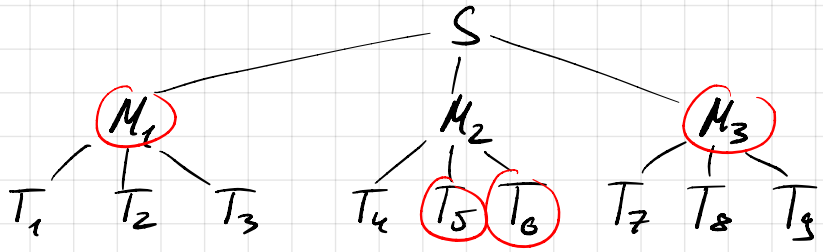
1) Suppose the attacker manages to find a $z \in \{0,1\}^{256}$ such that $SHA256(z) < \frac{2^{256}}{D}$. He can find z before any x is published by trying a large amount (D on average) of samples $\hat{z} \in \{0,1\}^{256}$ by brute force. Now, given a new x, he can simply set $y = x \oplus z$. This y solves the proof of work problem for the given x as:

$$SHA256(x \oplus y) = SHA256(x \oplus (x \oplus z))$$
$$= SHA256(z) < \frac{2^{256}}{D}$$

since applying bit-wise xor twice is the identity.

2) a)



The tree explains how S is created from the list. Each node is created by concatenating the values of all its children and hashing them. Alice can prove that T_4 is in S by providing Bob with (T_5, T_6, M_1, M_3). Bob can redo the calculation of S and as the hash function is collission resistant conclude that T_4 was correct.

b) The proof contains $\underbrace{(k-1)}_{\text{elements needed to get a layer closer to the root}} \cdot \underbrace{\lceil \log_k(n) \rceil}_{\text{tree depth}}$ elements.

c) $g_k(x) := (k-1)\log_k(x)$

$$\frac{g_2(x)}{g_3(x)} = \frac{\log_2(x)}{2\log_3(x)} = \frac{\log_2(3)}{2} < \frac{\log_2(4)}{2} = 1$$

$\Rightarrow$ Using $k=2$ leads to a shorter proof for large n. Hence, a binary tree should be used to construct the Merkle root.

3) a) ScriptSig: P

b) An attacker might guess that the password is only few characters long and starts bruteforcing all passwords starting from length 1 to length 6. When this can be done in reasonable time, he would stop when the hash of one of the tried passwords matches the hash in ScriptPK. This would allow him to redeem the UTXO.

c) In order to redeem her bitcoins, she could create a new UTXO containing her password in ScriptSig. The moment she sends this UTXO out, any miner could simply change the UTXO such that Alice's money goes to one of his own addresses as the password needed to make ScriptPK return true is readily available.

4) a) Using locktime to prevent the prize from being claimed early:

ScriptPK: <locktime>

OP_CHECKLOCKTIMEVERIFY

OP_DROP

OP_DUP

OP_HASH256

<addr>

OP_EQVERIFY

OP_CHECKSIG

The UTXO with this ScriptPK can only be redeemed with a ScriptSig containing <sig> <pk> where sig is the signature of the spending UTXO using the private key of the lottery. Moreover, the locktime in ScriptPK has to be smaller than the time specified in the spending UTXO.

b) One way to do this would be to create UTXOs that can be redeemed not only by this week's private key but also by all the future winning private keys, given these are already known in week 1. This is possible using a 1-out-of-(1001-n) multisig for the UTXO in week n. In this implementation, the lottery has to pay attention that the winning private key for week n only goes into circulation at week n. Otherwise, a winner could redeem UTXOs of future weeks.

The ScriptPK should look like:

OP_1 <addr$_n$> ... <addr$_{1000}$> OP_(1001-n) OP_CHECKMULTISIG

A ScriptSig with any correct signature created using the privatekey from week n up to thousand can now redeem the UTXO:

ScriptSig: OP_0 <sig$_i$> $i \in \{n, ..., 1000\}$

As multisig allows for maximally 1000 public keys this approach only works for the first 1000 weeks or less.

5) a) Alice would need to show Bob

(i) the Merkle proof of the block her transaction is contained in

(ii) the Merkle roots of all blocks between the block of her transaction and the second to latest block

(iii) the hash of the block right before the block containing her transaction.

Given this information, Bob can calculate the Merkle root of the block containing Alice's transaction and then create the hash of this block given the hash of the previous block. With the subsequent Merkle roots, Bob can work his way through the chain until he verifies if his calculated hash of the previous block matches the actual hash as shown by his client.

b) Merkle proof size: $\lceil \log_2(n) \rceil \cdot 32$ bytes

Merkle roots size: $(k-1) \cdot 32 \; \mathbb{1}(k>0)$ bytes

Block hash size: $32 \; \mathbb{1}(k>0)$ bytes

$\Rightarrow$ Size of proof for $k = 6$, $n = 1024$:

$10 \cdot 32 + 5 \cdot 32 + 32 = 16 \cdot 32$

$= 512$ bytes