1. Algoritmi probabilistici

Capita a volte di vedersi la strada bloccata da una persona, spostarsi a destra per passare, proprio mentre quella persona si sposta anch'essa nella stessa direzione; allora ci si sposta a sinistra, e di nuovo ci si trova bloccati. Se l'algoritmo seguito è "**if** destra **then** sinistra **else** destra", questo balletto non avrà fine. Ma se invece di fare una scelta *deterministica* ci si affida alla sorte e si sceglie *casualmente* una direzione, la probabilità che la strada sia ancora bloccata dopo k mosse è $(1/2)^k$, un valore che diventa arbitrariamente piccolo con la crescita di k; prima o poi, sarà possibile passare.

Questo aneddoto suggerisce un approccio "audace" alla progettazione di algoritmi, che potrebbe essere riassunto da Virgilio in questo modo:

"Audentes fortuna iuvat" – "La sorte aiuta gli audaci".

Quando non si sa quale opzione prendere di fronte ad una scelta, oppure l'algoritmo per decidere in maniera corretta ha un costo computazionale troppo alto, è possibile affidarsi alla casualità: si sceglie un'opzione a caso.

Abbiamo già incontrato il concetto di probabilità parlando di analisi del caso medio, ove il tempo di calcolo è stato valutato mediando su tutti i possibili dati di ingresso, dopo aver individuato (o per lo più assunto arbitrariamente) una distribuzione di probabilità in accordo alla quale i dati di ingresso possono presentarsi. Per esempio, il caso medio del Quicksort [cfr. Capitolo 12] parte dall'assunzione che tutte le permutazioni di un vettore di n elementi siano equiprobabili.

Nel caso di algoritmi *probabilistici* (detti anche algoritmi *randomizzati*) il calcolo delle probabilità è applicato non tanto ai dati di ingresso, quanto piuttosto ai dati di uscita. In un algoritmo probabilistico, la computazione dipende dai valori prodotti da un generatore di numeri casuali. Si assuma di avere a disposizione una funzione $\mathsf{random}(h,k)$, che quando è richiamata restituisce con la stessa probabilità un qualsiasi intero generato casualmente tra h e k, estremi inclusi.[1]

[1] In pratica la funzione $\mathsf{random}()$ richiede tempo costante ed è realizzata con un generatore di numeri pseudocasuali, cioè da un algoritmo deterministico che fornisce numeri che dal punto di vista statistico si comportano come se fossero veramente casuali.

Esempio (Testa o croce?) random$(0,1)$ restituisce 0 oppure 1 con probabilità $1/2$ ciascuno, come se fosse effettuato il lancio di una moneta e restituita testa oppure croce. ■

Il più semplice utilizzo della funzione random() consiste nel permutare casualmente i dati di ingresso di un problema. In questo modo una distribuzione casuale sui dati d'ingresso viene imposta preventivamente, anziché assunta successivamente nell'analisi di complessità. Vedremo questa soluzione risolvendo il problema della SELEZIONE, in cui si chiede di identificare il k-esimo elemento più piccolo all'interno di un insieme, senza dover ordinare tale insieme.

Un utilizzo più sofisticato, invece, permette di progettare algoritmi che nel risolvere un problema possono fornire un risultato scorretto (o addirittura non fornirne alcuno), ma che risultano tuttavia utilissimi qualora la probabilità che tali eventi si verifichino possa essere resa arbitrariamente piccola.

Esempio ESPRESSIONE POLINOMIALE NULLA. *Data un'espressione algebrica polinomiale $p(x_1,\ldots,x_n)$ in n variabili, determinare se p è identicamente nulla oppure no.* Si osservi che se p fosse dato già in forma semplificata, cioè come somma di monomi, allora il problema si risolverebbe banalmente verificando se esiste oppure no un coefficiente diverso da zero. Se invece p è dato come espressione algebrica arbitraria, allora il migliore algoritmo deterministico noto, basato sul metodo di semplificazione, richiede un tempo di calcolo molto elevato. È possibile definire un algoritmo probabilistico che dapprima genera casualmente una n-pla di valori $v_1,\ldots,v_n$ e poi valuta $p(v_1,\ldots,v_n)$. Se $p(v_1,\ldots,v_n)\neq 0$, allora l'espressione polinomiale è sicuramente non nulla. Altrimenti, se $p(v_1,\ldots,v_n)=0$, c'è la possibilità che l'espressione sia nulla. Indicato con d il grado massimo delle variabili in p, è stato dimostrato che generando ciascun valore v_i tramite random$(1,2d)$, $1\leq i\leq n$, la probabilità di errore non supera $1/2$. Iterando la valutazione di p più volte e assumendo l'espressione nulla se $p(v_1,\ldots,v_n)=0$ per tutte le n-ple generate, la probabilità di errore può essere resa piccola a piacere. ■

Applicheremo tale tecnica ad un problema molto importante, quello della PRIMALITÀ: stabilire se un numero è primo oppure no.

Per concludere, si noti che questi non sono i primi esempi di algoritmi probabilistici del presente libro: la tabelle hash, infatti, si avvalgono di funzioni hash che distribuiscono le chiavi in maniera uniforme.

1.1 Selezione

Dato un vettore A di n interi, la *mediana* è il numero che si troverebbe nella posizione centrale se questi fossero ordinati. A parte una piccola difficoltà nel definire la posizione centrale nel caso di n pari (si sceglie arbitrariamente $\lceil n/2\rceil$ sia nel caso pari che nel caso dispari), è facile vedere che la mediana può essere calcolata in $O(n\log n)$ tempo ordinando il vettore. Ma è possibile domandarsi (i) se questo ordinamento sia necessario e, ancora più importante, (ii) se il tempo $\Omega(n\log n)$ sia necessario.

Ragionando su tali questioni, può essere interessante generalizzare e cercare di risolvere il problema della SELEZIONE.

SELEZIONE (SELECTION). *Dati un vettore A di n interi ed un intero k tale che $1\leq k\leq n$, trovare il k-esimo elemento più piccolo.*

Si noti che per $k=1$ ($k=n$) il problema si riduce a quello di trovare il minimo (massimo), mentre $k=n/2$ corrisponde al mediano.

Per trovare il k-esimo elemento più piccolo senza ordinare il vettore, si può utilizzare la procedura perno() [cfr. Capitolo 12]. Ricordiamo che la procedura perno$(A, primo, ultimo)$ scambia tra loro elementi di $A[primo \dots ultimo]$ e restituisce l'indice j del "perno" tale che $A[i] \leq A[j]$, per $primo \leq i \leq j$, e $A[i] \geq A[j]$, per $j \leq i \leq ultimo$. La seguente procedura selezione(), proposta da Hoare (1961), richiama la perno() ed individua così il q-esimo elemento più piccolo di $A[primo \dots ultimo]$, dove $q = j - primo + 1$ è il numero di elementi di $A[primo \dots j]$. Se $k \leq q$, allora selezione() è riapplicata ad $A[primo \dots j]$; se invece $k > q$, allora selezione() è riapplicata ad $A[j+1 \dots ultimo]$, ma per ricercare il $(k-q)$-esimo elemento più piccolo. La chiamata iniziale è selezione$(A, 1, n, k)$.

```
item selezione(item[] A, int primo, int ultimo, int k)
if primo = ultimo then
    return A[primo]
else
    int j = perno(A, primo, ultimo)
    int q = j - primo + 1
    if k = q then
        return A[j]
    else if k < q then
        return selezione(A, primo, j - 1, k)
    else
        return selezione(A, j + 1, ultimo, k - q)
```

Assumendo che perno() restituisca con la stessa probabilità una qualsiasi posizione j del vettore A, il numero medio di confronti $T(n)$ tra elementi di A effettuati da selezione() è dato dalla relazione di ricorrenza

$$
\begin{aligned}
T(n) &= 0, && \text{per } n = 0, 1 \\
T(n) &= n + \frac{1}{n} \sum_{q=1}^{n} T\left(\max\{q-1, n-q\}\right) \\
&\leq n + \frac{2}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} T(q), && \text{per } n \geq 2
\end{aligned}
$$

Infatti, nel caso n sia pari, è facile vedere che i valori $\lfloor n/2 \rfloor, \lfloor n/2 \rfloor + 1, \dots, n-1$ vengono ottenuti due volte dall'espressione $\max\{q-1, n-q\}$, per $q = 1 \dots n$; se invece n è dispari, i valori $\lfloor n/2 \rfloor + 1, \lfloor n/2 \rfloor + 2, \dots, n-1$ vengono ottenuti due volte, mentre $\lfloor n/2 \rfloor$ viene ottenuto una volta sola – da cui la disequazione.

Effettuiamo un ragionamento informale per cercare di ricavare una buona soluzione sulla quale applicare la tecnica di dimostrazione per tentativi. In media, selezione() dovrebbe essere richiamata ogni volta su una porzione dimezzata del vettore A. Poiché la perno() richiede tempo lineare nella lunghezza della porzione di A, il numero complessivo di confronti dovrebbe crescere come $n + n/2 + n/4 + \dots + 1 \leq 2n$. Pertanto, una soluzione promettente di $T(n)$ con la quale tentare sembra essere $O(n)$.

Tentiamo quindi con $T(n) \leq cn$ per qualche costante c. La base dell'induzione è banalmente verificata perché $0 = T(0) = T(1) \leq c$. Si assuma che valga l'ipotesi induttiva $T(q) \leq cq$ per ogni

$q \leq n-1$. Sostituendo, si ricava:

$$
\begin{aligned}
T(n) &\leq n + \frac{1}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} 2 \cdot cq \leq n + \frac{2c}{n} \sum_{q=\lfloor n/2 \rfloor}^{n-1} q && \text{Sostituzione, raccolgo } 2c \\
&\leq n + \frac{2c}{n} \sum_{q=n/2-1}^{n-1} q && \text{Estensione sommatoria} \\
&= n + \frac{2c}{n} \left(\sum_{q=1}^{n-1} q - \sum_{q=1}^{n/2-2} q \right) && \text{Sottrazione prima parte} \\
&= n + \frac{2c}{n} \cdot \left(\frac{n(n-1)}{2} - \frac{(n/2-1)(n/2-2)}{2} \right) \\
&= n + \frac{\cancel{2}c}{n} \cdot \frac{\left(n^2 - n - \left(1/4\, n^2 - 3/2\, n + 2\right)\right)}{\cancel{2}} \\
&= n + c/n \cdot \left(3/4\, n^2 - 1/2\, n - 2\right) \\
&\leq n + c/n \cdot \left(3/4\, n^2\right) = n + 3/4\, cn \overset{?}{\leq} cn && \text{Vera per } c \leq 4
\end{aligned}
$$

L'ultima disequazione è vera per $c \geq 4$. selezione() trova il k-esimo elemento più piccolo di A in tempo lineare nel caso medio.

Tutta questa analisi è partita dall'assunzione che perno() restituisca con la stessa probabilità una qualsiasi posizione j del vettore A; questo non è vero per particolari casi dell'input, quali ad esempio un vettore già ordinato. Per ovviare a questo problema, è possibile forzare una scelta casuale del valore su cui impostare il perno, scambiando subito prima della chiamata perno() un elemento casuale fra *primo* e *ultimo* con l'elemento $A[primo]$, in questo modo:

$$A[\mathsf{random}(primo, ultimo)] \leftrightarrow A[primo]$$

Gli effetti sfavorevoli dovuti al caso pessimo sono pertanto ridotti e l'analisi della complessità nel caso medio risulta così più affidabile.

Tanto per curiosità, esistono altri algoritmi più sofisticati per la selezione() che richiedono tempo $O(n)$ nel caso pessimo, il primo dei quali è stato quello di Blum-Floyd-Pratt-Rivest-Tarjan (1973).

1.2 Primalità

Un problema interessante che può essere risolto in maniera elegante tramite un algoritmo probabilistico è quello della PRIMALITÀ:

PRIMALITÀ (PRIMALITY). *Dato un numero n, è tale numero primo, cioè non esiste alcun numero diverso da* 1 *e da n stesso che lo divida esattamente?*

Il problema può essere risolto con la seguente procedura deterministica:

```
boolean primo(int n)
    int j = 2
    while j ≤ ⌊√n⌋ do
        if n mod j = 0 then return false
        j = j + 1
    return true
```

La funzione primo() prova a dividere n per ciascun numero j compreso tra 2 e $\lfloor\sqrt{n}\rfloor$, e restituisce come risultato **true** se non è stato trovato alcun j tale che $n \bmod j = 0$, cioè tale che il resto della divisione di n per j sia zero. La sua complessità è $O(n^{1/2})$, che non è polinomiale nella dimensione del problema. Infatti, poiché un numero intero n codificato in base 2 necessita di $d = \log n + 1$ bit, la dimensione del problema è d. La complessità della funzione primo() è pertanto $O(2^d)$, cioè esponenziale nella dimensione del problema (secondo il criterio di costo logaritmico).

L'algoritmo più famoso che risolve PRIMALITÀ è quello probabilistico di Rabin (1980). Tale algoritmo si basa sul concetto di *testimone di compostezza*.

Sia $n > 2$ il numero intero (dispari) in ingresso del quale si vuole verificare la primalità, e sia b un intero compreso tra 1 ed $n-1$. Se n è primo, devono valere le tre proprietà seguenti. In primo luogo, $\mathsf{mcd}(n,b) = 1$ per definizione di numero primo, dove mcd() è il massimo comun divisore. Poi deve essere $b^{n-1} \bmod n = 1$ per il Piccolo Teorema di Fermat.[2] Infine, si ha che $b^2 \bmod n = 1$ se e solo se $b = 1$ oppure $b = n-1$. Combinando queste tre proprietà dopo aver riscritto $n-1 = 2^v m$, dove v è il massimo esponente assegnabile a 2 ed m è dispari, si può dimostrare che se n è primo devono valere le condizioni:

(1) $\mathsf{mcd}(n,b) = 1$;

(2) $b^m \bmod n = 1$ oppure $\exists i, 0 \le i \le v-1 : b^{2^i m} \bmod n = -1$.

L'intero b è un testimone che n è composto (cioè non primo) se almeno una delle condizioni (1) e (2) è falsa. Se b è un testimone che n è composto, allora n è effettivamente un numero composto. Se invece b non testimonia che n è composto, allora n può essere primo oppure composto. La verifica che b sia un testimone di compostezza per n può essere effettuata usando $O(\log n)$ operazioni aritmetiche su numeri interi di $O(\log n)$ bit. Infatti, v ed m si possono calcolare dimezzando $n-1$ per $O(\log n)$ volte. La condizione (1) è verificabile in tempo $O(\log n)$ con il ben noto algoritmo di Euclide. Il calcolo di $b^m \bmod n$ si può svolgere ancora in tempo $O(\log n)$ per esponenziazioni successive di b [cfr. Esercizio 1.1], mentre le potenze $b^{2^i m} \bmod n$ si calcolano sempre con lo stesso tempo per quadrature successive di $b^m \bmod n$.

Rabin ha dimostrato che se n è un numero composto, allora ci sono almeno $\frac{3}{4}(n-1)$ numeri compresi tra 1 ed $n-1$ che ne testimoniano la compostezza. Ciò implica che scegliendo a caso un numero b compreso tra 1 ed $n-1$ e verificando se b testimonia la compostezza di n oppure no, c'è una probabilità inferiore ad $\frac{1}{4}$ di rispondere erroneamente che n è primo quando invece è composto. Questo porta alla progettazione del seguente algoritmo probabilistico.

```
boolean primo(int n)
for j = 1 to K do
    b = random(1, n-1)
    if b testimonia che n è composto then
        return false
return true
```

Il precedente algoritmo ripete la verifica di compostezza su K valori indipendenti e uniformemente distribuiti di b e assume che n sia primo se e solo se nessuno di tali K valori è un testimone della compostezza. Mentre la risposta **false** (cioè che n è composto) è sempre corretta, la risposta **true** (cioè che n è primo) è corretta con probabilità maggiore o uguale a $1 - (\frac{1}{4})^K$. In altri termini, la probabilità di errore, cioè di assumere che n sia primo quando invece è composto, non supera $(\frac{1}{4})^K$. Per $K = 50$, una costante moltiplicativa piccola dal punto di vista computazionale, questa probabilità è talmente piccola da essere di gran lunga inferiore alla probabilità che la risposta sbagliata sia dovuta a guasti occorsi nei circuiti hardware del calcolatore! Inoltre, poiché il valore K è una costante, il numero di operazioni aritmetiche della funzione primo() resta $O(\log n)$.

[2] Il Piccolo Teorema di Fermat è così chiamato per distinguerlo dal più famoso Ultimo Teorema di Fermat.

L'algoritmo di Rabin mostra come la probabilità di errore possa essere resa arbitrariamente piccola iterando il procedimento probabilistico per un opportuno numero di volte (50, nella funzione primo()), permettendo così la risoluzione probabilistica in tempo polinomiale di molti problemi decisionali. Si noti che questa tecnica di riduzione dell'errore non richiede che la probabilità iniziale di errore sia piccola, ma solo che sia minore di $1-\varepsilon$ per qualche $\varepsilon > 0$. Se invece di $\frac{1}{4}$ la probabilità di errore fosse 0.99, basterebbe aumentare opportunamente il numero di iterazioni per ottenere la stessa probabilità $(\frac{1}{4})^{50}$ di errore pur mantenendo la stessa complessità di $O(\log n)$ operazioni aritmetiche!

Per concludere, è interessante notare che fino a qualche anno fa l'approccio probabalistico di Rabin era l'unico conosciuto e non erano noti algoritmi deterministici di costo polinomiale. La situazione è cambiata nel 2002, quando i tre matematici indiani Agrawal, Kayal e Saxena introdussero un algoritmo di complessità polinomiale $O(\log^{12+\varepsilon} n)$. Da allora, enormi sforzi di ricerca hanno portato alla scoperta di innumerevoli varianti, alcune delle quali basate su congetture matematiche non ancora dimostrate. Al momento, la versione migliore sembra essere quella di Pomerance e Lenstra, che ha complessità $O(\log^{6+\varepsilon} n)$. Nonostante la presenza di algoritmi deterministici polinomiali, tuttavia, l'algoritmo di Rabin è ancora il più veloce e il più usato.

1.3 Reality check

Grazie all'algoritmo probabilistico di Rabin, decidere se un intero n è primo oppure composto non è un problema difficile. Al contrario, fattorizzare n esprimendolo come prodotto di numeri primi pare essere molto difficile. La difficoltà di questo problema, però, torna utile nella crittografia digitale a chiave pubblica, ove si sfrutta proprio tale difficoltà per proteggere la segretezza di messaggi scambiati attraverso canali di trasmissione insicuri, ad esempio in Internet.

La crittografia si usa per criptare un messaggio in modo che, anche se intercettato da una spia, non possa essere decifrato, se non da chi possieda la chiave di decodifica. La crittografia a chiave pubblica si applica quando il mittente vuole spedire un messaggio criptato ad un destinatario senza che i due condividano una chiave di codifica/decodifica prestabilita. Attraverso procedimenti correlati, basati sulle stesse proprietà, è anche possibile l'apposizione delle cosiddette firme digitali.

Se Bruno vuole spedire messaggi sicuri deve disporre di una coppia di chiavi: una chiave pubblica P, che può diffondere liberamente, e una corrispondente chiave privata S, che solo lui dovrà conoscere. Quando Alice vuole inviare un messaggio sicuro a Bruno, basta che codifichi il messaggio M attraverso la chiave pubblica di Bruno, ottenendo $C = P(M)$. Bruno, per riottenere il messaggio originale, non deve fare altro che codificare nuovamente il messaggio C ricevuto usando S, in questo modo riotterrà $M = S(C)$. Se qualcuno intercettasse il messaggio spedito da Alice non potrà conoscerne il contenuto, in quanto per decifrare il messaggio si deve conoscere S e solo Bruno conosce S. Affinché il sistema sia corretto deve valere $M = S(P(M))$ per ogni M. Affinché sia anche sicuro deve essere molto difficile risalire ad S nota P.

Il procedimento per apporre una firma digitale ad un documento D è analogo, con la differenza che stavolta lo si firma codificandolo per mezzo della chiave privata S, ottenendo così $S(D)$, e lo si riconosce per autentico decodificandolo tramite la chiave pubblica P, riottenendo $D = P(S(D))$.

Consideriamo il sistema crittografico a chiave pubblica noto come RSA, dalle iniziali degli inventori Rivest, Shamir e Adleman (1978), che utilizza l'algoritmo probabilistico di Rabin per generare in tempo polinomiale una coppia di chiavi P ed S (pubblica e privata). Il sistema si basa proprio sull'asimmetria tra la semplicità nel poter generare grandi numeri primi e la difficoltà nel trovare una fattorizzazione di grandi numeri interi.

Nel sistema RSA, ogni chiave è una coppia di interi: $P = (N, p)$ ed $S = (N, s)$. N ha circa 200 cifre decimali, mentre p e s circa 100 (questi sono valori indicativi, poiché al crescere del numero delle cifre cresce il livello di sicurezza offerto dal sistema, ma ad oggi sono consigliate chiavi di circa 1024-2048 bit). Assumiamo che il messaggio da trasmettere consista di una stringa binaria b e che

M sia il corrispondente numero intero rappresentato in binario da b, e che considerazioni analoghe valgano per il messaggio criptato C. La codifica e la decodifica sono operazioni semplicissime. Infatti, $C = P(M)$ si calcola come $C = M^p \bmod N$ mentre $M = S(C)$ si calcola come $M = C^s \bmod N$. Queste operazioni si possono eseguire in tempo polinomiale (si veda l'Esercizio 1.1).

Per ottenere p e s, si procede nel modo seguente. Si generano casualmente tre numeri primi s, x e y di 100 cifre (ciò è fattibile in tempo polinomiale usando la funzione random() per generare interi e poi l'algoritmo di Rabin per verificare se tali interi sono primi o composti). Indi si ricava $N = xy$, mentre p deve essere tale che $ps \bmod (x-1)(y-1) = 1$. Anche questa operazione è eseguibile in tempo polinomiale (si veda l'Esercizio 1.3). Si può dimostrare che, per ogni M, vale $M^{ps} \bmod N = M$, cioè che $S(P(M)) = M$. Per decifrare una codifica conoscendo $P = (N, p)$ bisogna ricavare s, ovvero risalire ad x e y. Ma anche sapendo che $xy = N$, questo resta un problema complicato perché la fattorizzazione di un numero in fattori primi pare essere un problema intrattabile. Per garantire la sicurezza, però, occorrono numeri di circa 100-200 cifre, poiché la fattorizzazione di interi piccoli non è un problema intrattabile. Comunque, finora non è stato provato formalmente alcun risultato generale: né che la fattorizzazione sia un problema risolubile in tempo polinomiale né che sia un problema inerentemente difficile [cfr. Capitolo 18].

1.4 Esercizi

Esercizio 1.1 (Potenze)**.** Si dimostri che l'elevamento a potenza $x^v \bmod m$ si può effettuare in tempo polinomiale, dove x, v ed m sono interi.

Esercizio 1.2 (Algoritmo esteso di Euclide)**.** Si dimostri che, dati s ed m interi, l'equazione $sp + mq = \mathsf{mcd}(s,m)$, si risolve in tempo polinomiale con una modifica dell'algoritmo di Euclide.

Esercizio 1.3 (Inverso moltiplicativo)**.** Si dimostri che, dati tre numeri primi s, x ed y, l'equazione $ps \bmod (x-1)(y-1) = 1$ nell'incognita p è risolvibile in tempo polinomiale usando l'algoritmo esteso di Euclide (vedi Esercizio 1.2).

Esercizio 1.4 (Prodotto di matrici)**.** Date tre matrici A, B e C, quadrate $n \times n$, si vuole decidere se $AB = C$. Si fornisca un algoritmo probabilistico di complessità $O(n^2)$ per verificare se $AB = C$ con probabilità di errore al più $1/2$ (Suggerimento: si generi un vettore x di n elementi uguali a -1 oppure $+1$ e si verifichi se $A(Bx) \neq Cx$).

Esercizio 1.5 ($\frac{1}{2}$-Soddisfattibilità)**.** Data una formula booleana, si vuole verificare se la formula è soddisfatta da più della metà dei possibili assegnamenti dei valori di verità alle variabili booleane. Si fornisca un algoritmo probabilistico.

1.5 Soluzioni

Soluzione Esercizio 1.1

Assumiamo che x, v ed m siano rappresentati in binario usando d bit. Scomponiamo v come somma di potenze di 2, con una scansione della rappresentazione binaria di v, ottenendo $v = 2^{i_1} + \ldots + 2^{i_k}$, dove $i_1 < \cdots < i_k$. Calcoliamo le potenze $x^{2^j} \bmod m$ per $j = 1, 2, \ldots, i_k$, dove ciascuna è ricavata facendo il quadrato della precedente. Infine si ottiene $x^v \bmod m$ effettuando il prodotto modulo m tra le potenze: $x^{2^{i_1}} \bmod m \times \cdots \times x^{2^{i_k}} \bmod m$. Il numero di moltiplicazioni è $O(\log v) = O(d)$, quindi lineare nella dimensione del problema. Se inoltre vogliamo anche contare il numero totale di operazioni tra bit, dato che ciascuna moltiplicazione ne richiede $O(d^2)$ (in pratica, come moltiplicare due polinomi di d coefficienti), l'intero algoritmo effettua $O(d^3)$ operazioni tra bit.

Soluzione Esercizio 1.2
Ricordiamo che, assunti $s > 0$, $m \geq 0$, ed $s \geq m$, l'algoritmo di Euclide calcola $\mathsf{mcd}(s,m)$ in tempo $O(\log m)$ tramite la ricorrenza:

$$\mathsf{mcd}(s,m) = \begin{cases} s & \text{se } m = 0 \\ \mathsf{mcd}(m, s \bmod m) & \text{altrimenti} \end{cases}$$

Il seguente algoritmo ricorsivo euclideEstesoriceve in input s ed m e restituisce in tempo $O(\log m)$ la terna $(\mathsf{mcd}(s,m), p, q)$ che verifica l'equazione $sp + mq = \mathsf{mcd}(s,m)$:

$\mathsf{euclideEsteso}(s,m) = (s,1,0)$, se $m = 0$;
$\mathsf{euclideEsteso}(s,m) = (d, p, r - \lfloor \frac{s}{m} \rfloor r)$, altrimenti,
dove $(d,p,r) = \mathsf{euclideEsteso}(m, s \bmod m)$.

Soluzione Esercizio 1.3
Innanzitutto, che la soluzione p (detta *inverso moltiplicativo*) esista e sia unica è garantito dal fatto che per costruzione s è un numero primo minore di $(x-1)(y-1)$. Ponendo $m = (x-1)(y-1)$, l'equazione $ps \bmod m = 1$ è equivalente all'equazione $sp = mz + 1$ per un valore opportuno di z. Inoltre, ponendo $q = -z$, si ottiene l'equazione $sp + mq = \mathsf{mcd}(s,m)$ poiché, essendo s ed m primi tra loro, il loro massimo comun divisore è proprio uguale ad 1. Quindi l'equazione di partenza è un caso particolare dell'equazione generale $sp + mq = \mathsf{mcd}(s,m)$, la cui risoluzione è stata discussa nell'Esercizio 1.2.