1. **(12 punti)** Una macchina di Turing con "copia e incolla" (CPTM) è una macchina di Turing deterministica a singolo nastro, che può copiare e incollare porzioni di nastro. Le operazioni che una CPTM può fare sono le seguenti:

   - selezionare l'inizio della porzione di nastro da copiare;
   - selezionare la fine della porzione di nastro da copiare;
   - copiare la porzione di nastro selezionata, sovrascrivendo il contenuto della cella corrente e di tante celle a destra della cella corrente quante sono le celle necessarie per effettuare la copia;
   - fare le normali operazioni di scrittura e spostamento a sinistra o a destra della testina.

   Fare una operazione di copia senza che sia stata selezionata una porzione di nastro non ha effetto.

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

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

a)

$$\delta: Q\ x\ \Gamma \mapsto Q\ x\ \Gamma\ x\ \{L, R, C, I\}$$

C = copia, I = incolla, Left / Right

b) Mostriamo come convertire una macchina di Turing con copia e incolla M in una TM deterministica a nastro singolo S equivalente. La simulazione usa il simbolo speciale # per segnare le celle che vengono eliminate.

S = Su input w:

- La simulazione della mossa di sinistra $(\delta, q, a) = (r, b, L)$ procede come la TM standard: si sposta a sinistra finché non trova un # a sinistra o a destra nel caso della lettura. Nel caso della scrittura, controlla sempre fino a dove è stata eseguita e controlla che ci sia spazio nei bounds.

- La simulazione della mossa di destra $(\delta, q, a) = (r, b, R)$ procede come la TM standard: si sposta a destra finché non trova un # a sinistra o a destra nel caso della lettura. Nel caso della scrittura, controlla sempre fino a dove è stata eseguita e controlla che ci sia spazio nei bounds.

- La simulazione della mossa di copia $(\delta, q, a) = (r, b, C)$ seleziona l'inizio della porzione di nastro da copiare dentro i cancelletti (#) che rappresentano il limite e verifico che la fine dell'input da copiare sia entro la fine della porzione di nastro copiabile (si intende sempre entro i # - ndr, non ci interessa scriverlo, ma è per ricordarcelo). L'operazione di copia inizia da una cella che viene sovrascritta (pre: questa cella deve essere valida) fino ad una cella (valida) seguendo le transizioni L e R (in memoria).

- La simulazione della mossa di incolla $(\delta, q, a) = (r, b, I)$ seleziona l'inizio della porzione di nastro da incollare dentro i cancelletti (#) che rappresentano il limite e verifico che la fine dell'input da copiare sia entro la fine della porzione di nastro copiabile (si intende sempre entro i # - ndr, non ci interessa scriverlo, ma è per ricordarcelo). L'operazione di incolla inizia da una cella che viene sovrascritta (pre: questa cella deve essere valida) fino ad una cella (valida) seguendo le transizioni L e R (in memoria).

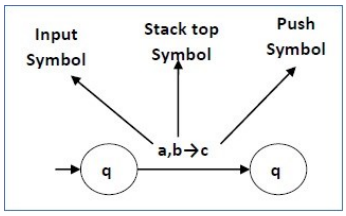
- Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora S termina con accettazione. Se in qualsiasi momento la simulazione raggiunge lo stato di rifiuto di M, allora S termina con rifiuto. Negli altri casi continua la simulazione dal "codice" di cui sopra.

3.9 Let a k-PDA be a pushdown automaton that has k stacks. Thus a 0-PDA is an NFA and a 1-PDA is a conventional PDA. You already know that 1-PDAs are more powerful (recognize a larger class of languages) than 0-PDAs.

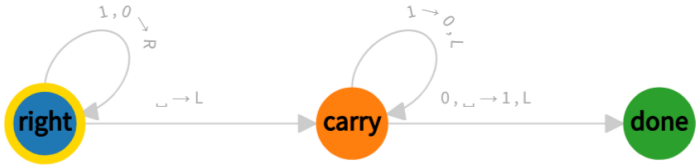
a. Show that 2-PDAs are more powerful than 1-PDAs.

b. Show that 3-PDAs are not more powerful than 2-PDAs. (Hint: Simulate a Turing machine tape with two stacks.)

Pushdown = Pila (Stack) – Pop/Push (Togli/Metti)



La TM "grafica" assomiglia ad un PDA = ecco perché (purtroppo per noi) esistono questi esercizi con i PDA per le TM!



a.

Primo modo:

- Usare una descrizione simil-TM per dire che i 2-PDA (PDA con 2 pile) sono più potenti dei PDA con 1 pila sola

Gli 1-PDA sono automi a pila con una sola coda: possiedono operazioni di pop e di push rispettando le operazioni della propria funzione di transizione – per l'operazione di pop, significa che scorro tutto a sinistra sull'input, recupero il primo input e lo elimino, rispostandomi verso la fine "shiftando" (spostando) tutto l'input. L'operazione di push scorre tutto l'input, arriva alla fine e inserisce il nuovo simbolo dell'alfabeto secondo la funzione di transizione.

Questo può essere facilmente simulato da una TM a nastro singolo:

- L'operazione di pop significa scorrere tutto il nastro fino all'inizio entro i confini, trovare il primo simbolo e toglierlo
- L'operazione di push significa scorrere tutto il nastro fino alla fine entro i confini, trovare l'ultimo simbolo e buttarne dentro un altro
- Lo spostamento a sinistra e a destra dipende sempre dal tipo di input e della funzione di transizione

I 2-PDA fanno la stessa cosa ma con due nastri: un nastro per la parte sinistra, un nastro per la parte destra, di fatto raddoppiando l'efficienza facendo il doppio delle operazioni date le stesse operazioni (dati due nastri, abbiamo il doppio della potenza computazionale)

Alternativamente:

a) Un 2-PDA può simulare una Macchina di Turing:

- La prima pila simula la parte sinistra del nastro della TM.

- La seconda pila simula la parte destra del nastro della TM.
- Lo stato interno del 2-PDA può tenere traccia dello stato della TM e del simbolo corrente sotto la testina.

b) Simulazione delle operazioni della TM:

- Lettura: Leggere il top della seconda pila.
- Scrittura: Modificare il top della seconda pila.
- Movimento a sinistra: Spostare un simbolo dalla prima pila alla seconda.
- Movimento a destra: Spostare un simbolo dalla seconda pila alla prima.

Secondo modo:

- Linguaggio riconosciuto dal più potente che non viene riconosciuto dal meno potente.

DFA/NFA = riconoscono linguaggi regolari

PDA = riconoscono linguaggi context-free

Per dimostrare questo, possiamo mostrare che un 2-PDA può riconoscere un linguaggio che un 1-PDA non può riconoscere.

Consideriamo il linguaggio L = {a^n b^n c^n | n ≥ 0}

1. Un 1-PDA non può riconoscere questo linguaggio perché:
    - Può usare la sua unica pila per contare 'a' e 'b', ma non ha modo di verificare il numero di 'c'.
2. Un 2-PDA può riconoscere questo linguaggio:
    - Usa la prima pila per contare 'a'.
    - Usa la seconda pila per contare 'b'.
    - Confronta entrambe le pile con il numero di 'c'.

Questo dimostra che i 2-PDA sono strettamente più potenti dei 1-PDA.

b)

Per dimostrare questo, possiamo mostrare che ogni 3-PDA può essere simulato da un 2-PDA.

Simulazione di un 3-PDA con un 2-PDA:

1. Usa la prima pila del 2-PDA per simulare le prime due pile del 3-PDA.
2. Usa la seconda pila del 2-PDA per simulare la terza pila del 3-PDA.
3. Codifica il contenuto delle prime due pile del 3-PDA nella prima pila del 2-PDA usando un alfabeto esteso.

Dimostra che è regolare:

- Trova l'esempio formale per capire **bene** come scrivere la funzione di transizione
- L'approccio più semplice è avere un NFA/DFA D' che inizia dallo stesso di D (che riconosce già il linguaggio L), ha gli stessi stati finali ma cambia nella funzione di transizione
- Poi, costruisci l'esempio formale intorno
    - Stato iniziale (uguale)
    - Alfabeto (uguale)
    - Transizione (cambia)
    - Stati finali (uguale)
    - Insieme degli stati (uguale)

1. (12 punti) Diciamo che una stringa x è un *prefisso* della stringa y se esiste una stringa z tale che $xz = y$, e che è un *prefisso proprio* di y se vale anche $x \neq y$. Dimostra che se $L \subseteq \Sigma^*$ è un linguaggio regolare allora anche il linguaggio

$$NOPREFIX(L) = \{w \in L \mid \text{ nessun prefisso proprio di } w \text{ appartiene ad } L\}$$

è un linguaggio regolare.

$$y = "abcd"$$

$$z = "cd",\ x = "ab",\ xz = "abcd"$$

Tale che $xz = y$ → $abcd = abcd$

Prefisso = viene prima Proprio = fa già parte della stringa

Condizione: $xz = y$ → NO prefissi propri! (NON vuoi la condizione! – vuol dire "Non esiste z")

Data una funzione di transizione: δ'((q, 0), a) = (δ(q, a), 0)

- Sintassi **generale** (Input = Output)
    - Dato lo stato "q" con input "0" secondo il carattere "a" appartenente all'alfabeto (input)
    - Arrivo allo stato "q" con output "0" dato il carattere "a" di input

Sia M' = (Q, Σ, δ, q_0, F) il DFA che riconosce L .

- L'alfabeto è composto solo da caratteri $\{a, \ldots z\} \in \Sigma$
- Lo stato iniziale accoglie uno qualsiasi dei simboli di input e avanza secondo la funzione di transizione
- La funzione di transizione
    - δ'((q, x), a) = (δ(q, a), x)
        - Da leggersi come
            - Dato lo stato "q" con input "x" secondo il carattere "a" appartenente all'alfabeto (input)
            - Arrivo allo stato "q" con output "x" dato il carattere "a" di input

Il DFA codifica la stringa "x" secondo tutti i suoi caratteri e avanza secondo la funzione di transizione.

$$x_1 \rightarrow x_2 \rightarrow \cdots x_n$$

    - δ'((q, y), a) = (δ(q, a), y)
        - Da leggersi come

- Dato lo stato “q” con input “y” secondo il carattere “a” appartenente all’alfabeto (input)
- Arrivo allo stato “q” con output “y” dato il carattere “a” di input

Il DFA codifica la stringa “y” secondo tutti i suoi caratteri e avanza secondo la funzione di transizione.

$$y_1 \rightarrow y_2 \rightarrow \cdots y_n$$

- δ'((q, z), a) = (δ(q, a), z)
    - Da leggersi come
        - Dato lo stato “q” con input “z” secondo il carattere “a” appartenente all’alfabeto (input)
        - Arrivo allo stato “q” con output “z” dato il carattere “a” di input

Il DFA avanza verso uno stato “pozzo” (mettiamo una fine “non finale” – “facciamo finta” di farlo finire per levarci dalle palle il simbolo e continuare felici secondo la funzione di transizione)

Approccio alternativo:

Costruiremo un automa a stati finiti deterministico (DFA) che riconosce NOPREFIX(L) utilizzando il DFA che riconosce L.

Passi della dimostrazione:

1. Sia M = (Q, Σ, δ, q_0, F) il DFA che riconosce L.
2. Costruiamo M' = (Q', Σ, δ', q_0', F') che riconosce NOPREFIX(L) come segue:
    - Q' = Q × {0, 1}
    - q_0' = (q_0, 0) se $q_0 \notin F$, altrimenti q_0' = (q_0, 1)
    - F' = {(q, 0) | q ∈ F}
    - Per ogni a ∈ Σ e q ∈ Q: δ'((q, 0), a) = (δ(q, a), 0) se δ(q, a) ∉ F δ'((q, 0), a) = (δ(q, a), 1) se δ(q, a) ∈ F δ'((q, 1), a) = (δ(q, a), 1)
3. Spiegazione del funzionamento di M':
    - Il secondo componente dello stato (0 o 1) indica se abbiamo incontrato un prefisso proprio che appartiene a L.
    - Iniziamo con 0 a meno che la stringa vuota non sia in L.
    - Passiamo a 1 non appena raggiungiamo uno stato finale di M, e rimaniamo in 1 da quel punto in poi.
    - Accettiamo solo se siamo in uno stato finale di M e il secondo componente è 0.
4. Correttezza:
    - M' accetta una stringa w se e solo se w ∈ L (raggiunge uno stato in F) e nessun prefisso proprio di w è in L (il secondo componente rimane 0).
    - Questo corrisponde esattamente alla definizione di NOPREFIX(L).

1.31 For any string $w = w_1 w_2 \cdots w_n$, the ***reverse*** of w, written $w^{\mathcal{R}}$, is the string w in reverse order, $w_n \cdots w_2 w_1$. For any language A, let $A^{\mathcal{R}} = \{w^{\mathcal{R}} \mid w \in A\}$.

Costruiremo un NFA che riconosce A^R a partire dal DFA che riconosce A.

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

Costruiamo M' = (Q', Σ, δ', q_0', F') che riconosce A^R come segue:

1. Q' = Q (gli stessi stati di M)

2. Σ rimane invariato

3. q_0' = F (l'insieme degli stati finali di M diventa lo stato iniziale di M')

4. F' = $\{q_0\}$ (lo stato iniziale di M diventa l'unico stato finale di M')

5. δ' è definita come:

Per ogni p, q ∈ Q e a ∈ Σ:

q ∈ δ'(p, a) se e solo se δ(q, a) = p in M

Specifichiamo che:

- M' inizia dagli stati finali di M

- M' segue le transizioni di M all'indietro

- M' accetta quando raggiunge lo stato iniziale di M

Dimostrazione di correttezza:

1. Se w ∈ A, allora esiste una sequenza di transizioni in M:

q_0 --(w_1)--> q_1 --(w_2)--> ... --(w_n)--> q_n, dove q_n ∈ F

2. Per w^R in M', avremo:

q_n --(w_n)--> ... --(w_2)--> q_1 --(w_1)--> q_0

3. Poiché q_n ∈ F = q_0' e q_0 ∈ F', M' accetta w^R

4. Viceversa, se M' accetta una stringa x, seguendo le transizioni all'indietro in M si otterrà una computazione accettante per x^R in M

Quindi, L(M') = A^R

13. $L = \{a^{2k}w \mid w \in \{a, b\}^*, |w| = k\}$

Esempio:

$$w = bb$$

$$a^{2(2)}bb$$

$$w = xy^iz = a^kb^k$$

Dato $k \geq 0$, dividiamo la stringa in tre parti di cui una non vuota ($y \neq \epsilon$) e siamo dentro il linguaggio ($xy \leq i$), scegliamo una stringa del tipo a^kb^k.

Per bilanciare questa cosa essendo che una delle due parti non deve essere vuota, significa che noi potremmo trovarci ad avere:

$$a^p, a^q, b^k$$

$$a^{p+q}b^{k-p-q}$$

Il numero di “a” è diverso dal numero di “b” e il linguaggio non è più regolare.

20. $L = \{ a^{n!} \mid n \geq 0 \}$

$$w = xy^iz, y \neq \epsilon, i \geq 0, |xy| \leq i$$

Supponiamo che L sia regolare.

1. Esiste una costante p > 0 (lunghezza di pumping) tale che ogni stringa s ∈ L con |s| ≥ p può essere scritta come s = xyz, dove:

- |xy| ≤ p

- |y| > 0

- ∀i ≥ 0, xy^i z ∈ L

2. Scegliamo s = a^(p!). Notiamo che s ∈ L e |s| ≥ p.

3. Per il Lemma di Pumping, s = xyz con le proprietà sopra menzionate.

4. Poiché |xy| ≤ p, sappiamo che |y| ≤ p.

5. Sia |y| = k, dove 0 < k ≤ p.

6. Consideriamo xy^2z:

- |xy^2z| = p! + k

7. Ma p! + k non è della forma n! per nessun n intero:

- p! < p! + k < (p+1)!

- Non esiste un n tale che n! = p! + k

Abbiamo trovato una stringa xy^2z che, secondo il Lemma di Pumping, dovrebbe essere in L, ma non lo è.

Questo contraddice l'ipotesi che L sia regolare.

Quindi, L = {a^(n!) | n ≥ 0} non è un linguaggio regolare.

3. Il linguaggio

$$L = \left\{ a^n b^m c^{n-m} : n > m > 0 \right\}$$

è regolare? Motivare in modo formale la risposta.

Soluzione: Il linguaggio non è regolare. Supponiamo per assurdo che lo sia:

- sia h la lunghezza data dal Pumping Lemma; possiamo supporre senza perdita di generalità che $h > 1$;
- consideriamo la parola $w = a^h b c^{h-1}$, che appartiene ad L ed è di lunghezza maggiore di h;
- sia $w = xyz$ una suddivisione di w tale che $y \neq \varepsilon$ e $|xy| \leq h$;
- poiché $|xy| \leq h$, allora xy è completamente contenuta nel prefisso a^h di w, e quindi sia x che y sono composte solo da a. Inoltre, siccome $y \neq \varepsilon$, possiamo dire che $y = a^p$ per qualche valore $p > 0$. Allora la parola xy^2z è nella forma $a^{h+p} b c^{h-1}$, e quindi non appartiene al linguaggio perché il numero di c non è uguale al numero di a meno il numero di b (dovrebbero essere $h + p - 1$ mentre sono solo $h - 1$).

Abbiamo trovato un assurdo quindi L non può essere regolare.

2.43 For strings w and t, write $w \triangleq t$ if the symbols of w are a permutation of the symbols of t. In other words, $w \triangleq t$ if t and w have the same symbols in the same quantities, but possibly in a different order.

For any string w, define $SCRAMBLE(w) = \{t \mid t \triangleq w\}$. For any language A, let $SCRAMBLE(A) = \{t \mid t \in SCRAMBLE(w) \text{ for some } w \in A\}$.

a. Show that if $\Sigma = \{0,1\}$, then the *SCRAMBLE* of a regular language is context free.

Se L è un linguaggio context-free, allora esiste una grammatica G in FNC che lo genera. Possiamo costruire una grammatica G' che genera il linguaggio SCRAMBLE(L):

- G' ha la stessa variabile iniziale di G e hanno lo stesso insieme di variabili
- L'alfabeto di entrambe è tra 0 ed 1
- Per ogni variabile V di G, esistono le seguenti regole per V':
    - S -> AB
    - A -> 0^n 1^m
    - B -> (prima permutazione) – tutte le combo
        - S-> AB
        - A -> 0011
        - B -> 0101
        - C -> 1010
        - ...
    - Oppure (generale – con una regola)
        - A -> wB
        - B -> (w) prima perm. | seconda perm. | ...

010 → 100 / 001 / 010 (permutazioni)

Scritta estesamente:

1. Partire da una grammatica lineare destra per A

2. Costruire una CFG per SCRAMBLE(A)

Sia G = (V, Σ, R, S) una grammatica lineare destra per A, dove:

- V è l'insieme delle variabili

- Σ = {0, 1}

- R è l'insieme delle regole

- S è il simbolo iniziale

Costruiamo G' = (V', Σ, R', S') per SCRAMBLE(A):

1. V' = V ∪ {S', Z}

2. Σ rimane {0, 1}

3. S' è il nuovo simbolo iniziale

4. R' contiene le seguenti regole:

a. S' → SZ

b. Per ogni regola A → wB in R (w è una stringa di 0 e 1, B ∈ V):

- Se w contiene n zeri e m uni:

A → 0^n1^mB | Permutazioni(0^n1^m)B

c. Per ogni regola A → w in R (w è una stringa terminale):

- Se w contiene n zeri e m uni:

A → 0^n1^mZ | Permutazioni(0^n1^m)Z

d. Z → 0Z | 1Z | ε

- La regola S' → SZ inizia la generazione e permette l'aggiunta di zeri e uni alla fine.

- Le regole di tipo (b) e (c) generano tutte le permutazioni possibili dei simboli nelle regole originali.

- La variabile Z alla fine permette di aggiungere un numero arbitrario di 0 e 1 in qualsiasi ordine.

Correttezza:

1. Se w ∈ A, allora G' può generare tutte le permutazioni di w:

- G' simula la derivazione di w in G, ma permette tutte le permutazioni ad ogni passo.

- Z alla fine consente di aggiungere simboli per ottenere permutazioni di lunghezza maggiore.

2. Se t ∈ SCRAMBLE(A), allora esiste w ∈ A tale che t ≅ w:

- G' può simulare la derivazione di w in G, generando una permutazione che corrisponde a t.

- Se t è più lunga di w, Z permette di aggiungere i simboli necessari.

Quindi, L(G') = SCRAMBLE(A)

2. (12 punti) I *grawlix* sono sequenze di simboli senza senso che sostituiscono le parolacce nei fumetti.

Un linguaggio è *volgare* se contiene almeno un grawlix. Considera il problema di determinare se il linguaggio di una TM è volgare.

(a) Formula questo problema come un linguaggio $GROSS_{TM}$.
(b) Dimostra che il linguaggio $GROSS_{TM}$ è indecidibile.

(a) $GROSS_{TM} = \{\langle M, L\rangle \mid M$ è $una\ TM\ che\ determina\ se\ L$ è $volgare\ (i.e. \exists w \in L \mid w = \Sigma \setminus \{a \dots z\}^*\}$

(b)

$$A_{TM} \leq_m GROSS_{TM}$$

F = Su input <M, L>, dove M è una TM ed L un linguaggio:

- Costruisci la macchina M' su input x
  - Copia tutto l'input sul nastro
  - Se $x \neq$ "#" (hai almeno un carattere grawlix), rifiuta
  - Se $x =$ "#", esegui M
    - Se M accetta, allora accetta; altrimenti rifiuta

- Restituisci $\langle M' \rangle$

Conclusioni:

1. Se $\langle M, w \rangle \in A_{TM} \Rightarrow M' \in GROSS_{TM}$, allora M' costruita dalla funzione accetta la parola x contenente almeno un grawlix ($\exists!\, w \mid w \in GROSS$), quindi la macchina M accetta fermandosi
2. Se $M' \in GROSS_{TM} \Rightarrow \langle M, w \rangle \in A_{TM}$, allora la computazione di M' genera almeno un carattere grawlix e la macchina M si ferma accettando la parola, dato che altrimenti la computazione non termina rifiutando oppure va in loop.

Per concludere, siccome A_TM è indecidibile, allora GROSS_TM è indecidibile.