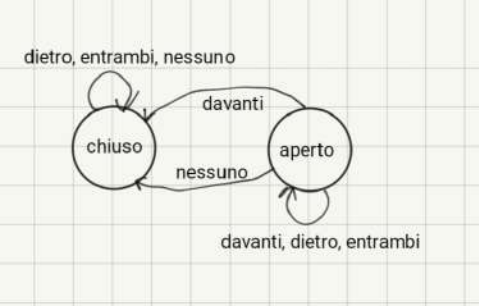
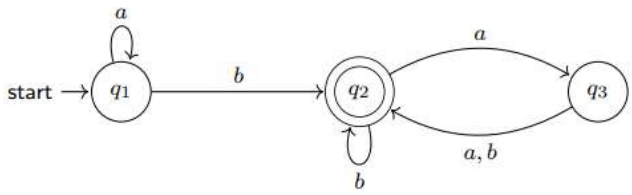
Automi Finiti Deterministici (DFA)

↗ Area	💻 Progetti personali
↗ Project	📐 Elementi di Teoria della Computazione
Day	Tuesday
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⭐

I computer moderni risultano essere modelli troppo complicati per permetterci di costruire un modello matematico efficace. Per questo, useremo un modello ideale, che per semplicità chiameremo **modello di computazione**. In questo capitolo ci concentreremo su uno dei modelli più semplici, ossia l'**automa a stati finiti**. Gli automi hanno una quantità molto limitata di memoria. Viene definito, inizialmente, un alfabeto Σ con un insieme finito di simboli. L'automa prende come input una stringa w su Σ e legge i simboli di w da sinistra a destra. Dopo aver letto l'ultimo simbolo, l'automa può accettare o rifiutare la stringa di input w. Potrebbe sembrare un modello infinitamente semplice, ma vediamo un attimo il sistema di controllo per una porta automatica. Partiamo col dire che il sistema può trovarsi in due stati, `aperto` o `chiuso`. Esso rileva quattro possibili input: `davanti`, `dietro`, `entrambi`, `nessuno`. Non sono altro che la posizione della persona. Prima di continuare, ci sono alcune regole da definire. Se la porta è chiusa, si apre solo se c'è una persona davanti. Se è aperta, si chiude solo se non c'è nessuno. Inoltre, dobbiamo supporre che il sistema parta da uno stato di chiuso. Detto questo, possiamo definire il nostro diagramma a stati finiti:



Partendo da `chiuso`, il sistema riceve in input `dietro`, `entrambi`, `nessuno` e `davanti`. Una volta letto `davanti`, passa nello stato `aperto`. Altrimenti, va in loop e resta `chiuso` ad oltranza. Allo stesso tempo, il sistema resta `aperto` se legge in input `davanti`, `dietro` e `entrambi`. Se legge `nessuno`, torna nello stato `chiuso`. Adesso, andiamo a definire un **automa finito deterministico**, **DFA**, in maniera un po' più tecnica:



Guardiamo questo DFA, che per semplicità chiameremo M_1. L'alfabeto su cui è definito è certamente $\Sigma = \{a, b\}$. C'è un insieme di stati, che definiamo con $Q = \{q_1, q_2, q_3\}$. Lo stato inziale è q_1, mentre lo stato finale, indicato con un doppio cerchio, è q_2. Per ogni stato, se si legge un simbolo dell'alfabeto, il diagramma specifica in quale stato si transisce. Se, leggendo l'ultimo simbolo è finale, nel nostro caso q_2, la stringa è accettata, altrimenti viene rifiutata. Vediamo un pratico esempio, verifichiamo se la stringa `abbaa` è accettata:

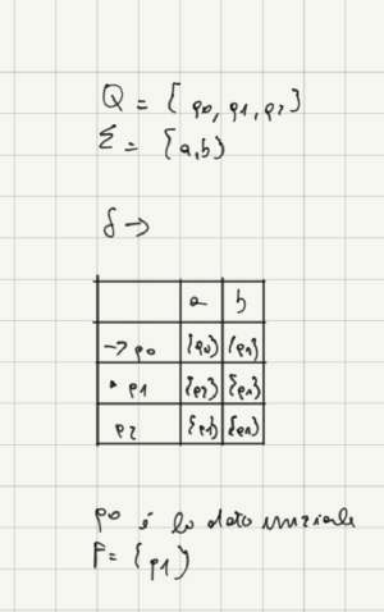
1. Partiamo dallo stato iniziale, ovviamente. Leggendo `a`, restiamo in q_1.
2. Se leggiamo `b`, andiamo in q_2.
3. Leggiamo ancora `b` e restiamo in q_2.
4. Ora leggiamo nuovamente `a` e andiamo in q_3.
5. Infine, leggendo ancora `a`, torniamo in q_2 e, quindi la stringa è accettata.

Ovviamente, una qualsiasi stringa le cui transizioni terminano in uno stato non accettante, essa sarà rifiutata. Nel nostro caso è rifiutata anche la stringa vuota ε, dato che lo stato iniziale non è accettante. Andiamo a dare una definizione un po’ più

tecnica. Un DFA è una **quintupla** $(Q, \Sigma, \delta, q_0, F)$:

- Q è un insieme finito, che indica l'**insieme degli stati**.
- Σ è un insieme finito, che indica l'**alfabeto** su cui è definito l'automa.
- $\delta : Q \times \Sigma \longrightarrow Q$ è la **funzione di transizione**.
- $q_0 \in Q$ è lo **stato iniziale**.
- $F \subseteq Q$ è l'**insieme degli stati accettanti**, o **finali**.

È di particolare importanza la funzione di transizione, poiché va ad indicare per ogni stato e per ogni simbolo di input, in quale stato si transisce. Per descriverla, metteremo una dicitura del tipo $\delta(q_i, a) \in Q$ che indica lo stato in cui si troverà il DFA quando, trovandosi nello stato q_i, legge il simbolo a. Nel nostro caso $\delta(q_i, a) = q_2$. Un automa è deterministico perché esiste, per ogni stato, una ed una sola transizione per ciascun simbolo dell'alfabeto. Per definire formalmente un automa, quindi, basterà rifarci alla quintupla:



Dato un DFA $(Q, \Sigma, \delta, q_0, F)$ e consideriamo una stringa di input $w = w_1 w_2 \dots w_n$ appartenente all'alfabeto. Il DFA accetta la stringa w se esiste una sequenza di stati $r_0, r_1, \dots, r_n$ tale che:

- $r_0 = q_0$;

- $\delta(r_i, w_i + 1) = r_i + 1$ per $i = 0, \ldots, n-1$;
- $r_n \in F$;

Questa dicitura appare complicata, ma possiamo spiegarla in modo semplice. In sostanza stiamo dicendo che il DFA accetta una stringa w se esiste un percorso attraverso gli stati del DFA che rappresenta la sequenza di caratteri nella stringa w. Inoltre, questo percorso deve terminare in uno stato finale del DFA. Chiamando A l'insieme di tutte le stringhe che la macchina accetta, diciamo che A è il **linguaggio della macchina** e scriveremo $L(M) = A$. La dicitura corretta è che il DFA riconosce, o accetta, A. Ogni macchina riconosce sempre e solo un linguaggio. La dicitura corretta è che il DFA **riconosce**, o **accetta**, A. Ogni macchina riconosce sempre e solo un linguaggio. Se la macchina non accetta nessuna stringa, riconosce il **linguaggio vuoto**. Un linguaggio si dice **linguaggio regolare** se esiste un automa finito che lo riconosce. Per esempio, il linguaggio riconosciuto dall'automa precedente è l'insieme di tutte le stringhe che contengono almeno una b e un numero pari di a segue l'ultima b.
Siano A e B due linguaggi. Definiamo le **operazioni regolari** di **unione**, **concatenazione** e **star** (o **kleene star**). Le possiamo sintetizzare benissimo così. Sia Σ = {a, b, c, …, z} l'alfabeto latino di 26 lettere e sia A = {`good`, `bad`} e B = {`boy`, `girl`}:

- unione: $A \cup B$ = {`good`, `bad`, `boy`, `girl`};
- concatenazione: $A \circ B$ = {`goodboy`, `goodgirl`, `badboy`, `badgirl`};
- star: A^* = {`ε`, `good`, `bad`, `goodgood`, `goodbad`, `badgood`, `badbad`, `goodgoodgood`, …};

Una proprietà di particolare importanza è la **chiusura**. Una collezione S di oggetti è **chiusa** per un'operazione regolare f se, applicando f a membri di S, restituisce un oggetto ancora in S. Possiamo introdurre il nostro primo teorema.

- **T.H.** = la classe dei linguaggi regolari è chiusa per l'operazione di unione. Cioè presi due linguaggi regolari L_1 e L_2, allora $L_1 \cup L_2$ è regolare.

- **DIM** = L'idea di base è quella di dimostrare che, se i due linguaggi sono riconosciuti da due DFA, M_1 e M_2, allora ci tocca costruire un nuovo DFA, M_3, che accetti la stringa w se e solo se essa è accettata da M_1 e M_2 contemporaneamente. Possiamo dare una definizione della quintupla di questo $M_3 = (Q_3, \Sigma, \delta_3, q_3, F_3)$:
  - $Q_3 = Q_1 \times Q_2$, quindi il prodotto cartesiano tra l'insieme di stati del primo DFA con quelli del secondo.
  - Σ, non cambia fondamentalmente nulla, l'alfabeto rimane lo stesso.

- δ_3, mettiamo caso di avere una coppia (x, y), che indica uno stato in Q_3, dove x è uno stato di M_1 e y di M_2 . Applichiamo la funzione di transizione δ_1 allo stato x e al simbolo a, ottenendo un nuovo stato x'. Applichiamo la funzione di transizione δ_2 allo stato y e al simbolo a, ottenendo un nuovo stato y'. Il nuovo stato (x', y') è lo stato raggiunto da (x, y) tramite l'input in M_3. In simboli, diremo che $\delta_3((x, y), a) = (\delta_1(x, a), \delta_2(y, a))$.
- Banalmente, q_3 è la coppia (q_1, q_2).
- Analogamente, l'insieme degli stati finali F_3 può essere definito facendo lo stesso ragionamento. Cioè, uno stato in M_3 deve essere finale se e solo se uno dei due stati corrispondenti in M_1 o M_2 è finale. In maniera formale diremo che $F_3 = \{(x, y) \in Q_3 \mid x \in F_1$ o $y \in F_2\}$.

Ma i linguaggi regolari sono chiusi rispetto all'operazione di concatenazione? C'è un teorema che possiamo dimostrare.

- **T.H.** = la classe dei linguaggi regolari è chiusa rispetto all'operazione di concatenazione. Presi due linguaggi regolari L_1 e L_2, allora $L_1 \circ L_2$.

- **DIM** = analogamente come abbiamo fatto con l'unione, dati due DFA M_1 e M_2, potremo costruire un nuovo automa M_3, che accetti la stringa w se e solo se essa può essere divisa in due parti tali che la prima parte è accettata da M_1 e la seconda da M_2. Ma qui, però, abbiamo un problema. Infatti M_3 dovrebbe tener traccia di tutte le possibilità, poiché non sa dove finisce la prima parte e dove comincia la seconda.

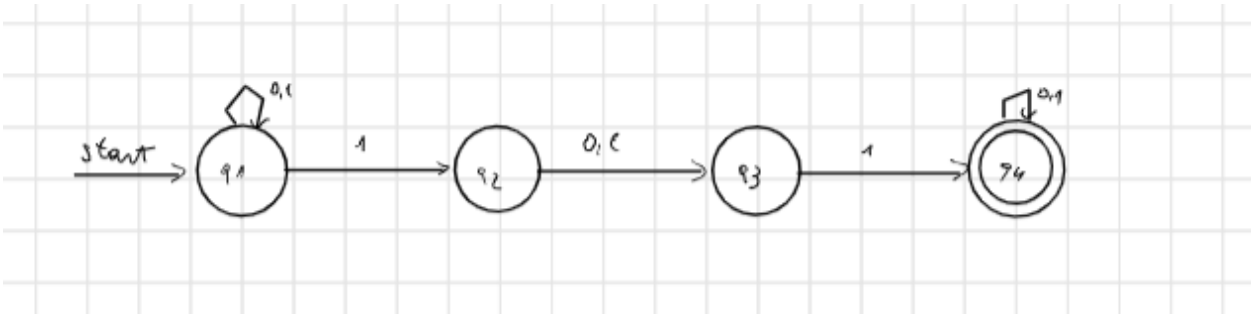
A questo punto, ci tocca lasciare in sospeso l'argomento, poiché dobbiamo introdurre in **non-determinismo**.

Automi Finiti Non Deterministici (NFA)

↗ Area	👩🏾‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

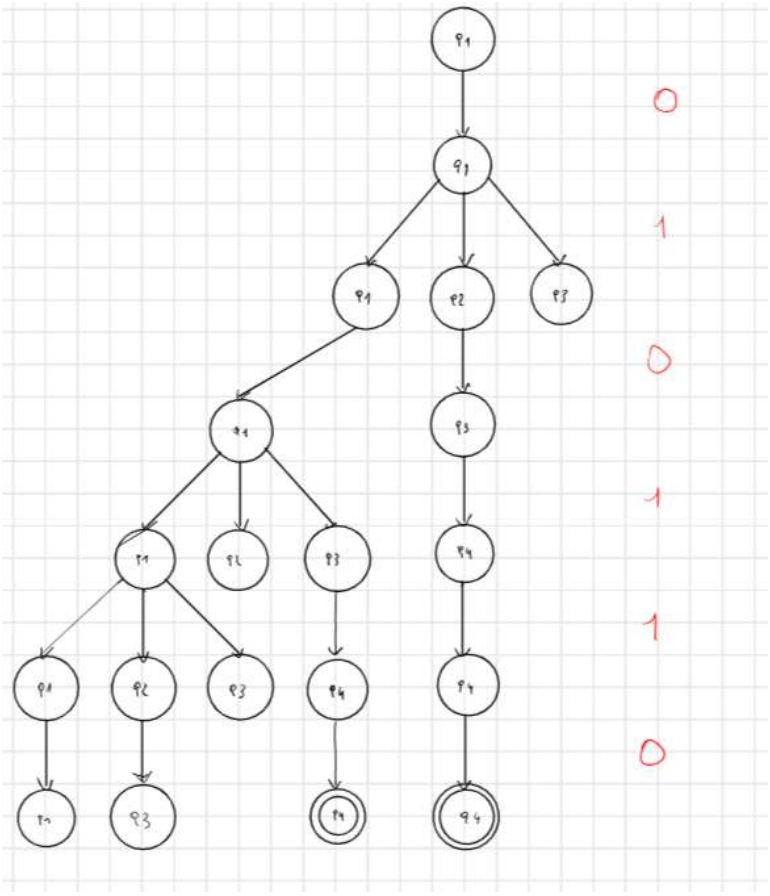
Fino ad ora abbiamo visto come una macchina deterministica, un DFA, si trova in uno stato, legge un certo simbolo e passa univocamente ad uno stato successivo. Ora, introduciamo una generalizzazione del determinismo, cioè il non-determinismo. Un **automa finito non deterministico**, abbreviato **NFA**, permette di avere:

- 0, 1 o anche più archi uscenti per ciascun simbolo dell'alfabeto.
- 0, 1 o anche più archi uscenti con un etichetta ε, che chiameremo **epsilon-transizione**.



Qui possiamo vedere un primo esempio di NFA. Vediamo un po’ nel dettaglio come funziona un NFA. Se un NFA è in uno stato e legge il simbolo a dell'alfabeto. Qui la macchina si divide in più copie di se stessa e segue tutte le possibilità in parallelo. Se una copia di un NFA legge un simbolo che non si trova su alcun arco uscente da quello

stato, allora quella copia muore, insieme a tutto il suo ramo di computazione. Se, almeno una copia, giunge in uno stato accettante, la stringa viene accettata. Se nessuna copia giunge in uno stato accettante, allora non è accettata. Vogliamo vedere un esempio pratico e, per questo, controlliamo se in questo NFA la stringa `010110` è accettata, vedendo il suo **albero di computazione**:



Possiamo dare una definizione un po' più formale dell'NFA. Così come il DFA, un NFA è una quintupla (Q, Σ, δ, q_0 , F):

- Q è un insieme finito, che indica l'**insieme degli stati**.
- Σ è un insieme finito, che indica l'**alfabeto** su cui è definito l'automa.
- $\delta : Q \times \Sigma_\varepsilon \longrightarrow P(Q)$ è la **funziona di transizione**.
- $q_0 \in Q$ è lo **stato iniziale**.
- $F \subseteq Q$ è l'**insieme degli stati accettanti**, o **finali**.

Come possiamo notare, la differenza sostanziale sta nella funzione di transizione. Mentre col DFA, essa è definita solo sull'alfabeto e restituisce uno stato, qui è definita sull'alfabeto, ma ammette anche mosse di tipo ε, oltre a restituire un insieme di stati.

- **N.B.** = ogni DFA può essere trasformato in un NFA.

Per quanto concerne i linguaggi, valgono più o meno le stesse regole dei DFA. La domanda è: dato un NFA, esiste sempre un DFA equivalente? Cioè NFA e DFA riconoscono la stessa classe di linguaggi? Per definire l'equivalenza tra DFA ed NFA, dobbiamo partire da una definizione. Due macchine si dicono equivalenti se riconoscono lo stesso linguaggio. Da questa definizione, possiamo definire il teorema.

- **T.H.** = Ogni NFA N ha un DFA M equivalente, tale che $L(M) = L(N)$.

- **DIM** = partiamo con un'idea di base. Avendo un NFA $N = \{Q_N, \Sigma, \delta_N, q_N, F_N\}$ e un DFA $M = \{Q_M, \Sigma, \delta_M, q_M, F_M\}$ tale che $L(M) = L(N)$. Dobbiamo distinguere due casi. Il primo è quando N non ha ε-archi. Sappiamo che $Q_M = P(Q_N)$ cioè ogni stato di M è un insieme di stati di N. Chiamando $R \in Q_M$, uno degli stati di M, e scegliendo un arbitrario a come simbolo dell'alfabeto, potremo scrivere la funzione di transizione come $\delta'(R, a) = \{q \in Q | q \in \delta(r, a)\}$. Quando M legge un simbolo a nello stato R, mostra dove a porta ogni stato in R. Poiché, da ogni stato, si può andare in un insieme di stati, dobbiamo prendere l'unione di tutti questi insiemi. Lo stato iniziale di M è lo stesso, mentre per gli stati finali, M accetta se uno dei possibili stati in cui N è in quel punto, è uno stato accettante. Il secondo caso è quando N ha ε-archi. Per proseguire con la dimostrazione, chiamiamo $E(R)$ l'insieme di stati che possono essere raggiunti attraverso 0 o più ε-archi. Una volta messa in chiaro questa cosa, andiamo a modificare la funzione di transizione, scrivendo $\delta'(R, a) = \{q \in Q | q \in E(\delta(r, a)\}$. Discorso diverso per lo stato iniziale, infatti dovremo scrivere $E(\{q_0\})$. Per quanto concerne gli stati finali, invece, non cambia niente.

La funzione di transizione di un DFA o un NFA può essere estesa per operare direttamente sulle stringhe e non sui simboli. Dato un DFA M, andiamo a definire una **funzione di transizione estesa** δ^*, in modo che per uno stato q e una stringa w, $\delta^*(q, w)$ sia lo stato raggiungibile quando, partendo da q, si abbia w come input. δ^* può essere definita in maniera diversa. Dato un DFA $M = \{Q_M, \Sigma, \delta_M, q_M, F_M\}$ la funzione di transizione estesa δ^*_M è così definita: $\delta^*_M(q, \varepsilon) = q$, con q un qualsiasi stato di M. Per ogni stringa $w = xa$, dove a è un simbolo dell'alfabeto, avremo $\delta^*_M(q, xa) = \delta_M(\delta^*_M(q, x), a)$. Nel primo punto, stiamo solo valutando nel caso

l'automa riceva una stringa vuota. Nel secondo caso, la nostra x indica zero o più simboli dell'alfabeto ed a è l'ultimo simbolo della stringa. Per ottenere la transizione di stato per la stringa w, si deve calcolare la transizione di stato ottenuta applicando la funzione di transizione del DFA allo stato ottenuto, applicando poi la funzione di transizione estesa alla stringa x e al simbolo a. Dobbiamo fare un discorso diverso nel caso dovessimo trovarci di fronte ad un NFA, poiché l'NFA può trovarsi in più di uno stato contemporaneamente. Valgono più o meno gli stessi discorsi fatti prima, soltanto che la funzione di transizione estesa restituisce l'insieme di tutti gli stati raggiungibili da q. Abbiamo detto che un qualsiasi NFA ha un DFA equivalente. Ovviamente, però, il DFA sarà esponenzialmente meno efficiente in termine di numero di stati. Introduciamo un nuovo teorema.

- **T.H.** = un linguaggio è regolare se e solo se esiste un automa finito non deterministico che lo riconosce.

- **DIM** = la dimostrazione, qui, è abbastanza banale. Infatti sia L un linguaggio regolare. Per definizione, sappiamo che esiste un DFA che lo riconosce. Ma ogni DFA è anche un NFA, quindi automaticamente esiste un NFA che lo riconosce.

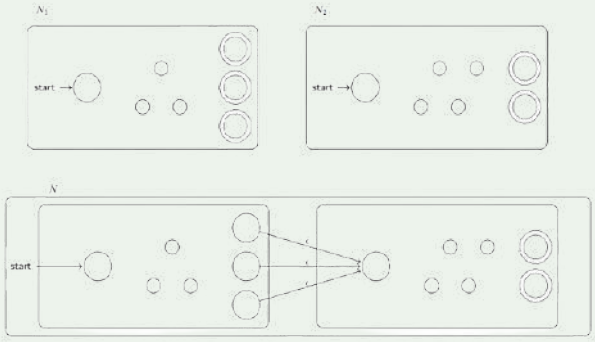
Possiamo passare, adesso, alla dimostrazione che la classe dei linguaggi regolari è chiusa per l'operazione di concatenazione.

- **T.H.** = la classe dei linguaggi regolari è chiusa per l'operazione di concatenazione. Cioè se L_1 e L_2 sono linguaggi regolari, allora lo saranno anche $L_1 \circ L_2$.

- **DIM** = Siano L_1 e L_2 due linguaggi definiti sullo stesso alfabeto. Sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ l'NFA che riconosce L_1. Sia, poi, $N_2 = (Q_2, \Sigma, \delta_2, q_2, F_2)$ l'NFA che riconosce L_2. Per la concatenazione, vogliamo costruire un NFA $N = (Q, \Sigma, \delta, q_1, F_2)$ che riconosca, appunto, $L_1 \circ L_2$. Analizziamo i vari elementi della quintupla:
  - $Q = Q_1 \cup Q_2$, ossia l'unione degli stati dei due NFA.
  - l'alfabeto è lo stesso, non si perdono generalità.
  - la funzione di transizione dobbiamo aprire un capitolo, infatti è definita come segue:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & q \in F_1,\ a \neq \varepsilon \\ \delta_1(q,a) \cup q_2 & q \in F_1,\ a = \varepsilon \\ \delta_1(q,a) & q \in Q_2 \end{cases}$$

- lo stato iniziale è lo stesso di N_1.
- gli stati finali sono gli stessi di N_2.



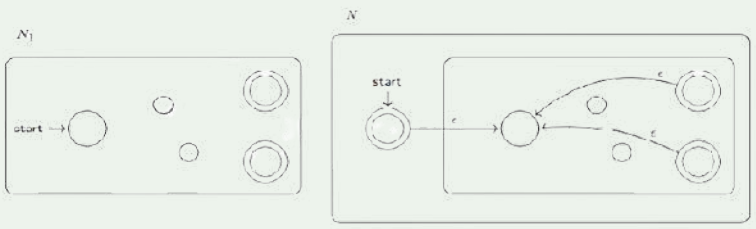
Con gli NFA, possiamo facilmente introdurre un nuovo teorema.

- **T.H.** = la classe dei linguaggi regolari è chiusa per l'operazione di kleene star. Cioè se L è un linguaggio regolare, allora lo è anche L^*.

- **DIM** = Sia L un linguaggio regolare e sia $N_1 = (Q_1, \Sigma, \delta_1, q_1, F_1)$ l'NFA che lo riconosce. Costruiamo, quindi, un NFA $N = (Q, \Sigma, \delta, q_0, F)$ che riconosce L^*. Analizziamo i vari elementi della quintupla:
  - $Q = \{q_0\} \cup Q_1$.
  - l'alfabeto resta lo stesso, non si perdono le generalità.
  - la funzione di transizione viene definita in questo modo:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1,\ q \notin F_1 \\ \delta_1(q,a) & q \in F_1,\ a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_1\} & q \in F_1,\ a = \varepsilon \\ \emptyset & q = q_0,\ a \neq \varepsilon \\ \{q_1\} & q = q_0,\ a = \varepsilon \end{cases}$$

  - lo stato iniziale è q_0.
  - $F = \{q_0\} \cup F_1$.


N_1
start
N
start
ε
ε
ε

Espressioni Regolari

↗ Area	👨🏽‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

Come in aritmetica possiamo usare le operazioni standard per costruire le espressioni più complesse, analogamente possiamo utilizzare le operazioni regolari, quindi unione, concatenazione e star, per poter costruire delle espressioni chiamate espressioni regolari:

- $(0 \cup 1)0^*$;

Mentre il valore di un'espressione aritmetica è un risultato numerico, il valore di un’espressione regolare è un linguaggio. Abbiamo visto che, con DFA ed NFA, possiamo definire linguaggi regolari, ma in maniera analoga possiamo farlo anche con le espressioni regolari. Prima di proseguire, dobbiamo dare una **definizione induttiva** di espressione regolare.

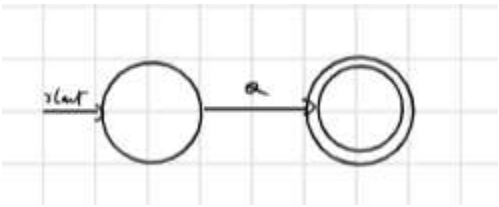
- **DIM** = come con ogni definizione induttiva che si rispetti, partiamo dalla base dell’induzione. Data una qualsiasi espressione regolare R, che soddisfa una certa proprietà, quest’ultima deve essere soddisfatta per i casi base, quindi $R = a$, con $a \in \Sigma$, $R = \varepsilon$ e $R = \emptyset$. Per ipotesi induttiva, supponiamo che la proprietà sia soddisfatta per due arbitrarie espressioni regolari R_1 e R_2. Supposta vera, allora essa sarà vera anche per le seguenti espressioni regolari, quindi $R = R_1 \cup R_2$, $R = R_1 \circ R_2$ e $R = R_1^*$.

Come in aritmetica, anche con le espressioni regolari, ci sono alcuni operatori che hanno la precedenza. Nel nostro caso, si prende in considerazione prima l’operazione di kleene star, poi la concatenazione e, infine, l’unione. Da qui, cominciamo a dimostrare il **teorema di Kleene**.

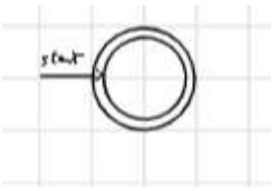
- **T.H.** = un linguaggio è regolare se e solo se esiste un'espressione regolare che lo descrive.
- **DIM** = l'idea alla base è ci viene fornita dalle nostre conoscenze pregresse. Sappiamo che, dato un linguaggio L, esso è regolare se e solo se L è riconosciuto da un NFA, a sua volta se e solo se è riconosciuto da un DFA. Il teorema di Kleene ci permette di dimostrare che, per ogni espressione regolare R, esiste un NFA N, tale che $L(N) = L(R)$, ma anche che per ogni DFA M, esiste un'espressione regolare R, tale che $L(R) = L(M)$.

Quindi, nella prima parte della nostra dimostrazione, supponiamo di avere un'espressione regolare R che descrive un linguaggio A. Dobbiamo trovare un NFA che riconosce A. Basandoci sulla definizione induttiva di espressione regolare, valutiamo i casi base:

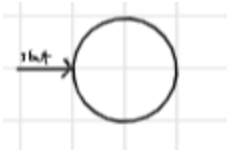
- $R = a$ per qualche $a \in \Sigma$. Allora $L(R) = \{a\}$. Costruiamo, quindi, un NFA che riconosca $L(R)$.



- $R = \varepsilon$. Allora $L(R) = \{\varepsilon\}$. Costruiamo, quindi, un NFA che riconosca $L(R)$.



- $R = \emptyset$. Allora $L(R) = \emptyset$. Costruiamo, quindi, un NFA che riconosca $L(R)$.



Per quanto riguarda il passo induttivo, esso è ancora più immediato. Questo perché abbiamo già dimostrato la chiusura per l'operazione di unione, concatenazione e kleene-star. La seconda parte della dimostrazione prevede una procedura per trasformare i DFA in espressioni regolari equivalenti. Per farlo, si utilizza una generalizzazione dell'NFA, chiamato **automa finito non deterministico generalizzato** (**GNFA**), ossia un NFA che permette archi etichettati con espressioni regolari. Come convertiamo un DFA in un GNFA? Fondamentalmente basterà aggiungere un nuovo stato iniziale, arbitrariamente etichettato con S, che ha un ε-arco verso il vecchio stato iniziale. Poi, concludiamo aggiungendo un unico stato finale, arbitrariamente etichettato con A, che ha uno o più ε-archi entranti provenienti dai vecchi stati accettanti. Il nostro obiettivo è quello di convertire questo GNFA ottenuto in un nuovo GNFA, ma con solo due stati, S ed A. Mettendo caso che il nostro GNFA di partenza abbia almeno k ≥ 2 stati, perché stato iniziale e finale devono essere per forza diversi, dobbiamo trovarci un GNFA equivalente con k-1 stati, iterando la procedura fino ad arrivare a due stati. Ok, sembra complicato, ma vediamo un esempio.

- **ES** = [`DFA` - $k = 3$] → [`GNFA` - $k = 5$] → [`GNFA` - $k = 4$] → [`GNFA` - $k = 3$] → [`GNFA` - `k = 2`] → [`REGEX`]

In pratica, ad ogni iterazione, si sceglie uno stato da eliminare e si ricavano, poi, le corrispondenti espressioni regolari, cancellando tutti gli archi coinvolti in questo stato.

L'ultimo argomento di questo capitolo riguarda i **linguaggi non regolari**. Esatto. Fino ad ora abbiamo visto solo linguaggi regolari, ma scommettiamo di avere questo tipo di linguaggio qui.

- $L = a^n b^n | n \geq 0$;

Una qualsiasi macchina che riconosce questo linguaggio deve essere in grado di controllare se il numero di a è uguale al numero di b. Siccome n è chiaramente infinito, la macchina dovrebbe avere un numero infinito di stati. Quando ci troviamo di fronte ad un caso del genere, esiste un teorema, detto **pumping lemma**, che ci permette di dare una dimostrazione formale della non-regolarità del linguaggio:

- **T.H.** = Se L è un linguaggio regolare, allora esiste una costante positiva p tale che per ogni stringa w in L di lunghezza $|w| \geq p$, esistono tre stringhe x, y, e z, con $w = xyz$, che devono soddisfare tre condizioni:
  - per ogni $i \geq 0$, $xy^i z \in L$;

- $|y| > 0$;
- $|xy| \leq p$;

  Sembra complicato, ma spieghiamo punto per punto. Il primo punto indica che, per ogni intero $i \geq 0$, la stringa $xy^i z$ appartiene al linguaggio. In altre parole, per qualsiasi valore di i, possiamo prendere la sottostringa y di w e *pomparla*, ossia ripeterla per i volte, in modo che la nuova stringa ottenuta appartenga ancora al linguaggio. Il punto due è banale, cioè semplicemente la lunghezza di y deve essere non vuota e la lunghezza della sottostringa xy deve essere minore o uguale di p.

- **DIM** = consideriamo un automa M che riconosce un linguaggio L e sia p il numero di stati. Consideriamo una qualsiasi stringa w tale che $|w| \geq p$. Se w viene accettata, esiste una sequenza $r_0, r_1, \ldots, r_w$ di $|w| + 1$ stati che parte dallo stato iniziale, legge la stringa un simbolo alla volta e, seguendo la funzione di transizione, termina in uno stato accettante. Notiamo subito una cosa, come il numero di stati della sequenza è maggiore del numero totale di stati, poiché $|w| + 1 \geq p$. Quindi, uno stato è ripetuto. Per semplicità lo chiameremo r. La stringa viene suddivisa in tre parti $w = xyz$, dove x porta dallo stato iniziale al primo r, y va dal primo al secondo r e z dal secondo r allo stato finale. Sembra difficile, ma è meccanico. Banalmente riusciamo a dimostrare che $|xy| \leq p$ è verificato, poiché, altrimenti, già in xy avremo lo stato ripetuto. Inoltre, è verificato anche $|y| > 0$, perché y ci permette di andare dalla prima occorrenza di r alla seconda, quindi non potrebbe essere vuota. Non ci resta che dimostrare $xy^i z$. M legge fino ad x e arriva al primo r, poi leggendo y passa dal primo al secondo r. Poi, prosegue la lettura con il secondo y e va ancora da r ad r. Fino ad arrivare a z e, quindi, allo stato finale, dimostrando l'asserzione. In parole povere, stiamo dicendo che, dimostrando queste tre condizioni, dimostreremo che un linguaggio è regolare e, allora, per dimostrare che un linguaggio non lo è, dovremo arrivare ad un assurdo durante la dimostrazione.

Macchine di Turing

↗ Area	👩🏾‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

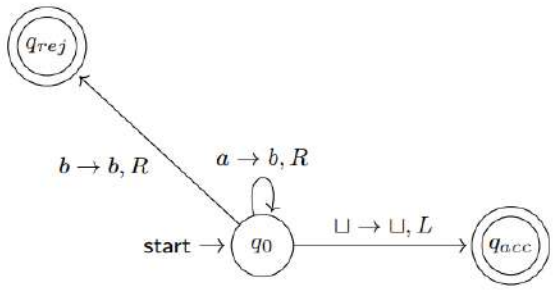
Fino ad ora, abbiamo visto modelli di computazione piuttosto semplici, come i DFA e gli NFA, che ovviamente hanno una potenza piuttosto limitata. Ci serve individuare una macchina altrettanto semplice, ma che sia potente come un computer. Ci viene in aiuto Alan Turing, matematico britannico, che schematizzò la costruzione della prima **macchina di Turing**. Si tratta di una macchina simile agli automi, ma con memoria illimitata e con la stessa potenza di un computer reale. Una MdT è una macchina a stati finiti, con un **nastro** infinito. Il nastro è diviso in celle, ognuna delle quali può contenere simboli. La computazione termina una volta giunti in uno stato di `accept` o di `reject`. Se non si raggiunge nessuno di questi stati, la computazione continua. All'inizio, il nastro contiene solo la stringa in input. Tutto il resto è vuoto e, anzi, per la precisione, è riempito dal carattere di $\sqcup$, **blank**. Il nastro è scorso da una **testina**, che si trova all'inizio della stringa in input. Andiamo, ora, a dare una definizione un po' più formale. Una MdT è una settupla $(Q, \Sigma, \Gamma, \delta, q_0, q_{accept}, q_{reject})$:

- Q è un insieme finito, che indica l'**insieme degli stati**.
- Σ è un insieme finito, che indica l'**alfabeto** su cui è definito la MdT ma, attenzione, perché $\sqcup \notin \Sigma$.
- Γ è un insieme finito, che indica l'**alfabeto del nastro**. In questo caso, $\sqcup \in \Gamma$ e, soprattutto $\Sigma \subset \Gamma$, cioè l'alfabeto principale è incluso in quello di nastro.
- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R\}$ è la **funzione di transizione**. A differenza degli automi, restituisce tre elementi: lo stato successivo della macchina, il simbolo da

scrivere sul nastro, preso dall'alfabeto di nastro e la direzione della testina, se deve muoversi a destra o a sinistra.

- $q_0 \in Q$ è lo **stato iniziale**.
- q_{accept} è lo **stato di accettazione**.
- q_{reject} è lo **stato di rifiuto**.

N.B. = gli stati di accettazione e rifiuto non hanno archi uscenti. Una volta che la computazione arriva in uno di essi, termina



Questo è un pratico esempio di come è composto il diagramma di stato di una MdT. Vediamo un po' la computazione. Mettiamo caso volessi vedere se la stringa `aba` è accettata. Mi trovo in q_0. Leggo `a` e resto in q_0, ma la testina si sposta a destra e scrive `b`. Ora, se legge `b`, vediamo che si sposta in q_{reject} e scrive `b`, spostandosi ancora a destra. Ovviamente, la computazione termina. La stringa `aba` non è accettata. Ad esempio, facendo lo stesso procedimento con la stringa `aaa`, noteremo come è accettata. Può anche capitare l'eventualità che gli stati di terminazione non vengano raggiunti e, in questo caso, la macchina va in loop. Durante la computazione di una MdT si possono verificare cambiamenti di tre tipi:

- stato, quando si passa da uno stato all'altro;
- contenuto del nastro, quando si scrive un simbolo diverso da quello contenuto in una cella del nastro.
- posizione della testina, quando la testina si sposta a destra o a sinistra.

Una **configurazione**, potremo dire, è una descrizione breve e concisa di questi tre elementi in un determinato istante della computazione. Quasi come se scattassimo una foto alla MdT in quel momento. È generalmente molto semplice effettuare la

configurazioni di una MdT data una determinata stringa, ma possiamo trovarci di fronte a casi particolari:

1. La testina si trova all'inizio del nastro, in questo modo $q_i bv$. La funzione di transizione potrebbe presentarsi in questo modo $\delta(q_i, b) = (q_j, c, L)$. In questo caso, la testina resta al suo posto, scrivendo il simbolo corretto, quindi $q_j cv$.
2. La testina si trova alla fine dell'input, in questo modo uaq_i. Semplicemente, la fine dell'input implica la presenza di un carattere $\sqcup$, quindi ci regoliamo in base alla funzione di transizione.

Una MdT accetta un input se esiste una sequenza di configurazioni $C_1, C_2, \ldots, C_k$, tale che C_1 e la configurazione iniziale e un ipotetico C_i produce $C_{(i+1)}$ per ogni $i = 1, \ldots, k-1$ e C_k è una qualsiasi configurazione di accettazione. Data una MdT M, il linguaggio riconosciuto da M è l'insieme delle stringhe M accetta. Tutte le stringhe che vengono accettate sono quelle che finiscono nello stato di q_{accept}. Quando una stringa non viene accettata, ci sono, però, due possibilità. La macchina può fermarsi in uno stato di q_{reject}, ma può anche essere che la MdT cicli all'infinito sull'input, non riuscendo a fermarsi mai. Un linguaggio si dice **Turing-riconoscibile** se esiste una MdT che lo riconosce. Prima, però, abbiamo detto che una MdT può anche finire in loop. Possiamo definire una MdT che si ferma su ogni input, chiamandola **decisore**. Un decisore decide un certo linguaggio se lo riconosce. Un linguaggio si dice **Turing-decidibile** se esiste una MdT che lo decide. Ogni linguaggio Turing-decidibile è anche Turing-riconoscibile, ma non viceversa. In quel caso, parleremo di **indecidibilità**, ma la approfondiremo successivamente.
Fino ad ora, abbiamo descritto una **MdT deterministica**. Ora vedremo come esistono molte **varianti** di MdT. Tutte quelle che vedremo hanno la stessa capacità computazionale, cioè riconoscono la stessa classe di linguaggi. Una prima variante che vedremo è la **MdT stayer**. Fino ad ora, abbiamo visto come la testina è obbligata a spostarsi a destra o a sinistra. Con la stayer, invece, aggiungiamo una nuova possibilità, ossia quella che la testina resti ferma. La definizione della settupla resta la stessa, ma basterà solo cambiare la funzione di transizione.

- $\delta : Q \times \Gamma \to Q \times \Gamma \times \{L, R, S\}$, dove S indica l'istante di computazione in cui la testina è ferma.

Prima abbiamo detto che le varianti della MdT riconoscono la stessa classe di linguaggi. Come proviamo che stayer e DMdT (MdT deterministica) hanno lo stesso potere computazionale? Allora, se volessimo provare che una stayer può simulare una DMdT,

è banale. Infatti, basta non usare mai S e la funzione di transizione sarà identica. Ora, però, vorremo essere in grado di provare che una DMdT riesca a simulare stayer. In realtà, anche qui, non è difficile giungere ad una conclusione. In poche parole, se durante la computazione di stayer ci troveremo di fronte ad un caso del genere: $\delta(q, a) = (q', a', S)$, allora la DMdT dovrà effettuare due passaggi: $\delta_M(q, a) = (\hat{q}, a', R)$ e $\delta_{M'}(\hat{q}, x) = (q, x, L)$. In parole povere, la testina si muove prima a destra, cambia il simbolo e raggiunge un altro stato. Poi, va a sinistra ed effettua il passaggio opposto.
Alcune operazioni computazionali, potrebbero aver bisogno di memorizzare una quantità maggiore di informazioni. È qui che introduciamo la **MdT multinastro**, che anziché avere un unico nastro, ha $k \geq 1$ nastri. L'idea di base è quella in cui l'input parte dall'inizio del nastro e gli altri nastri sono vuoti. La funzione di transizione cambia, infatti:

- $\delta : Q \times \Gamma^k \to Q \times \Gamma^k \times \{L, R, S\}^k$;

La transizione, oltre agli stati di partenza e di arrivo, dovrà specificare anche i k simboli letti, i k simboli da scrivere e i k movimenti delle k testine.
Le MdT multinastro sembrerebbero più potenti di quelle standard, ma ora dimostreremo che non è così. Prendiamo in considerazione due MdT, una multinastro, che chiameremo M e una a singolo nastro, che chiameremo S. Anche qui, la dimostrazione è abbastanza banale, poiché basterà che S simuli l'effetto dei k nastri di M memorizzando il loro contenuto su un unico nastro. Per separare i contenuti dei diversi nastri, si usa generalmente il simbolo $\#$. Quindi, una configurazione con k nastri corrisponderà ad una configurazione con k blocchi separati da $\#$. Per indicare la posizione della testina su ciascuno dei nastri, ogni simbolo viene sormontato da un punto, quindi scriveremo, supposto un $a \in \Gamma, \dot{a}$.
L'ultima variante di MdT che andremo ad analizzare è la **MdT non deterministica**, abbreviato **NMdT**. Anche qui, cambia la funzione di transizione, che sarà:

- $\delta : Q \times \Gamma \to \mathcal{P}(Q \times \Gamma \times \{L, R\})$;

Come possiamo vedere, è molto simile alla definizione data per gli NFA. Se la NMdT si trova in uno stato, leggendo un qualsiasi simbolo a, può trovarsi di fronte a più scelte e, in questo caso, si dividerà in più copie e la computazione viene eseguita in parallelo. Anche qui, quando andremo a definire le configurazioni, si andrà a creare un albero e un qualsiasi input viene accettato se e solo se l'albero termina con uno stato di accettazione. Anche qui vale lo stesso ragionamento con le precedenti varianti. Risulta

abbastanza semplice dimostrare che una MdT è anche una NMdT, dato che basterà disambiguare ogni ramo della computazione. Discorso diverso se vogliamo dimostrare che una NMdT è anche una MdT. Per semplicità, le chiameremo rispettivamente N e D. D, per simulare N, dovrà esplorare l'albero della computazione alla ricerca di una configurazione di accettazione. Ma, c'è un problema. Alcune diramazioni dell'albero potrebbero avere una lunghezza infinita. Quindi, nel caso D dovesse incappare in una diramazione infinita, non si fermerà mai e non si accorgerà mai di un'eventuale altra diramazione che termina in uno stato di accettazione. Per evitare ciò, utilizzeremo un algoritmo di ricerca degli alberi, chiamato **BFS** (***Breadth-First Search***), ossia la **ricerca in ampiezza**. Tutti i nodi vengono visitati in un dato livello, prima di passare al successivo. Quindi, in termini di MdT, eseguiamo il primo passo di ogni computazione. Se almeno una accetta, allora accetta. Poi col secondo e così via. Per semplicità, andremo a dividere la nostra D in una MdT multinastro, con tre nastri:

1. **Nastro dell'input**: contiene l'input che deve essere elaborato e viene letto solo dalla testina del primo nastro. Questo nastro non viene modificato durante la simulazione.
2. **Nastro di lavoro**: qui N viene simulata, in particolare vengono scritti i suoi stati e simboli correnti. Ogni volta che D deve eseguire la transizione, legge gli stati e i simboli correnti dal nastro di lavoro e determina la prossima transizione da eseguire.
3. **Nastro dell'indirizzo**: qui si tiene traccia delle posizioni correnti sui nastri di N.

Generalmente, si utilizza una MdT multinastro, perché abbiamo detto che, implementando una memoria più ampia, riesce a tener traccia di più input separatamente. E, ovviamente, abbiamo dimostrato che una qualsiasi MdT multinastro permette di simulare una MdT a singolo nastro e quindi possiamo ricondurci facilmente a D.
Abbiamo finito. Prima di passare al prossimo argomento, che sarà una sezione fondamentale della seconda parte del corso, dobbiamo definire un importantissimo concetto e ci serviranno le nozioni fornite da alcuni importanti matematici.

- **Alan Turing** (matematico inglese): padre, per eccellenza, dell'informatica. Oltre ad aver svolto un ruolo fondamentale durante la Seconda Guerra Mondiale, decifrando i codici segreti dei nazisti, ha dato la prima vera definizione di algoritmo e ha dimostrato che alcune questioni matematiche non possono essere risolte da alcun algoritmo o procedura finita.

- **Alonzo Church** (matematico americano): altro precursore dell'informatica, ha introdotto il concetto di funzione computabile o, più semplicemente, ricorsiva, dimostrando che era possibile definire una classe di funzioni matematiche che potevano essere calcolate da una MdT.
- **Stephen Kleene** (matematico americano): padre della teoria della computazione, ha generalizzato le teorie di Church, oltre ad aver fornito una base alla teoria dei linguaggi formali. Come abbiamo visto, ha anche introdotto l'operazione kleene-star ($*$), ossia tutte le stringhe che possono essere ottenute da un insieme di simboli attraverso un numero finito di operazioni di concatenazione e unione.
- **John Rosser** (matematico americano): ha avuto un impatto consistente nella logica matematica e nella teoria della computazione, teorizzando, col suo omonimo teorema, la completezza e la coerenza dei sistemi formali, alla base della dimostrazione dei linguaggi formali.

Le intuizioni di tutti questi matematici, hanno portato alla **Tesi di Church-Turing**.

- **T.H.** = Se esiste un algoritmo per eseguire un calcolo, allora questo calcolo può essere eseguito da una MdT (o variante equivalente).

- **N.B.** = si tratta di un'ipotesi non dimostrata, ma che si basa su ampie evidenze scientifiche ed empiriche ed è, quindi, accettata dalla comunità scientifica. Ovviamente, è possibile riscontrare problemi che risultano **indecidibili**, ossia non dimostrabili da una MdT. Lo vedremo presto.

Decidibilità

↗ Area	🧑🏾‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

Fino ad ora abbiamo parlato di automi e MdT. Nell'ultima parte del capitolo abbiamo parlato della testi di Church-Turing, *dimostrando* come un qualsiasi algoritmo può essere risolto da una MdT. In questo capitolo andremo ad analizzare i limiti della risoluzione dei problemi mediante gli algoritmi. Il nostro obiettivo sarà quello di dimostrare che alcuni problemi possono essere risolti algoritmicamente, e altri no. Vi starete chiedendo, perché? Perché studiare dei problemi che non possono essere risolti? Esistono due motivi principali:

1. Stabilire se un problema è irrisolvibile, può darci indicazioni in più su come semplificarlo o modificarlo, in modo da renderlo risolvibile algoritmicamente.
2. Teorizzare qualcosa di irrisolvibile, inoltre, può aiutarci ad aprire la mente di fronte alla teoria della computazione, stimolando la nostra capacità di risolvere problemi.

Quale sarà la nostra materia prima? In questo caso i **problemi di decisione.** Un problema di decisione è un problema che fornisce solo due tipi di risposte, `SÌ` o `NO`.

- $PRIMO$ = {Dato un numero x, x è primo?}

I problemi di decisione verranno codificati in linguaggi. Da tener conto che una MdT prende in input sempre una stringa, quindi altri oggetti più complessi dovranno diventare stringhe.
Ritornando al concetto precedente, dobbiamo trovare un problema indecidibile. Lo approfondiremo in seguito ma, un esempio pratico, è il **linguaggio** A_{TM}, noto anche come **problema della fermata**. A_{TM} consiste in un linguaggio di tutte le coppie (M, w) dove M è una MdT che accetta w, quindi la cui computazione si ferma in uno

stato di accettazione. È stato dimostrato che questo problema è indecidibile, ossia non esiste un algoritmo in grado di risolverlo. Questo linguaggio è divenuto, ben presto, simbolo dei limiti dei calcolatori. Ma come hanno fatto a dimostrarlo? Prima di proseguire, quindi, ci servono alcuni strumenti fondamentali. Introduciamo la **cardinalità degli insiemi infiniti**. Riflettiamo un attimo. Stabilire la cardinalità di un insieme finito è banale, basta contare i suoi elementi. Ma se l'insieme fosse infinito? Possiamo farlo? Partiamo dal concetto di infinito. Lo vediamo come una dimensione fissa, ma si può manipolare in qualche modo. Definiamo il **Paradosso di Hilbert**.

- Siamo in un albergo, con infinite stanze e con ospiti infiniti. Ad un certo punto, si presenta un nuovo ospite, che chiede una stanza. L'albergatore deve trovare un modo per alloggiarlo, come fa? Sposta tutti gli ospiti nella camera successiva (l'ospite della camera 1 nella 2, ecc). L'albergo è infinito, quindi non resterà nessuno fuori e, soprattutto, si potrà sistemare il nuovo ospite nella camera 1. Successivamente, si presente una comitiva di infiniti turisti. Come farà l'albergatore ad alloggiarli? Ogni ospite viene spostato nella stanza col numero doppio rispetto a quello attuale (l'ospite della 1 nella 2, l'ospite della 2 nella 4, ecc). Così facendo, le camere con i numeri dispari saranno tutte libere. Se invece avessimo infiniti alberghi, con infinite stanze. Tutti gli alberghi chiudono e ne resta solo uno. Tutte le persone devono alloggiare nell'unico albergo rimasto. Come fanno?

In questo caso, ad ogni persona viene assegnata una coppia di numeri (n, m), in cui n indica l'albergo di provenienza ed m la stanza relativa. Ci troveremo di fronte a qualcosa del genere:

- $$\begin{matrix} (1,1) & (1,2) & (1,3) & (1,4) & \cdots \\ (2,1) & (2,2) & (2,3) & (2,4) & \cdots \\ (3,1) & (3,2) & (3,3) & (3,4) & \cdots \\ (4,1) & (4,2) & (4,3) & (4,4) & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{matrix}$$

A questo punto, è facile assegnare le nuove stanze $[(1,1) \to 1; (1,2) \to 2]$. Abbiamo visto come l'infinito non è una dimensione fissa e possiamo, in qualche modo, estendere la sua definizione e, ovviamente, anche la sua cardinalità. Sappiamo che due insiemi finiti hanno la stessa cardinalità se gli elementi dell'uno possono essere messi in corrispondenza uno ad uno con quelli dell'altro. Si può fare con quelli infiniti? Dobbiamo definire la cardinalità in maniera astratta. Da qui, arriva il nostro primo teorema.

- **T.H.** = Due insiemi X e Y hanno la stessa cardinalità se e solo se esiste una funzione biettiva $f : X \to Y$ di X su Y.

Conosciamo già la definizione di funzione biettiva, ossia quando è sia iniettiva che suriettiva. Contestualizzato, ogni elemento di X viene mappato in un unico elemento di Y e per ogni elemento di X esiste un unico elemento di Y che viene mappato in esso. Usando questa definizione, possiamo definire un **insieme numerabile**. Un insieme si dice numerabile se possiamo, appunto, numerare i suoi elementi e scrivere una lista. Questo è facile per gli insiemi finiti. Quindi diremo che un insieme si dice numerabile se è finito oppure se i suoi elementi possono essere messi in corrispondenza biunivoca con i numeri naturali. Se un insieme numerabile possiede un numero infinito di elementi, viene detto **infinito numerabile**, e dato che può essere messo in corrispondenza biunivoca con i numeri naturali, si può dire che un insieme è infinito numerabile se ha la cardinalità di $\mathbb{N}$. Proviamo a prendere come esempio l'insieme $\mathbb{Q}$ dei numeri razionali. Creiamo una lista, un rettangolo, come abbiamo fatto prima:

- $$\begin{matrix} 1/1 & 1/2 & 1/3 & 1/4 & \cdots \\ 2/1 & 2/2 & 2/3 & 2/4 & \cdots \\ 3/1 & 3/2 & 3/3 & 3/4 & \cdots \\ 4/1 & 4/2 & 4/3 & 4/4 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{matrix}$$

Vista così, sembra impossibile numerare questo insieme. Cioè, se dovessimo partire dalla prima riga, non giungeremo mai alla seconda. Ci viene in aiuto un matematico tedesco, **Georg Cantor**, che introdusse il metodo della **diagonalizzazione**. Proviamo a dividere questo rettangolo fittizio in diagonali.

- $$\begin{matrix} 1/1 & 1/1 & 1/3 & 1/4 & \cdots \\ 2/1 & 2/2 & 2/3 & 2/4 & \cdots \\ 3/1 & 3/2 & 3/3 & 3/4 & \cdots \\ 4/1 & 4/2 & 4/3 & 4/4 & \cdots \\ \cdots & \cdots & \cdots & \cdots & \cdots \end{matrix}$$

Così facendo, ci troveremo sempre di fronte ad una diagonale finita, in questo caso quelle evidenziate con lo stesso colore. Tuttavia, non è sempre così. Non è sempre possibile stabilire una biezione tra due insiemi infiniti per poterne studiare la cardinalità. Per esempio, usando la diagonalizzazione possiamo dimostrare che l'insieme $\mathbb{R}$ dei numeri reali non è numerabile. Per farlo, dobbiamo giungere ad una contraddizione, quindi dimostriamo che non esiste una biezione tra $\mathbb{N}$ ed $\mathbb{R}$, supponendo che essa ci sia.

Possiamo supporre per assurdo che l'insieme dei numeri reali sia numerabile, cioè che esista una lista $f(1)$, $f(2)$, $f(3)$ di numeri reali, in cui essi sono rappresentati in un certo modo.

- $f_0(i), f_1(i), f_2(i)$;

Per essere più chiari, se $f(1) = 4,256\ldots$, allora $f_0(1) = 4$, $f_1(1) = 2$, ecc. Costruiamo, ora, un numero reale x, facendo si che sia compreso tra 0 e 1 e che la sua n-esima cifra decimale sia diversa dalla n-esima cifra decimale di uno dei numeri reali nella lista che abbiamo definito prima. Così facendo, giungiamo subito ad una contraddizione, poiché x risulterà non presente nella lista originale. Ma se abbiamo supposto che la lista contiene tutti i numeri reali, allora è banalmente assurdo. Giungiamo, quindi, alla conclusone che l'insieme dei numeri reali non è numerabile. Ma perché tutto questo? La tesi di Church-Turing ci dice che, se un insieme è numerabile, quindi è possibile risolverlo algoritmicamente, allora esiste una MdT che lo riconosce. Ma, abbiamo trovato un insieme non numerabile. Da qui, ad un teorema.

- **T.H.** = esistono alcuni linguaggi che non sono **Turing-riconoscibili**.

Per continuare con la nostra dimostrazione, dobbiamo definire il concetto di **MdT universale** U. Essa simula la computazione di una qualsiasi altra MdT. Prima abbiamo introdotto il problema della fermata A_{TM}, e ora dimostreremo se è Turing-riconoscibile o no. Definiamo un'arbitraria MdT universale U che riconosce A_{TM} sull'input $< M, w >$, dove M è una qualsiasi MdT e w è una stringa. L'idea alla base è che U simuli M sull'input w. Se M accetta, allora accetta, altrimenti se M rifiuta, allora anche U rifiuta. Poiché A_{TM} è costituito da tutte le possibili coppie (M, w), allora giungiamo alla conclusione che A_{TM} è Turing-riconoscibile. Abbiamo dimostrato che U riconosce A_{TM}. Ma lo decide anche? Beh, no. Infatti se M cicla su w, allora cicla anche U. Come facciamo a dimostrare che A_{TM} non è **Turing-decidibile**. Prima di tutto, parliamo del **Paradosso di Russell**.

- In un paese c'è un unico barbiere, che è un uomo ben sbarbato, che rade tutti e soli gli uomini del villaggio che non si radono da soli. La domanda è, chi rade il barbiere? Ci sono due possibilità:
  1. il barbiere rade se stesso? No, per definizione rade solo chi non si rade da solo.
  2. il barbiere non rade se stesso? Dato che il barbiere rade solo quelli che non si radono da soli, allora rade se stesso.

È chiaro che ci troviamo di fronte ad una contraddizione. O un **antinomia**, cioè due informazioni contraddittorie che possono essere entrambe dimostrate. Questo tipo di problema, detto **autoreferenza**, ci consente di dimostrare che alcuni problemi non possono essere decisi. Torniamo a noi. Per dimostrare che A_{TM} è indecidibile, assumiamo che esso lo sia. Chiamiamo H un decisore di A_{TM}. Quindi ci sono due possibili risultati di computazione: H accetta se M accetta w, allora H rifiuta se M non accetta w. Costruiamo una MdT che sfida M, usando H come sottoprogramma. La chiameremo D. Essa simula H sull'input $< M, < M >>$ e fornisce l'output opposto, ossia se H accetta, allora rifiuta e viceversa. È simile all'esecuzione di un programma che chiama se stesso in input (un compilatore per Python, può essere scritto in Python, quindi avrebbe senso eseguirlo). Eseguendo D con la sua stessa descrizione, otterremmo l'output opposto. Ed eccoci alla nostra contraddizione. Possiamo usare il metodo della diagonalizzazione per avere le idee più chiare.

	$< M_1 >$	$< M_2 >$	$< M_3 >$	$< M_4 >$	$\cdots$	$< D >$	$\cdots$
M_1	acc	rej	acc	rej	$\cdots$	$\cdots$	$\cdots$
M_2	acc	acc	acc	acc	$\cdots$	$\cdots$	$\cdots$
M_3	rej	rej	rej	rej	$\cdots$	$\cdots$	$\cdots$
• M_2	acc	acc	rej	rej	$\cdots$	$\cdots$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$
D	acc	acc	rej	rej	$\cdots$	$???$	$\cdots$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

Banalmente, la contraddizione è $???$, dove appunto ci troveremo un'entrata che è l'opposto di sé stessa. In conclusione, A_{TM} è Turing-riconoscibile, ma è indecidibile. Cerchiamo di trovare un linguaggio che non è neanche Turing-riconoscibile. Non possiamo usare A_{TM} come esempio, poiché abbiamo dimostrato che esso è Turing-riconoscibile. Il teorema che ora andremo a dimostrare dice che, se un linguaggio e il suo complemento sono entrambi Turing-riconoscibili, allora il linguaggio è decidibile. Quindi, in caso di linguaggio indecidibile, esso sicuramente non sarà Turing-riconoscibile, oppure non lo sarà il suo complemento. Il complemento è fondamentalmente tutte quelle stringhe che non fanno parte del linguaggio. Diremo che un linguaggio è **co-Turing-riconoscibile** se esso è il complemento di un linguaggio Turing-riconoscibile.

- **T.H.** = un linguaggio è decidibile se e solo se è Turing-riconoscibile e co-Turing-riconoscibile.

- **DIM** = se L è un linguaggio decidibile, allora esiste una MdT M con due possibili risultati di computazione, accettazione o rifiuto, tale che M accetta una stringa w se e solo se $w \in L$. Allora L è Turing-riconoscibile. Da qui risulta facile dimostrare che una MdT M accetta w se e solo se $w \in L$.
  - $w \rightarrow [M] \rightarrow \begin{cases} accetta\ se & w \in L \\ rifiuta\ se & w \notin L \end{cases} \rightarrow \begin{cases} rifiuta\ se\ M\ accetta \\ accetta\ se\ M\ rifiuta \end{cases}$

  Supponiamo che L e il suo complemento siano entrambi Turing-riconoscibili. Sia M_1 una MdT che riconosce L, e sia M_2 una MdT che riconosce L. Costruiamo una MdT N a due nastri. Sull'input x copia x sui nastri di M_1 e M_2. Simula M_1 e M_2 in parallelo, mettendoli sui due nastri di N. Se M_1 accetta, accetta; se M_2 accetta, rifiuta. Possiamo dire che N decide L, perché nel caso $x \in L$, questo si verifica se e solo se si arresta e accetta x. Quindi N accetta x. Nel caso in cui $x \notin L$, anche questo si verifica se e solo M_2 se si arresta e accetta x. Quindi N rifiuta x. Solo una delle due MdT accetta x, quindi N è una MdT con solo due possibili stati di computazione. Concludiamo che A_{TM} non è Turing-decidibile.

Ne abbiamo detti di teoremi, finora. Bene, giungiamo all'ultimo teorema di questo capitolo.

- **T.H.** = $\overline{A_{TM}}$ non è Turing-riconoscibile.
- **DIM** = in realtà, è abbastanza banale. Supponendo che $\overline{A_{TM}}$ sia Turing-riconoscibile, allora sarebbe decidibile. Ma questo abbiamo dimostrato che non è possibile, infatti A_{TM} non è Turing-decidibile. Contraddizione e, quindi, il teorema è dimostrato.

Riducibilità

↗ Area	👩🏾‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

Nel capitolo precedente, abbiamo presentato un problema computazionalmente irrisolvibile, cioè A_{TM}, che non può essere riconosciuto da una MdT e, quindi, da nessun'altro calcolatore. Andremo a presentare altri problemi irrisolvibili, con il metodo principale per dimostrare la loro irrisolvibilità, ossia la riducibilità. Una **riduzione** è la conversione di un problema in un altro problema più semplice o già risolto. L'idea alla base è che, se possiamo risolvere il secondo problema, allora si può risolvere il primo. Immaginiamo di avere due problemi A e B, con A molto complicato e B più semplice. Se riusciamo a trasformare un'istanza di A in un'istanza di B, in modo che, una volta risolto B, risolveremo anche A, allora staremo effettuando una riduzione. Utilizzando uno scenario matematico, il problema di calcolare l'area del rettangolo si riduce al problema di misurare la larghezza e la lunghezza. E così via. Così, per dimostrare che un problema è indecidibile, dimostreremo che un problema già noto e quindi già indecidibile, si riduce ad esso. Ora, possiamo vedere un esempio pratico. Nel capitolo precedente abbiamo dimostrato che è A_{TM} indecidibile. Ci piacerebbe dimostrarlo, ora, anche per $HALT_{TM}$, ossia il **problema della fermata**, che si chiede se una MdT si ferma su un determinato input. Useremo l'indecidibilità di A_{TM} per dimostrare che $HALT_{TM}$ è indecidibile, effettuando proprio una riduzione.

- **T.H.** = $HALT_{TM}$ è indecidibile.
- **DIM** = prima abbiamo parlato di un problema A e di un problema B. Ora, applichiamolo a questa dimostrazione. Sappiamo che A_{TM} risulta indecidibile. Vogliamo provare, ora, che $HALT_{TM}$ risulta indecidibile. Assumiamo, per assurdo, che $HALT_{TM}$ sia decidibile ed usiamo questa assunzione per dimostrare che A_{TM} è decidibile, arrivando ad una contraddizione. Sia R una MdT che decide

$HALT_{TM}$. Usiamo R per costruire una nuova MdT S, che decide, invece, A_{TM}. S simula R sull'input $< M, w >$. Se R rifiuta, allora rifiuta. Se R accetta, simula M su w finché non si ferma. Se M ha accettato, accetta. Se M ha rifiutato, rifiuta. Siamo giunti alla conclusione che R decide $HALT_{TM}$, ma allora anche S decide A_{TM}. Ma sappiamo che non è possibile, quindi siamo giunti ad una contraddizione e, automaticamente, concludiamo che $HALT_{TM}$ è indecidibile.

Dimostreremo, adesso, l'indecidibilità di altri problemi, per avere un quadro chiaro della situazione. Introduciamo E_{TM}, ossia il **problema del vuoto**, che si chiede se una MdT accetta o no il vuoto, ossia non accetta nessuna stringa.

- **T.H.** = E_{TM} è indecidibile.

- **DIM** = possiamo seguire la stessa falsa riga della dimostrazione precedente, istituendo un decisore R per costruire S, che decide A_{TM}. Se eseguissimo semplicemente R su M, otterremo solo informazioni parziali. Infatti, se R accetta M, sappiamo che il linguaggio è vuoto e che quindi M non accetta w. Ma se R rifiuta M, sappiamo solo che il linguaggio non è vuoto, ma non se effettivamente M accetta w. Dobbiamo cambiare la descrizione di M in modo che M rifiuti tutte le stringhe tranne w, ma su input w funziona come al solito. Eseguiamo R sulla macchina modificata. Se R accetta la descrizione della macchina modificata, vorrà dire che essa non accetta nulla e quindi M non accetta w. La macchina modificata la chiameremo, per semplicità, M_1. M_1 prende in input una stringa x e ha la stringa w incorporata nella sua descrizione. Se $x \neq w$, allora M_1 rifiuta immediatamente. Se invece $x = w$, allora M_1 esegue la macchina M sull'input w. Se M accetta w, allora anche M_1 accetta. In parole povere M_1 simula M. M_1 è costruita in modo tale che accetta solo la stringa w e rifiuta tutte le altre stringhe. S deve poter simulare M_1 partendo da una descrizione di M e w. Può farlo, perché deve aggiungere ad M alcuni stati in più per verificare che $x = w$. Concludiamo che, quindi, S è un decisore di A_{TM}. Ma non è possibile, perché sappiamo che A_{TM} è indecidibile.

Un altro interessante problema è $REGULAR_{TM}$, il **problema di equivalenza degli automi**. $REGULAR_{TM}$ si chiede se una MdT ha un automa finito equivalente, che sia un DFA o un NFA.

- **T.H.** = $REGULAR_{TM}$ è indecidibile.

- **DIM** = anche qui, effettuiamo una riduzione da A_{TM}. Costruiamo una MdT, M_1, tale che per ogni coppia $< M, w >$, si comporta in modo seguente su un input x. Se x è nella forma 0^n1^n, allora M_1 accetta. Altrimenti, simula M su w. Se M accetta w, allora M_1 accetta x. Se M rifiuta, allora M_1 rifiuta x. A questo punto, $L(M_1)$ è regolare se e solo se M accetta w. Ma quindi, abbiamo stabilito indirettamente che A_{TM} è decidibile. E, anche qui, contraddizione.

Fino ad ora, abbiamo usato riduzioni da A_{TM}. Ma possiamo anche usare una riduzione da E_{TM}, come per esempio nel caso del problema EQ_{TM}, ossia il **problema di equivalenza di due MdT**, che si chiede se due MdT riconoscono lo stesso linguaggio.

- **T.H.** = EQ_{TM} è indecidibile.

- **DIM** = come anticipato, partiamo da E_{TM}. Chiamando S un suo decisore, S costruisce una nuova MdT M_1, tale che $L(M_1) = \emptyset$. Chiamando R decisore di EQ_{TM}, esegue l'input su $< M, M_1 >$. Se R accetta, S accetta, vale a dire che si verifica una cosa del genere $L(M) = L(M_1) = \emptyset$. Altrimenti, potrebbe verificarsi che $L(M) \neq L(M_1)$. In questo caso, il linguaggio riconosciuto da M non è vuoto. E concludiamo che S non può esistere e che, quindi, EQ_{TM} è indecidibile.

Abbiamo visto vari problemi e abbiamo dimostrato la loro indecidibilità grazie a riduzioni. Ma ora tocca formalizzare il concetto di riducibilità. Lo faremo usando la **riducibilità tramite funzione**. Per esempio, se volessimo ridurre un problema A al problema B utilizzando una funzione, significa che esiste una **funzione calcolabile** che trasforma istanze di A in istanze di B. Una funzione $f : \Sigma^* \to \Sigma^*$ è una funzione calcolabile se esiste una MdT M che, su qualsiasi input w, si ferma avendo solo $f(w)$ sul nastro. Per farla breve, è una funzione che può essere calcolata da una MdT. Ora, diamo una definizione formale usando linguaggi. Diremo che un linguaggio A è riducibile tramite funzione al linguaggio B, e si denota con $A \leq_m B$, se esiste una funzione calcolabile $f : \Sigma^* \to \Sigma^*$, dove per ogni w, $w \in A \Leftrightarrow f(w) \in B$. Vale a dire, presa una stringa qualsiasi w, essa sarà accettata da A se e solo se la sua immagine tramite la funzione f appartiene a B. La funzione f mappa le stringhe che appartengono ad A nelle stringhe che appartengono a B, e viceversa. Se una stringa w appartiene ad A, la sua immagine $f(w)$ deve appartenere a B, e se una stringa w non appartiene ad A, allora la sua immagine $f(w)$ non appartiene a B. Analogamente alla definizione di riduzione, se un problema è riducibile tramite funzione ad un secondo

problema precedentemente risolto, allora avremo dimostrato l'asserto. Da qui, deriva il teorema.

- **T.H.** = Se $A \leq_m B$ e B è decidibile, allora anche A è decidibile.
- **DIM** = Sia M un decisore per B ed f la riduzione da A a B. Costruiamo un nuovo decisore N per A, tale che, su un input w, computa $f(w)$ ed esegue M su input $f(w)$, restituendo lo stesso output di M. Data la definizione di riduzione $w \in A \Leftrightarrow f(w) \in B$, M accetta $f(w)$ ogni volta che $w \in A$. Quindi N decide A e abbiamo dimostrato il teorema.

Da questa dimostrazione, deriva un altro teorema.

- **T.H.** = Se $A \leq_m B$ e A non è Turing-riconoscibile, allora anche B non sarà Turing-riconoscibile.
- **DIM** = La dimostrazione vale la stessa per questo teorema, solo che M ed N diventano riconoscitori e non decisori.

Un corollario, cioè una conseguenza immediata di questo teorema, può essere l'applicazione contraria, ossia che dato $A \leq_m B$, se A non è Turing-riconoscibile, allora non lo è neanche B. Possiamo usare la riducibilità tramite funzione per dimostrare che alcuni problemi non sono né Turing-riconoscibili e né co-Turing-riconoscibili. Questo perché la definizione implica che, se $A \leq_m B$, allora varrà anche $\overline{A} \leq_m \overline{B}$. Allora, possiamo dimostrare che alcuni problemi non sono né Turing-riconoscibili e neanche co-Turing-riconoscibili.

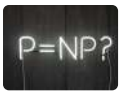
- **T.H.** = EQ_{TM} non è né Turing-riconoscibile e né co-Turing-riconoscibile.
- **DIM** = precedentemente abbiamo dimostrato che A_{TM} non è Turing-riconoscibile e neanche co-Turing-riconoscibile. Possiamo, quindi, ridurre A_{TM} a $\overline{EQ_{TM}}$. Per la prima asserzione, possiamo suppore che EQ_{TM} è Turing-riconoscibile. Si verificherebbe una cosa del genere $A_{TM} \leq_m \overline{EQ_{TM}}$, che è analoga a $\overline{A_{TM}} \leq_m EQ_{TM}$. Ma è impossibile, perché non è Turing-riconoscibile. Quindi concludiamo che non è EQ_{TM} Turing-riconoscibile. Senza scriverlo, supponendo che anche EQ_{TM} sia co-Turing-riconoscibile, ci troveremo di fronte alla stessa dimostrazione.

Un altro strumento molto importante riguardo le funzioni calcolabili, è il **teorema di Rice**. Esso afferma che, per ogni proprietà non banale delle funzioni calcolabili, il problema di decidere quali funzioni soddisfino tale proprietà e quali no, è indecidibile.

Una **proprietà non banale** è una proprietà che non effettua alcuna discriminazione sulle funzioni calcolabili, ossia o vale per tutte o per nessuna.

- **T.H.** = sia P = {$< M >$ | M è una MdT che verifica una proprietà $\mathcal{P}$} un linguaggio. Se esso verifica queste due condizioni, allora P è indecidibile.
  1. $\forall\ M_1, M_2$ MdT tali che $L(M_1) = L(M_2)$, $\langle M_1 \rangle \in P \Leftrightarrow \langle M_2 \rangle \in P$;
  2. $\exists\ M_1, M_2$ MdT tali che $\langle M_1 \rangle \in P$, $\langle M_2 \rangle \notin P$;

- **DIM** = la dimostrazione del teorema di Rice utilizzando una riduzione da A_{TM}. Andremo a fare una riduzione di questo tipo $A_{TM} \leq_m L_P$, con L un arbitrario linguaggio che gode di una proprietà P. Sia $T_\emptyset$ una MdT tale che $L(T_\emptyset) = \emptyset$. Assumiamo che $T_\emptyset \notin L_P$. Poiché, per definizione, L_P non è banale, esiste una MdT T tale che $\langle T \rangle \in L_P$. Progettiamo, ora, una funzione f, tale che $f(\langle M, w \rangle) = \langle S \rangle$, con S decisore per L_P. S, su input x, simula M su w. Se M rifiuta w, S rifiuta x. Se M accetta w, S simula T su input x. Se M va in loop, S rifiuta. Quindi, $\langle M, w \rangle \in A_{TM}$ se e solo se $L(S) = L(T)$, a sua volta se e solo se $\langle S \rangle \in L_P$. Allora, $\langle M, w \rangle \notin A_{TM}$ se e solo se $L(S) = \emptyset$, ma che equivale a dire che $L(S) = L(T_\emptyset)$, se e solo se $\langle S \rangle \notin L_P$.



Complessità

↗ Area	👩🏾‍🎓 Studio
↗ Project	📐 Elementi di Teoria della Computazione
☑ Done	☑️
∑ Due Status	🎉 Done!
∑ Current Task	⚪

Nel capitolo precedente, abbiamo affrontato la riducibilità di alcuni problemi computazionali che non possono essere risolti. Ma, anche quando un problema è decidibile, ossia che è computazionalmente risolvibile, la sua ricerca della soluzione potrebbe richiedere una grande quantità di tempo e di memoria. Per questo motivo, introduciamo la **complessità**, che studia il tempo, la memoria e le risorse che servono per risolvere un problema computazionale. Ci concentreremo, innanzitutto, sul **tempo**. Per proseguire, facciamo un esempio pratico. Consideriamo un linguaggio $A = \{0^k1^k | k \geq 0\}$. Di quanto tempo necessita una qualsiasi MdT a singolo nastro per decidere A? Il **tempo di esecuzione** si calcola in base alla lunghezza della stringa che rappresenta l'input della nostra MdT. Si analizza, di solito, il **caso peggiore**, cioè tempo di esecuzione massimo tra tutti gli input di una determinata lunghezza, e il **caso medio**, cioè la lunghezza media dei tempi di esecuzione su tutti gli input di una determinata lunghezza. Da qui, deriva una definizione.

- **DEF** = sia M una MdT che si arresta su tutti gli input. La complessità di tempo di M è la funzione $f : N \to N$, dove $f(n)$ è il numero massimo di passi che M utilizza su un qualsiasi input di lunghezza n. Se $f(n)$ è il tempo di esecuzione di M, diciamo che M ha tempo di esecuzione $f(n)$ e che M è una MdT di tempo $f(n)$.

Non è sempre facile stabilire il tempo esatto di esecuzione di un algoritmo, per questo se ne fa una stima e, generalmente, si utilizza l'**analisi asintotica**. Per fare una buona analisi asintotica, si prende in considerazione il termine di ordine maggiore del tempo di esecuzione.

- **ES** = $f(n) = 6n^3 + 2n^2 + 20n + 45$. Il termine di ordine maggiore è $6n^3$, ma si trascura la costante 6 e quindi la notazione asintotica è $f(n) = O(n^3)$, che noi chiameremo **notazione O-grande**.

Da qui, deriva un'altra definizione.

- **DEF** = Siano f e g due funzioni tali che $f, g : \mathbb{N} \to \mathbb{R}^+$. Si dice che $f(n) = O(g(n))$ se esistono interi positivi c e n_0 tali che, per ogni $n \geq n_0$, $f(n) \leq_c g(n)$. Verificate queste condizioni, diremo che $g(n)$ è un **limite superiore** per $f(n)$.

Fino ad ora, come modello di calcolo, abbiamo usato una MdT a singolo nastro deterministica. Ma, la scelta del modello di calcolo, può influenzare la complessità di tempo, infatti potremo prendere in considerazione anche MdT multinastro e non deterministiche. Da qui, nasce un teorema.

- **T.H.** = Sia $t(n)$ una funzione, tale che $t(n) \geq n$. Ogni MdT multinastro, di tempo $t(n)$ ammette una MdT equivalente a nastro singolo di tempo $O(t^2(n))$.

- **DIM** = per la dimostrazione, ci basta ricordare la dimostrazione che già abbiamo visto, ossia che una MdT multinastro ha sempre una MdT a singolo nastro che la simula ed è equivalente. Sia M una MdT a k nastri e sia S una MdT a singolo nastro. Ricordiamo che S, per simulare M, utilizza il suo unico nastro per rappresentare il contenuto di k nastri. Inizialmente, S mette sul proprio nastro tutti i nastri di M. Per simulare un passo, S scorre tutte le informazioni memorizzate sul suo nastro per determinare i simboli presenti sotto le testine di M. Poi, S esegue una nuova scansione e aggiorna il contenuto dei nastri e le posizioni delle testine. Analizziamo questa simulazione in termini di complessità. È acclarato che S effettua due passi sul suo nastro: il primo ottiene informazioni, il secondo le esegue. Chiamando **parte attiva** la porzione di nastro di S che viene effettivamente utilizzata, la sua lunghezza indica il tempo che S impiega per fare la scansione, quindi dobbiamo determinare un limite superiore per questa lunghezza. Sapendo che M utilizza $t(n)$ celle del nastro in $t(n)$ passi, allora la lunghezza della parte attiva di S sarà al più $t(n)$. Quindi, una scansione della parte attiva di S impiega $O(t(n))$ passi, che è anche il tempo totale che S impiega per eseguire un passo di M. Ora ci tocca limitare il tempo totale. Quindi, S simula ciascuno dei $t(n)$ passi di M, utilizzando $O(t(n))$ passi, quindi $t(n) \times O(t(n)) = O(t^2(n))$. L'asserto è verificato.

Prima di proseguire, c'è da fare alcune osservazioni sulla complessità temporale. Infatti, occorre considerare **codifiche ragionevoli**, vale a dire non prolisse, tali che le loro istanze non abbiano una rappresentazione eccessivamente lunga. Diremo che le codifiche ragionevoli sono **polinomialmente correlate**, cioè la lunghezza della codifica è al più polinomiale rispetto alla dimensione dell'istanza. Questa premessa ci permette di definire un concetto molto importante di questo capitolo. Fino ad ora, abbiamo dimostrato relazioni tra modelli diversi di computazione, evidenziando una differenza polinomiale tra le complessità temporali di una MdT a singolo nastro e una multinastro e una differenza esponenziale tra una MdT deterministica e una NMdT. Ora, però, ci tocca superare queste relazioni, poiché andremo a prendere in considerazione problemi la cui complessità temporale non è influenzata dal tipo di modello, poiché ci serve avere un aspetto più astratto della computazione e non solo della MdT, che è un modello e basta e non rappresenta tutto l'insieme computabile. Per questo, introduciamo la **classe P**. P è la classe dei linguaggi che sono decidibili in tempo polinomiale su una MdT a singolo nastro. In altre parole, un problema appartiene a P se esiste una MdT a singolo nastro che la risolve e che richiede un numero di passi polinomiale rispetto alla dimensione dell'istanza del problema. P è molto importante, poiché rappresenta la classe di quei problemi che sono realisticamente risolvibili con un calcolatore. La classe P ci fa capire che molte domande naturali non possono essere risolte in modo efficiente, ma la loro correttezza può essere verificata. Supponiamo di avere un problema X, che sappiamo è possibile risolvere in tempo polinomiale. Basandoci su questa definizione, quanti altri problemi possiamo individuare, risolvibili in tempo polinomiale? Introduciamo le **riduzioni polinomiali**. Un problema X si riduce in tempo polinomiale al problema Y se arbitrarie istanze di X possono essere risolte usando un numero polinomiale di passi di computazione più un numero polinomiale di chiamare ad un **oracolo** che risolve Y. Un oracolo è una specie di *black-box* che risolve istanze di Y in un solo passo. Come notazione diremo che $X \leq_P Y$. Le riduzioni polinomiali sono molto importanti, perché ci permettono di classificare i problemi in accordo alla loro difficoltà relativa, fornendo algoritmi, stabilendo intrattabilità ed equivalenza. Abbiamo tre tipi di riduzioni polinomiali:

- riduzioni mediante **equivalenza semplice**.
- riduzioni **da caso speciale a caso generale**.
- riduzioni mediante **codifica con gadgets**.

Andiamo passo per passo e definiamo i primi due problemi, che ridurremo tramite il primo metodo di equivalenza semplice.

- $INDIPENDENT-SET$ = dato un grafo G = (V, E), dove V sono i vertici e E gli archi, e un intero k, ci chiediamo se esiste un sottoinsieme di vertici, $S \subseteq V$,tale che la sua cardinalità è almeno k, $|S| \geq k$, e per ogni arco, al più uno dei suoi vertici è in S. In parole povere significa che se un arco ha un vertice in S, l'altro vertice non può essere in S.
- $VERTEX-COVER$ = dato un grafo G = (V, E), dove V sono i vertici e E gli archi, e un intero k, ci chiediamo se esiste un sottoinsieme di vertici, $S \subseteq V$,tale che la sua cardinalità è minore o uguale di k, $|S| \leq k$, e per ogni arco, almeno uno dei suoi vertici è in S. In altre parole, stiamo cercando di trovare un gruppo di vertici in cui ogni arco abbia almeno un estremo in questo gruppo, che sarà, appunto, il nostro *vertex-cover*. Ci chiediamo se esiste un *vertex-cover* di dimensione k.

Vogliamo dimostrare che $INDIPENDENT-SET \equiv_P VERTEX-COVER$. La riduzione tra questi due problemi si basa sul fatto che essi sono strettamente correlati. Se abbiamo un sottoinsieme di vertici S che è un *indipendent-set*, se prendiamo il complemento di quell'insieme, quindi in questo caso $V - S$, avremo il nostro *vertex-cover*. Per dimostrarlo, dobbiamo fornire una prova da ambo i lati. Preso S come *indipendent-set*, prendiamo in considerazione un arco arbitrario (u, v) del grafico. Se S è un *indipendent-set*, allora almeno uno dei due vertici deve essere contenuto in S. $u \in S$ - $v \notin S$ oppure $u \notin S$ - $v \in S$. Questo vale anche per $V - S$. E quindi arriviamo a dimostrare che $V - S$ è un *vertex-cover*. Procedendo in direzione opposta, supponiamo che S sia un *vertex-cover*. Consideriamo entrami i vertici, quindi $u \in V - S$ e $v \in V - S$. Ma se S è un *vertex-cover*, questo significa che (u, v) non può essere un arco nel grafo, perché quell'arco non sarebbe coperto, dato che nessuno dei due estremi è in S. Concludiamo, quindi, che $V - S$ è un *indipendent-set*.
Per spiegare il secondo tipo di riduzioni, introduciamo un nuovo problema.

- $SET-COVER$ = dato un insieme U di elementi, una collezione $S_1, S_2, \ldots, S_m$ di sottoinsiemi di U e un intero k, ci chiediamo se esiste una collezione $\leq k$ di questi insiemi la cui unione è uguale ad U. Per capirci meglio.
  - $U = \{1, 2, 3, 4, 5, 6, 7\}$
    $k = 2$
    $S_1 = \{3, 7\}$
    $S_2 = \{3, 4, 5, 6\}$
    $S_3 = \{1\}$
    $S_4 = \{2, 4\}$

$S_5 = \{5\}$
$S_6 = \{1, 2, 6, 7\}$

In questo caso, ho evidenziato di rosso i 2 sottoinsiemi la cui unione restituisce proprio U, per avere più chiaro il concetto.

Il nostro obiettivo, ora, è quello di dimostrare questa riduzione $VERTEX - COVER \leq_P SET - COVER$. In un'istanza di $VERTEX - COVER$ ci viene dato un grafo $G = (V, E)$ e un intero k come input. Partiamo dall'intero k, che è pressoché lo stesso. Poi avremo $U = E$, poiché il nostro insieme universo sarà uguale all'insieme degli archi. Infine, i nostri sottoinsiemi, che chiameremo S_v, saranno tutti gli archi e che sono incidenti su uno o più vertici v. Quindi avremo un *set-cover* al massimo k se e solo se il *vertex-cover* è al massimo k.
Per definire l'ultimo tipo di riduzione, dobbiamo introdurre un nuovo problema. Esso è detto **problema della soddisfacibilità**, ma prima di definirlo dovremo un attimo soffermarci su alcune sue proprietà. Un **letterale** è una variabile booleana o la sua negazione; una **clausola** è una disgiunzione ($\vee$) di letterali. Una **forma normale congiuntiva** (**CNF**) è una formula Φ che è una congiunzione ($\wedge$) di clausole. Per avere un'idea più chiara, ecco degli esempi.

- x_i - $\overline{x_i}$ → letterale
- $C_j = x_1 \vee \overline{x_2} \vee x_3$ → clausola
- $\Phi = C_1 \wedge C_2 \wedge C_3 \wedge C_4$ → CNF

Ora, possiamo introdurre il nostro problema.

- SAT = data una CNF Φ, ci chiediamo se ha un'assegnazione di verità che la soddisfa. Un caso particolare di SAT, che vedremo più nel dettaglio è $3 - SAT$, che è la stessa cosa, solo che ogni clausola ha esattamente tre letterali. Fondamentalmente, possiamo assegnare un valore, che sia `true` o `false`, ad ogni letterale. Se, in ogni clausola, almeno un letterale è soddisfatto, allora in quel caso sarà soddisfatta tutta la formula.

Ci poniamo come obiettivo quello di dimostrare che $3 - SAT \leq_P INDIPENDENT - SET$. Ora, sembra piuttosto complicato partire da una semplice Φ a finire in un grafo. Partiamo proprio da un'istanza di $3 - SAT$, quindi da una Φ le cui clausole hanno esattamente tre letterali. Vogliamo costruire un'istanza (G, k) di $INDIPENDENT - SET$, il cui *indipendent-set* ha dimensione k se e solo se Φ è soddisfacibile. In $3 -$

SAT noi assegniamo ad ogni letterale un valore, che sia `true` e `false`. E, banalmente, facciamo una scelta del genere anche in $INDIPENDENT-SET$, infatti scegliamo se includere o no i nostri vertici nell'*indipendent-set*. Quello che andremo a fare è creare, per ogni letterale, un vertice e sistemarli in un triangolo. Ogni clausola sarà, quindi, un triangolo. In ogni triangolo, dato che vogliamo un *indipendent-set*, siamo tenuti a scegliere al massimo un vertice. Ogni letterale può avere un valore, ma non avrebbe molto senso scegliere un letterale e anche il suo negato all'interno dell'*indipendent-set*, poiché ci troveremo di fronte ad una contraddizione. Quindi, per evitare ciò, andremo a collegare ogni letterale col suo negato, in modo da avere scelte consistenti. Ora dobbiamo chiederci quale sarà il valore del nostro k. È banale, in realtà, perché k sarà uguale al numero delle clausole.
Abbiamo, fino ad ora, descritto le principale metodologie di riduzione polinomiale. In ambito di riduzioni polinomiali, è di particolare importanza la transitività.

- se $X \leq_P Y$ e $Y \leq_P Z$, allora $X \leq_P Z$.

Basandoci su questa asserzione, potenzialmente, possiamo partire da un'istanza di $3-SAT$ e arrivare ad una di $SET-COVER$. Come? Esattamente come prima.

- $3-SAT \leq_P INDIPENDENT-SET \leq_P VERTEX-COVER \leq_P SET-COVER$;

Prima di passare al prossimo macro-argomento di questo capitolo, parliamo un attimo di **self-reducibility**. Tutte le riduzioni che abbiamo visto finora, si pongono una domanda. Per quanto concerne $VERTEX-COVER$, esso vuole trovare un *vertex-cover* di minima cardinalità, per fare un esempio. Questo è un **problema di ricerca**. Esso, però, per poter entrare nella classe P, deve diventare un **problema di decisione**, infatti andremo a chiederci se esiste un *vertex-cover* di cardinalità minima. Fondamentalmente, avviene una cosa di questo tipo.

- $SEARCH-PROBLEMS \leq_P DECISION-PROBLEMS$;

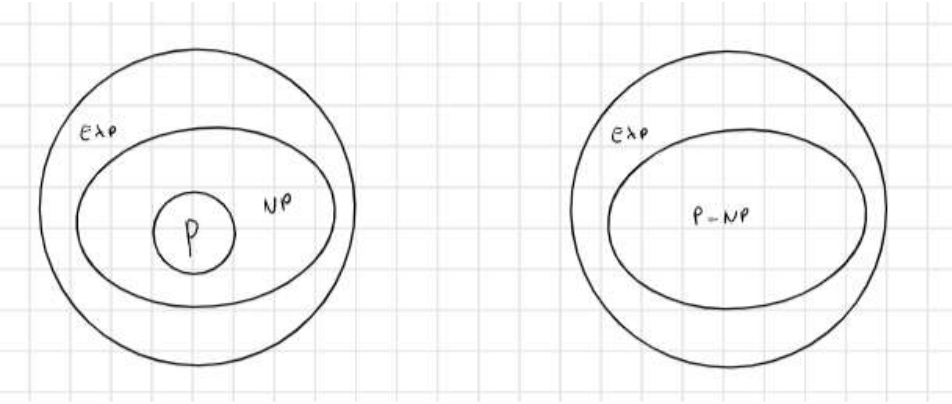
Questo è un concetto fondamentale che ci aiuterà nella comprensione di una nuova classe di problemi, la **classe** NP. Prima di proseguire, resta da tener conto di alcuni strumenti importanti. Un **certificatore** è un algoritmo che non controlla se un qualsiasi elemento s appartiene al problema X. Un certificatore si assicura che, una data dimostrazione t, sia efficiente nel verificare che $s \in X$.

- **DEF** = Un algoritmo $C(s,t)$ è un certificatore per il problema X se, per ogni stringa s, $s \in X$ se esiste una stringa t, detta **certificato**, tale che $C(s,t)$ = `yes`.

Un qualsiasi problema di decisione appartiene alla classe NP se possiede un certificatore in tempo polinomiale. Prima di proseguire, ci tocca fare chiarezza su alcune asserzioni. Abbiamo detto che appartengono alla classe P tutti quei problemi di decisione per cui esiste un algoritmo polinomiale. Appartengono, invece, alla classe NP tutti quei problemi di decisione per cui esiste un certificatore polinomiale. Una nuova classe che introduciamo è la **classe EXP**, che raccoglie tutti quei problemi di decisione per cui esiste un algoritmo esponenziale. Vogliamo verificare alcuni fatti.

- $P \subseteq NP$? Consideriamo un qualsiasi problema X in P. Per definizione sappiamo che esiste un algoritmo polinomiale $A(s)$ che risolve X. Con il nostro certificato $t = \varepsilon$, avremo che $C(s, t) = A(s)$.
- $NP \subseteq EXP$? Consideriamo un qualsiasi problema X in NP. Per definizione, sappiamo che esiste un certificatore polinomiale $C(s, t)$ per X. Per risolvere su input s, usiamo $C(s, t)$ su tutte le stringhe t, con $|t| \leq p(|s|)$. Viene restituito `yes` se e solo se $C(s, t)$ restituisce `yes` per qualsiasi stringa.
- $P = NP$? Qui, tocca aprire e chiudere una piccola parentesi. Se questa asserzione dovesse essere vera, per ogni problema per cui esiste un algoritmo polinomiale, allora esiste anche un certificatore polinomiale che ci garantisce una soluzione efficiente. La questione è ancora aperta, dato che questo è uno dei **problemi del millennio**. Nonostante non sia stato ancora dimostrato, e neanche il contrario, l'opinione comune di molti scienziati è che no, $P \neq NP$. Questo perché, se venisse dimostrato il vero, verrebbero messe in dubbio molte teorie che abbiamo sulla crittografia moderna, individuando vulnerabilità in molti sistemi.

Per avere le idee più chiare, ecco uno schema che mette in relazione questi problemi, sia se $P = NP$ e sia se $P \neq NP$.



Prima di proseguire, dobbiamo un attimo definire la **Tesi di Cook** e la **Tesi di Karp**.

- **DEF** = Cook afferma che il problema X **si riduce in modo polinomiale** al problema Y se istanze arbitrarie del problema X possono essere risolte usando un numero polinomiale di passi di computazione e un numero polinomiale di chiamate ad un oracolo che risolve Y.

- **DEF** = Karp afferma che il problema X **si trasforma in modo polinomiale** al problema Y se, dato un qualsiasi input x di X, possiamo costruire un input y tale che x è istanza `yes` di X se e solo se y è istanza `yes` di Y.

Fino ad ora, tutte le riduzioni che abbiamo visto, sono fondamentalmente trasformazioni, con una sola chiamata all'oracolo per Y, alla fine dell'algoritmo per X. Non si sa se queste due definizioni siano equivalenti, ma su di esse si basa la NP**-Completezza**. Un problema NP**-Completo** è un problema Y in NP, con la proprietà che per ogni problema X in NP, allora $X \leq_P Y$. Vediamo, adesso, un primo problema NP-Completo.

- $CIRCUIT-SAT$ = dato un circuito combinatoriale composto da porte logiche $\vee$, $\wedge$ e $\neg$, ci chiediamo se esso è soddisfacibile. Un circuito combinatoriale è soddisfacibile quando, data un'assegnazione di verità, l'output del circuito restituisce tutti `1`.

Vogliamo dimostrare che $CIRCUIT-SAT$ è un problema NP-Completo. Alla base di questa dimostrazione c'è il fatto che un qualsiasi algoritmo che prende in input un numero fissato n di bits e produce una risposta `yes` o `no`, può essere in qualche modo rappresentato da questo circuito. Per ora andiamo a considerare un qualsiasi problema X in NP. Per definizione, sappiamo allora che esiste un certificatore polinomiale $C(s,t)$. Per determinare se $s \in X$, dobbiamo sapere se esiste un certificato t di lunghezza $p(|s|)$ tale che $C(s,t)$ = `yes`. Consideriamo $C(s,t)$ come un algoritmo su $|s|+p(|s|)$ bit e convertiamolo in un circuito di dimensione polinomiale K. I primi $|s|$ bit sono noti con s, mentre i restanti $p(|s|)$ rappresentano i bit di t. Giungiamo alla conclusione che il circuito è soddisfacibile se e solo se $C(s,t)$ = `yes`.
Una volta individuato il primo problema NP-Completo, gli altri seguono. Infatti, possiamo sintetizzare i passi per stabilire l'NP-Completezza di un qualsiasi problema Y .

1. Dimostriamo che Y è in NP.
2. Scegliamo un problema NP-Completo X.

3. Dimostriamo che $X \leq_P Y$.

- **N.B.** = per la transitività, sia W un qualsiasi problema in NP. Se $W \leq_P X \leq_P Y$, allora $W \leq_P Y$ e, quindi, Y è NP-Completo.

A riguardo, la prima dimostrazione che vedremo è quella per $3-SAT$, in cui dimostreremo che è NP-Completo. Come? Facciamo la riduzione da $CIRCUIT-SAT \leq_P 3-SAT$. Ovviamente, partiamo da un'istanza di $CIRCUIT-SAT$, quindi da un circuito di porte logiche. Dobbiamo convertirlo in una formula Φ le cui clausole abbiano esattamente tre letterali. Analizziamo le varie porte logiche:

- $\neg$ = una porta $\neg$, con input x, corrisponde alla clausola $\neg x$.
- $\wedge$ = una porta $\wedge$ con input x e y corrisponde alla clausola $(\neg x \vee \neg y \vee z)$, dove z è la nostra nuova variabile output della porta $\wedge$.
- $\vee$ = una porta $\vee$ con input x e y corrisponde alle due clausole $(x \vee \neg z)$ e $(y \vee \neg z)$, dove z è la nostra nuova variabile output della porta $\vee$.

Ora, colleghiamo correttamente le clausole per garantire la corrispondenza logica tra le porte del circuito e le clausole di Φ. Ciò significa che le clausole corrispondenti alle porte di input devono essere collegate alle clausole corrispondenti alle porte di output. Infine, si aggiunge una clausola finale che richiede che l'output del circuito sia vero. Questo può essere ottenuto collegando una nuova variabile booleana alla porta di output e aggiungendo la clausola corrispondente a questa variabile. Ora che abbiamo dimostrato questa riduzione, possiamo facilmente dimostrare che altri problemi già noti sono NP-Completi. Avendo dimostrato $3-SAT$ e avendo dimostrato che da $3-SAT$ possiamo giungere a $SET-COVER$, automaticamente avremo dimostrato che anche $SET-COVER$ è NP-Completo. Infatti, proprio basandoci su questo asserto, andremo a dimostrato che altri problemi noti sono NP-Completi.
Prima di proseguire, dobbiamo soffermarci in un altro classe di problemi complementari, ossia $co-NP$. Un problema complementare a un problema in NP è il problema che richiede di trovare controesempi che dimostrano che una determinata soluzione proposta non è valida. In altre parole, un problema è in $co-NP$ se e solo se il suo complemento è in NP.

- **ES** = problema SAT. Si chiede se, data un'assegnazione di verità, una formula Φ è soddisfacibile. Ma se invece volessimo trovare una formula non-soddisfacibile?

Più che altro, dato un problema di decisione X, ci chiediamo se il suo complemento $\overline{X}$ è lo stesso, ma con le istanze `yes` e `no` invertite. Quindi, $NP = co-NP$? Questo è

un altro grande problema irrisolto. L'opinione comune è che non lo siano, anche perché se fosse vero, allora sarebbe vero anche $P = NP$, dato che P è chiusa rispetto al complemento e, a sua volta, lo sarebbe anche NP. Una cosa che è possibile verificare è che $NP \cap co - NP$, ossia se un problema X è sia NP che in $co - NP$. In altre parole, non esistono problemi per i quali sia possibile verificare una soluzione proposta in modo efficiente e anche trovare una soluzione in modo efficiente.
Concluse queste dimostrazioni, possiamo passare alla definizione di altri due problemi, strettamente correlati.

- $HAM - CYCLE$ = dato un grafo non orientato $G = (V, E)$, ci chiediamo se esiste un ciclo semplice Γ (**ciclo hamiltoniano**) che contiene ogni vertice di V.
- $DIR - HAM - CYCLE$ = dato un **digrafo**, quindi un grafo orientato, $G = (V, E)$, ci chiediamo se esiste un ciclo semplice Γ che contiene ogni vertice di V.

Si può facilmente dimostrare questa riduzione $DIR - HAM - CYCLE \leq_P HAM - CYCLE$. La riduzione non è eccessivamente complicata, anzi. Essendo un grafo orientato, ci troviamo di fronte a dei vertici che hanno archi entranti o uscenti. Per farlo diventare un grafo non orientato, ci basterà semplicemente sostituire ogni vertice con altri tre vertici. Per esempio, se avessimo il nodo v, dovremo avere v_{in}, v e v_{out}, a rappresentare rispettivamente l'input e l'output, quindi le direzioni delle frecce. In pratica, dato un grafo G di partenza, verrà costruito un grafo non orientato G' con $3n$ nodi. Ora, se volessimo dimostrare l'NP-Completezza di questi due problemi, ci tocca dover dimostrare un'altra riduzione, ossia $3 - SAT \leq_P DIR - HAM - CYCLE$. Sembrano due problemi estremamente differenti tra di loro. In $3 - SAT$ abbiamo vari letterali che possono assumere i valori di `true` o `false`. Dobbiamo costruire un grafo con due possibili opzioni per ogni variabile. Quindi, per ogni variabile, creiamo una linea di vertici, in modo bidirezionale collegati tra di loro. Il numero di vertici dipende dal numero di clausole, infatti bisognerà fare questo calcolo, $3k + 3$ per calcolare il numero totale di vertici. Per esempio, se avessimo due clausole, allora $3 * 2 + 3$, quindi 9 vertici per ogni linea. Dopodiché, aggiungiamo un nodo iniziale, chiamato arbitrariamente S, che si collega al primo e all'ultimo vertice della prima linea. Il primo vertice si collega, inoltre, al primo vertice della linea successiva e all'ultimo. Così come l'ultimo vertice si collega al primo della linea successiva e all'ultimo. Infine, il primo e ultimo nodo dell'ultima riga, si collegano ad un vertice finale, chiamato arbitrariamente T. Inoltre, aggiungiamo un arco che va da T ad S. Ci troveremo di fronte ad un grafo che ha 2^n cicli hamiltoniani, con n uguale al numero di letterali. Per concludere il grafo, dobbiamo aggiungere un vertice in più per ogni clausola. Per ogni letterale, vengono creati due archi nel grafo diretto: un

arco per la variabile stessa e un arco per la negazione della variabile. Questi archi rappresentano le assegnazioni possibili alla variabile.
Un altro problema di cui parleremo è il **problema del commesso viaggiatore**.

- TSP = dato un insieme di n città e di distanze $d(u, v)$, ci chiediamo se esiste un tour di lunghezza $\leq D$, ossia un tour che tocca tutte le città almeno una volta.

Quello che ci interessa è effettuare una riduzione $HAM-CYCLE \leq_P TSP$. È molto più semplice di quanto si creda, infatti noi partiamo da un'istanza di $HAM-CYCLE$, prendendo in considerazione $G = (V, E)$. Andiamo a creare n città, usando questa dicitura.

- $d(u, v) = \begin{cases} 1 & : (u, v) \in E \\ 2 & : (u, v) \notin E \end{cases}$

Otterremo un'istanza di TSP il cui tour ha lunghezza $\leq n$ se G è hamiltoniano.
Andiamo, ora, a definire altre classi di problemi.

- $K-COLOR$: dato un grafo non orientato G, ci chiediamo esiste un modo per colorare i vertici usando k colori in modo che non ci siano vertici adiacenti con lo stesso colore. Un caso particolare di questo problema è $3-COLOR$, che si chiede la stessa cosa, soltanto che i colori sono solo tre, arbitrariamente rosso, blu e verde.

Vogliamo effettuare questa riduzione $3-SAT \leq_P 3-COLOR$. Vogliamo, quindi, costruire un grafo che rispetti le specifiche di $3-COLOR$ se e solo se Φ è soddisfacibile. La prima cosa da fare è creare tre vertici e assemblarli in un triangolo. Ora, dobbiamo decidere quale colore assegnare a `true`, a `false` e ad un vertice chiamato `buffer`, che sarà una specie di cuscinetto. Ora, per ogni variabile e la sua negazione, creiamo due vertici nel grafo. Ovviamente, i letterale e le loro negazioni saranno collegati e poi, tutti i letterali andranno a collegarsi col vertice B. Questo garantisce che i letterali siano o true o false e non entrambe le cose. Ora, per ogni clausola, andremo ad introdurre un gadget nel nostro grafo. Un gadget, in questo caso, è una specie di sotto-grafo e contiene sei nodi e tredici archi. Ai lati del nostro gadget andiamo a piazzare due nodi, uno verde, T e uno rosso, F, che si collegano entrambi ai nodi estremi più in basso. Il nodo true si collega anche ai tre nodi superiori. Una volta fatta questa operazione, andremo ad assegnare valori di verità alle nostre variabili, andandole a colorare opportunamente. Starà a noi cercare un'istanza correte di $3-COLOR$, quindi nessuno nodo adiacente con lo stesso colore.
L'ultimo problema che andremo ad analizzare è un **problema numerico**:

- $SUBSET-SUM$: dato un insieme di numeri naturali $w_1, w_2, \ldots, w_n$ e un intero W, esiste un sottoinsieme la cui somma è esattamente W?

Ci interessa effettuare una riduzione $3-SAT \leq_P SUBSET-SUM$. Vogliamo assicurarci che la forma Φ sia soddisfacibile se e solo se esiste un sottoinsieme la cui somma è W. Questa è una delle riduzioni più lunghe, ma anche una delle meno complesse. Inizializziamo una tabella. Innanzitutto, le colonne saranno contrassegnate con i nomi di tutte le variabili e delle clausole, mentre nelle righe andranno messi tutti i possibili valori di verità delle variabili, quindi se c'è x_1, andremo a mettere x_1 e $\neg x_1$. Le celle saranno riempite nel seguente modo. Per le variabili, andiamo ad assegnare valori di `0` ed `1` in modo che venga scelta la variabile o la sua negazione. Nelle celle riservate alle clausole, in base ai valori che hanno la variabili nella Φ, riempiremo di `1` se l'assegnazione è vera e `0` se è falsa. A questa tabella, aggiungiamo poi altre righe. Quante? Deve avere $2n + 2k$ righe, dove n sono le variabili e k le clausole. Per esempio, 3 variabili e 3 clausole, la tabella dovrà avere 12 righe. Nelle colonne riservate alle variabili, riempiremo con `0`, mentre in quelle riservate alle clausole andremo ad apporre `1` e `2` a scaletta, per ogni clausola. Questo serve perché a noi interessa giungere al numero finale W, che ha questa forma $111\ldots444\ldots$. I punti segnalano che potrebbero esserci altri `1` e `4`, in base a variabili e clausole. A questo punto, assegniamo valori di verità alle nostre variabili e verifichiamo di giungere al nostro numero W in maniera corretta.