1. **Alternate proof-of-work**: For a hash function $H: \{0,1\}^* \rightarrow \{1,\ldots,2^{256}\}$, consider the following puzzle: given a challenge $c$ and a difficulty $d$, find nonces $n_1 \neq n_2$ such that:

$$H(c,n_1) = H(c,n_2) \pmod{d}$$

That is, the miner must find two nonces that collide under $H$ modulo $d$. Clearly, puzzle solutions are easy to verify and the difficulty can be adjusted granularly.

   a. A simple algorithm is to repeatedly choose a random nonce $n$ and add $(n, H(c,n))$ to a set $L$ stored to allow efficient search by $n$ (such as a hash map). The algorithm terminates when the last hash value added to $L$ collides with some hash value already in $L$. For given values of $d$, approximately how many invocations of $H$ are required in expectation to generate a solution $n_1,n_2$? How much memory does this use?

   **Hint**: it will be helpful to familiarize yourself with the birthday paradox.

   b. Consider an algorithm with limited memory that chooses one random nonce $n_1$ and then repeatedly chooses a random nonce $n_2$ until it finds an $n_2$ that collides with $n_1$. How many invocations of $H$ will this algorithm use in expectation? We note that there is a clever algorithm that finds a solution in the same asymptotic time as part (A), but using only constant memory.

   c. A puzzle is *progress-free* if for all $h,k$ (where $h\cdot k < B$ for some large bound $B$) the probability of finding a solution after computing $h\cdot k$ hashes is $k$ times higher than the probability of finding a solution after computing $h$ hashes. Is this puzzle progress-free?
2. Bitcoin-NG. In Bitcoin, every block contains a set of verified transactions along with a proof of work. The Bitcoin-NG proposal turns the process on its head. Instead of publishing a set of transactions in each block, the miner who finds a block simply includes their signing public key. This is called a key block. They then sign a series of micro blocks containing actual transactions using this key (and no further proof of work) until another miner solves the proof of work puzzle and takes over.

As usual, in case of a fork, the longest chain wins. The transaction fees from each transaction are split, with a portion $\beta$ going to the miner publishing them and the remaining $(1-\beta)$ proportion going to the next miner to find a key block, for some fixed $\beta$ in $[0,1]$.

a. How does this proposal affect scalability? Specifically, how does it impact the number of transactions which can fit into a fixed amount of space on the blockchain?

b. What is the benefit of this proposal?

c. Consider a miner who has found a key block and has a proportion $\alpha \in [0,1]$ of the total mining power. They might keep their chain of microblocks secret, only publishing it if they themselves find the next key block, otherwise allowing the next key block to mine directly on top of their key block with no intervening microblocks. When will this devious strategy be in the miner’s interest, in terms of $\alpha$ and $\beta$?

d. Similarly, consider a miner looking for a key block. They can mine on top of the previous miner’s published micro blocks as usual, or they can mine directly on top of the previous key block in hopes of finding two blocks in a row and keeping all of the previous microblock’s fees for themselves. When will this devious strategy be in the miner’s interest, in terms of $\alpha$ and $\beta$?

e. Given your results in parts (c) and (d), what is a reasonable choice for $\beta$ in practice?
3. **Ethereum micropayments.** In class we saw a protocol for serial micropayments in Bitcoin: Alice wants to send a series of payments to Bob, so she puts $n$ units of currency into an escrow account which is a multisig account shared by Alice and Bob. She then repeatedly signs transactions, the first paying 1 unit to Bob and returning $n-1$ to Alice, the second paying 2 units to Bob and returning $n-2$ units to Alice, and so on. Alice stops sending transactions whenever she wishes, and Bob can sign and publish the last transaction received. A time-locked refund transaction is used to ensure Alice can recover her money if Bob goes offline.

Let’s consider solving this problem in Ethereum. It is very easy to implement the above logic. We can actually add a cool feature: each micropayment need only require hashing, rather than signatures. This is perfect for lightweight clients (although initialization will require a signed message). The scheme works as follows: Alice picks a random $X$ and computes $Y = H^n(X)$ [that is, iterate the function $H$ $n$ times starting at $X$, e.g., for $n=2$ we have $Y = H(H(X))$]. She then creates a contract with $n$ units of currency and embeds the value $Y$ in the contract (and sends $Y$ to Bob).

To pay Bob $k$ units (for $k<n$), she sends Bob the value $Z = H^{n-k}(X)$.

a. Describe (in pseudocode or prose) how the contract should decide how much money to send Alice and/or Bob (and then call SUICIDE).

b. Describe a frontrunning attack whereby Alice can try to convince the contract to pay less to Bob than she actually authorized. How might you design the contract to prevent such an attack?

c. What security property should the hash function satisfy to ensure that Bob cannot steal more money than Alice intended to give him?

d. For a given $n$ and $k$, how expensive is this protocol: how many hashes does the contract need to compute before distributing funds? How much data will Alice and Bob need to store?

e. In practice, we would like to minimize the resources consumed by the contract above all else as gas is expensive. Since the above scheme is a linear chain, you might guess it can be improved using a tree structure. You’d be right! Describe this tree structure. What are the storage and computation costs (in terms of $n$ and $k$) for Alice, Bob, and the contract?
4. **Off-chain payment networks.** In class we discussed the use of payment channel networks to improve Bitcoin scaling. Consider two possible network scenarios for a group of 4 people. Assume all payment channels are bidirectional. Each has an initial capacity, the maximum the balance can deviate from zero in either direction. If a payment is executed which increases the channel balance to capacity, the channel is reset. We’ll consider a simple model where every second, a random user sends 1 unit to a different random user. Assume there are $N$ users total and each user is willing to put $M$ total coins on deposit.

- **a.** For each scenario, how many channels will be required for $N$ users?
- **b.** For each scenario, what will be the capacity of each channel given that each user will put $M$ total coins on deposit?
- **c.** For each scenario, for what fraction of random payments will a given channel be used?
- **d.** For each scenario, after how many transactions (on average) will some channel need to be reset?

**Hint:** A useful fact from the theory of random walks is that the “time to complete” (starting from 0) with upper and lower boundaries $(n, -m)$ is $nm$. For example, in a single channel of capacity $n$ with random payments back and forth, an average of $n^2$ payments can be made before one will push the balance to $n$, requiring a channel reset.

**Note:** Your answer, divided by 2, reflects the scalability increase of using payment channels over conventional blockchain payments, since every reset requires two transactions on the blockchain.

- **e.** What are some disadvantages of the hub-and-spoke model?
- **f.** Why is the random payment model in this question unrealistic? Briefly describe how you might improve efficiency by taking advantage of a more realistic distribution of payments.