PER ALTRI APPUNTI CONSULTARE IL SITO:

https://luigi-v.github.io/Appunti_Universita/

Per altri appunti consultare il sito: *https://luigi-v.github.io/Appunti_Universita/*

By *Vollono Luigi*

Sia ***A*** un ***automa finito deterministico*** (***DFA***) è 5-tupla, M = (Q, Σ, f, q_0, F), dove:

1. Q è ***insieme finito*** di stati.
2. Σ è ***alfabeto***, e il DFA processa stringhe su Σ.
3. f: Q × Σ -> Q è la ***funzione di transizione*** (definisce le regole per il cambio di stato).
4. $q_0 \in$ Q è lo ***stato start (o iniziale)***.
5. F⊆Q è l'insieme di ***stati accettanti*** (o ***finali***).

Un ***automa finito non-deterministico NFA A*** è una 5-tupla M= (Q, ∑, f, q_0, F), con

1. Q è ***l'insieme finiti*** di stati
2. ∑ è un ***alfabeto***, e il DFA processa stringhe su Σ;
3. f: Q × $\sum_\varepsilon$ -> P(Q) ***funzione di transizione***, dove P(Q) è l'insieme di tutti i sottoinsiemi possibili di Q;
4. q_0∈Q è ***stato start***
5. F⊆Q è ***insieme di stati accettanti***.

Sia ***M = (Q, Σ, f, q_0, F),*** consideriamo la stringa ***w = w_1 w_2 … w_n su Σ***, dove ogni w_i in Σ, (o in $\sum\varepsilon$ se è un NFA):
Allora ***M accetta w*** se esiste una ***sequenza di stati r_0, r_1, …, r_n*** in Q con tre condizioni:

1. **$r_0 = q_0$** (primo stato della sequenza è quello iniziale)
2. **$r_n \in F$** (ultimo stato in sequenza è uno stato accettante)
3. **$f(r_i, w_{i+1}) = r_{i+1}$, per ogni i = 0,1, 2, . . . , n-1** (sequenza di stati corrisponde a transizioni valide per la stringa w).

Verificate le tre condizioni, si può affermare che M ***riconosce il linguaggio*** A, se A = {w | M accetta w}.

Complemento:

Se si ha il linguaggio L su alfabeto Σ, ha un DFA M= (Q, Σ, f, q_1, F), allora il DFA per il ***complemento*** di C(L) è: ***M'=(Q, Σ, f, q_1, Q-F)***.

L' insieme dei linguaggi regolari è ***chiuso*** per l'operazione di ***complemento***.

Dim:

Abbiamo visto che dati DFA M_1 per Linguaggio L, possiamo costruire DFA M_2 per Linguaggio complemento L':

- Rendi tutti stati accetta in M_1 in non-accetta in M_2.
- Rendi tutti stati non-accetta in M_1 in accetta in M_2.

Quindi L regolare → C(L) regolare.

Intersezione:

La classe dei linguaggi regolari è ***chiusa*** per ***l'intersezione***. Cioè, se L_1 e L_2 sono **linguaggi regolari**, allora lo ***è $L_1 \cap L_2$***.

Dim:

A tal punto, siano L_1 e L_2 definiti su uno stesso alfabeto Σ. Supponiamo che il DFA M_1 riconosce L_1, dove $M_1 = (Q_1, \Sigma, f_1, q_1, F_1)$, e che il DFA M_2 riconosce L_2, dove $M_2 = (Q_2, \Sigma, f_2, q_2, F_2)$. Costruiamo il DFA $M_3 = (Q_3, \Sigma, f_3, q_3, F_3)$ in questo modo:

- $Q_3 = Q_1 \times Q_2 = \{(x, y) \mid x \text{ in } Q_1, y \text{ in } Q_2\}$;
- l'alfabeto di M_3 è Σ;
- M_3 ha f_3: $Q_3 \times \Sigma \rightarrow Q_3$ tale che per ogni x in Q_1, y in Q_2, a in Σ: $f_3((x, y), a) = (f_1(x, a), f_2(y, a))$;
- lo stato iniziale di M_3 è $q_3 = (q_1, q_2)$ in Q_3;
- l'insieme di stati accetta di M_3 è $F_3 = \{(x, y) \text{ in } Q_3 \mid x \text{ in } F_1$ **e** $y \text{ in } F_2\}$.

M_3 è un DFA che riconosce l'intersezione. Poiché $Q_3 = Q_1 \times Q_2$, allora il numero di stati in M_3 è $|Q_3| = |Q_1| \cdot |Q_2|$.

Unione:

La classe dei ***linguaggi regolari*** è ***chiusa*** per l'operazione di ***unione***, cioè, se L_1 e L_2 sono linguaggi regolari, allora lo è anche $L_1 \cup L_2$.

Dim:

Supponiamo che M_1 riconosca L_1, dove $M_1 = (Q_1, \sum, f_1, q_1, F_1)$ ed M_2 riconosca L_2, dove $M_2 = (Q_2, \sum, f_2, q_2, F_2)$.

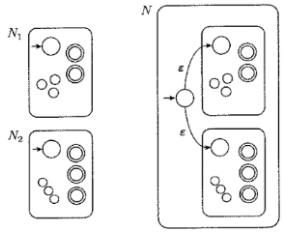
Costruiamo M_3 che riconosce $L_1 \cup L_2$, dove $M_3 = (Q_3, \sum, f_3, q_3, F_3)$:

1. $Q_3 = Q_1 \times Q_2 = \{(x, y) \mid x \in Q_1, y \in Q_2\}$; (*Prodotto cartesiano di Q_1 e Q_2*)
2. Alfabeto di M_3 è ∑, ovvero lo stesso di M_1 ed M_2. (*Se gli alfabeti sono diversi, era: $\sum = \sum_1 \cup \sum_2$*)
3. M_3 ha funzione di transizione f_3: $Q_3 \times \sum$ -> Q_3, tale che per ogni x in Q_1, y in Q_2 ed a in ∑: $f_3((x, y), a) = (f_1(x, a), f_2(y, a))$;
4. Lo stato start di M_3 è $q_3 = (q_1, q_2)$ in Q_3.
5. L' insieme di stati accetta di M_3 è $F_3 = \{(x, y) \text{ in } Q_3 \mid x \text{ in } F_1$ **o** $y \text{ in } F_2\}$

Poiché $Q_3 = Q_1 \times Q_2$, il numero di stati in M_3 è $|Q_3| = |Q_1| \cdot |Q_2|$.

Oppure tramite NFA:

La macchina N deve accettare il suo input se N_1 o N_2 accetta questo input. La nuova macchina ha un nuovo stato iniziale che si dirama negli stati iniziali delle vecchie macchine con ε-archi. Se una di essi accetta, allora anche N lo accetterà.



Sia $N_1 = (Q_1, \sum, f_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \sum, f_2, q_2, F_2)$ che riconosce A_2.

Costruiamo $N = (Q, \sum, f, q, F)$ per $A_1 \cup A_2$:

1. $Q = \{q_0\} \cup Q_1 \cup Q_2$
   Gli stati di N sono tutti gli stati di N_1 ed N_2, con l'aggiunta di un nuovo stato iniziale q_0.
2. Lo stato q_0 è lo stato iniziale di N.
3. L'insieme degli stati accettanti $F = F_1 \cup F_2$
   Gli stati accettanti di N sono tutti gli stati accettanti di N_1 ed N_2. In questo modo, N accetta se N_1 accetta o N_2 accetta.
4. Definiamo f in modo che per ogni *q* in *Q* e ogni *a* in $\sum_\varepsilon$:

$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \\ \delta_2(q,a) & q \in Q_2 \\ \{q_1, q_2\} & q = q_0 \text{ e } a = \varepsilon \\ \emptyset & q = q_0 \text{ e } a \neq \varepsilon. \end{cases}$$

Concatenazione:

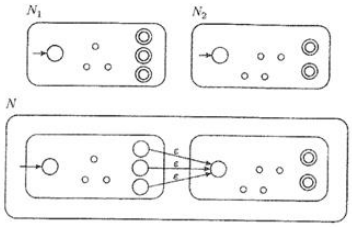
La classe dei ***linguaggi regolari*** è ***chiusa*** per la ***concatenazione***. Cioè, se L_1 e L_2 sono linguaggi regolari, allora lo è anche L_1L_2.

Dim:

Poniamo come stato iniziale di N, lo stato iniziale di N_1. Gli stati accettanti di N_1 hanno degli ulteriori ε-archi che permettono di diramarsi in N_2 ogni volta che N_1 è in uno stato accettante. Gli stati accettanti di N sono solo gli stati accettanti di N_2, esso accetta quando l'input può essere diviso in due parti, la prima accettata da N_1 e la seconda da N_2.

Sia $N_1 = (Q_1, \sum, f_1, q_1, F_1)$ che riconosce A_1 ed $N_2 = (Q_2, \sum, f_2, q_2, F_2)$ che riconosce A_2. Costruiamo $N = (Q, \sum, f, q, F)$ per A_1 o A_2:

1. $Q = Q_1 \cup Q_2$
   Gli stati di N sono tutti gli stati di N_1 ed N_2.
2. L'alfabeto $\sum$ è lo stesso dei due linguaggi L_1 ed L_2.
3. Lo stato q_0 è uguale allo stato iniziale di N_1.
4. Gli stati accettanti F_2 sono uguali agli stati accettanti di N_2.
5. Definiamo f in modo che per ogni *q* in *Q* e ogni *a* in $\sum_\varepsilon$:



$$\delta(q,a) = \begin{cases} \delta_1(q,a) & q \in Q_1 \text{ e } q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \text{ e } a \neq \varepsilon \\ \delta_1(q,a) \cup \{q_2\} & q \in F_1 \text{ e } a = \varepsilon \\ \delta_2(q,a) & q \in Q_2. \end{cases}$$

Kleene Star:

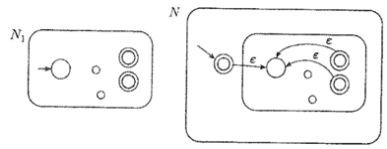
La classe dei ***linguaggi regolari*** è chiusa rispetto all'operazione ***star***.

Dim:

Possiamo costruire N come N_1 con ε-archi supplementari che dagli stati accettanti ritornano allo stato iniziale. Quando l'elaborazione giunge alla fine di una parte che N_1 accetta, la macchina N ha la scelta di tornare indietro allo stato iniziale per provare a leggere un'altra parte che N_1 accetta. Inoltre, si aggiunge un nuovo stato iniziale, che è anche uno stato accettante, e che ha un ε-arco entrante nel vecchio stato iniziale.

Sia $N_1 = (Q_1, \sum, f_1, q_1, F_1)$ che riconosce A_1. Costruiamo $N = (Q, \sum, f, q_0, F)$ per A_1^*:

1. $Q = \{q_0\} \cup Q_1$
   Gli stati di N sono tutti gli stati di N_1 più un nuovo stato iniziale.
2. Lo stato q_0 è il nuovo stato iniziale.
3. $F = \{q_0\} \cup F_1$
   Gli stati accettanti sono i vecchi stati accettanti più il nuovo stato iniziale.
4. Definiamo f in modo che per ogni *q* in *Q* e ogni *a* in $\sum_\varepsilon$:



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

Per ogni automa finito non deterministico NFA esiste un automa finito deterministico DFA ***equivalente***.
Ogni NFA N ha un equivalente DFA M, cioè se N è un NFA, allora esiste DFA M t.c. ***L(M) = L(N).***

Sia L = L(N) per un NFA $N = (Q_N, \Sigma, f_N, q_N, F_N)$; allora, costruiamo un DFA $D = (Q_D, \Sigma, f_D, q_D, F_D)$.

Partiamo da un automa D' che non considera le ε-transition, dove, per ogni $r \in Q$ e $a \in \Sigma$:

- $Q = P(Q_N)$, ossia ogni stato in D' sarà un sottoinsieme dell'insieme potenza degli stati del NFA N;
- $f(r, a) = \cup_{r \in R} f_N(r, a)$, ossia la funzione di transizione darà come risultato l'unione dei risultati (per ogni elemento r dell'insieme R), della funzione di transizione $f_N(r, a)$;
- $q = \{q_N\}$, ossia lo stato iniziale sarà l'insieme in cui compare lo stato iniziale del NFA N;
- $f = \{r \in Q \mid r \cap F_N \neq \emptyset\}$, ossia gli stati finali saranno tutti quegli stati che contengono al loro interno uno stato finale del NFA N.

Per ogni stato in f_N, se vi è una ε-transition dobbiamo definire l'insieme **E(r)** = r ∪ {q | q è raggiungibile da uno stato in r con 1 o più archi labellati ε}, cioè l'insieme che considera l'unione tra uno stato di D' (in quanto $r \in Q$) e l'insieme degli stati raggiungibili da uno stato in r che ammettono almeno una ε-transition.

Per la funzione di transizione estesa, si ha che:

- $f_{N*}(q_N,\varepsilon)=E(\{q_N\})$, quindi non ci sono lettere lette ed applichiamo la sola epsilon-transition;
- $f_{N*}(q_N,xa)=\cup E(f_N(r,a))_{r \in f_{N*}(q_N,x)}$, quindi si prendono tutti gli stati in cui potrebbe trovarsi l'automa dopo aver letto la stringa $x \in \Sigma^*$, e per ognuno di questi stati applichiamo la funzione di transizione del NFA, applichiamo poi la funzione E, facciamo l'unione di ciò che abbiamo ottenuto ed otteniamo l'insieme di stati raggiungibili quando l'input è la stringa xa.

Risulta, per ogni $x \in \Sigma^*$, $f_{D*}(q_D,x)=f_{N*}(q_N,x)$, cioè che le funzioni di transizione estese del NFA e del DFA corrispondente coincidono (ciò significa che i due automi accettano le stesse stringhe); questo risultato si può dimostrare induttivamente. Quindi, sappiamo che D simula N su ogni input x. Inoltre, D accetta x se e solo se N accetta x. Infine, il linguaggio L = L(N) = L(D). Ciò significa che D e N sono equivalenti.

---

2 – Espressione regolare

Definizione induttiva di espressione regolare (e.r.):

BASE:

- Se ε e ∅ sono espressioni regolari: L(ε) = {ε} e L(∅) = ∅;
- Ed $a \in \Sigma$, allora a è un'espressione regolare: L(a) = {a}.

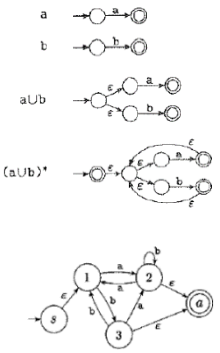
PASSO: Se R_1 e R_2 sono espressioni regolari, allora:

- $R_1 \cup R_2$ è un'espressione regolare che rappresenta $L(R_1) \cup L(R_2)$;
- R_1R_2 è un'espressione regolare che rappresenta $L(R_1)L(R_2)$;
- R_1^* è un'espressione regolare che rappresenta $(L(R_1))^*$.

Teorema di Kleene:

Un linguaggio è ***regolare*** se e solo se qualche ***espressione regolare*** lo descrive: L ↔ E.R.

Lemma1. Se un linguaggio è descritto da un'***espressione regolare***, allora esso è ***regolare***.



Dim:

Data una espressione regolare, la si scompone in sotto-espressioni più semplici (aiutandoci con una struttura ad albero) e si costruiscono una serie di NFA che riconoscono queste sotto-espressioni semplici.

Con quest'ultimi costruiamo altri NFA, tramite operazioni regolari con gli automi, che riconoscono sotto-espressioni più complesse, fino ad arrivare a costruire un NFA che riconosca l'espressione regolare data.

Lemma2. Se un linguaggio è ***regolare***, allora è descritto da un'***espressione regolare***.

Dim:

Dato un automa A, costruiamo un nuovo automa avente un nuovo stato iniziale senza archi entranti ed un nuovo stato finale senza archi uscenti. Si rimuove uno stato alla volta, che non siano i nuovi stati inseriti, modificando le etichette sugli archi in modo che l'automa risultante accetti le stesse stringhe, fino ad avere solo i due nuovi stati inseriti inizialmente.

Pumping Lemma:

Se ***A*** è un ***linguaggio regolare***, allora esiste un numero ***p*** (***lunghezza del pumping***) tale che ***s*** è una qualsiasi stringa in A di lunghezza almeno ***p***, allora ***s*** può essere divisa in ***tre parti***, ***s = x y z***, che soddisfa le seguenti condizioni:

1. ***Per ogni i ≥ 0, $xy^iz \in A$;***
2. ***$|y| > 0$;***
3. ***$|xy| \leq p$***.

3 - MdT

Una ***macchina di Turing*** è una 7-tupla (Q, ∑, Γ, f, q_0, q_{accept}, q_{reject}), dove Q, ∑, Γ sono tutti insiemi finiti e:

1. ***Q*** è ***l'insieme degli stati***;
2. ***∑*** è ***l'alfabeto di input*** non contenente il ***simbolo blank*** '***_***';
3. ***Γ*** è ***l'alfabeto del nastro*** con _ ∈ Γ e ∑ ⊆ Γ contenente tutti i simboli che possono essere scritti all'interno di una cella di memoria (tecnicamente, l'unione tra l'alfabeto di lavoro e l'insieme dei simboli non processabili dalla macchina, come _);
4. ***F: Q x Γ → Q x Γx {L, R}*** è la ***funzione di transizione***, dove le lettere lette sono all'interno del nastro (in ogni istante si hanno dei simboli nelle varie celle, e la cella a cui punta la testina rappresenta lo stato q in cui si trova la macchina).;
5. ***q_0*** ∈ Q è lo ***stato iniziale***;
6. ***q_{accept}*** ∈ Q è lo ***stato di accettazione***;

Una **configurazione** di una Macchina di Turing è una descrizione concisa di stato e contenuto del nastro. Trattasi di una stringa C = *uqv*, dove:
- *q* è lo *stato* occupato dalla macchina M;
- *uv* è il *contenuto del nastro* (sinistra – destra);
- la testina punta sul primo (cioè, più a sinistra) simbolo di *v* (su primo blank _ se v = ε);
- dopo *v* sono presenti solo simboli blank _.

Una ***MdT M*** accetta una ***stringa w*** se esiste una ***computazione*** (***sequenza di configurazione***) di M: $C_1, \dots C_k$ tale che

1. $C_1 = q_0w$ è la configurazione iniziale di M con input w;
2. Ogni C_i produce C_{i+1} per ogni i=1, …, k-1;
3. C_k è una configurazione di accept.

Un linguaggio si dice ***Turing-riconoscibile*** se esiste una macchina di Turing che lo riconosce.

Formalmente, un linguaggio L si dice Turing riconoscibile se esiste una Macchina di Turing M tale che L(M) = L. Di conseguenza, la macchina accetta tutte le stringhe del linguaggio. Se, invece, una stringa w∉L, allora la stringa può essere o *rifiutata* o mandare la macchina *in loop*.

Possiamo evitare il loop costruendo le macchine che si fermano (accettando o rifiutando) *su ogni input*: trattasi dei ***deciders***.
Si dice che un decider *decide* il linguaggio L se esso riconosce L.

Un linguaggio si dice ***Turing-decidibile*** o semplicemente ***decidibile*** se esiste una macchina di Turing che lo decide.

Formalmente, un linguaggio L si dice Turing decidibile se esiste una Macchina di Turing M tale che L(M) = L e M è un ***decider***.

La ***differenza tra un linguaggio L Turing riconoscibile e Turing decidibile*** risiede nel fatto che i primi possono mandare le MdT che li riconoscono in loop, mentre i secondi possono far sì che le MdT che li decidono o accettino o rifiutino le loro stringhe *su ogni input*.

Uno **stayer** è una Macchina di Turing la cui testina può rimanere sulla stessa cella del nastro durante una transizione; formalmente, essa è la stessa settupla M = (Q, Σ, Γ, δ, q_0, q_{accept}, q_{reject}), con la sola differenza che la funzione di transizione è δ: Q × Γ → Q × Γ × {*L, S, R*}, dove **S** indica che la testina resta ferma durante la transizione (S sta per "stay").

Una MdT **multinastro** (a k nastri) è una normale Macchina di Turing avente k nastri. Inizialmente, l'input compare sul primo nastro e gli altri sono vuoti. La sua funzione di transizione è **$\delta: Q \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{L, S, R\}^k$**; dunque, essa muove (in modo indipendente) le testine dei vari nastri.
Questa variante è simile ad una MdT convenzionale: usa k nastri, con k ≥ 1. Ciascun nastro ha una propria testina, e con una mossa si specificano (oltre al nuovo stato): i k simboli letti, i k simboli da scrivere e i k movimenti delle k testine. Come configurazione iniziale, essa avrà l'input sul primo nastro ed i rimanenti nastri saranno vuoti.

MdT e MdT multinastro sono modelli equivalenti.

Dim:

L'implicazione diretta è ovvia, in quanto una MdT è una MdT multinastro, con k = 1. Vogliamo dimostrare che per ogni MdT multinastro è possibile costruire una MdT convenzionale che riconosce lo stesso linguaggio. Effettuiamo una simulazione, utilizziamo una MdT multinastro, con k=3.
Una soluzione è immaginare i k nastri affiancati, ognuno con indicata la posizione della testina. Dobbiamo codificare questa informazione su un solo nastro. Per fare ciò possiamo concatenare il contenuto dei k nastri, su k blocchi consecutivi separati da un carattere particolare (con #):
- ogni blocco avrà lunghezza variabile che dipende dal contenuto del nastro corrispondente;
- un elemento marcato (con ˙) nel blocco i-esimo indica la posizione della testina i-esima (ad esempio, se la testina S punta ad un elemento del primo blocco e legge γ˙, allora la testina del primo nastro è in questa posizione e legge γ);
- usiamo un alfabeto esteso Γ_2 tale che γ˙ ∈ Γ_2 per ogni γ ∈ Γ.

5 - DECIDIBILITÀ

Un insieme è ***numerabile*** se ha la stessa cardinalità di un sottoinsieme di $\mathbb{N}$.

Se A è numerabile, allora possiamo "numerare" gli elementi di *A* e scrivere una lista (a_1, a_2, …), cioè, per ogni numero naturale *i*, allora possiamo specificare l'elemento *i*-esimo della lista.

In generale, *per determinare se un insieme è numerabile bisogna mostrare che esiste una biiezione con un sottoinsieme di* $\mathbb{N}$, cioè per ogni elemento possiamo far vedere che posizione occupa all'interno dell'insieme.

L'insieme Σ* è numerabile: per dimostrarlo, listiamo prima la stringa vuota, poi le stringhe (in ordine lessicografico) lunghe 1, poi 2, e così via. A questo punto possiamo numerare la lista (partendo da 1) e quindi *numerare* l'insieme Σ*.

L'insieme delle descrizioni di MdT $\{\langle M\rangle \mid M$ è una MdT sull'alfabeto Σ$\}$ è numerabile: è possibile codificare una MdT *M* con una stringa su un alfabeto Σ.

L'insieme dei numeri reali R non è numerabile:

Sia per assurdo $\mathbb{R}$ numerabile; allora possiamo costruire la lista f(1), f(2), f(3), …

Per ogni $i \geq 1$, scriviamo $f(i) = f_0(i),f_1(i)f_2(i)f_3(i)$… Cioè, sappiamo che ogni f(*i*) è un numero reale, ed essendo tale ha una parte intera ed una parte decimale. Nella rappresentazione precedente, $f_0(i)$ è la parte intera, separata da una virgola dalla parte decimale $f_1(i)f_2(i)f_3(i)$…

Organizziamo in una matrice in cui le colonne sono indicizzate con gli interi 1, 2, 3, …, *i* e la riga *i*-esima è l'elemento f(*i*) che compare nella lista, nella tabella inseriamo la parte decimale di ogni numero, ignorando la parte intera.

Consideriamo la diagonale principale di questa matrice e sia $x \in (0, 1)$ il numero $x=0,x_1x_2x_3…x_i…$ ottenuto scegliendo $x_i \neq f_i(i)$ per ogni $i \geq 1$.

Vediamo che $x \in \mathbb{R}$ quindi verifichiamo che x è nella lista. Se $x = f(j)$, allora il suo *j*-esimo digit soddisfa $x_j = f_j(j)$: ma $x_j \neq f_j(j)$ per definizione di *x*. Questa è una contraddizione. Quindi, $x \in \mathbb{R}$ non può comparire nella lista e $\mathbb{R}$ non è numerabile.

---

6 - RIDUCIBILITÀ

Un linguaggio *A* è ***riducibile a un linguaggio*** *B* ($\boldsymbol{A \leq_m B}$) se esiste una funzione calcolabile $f: \Sigma^* \to \Sigma^*$ tale che $\forall w,\ w \in A \Leftrightarrow f(w) \in B$.

Una funzione $f: \Sigma^* \to \Sigma^*$ è ***calcolabile*** se esiste una MdT *M* tale che, su ogni input *w*, *M* si arresta con f(*w*) (e solo con f(*w*)) sul suo nastro.

Cioè, una funzione è calcolabile se esiste una Macchina di Turing che la calcola.

$$A_{TM} = \{\langle M, w\rangle \mid M \text{ è una MdT e } w \in L(M)\},$$
$$E_{TM} = \{\langle M\rangle \mid M \text{ è una MdT e } L(M) = \emptyset\},$$
$$REGULAR_{TM} = \{\langle M\rangle \mid M \text{ è una MdT e } L(M) \text{ è regolare}\},$$
$$EQ_{TM} = \{\langle M_1, M_2\rangle \mid M_1, M_2 \text{ sono MdT e } L(M_1) = L(M_2)\}.$$

Teorema di Rice:

Sia $\boldsymbol{L_P = \{\langle M\rangle \mid M \text{ è una MdT che verifica la proprietà } P\}}$ un linguaggio che soddisfa le seguenti due condizioni:

1. L'appartenenza di M a L_P dipende solo da L(M), cioè: $\boldsymbol{\forall M_1, M_2 \text{ MdT tali che } L(M_1)=L(M_2), \langle M_1\rangle \in L_P \Leftrightarrow \langle M_2\rangle \in L_P}$
2. L_P è un ***problema non banale***, cioè: $\boldsymbol{\exists M_1, M_2 \text{ MdT tali che } \langle M_1\rangle \in L_P, \langle M_2\rangle \notin L_P}$

allora L_P ***è indecidibile***.

Idea dimostrativa di Rice:

A_{TM} si riduce a L se esiste una funzione calcolabile $f(\langle M, w\rangle) = \langle S\rangle$, tale che $\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle S\rangle \in L$. La MdT S su input x simula M su input w:

- Se M accetta w, allora S accetta x;
- Se M rifiuta w, allora S rifiuta x.

Mostriamo che f è una riduzione:

- f è calcolabile poiché semplicemente simula la MdT in input.
- Mostriamo che $\langle M, w\rangle \in A_{TM} \Leftrightarrow \langle S\rangle \in L$:
  - $\langle M, w\rangle \in A_{TM} \Rightarrow$ M accetta w $\Rightarrow$ S accetta x $\Rightarrow L(S) = \Sigma^* \Rightarrow \langle S\rangle \in L$
  - $\langle M, w\rangle \notin A_{TM} \Rightarrow$ M rifiuta w $\Rightarrow$ S rifiuta x $\Rightarrow L(S) = \emptyset \Rightarrow \langle S\rangle \notin L$

---

8 - COMPLESSITÀ

Classe P: È costituita da tutti i problemi che ammettono un ***algoritmo polinomiale*** che li risolve.

Classe NP: È costituita da tutti i problemi per cui esiste un ***certificatore polinomiale***.

Dove C(s,t) è un certificatore per X se per ogni istanza s di X, abbiamo $s \in X \Leftrightarrow \exists$ un certificatore t per cui C(s,t)="si".

Classe EXP: È costituita da tutti i problemi che ammettono un ***algoritmo esponenziale*** che li risolve.

Classe NP-Completi: Problemi in NP, poiché per definizione un problema X è NP-Completo se $X \in NP$ e per tutti $Y \in NP$, $Y \leq_P X$.

Classe CO-NP: Un problema $X \in$ CO-NP se e solo se il problema complemento $\overline{X} \in NP$.

Ovvero sono tutti quei problemi dove ogni istanza s di X, abbiamo $s \in \overline{X} \Leftrightarrow \forall$ certificatore t per cui C(s,t)="no".

Se $X \leq_P Y$ e $Y \leq_P Z$, allora $X \leq_P Z$:

Eseguiamo l'algoritmo per X usando un oracolo per Y; ma ogni volta che viene chiamato l'oracolo per Y, lo simuliamo in un numero polinomiale di passaggi usando l'algoritmo che risolve istanze di Y, ma in tal caso stiamo usando un oracolo per Z.

P⊆NP:

Dato un problema $X \in P$, allora esiste un algoritmo polinomiale A che risolve X. Per dimostrare che $X \in NP$, dobbiamo dimostrare che esiste un certificatore efficiente C per X, progettiamo C come segue:

Quando C viene eseguito con una coppia di input (s, t), il certificatore C restituisce il valore di A(s). C è un certificatore efficiente per X perché ha un tempo polinomiale, poiché esegue A (che ha un tempo polinomiale). Quindi, se una stringa $s \in X$, allora per ogni t abbiamo C(s, t)=sì. D'altra parte, se $s \notin X$, allora per ogni t abbiamo C(s, t) = no.

P=NP:

(⇐) Chiaramente, se P = NP, allora X può essere risolto in tempo polinomiale poiché appartiene a NP.

(⇒) Al contrario, supponiamo che X possa essere risolto in tempo polinomiale.

Se Y è un altro problema in NP, allora $Y \leq_P X$, e quindi Y può essere risolto in tempo polinomiale.

Ciò implica NP⊆P, ma già sappiamo che P⊆NP, e quindi P=NP.

Se NP≠co-NP, allora P≠NP:

Dimostriamo l'affermazione contrapposta, ovvero: se P = NP allora NP = co-NP.

Il punto è che P è chiusa per il complemento, quindi se P = NP, allora anche NP è chiusa per il complemento. Partendo dal presupposto P=NP, abbiamo:

$$X \in NP \Rightarrow X \in P \Rightarrow \overline{X} \in P \Rightarrow \overline{X} \in NP \Rightarrow X \in \text{co-NP} \qquad \text{e} \qquad X \in \text{co-NP} \Rightarrow \overline{X} \in NP \Rightarrow \overline{X} \in P \Rightarrow X \in P \Rightarrow X \in NP.$$

Quindi ne conseguirebbe che NP ⊆ co-NP e co-NP ⊆ NP, da cui NP = co-NP.

Riduzione Polinomiale:
Il problema X ***si riduce in modo polinomiale*** (Cook) al problema Y se istanze arbitrarie del problema X possono essere risolte usando:
Un numero polinomiale di passi di computazione standard, più un numero polinomiale di chiamate ad oracolo che risolve il problema Y.
Il problema X ***si trasforma in modo polinomiale*** (Karp) al problema Y se dato un qualsiasi input $x \in X$, possiamo costruire un input y tale che x è istanza "si" di X se e solo se y è istanza "si" di Y.

INDEPENDENT-SET:
Dato un grafo G = (V, E), diciamo che un sottoinsieme di nodi $S \subseteq V$ è un Indipendent-Set se non ci sono due nodi in S uniti da un arco. Il problema è:
Dato un grafo G = (V, E) e un intero k, esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \geq k$, e per ogni arco, ***almeno*** uno dei suoi estremi è in S?
VERTEX-COVER:
Dato un grafo G = (V, E), diciamo che un sottoinsieme di nodi $S \subseteq V$ è una Vertex Cover se ogni arco $e \in E$ ha almeno un'estremità (dell'arco) in S.
Dato un grafo G = (V, E) e un intero k, esiste un sottoinsieme di vertici $S \subseteq V$ tale che $|S| \leq k$, e per ogni arco, ***al più*** uno dei suoi estremi è in S?
VERTEX-COVER $\equiv_P$ INDEPENDENT-SET:
Mostriamo che ***S è un Independent-Set se e solo se il suo complemento V-S è un Vertex-Cover***. Mostrandolo nelle due direzioni:
(⇒) **INDEPENDENT-SET $\leq_P$ VERTEX-COVER**
Supponiamo che S sia un Independent Set e consideriamo un arco arbitrario e = (u, v), poiché S contiene vertici indipendenti, non è possibile che sia u e sia v siano in S quindi uno di essi deve essere per forza in V-S.
Ne consegue che ogni arco ha almeno un'estremità in V - S, e quindi V - S è una Vertex Cover.
(⇐) **VERTEX-COVER $\leq_P$ INDEPENDENT-SET**
Supponiamo che V - S sia una Vertex Cover e consideriamo due nodi u e v in S, se fossero uniti da un arco *e*, allora nessuna delle estremità di *e* risiederebbe in V - S, contraddicendo la nostra ipotesi che V - S sia una Vertex Cover.
Ne consegue che due nodi in S non sono uniti da un arco *e*, quindi S è un Independent Set.

SET-COVER:
Dato un insieme di elementi {1, …, n} (chiamato universo U) e una raccolta S di m sottoinsiemi (S_1, …, S_m) la cui unione è uguale ad U, il problema è:
Dato un insieme di elementi U, collezione S_1, …, S_m di sottoinsiemi di U, e un intero k, esiste una collezione $\leq k$ di questi sottoinsiemi la cui unione è U?
VERTEX-COVER $\leq_P$ SET-COVER:
Il nostro obiettivo è quello di coprire gli archi in E, quindi formuliamo un'istanza di Set Cover in cui l'insieme U è uguale ad E.
Ogni volta che selezioniamo un vertice nel Vertex Cover, copriamo tutti gli archi incidenti ad esso; quindi, per ogni vertice $v \in V$, aggiungiamo un insieme $S_v \subseteq U$ all'istanza Set Cover, costituito da tutti gli archi in G incidenti a v.
Ora ricordiamo che ***U può essere coperto con al massimo k degli insiemi S_1, …, S_n se e solo se G ha una Vertex Cover di dimensioni al massimo k.***
(⇒) Se S_{v1}, …, S_{vl} sono l ≤ k insiemi che coprono U, allora ogni arco in G è incidente a uno dei vertici v_1, …, v_l, e quindi l'insieme $\{v_1, …, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$.
(⇐) Al contrario, se $\{v_1, …, v_l\}$ è un Vertex Cover in G di dimensione $l \leq k$, allora gli insiemi S_{v1}, …, S_{vl} coprono U.

3-SAT:
Definiamo ***letterale*** una variabile booleana o il suo negato, e una ***clausola C_j*** che consiste in diversi letterali connessi tramite operatore **OR (∨)**, una formula booleana è in ***forma cnf***, se comprende diverse clausole connesse tramite operatore **AND (∧)**, mentre è una ***formula 3cnf*** se tutte le clausole della formula hanno esattamente tre letterali. Questo problema di verificare se una formula booleana è soddisfacibile, ovvero che valga 1 per un dato assegnamento di valori di verità. Il problema è il seguente:
Dato un insieme di clausole C_1, …, C_k, ciascuno di lunghezza 3, su un insieme di variabili $X = \{x_1, …, x_n\}$, esiste un assegnamento di verità soddisfacente?
3-SAT$\leq_P$Indipendent-Set:
Dobbiamo convertire la formula booleana in un grafo. Diciamo che due termini sono in "conflitto" se una variabile x_i ha in comune un arco col suo negato $\overline{x}_i$. Costruiamo il grafo G che contiene 3 vertici per ogni clausola, uno per ogni letterale, connettiamo i 3 letterali in una clausola (che forma un triangolo), ed infine connettiamo ogni letterale ad ogni suo negato. Un modo di risolvere un'istanza di 3-SAT è scegliere un termine da ciascuna clausola, non quelli in "conflitto", e uguagliarli ad 1, in modo da soddisfare tutte le clausole.
Possiamo dire quindi che ***G contiene un insieme indipendente di dimensione k=|φ| sse φ è soddisfacibile***.
(⇐) Supponiamo che G abbia un insieme indipendente S di dimensioni esattamente k, e deve consistere in un nodo per ogni triangolo.
Diciamo che esiste un'assegnazione di verità v per le variabili nell'istanza 3-SAT con la proprietà che le etichette di tutti i nodi in S valgono 1. Se un nodo in S fosse etichettato $\overline{x}_i$ e un altro x_i, allora ci sarebbe un arco tra questi due nodi, contraddicendo la nostra ipotesi. Pertanto, se x_i appare come un'etichetta di un nodo in S, impostiamo $v(x_i)=1$, altrimenti impostiamo $v(x_i)=0$ così che tutte le etichette dei nodi in S valgono 1.
(⇒) Se l'istanza 3-SAT è soddisfacibile, allora ogni triangolo nel grafo contiene almeno un nodo che ha valore 1. Sia S un insieme costituito da uno di questi nodi da ciascun triangolo. Sosteniamo che S è indipendente, poiché se ci fosse un arco tra due nodi u, v ∈ S, allora le etichette di u e v dovrebbero essere in conflitto (uno 0 e l'altro 1), ma questo non è possibile, poiché entrambi i vertici valgono 1.

CIRCUIT SATISFIABILITY:
Dato un circuito fisico K definiamolo come un grafo aciclico marcato e diretto, dove le foglie di sinistra sono etichettate con una delle costanti 0 o 1 e a destra con il nome di variabili distinte che rappresentano l'input al circuito. Ogni altro nodo è etichettato con uno degli operatori booleani ∧, ∨ o ¬.
Esiste un singolo nodo senza archi in uscita che rappresenta l'output: il risultato che viene calcolato dal circuito, ovvero la radice dell'albero.
CIRCUIT-SAT$\leq_P$3-SAT:
Quindi consideriamo un circuito arbitrario K ed associamo delle variabili 3-SAT x_v a ciascun nodo v del circuito. Ora dobbiamo codificare i nodi etichettati con operatori booleani, in modo da soddisfare le clausole.
A seconda della tipologia delle operazioni booleane, i nodi verranno trasformati in clausole in un modo preciso. Per i nodi etichettati come costanti, verranno aggiunte alle clausole con variabili x_5, se la costante è 1, o $\overline{x}_5$, se la costante è 0. Per il nodo di output, invece, aggiungiamo una clausola con la singola variabile, la quale richiederà che valga 1, concludendo la costruzione. Finora abbiamo creato una istanza SAT, per farla diventare 3-SAT inseriamo altre variabili nelle clausole e assegniamo il loro valore in modo che non cambi il risultato.

CLIQUE:
Un grafo G=(V,E) contiene una CLIQUE di dimensione k se esistono $v_1,…,v_k \in V$ tali che $(v_i,v_j) \in E$ per ogni i,j=1,…,k con $i \neq j$. Il problema consiste in:
Dato un grafo G e un intero k, G contiene una CLIQUE di dimensione k?

INDEPENDENT-SET ≤p CLIQUE:

Dato un grafo G=(V,E) e un intero k, S è un Independent-Set su G se $S \subseteq V$ con $|S|=k$ e per ogni e = (u,v) ∈ E, al più u o v sono in S.
Costruisco $G^c = (V, E^c)$ e per ogni i,j, in V $(i,j) \in E^c$ se e solo se $(i,j) \notin E$. S è un Independent-Set in G se e solo se S è una CLIQUE in G^c.

(⇒) Sia S un INDEPENDENT-SET in G con $|S|=k$, per definizione di Independent-Set, per ogni $u,v \in S$, $(u,v) \notin E$.
Per costruzione $(u,v) \in E^c$, generalizzando, S è una CLIQUE in G^c.

(⇐) Sia S una CLIQUE in G^c, per definizione di CLIQUE, per ogni $u,v \in S$, $(u,v) \in E^c$.
Per costruzione di E^c , $(u,v) \notin E$, generalizzando, S è un INDEPENDENT-SET in G.

HAMILTONIAN CYCLE PROBLEM:

Dato un grafo G=(V, E), diciamo che un ciclo C in G è un ciclo Hamiltoniano se visita ogni vertice esattamente una volta, senza ripetizioni. Il problema è: Dato un grafo *non orientato* G=(V, E), esiste un ciclo semplice Γ che contiene ogni nodo in V?

DIR-HAM-CYCLE≤PHAM-CYCLE:

Dobbiamo mostrare che ***G ha un ciclo Hamiltoniano se e solo se G' lo ha anche esso***.

(⇒) Ciascun nodo u di G, eccetto s e t, viene sostituito da una tripla di nodi u_{in}, u, u_{out} in G', collegati tra loro tramite archi, mentre i nodi s e t verranno sostituiti rispettivamente con s_{out} e t_{in}.
I nodi u e v in G, che sono collegati da un arco, in G´ saranno collegati da u_{out} e v_{in} come in figura sopra, ciò completa la costruzione in G.
Per dimostrare che ciò funziona, facciamo vedere che un cammino Hamiltoniano in G, ad esempio s-u-v-t, in G' sarà s_{out}-u_{in}-u-u_{out}-v_{in}-v-v_{out}-t_{in}.

(⇐) Affermiamo che qualsiasi cammino Hamiltoniano in G' da s_{out} a t_{in} deve andare da una tripla di nodi ad un'altra tripla, eccetto per l'inizio e la fine, perché qualsiasi cammino di questo tipo ha un cammino Hamiltoniano corrispondente in G.
Per confermare ciò, se si inizia il cammino da s_{out} e lo si segue, non è possibile andare al di fuori delle varie triple, fino a t_{in}.

3-SAT≤PDIR-HAM-CYCLE:

Mostriamo che DIR-HAM-CYCLE è in NP. Dato un grafo diretto G=(V, E) e un certificato che verifica polinomialmente che l'elenco dei vertici contenga ogni vertice esattamente una volta e che ogni coppia consecutiva nell'ordinamento sia unita da un arco, ciò stabilirebbe un ciclo Hamiltoniano.
Per ciascuna formula φ, facciamo vedere come costruire un grafo diretto G con due nodi, s e t, in cui ***esiste un cammino Hamiltoniano tra s e t se e solo se φ è soddisfacibile***.
Rappresentiamo ciascuna variabile x_i, con una struttura di forma romboidale che contiene una riga orizzontale di 3k+1 nodi con archi bidirezionali, in aggiunta altri 2 nodi sugli estremi del rombo, in più un nodo iniziale s collegato ad un nodo finale t, entrambi collegati a loro volta rispettivamente al primo rombo e all'ultimo. Rappresentiamo ciascuna clausola di φ con una coppia di nodo c_j all'interno delle varie strutture romboidali se quella variabile j-esima compare nella clausola. Questi nodi che rappresentano le clausole avranno frecce dipendenti al tipo di variabile booleana che compare in essa (negata o meno), completando la costruzione.

(⇐) Supponiamo che φ sia soddisfacibile, il cammino inizia da s, attraversa ciascun rombo in successione, e termina in t.
Per raggiungere i nodi orizzontali in un rombo, il cammino può procedere in due direzioni, ovvero da sinistra a destra e viceversa, l'assegnamento che soddisfa φ determina quale scegliere dei due versi. Se x_i =1 allora il cammino procede da sinistra a destra, altrimenti se x_i =0 va da destra a sinistra, formando così un ciclo Hamiltoniano.

(⇒) Se G ha un ciclo Hamiltoniano da s a t, facciamo vedere un assegnamento che soddisfa φ. Se il cammino Hamiltoniano passa attraverso i rombi in ordine da quello più in alto a quello più in basso e muovendosi da sinistra a destra o viceversa, possiamo determinare quali valori avranno le variabili booleane, ovvero se il cammino procede da sinistra a destra allora x_i =1, altrimenti se va da destra a sinistra x_i =0.

K-COLOR:

Dato un grafo non orientato G esiste un modo di colorare i nodi usando k colori in modo che nodi adiacenti NON hanno lo stesso colore?

3-SAT≤P3-COLOR:

Il problema si trova in NP. Dati G e k, un certificato che afferma che la risposta è "sì" è una colorazione k: si può verificare in tempo polinomiale che al massimo vengono usati i k colori e che nessuna coppia di nodi uniti da un arco riceve lo stesso colore.
Definiamo tre "nodi speciali" T, F e B, che chiamiamo Vero, Falso e Base ed uniamoli tra loro formando un triangolo. Creiamo un nodo per ogni letterale, unendoli coi loro negati, ed ognuno di esso lo colleghiamo al nodo BASE. Tramite la riduzione via "Gadget", aggiungiamo un gadget di 6 nodi e 13 archi ai nodi corrispondenti alle variabili che compaiono nelle clausole. Adesso assegniamo dei colori alle variabili e troviamo delle colorazioni corrette. Dimostriamo ora che ***l'istanza 3-SAT è soddisfacente se e solo se G ha una 3-COLOR***.

(⇒) Supponiamo che ci sia un'assegnazione soddisfacente per l'istanza 3-SAT. Definiamo una colorazione di G colorando prima **Base**, **Vero** e **Falso** arbitrariamente con i tre colori, quindi, per ogni i, assegnando a v_i il colore Vero se $x_i = 1$ e il colore Falso se $x_i = 0$.
Infine, è ora possibile estendere questa 3-COLOR in ciascun sottografo della clausola a sei nodi, risultando una 3-COLOR in tutto G.

(⇐) Supponiamo che G abbia un 3-COLOR. In questa colorazione, a ciascun nodo v_i viene assegnato il colore True o False; impostiamo la variabile x_i di conseguenza. Ora affermiamo che in ciascuna clausola dell'istanza 3-SAT, almeno uno dei termini nella clausola ha il valore di verità 1. In caso contrario, tutti e tre i nodi corrispondenti hanno il colore Falso nella 3-COLOR di G e non vi è alcuna 3-COLOR del sottografo della clausola corrispondente, ed abbiamo quindi una ***contraddizione***.

SUBSET-SUM:

Dati dei numeri naturali $w_1, \ldots, w_n$ e intero W, esiste un sottoinsieme la cui somma è esattamente W?

3-SAT≤PSUBSET-SUM:

SUBSET-SUM è in NP. Dati i numeri naturali $w_1, \ldots, w_n$ e un obiettivo W, un certificato verifica polinomialmente che un sottoinsieme di numeri interi dia come somma W. Dimostriamo che ***SUBSET-SUM ha una soluzione se e solo se φ è soddisfacibile*.**
Formiamo una tabella avente 2n+2k righe e n+k colonne, dove n è il numero di variabili booleane e k il numero di clausole. Ogni riga avrà assegnato 1 ad una variabile booleana e altri o nessuno 1 se quella variabile è in quella clausola. Le ultime 2k righe non sono indicizzate e hanno numeri fissati che fanno da supporto se per caso la somma delle righe indicizzate non arrivi al valora W.

(⇐) Supponiamo che la formula è soddisfacibile e prendiamo i numeri corrispondenti in tabella.
Adesso consideriamo i numeri posti a destra della tabella corrispondenti al nostro assegnamento di verità e sommiamoli provando ad ottenere l'ultimo numero ovvero 111,444, se non riusciamo ad ottenere quest'ultimo soltanto sommando i numeri presi dalle variabili dell'assegnamento di verità, possiamo prendere i numeri della seconda parte della tabella, per arrivare al numero obiettivo.

(⇒) Se ho dei sottoinsiemi di interi, e quindi una soluzione a SUBSET-SUM, considero le righe corrispondenti e vedo a che letterale corrisponde. Il numero 4 nell'ultima riga mi assicura che deve esserci almeno un 1 tra le righe selezionate, ma questo mi dice che c'è almeno un letterale posto a true e quindi la clausola corrispondente è soddisfatta.

SCHEDULE-RELEASE-TIMES:

Ogni job i ha un *release time* r_i quando è disponibile per la prima elaborazione, una *deadline* d_i entro la quale deve essere completato, e un *processing time* t_i. Supponiamo che tutti questi parametri siano numeri naturali.

Per essere completato, il lavoro i deve essere assegnato ad uno slot contiguo di unità di tempo t_i da qualche parte nell'intervallo $[r_i, d_i]$. La domanda è: possiamo pianificare tutti i lavori in modo tale che ciascuno venga completato entro la sua deadline?

Data un'istanza del problema, un certificato risolvibile sarà una specifica *release time* per ciascun lavoro. Potremmo quindi verificare che ogni lavoro venga eseguito per un intervallo di tempo distinto, tra la sua *release time* e la sua *deadline*. Quindi il problema è in NP.

Si consideri un'istanza di SUBSET-SUM con numeri $w_1, \ldots, w_n$ e un numero W, creiamo n jobs con *processing time* $t_i=w_i$, *release time* $r_i=0$ (tutti i jobs arrivano allo stesso tempo all'inizio quindi sono tutti immediatamente disponibili), e nessuna deadline, ma poniamo $d_j = \mathbf{1+}\sum_j^n w_j$.

Da quanto detto potremmo mettere tutti i jobs come vogliamo, non incontrando conflitti durante la schedulazione.

A questo punto, creiamo un job 0 con $t_0=1$, release time $r_0=W$ e deadline $d_0=W+1$, quindi il job 0 lo si deve schedulare quando arriva al tempo W ed eseguirlo immediatamente perché deve finire in tempo W+1. Ma adesso è come se il nostro intervallo fosse diviso in due, la parte sinistra da 0 a W, quindi ho W unità di tempo, e a destra c'è il resto, dovendo terminare entro $\mathbf{1+}\sum_j^n w_j$ (che indichiamo come S+1 per semplicità). Seguendo questa impostazione, abbiamo che le nostre unità di tempo adesso sono S, ovvero pari al tempo che abbiamo bisogno per andare a eseguire tutti i job, ma lo possiamo fare solo se riusciamo a distribuire i job metà prima del job 0 e l'altra metà dopo.

Mostriamo che ***SUBSET-SUM ha una soluzione se e solo se troviamo uno SCHEDULE-RELEASE-TIMES fattibile***.

(⇐) Quindi dobbiamo dividere gli interi $w_1, \ldots, w_n$ in qualche modo in un sottoinsieme, in modo tale che quest'ultimo sia eseguibile prima che arrivi il job 0 ed il resto sarà eseguito dopo che job 0 è finito, ma questa è proprio ciò che si deve fare per risolvere SUBSET-SUM su un insieme di interi.

(⇒) Quindi se l'istanza di SUBSET-SUM è un'istanza "si" allora posso dividere $w_1, \ldots, w_n$ in due parti, di cui una somma W, che corrisponde a tutti i jobs eseguiti prima del job 0, mentre i restanti sono eseguiti dopo il job 0, d'altra parte se abbiamo che i jobs possono essere eseguiti e quindi schedulati prima di job 0 ed il loro processing time totale è W, vuol dire che esiste un sottoinsieme dei w_i la cui somma è W.