Principles of Cryptocurrency Design

Esercitazione

Francesco Pasquale

30 marzo 2026

1 Il protocollo di Dolev-Strong

Consideriamo il protocollo di Dolev-Strong per il problema Byzantine Broadcast.

Algorithm 1 Dolev-Strong Protocol for Byzantine Broadcast

1: ROUND 0. Ogni nodo u inizializza un insieme vuoto $\mathcal{E}_u = \emptyset$.
2: La sorgente (il nodo 1) riceve in input b e invia $\langle b \rangle_1$ a tutti i nodi.
3: PER OGNI ROUND $r = 1, \ldots, f$. Ogni nodo u:
4: Per ogni messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1}}$ con r firme distinte (inclusa quella
5: della sorgente) ricevuto nel Round $r-1$:
6: SE $\hat{b} \notin \mathcal{E}_u$:
7: Aggiungi $\hat{b}$ a $\mathcal{E}_u$;
8: Firma il messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1}}$;
9: Invia il messaggio firmato $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_{r-1},\sigma_u}$ a tutti i nodi;
10: ROUND $f+1$. Ogni nodo u:
11: Per ogni messaggio $\langle \hat{b} \rangle_{1,\sigma_1,\sigma_2,\ldots,\sigma_f}$ con $f+1$ firme distinte (inclusa quella
12: della sorgente) ricevuto nel Round f:
13: SE $\hat{b} \notin \mathcal{E}_u$, aggiungi $\hat{b}$ a $\mathcal{E}_u$;
14: SE $\mathcal{E}_u$ contiene un solo elemento, allora OUTPUT l'elemento in $\mathcal{E}_u$,
15: Altrimenti OUTPUT 0

Esercizio 1. Alle linee 4 e 11 del protocollo ogni nodo onesto verifica, oltre al *numero* di firme distinte presenti (r firme per messaggi arrivati nel round $r-1$), che fra le firme ci sia anche quella della sorgente. Descrivere un attacco al protocollo che i nodi corrotti potrebbero mettere in atto se i nodi onesti non verificassero la presenza della firma della sorgente.

Esercizio 2. Descrivere un attacco al protocollo che i nodi corrotti potrebbero mettere in atto se fossero in numero maggiore di f.

Esercizio 3. Supponete che ci sia un *bug* nell'implementazione del sistema di firme digitali usato nel protocollo che consente a un nodo corrotto di falsificare le firme dei nodi onesti. Descrivere un attacco al protocollo che i nodi corrotti possono mettere in atto in questo caso.

Esercizio 4. Ripercorrere la dimostrazione vista a lezione del fatto che il protocollo di Dolev-Strong soddisfa *validity* e *consistency* se il numero di nodi corrotti è minore o uguale a f. (Capitolo 3 in [**?**])

Esercizio 5. Se l'insieme $F \subseteq [n]$ dei nodi corrotti è noto a priori, il problema Byzantine Broadcast (BB) diventa banale. Progettare un protocollo che prende in input l'insieme $F \subseteq [n]$ dei nodi corrotti, esegue un solo round di comunicazione, e soddisfa *validity* e *consistency* se tutti i nodi in $[n] \setminus F$ sono onesti.

2 Lower bound per Byzantine Broadcast senza PKI

Esercizio 6. A lezione abbiamo dimostrato che, in assenza di PKI non esistono protocolli per il problema Byzantine Broadcast che soddisfano *validity* e *consistency* per tre nodi, se almeno uno dei tre è corrotto.

Generalizzare la dimostrazione al caso di un numero di nodi arbitrario: dato un n qualunque, in assenza di PKI nessun protocollo per Byzantine Broadcast può soddisfare *validity* e *consistency*, se almeno $n/3$ dei nodi sono corrotti.

Esercizio 7. Considerate il sistema $\mathcal{G}$ e l'esperimento concettuale $\mathcal{H}$ in Figura 1.

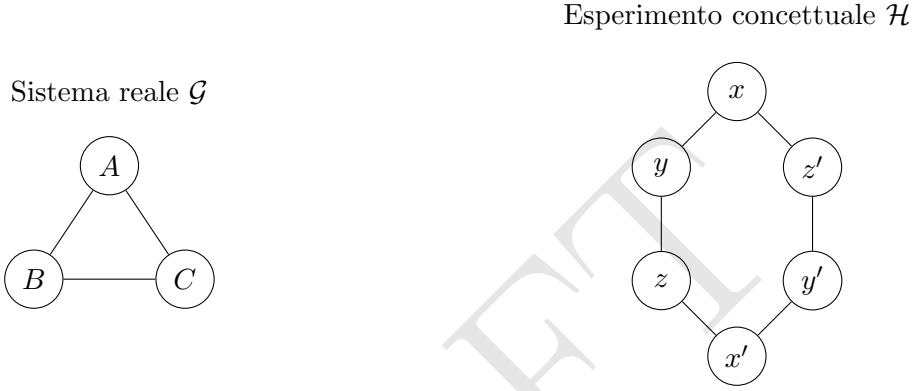

Figura 1: Il sistema reale $\mathcal{G}$ e l'esperimento concettuale $\mathcal{H}$

1. Eseguire il protocollo di Dolev-Strong con $f = 1$ nel sistema $\mathcal{G}$, quando tutti i nodi sono onesti, A è il nodo sorgente e riceve in input 1.

2. Simulare il protocollo di Dolev-Strong con $f = 1$ nell'esperimento concettuale $\mathcal{H}$ dove x e x' sono copie esatte[1] di A e ricevono in input 1, y e y' sono copie esatte di B e z e z' sono copie esatte di C (tutti i nodi sono onesti e seguono il protocollo).

3. Osservare che le esecuzioni dei due punti precedenti sono equivalenti (A, B e C non possono distinguere se sono nel sistema $\mathcal{G}$ o nell'esperimento concettuale $\mathcal{H}$).

4. Simulare il protocollo di Dolev-Strong con $f = 1$ nell'esperimento concettuale $\mathcal{H}$ in cui x e x' sono copie esatte di A, ma alla copia di A in x viene dato in input 1 mentre alla copia di A in x' viene dato in input 0; y e y' sono copie esatte di B e z e z' sono copie esatte di C (tutti i nodi sono onesti e seguono il protocollo). Quali sono gli output dei sei nodi? In che punto la dimostrazione dell'esercizio precedente non funziona nel caso del protocollo di Dolev-Strong? Perché?

3 Randomized Byzantine Broadcast senza PKI

Sia $H : \mathbb{N} \cup \{0\} \to [n]$ una funzione tale che $H(0) = 1$ e per ogni $t \in \mathbb{N}$ si comporta come un *random oracle*. A lezione abbiamo dimostrato che il protocollo seguente per Byzantine Broadcast soddisfa *Validity* con probabilità 1 e soddisfa *Consistency* con probabilità almeno $1 - (2/3)^{k-1}$.

[1] Ossia entrambe hanno la stessa chiave privata

Algorithm 2 Synchronous Randomized Byzantine Broadcast

1: ROUND 0. Ogni nodo u inizializza una variabile $\texttt{sb}_u = \perp$.
La sorgente (il nodo 1) riceve in input b e inizializza $\texttt{sb}_1 = b$.
2: PER OGNI ITERAZIONE $t = 0, \ldots, k-1$:
3: ROUND $3t$. Si seleziona il *leader* ℓ_t dell'iterazione utilizzando il random oracle $\ell_t = H(t)$
4: Se $\texttt{sb}_{\ell_t} \neq \perp$ allora ℓ_t invia a tutti $\texttt{sb}_{\ell_t}$;
5: Altrimenti ℓ_t sceglie un bit $\{0, 1\}$ u.a.r. e lo invia a tutti.
6: ROUND $3t + 1$. Ogni nodo u:
7: Se $\texttt{sb}_u \neq \perp$ allora u invia a tutti $\texttt{sb}_u$;
8: Altrimenti u invia a tutti il bit ricevuto nel round $3t$ dal leader ℓ_t (se non ha ricevuto nulla da ℓ_t o ha ricevuto entrambi i bit, invia 0 o 1 arbitrariamente)
9: ROUND $3t + 2$. Ogni nodo u:
10: Se c'è un bit $\hat{b}$ che u ha ricevuto nel round $3t + 1$ da almeno $2n/3$ nodi distinti allora u imposta $\texttt{sb}_u = \hat{b}$;
11: Altrimenti imposta $\texttt{sb}_u = \perp$.
12: ROUND $3k$. Ogni nodo u:
13: OUTPUT $\texttt{sb}_u$

Esercizio 8. Si consideri il protocollo in Algorithm 2. Dato $\delta > 0$, quanto deve essere grande il numero di iterazioni $k = k(\delta)$ affinché la probabilità che l'algoritmo soddifi la condizione di *consistency* sia almeno $1 - \delta$? Quanto deve essere grande k se scegliamo $\delta = 1/n$? Confrontare il numero di round di questo protocollo con quello del protocollo di Dolev-Strong.

Esercizio 9. Alle linee 10-11 il protocollo stabilisce che ogni nodo u deve decidere come impostare il bit $\texttt{sb}_u$ a seconda che abbia ricevuto o no almeno $2n/3$ "voti" per uno specifico bit $\hat{b}$ nel round precedente.

1. Mostrare un attacco che i nodi corrotti potrebbero mettere in atto se la decisione fosse presa in funzione di un numero di voti $h < 2n/3$;
2. Mostrare un attacco che i nodi corrotti potrebbero mettere in atto se la decisione fosse presa in funzione di un numero di voti $h > 2n/3$.

Esercizio 10. Cosa succederebbe se non imponessimo che $H(0) = 1$ (ossia che il leader ℓ_0 dell'iterazione $t = 0$ è il nodo sorgente)?

4 Byzantine Agreement

Nel problema *Byzantine Agreement (BA)* abbiamo n nodi di cui f sono corrotti. A ogni nodo i viene affidato in input un messaggio b_i. Vogliamo progettare un protocollo, che termini in un tempo finito, in cui ogni nodo onesto i dia in output un valore y_i tale che

- **Consistency**: Se i e j sono due nodi onesti, allora $y_i = y_j$;
- **Validity**: Se tutti i nodi onesti hanno in input lo stesso bit b, allora $y_i = b$ per ogni nodo onesto i.

Esercizio 11. Osservare che il problema Byzantine Agreement può essere risolto solo se il numero di nodi corrotti è $f < n/2$: Mostrare che dato un qualunque protocollo Π, se ci sono $f \geqslant n/2$ nodi corrotti questi possono sempre fare in modo che venga violata la condizione di *validity*.

Esercizio 12. Modificando opportunamente il protocollo di Dolev-Strong per il problema Byzantine Broadcast, mostrare che nel modello sincrono con PKI setup esiste un protocollo per Byzantine Agreement che soddisfa consistency e validity qualunque sia il numero di nodi corrotti $f < n/2$.

Esercizio 13. Modificando opportunamente la dimostrazione di impossibilità del problema Byzantine Broadcast con $n/3$ o più nodi corrotti, mostrare che senza PKI setup non esiste nessun protocollo per Byzantine Agreement che soddisfa consistency e validity in presenza di $n/3$ o più nodi corrotti.

5 Modello asincrono

Si ricordi che un *protocollo* nel modello asincrono è *event driven*, cioè specifica cosa fa un nodo nel momento in cui riceve un messaggio: il nodo può eseguire un numero arbitrario di operazioni locali, cambiare *stato* (lo *stato* di un nodo è costituito dai valori delle sue variabili locali, che possono dipendere dall'input e da tutti i messaggi ricevuti). Una configurazione C è una coppia $C = (S, M)$ dove S rappresenta lo stato di ogni nodo e M è l'insieme di messaggi non ancora consegnati, la *message pool*. Un messaggio m è una coppia $m = (p, x)$ dove p è il nodo destinatario del messaggio e x è il contenuto del messaggio.

Se Π è un protocollo nel modello asincrono, data una configurazione $C = (S, M)$ e un messaggio $m = (p, x)$, indichiamo con $m(C)$ la configurazione $C' = (S', M')$ che si ottiene in base al protocollo quando il messaggio m viene consegnato, ossia l'insieme S' è quello che si ottiene da S aggiornando lo stato del nodo p dopo l'elaborazione delle informazioni contenute in x e M' è la *message pool* che si ottiene da M togliendo il messaggio appena consegnato m e aggiungendo tutti i messaggi inviati da p a seguito della ricezione del messaggio m.

Esercizio 14. Sia Π un protocollo deterministico nel modello asincrono, sia $C = (S, M)$ una configurazione e siano $m_1 = (p_1, x_1)$ e $m_2 = (p_1, x_2)$ due messaggi in M. Osservare che se $p_1 \neq p_2$ allora

- m_2 sta nella *message pool* della configurazione $m_1(C)$ e m_1 sta nella *message pool* della configurazione $m_2(C)$;
- $m_2(m_1(C)) = m_1(m_2(C))$.

Uno *schedule* σ è una sequenza ordinata di messaggi $\sigma = \big((p_1, x_1), \ldots, (p_k, x_k)\big)$. Diciamo che uno schedul σ è *applicabile* a una configurazione C se $m_1 \in M$ e se per ogni $i = 2, \ldots, k$, il messaggio m_i appartiene alla *message pool* della configurazione $\sigma_{[1:i-1]}(C)$, dove con $\sigma_{[1:i-1]}$ intendiamo lo *schedule* formato dai primi $i-1$ messaggi dello *schedule* σ. Dato un protocollo Π, una configurazione C e uno schedule σ, indichiamo con $\sigma(C)$ la configurazione che si ottiene partendo da C in base al protocollo Π dopo la consegna di tutti i messaggi in σ (nell'ordine stabilito dalla sequenza).

Esercizio 15. Sia Π un protocollo deterministico nel modello asincrono, sia $C = (S, M)$ una configurazione e siano $\sigma_1 = \big((p_1, x_1), \ldots, (p_k, x_k)\big)$ e $\sigma_2 = \big((q_1, y_1), \ldots, (q_h, y_h)\big)$ due *schedule* applicabili a C. Osservare che se gli insiemi di nodi destinatari nei due schedule sono disgiunti, $\{p_1, \ldots, p_k\} \cap \{q_1, \ldots, q_h\} = \emptyset$, allora

- σ_2 è applicabile a $\sigma_1(C)$ e σ_1 è applicabile a $\sigma_2(C)$;
- $\sigma_2(\sigma_1(C)) = \sigma_1(\sigma_2(C))$.

Considerare il protocollo seguente.

Algorithm 3 Async BA protocol: Ogni nodo $u \in [n]$ esegue le istruzioni seguenti

Riceve in INPUT un bit $b \in \{0,1\}$.
$t = 1$
Invia $m = (b, t)$ a tutti
`decided` = FALSE
while `not decided` **do**
    when Arriva un messaggio $\hat{m} = (\hat{b}, h) \in \{0,1\} \times \mathbb{N}$ da un nodo $v \in [n]$
        if Hai già ricevuto un messaggio $(-, h)$ dal nodo v **then**
            Ignora $\hat{m}$
        else
            Aggiungi $\hat{m}$ all'insieme M_h, se esiste, altrimenti crea un insieme $M_h = \{\hat{m}\}$
        if $|M_t| \geqslant n - f$ **then**
            $v_0 = |\{(\hat{b}, t) \in M_t \,:\, \hat{b} = 0\}|$
            $v_1 = |\{(\hat{b}, t) \in M_t \,:\, \hat{b} = 1\}|$
            `decided` $= (\max\{v_0, v_1\} \geqslant n/2 + 3f + 1)$
            if $v_0 \geqslant n/2 + f + 1$ **then**
                $y = 0$
            else if $v_1 \geqslant n/2 + f + 1$ **then**
                $y = 1$
            else
                Scegli $y \in \{0,1\}$ u.a.r.
            $t = t + 1$
            Invia $m = (y, t)$ a tutti
OUTPUT y

Esercizio 16. Dimostrare che, se esistono un $t \in \mathbb{N}$ e un $b \in \{0,1\}$ tali che tutti i nodi onesti inviano (b, t), allora tutti i nodi onesti danno in output b.

Esercizio 17. Sia $t \in \mathbb{N}$. Dimostrare che se un nodo onesto u imposta la sua variabile $y_u(t)$ in modo deterministico, allora per ogni altro nodo onesto v che imposta la sua variabile $y_v(t)$ in modo deterministico deve essere $y_v(t) = y_u(t)$.

Esercizio 18. Osservare che dall'esercizio precedente segue anche che se due nodi onesti u e v impostano $\texttt{decided}_u(t) = \texttt{decided}_v(t) = \text{TRUE}$ allora $y_u(t) = y_v(t)$.

Esercizio 19. Siano $t \in \mathbb{N}$ e $b \in \{0,1\}$. Se un nodo onesto u imposta $\texttt{decided}_u(t) = \text{TRUE}$ e $y_u(t) = b$, allora ogni nodo onesto v invia il messaggio $(b, t+1)$.

Esercizio 20. Per ogni $t \in \mathbb{N}$ tale che `decided` = FALSE per tutti i nodi onesti, esiste un $b \in \{0,1\}$ tale che la probabilità che tutti i nodi onesti inviino $(b, t+1)$ è almeno 2^{-n}.

6 Funzioni Hash Crittografiche e Firme Digitali

Con il programma seguente (o con qualunque altra implementazione della funzione hash crittografica sha256) potete verificare che l'hash espresso in esadecimale della stringa *Francesco 16114492071* inizia con una sequenza di 9 zeri.

```python
from hashlib import sha256

nonce = '16114492071'
text = 'Francesco ' + nonce

print(sha256(text.encode('utf8')).hexdigest())
```

Esercizio 21. Scrivere un programma che trovi una stringa `<nonce>` tale che l'hash della stringa `<nome> <nonce>`, per il vostro `<nome>` inizi con una sequenza di 9 zeri. Stimare il *running time* del programma ed eseguirlo.

Le due linee seguenti sono un estratto di un file `/etc/shadow` contenente gli hash delle password degli utenti di un sistema GNU/Linux

```
satoshi:$y$j9T$2wTxPAjXwsoqXJc2eiYbb1$iERNSCCzND12k9zqBms.CqAC7CtzaGQhJnao/53Jjh2:
nakamoto:$y$j9T$RrPRoX82jNdhTLDHuVmhY1$UIzs5jXCaEiB8lgXyBbQ8uwiOYoulfguzeTl2mPUx41:
```

Dalla pagina di manuale shadow vediamo che in ogni riga i campi sono separati da ":" (nel caso in questione sono omessi tutti i campi esclusi i primi due). Il primo campo è lo *username* dell'utente. Il secondo campo è a sua volta diviso in tre parti separate dal simbolo $: la prima indica lo schema utilizzato per ottenere l'hash della password, in questo caso `y` indica che lo schema utilizzato è yescrypt. Le altre tre parti indicano rispettivamente i parametri passati allo schema, il *salt* e l'hash ottenuto.

Nonostante questo sistema di memorizzazione degli hash delle password sia molto sicuro, gli utenti possono sempre scegliere password "deboli". Nell'esempio in questione infatti uno dei due utenti ha scelto una password adeguatamente complessa, l'altro ha scelto una password molto debole.

Esercizio 22. Usando un opportuno software per il *password cracking* (per esempio, John the Ripper) determinare quale dei due utenti ha una password debole.

Esercizio 23. Il codice Python seguente[2]

Algorithm 4 Hashed RSA

```python
from Crypto.PublicKey import RSA
from hashlib import sha256

keyPair = RSA.generate(bits = 1024)

msg = b'Benvenuti a Principles of Cryptocurrency Design - AA 25/26!'
msg_hash = int.from_bytes(sha256(msg).digest(), byteorder = 'big')
msg_sig = pow(msg_hash, keyPair.d, keyPair.n)

print("Firma: ", hex(msg_sig))
```

ha scritto a video questa stringa:

```
Firma: 0x36ef68d9d5b6930e2cd9d2435ad171e4682828cb621b20daf836811bbca47f228539db3
02a66bf567e12240bac78e0a6699c230854a83dddac392287212608c5597f04d05dce7c7a61d7143b
c96f644ecb9d732dbd01f86e0009f7a64946fe7e38634b0db41673e8354f5895b8b103d8d5449155b
1a6daef2bfc67f559e7315e
```

[2]La libreria `hashlib` è built-in Python. Per `Crypto` usare la libreria `pycryptodome` `https://pypi.org/project/pycryptodome/`

Scrivere un programma per verificare che si tratta di una firma valida del messaggio *Benvenuti a Principles of Cryptocurrency Design - AA 25/26!* rispetto alla chiave pubblica $\texttt{pk} = (n, e)$ dove

```
n = 0xb9922e17467d351c454bbbe1a83ee5e6da8c5e650857c1cf9b980bbfd295eadd9b2aa09a97b
88d409820b7a6380a4c952b617b755a2eaef6b2aced5f8914610baf343edb9e5be3cb15d11c5f3684
27becd5969a8d1ab9d527ba5edad5d870600ea2b447cdfd45f227cba7076ede23087c4e7c581e6693
f0a67531250fa69d205
```

```
e = 0x10001
```

Esercizio 24. Supponiamo che, invece di firmare l'*hash* del messaggio (linee 7 e 8 in Algorithm 4), avessimo firmato direttamente il messaggio:

Algorithm 5 Textbook RSA

```
1. from Crypto.PublicKey import RSA
2.
3. keyPair = RSA.generate(bits = 1024)
4.
5. msg = b'Benvenuti a Principles of Cryptocurrency Design - AA 25/26!'
6. msg_sig = pow(int.from_bytes(msg, byteorder='big'), keyPair.d, keyPair.n)
7.
8. print("Firma: ", hex(msg_sig))
```

1. Scrivere un programma per verificare che la firma seguente, generata con lo schema in Algorithm 5,

   ```
   Firma: 0x96770d30c7f5370a6ffc1472e3638f8c58a86eae91aa675a17a9b2c924ebdaaefd
   d3ae56417b2ec0c88157427ed4dba55c3f926708c27d561ef4a06edd280061912d6f25b98e15
   82195da77547f5834b9b424541e6f6ca13313e0d1571cecb29e8d7518f9d3ce15865358b8c6b
   4b9947aa370f9140d25f3fd387457c3e5fc01a
   ```

   è una firma valida del messaggio *Benvenuti a Principles of Cryptocurrency Design - AA 25/26!* rispetto alla stessa chiave pubblica dell'esercizio precedente.

2. Mostrare che lo schema di firma digitale in Algorithm 5 non è sicuro, trovando una coppia di numeri interi (`fake_msg`, `fake_sig`) tale che `fake_sig` risulti una firma valida per `fake_msg` relativamente alla chiave pubblica del punto precedente.[3]

3. Notare il motivo per cui l'attacco del punto precedente invece non è attuabile sullo schema di firma digitale in Algorithm 4.

Esercizio 25. Scaricate un software che implementi lo standard OpenPGP e generate una vostra coppia di chiavi. Inviatemi poi una mail a `pasquale@mat.uniroma2.it` cifrandola con la mia chiave pubblica (che si trova sulla mia pagina web) e firmandola con la vostra chiave. Nella mail indicate nome, cognome e numero di matricola e allegate la vostra chiave pubblica.

7 Il test di primalità di Fermat

Esercizio 26. Si consideri il seguente *test di primalità*

1. Osservare che se l'algoritmo restituisce False, siamo sicuri che il numero in input non è primo (perché?), mentre se l'algoritmo restituisce True possiamo solo dire che il numero n in input ha *passato il test di Fermat*;

[3] Hint: Partire da una `fake_sig` arbitraria e trovare un `fake_msg` che ha `fake_sig` come firma.

Algorithm 6 Fermat primality test

```
1. from random import randint
2.
3. def Fermat_test(n, t = 10):
4.     for _ in range(t):
5.         a = randint(2, n-2)
6.         if pow(a, n-1, n) != 1:
7.             return False
8.     return True
```

2. Sia $n \in \mathbb{N}$ e indichiamo con W_n l'insieme

$$W_n = \left\{a \in \{1, \ldots, n-1\} \,:\, a^{n-1} \not\equiv 1 \bmod n\right\} .$$

Osservare che se n è *composto* (ossia, non è primo), allora la probabilità che n passi il test di Fermat è $(1 - |W_n|/n)^t$.

3. Sia n composto. Dimostrare che se esiste un $a \in \{1, \ldots, n-1\}$ tale che $\gcd(a, n) = 1$ e $a^{n-1} \not\equiv 1 \bmod n$ allora $|W_n| \geqslant n/2$.

(Hint: Sia a tale che $\gcd(a, n) = 1$ e $a^{n-1} \not\equiv 1 \bmod n$. Osservare che se $b \in [n] \setminus W_n$ allora $ab \bmod n \in W_n$ e che la funzione $f : [n] \setminus W_n \to W_n$, che associa a ogni $b \in [n] \setminus W_n$, $f(b) = ab \bmod n$ è iniettiva)

Esercizio 27. Un numero di Carmichael è un numero composto n, tale che $a^{n-1} \equiv 1$ per tutti gli $a \in \{1, \ldots, n-1\}$ tali che $\gcd(a, n) = 1$. Per i numeri di Carmichael perciò non possiamo dire che $|W_n| \geqslant n/2$.

- Scrivere un programma che prenda in input n e calcoli $|W_n|$;
- Calcolare $|W_n|$ per tutti i numeri minori di 10000, notando in particolare il valore di $|W_n|$ quando n è uno dei primi sette numeri di Carmichael: `561, 1105, 1729, 2465, 2821, 6601, 8911`.