Principles of Cryptocurrency Design

Esercitazione

Francesco Pasquale

7 maggio 2026

1 Consenso e Proof-of-Work

Per gli esercizi di questa sezione si vedano i file in `https://www.mat.uniroma2.it/~pasquale/dida/aa2526/pcd/pcd260413.zip`

Esercizio 1. Il programma `run.py` genera una sequenza di (header) di blocchi a partire dal genesis block di Bitcoin e li scrive a video.

1. Modificare il programma per fare in modo che ogni header di 80 byte venga scritto anche su un file `blockchain.dat`.
2. Scrivere un programma che legga un file `blockchain.dat` e verifichi se contiene una catena di blocchi consistente con lo schema implementato. Ossia che
   (a) I primi 80 byte coincidono con quelli del *genesis block* di Bitcoin.
   (b) Ogni blocco successivo di 80 byte contiene nel campo `prev_hash` l'hash del blocco precedente;
   (c) L'hash di ogni blocco è inferiore al `target` contenuto nel blocco.

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 (per esempio, `DELTA = 120` secondi) e un numero `EPOCH_LEN` che corrisponde al numero di blocchi prima di riaggiornare il target (per esempio, `EPOCH_LEN` $= 60$).

Modificare il codice in `block.py` in modo che ogni volta che il numero di blocchi creati è $k * \texttt{EPOCH\_LEN}$ 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 $k * \texttt{EPOCH\_LEN}$ e il `timestamp` del blocco $(k-1) * \texttt{EPOCH\_LEN}$.

Esercizio 3. Scrivere un programma che legga un file `blockchain.dat` e

1. Verifichi se contiene una catena che soddisfa i due punti dell'Esercizio 1;
2. Verifichi che ogni blocco contenga un target consistente con quello definito nell'Esercizio 2;
3. Restituisca la *proof-of-work* della catena, ossia la somma, per ogni blocco, di 2^{256} diviso `target` $+ 1$.

Esercizio 4. Scrivere un programma che legga un file `blockchain.dat` e, se contiene una catena che soddisfa i punti degli Esercizi 1 e 2, inizi ad aggiungere blocchi alla catena.

Esercizio 5. Prendiamo un blocco appena creato, il n. `945350`), della blockchain di Bitcoin.

1. Usando i programmi scritti nella lezione precedente e l'header del blocco, verificare che il timestamp del blocco è 1776355415, ossia lo unix time corrispondente a *gio 16 apr 2026, 18:03:35*, e che il target di quel blocco è $132740 * 2^{160}$.

2. Sapendo che il `target` è impostato in modo che, in media, viene creato un nuovo blocco ogni dieci minuti, stimare l'*hash rate* h (ossia il numero di hash per secondo) che i miner stanno complessivamente eseguendo attualmente.

3. Supponete che un extraterrestre arrivi sulla terra con un *hardware* che riesce a fare k volte più hash al secondo di quanti ne fanno tutti i miner attuali messi insieme, per qualche $k > 1$, e decida di voler usare il suo hardware per riscrivere l'intera blockchain di Bitcoin a partire dal *genesis block*. Assumendo che l'hash rate degli altri miner rimanga costante, calcolare approssimativamente quanto tempo passerebbe prima che, in accordo al protocollo Bitcoin, tutti riconoscano la blockchain dell'extraterreste come quella valida.

Esercizio 6. Assumendo che la funzione crittografica SHA256 sia un *random oracle*, ossia supponendo che dato qualunque $y \in \{0,1\}^{256}$ ogni volta che calcoliamo l'hash di un nuovo $x \in \{0,1\}^*$ si abbia che $\mathbf{P}\left(\text{SHA256}(x) = y\right) = 2^{-256}$. Qual è la probabilità che il prossimo blocco impieghi più di k minuti per essere creato, per $k > 10$?

2 Transazioni

Per gli esercizi di questa sezione si vedano i file in `https://www.mat.uniroma2.it/~pasquale/dida/aa2526/pcd/pcd260420.zip`

Esercizio 7. 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 8. Scrivere un metodo `hash` per la classe `Tx` che restituisca l'hash256 della serializzazione della classe.

Esercizio 9. Scrivere un metodo `tot_out` per la classe `Tx` che restituisca la somma degli `amount` contenuti negli output della transazione.

Esercizio 10. 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, per esempio `https://mempool.space/docs/api/rest#get-transaction-hex`.

3 Script

Esercizio 11. Qual è lo `unlocking_script` che consente di sbloccare un `locking_script` del tipo *p2pkh*?

Esercizio 12. L'istruzione OP_EQUAL estrae due elementi sottostanti dallo stack e inserisce nello stack TRUE se i due elementi sono uguali e FALSE altrimenti. Scrivere un `locking_script` che può essere sbloccato solo da chi conosce un messaggio `msg` il cui hash è `<msg_hash>`.

Esercizio 13. L'istruzione OP_NOT estrae un elemento dallo stack e, se è TRUE o FALSE ne inserisce la negazione nello stack, altrimenti inserisce nello stack FALSE. L'istruzione OP_VERIFY estrae un elemento dallo stack e, se è TRUE non fa nulla altrimenti interrompe l'esecuzione e la transazione è non valida. L'istruzione OP_SWAP estrae due elementi dallo stack e li reinserisce nello stack in ordine inverso. Usando queste e le altre istruzioni viste in precedenza, progettare un `locking_script` che può essere sbloccato solo con un `unlocking_script` che contenga un messaggio `<msg2>` diverso da un dato messaggio `<msg>` che però abbia lo stesso valore di hash (calcolato con OP_HASH160).

Per gli esercizi seguenti si vedano i file in `https://www.mat.uniroma2.it/~pasquale/dida/aa2526/pcd/pcd260427.zip`

Esercizio 14. Nel file `script.py` abbiamo implementato una classe *Script* e un metodo di classe `parse` che prende in input uno stream di byte consistente con la serializzazione degli script in Bitcoin e la lunghezza in byte dello script, e restituisce un oggetto *Script* con la sequenza di comandi codificati nello stream di byte.

Scrivere un metodo `serialize` per la classe *Script* che restituisca una sequenza di byte contenente la serializzazione dell'oggetto della classe.

Esercizio 15. 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.

Esercizio 16. Progettare un oggetto della classe Script che contenga due sequenze di byte, x e y, con y esattamente 32 byte, e tale che lo script restituisca TRUE se e solo se $\text{SHA256}(x) = y$.

Esercizio 17. Scrivere un interprete per un piccolo sottoinsieme del linguaggio *Script* (per esempio, considerando solo i comandi OP_DUP, OP_EQUAL, OP_VERIFY, OP_SWAP, OP_SHA256, OP_HASH256). L'interprete deve prendere in input l'istanza di un oggetto della classe Script, e restituire TRUE se lo *stack* al termine dell'esecuzione della lista dei comandi contiene un solo elemento che è 1, deve restituire FALSE in tutti gli altri casi.

4 Curve ellittiche e indirizzi Bitcoin

Possiamo descrivere una curva ellittica come l'insieme dei punti (x, y), a coordinate in un opportuno campo, che soddisfano l'equazione

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

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

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

Esercizio 18. 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 in (1) al variare di $x \in [x_0, x_1]$.

La curva ellittica che viene usata in Bitcoin si chiama secp256k1, ha come parametri $a = 0$ e $b = 7$ ed è definita sul campo $\mathbb{F}_p$ con $p = 2^{256} - 2^{32} - 2^9 - 2^8 - 2^7 - 2^6 - 2^4 - 1$.

Esercizio 19. Scrivere un programma per verificare che il numero p definito qui sopra è un numero primo.

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`

Esercizio 20. Verificare che le coordinate (x, y) qui sopra soddisfano l'equazione $y^2 = x^3 + 7 \mod p$.

Esercizio 21. L'ordine del gruppo generato dal punto base della curva è

$n =$ `0x FFFFFFFF FFFFFFFF FFFFFFFF FFFFFFFE BAAEDCE6 AF48A03B BFD25E8C D0364141`

Verificare che n è un numero primo.

Per gli esercizi seguenti si vedano i file in `https://www.mat.uniroma2.it/~pasquale/dida/aa2526/pcd/pcd260504.zip`

Esercizio 22. Usando la libreria `secrets` di python (o un altro generatore sicuro di numeri casuali) generare una chiave privata tale che l'indirizzo Bitcoin mainnet associato alla chiave inizi con `1PCD`.

Esercizio 23. Verificare che uno degli indirizzi contenuti negli output script P2PKH della transazione
b35d91a71f226ba961162ca18f321b4d9aada8a0e722430ad3d2d2e4dda9a2c0,
l'indirizzo `1CLJKodMLHhAwiDYFWDiwmz31cMfhqKnEc`, è l'hash di una chiave pubblica non compressa che ha come come chiave privata lo `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 24. 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.

Esercizio 25. A lezione abbiamo scritto un metodo WIF per la classe ADDR, che restituisce la chiave privata in formato WIF, e l'abbiamo usato per importare la chiave segreta in un wallet.

Scrivere un metodo di classe PARSE_WIF che legga una stringa di testo in *base58*, verifichi che gli ultimi quattro byte sono un *checksum* corretto dei glia altri byte secondo il formato WIF e restituisca l'oggetto corrispondente della classe ADDR.