PIANO LAUREE SCIENTIFICHE

LABORATORIO DI COMPLESSITÀ COMPUTAZIONALE

LEZIONE 3

GIORGIO GAMBOSI
DIPARTIMENTO DI MATEMATICA

UNIVERSITÀ DI ROMA "TOR VERGATA"

1. Trattabilità di problemi computazionali

L'esistenza di problemi indecidibili restringe la possibilità di progettare algoritmi e programmi ai soli problemi decidibili, quelli per cui esistono algoritmi di soluzione. In questo ambito, però, non tutti i problemi risultano risolvibili in tempo ragionevole.

1.1. **Un problema intrattabile: le torri di Hanoi.** Il problema delle **Torri di Hanoi** può essere formulato nel modo seguente: sono dati tre pioli di cui il primo contiene n dischi di diverso diametro impilati in ordine di diametro decrescente, con il disco più ampio in basso e quello più stretto in alto, mentre gli altri due pioli sono vuoti. Si vogliono spostare tutti i dischi, uno alla volta, dal primo al terzo piolo senza mai porre un disco più piccolo al di sopra di uno piİ grande.

Come si potrebbe fare?

Soluzione: La soluzione di questo gioco è semplice da descrivere usando un ragionamento di tipo ricorsivo. Supponiamo di avere già spostato i primi $n-1$ dischi sul secondo piolo, usando il terzo come appoggio. Possiamo ora spostare il disco più grande dal primo al terzo piolo, e quindi ricorsivamente spostare gli $n-1$ dischi dal secondo al terzo piolo usando il primo come appoggio. Naturalmente, se $n = 1$, dobbiamo solo spostare l'unico disco dal primo al terzo piolo.

```
SpostaDischi( n, da, a, appoggio )
  IF (n = 1) {
    sposta disco da piolo di origine a piolo destinazione;
  } ELSE {
    SpostaDischi( n - 1, da, appoggio, a );
    sposta disco da piolo di origine a piolo di appoggio;
    SpostaDischi( n - 1, appoggio, a, da );
  }
```

Quanti spostamenti dobbiamo fare?

Soluzione: Possiamo mostrare, ragionando per induzione sul numero n di dischi, che il numero di spostamenti di dischi effettuati dal programma, è pari a $2^n - 1$: se $n = 1$ allora è immediato che serve un solo spostamento e infatti $1 = 2^1 - 1$; nel caso $n > 1$ assumiamo che occorrano $2^{n-1} - 1$ mosse per ciascuno dei due spostamenti di $n - 1$ dischi (dal primo al secondo piolo e poi dal secondo al terzo), a cui aggiungiamo la mossa di spostamento dell'n-esimo piolo dal primo al terzo piolo, per un totale di $2 \times (2^{n-1} - 1) + 1 = 2^n - 1$ mosse.

Purtroppo, non c'è speranza di trovare un programma che effettui un numero di mosse inferiore a tale quantità, in quanto è stato dimostrato che le $2^n - 1$ mosse sono necessarie e non è possibile impiegarne di meno.

Originariamente, il problema delle Torri di Hanoi fu proposto con $n = 64$: in questo caso, supponendo che ogni mossa richieda un secondo, occorrono

$$2^{64} - 1 = 18\ 446\ 744\ 073\ 709\ 551\ 615$$

secondi, che equivalgono a circa 584 942 417 355 anni, ovvero quasi 585 miliardi di anni: si ricordi come l'Universo ha una età stimata di circa 14 miliardi di anni.

Le Torri di Hanoi mostrano dunque che, anche se un problema è decidibile, non è detto che l'algoritmo stesso possa sempre risolverlo in tempi ragionevoli: ciò è dovuto al fatto che il numero di mosse e quindi il tempo di esecuzione del programma, è *esponenziale* nel numero n di dischi (n appare all'esponente di $2^n - 1$). Il tempo necessario per spostare i dischi diventa dunque rapidamente insostenibile, anche per un numero limitato di dischi, come illustrato nella seguente tabella, in cui il tempo di esecuzione è espresso in secondi (s), minuti (m), ore (h), giorni (g) e anni (a).

n	5	10	15	20	25	30	35	40	45
tempo	32 s	17 m	9 h	12 g	1 a	34 a	1089 a	34.865 a	1.115.689 a

L'esponenzialità del tempo di esecuzione rende anche limitato l'effetto di eventuali miglioramenti nella velocità di esecuzione dei singoli passi, perchè in questo caso basta aumentare di poco il numero n di dischi per vanificare ogni miglioramento. Supponiamo infatti che un avanzamento tecnologico ci consenta ora di eseguire $m = 2^s$ operazioni in un secondo, invece di una sola: in tal caso, per spostare gli n dischi, anziché 2^n secondi, ne occorreranno $2^n/m = 2^{n-s}$. L'effetto di questo miglioramento viene però neutralizzato molto rapidamente al crescere del numero di dischi in quanto è sufficiente portare tale numero a $n + s$ per ottenere lo stesso tempo complessivo di esecuzione. In altre parole, un miglioramento delle prestazioni per un fattore *moltiplicativo* si traduce in un aumento solo *additivo* del numero di dischi trattabili nello stesso tempo. La tabella seguente esemplifica questo comportamento nel caso $n = 64$, mostrando il numero di dischi gestibili in un tempo pari a 18 446 744 073 709 551 615 secondi (più di 500 miliardi di anni), al variare della velocità di esecuzione: come possiamo vedere, miglioramenti anche molto importanti di quest'ultima si traducono in piccoli incrementi del numero di dischi che si possono spostare.

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero dischi	64	67	70	73	77	80	83	93

1.2. **Un problema probabilmente intrattabile: i rissosi cavalieri della tavola rotonda.** Consideriamo ora il caso di un problema che ci risulta intrattabile, anche se non possiamo essere certi di ciò. Il problema che consideriamo è quello della disposizione a tavola (rotonda) dei prodi e rissosi cavalieri di re Artù: i cavalieri in questione, anche se coraggiosi e fedeli al loro re, non sono tutti amici tra loro, ma accade spesso che coppie di cavalieri siano nemici da tenere lontani. Quel che Artù vuole è disporre i suoi n cavalieri intorno alla tavola rotonda (che ha n seggi intorno), in modo tale che due cavalieri seduti vicini non siano nemici.

Come si potrebbe fare?

Soluzione: Artù non sembra avere molto di meglio da fare che provare tutte le possibili disposizioni di cavalieri intorno al tavolo: per ogni disposizione, se due cavalieri nemici si ritrovano vicini la disposizione è scartata (e si passa a considerare la successiva) mentre se tutti i cavalieri vicini risultano amici la disposizione fornisce una soluzione al problema, e la ricerca termina.

Quante operazioni deve eseguire Artù?

Soluzione: Fissata una disposizione degli n cavalieri, Artù deve fare n verifiche per controllare che ogni coppia di cavalieri vicini non sia nemica (il primo cavaliere con il secondo, il secondo con il terzo, ..., l'ultimo con il primo). Quante disposizioni deve considerare Artù? Qui dipende da quanto è fortunato: chiaramente, se tutto va al meglio, la prima disposizione considerata va già bene, per cui deve effettuare in tutto n verifiche.

Noi siamo però più interessati a capire quante operazioni devono essere eseguite nel caso peggiore, in quanto questo ci fornisce un valore che non potremo in nessun caso superare. Il caso peggiore, nel problema di Artù, è ...

Qual è il caso peggiore per Artù?

Il caso peggiore per Artù è quello in cui deve esaminare tutte le disposizioni, il che può avvenire nel caso in cui non sia possibile disporre i cavalieri intorno alla tavola oppure nel caso in cui l'unica disposizione accettabile è l'ultima considerata.

Quante sono in tutto le disposizioni?

Il numero di disposizioni è derivabile nel modo seguente: nel determinare una disposizione da considerare, Artù può scegliere tra n cavalieri riguardo a chi considerare nella posizione 1; fissato il cavaliere al seggio 1, può scegliere tra gli altri $n-1$ cavalieri riguardo a chi considerare al seggio 2 (questo ci dà $n \cdot (n-1)$ possibili coppie di cavalieri ai primi due seggi); fissati i cavalieri ai seggi 1 e 2, può scegliere tra gli altri $n-2$ cavalieri riguardo a chi considerare al seggio 3, e così via. Alla fine, risultano $n \cdot (n-1) \cdot (n-2) \ldots 3 \cdot 2$ possibilità e quindi lo stesso numero di disposizioni. Questo numero prende il nome di *fattoriale* e si indica come $n!$.

In realtà Artù può considerare meno disposizioni, osservando che due disposizioni in cui i cavalieri sono disposti nello stesso ordine relativo (i cavalieri vicini sono gli stessi), rappresentano il medesimo caso per Artù. Data una disposizione, ce ne sono altre $n-1$ equivalenti ad essa da questo punto di vista, per cui in tutto Artù deve considerare $n!/n = (n-1)!$ casi diversi, effettuando n verifiche ciascuno, per un totale di $n!$ verifiche.

Nella tabella vediamo quante verifiche deve eseguire Artù al crescere del numero dei suoi cavalieri, e quanto tempo gli occorre, assumendo ad esempio che possa eseguire una verifica al secondo.

n	5	10	15	20	25	30	35	40	45
verifiche	120	3628800	1.3×10^{12}	2.4×10^{18}	1.5×10^{25}	2.6×10^{32}	1.0×10^{40}	8.1×10^{47}	1.2×10^{56}
tempo	2 m	42 g	41000 a	7×10^{10} a	5×10^{17} a	8×10^{24} a	3×10^{32} a	2×10^{40} a	3×10^{48} a

Ricordiamo come l'età dell'universo sia stimata pari a 1.4×10^{10} anni. Il tempo di risoluzione risulta in effetti (più che) esponenziale, in quanto è possibile mostrare che vale la cosiddetta approssimazione di Stirling (dove $e = 2.71828\ldots$)

$$n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

Contrariamente al caso delle torri di Hanoi, però, ora non sappiamo se non esista qualche modo più efficiente per Artù per risolvere il problema: tutto quello che abbiamo mostrato è che un particolare metodo (algoritmo) di soluzione richiede tempi unaccettabili. Potrebbero però esistere altri algoritmi, più efficienti, che magari permettono di ottenere una soluzione molto più rapidamente. Dobbiamo quindi distinguere tra la difficoltà *intrinseca* di risoluzione di un problema, come nel caso delle torri di Hanoi, e l'efficienza di un particolare algoritmo per la risoluzione del problema: naturalmente, un problema intrattabile non può essere risolto da nessun algoritmo efficiente, mentre un problema per cui abbiamo soltanto algoritmi non efficienti potrebbe magari essere risolto efficientemente da un nuovo algoritmo, che non conosciamo al momento.

1.3. **Un problema chiaramente trattabile: ordinamento.** Il problema di ordinare gli elementi in un insieme ci fornisce un esempio di problema trattabile. Supponiamo che re Artù, stanco di cercare una disposizione dei cavalieri intorno alla tavola rotonda che impedisca a cavalieri nemici di sedere vicini, imponga di autorità che i cavalieri vicini non possano litigare e sfidarsi a duello e, a quel punto, decida di disporre i cavalieri in ordine di anzianità di servizio (il più anziano al seggio 1, il secondo al seggio 2, fino al più novellino al seggio n).

Come si potrebbe fare?

Soluzione: Ad esempio, Artù potrebbe procedere in questo modo: ricercare il cavaliere più anziano tra tutti gli n e disporlo al seggio 1. A questo punto, ricercare il più anziano tra i rimanenti $n-1$ e disporlo al seggio 2, e così via, ricercando il cavaliere da disporre al seggio i (l'i-esimo in ordine di anzianità di servizio) come il più anziano tra gli $n-i+1$ cavalieri rimanenti dopo aver individuato ed estratto dall'insieme gli $i-1$ più anziani.

Artù procederebbe quindi nel modo seguente:

```
S = {insieme di tutti gli n cavalieri};
FOR (i = 1; i ≤ n; i = i + 1) {
   ricerca in S il cavaliere più anziano tra i rimanenti n − i + 1;
   assegna al cavaliere l'i-esimo seggio;
   elimina dall'insieme il cavaliere trovato
}
```

Quante operazioni deve eseguire Artù?

Soluzione: Artù deve ricercare, uno dopo l'altro, $n-1$ cavalieri (l'ultimo che rimane è chiaramente il più giovane). Per cercare il primo, deve confrontare l'anzianità di servizio di tutti e n i cavalieri (con $n-1$ confronti), per il secondo di $n-1$ cavalieri (tutti meno il più anziano, con $n-2$ confronti), per il terzo di $n-2$ e così via, fino al penultimo (l'$n-1$-esimo) per il quale ne deve esaminare l'anzianità di 2 cavalieri.

Quindi, Artù deve fare un numero di confronti pari a

$$(n-1)+(n-2)+\ldots+3+2+1=\frac{n(n-1)}{2}=\frac{n^2}{2}-\frac{n}{2}$$

Esaminiamo i numero di confronti effettuati da Artù al crescere del numero di cavalieri, insieme al tempo necessario per effettuarli, sempre nell'ipotesi che un confronto richieda 1 secondo.

n	5	10	15	20	25	30	35	40	45
confronti	10	45	105	190	300	435	595	780	990
tempo	10 s	45 s	105 s	3 m	5 m	7 m	10 m	13 m	16.5 s

Come si può vedere, nel problema di ordinare i cavalieri il numero di passi (confronti) da eseguire risulta molto inferiore se confrontato con il caso della disposizione introno alla tavola o anche delle torri di Hanoi: possiamo osservare, in particolare, come il numero di passi sia legato al numero di cavalieri da una relazione polinomiale, e non esponenziale. In generale, il passaggio da un andamento esponenziale a uno polinomiale ha due importanti conseguenze. In primo luogo, un polinomio cresce molto più lentamente di una qualunque funzione esponenziale, come si è potuto vedere nelle tabelle presentate sopra. In secondo luogo, la polinomialità del tempo di esecuzione rende molto più efficaci gli eventuali miglioramenti nella velocità di esecuzione dei singoli passi. Ad esempio, nel caso dell'ordinamento dei cavalieri, potendo eseguire m operazioni in un secondo, invece di una sola, occorrerebbero un numero di secondi pari a

$$\frac{n(n-1)}{2m}\simeq\frac{1}{2}\frac{n}{\sqrt{m}}\left(\frac{n}{\sqrt{m}}-1\right)$$

Quindi, un aumento delle prestazioni di un fattore m permette, nello stesso tempo, di ordinare circa $n\sqrt{m}$ cavalieri.

In altre parole, un miglioramento di un fattore *moltiplicativo* nelle prestazioni si traduce in un aumento anch'esso *moltiplicativo* del numero di cavalieri. La tabella seguente (analoga a quella vista nel caso della funzione 2^n) esemplifica questo comportamento nel caso $n=64$, mostrando il numero di cavalieri ordinabili in un tempo pari a 1 ora, al variare della velocità di esecuzione: come possiamo vedere, miglioramenti di quest'ultima si traducono in

incrementi significativi del numero di cavalieri che si possono ordinare nella stessa quantità di tempo.

operazioni/sec	1	10	100	10^3	10^4	10^5	10^6	10^9
numero cavalieri	85	268	848	2.683	8.485	26.832	84.852	2.683.282

Tranne che per piccole quantità di dati, un algoritmo che impiega un numero di passi esponenziale è impossibile da usare quanto un algoritmo che non termina. Con il termine *algoritmo polinomiale* si indica un algoritmo, per il quale esiste una costante $c > 0$, il cui numero di passi elementari sia al massimo pari a n^c per ogni dato in ingresso di dimensione n. Questa definizione porta a una classificazione dei problemi computazionali in cui, oltre alla divisione in problemi indecidibili e decidibili, abbiamo l'ulteriore suddivisione di quest'ultimi in *problemi trattabili* (per i quali esiste un algoritmo risolutivo polinomiale) e *problemi intrattabili* (per i quali un tale algoritmo non esiste).