Principles of Cryptocurrency Design

Esercitazione

Francesco Pasquale

16 maggio 2024

1 Consenso e Proof-of-Work

Si vedano i file in `pcd240416`.

Esercizio 1. Scrivere un programma che legga un file in cui ogni riga è una sequenza di 152 caratteri esadecimali e verifichi se la sequenza di righe definisce una catena *valida*. Una catena è valida se:

1. Ogni riga del file corrisponde alla serializzazione di un'istanza dell'oggetto `Block`, così come definito in dal metodo `serialize` alla riga 35 in `block.py`, espressa in esadecimale.
2. Il `prev_hash` del blocco nella prima riga è l'hash del *genesis block* che si trova nel file `config`, e per ogni $i > 1$ il `prev_hash` del blocco nella riga i-esima è l'hash del blocco nella riga $(i-1)$-esima
3. Per ogni riga, l'hash del blocco in quella riga è inferiore al target.

Nella directory `testcases/` trovate due file con due catene da mille blocchi ciascuna: una delle due catena è valida, l'altro no. Per il file che contiene la catena non valida, identificare la prima riga non valida e quale dei punti qui sopra non è soddisfatto.

Esercizio 2. Nel codice scritto a lezione abbiamo usato un un `target` costante. Definire un tempo medio `DELTA` che vogliamo imporre fra la creazione di due blocchi consecutivi (diciamo `DELTA = 120` secondi) e un numero `EPOCH_LEN` che corrisponde al numero di blocchi prima di riaggiornare il target (diciamo `EPOCH_LEN` $= 60$). Inserire i due valori nel file `config`.

Modificare il codice in `block.py` in modo che, `target`(0) sia quello del *genesis block* e viene usato per i primi `EPOCH_LEN` blocchi, ma ogni volta che il numero di blocchi creati è $k *$ `EPOCH_LEN` per qualche $k \geqslant 1$, per i successivi `EPOCH_LEN` blocchi il target sia dato dalla formula

$$\texttt{target}(k) = \texttt{target(k-1)} \cdot \frac{\delta}{\texttt{DELTA}}$$

dove con δ abbiamo indicato la differenza fra il `timestamp` del blocco $k *$ `EPOCH_LEN` e il `timestamp` del blocco $(k-1) *$ `EPOCH_LEN`.

Esercizio 3. Alla linea 58 del file `block.py` incrementiamo `block.nonce` di una unità, senza preoccuparci del fatto che questo numero deve essere a 4 byte e quindi non può in ogni caso superare $2^{32} - 1$.

Modificare il codice inserendo la condizione che, se `block.nonce` dovesse raggiungere 2^{32}, `block.nonce` ripartirebbe da 0 e `block.timestamp` verrebbe aggiornato.

Esercizio 4. Scrivere un programma che legga un file e, se contiene una catena *valida* così come definito nell'Esercizio 1, restituisca la *proof-of-work* della catena, ossia la somma, per ogni blocco, di 2^{256} diviso il `target` del blocco.

2 Transazioni

Si vedano i file in `pcd240423`.

Esercizio 5. Nel file `helpers.py` abbiamo implementato la funzione `varint2int` che prende in input uno *stream* di byte e restituisce il numero intero corrispondente, secondo le specifiche definite qui:
`https://developer.bitcoin.org/reference/transactions.html#compactsize-unsigned-integers`

Scrivere una funzione `int2varint` che esegua l'operazione inversa: prenda in input un intero non negativo minore di 2^{64} e restituisca una sequenza di byte che codifichi l'intero secondo le specifiche.

Esercizio 6. Nel file `transaction.py`, per ognuna delle classi `Tx`, `TxIn` e `TxOut` abbiamo implementato un metodo di classe `parse` che prende in input una sequenza di byte opportuna e restituisce l'oggetto codificato nella sequenza.

Per ognuna delle classi, scrivere un metodo `serialize` che restituisca una sequenza di byte contenente la serializzazione dell'oggetto della classe.

Esercizio 7. Scrivere un metodo `tot_out` per la classe `Tx` che restituisca la somma degli `amount` contenuti negli output della transazione.

Esercizio 8. Scrivere un metodo `fee` per la classe `Tx` che restituisca la *transaction fee* della transazione. La transaction fee è la differenza fra la somma degli `amount` contenuti negli output puntati dagli input della transazione e la somma degli `amount` contenuti negli output della transazione. Per recuperare gli output puntati dagli input di una transazione potete usare, per esempio, le api di un qualche sito che consente di recuperare dati dalla Blockchain, come mostrato per esempio nel file `get_tx.py`.

Esercizio 9. Progettare e implementare un metodo della classe `Tx` che restituisca in output un albero di cui l'istanza dell'oggetto è il nodo radice. I figli di un nodo/transazione u sono le transazioni i cui output sono puntati dagli input di u. Le foglie di un tale albero perciò saranno tutte transazioni *coinbase*.

3 Script

Si vedano i file in `pcd240423`.

Esercizio 10. Nel file `script.py` abbiamo implementato un metodo di classe `parse` che prende in input una sequenza di byte opportuna e restituisce l'oggetto codificato nella sequenza. Scrivere un metodo `serialize` che restituisca una sequenza di byte contenente la serializzazione dell'oggetto della classe.

Esercizio 11. Fare il parsing del seguente *locking script* `6e879169a87ca887`. Usando le descrizioni degli OP_CODES[1], determinare cosa dovrebbe contenere un *unlocking script* per ottenere uno script che restituisca TRUE.

[1] Le trovate, per esempio, qui https://wiki.bitcoinsv.io/index.php/Opcodes_used_in_Bitcoin_Script

4 Curve ellittiche e indirizzi Bitcoin

dove a e b sono parametri che definiscono la curva, con l'aggiunta di un punto che chiamiamo *punto all'infinito* $\{\infty\}$.

Esercizio 12. Scrivere un programma che prenda in input i parametri a e b e gli estremi di un intervallo di numeri reali, $[x_0, x_1] \subseteq \mathbb{R}$ e disegni il grafico della curva ellittica

$$y^2 = x^3 + ax + b \tag{1}$$

al variare di $x \in [x_0, x_1]$.

Esercizio 13. Scrivere un programma per verificare che

$$p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$$

è un numero primo.

Esercizio 14. Il punto base della curva SECP256K1 è $G = (x, y)$, dove

$x =$ `0x 79BE667E F9DCBBAC 55A06295 CE870B07 029BFCDB 2DCE28D9 59F2815B 16F81798`
$y =$ `0x 483ADA77 26A3C465 5DA4FBFC 0E1108A8 FD17B448 A6855419 9C47D08F FB10D4B8`

Verificare che le coordinate (x, y) qui sopra soddisfano l'equazione $y^2 = x^3 + 7 \mod p$.

Si vedano i file in pcd240509.

Esercizio 15. A lezione abbiamo implementato il metodo WIF, che restituisce la chiave privata nel formato *Wallet Import Format (WIF)*. Scrivere un metodo di classe PARSE_WIF che legga una stringa in formato WIF e restituisca l'oggetto corrispondente della classe ADDRESS.

Esercizio 16. Abbiamo visto che uno degli indirizzi contenuti negli output script P2PKH della transazione `b35d91a71f226ba961162ca18f321b4d9aada8a0e722430ad3d2d2e4dda9a2c0` ha come chiave privata l'hash `sha256` della stringa `Francesco`. Quella transazione ha 500 output del tipo P2PKH tutti con lo stesso ammontare. Scrivere un programma *brute-force* che cerchi di individuare se in quella transazione ci sono altri indirizzi che hanno come chiave privata l'hash `sha256` di altri nomi.

Esercizio 17. Con la transazione testnet `da9c8ebf9861da00cc0cdfb6b7acc5a082903c67d4e89209005e68b65509eb10` abbiamo inserito nell'output script P2PKH un indirizzo che aveva come chiave privata l'hash `sha256` della stringa `Matteo` e abbiamo visto che quell'output è stato immediatamente speso da qualche bot che sulla rete ha individuato facilmente la chiave privata.

Ottenere dei bitcoin testnet tramite qualche faucet, provare a eseguire delle transazioni verso indirizzi con chiavi private facilmente individuabili tramite brute-force e vedere quanto tempo "resistono" prima che qualche bot ne individui la chiave privata e li spenda.