# Esercizio 1 - TM e varianti - Save and Restore

1. **(12 punti)** Una macchina di Turing con "save e restore" (SRTM) è una macchina di Turing deterministica a singolo nastro, che può salvare la configurazione corrente per poi ripristinarla in un momento successivo. Oltre alle normali operazioni di spostamento a sinistra e a destra, una SRTM può effettuare l'operazione di SAVE, che salva la configurazione corrente, e l'operazione di RESTORE che ripristina la configurazione salvata. L'operazione di SAVE sovrascrive una eventuale configurazione salvata in precedenza. Fare il RESTORE in assenza di configurazione salvata non ha effetto: si mantiene inalterata la configurazione corrente.

   (a) Dai una definizione formale della funzione di transizione di una SRTM.

   (b) Dimostra che le SRTM riconoscono la classe dei linguaggi Turing-riconoscibili. Usa una descrizione a livello implementativo per definire le macchine di Turing.

# (a) Definizione formale della funzione di transizione di una SRTM

Una Macchina di Turing con "save e restore" (SRTM) è una tupla: **$M = (Q, \Sigma, \Gamma, \delta, q_0, q\_accept, q\_reject)$**

dove tutti i componenti sono definiti come per una TM standard, eccetto la funzione di transizione che ha la forma:

**$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, SAVE, RESTORE\}$**

Le operazioni SAVE e RESTORE operano sulla configurazione corrente della macchina (stato, posizione testina, contenuto del nastro). La macchina mantiene un registro per una configurazione salvata.

# (b) Dimostrazione dell'equivalenza con i linguaggi Turing-riconoscibili

Per dimostrare che le SRTM riconoscono esattamente la classe dei linguaggi Turing-riconoscibili, dobbiamo provare due inclusioni:

## Direzione 1: Ogni linguaggio Turing-riconoscibile è riconosciuto da una SRTM

Questa dimostrazione è banale: le TM deterministiche standard sono un caso particolare di SRTM che non effettuano mai le operazioni SAVE o RESTORE. Di conseguenza, ogni linguaggio Turing-riconoscibile è riconosciuto da una SRTM.

## Direzione 2: Ogni linguaggio riconosciuto da una SRTM è Turing-riconoscibile

Mostriamo come convertire una SRTM M in una TM deterministica a nastro singolo S equivalente.

**S = "Su input w:**

1. **Inizializzazione:** S organizza il suo nastro per simulare sia il nastro di lavoro di M che lo spazio per memorizzare una configurazione salvata. Il nastro ha il formato: `#w#⊥` dove `#` marca i confini della parte di lavoro e `⊥` segna l'inizio della zona di backup.
2. **Simulazione mosse L/R:** Per le transizioni δ(q,a) = (r,b,L) e δ(q,a) = (r,b,R), S opera come una TM standard: scrive b, sposta la testina nella direzione appropriata e cambia stato. Se raggiunge un marcatore `#`, gestisce il confine appropriatamente.
3. **Simulazione SAVE:** Per δ(q,a) = (r,b,SAVE), S:
   - Scrive b nella cella corrente
   - Copia l'intera configurazione corrente (contenuto nastro + posizione testina + stato r) nella zona di backup dopo `⊥`, sovrascrivendo eventuali backup precedenti
   - Continua nello stato r dalla posizione corrente
4. **Simulazione RESTORE:** Per δ(q,a) = (r,b,RESTORE), S:
   - Se non esiste configurazione salvata (zona dopo `⊥` è vuota), continua normalmente scrivendo b e andando nello stato r
   - Se esiste configurazione salvata, la ripristina: copia il contenuto del backup nella zona di lavoro, ripristina la posizione della testina e lo stato salvato
5. **Gestione stati finali:** Se M raggiunge q_accept o q_reject, S fa lo stesso."

Questa costruzione dimostra che S simula correttamente M, quindi ogni linguaggio riconosciuto da M è anche riconosciuto da S, che è una TM standard.

**Conclusione:** Le SRTM riconoscono esattamente la classe dei linguaggi Turing-riconoscibili. ■

# Esercizio 2 - TM e varianti - TM a griglia

**1.** *Una* macchina di Turing bidimensionale *utilizza una griglia bidimensionale infinita di celle come nastro. Ad ogni transizione, la testina può spostarsi dalla cella corrente ad una qualsiasi delle quattro celle adiacenti. La funzione di transizione di tale macchina ha la forma*

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{\uparrow, \downarrow, \rightarrow, \leftarrow\},$$

*dove le frecce indicano in quale direzione si muove la testina dopo aver scritto il simbolo sulla cella corrente.*

*Dimostra che ogni macchina di Turing bidimensionale può essere simulata da una macchina di Turing deterministica a nastro singolo.*

## Dimostrazione della simulazione di TM bidimensionale con TM a nastro singolo

**Teorema:** Ogni macchina di Turing bidimensionale può essere simulata da una macchina di Turing deterministica a nastro singolo.

**Dimostrazione:** Mostriamo come simulare una TM bidimensionale B con una TM deterministica a nastro singolo S.

La strategia consiste nel rappresentare la griglia bidimensionale sul nastro unidimensionale come una sequenza di stringhe (righe) separate da marcatori speciali.

**S = "Su input w:**

1. **Inizializzazione:** Sostituisce w con la configurazione iniziale `##w##` e marca con $\overline{b}$ il primo simbolo di w per indicare la posizione della testina.
2. **Localizzazione:** Scorre il nastro finché non trova la cella marcata con $\overline{b}$ (posizione corrente della testina bidimensionale).
3. **Simulazione delle transizioni:** Aggiorna il nastro secondo la funzione di transizione δ(q,a) = (r,b,dir) di B:
   - **Per δ(q,a) = (r,b,→):** Scrive b non marcato nella cella corrente, sposta $\overline{b}$ sulla cella immediatamente a destra. Sposta di una cella a destra tutte le marcature con pallino `•`. Se una marcatura raggiunge un `#`, scrive un blank marcato al posto di `#` e sposta il contenuto del nastro da questa cella fino al `##` finale di una cella più a destra.
   - **Per δ(q,a) = (r,b,←):** Scrive b non marcato nella cella corrente, sposta $\overline{b}$ sulla cella immediatamente a sinistra. Sposta di una cella a sinistra tutte le marcature con pallino `•`. Se una marcatura raggiunge un `#`, scrive un blank marcato al posto di `#` e sposta il contenuto del nastro da questa cella fino al `##` iniziale di una cella più a sinistra.
   - **Per δ(q,a) = (r,b,↑):** Scrive b marcato con pallino `•` nella cella corrente, e sposta $\overline{b}$ sulla prima cella marcata con pallino `•` posta a sinistra della cella corrente. Se questa cella non esiste, aggiunge una nuova riga composta solo da blank all'inizio della configurazione.
   - **Per δ(q,a) = (r,b,↓):** Scrive b marcato con pallino `•` nella cella corrente, e sposta $\overline{b}$ sulla prima cella marcata con pallino `•` posta a destra della cella corrente. Se questa cella non esiste, aggiunge una nuova riga composta solo da blank alla fine della configurazione.
4. **Gestione stati finali:** Se B raggiunge q_accept o q_reject, S termina nello stesso stato.
5. **Iterazione:** Se non si è in uno stato finale, ripeti dal punto 2."

**Spiegazione del meccanismo:**

- Il nastro rappresenta la griglia 2D come sequenza di righe separate da `#`
- `##` marca l'inizio e la fine della rappresentazione della griglia
- $\overline{b}$ indica la posizione corrente della testina bidimensionale
- `•` marca le colonne per facilitare i movimenti verticali

- L'espansione dinamica gestisce l'accesso a nuove celle della griglia

Questa costruzione dimostra che S simula correttamente ogni computazione di B, quindi le TM bidimensionali non hanno maggiore potere computazionale delle TM standard. ■

# 1. SIMULAZIONE CON TM A NASTRO SEMI-INFINITO

## Variante A: Codifica sequenziale delle righe

**Definizione formale:** Sia B = (Q_B, Σ, Γ_B, δ_B, $q_0$, q_accept, q_reject) una TM bidimensionale con δ_B : Q_B × Γ_B → Q_B × Γ_B × {↑, ↓, →, ←}.

Costruiamo S = (Q_S, Σ, Γ_S, δ_S, $q_0$', q_accept, q_reject) dove:

- Q_S = Q_B ∪ {stati ausiliari per navigazione}
- Γ_S = Γ_B ∪ {#, •, ⌐}
- δ_S : Q_S × Γ_S → Q_S × Γ_S × {L, R}

**S = "Su input w:**

1. **Inizializzazione:** Sostituisce w con ⌐#w#⌐ e marca con • il primo simbolo per indicare la posizione della testina 2D.
2. **Localizzazione:** Scorre a destra fino a trovare la cella marcata con •.
3. **Simulazione transizioni:**
   - **δ_B(q,a) = (r,b,→):** Scrive b, rimuove •, sposta • una cella a destra. Se incontra #, espande la riga corrente.
   - **δ_B(q,a) = (r,b,←):** Scrive b, rimuove •, sposta • una cella a sinistra. Se va oltre #, espande a sinistra.
   - **δ_B(q,a) = (r,b,↑):** Salta alla riga precedente: scorre a sinistra fino al # precedente, posiziona • nella colonna corrispondente.
   - **δ_B(q,a) = (r,b,↓):** Salta alla riga successiva: scorre a destra fino al # successivo, posiziona • nella colonna corrispondente.
4. **Iterazione:** Ripeti dal punto 2 finché non si raggiunge q_accept o q_reject."

## Variante B: Indirizzamento tramite coordinate

**Definizione formale:** Stessa struttura di base, ma con Γ_S = Γ_B ∪ {#, ⊥, ∇, (i,j)} dove (i,j) rappresenta coordinate.

**S = "Su input w:**

1. **Inizializzazione:** Crea nastro formato ⊥(0,0):$w_1$#(0,1):$w_2$#...#⊥ con coordinate esplicite.
2. **Navigazione:** Mantiene registro della posizione corrente (x,y) negli stati interni.
3. **Simulazione transizioni:**
   - **Movimenti orizzontali:** Incrementa/decrementa coordinata x, cerca cella target.

- **Movimenti verticali:** Incrementa/decrementa coordinata y, cerca/crea riga target.

4. **Gestione espansione:** Aggiunge nuove righe/colonne quando necessario."

**Definizione formale della TM simulatrice:**

Sia B = (Q_B, Σ, Γ_B, δ_B, $q_0$_B, q_accept, q_reject) una TM bidimensionale con:

- δ_B : Q_B × Γ_B → Q_B × Γ_B × {↑, ↓, →, ←}

Costruiamo S = (Q_S, Σ, Γ_S, δ_S, $q_0$_S, q_accept, q_reject) dove:

- **Q_S = Q_B × ℤ × ℤ × {SEEK, WRITE, EXPAND_H, EXPAND_V}**
  - (q,x,y,mode) dove (x,y) sono le coordinate correnti e mode è la modalità operativa
- **Γ_S = Γ_B ∪ {⊥, #, :, (, ), 0, 1, 2, ..., 9, -, •}**
  - ⊥: marcatore inizio/fine griglia
  - #: separatore tra celle
  - :: separatore coordinate-contenuto
  - (,): delimitatori coordinate
  - cifre: per rappresentare numeri nelle coordinate
  - •: marcatore posizione corrente

**Formato nastro:** $\perp(x_1,y_1):c_1\#(x_2,y_2):c_2\#...\#(x_n,y_n):c_n\perp$

**Algoritmo di simulazione S:**

**S = "Su input w = $a_1a_2...a_k$:**

# Fase 1: Inizializzazione

1. Scrive sul nastro: $\perp(0,0):a_1\bullet\#(0,1):a_2\#...\#(0,k\text{-}1):a_k\perp$
2. Inizializza stato interno: ($q_0$_B, 0, 0, SEEK)
3. Posiziona testina sul primo carattere dopo ⊥

# Fase 2: Localizzazione (mode = SEEK)

1. **Ricerca posizione target (x_target, y_target):**
   - Scorre il nastro da sinistra verso destra
   - Per ogni cella $(x_i,y_i):c_i$, confronta $(x_i,y_i)$ con (x_target, y_target)
   - Se trova corrispondenza: passa a mode = WRITE
   - Se raggiunge ⊥ finale senza trovare: passa a EXPAND

# Fase 3: Simulazione Transizioni (mode = WRITE)

Per δ_B(q, a) = (r, b, dir):

1. **Lettura:** Legge contenuto $c_i$ nella posizione corrente (x,y)
2. **Scrittura:** Sostituisce $c_i$ con b, rimuove • se presente
3. **Aggiornamento coordinate target:**
   - **dir = →:** (x_target, y_target) ← (x+1, y)
   - **dir = ←:** (x_target, y_target) ← (x-1, y)
   - **dir = ↓:** (x_target, y_target) ← (x, y+1)
   - **dir = ↑:** (x_target, y_target) ← (x, y-1)
4. **Aggiornamento stato:** (q,x,y,mode) ← (r, x_target, y_target, SEEK)

# Fase 4: Espansione Griglia

**Se SEEK non trova (x_target, y_target):**

1. **Localizzazione punto inserimento:**
   - Cerca posizione ordinata dove inserire (x_target, y_target)
   - Ordine lessicografico: prima per y, poi per x
2. **Inserimento nuova cella:**
   - Sposta contenuto esistente a destra per fare spazio
   - Inserisce (x_target, y_target):⊥#
   - Aggiorna • per marcare nuova posizione
3. **Ritorna a mode = SEEK**

# Fase 5: Gestione Stati Finali

- Se r = q_accept: termina con accettazione
- Se r = q_reject: termina con rifiuto
- Altrimenti: ritorna alla Fase 2

**Esempio di Evoluzione:**

```
Stato iniziale w="ab":
⊥(0,0):a•#(0,1):b⊥
Stato: (q₀, 0, 0, WRITE)

Dopo δ_B(q₀, a) = (q₁, x, →):
⊥(0,0):x#(0,1):b•⊥
Stato: (q₁, 1, 0, SEEK)

Cella (1,0) non esiste → Espansione:
⊥(0,0):x#(0,1):b#(1,0):⊥•⊥
Stato: (q₁, 1, 0, WRITE)
```

**Funzioni di Transizione Chiave:**

- **δ_S((q,x,y,SEEK), (i,j):c) =**
  - Se (i,j) = (x,y): ((q,x,y,WRITE), (i,j):c•, R)
  - Altrimenti: ((q,x,y,SEEK), (i,j):c, R)
- **δ_S((q,x,y,WRITE), c•) =**
  - Applica transizione di B: se δ_B(q,c) = (r,b,dir)
  - Aggiorna coordinate secondo dir
  - ((r,x',y',SEEK), b, calcola_direzione_per_ricerca)

**Correttezza:** La simulazione preserva la semantica bidimensionale mappando ogni configurazione (q, griglia_2D, (x,y)) di B in una configurazione equivalente (q', nastro_1D, posizione) di S.

**Complessità:** Ogni transizione di B richiede O(n) passi in S, dove n è il numero di celle utilizzate nella griglia 2D.

Questa costruzione dimostra formalmente che le TM bidimensionali non sono più potenti delle TM a nastro semi-infinito. ■

# 2. SIMULAZIONE CON TM MULTINASTRO

## Variante A: Nastro per righe + nastro indici

**Definizione formale:** Sia M = (Q_M, Σ, $\Gamma_1 \times \Gamma_2$, δ_M, $q_0$, q_accept, q_reject) una TM a 2 nastri con:

- δ_M : Q_M × ($\Gamma_1 \times \Gamma_2$) → Q_M × ($\Gamma_1 \times \Gamma_2$) × ({L,R} × {L,R})

**M = "Su input w:**

1. **Nastro 1 (Dati):** Memorizza le righe della griglia separate da #: ##$riga_0$#$riga_1$#$riga_2$##
2. **Nastro 2 (Indici):** Mantiene informazioni di posizionamento: [riga_corrente] [colonna_corrente]
3. **Simulazione:**
   - **δ_B(q,a) = (r,b,→):** Sposta testina 1 a destra nel nastro dati, aggiorna colonna nel nastro 2.
   - **δ_B(q,a) = (r,b,←):** Sposta testina 1 a sinistra nel nastro dati, aggiorna colonna nel nastro 2.
   - **δ_B(q,a) = (r,b,↑):** Cerca riga precedente nel nastro 1, aggiorna riga nel nastro 2.
   - **δ_B(q,a) = (r,b,↓):** Cerca riga successiva nel nastro 1, aggiorna riga nel nastro 2."

## Variante B: Nastro X + nastro Y + nastro ausiliario

**Definizione formale:** TM a 3 nastri M = (Q_M, Σ, $\Gamma_1 \times \Gamma_2 \times \Gamma_3$, δ_M, $q_0$, q_accept, q_reject) con:

- δ_M : Q_M × ($\Gamma_1 \times \Gamma_2 \times \Gamma_3$) → Q_M × ($\Gamma_1 \times \Gamma_2 \times \Gamma_3$) × ({L,R}³)

**M = "Su input w:**

1. **Nastro 1 (Asse X):** Memorizza una "striscia" orizzontale della griglia 2D contenente la posizione corrente.
2. **Nastro 2 (Asse Y):** Memorizza una "striscia" verticale della griglia 2D contenente la posizione corrente.
3. **Nastro 3 (Coordinamento):** Mantiene coordinate (x,y) correnti e gestisce la sincronizzazione.
4. **Simulazione:**
   - **Movimenti orizzontali:** Aggiorna nastro 1 e coordinate nel nastro 3.
   - **Movimenti verticali:** Aggiorna nastro 2 e coordinate nel nastro 3.
   - **Sincronizzazione:** Il nastro 3 orchestra l'aggiornamento degli altri nastri quando si cambia "striscia"."

**Conclusione:** Tutte e quattro le varianti dimostrano che le TM bidimensionali non possiedono maggiore potere computazionale delle TM standard, confermando la robustezza del modello di Turing attraverso diverse rappresentazioni implementative. ■

# Esercizio 3 - Dimostra L regolare - SWAP

1. **(12 punti)** Data una stringa $w \in \Sigma^*$, definiamo una operazione che scambia di posizione i caratteri della stringa a due a due:

$$\mathrm{SWAP}(w) = \begin{cases} \varepsilon & \text{se } w = \varepsilon \\ a & \text{se } w = a \text{ con } a \in \Sigma \\ a_1 a_0 \mathrm{SWAP}(u) & \text{se } w = a_0 a_1 u \text{ con } a_0, a_1 \in \Sigma, u \in \Sigma^* \end{cases}$$

Per esempio, SWAP(ABCDE) = BADCE.

Dimostra che se $L \subseteq \Sigma^*$ è un linguaggio regolare, allora anche il seguente linguaggio è regolare:

$$\mathrm{SWAP}(L) = \{\mathrm{SWAP}(w) \mid w \in L\}.$$

# Teorema: Chiusura dei Linguaggi Regolari sotto SWAP

**Enunciato:** Se L ⊆ Σ* è un linguaggio regolare, allora SWAP(L) = {SWAP(w) | w ∈ L} è regolare.

**Dimostrazione costruttiva:**

Sia L un linguaggio regolare. Allora esiste un DFA **M = (Q, Σ, δ, $q_0$, F)** che riconosce L.

Costruiamo un NFA **M' = (Q', Σ, δ', $q_0$', F')** che riconosce SWAP(L).

# Definizione formale di M':

**Q' = Q ∪ (Q × Σ) ∪ {$q_0$'}**

- Q: stati originali di M
- Q × Σ: stati "memoria" (q,a) che ricordano di aver letto il carattere a nello stato q
- $q_0$': nuovo stato iniziale

**$q_0$' ∉ Q ∪ (Q × Σ)** (stato iniziale fresco)

**F' = F ∪ {(q,a) | δ(q,a) ∈ F}**

- F: stati finali originali (per stringhe di lunghezza pari)
- {(q,a) | δ(q,a) ∈ F}: stati finali per stringhe di lunghezza dispari

# Funzione di transizione δ':

**1. Inizializzazione:** $\delta'(q_0', \varepsilon) = \{q_0\}$

**2. Lettura primo carattere di ogni coppia:** ∀q ∈ Q, ∀a ∈ Σ: δ'(q, a) = {(q, a)} ∪ (F se δ(q,a) ∈ F)

**3. Lettura secondo carattere e scambio:** ∀q ∈ Q, ∀a,b ∈ Σ: δ'((q, a), b) = {δ(δ(q, a), b)}

# Invariante di correttezza:

L'automa M' simula M sui caratteri "scambiati" a coppie:

- Quando legge $a_0a_1...a_{2n-1}a_{2n}u$ su M', simula M su $a_1a_0...a_{2n+1}a_{2n}$SWAP(u)

# Analisi dei casi:

**Caso 1: |w| = 0 (stringa vuota)**

- M' accetta ε tramite $\delta'(q_0', \varepsilon) = \{q_0\}$
- M' accetta iff $q_0 \in F$ iff $\varepsilon \in L$ iff SWAP(ε) = $\varepsilon \in$ SWAP(L) ✓

**Caso 2: |w| = 1 (w = a)**

- M' legge a: $q_0' \to^\varepsilon q_0 \to^a \{(q_0, a)\}$
- M' accetta iff $(q_0, a) \in F'$ iff $\delta(q_0, a) \in F$ iff $a \in L$ iff SWAP(a) = $a \in$ SWAP(L) ✓

**Caso 3: |w| ≥ 2 (w = $a_0a_1u$)**

- M' legge $a_0a_1$: $q_0' \to^\varepsilon q_0 \to^{a_0} (q_0, a_0) \to^{a_1} \delta(\delta(q_0, a_0), a_1) = \delta(q_0, a_1a_0)$
- Per induzione, M' simula correttamente SWAP(u) dalla posizione $\delta(q_0, a_1a_0)$

- M' accetta w iff M accetta $a_1a_0$SWAP(u) iff $a_0a_1u \in L$ iff SWAP($a_0a_1u$) = $a_1a_0$SWAP(u) $\in$ SWAP(L) ✓

## Esempio di esecuzione:

**Input:** w = ABCDE $\in$ L
**SWAP(w) = BADCE**

**Traccia di M' su ABCDE:**

1. $q_0' \to^\varepsilon q_0$
2. $q_0 \to^A (q_0, A)$
3. $(q_0, A) \to^B \delta(\delta(q_0, A), B) = \delta(q_0, BA)$
4. $\delta(q_0, BA) \to^C (\delta(q_0, BA), C)$
5. $(\delta(q_0, BA), C) \to^D \delta(\delta(\delta(q_0, BA), C), D) = \delta(q_0, BADC)$
6. $\delta(q_0, BADC) \to^E (\delta(q_0, BADC), E)$
7. Accetta iff $(\delta(q_0, BADC), E) \in F'$ iff $\delta(\delta(q_0, BADC), E) \in F$ iff $\delta(q_0, BADCE) \in F$ iff BADCE $\in$ L

**Conclusione:** M' riconosce esattamente SWAP(L), quindi SWAP(L) è regolare. ■

**Complessità:** $|Q'| = |Q| + |Q|\cdot|\Sigma| + 1 = O(|Q|\cdot|\Sigma|)$, quindi la costruzione è efficiente.

# Alternativa dimostrazione finale

# Teorema: w $\in$ L $\Leftrightarrow$ SWAP(w) $\in$ SWAP(L)

Usando la stessa tecnica bidirezionale della dimostrazione per **FLIP**, costruisco la dimostrazione:

## Setup iniziale:

Sia L un linguaggio regolare riconosciuto da DFA **A = (Q, Σ, δ, $q_0$, F)**.

Costruisco **A' = (Q', Σ, δ', $q_0'$, F')** che riconosce SWAP(L) dove:

- **Q' = Q $\cup$ (Q × Σ) $\cup$ {$q_0'$}**
- **F' = F $\cup$ {(q,a) | δ(q,a) $\in$ F}**

## Dimostrazione bidirezionale:

**$\Leftarrow$ Direzione:** Se **SWAP(w) $\in$ SWAP(L)** allora **w $\in$ L**

1. **Ipotesi:** SWAP(w) $\in$ SWAP(L)
2. **Per definizione di SWAP(L):** $\exists$v $\in$ L tale che SWAP(v) = SWAP(w)
3. **Proprietà di SWAP:** SWAP(SWAP(v)) = v, quindi v = SWAP(SWAP(w))

4. **Ma SWAP è involutiva:** SWAP(SWAP(w)) = w
5. **Quindi:** v = w, e poiché v ∈ L, abbiamo **w ∈ L** ✓

⟹ **Direzione:** Se **w ∈ L** allora **SWAP(w) ∈ SWAP(L)**

Dimostro per induzione sulla lunghezza di w:

**Caso base |w| ≤ 1:**

- Se w = ε: SWAP(ε) = ε, e ε ∈ L iff ε ∈ SWAP(L) ✓
- Se w = a: SWAP(a) = a, e a ∈ L iff a ∈ SWAP(L) ✓

**Passo induttivo |w| ≥ 2:**

- Sia w = $a_0a_1u$ dove |u| ≥ 0
- Per ipotesi: w = $a_0a_1u$ ∈ L
- Devo provare: SWAP(w) = $a_1a_0$SWAP(u) ∈ SWAP(L)

**Simulazione di A' su w = $a_0a_1u$:**

```
q₀' →^ε q₀                    [ε-transizione iniziale]
q₀ →^a₀ (q₀, a₀)              [memorizza primo carattere]
(q₀, a₀) →^a₁ δ(δ(q₀, a₀), a₁) = δ(q₀, a₁a₀)  [effettua lo scambio]
```

**Proprietà chiave:** $\delta(q_0, a_1a_0)$ è lo stato raggiunto da A leggendo $a_1a_0$.

**Per induzione su u:** A' simula correttamente SWAP(u) partendo da $\delta(q_0, a_1a_0)$.

**Quindi:** A' accetta w iff A accetta $a_1a_0$SWAP(u) = SWAP($a_0a_1u$) = SWAP(w).

**Poiché w ∈ L,** esiste una computazione accettante di A su w:

```
q₀ →^a₀ s₁ →^a₁ s₂ →^u sₙ ∈ F
```

**Invertendo lo scambio:** A ha anche una computazione:

```
q₀ →^a₁ s₁' →^a₀ s₂' →^SWAP(u) sₙ' ∈ F
```

dove gli stati $s_2'$, $s_n'$ sono determinati dalla struttura di SWAP.

**Conclusione:** A accetta SWAP(w), quindi **SWAP(w) ∈ L**.

Ma per definizione di SWAP(L): **SWAP(w) ∈ SWAP(L)** ✓

# Schema di correttezza (come nel diagramma):

```
w ∈ L  ⟺  [A accetta w]  ⟺  [A' simula correttamente]  ⟺  SWAP(w) ∈ SWAP(L)
  ↑                                                                    ↓
  └── Per involutività di SWAP: w = SWAP(SWAP(w)) ──────────────────────┘
```

## Conclusione:

La costruzione di A' preserva esattamente la relazione **w ∈ L ⟺ SWAP(w) ∈ SWAP(L)**, dimostrando che SWAP(L) è regolare quando L è regolare, utilizzando la stessa tecnica bidirezionale usata per FLIP. ■

**Nota:** La chiave è l'**involutività** di SWAP (come per FLIP), che garantisce la bidirezionalità della trasformazione.

# Esercizio 4 - Dimostra L regolare - CFG Lineare a dx

**3. (12 punti)** Una CFG è detta *lineare a destra* se il corpo di ogni regola ha al massimo una variabile, e la variabile si trova all'estremità di destra. In altre parole, tutte le regole di una grammatica lineare a destra sono nella forma $A \to wB$ o $A \to w$, dove $A$ e $B$ sono variabili e $w$ è una stringa di zero o più simboli terminali.

Dimostra che ogni grammatica lineare a destra genera un linguaggio regolare. *Suggerimento:* costruisci un $\varepsilon$-NFA che simula le derivazioni della grammatica.

# Teorema: CFG Lineari a Destra Generano Linguaggi Regolari

**Enunciato:** Ogni grammatica lineare a destra genera un linguaggio regolare.

**Dimostrazione costruttiva per ε-NFA:**

Sia **G = (V, Σ, R, S)** una CFG lineare a destra con regole della forma:

- A → wB (w ∈ Σ*, A,B ∈ V)
- A → w (w ∈ Σ*, A ∈ V)

## Osservazione strutturale delle derivazioni:

Le derivazioni di G hanno struttura particolare: ogni derivazione di una parola è formata da **n applicazioni** di regole A → wB seguite da **un'applicazione finale** di una regola A → w.

## Costruzione dell'ε-NFA:

Costruiamo **A = (Q, Σ, δ, $q_0$, F)** che simula le derivazioni di G.

**Notazione per transizioni estese:** Per stati p,q e stringa $w = w_1...w_n$, scriviamo **$q \in \delta(p, w)$** quando l'automa può andare da p a q consumando w, introducendo stati intermedi $r_1...r_{n-1}$:

- $\delta(p, w_1) \ni r_1$
- $\delta(r_1, w_2) = \{r_2\}$
- ⋮
- $\delta(r_{n-1}, w_n) = \{q\}$

# Definizione formale dell'automa:

**$Q = V \cup \{qf\} \cup E$** dove:

- V: insieme delle variabili della grammatica
- qf: unico stato finale
- E: stati ausiliari per transizioni estese

**$q_0 = S$** (variabile iniziale della grammatica)

**$F = \{qf\}$** (unico stato finale)

**Funzione di transizione δ:**

- **$B \in \delta(A, w)$** per ogni regola **$A \to wB \in R$**
- **$qf \in \delta(A, w)$** per ogni regola **$A \to w \in R$**

# Algoritmo di simulazione:

L'automa A simula le derivazioni di G come segue:

1. **Inizializzazione:** Parte dallo stato S (variabile iniziale)
2. **Ciclo di derivazione:** Ripete finché possibile:
   - **Stato corrente = variabile A:** Sceglie non deterministicamente una regola per A
   - **Regola $A \to wB$:** Consuma w dall'input e va nello stato B
   - **Regola $A \to w$:** Consuma w dall'input e va in qf (accettazione)
3. **Accettazione:** Se riesce a consumare tutto l'input e raggiungere qf

# Correttezza della costruzione:

**Lemma:** $w \in L(G) \Leftrightarrow w \in L(A)$

**$\Rightarrow$ Direzione (G ⊢ w implica A accetta w):**

Se $S \Rightarrow^* G\ w$, allora esiste una derivazione:

$$S \Rightarrow w_1B_1 \Rightarrow w_1w_2B_2 \Rightarrow ... \Rightarrow w_1w_2...w_nB_n \Rightarrow w_1w_2...w_nw_{n+1} = w$$

Corrispondentemente, A ha computazione:

```
S →^w₁ B₁ →^w₂ B₂ →^... →^wₙ Bₙ →^wₙ₊₁ qf
```

**⇐ Direzione (A accetta w implica G ⊢ w):**

Se A accetta w con computazione:

```
S →^w₁ B₁ →^w₂ B₂ →^... →^wₙ Bₙ →^wₙ₊₁ qf
```

Allora esistono regole corrispondenti che permettono la derivazione:

```
S ⇒ w₁B₁ ⇒ w₁w₂B₂ ⇒ ... ⇒ w₁w₂...wₙwₙ₊₁ = w
```

# Esempio di applicazione:

**Grammatica:** S → aS | bB, B → cS | ε

**ε-NFA corrispondente:**

- Q = {S, B, qf}
- δ(S, a) ∋ S, δ(S, b) ∋ B
- δ(B, c) ∋ S, δ(B, ε) ∋ qf

**Derivazione:** S ⇒ aS ⇒ abB ⇒ abcS ⇒ abcaS ⇒ abcabB ⇒ abcab

**Computazione:** S →^a S →^b B →^c S →^a S →^b B →^ε qf

# Conclusione:

L'ε-NFA A riconosce esattamente L(G). Poiché i linguaggi riconosciuti da ε-NFA sono regolari, **ogni CFG lineare a destra genera un linguaggio regolare**. ■

**Complessità:** |Q| = |V| + O(|R|), quindi la costruzione è lineare nelle dimensioni della grammatica.

# Considerazioni

# CFG Lineare a Destra ≠ Forma Normale di Chomsky

## Differenze strutturali:

**CFG Lineare a Destra:**

- Regole: A → wB o A → w
- w ∈ Σ* (stringa di terminali di lunghezza ≥ 0)
- Al più una variabile, sempre a destra

**Forma Normale di Chomsky (CNF):**

- Regole: A → BC o A → a
- Esattamente due variabili O un terminale
- Nessuna stringa di terminali multipli

## Esempio di incompatibilità:

```
CFG Lineare a Destra:
A → abcB     ✓ (valida)
A → xyz      ✓ (valida)

CNF equivalente richiederebbe:
A → AB₁
B₁ → BC₁
C₁ → CB
B → a, C → b, etc.
```

# CNF non è lineare a sinistra

**CFG Lineare a Sinistra** avrebbe regole: A → Bw (variabile a sinistra)

**CNF ha:** A → BC (due variabili, nessuna "stringa terminale")

## Gerarchia di Chomsky:

```
Regolari ⊂ Lineari ⊂ Context-Free ⊂ Context-Sensitive ⊂ Ricorsivamente Enumerabili
```

- **CFG Lineari** (dx/sx) → Linguaggi Regolari
- **CNF** → Qualsiasi Linguaggio Context-Free

## Perché l'esercizio ha senso:

L'esercizio dimostra che le **CFG lineari a destra generano linguaggi regolari**, non che siano una forma normale. La costruzione dell'ε-NFA sfrutta la struttura particolare delle regole lineari:

- A → wB: "consuma w e vai allo stato B"

- A → w: "consuma w e accetta"

Questa corrispondenza diretta con gli stati di un automa è ciò che rende i linguaggi lineari equivalenti ai regolari!

**Conclusione:** Le CFG lineari sono una **classe ristretta** che collassa sui regolari, non una forma normale per i context-free generali.

# Esercizio 5 - L non regolare - X_i diverso da X_j

**1.47** Let $\Sigma = \{1, \#\}$ and let

$$Y = \{w | \ w = x_1\#x_2\#\cdots\#x_k \text{ for } k \geq 0, \text{ each } x_i \in 1^*, \text{ and } x_i \neq x_j \text{ for } i \neq j\}.$$

Prove that $Y$ is not regular.

# Teorema: Y non è regolare

**Dimostrazione per assurdo tramite Pumping Lemma:**

Supponiamo per assurdo che Y sia regolare.

**• Sia p la lunghezza data dal Pumping Lemma**

**• Consideriamo la parola w = 1#11#111#...#1$^p$**, che appartiene a Y ed è di lunghezza maggiore di p

**Verifica che w ∈ Y:**

- w ha la forma $x_1\#x_2\#...\#x_p$ dove $x_i = 1^i$ per i = 1,2,...,p
- Ogni $x_i \in 1^*$ ✓
- $x_i \neq x_j$ per $i \neq j$ (poiché $1^i \neq 1^j$ quando $i \neq j$) ✓
- Quindi w ∈ Y

**Calcolo lunghezza:** $|w| = (1) + 1 + (2) + 1 + (3) + 1 + ... + (p) + (p-1) = \Sigma_{i=1}^{p} i + (p-1) = p(p+1)/2 + p-1 > p$

**• Sia w = xyz una suddivisione di w tale che y ≠ ε e |xy| ≤ p**

**Analisi delle possibili decomposizioni:**

Poiché |xy| ≤ p e w inizia con "1#11#111#...", la parte xy è contenuta nei primi caratteri di w.

**Caso 1:** y è contenuto nel primo segmento "1"

- Dato che |y| > 0 e il primo segmento ha lunghezza 1, necessariamente y = "1"
- Quindi: x = ε, y = "1", z = "#11#111#...#1$^p$"

- **Test con i = 0:** $xy^0z = \varepsilon z = "\#11\#111\#...\#1^p"$
- Questa stringa **inizia con #**, quindi non ha la forma $x_1\#x_2\#...\#x_k$ richiesta da Y
- **Conclusione:** $xy^0z \notin Y$ ✗

**Caso 2:** y attraversa il confine tra primo e secondo segmento

- Se y = "#", allora: x = "1", z = "11#111#...#1^p"
- **Test con i = 0:** $xy^0z$ = "1" + "11#111#...#1^p" = "111#111#...#1^p"
- I segmenti sono: "111", "111", "1⁴", ..., "1^p"
- **I primi due segmenti sono identici** (entrambi "111"), violando $x_i \neq x_j$
- **Conclusione:** $xy^0z \notin Y$ ✗

**Caso 3:** y = "1#"

- Allora: x = ε, z = "11#111#...#1^p"
- **Test con i = 2:** $xy^2z$ = "1#1#11#111#...#1^p"
- I segmenti sono: "1", "1", "11", "111", ..., "1^p"
- **I primi due segmenti sono identici** (entrambi "1"), violando $x_i \neq x_j$
- **Conclusione:** $xy^2z \notin Y$ ✗

**Generalizzazione:** In tutti i casi possibili, esiste un esponente $i \geq 0$ tale che $xy^iz$ produce una stringa che:

- O non rispetta la forma sintattica $x_1\#x_2\#...\#x_k$
- O contiene segmenti duplicati $x_i = x_j$ con $i \neq j$

Questo **contraddice il Pumping Lemma**, quindi l'ipotesi iniziale è falsa.

**Abbiamo trovato un assurdo, quindi Y non può essere regolare. ■**

**Intuizione:** Y richiede di "ricordare" tutti i segmenti precedenti per verificare che siano diversi dal segmento corrente. Questa proprietà di "memoria illimitata" non può essere implementata con un numero finito di stati.

# Alternativa 2

# Teorema: L non è regolare

**Dimostrazione per assurdo tramite Pumping Lemma:**

Supponiamo per assurdo che L sia regolare.

• **Sia p la lunghezza data dal Pumping Lemma**

• **Consideriamo la parola w = 0^(p·(p+1)) 1^(p+1)**, che appartiene a L ed è di lunghezza maggiore di p

**Verifica che w ∈ L:**

- m = p·(p+1), n = p+1
- m/n = p·(p+1)/(p+1) = p ∈ $\mathbb{Z}$ ✓
- Quindi w ∈ L

**Verifica lunghezza:** |w| = p·(p+1) + (p+1) = (p+1)² > p ✓

**• Sia w = xyz una suddivisione di w tale che y ≠ ε e |xy| ≤ p**

**Analisi della decomposizione:**

Poiché |xy| ≤ p e w inizia con p·(p+1) zeri (dove p·(p+1) >> p), la parte xy è **completamente contenuta nella sequenza di zeri**.

Quindi:

- x = 0^a per qualche a ≥ 0
- y = 0^b per qualche b ≥ 1 (dato che y ≠ ε)
- z = 0^c 1^(p+1) per qualche c ≥ 0

dove a + b + c = p·(p+1) e a + b ≤ p.

**Test con esponente i = 0:**

Consideriamo $xy^0z$ = xz = 0^(a+c) 1^(p+1) = 0^(p·(p+1)-b) 1^(p+1)

Per questa stringa essere in L, il rapporto (p·(p+1) - b)/(p+1) deve essere un intero.

**Calcolo:** (p·(p+1) - b)/(p+1) = p·(p+1)/(p+1) - b/(p+1) = p - b/(p+1)

**Condizione di appartenenza:** Per p - b/(p+1) ∈ $\mathbb{Z}$, deve valere b/(p+1) ∈ $\mathbb{Z}$, cioè (p+1) deve dividere b.

**Analisi del vincolo:** Dalle condizioni del Pumping Lemma:

- 1 ≤ b (dato che y ≠ ε e y contiene solo zeri)
- a + b ≤ p, quindi b ≤ p (dato che a ≥ 0)

**Contraddizione:** Abbiamo 1 ≤ b ≤ p < p+1.

Poiché b < p+1 e b ≥ 1, il numero (p+1) **non può dividere** b.

Quindi b/(p+1) ∉ $\mathbb{Z}$, il che implica p - b/(p+1) ∉ $\mathbb{Z}$.

**Conclusione:** $xy^0z$ ∉ L, contraddicendo il Pumping Lemma.

**Abbiamo trovato un assurdo, quindi L non può essere regolare. ■**

**Intuizione:** L richiede di verificare se m è un multiplo esatto di n, una relazione aritmetica che non può essere verificata con memoria finita per rapporti arbitrari.

# Bonus: Roba brutta

*1.57 If $A$ is any language, let $A_{\frac{1}{2}\_}$ be the set of all first halves of strings in $A$ so that

$$A_{\frac{1}{2}\_} = \{x \mid \text{ for some } y,\ |x| = |y| \text{ and } xy \in A\}.$$

Show that if $A$ is regular, then so is $A_{\frac{1}{2}\_}$.

*1.58 If $A$ is any language, let $A_{\frac{1}{3}\_\frac{1}{3}}$ be the set of all strings in $A$ with their middle thirds removed so that

$$A_{\frac{1}{3}\_\frac{1}{3}} = \{xz \mid \text{ for some } y,\ |x| = |y| = |z| \text{ and } xyz \in A\}.$$

Show that if $A$ is regular, then $A_{\frac{1}{3}\_\frac{1}{3}}$ is not necessarily regular.

## Esercizio 1.57: $A_{1/2}$ è regolare se A è regolare

**Teorema:** Se A è regolare, allora $A_{1/2}$ è regolare.

**Dimostrazione costruttiva:**

Sia M = (Q, Σ, δ, $q_0$, F) un DFA che riconosce A.

Costruisco **N = (Q', Σ, δ', $q_0'$, F')** NFA che riconosce $A_{1/2}$:

**Stati:** Q' = Q × Q × {0, 1}

- **Primo componente:** stato corrente nella simulazione di M
- **Secondo componente:** stato "memorizzato" dove inizia la seconda metà
- **Terzo componente:** fase (0 = prima metà, 1 = seconda metà)

**Stato iniziale:** $q_0'$ = ($q_0$, $q_0$, 0)

**Funzione di transizione:**

- δ'((q, r, 0), a) = {(δ(q,a), r, 0), (δ(q,a), δ(q,a), 1)}
- δ'((q, r, 1), a) = {(δ(q,a), r, 1)}

**Stati finali:** F' = {(q, r, 1) | q ∈ F}

**Algoritmo:**

1. **Fase 0:** Simula M normalmente, a ogni passo può "indovinare" che inizia la seconda metà
2. **Transizione critica:** (q, r, 0) $\rightarrow^{a}$ (δ(q,a), δ(q,a), 1) memorizza lo stato corrente

3. **Fase 1:** Continua simulazione, mantenendo memoria dello stato di inizio seconda metà
4. **Accettazione:** Se la simulazione completa (lunghezza 2n) termina in stato finale

**Correttezza:** N accetta x iff ∃y con |x| = |y| e xy ∈ A. ■

---

# Esercizio 1.58: $A_{1/3-1/3}$ non è necessariamente regolare

**Teorema:** Esiste A regolare tale che $A_{1/3-1/3}$ non è regolare.

**Controesepio devastante:**

Sia $_A = \{0,1,2\} \setminus \{0^n1^n2^n \mid n \geq 1\}_*$

**A è regolare:** È il complemento di un linguaggio context-free specifico, riconoscibile da DFA che rifiuta il pattern esatto $0^n1^n2^n$.

**Calcolo di $A_{1/3-1/3}$:**

Per $xz \in A_{1/3-1/3}$, deve esistere y con $|x| = |y| = |z| = n$ e $xyz \in A$.

**Claim cruciale:** $\{0^n2^n \mid n \geq 1\} \subseteq A_{1/3-1/3}$

**Dimostrazione del claim:** Fisso $n \geq 1$ e considero $x = 0^n$, $z = 2^n$.

Devo trovare y con $|y| = n$ tale che $xyz = 0^ny2^n \in A$.

**Condizione di appartenenza:** $0^ny2^n \in A \iff 0^ny2^n \notin \{0^k1^k2^k \mid k \geq 1\}$

**Condizione equivalente:** $y \neq 1^n$

**Costruzione esplicita:** Scelgo $y = 0^n$ (qualsiasi $y \neq 1^n$ funziona).

Allora $xyz = 0^n0^n2^n = 0^{2n}2^n \notin \{0^k1^k2^k \mid k \geq 1\}$ ✓

Quindi $xyz \in A$, e per definizione $(0^n, 2^n) \in A_{1/3-1/3}$.

**Conclusione del claim:** $\forall n \geq 1, 0^n2^n \in A_{1/3-1/3}$

**$\{0^n2^n \mid n \geq 1\}$ non è regolare:**

*Pumping Lemma applicato:*

- Sia p la lunghezza di pumping
- Considero $w = 0^p2^p \in \{0^n2^n \mid n \geq 1\}$
- Per qualsiasi decomposizione $w = xyz$ con $|xy| \leq p$, $y \neq \varepsilon$:
  - y contiene solo zeri (poiché $|xy| \leq p < 2p$)
  - xy^i z avrà più zeri che 2, violando il pattern $0^n2^n$ per $i \neq 1$

**Conclusione finale:**

$A_{1/3^{-1}/3} \supseteq \{0^n 2^n \mid n \geq 1\} \implies A_{1/3^{-1}/3}$ non regolare

Quindi esiste A regolare (il nostro A) tale che $A_{1/3^{-1}/3}$ non è regolare. ■

**Morale:** Mentre $A_{1/2}$ preserva la regolarità (proprietà di chiusura), $A_{1/3^{-1}/3}$ può distruggerla, rivelando dipendenze aritmetiche nascoste che richiedono memoria infinita.

# Esercizi Esotici

---

# Esercizio 1: Turing Machine con Testina Calda

## Consegna:

Una **Macchina di Turing con Testina Calda (Hot-Head TM)** è così impaziente di scrivere che non può mai lasciare una cella senza modificarla. Ad ogni transizione, la testina **deve obbligatoriamente** scrivere un simbolo diverso da quello che sta leggendo.

Formalmente: $\delta(q, a) = (r, b, dir)$ implica **$b \neq a$** sempre.

Dimostra che le Hot-Head TM riconoscono esattamente i linguaggi Turing-riconoscibili.

## Soluzione:

**Teorema:** Hot-Head TM ≡ TM Standard

**⇐ Direzione (Standard → Hot-Head):** Data TM standard M, costruisco Hot-Head H con alfabeto esteso $\Gamma\_H = \Gamma \cup \{a' \mid a \in \Gamma\}$.

**H = "Su input w:**

1. Per ogni mossa $\delta\_M(q, a) = (r, b, dir)$:
   - Se $b \neq a$: H fa $\delta\_H(q, a) = (r, b, dir)$
   - Se $b = a$: H fa $\delta\_H(q, a) = (r\_temp, a', dir)$ poi $\delta\_H(r\_temp, a') = (r, a, S)$
2. **Invariante:** a' significa "simbolo a temporaneamente modificato"

**⇒ Direzione (Hot-Head → Standard):** Data Hot-Head H, costruisco Standard S che simula H tracciando lo stato "prima della scrittura forzata".

**S = "Su input w:**

1. **Memoria aggiuntiva:** Stato di S = (stato_H, simbolo_originale)
2. **Simulazione:** Quando H scrive $b \neq a$, S deduce cosa H "voleva davvero fare"
3. **Decodifica:** S ricostruisce il comportamento logico ignorando scritture "forzate""

**Correttezza:** La scrittura forzata è solo "rumore termico" che non influenza la logica computazionale. ■

---

# Esercizio 2: Turing Machine Bells & Whistles

## Consegna:

Una **Bells & Whistles Turing Machine** è dotata di dispositivi audio decorativi. Oltre alle normali operazioni, può:

- 🔔 **BELL**: Suonare una campanella
- 🎵 **WHISTLE**: Fischiare una nota
- 🎺 **FANFARE**: Suonare una fanfara

La funzione di transizione ha forma: δ(q, a) = (r, b, dir, audio) dove audio ∈ {ø, BELL, WHISTLE, FANFARE}.

Il nastro di input è preceduto da una "partitura" che codifica la sequenza di suoni attesi. La macchina accetta solo se la computazione produce **esattamente** la sequenza musicale codificata.

## Definizione formale:

Input: ♪$s_1 s_2 ... s_k$♪w dove $s_i \in \{B,W,F\}$ è la partitura e w è l'input vero.

## Soluzione:

**Teorema:** Bells & Whistles TM ≡ TM Standard

**Intuizione:** La "musica" è solo un controllo di checksum sofisticato!

**⇐ Direzione (Standard → B&W):** Data TM standard M che riconosce L, costruisco B&W che riconosce: L' = {♪$s_1...s_k$♪w | w ∈ L e $s_1...s_k$ è una sequenza qualsiasi}

**Algoritmo triviale:** Ignora la partitura, simula M su w, suona suoni a caso.

**⇒ Direzione (B&W → Standard):** Data B&W M, il linguaggio accettato ha forma: L = {♪$s_1...s_k$♪w | M su w produce suoni $s_1...s_k$ e accetta}

**Costruzione Standard S: S = "Su input ♪$s_1...s_k$♪w:**

1. **Estrazione partitura:** Legge e memorizza $s_1...s_k$
2. **Simulazione silenziosa:** Simula M su w ignorando i suoni
3. **Tracking audio:** Per ogni transizione, registra l'audio prodotto
4. **Verifica finale:** Accetta iff M accetta E la sequenza audio coincide"

**Problema nascosto:** La "partitura" vincola il comportamento di M!

**Osservazione profonda:** Se M deve produrre una specifica sequenza audio su w, allora M deve già "conoscere" w in anticipo. Questo trasforma il riconoscimento in un problema di **predizione deterministica**.

**Conclusione:** B&W TM possono riconoscere linguaggi sofisticati dove l'input codifica sia i dati che il "comportamento atteso" della computazione. Rimangono comunque Turing-equivalenti. ■

---

# Esercizio 3: Turing Machine Telepatica

## Consegna:

Una **Turing Machine Telepatica (Psychic TM)** ha il potere soprannaturale di "sentire" il contenuto di celle lontane senza muovere la testina.

Può eseguire l'operazione speciale **PEEK(offset)** che legge il simbolo nella cella corrente + offset senza spostarsi.

La funzione di transizione diventa: δ(q, a, peek_data) = (r, b, dir, peek_action) dove:

- peek_data: simbolo letto dall'ultima PEEK (⊥ se mai usata)
- peek_action ∈ {∅, PEEK(k)} per k ∈ $\mathbb{Z}$

**Esercizio:** Dimostra che le Psychic TM sono equivalenti alle TM standard, ma attenzione alle implicazioni filosofiche!

## Soluzione:

**Teorema:** Psychic TM ≡ TM Standard (ma solleva questioni metafisiche!)

⇐ **Direzione (Standard → Psychic):** Banale: TM standard è Psychic che non usa mai PEEK. ✓

⇒ **Direzione (Psychic → Standard): Problema:** Come simulare "percezioni extrasensoriali" con movimento fisico?

**Costruzione Standard S: S = "Su input w:**

1. **Quando P fa PEEK(k):**
   - S salva posizione corrente pos
   - S si sposta a pos + k (aggiungendo celle se necessario)
   - S legge il simbolo target
   - S torna a pos

- S memorizza il simbolo in uno stato "memoria PEEK"

2. **Overhead spaziale:** S deve gestire espansione dinamica del nastro
3. **Overhead temporale:** Ogni PEEK costa O(|k|) mosse"

**Analisi filosofica profonda:**

**Implicazione 1:** La "telepatia" è solo **movimento ottimizzato**. Ogni informazione accessibile telepaticamente è accessibile fisicamente.

**Implicazione 2:** La **località** non limita la computazione, ma influenza l'**efficienza**.

**Paradosso temporale:** Se P fa PEEK(-1000) su cella mai visitata, S deve "creare" contenuto di celle future per poter rispondere. Questo suggerisce che la "telepatia" implica **predeterminazione** del nastro.

**Teorema Meta-Fisico:** Ogni TM telepatica equivale a una TM standard che "simula" l'universo completo in cui opera.

**Conclusione:** La telepatia computazionale è un'illusione: ogni "conoscenza a distanza" deriva da computazione locale nascosta. Il potere soprannaturale si riduce a bookkeeping! ■

---

**Morale Generale:** Tutte le varianti "esotiche" di TM collassano alla computabilità standard, confermando la robustezza universale del modello di Turing. L'universo computazionale è democratico: non esistono "superpoteri" algoritmici! 🎭