Problemi trattabili

"OK, mi sono sbagliato! Un calcolatore non può risolvere tutti i problemi computazionali. Però, tutti i problemi computazionali risolvibili sono risolvibili con algoritmi "efficienti"!"

E' vero? Ci crediamo?

Ancora un po' di esempi per ragionare insieme.

Algoritmi ricorsivi (informale, ci ritorneremo ampiamente...)

Un algoritmo è ricorsivo se è "espresso in termini di se stesso". L'esecuzione comporta la semplificazione o suddivisione dell'insieme dei dati (INPUT) e l'applicazione dello stesso algoritmo agli insiemi di dati più piccoli.

All'inizio gli algoritmi ricorsivi "lasciano scettici". Sembrano non risolvere il problema, un trucco, non convincono...

Esempio classico: fattoriale

$$f : N \to N \qquad n \longrightarrow n \cdot (n-1) \cdot \ldots \cdot 1 = n!$$

Usando un linguaggio di programmazione che permette all'utente di scrivere algoritmi (programmi) ricorsivi,

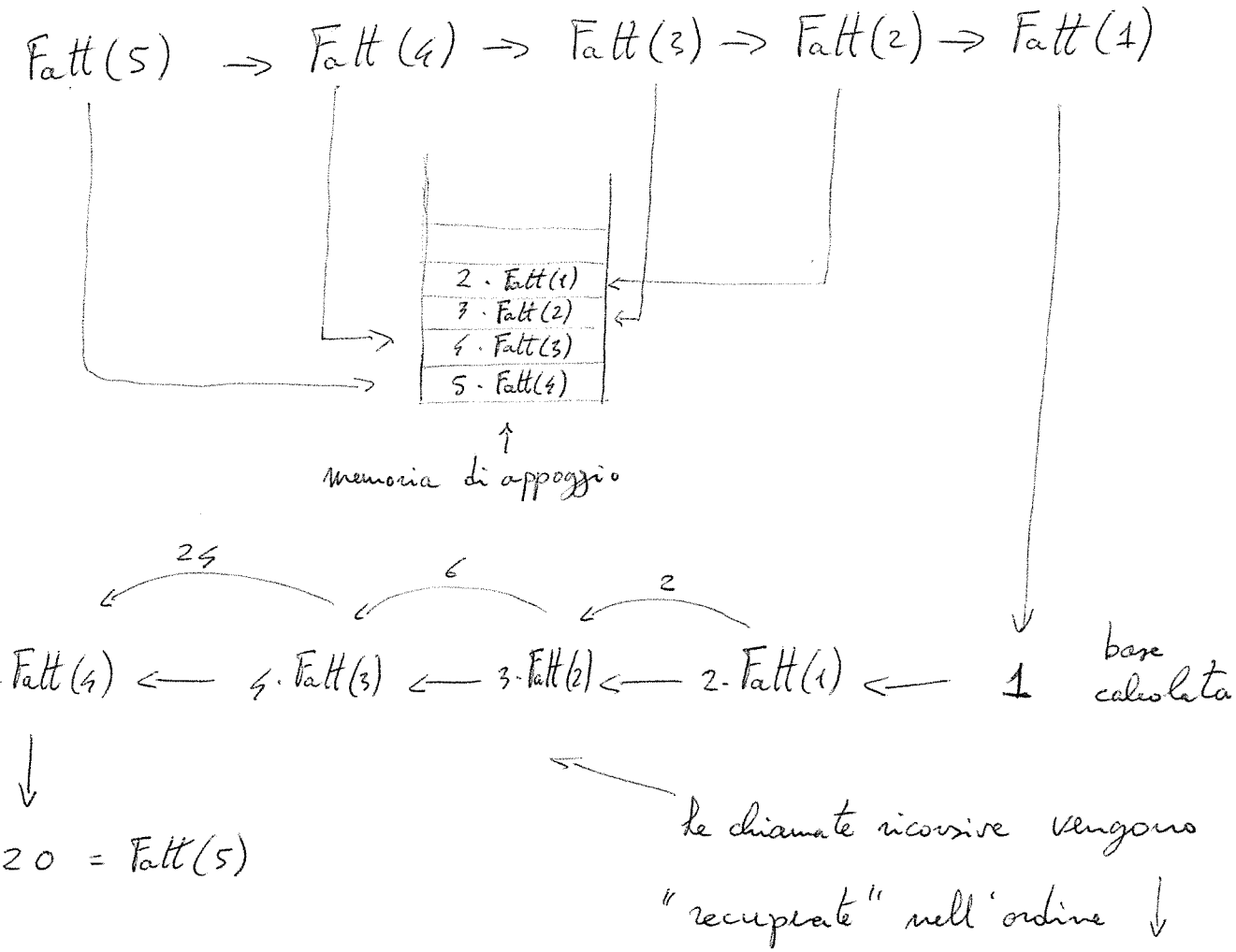
↓

(il linguaggio di programmazione usa una memoria di appoggio per memorizzare temporaneamente le chiamate ricorsive e "ricompone" i risultati da esse prodotti)

la funzione fattoriale può essere calcolata attraverso il seguente algoritmo ricorsivo

```
Fatt (n)
if (n == 1) return 1;        ---> base della ricorsione
else return n · Fatt(n-1);   ---> semplificazione e chiamata ricorsiva
```

Esempio: l'esecuzione di Fatt(5) comporta:

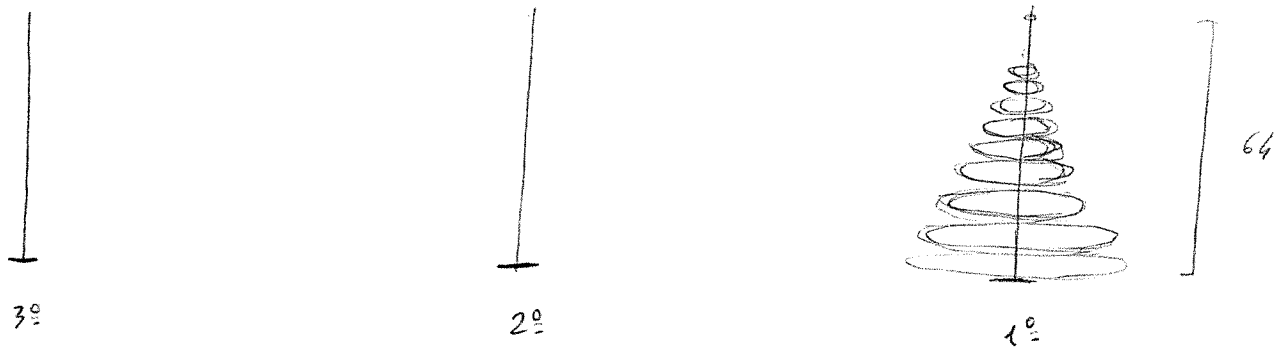


Problema delle Torri di Hanoi

Gioco del XIX secolo introdotto dal matematico francese Eduard Lucas.

Si rifà ad una leggenda indiana sulla divinità Brahma e sulla fine del mondo.

In un tempio induista: 3 pioli, il primo con 64 dischi d'oro impilati in ordine di diametro decrescente. Gli altri due sono vuoti. Precisamente:



Dei monaci sannyasin spostano i dischi dal primo al terzo piolo, usando il secondo come appoggio.

Regole:

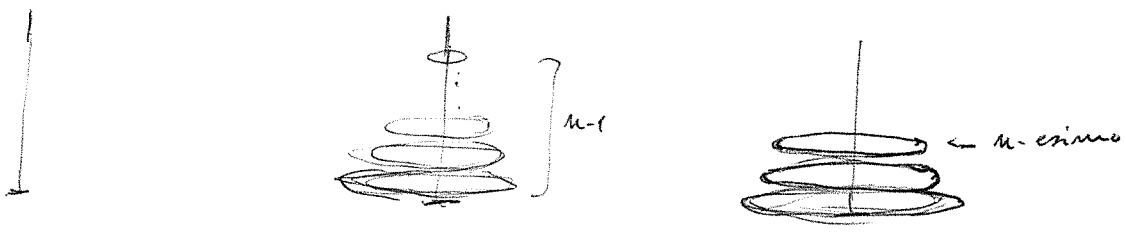
- un disco alla volta
- mai un disco di diametro maggiore su uno di diametro minore

Quando i monaci avranno finito di spostare i dischi avverrà la fine del mondo.

La soluzione del gioco è semplice usando un algoritmo ricorsivo. Idea:

Supponiamo di:

- aver spostato i primi $n-1$ dischi sul 2° piolo, usando il terzo come appoggio, cioè:



Allora:

- spostiamo l'n-esimo disco dal 1° al 3° piolo
- spostiamo gli n-1 dischi dal 2° piolo al 3° piolo usando il primo come appoggio

In codice, ciò si traduce in:

```
TORREHANOI ( n, PRIMO, SECONDO, TERZO)
                ↓      ↓       ↓
               DAL   APPOGGIO  AL

IF (n == 1) {
   SPOSTA   PRIMO ⟶ TERZO;
}
ELSE {
      TORREHANOI (n-1, PRIMO, TERZO, SECONDO) ;
                    ↓      ↓       ↓
                   DAL   APPOGGIO  AL

      SPOSTA   PRIMO → TERZO;

      TORREHANOI (n-1, SECONDO, PRIMO, TERZO) ;
                    ↓       ↓      ↓
                   DAL   APPOGGIO  AL
```

Sviluppiamo qualche caso, per n piccolo, per acquisire familiarità con l'algoritmo.

Esempio $n = 2$ PRIMO = P, SECONDO = S, TERZO = T

TORRE HANOI (2, P, S, T)

↓

TORREHANOI (1, P, T, S) ⟶ SPOSTA P → S

SPOSTA P → T

TORREHANOI (1, S, P, T) ⟶ SPOSTA S → T

Esempio $n = 3$

TORRE HANOI (3, P, S, T)

↓

TORREHANOI (2, P, T, S) ⟶ TORREHANOI (1, P, S, T) ⟶ SPOSTA P→T; (1)

SPOSTA P → S; (2)

TORREHANOI (1, T, P, S) ⟶ SPOSTA T→S; (3)

SPOSTA → P → T (4)

TORREHANOI (2, S, P, T) ⟶ TORREHANOI (1, S, T, P) ⟶ SPOSTA S→P; (5)

SPOSTA S → T; (6)

TORREHANOI (1, P, S, T) ⟶ SPOSTA P→T (7)

Osservazione. Numero di spostamenti (passi algoritmo essenzialmente)

$n = 2 \longrightarrow 2^2 - 1 = 3$

$P \to S$;
$P \to T$;
$S \to T$;

$n = 3 \longrightarrow 2^3 - 1 = 7$

$P \to T$;
$P \to S$;
$T \to S$;
$P \to T$;
$S \to P$;
$S \to T$;
$P \to T$;

(Se ancora non siete convinti, controllate "empiricamente" da soli/e, usando tre carte!)

Possiamo dimostrare che l'algoritmo impiega $2^n - 1$ mosse per spostare n dischi.

Per induzione. Base $n = 1$. $2^n - 1 = 2^1 - 1 = 1$.

Vera in quanto l'algoritmo sposta l'unico disco con una mossa.

Ipotesi induttiva. Supponiamo sia vero per spostare $n-1$ dischi.

Passo induttivo. Guardando il codice è immediato vedere che occorrono per n dischi:

$$\underbrace{(2^{n-1} - 1)}_{\text{due chiamate di TORREHANOI}(n-1, \dots)} + 1 + (2^{n-1} - 1) = 2 \cdot 2^{n-1} - 1 + 1 - 1$$

$$= 2^n - 1 \; ;$$

Purtroppo è stato dimostrato che $2^n - 1$ mosse sono necessarie. Non è possibile progettare un algoritmo risolutivo che richiede meno passi.

Per quanto ci riguarda, possiamo affermare con certezza che esistono problemi risolvibili ma INTRATTABILI, che ammettono cioè soltanto algoritmi che richiedono un numero di passi esponenziale.

Nota: $2^{64} - 1$ equivale a circa 585 miliardi di anni, assumendo che una mossa (spostamento di un disco) richieda un secondo.

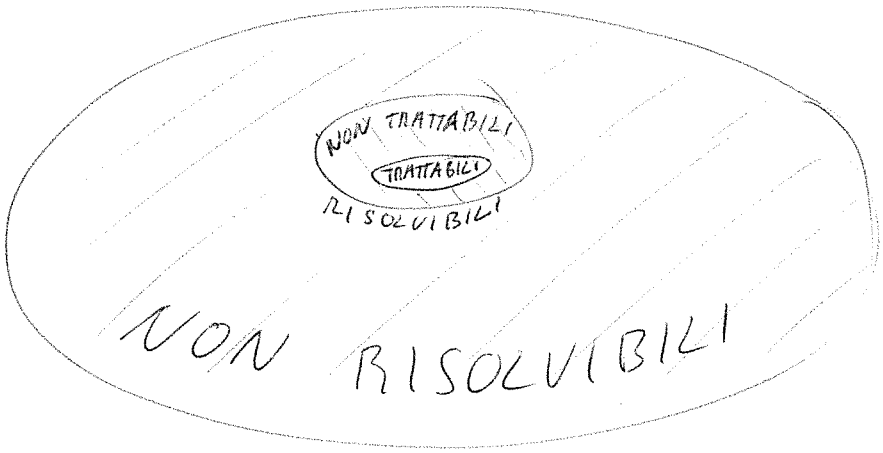
La teoria del Big Bang assicura che l'Universo ha una vita che varia tra i 10 e i 20 miliardi di anni.

Esempi di tempi di esecuzione per n "piccolo"

n	5	10	15	20	25	30	35	40
tempo	32 s	17 m	9 h	12 g	1 a	34 a	1089 a	38865 a

Sorprendentemente, si può far vedere che la versione del problema delle Torri di Hanoi con più pioli ($k > 3$) è risolvibile attraverso algoritmi "efficienti".

In conclusione, abbiamo ancora raffinato la nostra visione dell'universo dei problemi computazionali, che è al momento all'incirca la seguente



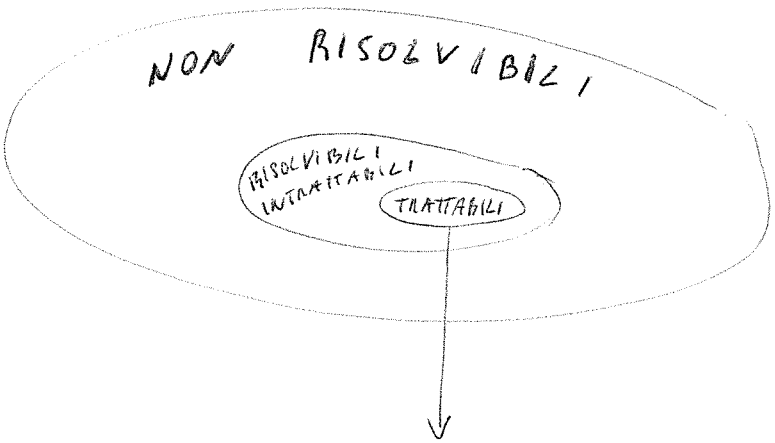
Riferimenti: materiale estratto dalla precedente versione del libro di testo, capitolo introduttivo.

Algoritmi polinomiali ed esponenziali

Tranne che per problemi computazionali di piccola taglia, un algoritmo che impiega un numero esponenziale di passi è

INUTILIZZABILE

Algoritmo polinomiale: 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 input di dimensione n.



$\exists$ un algoritmo risolutivo polinomiale

In questo corso focalizzeremo la nostra attenzione su problemi trattabili. Prima di addentrarci, però, raffiniamo ulteriormente la nostra visione dell'universo dei problemi computazionali.

Esiste una "zona grigia" localizzata tra i problemi trattabili e quelli intrattabili.

SUDOKU

(SU = GIOCO DOKU = SOLITARIO)

3	9							8
	7	1			3			
		8			9		6	
1								9
6								3
5								4
	4		1	5		9		
			9			8	2	
9							4	7

Giappone, 1984

USA 1979
"Number Place"

Scopo del gioco: riempire le caselle con numeri tra 1 e 9 in modo tale che

- ogni riga contenga tutti i numeri compresi tra 1 e 9
- ogni colonna contenga tutti i numeri compresi tra 1 e 9
- ogni sotto-tabella contenga tutti i numeri compresi tra 1 e 9

Nelle configurazioni "semplici" del gioco, il giocatore, prestando attenzione, può identificare univocamente il contenuto della casella (es. (7, 8))

Nelle configurazioni "complesse" non è così.

Si procede per "tentativi ed errori". Si segue un'ipotesi, assegnando uno tra i possibili valori alla casella che

si sta considerando e si procede.
Se ad un certo punto ci si trova in un vicolo cieco, si "torna indietro", svuotando le caselle precedentemente assegnate. Algoritmi di questo tipo si dicono algoritmi di "backtracking" (e saranno oggetto di corsi più avanzati di algoritmi, non di questo).

Tuttavia, allo scopo di avere un'idea più chiara di come una configurazione complessa possa essere risolta, formalizziamo la precedente strategia in una funzione ricorsiva che chiameremo SUDOKU.

Supporremo che esistano tre procedure di appoggio che chiameremo

PrimaVuota ()



restituisce la prima casella vuota nella scacchiera

SuccVuota()



restituisce la casella vuota successiva a quella che si sta considerando

UltimaVuota ()

restituisce un valore booleano T/F che dice se la casella in questione è l'ultima vuota nella scacchiera.

Le tre procedure scorrono la scacchiera attraverso un ordine prestabilito (e.g. dalla prima riga verso l'ultima).

```
casella = PrimaVuota();

SUDOKU (casella)

  elenco = { insieme cifre ammissibili per casella };

    FOR (i = 0;  i < |elenco|;  i = i+1) {

        Assegna (casella, elenco[i]);

        IF (!UltimaVuota(casella) && !SUDOKU(SuccVuota(casella))) {

            Svuota (casella);

        }

        ELSE {

            Return TRUE;

        }

  }

RETURN FALSE;
```

Se non è l'ultima invoca SUDOKU sulla casella successiva

Nota: analizi alla lavagna di casi semplici per spiegare il funzionamento e il "match" con l'intuizione.

Quanto "costa" l'algoritmo risolutivo?

Quanti passi impiega sulle configurazioni più difficili?

Se ci sono all'inizio $m < 9 \times 9$ caselle vuote nella scacchiera, poiché ogni casella vuota può assumere al più 9 valori, allora i "riempimenti" possibili delle m caselle sono 9^m.

Ragioniamo in termini più generali.

9×9 $\xrightarrow{\text{ridefinito}}$ scacchiera $n \times n$ $\quad n$ = quadrato ($n = 16$, $n = 25$ ...)

Se usiamo n cifre occorre una scacchiera $n \times n$.

Inoltre, se le caselle vuote iniziali sono $m < n \times n$, i "riempimenti" possibili delle m caselle sono $n^m < n^{n \times n}$.

Ricordando che possiamo scrivere $n = 2^{\log_2 n}$, risulta

$$n^{n \times n} = 2^{\log n \cdot (n \times n)} = 2^{n^2 \cdot \log_2 n}$$

Quindi SUDOKU (algoritmo ricorsivo) nei casi difficili effettua un numero esponenziale di passi!

↓

Come nel caso dell'algoritmo che risolve il problema delle Torri di Hanoi ... ma ...

① A differenza del problema delle Torri di Hanoi, non possiamo escludere ad oggi che ne esista uno polinomiale. Non è stato trovato, ma nessuno ha dimostrato che non può esistere.

② Quando il problema ammette soluzione, esiste sempre una prova dell'esistenza di una tale soluzione che può essere verificata da un algoritmo polinomiale

Per esempio, se l'editore svela la sequenza di cifre per le m caselle vuote, l'algoritmo che segue verifica efficientemente la soluzione:

```
VerificaSudoku (sequenza)
casella = PrimaVuota();
FOR (i=0; i<m; i=i+1) {
  CIFRA = sequenza[i];
  IF (CIFRA appare in casella.riga) RETURN FALSE;
  IF (CIFRA appare in casella.colonna) RETURN FALSE;
  IF (CIFRA appare in casella.sotto_tabella) RETURN FALSE;
  Assegna (casella, CIFRA);
  casella = SuccVuota (casella);
}
RETURN TRUE
```

Quanti passi? n cifre da verificare, $n < n^2$

Ogni cifra va confrontata con:

- tutti gli elementi della riga n
- tutti gli elementi della colonna n
- tutti gli elementi della sottotabella n

$3n$ confronti

In totale occorrono al più $3 \cdot n \cdot n^2 = 3n^3$ passi.

↓ polinomio

Il Sudoku appartiene ad una classe di problemi che sono caratterizzati dall'ammettere particolari sequenze di lunghezza polinomiale, chiamate CERTIFICATI polinomiali, che permettono di verificare che il problema ha soluzione in un numero polinomiale di passi.

Nel nostro caso la sequenza di cifre è lunga $n < n^2$

↓ polinomio

e l'algoritmo di verifica, che prende in input la sequenza, richiede al più $3 \cdot n^3$ passi

↓ polinomio

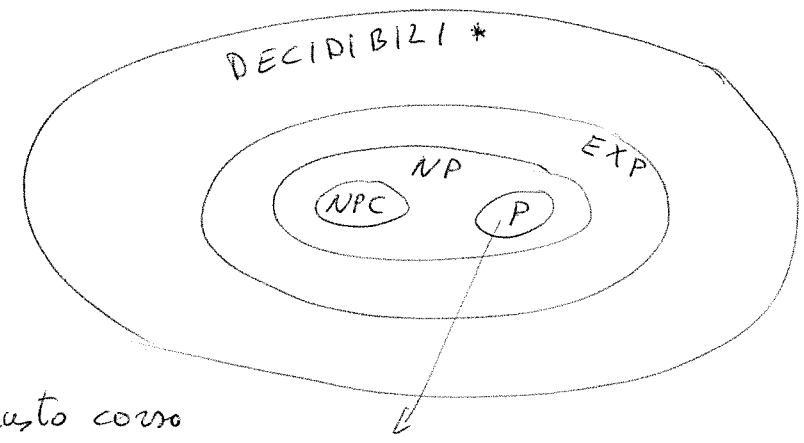
Per motivi che non ci interessano al momento, questa classe di problemi viene chiamata "Classe NP" (dove NP NON sta per "non polinomiale"!)

I problemi risolvibili tramite algoritmi polinomiali appartengono alla "Classe P" (dove P sta per polinomiale)

La Classe P $\subseteq$ Classe NP in quanto per ogni problema che ammette un algoritmo risolutivo polinomiale, possiamo usare tale algoritmo per produrne una soluzione e, quindi, un certificato polinomiale.

In realtà il Sudoku appartiene alla classe dei problemi "più difficili" di NP, denotata con NPC.

La nostra visione più raffinata dei problemi computazionali allora diventa



* la teoria della "complessità" dei problemi computazionali considera problemi di decisione (OUTPUT 0/1) e problemi di decisione risolvibili si dicono DECIDIBILI

In questo corso ci occuperemo di problemi computazionali che appartengono alla Classe P.

Riepilogando, in queste prime 3 lezioni, abbiamo appreso:

① Che cos'è un "problema computazionale" e cosa intendiamo per "algoritmo che lo risolve"

② Esiste un fenomeno noto come crescita esponenziale che in alcuni contesti apporta vantaggi ma, in altri, per esempio nell'analisi del "costo" degli algoritmi, è deleterio.
Un algoritmo esponenziale è inutilizzabile

③ Esistono problemi computazionali non risolvibili da un dispositivo di calcolo (HALTING PROBLEM)

④ Esistono problemi risolvibili da un calcolatore ma "intrattabili". Non esistono cioè algoritmi efficienti per essi
(TORRI DI HANOI)

⑤ I problemi risolvibili in modo efficiente sono quelli che ammettono algoritmi polinomiali.

⑥ Esistono problemi per cui non si conoscono algoritmi efficienti, non esistono prove della loro INESISTENZA, ma esistono algoritmi efficienti per verificare che una soluzione c'è.
(SUDOKU)