Algoritmi di Crittografia (2023/24)

Notebook 2

In [ ]:
```
from IPython.display import HTML
HTML('<style>{}</style>'.format(open('/home/mauro/.jupyter/custom/custom.
```

Latex definitions

Il Santo Graal della crittografia asimmetrica

- La mera possibilità di "pensare" ad una crittografia asimmetrica si regge sull'esistenza di funzioni che siano facili da calcolare ma difficili da invertire, dette per questo funzioni a "senso unico" (one-way)
- Osservazione 1 Facile o difficile sono nozioni poco precise in generale. Nel contesto computazionale esse si traducono in principi effettivi mediante il criterio di efficienza polinomiale
- Secondo tale criterio un problema è facile se per esso esiste un algoritmo di costo polinomiale (o semplicemente polinomiale) nella dimensione dell'input, in un modello di calcolo appropriato
- Problemi per cui non esistono o non sono noti algoritmi polinomiali sono detti rispettivamente intrattabili o presumibilmente intrattabili

Problemi NP-completi

- Una classe di problemi presumibilmente intrattabili che riveste una straordinaria importanza teorica e pratica è quella dei cosiddetti problemi NP-completi
- Per un problema NP-completo non è nota l'esistenza di algoritmi polinomiali e la congettura è che non esistano. Si tratta della cosiddetta questione aperta P vs NP, che è probabilmente la più importante in Informatica e una delle più importanti della Matematica

Funzioni one-way

- L'esistenza di funzioni one-way implica P$\neq$NP ed è quindi improbabile che, in tempi brevi, si possa giungere all'individuazione di una funzione dimostrabilmente one-way
- Osservazione 2 Una funzione f che fosse "dimostrabilmente" one-way sarebbe comunque difficile da invertire anche per il non malintanzionato, in particolare proprio il destinatario legittimo del messaggio.
- Un esempio famoso, di natura non matematica, illustra bene la situazione
- Alice e Bob vogliono scegliere una certa quantità di vernice di un colore segreto, noto solo a loro.
- Non possono scambiarsi l'informazione pubblicamente, perché le loro

conversazioni sono pubbliche (o servegliate).

- Alice ha un'idea, che spiega a Bob apertamente
  - Bob, scegli un colore che vuoi ma non dire a nessuno, neppure a me, che cosa hai scelto
  - Fatto? Ora mischia il colore che hai scelto in parti uguali con vernice gialla. Se per caso avevi scelto proprio il giallo come colore segreto, allora ricominciamo tutto da capo
  - No? Bene! Anche io ho scelto un colore segreto e l'ho mischiato al giallo. Ora io ti mando un litro della mia miscela e tu mi mandi un litro della tua.
  - Ricevuto? Ok. Ora mischia mezzo litro del tuo colore segreto con il litro della mia miscela. Io farò altrettanto con il mio colore segreto e la tua miscela.
  - Ecco, il prodotto finale (1 litro e mezzo di miscela ciascuno) sarà il nostro colore segreto

- Eva "vede" (e supponiamo pure che possa "campionare") due miscele con metà quantità di giallo e naturalmente sa che il colore comune in entrambe è proprio il giallo.
- Ogni combinazione delle due porterebbe ad una terza miscela con troppo giallo.
- L'unica "soluzione" (apparente) consiste nel riuscire a separare colori già mischiati!
- Come si capisce da questo esempio (trascurandone i particolari poco realistici), neanche Alice e Bob sarebbero in grado di "invertire" la funzione.
- Loro però possono aggirare il problema grazie alla disponibilità di informazione segreta

Funzioni trapdoor

- Prima di calare il prededente protocollo in un contesto matematico rigoroso, anticipiamo un secondo tipo di funzione, che risulta altrettanto appetibile in contesti crittografici.
- Si tratta delle cosiddette funzioni trapdoor.
- Queste sono funziono facili da calcolare e facile anche da invertire qualora si disponga di una qualche informazione aggiuntiva
- In mancanza di tale informazione, la funzione è invece computazionalmente difficile da invertire.
- L'esistenza sia di funzioni one-way che di funzioni trapdoor non è dimostrata. Esistono candidati ed un ragionevole livello di confidenza che si tratti di funzioni con le caratteristiche desiderate, tanto è vero che su alcune di esse poggiano i protocolli asimmetrici moderni, ma un risultato teorico a supporto non lo abbiamo.

Aritmetica modulare

- Allo scopo di introdurre il primo candidato di funzione one-way, e quindi il protocollo matematico corrispondente allo scambio di vernice colorata, è necessario aprire un'ampia parentesi sull'aritmetica modulare
- In realtà l'aritmetica modulare soggiace a tutti i protocolli che vedremo nel resto del corso e dunque una sua conoscenza non marginale è necessaria per la comprensione di detti protocolli.

Insiemi $\mathbf{Z}_n$

- Aritmetica modulare è il termine che viene utilzzato per indicare un settore della Matematica che si occupa dello studio di particolari insiemi numerici e delle operazioni definite in essi.
- Gli insiemi oggetto di studio sono sottoinsiemi finiti dei numeri interi, ciascuno definito tramite un intero positivo n, detto modulo; è dunque naturale che essi vengano denotati con la scrittura $\mathbf{Z}_n$.
$$\mathbf{Z}_n = \{0, 1, \ldots, n-1\}$$
- In Informatica troviamo alcune fra le più importanti applicazioni dell'aritmetica modulare: oltre alla crittografia, la generazione di numeri pseudo-causali, le funzioni hash, i codici correttori di errore, implementazione di strutture dati, ...
- $\mathbf{Z}_n$ può essere opportunamente visto come l'insieme dei possibili resti quando si divide un numero intero per n.

Una precisazione "tecnica"

- Nella teoria dei gruppi, $\mathbf{Z}_n$ è definito come il gruppo quoziente $\mathbf{Z}/n\mathbf{Z}$.
- $n\mathbf{Z}$ è il sottogruppo degli interi costituito dai multipli di n, ovvero dai numeri che soddisfano la relazione di equivalenza R così definita:
$$x\mathrm{R}y \text{ se e solo se } (x-y) \bmod n = 0$$
dove $\bmod$ indica proprio l'operazione di calcolo del resto.
- $\mathbf{Z}_n$ è dunque il gruppo costituito dalle classi di equivalenza rispetto a tale relazione.

Nota terminologica (e non solo)

- Se $y \bmod n = x \bmod n$, ovvero se x e y danno lo resto nella divisione per n (o ancora, se la differenza $x-y$ è divisibile per n) allora si dice anche che x è congruo a y modulo n (o che x e y sono congrui modulo n), scritto:
$$x \equiv y \pmod n$$
o anche, più semplicemente,
$$x = y \pmod n$$
- Alla luce di questa definizione, potremmo dire che qualsiasi intero x fa parte di $\mathbf{Z}_n$ nel senso che o $x \in \{0, 1, \ldots, n-1\}$, oppure x è rappresentato quell'unico $y \in \{0, 1, \ldots, n-1\}$ tale che $x \equiv y \pmod n$.

- Come vedremo più avanti, nelle operazioni modulari quel che conta sono proprio i rappresentanti ed è per questa ragione che ignoriamo quanto riportate nella precedente "precisazione tecnica".

Operazioni modulari

- Negli insiemi $\mathbf{Z}_n$ sono definite le quattro operazioni aritmetiche, anche se per la divisione è necessario che sia verificata una ben precisa condizione.
- Fissato il modulo n, inizialmente definiremo le operazioni modulari usando la simbologia op_n, dove $\mathrm{op} \in \{+, -, \times, \div\}$ (indicheremo inoltre la moltiplicazione indifferentemente con $\times$ o con $\cdot$).
- Nel seguito però, per semplicità e quando sia perfettamente chiaro che stiamo operando in $\mathbf{Z}_n$, eviteremo di inserire il pedice n.
- Lasciando da parte, per il momento, la divisione, le altre tre operazioni base sono definite nel modo seguente:

$$\begin{aligned} x +_n y &= (x+y) \bmod n \\ x -_n y &= (x-y) \bmod n \\ x \cdot_n y &= (x \cdot y) \bmod n \end{aligned}$$

- In parole semplici, ogni operazione coincide con la medesima operazione sugli interi seguita dal calcolo del resto modulo n.

Riguardo la sottrazione e il caso di dividendo negativo

- Si noti che anche la sottrazione è perfettamente definita.
- Infatti, anche se il valore $x - y$ è negativo, il resto nella divisione per n è comunque un numero in $\mathbf{Z}_n$.
- Se indichiamo con Q e R rispettivamente quoziente e resto della divisione di a per b, deve innanzitutto valere
$$a = b \cdot Q + R$$
- Tuttavia l'equazione ha infinite soluzioni: ad esempio, se $a = 7$ e $b = 2$ potremmo correttamente scrivere $7 = 3 \cdot 2 + 1$ ma anche $7 = 3 \cdot 1 + 4$ come pure $7 = 3 \cdot 3 - 2$ e così via a piacimento.
- Per rendere unici quoziente e resto ci vuole un'ulteriore condizione e questa, come è ben noto, è che $0 \leq R < b$
- La "soluzione" dell'equazione è resa unica proprio dalla condizione richiesta sul resto, ovvero $R \in \{0, 1, \ldots, b-1\}$.
- Questa condizione deve essere sempre rispettata, anche quando il dividendo è negativo
- Quindi, ad esempio, $-8 \bmod 3 = 1$ perché $-8 = 3 \cdot (-3) + 1$; ogni altro valore di Q, diverso da -3, "costringerebbe" infatti ad avere un resto al di fuori dell'insieme $\{0, 1, 2\}$.
- In particolare, vale sempre
$$-1 \bmod n = n - 1$$
perché

$$-1 = n \cdot (-1) + n - 1$$

Elemento neutro e inverso negli insiemi numerici $\mathbf{Z}$ e $\mathbf{Q}$

- Sugli interi (e sui razionali) i numeri 0 e 1 giocano un ruolo speciale.
- Lo 0 è il c.d. elemento neutro dell'addizione perché, qualsiasi sia $x \in \mathbf{Z}$, vale $x + 0 = 0 + x = x$.
- Analogamente, 1 è elemento neutro della moltiplicazione perché, qualsiasi sia $x \in \mathbf{Z}$, vale $x \cdot 1 = 1 \cdot x = x$.
- L'elemento neutro entra poi nella definizione di (elemento) inverso.
- Sugli interi, ogni numero ha un elemento inverso rispetto all'addizione (detto anche opposto).
- L'opposto di x, denotato con $-x$, è precisamente quel numero y tale che $x + y = 0$.
- L'inverso rispetto alla moltiplicazione (detto anche reciproco) non esiste ovviamente per lo 0 ma, sugli interi, non esiste se non per 1 e -1 (che sono i reciproci di se stessi).
- L'inverso moltiplicativo esiste invece (tranne sempre che per lo 0) per ogni numero razionale.

Elemento neutro e inverso in $\mathbf{Z}_n$

- Per come sono definite le operazioni in $\mathbf{Z}_n$, è banale che 0 e 1 siano ancora, rispettivamente, l'elemento neutro di addizione e moltiplicazione.
- L'opposto di $x \in \mathbf{Z}_n$ (cioè l'inverso additivo) si trova facilmente, applicando la definizione: cerchiamo $z \in \mathbf{Z}_n$ tale che $x +_n z = 0$ da cui z=n-x.
- L'opposto si può ancora scrivere come $-x$, ricordando anche quanto già detto a proposito di quoziente e resto quando il dividendo è negativo.
- Per l'inverso moltiplicativo vedremo a breve le condizioni di esistenza e il metodo di calcolo.

Aritmetica modulare in Python

- Definiamo una funzione zmod che "restituisce" una classe che rappresenta un insieme $\mathbf{Z}_n$ (dove n è l'unico parametro della funzione).
- La classe "supporta" le operazioni modulari (moltiplicazione esclusa per il momento)
- È sufficiente che uno dei due operandi sia un elemento di $\mathbf{Z}_n$ affinché il risultato sia un elemento di $\mathbf{Z}_n$ (anche se l'altro operando è un intero arbitrario).

In [ ]:
```
# Per facilitare la lettura (e dunque la comprensione) della classe,
# i metodi sono definiti prima, esternamente alla classe stessa
# Ogni funzione effettua l'operazione specifica, che però restituisce
# un numero intero. La successima conversione (cast) restituisce un ogget
# il cui tipo è quello del primo operando

def addm(self,other):
```

```
        'Addizione modulare'
        return self.__class__(int.__add__(self,other)%self.mod)
    def subm(self,other):
        'Sottrazione modulare'
        return self.__class__(int.__sub__(self,other)%self.mod)
    def mulm(self,other):
        'Moltiplicazione modulare'
        return self.__class__(int.__mul__(self,other)%self.mod)
    def umin(self):
        return self.__class__(int.__neg__(self))
    # Metodo new per zn. Oltre al controllo iniziale, forza il numero nel ran
    def newm(cls,x):
        assert type(x)==int,"Supporta solo numeri interi"
        return int.__new__(cls,x%cls.mod)
```

In [ ]:

```
# L'inserimento in una funzione, consente la definizione "parametrica" di
# con il valore del modulo determinato a tempo di esecuzione. La definizi
# __radd__, __rsub__ e __rmul__ fa sì che venga applicata l'operazione mo
# quando l'elemento di zn è il secondo operando (e il primo è un intero a
def zmod(n):
    return type('zn',(int,),{'mod':n,\
                             '__new__':newm,\
                             '__add__':addm,\
                             '__radd__':addm,\
                             '__sub__':subm,\
                             '__rsub__':lambda x,y: x.__class__.__neg__(x
                             '__neg__':umin,\
                             '__mul__':mulm,\
                             '__rmul__':mulm})
```

- Possiamo ora creare una specifica classe, istanziarla ed eseguire qualche operazione

In [ ]:

```
z13 = zmod(13)
```

In [ ]:

```
x = z13(2)
y = z13(28)
z = z13(-11)
print(x)
print(y)
print(z)
```

In [ ]:

```
print(x+11)
print(8*y)
print(2-z)
print(x*(z-2*y))
print(-x)
```

- Nelle espressioni, se almeno uno degli operandi è un elemento di $\mathbf{Z}_n$, il risultato è un elemento di $\mathbf{Z}_n$.

In [ ]:

```
print(4*x+7)
type(2*6-5*x)
```

- Altrimenti è necessario un cast esplicito finale

In [ ]:
```
print(z13(10+3*5))
```

Inverso moltiplicativo e divisione modulare

- Negli insiemi $\mathbf{Z}_n$ la divisione non può essere definita a partire dalle corrispondenti operazioni sugli interi.
- Definiamo invece inverso moltiplicativo di un elemento $x \in \mathbf{Z}_n$ quel numero z (se esiste) t.c.
$$x \cdot_n z = (x \cdot z) \bmod n = 1$$
- L'inverso di x viene indicato (secondo solito) scrivendo x^{-1}
- Se l'inverso di un numero $x \in \mathbf{Z}_n$ esiste, allora "potremmo" anche definire $y \div_n x = y \cdot x^{-1} \bmod n$, anche se questo è raramente necessario nelle applicazioni.
- Ma quand'è che l'inverso di un numero $x \in \mathbf{Z}_n$ esiste? La risposta è semplice: esiste se e solo se $\mathrm{MCD}(x, n) = 1$, cioè x ed n sono relativamente primi. La dimostrazione (due celle sotto) è facoltativa.
- Come immediata conseguenza, se n è un numero primo, ogni $x \in \mathbf{Z}_n$, $x \neq 0$, ha inverso moltiplicativo.

Qualche semplice esempio

- In $\mathbf{Z}_{12}$ sono invertibili i numeri 1, 5, 7 e 11, e abbiamo:

$$\begin{aligned}
1^{-1} &= 1 \pmod{12} \\
5^{-1} &= 5 \pmod{12} \\
7^{-1} &= 7 \pmod{12} \\
11^{-1} &= 11 \pmod{12}
\end{aligned}$$

In altri termini, ogni numero invertibile ha per inverso se stesso!

- In $\mathbf{Z}_7$ abbiamo invece:

$$\begin{aligned}
1^{-1} &= 1 \pmod{7} \\
2^{-1} &= 4 \pmod{7} \\
3^{-1} &= 5 \pmod{7} \\
4^{-1} &= 2 \pmod{7} \\
5^{-1} &= 3 \pmod{7} \\
6^{-1} &= 6 \pmod{7}
\end{aligned}$$

Condizione necessaria e sufficiente per l'nvertibilità di x modulo n è che $\mathrm{MCD}(x, n) = 1$

- Il risultato È una immediata conseguenza della seguente Identità di Bezout (che non dimostriamo):

> Dati due numeri interi x e n, non entrambi nulli, esistono due numeri interi a e b tali che
> $$\mathrm{MCD}(x, n) = a \cdot x + b \cdot n$$
> Inoltre, $\mathrm{MCD}(x, n)$ è il più piccolo intero positivo esprimibile come combinazione lineare intera di x e n.

- Sufficienza. Se $MCD(x, n) = 1$, allora dall'identità di Bezout
$$1 = a \cdot x + b \cdot n$$
per opportuni interi a e b, ricaviamo
$$a \cdot x = 1 - b \cdot n$$
da cui
$$(a \cdot x) \bmod n = 1$$
e questo vuol proprio dire che $a \equiv x^{-1} (\bmod\, n)$.
- Necessità. Se x ammette inverso moltiplicativo y, allora $(x \cdot y) \bmod n = 1$, ovvero $x \cdot y = 1 + k \cdot n$, per un qualche valore intero k. Riordinando
$$x \cdot y - k \cdot n = 1$$
e poiché 1 è chiaramente il più piccolo intero positvo, necessariamente deve risultare $MCD(x, n) = 1$. In questo caso, il ruolo di a e b è giocato, rispettivamente, da y e $-k$.

Calcoli in $\mathbf{Z}_n$

- L'aritmetica modulare è molto "flessibile" riguardo al calcolo del risultato finale di un'espressione, nel senso precisato di seguito.
- Supponiamo di dover calcolare un'espressione $\mathcal{E}_{\mathbf{Z}_n}$ in $\mathbf{Z}_n$, cioè un'espressione in cui tutte le operazioni sono modulari, e sia $\mathcal{E}_{\mathbf{Z}}$ la corrispondente espressione in $\mathbf{Z}_n$. Vale
$$\mathcal{E}_{\mathbf{Z}_n} = \mathcal{E}_Z \bmod n$$
- In altri termini, possiamo svolgere i calcoli eseguendo ogni singola operazione (in aritmetica intera, cioè senza il calcolo del resto), ed eseguire il calcolo del resto solo sull'ultimo valore calcolato.
- Esempio
$$\begin{aligned}(5 \cdot_{11} 8) +_{11} (4 -_{11} 9) &= 7 +_{11} 6 \\ &= 2\end{aligned}$$
e, equivalentemente,
$$\begin{aligned}((5 \cdot 8) + (4 - 9)) \bmod 11 &= (40 + (-5) \bmod 11 \\ &= 35 \bmod 11 \\ &= 2\end{aligned}$$
- Vale ovviamente anche il contrario. Se dobbiamo calcolare la quantità

$\mathcal{E}_{\mathbf{Z}} \bmod n$, dove le operazioni sono eseguite in aritmetica intera, è possibile su qualsiasi valore intermedio eseguire preliminarmente una riduzione modulo n.

- Usando lo stesso esempio di prima, decidiamo di applicare il modulo al risultato della sottrazione:

$$\begin{aligned}((5 \cdot 8) + (4 - 9)) \bmod 11 &= (40 + (-5 \bmod 11)) \bmod 11 \\ &= (40 + 6) \bmod 11 \\ &= 46 \bmod 11 \\ &= 2\end{aligned}$$

- Applicare i moduli già ai risultati intermedi ha "enormi" ricadute positive sui calcoli, in quanto consente di limitare la grandezza delle quantità generate.

Un esempio di calcolo "manuale" efficiente

- Ci proponiamo di calcolare la quantità $2^{100} \bmod 29$.
- Naturalmente possiamo prima calcolare $2^{100} = 1267650600228229401496703205376$ e poi eseguire la divisione intera per 29.
- Un modo alternativo consiste nell'applicare ripetutamente le proprietà dei moduli (oltreché quelle dell'algebra "classica"):

$$\begin{aligned}2^{100} \bmod 29 &= (2^5)^{20} \bmod 29 \\ &= (2^5 \bmod 29)^{20} \bmod 29 \\ &= (32 \bmod 29)^{20} \bmod 29 \\ &= 3^{20} \bmod 29 \\ &= (3^3)^6 3^2 \bmod 29 \\ &= (3^3 \bmod 29)^6 3^2 \bmod 29 \\ &= (27 \bmod 29)^6 3^2 \bmod 29 \\ &= (-2)^6 3^2 \bmod 29 \\ &= 2^6 3^2 \bmod 29 \\ &= 2(2^5) 3^2 \bmod 29 \\ &= 2(2^5 \bmod 29) 3^2 \bmod 29 \\ &= 2 \cdot 3 \cdot 3^2 \bmod 29 \\ &= 2 \cdot 3^3 \bmod 29 \\ &= 2 \cdot 27 \bmod 29 \\ &= 2 \cdot (-2) \bmod 29 \\ &= -4 \bmod 29 \\ &= 25\end{aligned}$$

- Come si può notare, il più grande valore intermedio generato è 32, un bel risparmio in termini di memoria.
- Come vedreni, per i valori che vengono generati nel caso dei calcoli eseguiti dai protocolli crittografici, la possibilità di operare riduzioni sui valori intermedi demarca il confine fra fattibilità e non fattibilità delle operazioni.

Ancora una proprietà che useremo in seguito

Se m divide n, allora per qualsiasi $x \in \mathbf{Z}$ vale
$$(x \bmod n) \bmod m = x \bmod m$$

* Per dimostrare questa proprietà (peraltro intuitiva) è sufficiente applicare la definizione di resto.

Gli algoritmi di Euclide ed Euclide esteso

- L'algoritmo di Euclide esteso costituisce un metodo efficiente per calcolare l'inverso di un elemento $x \in \mathbf{Z}_n$, se esiste.
- Iniziamo però a descrivere l'algoritmo di Euclide, che ci permette di calcolare "solo" il MCD di due numeri interi.
- L'algoritmo (uno dei più antichi noti) è basato sulla seguente uguaglianza
$$\mathrm{MCD}(x,y) = \mathrm{MCD}(y, x \bmod y), \qquad y \neq 0$$
unitamente "condizione base" $\mathrm{MCD}(x,0) = x$, nel caso in cui y sia uguale a 0.
- Possiamo quindi formulare la seguente definizione per ricorrenza

$$\mathtt{MCD}(x,y) = \begin{cases} x & \mathtt{se}\ \ y = 0 \\ \\ \mathtt{MCD}(y, x \bmod y) & \mathtt{altrimenti} \end{cases}$$

che ha una traduzione "letterale" e immediata in codice Python.

In [ ]:
```python
def Euclid(x,y,pr=False):
    if pr:
        print(x,y)
    if y==0:
        return x
    return Euclid(y,x%y,pr)
```

In [ ]:
```python
Euclid(14,49,True)
```

In [ ]:
```python
Euclid(49,18,True)
```

Correttezza dell'algoritmo di Euclide

- Dimostriamo che valgono entrambe le diseguaglianze:
$$\mathrm{MCD}(x,y) \leq \mathrm{MCD}(y, x \bmod y)$$
e
$$\mathrm{MCD}(y, x \bmod y) \leq \mathrm{MCD}(x,y)$$
- Per questo scopo usiamo due semplici fatti:

se un numero m divide x e y allora divide una qualsiasi combinazione lineare a coefficienti interi di x e y. In formule

$$m|a \wedge m|b \Rightarrow m|(a \cdot x + b \cdot y) \quad a, b \in \mathbf{Z}$$

e

$x \bmod y$ è una combinazione lineare di x e y

$$x \bmod y = x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y$$

- $\mathrm{MCD}(x, y) \leq \mathrm{MCD}(y, x \bmod y)$.
- Poiché $\mathrm{MCD}(x, y)$ divide sia x che y, divide anche una qualsiasi combinazione lineare a coefficienti interi di x e y
- In particolare divide la combinazione $x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y$ che è esattamente $x \bmod y$.
- Dunque $\mathrm{MCD}(x, y)$ divide anche $x \bmod y$ oltre che y, cioè è un loro divisore comune. Ma poiché fra i divisori comuni a y e $x \bmod y$ il valore $\mathrm{MCD}(y, x \bmod y)$ è (per definizione) il massimo, ne consegue che esso è non minore di $\mathrm{MCD}(x, y)$.

- $\mathrm{MCD}(y, x \bmod y) \leq \mathrm{MCD}(x, y)$
- Il procedimento è lo stesso: basta far vedere che $\mathrm{MCD}(y, x \bmod y)$ è divisore comune, oltre che di y, anche di x. Se infatti questo è il caso la tesi è dimostrata perché $\mathrm{MCD}(x, y)$ è per definizione il massimo di tutti i divisori comuni.
- Ma $\mathrm{MCD}(y, x \bmod y)$ divide x perché x può essere espresso come combinazione lineare di y e $x \bmod y$ nel mdo seguente:

$$x = a \cdot y + b \cdot x \bmod y = a \cdot y + b \left(x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y \right) = \left\lfloor \frac{x}{y} \right\rfloor \cdot y + 1 \left(x - \left\lfloor \frac{x}{y} \right\rfloor$$

Algoritmo di Euclide esteso

- Possiamo modificare la ricorsione del metodo di Euclide in modo da calcolare, oltreché il MCD m di x e y, anche due valori interi tali a e b che soddisfano l'identità di Bezout.

$$m = a \cdot x + b \cdot y$$

- Il caso base, in cui $y = 0$, è immediato: possiamo infatti fissare $a = 1$ e $b = 0$

$$x = \mathrm{MCD}(x, 0) = 1 \cdot x + 0 \cdot 0$$

e la formulazione generale della ricorrenza si può esprimere nel modo seguente

$$\mathtt{EEuclid}(x,y) = \begin{cases} x, 1, 0 & \text{se } y = 0 \\ \\ m, b, a - b\left\lfloor \frac{x}{y} \right\rfloor & \text{se } m, a, b = \mathtt{EEuclid}(y, x \bmod y) \end{cases}$$

- La dimostrazione di correttezza è una semplice applicazione del principio di induzione.
- Abbiamo già esaminato il caso base: se $y = 0$ vale effettivamente $x = \mathrm{MCD}(x, y)$ ed inoltre $x = 1 \cdot x + 0 \cdot y$.
- Supponendo, per ipotesi induttiva, che la terna di numeri $m, a, b = \mathtt{EEuclid}(y, x \bmod y)$ soddisfi:
  1. $m = \mathrm{MCD}(y, x \bmod y)$
  2. $m = a \cdot y + b \cdot (x \bmod y)$ anche la terna associata a $\mathtt{EEuclid}(x, y)$ risulta corretta.
- Infatti, il valore m, come primo elemento della terna, è certamente corretto perché, come già sappiamo, $\mathrm{MCD}(x, y) = \mathrm{MCD}(y, x \bmod y)$.
- Per verificare la correttezza degli altri due valori sono sufficienti pochi passaggi a partire dall'ipotesi induttiva:

$$\begin{aligned} m &= a \cdot y + b \cdot (x \bmod y) \\ &= a \cdot y + b \cdot \left(x - \left\lfloor \frac{x}{y} \right\rfloor \cdot y \right) \\ &= b \cdot x + \left(a - b \left\lfloor \frac{x}{y} \right\rfloor \right) \cdot y \end{aligned}$$

- L'implementazione in Python è, ancora una volta, immediata.

In [ ]:
```
def extended_Euclid(x,y):
    '''Computes integer m,a,b such that m = gcd(x,y) and
       m = ax+by holds.'''
    if y==0:
        return x,1,0
    m,a,b = extended_Euclid(y,x%y)
    # Per ipotesi induttiva vale: m = a y + b (x%y)  e m = MCD(y,x mod y)
    return m,b,a-b*int(x/y)
```

In [ ]:
```
x = 49
y = 14
m,a,b = extended_Euclid(x,y)
sign = '+' if b>0 else ''
print(f"m={m},\ta={a},\tb={b},\ta*x+b*y={a}*{x}{sign}{b}*{y}={a*x+b*y}")
```

In [ ]:
```
x = 49
y = 18
m,a,b = extended_Euclid(x,y)
sign = '+' if b>0 else ''
```

```
print(f"m={m},\ta={a},\tb={b},\ta*x+b*y={a}*{x}{sign}{b}*{y}={a*x+b*y}")
```

Calcolo dell'inverso

- Come possiamo usare l'algoritmo di Euclide esteso per il calcolo dell'inverso moltiplicativo di x modulo n?
- Sia $m, \alpha, \beta = \mathtt{extended\_Euclid}(x, n)$
- Se $m > 1$ possiamo immediatamente concludere che $x^{-1} \bmod n$ non esiste.
- Altrimenti (se cioè $m = 1$), riscrivendo l'uguaglianza

  $$1 = a \cdot x + b \cdot n$$

  nel modo seguente

  $$a \cdot x = 1 - b \cdot n$$

  e riducendo entrambi i membri modulo n, otteniamo

  $$(a \cdot x) \bmod n = 1$$

  e questo vuol proprio dire che $a = x^{-1} \bmod n$.

In [ ]:
```
m,a,b = extended_Euclid(5,7)
print(a)
(a*5)%7   # a è l'inverso di 5 modulo 7
```

- Si noti però che a potrebbe essere un valore negativo. In tal caso un'ulteriore operazione modulo n restituisce l'inverso come numero propriamente di $\mathbf{Z}_n$

In [ ]:
```
m,a,b = extended_Euclid(5,31)
print(a)
print(a*5%31)  # a è comunque l'inverso
print(a%31)    # Se vogliamo l'inverso come elemento di Z_31
```

- Possiamo quindi definire una funzione *modular_inverse* che usa l'algoritmo di Euclide esteso ed esegue l'operazione modulare finale

In [ ]:
```
def modular_inverse(x,n):
    '''Computes 1/x mod n, if exists'''
    m,a,b=extended_Euclid(x,n)
    if m == 1:
        return a%n
```

In [ ]:
```
modular_inverse(5,31)
```

Teorema cinese dei resti

- Questo "teorema" dell'aritmetica modulare (ascritto al matematico cinese Sun-Tsu, intorno all'anno 100) ha importanti utilizzi in crittografia.
- Lo ritroveremo come strumento teorico in alcuni protocolli ma, almeno nel caso dell'RSA, anche come utile strumento per ridurre il tempo di calcolo nelle operazioni di cifratura.

- Ecco l'enunciato del teorema.

  Sia n un numero intero esprimibile come prodotto di $r > 1$ numeri interi fra loro relativamente primi

  $$n = n_1 \cdot n_2 \cdot \ldots \cdot n_r$$

  Allora il resto della divisione di un qualsiasi intero a per n è completamente determinato dai resti delle divisioni per $n_1 \cdot n_2 \cdot \ldots \cdot n_r$. In altri termini, esiste una corrispondenza biunivoca fra $\mathbf{Z}_n$ e il prodotto cartesiano $\mathbf{Z}_{n1} \times \mathbf{Z}_{n2} \times \ldots \times \mathbf{Z}_{nr}$.

- La dimostrazione è costruttiva.
- Dato un valore $a \in \mathbf{Z}_n$, la r-upla corrispondente si trova facilmente:
  $$a \longrightarrow (a \bmod n_1, a \bmod n_2, \ldots, a \bmod n_r)$$
- Ad esempio se $n = 4 \cdot 7 \cdot 9 = 252$ e $a = 3413$, risulta $a \longrightarrow (1, 4, 2)$.

In [ ]:
```
n1=4
n2=7
n3=9
n = n1*n2*n3
print(n1,n2,n3,n)
```

In [ ]:
```
a = 3413
a1=a%n1
a2=a%n2
a3=a%n3
print(f"{a%n} --> ({a1},{a2},{a3})")
```

- Per la trasformazione inversa, l'idea è di "trovare" coefficienti $c_i \in \{0, 1\}$ tali che la combinazione lineare:

  $$C = c_1 a_1 + c_2 a_2 + \ldots + c_r a_r$$

  verifichi la proprietà che $C \bmod n_i = a_i$, $i = 1, \ldots, r$. In tal caso, per un semplice argomento legato alla cardinalità dei due insiemi, potremo concludere che $C = a$.

- Per trovare tali coefficienti c_i si procede nel modo indicato di seguito.

  1. Per $i = 1, 2, \ldots, r$ calcoliamo le quantità $m_i = \prod_{j \neq i} n_j$, dove cioè m_i è il prodotto di tutti i moduli ad eccezione proprio dell'i-esimo n_i.
  2. Per "costruzione" $\mathrm{MCD}(m_i, n_i) = 1$, e dunque esiste l'inverso $m_i^{-1} \bmod n_i$, che possiamo calcolare utilizzando l'algoritmo di Euclide esteso.
  3. I valori m_i e i loro inversi consentono ora di definire i valori c_i con le caratteristiche richieste:

  $$c_i = m_i \cdot \left(m_i^{-1} \bmod n_i\right)$$

- Risulta infatti, per $j \neq i$,

$$c_i \bmod n_j = 0$$

perché c_i è multiplo di n_j

- E risulta invece

$$c_i \bmod n_i = \left(m_i \cdot \left(m_i^{-1} \bmod n_i\right)\right) \bmod n_i = \left(m_i \cdot m_i^{-1}\right) \bmod n_i = 1 \bmod n_i =$$

- Se dunque poniamo $C = \sum_{i=1}^{r} c_i a_i$, risulta

$$C \bmod n_i = a_i, \quad i = 1, \ldots, r$$

- Dunque C ha gli stessi resti di a, modulo ciascuno degli n_i.

- Poiché la cardinalità del prodotto cartesiano $\mathbf{Z}_{n1} \times \mathbf{Z}_{n2} \times \ldots \times \mathbf{Z}_{nr}$ è la stessa di $\mathbf{Z}_n$ deve necessariamente risultare $C = a$. In caso contrario ci dovrebbe essere almeno un elemento di $\mathbf{Z}_n$ cui non corrisponde nessuna r-upla in $\mathbf{Z}_{n1} \times \mathbf{Z}_{n2} \times \ldots \times \mathbf{Z}_{nr}$, il che è chiaramente falso.

In [ ]:
```
# Moduli e loro prodotto
n1 = 27
n2 = 40
n3 = 11
n = n1*n2*n3
print(f"n={n}\tn1={n1}\tn2={n2}\tn3={n3}")
# Coefficienti per la trasformazione inversa
m1 = n2*n3
m2 = n1*n3
m3 = n1*n2
invm1=modular_inverse(m1,n1)
invm2=modular_inverse(m2,n2)
invm3=modular_inverse(m3,n3)
c1 = m1*invm1
c2 = m2*invm2
c3 = m3*invm3
print(f"c1={c1}\tc2={c2}\tc3={c3}")
```

- Si noti che tutto ciò che abbiamo fatto sinora è rubricabile come "pre-computazione". I calcoli dipendono cioè solo dal modulo n e dai divisori coprimi n_i.
- Se dobbiamo eseguire molti calcoli in $\mathbf{Z}_n$, le quantità pre-determinate ovviamente non cambiano
- Supponiamo, ad esempio, di voler calcolare la seguente funzione per diversi valori del parametro:

$$f(x) = (5x^2 - 12x + 1) \bmod n$$

- Dato a, invece di calcolare $f(a)$, calcoliamo $f_{a_i} = f(a_i) \bmod n_1$, $i = 1, \ldots r$ e poi "ricostruiamo" $f(a)$ grazie al teorema cinese del resto.

In [ ]:
```
a = 5191
a1 = a%n1
a2 = a%n2
a3 = a%n3
```

```
print(f"a={a}\t\ta1={a1}\ta2={a2}\ta3={a3}")
fa = (5*a**2-12*a+1)%n
fa1 = (5*a1**2-12*a1+1)%n1
fa2 = (5*a2**2-12*a2+1)%n2
fa3 = (5*a3**2-12*a3+1)%n3
print(f"fa={fa}\tfa1={fa1}\tfa2={fa2}\tfa3={fa3}")
C = (c1*fa1+c2*fa2+c3*fa3)%n
print(f"C={C}")
```

Gruppi e gruppi ciclici $\mathbf{Z}_n$ e $\mathbf{Z}_n^*$

- L'operazione di somma modulare in $\mathbf{Z}_n$ gode (fra le altre) delle seguenti quattro le proprietà
  1. Chiusura: la somma di due elementi di $\mathbf{Z}_n$ è ancora un elemento di $\mathbf{Z}_n$
  2. Associatività: $(a +_n b) +_n c = a +_n (b +_n c)$
  3. Esistenza el. neutro: chiaramente lo 0
  4. Esistenza inverso: l'inverso (o opposto) di x è $n - x$
- L'insieme $\mathbf{Z}_n$, rispetto all'operazione di somma modulare, è dunque un *gruppo*

- Che cosa possiamo dire invece della "struttura" di $\mathbf{Z}_n$ rispetto alla moltiplicazione?
- Sappiamo che se n è composto non tutti gli elementi di $\mathbf{Z}_n$ hanno inverso moltiplicativo.
- Se dunque n è composto, l'insieme $\mathbf{Z}_n$ non è un gruppo rispetto alla moltplicazione modulare.
- Sappiamo però che l'inverso è posseduto dagli elementi $x \in \mathbf{Z}_n$ che sono coprimi con n
- ... e l'insieme di tali elementi forma effettivamente un gruppo rispetto alla moltiplicazione modulare.
- Si usa indicare tale insieme con la scrittura $\mathbf{Z}_n^*$.
- Ad esempio, i numeri di $\mathbf{Z}_{12}$ che sono coprimi con 12 sono 1, 5, 7 e 11. Allora $\mathbf{Z}_{12}^* = \{1, 5, 7, 11\}$ è un gruppo moltiplicativo.
- Come esercizio, si verifichi dapprima che $\mathbf{Z}_{12}^*$ è un gruppo rispetto alla moltiplicazione modulate e poi si provi a dimostrare il fatto in generale per $\mathbf{Z}_n^*$.

Se n è primo $\mathbf{Z}_n$ è un campo

- Ovviamente, se n è primo, tutti gli elementi di $\mathbf{Z}_n$, ad eccezione dello zero, sono coprimi con n e dunque hanno inverso.
- Poiché vale (banalmente) anche la proprietà distributiva della moltiplicazione rispetto all'addizione, se n è primo allora $\mathbf{Z}_n$ è un *campo finito* (o *campo di Galois*)

Ordine di un elemento in un gruppo

- Consideriamo un generico elemento x di un gruppo G.
- Si chiama ordine di x il minimo numero di volte che possiamo

sommare/moltiplicare x all'elemento neutro prima di ottenere come risultato nuovamente l'elemento neutro.

- Più difficile da spiegare che da capire con semplici esempi.
- Nel gruppo additivo $\mathbf{Z}_{12}$ l'operazione è l'addizione e l'elemento neutro è lo zero. In tal caso (tralasciando l'indicazione del modulo per semplicità):
  1. l'ordine di 2 è 6 perché $(((((0+2)+2)+2)+2)+2)+2=0$ (cioè applicando 6 volte l'addizione modulare si ottiene 0) e con un minore di addizione non si ottiene 0.
  2. L'ordine di 7 è 12; infatti

$$\begin{aligned} 0+7&=7\\ 7+7&=2\\ 2+7&=9\\ 9+7&=4\\ 4+7&=11\\ 11+7&=6\\ 6+7&=1\\ 1+7&=8\\ 8+7&=3\\ 3+7&=10\\ 10+7&=5\\ 5+7&=0 \end{aligned}$$

- Nel gruppo moltiplicativo $\mathbf{Z}_7^*$ (elemento neutro 1):
  1. l'ordine di 2 è 3 perché $((1\cdot 2)\cdot 2)\cdot 2=1$ e con un numero minore di moltiplicazioni non si ottiene 1;
  2. l'ordine di 3 è 6 perché

$$\begin{aligned} 1\cdot 3&=3\\ 3\cdot 3&=2\\ 2\cdot 3&=6\\ 6\cdot 3&=4\\ 4\cdot 3&=5\\ 5\cdot 3&=1 \end{aligned}$$

Gruppi ciclici e generatori

- Il numero di elementi che fanno parte di un gruppo G è poi detto ordine di G stesso, e viene indicato con $\mathrm{ord}(G)$: per $\mathbf{Z}_{12}$ l'ordine è 12 mentre per $\mathbf{Z}_7^*$ l'ordine è 6.
- Un gruppo si dice ciclico se esiste un elemento g il cui ordine coincide con l'ordine del gruppo.

$$\mathrm{ord}(g)=\mathrm{ord}(G)$$

- Tale elemento viene detto generatore del gruppo (proprio perché con operazioni successive si ottengono tutti gli elementi del gruppo) o anche radice primitiva
- In base agli esempi precedenti possiamo quindi dire che 7 è un generatore del

gruppo additivo $\mathbf{Z}_{12}$ e che 3 è un generatore del gruppo moltiplicativo $\mathbf{Z}_7^*$
- I gruppi additivi $\mathbf{Z}_n$ e moltiplicativi $\mathbf{Z}_n^*$, con n primo, sono gruppi ciclici.

- Consideriamo ora, per fissare le idee, gruppi moltiplicativi
- Se g è un generatore, vale chiaramente

$$\mathbf{Z}_p^* = \{g^i \bmod p | i = 1, \dots, p-1\}$$

- Per un elemento h di ordine $s < p-1$ (cioè un elemento che non è un generatore) è comunque interessante considerare l'insieme:

$$H = \{h^i \bmod p | i = 1, \dots, s\}$$

- È facile dimostrare che H è un sottogruppo ciclico di $\mathbf{Z}_p^*$.
- Sottogruppo vuol dire che è un sottoinsieme di G e che è un gruppo!
- Va da se che ogni sottogruppo, per essere tale, deve comunque incudere 1.
- h è chiaramente il generatore del sottogruppo e, se $x = h^i \bmod p$ (per un qualche $i \leq s$), allora, banalmente $x^{-1} \bmod p = h^{s-i} \bmod p$, proprio perché

$$x \cdot x^{-1} \bmod p = x^i \cdot x^{s-i} \bmod p = x^s \bmod p = 1$$

In [ ]:
```
from ACLIB.utils import Euclid, modexp
```

In [ ]:
```
def ord(h,p):
    if h == 1:
        return 1
    v = h
    o = 2
    while (v:=(v*h)%p) and v!=1:
        o+=1
    return o
```

In [ ]:
```
ord(1,11)
```

In [ ]:
```
def subgroup(h,p):
    return [(h**i)%p for i in range(1,ord(h,p)+1)]
```

In [ ]:
```
subgroup(2,11)
```

Teorema di Lagrange e teorema fondamentale dei gruppi ciclici

> *Teorema di Lagrange* Qualsiasi sottogruppo di un gruppo di ordine k ha necessariamente ordine che è un divisore di k.

- Per i gruppi ciclici esiste un risultato più forte.

> *Teorema fondamentale dei gruppi ciclici* Per ogni divisore k dell'ordine del gruppo esiste uno ed un solo sottogruppo di ordine k

- Esempio: I sottogruppi di $\mathbf{Z}_{19}^*$ sono elencati di seguito, con l'aiuto di un semplice programma Python. Si tenga presente che $\mathbf{Z}_{19}^*$ ha ordine 18 e che i divisori non banali di 18 sono 2, 3, 6 e 9.

In [ ]:
```
p = 19
subgroups = sorted([(h,subgroup(h,p)) for h in range(2,p)], key=lambda pa
for H in subgroups:
    print(f"Generatore: {H[0]}\tOrdine: {len(H[1])}\tSottogruppo: {sorted
```

Primi sicuri (safe prime)

- Per ragioni che saranno chiare non appena esamineremo il primo protocollo di scambio di chiavi, numeri primi molto importanti sono quelli della forma: $p = 2q + 1$, dove anche q è primo.
- Esempi di tali numeri, detti *primi sicuri*, sono: 11, 23, 59 e 83.
- Per un tale numero, il teorema di Lagrange garantisce che i sottogruppi non banali possono avere solo 2 o q elementi.
- Sappiamo però anche di più (perché abbiamo a che fare con gruppi ciclici) e cioè che esiste un solo sottogruppo di ordine 2 e un solo sottogruppo di ordine q.
- Vediamo un esempio esaustivo nel caso di $p = 23$

In [ ]:
```
p = 23
subgroups = sorted([(h,subgroup(h,p)) for h in range(2,p)], key=lambda pa
for H in subgroups:
    print(f"Generatore: {H[0]}\tOrdine: {len(H[1])}\tSottogruppo: {sorted
```

- Dall'esempio di vede anche che, se tralasciamo il sottogruppo generato da 22 (che naturalmente è $\{1, -1 \bmod 23\} = \{1, 22\}$), ci sono esattamente metà elementi che generano il gruppo intero e metà che generano il sottogruppo di ordine q.
- Questa è una situazione generale e possiamo anche affermare che, per i primi sicuri (ed escludendo naturalmente 1 e -1), è facile stabilire da che cosa sia formato il sottogruppo di q elementi.
- Esso è composto precisamente da quegli elementi che sono l'equivalente (in aritmetica modulare) dei quadrati perfetti e che qui vengono detti residui quadratici (modulo p)

- Per continuare con l'esempio di $\mathbf{Z}_{23}^*$, il numero 3 è un residuo quadratico perché $(7 \cdot 7) \bmod 23 = 3$
- Se il modulo è primo si può poi dimostrare che ogni quadrato perfetto ha esattamente due radici (come nel caso reale) che sono una l'opposta dell'altra.
- La seconda "radice quadrata" di 3 in $\mathbf{Z}_{23}^*$ è dunque $23 - 7 = 16$

In [ ]:
```
(16**2)%23
```

- Nel caso di primi sicuri, e se escludiamo 1 e -1, il teorema di Lagrange garantisce che un qualsiasi elemento genera o l'intero gruppo o il sottogruppo dei residui quadratici.
- E sempre grazie al teorema di Lagrange risulta facile capire, per un elemento x scelto a case (e diverso da 1 e -1), in quale caso ci si trovi.
- Infatti, sapendo che l'ordine di x può solo essere q o $p-1$, è sufficiente calcolare (nel modo che vedremo) $x^q \bmod p$.
- Se il risultato è 1, allora l'ordine di x è q e dunque x genera il sottogruppo, altrimenti genera l'intero gruppo $\mathbf{Z}_p^*$

- Nella libreria OPENSSL si adotta un approccia ancora più estremo, in cui il generatre è fisso
- Infatti, quando richiesto di generare un safe prime, OPENSSL restituisce un valore p tale che, per default, 2 è sempre un generatore del gruppo $\mathbf{Z}_p^*$. L'utente ha solo la possibilità di richiedere un diverso generatore, ma solo 3 o 5.

In [ ]:
```
%%sh
openssl dhparam -text 2048
```

In [ ]:
```
%%sh
ls -ltr
```

In [ ]:
```
%%sh
openssl dhparam -outform PEM -out dhcert.pem 2048
openssl dhparam -inform PEM -in dhcert.pem -check -text
```

Un funzione candidata one-way: l'esponenziale modulare

- Una computazione fondamentale in crittografia asimmetrica riguarda il calcolo della funzione esponenziale modulare.
- Per essere one-way la funzione deve essere innanzitutto (computazionalmente) facile da calcolare.
- Dobbiamo quindi capire se sia possibile calcolare in maniera efficiente quantità del tipo $a^b \bmod n$.

- Il problema è delicato per via della grandezza dei numeri in gioco.
- In applicazioni crittografiche, sia a che b (oltreché n), possono essere dell'ordine del migliaio di bit ma, giusto per fornire un'idea, supponiamo che la lunghezza sia limitata a 128 bit.
- In tal caso a^b è un numero che può arrivare ad avere una quantità di bit pari a

$$\log_2 (2^{128})^{2^{128}} = \log_2 2^{128 \cdot 2^{128}} = 128 \cdot 2^{128} = 2^{135}$$

  che corrispondono a più di 10^{40} cifre decimali. Evidentemente non esiste alcuna possibilità di manipolare numeri di tale grandezza!
- Poiché però il risultato finale è un numero di "soli" 128 bit, si capisce che

bisognerà utilizzare le proprietà dell'aritmetica modulare (che abbiamo visto) per limitare la grandezza dei risultati intermedi.

- Questo per quanto riguarda lo spazio.

- Anche il tempo però è un problema se si adotta l'algoritmo "banale" che calcola a^b (sia pure con le opportune riduzioni modulo n dei risultati intermedi) come prodotto iterato $\underbrace{a \times a \times \ldots \times a}_{b-1 \text{ prodotti modulari}}$.
- In tal caso, infatti, si eseguono esattamente $b-1$ moltiplicazioni modulari, una quantità improponibile se b ha anche "solo" 128 bit.

Prodotto modulare

- L'esponenziale è basata comunque sul prodotto e dunque introduciamo prima un algoritmo per il calcolo della quantità $a \cdot b \bmod n$
- Tutto ciò non è inutile anche perché la tecnica è la stessa che adotteremo per l'esponenziale.
- Questo non dovrebbe sorprendere per il semplice motivo che, come l'esponenziale è un'iterazione della moltiplicazione, il prodotto è un'iterazione della somma.
- Possiamo infatti calcolare $a \cdot b$ come $\underbrace{a + a + \ldots + a}_{b-1 \text{ addizioni modulari}}$, cosa che però vogliamo evitare.
- Presentiamo l'algoritmo utilizzando come esempio il calcolo di $z = a \cdot b$, con $b = 26$.
  1. Scriviamo b in base 2,
  $$26_{10} = 11010_2 = 2^4 + 2^3 + 2^1$$
  2. Utilizzando ora la proprietà distributiva abbiamo
  $$a \cdot b = a \cdot 26 = a \cdot (2^4 + 2^3 + 2^1) = a \cdot 2^4 + a \cdot 2^3 + a \cdot 2^1$$
  3. Tutte le quantità del tipo $a \cdot 2^k$, sia quelle utilizzate nel prodotto (ovvero quelle con $k = 1, 3, 4$) sia quelle che "non servono" ($k = 0, 2$), possono essere calcolate partendo da $a_0 = a \cdot 2^0 = a$ e raddoppiando il valore ad ogni iterazione: $a_{k+1} = 2 \cdot a_k$
  4. Ad inizio poniamo $z = 0$ e ad ogni iterazione sommiamo a_k a z se e solo se a_k "serve", cioè se il k-esimo bit di b è 1.

- Nel codice Python che segue: (1) le moltiplicazioni per 2 sono eseguite mediante shift, (2) dopo ogni operazione viene eseguita la riduzione modulo n, e (3) i bit di b vengono rivelati mediante successive divisioni per 2, calcolate ancora mediante shift.

In [ ]:
```
def modprod(a,b,n):
    '''Computes the product ab mod n using additions and shifts only.
```

```
       For the computation of remainders x mod n, we note that x is alway
       less than 2n (except possibly for the input parameters a and b).
       Hence integer division is not required.
    '''
    def easymod(x):
        'x<2n, always'
        nonlocal n
        if x>=n:
            return x-n
        return x
    if b==0:
        return 0
    a = a%n                  # If a>n replace a with a mod n
    b = b%n                  # Idem
    z = 0                    # z accumulates the partial sums a2^k
    while b>0:
      if b&1:                # Use the current value of a if needed
        z = easymod(z+a)
      a = easymod(a<<1)      # prepare for the next step
      b = b >> 1             # shift b
    return z
```

In [ ]:
```
modprod(2**253-1,2*135-5,7101)==((2**253-1)*(2*135-5))%7101
```

L'algoritmo di esponenziazione

- Utilizza la stessa tecnica impiegata per il calcolo del prodotto.
- Supponiamo di voler calcolare $a^b \bmod n$, e supponiamo ancora $b = 26_{10} = 11010_2$.
- Abbiamo quindi $a^{26} = a^{2^4+2^3+2^1} = a^{2^4}a^{2^3}a^{2^1}$.
- Allora, similmente al modo con cui si procede nel caso del prodotto, possiamo determinare tutte le potenze $a_k = a^{2^k}$ partendo da $a_0 = a^{2^0} = a$ e calcolando ad ogni iterazione

$$a_{k+1} = a_k^2 = \left(a^{2^k}\right)^2 = a^{2\cdot 2^k} = a^{2^{k+1}}$$

- Nel caso del prodotto, i risultati vengono "accumulati" sommando i contributi a_k ad una variabile inizializzata al valore 0.
- Nel caso dell'esponenziale, i risultati vengono invece accumulati moltiplicando i contributi a_k ad una variabile inizializzata a 1.
- Il codice Python è essenzialmente "identico" a quello del prodotto

In [ ]:
```
def modexp(a,b,n):
    'Efficiently computes a^b mod n.'
    a = a%n      # If a>n, replaces it with a mod n (of course, this canno
    p = 1        # p accumulates the partial products a^(2^k)
    while b>0:
        if b&1: # If the rightmost bit of b is 1, the current value of a
            p = modprod(p,a,n)
        a = modprod(a,a,n)   # prepare for the next step
        b = b >> 1           # shift b
    return p
```

In [ ]:
```
modexp(1138,323,101)==(1138**323)%101
```

L'inversa dell'esponenziale modulare: il logaritmo discreto

- Come nel caso dei numeri reali, se in un insieme $\mathbf{Z}_n^*$ vale $b^e = x \bmod n$, l'esponente e è detto logaritmo (discreto) in base b di x, e si scrive $e = \log_b x \bmod n$.
- Da quanto abbiamo visto a proposito dei gruppi ciclici, se p è primo esiste sempre una "base" g per cui il logaritmo discreto è definito per ogni elemento di $\mathbf{Z}_p^*$. Tale base deve essere un generatore del gruppo.
- Si parla poi di logaritmo anche nel caso di gruppi additivi. Il valore di k tale che $k \cdot g = x \bmod n$ è cioè anch'esso detto logaritmo in base g di x.
- Quest'ultima definizione ci "farà comodo" quando parleremo di crittografia su curve ellittiche.

Radici primitive (generatori)

- Per un gruppo $\mathbf{Z}_p^*$ arbitrario, non sono però noti algoritmi polinomiali per decidere se un elemento $a \in \mathbf{Z}_p^*$ è una radice primitiva né tantomeno algoritmi polinomiali per trovarne una.
- Questo anche se il loro numero è relativamente elevato.
- Più precisamente, si può dimostrare che il numero di radici primitive di $\mathbf{Z}_n^*$ è pari al valore di una funzione molto nota in teoria dei numeri, e precisamente la funzione (toziente) di Eulero, denotata con $\phi(n)$.

$$\phi(n) = |\{i | 1 \leq i < n \;\text{ e }\; \mathrm{MCD}(i,n) = 1\}|$$

- A parole, $\phi(n)$ è il numero di interi minori di n e coprimi con n (ivi incluso 1).
- La seguente funzione (particolarmente inefficiente...) calcola $\phi(n)$

In [ ]:
```
def phi(n):
    return [x for x in range(1,n) if Euclid(x,n)==1]
```

In [ ]:
```
phi(22)
```

- Incontreremo ancora, a proposito del protocollo RSA, la funzione ϕ.
- Ci interessa in particolare il caso in cui n è il prodotto di due numeri primi $n = p \cdot q$. In tal è di facile verifica che vale $\phi(n) = (p-1)(q-1)$
- Infatti, fra i numeri minori n, 1 ogni p numeri non è coprimo con n. Si tratta dei $q-1$ numeri $p, 2p, \ldots, (q-1)p$.
- Analogamente, 1 ogni q numeri non è coprimo con n e si tratta dei $p-1$ numeri $q, 2q, \ldots, (p-1)q$.
- In queste due sequenze non ci sono elementi comuni proprio perché p e q sono coprimi fra loro.
- Tenendo conto del numero 1, abbiamo dunque

$$\begin{aligned}\phi(n) &= n - (p-1) - (q-1) - 1 \\ &= p \cdot q - p - (q-1) \\ &= p(q-1) - (q-1) \\ &= (p-1)(q-1)\end{aligned}$$

- Questo ragionamento può essere comunque esteso ad ogni possibile valore di n, utilizzando la scomposizione in primi

Il calcolo del logaritmo è (apparentemente) difficile

- Non sono noti algoritmi polinomiali per il calcolo del logaritmo discreto.
- Tutti gli algoritmi noti hanno worst-case esponenziale nella lunghezza dei numeri.
- Essi non fanno cioè "sostanzialmente" meglio dell'approccio a forza bruta, che prova tutti gli esponenti fino a trovare quello corretto (tempo atteso $O(2^{n-1})$ per numeri di n bit).
- In questo corso ne vedremo solo uno, perché realmente molto semplice, il cui costo atteso è $O(2^{n/2})$, ma che è meglio caratterizzato dal prodotto spazio$\times$ tempo impiegato, come vedremo subito.

Algoritmo baby-steps giant-steps BSGS)

- Si tratta sempre di un algoritmo brute-force che però consente di stabilire, entro certi limiti, se privilegiare lo spazio oppure il tempo
- L'algoritmo necessita solo della conoscenza del modulo p, oltre che della base g del logaritmo da calcolare (la radice primitiva), e procede calcolando due successioni di numeri, una con incrementi "piccoli", l'altra con incrementi "grandi", da cui anche il nome dell'algoritmo (cosa che però è più evidente nel caso di gruppo additivo)
- Un termine che compare in entrambe le successioni (si parla anche di collisione) consente di calcolare il logaritmo.

Pseudocodice

1. Scegli due interi r e s tali che $r \cdot s \geq p$. Si noti che ogni numero t in $\mathbf{Z}_p$ può essere espresso come $t = j + i \cdot r$, $j = 0, 1, \ldots, r-1$ e $i = 0, 1, \ldots, (s-1)$.

$$\begin{array}{rcl} (0,0) & \rightarrow & 0 \\ (0,1) & \rightarrow & 1 \\ & \cdots & \\ (0,r-1) & \rightarrow & r-1 \\ (1,0) & \rightarrow & r \\ (1,1) & \rightarrow & r+1 \\ & \cdots & \\ (s-1,r-1) & \rightarrow & (s-1)\cdot r + r - 1 = s \cdot r - 1 \end{array}$$

2. Calcola le due successioni (per semplicità omettiamo l'operazione di calcolo del resto)
    - $1, g, g^2, \ldots, g^{r-1}$ baby steps
    - $x, xg^{-r}, xg^{-2r}, \ldots, xg^{-(s-1)r}$ giant steps
3. Se, per qualche valore di i e j, risulta $g^i = xg^{-jr}$ e, riordinando:

$$g^{i+jr} = x, \qquad \text{da cui} \qquad i + jr = \log_g x$$

- La correttezza dipende dal fatto che, come già sottolineato, ogni numero in $\mathbf{Z}_p$ può essere espresso come somma $i + jr$ se i e j variano nel range indicato.

Implementazione

- L'algoritmo viene implementato memorizzando i termini della successione baby-staps in una lookup table T, che viene poi consultata con i termini della successione giant-steps, alla ricerca di una collisione
- Il tempo di esecuzione dell'algoritmo è $O(r+s)$, sempre naturalmente che T supporti le ricerche in tempo $O(1)$, e poiché $r \cdot s \geq p$ il minimo si ottiene ponendo $r = s = \lceil \sqrt{p} \rceil$
- Il consumo di spazio è $O(r)$ e dunque si può ridurre arbitrariamente, al prezzo di aumentare il numero di termini della successione giant-steps e dunque il tempo di esecuzione

Qualche esperimento

- Definiamo una funzione, safe group, che restituisce un safe prime p e un suo generatore g

In [ ]:
```
import Crypto.Util.number as utils
```

In [ ]:
```
utils.isPrime(11)
```

In [ ]:
```
from os import urandom
def safeprime(p):
    '''Checks whether p=2*q+1 and q are primes'''
    if not p&1:
        return False
    q = p-1>>1
    return q&1 and utils.isPrime(p) and utils.isPrime(q)
def safegroup(b):
    '''Returns a pair p,g where p is a 8*b bit safe prime
       and g is either 2 or 5
    '''
    while True:
        while (q:=int.from_bytes(urandom(b),'big')|1) and not safeprime((
            pass
        p = (q<<1)+1
        if p%8==3:
            return p,2
        elif p%5 != 4:
            return p,5
```

In [ ]:
```
p,g = safegroup(5)
print(p,g)
```

In [ ]:
```
# Implementazione come dizionario
class lookuptabled(dict):
    def __new__(cls,*args):
        self = super().__new__(cls)
        return self
    def __init__(self,k=None,v=None):
        if k is not None:
            self[k]=v
    def __getitem__(self,k):
        v = super().get(k,None)
        return v
```

In [ ]:
```
# Implementazione come lista ordinata di coppie (chiave,valore)
from bisect import bisect_left,insort_left
class lookuptablel:
    def __init__(self,k=None,v=None):
        self.L = []
        if k is not None:
            self.L.append((k,v))
    def __getitem__(self,k):
        pos = bisect_left(self.L,(k,-1))
        if pos<len(self.L):
            p = self.L[pos]
            if p[0]==k:
                return p[1]
        return None
    def __setitem__(self,k,v):
        insort_left(self.L,(k,v))
    def __repr__(self):
        return str(self.L)
```

In [ ]:
```
from ACLIB.utils import intsqrt, modprod, modexp, modular_inverse
def BSGS(g,x,n,lookuptable=lookuptabled,rmax=10**8):
    '''Computes the discrete logarithm a=log_g x mod n,
       where g is a primitive root of the multiplicative group Z*_n.
       Uses the Baby-steps Giant-steps algorithm with rmax default
       maximum number of baby steps.
    '''
    r = min(intsqrt(n)+1,rmax); s = (n // r) + 1
    # Start baby-steps
    bs = lookuptable(1,0)     # First pair in the baby-steps lookup table
    v = g                     # v = g^1 mod n = g (assuming g<n)
    for i in range(1,r):
        bs[v]=i
        v = modprod(v,g,n)
    # Start giant-steps
    gr = modexp(modular_inverse(g,n),r,n)
    R = 0
    xr = x
    while R<s*r:
        e = bs[xr]
        if e is not None:
            return (e+R)%n
        xr = modprod(xr,gr,n)
        R += r
    return None
```

In [ ]:
```
from time import time
```

In [ ]:
```
# Calcoliamo logaritmo discreto di 19 modulo p in base g
start = time()
e = BSGS(g,19,p,lookuptable=lookuptablel)
print(f"{time()-start:.2f}")
print(e)
```

In [ ]:
```
modexp(g,e,p)
```

In [ ]:
```
start = time()
e = BSGS(g,19,p)
print(f"{time()-start:.2f}")
print(e)
```

In [ ]:
```
modexp(g,e,p)
```