders need to collaborate. Here's how it works. Let's say we have n trustees, and each has secret keys a_i and public keys g^{a_i}. We can define a joint public key by simply multiplying them all together:

$$\begin{aligned} k_{\text{joint}} &= g^{a_1} g^{a_2} \cdots g^{a_n} \\ &= g^{a_1 + a_2 + \cdots a_n} \end{aligned}$$

Multiplying together the public keys yields a joint key, where we know the *formula* for the exponent, even though we can't derive the *value* for it. So, if a message is encrypted with this joint key, we need a way for the trustees to collaboratively decrypt it, without revealing their secret keys.

$$\begin{aligned} \text{PartialDecrypt}(c, a_i) &= \text{PartialDecrypt}\left(\left\langle g^r, g^{(a_1 + a_2 \cdots a_n) r} g^m \right\rangle, a_i\right) \\ \text{DecryptionShare}_i &= (g^r)^{a_i} = g^{a_i r} \\ \prod_i \text{DecryptionShare}_i &= g^{a_1 r} g^{a_2 r} \cdots g^{a_n r} \\ &= g^{(a_1 + a_2 \cdots a_n) r} \end{aligned}$$

Each decryption share is nothing more than the nonce padding g^r raised to the secret key share a_i, and then we can just multiply those shares together and we get exactly the value that's multiplied with g^m. As before, multiplicative inverses are easy to find, so we have everything we need to combine "partial decryptions" together to yield the complete decryption. (This is a warm-up exercise that you'll do to get things rolling for part 1 of the assignment.)

5.3 Chaum-Pedersen proofs

You'll see several different kinds of Chaum-Pedersen proof objects in `chaum_pedersen.py`:

DisjunctiveChaumPedersenProofKnownNonce demonstrates that the prover knows the secret nonce used to create an encryption that's either a 1 or 0.

ConstantChaumPedersenProofKnownNonce demonstrates that the prover knows the secret nonce used to create an encryption that's a constant (not just 0 or 1).

ConstantChaumPedersenProofKnownSecretKey demonstrates that the prover knows the secret key used to create an encryption that's a constant (not just 0 or 1).

GenericChaumPedersenProof is used by all of the above proofs. It provides a proof that two tuples $(g, g^x), (h, h^x)$ share the same exponent x without revealing it.

So, when do you use which sort of proof? When you are *encrypting a counter that might be one or zero*, you make a disjunctive proof. You'll need to hang on to the nonce that you used to do the encryption. You'll ultimately output one proof per encrypted counter.

When you are *proving that the voter cast at most k votes on the ballot*, you could theoretically sum them up and use a disjunctive proof, but that would only really work with $k = 1$, and we want to be more general, which would make a disjunctive proof much more complicated. Instead, what you'll do is create zero or more "placeholder" candidates (literally, their name will be "PLACEHOLDER" with an integer after that). If all the other candidates got zero votes, then the placeholder(s) gets one. Otherwise, the placeholder(s) gets zero, such that the sum of the candidate slots plus the placeholder slots is exactly k. Each placeholder ciphertext has the same disjunctive proof as all the real candidates. And now here's the cool part: you just use the additive homomorphism of ElGamal, and generate a *constant* Chaum-Pedersen proof, that the sum of all the candidates, including the placeholder(s), is exactly k.

When you are *decrypting the tally and proving that the announced plaintext is correct*, you use the decryption proof, which doesn't require the nonce, but does require the decryption (secret) key. This corresponds to something the election official might compute when the election is complete. You'll see each of these Chaum-Pedersen proof types listed as part of the various data structures in `simple_election_data.py`, so you'll know which one to use where.

Fun fact: in the special case where $k = n$, we wouldn't need this placeholder construction, because we wouldn't need to restrict the total number of cast votes. This voting method is called "approval voting"[11]; not only does it simplify our proofs, but it also improves usability for voters by eliminating the possibility of a ballot being discarded by virtue of being "overvoted". Of course, if a voter selected *all n* candidates, their vote would be valid, but it wouldn't have any meaningful influence on the winner.

So, how do those proofs work? We cover this in more detail in our lecture. For the purposes of this assignment, what matters is that you understand what a "generic" Chaum-Pedersen proof actually proves, which then enables the more specific versions of the proof to use it as a subroutine. So, if we want to prove that the encryption of $m = 3$ is actually three, based on our knowledge of the encryption nonce r, then we're going to be generating a proof that (g, g^r) and (g^a, g^{ar}) share the nonce r without revealing r. And that value g^{ar} is just the right-hand side of the ElGamal encryption, after we multiply it with g^{-3} (i.e., canceling out the g^m where $m = 3$).

This might not seem very exciting, but it's the same math when we're proving that the *sum of the encrypted counters* is exactly the constant k for a k-of-n election. When you combine that with proofs, for each counter, that they are exactly 0 or 1, you end up with a proof, publicly verifiable, that you have a *well-formed ballot*. No overvotes. No funny business. This gives public observers of an election the ability to verify important properties of the election tally without compromising any individual ballot's privacy!

What if the voter selects more than k candidates? The voter has expressed a ballot that is not valid. There's no way for you to produce an encrypted ballots, with all of the valid proofs. Since

[11] https://en.wikipedia.org/wiki/Approval_voting

exactly this can and will happen in the real world, you'll need to deal with this, in part 2 of the assignment. You'll need to "interpret" these overvoted ballots into undervotes (i.e., no selections at all) before encrypting them. Note that this means that the decryption of the encryption of an overvoted ballot is going to be different from the original selections!

5.4 Nonces and nondeterminism

You'll see `seed`, `hash_header`, and other such names for arguments to many of your functions. The important concept is that **all your code should be deterministic**. The same inputs should always yield the same outputs. These seed values are where you get your nondeterminism. When you're encrypting a ballot that has five counters, you'll need five individual random numbers. You can derive them from the seed using the code in `nonces.py`.

The broad idea, then, is that every "real world" ballot will start with a single, true-random number, and from that, all of the other random r values for the encryptions, and everywhere else they're needed, will be derived from that initial seed.

You want to implement this as a hierarchical process. From the master seed, you'll derive seeds for each ballot. Then from the ballot seeds, you'll derive seeds for each selection. Each layer of code passes the seed values down to the next layer of code. You can convert one seed into as many as you need by using the code in `nonces.py`.

Extra fun. Imagine that we have hand-marked paper ballots being fed into a scanner with a reasonably fast computer but not enough storage for the ballot ciphertexts (real world ElectionGuard ciphertexts can be multiple megabytes of data per ballot). If we printed a unique master seed as a barcode on the bottom of each ballot, then the scanner could compute the full encryption, if it wanted, printing the voter a "receipt" which is really just the hash of their encrypted ballot. *Then the machine can completely forget the ciphertext* because it can be recomputed later, identically, due to the deterministic architecture of how nonces are derived from seeds.

6 The ElectionGuard library

If you look in the `src/electionguard` directory, you'll see a number of files. Here's what each of them does:

chaum_pedersen.py Implementations of several different Chaum-Pedersen proofs (see Section 5.3).

dlog.py Implements a caching, brute-force discrete log engine. So long as the exponent is on the small side (millions, not billions), this will run efficiently. We need this to undo the g^v exponentiation applied to the plaintext vote counter v.

elgamal.py Implements "exponential ElGamal" encryption and decryption. Also, this is where you'll implement "key sharing" (see Section 5.2).

group.py Implements `ElementModQ`, `ElementModP`, and a variety of function wrappers to compute with them. All of this bottoms out to calls to the GNU MultiPrecision library, which has fast, hand-tuned C and assembly code.

hash.py Implements a wrapper around SHA-256 hashing, where you can hash just about anything, including lists, and get out an `ElementModQ`.

logs.py Implements a wrapper around Python's logging facility. Logs go to the screen, and you have some flexibility to configure things, where everything goes to a file and only some things go to the screen. You'll see lots of calls to the logging functions throughout the rest of the code.

nonces.py Often you'll want to start with some seed value and generate a stream of nonces that you need to be different from one another, but deterministically derived from the seed. This code gives you a Python Sequence, which you can treat as if it's an array of infinite length, giving you random access to any element. Optional "headers" (typically, just another string) will be hashed together with the given seed value, making it easy for you to create multiple nonce streams, which you might use for different purposes, from the same original seed value (see Section 5.4).

simple_election_data.py This defines all the input and output types for the code you'll be writing. This is the Python equivalent of C `struct` definitions. (We deliberately avoided using too many fancy object-oriented features, to keep our code cleaner, making it hopefully more readable by non-experts in the language.)

simple_elections.py This is where you'll be doing most of your work. We've defined all the functions you need to implement, including their argument and return types, but we've left most of the bodies empty.

utils.py A handful of general-purpose utility functions that are used in various places.

6.1 Testing

You'll find corresponding unit test files in the `tests` directory. If you read those tests, you'll see that they make extensive use of the Hypothesis property-based testing library, which allows inputs to be generated at random. In `src/electionguardtest`, you'll see the "generator" functions (called "composites" in Hypothesis) that can produce these random values. Unfortunately, Hypothesis and mypy (Python's static type annotation checker) don't get along nicely, so we cannot annotate the generators with their return type. If you follow the patterns of other unit tests, you should be able to avoid too many weird type errors.

6.2 Property-based testing

If you've never worked with property-based testing before, you can think of it as a form of fuzz testing, where the computer generates randomly chosen inputs, as part of the unit testing process,

and then assertions are made about the code. A simple example property might be asserting that decryption is the inverse of encryption, i.e., $\forall x : \text{Decrypt}(\text{Encrypt}(x)) = x$. A property-based testing library will automate the process of generating inputs and running the tests. If it finds a *counterexample*, an optional but helpful feature is called *shrinking*, where it will try to search for a "simpler" counterexample to present to you, which will help you better understand your bug. We've disabled shrinking, by default for most Hypothesis tests, because it's very slow.

Scott Wlaschin has produced an excellent introduction to property-based testing[12] – a 50-minute YouTube video plus a variety of written blog posts. His example code is all in F#, but his explanations apply perfectly to our use of Hypothesis in Python.

7 Submission Strategy & Engineering

All of the code elements that you must implement are conveniently labeled `TODO` for you and will mention "part 1" or "part 2". You'll notice some of these `TODO` comments are in the unit tests, themselves. Each `TODO` item explains what you're supposed to do, so we won't reiterate those here.

Generally speaking, part 1 is a warm-up, where you're learning your way around the codebase, and part 2 is where you're dealing with ballots having k-of-n votes cast. Of note, in part 1, you'll start off by implementing ElGamal key sharing (see 5.2), which involves editing code in `elgamal.py`. Everything else you'll do is in `simple_elections.py` and the associated unit test files.

You should adopt a "commit and push, early and often" strategy. While we're not specifically instructing you in how you should collaborate with your partner, some variant of pair programming will be invaluable, where one of you is typing and sharing their screen, while the other is looking over and offering suggestions. Certainly, Git makes it straightforward for you to work on separate items and merge them later, but you're less likely to understand the full project. Trade-off who is typing and who is assisting. Don't just say "I know how to do this. You just watch and maybe clean it up later."

The unit tests will all run on GitHub as part of GitHub Actions. It's generally faster and easier to run them on your own computer. You can do this inside your favorite IDE, or you can run `make test` from your commandline. If you get weird errors, you might run `pipenv shell` beforehand, which ensures that you're using the Python virtual environment for the subsequent commands you want to run. You should also run `make lint`, which will run `black` (the code indenter) and `mypy` (the static type checker). If `black` rejects your code indentation, you can fix that by instead running `make auto-lint` or running `black src tests` on the command-line.

The autograder is also available to you from the commandline. Just run `make autograder`. That should get you results identical to what the graders see on GitHub Actions.

When you finish part 1. You're going to want to use a Git tag. You do this after you're satisfied that you've completed the requirements. You then say `git tag part1` and `git push --tags`. Alternately, you can use the GitHub website to create a "release", which in turn creates a Git tag. Your grader will examine your code, at this tag, and the output of GitHub Actions when running the

[12] https://fsharpforfunandprofit.com/pbt/

unit tests on this tag. Or, if you haven't done this, your grader will just look at whatever you've pushed prior to the deadline.

Continuing with part 2. You can march right ahead with part 2, even if that means breaking something from part 1. Your grader will examine whatever you have committed and pushed prior to the final deadline. If you want to keep playing with the code afterwards, please make a `part2` tag, as above, and then that will be the focus of the graders' attention.

What if it works on my computer and fails on GitHub Actions? Your grade is based on what the grader sees running on GitHub Actions. Perhaps you changed a file on your local machine and didn't commit or push the changes to the server. When in doubt, find one of the TAs or the professor and ask for help if you're seeing these sorts of discrepancies.

Am I allowed to add imports to the files? Absolutely. If there's a function you need that isn't already imported, then feel free to import it. That includes functions from the ElectionGuard library, or otherwise available from the standard Python libraries. You shouldn't need to add or change any external dependencies in Pipfile.

Am I allowed to add helper functions and/or additional unit tests? Sure. Just don't change the type signatures or any of the other existing code that we've provided for you. In particular, if you make changes to any of the unit tests used by the autograder, your human grader will treat that test as having failed and will subtract the appropriate points.

I'm seeing a ton of warnings on the console... Many of the functions that you're given, when they discover an error, will log about it, and then indicate a failure as part of their return value. Some unit tests will exercise these errors, causing all the logging output as a side-effect, but what really matters to you is whether the unit tests pass or fail.

8 Grading

This project is worth a total of twenty points.

Part 1 (worth 10 points). You must implement several functions and unit tests. Your grader will look to make sure all tests from `test_simple_elections_part1.py` pass. You get one point for your code passing `mypy` with no warnings or errors.

Part 2 (worth 10 points). The remaining unit tests, in `test_simple_elections_part2.py` , are collectively worth 9/10 points. The remaining point is assessed by your human grader, looking at your `README.md` file, where you're asked to answer some questions, based on building a simple benchmark.

We're only including `mypy` once in the grading rubric, but if your part 1 submission is fine and your final submission has warnings or errors, we'll still take off the point.

9 Optional Part 3: Performance

Your instructor will tell you whether this is required for you, or whether it's just here for you to play with on your own.

Right now, the code, as you have it is just using the modular exponentiation that's built into Python / GnuMP, and those modular exponentiation operations are far-and-away the performance bottleneck in this computation. You can even imagine real-world scenarios, such as an in-precinct optical paper-ballot scanner, which might have a relatively limited CPU. If it's too slow, then voters might have to queue up to deposit their ballots!

There are a *lot* of ways that you could improve the performance of ElectionGuard without changing what you're computing. For now, we'll focus on the easiest "low hanging fruit" optimization available to you. For *most* of the modular exponentiation computations, they tend to have either g (the generator) or g^a (the public key) as their base, while the exponent is going to be an arbitrary number in $\mathbb{Z}_q^*$. We're going to speed these up through pre-computation.

- To optimize `g_pow_p`, We might precompute an array of exponentiations of g: $\left[g, g^2, g^4, \ldots, g^{\left(2^{255}\right)}\right]$. We can then decompose any value $x \in \mathbb{Z}_q^*$ into its component bits $x_0, x_1, \ldots x_{255}$ allowing us to restate $g^x = \prod_i g^{\left(x_i 2^i\right)}$. And if any x_i is zero, we can just skip over it and move on to the next one. We might expect half of those x_i's to be zero, so we only have to do around 128 multiplies.

- Instead, we're going to do even better by doing additional precomputation. Let's say you break down x into its constituent bytes. It only takes 32 bytes to represent any value in $\mathbb{Z}_q^*$. For each of those 32 bytes, you need a separate array of 256 entries. You look up the result for each byte, and multiply all those together. A modular exponentation now requires only 32 lookups and 31 multiplies!

- The more effort and storage we dedicate to precomputation, the faster we'll be able to compute our modular exponentiations, but it's a time-space tradeoff with diminishing returns. For example, if we instead did 10-bit lookups rather than 8-bit lookups, we would now have 26 arrays of size 2^{10} rather than 32 arrays of size 2^8. Storage would grow by a factor of 3.25x and in return we would only shrink the computation from 31 multiplies down to 25 multiplies.

- You can repeat exactly the same process for using g^a (the public key) as the base, generating another table of precomputed exponentiations. Exactly where should you put these acceleration structures? How should you reengineer the existing `pow_p` function? That's really up to you. You may choose to get fancy with object-oriented subclassing of `ElementModP`, or you may choose to strategically deploy `if` statements and/or global variables. Ideally, the rest of the codebase should be completely unaware of the optimizations that you've done.

- As you work on `g_pow_p` and `pow_p` in `group.py`, you'll most likely want to take advantage of the `q_to_bytes` function, already present there, to extract the bytes that you'll be using. Note that you won't always get 32 bytes out. Smaller numbers will yield shorter arrays.
- What happens if somebody passes you an exponent that's outside of $[0, Q)$? Is that even possible? You can always add a precomputation step that computes your exponent mod Q.
- All the existing unit tests will help you validate that you haven't broken anything. Feel free to create additional unit tests specifically to exercise your optimized code against the original tried-and-true code. Once you're confident that your code is correct, rerun your benchmark from part 2 to measure how much you sped things up. You should make suitable edits to your `README.md` file to indicate the performance boost that you were able to achieve.