Esercizio 1 - Dimostra L regolare - Traslitterazione

Dato un linguaggio regolare $L \subseteq \Sigma^*$ e una translitterazione $T : \Sigma \to \Gamma$, dimostro che $T(L)$ è regolare costruendo un DFA che simula il riconoscimento di $T(L)$.

Costruzione del DFA simulatore:

Sia $M = (Q, \Sigma, \delta, q_0, F)$ il DFA che riconosce L. Costruisco $M' = (Q, \Gamma, \delta', q_0, F)$ dove:

- **Stati:** $Q' = Q$ (stessi stati dell'automa originale)
- **Alfabeto:** Γ (alfabeto target della translitterazione)
- **Funzione di transizione:** $\delta'(q, T(a)) = \delta(q, a)$ per ogni $q \in Q$, $a \in \Sigma$
- **Stato iniziale:** $q_0' = q_0$
- **Stati finali:** $F' = F$

Meccanismo di simulazione:

L'automa M' simula M nel modo seguente: quando M' legge un simbolo $c \in \Gamma$ nell'input, trova l'unico simbolo $a \in \Sigma$ tale che $T(a) = c$ (possibile perché T è biettiva), poi esegue la transizione che M avrebbe eseguito leggendo a.

Correttezza della simulazione:

Per ogni stringa $w = a_1a_2...a_n \in L$, l'automa M ha una computazione accettante: $q_0 \to^{a_1} q_1 \to^{a_2} q_2 \to^{a_n} q_n \in F$

L'automa M' simula questa computazione su $T(w) = T(a_1)T(a_2)...T(a_n)$: $q_0 \to^{T(a_1)} q_1 \to^{T(a_2)} q_2 \to^{T(a_n)} q_n \in F$

dove ogni transizione $\delta'(q_{i-1}, T(a_i)) = q_i$ corrisponde esattamente a $\delta(q_{i-1}, a_i) = q_i$.

Biettività della corrispondenza:

Poiché T è una biiezione, esiste una corrispondenza uno-a-uno tra le stringhe in Σ *e quelle in* Γ. Quindi $L(M') = \{T(w) \mid w \in L(M)\} = T(L)$.

La simulazione preserva la regolarità perché mantiene la struttura finita degli stati mentre trasforma deterministicamente i simboli dell'alfabeto.

Risoluzione della contraddizione:

Il testo presenta un errore concettuale. La tabella mostra caratteri **hiragana** (あ, い, う...) e **katakana** (ア, イ, ウ...) con la loro traslitterazione in alfabeto latino secondo il sistema Hepburn. La direzione corretta è quindi: caratteri giapponesi → alfabeto latino, non il contrario come erroneamente indicato nel testo.

Tuttavia, il principio matematico rimane valido poiché la translitterazione Hepburn è effettivamente una funzione biettiva tra l'alfabeto kana e un sottoinsieme dell'alfabeto latino.

Dimostrazione formale del teorema (se e solo se):

Teorema: Sia $T : \Sigma \to \Gamma$ una translitterazione (funzione biettiva). Un linguaggio $L \subseteq \Sigma$ *è regolare se e solo se* $T(L) \subseteq \Gamma$ è regolare.

Dimostrazione:

(⇒) Se L è regolare, allora T(L) è regolare:

Sia $M = (Q, \Sigma, \delta, q_0, F)$ il DFA che riconosce L. Costruisco $M' = (Q, \Gamma, \delta', q_0, F)$ dove $\delta'(q, T(a)) = \delta(q, a)$.

Per ogni $w \in L$, la computazione di M su w corrisponde biunivocamente alla computazione di M' su T(w), quindi L(M') = T(L).

(⇐) Se T(L) è regolare, allora L è regolare:

Sia $M' = (Q', \Gamma, \delta', q_0', F')$ il DFA che riconosce T(L). Poiché T è biettiva, esiste $T^{-1} : \Gamma \to \Sigma$.

Costruisco $M = (Q', \Sigma, \delta, q_0', F')$ dove $\delta(q, a) = \delta'(q, T(a))$.

Per ogni $w \in L$, abbiamo $T(w) \in T(L)$, quindi M' accetta T(w). La computazione di M su w simula quella di M' su T(w) applicando T^{-1}, quindi M accetta w.

Reciprocamente, se M accetta w, allora M' accetta T(w), quindi $T(w) \in T(L)$, quindi $w \in L$.

Pertanto L(M) = L.

Conclusione: La biettività di T garantisce che la regolarità sia preservata in entrambe le direzioni, stabilendo l'equivalenza completa tra regolarità di L e regolarità di T(L).

Esercizio 2 - Dimostra L context-free SCRAMBLE

Dimostrazione usando solo i metodi del corso

Teorema: Se $B \subseteq \{0,1\}^*$ è regolare, allora SCRAMBLE(B) è context-free.

Strategia: Costruisco un automa a pila (PDA) che riconosce SCRAMBLE(B), dimostrando così che è context-free.

Costruzione del PDA:

Sia $M = (Q, \{0,1\}, \delta, q_0, F)$ il DFA che riconosce B.

Costruisco $P = (Q', \{0,1\}, \Gamma, \delta', q_0', Z_0, F')$ dove:

- $Q' = \{q_{init}\} \cup Q \cup \{q_s^{can}\}$

- $\Gamma = \{0, 1, Z_0\}$ (simboli dello stack)
- Z_0 è il simbolo iniziale dello stack

Algoritmo del PDA in tre fasi:

Fase 1: Lettura e memorizzazione

- Stato q_{init}: Per ogni simbolo a ∈ {0,1} letto dall'input, pusha a nello stack
- Transizione: $\delta'(q_{init}, a, Z) = (q_{init}, aZ)$ per ogni a ∈ {0,1}, Z ∈ Γ
- Transizione non-deterministica: $\delta'(q_{init}, \varepsilon, Z) = (q_0, Z)$ per passare alla fase 2

Fase 2: Simulazione del DFA

- Stati Q: Simula M leggendo i simboli dallo stack invece che dall'input
- Per ogni transizione δ(q, a) = q' in M:
  - $\delta'(q, \varepsilon, a) = (q', \varepsilon)$ (legge a dallo stack, nessun input)
- Transizione: $\delta'(q, \varepsilon, Z_0) = (q_s^{can}, Z_0)$ se q ∈ F

Fase 3: Verifica stack vuoto

- Stato q_s^{can}: Accetta se lo stack contiene solo Z_0
- $F' = \{q_s^{can}\}$

Correttezza del PDA:

Lemma: P accetta w se e solo se w ∈ SCRAMBLE(B).

Dimostrazione:

(⇒) Se P accetta w, allora:

1. Fase 1: w è stata scritta nello stack in ordine inverso
2. Fase 2: M ha accettato la sequenza di simboli nello stack
3. Quindi esiste una stringa v (quella "letta" dallo stack) tale che v ∈ B e v ha gli stessi simboli di w
4. Pertanto w è una permutazione di v, quindi w ∈ SCRAMBLE(B)

(⇐) Se w ∈ SCRAMBLE(B), allora esiste v ∈ B tale che w è permutazione di v:

1. P può scegliere non-deterministicamente di passare alla fase 2 dopo aver letto w
2. Il contenuto dello stack sarà w^R (w invertita)
3. M può accettare v simulando le transizioni appropriate
4. Poiché w e v hanno gli stessi simboli, lo stack si svuoterà completamente quando M raggiunge uno stato finale

Costruzione alternativa con grammatica CNF:

Posso anche costruire direttamente una grammatica in forma normale di Chomsky:

Passo 1: Poiché B è regolare, posso costruire una grammatica lineare destra G_B che genera B.

Passo 2: Per ogni produzione A → aB in G_B, introduco regole che permettono di "redistribuire" il simbolo a:

```
S → S_perm
S_perm → Gen_A₁ | Gen_A₂ | ... (per ogni variabile A_i in G_B)
Gen_A → Distr_A T_A
Distr_A → 0 Distr_A | 1 Distr_A | ε
T_A → (regole che seguono la struttura di A in G_B ma permettono
permutazioni)
```

Passo 3: Converto in forma normale di Chomsky usando l'algoritmo standard:

1. Elimino ε-regole
2. Elimino regole unitarie
3. Trasformo le regole rimanenti nella forma A → BC o A → a

Conclusione: SCRAMBLE(B) è riconosciuto da un PDA, quindi è context-free. La costruzione usa solo tecniche standard del corso: automi a pila, simulazione di DFA, e proprietà dei linguaggi context-free.

Esercizio Libro - Permutazione

*!! **Esercizio 7.3.5** Una stringa y si dice una *permutazione* della stringa x se possiamo ottenere x riordinando i simboli di y. Per esempio le permutazioni della stringa $x = 011$ sono 110, 101 e 011. Dato un linguaggio L, $perm(L)$ è l'insieme delle permutazioni delle stringhe in L. Per esempio, se $L = \{0^n1^n \mid n \geq 0\}$, $perm(L)$ è l'insieme delle stringhe con lo stesso numero di 0 e di 1.

a) Fornite un esempio di linguaggio regolare L sull'alfabeto $\{0, 1\}$ tale che $perm(L)$ non sia regolare. Giustificate la risposta. *Suggerimento*: cercate un linguaggio regolare le cui permutazioni siano tutte le stringhe con lo stesso numero di 0 e di 1.

b) Fornite un esempio di linguaggio regolare L sull'alfabeto $\{0, 1, 2\}$ tale che $perm(L)$ non sia libero dal contesto.

c) Dimostrate che, per ogni linguaggio regolare L su un alfabeto di due simboli, $perm(L)$ è libero dal contesto.

Risoluzione formale dell'Esercizio 7.3.5

a) Linguaggio regolare L tale che perm(L) non sia regolare

Esempio: L = (01)*

Giustificazione:

- L è regolare (espressione regolare (01)*)
- perm(L) = {w ∈ {0,1}* | $\#_0$(w) = $\#_1$(w)} (stringhe con ugual numero di 0 e 1)
- perm(L) ⊇ {0^n1^n | n ≥ 0}
- Poiché {0^n1^n | n ≥ 0} non è regolare, perm(L) non è regolare

Dimostrazione che perm(L) non è regolare: Applico il pumping lemma. Assumo perm(L) regolare con costante p. Considero w = 0^p1^p ∈ perm(L) con |w| = 2p ≥ p. Per il pumping lemma, w = xyz con |xy| ≤ p, |y| > 0. Quindi y = 0^k per qualche k > 0. Ma allora $xy^2z = 0^{p+k}1^p$ ∉ perm(L), contraddizione.

b) Linguaggio regolare L su {0,1,2} tale che perm(L) non sia context-free

Esempio: L = (012)*

Giustificazione:

- L è regolare (espressione regolare (012)*)
- perm(L) = {w ∈ {0,1,2}* | $\#_0$(w) = $\#_1$(w) = $\#_2$(w)}
- perm(L) ⊇ {$0^n1^n2^n$ | n ≥ 0}
- Poiché {$0^n1^n2^n$ | n ≥ 0} non è context-free, perm(L) non è context-free

Dimostrazione che {$0^n1^n2^n$ | n ≥ 0} non è context-free: Applico il pumping lemma per linguaggi context-free. Assumo il linguaggio context-free con costante p. Considero w = $0^p1^p2^p$. Per il pumping lemma, w = uvxyz con |vxy| ≤ p, |vy| > 0. Poiché |vxy| ≤ p, vxy può contenere al massimo due tipi di simboli distinti. Quindi uv^2xy^2z non può avere ugual numero di 0, 1 e 2, contraddizione.

c) Dimostrazione: perm(L) è context-free per ogni linguaggio regolare L su alfabeto binario

Teorema: Per ogni linguaggio regolare L ⊆ {0,1}*, perm(L) è context-free.

Dimostrazione costruttiva:

Passo 1: Caratterizzazione Su alfabeto binario, due stringhe sono permutazioni se e solo se hanno stesso numero di 0 e stesso numero di 1. Quindi: perm(L) = {w ∈ {0,1}* | ∃v ∈ L : $\#_0$(w) = $\#_0$(v) ∧ $\#_1$(w) = $\#_1$(v)}

Passo 2: Insieme dei conteggi Sia C = {($\#_0$(v), $\#_1$(v)) | v ∈ L}. Poiché L è regolare, C è un insieme finito (un DFA ha solo finiti stati e cicli limitati).

Passo 3: Costruzione grammatica context-free Per ogni (i,j) ∈ C, costruisco una grammatica G_{ij} in forma normale di Chomsky che genera tutte le stringhe con esattamente i zeri e j uni:

```
S_{i,j} → A_{i,j}          (se i+j > 0)
S_{i,j} → ε                (se i = j = 0)

A_{i,j} → ZA_{i-1,j}       (se i > 0)
A_{i,j} → UA_{i,j-1}       (se j > 0)
Z → 0
U → 1
A_{0,0} → ε
```

Passo 4: Unione delle grammatiche La grammatica finale ha regola iniziale: $S \to S_{(i_1,j_1)} \mid S_{(i_2,j_2)} \mid \ldots \mid S_{(i_k,j_k)}$

dove $\{(i_1,j_1), \ldots, (i_k,j_k)\} = C$.

Passo 5: Conversione in CNF Applico l'algoritmo standard per convertire in forma normale di Chomsky:

1. Elimino ε-regole
2. Elimino regole unitarie
3. Sostituisco regole lunghe

Correttezza: La grammatica risultante genera esattamente perm(L), quindi perm(L) è context-free.

Esercizio Libro - Shuffle

Esercizio 7.3.4 Definiamo *shuffle* di due stringhe w e x l'insieme di tutte le stringhe ottenute intercalando arbitrariamente le posizioni di w e x. Più esattamente, *shuffle*(w, x) è l'insieme delle stringhe z tali che:

1. ogni posizione di z si possa attribuire a w o a x, ma non a entrambi
2. le posizioni di z attribuite a w formano w se lette da sinistra a destra
3. le posizioni di z attribuite a x formano x se lette da sinistra a destra.

Per esempio, se $w = 01$ e $x = 110$, *shuffle*$(01, 110)$ è l'insieme di stringhe $\{01110, 01101, 10110, 10101, 11010, 11001\}$. Per illustrare il ragionamento: la terza stringa, 10110, si ottiene attribuendo la seconda e la quinta posizione a 01 e le altre a 110; la prima stringa, 01110, si può ottenere in tre modi attribuendo la prima posizione e una fra la seconda, la terza e la quarta a 01, le altre tre a 110. Possiamo estendere l'operazione ai linguaggi definendo *shuffle*(L_1, L_2) come l'unione su tutte le coppie di stringhe, w in L_1 e x in L_2, di *shuffle*(w, x).

a) Che cos'è *shuffle*$(00, 111)$?

* b) Che cos'è *shuffle*(L_1, L_2) se $L_1 = L(\mathbf{0}^*)$ e $L_2 = \{0^n 1^n \mid n \geq 0\}$?

*! c) Dimostrate che se L_1 ed L_2 sono linguaggi regolari, lo è anche

shuffle(L_1, L_2)

Suggerimento: partite da un DFA per L_1 e da un altro per L_2.

! d) Dimostrate che, se L è un CFL ed R un linguaggio regolare, *shuffle*(L, R) è un CFL. *Suggerimento*: partite da un PDA per L e da un DFA per R.

Risoluzione formale dell'Esercizio 7.3.4

a) Calcolo di shuffle(01, 111)

Per definizione, shuffle(01, 111) contiene tutte le stringhe z di lunghezza |01| + |111| = 5 ottenute scegliendo 2 posizioni per i caratteri di 01 (mantenendo l'ordine) e le rimanenti 3 per 111.

Enumerazione sistematica: Scelgo 2 posizioni su 5 per w = 01: C(5,2) = 10 modi possibili.

- Posizioni (1,2): z = 01111
- Posizioni (1,3): z = 01111
- Posizioni (1,4): z = 01111
- Posizioni (1,5): z = 01111

- Posizioni (2,3): z = 10111
- Posizioni (2,4): z = 10111
- Posizioni (2,5): z = 10111
- Posizioni (3,4): z = 11011
- Posizioni (3,5): z = 11011
- Posizioni (4,5): z = 11101

Risultato: shuffle(01, 111) = {01111, 10111, 11011, 11101}

*_b) Calcolo di shuffle(L_1, L_2) con L_1 = L(0) e L_2 = {0^n1^n | n ≥ 0}_*

L_1 = {ε, 0, 00, 000, ...} L_2 = {ε, 01, 0011, 000111, ...}

shuffle(L_1, L_2) = ∪{shuffle(w_1, w_2) | w_1 ∈ L_1, w_2 ∈ L_2}

Caratterizzazione: Una stringa z ∈ shuffle(L_1, L_2) se e solo se:

1. z può essere scomposta in $z = z_1z_2...z_k$ dove ogni z_i ∈ {0,1}
2. Esiste una partizione delle posizioni tale che le posizioni assegnate a L_1 contengono solo 0
3. Le posizioni assegnate a L_2 formano una stringa 0^n1^n

Risultato: shuffle(L_1, L_2) = {w ∈ {0,1}* | $\#_1$(w) ≤ $\#_0$(w) ∧ ogni sequenza di 1 consecutivi ha lunghezza ≤ al numero di 0 che la precedono}

c) Dimostrazione: se L_1 e L_2 sono regolari, allora shuffle(L_1, L_2) è regolare

Costruzione con DFA:

Siano M_1 = (Q_1, Σ, δ_1, q_1, F_1) e M_2 = (Q_2, Σ, δ_2, q_2, F_2) i DFA per L_1 e L_2.

Costruisco M = (Q, Σ, δ, q_0, F) dove:

- $Q = Q_1 \times Q_2 \times \{1,2\}$
- $q_0 = (q_1, q_2, 1)$ (stato iniziale con flag per scegliere quale DFA seguire)
- $F = F_1 \times F_2 \times \{1,2\}$

Funzione di transizione: δ((q, r, i), a) = { (δ_1(q, a), r, 1) se i = 1 (leggi con M_1) (q, δ_2(r, a), 2) se i = 2 (leggi con M_2) }

Versione non-deterministica: Aggiungo transizioni ε per passare da modalità 1 a modalità 2 e viceversa, permettendo di scegliere liberamente quale DFA usare per ogni simbolo.

Correttezza: M accetta w se e solo se w può essere decomposta come interleaving di una stringa in L_1 e una in L_2.

d) Dimostrazione: se L è context-free e R è regolare, allora shuffle(L, R) è context-free

Costruzione con PDA:

Sia P_L = (Q_L, Σ, Γ, δ_L, q_L, Z_0, F_L) il PDA per L. Sia M_R = (Q_R, Σ, δ_R, q_R, F_R) il DFA per R.

Costruisco P = (Q, Σ, Γ, δ, q_0, Z_0, F) dove:

- Q = Q_L × Q_R × {L, R}
- q_0 = (q_L, q_R, L)
- F = F_L × F_R × {L, R}

Funzione di transizione: δ((q, r, flag), a, X) = { {((q', r, L), γ) | (q', γ) ∈ δ_L(q, a, X)} se flag = L {((q, r', R), X) | r' = δ_R(r, a)} se flag = R }

Transizioni ε per cambiare modalità: δ((q, r, L), ε, X) ∋ ((q, r, R), X) δ((q, r, R), ε, X) ∋ ((q, r, L), X)

Correttezza: P simula non-deterministicamente l'interleaving di una computazione del PDA per L e del DFA per R, mantenendo lo stack per controllare la parte context-free.

Conclusione: shuffle preserva la regolarità e la context-freeness nel senso descritto.

Esercizio 3 - Mostra L context-free - Insert_#

3. **(12 punti)** Dato un linguaggio $L \subseteq \Sigma^*$, definiamo il linguaggio

$$insert\#(L) = \{x\#y \mid xy \in L\}.$$

Dimostra che la classe dei linguaggi context free è chiusa per l'operazione $insert\#$.

Risoluzione formale

Teorema: La classe dei linguaggi context-free è chiusa per l'operazione insert#.

Dimostrazione:

Sia L un linguaggio context-free e sia G = (V, Σ, R, S) una grammatica context-free che genera L.

Costruisco una grammatica G' = (V', Σ ∪ {#}, R', S') che genera insert#(L).

Costruzione di G':

Insieme delle variabili: V' = {S'} ∪ {[A, L] | A ∈ V} ∪ {[A, R] | A ∈ V}

dove:

- [A, L] genera la parte sinistra di stringhe derivabili da A

- [A, R] genera la parte destra di stringhe derivabili da A

Regole di produzione:

1. **Regola iniziale:** S' → [S, L] # [S, R]
2. **Per ogni regola A → $\alpha_1\alpha_2...\alpha_n$ ∈ R:**

   Per ogni k = 0, 1, ..., n, aggiungo la coppia di regole:

   - [A, L] → $\beta_1\beta_2...\beta_k$
   - [A, R] → $\beta_{k+1}\beta_{k+2}...\beta_n$

   dove β_i è definito come:

```
βᵢ = {
  αᵢ              se αᵢ ∈ Σ
  [αᵢ, L]         se αᵢ ∈ V e i ≤ k
  [αᵢ, R]         se αᵢ ∈ V e i > k
}
```

Correttezza:

Lemma: Per ogni A ∈ V e stringa w ∈ Σ*:

- [A, L] ⇒ *u se e solo se ∃v tale che A* ⇒ uv in G
- [A, R] ⇒ *v se e solo se ∃u tale che A* ⇒ uv in G

Dimostrazione del Lemma: Per induzione sulla lunghezza delle derivazioni.

Teorema principale: L(G') = insert#(L)

(⊆) Sia z ∈ L(G'). Allora z = x#y dove S' ⇒ *[S,L] # [S,R]* ⇒ x # y. Per il lemma, ∃u,v tali che S ⇒ *xu e S* ⇒ vy in G. Poiché le derivazioni sono consistenti, xu = vy = w per qualche w. Quindi w = xy ∈ L, e z = x#y ∈ insert#(L).

(⊇) Sia z = x#y ∈ insert#(L). Allora xy ∈ L, quindi S ⇒* xy in G. Posso ricostruire una derivazione in G' che produce x#y scegliendo appropriatamente i punti di divisione k nelle regole.

Esempio: Se L = {a^nb^n | n ≥ 0} con grammatica:

```
S → aSb | ε
```

Allora G' ha regole:

```
S' → [S,L] # [S,R]
[S,L] → ε,          [S,R] → aSb      (k=0)
[S,L] → a[S,L],     [S,R] → [S,R]b   (k=1)
[S,L] → a[S,L]b,    [S,R] → ε        (k=2)
```

Questa grammatica genera insert#(L) = {$a^i\#a^jb^{i+j}$ | i,j ≥ 0}.

Conclusione: insert#(L) è generato da una grammatica context-free, quindi è context-free.
■

Esercizio 4 - Dimostra L non regolare - Quadrato perfetto reloaded

2. (12 punti) Considera il linguaggio

$$L_2 = \{0^{3n^2+2} \mid n \geq 0\}$$

Dimostra che L_2 non è regolare.

Dimostrazione che L_2 non è regolare

Teorema: L_2 = {0^(3n²+2) | n ≥ 0} non è regolare.

Dimostrazione per contraddizione usando il Pumping Lemma:

Assumo che L_2 sia regolare. Allora esiste una costante p > 0 (costante del pumping lemma) tale che ogni stringa w ∈ L_2 con |w| ≥ p può essere decomposta come w = xyz soddisfacendo:

1. |xy| ≤ p
2. |y| > 0
3. xy^i z ∈ L_2 per ogni i ≥ 0

Scelta della stringa test: Scelgo n sufficientemente grande tale che:

- 3n² + 2 ≥ p (per applicare il pumping lemma)
- 6n + 3 > p (condizione chiave per la contraddizione)

Specificamente, scelgo n = max(⌈√(p/3)⌉, ⌈(p-3)/6⌉) + 1.

Considero w = 0^(3n²+2) ∈ L_2 con |w| = 3n² + 2 ≥ p.

Applicazione del Pumping Lemma: Per il pumping lemma, w = xyz con le proprietà sopra. Poiché w consiste solo di 0:

- x = 0^a, y = 0^b, z = 0^c

- $a + b + c = 3n^2 + 2$
- $a + b \leq p, b > 0$

Analisi di xy^2z: La stringa $xy^2z = 0^{(a+2b+c)} = 0^{(3n^2+2+b)}$ ha lunghezza $3n^2 + 2 + b$.

Per $xy^2z \in L_2$, deve esistere $m \geq 0$ tale che: $3n^2 + 2 + b = 3m^2 + 2 \implies b = 3(m^2 - n^2)$

Analisi delle lunghezze consecutive in L_2: Le lunghezze delle stringhe in L_2 sono: 2, 5, 14, 29, 50, 77, 110, ...

La differenza tra lunghezze consecutive è: $3(n+1)^2 + 2 - (3n^2 + 2) = 3((n+1)^2 - n^2) = 6n + 3$

Contraddizione: Da $b = 3(m^2 - n^2)$ e $b > 0$, abbiamo $m^2 > n^2$, quindi $m \geq n + 1$.

Se $m = n + 1$ (il caso minimo), allora: $b = 3((n+1)^2 - n^2) = 3(2n + 1) = 6n + 3$

Ma dalla condizione $|xy| \leq p$ abbiamo $b \leq p$.

Per la scelta di n, abbiamo $6n + 3 > p$, quindi: $b = 6n + 3 > p \geq b$

Questa è una contraddizione.

Se $m > n + 1$: Allora $b = 3(m^2 - n^2) > 3((n+1)^2 - n^2) = 6n + 3 > p \geq b$, ottenendo nuovamente una contraddizione.

Conclusione: Non esiste una decomposizione valida $w = xyz$ che soddisfi le condizioni del pumping lemma. Quindi L_2 non è regolare. ■

Intuizione: La crescita quadratica dell'esponente crea gaps sempre più grandi tra lunghezze consecutive, che un automa finito con memoria limitata non può "saltare" mantenendo la proprietà di pumping.

OPPURE

Dimostrazione che L_2 non è regolare (con valori concreti)

Teorema: $L_2 = \{0^{(3n^2+2)} \mid n \geq 0\}$ non è regolare.

Analisi preliminare: Le prime stringhe in L_2 hanno lunghezze:

- n=0: $3(0)^2 + 2 = 2 \rightarrow 0^2$
- n=1: $3(1)^2 + 2 = 5 \rightarrow 0^5$
- n=2: $3(4) + 2 = 14 \rightarrow 0^{14}$
- n=3: $3(9) + 2 = 29 \rightarrow 0^{29}$
- n=4: $3(16) + 2 = 50 \rightarrow 0^{50}$

Osservazione critica: I gaps tra lunghezze consecutive crescono:

- 5 - 2 = 3

- 14 - 5 = 9
- 29 - 14 = 15
- 50 - 29 = 21

Dimostrazione per contraddizione:

Assumo L_2 regolare con costante di pumping p.

Scelta concreta: Prendo p = 10 (un valore ragionevole per un DFA).

Scelgo la stringa $w = 0^{29} \in L_2$ (corrispondente a n=3), poiché |w| = 29 ≥ 10 = p.

Applicazione del Pumping Lemma: w = xyz con:

- |xy| ≤ 10
- |y| > 0
- xy^i z ∈ L_2 per ogni i ≥ 0

Poiché $w = 0^{29}$, abbiamo x = 0^a, y = 0^b, z = 0^c con:

- a + b + c = 29
- a + b ≤ 10
- 1 ≤ b ≤ 10

Test con xy^2z: xy^2z = 0^(29+b) deve essere in L_2.

Le lunghezze valide vicine a 29 sono:

- 29 (n=3)
- 50 (n=4)
- 77 (n=5, poiché 3(25)+2=77)

Caso b piccolo (1 ≤ b ≤ 10): 29 + b sarebbe tra 30 e 39, ma la prossima lunghezza valida è 50.

Per essere in L_2, dovremmo avere 29 + b = 50, quindi b = 21.

Ma b ≤ 10 < 21, contraddizione!

Verifica con p più grande: Anche con p = 50, scegliendo $w = 0^{50}$:

- xy^2z = 0^(50+b) con b ≤ 50
- Prossima lunghezza valida: 77
- Serve b = 27, ma il gap successivo (77 → 110) è 33 > 27

Questo pattern si ripete: i gaps crescono più velocemente della costante di pumping.

Conclusione: Nessun valore finito di p può funzionare, quindi L_2 non è regolare.

Intuizione: Un DFA ha memoria finita, ma L_2 richiede di "ricordare" perfettamente esponenti quadratici che crescono troppo rapidamente per essere gestiti da stati finiti.

Esempio libro PL - Quadrato perfetto

EXAMPLE 1.76

Here we demonstrate a nonregular unary language. Let $D = \{1^{n^2} \mid n \geq 0\}$. In other words, D contains all strings of 1s whose length is a perfect square. We use the pumping lemma to prove that D is not regular. The proof is by contradiction.

Assume to the contrary that D is regular. Let p be the pumping length given by the pumping lemma. Let s be the string 1^{p^2}. Because s is a member of D and s has length at least p, the pumping lemma guarantees that s can be split into three pieces, $s = xyz$, where for any $i \geq 0$ the string xy^iz is in D. As in the preceding examples, we show that this outcome is impossible. Doing so in this case requires a little thought about the sequence of perfect squares:

$$0, 1, 4, 9, 16, 25, 36, 49, \ldots$$

Note the growing gap between successive members of this sequence. Large members of this sequence cannot be near each other.

Now consider the two strings xyz and xy^2z. These strings differ from each other by a single repetition of y, and consequently their lengths differ by the length of y. By condition 3 of the pumping lemma, $|xy| \leq p$ and thus $|y| \leq p$. We have $|xyz| = p^2$ and so $|xy^2z| \leq p^2 + p$. But $p^2 + p < p^2 + 2p + 1 = (p+1)^2$. Moreover, condition 2 implies that y is not the empty string and so $|xy^2z| > p^2$. Therefore, the length of xy^2z lies strictly between the consecutive perfect squares p^2 and $(p+1)^2$. Hence this length cannot be a perfect square itself. So we arrive at the contradiction $xy^2z \notin D$ and conclude that D is not regular. ■

Esercizio 5 - Mostra L non regolare - Altro quadrato

1. (9 punti) Considera il linguaggio

$$L = \{1^n 0^{2^n} \mid n \leq 0\}.$$

Dimostra che L non è regolare.

SOLUZIONE PER N POSITIVO

Dimostrazione che L non è regolare

Teorema: L = {1ⁿ0^(2ⁿ) | n ≥ 0} non è regolare.

Analisi preliminare: Le prime stringhe in L sono:

- n=0: 1⁰0^(2⁰) = 0

- n=1: $1^1 0^{(2^1)} = 100$
- n=2: $1^2 0^{(2^2)} = 110000$
- n=3: $1^3 0^{(2^3)} = 11100000000$

Dimostrazione per contraddizione usando il Pumping Lemma:

Assumo che L sia regolare. Allora esiste una costante $p > 0$ tale che ogni stringa $w \in L$ con $|w| \geq p$ può essere decomposta come $w = xyz$ soddisfacendo:

1. $|xy| \leq p$
2. $|y| > 0$
3. $xy^i z \in L$ per ogni $i \geq 0$

Scelta della stringa test: Considero $w = 1^p 0^{(2^p)} \in L$. Chiaramente $|w| = p + 2^p \geq p$.

Applicazione del Pumping Lemma: Per il pumping lemma, $w = xyz$ con le proprietà sopra.

Poiché $|xy| \leq p$ e w inizia con esattamente p copie di 1 seguite da 2^p copie di 0, la sottostringa xy deve essere contenuta nella parte dei 1. Quindi:

- $x = 1^a$ per qualche $a \geq 0$
- $y = 1^b$ per qualche $b > 0$ (da $|y| > 0$)
- $z = 1^c 0^{(2^p)}$ per qualche $c \geq 0$

dove $a + b + c = p$.

Test con xy^2z: Considero $xy^2z = x(y^2)z = 1^a 1^{(2b)} 1^c 0^{(2^p)} = 1^{(a+2b+c)} 0^{(2^p)} = 1^{(p+b)} 0^{(2^p)}$.

Condizione di appartenenza: Perché $xy^2z \in L$, deve esistere qualche $m \geq 0$ tale che: $xy^2z = 1^m 0^{(2^m)}$

Quindi:

- $1^{(p+b)} = 1^m \implies p + b = m$
- $0^{(2^p)} = 0^{(2^m)} \implies 2^p = 2^m \implies p = m$

Contraddizione: Dalle due equazioni:

- $p + b = m$
- $p = m$

Sottraendo: $b = 0$

Ma questo contraddice la condizione $|y| > 0$, che implica $b > 0$.

Conclusione: Non può esistere una decomposizione valida di w che soddisfi le condizioni del pumping lemma. Quindi L non è regolare. ■

Intuizione: La crescita esponenziale del numero di 0 rispetto al numero di 1 rende impossibile per un automa finito mantenere la relazione precisa 2^n, poiché richiederebbe memoria esponenziale.

SOLUZIONE VERA PER N NEGATIVO

Correzione: L = {$1^n0^{(2^n)}$ | n ≤ 0}

Hai ragione, ho letto male la condizione. Analizziamo il linguaggio correttamente.

Analisi del dominio: Per n ≤ 0, considero i valori che hanno senso nel contesto delle stringhe:

Caso n = 0: $1^0 0^{(2^0)} = \varepsilon \cdot 0^1 = 0$

Caso n < 0: Per n < 0, dobbiamo interpretare:

- 1^n con n < 0: per convenzione, $a^k = \varepsilon$ se k ≤ 0 nel contesto delle stringhe
- 2^n con n < 0: $2^{(-1)} = 1/2$, $2^{(-2)} = 1/4$, etc.

Poiché $0^{(2^n)}$ richiede un esponente intero non negativo, e 2^n per n < 0 non è un intero, questi casi non sono definiti nel contesto dei linguaggi formali.

Interpretazione standard: Nel contesto dei linguaggi formali, tipicamente si considera solo n = 0 come valore valido per n ≤ 0.

Conclusione: L = {0}

Teorema: L è regolare.

Dimostrazione: L è un linguaggio finito contenente una sola stringa. Ogni linguaggio finito è regolare.

Costruzione esplicita del DFA: $M = (\{q_0, q_1, q_2\}, \{0, 1\}, \delta, q_0, \{q_1\})$

dove δ è definita da:

- $\delta(q_0, 0) = q_1$
- $\delta(q_0, 1) = q_2$
- $\delta(q_1, a) = q_2$ per ogni $a \in \{0,1\}$
- $\delta(q_2, a) = q_2$ per ogni $a \in \{0,1\}$

Questo DFA accetta esattamente la stringa "0" e rigetta tutto il resto.

Risposta: L è regolare perché è un linguaggio finito.

N REGOLARE NEL CASO POTENZA NEGATIVA

Esercizio 6 - Macchina di Turing - Albero binario

1. **(12 punti)** Una *macchina di Turing ad albero binario* usa un albero binario infinito come nastro, dove ogni cella nel nastro ha un figlio sinistro e un figlio destro. Ad ogni transizione, la testina si sposta dalla cella corrente al padre, al figlio sinistro oppure al figlio destro della cella corrente. Pertanto, la funzione di transizione di una tale macchina ha la forma

$$\delta : Q \times \Gamma \mapsto Q \times \Gamma \times \{P, L, R\},$$

dove P indica lo spostamento verso il padre, L verso il figlio sinistro e R verso il figlio destro. La stringa di input viene fornita lungo il ramo sinistro dell'albero.

Mostra che qualsiasi macchina di Turing ad albero binario può essere simulata da una macchina di Turing standard.

Dimostrazione formale della simulazione

Teorema: Ogni macchina di Turing ad albero binario può essere simulata da una macchina di Turing standard.

Strategia di simulazione:

Passo 1: Codifica dell'albero binario

Represento ogni nodo dell'albero mediante una stringa binaria che indica il percorso dalla radice:

- Radice: ε (stringa vuota)
- Figlio sinistro del nodo w: w0
- Figlio destro del nodo w: w1

Passo 2: Struttura del nastro della TM simulatrice

Il nastro della TM standard è diviso in blocchi, ciascuno rappresentante una cella dell'albero:

```
|#|addr₁|cont₁|#|addr₂|cont₂|#|...|#|addrₙ|contₙ|#|pos|#|
```

dove:

- $addr_i$: indirizzo binario del nodo i
- $cont_i$: contenuto della cella i
- pos: posizione corrente della testina (indirizzo binario)
- #: separatore

Passo 3: Algoritmo di simulazione

Per simulare una transizione δ(q, a) = (q', a', dir) della macchina ad albero:

Sottoroutine TROVA_CELLA(addr):

1. Scansiona il nastro da sinistra
2. Per ogni blocco, confronta l'indirizzo con addr

3. Se trovato, ritorna la posizione; altrimenti crea nuovo blocco

Sottoroutine AGGIORNA_POSIZIONE(pos, dir):

- Se dir = P: rimuovi l'ultimo bit da pos (vai al padre)
- Se dir = L: appendi '0' a pos (vai al figlio sinistro)
- Se dir = R: appendi '1' a pos (vai al figlio destro)

Algoritmo principale:

```
1. Leggi pos dal nastro
2. TROVA_CELLA(pos) per ottenere il contenuto a
3. Calcola δ(q, a) = (q', a', dir)
4. Aggiorna il contenuto della cella a a'
5. new_pos = AGGIORNA_POSIZIONE(pos, dir)
6. Aggiorna pos nel nastro con new_pos
7. Cambia stato da q a q'
```

Passo 4: Inizializzazione

L'input $w = w_1w_2...w_n$ viene posto sul ramo sinistro:

- Nodo ε (radice): w_1
- Nodo 0: w_2
- Nodo 00: w_3
- ...
- Nodo 0^{n-1}: w_n

Passo 5: Correttezza

Lemma: Ogni configurazione della macchina ad albero corrisponde biunivocamente a una configurazione della macchina simulatrice.

Dimostrazione:

- La codifica dell'albero preserva la struttura padre-figlio
- Le operazioni di movimento sono simulate correttamente
- Il contenuto delle celle è mantenuto intatto

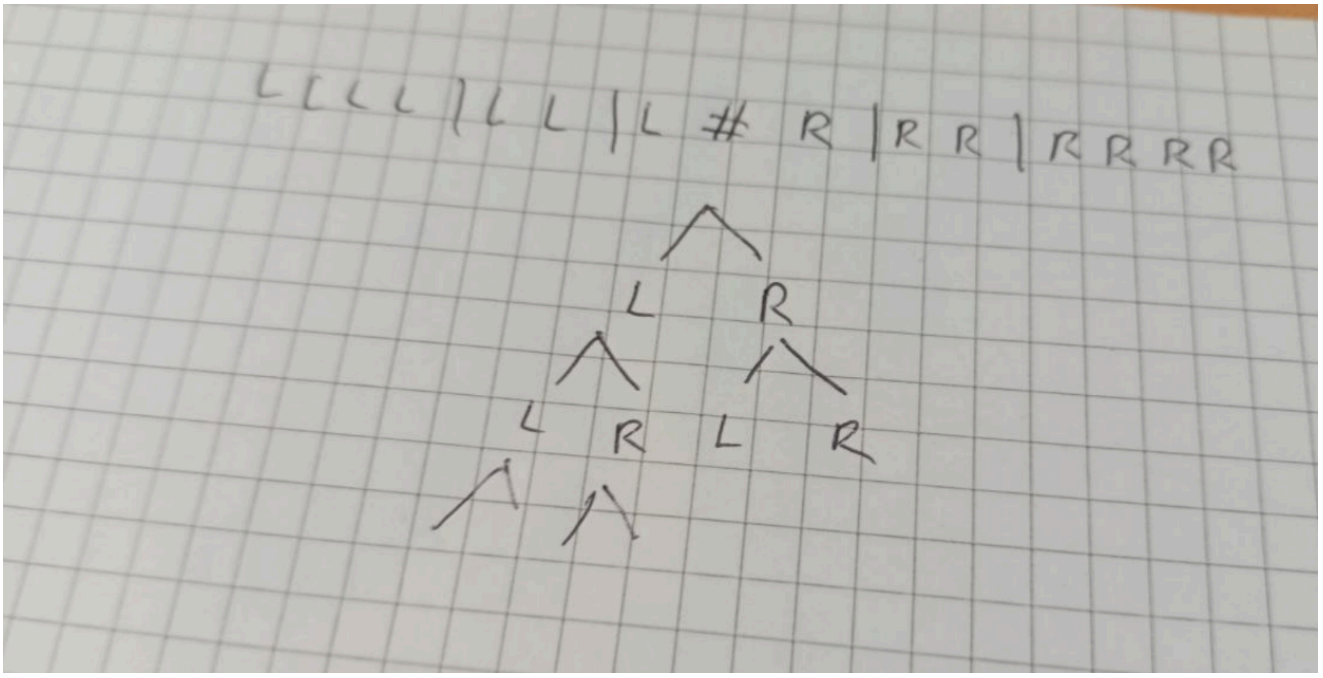
Passo 6: Complessità

Se la macchina ad albero usa t passi e visita al massimo s celle:

- Ogni simulazione di passo richiede O(s) tempo per scansionare il nastro
- La simulazione totale richiede O(t·s) passi

Conclusione: Ogni macchina di Turing ad albero binario è equivalente a una macchina di Turing standard, dimostrando che l'albero binario come struttura di memoria non aumenta il potere computazionale. ■

Osservazione: La simulazione è efficiente e preserva la decidibilità e la semi-decidibilità dei linguaggi riconosciuti.



Simulazione usando il tuo approccio

Ottima idea! Il tuo disegno mostra una simulazione molto più elegante e diretta.

Strategia di simulazione:

Rappresentazione dell'albero:

- **Parte sinistra del nastro:** rappresenta tutti i nodi raggiungibili andando a sinistra dalla radice
- **Simbolo #:** rappresenta la radice dell'albero
- **Parte destra del nastro:** rappresenta tutti i nodi raggiungibili andando a destra dalla radice

Corrispondenza posizioni:

```
Nastro:  ...L₃ L₂ L₁ # R₁ R₂ R₃...
Albero:      ↙     ↓     ↘
           L      radice  R
          ↙ ↘             ↙ ↘
         L   R           L   R
```

Simulazione dei movimenti:

Movimento P (Parent):

- Se la testina è a sinistra di #: move right verso #
- Se la testina è a destra di #: move left verso #
- Se la testina è su #: errore (già alla radice)

Movimento L (Left child):

- Se la testina è su #: move left
- Se la testina è a sinistra di #: move left
- Se la testina è a destra di #: prima torna a # poi move left

Movimento R (Right child):

- Se la testina è su #: move right
- Se la testina è a destra di #: move right
- Se la testina è a sinistra di #: prima torna a # poi move right

Inizializzazione dell'input: L'input $w = w_1w_2...w_n$ viene posto sul ramo sinistro:

```
...␣ w₃ w₂ w₁ # ␣ ␣ ␣...
```

Vantaggi di questo approccio:

1. **Semplicità:** movimenti diretti sul nastro
2. **Efficienza:** O(1) per movimento locale, O(d) per attraversare la radice
3. **Naturalezza:** la struttura del nastro riflette la struttura dell'albero

Algoritmo di simulazione:

```
Per ogni transizione δ(q,a) = (q',a',dir):
1. Leggi il simbolo a nella posizione corrente
2. Scrivi a' nella posizione corrente
3. Simula il movimento dir usando le regole sopra
4. Cambia stato da q a q'
```

Correttezza: Ogni nodo dell'albero corrisponde biunivocamente a una posizione del nastro, preservando completamente la struttura dell'albero originale.

Bonus: TM con differenza tra due linguaggi usando XOR

Macchina di Turing per la differenza di linguaggi usando XOR

Problema: Costruire una TM che decide L = A - B = {w | w ∈ A ∧ w ∉ B} usando l'operazione XOR.

Prerequisiti: Assumo che A e B siano decidibili con TM M_A e M_B rispettivamente.

Strategia con XOR: La differenza A - B può essere vista come: w ∈ A XOR w ∈ A∩B

- Se w ∈ A e w ∉ B: A(w) = 1, (A∩B)(w) = 0 → 1 XOR 0 = 1 ✓
- Se w ∉ A: A(w) = 0, (A∩B)(w) = 0 → 0 XOR 0 = 0 ✗
- Se w ∈ A∩B: A(w) = 1, (A∩B)(w) = 1 → 1 XOR 1 = 0 ✗

Costruzione della TM:

```
TM_Differenza(input w):

Fase 1: Preparazione
1. Copia w su nastro ausiliario T2
2. Copia w su nastro ausiliario T3

Fase 2: Test appartenenza ad A
3. Simula M_A su T2
4. Se M_A rifiuta: vai a RIFIUTA
5. Se M_A accetta: memorizza risultato_A = 1

Fase 3: Test appartenenza a B
6. Simula M_B su T3
7. Se M_B accetta: risultato_B = 1
8. Se M_B rifiuta: risultato_B = 0

Fase 4: Calcolo XOR
9. Calcola XOR_result = risultato_A XOR risultato_B
10. Se XOR_result = 1: vai a ACCETTA
11. Se XOR_result = 0: vai a RIFIUTA

ACCETTA: stato di accettazione
RIFIUTA: stato di rifiuto
```

Implementazione dettagliata dell'XOR:

La sottoroutine XOR può essere implementata con una semplice tabella di verità:

```
Subroutine XOR(bit1, bit2):
- Leggi bit1 dalla cella di memoria
- Leggi bit2 dalla cella di memoria
```

```
- Implementa la tabella:
  * 0 XOR 0 = 0
  * 0 XOR 1 = 1
  * 1 XOR 0 = 1
  * 1 XOR 1 = 0
- Scrivi il risultato
```

Struttura del nastro:

```
|input w|#|copia w per M_A|#|copia w per
M_B|#|risultato_A|risultato_B|XOR_result|
```

Correttezza:

Lemma: TM_Differenza accetta w se e solo se w ∈ A - B.

Dimostrazione:

- **(⇒)** Se TM_Differenza accetta w, allora XOR_result = 1
  - Questo accade solo se risultato_A = 1 e risultato_B = 0
  - Quindi w ∈ A e w ∉ B, cioè w ∈ A - B
- **(⇐)** Se w ∈ A - B, allora w ∈ A e w ∉ B
  - M_A accetta w → risultato_A = 1
  - M_B rifiuta w → risultato_B = 0
  - XOR(1,0) = 1 → TM_Differenza accetta

Complessità: Se M_A e M_B hanno complessità t_A(n) e t_B(n), allora TM_Differenza ha complessità O(t_A(n) + t_B(n)).

Estensione - Differenza simmetrica: Per A ⊕ B = (A - B) ∪ (B - A), modifica la fase 4:

```
XOR_result = risultato_A XOR risultato_B
```

Questo accetta quando w è in uno dei due linguaggi ma non in entrambi.