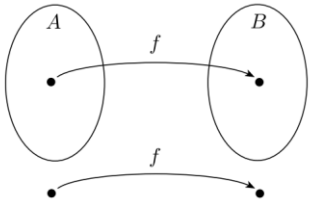
- **P** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing deterministica in tempo $O(|x|^k)$
- **NP** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere verificata da un verificatore in tempo $O(|x|^k)$.
- **Equivalente:** è la classe dei linguaggi in cui l'appartenenza di una stringa $x \in \Sigma^*$ al linguaggio può essere decisa da una macchina di Turing nondeterministica in tempo $O(|x|^k)$.
- **coNP** è la classe dei linguaggi tali che il loro complementare è in *NP*

Definition

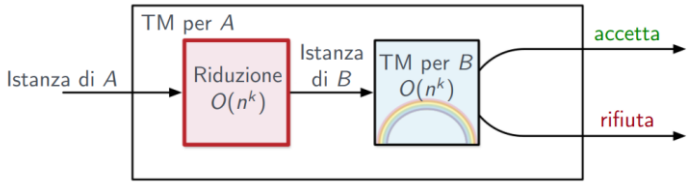
Un linguaggio A è riducibile in tempo polinomiale al linguaggio B ($A \leq_P B$), se esiste una funzione calcolabile in tempo polinomiale $f : \Sigma^* \mapsto \Sigma^*$ tale che

per ogni $w \in \Sigma^* : w \in A$ se e solo se $f(w) \in B$



f è la riduzione polinomiale da A a B

Se $A \leq_P B$, e $B \in P$, allora $A \in P$:



A ***Boolean formula*** is an expression involving Boolean variables and operations. For example,

$$\phi = (\overline{x} \wedge y) \vee (x \wedge \overline{z})$$

is a Boolean formula. A Boolean formula is ***satisfiable*** if some assignment of 0s and 1s to the variables makes the formula evaluate to 1. The preceding formula is satisfiable because the assignment $x = 0$, $y = 1$, and $z = 0$ makes ϕ evaluate to 1. We say the assignment *satisfies* ϕ. The ***satisfiability problem*** is to test whether a Boolean formula is satisfiable. Let

$$SAT = \{\langle\phi\rangle \mid \phi \text{ is a satisfiable Boolean formula}\}.$$

THEOREM 7.27

form. A ***literal*** is a Boolean variable or a negated Boolean variable, as in x or $\overline{x}$. A ***clause*** is several literals connected with $\vee$s, as in $(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4)$. A Boolean formula is in ***conjunctive normal form***, called a ***cnf-formula***, if it comprises several clauses connected with $\wedge$s, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3} \vee x_4) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6}).$$

It is a ***3cnf-formula*** if all the clauses have three literals, as in

$$(x_1 \vee \overline{x_2} \vee \overline{x_3}) \wedge (x_3 \vee \overline{x_5} \vee x_6) \wedge (x_3 \vee \overline{x_6} \vee x_4) \wedge (x_4 \vee x_5 \vee x_6).$$

1. Trovare un certificato di appartenenza:
   $\Rightarrow$ l'assegnamento di verità alle variabili $a, b, c, \ldots$
2. Trovare un algoritmo polinomiale per verificare il certificato.

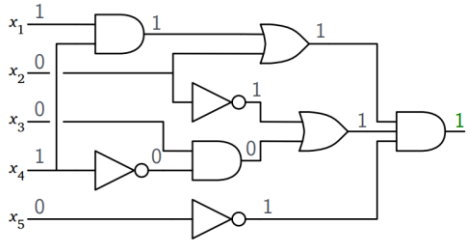
Theorem (Cook e Levin, 1973)

*L'esistenza di una macchina di Turing polinomiale per risolvere **SAT** implica che $P = NP$.*

SAT è Re tra i problemi in NP:

- Ogni problema NP può essere trasformato in una istanza di **SAT** in tempo polinomiale
- **SAT** può essere usato per risolvere tutti i problemi NP
- chi scopre un algoritmo polinomiale per **SAT** sa risolvere tutti i problemi NP in tempo polinomiale!

Certificato: $x_1 = 1, x_2 = 0, x_3 = 0, x_4 = 1, x_5 = 0$

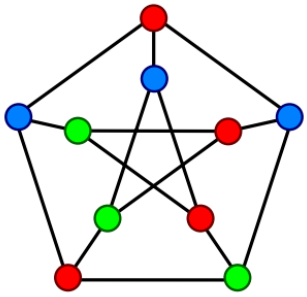


5. Calcola l'output dell'intero circuito: se $= 1$, SI, altrimenti NO

- Un problema è NP-hard se l'esistenza di un algoritmo polinomiale per risolverlo implica l'esistenza di un algoritmo polinomiale per ogni problema in NP.
- Se siamo in grado di risolvere un problema NP-hard in modo efficiente, allora possiamo risolvere in modo efficiente ogni problema di cui possiamo verificare facilmente una soluzione, usando la soluzione del problema NP-hard come sottoprocedura.
- Un problema è NP-completo se è sia NP-hard che appartenente alla classe NP (o "NP-easy").
  - Esempio: **CircuitSAT**!

Esercizio (A)

"Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici collegati da un arco condivida lo stesso colore. La figura seguente mostra un esempio di colorazione di un grafo con 10 vertici che usa 3 colori (rosso, verde, blu).



Chiamiamo 3-COLOR il problema di trovare, se esiste, una colorazione di un grafo che usa 3 colori diversi.

Dimostrate che 3-COLOR è un problema in NP nel modo seguente:

1. definite com'è fatto un certificato per 3-COLOR
2. definite un verificatore polinomiale per 3-COLOR

Per dimostrare che 3-Color è un problema in NP, dobbiamo:

1. Definire cos'è un certificato per 3-Color

2. Definire un verificatore polinomiale per 3-Color

Un certificato per 3-Color è semplicemente un'assegnazione di 3 colori ai vertici del grafo dato, in modo che vertici adiacenti abbiano colori diversi. La dimensione del certificato è O(n), dove n è il numero di vertici, poiché specifica un colore per ogni vertice.

Un verificatore polinomiale per 3-Color, dato un grafo G e un certificato (assegnazione di colori), fa quanto segue:

1. Controlla che il certificato usi solo 3 colori. Questo richiede tempo O(n).

2. Per ogni coppia di vertici adiacenti in G, controlla che abbiano colori diversi nel certificato. Poiché G può avere al massimo O(n^2) coppie di vertici adiacenti, questo passaggio richiede tempo O(n^2).

Se entrambi i controlli vanno a buon fine, il verificatore accetta, altrimenti rifiuta. Il tempo totale è O(n^2), quindi polinomiale nella dimensione dell'input.

Esercizio (B)

Considerate il seguente problema, che chiameremo SUBSETSUM:
dato un insieme di numeri interi S ed un valore obiettivo t, stabilire se esiste un sottoinsieme $S' \subseteq S$ tale che la somma dei numeri in S' è uguale a t.

Esempio: se $S = \{4, 11, 16, 21, 27\}$ e $t = 25$, allora il sottoinsieme $S' = \{4, 21\}$ è una soluzione di SUBSETSUM perché $4 + 21 = 25$.

Dimostrate che SUBSETSUM è un problema in NP nel modo seguente:

1. definite com'è fatto un certificato per SUBSETSUM
2. definite un verificatore polinomiale per SUBSETSUM

Per dimostrare che SubsetSum è un problema in NP, dobbiamo:

1. Definire cos'è un certificato per SubsetSum

2. Definire un verificatore polinomiale per SubsetSum

Un certificato per SubsetSum è un sottoinsieme S' dell'insieme dato S tale che la somma dei numeri in S' sia uguale al target t. La dimensione del certificato è O(n), dove n è la dimensione di S, poiché nel caso peggiore S' potrebbe includere tutti gli elementi di S.

Un verificatore polinomiale per SubsetSum, dato S, t, e un certificato S', fa quanto segue:

1. Controlla che S' sia un sottoinsieme di S. Questo può essere fatto in tempo O(n^2) confrontando ogni elemento di S' con ogni elemento di S.

2. Somma tutti i numeri in S' e controlla se la somma è uguale a t. Questo richiede tempo O(n).

Se entrambi i controlli vanno a buon fine, il verificatore accetta, altrimenti rifiuta. Il tempo totale è O(n^2), quindi polinomiale nella dimensione dell'input.

Nell'esempio dato, un certificato sarebbe S' = {4, 21}, e il verificatore controllerebbe:

1. {4, 21} è un sottoinsieme di {4, 11, 16, 21, 27}

2. 4 + 21 = 25, che è uguale al target t = 25

Quindi, il verificatore accetterebbe questo certificato.

- Un problema è NP-hard se l'esistenza di un algoritmo polinomiale per risolverlo implica l'esistenza di un algoritmo polinomiale per ogni problema in NP.
- Se siamo in grado di risolvere un problema NP-hard in modo efficiente, allora possiamo risolvere in modo efficiente ogni problema di cui possiamo verificare facilmente una soluzione, usando la soluzione del problema NP-hard come sottoprocedura.

A language B is ***NP-complete*** if it satisfies two conditions:

1. B is in NP, and
2. every A in NP is polynomial time reducible to B.

Ogni dimostrazione di NP-completezza di compone di due parti:

1. dimostrare che il problema appartiene alla classe NP;
2. dimostrare che il problema è NP-hard.
   - Dimostrare che un problema è in NP vuol dire dimostrare l'esistenza di un verificatore polinomiale.
   - Le tecniche che si usano per dimostrare che un problema è NP-hard sono fondamentalmente diverse.

Per dimostrare che un certo problema è NP-hard si procede tipicamente con una dimostrazione per riduzione polinomiale.

Per dimostrare che un problema B è NP-hard:

1. Scegli un problema A che sai essere NP-hard.
2. Descrivi una riduzione polinomiale da A a B:
   - data un'istanza di A, trasformala in un'istanza di B,
   - con una funzione che opera in tempo polinomiale.
3. Dimostra che la riduzione è corretta:
   - Dimostra che la funzione trasforma istanze "buone" di A in istanze "buone" di B.
   - Dimostra che la funzione trasforma istanze "cattive" di A in istanze "cattive" di B.
     Equivalente: se la tua funzione produce un'istanza "buona" di B, allora era partita da un'istanza "buona" di A.
4. Mostra che la funzione impiega tempo polinomiale.

- Per dimostrare che un certo problema è NP-hard si procede tipicamente con una dimostrazione per riduzione polinomiale

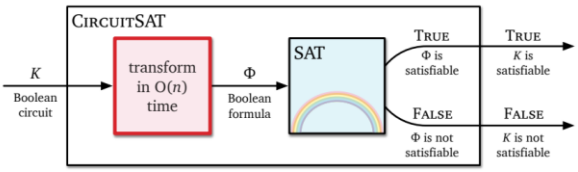
Per dimostrare che B è NP-hard dobbiamo ridurre un problema NP-hard a B.

- Abbiamo bisogno di un problema NP-hard da cui partire: CircuitSAT

3SAT = {⟨φ⟩ | φ è una formula booleana in 3-CNF soddisfacibile}

- **3SAT** è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- **3SAT** è NP-hard:
  dimostrazione per riduzione di **SAT** a **3SAT**

- **SAT** è in NP:
  il certificato è l'assegnamento di verità alle variabili $a, b, c, \ldots$
- **SAT** è NP-hard:
  dimostrazione per riduzione di **CircuitSAT** a **SAT**

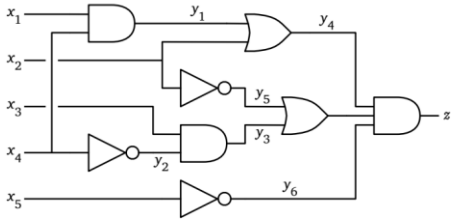


Ora dobbiamo mostrare che il circuito originale K è soddisfacibile se e solo se la formula risultante Φ è soddisfacibile.
Dimostriamo questa affermazione in due passaggi:

- $\Rightarrow$ Dato un insieme di input che rende vero il circuito K, possiamo ottenere i valori di verità per le variabili nella formula Φ calcolando l'output di ogni porta logica di K.
- $\Leftarrow$ Dati i valori di verità delle variabili nella formula Φ, possiamo ottenere gli input del circuito semplicemente ignorando le variabili delle porte logiche interne y_i e la variabile di uscita z.

L'intera trasformazione da circuito a formula può essere eseguita in tempo lineare. Inoltre, la dimensione della formula risultante cresce di un fattore costante rispetto a qualsiasi ragionevole rappresentazione del circuito.

1. Dare un nome agli output delle porte logiche:



2. Scrivere le espressioni booleane per ogni porta logica e metterle in and logico, aggiungendo "AND z" alla fine:

$$(y_1 = x_1 \wedge x_4) \wedge (y_2 = \overline{x_4}) \wedge (y_3 = x_3 \wedge y_2) \wedge (y_4 = y_1 \vee x_2) \wedge (y_5 = \overline{x_2}) \wedge (y_6 = \overline{x_5}) \wedge (y_7 = y_3 \vee y_5) \wedge (z = y_4 \wedge y_7 \wedge y_6) \wedge z$$

Riduzione di CircuitSAT a 3SAT

1. Fai in modo che ogni porta logica abbia al massimo due input
2. Trasforma il circuito in una formula booleana come fatto per SAT
3. Trasforma la formula in CNF usando le regole seguenti:

$$a = b \wedge c \mapsto (a \vee \overline{b} \vee \overline{c}) \wedge (\overline{a} \vee b) \wedge (\overline{a} \vee c)$$
$$a = b \vee c \mapsto (\overline{a} \vee b \vee c) \wedge (a \vee \overline{b}) \wedge (a \vee \overline{c})$$
$$a = \overline{b} \mapsto (a \vee b) \wedge (\overline{a} \vee \overline{b})$$

4. Trasforma la formula in 3CNF aggiungendo variabili alle clausole con meno di tre letterali:

$$a \vee b \mapsto (a \vee b \vee x) \wedge (a \vee b \vee \overline{x})$$
$$a \mapsto (a \vee x \vee y) \wedge (a \vee \overline{x} \vee y) \wedge (a \vee x \vee \overline{y}) \wedge (a \vee \overline{x} \vee \overline{y})$$

If G is an undirected graph, a ***vertex cover*** of G is a subset of the nodes where every edge of G touches one of those nodes. The vertex cover problem asks whether a graph contains a vertex cover of a specified size:

$$\mathit{VERTEX\text{-}COVER} = \{\langle G, k\rangle \mid G \text{ is an undirected graph that has a } k\text{-node vertex cover}\}.$$

THEOREM 7.44

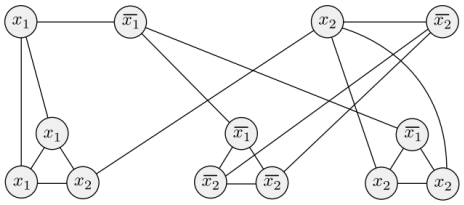
VERTEX-COVER is NP-complete.

PROOF Here are the details of a reduction from *3SAT* to *VERTEX-COVER* that operates in polynomial time. The reduction maps a Boolean formula ϕ to a graph G and a value k. For each variable x in ϕ, we produce an edge connecting two nodes. We label the two nodes in this gadget x and $\overline{x}$. Setting x to be TRUE corresponds to selecting the node labeled x for the vertex cover, whereas FALSE corresponds to the node labeled $\overline{x}$.

To prove that this reduction works, we need to show that ϕ is satisfiable if and only if G has a vertex cover with k nodes. We start with a satisfying assignment. We first put the nodes of the variable gadgets that correspond to the true literals in the assignment into the vertex cover. Then, we select one true literal in every clause and put the remaining two nodes from every clause gadget into the vertex cover. Now we have a total of k nodes. They cover all edges because every variable gadget edge is clearly covered, all three edges within every clause gadget are covered, and all edges between variable and clause gadgets are covered. Hence G has a vertex cover with k nodes.

The gadgets for the clauses are a bit more complex. Each clause gadget is a triple of nodes that are labeled with the three literals of the clause. These three nodes are connected to each other and to the nodes in the variable gadgets that have the identical labels. Thus, the total number of nodes that appear in G is $2m + 3l$, where ϕ has m variables and l clauses. Let k be $m + 2l$.

For example, if $\phi = (x_1 \vee x_1 \vee x_2) \wedge (\overline{x_1} \vee \overline{x_2} \vee \overline{x_2}) \wedge (\overline{x_1} \vee x_2 \vee x_2)$, the reduction produces $\langle G, k\rangle$ from ϕ, where $k = 8$ and G takes the form shown in the following figure.



3. (12 punti) Una 3-colorazione di un grafo non orientato G è una funzione che assegna a ciascun vertice di G un "colore" preso dall'insieme $\{1, 2, 3\}$, in modo tale che per qualsiasi arco $\{u, v\}$ i colori associati ai vertici u e v sono diversi. Una 3-colorazione è *sbilanciata* se esiste un colore che colora più di metà dei vertici del grafo.

UNBALANCED-3-COLOR è il problema di trovare una 3-colorazione sbilanciata:

$$\text{UNBALANCED-3-COLOR} = \{\langle G \rangle \mid G \text{ è un grafo che ammette una 3-colorazione sbilanciata}\}$$

(a) Dimostra che UNBALANCED-3-COLOR è un problema NP
(b) Dimostra che UNBALANCED-3-COLOR è NP-hard, usando 3-COLOR come problema NP-hard di riferimento.

(a) Il certificato è un vettore c dove ogni elemento $c[i]$ è il colore assegnato al vertice i. Il seguente algoritmo è un verificatore V per UNBALANCED-3-COLOR:
$V =$ "Su input $\langle \langle G \rangle, c \rangle$:
1. Controlla se c è un vettore di n elementi, dove n è il numero di vertici di G.
2. Controlla se $c[i] \in \{1, 2, 3\}$ per ogni i.
3. Controlla se per ogni arco (i, j) di G, $c[i] \neq c[j]$.
4. Controlla se esiste un colore che compare in più di metà degli elementi di c.
5. Se tutte le condizioni sono vere, *accetta*, altrimenti *rifiuta*."

Ognuno dei passi dell'algoritmo si può eseguire in tempo polinomiale.

(b) Dimostriamo che UNBALANCED-3-COLOR è NP-Hard per riduzione polinomiale da 3-COLOR a UNBALANCED-3-COLOR. La funzione di riduzione polinomiale f prende in input un grafo $\langle G \rangle$ e produce come output un grafo $\langle G' \rangle$. Se n è il numero di vertici di G, G' è costruito aggiungendo $n + 1$ vertici isolati a G. Un vertice è isolato se non ci sono archi che lo collegano ad altri vertici.
Dimostriamo che la riduzione è corretta:

- Se $\langle G \rangle \in$ 3-COLOR, allora esiste un modo per colorare G con tre colori $1, 2, 3$. Sia n il numero di vertici di G. Posso costruire una colorazione sbilanciata per il grafo G' come segue:
  - i colori dei vertici di G' che appartengono anche a G sono colorati come in G;
  - i vertici isolati sono colorati tutti con il colore 1.

  In questo modo il colore 1 è assegnato ad almeno metà dei vertici di G' e la colorazione è sbilanciata.
- Se $\langle G' \rangle \in$ UNBALANCED-3-COLOR, allora esiste una colorazione sbilanciata di G'. Se elimino da G' i vertici isolati aggiunti dalla riduzione ottengo una 3-colorazione di G.

La funzione di riduzione deve contare i vertici del grafo G e aggiungere $n + 1$ vertici al grafo, operazioni che si possono fare in tempo polinomiale.

1. Una *tag-Turing machine* è una macchina di Turing con un singolo nastro e due testine: una testina può solo leggere, l'altra può solo scrivere. All'inizio della computazione la testina di lettura si trova sopra il primo simbolo dell'input e la testina di scrittura si trova sopra la cella vuota posta immediatamente dopo la stringa di input. Ad ogni transizione, la testina di lettura può spostarsi di una cella a destra o rimanere ferma, mentre la testina di scrittura deve scrivere un simbolo nella cella corrente e spostarsi di una cella a destra. Nessuna delle due testine può spostarsi a sinistra.

Dimostra che che le tag-Turing machine riconoscono la classe dei linguaggi Turing-riconoscibili.

1. Per risolvere l'esercizio dobbiamo dimostrare che *(a)* ogni linguaggio riconosciuto da una tag-Turing machine è Turing-riconoscibile e *(b)* ogni linguaggio Turing-riconoscibile è riconosciuto da una tag-Turing machine.

(a) Mostriamo come convertire una tag-Turing machine M in una TM deterministica a nastro singolo S equivalente. S simula il comportamento di M tenendo traccia delle posizioni delle due testine marcando la cella dove si trova la testina di lettura con un pallino sopra il simbolo, e marcando la cella la cella dove si trova la testina di scrittura con un pallino sotto il simbolo. Per simulare il comportamento di M la TM S scorre il nastro e aggiorna le posizioni delle testine ed il contenuto delle celle come indicato dalla funzione di transizione di M.

$S =$ "Su input $w = w_1 w_2 \dots w_n$:

1. Scrivi un pallino sopra il primo simbolo di w e un pallino sotto la prima cella vuota dopo l'input, in modo che il nastro contenga

$$\overset{\bullet}{w}_1 w_2 \dots w_n \underset{\bullet}{\sqcup}.$$

2. Per simulare una transizione, S scorre il nastro per trovare la posizione della testina di lettura e determinare il simbolo letto da M. Se la funzione di transizione stabilisce che la testina di lettura deve spostarsi a destra, allora S sposta il pallino nella cella immediatamente a destra, altrimenti lo lascia dov'è. Successivamente S si sposta verso destra finché non trova la cella dove si trova la testina di scrittura, scrive il simbolo stabilito dalla funzione di transizione nella cella e sposta la marcatura nella cella immediatamente a destra.
3. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di M, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di M allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

(b) Mostriamo come convertire una TM deterministica a nastro singolo S in una tag-Turing machine M equivalente. M simula il comportamento di S memorizzando sul nastro una sequenza di configurazioni di S separate dal simbolo #. All'interno di ogni configurazione un pallino marca il simbolo sotto la testina di S. Per simulare il comportamento di S la tag-Turing machine M scorre la configurazione corrente e scrivendo man mano la prossima configurazione sul nastro.

$M =$ "Su input $w = w_1 w_2 \dots w_n$:

1. Scrive il simbolo # subito dopo l'input, seguito dalla configurazione iniziale, in modo che il nastro contenga

$$w_1 w_2 \dots w_n \# \overset{\bullet}{w}_1 w_2 \dots w_n \sqcup,$$

che la testina di lettura si trovi in corrispondenza del $\overset{\bullet}{w}_1$ e quella di lettura in corrispondenza del blank dopo la configurazione iniziale. Imposta lo stato corrente della simulazione st allo stato iniziale di S e memorizza l'ultimo simbolo letto $prec = \#$. L'informazione sui valori di st e $prec$ sono codificate all'interno degli stato di M.
2. Finché il simbolo sotto la testina di lettura non è marcato, scrive il simbolo precedente $prec$ e muove a destra. Aggiorna il valore di $prec$ con il simbolo letto.
3. Quando si trova un simbolo marcato $\overset{\bullet}{a}$ e $\delta(st, a) = (q, b, R)$:
   - aggiorna lo stato della simulazione $st = q$;
   - scrive $prec$ seguito da b, poi muove la testina di lettura a destra;
   - scrive il simbolo sotto la testina marcandolo con un pallino.
4. Quando si trova un simbolo marcato $\overset{\bullet}{a}$ e $\delta(st, a) = (q, b, L)$:
   - aggiorna lo stato della simulazione $st = q$;
   - scrive $\overset{\bullet}{prec}$; se $prec = \#$ scrive $\#\underset{\bullet}{\sqcup}$;
   - scrive b.
5. Copia il resto della configurazione fino al # escluso. Al termine della copia la testina di lettura di trova in corrispondenza della prima cella nella configurazione corrente, e quella di lettura sulla cella vuota dopo la configurazione.
6. Se in qualsiasi momento la simulazione raggiunge lo stato di accettazione di S, allora *accetta*; se la simulazione raggiunge lo stato di rifiuto di M allora *rifiuta*; altrimenti prosegue con la simulazione dal punto 2."

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.

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 $\langle M' \rangle \notin SO_{\text{TM}}$.

5.26 Define a ***two-headed finite automaton*** (2DFA) to be a deterministic finite automaton that has two read-only, bidirectional heads that start at the left-hand end of the input tape and can be independently controlled to move in either direction. The tape of a 2DFA is finite and is just large enough to contain the input plus two additional blank tape cells, one on the left-hand end and one on the right-hand end, that serve as delimiters. A 2DFA accepts its input by entering a special accept state. For example, a 2DFA can recognize the language $\{\mathtt{a}^n\mathtt{b}^n\mathtt{c}^n \mid n \geq 0\}$.

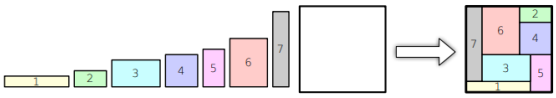
a. Let $A_{\text{2DFA}} = \{\langle M, x \rangle \mid M \text{ is a 2DFA and } M \text{ accepts } x\}$. Show that A_{2DFA} is decidable.

b. Let $E_{\text{2DFA}} = \{\langle M \rangle \mid M \text{ is a 2DFA and } L(M) = \emptyset\}$. Show that E_{2DFA} is not decidable.

1. **Simulazione del 2DFA**:
   - Codifichiamo gli stati del 2DFA, le posizioni delle due testine e il contenuto del nastro.
   - Simuliamo le transizioni del 2DFA passo dopo passo.
   - Poiché il nastro è finito e le testine possono muoversi solo su questo nastro finito, la simulazione terminerà in un numero finito di passi.
2. **Decisione dell'Accettazione**:
   - Se durante la simulazione M entra in uno stato di accettazione, accettiamo $\langle M, x \rangle$.
   - Se M termina senza entrare in uno stato di accettazione, rifiutiamo $\langle M, x \rangle$.
2. **Costruzione della riduzione**:
   - Data un'istanza $\langle M, w \rangle$ di $HALT_{TM}$, costruiremo un 2DFA M' che simula il comportamento di M su w.
   - M' accetta l'input solo se M si arresta su w.
3. **Descrizione di M'**:
   - M' utilizza i due capi di lettura per simulare M su w.
   - Se M si arresta su w, M' entra in uno stato di accettazione.
   - Se M non si arresta su w, M' non accetta nessun input.
4. **Riduzione a E_{2DFA}**:
   - Costruiamo un 2DFA M' tale che:
     M' accetta l'input w se e solo se M si arresta su w.
   - Se M non si arresta su w, allora $L(M') = \emptyset$.
   - Se M si arresta su w, allora $L(M') \neq \emptyset$.

3. Il problema SetPartitioning chiede di stabilire se un insieme di numeri interi S può essere suddiviso in due sottoinsiemi disgiunti S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2. Sappiamo che questo problema è NP-hard.

Il problema RectangleTiling è definito come segue: dato un rettangolo grande e diversi rettangoli più piccoli, determinare se i rettangoli più piccoli possono essere posizionati all'interno del rettangolo grande senza sovrapposizioni e senza lasciare spazi vuoti.



Un'istanza positiva di RectangleTiling.

Dimostra che RectangleTiling è NP-hard, usando SetPartitioning come problema di riferimento.

Dimostriamo che RectangleTiling è NP-Hard per riduzione polinomiale da SetPartitioning. La funzione di riduzione polinomiale prende in input un insieme di interi positivi $S = \{s_1, \ldots, s_n\}$ e costruisce un'istanza di RectangleTiling come segue:

- i rettangoli piccoli hanno altezza 1 e base uguale ai numeri in S moltiplicati per 3: $(3s_1, 1), \ldots, (3s_n, 1)$;
- il rettangolo grande ha altezza 2 e base $\frac{3}{2}N$, dove $N = \sum_{i=1}^{n} s_i$ è la somma dei numeri in S.

Dimostriamo che esiste un modo per suddividere S in due insiemi S_1 e S_2 tali che la somma dei numeri in S_1 è uguale alla somma dei numeri in S_2 se e solo se esiste un tiling corretto:

- Supponiamo esista un modo per suddividere S nei due insiemi S_1 e S_2. Posizioniamo i rettangoli che corrispondono ai numeri in S_1 in una fila orizzontale, ed i rettangoli che corrispondono ad S_2 in un'altra fila orizzontale. Le due file hanno altezza 1 e base $\frac{3}{2}N$, quindi formano un tiling corretto.
- Supponiamo che esista un modo per disporre i rettangoli piccoli all'interno del rettangolo grande senza sovrapposizioni né spazi vuoti. Moltiplicare le base dei rettangoli per 3 serve ad impedire che un rettangolo piccolo possa essere disposto in verticale all'interno del rettangolo grande. Quindi il tiling valido è composto da due file di rettangoli disposti in orizzontale. Mettiamo i numeri corrispondenti ai rettangoli in una fila in S_1 e quelli corrispondenti all'altra fila in S_2. La somma dei numeri in S_1 ed S_2 è pari ad $N/2$, e quindi rappresenta una soluzione per SetPartitioning.

Per costruire l'istanza di RectangleTiling basta scorrere una volta l'insieme S, con un costo polinomiale.