Esercizio 1 - TM

1. **(12 punti)** Una macchina di Turing spreca-nastro è simile a una normale macchina di Turing deterministica a nastro singolo semi-infinito, ma può spostare la testina nella parte non ancora utilizzata del nastro. In particolare, se tutte le celle dopo la cella numero s del nastro sono vuote, e la cella s è non vuota, allora la testina può spostarsi nella cella numero $2s$. A ogni passo, la testina della TM spreca-nastro può spostarsi a sinistra di una cella (L), a destra di una cella (R) o dopo la parte non vuota del nastro (J).

   (a) Dai una definizione formale della funzione di transizione di una TM spreca-nastro.

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

1.a - Funzione di transizione

δ = Transizione
Q = Stati
Γ = Nastro

L = Left
R = Right
J = Jump di $2s$ celle

$\mapsto$ = Mappa l'input all'output

$\delta : Q \times \Gamma \rightarrow Q \times \Gamma \times \{L, R, J\}$

1.b - Dimostrazione

"Classe dei linguaggi Turing-riconoscibili"
3 Scelte:

1. TM a nastro singolo
2. TM multinastro
3. TM nastro semi-infinito
    1. Di solito infinito a destra

(Andata $\Rightarrow$)

TM Singolo $\Rightarrow$ TM Variante
$(L, R) \Rightarrow (L, R, J)$

|_| = Blank = La cella è vuota

Mostriamo come usare una macchina a nastro singolo equivalente.

S = Su input w:

- $\delta(q, a) = (r, b, L)$
  - Da stato "q", input "a", vai allo stato "r" facendo "L" scrivendo "b"

S procede come nella TM standard: S
scrive b sul nastro e muove la testina di una cella a sinistra.

Se la simulazione arriva ad inizio nastro (sopra il #), ti sposti immediatamente di una cella a destra rimanendo dentro il resto del nastro.

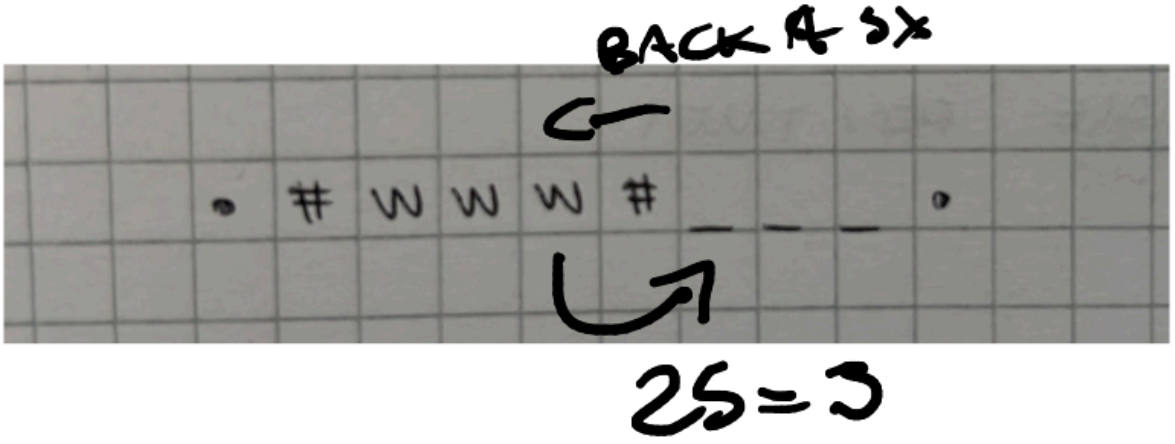
- $\delta(q, a) = (r, b, R)$
  - Da stato "q", input "a", vai allo stato "r" facendo "R" scrivendo "b"

S procede come nella TM standard: S
scrive b sul nastro e muove la testina di una cella a destra scrivendo un pallino.

Se la simulazione arriva ad un cancelletto che delimita la fine del nastro allora S si sposta immediatamente di una cella a sinistra scrivendo un blank al posto del cancelletto e poi riscrivendo il cancelletto sulla cella attuale.

(Traduzione Paragrafo 2 = Se fine nastro, non andare oltre, altrimenti scrivi sempre!)

- $\delta(q, a) = (r, b, J)$
  - Da stato "q", input "a", vai allo stato "r" facendo "J" scrivendo "b"



S scrive "b" nella cella corrente e arriva ad un # che rappresenta la fine del nastro. Se arriva ad un #, si sposta di una cella a sinistra e poi realizza la "J" spostandosi di 2s celle per prendere la porzione di nastro non scritta.

Continua in questo modo marcando il nuovo # che è il nuovo inizio nastro e parto da lì realizzando tutte le altre transizioni (= mi muovo a zig-zag) finché non esaurisco tutti i possibili input.

In questa ultima fase, se una delle celle marcate è il # alla fine del nastro, allora la TM lo sostituisce con un blank e scrive un # immediatamente a destra dell'ultima cella marcata prima di continuare con la simulazione.

La fine dice sempre:

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.

Altrimenti, torna allo stato (2) = continua con le transizioni (in questo caso da sinistra).

(Ritorno $\Leftarrow$)

TM Singolo $\Leftarrow$ TM Variante
$(L, R) \Leftarrow (L, R, J)$

Il verso di ritorno è "ovvio", perché la sprecanastro ha già questi movimenti della macchina a nastro singolo.

La prima dimostrazione è banale: le TM deterministiche a singolo nastro sono un caso particolare di TM spreca-nastro che non effettuano mai la mossa J per saltare oltre la parte non vuota del nastro. Di conseguenza, ogni linguaggio Turing-riconoscibile è riconosciuto da una TM spreca-nastro.

Per dimostrare che ogni linguaggio riconosciuto da una TM spreca-nastro è Turing-riconoscibile, mostriamo come convertire una macchina di Turing spreca-nastro M in una TM deterministica a nastro singolo S equivalente.

S = "Su input w:

1. Inizialmente S mette il suo nastro in un formato che gli consente di implementare l'operazione di salto oltre la parte non vuota del nastro, usando il simbolo speciale # per marcare l'inizio e la fine della porzione usata del nastro. Se w è l'input della TM, la configurazione iniziale del nastro è $\#w\#$.
2. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, L)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a sinistra. Se lo spostamento a sinistra porta la testina sopra il # che marca l'inizio del nastro, S si muove immediatamente di una cella a destra, lasciando inalterato il #. La simulazione continua con la testina in corrispondenza del simbolo subito dopo il #.
3. La simulazione delle mosse del tipo $\delta(q, a) = (r, b, R)$ procede come nella TM standard: S scrive b sul nastro e muove la testina di una cella a destra. Se lo spostamento a destra porta la testina sopra il # che marca la fine del nastro, S scrive un blank al posto del #, e scrive un # nella cella immediatamente più a destra. La simulazione continua con la testina in corrispondenza del blank.
4. Per simulare una mossa del tipo $\delta(q, a) = (r, b, J)$ la TM S scrive b nella cella corrente, e poi sposta la testina a destra fino ad arrivare in corrispondenza del # che marca la fine del nastro. A questo punto si sposta a sinistra finché non trova un simbolo diverso dal blank. Marca con un pallino il primo simbolo non blank che trova, poi si sposta di una cella a destra e la marca con il pallino. Continua procedendo a zig-zag, marcando via via una cella all'inizio e una alla fine della sequenza di pallini. Quando la prossima cella da marcare è il # all'inizio del nastro, la TM non la marca e inizia a spostarsi a destra, scorrendo il nastro per togliere tutti i pallini, e riprendere la simulazione con la testina in corrispondenza dell'ultima cella marcata. In questa ultima fase, se una delle celle marcate è il # alla fine del nastro, allora la TM lo sostituisce con un blank e scrive un # immediatamente a destra dell'ultima cella marcata prima di continuare con la simulazione.
5. 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 punto 2."

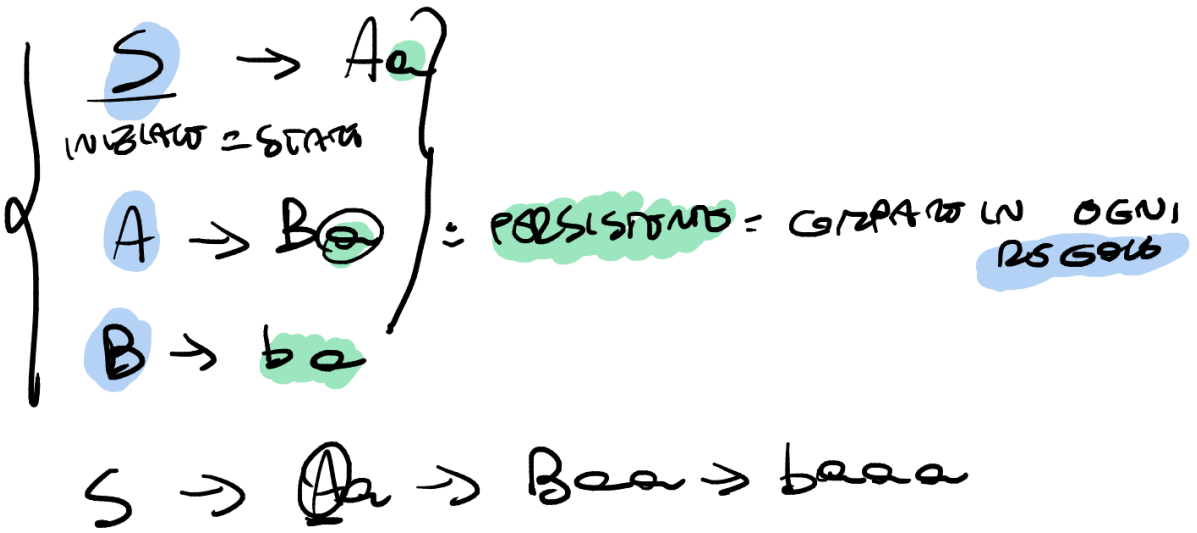
Esercizio 2 - Decidibile

2. (12 punti) Una variabile A in una grammatica context-free G è *persistente* se compare in ogni derivazione di ogni stringa w in $L(G)$. Data una grammatica context-free G e una variabile A, considera il problema di verificare se A è persistente.

(a) Formula questo problema come un linguaggio $PERSISTENT_{\text{CFG}}$.

(b) Dimostra che $PERSISTENT_{\text{CFG}}$ è decidibile.

Logica delle grammatiche context-free e in questo caso della persistenza.



2.1 - Formula il problema come linguaggio

PERSISTENT_CFG = {<T, w> | T è una macchina di Turing e L(w) è persistente}

Persistente = L = G, A = G è una CFG persistente, A variabile di G

ALTRIMENTI

PERSISTENT_CFG = { <G, A> | G è una CFG, A è una variabile persistente}

2.2 - Dimostra L decidibile

CFG = Grammatiche context-free
DFA = Automi a stati finiti deterministici
NFA = Automi a stati finiti non-deterministici

Una macchina di Turing riconosce sempre queste.

A_CFG / A_DFA / A_NFA = Macchine di Turing che te le riconosce

MA puoi usare il fatto che se so che i problemi "notevoli" sono decidibili, allora anche il mio lo diventa.

(Riferimento: Slide - Decidibilità)

Dimostrazione

Usiamo una macchina di Turing M che decide PERSISTENT_CFG (sapendo che A_CFG è un problema decidibile, allora esiste sicuramente una macchina di Turing che [riconosce = termina con accettazione] G)

M = Su input <G, A>, dove G è una CFG e A è una variabile:

Digressione per capire bene per le dimostrazioni delle CFG:

Forma Normale: Siamo sicuri che la grammatica si deriva sempre in maniera non ambigua = Potresti avere derivazioni diverse

Forma Normale di Chomsky:
A -> BC, A -> a
Tradotta in italiano: Trasformiamo le regole sempre in questa forma per fare in modo che a prescindere la mia G sia non ambigua

La logica è questa: nelle soluzioni vedi sempre o quasi (almeno anche nella prima parte) che le CFG sono in forma Chomsky = saranno sempre derivabili nello stesso modo.

Tornando alla dimostrazione

Convertiamo G in forma normale di Chomsky e:

- Verifica che A appartenga alle variabili di G
- Se ogni regola produce sempre "A -> a" allora se la grammatica termina allora anche M esegue su input G
- Se M accetta, allora G termina, altrimenti rifiuta

La parte finale è sempre un se e solo se:

- Se (G, A) sta in PERSISTENT ALLORA T(M) decide G

G è un a grammatica persistente, allora essendo in CNF = Forma Normale di Chomsky, esiste sicuramente una TM che la decide. In questo caso, quindi, esiste sempre "a" in tutte le regole, dato che tutte le regole intermedie producono sempre delle derivazioni non ambigue (=a una certa termini sempre).

Se G termina allora M termina con stato di accettazione, altrimenti no.

- Se T(M) decide G, allora PERSISTENT è una grammatica persistente.

Esiste sempre almeno una derivazione dove A compare come regola. In questo caso, essendo in FNC allora otteniamo sempre una grammatica in cui compare anche A. Se non dovesse comparire, allora la TM rifiuta (perché non c'è sempre A -> a)

(a) $PERSISTENT_{\text{CFG}} = \{\langle G, A\rangle \mid G \text{ è una CFG, } A \text{ è una variabile persistente}\}$

(b) La seguente macchina N usa la Turing machine M che decide E_{CFG} per decidere $PERSISTENT_{\text{CFG}}$:

N = "su input $\langle G, A\rangle$, dove G è una CFG e A una variabile:

1. Verifica che A appartenga alle variabili di G. In caso negativo, rifiuta.
2. Costruisci una CFG G' eliminando tutte le regole dove compare A dalla grammatica G.
3. Esegui M su input $\langle G'\rangle$, e ritorna lo stesso risultato di M."

Mostriamo che N è un decisore dimostrando che termina sempre e che ritorna il risultato corretto. Verificare che una variabile appartenga alle variabili di G è una operazione che si può implementare scorrendo la codifica di G per controllare se A compare nella codifica. Il secondo passo si può implementare copiando la codifica di G senza riportare le regole dove compare A. Di conseguenza, il primo ed il secondo step terminano sempre. Anche il terzo step termina sempre perché sappiamo che E_{CFG} è un linguaggio decidibile. Quindi N termina sempre la computazione.
Vediamo ora che N dà la risposta corretta:

- Se $\langle G, A\rangle \in PERSISTENT_{\text{CFG}}$ allora A è una variabile persistente, quindi compare in ogni derivazione di ogni stringa $w \in L(G)$. Se la eliminiamo dalla grammatica, eliminando tutte le regole dove compare A, allora otteniamo una grammatica G' dove non esistono derivazioni che permettano di derivare una stringa di soli simboli terminali, e di conseguenza G' ha linguaggio vuoto. Quindi $\langle G'\rangle \in E_{\text{CFG}}$, e l'esecuzione di M terminerà con accettazione. N ritorna lo stesso risultato di M, quindi accetta.
- Viceversa, se $\langle G, A\rangle \in PERSISTENT_{\text{CFG}}$ allora A non è una variabile persistente, quindi esiste almeno una derivazione di una parola $w \in L(G)$ dove A non compare. Se eliminiamo A dalla grammatica, eliminando tutte le regole dove compare, allora otteniamo una grammatica G' che può derivare w, e di conseguenza G' ha linguaggio vuoto. Quindi $\langle G'\rangle \notin E_{\text{CFG}}$, e l'esecuzione di M terminerà con rifiuto. N ritorna lo stesso risultato di M, quindi rifiuta.

Esercizio 3 - Indecidibile

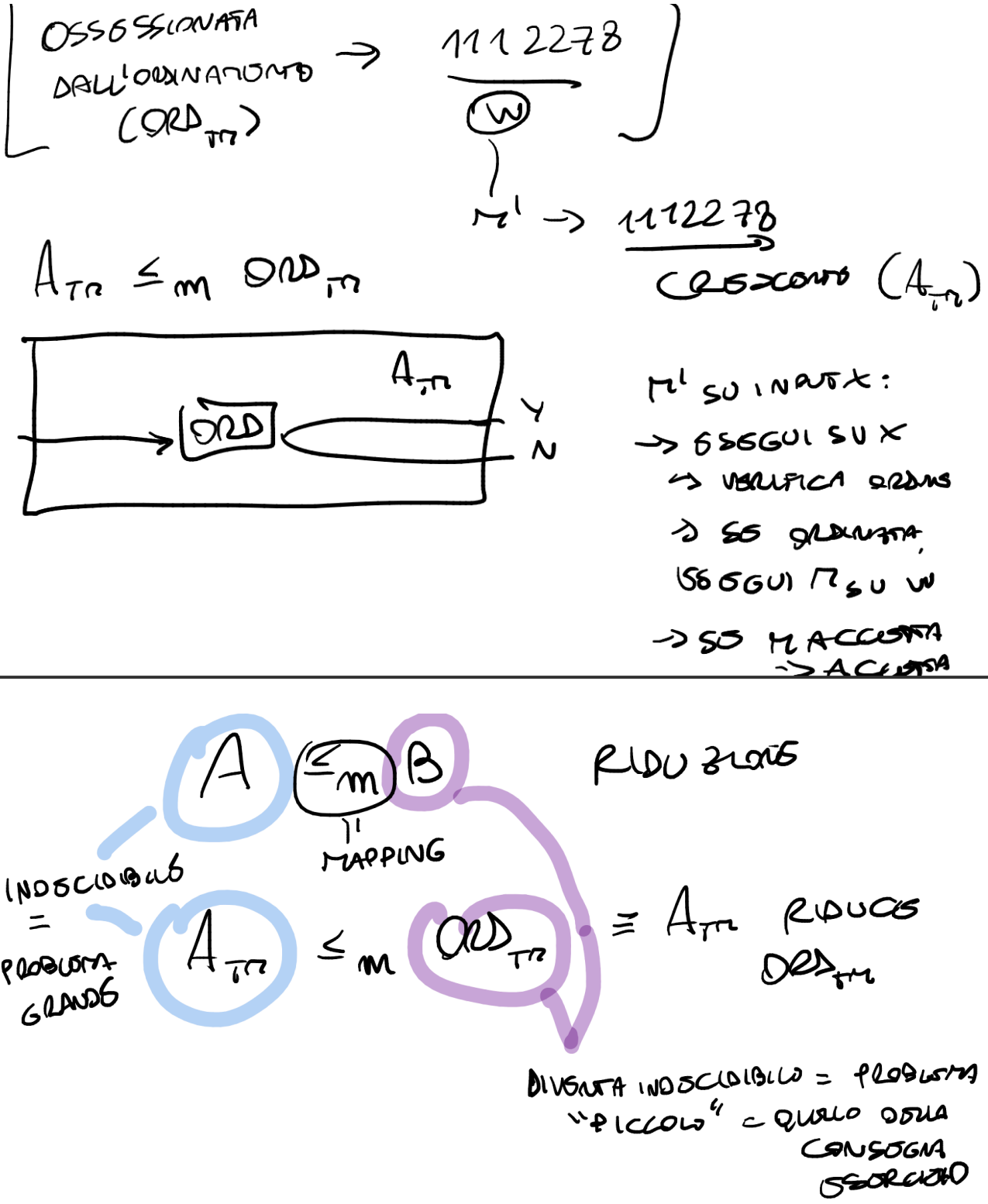
3. (12 punti) Considera le stringhe sull'alfabeto $\Sigma = \{1, 2, \ldots, 9\}$. Una stringa w di lunghezza n su Σ si dice *ordinata* se $w = w_1 w_2 \ldots w_n$ e tutti i caratteri $w_1, w_2, \ldots, w_n \in \Sigma$ sono tali che $w_1 \leq w_2 \leq \cdots \leq w_n$. Ad esempio, la stringa 1112778 è ordinata, ma le stringhe 5531 e 44427 non lo sono (la stringa vuota viene considerata ordinata). Diciamo che una Turing machine è *ossessionata dall'ordinamento* se ogni stringa che accetta è ordinata (ma non è necessario che accetti tutte queste stringhe). Considera il problema di determinare se una TM con alfabeto $\Sigma = \{1, 2, \ldots, 9\}$ è ossessionata dall'ordinamento.

(a) Formula questo problema come un linguaggio SO_{TM}.

(b) Dimostra che il linguaggio SO_{TM} è indecidibile.

Ossessionata dall'ordinamento: Le stringhe hanno i caratteri SEMPRE ordinati in senso crescente.

> Es. 1112778 = Ordinata
> 5531 oppure 44417: NON ordinate



3.1 Formula L come linguaggio

ORD_TM = M è una TM che accetta solo parole ordinate

3.2 Dimostra L indecidibile

$A_{TM} \leq_m ORD_{TM}$

F = Su input <M, w> dove M è una TM, w è una stringa:

- Costruisci M' su input "x"

    - Verifica se "x" è un input ordinato in senso crescente
    - Se x è ordinato, esegui M su input w
    - Se M accetta, allora accetta, altrimenti rifiuta oppure va in loop
- Ritorna <M'>

$x \in A \iff f(x) \in B$

Pattern di Esempio 1 (da A_TM):

```
F = Funzione di riduzione = Su input ⟨M,w⟩
M' = "Su input x: (ugualmente: "Simula M' su x")
    1. [Ignora x / Usa x secondo necessità]
    2. Simula M su w
    3. Se M accetta w → [COMPORTAMENTO CHE SODDISFA P]
    4. Se M rifiuta w → [COMPORTAMENTO CHE NON SODDISFA P]"
```

Struttura della dimostrazione per indecidibilità

1. Definizione e setup

- Enunciare chiaramente il linguaggio L di cui si vuole dimostrare l'indecidibilità
- Specificare l'alfabeto e la codifica delle istanze
- Assumere per assurdo che L sia decidibile (esista una MT M che decide L)

2. Scelta del problema di riduzione

- Identificare un problema già noto come indecidibile (tipicamente HALT, A_TM, E_TM, o EQ_TM)
- Giustificare brevemente perché tale problema è indecidibile

3. Costruzione della riduzione

- Definire la funzione di riduzione f: Σ → *Σ* che trasforma istanze del problema indecidibile in istanze di L
- Dimostrare che f è calcolabile (descrivere l'algoritmo che la computa)
- Verificare la correttezza della riduzione: x ∈ PROBLEMA_INDECIDIBILE ⇔ f(x) ∈ L

4. Derivazione della contraddizione

- Mostrare che se esistesse M (decisore per L), allora si potrebbe decidere il problema indecidibile

- Costruire l'algoritmo: su input x, calcolare f(x) e usare M per determinare se f(x) ∈ L
- Concludere che questo contraddirebbe l'indecidibilità del problema di partenza

5. Conclusione

- L'assunzione iniziale era falsa, quindi L è indecidibile

La chiave è la correttezza della riduzione al punto 3: ogni dettaglio deve essere verificato rigorosamente per entrambe le direzioni dell'equivalenza.

Soluzione.

(a) $SO_{\text{TM}} = \{\langle M \rangle \mid M \text{ è una TM con alfabeto } \Sigma = \{1, 2, \ldots, 9\} \text{ che accetta solo parole ordinate}\}$

(b) La seguente macchina F calcola una riduzione $\overline{A_{TM}} \leq_m UA$:

$F =$ "su input $\langle M, w \rangle$, dove M è una TM e w una stringa:

1. Costruisci la seguente macchina M':

   $M' =$ "Su input x:
   1. Se $x = 111$, accetta.
   2. Se $x = 211$, esegue M su input w e ritorna lo stesso risultato di M.
   3. In tutti gli altri casi, rifiuta.

2. Ritorna $\langle M' \rangle$."

Mostriamo che F calcola una funzione di riduzione da $\overline{A_{TM}}$ a SO_{TM}, cioè una funzione tale che

$$\langle M, w \rangle \in \overline{A_{TM}} \text{ se e solo se } \langle M' \rangle \in SO_{\text{TM}}.$$

- Se $\langle M, w \rangle \in \overline{A_{TM}}$ allora la macchina M rifiuta o va in loop su w. In questo caso la macchina M' accetta la parola ordinata 111 e rifiuta tutte le altre, quindi è ossessionata dall'ordinamento e di conseguenza $\langle M' \rangle \in SO_{\text{TM}}$.
- Viceversa, se $\langle M, w \rangle \notin \overline{A_{TM}}$ allora la macchina M accetta w. Di conseguenza, la macchina M' accetta sia la parola ordinata 111 che la parola non ordinata 211, quindi non è ossessionata dall'ordinamento. Di conseguenza $\langle M' \rangle \notin SO_{\text{TM}}$.