Automi e Linguaggi Formali
Soluzione della prima prova intermedia

1. Per dimostrare che gli all-ε-NFA riconoscono esattamente la classe dei linguaggi regolari occorre procedere in due versi: dimostrare che ogni linguaggio regolare è riconosciuto da un all-ε-NFA, e che ogni linguaggio riconosciuto da un all-ε-NFA è regolare.

- Per dimostrare che ogni linguaggio regolare è riconosciuto da un all-ε-NFA si parte dal fatto che ogni linguaggio regolare ha un DFA che lo riconosce. È facile vedere che ogni DFA è anche un all-ε-NFA, che non ha ε-transizioni e dove $\delta(q, a) = \{q'\}$ per ogni stato $q \in Q$ e simbolo dell'alfabeto $a \in \Sigma$. In un DFA c'è una sola computazione possibile, quindi lo stato in cui si trova l'automa dopo aver consumato l'input è unicamente determinato: se questo stato è finale il DFA accetta, altrimenti rifiuta. Questo è coerente con la condizione di accettazione degli all-ε-NFA quando c'è un solo possibile stato dove si può trovare l'automa dopo aver consumato l'input.
- Per dimostrare che ogni linguaggio riconosciuto da un all-ε-NFA è regolare, mostriamo come possiamo trasformare un all-ε-NFA in un DFA equivalente. La trasformazione è descritta dal seguente algoritmo:

```
Require: Un all-ε-NFA N = (Q_N, Σ, q_0, δ_N, F_N)
Ensure: Un DFA D = (Q_D, Σ, S_0, δ_D, F_D) equivalente a N
  S_0 ← Eclose(q_0)                        ▷ Lo stato iniziale è la chiusura di q_0
  Q_D ← {S_0}                              ▷ Q_D sarà l'insieme degli stati del DFA
  if S_0 ⊆ F_N then                        ▷ Se S_0 contiene solamente stati finali dell'all-ε-NFA ...
      F_D ← {S_0}                          ▷ ... allora S_0 è stato finale del DFA, ...
  else
      F_D ← ∅                              ▷ ... altrimenti no
  end if
  while Q_D contiene stati senza transizioni uscenti do      ▷ Ciclo principale
      Scegli S ∈ Q_D senza transizioni uscenti
      for all a ∈ Σ do                     ▷ una transizione per ogni simbolo dell'alfabeto
          S' ← ∅                           ▷ stato di arrivo della transizione
          for all q ∈ S do                 ▷ lo stato di partenza S è un insieme di stati di N
              S' ← S' ∪ δ_N(q, a)          ▷ aggiungi gli stati δ_N(q, a) ad S'
          end for
          S' ← Eclose(S')                  ▷ Chiusura di S' per ϵ-transizioni
          Q_D ← Q_D ∪ {S'}                 ▷ Aggiungi lo stato S' al DFA
          if S' ⊆ F_N then                 ▷ Se S' contiene solamente stati finali ...
              F_D ← F_D ∪ {S'}             ▷ ... allora S' è finale per il DFA
          end if
          δ_D(S, a) ← S'                   ▷ Aggiungi la transizione da S ad S' con input a al DFA
      end for
  end while
  return D = (Q_D, Σ, S_0, δ_D, F_D)
```

  L'algoritmo è del tutto analogo a quello usato per trasformare un ε-NFA in un DFA, con la sola differenza della definizione degli stati finali, che in questo caso sono tutti gli insiemi di stati S che contengono solamente stati finali, per rappresentare il fatto che un all-ε-NFA accetta quando tutti i possibili stati in cui si può trovare dopo aver consumato l'input sono stati finali.

  Soluzione alternativa: in alternativa all'algoritmo si può dare la definizione del DFA $D = (Q_D, \Sigma, S_0, \delta_D, F_D)$ equivalente all'all-ε-NFA $N = (Q_N, \Sigma, q_0, \delta_N, F_N)$ specificando le componenti del DFA:
  - l'insieme degli stati è l'insieme delle parti di Q_N: $Q_D = \{S \mid S \subseteq Q_N\}$;
  - lo stato iniziale è la ε-chiusura di q_0: $S_0 = \textsc{Eclose}(q_0)$;
  - la funzione di transizione "simula" le transzioni di N:

  $$\delta_D(S, a) = \textsc{Eclose}\Big(\bigcup_{p \in S} \delta_N(p, a)\Big)$$

  - l'insieme degli stati finali è l'insieme delle parti di F_N: $F_D = \{S \mid S \subseteq F_N\}$.

  Anche questa definizione è analoga a quella che trasforma un ε-NFA in un DFA, con la sola differenza della definizione degli stati finali.

2. (a) **Prima alternativa:** Possiamo dimostrare che L_1 non è regolare modificando la dimostrazione che il linguaggio $\{0^n1^n \mid n \geq 0\}$ non è regolare. Supponiamo che L_1 sia regolare: allora deve esistere un DFA A che lo riconosce. Il DFA avrà un certo numero di stati k. Consideriamo la computazione di A con l'input ab^k:

$$\text{start} \longrightarrow p_0 \xrightarrow{a} \underbrace{p_1 \xrightarrow{b} p_2 \xrightarrow{b} \cdots \xrightarrow{b} p_{k+1}}_{k+1 \text{ stati}}$$

Poiché la sequenza di stati $p_1, p_2, \ldots, p_{k+1}$ che legge b^k è composta da $k+1$ stati, allora esiste uno stato che si ripete: possiamo trovare $i < j$ tali che $p_i = p_j$. Chiamiamo q questo stato. Cosa succede quando l'automa A legge c^i partendo da q?

- Se termina in uno stato finale, allora l'automa accetta, sbagliando, la parola ab^jc^i.
- Se termina in uno stato non finale allora l'automa rifiuta, sbagliando, la parola ab^ic^i

In entrambi i casi abbiamo trovato un assurdo, quindi L_1 non può essere regolare.

Seconda alternativa: Per le proprietà di chiusura dei linguaggi regolari, sappiamo che l'intersezione di linguaggi regolari è un linguaggio regolare. Se intersechiamo L_1 con un linguaggio regolare e quello che otteniamo non è un linguaggio regolare, allora possiamo concludere che L_1 non può essere regolare. Consideriamo il linguaggio $L' = L_1 \cap \{a^*b^*c^*\} = \{ab^mc^m \mid m \geq 0\}$, e usiamo il Pumping Lemma per dimostrare che non è regolare. Supponiamo per assurdo che L' sia regolare:

- sia k la lunghezza data dal Pumping Lemma;
- consideriamo la parola $w = ab^kc^k$, che appartiene ad L' ed è di lunghezza maggiore di k;
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq k$;
- siccome $|xy| \leq k$, allora x e y devono cadere all'interno del prefisso ab^k della parola w. Ci sono due casi possibili, secondo la struttura di y:
  - y contiene la a iniziale. In questo caso la parola xy^2z non appartiene ad L' perché contiene due a;
  - y contiene solamente b. In questo caso la parola xy^2z non appartiene ad L' perché contiene più b che c.

In entrambi i casi abbiamo trovato un assurdo quindi L' non è regolare, e possiamo concludere che neanche L_1 può essere regolare.

(b) Mostriamo che L_1 si comporta come un linguaggio regolare rispetto al Pumping Lemma.

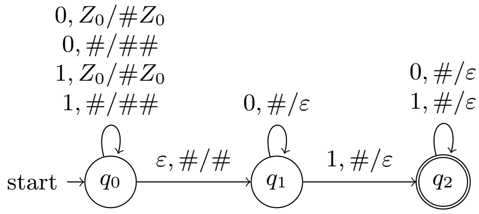
- Poniamo come lunghezza del pumping $k = 2$.
- Data una qualsiasi parola $w = a^\ell b^m c^n \in L_1$ di lunghezza maggiore o uguale a 2, si possono presentare vari casi, secondo il numero di a presenti nella parola:
  - se c'è una sola a, allora $w = ab^mc^m$. Scegliamo la suddivisione $x = \varepsilon$, $y = a$ e $z = b^mc^m$. Per ogni esponente $i \geq 0$, la parola $xy^iz = a^ib^mc^m$ appartiene a L_1: se $i = 1$ allora il numero di b è uguale al numero di c come richiesto, mentre se $i \neq 1$ il linguaggio non pone condizioni sul numero di b e c;
  - se ci sono esattamente due a, allora $w = aab^mc^n$. Scegliamo la suddivisione $x = \varepsilon$, $y = aa$ e $z = b^mc^n$. Per ogni esponente $i \geq 0$, la parola $xy^iz = a^{2i}b^mc^n$ appartiene a L_1: il numero di a è pari, quindi sempre diverso da 1, e ricadiamo nelle situazioni in cui il linguaggio non pone condizioni sul numero di b e c;
  - se ci sono almeno tre a, allora $w = a^\ell b^mc^n$ con $\ell \geq 3$. Scegliamo la suddivisione $x = \varepsilon$, $y = a$ e $z = a^{\ell-1}b^mc^n$. Per ogni esponente $i \geq 0$, la parola $xy^iz = a^{i+\ell-1}b^mc^n$ contiene almeno due a, e quindi appartiene a L_1, perché rientra nelle situazioni in cui il linguaggio non pone condizioni sul numero di b e c;
  - se non ci sono a, allora $w = b^mc^n$. Scegliamo la suddivisione che pone $x = \varepsilon$, y uguale al primo carattere della parola e z uguale al resto della parola. Per ogni esponente $i \geq 0$, la parola xy^iz sarà nella forma b^pc^q per qualche $p, q \geq 0$ e quindi appartenente a L_1, perché quando non ci sono a il linguaggio non pone condizioni sul numero di b e c.

In tutti i casi possibili la parola può essere pompata senza uscire dal linguaggio, quindi L_1 rispetta le condizioni del Pumping Lemma.

(c) Il Pumping Lemma stabilisce che se un linguaggio è regolare, allora deve rispettare certe condizioni. Il verso opposto dell'implicazione non è vero: possono esistere linguaggi, come L_1, che rispettano le condizioni ma non sono regolari. Di conseguenza, i punti (a) e (b) non contraddicono il lemma.

3. (a) Il PDA che riconosce L_2 opera nel modo seguente:

- inizia a consumare l'input ed inserisce un carattere # per ogni simbolo che consuma, rimanendo nello stato q_0;
- ad un certo punto, sceglie nondeterministicamente che ha consumato la prima metà della parola, e si sposta nello stato q_1;
- in q_1, estrae un carattere # dalla pila per ogni 0 che consuma dall'input;
- quando legge il primo 1 nella seconda parte della parola, estrae un # dalla pila e si sposta in q_2, che è uno stato finale;
- in q_2, continua ad estrarre un # dalla pila per ogni carattere che consuma (0 o 1).



Per accettare una parola, il PDA deve:

- inserire nella pila un certo numero di #
- estrarre dalla pila un numero di # minore o uguale di quanti ne ha inserito in pila
- consumare almeno un 1 durante lo svuotamento della pila

Quindi l'automa accetta solo parole $w = uv$ dove u è la parte di parola consumata durante la fase di riempimento della pila e v è la parte di parola consumata durante lo svuotamento della pila. La parola v contiene almeno un 1 ed è di lunghezza minore o uguale a u. Se u è più corta di v il PDA svuota la pila prima di riuscire a consumare tutta la parola e si blocca. Se invece v non contiene 1 allora il PDA termina la computazione nello stato q_1 che non è finale.

(b) Per costruire una CFG che genera L_2 prendiamo una qualsiasi parola w che sta in L_2. Se consideriamo l'ultima occorrenza di un 1 nella parola, possiamo riscrivere la parola come $w = u10^k$, con $k \geq 0$ e $|u| \geq k+1$, perché l'ultima occorrenza di 1 deve stare nella seconda metà della parola. Se spezziamo ulteriormente u in $u = xy$ con $|x| = k+1$ e $|y| \geq 0$, allora possiamo definire la grammatica che genera L_2 come segue:

$$
\begin{aligned}
S &\to 0S0 \mid 1S0 \mid 0T1 \mid 1T1 \\
T &\to 0T \mid 1T \mid \varepsilon
\end{aligned}
$$

Nella grammatica, la variabile S genera stringhe del tipo $xT10^k$ con $x \in \{0,1\}^*$ e $|x| = k+1$, mentre T genera stringhe $y \in \{0,1\}^*$ con $|y| \geq 0$. Quindi la grammatica genera tutte e sole le stringhe del tipo $xy10^k$ dove $|xy| \geq k+1$, che corrispondono alle stringhe che stanno nel linguaggio L_2.