SAPIENZA
UNIVERSITÀ DI ROMA

DIPARTIMENTO DI MATEMATICA GUIDO CASTELNUOVO

Algoritmo di fattorizzazione di Lenstra

ISTITUZIONI DI ALGEBRA E GEOMETRIA

Professore:

Domenico Fiorenza

Studenti:

Umberto Gallo (1937779)

Giorgia Iannarelli (1962972)

Anno Accademico 2024/2025

Indice

1 Introduzione

Obiettivo di questa tesina è quello di presentare il codice, realizzato in *MATLAB*, che implementa l'algoritmo di Lenstra. Ideato nel 1985 dal matematico olandese Hendrik Willem Lenstra Jr., tale algoritmo riguarda la fattorizzazione di un numero intero, positivo e composto, attraverso l'utilizzo delle curve ellittiche. L'algoritmo di Lenstra lo possiamo vedere come una generalizzazione del metodo $p-1$ di Pollard, che a sua volta sfrutta il concetto di ricerca dei divisori di un numero n attraverso l'esecuzione di operazioni modulari. In particolare, l'idea per entrambi gli algoritmi, è quella di considerare una certa famiglia di gruppi (che per Pollard sono gli $(\mathbb{Z}/(p))^*$ con p tale che $p|n$, mentre per Lenstra sono le curve ellittiche) e "scommettere" che almeno uno di questi sia γ-liscio.
Il vantaggio dell'algoritmo di Lenstra, che illustreremo in dettaglio nel seguito, sta nel fatto che il numero di gruppi su cui è possibile "scommettere" è dell'ordine di n^3, dove n è il numero da fattorizzare. Quindi in questo caso abbiamo la quasi certezza che si riesca a trovare un divisore di n.
Gli algoritmi di fattorizzazione sono di grande interesse nel campo della crittografia, in particolare nelle applicazioni che fanno uso di numeri primi di grandi dimensioni, come nel caso degli scambi di chiavi (ad esempio, nel protocollo RSA) e altre tecniche di crittografia basate sulla difficoltà di fattorizzare numeri interi grandi. L'efficienza e la versatilità del metodo di Lenstra nel trovare fattori di numeri composti, lo rendono utile quando si cerca di analizzare la sicurezza di sistemi crittografici che si basano proprio sulla fattorizzazione di un certo numero n in due grandi numeri primi.
La seguente tesina si sviluppa in tre capitoli:

- Nel **primo capitolo** vi sono brevi richiami sulle curve ellittiche e qualche loro proprietà;

- Nel **secondo capitolo** è presente il corpo principale della tesina, ossia la descrizione del metodo di fattorizzazione di Lenstra e il relativo codice in *MATLAB*;

- Nel **terzo capitolo**, suddiviso in varie sezioni, sono illustrati e spiegati i diversi sottoprogrammi che intervengono nel codice principale.

2 Richiami: le curve ellittiche

2.1 Introduzione alle curve ellittiche

Le curve ellittiche nascono alla fine dell'800, a partire dall'analisi complessa. L'obiettivo era quello di cercare una funzione $f : \mathbb{C} \to \mathbb{C}$ che fosse biperiodica (ossia tale che $f(z) = f(z + \lambda)$ con $\lambda \in \Lambda$ dove Λ indica un reticolo discreto) ed olomorfa.
Dato che solo le funzioni costanti hanno questa proprietà, l'idea fu quella di non richiedere che la funzione f fosse olomorfa ovunque, ma supporre che questa ammettesse delle singolarità sui punti del reticolo.
In questo modo si pensò ad f come una funzione del tipo:

$$f : \mathbb{C} \,/\, \Lambda \to \mathbb{C}$$

dove $\mathbb{C} \,/\, \Lambda$ è un toro, dunque in particolare è compatto.

A tal proposito, Weierstrass introdusse la cosiddetta funzione $\mathcal{P}$ di Weierstrass, definita come:

$$\mathcal{P}(z) = \tfrac{1}{z^2} + \textstyle\sum_{\lambda \neq 0} \big(\tfrac{1}{(z-\lambda)^2} - \tfrac{1}{\lambda^2} \big)$$

Questa funzione ha le seguenti proprietà:

- E' olomorfa su $\mathbb{C} \setminus \Lambda$;
- $\mathcal{P}(z + \lambda) = \mathcal{P}(z)$;
- Ha un polo di ordine 2 in $z = 0$;
- $\mathcal{P}(z) = \frac{1}{z^2} + A(z)$, dove $A(z)$ si può scrivere come $A(z) = z\, a(z)$ con $a(z)$ che avrà poli nei punti non nulli di Λ;
- $\mathcal{P}(-z) = \mathcal{P}(z)$, dunque $\mathcal{P}(z)$ è pari;
- Ha derivata pari a

  $$\mathcal{P}'(z) = -\tfrac{2}{z^3} + \textstyle\sum_{\lambda \neq 0} -\tfrac{2}{(z-\lambda)^3} = -2 \sum_{\lambda \in \Lambda} \tfrac{1}{(z-\lambda)^3},$$

  che può essere riscritta come

  $$\mathcal{P}'(z) = -\tfrac{2}{z^3} + B(z),$$

  con $B(z) = z\, b(z)$, dove $b(z)$ avrà poli nei punti non nulli di Λ;
- $P'(z)$ è olomorfa su $\mathbb{C} \,/\, \Lambda$ ed è dispari.

Abbiamo allora due funzioni:

1. $\mathcal{P}(z) = \frac{1}{z^2} + z\ a(z)$
2. $\mathcal{P}'(z) = -\frac{2}{z^3} + z\ b(z)$

Calcolando $\mathcal{P}(z)^3$ e $\mathcal{P}'(z)^2$, si può osservare che

$$(\mathcal{P}'(z))^2 = 4\ \mathcal{P}(z)^3 + \alpha_1\ \mathcal{P}(z) + \alpha_2$$

da cui, dividendo per 4, si ha:

$$\left(\tfrac{\mathcal{P}'(z)}{2}\right)^2 = \mathcal{P}(z)^3 + \tilde{\alpha_1}\ \mathcal{P}(z) + \tilde{\alpha_2}$$

Ponendo $x(z) = \mathcal{P}(z)$ e $y(z) = \frac{\mathcal{P}'(z)}{2}$, concludiamo che la coppia $(x(z), y(z))$ soddisfa l'equazione

$$y^2 = x^3 + \tilde{\alpha_1}\ x + \tilde{\alpha_2}$$

che è l'equazione di una curva ellittica.

A questo punto abbiamo una mappa

$$\varphi : \mathbb{C}\ /\ \Lambda \setminus \{(0,0)\} \to \mathbb{C}^2,$$

tale che $\varphi(z) = \left(\mathcal{P}(z), \frac{\mathcal{P}'(z)}{2}\right)$ e la sua immagine è la cubica di equazione

$$y^3 = x^2 + ax + b,$$

per certi valori di a e b.

Quello che vogliamo fare ora è estendere l'applicazione φ in 0. Notiamo che:

$$\varphi(0) = \left(\mathcal{P}(0), \tfrac{\mathcal{P}'(0)}{2}\right) = (\infty, \infty)$$

Il polo in $\mathcal{P}(z)$ è di ordine 2 (poichè abbiamo $\frac{1}{z^2}$), mentre il polo in $\mathcal{P}'(z)$ è di ordine 3 (poichè abbiamo $\frac{1}{z^3}$), dunque per gesitre la presenza di questi ∞, basta passare da $\mathbb{C}^2$ ad una versione che contenga i punti all'∞.

A tal proposito consideriamo l'inclusione $\mathbb{C}^2 \subseteq \mathbb{P}^2\,\mathbb{C}$, che è data da $(x, y) \to [x : y : 1]$. Guardando la mappa

$$\varphi : \mathbb{C}\ /\ \Lambda \setminus \{(0,0)\} \to \mathbb{P}^2\,\mathbb{C},$$

posso dire che

$$\varphi(z) = \left[\mathcal{P}(z) : \tfrac{\mathcal{P}'(z)}{2} : 1\right] = \left[\tfrac{\mathcal{P}(z)}{\frac{\mathcal{P}'(z)}{2}} : 1 : \tfrac{1}{\frac{\mathcal{P}'(z)}{2}}\right]$$

dove nell'ultima uguaglianza abbiamo usato il fatto che se moltiplichiamo le coordinate per una stessa costante, il punto proiettivo non cambia.

A questo punto, allora, possiamo dire che

$$\varphi(z) = \left[\frac{\mathcal{P}(z)}{\frac{\mathcal{P}'(z)}{2}} : 1 : \frac{1}{\frac{\mathcal{P}'(z)}{2}} \right] \xrightarrow{z \to 0} [0:1:0]$$

Concludiamo che φ si estende ad un'applicazione del tipo

$$\varphi : \mathbb{C} \,/\, \Lambda \to \{\ y^2 = x^3 + ax + b \cup [0:1:0]\ \} \subseteq \mathbb{P}^2\,\mathbb{C},$$

che si può dimostrare essere un isomorfismo.

2.2 Struttura di gruppo per le curve ellittiche

Notiamo che $(\mathbb{C}, +)$ è un gruppo abeliano e $(\mathbb{C} \,/\, \Lambda, +)$ è un suo sottogruppo abeliano. Ma allora, dato che φ è un isomorfismo, anche l'insieme

$$\{\ y^2 = x^3 + ax + b \cup [0:1:0]\ \}$$

dovrà avere la struttura di gruppo abeliano. Vediamo come è fatta.

- Elemento neutro: è l'immagine di $[0]$ tramite l'isomorfismo, ossia $\varphi(0) = [0:1:0]$
- Opposto: $\varphi(-z) = \big(\mathcal{P}(-z), \frac{\mathcal{P}'(-z)}{2}\big) = \big(\mathcal{P}(z), -\frac{\mathcal{P}'(z)}{2}\big)$. Dunque l'opposto di (x, y) è $(x, -y)$.
- Operazione di somma: per capire quanto fa $\varphi(z+w)$ usiamo la seguente relazione:

$$\det \begin{pmatrix} \mathcal{P}(z) & \mathcal{P}'(z) & 1 \\ \mathcal{P}(w) & \mathcal{P}'(w) & 1 \\ \mathcal{P}(z+w) & -\mathcal{P}'(z+w) & 1 \end{pmatrix} = 0$$

  e, dividendo per 2 la seconda colonna, si ha

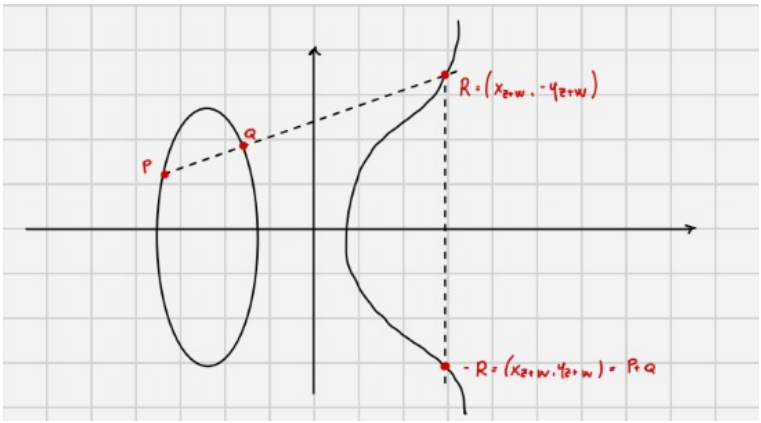
$$\det \begin{pmatrix} \mathcal{P}(z) & \frac{\mathcal{P}'(z)}{2} & 1 \\ \mathcal{P}(w) & \frac{\mathcal{P}'(w)}{2} & 1 \\ \mathcal{P}(z+w) & -\frac{\mathcal{P}'(z+w)}{2} & 1 \end{pmatrix} = 0$$

  Ma dire che

$$\det \begin{pmatrix} x_z & y_z & 1 \\ x_w & y_w & 1 \\ x_{z+w} & -y_{z+w} & 1 \end{pmatrix} = 0$$

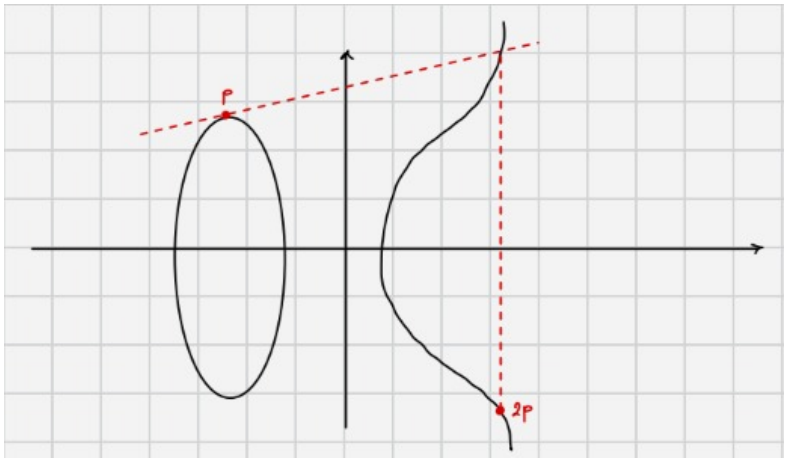
vuol dire che i punti $P = (x_z, y_z)$, $Q = (x_w, y_w)$ e $R = (x_{z+w}, -y_{z+w})$ sono allineati.

Vediamo graficamente questo cosa vuol dire:



Dal punto di vista grafico si può osservare che nella somma sulla curva ereditata dalla somma sul toro, il punto $P + Q$ è proprio il punto $-R$.

Quando P e Q coincidono, fare $P + Q = 2P$ vuol dire tracciare la tangente, come si vede nella figura sottostante.



Le formule per $P + Q$ e $2P$ sono algebriche, dunque possiamo generalizzare quanto fatto in $\mathbb{C}$ e definire questa struttura di gruppo con a e b in un campo $\mathbb{K}$ qualsiasi.

Attenzione. La curva ellittica $\mathcal{C}_{a,b}$ di equazione $y^2 = x^3 + ax + b$ ha una struttura di gruppo a patto che a e $b \in \mathbb{K}$ siano scelti in modo che la curva $\mathcal{C}_{a,b}$ sia liscia.

2.3 Come stabilire che una curva f(x,y)=0 è singolare

Data una curva di equazione $f(x, y) = 0$, si dice che è singolare se $\exists\ (x_0, y_0)$ tale che:

$$\begin{cases} f(x_0, y_0) = 0 \\ \frac{\partial f(x_0,y_0)}{\partial x} = 0 \\ \frac{\partial f(x_0,y_0)}{\partial y} = 0 \end{cases}$$

Nel nostro caso $f(x, y) = y^2 - (x^3 + ax + b)$, quindi per trovare la condizione di singolarità, sostituiamo nel sistema precedente ottenendo:

$$\begin{cases} y_0 - x_0^3 - ax_0 - b = 0 \\ -3x_0^2 - a = 0 \\ 2y_0 = 0 \end{cases}$$

A questo punto, affinché dalla terza condizione valga $y_0 = 0$, necessariamente la caratteristica del campo $\mathbb{K}$ deve essere diversa da 2. In questo modo il sistema si riduce a:

$$\begin{cases} -(x_0^3 + ax_0 + b) = 0 \\ -3x_0^2 - a = 0 \\ y_0 = 0 \end{cases}$$

Affinché il sistema sia soddisfatto, le prime due equazioni devono avere uno zero in comune, dunque il $MCD(x^3 + ax + b, 3x^2 + a)$ deve essere diverso da 1 ed in particolare deve aversi $MCD(x^3 + ax + b, 3x^2 + a) = (x - x_0)$ con x_0 zero in comune. Per poter procedere con il calcolo, supponiamo anche che $char(\mathbb{K}) \neq 3$, dato che vogliamo dividere per $3x^2$:

$$(x^3 + ax + b, 3x^2 + a) = \left(3x^2 + a, \frac{3}{2}ax + b\right) = \left(\frac{3}{2}ax + b, \frac{27b^2}{4a^2} + a\right)$$

Poiché $\frac{27b^2}{4a^2} + a$ è una costante e noi vogliamo che $MCD(x^3 + ax + b, 3x^2 + a)$ sia un polinomio di primo grado, dobbiamo imporre che $\frac{27b^2}{4a^2} + a = 0$.
Da qui ricaviamo che $\mathcal{C}_{a,b}$ è singolare se e solo se $\Delta = 27b^2 + 4a^3 = 0$.

2.4 Formule esplicite per P+Q e 2P

Consideriamo una curva ellittica di equazione $y^2 = x^3 + ax + b$, tale che sia liscia, ossia $\Delta = 27b^2 + 4a^3 \neq 0$. Abbiamo visto che in $\mathbb{C}$ ad esempio, la nostra curva ha una struttura di gruppo e abbiamo anche visto graficamente come trovare il punto $R = P+Q$. Passiamo ora a determinare delle formule esplicite per calcolare $(x_R, y_R) = (x_{P+Q}, -y_{P+Q})$:

- Consideriamo due punti P e Q appartenenti alla curva $\mathcal{C}_{a,b}$, rispettivamente di coordinate $P = (x_P, y_P)$ e $Q = (x_Q, y_Q)$

- Scriviamo l'equazione della retta passante per i punti P e Q in forma parametrica

$$r : P + t\vec{PQ} = \begin{pmatrix} x_P \\ y_P \end{pmatrix} + t \begin{pmatrix} x_Q - x_P \\ y_Q - y_P \end{pmatrix} = \begin{pmatrix} x_P + t(x_Q - x_P) \\ y_P + t(y_Q - y_P) \end{pmatrix}$$

- A questo punto intersechiamo la retta r con la curva ellittica $\mathcal{C}_{a,b}$, in modo da determinare i valori di t:

$$(y_P + t(y_Q - y_P))^2 - (x_P + t(x_Q - x_P))^3 - a(x_P + t(x_Q - x_P)) - b = 0$$

  Notiamo che questa equazione ammette tre soluzioni, ma due di queste le conosciamo già:

  - $t = 0$ corrisponde al punto P;
  - $t = 1$ corrisponde al punto Q;
  - Essendo $t = 0$ e $t = 1$ soluzioni, l'equazione precedente può essere riscritta nella forma:

$$(y_P+t(y_Q-y_P))^2-(x_P+t(x_Q-x_P))^3-a(x_P+t(x_Q-x_P))-b = \alpha\, t(t-1)(t-\lambda)$$

- Escplicitando il polinomio a sinistra otteniamo:

$$t\,[2y_P(y_Q-y_P)+(y_Q-y_P)^2t-3x_P^2(x_Q-x_P)-3x_P(x_Q-x_P)^2t-(x_Q-x_P)^3t^2-a(x_Q-x_P)] \quad (1)$$

  Uguagliando quanto ottenuto ad $\alpha\, t(t-1)(t-\lambda)$, ricaviamo che α è proprio il coefficiente di t^2, dunque:

$$\alpha = -(x_Q - x_P)^3$$

- Infine notiamo che $\alpha\lambda$ è il termine noto di (1), dunque:

$$\lambda = \frac{\alpha\lambda}{\alpha} = \frac{a}{(x_Q - x_P)^2} + \frac{3x_P^2}{(x_Q - x_P)^2} - \frac{2y_P(y_Q - y_P)}{(x_Q - x_P)^3}$$

  questo però a patto che $x_Q \neq x_P$.

Concludiamo quindi che le formule esplicite per $P+Q$ sono date da:

- Se $x_Q = x_P$ si ha che
$$P+Q = [0:1:0]$$
cioè è il punto all'infinito;

- Se $x_Q \neq x_P$ allora:
$$\begin{cases} x_{P+Q} = x_P + \left(\frac{a+3x_P^2}{x_Q-x_P} - \frac{2y_P(y_Q-y_P)}{(x_Q-x_P)^2}\right) \\ y_{P+Q} = -y_P - \left(\frac{a+3x_P^2}{(x_Q-x_P)^2} - \frac{2y_P(y_Q-y_P)}{(x_Q-x_P)^3}\right)(y_Q-y_P) \end{cases}$$

A questo punto cerchiamo le formule per il raddoppio $2P$:

- Consideriamo il punto P appartenente alla curva $\mathcal{C}_{a,b}$, rispettivamente di coordinate $P = (x_P, y_P)$
- Cerchiamo l'equazione della retta tangente alla curva $\mathcal{C}_{a,b}$ passante per P
$$r : P + t\vec{v} = \begin{pmatrix} x_P \\ y_P \end{pmatrix} + t \begin{pmatrix} x_v \\ y_v \end{pmatrix} = \begin{pmatrix} x_P + tx_v \\ y_P + ty_v \end{pmatrix}$$
- A questo punto sostituiamo nella curva $\mathcal{C}_{a,b}$:
$$(y_P + ty_v)^2 - (x_p + tx_v)^3 - a(x_P + tx_v) - b = 0,$$
da cui
$$y_P^2 + 2y_p y_v t + t^2 y_v^2 - x_P^3 - 3x_P^2 x_v t - 3x_P x_v^2 t^2 - t^3 x_v^3 - ax_P - ax_v t - b = 0 \qquad (1)$$
Notiamo che, dato che il punto P appartiene alla curva $\mathcal{C}_{a,b}$, si ha
$$y_P^2 - x_P^3 - ax_P - b = 0$$
Dunque questo termine può essere cancellato nell'espressione (1).
- Dato che ci interessa la retta tangente, anche il termine lineare in t deve sparire, per cui si ha un'equazione lineare in (x_v, y_v), che è
$$2y_P y_v - 3x_P^2 x_v - ax_v = 0,$$
da cui si ha
$$(2y_P)y_v - (3x_P^2 + a)x_v = 0,$$

e, dato che ci basta una sola soluzione di questa equazione, prendiamo

$$\begin{cases} y_v = 3x_P^2 + a \\ x_v = 2y_P \end{cases}$$

- Notiamo che:

  - Se $y_P = 0$ si ha che
  $$2P = [0 : 1 : 0]$$
  cioè è il punto all'infinito;

  - Se $y_P \neq 0$, allora sostituiamo (x_v, y_v) nell'equazione (1).

  A questo punto i termini di grado 1 spariscono, e resta:

  $$t^2(3x_P^2 + a)^2 - t^3(2y_P)^3 - 3x_P(2y_P)^2t^2 = 0$$

  e mettendo t^2 in evidenza si ottiene

  $$t^2[(3x_P^2 + a)^2 - 12y_P^2x_P - 8y_P^3t] = 0$$

  da cui si ha

  - $t = 0$ che non consideriamo perché corrisponde al punto P;
  - $t = \frac{(3x_P^2+a)^2-12y_P^2x_P}{8y_P^3} = \frac{(3x_P^2+a)^2}{8y_P^3} - \frac{3x_P}{2y_P}$

- Concludiamo che le formule esplicite per $2P$ sono date da:

  - Se $y_P = 0$ si ha che
  $$2P = [0 : 1 : 0]$$
  cioè è il punto all'infinito;

  - Se $y_P \neq 0$ allora:

  $$\begin{cases} x_{2P} = x_P + \left(\frac{(3x_P^2+a)^2}{8y_P^3} - \frac{3x_P}{2y_P}\right)2y_P = x_P + \frac{(3x_P^2+a)^2}{4y_P^2} - 3x_P = -2x_P + \frac{(3x_P^2+a)^2}{4y_P^2} \\ y_{2P} = -y_P - \left(\frac{(3x_P^2+a)^2}{8y_P^3} - \frac{3x_P}{2y_P}\right)(3x_P^2 + a) \end{cases}$$

3 Algoritmo di Fattorizzazione di Lenstra

3.1 Descrizione del metodo di Lenstra

Una volta che abbiamo richiamato alcune proprietà generali delle curve ellittiche, procediamo con la descrizione del metodo di Lenstra. L'obiettivo è quello di fattorizzare un certo numero $n \in \mathbb{N}$ dispari e richiediamo anche che sia composto, perché altrimenti avrebbe fattorizzazione banale.
Dunque, dato n, consideriamo un certo punto $P = (x_p, y_p)$ con x_p e $y_p \in \mathbb{Z}/(n)$ e, scelto un parametro $a \in \mathbb{Z}/(n)$, definiamo il parametro b in modo che il punto P considerato appartenga alla cubica di equazione $y^2 = x^3 + ax + b$. Una volta determinato il valore di b, vogliamo che la nostra curva sia liscia, dunque è necessario controllare che il discriminante $\Delta = 4a^3 + 27b^2 \neq 0 \ mod(n)$. Per farlo conviene calcolare $d = MCD(\Delta, n)$ e distinguere 3 casi:

1. Se $d \neq 1$ e $d \neq n$, vuol dire che abbiamo trovato un divisore di n;
2. Se $d = n$, vuol dire che $n|\Delta$, dunque $\Delta = 0 \ mod(n)$. Se siamo in questo caso, vuol dire che la curva $\mathcal{C}_{a,b}$ costruita non è liscia e non è possibile trovare un primo p tale che $p|n$ e $\Delta \neq 0 \ mod(p)$, dunque dobbiamo cambiare il parametro a e ricominciare;
3. Se invece $d = 1$, vuol dire che la curva costruita con i parametri a e b è liscia e inoltre contiene il punto P. Se siamo in questo caso, vuol dire che $\forall$ p tale che $p|n$, allora $\Delta \neq 0 \ mod(p)$, quindi se consideriamo la riduzione della curva di equazione $y^2 = x^3 + ax + b \ mod(p)$ e riduciamo anche le coordinate di $P \ mod(p)$, otteniamo una curva ellittica puntata, cioè un gruppo abeliano puntato.

Attenzione. A priori, se consideriamo la curva $mod(n)$, non si può parlare di curva ellittica, perché $\mathbb{Z}/(n)$ non è un campo.

A questo punto l'idea è la seguente: fissato un certo $\gamma \in \mathbb{N}$, vogliamo scommettere che per qualche p tale che $p|n$, $|\mathcal{C}_{a,b}|$ sia γ-liscia, cioè

$$\text{se } |\mathcal{C}_{a,b}| = p_1^{\beta_1} * \cdots * p_k^{\beta_k}, \text{ allora } \forall\ k,\ p_k^{\beta_k} \leq \gamma \quad .$$

Osservazione 1. Azzardando di più, possiamo fissare un certo insieme di primi $I = \{2, 3, 5, 7, 11\}$ e scommettere che $|\mathcal{C}_{a,b}|$ sia γ-liscia rispetto a I.

Per scegliere γ usiamo la **stima di Hasse**, che afferma che, data una curva ellittica $\mathcal{C}$ su un campo finito $\mathbb{F}_p$, allora

$$|\mathcal{C}| \leq (\sqrt{p} + 1)^2$$

Nel nostro caso quindi, poiché stiamo considerando una curva ellittica $\mathcal{C}_{a,b}$ con p tale che $p|n$, allora possiamo dire che $p \leq \sqrt{n}$, dunque

$$|\mathcal{C}_{a,b}| \leq (\sqrt[4]{n}+1)^2 = \gamma$$

Una volta fissato $\gamma \in \mathbb{N}$, andiamo a considerare un multiplo comune di tutti i numeri naturali da 1 a γ e scegliamo proprio

$$m = mcm(1,\dots,\gamma) = \prod_{p\leq\gamma} p^{e_p} \text{ con } e_p = \lfloor log_p(\gamma) \rfloor = \lfloor \frac{log(\gamma)}{\log(p)} \rfloor$$

Con questa scelta mostriamo che, se la nostra scommessa dovesse essere vincente, allora riusciamo a fattorizzare n. Supponiamo, dunque, di aver vinto la scommessa:

- In questo caso allora $|\mathcal{C}_{a,b}|$ è γ-liscia, quindi per definizione $|\mathcal{C}_{a,b}| = p_1^{\beta_1} * \dots * p_k^{\beta_k}$, allora $\forall\ k$, $p_k^{\beta_k} \leq \gamma$. Tuttavia sappiamo anche che $m = \prod_{p\leq\gamma} p^{e_p}$ con $e_p =$ massimo esponente per cui $p^{e_p} \leq \gamma$. Da questo segue dunque che $\forall\ k$, $\beta_k \leq e_k$, da cui $|\mathcal{C}_{a,b}| \mid m$.

- Dire che $|\mathcal{C}_{a,b}| \mid m$, vuol dire che $\exists\ l : m = l * |\mathcal{C}_{a,b}|$, dunque $m * P = l * |\mathcal{C}_{a,b}| * P = 0$, cioè è l'elemento neutro del nostro gruppo additivo, che non è altro che il punto all'infinito.

- Quindi, se abbiamo vinto la scommessa, necessariamente $m * P = 0$. Tuttavia, calcolare $m*P$ vuol dire calcolare $p^{e_p}(\dots(3^{e_3}(2^{e_2}*P)))$. Partiamo calcolando $2^{e_2}*P$ con le formule del raddoppio definite nella sezione precedente e successivamente procediamo con il calcolo di $k*P$, con $k \in \{3,5,7,11,\dots\}$ usando sia la formula del raddoppio che quella della somma.

- Ogni volta che applichiamo una di queste formule, i denominatori potrebbero annullarsi e prima o poi lo faranno sicuramente, dato che stiamo supponendo di aver vinto la scommessa. Quando uno dei denominatori (che sono o $2y_P$ o $x_Q - x_P$) si annullano $mod(p)$ (con p generico tale che $p|n$), vuol dire che $p|2y_P$ oppure $p|(x_Q - x_P)$. Ma dato che per ipotesi $p|n$, allora $p|(2y_P, n)$ oppure $p|(x_Q - x_P, n)$, dunque in particolare $(2y_P, n) \neq 1$ e $(x_Q - x_P, n) \neq 1$. Allora le possibilità sono due:

  1. $(2y_P, n) = d$ o $(x_Q - x_P, n) = d$, cioè abbiamo trovato un divisore proprio di n;
  2. $(2y_P, n) = n$ o $(x_Q - x_P, n) = n$, allora cambiamo a o il punto P e ricominciamo.

Osservazione 2. Noi non conosciamo p tale che $p|n$, ma solo n. Dunque per poter costruire il nostro algoritmo, invece di lavorare $mod(p)$, lavoriamo $mod(n)$. Tuttavia in questo caso dobbiamo stare attenti all'invertibilità dei denominatori, quindi l'algoritmo, ogni volta che richiamerà le formule di raddoppio o di somma, dovrà controllare che $(2y_P, n) = 1$ e $(x_Q - x_P, n) = 1$ per poter andare avanti.

Osservazione 3. L'algoritmo di Lenstra risulta essere molto efficiente, poiché il numero di gruppi abeliani su cui possiamo scommettere è molto elevato. In particolare abbiamo per ogni punto $P = (x_P, y_P)$ fissato, n possibili modi di scegliere $a \in \mathbb{Z}/(n)$. Se facciamo variare anche le coordinate di P abbiamo altre n^2 possibilità, quindi in totale possiamo scommettere su n^3 gruppi.

3.2 Codice Matlab

```
% Progetto di Umberto Gallo & Giorgia Iannarelli

clc
clear

% Chiedo in input un n da fattorizzare
prompt = "Inserire n da fattorizzare: ";
n = input(prompt);

% Controllo che n sia intero e positivo
while n ~= floor(n) || n <= 0
    warning('bisogna inserire un numero intero e positivo.')
    prompt = "Inserire n da fattorizzare: ";
    n = input(prompt);
end

% Escludo il caso banale n = 1
if n == 1
    disp(['Il valore inserito in input  pari a 1, quindi ha ' ...
        'fattorizzazione banale.'])
    return;
```

```
22 end
23
24 % Escludo il caso banale in cui n sia pari
25 if mod(n,2)==0
26     q=n/2;
27     disp(['Il valore inserito  pari, dunque ha una ' ...
28         'fattorizzazione banale n = 2 * ', num2str(q)])
29     return;
30 end
31
32 % Controlliamo l'eventuale primalit di n usando uno dei test di primalit visti.
33 % Se il test usato d esito positivo, segnaliamo all'utente che n  primo ed
34 % usciamo dal programma.
35
36 if (test_Miller_Rabin(n, 100) == true)
37     disp(['Il valore n inserito in input  un numero primo. Il programma ' ...
38         'verr interrotto.'])
39     return;
40 end
41
42 % Chiedo in input un punto P
43 disp('Inserire le coordinate del punto P.')
44
45 prompt = "Inserire xP: ";
46 xP = input(prompt);
47 % Controllo che xP sia intero
48 while xP ~= floor(xP)
49     disp('Errore: inserire un numero intero')
50     prompt = "Inserire xP: ";
51     xP = input(prompt);
52 end
53
54 prompt = "Inserire yP: ";
55 yP = input(prompt);
```

```
% Controllo che yP sia intero
while yP ~= floor(yP)
    disp('Errore: inserire un numero intero')
    prompt = "Inserire yP: ";
    yP = input(prompt);
end

% Salvo P in un vettore
P = [xP, yP];

% Costruisco la curva ellittica:
% Prendo 'a' pari ad 1 e incremento fino ad n - 1 se necessario
% (a \in Z/(n)).
a = 1;
B = (nthroot(n, 4) + 1)^2; % Soglia determinata mediante la disuguaglianza
% di Hasse

while a < n
    % Trovo b imponendo il passaggio di P per l'equazione della curva
    b = (yP^2) - (xP^3) - (a * xP);
    b = mod(abs(b), n);
    % Calcolo il discriminante 4a^3 + 27b^2 mod n
    disc = mod((4 * a^3) + (27 * b^2), n);

    % Calcolo il MCD fra il discriminante d ed n
    d = MCD(disc, n);
    if d ~= 1 && d ~= n
        disp(['Ho trovato un divisore d = ', num2str(d)])
        return
    end
    if d == n
        % Se d = n, allora n divide il discriminante, ossia disc = 0
        % mod n
        a = a + 1;
```

```
90      end
91      if d == 1
92          p = 2; % perch partiamo calcolando (2^e2)*P
93          e2 = floor(log(B) / log(p));
94          j = 1;
95          while j <= e2
96              % In Q salvo un punto della forma 2^k * P
97              [Q, flag] = raddoppia(P, n, a);
98              if flag == 2
99                  % flag = 2 se raddoppia ha un trovato e stampato un
100                 % divisore, quindi l'algoritmo termina
101                 return
102             end
103             if flag == 1
104                 % flag = 1 se MCD(yP, n) = n, quindi bisogna
105                 % cambiare a e ricominciare
106                 a = a + 1;
107                 break
108             end
109             if flag == 0
110                 j = j + 1;
111                 P = Q;
112             end
113         end
114
115         if flag == 1
116             % vuol dire che ho cambiato a nel ciclo precedente
117             % e devo passare alla prossima iterazione del while a<n
118             continue
119         end
120
121         % Se nel while non abbiamo trovato un divisore si ha che
122         % Q = 2^(e_2) * P.
123
```

```
124         % In p salvo una lista di primi
125         p = [3, 5, 7, 11];
126
127         for i = 1:length(p)
128             % Salvo in T l'ultimo punto trovato nel ciclo precedente
129             T = Q;
130             k = p(i);
131             e = floor(log(B) / log(k));
132             j = 1;
133
134             while j <= e
135                 [Q,flag] = calcola_kP(T, k, n, a);
136                 if flag == 2
137                     % flag = 2 se nel sottoprogramma calcola_kP, somma o
138                     % raddoppia hanno trovato e stampato un divisore,
139                     % quindi l'algoritmo termina
140                     return
141                 end
142                 if flag == 1
143                     % flag = 1 vuol dire che o il denominatore di somma o
144                     % di raddoppia si  annullato mod(n), quindi bisogna
145                     % cambiare a e ricominciare
146                     a = a + 1;
147                     break
148                 end
149                 if flag == 0
150                     % flag = 0, quindi  stato possibile calcolare k *
151                     % Q mod n e non sono stati trovati divisori propri
152                     % di n procedendo con le somme/raddoppi. Incremento
153                     % dunque il valore di j -> j + 1 e  vado avanti
154                     j = j + 1;
155
156                     % salvo dentro T il Q calcolato con calcola_kP
157                     T=Q;
```

```
158                 end
159             end
160
161             if flag == 1
162                 % vuol dire che ho cambiato a e devo uscire dal ciclo for e
163                 % andare alla prossima iterazione del while (a<n)
164                 break
165             end
166         end
167     end
168 end
169
170 disp("Non sono riuscito a fattorizzare n provando con tutti i possibili a" + ...
171     "in Z/(n). Prova a rilanciare il programma cambiando le coordinate" + ...
172     "del punto P...")
```

3.3 Spiegazione dell'algoritmo

Andiamo ora ad analizzare cosa fa nello specifico il programma, mentre nella sezione successiva descriveremo il funzionamento dei vari sottoprogrammi utilizzati.

Inserimento da tastiera del numero che si vuole fattorizzare

```
1  % Chiedo in input un n da fattorizzare
2  prompt = "Inserire n da fattorizzare: ";
3  n = input(prompt);
4
5  % Controllo che n sia intero e positivo
6  while n ~= floor(n) || n <= 0
7      warning('bisogna inserire un numero intero e positivo.')
8      prompt = "Inserire n da fattorizzare: ";
9      n = input(prompt);
10 end
```

- Chiediamo innanzitutto l'inserimento da parte dell'utente del numero n che si vuole

fattorizzare.

- A questo punto, controlliamo che il numero n inserito sia intero e positivo. Per farlo, usiamo un ciclo *while* in modo tale che ogni volta che n non rispetta le condizioni, stampa un messaggio di errore e ne richiede l'inserimento.

Controllo di non essere in un caso banale

```
% Escludo il caso banale n = 1
if n == 1
    disp(['Il valore inserito in input  pari a 1, quindi ha ' ...
        'fattorizzazione banale.'])
    return;
end

% Escludo il caso banale in cui n sia pari
if mod(n,2)==0
    q=n/2;
    disp(['Il valore inserito  pari, dunque ha una ' ...
        'fattorizzazione banale n = 2 * ', num2str(q)])
    return;
end
```

- Controlliamo che n non sia 1. Se $n = 1$ allora stampiamo il messaggio *"Il valore inserito in input è pari a 1, quindi ha fattorizzazione banale"* e terminiamo l'esecuzione.

- Controlliamo che n non sia pari. Nel caso in cui n sia pari, allora avrà fattorizzazione banale, quindi calcoliamo il quoziente q e successivamente stampiamo il messaggio *"Il valore inserito in input è pari, quindi ha fattorizzazione banale* $n = 2 * q$*"*, terminando così l'esecuzione.

Controllo che n non sia primo

```
if (test_Miller_Rabin(n, 100) == true)
    disp(['Il valore n inserito in input  un numero primo. Il programma ' ...
        'verr interrotto.'])
    return;
end
```

- Controlliamo ora che il numero n inserito non sia primo, perché se lo fosse, allora non riusciremmo a fattorizzarlo e il programma terminerebbe. Per farlo usiamo uno dei test di primalità visti a lezione, in particolare quello di Miller-Rabin, essendo il test più efficiente.
- Il test di Miller-Rabin lo iteriamo al massimo 100 volte. Se per ogni iterazione il test dà esito *true*, allora possiamo concludere che con buona probabilità il numero n inserito è primo, quindi terminiamo l'esecuzione del programma stampando un messaggio in uscita.

Inserimento delle coordinate del punto P

```
% Chiedo in input un punto P
disp('Inserire le coordinate del punto P.')

prompt = "Inserire xP: ";
xP = input(prompt);
% Controllo che xP sia intero
while xP ~= floor(xP)
    disp('Errore: inserire un numero intero')
    prompt = "Inserire xP: ";
    xP = input(prompt);
end

prompt = "Inserire yP: ";
yP = input(prompt);
% Controllo che yP sia intero
```

```
while yP ~= floor(yP)
    disp('Errore: inserire un numero intero')
    prompt = "Inserire yP: ";
    yP = input(prompt);
end

% Salvo P in un vettore
P = [xP, yP];
```

- Se siamo arrivati a questo punto del codice, vuol dire che il numero n inserito è dispari ed è composto, quindi andiamo a cercare un divisore. Iniziamo quindi a descrivere il funzionamento dell'algoritmo di Lenstra.
- Innanzitutto chiediamo all'utente di inserire le coordinate x_P e y_P di un punto P che poi ci servirà per costruire la nostra curva ellittica puntata.
- Nel caso in cui i valori inseriti in input non fossero interi, allora tramite dei cicli *while* richiediamo l'inserimento.
- Infine, salviamo le coordinate (x_P,y_P) in un vettore che chiamiamo P.

Costruzione della curva ellittica

```
a = 1;
B = (nthroot(n, 4) + 1)^2; % Soglia determinata mediante la disuguaglianza
% di Hasse

while a < n
    % Trovo b imponendo il passaggio di P per l'equazione della curva
    b = (yP^2) - (xP^3) - (a * xP);
    b = mod(abs(b), n);
    % Calcolo il discriminante 4a^3 + 27b^2 mod n
    disc = mod((4 * a^3) + (27 * b^2), n);

    % Calcolo il MCD fra il discriminante d ed n
    d = MCD(disc, n);
```

```
14      if d ~= 1 && d ~= n
15          disp(['Ho trovato un divisore d = ', num2str(d)])
16          return
17      end
18      if d == n
19          % Se d = n, allora n divide il discriminante, ossia disc = 0
20          % mod n
21          a = a + 1;
22      end
23      if d == 1
```

- Andiamo ora a costruire la nostra curva ellittica. Per farlo prendiamo un certo $a \in \mathbb{Z}/(n)$. Per semplicità nel nostro programma prendiamo $a = 1$.

- Definiamo poi $B = (\sqrt[4]{n} + 1)^2$ seguendo la stima di Hasse. In particolare, il valore di B rappresenta la nostra "scommessa" sulla cardinalità della curva ellittica $\mathcal{C}_{a,b}$ e ci servirà nel seguito.

- Facciamo partire un ciclo *while* per $a < n$, dato che $a \in \mathbb{Z}/(n)$, e a questo punto troviamo il parametro b della curva di equazione $y^2 = x^3 + ax + b$, imponendo il passaggio per P in essa. Successivamente riduciamo $mod(n)$ tale valore di b.

- A questo punto, affinché la cubica $y^2 = x^3 + ax + b$ sia una curva ellittica, abbiamo bisogno che sia liscia, quindi calcoliamo il discriminante $\Delta = 4a^3 + 27b^2$ $mod(n)$ e controlliamo che sia diverso da 0. Per farlo conviene calcolare $d = MCD(\Delta, n)$ e distinguere 3 casi:

  1. Se $d \neq 1$ e $d \neq n$, vuol dire che abbiamo trovato un divisore di n, quindi terminiamo l'esecuzione stampando in output il divisore d;
  2. Se $d = n$, vuol dire che $n|\Delta$, dunque $\Delta = 0$ $mod(n)$. Se siamo in questo caso, vuol dire che la curva $\mathcal{C}_{a,b}$ costruita non è liscia, dunque dobbiamo cambiare il parametro a e ricominciare;
  3. Se invece $d = 1$, vuol dire che la curva costruita con i parametri a e b è liscia e inoltre contiene il punto P. Se siamo in questo caso, procediamo con la ricerca del divisore di n.

Calcolo di m*P

```
    if d == 1
        p = 2; % perch partiamo calcolando (2^e2)*P
        e2 = floor(log(B) / log(p));
        j = 1;
        while j <= e2
            % In Q salvo un punto della forma 2^k * P
            [Q, flag] = raddoppia(P, n, a);
            if flag == 2
                % flag = 2 se raddoppia ha un trovato e stampato un
                % divisore, quindi l'algoritmo termina
                return
            end
            if flag == 1
                % flag = 1 se MCD(yP, n) = n, quindi bisogna
                % cambiare a e ricominciare
                a = a + 1;
                break
            end
            if flag == 0
                j = j + 1;
                P = Q;
            end
        end

        if flag == 1
            % vuol dire che ho cambiato a nel ciclo precedente
            % e devo passare alla prossima iterazione del while a<n
            continue
        end
```

- Se siamo arrivati a questo punto del codice, vuol dire che abbiamo costruito la nostra curva ellittica liscia e vogliamo andare a cercare un divisore di n. Seguendo quello che abbiamo visto nella spiegazione del metodo di fattorizzazione di Lenstra,

sappiamo che nel caso in cui avessimo vinto la nostra scommessa data da B, allora riusciremmo a fattorizzare n. Ma, vincere la scommessa, vuol dire che $|\mathcal{C}_{a,b}|$ divide m con $m = mcm\{1, \dots, B\} = \prod_{p \leq B} p^{e_p}$. Quindi, per fattorizzare n necessariamente nel calcolo di $m * P$, ad un certo punto dobbiamo trovare l'elemento neutro della curva ellittica $\mathcal{C}_{a,b}$, cioè il punto all'infinito. Ma trovare il punto all'infinito, vuol dire che nelle formule per $P + Q$ o $2P$, i denominatori si sono annullati $mod(n)$.

- Procediamo quindi con il calcolo di $m * P$, da $2^{e_2} * P$. Per farlo, usiamo un ciclo *while* per $j \leq e_2$ e applichiamo la funzione *raddoppia(P,n,a)* e_2 volte.

- Ogni volta che usiamo la funzione *raddoppia(P,n,a)*, controlliamo il valore del marcatore *flag*, che ci permette di capire se i denominatori si sono annullati $mod(n)$:

  1. Se *flag* $= 2$ allora vuol dire che $MCD(2y_p, n) = d$, cioè abbiamo trovato un divisore di n, quindi terminiamo l'esecuzione e stampiamo il divisore trovato;
  2. Se *flag* $= 1$ allora vuol dire che $MCD(2y_p, n) = n$, dunque dobbiamo cambiare a e ricominciare, perché il denominatore in questo caso è proprio 0 e non possiamo procedere con i calcoli. In tal caso, usiamo il comando *break* che ci permette di uscire dal ciclo *while* in cui ci troviamo;
  3. Se *flag* $= 0$, vuol dire che $MCD(2y_p, n) = 1$, cioè $2y_p$ è invertibile $mod(n)$. In questo caso possiamo procedere con il calcolo di $2^{e_2} * P$, quindi aggiorniamo j e definiamo $P = Q$ che è il risultato ottenuto dalla funzione *raddoppia(P,n,a)*.

- L'ultimo *if* è necessario, perché nel caso in cui avessimo cambiato a, ci permetterebbe di saltare tutte le istruzioni successive contenute nel ciclo *while* $a < n$ e procedere con l'iterazione successiva.

```
% In p salvo una lista di primi
p = [3, 5, 7, 11];

for i = 1:length(p)
    % Salvo in T l'ultimo punto trovato nel ciclo precedente
    T = Q;
    k = p(i);
    e = floor(log(B) / log(k));
```

```
9               j = 1;
10
11              while j <= e
12                  [Q,flag] = calcola_kP(T, k, n, a);
13                  if flag == 2
14                      % flag = 2 se nel sottoprogramma calcola_kP, somma o
15                      % raddoppia hanno trovato e stampato un divisore,
16                      % quindi l'algoritmo termina
17                      return
18                  end
19                  if flag == 1
20                      % flag = 1 vuol dire che o il denominatore di somma o
21                      % di raddoppia si  annullato mod(n), quindi bisogna
22                      % cambiare a e ricominciare
23                      a = a + 1;
24                      break
25                  end
26                  if flag == 0
27                      % flag = 0, quindi  stato possibile calcolare k *
28                      % Q mod n e non sono stati trovati divisori propri
29                      % di n procedendo con le somme/raddoppi. Incremento
30                      % dunque il valore di j -> j + 1 e  vado avanti
31                      j = j + 1;
32
33                      % salvo dentro T il Q calcolato con calcola_kP
34                      T=Q;
35                  end
36              end
37
38              if flag == 1
39                  % vuol dire che ho cambiato a e devo uscire dal ciclo for e
40                  % andare alla prossima iterazione del while (a<n)
41                  break
42              end
```

```
43          end
44      end
45  end
46
47  disp("Non sono riuscito a fattorizzare n provando con tutti i possibili a" + ...
48      "in Z/(n). Prova a rilanciare il programma cambiando le coordinate" + ...
49      "del punto P...")
```

- Se siamo arrivati qui, vuol dire $2^{e_2} * P$ è stato determinato correttamente, quindi procediamo con il calcolo di $m * P$. Dato che l'algoritmo di Lenstra generalmente trova un divisore abbastanza velocemente, invece di considerare tutti i primi $p \leq B$, prendiamo solo i primi numeri primi $\{3, 5, 7, 11\}$, che salviamo in una lista.

- A questo punto, scorriamo la lista di primi. Salviamo in una variabile T l'ultimo punto Q trovato nel ciclo precedente, che al passo 1 sarà proprio $2^{e_2} * P$. Inoltre definiamo $k = p(i)$ che al passo 1 sarà proprio $k = 3$ e calcoliamo $e_k = \lfloor \frac{\log(B)}{\log(k)} \rfloor$ che ci dà il numero di volte per cui dobbiamo calcolare $k * Q$.

- Tramite un ciclo *while* per $j \leq e_k$, procediamo con il calcolo di $k^{e_k} * Q$. Per farlo, richiamiamo la funzione *calcola_kP(T,k,n,a)* ed esattamente come prima, controlliamo ad ogni iterazione se i denominatori si annullano oppure no. Se non si annullano (i.e. *flag* $= 0$), allora passiamo all'iterazione successiva salvando in T l'ultimo Q trovato. Se, invece, ad un certo punto i denominatori si annullano, controlliamo il marcatore *flag*:

  1. Se *flag* $= 2$ allora vuol dire che $MCD(2y_p, n) = d$, cioè abbiamo trovato un divisore di n, quindi terminiamo l'esecuzione e stampiamo il divisore trovato;
  2. Se *flag* $= 1$ allora vuol dire che $MCD(2y_p, n) = n$, dunque dobbiamo cambiare a e ricominciare. Per farlo, usiamo il comando *break* che ci permette di uscire dal ciclo *while* in cui ci troviamo, mentre l'ultimo *if* è quello che ci permette di usicre dal ciclo *for* se *flag* $= 1$ per ricominciare tutto.

- Nel caso in cui il ciclo *while* per $a < n$ dovesse terminare senza aver trovato un divisore, allora stampiamo un messaggio di uscita, in cui chiediamo di rilanciare il programma cambiando le coordinate del punto P appartenente alla curva ellittica.

4 Sottoprogrammi presenti nel codice di Lenstra

Come sopra anticipato, in questa parte della tesina vogliamo spiegare brevemente i vari sottoprogrammi che vengono utilizzati nel codice di Lenstra.

4.1 Calcolo dell'MCD

```
function d = MCD(a, b)
    % Funzione per il calcolo del massimo comun divisore di due interi a e
    % b (con a>b)

    % Ci riduciamo a lavorare con numeri positivi.
    a = abs(a);
    b = abs(b);

    % Controllo che a>b, altrimenti scambio i due numeri
    if a<b
        temp=a;
        a=b;
        b=temp;
    end

    if(b == 0)
        % Per definizione, MCD(a, 0) := a
        d = a;
    elseif(b == 1)
        % a  coprimo con b
        d = 1;
    else
        % Eseguiamo ricorsviamente il calcolo del MCD finch non troviamo
        % resto 0
        d = MCD(b, mod(a, b));
    end
end
```

Questo sottoprogramma calcola l'MCD tra due numeri. Prende in input due numeri a e b e restituisce in output d che è proprio l'MCD cercato.
Andiamo ad analizzare nello specifico i vari passi del codice.

- Rendiamo positivi i nostri numeri a e b e lo facciamo tramite la funzione *abs* di *MATLAB*.
- Dato che vogliamo $a > b$, controlliamo se vale questa condizione. In caso contrario, scambiamo a e b e lo facciamo tramite una variabile di appoggio chiamata *temp*.
- A questo punto, andiamo ad analizzare i casi banali:
  - Se $b = 0$, per definizione $MCD(a, 0) := a$, dunque $d = a$;
  - Se $b = 1$, $MCD(a, 1) = 1$, dunque $d = 1$;
- Se non siamo nei due casi banali, quello che si fa è applicare l'algoritmo di Euclide delle divisioni successive. Eseguiamo ricorsivamente il calcolo dell'MCD tra b e il resto della divisione di a con b, finchè non troviamo resto 0.
  Una volta trovato il resto 0, abbiamo trovato anche l'MCD tra a e b.

4.2 Calcolo dell'identità di Bézout

```
1  function [x, y] = calcola_bezout(n, m)
2
3      % La funzione calcola i coefficienti di Bzout per nx + my = MCD(n, m).
4
5      check=false;
6
7      % Controllo se n>m. Se non lo  li scambio
8      if n<m
9          check=true;
10         temp=n;
11         n=m;
12         m=temp;
13     end
14
```

```
15      % Creo la matrice che mi permetter di trovare i coefficienti di Bezout
16      A=zeros(2);
17      A(1,2)=1;
18      A(2,1)=1;
19      B=eye(2);
20
21      while mod(n,m)~=0
22          q=floor(n/m);
23          r=n-q*m;
24          A(2,2)=-q;
25          B=A*B; % Devo moltiplicare a sinistra
26          n=m;
27          m=r;
28      end
29
30      % Devo fare un ultima iterazione
31      q=floor(n/m);
32      A(2,2)=-q;
33      B=A*B;
34
35      coeff=[1,0]*B;
36
37      % Se n ed m sono stati scambiati all'inizio, allora il coefficiente di
38      % n  coeff(2) e quello di m  coeff(1)
39      if check==true
40          x=coeff(2);
41          y=coeff(1);
42      else
43          x=coeff(1);
44          y=coeff(2);
45      end
46  end
```

Questo sottoprogramma calcola i coefficienti dell'identità di Bézout. In particolare prende in input due numeri n ed m e restituisce in output x e y che sono i coefficienti dell'identità

di Bézout, rispettivamente di n ed m. Andiamo ad analizzare nello specifico come funziona il codice:

- Inizializziamo una variabile booleana *check=false*, che ci serivirà nel seguito per capire se $n > m$.
- Controlliamo se $n > m$. Se non lo è, scambiamo i due numeri in modo che valga sempre che $n > m$ e definiamo *check=true* per tener traccia che n ed m sono stati scambiati.
- A questo punto, creiamo la matrice $A = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$ e definiamo un'altra matrice di appoggio $B = Id$, che verrà aggiornata ad ogni divisione successiva.
- A questo punto, applichiamo l'algoritmo di Euclide delle divisioni successive. Quindi finché il resto della divisione tra n ed m è $\neq 0$, continuiamo a dividere. Calcoliamo la parte intera inferiore del quoziente q tra n ed m, poi calcoliamo il resto e aggiorniamo la matrice A facendola diventare $A = \begin{pmatrix} 0 & 1 \\ 1 & -q \end{pmatrix}$. Fatto ciò, moltiplichiamo la matrice B a sinistra per A e infine ridefiniamo $n = m$ e $m = r$.
- Dato che usciamo dal ciclo quando il resto della divisione tra n ed m è 0, una volta usciti, per trovare i coefficienti giusti, dobbiamo fare un'ultima iterazione, altrimenti non staremmo considerando l'ultimo quoziente. A questo punto i coefficienti dell'identità di Bèzout sono proprio dati da:

$$\begin{pmatrix} x \\ y \end{pmatrix} = \begin{pmatrix} 0 & 1 \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q_n \end{pmatrix} \begin{pmatrix} 0 & 1 \\ 1 & -q_{n-1} \end{pmatrix} \cdots \begin{pmatrix} 0 & 1 \\ 1 & -q_1 \end{pmatrix}$$

- Come ultimo passaggio, tramite la variabile *check* controlliamo se all'inizio del programma è avvenuto lo scambio tra n ed m. Se questo è vero, *check=true* e quindi scambiamo anche i coefficienti dell'identità di Bézout, per cui $x = coeff(2)$ mentre $y = coeff(1)$.

4.3 Calcolo dell'inverso

```
function x = inverso(n, m)
    % La funzione prende in input due interi n ed m e restituisce, se esiste,
    % un inverso di n mod m.

    % ATTENZIONE: affinch l'inverso di n modulo m sia ben definito (ossia
    % n sia un elemento di (Z/(m))*), n dev'essere coprimo con m. Pertanto
    % verifichiamo se MCD(n, m) = 1: se ci non avviene, usciamo dal
    % programma.

    if(MCD(n, m) ~= 1)
        error('I due argomenti inseriti non sono fra loro coprimi, pertanto non'...
        'e possibile calcolare un inverso.');
        return;
    end

    % Una volta verificato che n ed m sono coprimi, l'inverso di n mod(m)
    % lo troviamo usando l'identit di Bezout. In particolare l'inverso di
    % n mod(m)  il coefficienti di n nell'identit

    [x,~]=calcola_bezout(n,m);
end
```

Questo sottoprogramma calcola l'inverso di due numeri.
In particolare, il programma prende in input due numeri n ed m e restituisce in output x che è l'inverso di n $mod(m)$.
L'idea è quella di partire dall'identità di Bèzout e ricavare da essa l'inverso.
Vediamo come è strutturato il codice.

- Come prima cosa, controlliamo se n è coprimo con m (ossia se $MCD(n, m) = 1$), perchè altrimenti non sarebbe ben definito l'inverso di n $mod(m)$. Se n ed m non sono coprimi, il nostro programma termina stampando un messaggio di errore.

- Se, invece, ci troviamo nella condizione in cui n ed m sono coprimi allora procediamo con il calcolo dell'inverso di n $mod(m)$ e richiamiamo la funzione che calcola i

coefficienti dell'identità di Bèzout. In particolare, l'inverso di n $mod(m)$ è proprio il coefficiente di n nell'identità.

4.4 Calcolo della potenza modulare

```
function potenza = potenza_mod(x, k, n)
    % Funzione che, presi in input tre interi x, k, n, calcola x^k mod n.

    % Verifichiamo se k  un intero non-negativo. In caso contrario,
    % interrompiamo l'esecuzione.
    if(k ~= floor(k) || k < 0)
        error('La potenza deve essere un numero intero non-negativo.')
        return;
    end

    % Verifichiamo se x  un numero intero. In caso contrario,
    % interrompiamo l'esecuzione.
    if(x ~= floor(x))
        error('La base della potenza deve essere un numero intero.')
        return;
    end

    % Infine, verifichiamo che n sia un numero intero positivo.
    if(n ~= floor(n) || n <= 0)
        error(['Il numero rispetto cui si sta calcolando la classe di resto'...
        'deve essere intero e positivo.']);
        return;
    end

    % Calcoliamo l'espressione binaria di k come stringa di 0 e 1.
    % Attenzione: l'output  di tipo char.
    bin = dec2bin(k);

    % Attenzione: l'output  un vettore char. Inoltre, bin(1) si
```

```
30      % riferisce alla potenza di ordine pi alto, mentre bin(length(bin)) a
31      % quella di ordine pi basso.
32
33      % Scorriamo in senso DECRESCENTE il vettore bin (ci serve partire dalla
34      % potenza 0-esima, ndr) e calcoliamo le potenze di x^(2^(i - 1)) mod n
35      % iterativamente. Se bin(i) == '1', calcoliamo potenza = potenza *
36      % x^(2^(i - 1)) mod n iterativamente.
37
38      potenza = 1;
39      m = x; % Variabile di appoggio per il calcolo delle potenze x^(2^(i-1))
40
41      for i = length(bin):-1:1
42          % Attenzione: all'indice i-esimo  associata la potenza 2^(i-1)-esima
43          % di x poich quella di ordine pi basso  x^(2^0) = x.
44          if(bin(i) == '1')
45              potenza = mod(potenza * m, n);
46          end
47          m = mod(m^2, n);
48      end
49  end
```

Questo sottoprogramma calcola la potenza modulare $x^k \ mod(n)$, quindi prende in input la base x, l'esponente k, il modulo n e restituisce in output la potenza modulare.
Andiamo ad analizzare nello specifico come funziona il codice:

- Iniziamo controllando che gli input k, x ed n soddisfano alcune condizioni. In particolare, k deve essere un intero non negativo, x un intero e n un intero positivo. Se non sono rispettate tali condizioni, allora terminiamo l'esecuzione stampando un messaggio di errore.

- A questo punto, per calcolare la potenza modulare in modo efficiente, conviene scrivere l'esponente k in binario e per farlo usiamo la funzione *dec2bin* di *MATLAB*, che trasforma k da decimale a binario, restituendo un vettore di caratteri.

- Inizializziamo la variabile *potenza* a 1 e definiamo una variabile di appoggio m per il calcolo delle potenze.

- Scorriamo il vettore di caratteri che contiene la scrittura binaria di k a partire dall'ultima posizione e ogni volta che troviamo un 1, andiamo ad aggiornare la variabile *potenza* moltiplicando $potenza * m$ e calcolandone poi la classe di resto $mod(n)$. Ad ogni ciclo calcoliamo sempre le potenze m^2 $mod(n)$ e moltiplichiamo per *potenza* solo se $bin = '1'$.

4.5 Test di primalità di Miller-Rabin

```
1  function test = test_Miller_Rabin(n, iter)
2      % Test di primalit di Miller-Rabin sul numero intero n e con massimo numero
3      % di iterazioni pari a iter.
4      % Il test restituisce false=0 quando il numero inserito non  sicuramente
5      % primo e restituisce true=1 se il numero inserito ha probabilit di
6      % esserlo
7
8      % N.B. Per questo test ci sono almeno i 3/4 degli elementi in
9      % (Z/(n))* che ci permettono di scoprire che n non  primo.
10
11
12     % Verifichiamo che n sia intero e positivo
13     if(n ~= floor(n) || n <= 0)
14         test = false;
15         error('Il valore di n deve essere intero e positivo.');
16         return;
17     end
18
19     % Escludiamo i casi banali n = 2, n pari ed n = 1
20     if(n == 2)
21         test = true;
22         return;
23     elseif(mod(n, 2) == 0 || n == 1)
24         test = false;
25         return;
26     end
```

```
27
28      % Inizializziamo test al valore true: non appena rileviamo che il test
29      % di Fermat NON  soddisfatto, cambiamo il suo valore a false
30      test = true;
31
32      for i = 1:iter
33          a = randi([2 (n -1)]);
34          if(MCD(a, n) > 1)
35              % se il massimo comun divisore fra a ed n  > 1, esso  un
36              % divisore proprio di n -> n non  primo.
37              test = false;
38              return;
39          end
40
41          % Arrivati qui sappiamo che a  un invertibile di Z/n. Inoltre, n
42          % dispari -> n - 1 = 2^k * d, con d dispari. Calcoliamo i valori di
43          % k=#numero di elementi della successione di M-R (con divisioni
44          % successive) e d=divisore dispari di n-1.
45
46          bin=dec2bin(n-1);
47          for j=1:length(bin)
48              % cerco l'ultimo 1 dello sviluppo binario e il #zeri in fondo
49              % corrisponde a k
50              if bin(j)=='1'
51                  indice=j;
52              end
53          end
54          k=length(bin)-indice;
55          d = bin2dec(bin(1:indice));
56
57          % Verifichiamo se a^d = 1 mod n: se tale relazione non  soddisfatta,
58          % procediamo con il calcolo della successione di M-R
59
60          x_0=potenza_mod(a,d,n); % calcolo primo elemento della successione
```

```
61          if(x_0 ~= mod(1, n))
62              x_old = x_0;
63              x_new = potenza_mod(x_old, 2, n);
64              j = 0;
65              flag = false; % Variabile flag per il ciclo while
66              while(j <= k - 1)
67                  if(mod(x_new, n) == mod(1, n))
68                      % Se trovo un termine della successione pari a 1,
69                      % allora fermo il ciclo e vado a guardare il termine
70                      % della successione precedente. Se trovo -1, allora il
71                      % test  fallito e devo cambiare a. Se invece non
72                      % troviamo -1, allora concludiamo che n NON  primo
73                      flag = true;
74                      break;
75                  end
76                  x_old = x_new;
77                  x_new = potenza_mod(x_old, 2, n);
78                  j = j + 1;
79              end
80
81              if(flag == true)
82                  % Se arriviamo qui, vuol dire che abbiamo trovato un 1 e
83                  % dobbiamo controllare il termine della successione
84                  % precedente
85                  if x_old~=mod(-1,n)
86                      test = false;
87                      return;
88                  end
89              else
90                  % se siao entrati qui vuol dire che non abbiamo mai trovato
91                  % 1, dunque il test di M-R  fallito, perch in realt
92                  % fallito Fermat --> dunque n NON  primo
93                  test=false;
94                  return
```

```
95              end
96          end
97      end
```

Questo sottoprogramma implementa il test di primalità di Miller-Rabin, che ci permette di capire se un certo numero n è primo oppure no. In particolare, il programma prende in input il numero n e un massimo numero di iterazioni $iter$ e restituisce in output una variabile booleana $test$ che può essere $false = 0$, quando il numero inserito non è sicuramente primo oppure $true = 1$, se il numero inserito ha buone probabilità di esserlo. Bisogna, però, fare attenzione al fatto che il test di Miller-Rabin, così come gli altri test di primalità visti, non assicura, se viene superato, che il numero inserito sia primo. Tuttavia per il test di Miller-Rabin, almeno il 75% degli elementi in $\mathbb{Z}/(n)$ sono testimoni della NON primalità di n, quindi è molto improbabile che se il test venga superato più volte, il numero non sia primo. Andiamo ad analizzare nello specifico come funziona il codice:

- Controlliamo innanzitutto che il numero n inserito sia intero e positivo. In caso contrario, resituiamo $false$ e terminiamo l'esecuzione con un messaggio di errore.
- A questo punto, escludiamo i casi banali $n = 1$ oppure n pari. Se $n = 2$, allora il test restituisce $true$ e termina l'esecuzione, se invece $n = 1$ oppure n è pari, allora il test restituisce $false$.
- Se siamo arrivati a questo punto del codice, allora non siamo in un caso banale e quindi inizializziamo la variabile di output $test = true$. Non appena il test rileva che n non è primo, cambiamo il suo valore in $false$ e terminiamo l'esecuzione.
- Iteriamo il test di Miller-Rabin tramite un ciclo for per un massimo di $iter$ volte. Quindi consideriamo una base a presa in maniera random tra 2 e $n-1$ e andiamo a calcolare $MCD(a, n)$. Se $MCD(a, n) > 1$, allora abbiamo trovato un divisore di n, quindi sicuramente n NON è primo. Restituiamo $false$ e terminiamo l'esecuzione.
- Se invece $MCD(a, n) = 1$, allora vuol dire che a è invertibile $mod(n)$ ed è anche dispari, quindi sicuramente $n - 1 = 2^k * d$, con d dispari. Calcoliamo quindi k, che è il numero di elementi della successione di Miller-Rabin e d il divisore dispari. Per farlo, consideriamo la scrittura in binario di $n-1$ e andiamo a cercare l'ultimo 1 presente nel vettore bin. Così facendo, possiamo concludere che k è proprio il numero di zeri presenti in fondo alla scrittura binaria di $n - 1$, mentre d si ottiene

passando nuovamente alla scrittura decimale di ciò che rimane, eliminando gli ultimi zeri.

- Costruiamo quindi la successione di Miller-Rabin. Il primo elemento della successione è $\alpha = x_0 = a^d \ mod(n)$, mentre gli altri si ottengono elevando al quadrato l'elemento precedente e calcolando poi la classe di resto $mod(n)$. Il numero di elementi della successione è dato da k. Controlliamo innanzitutto che $x_0 \neq 1 \ mod(n)$, perché se fosse uguale a 1, allora la successione di M-R sarebbe banale e il test sarebbe superato.

- Se $x_0 \neq 1 \ mod(n)$, procediamo con il calcolo della successione. Salviamo quindi in una variabile x_{old} l'ultimo elemento della successione calcolato e salviamo in x_{new} il suo quadrato $mod(n)$. Inizializziamo un contatore j a 0 e un marcatore $flag$ a $false$, che ci permette di terminare il ciclo $while$ nel caso in cui $x_{new} = 1 \ mod(n)$.

- Costruiamo i vari elementi della successione di M-R tramite il ciclo $while$ iterato k volte e ogni volta controlliamo se $x_{new} = 1 \ mod(n)$. Se lo è, allora $flag$ diventa $true$ e interrompiamo il ciclo.

- A questo punto, controlliamo il valore di $flag$. Se $flag = false$, allora vuol dire che il test di Miller-Rabin è superato (perché in realtà è superato quello di Fermat), quindi dobbiamo cambiare a. Se invece $flag = true$, dobbiamo controllare il valore di x_{old}. Se $x_{old} \neq -1 \mod (n)$, allora il test è fallito e quindi n sicuramente NON è primo. Se invece $x_{old} = -1 \mod (n)$, il test di Miller-Rabin è superato e quindi dobbiamo cambiare $a \in \mathbb{Z}/(n)$.

4.6 Calcolo del raddoppio 2P

```
1  function [Q, flag] = raddoppia(P, n, a)
2      % Funzione che prende in input un punto P mod n sulla una curva
3      % ellittica di parametro a e che restituisce 2 * P mod n.
4
5      xP = P(1);
6      yP = P(2);
7
8      d = MCD(yP, n); % calcolo questo perch  il termine che sta al
```

```
    % denominatore nelle formule
    flag = 0;

    if d == n
        flag = 1; % bisogna cambiare a nell'algoritmo di Lenstra.
        Q = 0; % Punto all'infinito

    elseif d ~= 1 && d ~= n
        disp(['Ho trovato un divisore d = ', num2str(d)]);
        flag = 2; %bisogna uscire da tutto
        Q = 0; % Punto all'infinito

    else
        % 2 * yP  invertibile.
        i = inverso(2 * yP, n);

        % Calcolo le coordinate di Q = 2 * P mod n con la formula del
        % raddoppio
        xQ = (i^2) * ((3 * (xP^2) + a)^2) - 2 * xP;
        xQ = mod(xQ, n);
        yQ = -yP - (3 * (xP^2) + a)*((i^3)*(3 * (xP^2) + a)^2 - i*3*xP);
        yQ = mod(yQ, n);
        Q = [xQ, yQ];
    end
end
```

Questo sottoprogramma calcola $2P\ mod(n)$, dove P è un punto sulla curva ellittica $\mathcal{C}_{a,b}$ di equazione $y^2 = x^3 + ax + b$.
Il programma prende in input P, l'intero $a \in \mathbb{Z}\ /\ (n)$, che è il parametro della curva ellittica, ed infine l'intero n.
Restituisce in output Q che è proprio $2P\ mod(n)$ e il marcatore $flag$, che controlla se i denominatori delle formule si annullano o meno.
Vediamo nello specifico i vari passi del codice.

- Innanzitutto chiamiamo x_P e y_P le coordinate di P, e calcoliamo l'MCD tra y_P ed n, tramite la funzione MCD. Poniamo il marcatore $flag$ uguale a 0.

Facciamo ora dei controlli:

- Se $d = n$, il marcatore $flag$ diventa 1 e questo corrisponde a cambiare il parametro a nell'algoritmo di Lenstra;
- Se, invece, d è diverso da 1 e da n, allora abbiamo trovato un divisore, dunque $flag$ diventa 2 e usciamo dal ciclo.
  In entrambi i casi abbiamo chiamato Q il punto all'infinito [0:1:0], e lo abbiamo posto uguale a 0.

- Se non ci troviamo in nessuno dei casi esplicitati al punto precedente, allora possiamo invertire $2y_P$ e a quel punto possiamo utilizzare le formule del raddoppio. Scriviamo le coordinate di Q, x_Q e y_Q e le riduciamo $mod(n)$. Dopodichè, l'ultimo passaggio consiste nel creare il vettore Q di coordinate x_Q e y_Q, che è proprio il raddoppio $2P$ $mod(n)$.

4.7 Calcolo della somma P+Q

```
1  function [T, flag] = somma(P, Q, n, a)
2      % Presi in input due punti P e Q e due interi n ed a,
3      % la funzione calcola la somma P + Q mod n come punto della curva
4      % ellittica di parametro a modulo n.
5
6      if P == Q
7          % Richiamo raddoppia
8          [T, flag] = raddoppia(P, n, a);
9      else
10         % Estraggo le coordinate
11         xP = P(1);
12         xQ = Q(1);
13         yP = P(2);
14         yQ = Q(2);
15
16         % Calcolo del MCD fra il denominatore xQ - xP ed n.
17         d = MCD(xQ - xP, n);
18         flag = 0;
```

```
19
20      if d == n
21          % Se flag  1, bisogna cambiare a nell'algoritmo di Lenstra.
22          flag = 1;
23          T = 0; % Punto all'infinito
24
25      elseif d ~= 1 && d ~= n
26          disp(['Ho trovato un divisore d = ', num2str(d)]);
27          flag = 2; %bisogna uscire da tutto
28          T = 0; % Punto all'infinito
29      else
30          % xQ - xP  invertibile
31          % Calcolo l'inverso di xQ - xP
32          i = inverso(xQ - xP, n);
33
34          % Calcolo le coordinate di P+Q con la formula della somma
35          xT = xP + ( i*(a + 3*(xP^2) ) - (i^2)*2*yP*(yQ-yP) );
36          xT = mod(xT, n);
37          yT = -yP - ( (i^2)*(a + 3*(xP^2)) - (i^3)*2*yP*(yQ-yP) )*(yQ-yP);
38          yT = mod(yT, n);
39          T = [xT, yT];
40      end
41  end
```

Questo sottoprogramma calcola la somma $mod(n)$ tra P e Q, dove P e Q sono due punti presi sulla curva ellittica $\mathcal{C}_{a,b}$ di equazione $y^2 = x^3 + ax + b$.
In particolare, il programma prende in input i due punti della curva P e Q, l'intero $a \in \mathbb{Z} \,/\, (n)$, che è il parametro della curva ellittica, ed infine l'intero n.
Resistuisce in output T che è proprio $P + Q \ mod(n)$ e il marcatore $flag$, che controlla se i denominatori delle formule si annullano o meno.
Vediamo nello specifico i vari passi del codice.

- Come prima cosa, controlliamo se i punti P e Q coincidono; in tal caso, la somma $P + Q \ mod(n)$ equivale a fare $2P \ mod(n)$ e per questo motivo richiamiamo la funzione *raddoppia* con input P, n ed a.

- Se, invece, P e Q sono distinti estraiamo le loro coordinate, chiamando x_P e y_P quelle di P e x_Q e y_Q quelle di Q.

- A questo punto, tramite la funzione MCD, calcoliamo l'MCD tra il denominatore $x_Q - x_P$ che figura nella formula della somma richiamata alla sezione 1, ed n.
  Poniamo il marcatore $flag$ uguale a 0 e facciamo dei controlli.

  - Se $d = n$, il marcatore $flag$ diventa 1 e questo corrisponde a cambiare il parametro a nell'algoritmo di Lenstra;
  - Se, invece, d è diverso da 1 e da n, allora abbiamo trovato un divisore, dunque $flag$ diventa 2 e usciamo dal ciclo.
    In entrambi i casi abbiamo chiamato T il punto all'infinito [0:1:0], e lo abbiamo posto uguale a 0.

- Se non ci troviamo in nessuno dei casi esplicitati al punto precedente, allora possiamo invertire $x_Q - x_P$ e possiamo utilizzare le formule della somma. Scriviamo le coordinate di T, x_T e y_T e le riduciamo $mod(n)$. Dopodichè, l'ultimo passaggio consiste nel creare il vettore T di coordinate x_T e y_T, che è proprio la somma $P+Q$ $mod(n)$.

4.8 Calcolo di k*P

```
function [Q, flag] = calcola_kP(T, k, n, a)
    % Presi in input un punto T e tre interi k, n ed a, la funzione calcola
    % k * P mod n come punto sulla curva ellittica di parametro a modulo n.

    % Definisco Q come (0,0) per iniziare il calcolo kP
    Q=[0,0];

    % scrivo k in binario con la funzione dec2bin che per mi da un char
    % come output
    bin = dec2bin(k);

    for i = length(bin) : -1 : 1
        if bin(i) == '1'
```

```
14              [Q, flag]=somma(Q, T, n, a);
15              if flag == 2
16                  % In questo caso somma ha trovato e stampato un divisore,
17                  % quindi resituisco al main flag=2 per terminare tutto
18                  return;
19              end
20              if flag == 1
21                  % flag = 1 vuol dire che il denominatore delle somme
22                  % divide n, quindi bisogna cambiare a e ricominciare
23                  return;
24              end
25          end
26
27          % Raddoppio T
28          [T, flag] = raddoppia(T, n, a);
29          if flag == 2
30              % In questo caso raddoppia ha un trovato e stampato un
31              % divisore, quindi restituisco al main flag=2 per terminare
32              % tutto
33              return
34          end
35          if flag == 1
36               % se flag = 1, allora MCD(yP, n) = n, quindi bisogna
37               % cambiare a.
38               return
39          end
40      end
```

Questo sottoprogramma calcola le coordinate del punto $k * P$. In particolare, prende in input il punto T (che rappresenta l'ultimo punto calcolato nel ciclo precedente), k, n e il parametro a della curva ellittica. Restituisce in output il punto $Q = k * T$ e il marcatore $flag$ che controlla che i denominatori nelle formule di somma o di raddoppio non si annullino. Andiamo ad analizzare nello specifico come funziona il codice:

- Innanzitutto salviamo nella variabile di output Q il punto $(0, 0)$ per inizializzare il calcolo e successivamente andiamo a calcolare la scrittura binaria di k, usando la

funzione *dec2bin* di *MATLAB*.

- Se, scorrendo il vettore *bin* troviamo un 1 nella posizione $i-esima$, allora aggiorniamo Q con $Q+T$, dove ricordiamo che in T è salvato $2^{length(bin)-i}T$. Per farlo, usiamo la formula di somma richiamata nella sezione 1, controllando però che i denominatori non si annullino tramite il marcatore $flag$.

- A questo punto raddoppiamo T chiamando la funzione $raddoppia(T, n, a)$. Tuttavia, prima di procedere con il calcolo di $k * T$, dobbiamo assicurarci che i denominatori $2y_P$ non si siano annullati e lo facciamo nuovamente controllando il marcatore $flag$.

- Se il ciclo for termina senza interruzioni, vuol dire che il programma è riuscito a calcolare con successo $k * T$ che sarà salvato nella variabile di output Q.