Nome: Alberto Biliotti
Matricola: 7109894

Data security and privacy
Seconda parte esercizi

---

1 Esercizi di programmazione

1.1 Esercizio 3.1

Ho implementato gli algoritmi descritti di seguito nel file "algoritmiCrittografiaRSA.py" allegato

a. L'algoritmo esteso di Euclide è utile per ricavare il massimo comun divisore (MCD) tra due numeri a ed n e, se questo risulta essere pari ad 1, gli inversi dei due numeri in modulo pari all'altro numero in ingresso.
Tale algoritmo consiste di un unico ciclo nel quale, in ogni iterazione, non faccio altro che ricavarmi i resti delle divisioni dei resti trovati alle due iterazioni precedenti, con i primi due resti con i i quali inizio le iterazioni che sono posti uguali ad a ed n. In pratica quindi, per ricavare il MCD ad ogni iterazione calcolerò:

$$r_i = r_{i-1} - q_i * r_{i-2},\ r_0 = a, r_1 = n \tag{1}$$

Dove q_i è la divisione intera tra r_{i-1} e r_{i-2}. Tale ciclo si interromperà quando il resto diventerà pari a zero, e il MCD corrisponderà al precedente valore del resto. Contemporaneamente a ciò, calcolerò iterativamente i coefficienti di Bezout in maniera simile a quanto visto per il MCD, solo che invece di partire da valori pari ad a ed n, partirò con valori pari a 0 ed 1 per il coefficiente associato ad a mentre 1 e 0 per quello associato ad n, ossia, detti x ed y i coefficienti di Bezout associati rispettivamente ad a ed n:

$$x_i = x_{i-1} - q_i * x_{i-2},\ x_0 = 1, x_1 = 0 \tag{2}$$

$$y_i = y_{i-1} - q_i * y_{i-2},\ y_0 = 0, y_1 = 1 \tag{3}$$

Una volta ricavati i coefficienti di Bezout, che saranno pari ai valori x ed y ricavati nella penultima esecuzione del ciclo, volendo ricavare l'inverso di a modulo n e l'inverso di n modulo a dovrò applicare ad x il modulo rispetto ad n e ad y il modulo rispetto ad a, ottenendo così i valori desiderati che andrò a restituire. Tale funzione è implementata con il nome: "algoritmoEstesoEuclide".

b. Per quanto riguarda l'algoritmo di esponenziazione veloce, nella mia implementazione ho leggermente modificato l'algoritmo "classico" visto a lezione per renderlo più semplice, evitando di dovermi ricavare e salvare direttamente la rappresentazione binaria di un numero. Infatti, invece di partire dal bit più significativo dell'esponente, ciclo partendo da quello meno significativo, procedendo in maniera opposta a quanto visto nell'algoritmo tradizionale, dovendo quindi eseguire prima il controllo sul bit in analisi, e in caso risulti pari ad uno, moltiplicare modularmente la variabile di accumulo, che conterrà alla fine delle iterazioni il risultato, per la base elevata a due a sua volta elevato al numero dell'iterazione meno uno, e successivamente elevare modularmente al quadrato la base. Facendo così ottengo lo stesso risultato che otterrei con l'algoritmo tradizionale ma senza dover ricavare direttamente la rappresentazione binaria dell'esponente, dovendo solamente eseguire un operazione di modulo due sull'esponente per ricavare il bit ed una successiva divisione intera sempre per due dell'esponente fino a che non ottengo un valore pari a 0. Tale funzione è implementata con il nome: "esponenziazioneVeloce"

c. Il test di Miller-Rabin, utile per testare se un intero n dispari è composto o meno attraverso un algoritmo di tipo Monte Carlo, restituisce vero quando il numero in analisi è effettivamente composto. Però se il test restituisce falso, concludendo quindi che il numero non risulta essere composto, esiste una probabilità di errore che si può dimostrare essere inferiore ad 1/4 (dimostreremo in un altro esercizio che tale soglia di errore è pari al massimo ad 1/2).
Tale algoritmo prende in ingresso, oltre all'intero positivo da testare, anche un altro intero positivo, minore del numero da testare, chiamato x (nell'implementazione tale numero è chiamato invece test) coprimo con n. Il test consiste, in primo luogo, nel dividere n-1 per due fino a che non ottengo un numero dispari m, dopo aver diviso per due per r volte. Fatto ciò mi ricavo il termine iniziale x_0 come:

$$x_0 = x^m \; mod \, n \tag{4}$$

Se tale valore è pari a più o meno uno termino direttamente restituendo falso, altrimenti procedo in un ciclo che calcola ripetutamente per un massimo di r volte la seguente formula:

$$x_i = x_{i-1}^2 \; mod \, n \tag{5}$$

Tale ciclo verrà interrotto sia in caso trovi un valore di x_i pari ad 1, restituendo vero, in quanto avrei trovato una radice quadrata non banale dell'unità, che i numeri primi non possono possedere per la proprietà dei resti quadratici, sia nel caso trovi un valore pari a meno uno restituendo falso. Se invece non incontro nessun valore pari a meno uno in nessuna iterazione eccetto l'ultima, restituisco vero in quanto violerei il teorema di Fermat, poiché avrei trovato che x è un numero intero, coprimo con n per ipotesi, tale che $x^{n-1} \not\equiv_n 1$.
Tale funzione è stata implementata nel metodo "testMillerRabin".

d. A questo punto posso implementare anche un metodo per generare numeri primi, dato un ordine di grandezza (rispetto alla base due) desiderato n ed una soglia di tolleranza, che consiste nello scegliere casualmente un intero nell'intervallo compreso tra 2^{n-1} e 2^n che sarà il nostro candidato numero primo su cui andrò ad eseguire il test di Miller-Rabin definito al punto precedente con un numero di test compreso nell'intervallo tra due ed il candidato meno uno, ricordando che se il test restituisce falso, ovvero il numero risulta potenzialmente primo, tale risultato ha una possibilità di errore al più pari ad 1/4. Ripeterò quindi il test con interi di test differenti finché la probabilità di errore non decresce sotto la soglia indicata in input, considerando le probabilità di errore indipendenti per ogni test, restituendo il candidato. Se invece il test restituisce valore vero, avrò la certezza che il candidato è in realta composto e procederò a ripetere il procedimento scegliendo casualmente un nuovo numero intero candidato primo nell'intervallo desiderato. Tale funzione è implementata nel metodo "generaNumPrimo".

e. A questo punto ho tutti gli ingredienti per poter implementare uno schema di cifratura RSA. In primo luogo, è stato implementato un metodo, chiamato "generaFattoriModulo", che, dato in input l'ordine di grandezza in base due desiderato per il modulo RSA, genera due fattori primi casuali p e q, il cui prodotto sarà proprio il modulo dello schema RSA desiderato.
Posso quindi implementare un metodo per generare le chiavi pubbliche e private per la cifratura e decifratura, calcolando prima il valore della funzione $\phi(n) = (p-1)(q-1)$ (equivalente alla cardinalità di $\mathbb{Z}_n^*$, ovvero il numero di interi minori di n coprimi con esso). Successivamente scelgo arbitrariamente una chiave pubblica, di partenza chiamata e, nel mio caso 3, controllando che sia coprima con $\phi(n)$, altrimenti incremento di uno il valore di e fino a che non trovo un valore coprimo con $\phi(n)$. Una volta stabilito il valore di e, lo inverto in modulo $\phi(n)$ ottenendo così la chiave privata della fattorizzazione. Tale metodo è stato chiamato "generaChiavi".
Ora posso quindi implementare un metodo che, dato un ordine di grandezza desiderato per il modulo RSA, mi generi i fattori p e q, il modulo e le chiavi direttamente, sfruttando i due metodi implementati in precedenza, chiamato "generaModuloChiavi".

A questo punto il metodo per la cifratura e la decifratura "classica", chiamato "crittaRSA" consiste semplicemente nello sfruttare l'esponenziazione veloce per ricavare sia il messaggio cifrato c che il plaintext m in questo modo:

$$c = E_e[m] = m^e \, mod \, n, \; m = E_d[c] = c^d \, mod \, n \tag{6}$$

. Per quanto riguarda infine il metodo di decifratura/firma digitale che sfrutta il Teorema Cinese del Resto, nel quale devo essere a conoscenza dei due fattori p e q del modulo RSA, dato m un messaggio (cifrato o meno), mi ricavo prima i due termini S_p ed S_q come:

$$S_p = m^{d \, mod \, p-1} \, mod \, p, \; S_q = m^{d \, mod \, q-1} \, mod \, q \tag{7}$$

. Per poi ricavare il risultato S come:

$$S = (q * (q^{-1} \, mod \, p) * S_p + p * (p^{-1} \, mod \, q) * S_q) \, mod \, n \tag{8}$$

. Dove l'inverso di q modulo p e l'inverso di p modulo q li ho ricavati attraverso un'unica applicazione del algoritmo esteso di Euclide. Tale ottimizzazione possiede uno speed up teorico di circa 1/4 rispetto all'algoritmo di cifratura classico.

Nel file in allegato sono implementati inoltre alcuni test per verificare la correttezza delle implementazioni, insieme a quello per controllare lo speedup che ottengo decifrando 100 messaggi generati casualmente, con un modulo precomputato dell'ordine di $2^3$000 ossia un numero con oltre 900 cifre, in cui si evince che effettivamente, se pur con risultati variabili a causa dello scheduler del sistema operativo, lo speed è pari a circa 1/4 tra l'implementazione della decryption con e senza CRT speed up (il test impiega qualche istante per terminare, vista la notevole dimensione dei numeri necessaria per evidenziare maggiormente tale speed up).

1.2 Esercizio 3.2

La soluzione di questo esercizio, implementata in Python, è contenuta nel file "decryptionExponentAttackRSA.py" in allegato.

a. Tale indice esiste in quanto il valore di x all'ultimo passo r sarà pari a uno in modulo n in quanto:

$$x_r \equiv_n x^{2^r m} = x^{e*d-1} \equiv_n x^{(e*d-1) \, mod \, \phi(n)} = x^0 = 1 \tag{9}$$

Poiché nell'algoritmo ho elevato prima x alla m per poi elevarlo al quadrato per r volte, avendo posto $2^r m = e * d - 1$. Tale risultato è possibile grazie al teorema di Eulero, che afferma che, per qualsiasi x intero positivo, se x ed n sono coprimi, ho che $x^{\phi(n)} \equiv_n 1$.
Quindi, calcolando i quadrati successivi di x^m, troverò prima o poi un valore pari ad uno in modulo n. Se il valore di x_j diverso da 1, che elevato al quadrato modulo n restituisce 1, è diverso da -1, saprò che $x_j - 1$ e $x_j + 1$ avranno ciascuno un fattore in comune distinto con il modulo che stiamo cercando di fattorizzare. Questo poiché ho trovato un valore z che è radice quadrata non banale dell'unità, ossia: $z^2 \equiv_n 1$. Tale uguaglianza modulare può essere scritta come: $(z-1)(z+1) \equiv_n 0$. Da ciò si può dedurre che $n|(z-1)(z+1)$, ossia che n divide il prodotto tra z-1 e z+1, ma non uno dei due termini singolarmente, in quanto altrimenti avrei che $z \equiv_n \pm 1$, il che è assurdo per quanto detto in precedenza.
Per questo sono certo che i due termini z-1 e z+1 abbiano un termine non banale in comune con n, che posso ricavare calcolando il MCD tra z-1 ed n e tra z+1 ed n.

b. Per implementare tale algoritmo posso procedere in maniera simile a quanto visto per il test di Miller Rabin, implementando prima il ciclo che divide per due il numero $e * d - 1$ fino a che non ottengo un valore m dispari,tale che $e * d - 1 = 2^r m$, dove r è il numero di volte che ho dovuto dividere per 2 per ottenere m. Successivamente, dentro un ciclo while potenzialmente infinito, scelgo casualmente un valore x in $\mathbb{Z}_n^*$, quindi mi ricavo il valore $x_0 = x^m$, come accade per l'algoritmo di Miller Rabin. Una volta ricavato x_0, se tale valore è diverso da ± 1, parto con un ciclo for calcolando $x_i = x_{i-1}^2$ finché non trovo un valore pari a ± 1, altrimenti scelgo un nuovo valore x. Se dentro il ciclo for trovo un indice j tale che $x_j = -1$, sceglierò un nuovo valore iniziale interrompendo il ciclo e ripeterò il procedimento con un nuovo valore casuale di x. Se invece trovo un valore $x_{j+1} = 1$ mi ricaverò uno dei due fattori di n calcolando il MCD tra n e x_j, che mi sono salvato in una variabile temporanea, per poi ricavarmi l'altro dividendo n per il fattore appena ricavato. Terrò conto inoltre di quante volte ho dovuto ricalcolare il valore di x, che è pari al numero di iterazioni dell'algoritmo, per arrivare alla fattorizzazione, restituendo tale valore insieme ad i due fattori del modulo ricavati.

Nel file allegato ho testato l'implementazione appena illustrata su 100 moduli RSA di ordine di grandezza di 2^{500} o equivalentemente 10^{150} generati casualmente con i metodi visti nell'esercizio precedente. I risultati mostrano come la mia implementazione sia anzitutto corretta, in quanto i valori generati dall'attacco sono equivalenti alla fattorizzazione desiderata, ma anche che essa è estremamente efficiente, in quanto il tempo medio di esecuzione dell'attacco è pari a circa un millisecondo, con una varianza di circa $7 * 10^{-7}$ secondi quadrati ed una deviazione standard di circa 0.0008 secondi.
Per quanto riguarda invece il numero medio di iterazioni dell'algoritmo, tale valore è pari a circa 1.5, in linea con quanto affermato nel testo dell'esercizio che poneva tale valore inferiore a 2.

2 Esercizi di approfondimento

2.1 Esercizio 2.1

a. Per dimostrare ciò è sufficiente applicare la proprietà del prodotto modulo n che afferma che il prodotto di due interi maggiori o uguali di uno modulo n è uguale al prodotto del loro modulo n, ossia:

$$(a * b) \, mod \, n = (a \, mod \, n) * (b \, mod \, n) \; \forall n \geq 1 \tag{10}$$

Sapendo inoltre, per ipotesi, che x è testimone di Fermat, ossia $x^{n-1} \equiv_n z$ con z diverso da uno, mentre y non lo è, ossia $y^{n-1} \equiv_n 1$, posso concludere che:

$$(x * y)^{n-1} \, mod \, n = (x^{n-1} \, mod \, n) * (y^{n-1} \, mod \, n) \equiv_n z * 1 = z \tag{11}$$

Ossia $(x * y) \, mod \, n$ è testimone di Fermat di n in $\mathbb{Z}_n^*$

b. Per assurdo, se poniamo xy e xy' congruenti modulo n, arriviamo ad un assurdo in quanto, ricordando che per essere testimone di Fermat di un intero positivo n un numero deve avere massimo comune divisore pari ad 1 con n, ossia $(x, n) = 1$, si ha:

$$xy \equiv_n xy' -> y \equiv_n y' \tag{12}$$

Posso dividere x da entrambe le parti poiché esso è invertibile in quanto il suo massimo comune divisore con n è pari ad uno per quanto mostrato prima. Il risultato ottenuto è assurdo in quanto y e y' per ipotesi sono due diversi non testimoni di Fermat appartenenti a $\mathbb{Z}_n^*$.

c. Se esiste almeno un testimone di Fermat x per n, allora, chiamato y un altro qualsiasi intero in $\mathbb{Z}_n^*$ non testimone di Fermat per n, avremo che il prodotto di x ed y, in modulo n, risulta ancora essere un testimone di Fermat, per la proprietà dimostrata nel punto a. Inoltre tale valore sarà unico per ciascuno dei possibili y, numeri non testimoni di Fermat appartenenti a $\mathbb{Z}_n^*$, per quanto affermato

dalla proprietà b. Di conseguenza, se in $\mathbb{Z}_n^*$ esiste almeno un testimone di Fermat, avremo che il numero totale di testimoni di Fermat è almeno pari a quello dei non testimoni di Fermat, ossia abbiamo dimostrato come ci siano almeno $\frac{|\mathbb{Z}_n^*|}{2} = \frac{\phi(n)}{2}$ testimoni di Fermat in $\mathbb{Z}_n^*$.

d. Se n è composto dispari non Carmichael, allora possiede almeno un testimone di Fermat. Per quanto visto al punto precedente, n possiede quindi almeno $\frac{|\mathbb{Z}_n^*|}{2}$ testimoni di Fermat. Siccome l'algoritmo di Fermat consiste nello scegliere casualmente un numero intero x compreso tra 1 ed n-1, se tale numero ha massimo comune divisore diverso da 1 con n non posso sbagliare in quanto so già che tale valore è un fattore di n e di conseguenza ritorno vero poichè n è sicuramente composto. Se invece, come è probabile che accada, il numero x risulta essere coprimo con n, applico il teorema di Fermat controllando il valore di $x^{n-1}\ mod\ n$. Se x è un testimone di Fermat, ossia in almeno il 50% dei casi per quanto detto in precedenza, il risultato torna essere diverso da uno e termino correttamente restituendo vero. Quindi abbiamo dimostrato come l'algoritmo di Fermat possiede probabilità di errore minore di $\frac{1}{2}$ applicato su un numero composto non Carmichael. Siccome l'algoritmo di Miller-Rabin include al suo interno anche quello di Fermat, posso concludere che anche tale algoritmo possiede una probabilità di successo pari o superiore ad $\frac{1}{2}$.

2.2 Esercizio 2.4

Le implementazioni dei metodi descritti di seguito, utili per sfruttare il commons modulus failure per ricavare il messaggio in chiaro a partire da due testi cifrati a partire dallo stesso messaggio, con lo stesso modulo ma con chiavi differenti si trovano nel file "commonModuleFailure.py" allegato.
Come anticipato l'obbiettivo è quello di ricavare il messaggio m conoscendo unicamente due messaggi cifrati c_1 e c_2 con crittografia RSA a partire da m con lo stesso modulo ma con chiavi di cifratura e_1 ed e_2 differenti, anche esse note in quanto pubbliche.
Per fare ciò anzitutto devo controllare che e_1 ed e_2 siano coprimi tra loro; fortunatamente ciò si verifica quasi sempre. Appurato che le due chiavi siano coprime tra loro è sufficiente sfruttare la formula illustrata di seguito per ricavare il messaggio in chiaro:

$$c_1^x * c_2^y \equiv_n m^{e_1 x + e_2 y} \equiv_n m^1 = m \tag{13}$$

Dove x ed y sono i coefficienti dell'identità di Bezout, che dati due interi non nulli, nel nostro caso le due chiavi pubbliche, permette di scrivere il loro massimo comun divisore come combinazione lineare di tali numeri; in questa identità i numeri di cui sto cercando di ricavare il MCD sono moltiplicati rispettivamente per i valori x ed y che sono chiamati coefficienti di Bezout.
Tuttavia, per poter sfruttare effettivamente la formula (13) devo prima implementare due funzioni importanti: una che mi permetta, dati due numeri, di ricavarmi i loro coefficienti di Bezout ed una seconda che mi permetta di esponenziare modularmente in maniera efficiente un intero anche con esponenti negativi, in quanto uno dei due coeffcienti di Bezout risulterà essere sicuramente negativo se le chiavi sono coprime.
Per quanto riguarda la prima funzione, per ricavare i coefficienti di Bezout non devo fare altro che applicare l'algoritmo esteso di Euclide, che avevamo già implementato negli esercizi di programmazione, anche se in quel caso non restituivamo i coefficienti ma direttamente gli inversi modulari dei due numeri in input, tuttavia per risolvere ciò è sufficiente eliminare l'operazione di modulo che andavo ad applicare ad i coefficienti che avevo ricavato dall'esecuzione dell'algoritmo. Tale funzione è stata implementata nel codice con il nome: "algoritmoEstesoEuclideBezout".
Invece per la seconda funzione devo anzitutto definire l'esponenziazione modulare di una base positiva, coprima con il modulo, per un esponente negativo. Tale operazione consiste nel calcolare prima l'inverso in modulo della base per poi elevare modularmente tale valore per l'esponente cambiato di segno. Di conseguenza la mia implementazione di tale funzione non farà altro che controllare se l'esponente è positivo o negativo: nel primo caso non faccio altro che sfruttare il metodo per l'esponenziazione veloce "classico", implementato anch'esso negli esercizi pratici, se invece l'esponente è negativo applicherò prima l'algortimo

esteso di Euclide per trovare il MCD tra la base ed il modulo e, in caso la base risulti invertibile (MCD uguale ad uno), l'inverso modulare della base. Una volta ricavata la base invertita sfrutterò l'algoritmo di esponenziazione modulare veloce per elevare l'inverso della base all'esponente cambiato di segno. Tale funzione è implementata nel mio codice con il nome "esponenziazioneVeloceNegativa".
Una volta implementate queste due funzioni non mi resta che implementare il common modulus failure grazie alla formula 13, controllando prima che le due chiavi siano coprime tra loro. La formula è stata quindi implementata nel metodo "commonModulusFailure".
Fatto ciò posso quindi risolvere l'esercizio ricavando m dai messaggi cifrati che conosciamo insieme alle relative chiavi, che risulterà pari a 34634564567767777777. Inoltre per dimostrare la correttezza di tale risultato posso crittografare nuovamente il messaggio trovato con le stesse chiavi ed osservare come i testi cifrati combacino con quelli da cui sono partito.