Exercise 1

Inspired by proof-of-stake, Amir would like to create an Ethereum smart contract in which people can lock up their funds (in ether) and then vote on a very important matter. The matter can be resolved in one of k ways, thus each person's vote is an integer $\{1, 2, ..., k\}$. Amir is asking for your help. Specifically, he would like to have a protocol and a smart contract that provides as many of the following functionality/desired properties as possible:

1. Anyone on the Ethereum network can sign up in the protocol by depositing 1 ether. One may sign up several times and thus pay a larger overall deposit.
2. At the end of the protocol, everyone can get their deposits back. The returned value may be a bit smaller than the original deposit, e.g. 0.9 ether, if you need to pay for something in your protocol.
3. Each deposit of 1 ether entitles the depositor to one vote.
4. After a particular deadline t_1, the contract stops accepting deposits and instead only accepts votes by those who have already deposited money.
5. No one can vote twice for the same deposit.
6. After a later deadline t_2, the contract stops accepting votes.
7. After an even later deadline t_3, everyone with access to the blockchain should be able to figure out which option from the set $\{1, 2, ..., k\}$ received the maximum number of votes. If there is a tie, you can break it arbitrarily as you wish.
8. No one should be able to know another person's vote.
9. If Alice has voted for $i \in \{1, 2, ..., k\}$ and she wants to prove this to Bob, she should be able to do so. You can assume that Bob has access to the blockchain.
10. Your contract should not have any of the security vulnerabilities discussed in the lectures.
11. Your contract should use as little gas as possible and be affordable. You should find out exactly how much gas each one of your functions uses in the worst case and report it in your pdf file.

Finally, you are allowed to have participants in the contract who take roles other than voting, but any such participants should be well-incentivized, i.e. you should argue why they can make money by helping you run your contract if your protocol depends on their participation.

1. **Protocol**

   (a) **Phase 0: Initialization** When a election contract is constructed:
   - The deployed block number is recorded in variable deployedBlockNumber.
   - The address of the signer, who's a third party responsible for blind signing in phase 2, is specified as variable signer.
   - The modulo N to be used in the blind signing phase is specified as variable N.
   - The number of options/candidates is specified as variable k.

   (b) **Phase 1: Registration**
   - The voters deposit 0.5 ETH to register for a vote calling the function voter_register().
     - The address of the voter corresponds to a vote ID is stored in the mapping voter.
     - The deposit corresponds to a vote ID is stored in the mapping D.
     - Registering for multiple votes is allowed by repeatedly deposit.
     - Noted that the other deposit of 0.5 ETH will be made later at the voting phase.
   - **Time range**: block.number $< t_1$

   (c) **Phase 2: Blind Signing**

   i. Pre-signing
   - Each voter creates a random 256 bits serial number sn_i for a vote.
     (The following procedures are all for this vote.)
   - The voter creates a message m_i which is the hashed sn_i, denoted as $m_i = sha256(sn_i)$.
   - The voter chooses a random number r_i and computes $m_i' = m_i {r_i}^e$, where e is the public key of the signer.

   ii. Request signing
   - The voter requests a blind signature from the signer by calling request_sign().
     `request_sign(depositID)`
     - When the voter request for a blind signature, 0.2 ETH of their deposit stored in the mapping D will be moved to the another mapping signD, which is storing deposit related to signing.
       The first 0.1 ETH will be used as deposit to make sure that the voter will honestly claim that he received the signature later. The other 0.1 ETH will be paid to the signer as the signing fee.
     - The function update the signing status of the vote stored in the mapping signStatus to 1.
     - **Time range**: $t_1 \leqslant$ block.number $< t_{1.1}$
   - The voter sends the m_i' to the signer off-chain.

   iii. Sign
   - The signer signs on m_i' and sends the signature $s_i' = {m_i'}^d$ to the voter, then he calls signed() to claim that he has signed.
     `signed(depositID)`
     - When the signer call signed(), he needs to deposit 0.1 ETH to prevent that he fake that he signed.
     - The function update the signing status of the vote to 2.
     - **Time range**: $t_{1.1} \leqslant$ block.number $< t_{1.2}$

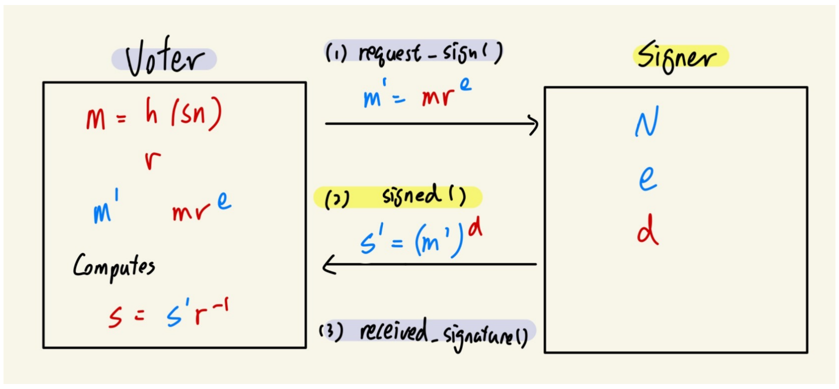
   iv. Receive signature
   - Upon receiving s_i', the voter calls received_signature() to claim that she has received the signature.
     `received_signature(uint depositID, uint256 mdPrime)`
     - The voter also commits the ${m'}^d$ when calling received_signature().
       This is in case of the voter wants to prove her choice to other later. (Check Analysis().)

- 0.1 ETH will be moved back to D[depositID] as the refund of the voter's deposit.
- 0.2 ETH will be transferred to the signer as the refund of his deposit plus the award of signing.
- **Time range**: $t_{1.2} \leqslant$ block.number $< t_{1.3}$

v. Post-signing

- The voter can compute $s_i = s_i' r_i^{-1}$ and get the signature she needs.

vi. **Over all time range**: $t_1 \leqslant$ block.number $< t_{1.3}$



(d) **Phase 3: Voting**

- Each voter decides on her vote(s) $v_i \in \{1, 2, ..., k-1\}$.
- The voter chooses a nonce n_i and computes the hash $hv_i = sha256(v_i, n_i)$.
- The voter computes the hash of the signature s_i she got from the signer hs_i
- The voter votes by calling vote() with a freshly generated blockchain address.

  `vote(sn, hs, hv)`

  - The voter needs to deposit the other 0.5 ETH when calling vote() to make sure that she'll reveal her vote later. Thus, once she called vote(), the deposit she made should be 1 ETH in total.
  - The address of the anonymous voter will be stored in the mapping hiddenVoter.
  - hs_i corresponds to sn_i will be stored in the mapping signatureHahses
  - hv_i corresponds to sn_i will be stored in the mapping voteHashes
  - If Alice has multiple votes, she should generate different addresses for calling vote() for each vote.
- **Time range**: $t_{1.3} \leqslant$ block.number $< t_2$

(e) **Phase 4: Vote revelation**

- Each participant calls the function reveal() to reveal the real vote.

  `reveal(sn, s, v, n)`

  - The revealing vote v_i must be in $\{0, 1, ..., k-1\}$, else the revelation will fail.
  - The s_i must be the signer's blind signature on sn_i. This is verified by checking whether $(s_i)^e \mod N = sha256(sn_i)$.
  - If the hashes hv_i, hs_i both match, the vote is added to the corresponding candidate in the mapping count.
  - The current highest number of votes obtained and the candidate will also be updated if needed.
  - If the revelation is done successfully, the voter can refund her voting deposit (0.5 ETH).
- **Time range**: $t_2 \leqslant$ block.number $< t_3$

(f) **Phase 5: Deposit Withdrawal**

- Each participant withdraw his initial deposit during registration by calling the function withdraw().

- The amount of withdrawl should be:
  - 0.5 ETH if the voter requested for signature but the signer didn't respond.
  - 0.4 ETH if the voter requested for signature and received the signature successfully.
  - 0.3 ETH if the either the voter or the signer cheated by faking that they did not receive signature/sent the signature.
- **Time range**: $t_3 \leqslant$ block.number $< t_4$

2. **Analysis**

(a) **Why does the voter only deposit 0.5 ETH during registration?**
It is stated that each deposit of 1 ETH grants the depositor one vote in the exercise description. However, in this protocol, a commit-reveal scheme is implemented to ensure the fairness of the vote. To penalize malicious participants who refuse to reveal their votes, a deposit during the vote commitment phase (Phase 3) is necessary. Additionally, since a blind signature protocol is implemented in Phase 2 to conceal the voters' true identities, we cannot deduct money from their initial deposit during registration, as it would compromise their anonymity. Hence, a second deposit is required during Phase 3, which will be paid by the address newly generated by the voter for voting. In summary, the smart contract ask for a deposit of 0.5 ETH during the registration, and the other 0.5 ETH during the voting. That is, 1 ETH is required for each vote.

As a voter should have already paid the signer a 0.1 ETH signing fee during Phase 2, they should be willing to vote and, therefore, deposit the second 0.5 ETH when entering Phase 3. If a voter decides to forfeit their vote, they simply refrain from depositing and voting during Phase 3 and can still retrieve the remaining registration deposit during Phase 5.

(b) **Can the voter vote twice for the same deposit?**
In the function request_sign(), the signing status of the corresponding depositID (signStatus[depositID]) must be 0, which is the default value in the mapping. Upon a successful call to request_sign(), the signStatus[depositID] is updated to 1. Therefore, each voter can only request a signature once for each deposit. As one blind signature corresponds to one vote in the voting phase, the voter can only cast a single vote for the same deposit.

(c) **What's the incentives for the signer to sign**
The signer can earn 0.1 ETH as the signing fee for each blind signature he signed.

Signing Status after Phase 2	Description	Voter/Signer payoffs
1	Voter requested but no response from the signer.	0/0
2	(Voter cheated that no signature received) OR (Signer cheated that he had signed.)	-0.2/-0.1
3	Voter received signature successfully.	-0.1/0.1

In the case that either party intends to engage in dishonest behavior, the signing status of the deposit will ultimately be marked as status 2 during phase 2. This outcome is unfavorable for both parties involved, resulting in the highest potential loss. Assuming that the signer is rational, he should follow the protocol honestly to maximize his payoff. Similarly, the voter should also honestly report the receipt of the signature to ensure the refund of his deposit.

(d) **Can anyone know another person's vote?**
No, it is not possible for anyone to know another person's vote.
The blind signature protocol implemented in Phase 2 ensures that the voter's identity is completely dissociated from their vote. Each voter is required to use a newly generated address for voting, which helps maintain anonymity. If a voter has multiple votes, they are instructed to generate different addresses for each vote. This practice prevents any correlation between the voting address and the deposit address based on the number of votes or deposits made. Thus, the system safeguards the privacy of each individual's vote.

(e) **How does Alice prove her vote to Bob?**
Suppose Alice wants to prove to Bob that she voted for option 2, and the serial number of the blind signature used for the vote is sn_A. Since the protocol disassociated the depositor's identity and the voter's identity using blind signature, Alice needs to associate the two together in order to complete the proof.

Recall that Alice committed ${m'_A}^d$ when she called received_signature() in phase 2, which made her ownership of ${m'_A}^d$ public on the blockchain. Now, Alice shows that the blind signature s_A belongs to her by sending Bob r'_A. Using the equation ${m'_A}^d = {m_A}^d {r_A}^{ed} = {m_A}^d r_A = s_A r_A$, Bob can verify whether ${m'_A}^d {r_A}^{-1} = s_A$ upon receiving r_A. If it is confirmed that s_A belongs to Alice, Bob can further check on the blockchain to see if Alice called reveal(sn_A, s_A, 2, n_A). This step allows Alice to complete the proof.

(f) **Attack by tampering with committed vote during phase 3**
When someone commit his vote on the blockchain, his serial number(sn), hash of blind signature(hs), and hash of vote option (hv) become public on the blockchain. A malicious player may want to change his hv by calling vote() again with the same sn, hs but a different hv. This is prevented by the following lines in vote():

```
require(voteHashes[sn] == bytes32(0));
voteHashes[sn] = hv;
```

Since the voteHashes[sn] will be set to the address of the voter after the first function call. and the vote() requires voteHashes[sn] to be default, the malicious player's will fail to modify a committed hv.

(g) **Re-entrancy attack on reveal()**
Since the voter also gets refund of his voting deposit during revelation, a malicious voter may want to call reveal() multiple times with the same vote to steal the money. This is prevented by setting the corresponding hash of the vote stored in the mapping voteHashes to 0 before the contract sends the refund in his first function call on withdraw().

```
voteHashes[sn] = 0;
```

(h) **Re-entrancy attack on withdraw()**
A malicious depositor may want to call withdraw() function multiple times for the same deposit to steal the money. This is prevented by setting the corresponding registration deposit stored in the mapping D to 0 before the contract sends the refund to him in his first function call on withdraw().

```
D[depositID] = 0;
```

(i) **Gas**

- voter_register(): 103885
- request_sign(): 84897
- signed(): 39396
- received_signature(): 87620

- vote(): 105545
- reveal(): 117389
- withdraw(): 78647