Cenni di teoria degli algoritmi

E.Mumolo. DEEI

mumolo@units.it

Cenni di computabilita'.

Definizione di algoritmo

- 1ª definizione (Goedel)

  → E' una sequenza di regole per formare funzioni matematiche complesse da funzioni piu' semplici
- 2ª definizione (Church)

  → Un algoritmo e' un formalismo chiamato λ-calcolo
- 3ª definizione (Turing)

  → Un algoritmo e' una sequenza di istruzioni (programma) per il calcolatore ipotetico chiamato "macchina di Turing"
- 4ª definizione

  → Un algoritmo e' una sequenza di operazioni di base effettuate su una struttura dati e controllata da schemi operativi, sequenziali e di selezione
- 5ª definizione

  → Un algoritmo e' qualsiasi funzione complessa svolta da un calcolatore
- Tesi di Church-Turing
  - Tutte queste definizioni sono tra loro equivalenti
  - Qualsiasi definizione ragionevole di algoritmo e' equivalente alle definizioni che conosciamo
- Inoltre

Ogni volta che scriviamo una serie di passi discreti per risolvere un problema, esiste un algoritmo per risolvere il problema

Tutti i calcolatori sono equivalenti (interpreti) e sono equivalenti alla macchina di Turing

Caratteristiche di un algoritmo

- Il programma che descrive un algoritmo deve essere di lunghezza finita

→quindi, c'e' un limite al numero e alla complessita' delle istruzioni

- Ci deve essere una *macchina* (ipotetica o no) in grado di eseguire le istruzioni
- La *macchina* deve avere una memoria per immagazzinare i risultati intermedi
- Il calcolo non e' probabilistico (il prossimo passo non viene scelto a caso)
- Non c'e' nessun limite alla lunghezza dei dati di ingresso
  →quindi, non ci deve essere nessun limite alla quantita' di memoria
- Non ci deve essere nessun limite al numero di passi
- Ci possono essere esecuzioni con un numero infinito di passi (algoritmo che non termina!)

Preliminari: dimensione di un'insieme

- Cardinalita' di un insieme finito: numero di elementi dell'insieme
  - Es: se S e' un insieme finito di M elementi, *card(*S*)*=M
- Ma: se l'insieme e' infinito?
- **Due insiemi hanno la stessa cardinalita' se i loro elementi possono essere messi in corrispondenza biunivoca** (es. numeri naturali, numeri pari)
- Se gli elementi di un insieme S_1 possono essere messi in corrispondenza biunivoca con gli elementi di un sottoinsieme di un altro insieme, S_2, allora si dice $card(S_1) \leq card(S_2)$
- Se gli elementi di un insieme S_1 possono essere messi in corrispondenza biunivoca con gli elementi di un sottoinsieme PROPRIO di S_2 allora $card(S_1) < card(S_2)$
- Insieme finito S di M elementi: un sottoinsieme puo' essere rappresentato con una parola binaria di M bit
- Numero di sottoinsiemi: 2^M → *card(*insieme di sottoinsiemi di S*)* = 2^M
  → *card(*S*)* < *card(*insieme di sottoinsiemi di S*)*

Preliminari: dimensione di un'insieme (cont)

- INSIEMI INFINITI: Per es. **Cardinalità** dei numeri naturali, N, e dell'insieme dei sottoinsiemi di N
- Un sottoinsieme di N si puo' rappresentare con un numero binario di infinite cifre
- C'e' corrispondenza biunivoca tra N e numeri binari di infinite cifre?

Supponiamo che sia possibile. Cifre binarie: $b_i\ b_j\ b_k \ldots b_n\ b_m \ldots$

Elenco dei numeri naturali →

	0	1	2	3	...
0	b_{00}	b_{01}	b_{02}	b_{03}	...
1	b_{10}	b_{11}	b_{12}	b_{13}	...
2	b_{20}	b_{21}	b_{22}	b_{23}	...
3	b_{30}	b_{31}	b_{32}	b_{33}	...
.	...				
.	...				
.	...				
i	b_{i0}	b_{i1}	b_{i2}	b_{i3}	...
.	...				

Prendiamo il numero binario di infinite cifre sulla diagonale: B= $b_{00}\ b_{11}\ b_{22}\ b_{33} \ldots$

Esiste sicuramente un numero naturale dell'elenco. B_k= $b_{k0}\ b_{k1}\ b_{k2}\ b_{k3} \ldots$, uguale a B: basta che $b_{ki} = b_{ii}$ per ogni i

MA: il numero $\overline{B}$? Se B=$\overline{B_k}$, sarebbe $b_{ki} = \overline{b_{kk}}$ per ogni i → assurdo! → $\overline{B}$ non compare nell'elenco ➔ *card(*N*)* < *card(*insieme di sottoinsiemi di N*)*

Preliminari: dimensione di un'insieme (cont)

- Consideriamo l'insieme delle funzioni binarie su N: $F_b = \{f : N \xrightarrow{f} \{0, 1\}\}$
- Facciamo dapprima riferimento per semplicità a insiemi finiti
  - E' noto che le applicazioni di un insieme A di k elementi in un insieme di n elementi sono n^k.
  - Quindi le funzioni binarie su un insieme di n elementi è 2^n
  - E' altresi' noto che il numero di sottoinsiemi di un insieme di n elementi è 2^n
  - Quindi, nel caso di insiemi finiti, il numero delle funzioni binarie su un insieme di n elementi è uguale al numero di sottoinsiemi dell'insieme di n elementi
- Analogamente si può dimostrare che il numero delle funzioni binarie su N (Fb) ha la cardinalità dell'insieme di sottoinsiemi di N
- Infatti, gli elementi di Fb possono essere messi in corrispondenza biunivoca con i numeri binari di infinite cifre! In altri termini:
  B= $b_0\ b_1\ b_2\ b_3$ … definisce biunivocamente una funzione f(k) = b_k appartenente a Fb

  Quindi: *card*(F_b) = *card*(insieme di sottoinsiemi di N)
- Passiamo ora all'insieme F = Insieme delle funzioni N su N: $F = \{f : N \xrightarrow{f} N\}$
- Naturalmente $F_b \subset F$ ➔ *card*(F_b) = *card(insieme sottoinsiemi di N)* < *card*(F)

Preliminari: dimensione di un'insieme (cont)

- Quindi:

  $card(\mathbb{N}) < card(\text{insieme di sottoinsiemi di } \mathbb{N}) = card(\mathbb{F}_b) < card(\mathbb{F})$

  ➔ $card(\mathbb{N}) < card(\mathbb{F})$

- Ovvero: l'insieme delle funzioni N su N non è numerabile

- possiamo renderci già conto ora che devono esserci funzioni per cui non esistono programmi di calcolatore per calcolarle
  - infatti, l'insieme dei programmi è numerabile (basta metterli in ordine lessicografico)
  - mentre l'insieme delle funzioni non è numerabile

➔ vediamolo un pò meglio

Primo problema

- Ci sono moltissimi compiti che si possono fare con un algoritmo
- Domanda: *esistono compiti per cui non esiste un algoritmo?*

Sia S = {linguaggio, macchina di calcolo}.

➔ S e' un sistema formale nel quale si possono scrivere algoritmi.

Algoritmo A in S che termina = funzione f_A, calcolata da A

➔ Un problema è computabile se esiste un algoritmo (che termina) per risolverlo

Funzione totale = termina per tutti i valori di ingresso

Funzione parziale = termina solo per alcuni valori

- Insieme di tutti gli algoritmi in S = A_s
- Insieme di tutte le funzioni calcolabili in S = F_s

Esiste un sistema formale S dove si possono risolvere tutti i problemi esistenti?

Cioe', esiste S tale che F_s comprende tutte le funzioni?

Primo problema (cont.)

In S, il linguaggio sara' fatto da un numero finito di simboli.

→ Insieme di tutti gli algoritmi definiti in S: A_s

Ordiniamo e numeriamo gli infiniti algoritmi scritti con il linguaggio di S

$card(A_s) \leq card(N)$ e $card(N) < card(F)$ ➔ $card(A_s) < card(F)$

→ Insieme di tutte le funziono calcolabili in S: F_s

Corrispondenza biunivoca tra F_s e il sottoinsieme di A_s che calcola la stessa funzione

$card(F_s) \leq card(A_s)$ e $card(A_s) < card(F)$

➔ $card(F_s) < card(F)$

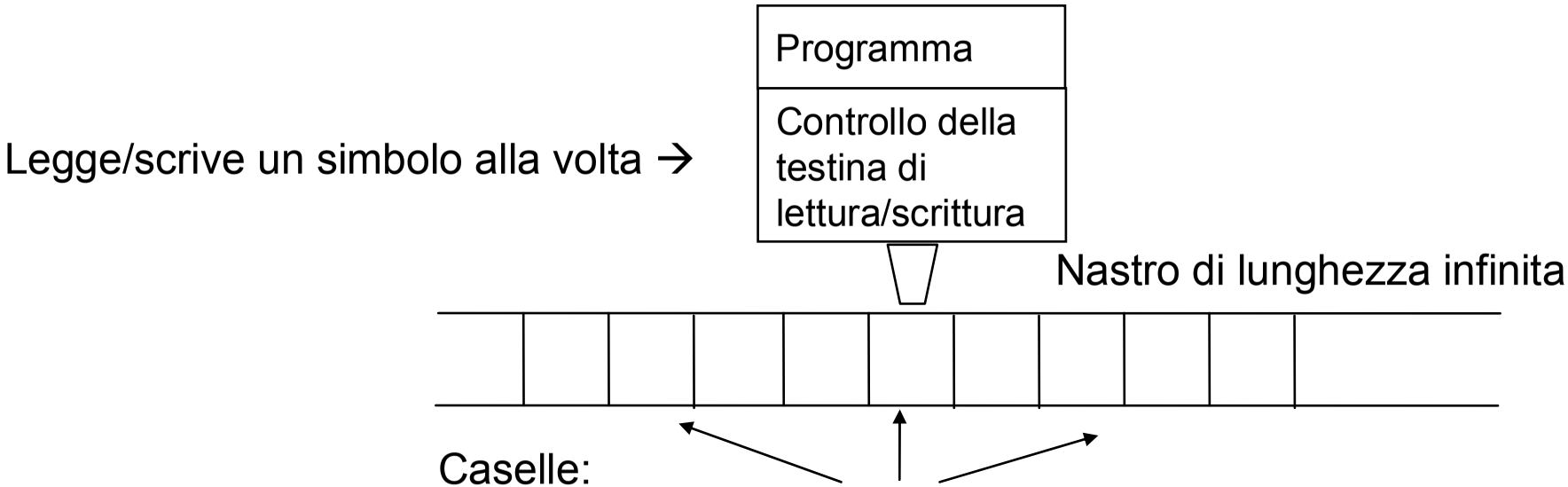
$Nota_1$: F={funzioni intere su S} F_s={funzioni calcolabili su S}

$Nota_2$: $F_s \subseteq F$ ➔ $F_s \subset F$ anzi F_s e' un piccolissimo sottoinsieme di F

- Cioè, la risposta alla domanda della slide precedente è:

  il numero di compiti per cui non esiste nessun algoritmo e' infinitamente maggiore di quelli per cui esiste un algoritmo!

Il modello di Turing



- il controllo si trova in uno degli stati q_0, q_1, …q_m.
- la macchina si trova nello stato q_i e legge il simbolo s_j.
- il controllo contiene una descrizione dell'azione che deve essere eseguita: $s_k q_r x_t$
- azione: scrivere un simbolo s_k, portarsi nello stato q_r, spostarsi a destra (x_t=D) o sinistra(x_t=S) o restare fermo (-)
- il comportamento del controllo puo' essere descritto con la matrice funzionale (stati,simboli)

Il modello di Turing

Esempio: sostituisce tutti gli 0 con ‘Z’ e tutti gli 1 con ‘U’

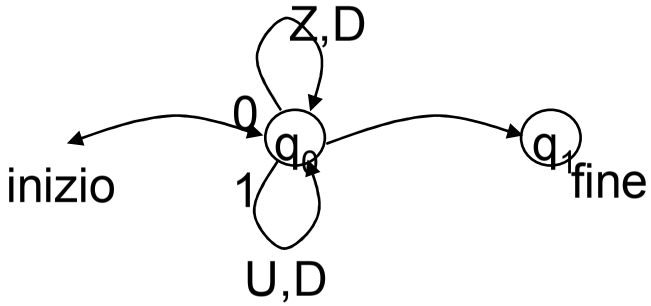
	0	1	b
q_0	Zq_0D	Uq_0D	bq_1S
q_1			

stati

Simboli di ingresso

stato iniziale=q_0 stato finale (fermata)=q_1 b=casella vuota

Rappresentazione grafica (**AUTOMA**)

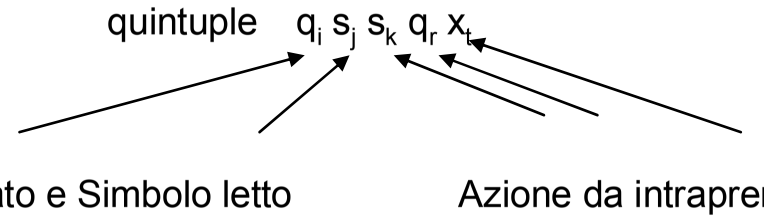


Rappresentazione alternativa (**CINQUINE**)

Nel caso sopra: $q_0 0 Z q_0 D$

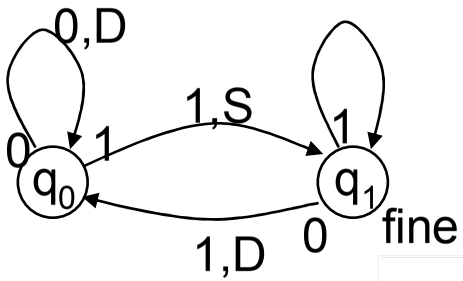
$q_0 1 U q_1 D$

$q_0 b b q_1 S$



Il modello di Turing

Esempio: programma per una MdT che conta in binario.
Ipotesi: la stringa e' azzerata e un marcatore (1). I numeri binari sono scritti a sinistra del marcatore, in occorrenza del primo stato q_0 o dell'ultimo stato q_1 di ogni sequenza



Matrice funzionale

	0	1
q_0	0 q_0 D	1 q_1 S
q_1	1 q_0 D	0 q_1 S

q0	…00<u>0</u>100…	q0	…001<u>0</u>100…
q0	…000<u>1</u>00…	q0	…0010<u>1</u>00…
q1	…000100…	q1	…0010100…
q0	…001<u>1</u>00…	q0	…0011<u>1</u>00…
q1	…001100…	q1	…0011100…
q1	…0<u>1</u>0100…	…	

Il modello di Turing

Esempio: programma per la MdT che valuti se la stringa di ingresso contiene un numero pari o dispari di '1' (controllo di parita')

Ipotesi: la stringa binaria e' circondata da caselle bianche ('b')

Inizialmente la testina sta sul primo bit a sinistra.

Macchina a quattro stati: stato iniziale q_0, stati finali q_2 (pari), q_3 (dispari)

Descrizione funzionale

	0	1	b
q0	0 q0 D	1 q1 D	b q2 S
q1	0 q1 D	1 q0 D	b q3 S
q2			
q3			

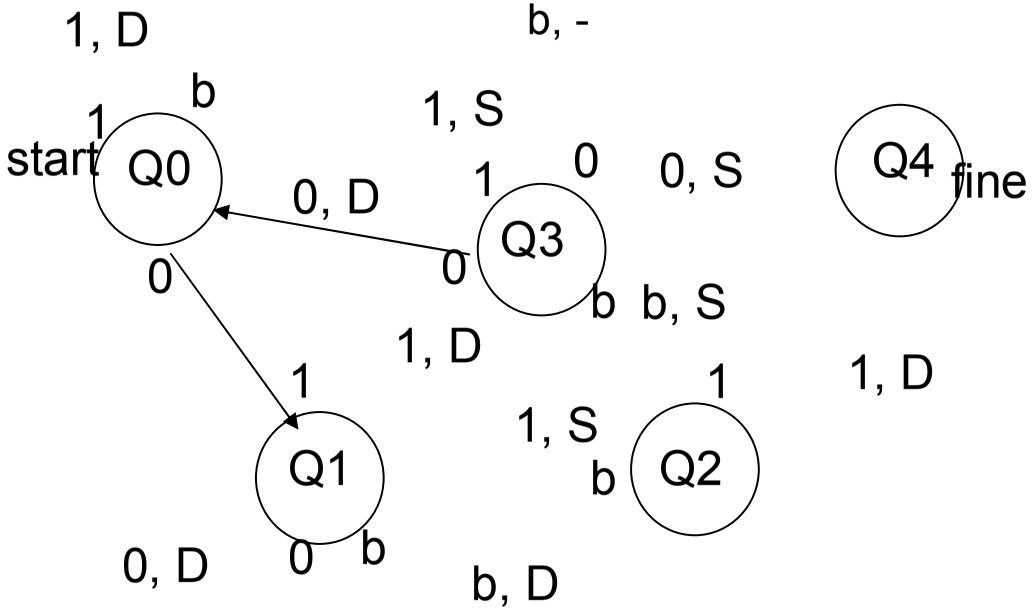
Descrizione alternativa

q_0 0 0 q_0 D
q_0 1 1 q_1 D
q_0 b b q_2 S
q_1 0 0 q_0 D
q_1 1 1 q_0 D
q_1 b b q_3 S

Il modello di Turing

Esempio: programma per la MdT che conta gli zeri di una stringa binaria di ingresso in 'unario' (sistema di numerazione posizionale in base 1)

	0	1	b	*
Q0	Q_1 * D	Q_0 1 D	Q_4 b S	
Q1	Q_1 0 D	Q_1 1 D	Q_2 b D	
Q2		Q_2 1 D	Q_3 1 S	
Q3	Q_3 0 S	Q_3 1 S	Q_3 b S	Q_0 0 D
Q4				



Esempio:

prima

	b	1	0	0	1	b	b	b	b	

dopo

	b	1	0	0	1	b	1	1	b	

Il modello di Turing

Esempio: programma per una MdT che conta il nr. di '1' e lo scrive in decimale
Ipotesi: stato iniziale q_0 separatore: # alfabeto S={0,1,2, …, 8,9,b,#,*}

Prima:

b	#	0	1	1	0	0	b

dopo:

2	#	0	1	1	0	0	b

Matrice funzionale:

	0	1	2	3	4	5	6	7	8	9	#	*	b
q0											#q1D		
q1	0q1D	*q2S											
q2	0q2S	1q2S									#q3S		
q3	1q4D	2q4D	3q4D	4q4D	5q4D	6q4D	7q4D	8q4D	9q4D	0q3S			1q4D
q4	0q4D	1q4D									#q4D	1q1D	

Il modello di Turing

Esempio: programma per una MdT che calcola il successivo di un numero in decimale
Ipotesi: sul nastro c'e' il numero iniziale, la testina e' sull'ultima cifra a destra

Es.

b	b	0	1	9	b	b	

MdT a due stati, alfabeto: S={b,0,1,2,…,8,9} stato iniziale q_0

Matrice funzionale

	0	1	2	3	4	5	6	7	8	9	b
q_0	1 q_1 D	2 q_1 D	3 q_1 D	4 q_1 D	5 q_1 D	6 q_1 D	7 q_1 D	8 q_1 D	9 q_1 D	0 q_0 S	1 q_1 D
q1											

Descrizione con cinquine

q_0 0 1 q_1 D
q_0 1 2 q_1 D
q_0 2 3 q_1 D
q_0 3 4 q_1 D
q_0 4 5 q_1 D
q_0 5 6 q_1 D
q_0 6 7 q_1 D
q_0 7 8 q_1 D
q_0 8 9 q_1 D
q_0 9 0 q_0 S
q_0 b 1 q_1 D

Il modello di Turing

Esempio: programma per una MdT che conta in decimale

Ipotesi: nastro tutto bianco o con un numero di partenza: in questo caso la testina sta sul numero. Due stati, stato iniziale q_0

MdT a due stati, alfabeto: S={b,0,1,2,…,8,9} stato iniziale q_0

Matrice funzionale

	0	1	2	3	4	5	6	7	8	9	b
q_0	1 q_1 D	2 q_1 D	3 q_1 D	4 q_1 D	5 q_1 D	6 q_1 D	7 q_1 D	8 q_1 D	9 q_1 D	0 q_0 S	1 q_1 D
q1	0 q1 D	1 q1 D	2 q1 D	3 q1 D	4 q1 D	5 q1 D	6 q1 D	7 q1 D	8 q1 D	9 q1 D	b q0 S

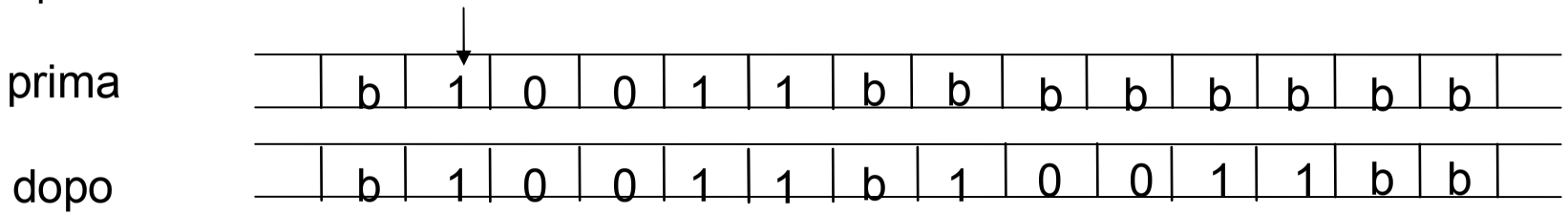
Il modello di Turing

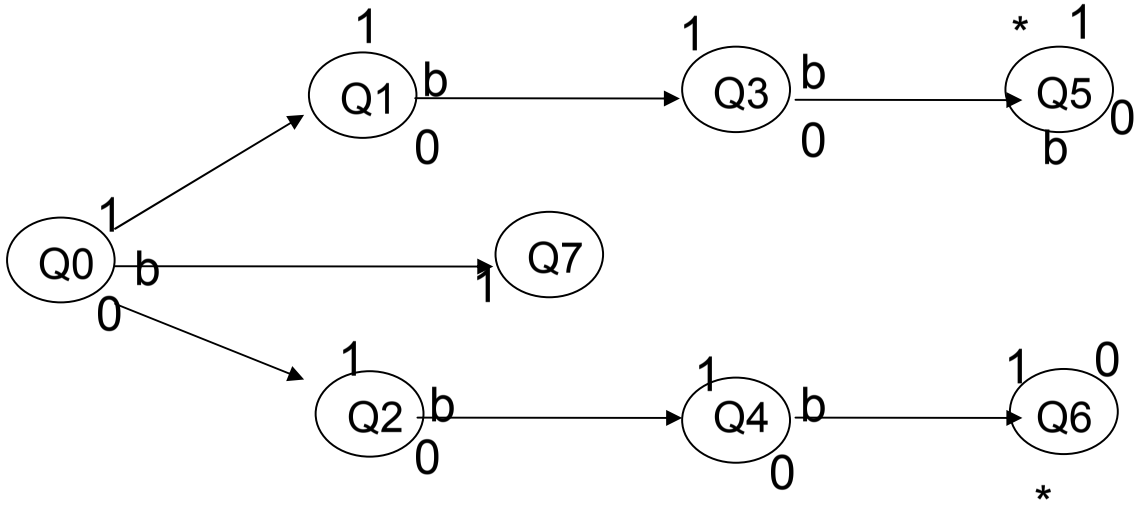
Esempio: programma per la MdT che copia una stringa di 1 e 0 a destra dopo il blank. Il programma parte dallo stato Q0 (stato iniziale) e finisce nello stato Q7 (stato finale).
La posizione iniziale della testina e' indicata con la freccia.
L'idea è di marcare lo 0 o 1 con un * e andare a destra fino a quando si incontra b. A questo punto si riscrive b, si va' a destra e si cambia stato (diverso se partiva da 0 o da 1). In questo nuovo stato si va' a destra fino a quando si incontra un nuovo b, alposto del quale si copia lo 0 o 1 dal quale si era partiti e si torna a sinistra fino a incontrare *. Si riscrive lo 0 o l'1 dal quale si era partiti e si ricomincia.

	1	0	b	*
Q0	Q1,*,D	Q2,*,D	Q7,b,-	
Q1	Q1,1,D	Q1,0,D	Q3,b,D	
Q2	Q2,1,D	Q2,0,D	Q4,b,D	
Q3	Q3,1,D	Q3,0,D	Q5,1,S	
Q4	Q4,1,D	Q4,0,D	Q6,0,S	
Q5	Q5,1,S	Q5,0,S	Q5,b,S	Q0,1,D
Q6	Q6,1,S	Q6,0,S	Q6,b,S	Q0,0,D

Il modello di Turing

Dalla slide precedente: programma per la MdT che copia una stringa di 1 e 0 a destra dopo il blank





Il modello di Turing

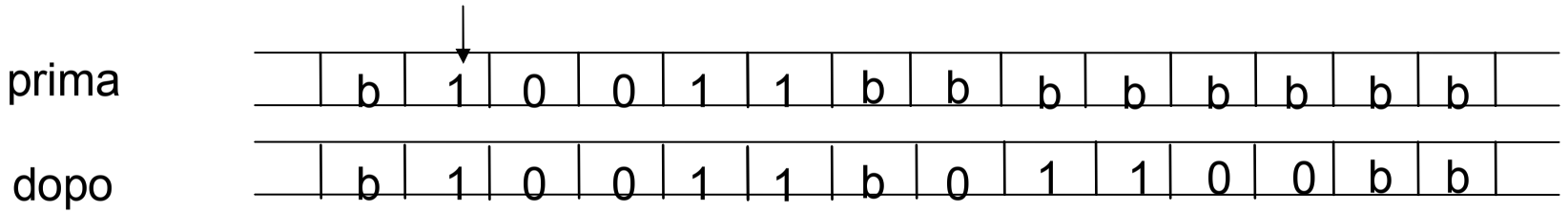
Programma per la MdT che fa la negazione logica di una stringa di 1 e 0 e la riscrive a destra dopo il blank.

Questo programma si puo' facilmente ricavare dal programma precedente. Basta sostituire la riga Q3 e Q4 con:

Q3	Q3,1,D	Q3,0,D	Q5,0,S	
Q4	Q4,1,D	Q4,0,D	Q6,1,S	

Cosi' facendo quando si riscrive la stringa a destra, se si parte da un 1 si scrive 0 e se si parte da 0 si scrive 1.

Funzionamento sulla stringa di prima:



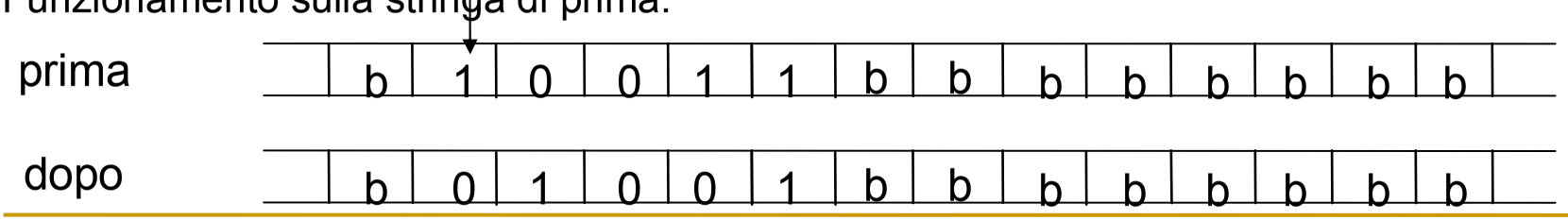
Il modello di Turing

Programma per la MdT che fa la divisione per 2 di una stringa binaria e la riscrive al posto della stringa iniziale.

Questo corrisponde a fare lo shift a destra della stringa di1 e 0.

	0	1	b
Q0	Q0,0,D	Q0,1,D	Q1,b,S
Q1	Q2,b,S	Q2,b,S	
Q2	Q3,0,D	Q4,1,D	Q5,b,D
Q3			Q1,0,S
Q4			Q1,1,S
Q5			Q6,0,-
Q6			

Funzionamento sulla stringa di prima:



Il modello di Turing

Programma per la MdT che riconosce se una stringa con caratteri A e B e' palindroma (si legge da entrambi i lati, es. ABBA).

Se lo stato finale e' Q6, allora la stringa e' palindroma, se lo stato finale e' Q7, no.

L'idea e' di cancellare a destra e sinistra lo stesso carattere: A con A, B con B. Se non si trova lo stesso carattere la stringa non e' palindroma.

	A	B	b
Q0	Q1, b, D	Q2, b, D	Q6, b, -
Q1	Q1, A, D	Q1, B, D	Q3, b, S
Q2	Q2, A, D	Q2, B, D	Q4, b,S
Q3	Q5, b, S	Q7, B, -	
Q4	Q7, A, -	Q5, b, S	
Q5	Q5, A, S	Q5, B, S	Q0, b, D
Q6			
Q7			

Automi

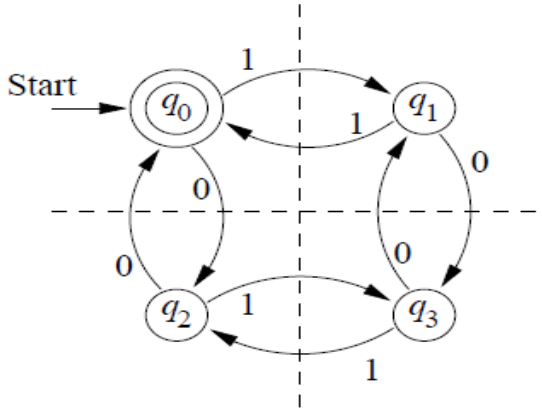
- Una macchina di Turig rappresenta un automa a stati finiti

- Un automa a stati finiti è un sistema dinamico, discreto ed invariante, in cui gli insiemi d'ingresso, di uscita e di stato sono finiti.

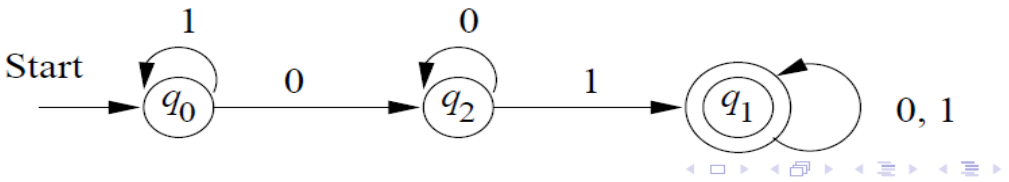
- Un automa a numero di stati finito deterministico è definito come un sistema A = {I, U, S, f, g}, dove
  - I = {i1, i2, …, in} è l'insieme finito dei possibili simboli in ingresso
  - U = {u1, u2, …, um} è l'insieme finito dei possibili simboli in uscita
  - S = {s1, s2, …, sh} è un insieme finito degli stati
  - f: $I \times S \rightarrow U$ è una funzione delle uscite (eventualmente parziale), che collega l'uscita al valore attuale dell'ingresso e dello stato, U(t)=f (S(t), I(t))
  - g: $I \times S \rightarrow S$ è la funzione di transizione degli stati interni successivi, che collega lo stato nell'istante successivo al valore attuale dell'ingresso e dello stato, S(t+1)=g(S(t), I(t))
- Automa di Mealy e Automa di Moore
  - Automa di Moore: la funzione f dipende solo dallo stato: $f = S \rightarrow U$ e dunque U(t)=f (S(t)).
  - Automa di Mealy: la funzione f dipende dallo stato e dagli ingressi.

Automi-esempi

- Automa a stati finiti deterministico che accetta tutte e sole le stringhe con un numero pari di zeri e un numero pari di uni

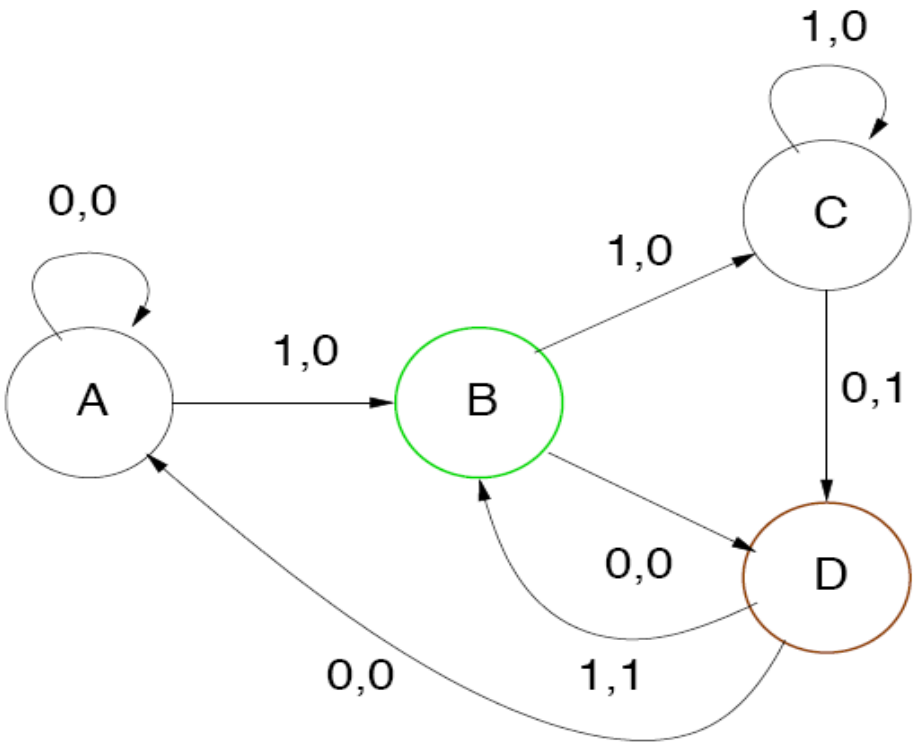


- Automa che accetta la stringa L = {x01y : x, y ∈ {0, 1}* }



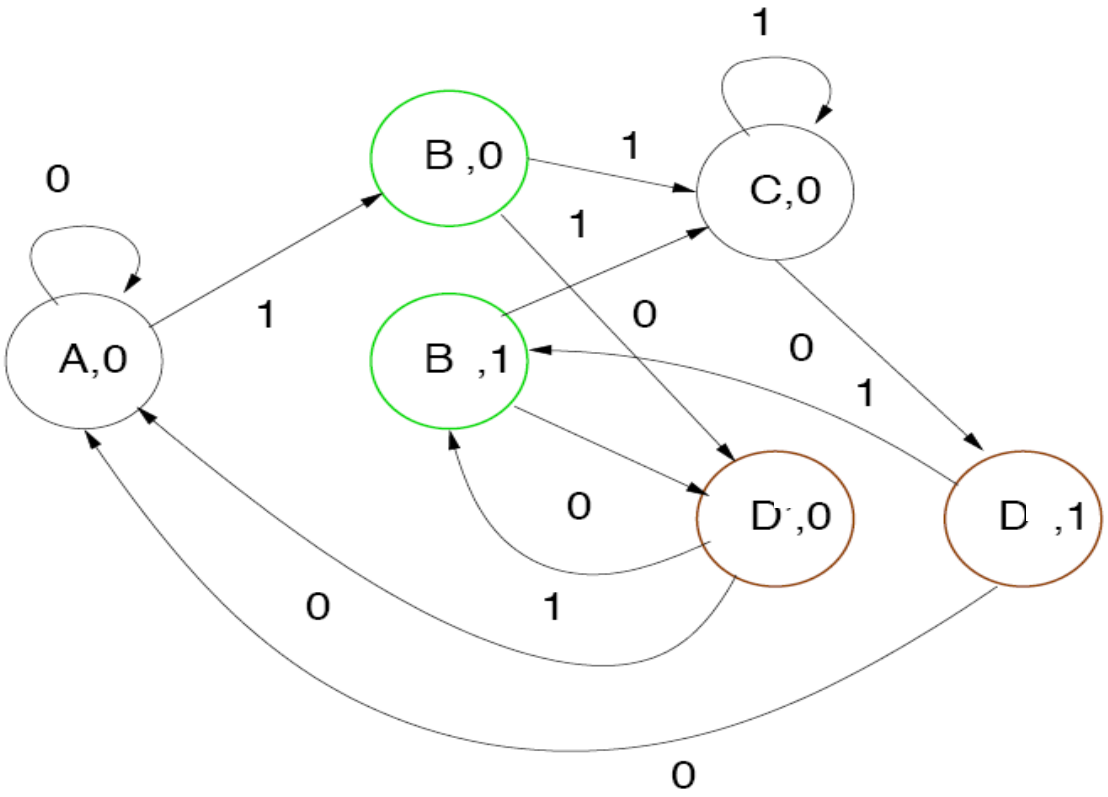
Automi-esempi

Automa di Mealy che “riconosce” le sequenze 101 e 110



Automi-esempi

Automa di Moore equivalente



Calcolo con il modello di Turing

- Calcolo della MdT: sequenza (eventualmente infinita) di configurazioni della MdT
- Funzioni T-calcolabili: calcolabili con la MdT
- *NB: per motivi storici si usa spesso l'aggettivo 'ricorsivo' al posto di 'calcolabile'*
- Ad ogni MdT puo' essere associata la funzione calcolata dalla MdT
- Estensioni:
  - Una MdT con h testine e un nastro puo' essere simulata da una MdT con 1 testina e 1 nastro
  - Una MdT con n nastri e h testine puo' essere simulata da una MdT con 1 nastro e h testine
  - Una MdT con n nastri, dove il j-esimo nastro e' kj-dimensionale ed ha hj testine, puo' essere simulato da una MdT con nastro monodimensionale e 1 testina

→ Complicando la macchina di Turing, l'insieme delle funzioni T-calcolabili non cambia, Cambia solo la velocita' di calcolo

Enumerazione delle macchine di Turing

- Semplificazione: MdT con 2 simboli e k stati.
- Matrice funzionale: 2k celle, ciascuna contiene una terna sqx.

	s0	s1
Es. con 1 stato:	q0	sqx sqx

- Rappresentiamo la matrice funzionale per righe: ogni stringa e' una MdT!
- Quante terne?
  2k2 terne sqx piu' cella vuota = (4k +1)
- Quante celle?
  2k
- Quante stringhe (cioè quante MdT?)
  $N_k = (1+4k)^{2k}$
- Cioe': se k=1 (1 stato) → n_1 = 25 MdT
  se k=2 (2 stati) → N_2 = 6561 MdT
- Funzioni non calcolabili
  - Un problema e' non calcolabile se non esiste nessun algoritmo per risolvere il problema
    → non vuol dire che non conosciamo nessun algoritmo per risolvere il problema, ma che non lo conosceremo mai!

Il problema dell'arresto (cont.)

- Data una corrispondenza biunivoca tra N e l'insieme dei programmi P, si consideri la seguente MdT h: N → {0,1}:

$$h(x) = \begin{cases} \textbf{1 se il programma x-esimo si ferma con input x} \\ \textbf{0 se il programma x-esimo non si ferma con input x} \end{cases}$$

- Supponiamo che h(x) esista. E' lecito allora costruire la MdT che calcola f(.):

$$f(x) = \begin{cases} \textbf{1 se h(x) = 0} \\ \textbf{non si ferma se h(x) = 1} \end{cases}$$

- Analizzando la corrispondenza biunivoca, f(.) è il programma i-esimo.
- Ma allora consideriamo f(i).
  - Se f(i) = 1, allora h(i) = 0 cioè, dalla definizione di h(.), il programma i-esimo, NON si ferma con input i , cioè f(i) non si ferma che è contraddetto dal fatto che f(i)=1
  - Se f(i) NON si ferma, allora h(i) =1 cioè f(i) si ferma
  - IN OGNI CASO ABBIAMO UNA CONTRADDIZIONE
    - → l'ipotesi che h(x) esista è assurda!
    - → h(x) non esiste!

Il problema dell'arresto della MdT (cont.)

- Conclusione: non esiste nessun algoritmo che risolve il problema dell'arresto! → il problema dell'arresto non e' computabile!
- Altri problemi non computabili:

*Il problema della totalita': una funzione (programma) F(x)e' definita per tutti gli x ?

→ non e' computabile!

*Cambio calcolatore e programmi. Problema: i nuovi programmi sono equivalenti a quelli vecchi?
→ non e' computabile!

Cioe': dati due programmi P e Q, essi calcolano la stessa funzione? Ovvero, P(x)=A(x) per tutti gli x? → non e' computabile!

*Linguaggi di programmazione descritto in BNF.

es.:identificatore:<letter> ::= A | B | ... | Z

<digit> ::= 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9

<ident> ::= <letter> | <ident> <letter> | <ident> <digit>

Problema: date due BNF, esiste un algoritmo per decidere se le due grammatiche rappresentano lo stesso linguaggio?

→ non e' computabile!

*Equazione Diofantee, per esempio $(a+1)^2 + (b+1)^3 = (c+1)^4$ solo risolvibili con numeri interi? → non e' computabile!

*Decimo problema di Hilbert: una generica equazione polinomiale a coefficienti interi ha soluzione intera? → non e' computabile!

...

Cenni sulla teoria della Complessita'

- Studio della computabilita' →quali problemi ammettono una soluzione algoritmica?
- Studio della complessita' → i problemi risolvibili, richiedono una quantita' di risorse accettabile?
- Problemi computabili: fattibili, non fattibili
- Risorse del computer:
  - Tempo (di esecuzione di un programma sulla unita' elaboratrice)
  - Memoria (spazio richiesto dall'algoritmo per immagazzinare i dati)
  - Potenza di calcolo (potenza o numero di unita' elaboratrici)
- Tempo di esecuzione di un programma: dipende da
  - Complessita' dell'algoritmo (non dipende dal calcolatore)
  - Efficienza del sw di sistema
  - Potenza dell'elaboratore
- Complessita' dell'algoritmo:
  - Indipendente dal calcolatore, dal linguaggio, dal compilatore
  - Dipende solo dal numero di dati usati

Complessita'

- Valutazione della complessita' di un algoritmo:
  - Funzione esatta della quantita' dei dati di ingresso
    - → troppo difficile
    - → fuorviante
  - Comportamento asintotico
  - Fattibilità o no stabilita sulla base del comportamento asintotico
  - Complessità lineare (n), polinomiale (n^c), esponenziale (c^n), logaritmica ($\log_2 n$)
  - Complessità nel peggiore dei casi (worst case), caso intermedio (average case)
- Valutazione della complessita' dei problemi
  - complessità del miglior algoritmo per risolvere un problema

Quantita' n dei dati di ingresso	$\log_2 n$ [μs]	n [μs]	n^2 [μs]	2^n [μs]
10	0.000003 s	0.00001 s	0.0001 s	0.001 s
1000	0.00001 s	0.001 s	1 s	> 10^{287} anni
100000	0.000017 s	0.1 s	2.8 ore	> 10^{30000} anni

Ordine di complessita'

- Ordine di complessita' superiore O(.)
  f(n) e' dell'ordine O(g(n)) se esistono due costanti c e n_0 tali che $|f(n)| \leq c\,|g(n)|$ per ogni $n>n_0$
  es: $f(n) = 100\,n^2 + 150 \rightarrow f(n) \leq 200\,n^2$ per $n>1 \rightarrow f(n)$ e' $O(n^2)$

- Ordine di complessita' inferiore $\Omega(.)$
  f(n) e' dell'ordine $\Omega(g(n))$ se esistono due costanti c e n_0 tali che $|f(n)| \geq c\,|g(n)|$ per ogni $n>n_0$
  es: $f(n) = 100\,n^2 + 150 \rightarrow$ f(n) e' $\Omega(n^2)$ ma anche $\Omega(n)$

- Ordine di complessita' equivalente ~
  $f(n) \sim g(n)$ se esiste una costante n_0 tale che $\lim f(n)/g(n) = 1$ per $n \rightarrow \infty$, $n > n_0$
  es: $f(n) = 100\,n^2 + 150 \rightarrow$ f(n) e' $\Omega(n^2)$ ma anche $\Omega(n)$

- Ordine di complessita' $\Theta(g(n))$
  f(n) e' dell'ordine $\Theta(g(n))$ se esistono c1, c2, n_0 tali che $c1|g(n)| \leq |f(n)| \leq c2\,|g(n)|$ per ogni $n>n_0$
  es: $f(n) = 100\,n^2 + 150 \rightarrow n^2 \leq f(n) \leq 200\,n^2$ per $n>1 \rightarrow f(n)$ e' $\Theta(n^2)$

Ordine di complessita'

- Esempio $6n\log n + \sqrt{n}\log^2 n$ e' di O(*n*log*n*).

Infatti $6n\log n + \sqrt{n}\log^2 n \le cn\log n$ per c =7 e n>1:

$6n\log n + \sqrt{n}\log^2 n \le 7n\log n$ dato che n> 1 posso dividere per log*n*

→ $6n + \sqrt{n}\log n \le 7n$ → $\sqrt{n}\log n \le n$ → dato che n>1:

→ $\log n \le \sqrt{n}$ Cvd

- Ma anche $6n\log n + \sqrt{n}\log^2 n$ e' di Ω(*n*log*n*).

Infatti $6n\log n + \sqrt{n}\log^2 n \ge cn\log n$ per c =5 e n>1: $6n\log n + \sqrt{n}\log^2 n \ge 5n\log n$

→ $6 + \frac{\log n}{\sqrt{n}} \ge 5$ Cvd

- Quindi: $6n\log n + \sqrt{n}\log^2 n$ e' di Θ(*n*log*n*)

Complessita'

- Differenza tra complessita' polinomiale e esponenziale
- Algoritmi non fattibili: quelli esponenziali
- Algoritmi fattibili: quelli polinomiali
  - Chiusura degli algoritmi fattibili: vera se polinomiali
  - Indipendenza del tempo polinomiale dal calcolatore: un algoritmo polinomiale su un calcolatore e' polinomiale anche su un altro calcolatore
  - Tesi del calcolo sequenziale: i tempi di esecuzione dei calcolatori sequenziali sono collegati polinomialmente
  - Riassumendo: tesi di Church-Turing: tutte le definizioni di algoritmo sono tra loro equivalenti. Tesi del calcolo sequenziale: gli algoritmi fattibili sono uguali per tutti i calcolatori sequenziali
- Complessita' di un algoritmo: modifiche intelligenti e/o strutture dati intelligenti
  - Esempio: valutazione di un polinomio di ordine n p(x): $p(x) = \sum_{i=0}^{n} a_i x^i$

se valutato normalmente il numero di prodotti e' $O(n^2)$.

se fatto cosi' (metodo di Horner), il numero di prodotti e' O(n):

$$p(x) = a_0 + x(a_1 + x(a_2 + x(a_3 + \ldots + x(a_{n-1} + xa_n))\ldots)$$

Complessita'

- La asintoticità significa che vediamo solo il comportamento per grandi valori della quantità dei dati di ingresso.
- Per valori più piccoli dobbiamo vedere anche le costanti
  - differenza tra n^2 e log n: se è n^2 e se una istruzione richiede 1 microsecondo, in 1 secondo la dimensione dei dati di ingresso è $1=n^2\ 10^{-6}$ cioè n=1000 mentre se è 100.000log n in un secondo n=1024. cioè è più conveniente n^2.
- La bassa complessità è importante anche perchè i miglioramenti tecnologici sono più importanti per basse complessità:
  - si supponga che un algoritmo abbia complessità lineare: cn. Se una istruzione richiede 1 microsecondo, in un secondo l'ampiezza di dati è di 10^6 /c. Se la tecnologia migliora, e una struzione richiede 0.1 microsecondi, in 1 secondo posso effettivamente aumentare la dimensione dei dati di 10
  - se la complessità è quadratica, se l'algoritmo esegue 1 secondo, la dimensione dei dati è $1000/\sqrt{c}$ ma se le istruzioni richiedono 0.1 microsecondi in un secondo posso aumentare la dimensione dei dati solo di 3
  - se la complessità aumenta, l'effetto è sempre più evidente (per esempio se la complessità è 2^n posso aumentare la dimensione dei dati solo di 1.2 circa)
  - NB: per la stessa dimensione il vantaggio c'è: l'algoritmo è effettivamente 10 volte più veloce, quello che è penalizzato è l'aumento della dimensione dei dati

Complessita'

- Esempio: il prodotto fra interi di n cifre ha un numero di prodotti fra numeri a 1 cifra di complessita' $O(n^2)$:

```
1422 x
1343 =
4266
   5688-
4266-
  1422-
1908746
```

- Ripartiamo l'intero di n cifre in due interi di n/2 cifre: 14|22 x 13|43

  $A|B \times C|D = (A10^2+B) \times (C10^2+D) = AC10^4+(AD+BC)\,10^2+BD$

  → ha 4 prodotti tra n/2 cifre cioe' 4 $(n/2)^2 = n^2$ prodotti (non cambia niente)

Allora: $A|B \times C|D = AC10^4 + ((A+B)(C+D) - AC - BD)\,10^2 + BD$

  → ha 3 prodotti tra n/2 cifre: AC=182 , BD=946, ((A+B)(C+D) – AC –BD)=888

```
946 +
  888-- +
 182--   =
1908746
```

- Se T(n) e' il tempo per moltiplicare due interi a n cifre, allora T(n)=3T(n/2)

  → relazione di ricorrenza!

Complessita'

- Ci sono molti metodi per la soluzione delle relazioni di ricorrenza. Due di questi:
  - Metodo per iterazioni
  - Metodo per sostituzioni

- Metodo per *iterazioni*. Esempio:

T(0)=a;

T(n)=b+T(n-1):

T(n) = b+T(n-1)=b+b+T(n-2)=.b+b+b+T(n-3)=…=nb+T(n-n)=nb+T(0)=nb+a → O(n)

- Altro esempio del metodo per iterazioni:

T(0)=a;

T(n)=b+2T(n-1):

T(n) = b+2T(n-1)=b+2(b+2T(n-2))=b+2b+4T(n-2)=3b+4(b+T(n-3))=7b+4T(n-3)=

…=$(2^n-1)b+ 2^n$ T(0)=$b(2^n-1)+ a2^n = (a+b)\ 2^n$ → $O(2^n)$

Complessita'

- Metodo per *sostituzioni* per risolvere le relazioni di ricorrenza

  Riprendiamo il prodotto fra numeri interi di n cifre:

  $T(1) = c$

  $T(n) = 3T(n/2)$ sostituiamo $n=2^k$ (cioe' $k=\log_2 n=\lg n$)

  $T(2^k) = 3T(2^k/2)=3T(2^{k-1})$

  $T(2^k)=3T(2^{k-1})=3(3T(2^{k-2}))=\ldots=3^kT(2^{k-k})= 3^kT(1)=c\ 3^k= c\ 3^{\lg n}$ (vedi)

  NB: $3^{\lg n}= n^{\lg 3}$.

  Infatti $\lg 3^{\lg n} = \lg n \lg 3=\lg 3 \lg n = \lg n^{\lg 3}$. Dato che lg e' monotona $3^{\lg n}= n^{\lg 3}$ → cvd

  quindi $T(n) = c\ n^{\lg 3} = n^{1.59}$

  ➔ è l'algoritmo di Karatsuba

- Tornando al prodotto tra interi ripartendo i numeri, la complessita' e' $O(n^{1.59})$ e non $O(n^2)$ cioe' migliore del metodo tradizionale
- Approccio "**DIVIDI ET IMPERA**" → formula l'algoritmo in parti piu' piccole, piu' semplici da risolvere

Complessita'

- Prodotto matriciale:
  || A (nxn) || x || B (nxn) || = || C (nxn) || dove $c_{ij} = \Sigma_{k=1,n}\ a_{ik}\ b_{kj}$
  ogni elemento di C richiede n prodotti, ci sono n^2 elementi → $O(n^3)$
- Proviamo a partizionare le matrici:

$$\begin{Vmatrix} A & B \\ C & D \end{Vmatrix} \times \begin{Vmatrix} E & F \\ G & H \end{Vmatrix} = \begin{Vmatrix} I & L \\ M & N \end{Vmatrix}$$

  dove I=AE + BG, L=AF + BH, M=CE +DG, N=CF + DH
  cioe' 8 prodotti tra matrici di dimensione n/2. Quindi T(n)=8T(n/2) e T(1)=c
  Per sostituzione, vedi la pagina precedente (k=lgn):
  $T(n)=8^kT(1)=c8^k=c8^{lgn}=cn^{lg8}=cn^3$ → $O(n^3)$
- Facciamo allora cosi':
  $P_1=(A+D)(E+H)$, $P_2=(C+D)E$, $P_3=A(F-H)$, $P_4=D(G-E)$, $P_5=(A+B)H$
  $P_6=(C-A)(E+F)$, $P_7=(B-D)(G+H)$ → cioe' 7 prodotti tra matrici (n/2 x n/2)
  Allora: $I=P_1+P_4-P_5+P_7$, $L=P_3+P_5$, $M=P_2+P_4$, $N=P_1+P_3-P_2+P_6$
  (verificare algebricamente!).
  Allora: $T(n)=7T(n/2)=7^kT(1)=c7^k=c7^{lgn}=cn^{lg7}=cn^{2.81}$ → $O(n^{2.81})$
  questo algoritmo (Strassen) e' migliore di quello tradizionale!!

Complessita'

- Algoritmo molto inefficiente per calcolare 2^n:

```
#include <stdio.h>
int sciocca(int n)
{
     if(n<=0)  return (1);
     else      return (sciocca(n-1)+sciocca(n-1));
}
main()
{
     printf("2^4 = %d\n",sciocca(4));
     printf("2^5 = %d\n",sciocca(5));
     printf("2^6 = %d\n",sciocca(6));
}
```

  Che complessita' ha?

  $T(0)=c$, $T(n)=2T(n-1)=2(2T(n-2))=2(2(2T(n-3)))=\ldots=2^nT(0)=c2^n$

  → complessita' esponenziale

- Fattoriale: `int fact(int n) { if(n<=1) return 1 else return n*fact(n-1); }`

  relazioni di ricorrenza:

  $T(1)=a$ //peso di un return

  $T(n)=b+T(n-1)$ //un return piu' un prodotto piu' $T(n-1)$

  per iterazione $T(n)=b+T(n-1)=b+b+T(n-2)=\ldots=(n-1)b+T(1)=(n-1)b+a=nb+a-b$

  → $O(n)$

Complessita' dell'ordinamento per selezione – versione iterativa

```
void selectionsort ( int a[], int n )
{
  int i,j,k,t;
  for (i=n-1; i > 0; i-- ){
        k = 0;
        for (j=1; j <= i; j++ )
            if ( a[j] > a[k] )  k=j;
        t = a[k];
        a[k] = a[i];
        a[i] = t;
  }
}
```

- Complessita' (in numero di confronti):

i=(n-1) → j=1,2,…,n-1

i=(n-2) → j=1,2,…,n-2

…

i=1 →j=1

Numero di confronti = (n-1)+(n-2)+…+1 = $n(n-1)/2=n^2/2-n/2$ → $O(n^2)$

Complessita' dell'ordinamento per selezione – versione ricorsiva

```
#include <stdio.h>
void sel(int *s,int n)
{     int max=0,t,i;
      if(n==0) return;
      for(i=1;i<=n;i++)
            if(s[i]>s[max]) max=i;
      t=s[n];s[n]=s[max];s[max]=t;
      sel(s,n-1);
}
main()
{  int i, s[6]={80,22,1,0,3,7};
   for(i=0;i<6;i++) printf("%d ",s[i]);printf("\n");
   sel(s,5);
   for(i=0;i<6;i++) printf("%d ",s[i]);
}
```

- Relazione di ricorrenza: T(n)=T(n-1)+(n-1). Iterativamente:

T(n)=T(n-1)+(n-1)=T(n-2)+(n-2)+(n-1)=T(n-3)+(n-3)+(n-2)+(n-1)=
=1+2+…+(n-1)=n(n-1)/2=$n^2/2-n/2$ → $O(n^2)$

Complessita' della ricerca binaria – versione ricorsiva

```
int bin(int *s,int i, int j, int key)
{    int k;
     if(i>j) return(-1);
     k=(i+j)/2;
     if (key==s[k]) return(k);
     if (key<s[k]) j=k-1;  else i=k+1;
     return(bin(s,i,j,key));
}
main()
{    int i,s[6]={80,22,1,0,3,7};
     sel(s,5); //devo prima ordinarlo!!
     for(i=0;i<6;i++) printf("%d ",s[i]);printf("\n");
     printf("indice %d\n",bin(s,0,5,8));
}
```

- Relazione di ricorrenza: $T(n)=1+T(n/2)$. Per sostituzione, $n=2^k \rightarrow k=\lg n$

  $T(2^k) = T(2^{k-1}) +1 = T(2^{k-2})+2=\ldots=T(2^{k-k})+k=T(1)+\lg n \rightarrow O(\lg n)$

Ordinamento merge-sort

- Fusione di due array ordinati (merge)

```
void merge(int a[], int na, int b[], int nb, int c[])
{
   int ia = 0, ib = 0, ic = 0;

   while (ia < na && ib < nb)
      if (a[ia] < b[ib])
         c[ic++] = a[ia++];
      else
         c[ic++] = b[ib++];
   while (ia < na)
         c[ic++] = a[ia++];
   while (ib < nb)
         c[ic++] = b[ib++];
}
```

- Complessita': O(n)

Ordinamento merge-sort

- Suddivisione dell'array di ingresso a meta' fino ad arrivare ad 1 elemento

```
     void msort(int a[], int n){
int i, left, right, p[N];

if (n<2)  return;

msort(a, left = n/2);
msort(&a[left], right = n-left);

copy_array(a, p, left);
merge(p, left, &a[left], right, a);
}
```

- Complessita': T(n) = 2 T(n/2) + cn (ricorsione + merge)
- Per sostituzione ($n=2^k$, $k=\lg n$):

$T(2^k) = 2T(2^{k-1})+c2^k = 2(2T(2^{k-2})+c2^{k-1})+c2^k =$

$= 2(2(2T(2^{k-3})+c2^{k-2})+c2^{k-1})+c2^k =$

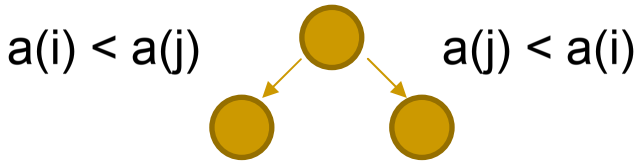
$= 2^3T(2^{k-3})+2^2c2^{k-2}+2c2^{k-1}+c2^k = 2^3T(2^{k-3})+2c2^k = \ldots =$

$= 2^kT(1)+kc2^k = 2^{\lg n}(T(1)+c\lg n) = n^{\lg 2}(T(1)+c\lg n) =$

$= nT(1)+ cn\lg n \rightarrow O(n\lg n)$

Complessità inferiore dell'ordinamento

- Ipotesi: n numeri da ordinare, tutti distinti
- Osservazioni
  - Ordinare n numeri vuol dire cercare una particolare permutazione
  - n numeri → n! permutazioni
  - Ogni algoritmo può essere associato a un albero binario:



  - Ogni foglia rappresenta una permutazione → almeno n! foglie
  - Profondità dell'albero = numero di confronti
  - In generale numero_foglie = $2^{\text{profondità}}$ → profondità=log(numero_foglie)
  - Nel caso dell'ordinamento, numero_foglie = n!
  - ➔ numero confronti=profondità=log(n!)=log($(n/e)^n$)=nlogn - nlogn=O(nlogn)

Prove per induzione

- Tecnica per provare la correttezza di semplici procedure
- Dimostrazione di ricorrenze
- Idea:
  - Verificare la ricorrenza in un caso iniziale
  - Dimostrare che ogni volta la ricorrenza e' vera in un caso ristretto, e' vera anche per un caso successivo → e' vera sempre
- Esempio, merge-sort: $T(n)=2T(n/2)+cn$ (**a**) → $T(n)=cn\lg n+nT(1)$ (**b**)

  - (**b**) e' verificata per n=1 (da (**b**), T(n)=T(1), come deve essere da (**a**))

  - supponiamo (**b**) sia vera per n generico.

  - mostriamo che, allora, (**b**) e' vera per 2n:

  $T(2n)=2T(n)+c\,2n=2(cn\lg n+nT(1)) + c\,2n=$ (perche' (**b**) e' vera per n)

  $=2cn\lg n+2nT(1)+2cn=2cn(\lg n+1) + 2nT(1)=$

  (nota: $\lg 2n=\lg 2+\lg n=1+\lg n$)

  $=2cn\lg 2n+2nT(1)=c2n\lg 2n+2nT(1)$

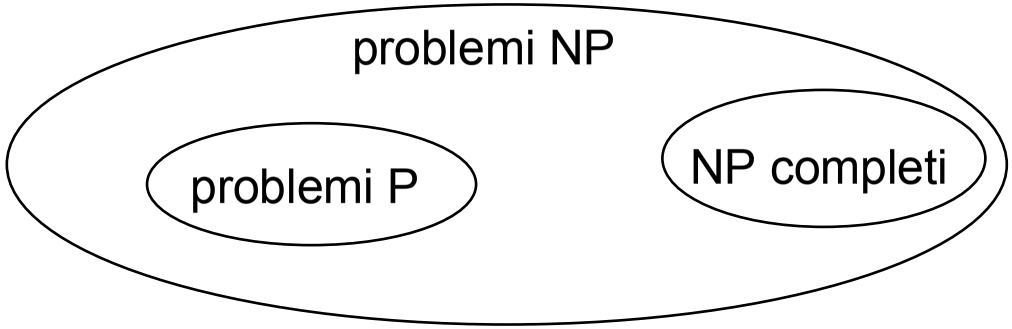
  → (**b**) vale per 2n → sempre vera

Prove per induzione - esercizi

- Esercizi:
  - dimostrare $\sum_{k=0}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$
  - dimostrare $\sum_{k=0}^{n} k^3 = \frac{n^2(n+1)^2}{4}$
  - dimostrare $a + ar + ar^2 + \cdots + ar^n = \frac{a(1-r^{n+1})}{1-r}$
  - dimostrare $n^2 \geq 2n + 1$
  - I numeri armonici H_1, H_2, H_3, ... sono definiti da $H_j = 1 + \frac{1}{2} + \frac{1}{3} + \cdots + \frac{1}{j}$
    - dimostrare che $1 + \frac{n}{2} \leq H_{2^n} \leq 1 + n$ ..

Classi di complessità

- Quali problemi sono computabili in tempo polinomiale? Ordinamento, ricerca, prodotto matriciale, ecc.
- *Classe di complessità **P** = insieme dei problemi che sono risolvibili in tempo polinomiale*
- Molti problemi non sono computabili in tempo polinomiale: bisogna dimostrare che non esistono algoritmi polinomiali
- Se per un un problema non esistono soluzioni polinomiali, ma esistono verifiche polinomiali, allora il problema è classificato **NP**
  - esempio: dato un insieme di interi, ci si chiede se esiste un suo sottoinsieme di un certo numero di elementi la cui somma sia zero
  - la ricerca del sottoinsieme è laboriosa, ma la verifica è veloce



Classi di complessità: esempi di problemi

- Esempio: scacchi → tempo esponenziale(se si scoprisse un algoritmo 8x8, si generalizzi nxn)
- Esempio: stivaggio contenitori → tempo esponenziale (C camion che portano un peso P, bastano per trasportare N casse di peso diverso?)
- Esempio: commesso viaggiatore → tempo esponenziale (visita di N citta' entro un dato chilometraggio, visitando una sola volta ogni citta')
- Esempio: ciclo di Hamilton → tempo esponenziale (visita di una sola volta dei nodi di un grafo)
- Esempio: stesura di un orario→ tempo esponenziale
- Soluzioni semplici: approssimazioni, soluzioni mediamente veloci (non nel caso peggiore), sviluppo di algoritmi che si sa che contengono un errore

Problemi NP-completi

- Tutti i problemi fattibili sono **NP**
- Anche i problemi non fattibili sono **NP → P = NP ??**
- Tra i problemi **NP** non fattibili, ce ne sono alcuni piu' difficili:
  → Problemi NP-completi
- Se si trova un algoritmo polinomiale per uno di questi, esiste un algoritmo polinomiale per ogni problema **NP**
- Esempio: stivaggio contenitori, ciclo di Hamilton, commesso viaggiatore, schedulazione sono **NP-completi**
- I problemi **NP-completi** non sono fattibili
- Se e' difficile trovare una soluzione ad un problema, si puo' provare a dimostrare se e' **NP-completo**
- Conclusione:
  - problemi fattibili → algoritmo polinomiale
  - Problemi NP → verifica polinomiale
  - Problemi NP-completi → molto probabilmente non ammettono algoritmo veloci

Bibliografia

- Generalità
  - Fondamenti di programmazione dei calcolatori elettronici, Batini, Carlucci Aiello, Lenzerini, Marcetti Spaccamela, Miola, Franco Angeli 2002
- Calcolabilità
  - Modelli di calcolo e calcolabilità, Marchetti Spaccamela, Protasi, Franco Angeli 1988
- Complessità
  - Introduzione agli algoritmi-Vol.3, Cormen, Leiserson, Rivest,Jackson Libri 1995
  - Linguaggi, Modelli, Complessità, Ausiello, D'Amore, Gambosi, Franco Angeli 2003