Aniello Murano

Automi e Pushdown

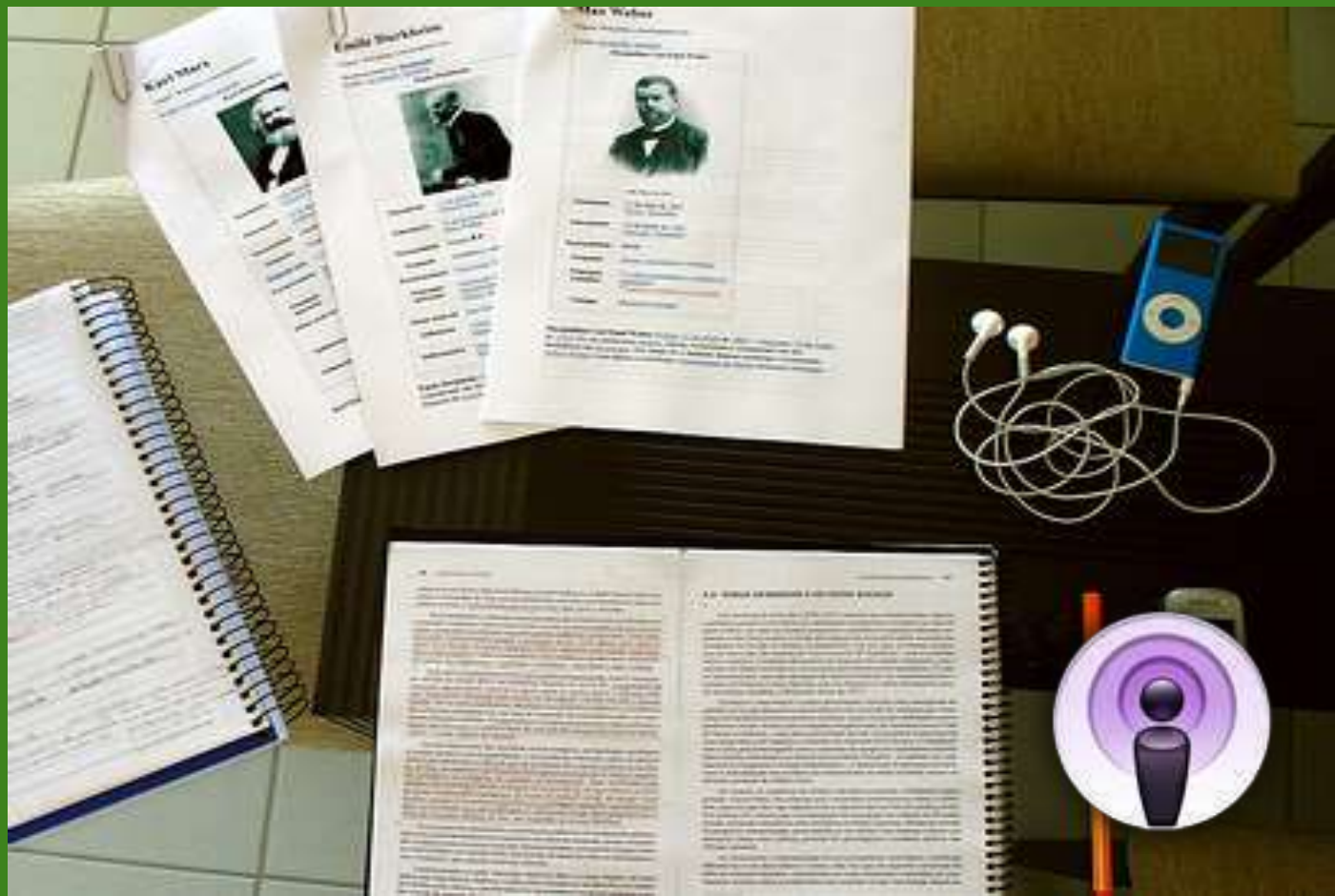
Lezione n.2
Parole chiave:
Automi e PDA

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Calcolabilità, complessità e macchine computazionali

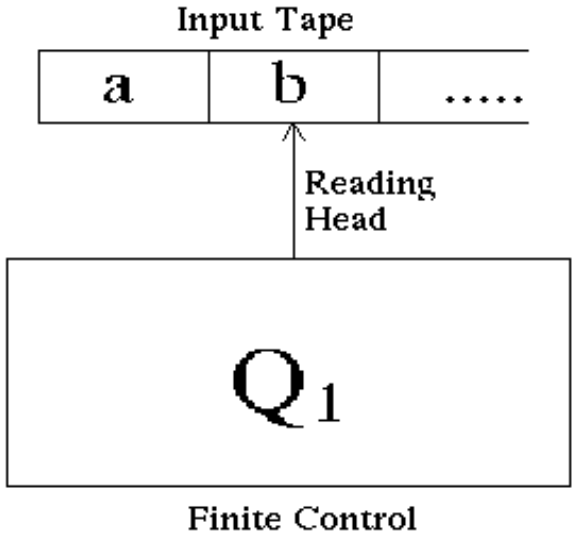
- Per presentare in modo formale i concetti basilari delle teorie della calcolabilità e della complessità, abbiamo innanzitutto bisogno di una definizione precisa di cosa è un computer.
- I computer di uso quotidiano, come un personal computer, sono troppo complessi per essere utilizzati come modelli
- Si preferisce invece utilizzare delle macchine computazionali "semplificate".
- Esistono diverse macchine che si possono usare, con diversa capacità computazionale
- Solitamente, all'aumentare della loro capacità computazionale, aumenta la difficoltà gestionale e valutativa di queste macchine.
- Tra le macchine computazionali più semplici, particolarmente adatte al nostro scopo, ci sono gli automi.
- Sebbene gli automi siano costituiti da una memoria finita, esistono svariati contesti reali in cui essi sono impegnati con successo.

Perchè studiare la teoria degli automi?

- La teoria degli automi e dei linguaggi formali è lo studio di dispositivi computazionali o "macchine" computazionali.
- Gli automi, originariamente, furono proposti per creare un modello matematico che riproducesse il funzionamento del cervello.
- La teoria degli automi è alla base della descrizione e della modellazione dei linguaggi di programmazione, della costruzione dei loro riconoscitori e traduttori, della realizzazione di strumenti di elaborazione testuale.
- Ecco alcuni esempi in cui gli automi finiti sono possono essere utilizzati come modelli per il software:
  - software per la progettazione e la verifica del comportamento dei circuiti digitali;
  - "analizzatore lessicale" di un compilatore;
  - software che eseguono una scansione di testi molto lunghi per trovare parole, frasi, ecc.;
  - software per verificare i protocolli di comunicazione o protocolli per lo scambio sicuro di informazioni.
- Riferimenti: Hopcroft, J.E., Ullman J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, Mass., 1979.

Descrizione

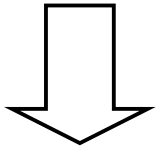
- Un automa è un dispositivo sequenziale creato per eseguire un particolare compito che, ad ogni istante, può trovarsi in un determinato "**stato**".
- Quando l'automa si trova in uno stato, può eseguire una transizione (in un altro stato) in base ad un input esterno, rappresentato da un simbolo del suo alfabeto
- Lo scopo dello stato è quello di ricordare la parte rilevante della **storia** del sistema. Dunque lo stato funge da **memoria.**
- Il modello di un automa consiste di un dispositivo di controllo, con un numero finito di stati, una testina di lettura e un nastro infinito diviso in celle.



Rappresentazione grafica di un automa

Descrizione(2)

- Gli automi sono spesso utilizzati per descrivere linguaggi formali, e per questo sono chiamati accettori o riconoscitori di un linguaggio.
- L'evoluzione di un automa parte da un particolare stato detto **stato iniziale**.
- Un sottoinsieme privilegiato degli stati di un automa è detto insieme degli **stati finali** e corrispondono agli stati "accettanti".
- Una sequenza di simboli (detto anche stringa o parola) appartiene al linguaggio accettato da un automa se essa porta l'automa, in un certo numero di passi, dallo stato iniziale ad uno accettante.
- A diverse classi di automi corrispondono diverse classi di linguaggi, caratterizzate da diversi livelli di complessità.
- Analizziamo in dettaglio i vari tipi di automi



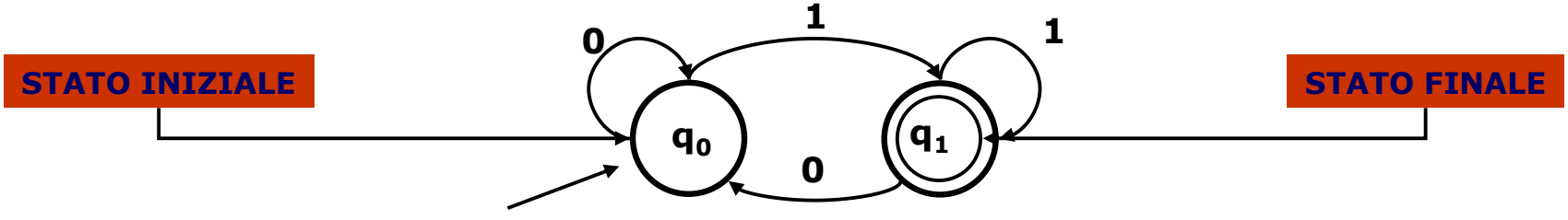
Definizione di un DFA

- Un "**deterministic finite automaton**" o "**DFA**" (automa a stati finiti determini-stico) è formalmente definito come una quintupla (Q,Σ,δ,q_0,F) dove

  - Q è un insieme finito di **stati**
  - Σ è un insieme finito di caratteri che costituiscono l'**alfabeto**
  - $\delta : Q \times \Sigma \rightarrow Q$ è la **funzione di transizione** che associa ad ogni coppia (stato, carattere) uno stato
  - $q_0 \in Q$ è lo **stato iniziale**
  - F sottoinsieme di Q è l'insieme degli **stati finali**

Run di accettazione di un DFA

- Un DFA $A = (Q,\Sigma,\delta,q_0,F)$ accetta una stringa $u = u_1\ u_2\ \dots\ u_n$ se (e solo se) c'è una sequenza di stati $r = r_1,\ r_2, \dots, r_n,\ r_{n+1}$ tale che:
  - $r_1 = q_0$
  - $r_{i+1} = \delta(r_i,u_i)$, per ogni i con $1 \leq i \leq n$
  - $r_{n+1} \in F$
- Il linguaggio **accettato** da un DFA è l'insieme di tutte le **parole** (stringhe) formate con i simboli di Σ per il quale l'automa partendo dallo stato iniziale raggiunge lo stato finale alla lettura dell'ultimo simbolo della parola.
- Un linguaggio è **regolare** se esiste un DFA che accetta tutte e sole le parole che esso contiene.
- **Esempio:** In figura è mostrato l'automa che accetta il linguaggio di tutte le parole composte di 0 e 1 e che terminano con 1.

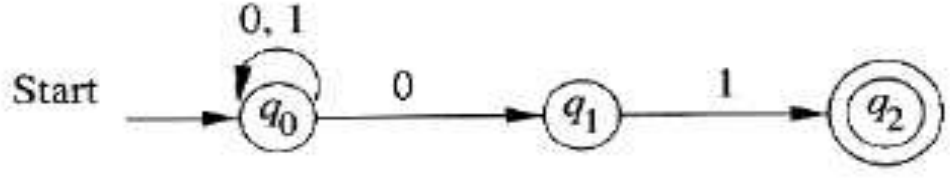


Esercizio

- Scrivere un DFA che accetti il linguaggio di tutte le parole composte di 0 e 1 e che contengono un numero pari di 1.
- Scrivere un DFA che accetti il linguaggio di tutte le parole composte di 0 e 1, il cui numero binario corrispondente è pari.
- Scrivere un DFA che accetti il linguaggio di tutte le parole composte di 0 e 1 e che terminano con “01”.

Definizione di un NFA

- Un "**nondeterministic finite automaton**" o "**NFA**" (automa a stati finiti non-deterministico) differisce da un DFA per la definizione della funzione di transizione.
- Un NFA, stando in uno stato e leggendo un simbolo può andare in più stati.
- Il funzionamento di un NFA si può immaginare come se esistessero più copie dello stesso automa.
- Formalmente, un NFA è definito come per i DFA da una quintupla (Q,Σ,δ,q_0,F), con l'unica differenza che la δ è così definita:
  - $\delta : Q \times \Sigma \rightarrow 2^Q$ (con 2^Q denotiamo l'insieme di tutti i sottoinsiemi di Q)
- Nota: data una parola su Σ, un NFA può avere una, nessuna o più sequenze di stati associati dalla funzione di transizione.
- Una parola è accettata da un NFA se almeno una delle sequenze di stati associate dalla δ terminano in uno stato finale.
- **Esempio:** In figura è mostrato l'automa che accetta il linguaggio di tutte le parole composte di 0 e 1 e che terminano con "01".



Confronto tra run di un DFA e di un NFA

$A = (Q, \Sigma, \delta, s, F)$

Quando A è un DFA

A ***accetta*** la stringa

$u_1\ u_2\ \ldots\ u_n$

se c'è una sequenza di stati

$r_1,\ r_2, \ldots, r_n,\ r_{n+1}$

tale che:

- $r_1 = s$
- $r_{i+1} = \delta(r_i, u_i)$, per ogni ***i*** con $\mathbf{1 \leq i \leq n}$
- $r_{n+1} \in F$

Quando A è un NFA

A ***accetta*** la stringa

$u_1\ u_2\ \ldots\ u_n$

se c'è una sequenza di stati

$r_1,\ r_2, \ldots, r_n,\ r_{n+1}$

tale che:

- $r_1 = s$
- $r_{i+1} \in \delta(r_i, u_i)$, per ogni ***i*** con $\mathbf{1 \leq i \leq n}$
- $r_{n+1} \in F$

Equivalenza tra DFA e NFA

- Due automi A e B sono detti equivalenti se accettano lo stesso linguaggio, ovvero L(A)=L(B)
- **Teorema: Per ogni NFA N esiste un DFA D equivalente.**
- Per provare questo teorema si usa la **subset construction:**
  - gli stati di D rappresentano tutti i possibili sottoinsiemi degli stati raggiungibili in N, sulla stessa parola.

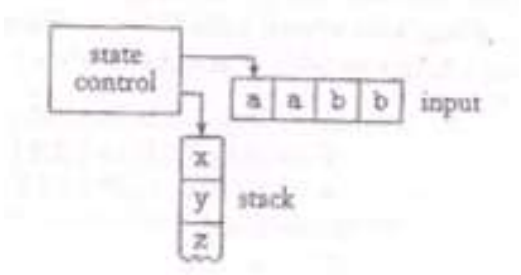
- Sia N = (Q, Σ, δ, s, F) un NFA, la subset construction permette di costruire un DFA D = (Q', Σ, δ', s', F') nel modo seguente:
  - Q' = P (Q)
  - $\delta'(R,a) = \{q \mid q=\delta(p,a) \text{ per ogni } p \in R\}$
  - s' = {s}
  - F'= {R | R è un sottoinsieme di Q che contiene uno stato finale di N}

- **Osservazione:** La subset constuction è una traduzione esponenziale!

- Scrivere un NFA e un DFA per il linguaggio accettante tutte le parole terminanti per “001”
- Scrivere un NFA e un DFA per il linguaggio accettante tutte le parole che hanno un “1” nella terzultima posizione.
- Scrivere un DFA per il linguaggio D = {w | w contiene lo stesso numero di “01”e “10”}. Per esempio 010 è in D

- Come si è detto precedentemente, gli NFA accettano (tutti e solo) i linguaggi regolari.
- Questa classe gode della proprietà di essere **chiusa** rispetto alle operazioni di unione, intersezione e complemento [Prova lasciata per esercizio agli studenti]
- Ogni linguaggio appartenente a questa classe può essere costruito tramite una **grammatica regolare**
- Ogni linguaggio appartenente a questa classe può essere univocamente determinato tramite una **espressione regolare**.
- Gli ultimi due concetti, sebbene siano marginali per questo corso, sono fondamentali per uno studio completo sulla teoria degli automi.
- Informazioni più dettagliate possono essere reperite dal libro di testo consigliato: Hopcroft, J.E., Ullman J.D., *Introduction to Automata Theory, Languages, and Computation*, Addison Wesley, Reading, Mass., 1979.

- Si consideri il linguaggio L = $\{a^n b^n$ con $n>1\}$
- Domanda: Questo linguaggio può essere accettato da un automa a stati finiti?
- Risposta: No! Gli automi a stati finiti posseggono una memoria finita e quindi non sono in grado di riconoscere linguaggi che, per la loro struttura, necessitano di ricordare una quantità di "informazioni" non limitate.
- Il linguaggio in oggetto può essere accettato da un **Pushdown Automaton** o **PDA**
- Un PDA è un automa a stati finiti con stack. Lo stack è un contenitore (anche infinito) che rispetta un accesso LIFO ed ha capienza infinita.
- Ad ogni transizione di un PDA è dunque possibile leggere il contenuto di una cella del nastro ed il simbolo in cima allo stack, passare in un nuovo stato e sostituire il simbolo letto sul top dello stack con una stringa.



Schema di un automa pushdown

- I PDA sono stati introdotti da Oettinger nel 1961 e da Schutzenberger nel 1963

- Schutzenberger (1920-1996) è stato un ricercatore e un punto di riferimento per varie generazioni di ricercatori per i linguaggi formali e l'informatica teorica. Ha insegnato alla Facoltà di Scienze dell'Università di Parigi (VI e VII).

Definizione di PDA

- Un PDA nondeterministico è formalmente definito da una 6-pla $(Q,\Sigma,\Gamma,\delta,q_0,F)$ dove
  - Q è un insieme finito di **stati**
  - Σ è un insieme finito di caratteri che costituiscono l'**alfabeto del nastro**
  - Γ è un insieme finito di caratteri che costituiscono l'**alfabeto dello stack**
  - $\delta : Q \times \Sigma_\varepsilon \times \Gamma_\varepsilon \rightarrow P(Q \times \Gamma^*)$ è la **funzione di transizione** che associa ad ogni tripla (stato, carattere letto sul nastro, carattere letto sul top dello stack) un insieme di possibili coppie (stato, stringa), dove la stringa sostituisce il top dello stack
  - $q_0 \in Q$ è lo **stato iniziale**
  - F sottoinsieme di Q è l'insieme degli **stati finali**

Passi di Calcolo

- Se (q,B) ϵ δ(p,a,A), con a ϵ Σ_ε, A ϵ Γ_ε e B ϵ Γ_ε allora è possibile che sia eseguito uno dei seguenti passi di calcolo,
  - (POP-PUSH) se a ϵ Σ, A, B ϵ Γ con a in lettura sul nastro di input e A in cima alla pila passa nello stato q, rimpiazza A con B in cima alla pila e muove la testina di lettura del nastro di input di una cella verso destra.
  - (PUSH) se a ϵ Σ, A = ε , B ϵ Γ allora, con a in lettura sul nastro di input e indipendentemente dal simbolo in cima alla pila, passa nello stato q, impila B in cima alla pila e muove la testina di lettura del nastro di input di una cella verso destra.
  - (POP) se a ϵ Σ, A ϵ Γ, B = ε allora, con a in lettura sul nastro di input e A in cima alla pila, passa nello stato q, elimina A dalla cima della pila e muove la testina di lettura del nastro di input di una cella verso destra.
- Se a = ε , i tre tipi di mosse avvengono senza che la testina di lettura si sposti.

Configurazione

- Come nel caso degli NFA, il concetto di configurazione è utile per descrivere la situazione in cui si trova la macchina complessivamente:
  - lo stato,
  - l'input ancora da esaminare e
  - il contenuto della pila.
- Quindi una configurazione è un elemento di $Q \times \Sigma^* \times \Gamma^*$.
- La configurazione iniziale è (q_0,x,ε)

Accettazione

- Un PDA $M = (Q,\Sigma,\Gamma,\delta,q_0,F)$ accetta una stringa $w= w_1w_2 \ldots w_n$, con $w_i \in \Sigma_\varepsilon$, se esistono una sequenza di stati, $r_0, r_1,\ldots,r_n$ di Q e una stringa $s_0, s_1, \ldots, s_m$ di Γ^* che soddisfino le seguenti tre condizioni:
  1. $r_0 = q_0$ e $s_0 = \varepsilon$.
  2. Per $i = 0, \ldots, m - 1$, si ha che $(r_{i+1}, b) \in \delta(r_i, w_{i+1}, a)$, dove $s_i = a \cdot t$ e $s_{i+1} = b \cdot t$, per qualche a,b in Γ_ε e t in Γ^*.
  3. r_n appartiene a F.

- Un esempio di PDA che accetta il linguaggio $L=\{0^n1^n|n\geq 0\}$ è il seguente:

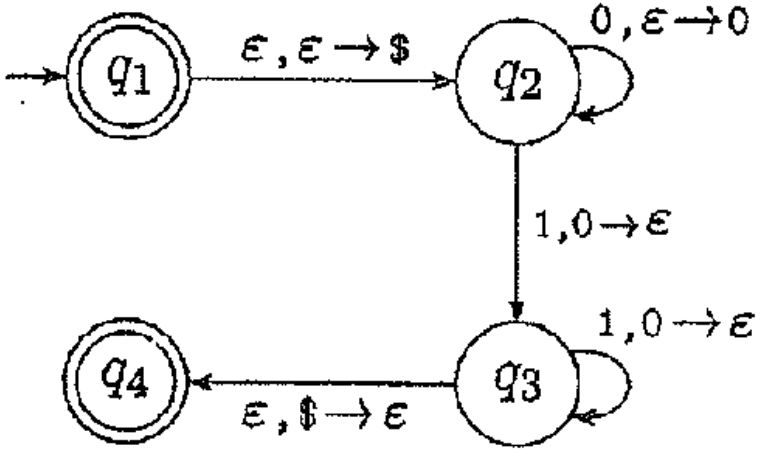
$Q = \{q_1, q_2, q_3, q_4\}$,

$\Sigma = \{0,1\}$,

$\Gamma = \{0, \$\}$,

$F = \{q_1, q_4\}$, and

Input:	0			1			ε		
Stack:	0	$	ε	0	$	ε	0	$	ε
q_1									$\{(q_2, \$)\}$
q_2			$\{(q_2, 0)\}$		$\{(q_3, \varepsilon)\}$				
q_3					$\{(q_3, \varepsilon)\}$			$\{(q_4, \varepsilon)\}$	
q_4									



Proprietà dei PDA

- I PDA sono schiusi rispetto all'unione, ma **non** rispetto al complemento e all'intersezione.
- I PDA sono invece chiusi rispetto all'intersezione e al complemento con linguaggi regolari, cioè l'intersezione (o il complemento) di un linguaggio accettato da un PDA con un linguaggio regolare è sempre accettato da un PDA.
- Anche per i PDA esiste la possibilità di usare delle grammatiche in grado di generare tutti i soli i linguaggi da essi accettati: le **grammatiche context-free**

Esercizi

- Definire un PDA che accetti il linguaggio $\{a^n b^n$, con $n>1\}$
- Definire un PDA che accetti il linguaggio: $\{a^i b^j c^k \mid i,j,k > 0$ e $i = j$ oppure $i = k\}$.
- Definire un PDA che accetti il linguaggio: $\{ww^R \mid w \in \{0,1\}^*\}$. "R sta per reverse" (linguaggio delle palindrome)

PDA deterministici

- Un PDA $A = (Q,\Sigma,\Gamma,\delta,q_0,F)$ è deterministico (DPDA) se
  - per ogni $q \in Q$, $a \in \Sigma_\varepsilon$ e $B \in \Gamma$, abbiamo che $card(\delta(q,a,B)) \leq 1$,
  - per ogni $q \in Q$ e $B \in \Gamma$, se $\delta(q,\varepsilon,B) \neq \emptyset$, allora $\delta(q,a,B) = \emptyset$ per ogni $a \in \Sigma$

- Dunque, se c'è una ε-mossa eseguibile in un certo stato e con un certo simbolo in cima alla pila allora quella è l'unica mossa eseguibile in quello stato e con quel simbolo in cima alla pila.

- A differenza del caso dei DFA, esistono linguaggi accettati da un PDA che non possono essere accettati da un DPDA (per esempio il linguaggio delle palindrome)

- Quindi L(PDA), la classe dei linguaggi accettati dai PDA, contiene strettamente L(DPDA), la classe dei linguaggi accettati dai PDA deterministici,

- Per quanto un PDA possa essere più potente in un NFA, esistono linguaggi che non possono essere riconosciuti da un PDA. Per esempio, il linguaggio $\{a^n b^n c^n$, con $n>1\}$ necessita di una **macchina di Turing** per essere accettato.
- Le macchine di Turing saranno oggetto delle prossime lezioni.

Aniello Murano

Macchine di Turing

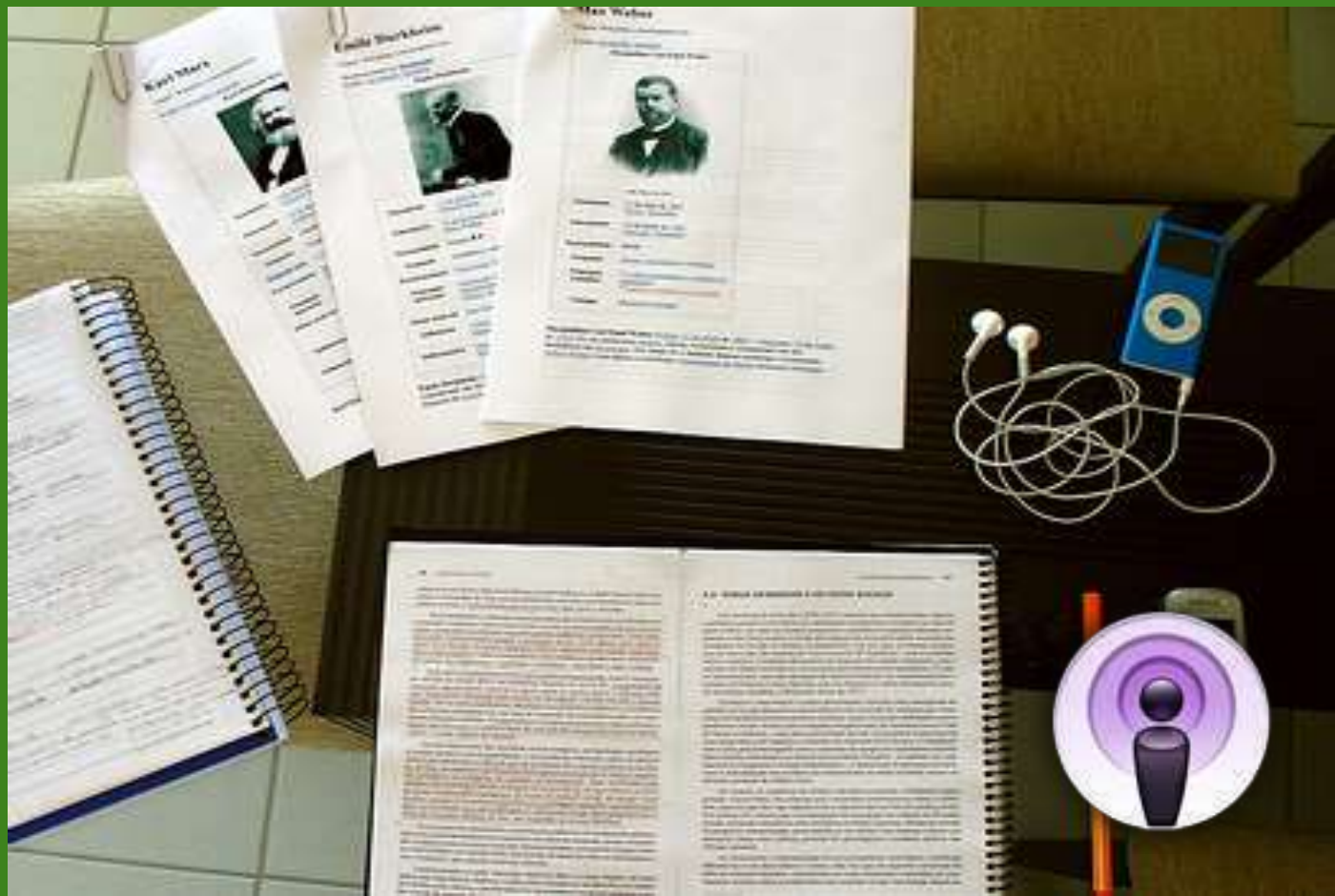
Lezione n.3
Parole chiave:
Turing machine

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Potere computazionale di MdT

- Tra le macchine computazionali, la **Macchina di Turing** (**MdT**) è senza dubbio la più potente
- In letteratura, è stata dimostrata l'equivalenza computazionale (ossia in grado di effettuare le stesse elaborazioni) con diversi altri modelli di calcolo di più ampia portata come:
  - le funzioni ricorsive di Jacques Herbrand e Kurt Gödel,
  - il lambda calcolo di Alonzo Church e Stephen Kleene,
  - la logica combinatoria di Moses Schönfinkel e Haskell Curry,
  - gli algoritmi di Markov,
  - i sistemi di Thue,
  - i sistemi di Post,
  - le macchine di Hao Wang
  - le macchine a registri elementari o RAM astratte di Marvin Minsky.
  - ….
  - ….
- **TESI DI CHURCH-TURING:**

  Per ogni problema calcolabile esista una MdT in grado di risolverlo

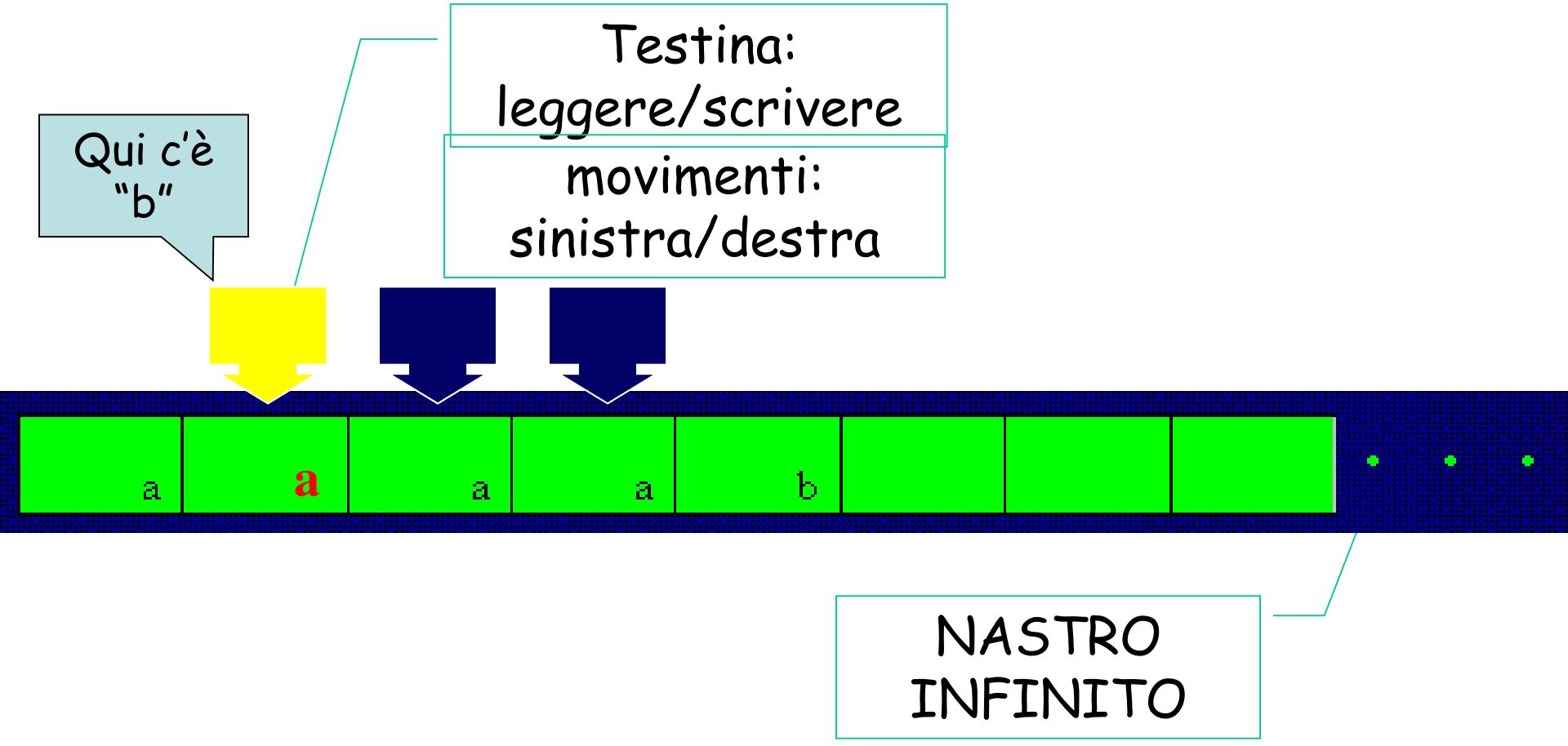
- La MdT come modello di calcolo è stato introdotta nel 1936 da Alan Turing.
- Nel 1936 venne pubblicato un articolo di Alan Turing intitolato "On computable numbers, with an application to the Entscheidungsproblem", in cui l'autore risolveva negativamente il problema della decidibilità (Entscheidungsproblem) lanciato nel 1900 da David Hilbert e Wilhelm Ackermann che recitava così:

  «esiste sempre, almeno in linea di principio, un metodo meccanico (cioè una maniera rigorosa) attraverso cui, dato un qualsiasi enunciato matematico, si possa stabilire se esso sia vero o falso?»
- L'utilità di un tale algoritmo sta nel fato che sarebbe in grado di risolvere tutti i problemi matematici e ancora più importante che ogni ragionamento umano poteva essere ridotto a mero calcolo meccanizzabile.
- E' da citare anche che una prima risposta negativa al problema della decidibilità la diede il matematico Gödel nel 1931 con il lavoro sull'incompletezza dei sistemi formali coerenti ("primo teorema di incompletezza");
- Gödel dimostrò che la semplice coerenza di un sistema formale non può garantire che ciò che in esso viene dimostrato sia vero oppure falso.

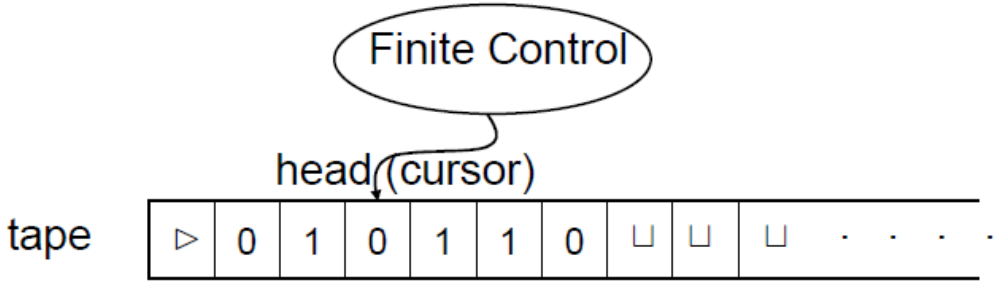
Descrizione della macchina di Turing

- Una macchina di Tuding opera su un nastro (potenzialmente infinito) che come per gli NFA si presenta come una sequenza di caselle nelle quali possono essere registrati simboli di un ben determinato alfabeto finito.
- A differenza degli NFA, la testina di una MdT può, oltre che leggere, anche scrivere sul nastro. Per questo motivo la testina viene chiamata di I/O. Inoltre, la testina può spostarsi a destra o a sinistra.
- Ogni passo dell'evoluzione viene determinato dallo stato attuale *s* nel quale la macchina si trova e dal carattere che la testina di I/O trova sulla casella del nastro su cui è posizionata e si concretizza nell'eventuale modifica del contenuto della casella, nell'eventuale spostamento della testina e nell'eventuale cambiamento dello stato.
- Una **evoluzione** della macchina consiste in una sequenza di sue possibili "**configurazioni**".
- L'evoluzione di una MdT può sia arrestarsi in una certa configurazione (che può essere "utile" o "meno inutile" oppure può anche accadere che l'evoluzione non abbia mai fine.

Funzionamento schematico di una MdT



MACCHINA DI TURING: definizione formale



- Una MdT (deterministica) è formata da una 7-pla (Q, Σ, Γ, δ, q_0, accept, reject):
  - **Q** è un insieme finito di **stati**;
  - **STATI SPECIALI**: q_0 è lo **stato iniziale**
    "accept" è lo **stato accettante**;
    "reject" è lo **stato rifiutante**;
  - **Γ** è l'**alfabeto del nastro**. Include il simbolo speciale "-" oppure "␣" (simbolo di blank);
  - **Σ** ⊆ **Γ** \{-} è l'**alfabeto di input**;
  - **δ** : Q\{accept, recject} × Γ → Q× Γ ×{L,R} è la **funzione di transizione** (parziale).
- Nota: accept ≠ reject e sono degli **halting states,** cioè non esistono transizioni da questi stati.
- **Osservazione**: nella definizione classica, la MdT ha anche un simbolo speciale di starting sul nastro, uno stato speciale aggiuntivo "halt" di halting e la testina ha la possibilità di rimanere ferma in una transizione. Queste modifiche non alterano il potere computazionale della macchina. Talvolta, noi useremo queste varianti per semplicità.

Funzionamento di una MdT

- La funzione di transizione $\delta : Q\backslash\{accept, recject\} \times \Gamma \rightarrow Q \times \Gamma \times \{L,R\}$ permette alla MdT, a partire da un certo stato e leggendo un simbolo sul nastro, di transire in un nuovo stato, sostituire il simbolo letto sul nastro, e muovere la testina di lettura a sinistra o a destra.

- INIZIALMENTE: Stato=q_0 ; nastro in input … - - $w_1 \ldots w_n$ - -… testina posizionata su w_1 (su $\triangleright$ se esiste, che si troverà subito a sinistra di w_1);

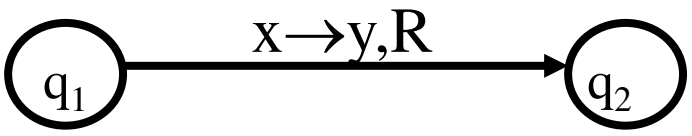
- AD OGNI PASSO: nuova configurazione conforme alla funzione di transizione δ;

- La testina non può muoversi a sinistra del primo simbolo della parola in input.

- Se si fa uso del marcatore di inizio stringa "$\triangleright$", questo simbolo non viene mai sovrascritto e la testina non va mai oltre questo simbolo.

- Per semplicità, assumiamo che quando la testina si trova negli stati "accept" o "reject" (anche "halt" se previsto) non si muove.

Rappresentazione grafica



La rappresentazione grafica in figura rappresenta la situazione in cui:

- il simbolo corrente sul nastro è x,
- Lo stato corrente è q_1,
- La testina legge x, lo sostituisce con y, si sposta a destra e transisce nello stato q_2

Se x=y, allora l'etichettatura x→x,R si può semplicemente scrivere come x→R.
Se invece di R, ci fosse stato L, allora la testina si sposta a sinistra, a meno che si tratti della prima cella non blank, nel qual caso, la testina non si sposta.

CONFIGURAZIONE DI UNA MdT

- Una **configurazione** di una MdT è data dalle seguenti informazioni:
  1. Stato corrente;
  2. Contenuto del nastro (i simboli appartenenti a Σ)
  3. La posizione della testina.

- Una configurazione può essere rappresentata come una tupla **(u,q,v)** o con la stringa **uqv** dove:
  - q è stato corrente;
  - u ϵ Σ* è la stringa di simboli di Σ che si trova a sinistra della testina;
  - v ϵ Σ* è la stringa di simboli di Σ che si trova a destra della testina;

- Per esempio, $11q_7011$ rappresenta la configurazione dove il nastro è 11011, lo stato corrente è q_7 e la testina è sullo zero.

- Una configurazione u accept v è detta di **accettazione**; u reject v è invece di **rifiuto**

- La relazione di esecuzione tra configurazioni è ($\rightarrow$ si legge "porta a") :
  - $C_1 \rightarrow C_2$: La MdT esegue un passo da C1 a C2;
  - $C_1 \rightarrow^* C_2$: chiusura transitiva - La MdT esegue in 0,1,2 o più passi uno spostamento da C_1 a C_2;

Accettazione di Linguaggi

- Una MdT M ***accetta*** una stringa in input se, prima o poi incontra lo stato "**accept**". Altrimenti la stringa è non accettata. L(M)={x ∈ Σ^* | M su input x raggiunge lo stato "accept"}

- Dunque, l'input è non accettato da M in due casi:

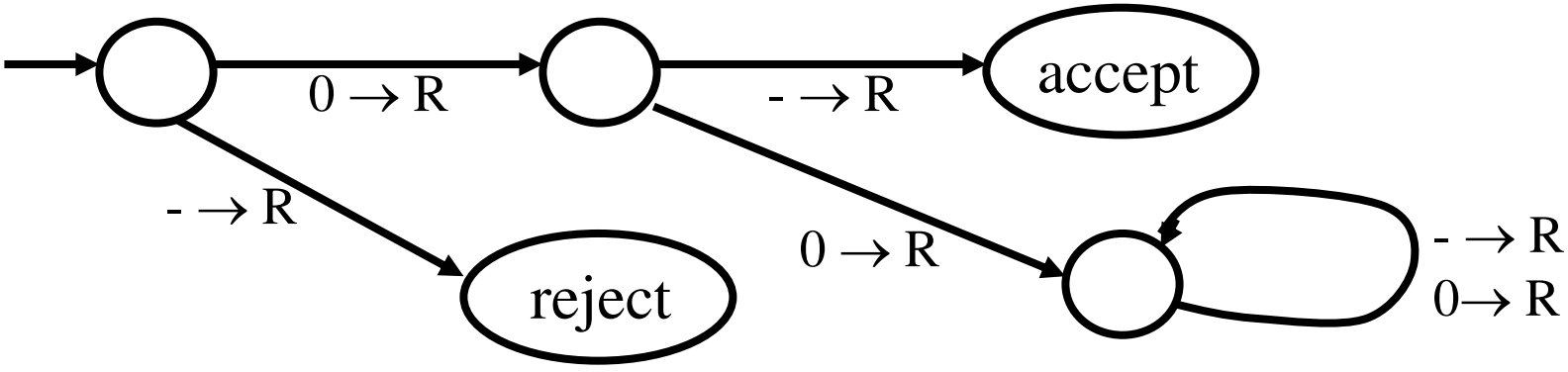
  1. M incontra ad un certo punto lo stato reject;

  2. M non si ferma mai (non incontra mai uno stato di halting)

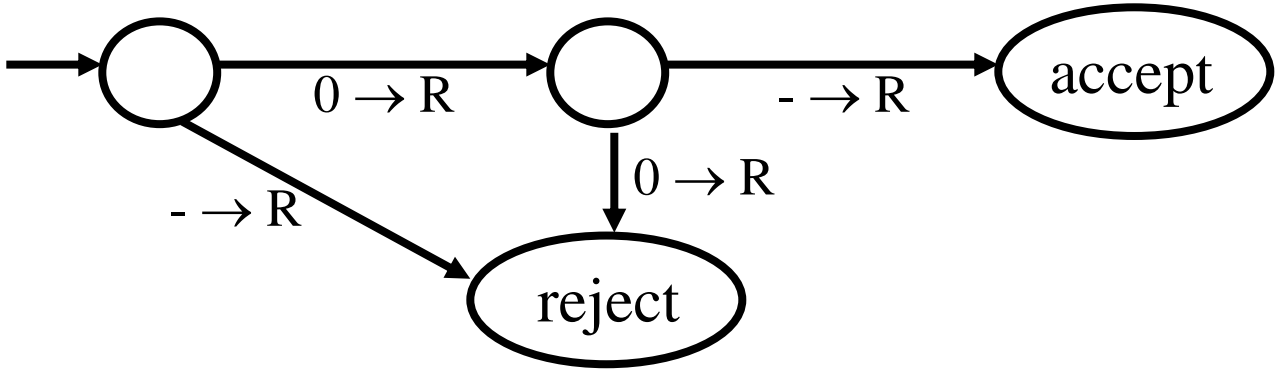
- Il linguaggio **riconosciuto** (accettato) da una MdT è l'insieme delle stringhe che esso accetta

- Una MdT è detta **decisore** se si ferma su ogni input
- Se la macchina è un **decisore**, allora diremo che non solo accetta un linguaggio, ma lo decide anche. Formalmente, M decide un linguaggio $L \subseteq \Sigma^*$ se $L=L(M)$ e per ogni x in Σ^*-L, M termina in uno stato "reject";
- Un linguaggio è detto **Turing-riconoscibile** (o **ricorsivo-enumerabile**) se esiste una MdT che lo riconosce
- Un linguaggio è detto **Turing-decidibile** (o **ricorsivo**) se esiste una MdT che lo decide
- LINGUAGGI RICORSIVI $\subseteq$ LINGUAGGI RICORSIVI ENUMERABILI;

Quale linguaggio riconosce la seguente MdT?
Questo linguaggio è solo accettato o anche deciso?



Quale linguaggio riconosce la seguente MdT?
Questo linguaggio è solo accettato o anche deciso?



Esempio 1

- $L=\{x \in \{0,1\}^* \mid x \text{ ha un numero dispari di } 1\}$

  Metodo: La testina si muove a destra e si mantiene nello stato la parità di 1 letti;

  Stati $Q=\{q_0, q_1, \text{accept}, \text{reject}\}$

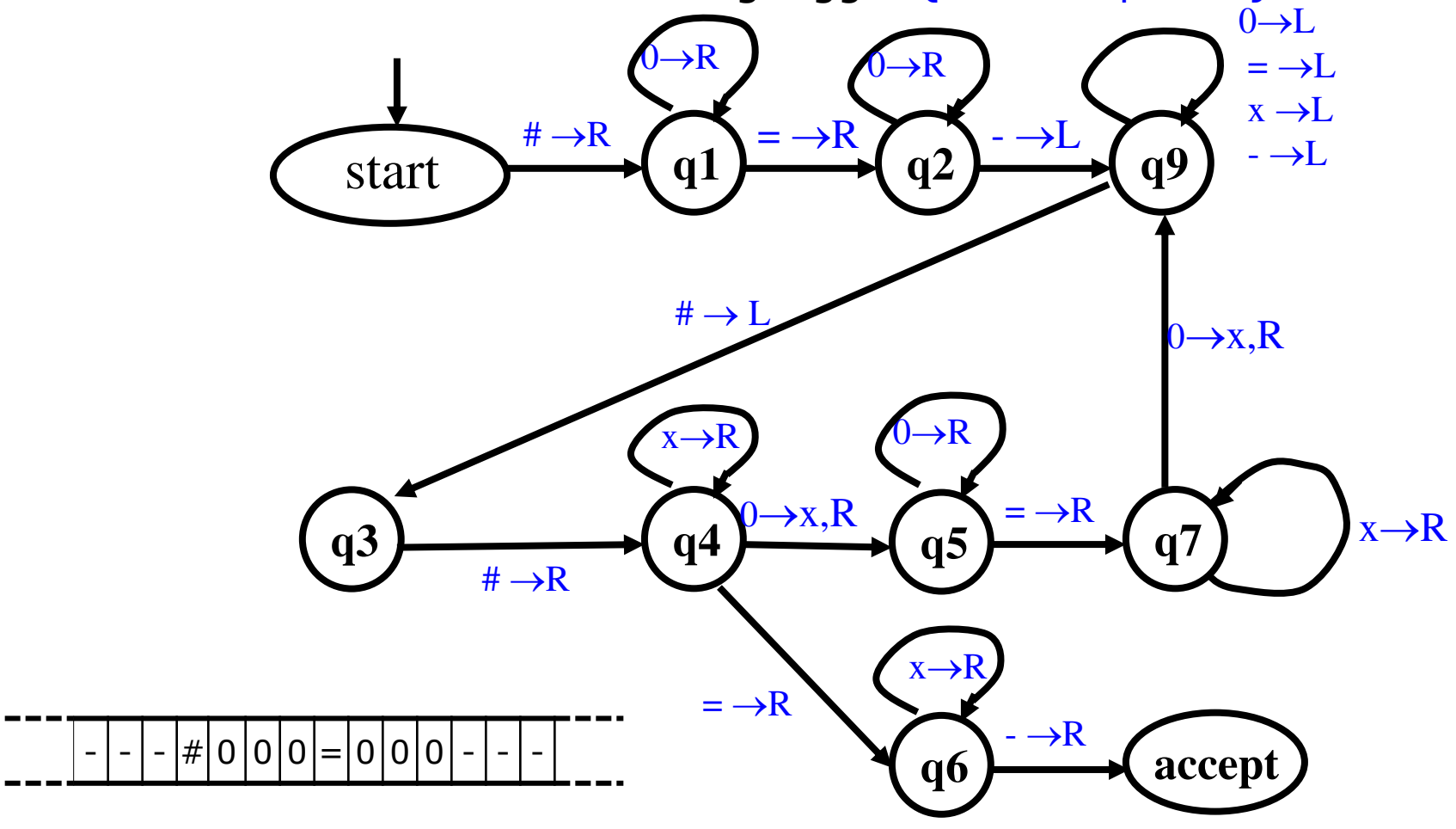
	0	1	-
q_0	$q_0 \rightarrow R$	$q_1 \rightarrow R$	reject$\rightarrow L$
q_1	$q_1 \rightarrow R$	$q_0 \rightarrow R$	accept$\rightarrow L$

Funzione di transizione rappresentata attraverso la tabella di transizione

- Ogni linguaggio regolare può essere analogamente deciso da una TM che muove la testina dall'inizio del nastro verso destra leggendo l'input;

Esempio 2

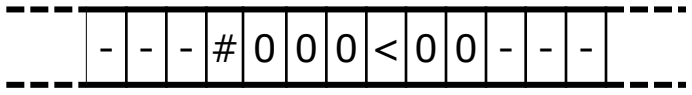
Si definica una MdT che decida il linguaggio $\{\#0^n=0^m \mid n=m\}$



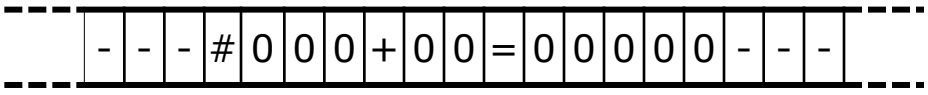
Per semplicità assumiamo che tutte le transizioni mancanti vadano nello stato **reject**

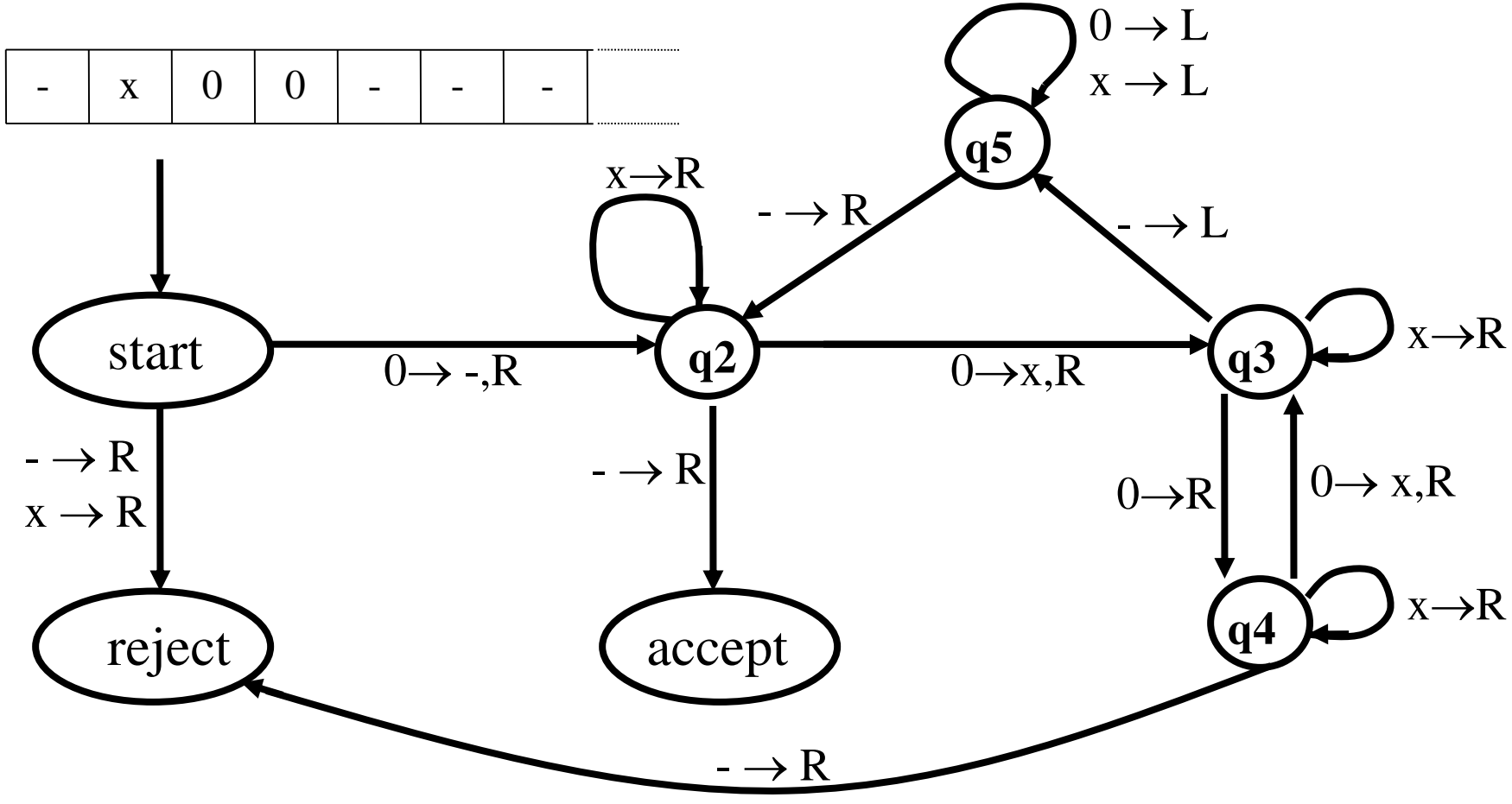
Esercizi

Si definica una MdT che decida il linguaggio $\{\#0^n<0^m \mid n<m\}$



Si definica una MdT che decida il linguaggio $\{\#0^n+0^m=0^k \mid n+m=k\}$





Esempio di un MdT che decide il linguaggio $\{0^{2^n} \mid n > 0\}$ costituito da tutte le parole di 0 la cui lunghezza è una potenza di 2

LA MACCHINA DI TURING COME FUNZIONE COMPUTABILE

- Consideriamo la variante della MdT con lo stato aggiuntivo “halt”

- FUNZIONE COMPUTABILE con M: su input x, se M termina in uno stato di “accept” allora M(x)=“accept”, altrimenti se termina in uno stato di “no” M(x)=“reject”, se invece termina in uno stato “halt” allora M(x)= contenuto del nastro che si trova tra il primo simbolo “blank” (o il simbolo speciale “▷”) e l’insieme infinito di simboli blank “-”;

- Su altri input x (es. quando M gira all’infinito): M(x)= indefinito;

- Per le funzioni possiamo avere per input e output alfabeti differenti;

- Una funzione è ricorsiva se è computabile da qualche Macchina di Turing;

- In pratica, una MdT che calcola una funzione (anche parziale), può essere vista come un **Trasduttore**
- A partire da un input sul nastro, se la macchina ad un certo punto si ferma, l'output è la stringa presente sul nastro in quel momento.
- Questo sarà ancora più evidente quando vedremo le macchine di Turing a più nastri, dove un nastro può essere usato per l'input, uno per l'output e altri nastri per l'esecuzione della macchina
- Esercizio: Definire una macchina trasduttrice che calcoli la funzione prodotto di due interi positivi in notazione unaria, seconde le seguenti convenzioni:
  - Sul nastro memorizzata la stringa $1^n\#1^m$, dove le due sequenze non vuote di 1 rappresentano i numeri da moltiplicare in notazione unaria.
  - Quando la macchina si ferma, sul nastro verrà memorizzata la stringa 1^{nm}.

- INCREMENTO DI UN NUMERO BINARIO: $f(x)=x+1$ dove $x \in \{0,1\}^+$

Metodo: Muovi la testina verso destra fino alla fine(al primo simbolo di ␣)
e vedi se $x=1^n$(tutti i bit=1) :

- se $x=1^n$ allora cambia in 10^n;
- altrimenti sostituisci tutti gli 1 con gli 0 e l'ultimo 0 a 1;

Stati: $Q=\{s,p,q,r,h\}$ (h è accept)

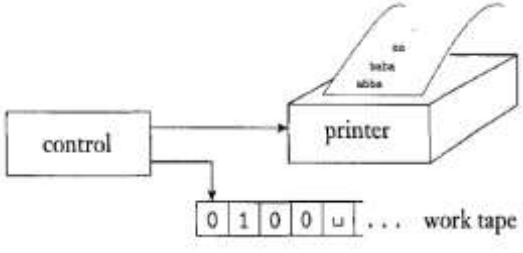
	▷	0	1	␣
s	s,▷,R	p,0,R	s,1,R	q,0,L
p	p,▷,R	p,0,R	p,1,R	q,␣,L
q	r,▷,R	h,1,S	q,0,L	q,␣,L
r	r,▷,R	h,1,S	h,1,S	h,␣,S

- LINGUAGGIO DI PALINDROME: tutte le stringhe x tali che x è uguale a x letta in senso contrario;
  - Esempio: amanama,010010 ;
- Macchina di Turing per l'accettazione del linguaggio di palindrome: Muovi la testina avanti e indietro in modo che:
  - verifica il primo simbolo con l'ultimo simbolo e marcali, ad esempio sostituendoli il primo con ▷ e l'ultimo con ␣ ;
  - verifica il secondo simbolo con il penultimo, e cosi via…

	▷	0	1	␣
s	s,▷,R	q0,▷,R	q1,▷,R	yes,␣,S
q0	q0,▷,R	q0,0,R	q0,1,R	p0,␣,L
q1	q1,▷,R	q1,0,R	q1,1,R	p1,␣,L
p0	yes,▷,S	r,␣,L	no,1,S	r,␣,S
p1	yes,▷,S	no,0,S	r,␣,L	r,␣,S
r	s,▷,R	r,0,L	r,1,L	r,␣,S

Perché "recursively enumerable"

- Come abbiamo detto in precedenza, la classe dei linguaggi Turing-riconoscibile (Turing-recognizable) è anche chiamata *ricorsiva enumerabile* (*recursively enumerable).*
- Questo termine ha origine da una variante delle macchine di Turing: l'***enumeratore.***
- Un enumeratore è una macchina di Turing con stampante usata per stampare stringhe.
- Ogni qualvolta la macchina di Turing vuole aggiungere una stringa alla lista, semplicemente manda la stringa alla stampante.
- Nella figura è mostrato lo schema dell'enumeratore.
- Un enumeratore E inizia a lavorare con un nastro vuoto.
- Se l'enumeratore lavora per sempre, può produrre una lista infinita di stringhe
- Il linguaggio enumerato da E è l'insieme di tutte le possibili stringhe che E riesce a stampare.
- In particolare, è utile notare che E può generare le stringhe in qualsiasi ordine, anche con ripetizione.
- **Teorema:** Un linguaggio è Turing-riconoscibile (Turing-recognizable) se e solo se esiste un enumeratore che lo enumera.



- Un algoritmo si può definire come una collezione di istruzioni semplici per la realizzazione di un determinato compito.
- Gli algoritmi sono anche conosciuti in ambiti non informatici come procedure o ricette.
- Gli algoritmi giocano un ruolo importante in varie discipline (ingegneria, matematica, medicina, ecc.)
- Anche la matematica classica mostra una varietà di algoritmi per la soluzione dei problemi più disparati, come per esempio trovare numeri primi, il massimo comune divisore ecc.)
- Anche se gli algoritmi hanno una lunga storia nella letteratura, la loro effettiva formalizzazione risale al ventesimo secolo.
- La storia delle prossime slide mostra come la precisa definizione di algoritmo è stata fondamentale per la soluzione di un importante problema matematico.

I problemi di Hilbert

- Nel 1900, il matematico David Hilbert, in un importante congresso matematico tenutosi a Parigi, identificò 23 problemi matematici difficili e li consegnò alla comunità scientifica come "problemi competitivi" per il secolo che stava per iniziare.
- Alcuni problemi sono stati risolti in pochi anni, altri hanno visto la soluzione solo pochi anni fa, ed altri ancora non hanno avuto tuttora risposta .
- Il decimo problema della lista riguardava gli algoritmi ed in particolare chiedeva di definire un algoritmo che "dato in input un polinomio, fosse in grado di testare se quel polinomio ammettesse o meno radice intera".
- Una radice di un polinomio è un assegnamento di valori alle sua variabili in modo tale che il valore del polinomio diventi 0.
- Un polinomio è la somma di termini, dove ciascun termine è il prodotto di una costante (chiamato coefficiente) e di alcune variabili.
- Per esempio, $6x^3\,yz^2 + 3xy^2 - x^3 - 10$ è un polinomio di 4 termini con variabili x, y e z. La usa radice che $x = 5$, y = 3, and z = 0. Inoltre, questa radice è intera perché tutte gli assegnamenti alle variabili sono numeri interi. Si osservi che esistono polinomi con radici non intere.

L'indecidibilità del decimo problema di Hilbert

- Una osservazione sul decimo problema di Hilbert:
- Hilbert non usò il termine algoritmo per questo problema, ma quello di "processo" che, per definizione, è determinato da un numero finito di operazioni.
- Dunque, Hilbert non chiedeva di controllare l'esistenza di un algoritmo ma solo di implementarlo. Questo perché era convinzione comune che ogni problema dovesse ammettere una procedura risolutiva.
- Oggi sappiamo che il decimo problema di Hilbert è indecidibile! (questo risultato è stato formalmente provato nel 1970 da Yuri Matijasevic che estese una idea di Martin Davis, Hilary Putnam, and Julia Robinson)
- Ai tempi di Hilbert, gli strumenti a disposizione dei matematici erano certamente adeguati per mostrare procedure per alcuni problemi, ma erano completamente inadeguati per provare l'inesistenza di una procedura per quelli che oggi sappiamo essere indecidibili.
- I primi risultati sul decimo problemi di Hilbert arrivarono negli anni '30 e passarono per una definizione rigorosa del concetto di algoritmo.
- Tale definizione arrivò ad opera di Alonzo Church and Alan Turing:
  - Church usò un sistema notazionale chiamato λ-calculus.
  - Turing usò un modello di macchina computazionale (la macchina di Turing)
- Queste due definizioni furono poi mostrate essere equivalenti. Questa equivalenza diede origine alla seguente tesi chiamata tesi di ***Church-Turing.***

Tesi di Church-Turing

- Un problema (o una funzione) è **calcolabile (o decidibile)** in modo algoritmico (o calcolabile in modo effettivo, o effettivamente calcolabile) se esiste un algoritmo che consente di calcolarne i valori per tutti gli argomenti.
- Nel 1936 il logico americano Alonzo Church, in seguito alle sue ricerche sulla computabilità effettiva, propose di identificare la classe delle funzioni calcolabili mediante un algoritmo (o funzioni effettivamente calcolabili) con una particolare classe di funzioni aritmetiche (funzioni ricorsive). Tale identificazione è oggi nota col nome di **Tesi di Church**.
- È possibile dimostrare l'equivalenza tra la classe delle funzioni ricorsive e la classe delle funzioni **Turing-computabili** (decidibili), in quanto ogni funzione Turing-computabile è ricorsiva, e viceversa (Turing 1937).
- La Tesi di Church può quindi essere formulata come segue:

"una funzione è effettivamente calcolabile
se solo se
è Turing-computabile"

Il decimo problema di Hilbert è Turing-recognizable

- Mostriamo che il decimo problema di Hilbert è Turing-riconoscibile (Turing-recognizable).
- Dapprima consideriamo il problema per il caso semplificato di polinomi ad una sola varabile, tali come $4x^3 - 2x^2 + x - 7$.
- In questo caso, valutiamo se l'insieme D = {p | p è un polinomio su x con radice intera su x} è Turing-riconoscibile (Turing-recognizable) o meno.
- Una macchina di Turing M in grado di accettare D è la seguente:
  1. M prende input un polinomio p sulla variabile x.
  2. Poi M valuta p assegnando a x successivamente i valori 0, 1, -1, 2, -2, 3, 3....
  3. Se ad un certo punto il polinomio si valuta 0' allora M va in accettazione.
- Se p ha una radice intera, allora M prima o poi la troverà e dunque accetterà p. Se invece p non ha una radice intera allora M continuerà a computare all'infinito.
- Per polinomi a n variabili, si può utilizzare una macchina di Turing M' equivalente ad M, dove il test è fatto su insiemi ordinati di valori interi differenti.
- Sia M che M' così come sono state definite sono riconoscitori ma non decisori di linguaggi.
- Tuttavia, è possibile convertire M in un decisore utilizzando il seguente risultato:

  "La radice intera di un polinomio a singola varabile se esiste è compresa tra $\pm k(c_{max}/c_1)$ dove k è il numero dei termini, c_{max} è il coefficiente con valore assoluto più grade e c_1 è il coefficiente del termine di grado massimo.
- Matijasevic ha mostrato invece che è impossibile calcolare questo limite per polinomi con più variabili.

Proprietà delle Turing Machine

- **Teorema:** Le MdT sono chiuse rispetto a unione e intersezione.

- **Teorema:** La classe dei linguaggi decidibili è chiusa rispetto al complemento.

- La dimostrazione dei precedenti teoremi è lasciata per esercizio agli studenti. Sarà ancora più semplice provare tali teoremi dopo aver trattato le macchine di Turing multinastro

Aniello Murano

Macchine di Turing multinastro

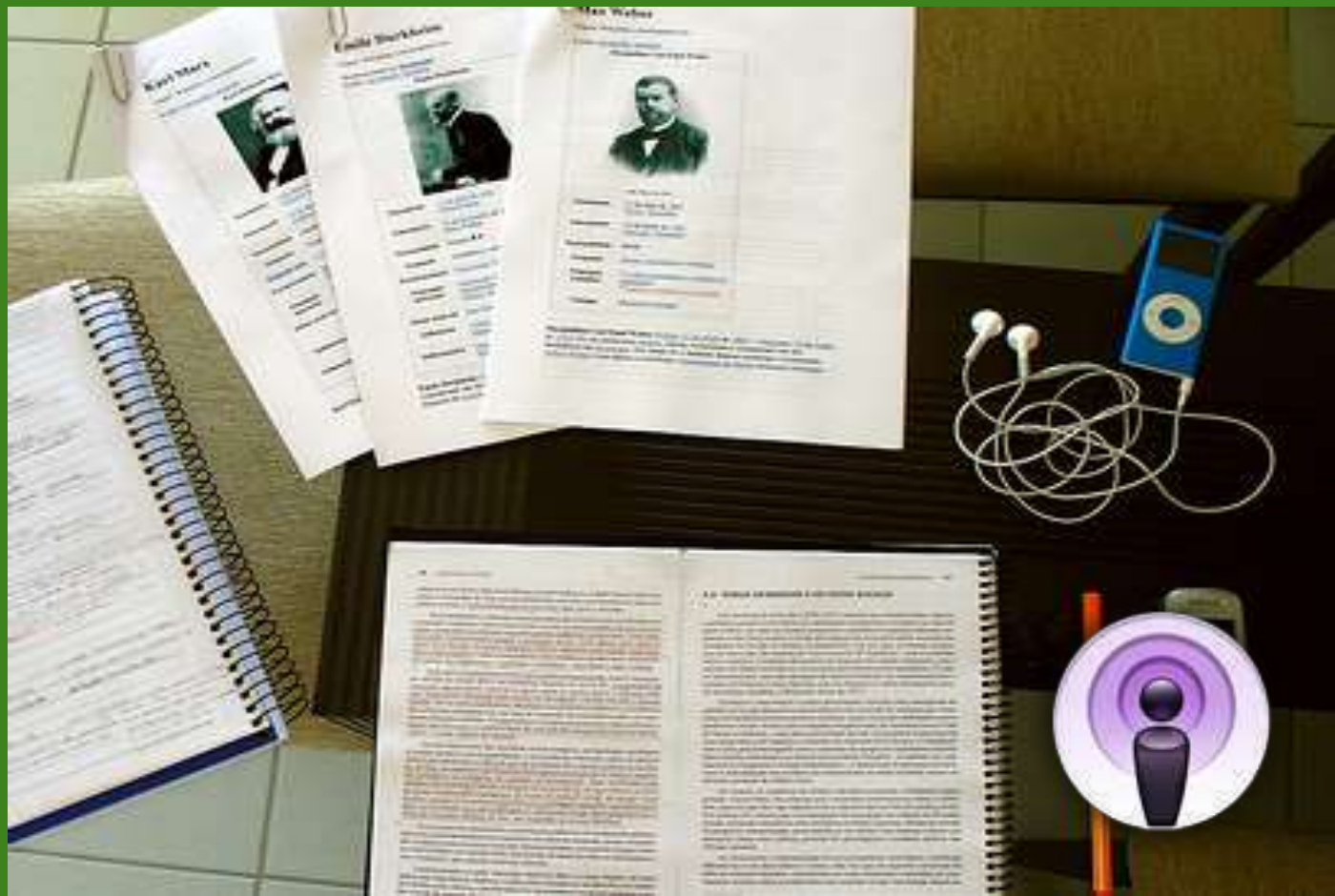
Lezione n.4
Parole chiave:
Macchina di Turing
Multinastro

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Definizione delle TM multinastro

- Una macchina di Turing multinastro è paragonabile ad un ordinaria Macchina di Turing a singolo nastro ma con più nastri.
- Ogni nastro ha la sua testina per leggere e scrivere.
- Inizialmente l'input risiede sul I° nastro e gli altri sono inizializzati con caratteri di blank.
- La funzione di transizione cambia per permettere di leggere,scrivere e muovere le testine su alcuni o tutti i nastri simultaneamente. Formalmente è descritta in questo modo:

$$\boldsymbol{\delta} : Q\backslash\{accept,reject\} \times \Gamma^k \rightarrow Q \times \Sigma^k \times \{L,R,S\}^k \text{ dove k è il numero di nastri;}$$

- L'espressione $\delta(q_i,a_1,\ldots, a_k)=(q_j,b_1,\ldots, b_k,L,R,\ldots,L)$ sta a significare che,

  Se la macchina è nello stato q_i e le testine da 1 a k stanno leggendo i simboli $a_1 \ldots a_k$, la macchina va nello stato q_j, scrive i simboli $b_1 \ldots b_k$, e direziona ogni testina per muoversi a sinistra o destra, o rimanere ferma, in base a quanto specificato dalla relazione di transizione.

- Sebbene le Macchine di Turing multinastro sembrano essere più potenti delle classiche TM, possiamo facilmente dimostrare che sono equivalenti.
- Ricordiamo che due Macchine di Turing sono equivalenti se riconoscono lo stesso linguaggio.

Equivalenza tra TM multinastro e TM a singolo nastro

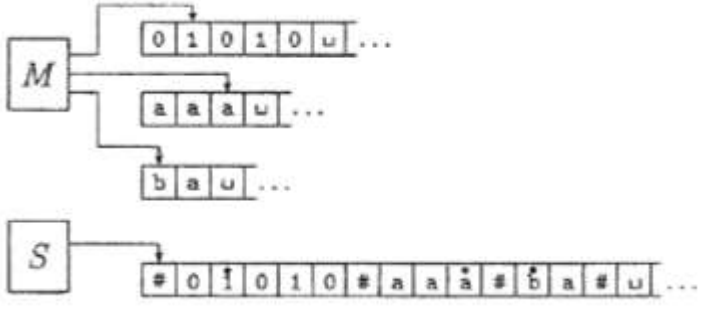
- TEOREMA

  Per ogni Macchina di Turing multinastro esiste una macchina di Turing a singolo nastro equivalente.

- DIMOSTRAZIONE

  Mostriamo come convertire una TM multinastro M in una TM S a singolo nastro. L'idea chiave è quella di mostrare come simulare M con S.

- Assumiamo che M abbia k nastri. Allora S simula l'effetto dei k nastri di M memorizzando il loro contenuto sul suo unico nastro. S utilizza il simbolo # come delimitatore per separare i contenuti dei diversi nastri. Inoltre, S deve tener traccia delle posizioni delle varie testine. Per fare questo, per ogni simbolo nella posizione di una testina dei k nastri, S scrive lo stesso simbolo con l'aggiunta di un punto sopra. Questo nuovo simbolo sarà poi aggiunto all'alfabeto del nastro di S. Per maggiore chiarezza di seguito è rappresentata una TM a 3 nastri e la sua corrispondente riduzione in una una TM a singolo nastro:



- Di seguito è descritto il funzionamento della TM S vista nella precedente diapositiva:

S=" Su input $w=w_1 \ldots w_n$: $\#\dot{w}_1 w_2 \cdots w_n \#\dot{\sqcup}\#\dot{\sqcup}\# \cdots \#$

1. Di seguito è descritto il funzionamento della TM S vista nella precedente diapositiva:

1. S in prima istanza copia il contenuto dei k nastri della TM M sul suo nastro:

2. Per simulare un singolo movimento, S scandisce il nastro dal primo simbolo # il quale indica il limite sinistro, fino al (k+1)-esimo #, il quale indica il limite destro, per individuare il simbolo che rappresenta la testina virtuale. Come secondo passo S aggiorna i nastri, in base a come è definita la funzione di transizione di M.

3. Se in qualche punto S muove una delle testine virtuali a destra di un simbolo #, questa azione indica che M ha mosso la testina corrispondente in una porzione del nastro dov'è presente un simbolo di blank. Cosi S scrive un simbolo blank su questa cella del nastro e trasla di un unità i contenuti del nastro, da questa cella fino al simbolo # più a destra. La simulazione continua poi ciclicamente.

- COROLLARIO:

  Un linguaggio è Turing-Riconoscibile se e solo se esiste almeno una macchina di Turing Multinastro che lo riconosce.

- DIMOSTRAZIONE:

  ->:

  Un linguaggio Turing riconoscibile è riconosciuto da una normale Macchina di Turing (a singolo nastro), la quale è un caso particolare di una Macchina di Turing multinastro.

  <-:

  Questo si dimostra attraverso il teorema visto nella precedente diapositiva.

- Si ricordi che una MdT che calcola una funzione (anche parziale) può essere vista come un **Trasduttore**
- Questo è ancora più evidente nel caso di una macchine di Turing a più nastri, dove un nastro può essere usato per l'input, uno per l'output ed eventualmente altri nastri possono essere usati per l'esecuzione della macchina
- Esercizio: Definire una macchina trasduttrice a due nastri che calcoli la funzione prodotto di due interi positivi in notazione unaria, seconde le seguenti convenzioni:
  - Sul primo nastro (nastro di input) è memorizzata la stringa $1^n\#1^m$, dove le due sequenze non vuote di 1 rappresentano i numeri da moltiplicare in notazione unaria.
  - Quando la macchina si ferma, sul secondo nastro (nastro di output) verrà memorizzata la stringa 1^{nm}.

Aniello Murano

Macchina di Turing universale e problema della fermata

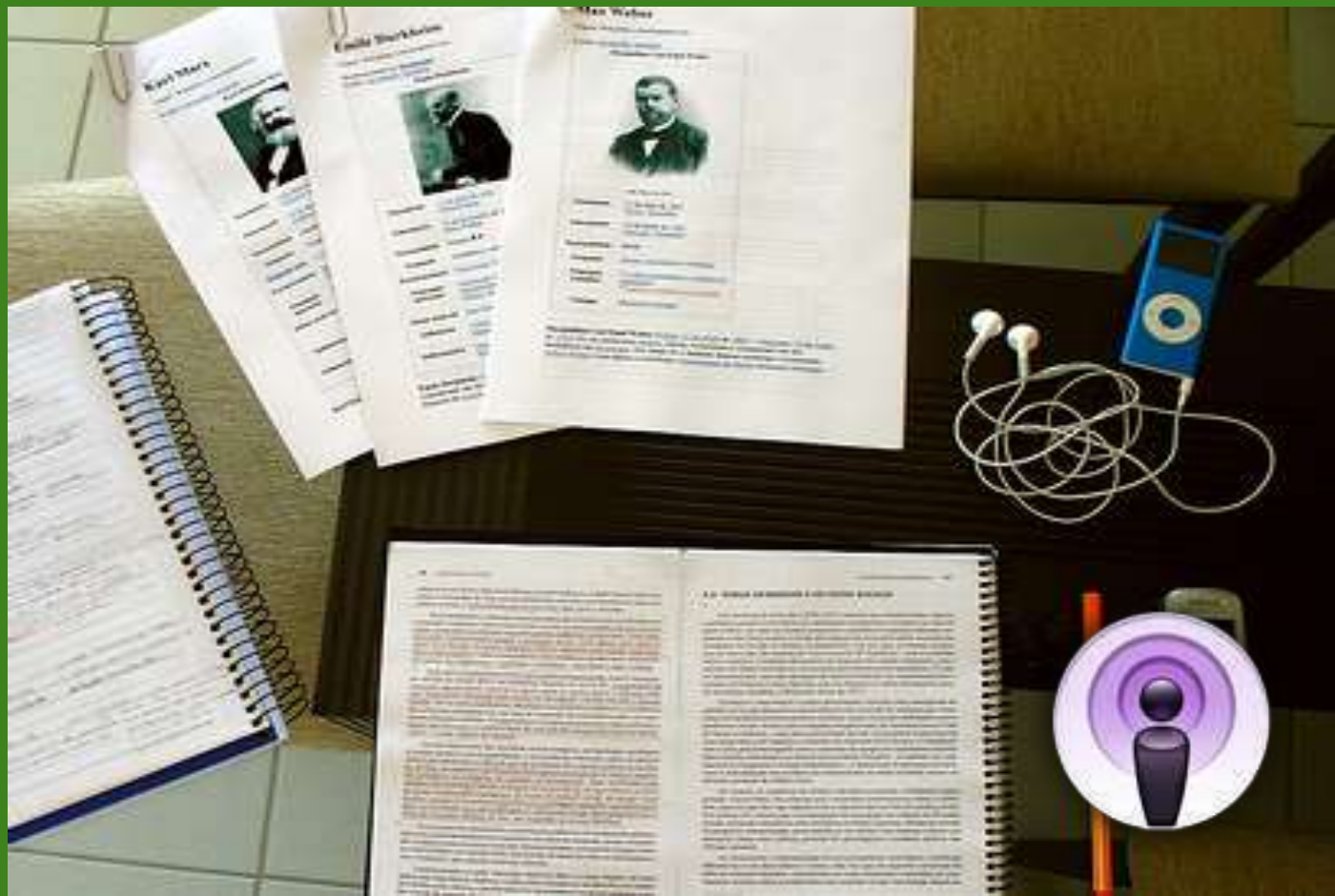
Lezione n.6
Parole chiave:
Universal Turing machine

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Riassunto delle lezioni precedenti

- Nelle lezioni precedenti abbiamo introdotto il concetto di macchina di Turing deterministica e non deterministica e abbiamo mostrato alcuni semplici linguaggi accettati.
- Abbiamo visto che le due versioni sono equivalenti ma passare da una macchina non deterministica a una deterministica può richiedere un salto esponenziale
- Con le macchine di Turing deterministiche è stato possibile caratterizzare la classe di complessità TIME(f(n)) di cui fa parte la classe P, e la classe SPACE(f(n)) di cui va parte la classe L (logarithmic)
- Con le macchine di Turing nondeterministiche è stato possibile caratterizzare la classe di complessità NTIME(f(n)) di cui fa parte la classe NP, e la classe NSPACE(f(n)) di cui va parte la classe NL.

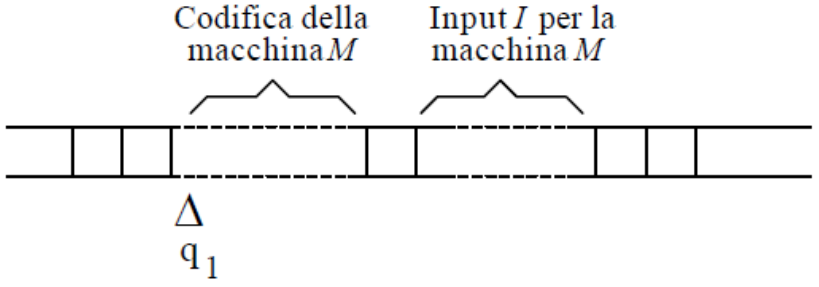
L’interesse nella macchina di Turing

- L'interesse nella macchina di Turing (TM) per gli informatici risiede soprattutto nel fatto che essa rappresenta un **modello di calcolo algoritmico**, di un tipo di calcolo cioè che è automatizzabile in quanto eseguibile da un dispositivo meccanico.
- Ogni TM è il modello astratto di un calcolatore - astratto in quanto prescinde da alcuni vincoli di limitatezza cui i calcolatori reali devono sottostare;
- Per esempio, la memoria di una TM (vale a dire il suo nastro) è potenzialmente estendibile all'infinito (anche se, in ogni fase del calcolo, una TM può sempre utilizzarne solo una porzione finita), mentre un calcolatore reale ha sempre limiti ben definiti di memoria.
- Dunque, una TM M che accetta un linguaggio è analoga a un programma che implementa un algoritmo.
- In questa lezione, mostreremo come una TM si può in realtà vedere come equivalente a un moderno calcolatore con un linguaggio di programmazione e una riserva illimitata di memoria.

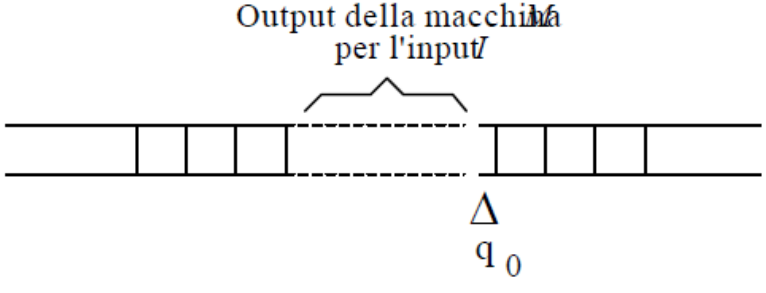
- Sino ad ora abbiamo considerato TM in grado di effettuare un solo tipo di calcolo, sono cioè dotate di un insieme di oggetti che consente loro di calcolare una singola funzione (ad esempio la somma, il prodotto, ecc.).
- Tuttavia, è possibile definire una TM, detta **Macchina di Turing Universale** (MTU), in grado di simulare il comportamento di ogni altra TM.
- Questo è reso possibile dal fatto che gli oggetti che definiscono ogni TM possono essere rappresentati in modo tale da essere scritte sul nastro di una TM.
- In particolare, è possibile sviluppare un metodo per codificare mediante numeri naturali la tavola di transizione di una qualsiasi TM.
- In questo modo, il codice di una TM può essere scritto sul nastro e dato in input a un'altra TM.
- La codifica può essere definita in maniera tale che, dato un codice, si possa ottenere la tavola di transizione corrispondente e viceversa mediante un procedimento algoritmico (una codifica che goda di questa proprietà è detta una **codifica effettiva**).

L'interesse nella macchina di Turing

- Si può dimostrare che esiste un TM (la MTU appunto) che, preso in input un opportuno codice effettivo delle componenti di un'altra macchina, ne simula il comportamento.
- Più formalmente, la MTU U è una macchina il cui input è composto dalla concatenazione di due elementi (si veda la figura in alto a lato):
  1. la codifica della tavola di transizioni di una TM M;
  2. un input I per M.
- Per ogni M e per ogni I, la MTU "decodifica" le tuple che definiscono M, e le applica ad I, ottenendo lo stesso output che M avrebbe ottenuto a partire da I (come mostrato nella figura a lato in basso).
- Formalmente si dice che U(M;I)=M(I)



Inizio della simulazione di M



Fine della simulazione di M

Dettagli sulla simulazione (1)

- Siccome U deve poter simulare qualsiasi TM M, non può essere considerato un limite superiore "a priori" per il numero di stati e di simboli di M che U deve considerare.
- Per questo motivo si assume che stati e simboli di M sono numeri interi.
- In particolare, si assume che
  - l'alfabeto Σ di M sia $\{1, 2, \dots, |\Sigma|\}$,
  - l'insieme di stati K sia $\{|\Sigma|+1, |\Sigma|+2, \dots, |\Sigma|+|K|\}$,
  - lo stato iniziale $s=|\Sigma|+1$,
  - Gli stati "accept", "reject" sono codificati con $|\Sigma|+2$ e $|\Sigma|+3$
  - I numeri $|\Sigma|+|K|+1$ e $|\Sigma|+|K|+2$ codificano gli spostamenti "left" e "right"
  - La funzione di transizione è ottenuta in modo ovvio, rappresentano le regole come tuple di numeri.
  - Tutti i numeri saranno codificati come numeri binari di lunghezza $\lceil \log(|K| + |\Sigma|) \rceil$

Dettagli sulla simulazione (2)

- La codifica della TM M in input per U comincerà con il numero |K| e poi |Σ| entrambi in binario e separati da virgole.
- Segue poi una descrizione di δ in termini di quintuple ((q,a),(p,b,*d*)), con *d* in {left, right}.
- Poi segue un ";" che ha il compito di segnalare la fine della descrizione di M.
- Ancora, si inserisce la codifica in binario della parola input $x = x_1, \ldots, x_k$, con la virgola usata come separatore degli interi binari che codificano i singoli simboli.
- Gli oggetti aggiuntivi (parentesi, virgola, punto e virgola, ecc.) possono anche essere codificati con altri interi successivi a quelli utilizzati.
- **Nota:** Ogni codifica "algoritmica" effettiva va bene.
- **Nota:** La rappresentazione di M e la rappresentazione del suo input possono anche essere messi su due nastri differenti, vista l'equivalenza (polinomiale) tra una macchina di Turing a più nastri e una ad un solo nastro.

Dettagli sulla simulazione (3)

- La MTU sull'input $<M,x_1 \ldots x_k>$ ha due nastri:
- il primo contiene l'input $<M,x_1 \ldots x_k>$
- il secondo contiene la (codifica della) configurazione corrente di M, nella forma (q,u,v) dove uv è il contenuto del nastro di M ad un certo punto della sua computazione, q è lo stato in cui si trova M e il simbolo in lettura è il primo di v.
- I primi passi della MTU servono per scrivere sul secondo nastro la codifica della configurazione iniziale, $(s,x_1 \ldots x_k)$
- Per simulare un passo di M:
  - U esamina il secondo nastro fino a trovare la codifica binaria dello stato corrente q (ricordiamo che è un numero tra $\{|\Sigma|+1, |\Sigma|+2, \ldots, |\Sigma|+|K|\}$)
  - cerca sul primo nastro una regola per q
  - poi muove la testina del secondo nastro per individuare il simbolo in lettura per M e controlla se la regola in lettura sul primo nastro coinvolge lo stesso simbolo input; se sì la regola viene implementata (cambiando la configurazione sul secondo nastro in corrispondenza) altrimenti si controlla la regola successiva
- Chiaramente, quando M si ferma, anche U si ferma.

Macchine di Turing programmabili

- Poiché la MTU è in grado di simulare il comportamento di qualsiasi TM, allora essa, in virtù della Tesi di Church, è in grado di calcolare qualsiasi funzione che sia calcolabile mediante un algoritmo.
- Ciò che caratterizza la MTU rispetto alle TM usuali è costituito dal fatto di essere una macchina calcolatrice **programmabile**.
- Mentre infatti le normali macchine di Turing eseguono un solo programma, che è "incorporato" nella tavola di transizione, la MTU assume in input il programma che deve eseguire.
- In pratica la MTU riceve in input la codifica di una TM M che deve simulare, dove la codifica di M ha proprio la funzione di consentire alla MTU di interpretare e di eseguire il “programma” rappresentato dalla TM M.

Macchine di Turing e l'architettura di Von Neumann (1)

- Un'altra caratteristica fondamentale della MTU è data dal tipo di trattamento riservato ai programmi.

- La MTU tratta i programmi (cioè la codifica delle tuple della TM da simulare) e i dati (l'input della TM da simulare) in maniera sostanzialmente analoga: essi vengono memorizzati sullo stesso supporto (il nastro), rappresentati utilizzando lo stesso alfabeto di simboli ed elaborati in modo simile.

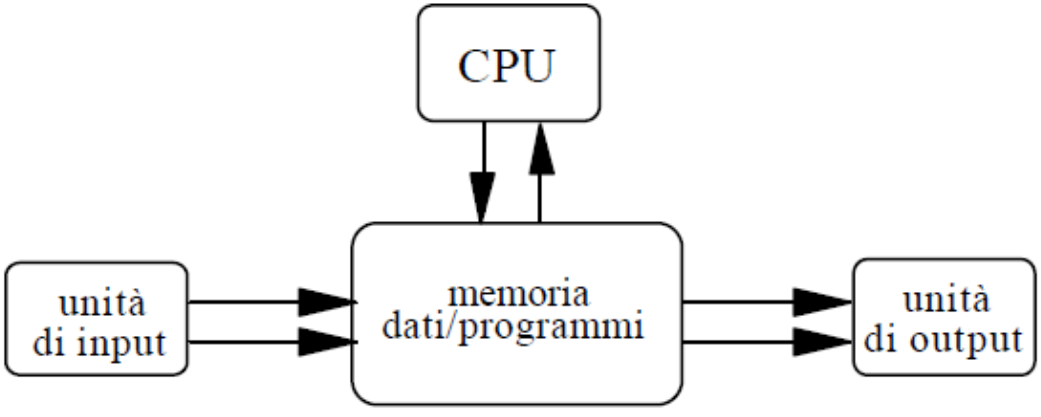
- Queste caratteristiche sono condivise dagli attuali calcolatori, che presentano la struttura nota come *architettura di von Neumann* (dal nome dello scienziato di origine ungherese John von Neumann che la ideò).

Macchine di Turing e l'architettura di Von Neumann (2)

- La struttura di un calcolatore di von Neumann è raffigurata, molto schematicamente, nella figura.

- Un dispositivo di input e un dispositivo di output permettono di accedere dall'esterno alla memoria del calcolatore, consentendo, rispettivamente, di inserirvi e di estrarne dei dati.

- Le informazioni contenute in memoria vengono elaborate da una singola unità di calcolo (detta CPU - *Central Processing Unit*), che opera sequenzialmente su di essi.



Macchine di Turing e l'architettura di Von Neumann (3)

- La caratteristica più importante della macchina di von Neumann è costituita dal fatto che sia dati che programmi vengono trattati in modo sostanzialmente omogeneo, ed immagazzinati nella stessa unità di memoria.

- Questo consente una grande flessibilità al sistema.

- Ad esempio, poiché dati e programmi sono oggetti di natura omogenea, è possibile costruire programmi che prendano in input altri programmi e li elaborino, e che producano programmi in output.

- Queste possibilità sono ampiamente sfruttate negli attuali calcolatori digitali, e da esse deriva gran parte della loro potenza e della loro facilità d'uso (ad esempio, un compilatore o un sistema operativo sono essenzialmente programmi che operano su altri programmi).

- In questo senso limitato, un calcolatore di von Neumann costituisce una realizzazione concreta della MTU (e la memoria dati/programmi può essere considerata l'equivalente del nastro della MTU).

Macchine di Turing e l'architettura di Von Neumann (4)

- Anche la potenza computazionale tra il calcolatore di von Neumann e la macchina di Turing è la stessa:
- se si suppone che il calcolatore di von Neumann è dotato di memoria e tempi di calcolo illimitati, esso è in grado di calcolare tutte le funzioni computabili secondo la Tesi di Church (per questo si dice che una macchina di von Neumann è un *calcolatore universale*).
- La MTU costituisce quindi un modello astratto degli attuali calcolatori digitali (elaborato prima della loro realizzazione fisica).
- Si noti che anche i vari linguaggi di programmazione sviluppati in informatica consentono di definire tutte e sole le funzioni ricorsive (purché, ovviamente, si supponga che tali linguaggi "girino" su calcolatori ideali con memoria e tempi di calcolo illimitati).
- Questo vale sia per i linguaggi di programmazione di alto livello (come PASCAL, FORTRAN, BASIC, VISUAL BASIC, C, C++, JAVA, LISP, PROLOG, eccetera), sia per i vari tipi di *codice assembler*.

Aniello Murano

Problemi decidibili e non decidibili

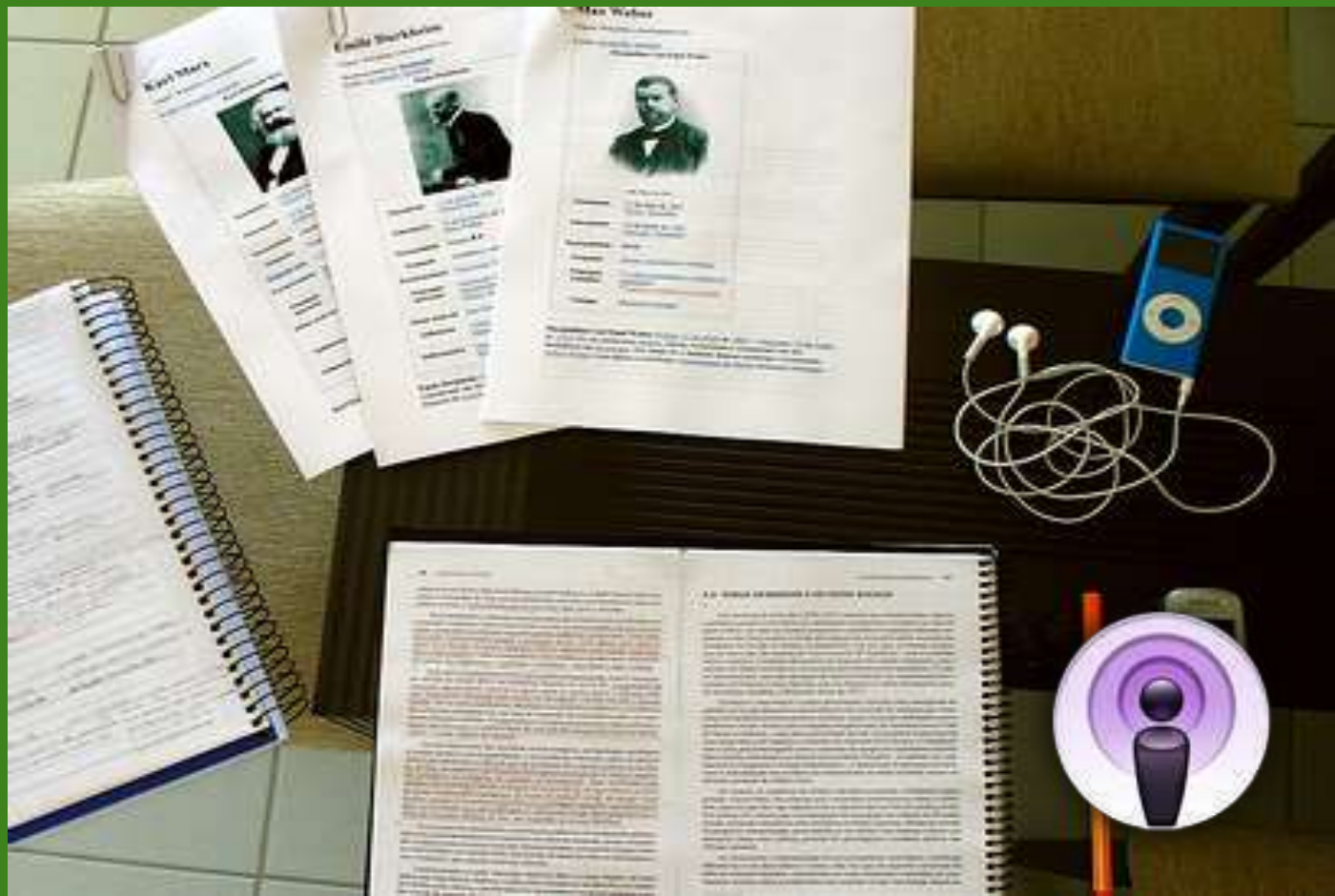
Lezione n.7
Parole chiave:
Decidibilità

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Overview

- In questa lezione mostreremo alcuni problemi decidibili per le macchine di Turing.
- In particolare, mostreremo su automi finiti che possono essere decisi da macchine di Turing (membership, vuoto, equivalenza, ecc)
- Mostreremo poi che già il semplice caso della membership (cioè stabilire se una data parola è accettata da una data macchina di Turing è non decidibile, sebbene Turing riconoscibile.
- Per mostrare l'indecidibilità del problema precedente utilizzeremo la tecnica della diagonalizzazione di Cantor.

- Abbiamo già detto che le macchine di Turing sono modelli computazionali più potenti degli NFA.
- In particolare è possibile definire degli algoritmi per testare se
  - Un DFA/NFA accetta una data stringa
  - Un linguaggio di un DFA/NFA è vuoto o meno
  - Due automi finiti sono equivalenti (cioè accettano lo stesso linguaggio).
- Tutti questi problemi possono essere rifrasati in termini di linguaggi. Per esempio, il problema di accettazione per un DFA può essere espresso tramite un linguaggio A_{DFA} contenente la codifica di tutti i possibili DFA e le stringhe che essi accettano nel modo seguente:

$$\mathbf{A_{DFA} = \{ (B, w) \mid B \text{ è un DFA che accetta la stringa in input } w\}}$$

- Il problema di verificare se UN DFA *B* accetta un input w equivale al problema di verificare se ***(B,*** *w) è un elemento di* A_{DFA}.
- Similmente, possiamo riformulare gli altri problemi in termini di verifica di appartenenza di un elemento in un linguaggio.
- Mostrare che il linguaggio A_{DFA} è decidibile equivale a mostrare che il relativo problema computazionale è decidibile.

A_{DFA} è decidibile

- Una macchina di Turing M in grado di decidere A_{DFA} è la seguente:
  - *M prende in input una codifica di B e una di w dove B* è un DFA e w è una stringa. Più precisamente, la codifica di B è la codifica dei suoi 5 componenti. Ogni codifica algoritmica (rigorosa) va bene!
  - M simula B su w, similmente a come visto per la macchina di Turing universale*. In pratica M legge B e w e in base alla transizione simulata aggiorna la configurazione corrente*
  - Se B finisce in uno stato di accettazione alla fine della lettura della stringa w (cioè la configurazione finale è di accettazione) allora M transisce nello stato di accettazione "yes"; altrimenti M rifiuta transendo nello stato "no".
- Un procedimento simile si può applicare nel caso di
  - NFA: l'input di M sarà la codifica del DFA corrispondente.
  - A_{REX} = { (R, w) | R è una espressione regolare che genera la stringa w}: l'input di M sarà la codifica del DFA corrispondente a R.

Il controllo del vuoto per i DFA è decidibile

- Sia E_{DFA} {A | A è un DFA e L(A) = Φ}
- **Teorema:** E_{DFA} è un linguaggio decidibile.
- **Prova:** Una macchina di Turing in grado di decidere questo linguaggio è quella mostrata nelle lezioni precedenti per il calcolo della raggiungibilità. In pratica, presa una codifica del DFA, la macchina di Turing verifica se seguendo le regole di transizione è possibile raggiungere uno stato finale da uno iniziale.

  L'algoritmo opera con un marcatore degli stati raggiungibili.

  Se ad un certo punto non è possibile marcare nuovi stati e non si è ancora raggiunto uno stato si accettazione per il DFA allora la macchina di Turing non accetta.
- La decidibilità del vuoto per un DFA permette di decidere anche l'equivalenza di due DFA. Infatti, ricordando che i DFA sono chiusi rispetto al complemento, all'unione e all'intersezione, l'equivalenza di due DFA A e B equivale a decidere il vuoto di

$$L(C) = \left(L(A) \cap \overline{L(B)}\right) \cup \left(\overline{L(A)} \cap L(B)\right)$$

Problemi decidibili per i PDA e i DPDA

- Alcuni dei problemi visti per gli NFA sono decidibili anche per i PDA, ma richiedono prove più complesse.
- Per esempio, per verificare il vuoto di un PDA non si può semplicemente simulare le sue configurazioni visto che sono infinite.
- Di contro però, le tavole di transizione dei PDA sono finite (tutti gli oggetti che li compongono sono finiti). Se si opera dunque sui componenti, allora si può verificare se una configurazione finale è raggiungibile o meno da una iniziale.
- Infatti, sono decidibili per i PDA (e dunque anche per i DPDA) il problema dell'appartenenza (di una stringa al linguaggio accettato da un NPDA o generato da una (D)CFL) e del vuoto.
- Per testare l'equivalenza di due PDA non possiamo semplicemente usare la stessa idea mostrata per i DFA visto che i PDA non sono chiusi rispetto al complemento. Per i PDA il problema dell'equivalenza è indecidibile.
- Per i DPDA il problema dell'equivalenza*,** è invece decidibile. Anche per i DPDA non è possibile usare lo stesso ragionamento usato per i DFA visto che i DPDA non sono chiusi rispetto a unione e intersezione.

Note bibliografiche:

*Geraud Senizergues. The equivalence problem for deterministic pushdown automata is decidable. In Automata, languages and programming (Bologna, 1997), volume 1256 of Lecture Notes in Comput. Sci., pp. 671- 681. Springer, Berlin, 1997.

**Geraud Senizergues. L(A) = L(B)? A simplied decidability proof. Theoret. Comput. Sci., 281(1-2):555-608, 2002. Selected papers in honour of Maurice Nivat.

Problemi indecidibili per le macchine di Turing

- **Teorema:** Il problema dell'appartenenza è indecidibile per la macchina di Turing.
- In seguito mostriamo la prova della sua indecidibilità così come di altri problemi. Questo al fine di introdurre tecniche di prova di indecidibilità.
- Il problema di determinare se una data macchina di Turing accetta una data stringa si può ridurre al problema di decisione del linguaggio seguente A_{TM}, in linea con quanto fatto per gli NFA:

$$A_{TM} = \{(M, w) \mid M \text{ è una macchina di Turing e } M \text{ accetta } w\}$$

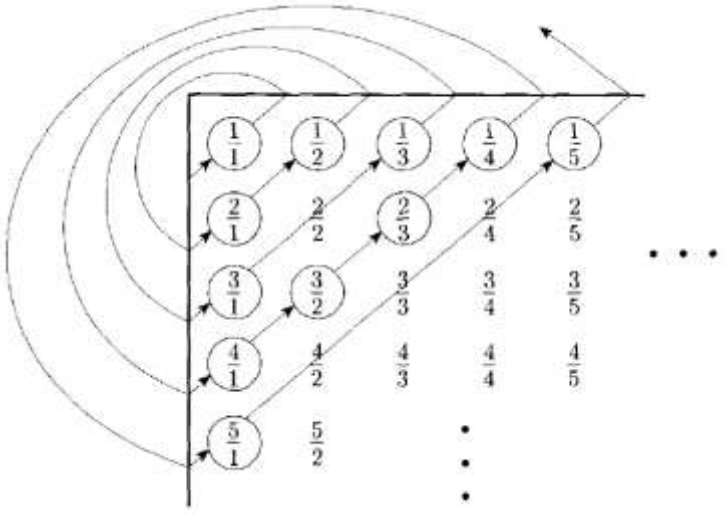
- Prima di mostrare che A_{TM} è indecidibile, mostriamo che è Turing-riconoscibile (Turing-recognizable).
- Questo mostra formalmente che i riconoscitori sono più potenti dei decisori.
- Una macchina di Turing U capace di accettare A_{TM} è la seguente:
- Su un input (M, w), dove M è la codifica di una TM e w è una stringa:
  1. U simula M su w.
  2. Se M non entra mai in uno stato di accettazione, allora U accetta; se M non entra mai in uno stato di rifiuto, allora U accetta.
- Si noti che U cicla su un input (M, w) se M cicla su w, il che dimostra perché U non decide A_{TM}.
- Se U avesse un modo per determinare che M non si fermerà mai su un certo input w, allora U potrebbe non accettare un tale input.
- Come vedremo in seguito, A_{TM} permette di dimostrare che il **problema della fermata** (**halting problem**) è indecidibile. Mostreremo cioè che nessuna macchina di Turing è in grado di determinare se un'altra macchina di Turing si fermerà prima o poi, o meno.

Il metodo di Cantor

- Per provare la non decidibilità di A_{TM}, si può ricorrere al metodo della diagonalizzazione introdotto da Georg Cantor nel 1873.
- A quell'epoca, Cantor si occupava della misurazione degli insiemi.
- Confrontare due insiemi finiti è semplice: basta contare gli elementi che essi contengono.
- Nel caso di insiemi infiniti, invece, non possiamo utilizzare questo metodo, perché chiaramente non finirebbe mai.
- Cantor propose un metodo alternativo basato sull'osservazione che due insiemi (finiti o infiniti) hanno la stessa taglia se gli elementi di un insieme possono essere accoppiati con gli elementi dell'altro.
- Prima di descrivere formalmente questo metodo, introduciamo alcune notazioni matematiche sulle funzioni:
- Si assuma di avere due insiemi A e B, e una funzione f da A a B. La funzione f è detta corrispondenza (biettiva) se
  - $f(a) \neq f(b)$ se e solo se $a \neq b$
  - per ogni $b \in B$ c'è un $a \in A$ tale che $f(a)=b$.
- Due insiemi A and B sono della stessa taglia se c'è una corrispondenza $f:A \rightarrow B$.
- In una corrispondenza, ogni elemento di A corrisponde ad un solo elemento di B e ogni elemento di B ha un unico elemento di A che corrisponde ad esso.

Insiemi contabili

- Con il metodo di Cantor, è possibile mostrare, ad esempio, che l'insieme dei numeri naturali e l'insieme dei numeri naturali pari hanno la stesa taglia.
- Infatti la corrispondenza f tra i due insiemi è semplicemente $f(n) = 2n$.
- **Definizione:** Un insieme è contabile se è finito oppure se ha la stessa taglia dell'insieme dei numeri naturali N.
- Si consideri l'insieme dei numeri razionali positivi $Q = \{m/n \mid m,n \in N\}$. Questo insieme è contabile?
- La risposta è **sì** perché esiste un modo per associare tutti gli elementi di Q agli elementi di N tramite una corrispondenza. Il metodo è il seguente:
- Si dispongono tutti i numeri di Q in una matrice infinita dove la i-esima riga contiene tutti i numeri con numeratore i e la j-esima colonna tutti i numeri con denominatore j.
- La matrice si può trasformare in una lista leggendo i suoi elementi in diagonale. Si noti che non leggiamo gli elementi per riga perché le righe sono infinite, altrimenti partendo dalla prima riga non visiteremmo mai la seconda.



La corrispondenza di N e Q

Insiemi non contabili

- Non tutti gli insiemi infiniti possono essere messi in corrispondenza con N. Questi insiemi sono detti non contabili.
- Per esempio, l'insieme dei numeri reali R non è contabile. La prova procede per assurdo, mostrando che esiste un termine che non ha corrispondenza in N.
  - Il numero si costruisce nel seguente modo: da tutti i numeri f(i), si costruisce il termine t= $0,c_1c_2\ldots$ dove ogni c_i è tale da essere differente dalla i-esima cifra decimale di f(i). Per esempio se f(1)= 4,14… e f(2)=6,45… allora t può essere 0,27… Questa costruzione assicura che t non corrisponde a nessun f(i).
- Trasportando questo risultato nella teoria della computazione, se uno mostra che **i linguaggi non sono contabili**, mentre **le macchini di Turing sono contabili**, segue che alcuni linguaggi non sono decidibili o addirittura Turing-riconoscibile.
- Per mostrare che l'insieme delle macchine di Turing è contabile si osservi che l'insieme di tutte le stringhe **Σ^* è contabile**. La funzione di corrispondenza è tra le stringhe di lunghezza i e l'insieme dei numeri necessari per listare tutte le parole di quella lunghezza.
- L'insieme delle macchine di Turing è contabile perchè ogni macchina è una codifica su Σ.

I linguaggi sono non contabili

- Per mostrare che l'insieme dei linguaggi è non contabile, prima si osservi che l'insieme B delle sequenze infinite binarie è non contabile (prova simile al caso dei reali).
- Sia C l'insieme di tutti i linguaggi sull'alfabeto Σ. Mostriamo adesso che anche C è non contabile, dando una corrispondenza con B.
- Sia $\Sigma^* = \{s_1, s_2, s_3, \ldots\}$. Adesso associamo i linguaggi di C alle stringhe di B in modo che ciascun linguaggio A in C abbia un'unica sequenza in B.
- La sequenza binaria è così costituita: l'i-esimo bit della sequenza è 1 se $s_i \in A$ e 0 altrimenti (formalmente questa sequenza è chiamata la X_A caratteristica di A). Per esempio, se A è il linguaggio delle stringhe che iniziano per 0 sull'alfabeto $\Sigma = \{0,1\}$, la sua caratteristica X_A è la seguente:

$$\begin{aligned}
\Sigma^* &= \{\ \varepsilon\ ,\ 0\ ,\ 1\ ,\ 00\ ,\ 01\ ,\ 10\ ,\ 11\ ,\ 000\ ,\ 001\ ,\ \cdots\ \}\ ; \\
A &= \{\ \quad 0\ ,\ \quad 00\ ,\ 01\ ,\ \quad\quad 000\ ,\ 001\ ,\ \cdots\ \}\ ; \\
\chi_A &= \quad 0 \quad 1 \quad 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \quad 1 \quad \cdots \quad .
\end{aligned}$$

- <u>Esercizio per gli studenti: Provare che X_A è una biezione</u>

Prova della indecidibilità di A_{TM}

- Supponiamo per assurdo che il problema dell'appartenenza è decidibile. Dimostreremo che tale ipotesi produce una contraddizione.
- Se A_{TM} è decidibile, per la universalità esiste una macchina di Turing H che decide M, cioè, data M (più precisamente, una sua codifica) e un qualsiasi input w di M, la macchina H si ferma e accetta se M accetta w, mentre H si ferma e non accetta se M non riesce ad accettare w. Schematicamente H(M,w) si comporta nel modo seguente:
  - accetta se la computazione di *M* con ingresso *w* termina con accettazione,
  - rifiuta se la computazione di *M* con ingresso *w* non accetta (cicla o rifiuta).
- Adesso, si consideri una nuova macchina di Turing D che utilizzi H come subroutine, tale che D chiama H per stabilire come si comporta M su input M e restituisce l'opposto del valore ottenuto. In pratica, D si comporta come segue:
  - rifiuta se la computazione di *M* con ingresso *M* termina con accettazione,
  - accetta se la computazione di *M* con ingresso *M* non accetta.
- **Nota:** Il fatto di eseguire una macchina sulla propria descrizione non deve preoccupare. Questo è simile al caso in cui un programma lavora con se stesso come input. Per esempio, un compilatore è un programma che trasforma altri programmi ed entrambi possono essere scritti con lo stesso linguaggio.

Prova del teorema della fermata

- Cosa succede se a D diamo in input la stessa macchina D (in pratica la sua codifica).
- Ci chiediamo cioè se D accetta o meno con input D?
- Abbiamo che *D(D) accetta se D* non accetta *D. Viceversa, rifiuta D quando D accetta D.*
- Questa è ovviamene una contraddizione. Dunque, H e D non possono esistere
- In pratica, abbiamo dimostrato, in base alla definizione di D, che D con input D da origine a un calcolo che termina se e soltanto se il calcolo di D per l'input D non termina, il che è palesemente assurdo.
- Ne consegue quindi che una macchina che si comporta come H non può esistere, e che quindi, se è vera la tesi di Church, non può esistere un algoritmo che permette di decidere il problema dell'appartenenza.
- Dunque il problema dell'appartenenza è indecidibile (o non ricorsivo).

L'halting problem e la tecnica della diagonalizzazione (1)

- Per meglio rendersi conto di questo risultato, consideriamo il metodo della diagonalizzazione di Cantor.
- Ancora una volta, si assuma che H sia in grado di decidere A_{TM}, che D, costruita a partire da H, su input M accetta esattamente quando M non accetta l'input M. Infine, si dia in input a D lo stesso D. Allora si ha che
  - H accetta *(M, w) esattamente quando M accetta w*
  - *D rifiuta M esattamente quando M accetta M (come input)*
  - *D rifiuta D quando D accetta D (contraddizione)*
- Vediamo adesso come la diagonalizzazione porta a mostrare un assurdo, sotto l'ipotedi che A_{TM} è decidibile
- Si consideri la tabella dei comportamenti di *H* e *D. In questa tabella riportiamo sulle righe le macchine di Turing M_1, M_2 ecc, e sulle colonne le loro descrizioni*
- Ogni combinazione riga i colonna j dice se una macchina M_i accetta o meno M_j. In particolare in (i,j) scriviamo *accept se* M_i accetta o meno M_j e lasciamo la cella vuota se M_i rifiuta o cicla su quell'input

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*		*accept*		
M_2	*accept*	*accept*	*accept*	*accept*	
M_3					$\cdots$
M_4	*accept*	*accept*			
$\vdots$			$\vdots$		

- ,

L'halting problem e la tecnica della diagonalizzazione (2)

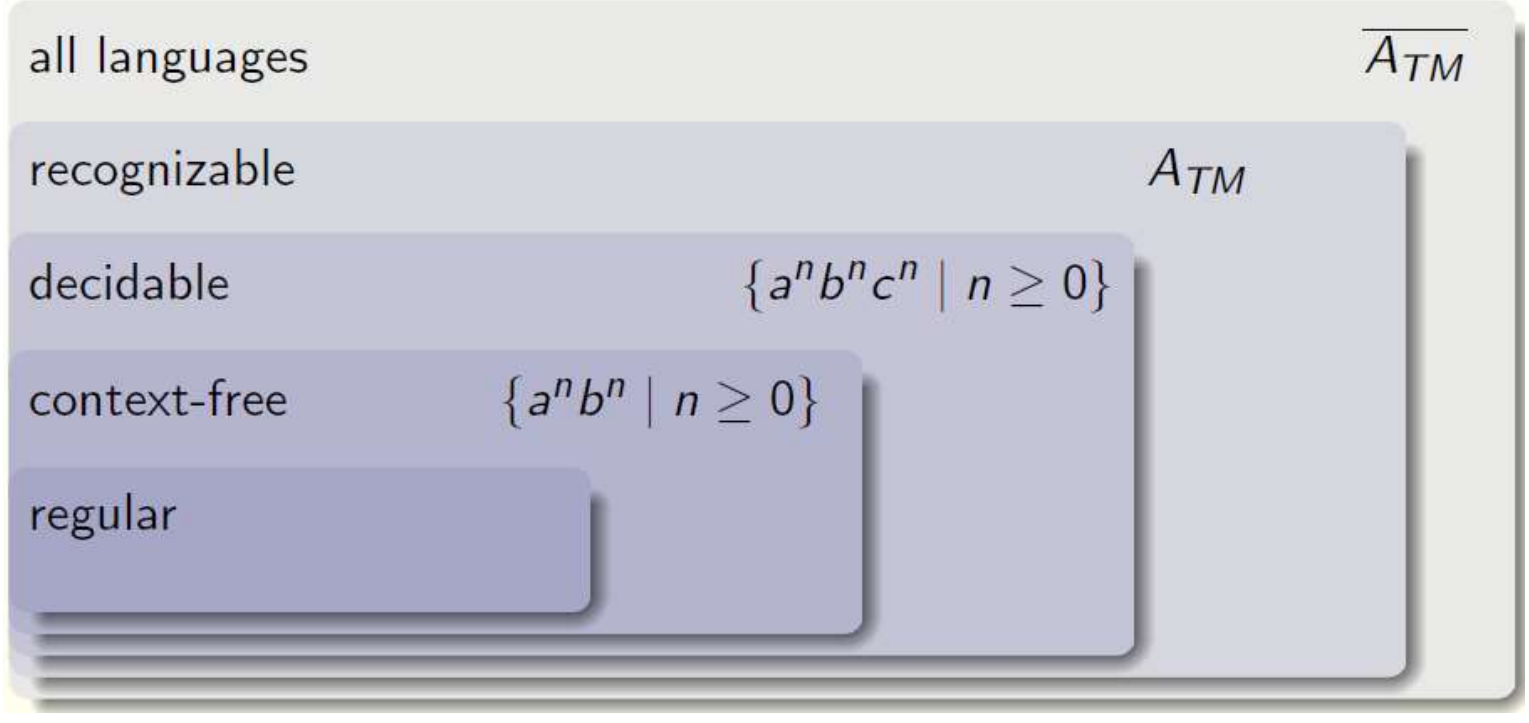
- Nella figura in alto al lato, ogni cella (i, j) rappresenta il risultato di H sugli input della figura mostrata nella slide precedente. Dunque, se M_3 no accetta M_2, nella cella (3,2) scriviamo reject, perché H rifiuta l'input (M_3, M_2).
- Nella figura in basso al lato, aggiungiamo la riga e la colonna D alla figura in alto.
- Si noti che D computa l'opposto dei valori presenti sulla diagonale. La contraddizione arriva in corrispondenza del punto interrogavo dove il valore deve essere l'opposto di se stesso.

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*	
M_4	*accept*	*accept*	*reject*	*reject*	
$\vdots$			$\vdots$		

	$\langle M_1 \rangle$	$\langle M_2 \rangle$	$\langle M_3 \rangle$	$\langle M_4 \rangle$	$\cdots$	$\langle D \rangle$	$\cdots$
M_1	*accept*	*reject*	*accept*	*reject*		*accept*	
M_2	*accept*	*accept*	*accept*	*accept*	$\cdots$	*accept*	$\cdots$
M_3	*reject*	*reject*	*reject*	*reject*		*reject*	
M_4	*accept*	*accept*	*reject*	*reject*		*accept*	
$\vdots$			$\vdots$		$\ddots$		
D	*reject*	*reject*	*accept*	*accept*		?	
$\vdots$			$\vdots$				$\ddots$

Un linguaggio non Turing-recognizable

- Completiamo questa lezione con due importanti risultati.
  1. Mostriamo una condizione necessaria e sufficiente per un linguaggio affinché sia ricorsivo(recursive)
  2. Mostriamo l'esistenza di linguaggi che non sono Turing-riconoscibile (Turing-recognizable).
- Definizione: Un linguaggio è co-Turing-riconoscibile (co-Turing-recognizable) se e solo se il suo complemento è Turing-riconoscibile (Turing-recognizable).
- **Teorema:** Un linguaggio è decidibile se e solo se è **Turing-riconoscibile** e **co-Turing-riconoscibile**.
- **Prova:** Ci sono due direzioni da provare. Dapprima assumiamo che A sia decidibile. Questa direzione segue dal fatto che il complemento di un linguaggio decidibile è decidibile e dal fatto che ogni linguaggio decidibile è anche Turing-riconoscibile.
- Per l'altra direzione, se A e il suo complemento comp(A) sono Turing-riconoscibile, possiamo usare una Macchina di Turing a 2 nastri dove sul primo decidiamo A e sul secondo decidiamo comp(A). Data una qualsiasi parola w essa appartiene ad A o a comp(A). Qualsiasi sia il caso, si ha che comunque la macchina su w si fermerà, per definizione di accettazione.
- **Corollario:** Comp(A_{TM}) non è Turing-riconoscibile.
- **Prova:** Noi sappiamo che A_{TM} è Turing-riconoscibile. Se anche comp(A_{TM}) fosse Turing-riconoscibile, allora per il teorema precedente anche A_{TM} sarebbe decidibile. Ma noi sappiamo che A_{TM} non è decidibile. Dunque, comp(A_{TM}) non può essere Turing-riconoscibile.


all languages
$\overline{A_{TM}}$
recognizable
A_{TM}
decidable
$\{a^n b^n c^n \mid n \geq 0\}$
context-free
$\{a^n b^n \mid n \geq 0\}$
regular

Aniello Murano

Problemi non decidibili e riducibilità

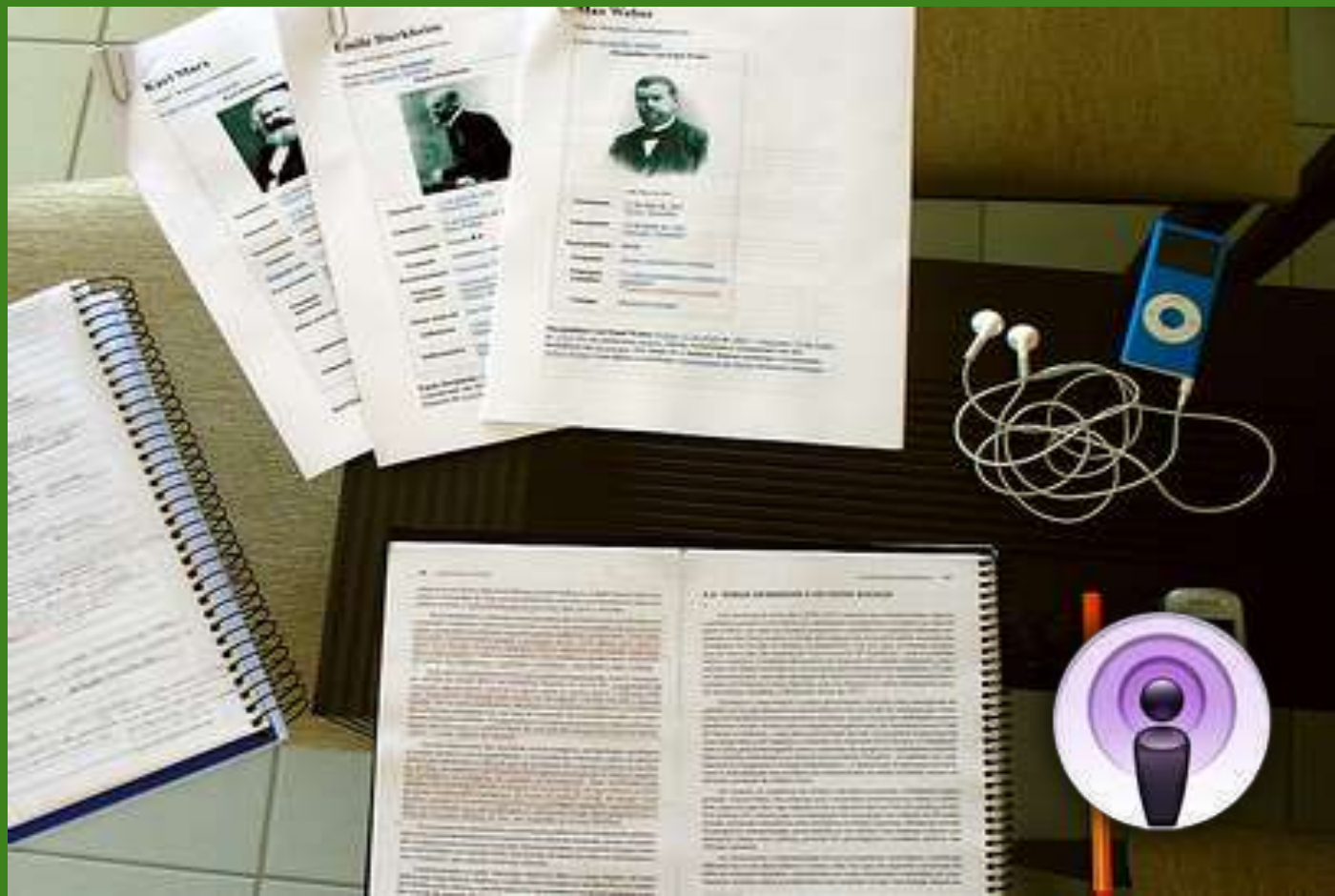
Lezione n.9
Parole chiave:
LBA e PCP

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



LBA – Linear bounded automaton

DEFINIZIONE:

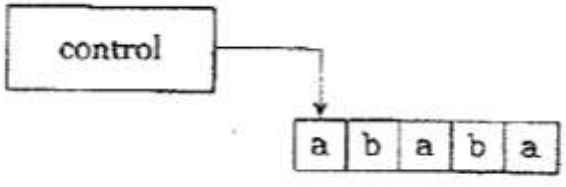
- Le **linear bounded automaton** (LBA) sono Macchine di Turing con una sola limitazione:

la testina sul nastro non può muoversi
oltre la porzione di nastro contenente l'input.

- Se un LBA prova a muovere la sua testina oltre l'input, la testina rimane ferma in quel punto come accade nella Macchina di Turing classica quando la testina tenta di andare più a sinistra del primo simbolo in input (o il simbolo di start se utilizzato).

- Un **LBA** è una Macchina di Turing con memoria limitata, com'è mostrato nella figura sottostante. Può risolvere solo problemi il cui input non superi la capienza del nastro.

- Usando un alfabeto del nastro più ampio dell'alfabeto di input si ha che la memoria disponibile viene incrementata di un fattore costante. Perciò per un input di lunghezza n, la quantità di memoria disponibile è lineare in n. Da questo deriva il nome di questo modello.



Potenza espressiva degli LBA

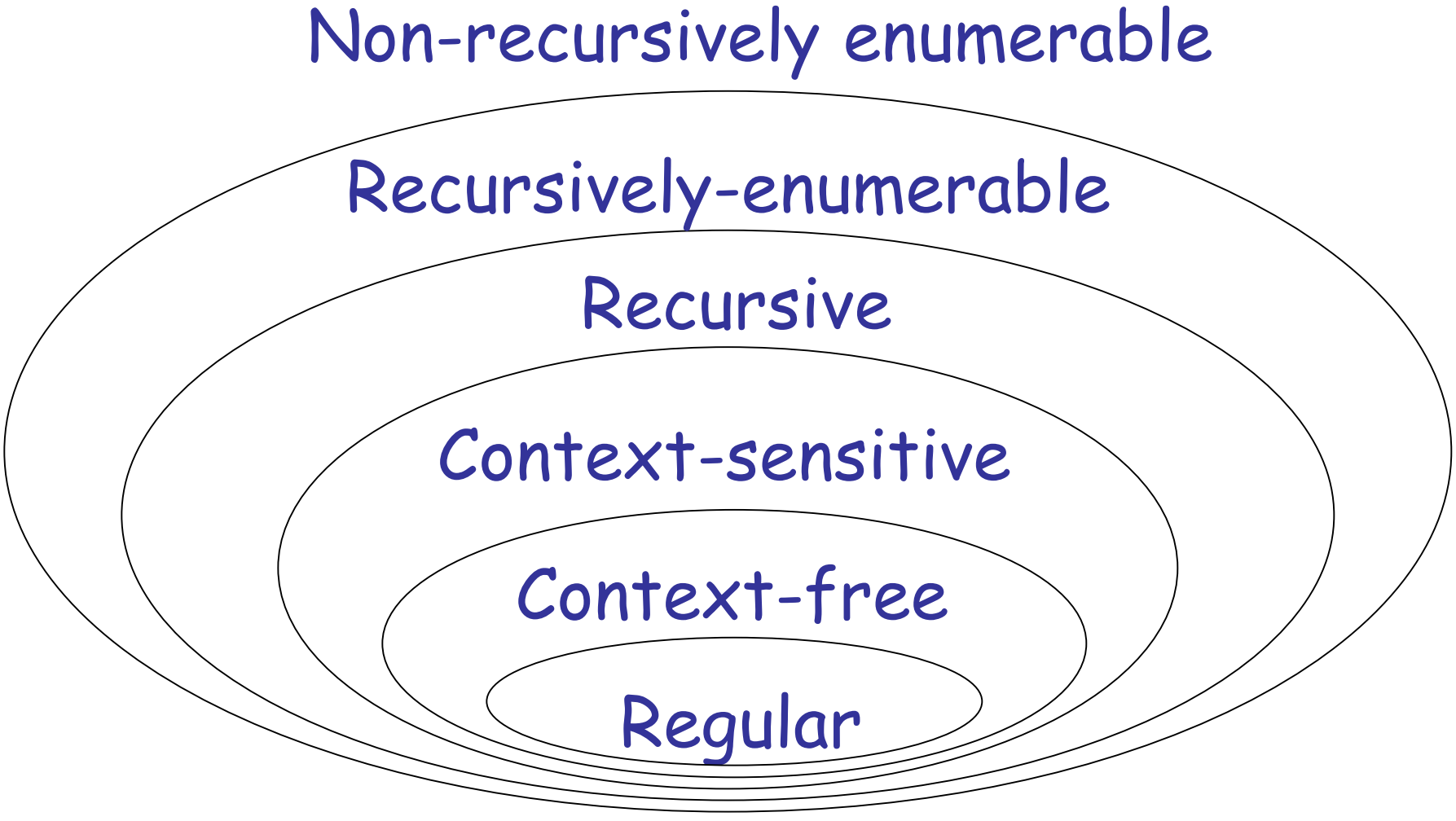
- Nonostante questi vincoli sulla memoria, gli LBA sono piuttosto potenti in termini di accettori di linguaggi. Ad esempio, i decisori per A_{DFA} e E_{DFA} sono tutti LBA.

- Gli LBA sono strettamente più potenti dei PDA. Infatti, si può provare che ogni CFL può essere deciso da un LBA. Inoltre esistono linguaggi accettati da LBA che non sono context-free come ad esempio $L=\{a^n b^n c^n\}$ e $L=\{a^{n!}\}$.

- Gli LBA sono anche meno potenti delle Macchine di Turing (si pensi a qualsiasi linguaggio che ha bisogno di memoria "non constante" in aggiunta all'input per poterlo processare)

- I linguaggi accettati dagli LBA sono generati da grammatiche denominate ***"context-sensitive".***

- ***Problema aperto:*** La versione deterministica degli LBA è meno potente?


Non-recursively enumerable
Recursively-enumerable
Recursive
Context-sensitive
Context-free
Regular

- Come accennato, gli LBA sono piuttosto potenti in termini di accettori di linguaggi. Vediamone un esempio:

- A_{LBA} è il problema di determinare se un LBA accetta il suo input.
- Formalmente,

$$A_{LBA} = \{<M,w> \mid M \text{ è un LBA che accetta una stringa } w\}$$

- Ricordiamo che il problema analogo per le MT, ovvero A_{TM}, è indecidibile

- A_{LBA} è decidibile.
- La dimostrazione della decidibilità di A_{LBA} è semplice qualora si possa asserire che il numero di configurazioni possibili è limitato, come mostrato dal Lemma che presentiamo nella prossima diapositiva.

PARTICOLARITA' DI UN LBA

LEMMA:

- Preso un LBA M con q stati e g simboli nell'alfabeto del nastro. Ci sono esattamente qng^n configurazioni distinte di M per un nastro di lunghezza n.

DIMOSTRAZIONE:

- Ricordando che una configurazione di M è come uno "snaphshot" nel mezzo di una computazione.
- Una configurazione consiste dello stato di controllo, posizione della testina e il contenuto del nastro.
- Per quanto riguarda le "grandezze" degli oggetti coinvolti nelle configurazioni, notiamo che:
  - M ha q stati.
  - la lunghezza del nastro è n, cosi la testina può essere al più in n posizioni differenti,
  - g^n sono le possibili stringhe di simboli del nastro che si possono trovare sul nastro.
- Il prodotto di queste tre quantità è il numero totale di configurazioni differenti di M con un nastro di lunghezza n, ovvero qng^n.

TEOREMA

A_{LBA} è decidibile

IDEA PER LA DIMOSTRAZIONE:

- Prima di tutto per decidere se un LBA M accetta l'input w, simuliamo M su w con una MdT M'
- Durante il corso della simulazione, se M si ferma e accetta o rigetta, allora la prova termina perché M' si ferma. La difficoltà nasce se M gira all'infinito su w.
- Questo si può ovviare individuando quando la macchina va in loop, cosi possiamo fermarci e rifiutare.
- L'idea per capire quando M va in loop è questa:
  - Quando M computa su w, M passa da una configurazione ad un'altra. Se M ripete più volte una stessa configurazione significa che è entrata in un loop.
  - Poiché M è un LBA, sappiamo che la capienza del nastro è limitata. Attraverso il lemma visto nella precedente slide, sappiamo che M può avere solo un numero limitato di configurazioni in base alla capienza del suo nastro.
  - Quindi M ha un limite di volte prima di entrare in qualche configurazione che ha già fatto precedentemente. Come conseguenza del lemma, possiamo dire che se M non si ferma dopo un certo numero di passi significa che è in loop. Questo M' è in grado di rilevarlo.

Dimostrazione della decidibilità di A_{LBA}

DIMOSTRAZIONE:

- L'Algoritmo che decide A_{LBA} è il seguente.

M':="Su input <M,w>, dove M è un LBA e w è una stringa:

1. Simula M su w finché non si ferma o, alternativamente, per qng^n passi.
2. Se M si ferma, abbiamo i seguenti casi:
   I. Se M accetta allora M' accetta
   II. Se M rifiuta allora M' rifiuta
3. Se M non si ferma, allora M' rifiuta."

- **Domanda:** Perché l'algoritmo può rifiutare se la macchina non si ferma in qng^n passi?

- **Risposta:** Per il lemma precedente, M deve aver incontrato almeno una volta una configurazione precedentemente incontrata, dunque si trova in un loop.

Il linguaggio E_{LBA}

- Abbiamo visto un problema decidibile per un LBA ma che non lo è per una Macchina di Turing classica.
- Esistono tuttavia problemi indecidibili per le TM che rimangono tali anche per le LBA. Un esempio è il seguente linguaggio:

$$E_{LBA} = \{ \langle M \rangle \mid M \text{ è un LBA dove } L(M) = \varnothing \}$$

Per la dimostrazione di indecidibilità, dobbiamo prima introdurre il concetto di "computation histories"

Computation Histories

- Una **Computation Historie** (CH) di una MdT M è la sequenza finita di configurazioni che M attraversa per decidere un input.
-
- Sia *M* una MdT e *w* un ingresso per *M*. Una **CH accettante** per *M* su *w* è una sequenza di configurazioni $C_1, C_2, \ldots, C_l$ dove:
  - C_1 è la configurazione di partenza di *M* su *w*
  - C_l è una configurazione accettante
  - ogni C_i segue C_{i-1} secondo le regole di *M*

- Una **CH non accettante** è simile ad una CH accettante tranne che C_l è una configurazione non accettante.

- Nota: Le MdT deterministiche hanno una sola CH per un dato input, mentre le MdT non deterministiche ne possono avere diverse.

Prova che E_{LBA} è indecidibile

- Riduciamo A_{TM} a E_{LBA}.
- Per ogni MdT M e ingresso w, definiamo un LBA *B* tale che

  L(B)={CH | CH è accettante per una data MdT M e un dato ingresso w}

  - Se *L(B)* = Ø vuol dire che *M* non accetta *w* e dunque (M,w) non è in A_{TM}.
  - Viceversa se L(B) non è vuoto allora *M* accetta *w* e dunque (M,w) è in A_{TM}.

- Per verificare che ogni sequenza in ingresso per *B* è una CH accettante devono essere verificate le condizioni descritte in precedenza.
- Costruiamo il decisore *S* per A_{TM}, supponendo che esiste un decisore *R* per L(B)
- *S* := su ogni ingresso <*M, w*>,
  - costruisce un LBA *B* da (*M*,*w*) come definito sopra.
  - esegue *R* su ingresso *(B)*
  - se *R* accetta → S rifiuta (*L(B)* è vuoto → *M* non accetta *w*)
  - se *R* non accetta → S accetta (*L(B)* è non vuoto → *M* ha una CH accettante per *w*)
- Quindi, l'esistenza di un decisore per E_{LBA} implica un decisore per A_{TM}. Sappiamo però che A_{TM} è indecidibile, quindi non esiste un decisore per E_{LBA}.

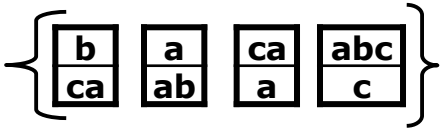
PCP – POST CORRESPONDENCE PROBLEM

- Il fenomeno dell'indecidibilità non è un problema che riguarda solo gli automi. Per mostrare un esempio, consideriamo un problema indecidibile riguardante una "semplice" manipolazione di stringhe: il **Post correspondence problem** (PCP).

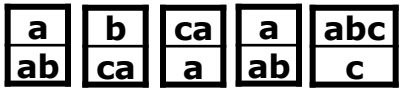
- Il PCP fu introdotto e provato indecidibile per la prima volta da E.L.Post in "A variant of Recursivley Unsolvable problem", Bull. Amer. Math. Soc.52 (1946)

- Possiamo descrivere questo problema paragonandolo ad un tipo di puzzle. Iniziamo con un insieme di domini, ognuno contenente due stringhe, ognuna su un lato.

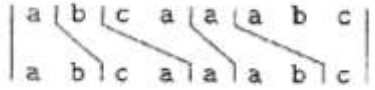
- Un pezzo singolo del domino è rappresentato in questo modo: 

- Mentre un insieme di domini sarà di questo tipo: 

- L'obbiettivo è quello di costruire una lista di questi domini (le ripetizioni sono ammesse) in modo che la stringa dei simboli scritti nella parte superiore dei domini è uguale a quella dei simboli inferiori. Questa lista è chiamata un **match**.

- Ad esempio la seguente lista è un match per questo puzzle: 

- Leggendo l'inizio della stringa otteniamo "abcaaabc", che è uguale alla stringa sotto. Possiamo anche rappresentare questo match deformando i domini in modo che creiamo la corrispondenza da sopra a sotto la linea.



- Per alcuni insiemi di domini non è sempre possibile trovare un match. Per esempio l'insieme:

$$\left\{\left[\frac{abc}{ab}\right], \left[\frac{ca}{a}\right], \left[\frac{acc}{ba}\right]\right\}$$

non può contenere un match perché ogni stringa che si trova sopra è più lunga rispetto alla stringa che si trova sotto.

- Il **Post correspondence problem** riguarda la possibilità di determinare se una collezione di domini ha un match. Questo problema non è risolvibile con algoritmi. Prima di dare la definizione formale di questo teorema con la sua definizione, formuliamo precisamente il problema e in seguito definiamo il suo linguaggio.

INDECIDIBILITA' DEL PCP

- Un instanza del PCP è un insieme P di domini: $\left[\begin{smallmatrix} t_1 \\ b_a \end{smallmatrix}\right]$, $\left[\begin{smallmatrix} t_2 \\ b_2 \end{smallmatrix}\right]$,…, $\left[\begin{smallmatrix} t_k \\ b_k \end{smallmatrix}\right]$
- Un match è una sequenza di interi $i_1, i_2,\ldots, i_l$, dove $t_{i1}, t_{i2},\ldots t_{in} = b_{i1}, b_{i2},\ldots, b_{in}$.
- Nota: la stessa coppia può apparire più volte, ovvero, può essere che per certi $m \neq j$, si ha che $i_j = i_m$.

- Il problema del PCP è quello di determinare se P ha un match.

- Il linguaggio corrispondente al problema è

  PCP={<P>| P è un istanza del Post Correspondence Problem con un match}

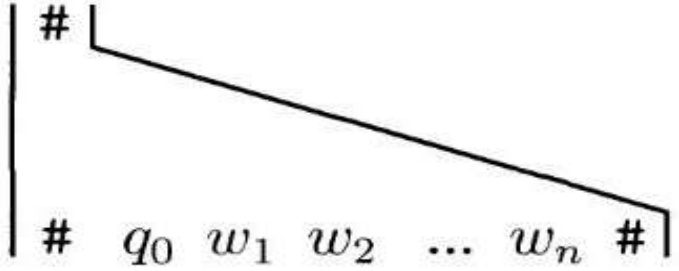
TEOREMA: **PCP è indecidibile**

PCP è indecidibile

- Per la prova riduciamo A_{TM} al PCP attraverso le Computation Histories.
- Dati $\langle M, w \rangle$, costruiamo un insieme P di domino nel modo seguente:
- Iniziamo con un domino che ha nella parte di sotto la configurazione iniziale di M e sopra una stringa vuota.
- Aggiungiamo a P dei domino la cui parte superiore corrisponde alla configurazione corrente e la parte inferiore è la configurazione successiva.
- Questi domino devono anche tenere conto di
  - Il movimento della testina.
  - Aumentare lo spazio del nastro con degli spazi vuoti (quando richiesto).
  - Forzare ad usare come primo domino della soluzione del match quello corrispondente alla configurazione iniziale di M.
- Per semplicità, assumiamo che ogni soluzione possibile deve sempre inizialre con il primo domino.
- Chiameremo il PCP con questa regola aggiuntiva "**modified PCP**" o semplicemente MPCP.
- Successivamente vedremo che questa non è una limitazione e che può essere rimossa.

Prova di ind. per PCP: Step 1. Inizializzazione

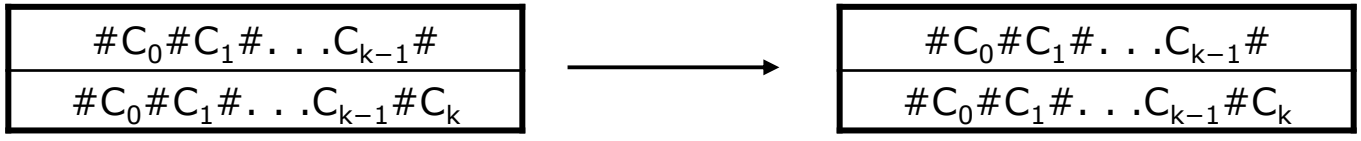
- Dati ⟨M, w⟩, costruiamo il primo domino dell'insieme P come corrispondente alla configurazione iniziale è q_0w nel modo seguente:



- Il simbolo # è un nuovo simbolo non appartenente all'alfabeto del nastro di M e serve come marcatore

Prova di ind. per PCP: Step2. transizione

- Ad ogni passo di computazione di M facciamo corrispondere dei domino in P in modo da poter copiare la configurazione corrente di M dalla parte di sotto dei domino (delimitata tra #) nella parte superiore e sostituiamo nella parte sottostante corrispondente la prossima configurazione



- Per fare questo, abbiamo bisogno di fare più passi elementari nel modo seguente:
- Una configurazione racchiusa tra # avrà sempre la forma del tipo α**bqc**β.
- Per calcolare la prossima configurazione, occorre copiare α sia nella parte superiore che inferiore
- Copiare bqc sulla parte superiore e la prossima configurazione sulla parte inferiore
- Copiare β sia nella parte superiore che inferiore.
- Per copiare α e β, aggiungiamo il sequente domino in P, per ogni a in Γ:

a
a

- Nelle slide successive indichiamo come trattare le transizioni.

Prova di ind. per PCP: Step 3. transizione R

- Per ogni transizione del tipo δ(q,c)=(p,d,R), noi aggiungiamo a P il seguente domino

qc
dp

- Questo domino ci permetterà di muovere da

#. . . # αb
#. . . #α**bqc**β # αb

#. . . # αbqc
#. . . #α**bqc**β # αbdp

- Per il caso speciale in cui c può essere il carattere blank "-", aggiungiamo in P il domino

q#
dp#

- L'ultimo domino sarà utile quando M leggerà spazi blank a destra della stringa di input

Prova di ind. per PCP: Step 4. transizione L

- Per ogni transizione del tipo δ(q,c)=(p,d,L), noi aggiungiamo a P il seguente domino

bqc
pbd

- Questo domino ci permetterà di muovere da

#. . . # α
#. . . #α**bqc**β # α

→

#. . . # αbqc
#. . . #α**bqc**β # αpbd

- Per il caso speciale in cui b può essere il carattere blank "-", aggiungiamo in P il domino

#qc
#pd

- Quest'ultimo domino sarà utile quando M è all'inizio dell'input e cerca di andare a sinistra

Prova di ind. per PCP: Step 5. matching con q_{accept}

- M accetta w se possiamo raggiungere nel PCP la seguente situazione:

$\#C_0\#...\#C_{n-1}\#$
$\#C_0\#...\#C_{n-1}\#\alpha bq_{accept}\beta\#$

- Prima di terminare con la riduzione, dobbiamo sistemare il fatto che la stringa nella parte inferiore dei domino è più lunga di una configurazione della parte superiore. Per questo motivo, aggiungiamo a P i domino del tipo:

cq_{accept}
q_{accept}

e

$q_{accept}c$
q_{accept}

- Gli ultimi domino permetteranno di scartare uno alla volta i simboli in eccesso, fino ad arrivare alla seguente situazione:

$\#C_0\#...\#q_{accept}c\#$
$\#C_0\#...\#q_{accept}c\#q_{accept}$

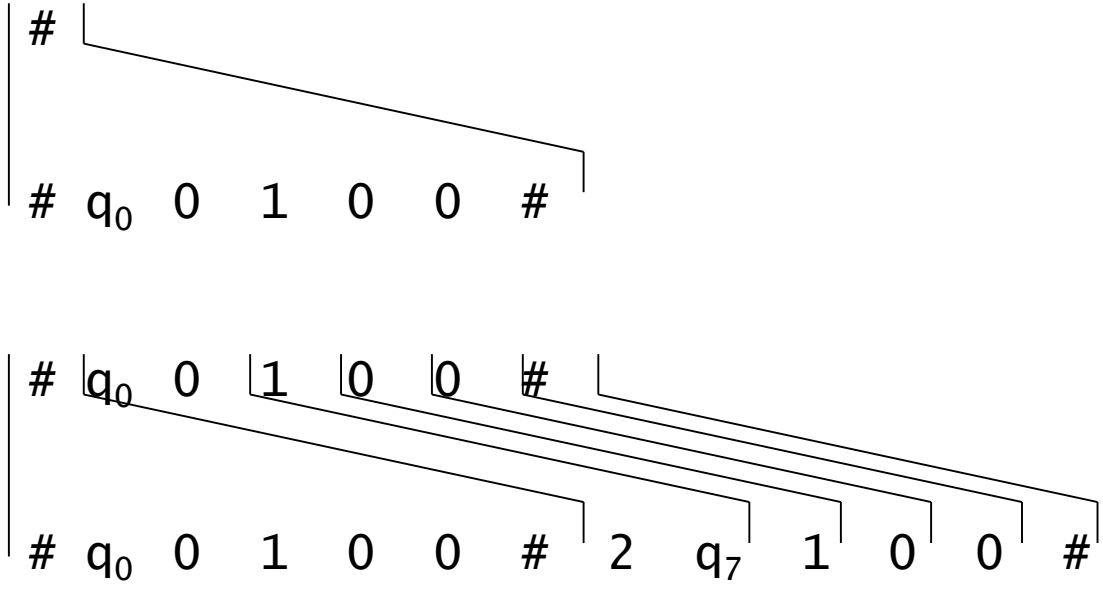
Prova di ind. per PCP: Step 6. Chiusura con q_{accept}

- Per finire dobbiamo aggiungere il seguente ulteriore domino a P:

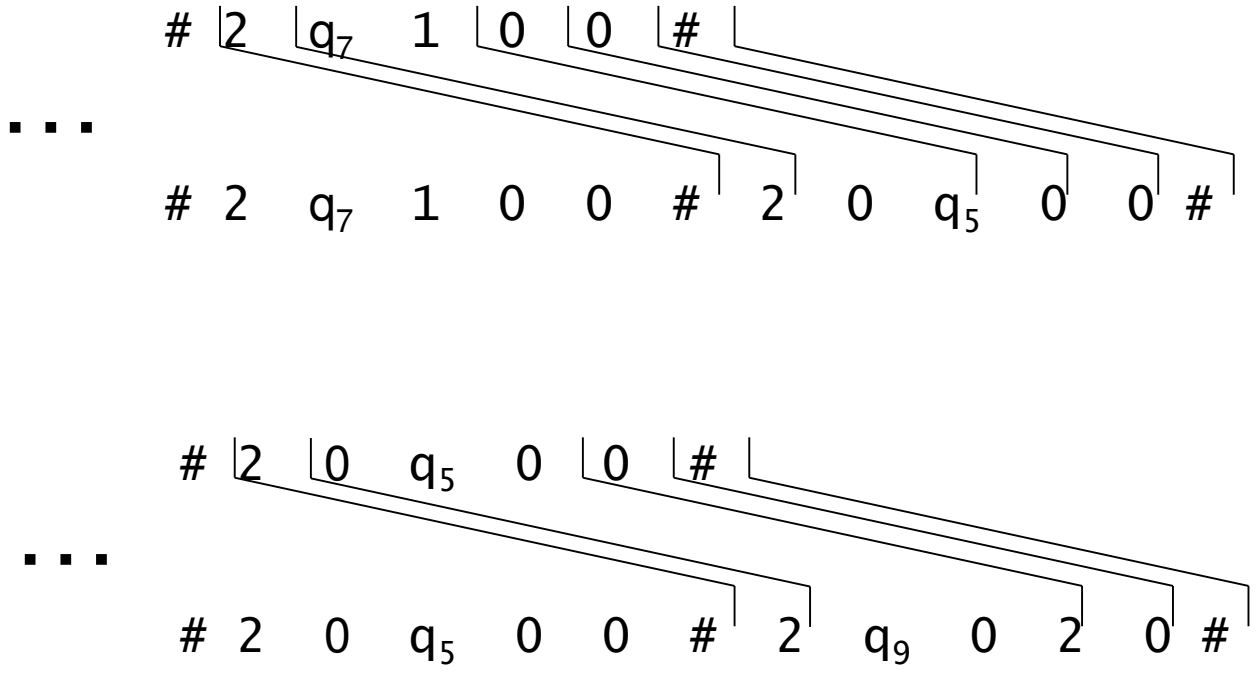
q_{accept}##
#

- Con questa costruzione, abbiamo una istanza per MPCP che ammette soluzione se e solo se M accetta w.

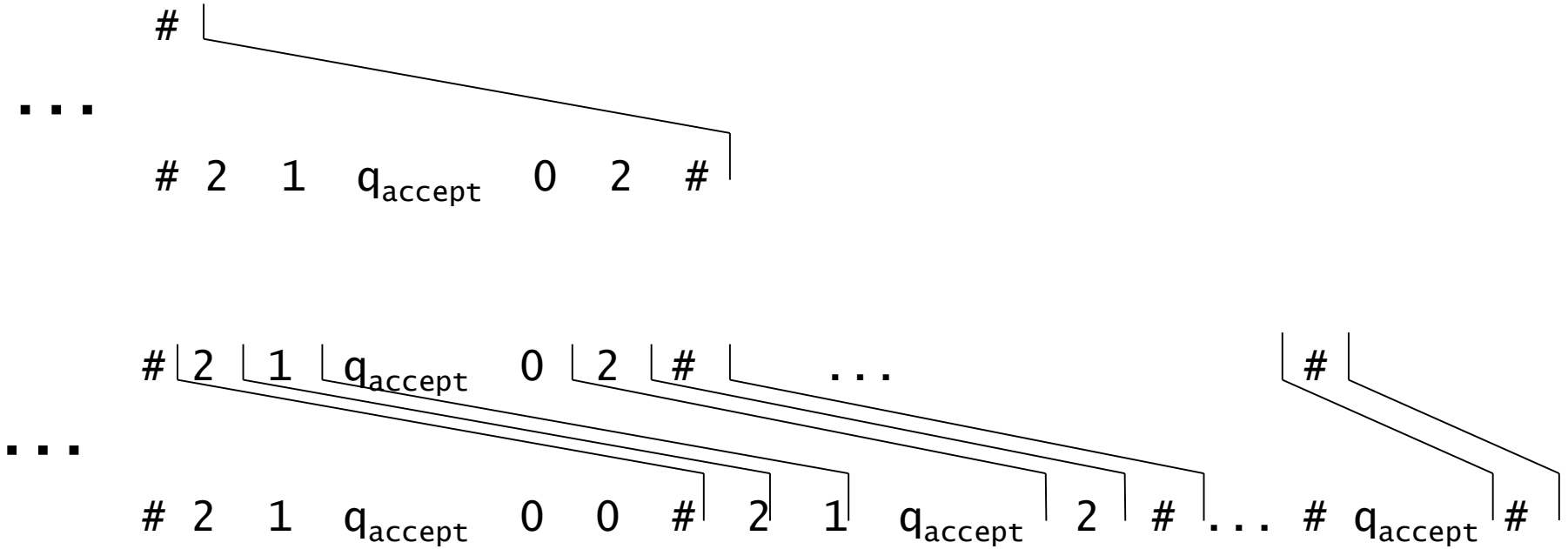
Esempio: Step 1, Step 2 + Step 3

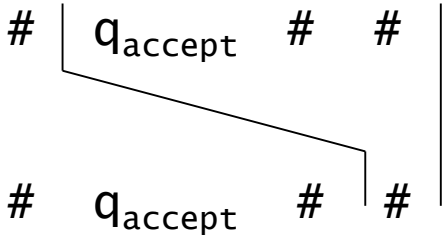


Esempio: Step 2+3, Step 2+4



Esempio: Step 2+4, Step 2+5




#
q_{accept}
#
#
#
q_{accept}
#
#

Da MPCP a PCP

- Per completare la dimostrazione, dobbiamo forzare il fatto che il primo domino deve essere il primo ad essere usato in ogni possibile soluzione del PCP.
- Sia "⋆" un nuovo simbolo (cioè non presente in Γ U {#}).
- Per ogni stringa s, sia⋆s la stringa ottenuta inserendo il simbolo "⋆"
- Prima di ogni simbolo in s. Per esempio, ⋆(abc) = ⋆a ⋆ b ⋆ c.
- Analogamente, definiamo s⋆ e ⋆s⋆ come la stringa ottenuta aggiungendo solo dopo, oppure sia prima che dopo, ogni simbolo di s il simbolo ⋆
- Dato l'insieme di domino P costruiti precedentemente, si sostituiscano i domino nel modo seguente:
  - t1/b1 = t1⋆/ ⋆ b1⋆
  - ogni altro ti/bi= ⋆ ti / b1⋆
  - il domino finale q_{accept}##/ # = ⋆q_{accept}# ⋆ #/#
- In questo modo, il primo domino deve essere assolutamente il primo altrimenti non ci sarà matching

- Provare che il seguente problema è indecidibile.

Problema del domino:

Dato un insieme finito di colori e un insieme di piastrelle quadrate con i quattro lati colorati di un qualche colore tra quelli a disposizione, trovare, se esiste, un modo per piastrellare una parete infinita in lunghezza e larghezza assicurando che i lati adiacenti delle piastrelle abbiano lo stesso colore.

Aniello Murano

Decidibilità delle teorie logiche

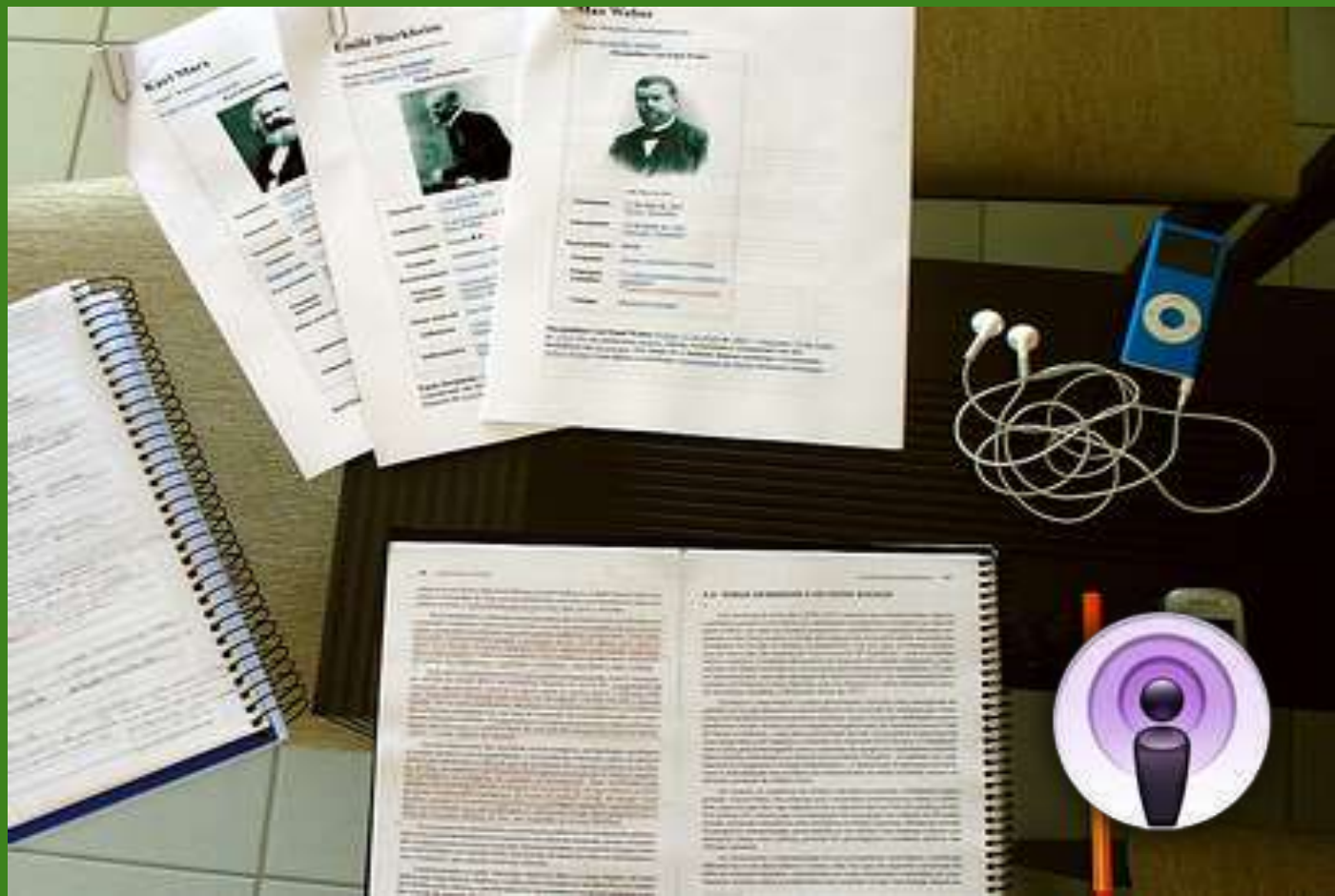
Lezione n.11
Parole chiave:
Teorie logiche

Corso di Laurea:
Informatica

Codice:

Email Docente:
murano@ na.infn.it

A.A. 2008-2009



Prefazione

- Nelle lezioni precedenti abbiamo trattato il concetto di decidibilià e indecidibilità nella teoria della computabilità.
- In questo contesto, possiamo dire che un insieme A è detto decidibile o ricorsivo se esiste un algoritmo che ricevuto in input un qualsiasi elemento, termina restituendo in output 0 o 1 a seconda che il valore appartenga o no all'insieme A.
- In questa lezione tratteremo la decidibilità e l'indecidibilità di teorie nella logica matematica.
- In particolare, concentreremo la nostra attenzione sul problema di determinare se affermazioni matematiche sono vere o false e investigheremo la decidibilità di questo problema.
- Si vedrà che la decidibilità dipende dal particolare dominio matematico in cui le affermazioni sono descritte.

Esempi di affermazioni matematiche

- Esempi di affermazioni matematiche sono i seguenti

1. $\forall q\,\exists p\,\forall x,y\,\left[p>q \wedge (x,y>1 \rightarrow xy\neq p)\right]$,
2. $\forall a,b,c,n\,\left[(a,b,c>0 \wedge n>2) \rightarrow a^n+b^n\neq c^n\right]$, and
3. $\forall q\,\exists p\,\forall x,y\,\left[p>q \wedge (x,y>1 \rightarrow (xy\neq p \wedge xy\neq p+2))\right]$

- La prima formula asserisce che esistono infiniti numeri primi. Questa affermazione è nota essere vera dal tempo di Euclide (più di 2300 anni fa).
- La seconda formula corrisponde all'ultimo teorema di Fermat che è stato dimostrato pochi anni fa ad opera di Andrew Wiles.
- La terza formula asserisce che esistono infinite coppie di numeri primi che differiscono di solo due unità. Questa è solo una congettura ed è tuttora non dimostrata ne confutata.
- **Nota:** Spiegheremo formalmente il loro significato nelle prossime diapositive…

Dalle logiche ai linguaggi

- Al fine di automatizzare il processo di determinazione di verità delle affermazioni matematiche è utile considerare queste affermazioni come stringhe e definire un linguaggio formato da tutte le affermazioni vere.
- Il problema della determinazione di verità delle affermazioni si riduce a alla decidibilità di questo linguaggio

Definizione del linguaggio

- Per la definizione del linguaggio si consideri il seguente alfabeto:

$$\{\wedge, \vee, \neg, (,), \forall, x, \exists, R_1, \ldots, R_k\}$$

  - Λ, V, e ¬, corrispondo alle operazioni booleane and,or e not*;*
  - *"(" e* ")" sono le parentesi*;*
  - *forall ed* *exists* sono i quantificatori universale ed esistenziale*;*
  - x denota variabili*;*
  - R_1 , ... , R_k sono *relazioni.*
- *Una formula è una stringa sull'alfabeto dato*
- Una stringa della forma $R_i(x_i, \ldots, x_j)$ *è una formula atomica.* Il valore j è l'arità della relazione R_i.
- Una formula (ben formata), (in breve fbf)
  1. è una formula atomica,
  2. è una combinazione booleana di altre formule più piccole
  3. è una quantificazione su altre formule f del tipo \exist x_i [f] oppure \forall x_i [f]
- **Nota:** I quantificatori legano le variabili all'interno del loro "scope" (parentesi quadre). Se una variabile non è legata in una formula allora la variabile è detta **libera**. Le formule senza variabili libere sono dette **sentenze** o ***statements***.

Logica del primo ordine(1)

Formule ben Formate:

- $R_1(x1) \Lambda R_2(x1, x2, x3)$
- $\forall x_1[R_1(x_1) \Lambda R(x_1, x_2, x_3)]$
- $\forall x_1 \exists x_2 \exists x_3[R_1(x_1) \Lambda R_2(x_1, x_2, x_3)]$

Osservazione: solo l'ultima fbf è una sentenza.

- L'ultima si legge, per ogni x_1 esistono x_2 e x_3 tali che $R_1(x_1)$ e $R_2(x_1, x_2, x_3)$ sono veri

Logica del primo ordine(2)

Costruendo tale sistema possiamo ragionare su sentenze del tipo

1. $\forall q, \exists x, y[p>q \wedge (x,y>1 \rightarrow xy \neq p)]$. (infiniti numeri primi)

2. $\forall a, b, c, n[(a,b,c>0 \wedge n>2) \rightarrow a^n + b^n \neq cn]$. (ultimo teorema di Fermat)

3. $\forall q \exists p \forall x, y[p>q \wedge (x,y>1 \rightarrow (xy \neq p \wedge xy \neq p+2))]$.
   (congettura sui numeri primi gemelli)

Logica del primo ordine(3)

- Per avere senso, una logica ha bisogno che le venga assegnato un significato. Per fare questo, abbiamo bisogno di assegnare la sintassi a uno specifico costrutto matematico, chiamato modello.

- Un modello è composto da un **universo** e un insieme di **relazioni**, una per ogni simbolo di relazione nella logica.

- Esempio:
  - sia Σ = {Λ, V, e $\neg$, (,), \forall, \exists, x,$R_1(\cdot\ ,\ \cdot)$}.
  - Un modello per questa logica è $M_1 = (N,\leq)$, con x $\rightarrow$ N and R1 $\rightarrow \leq$.
  - N è l'universo e la relazione $\leq \epsilon$ N $\times$ N è l'*interpretazione* per il simbolo di relazione binaria R_1.

Logica del primo ordine(4)

- Data una logica e un modello, possiamo verificare se una particolare sentenza è vera nel modello.
- Esempio 1:
  - Data la logica Σ = {Λ, V, e ¬, (,), \forall, \exists, x,R_1(·, ·)}, col modello M_1 = (N,≤).
  - Possiamo chiederci se la sentenza \forall x \exists y[R_1(x, y) V R_1(y, x)] è vera.
  - Chiaramente la sentenza è vera, visto che per ogni assegnamento x → a e y → b per a, b ϵ N, abbiamo che a ≤ b or b ≤ a.
- Esempio 2:
  - Data la logica Σ = {Λ, V, e ¬, (,), \forall, \exists, x,R_1(·, ·)}, col modello M_2 = (N,<)
  - Possiamo dire che la sentenza \forall x \forall y[R_1(x, y) V R_1(y, x)] non è vera. Questo perché per l'assegnamento x → a e y → a con a ϵ N abbiamo a < a or a < a, che è chiaramente falso.
- Esempio 3:
  - Data la logica Σ = {Λ, V, e ¬, (,), \forall, \exists, x,R_1(· , · , ·)}, col modello M_3 = (R,+) e R_1 relazione ternaria
  - possiamo dire che la sentenza \forall y \exists x[R_1(x, x, y)] è vera. Infatti per ogni assegnamento x → a e y → b con a, b ϵ R abbiamo che +(a, a, b), o nella classica notazione matematica b = a + a, è vera. Falso se il dominio è N

Una teoria decidibile

- Sia M un modello. Diremo che la collezione di tutte le sentenze vere sotto quel modello è la *teoria* del modello e scriveremo Th(M).
- **Teorema: la teoria Th(N,+) è decidibile.**
- Cosa significa che una teoria è decidibile? Significa che noi possiamo decidere se una particolare sentenza appartiene alla teoria o no. Quindi possiamo trattare la teoria Th(N,+) come un linguaggio e possiamo costruire un decisore per questo linguaggio.
- Consideriamo la sentenza \forall x \exists y[y = x + x]. Questa sentenza è vera ed è anche un elemento della teoria Th(N,+).
- Consideriamo ora \exists x \forall y[y = x + x]. Questa sentenza è falsa è quindi non è un membro della teoria.
- E' possibile mostrare la decidibilità della teoria Th(N,+) costruendo un automa finito che decide il linguaggio.

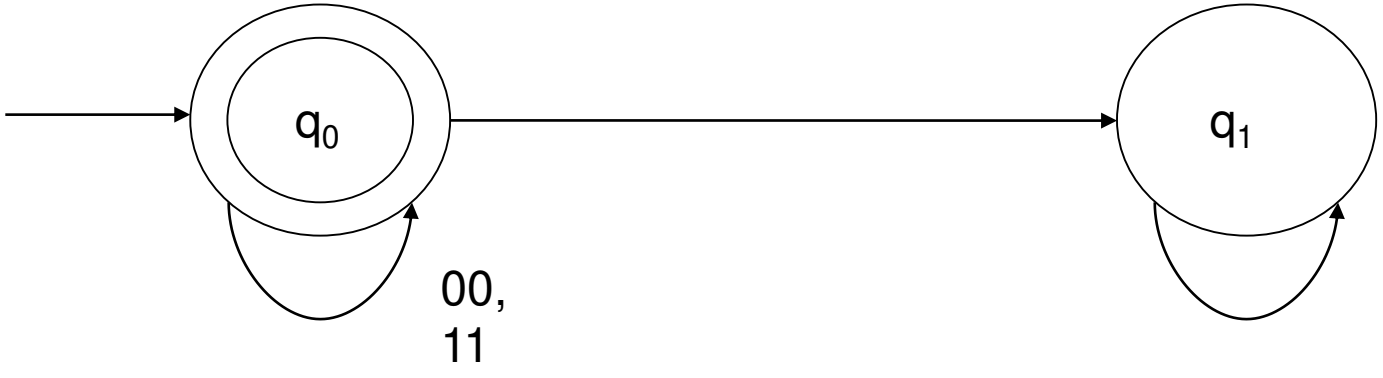
Premessa alla prova: operazioni con automi(1)

Sia Σ = { 00, 01, 10, 11} dove la coppia di nuperi ij rappresenta un elemento di una matrice trasposta 2 x 1 di binari.

Si noti che ogni parola costruita su Σ rappresenta due numeri binari. Per esempio 00 11 10 10 rappresenta i numeri 0111 e 0100

Sia A = { w ϵ Σ^* | la prima riga sia uguale alla seconda}.

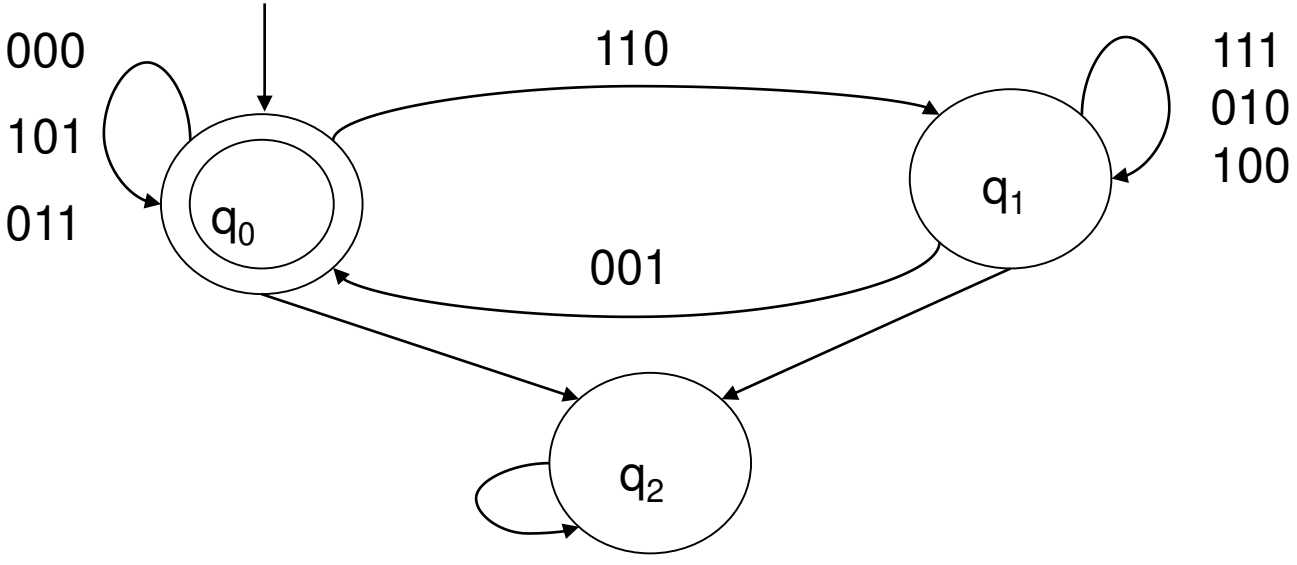
Esempio: 00 11 00 11 11 11 ϵ A and $\neg$(11 00 10 00 11 01 00 ϵ A)



Premessa alla prova: operazioni con automi(2)

- Sia Σ = {000, 001, 010, 011, 100, 101, 110, 111}.

- Si consideri il seguente linguaggio:
  B = { w ϵ Σ* | la somma delle prime due righe è uguale alla terza}
  Per esempio, 001 110 011 ϵ B, mentre ¬ (001 110 010 001 ϵ B)



Prova di decidibilità di Th(N,+)

- Sia $\varphi = Q_1 x_1 \ldots Q_n x_n(\psi)$ una sentenza di Th(N,+), dove
  - ciascun Q_i è un quantificatore esistenziale ($\exists$) o universale ($\forall$)
  - ψ non ha quantificatori.
- Sia inoltre $\varphi_i = Q_i x_i \ldots Q_n x_n (\psi)$. In particolare siano $\varphi_0 = \varphi$ e $\varphi_n = \psi$.
- Sia Σ_i l'alfabeto di tutte le parole binarie di lunghezza ì.
- Si costruisca A_n su Σ_n che accetti tutte le parole che rendano vera $\varphi_n = \psi$.
  - Si noti che ψ non ha quantificatori e solo operazioni di somma.
- Si costruisca A_i a partire da A_{i+1}, nel seguente modo:
- Si assuma che Q_i sia esistenziale. Allora costruire A_i in modo da fare una scelta non deterministica sull'i+1-esimo elemento di Σ.
- Se Q_i è universale, allora a fronte della equivalenza $\forall x_{i+1}\, \varphi_{i+i} = \neg\exists x_{i+1}\, \neg\varphi_{i+i}$ si costruisce prima il complemento di A_{i+1} poi si applica il procedimento precedente per Q_i esistenziale e poi si complementa l'automa ottenuto
- L'automa A_0 accetta qualche input se e solo se $\varphi_0 = \varphi$ è vera.
- Dunque, l'ultimo passo dell'algoritmo è testare il vuoto A_0.

Una teoria non decidibile

- Il seguente teorema ha delle conseguenze filosofiche molto profonde.
- Esso mostra che la matematica non può essere "meccanizzata".
- Mostra inoltre che alcuni problemi nella teoria dei numeri non sono algoritmici, cosa che provocò una grande sorpresa nei matematici all'inizio del 1900.
- Allora infatti si credeva che tutti i problemi matematici potessero essere risolti algoritmicamente e che bisognasse solo trovare l'algoritmo per risolvere un dato problema.
- **Teorema: la teoria Th(N,+,×) è indecidibile.**
- Questo significa che non esiste un algoritmo che si ferma su tutte le sentenze φ sull'alfabeto appropriato. Quello che più sorprende è la semplicità della struttura di questa logica indecidibile.
- Questo vuol dire che ci sono delle fondamentali limitazioni algoritmiche nella matematica.
- La dimostrazione si ottiene tramite una riduzione del problema A_{TM} alla teoria Th(N,+,×).

Teorema di incompletezza(1)

- **Teorema: la collezione di sentenze provabili in Th(N,+,×) è Turing-riconoscibile.**
- **Dimostrazione:**
  - il seguente algoritmo P accetta il suo input φ se e solo se φ è dimostrabile.
  - L'algoritmo P prova tutte le possibili stringhe come candidate per una prova π di φ usando un "proof checker"(verificatore della prova).
  - Se viene trovata una stringa che è una prova, allora l'algoritmo accetta.

Teorema di incompletezza(2)

- **Teorema (di incompletezza di Kurt Gödels): alcune sentenze vere in Th(N,+,×) non sono dimostrabili**.
- Con qualche semplificazione, questo teorema afferma che

  "In ogni formalizzazione coerente della matematica che sia sufficientemente potente da poter assiomatizzare la teoria elementare dei numeri naturali — vale a dire, sufficientemente potente da definire la struttura dei numeri naturali dotati delle operazioni di somma e prodotto — è possibile costruire una proposizione sintatticamente corretta che non può essere né dimostrata né confutata all'interno dello stesso sistema."