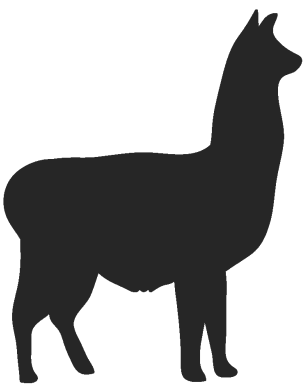
Algoritmi e Principi dell'Informatica

Lorenzo Rossi e tutti coloro che mi hanno aiutato!

AA 2021/2022

Ultimo aggiornamento: 2022-06-13

Questi appunti sono distribuiti con licenza Creative Commons 4.0 - CC BY-NC



nessun alpaca è stato ferito nella scrittura di questi appunti

Indice

Introduzione

Questi appunti si riferiscono al corso di *Algoritmi e principi dell'Informatica*, tenuto nell'anno accademico *2021/2022* dal professor *Marco Martinenghi*. La versione più aggiornata può essere trovata sulla repo di GitHub: github.com/lorossi/appunti-di-algoritmi-e-principi-dell-informatica.

Il contenuto di questo documento è tratto parzialmente dalle slide mostrate in aula e dai libri indicati nel manifesto degli studi, (*Informatica Teorica. Mandrioli, Spoletini* e *Introduzione agli algoritmi e strutture dati. Cormen, Leiserson, Rivest, Stein*). Spesso, le illustrazioni e le tabelle saranno uguali *(a meno di palesi errori da parte mia)*.

Alcuni argomenti, paragrafi e sezioni potrebbero talora seguire le lezioni, talora seguire i libri perché mi sono preso la libertà di riordinare il corso a piacimento, in modo che seguisse il mio schema mentale.

Il lettore si senta libero di avvisarmi qualora trovasse un errore! Vi ringrazio in anticipo.

Lorenzo Rossi

1 Linguaggi Formali

Con **linguaggi formali** si intende un'insieme di **stringhe** costruito su un **alfabeto**, secondo uno specifico insieme di regole.
Formalmente:

- **Alfabeto**, o *vocabolario*
  - Insieme **finito** di simboli di base
- **Stringa** su un alfabeto A
  - Sequenza **finita** di simboli dell'alfabeto A
    $\rightarrow$ c'è un ordine tra gli elementi
    $\rightarrow$ non c'è un limite superiore alla lunghezza
  - Sono consentite le **ripetizioni**

1.1 Stringhe

Una **stringa**, è caratterizzata dalla sua **lunghezza**:

- Equivale al numero di simboli che contiene
- La lunghezza della generica stringa x si indica con $|x|$

La stringa **vuota** è una stringa:

- Indicata come ϵ
- Per **definizione**, $|\epsilon| = 0$
- Un insieme formato dalla stringa vuota non corrisponde all'insieme vuoto
  $\rightarrow$ *in simboli:* $\{\epsilon\} \neq \emptyset$

La stringa vuota è definita su qualsiasi alfabeto.

1.1.1 Confronto di stringhe

Date due stringhe

$$x = x_1 x_2 \dots x_n$$
$$y = y_1 y_2 \dots y_m$$

esse si dicono **uguali** se e solo se sono valide le seguenti due proposizioni:

1. Le due stringhe hanno la stessa **lunghezza**
$$|x| = |y| \Leftrightarrow n = m$$
2. Gli elementi in posizioni corrispondenti sono **uguali**
$$x_i = y_i \; \forall 1 \leq i \leq n$$

1.1.2 Concatenazione di stringhe

Date due stringhe x e y, la loro **concatenazione** (detta anche *prodotto*) è una stringa xy (*oppure* $x \cdot y$) dove x è seguita da y.

Proprietà della concatenazione:

1. Una stringa x concatenata con ϵ è ancora x *(non viene alterata)*
   $\rightarrow$ $\{\epsilon\} \cdot x = x$
2. La concatenazione è **associativa** e **non commutativa**

$\rightarrow\ a \cdot (x \cdot y) = (a \cdot x) \cdot y$
$\rightarrow\ x \cdot y \neq x \cdot y$

3. Le ripetizioni di un carattere all'interno di una stringa vengono abbreviate tramite elevazione a potenza
   $\rightarrow\ xx \rightarrow x^2$
   $\rightarrow\ yyy \rightarrow y^3$
   $\rightarrow\ yyyyxx \rightarrow y^4x^2$

1.1.3 Sottostringhe

Una stringa x è una **sottostringa** (detta anche *fattore*) di una stringa s se esistono due stringhe y, z

$$y = y_1 y_2 \dots y_m$$
$$z = z_1 z_2 \dots z_m$$

tali che:

$$s = yxz$$

Proprietà delle sottostringhe:

- sia y che z possono essere ϵ
  - se $y = \epsilon \Rightarrow x$ è detta **prefisso** e s inizia con i caratteri di x
  - se $z = \epsilon \Rightarrow x$ è detta **suffisso** e s finisce con i caratteri di x
- Se $y = \epsilon, z = \epsilon$ allora x è *uguale* a s

1.2 Stella di Kleene

La **stella di Kleene** è un operatore unario che si applica a un insieme di simboli o a un insieme di stringhe. Si indica con il simbolo $*$ e si pronuncia *"A star"*.
Se A è un alfabeto, allora A^* è l'insieme di tutte le stringhe su simboli di A, inclusa la stringa vuota ϵ *a patto che essa faccia parte dell'alfabeto.* Non è imposto un limite superiore alla lunghezza delle stringhe prodotte.

1.2.1 Definizione formale della Stella di Kleene

È possibile definire la stella di Kleene tramite una trattazione più algebrica:

- Un **semigruppo** è una coppia $\langle S, \circ \rangle$ in cui:
  - S è un **insieme** ed è **chiuso** rispetto a $\circ$
  - $\circ$ è un'operazione **associativa** su S
    $\rightarrow$ le operazioni possono essere associate a piacere
    $\rightarrow$ è *distributiva* e *commutativa*
- Un **monoide** è un semigruppo tale per cui:
  - $\exists\, u \forall\, x \mid x \circ u = u \circ x = x$
  - u è *l'elemento neutro* rispetto all'operazione $\circ$
- Un **gruppo** è un monoide che tale per cui:
  - $\forall\, x \exists\, x^{-1} \mid x \circ x^{-1} = x^{-1} \circ x = u$
  - x^{-1} è *l'elemento inverso* di x rispetto all'operazione $\circ$

Tra i 3 insiemi è valida la relazione:

$$semigruppo \subseteq monoide \subseteq gruppo$$

A valle di queste definizioni, è possibile inoltre affermare che:

- Dato un semigruppo $\langle S, \circ \rangle$ e un sottoinsieme X di S:
  - X^+ *(detto "più di Kleene")* denota il sottoinsieme di S generato da X, cioè tutte le sequenza della forma
    $$x_1 \circ \ldots \circ x_n \quad x_i \in X, n \geq 1$$
  - Per un monoide $\langle S, \circ \rangle$ con unità u:
    - $X^* = X^+ \cup \{u\}$
    - X^* è detto il **monoide libero** generato da X

1.3 Linguaggi

È detto **linguaggio** un qualsiasi insieme di stringhe definite su un alfabeto. Sono linguaggi:

- Italiano, Inglese, Francese, ...
- `C`, `Java`, `Pascal`, ...
- Linguaggi grafici, Musica, Multimedia, ...

Inoltre:

- Una lingua come l'Italiano è *infinita*, perché è possibile scrivere frasi di lunghezza infinita
- Analogamente, un linguaggio come il `C` è un insieme *potenzialmente infinito* poiché l'insieme di programmi corretti è infinito

Formalmente, un linguaggio L definito su un alfabeto A è un *sottoinsieme* di A^*.
I linguaggi formali, contrariamente a quanto possa sembrare, *non sono solo rappresentazioni matematiche astratte*. Essi infatti sono metodi utili a *rappresentare* o *comunicare* una informazione, quindi non solo stringhe senza significato. Usando le operazioni descritte nella Sezione 1.3.1, in base ai vari contesti, è possibile interpretare un linguaggio in modi consoni. Anche i *calcoli* possono essere rappresentati tramite linguaggi formali.
Esistono molti tipi di linguaggi, tra i quali si riconoscono soprattutto:

- Linguaggi *naturali*
- Linguaggi *di programmazione*
- Linguaggi *logici*

Il significato di queste definizioni verrà affrontato in seguito.

1.3.1 Operazioni sui linguaggi

Poiché un linguaggio non è altro che un insieme di stringhe, su di loro si applicano le operazioni insiemistiche. Esse sono:

1. **Unione** - $\cup$
2. **Intersezione** - $\cap$
3. **Differenza** - $\setminus$ *oppure* $-$
4. **Complemento** - L^c
5. **Concatenazione** - $\cdot$
6. **Potenza n-esima** - L^n
7. **Chiusura di Kleene** - L^*

Le operazioni sui linguaggi creano nuove classi di linguaggi. Esempi e le loro rispettive proprietà sono descritte nelle Sezioni seguenti (1.3.1.1 - 1.3.1.6).

1.3.1.1 Unione

Siano L_1, L_2 due linguaggi:

$$L_1 \cup L_2 = \{w \,|\, w \in L_1 \vee w \in L_2\}$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro unione sarà:

$$L_1 \cup L_2 = \{\epsilon, a, b, c, ba, bb, bc, ca, cb, cc\}$$

1.3.1.2 Intersezione

Siano L_1, L_2 due linguaggi:

$$L_1 \cup L_2 = \{w \,|\, w \in L_1 \wedge w \in L_2\}$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro intersezione sarà:

$$L_1 \cap L_2 = \{bc, ca\}$$

1.3.1.3 Differenza

Siano L_1, L_2 due linguaggi:

$$L_1 \setminus L_2 = L_1 - L_2 = \{w \,|\, w \in L_1, w \notin L_2\}$$

È un'operazione che viene generalmente usata quando $L_2 \subseteq L_1$.

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{ba, bb, bc, ca, cb, cc\} \qquad L_2 = \{bc, ca\}$$

La loro differenza sarà:

$$L_1 \setminus L_2 = \{ba, bb, cb, cc\}$$

1.3.1.4 Complemento

Sia L_1 un linguaggio:

$$\neg L_1 = \{w \,|\, w \notin L_1\}$$

Sia L un linguaggio definito su un alfabeto A. Allora la sua operazione di complementazione sarà definita come:

$$L^c = A^* \setminus L$$

1.3.1.5 Concatenazione

Siano L_1, L_2 due linguaggi:

$$L_1 \cdot L_2 = \{wz \,|\, w \in L_1, z \in L_2\}$$

- L'operazione non è commutativa: $L_1 \cdot L_2 \neq L_2 \cdot L_1$
- Si fa scorrere ogni elemento di L_1 associandolo a ogni elemento di L_2
- La concatenazione di un linguaggio con un linguaggio vuoto dà origine al linguaggio stesso, mentre la concatenazione di un linguaggio con un insieme vuoto dà un insieme vuoto:

$$L \cdot \{\epsilon\} = L \quad L \cdot \emptyset = \emptyset$$

- Il numero delle stringhe in $L_1 \cdot L_2$ sarà pari al prodotto del numero di stringhe in ciascun linguaggio:

$$|L_1 \cdot L_2| = |L_1| \cdot |L_2|$$

Esempio: siano L_1, L_2 due linguaggi:

$$L_1 = \{\epsilon, a, b, c, bc, ca\} \qquad L_2 = \{ba, bb, bc, ca, cb, cc\}$$

La loro concatenazione sarà:

$$\begin{aligned} L_1 \cdot L_2 = \{\ & ba, bb, bc, ca, cb, cc, aba, abb, abc, aca, acb, acc, bba, bbb, bbc, bca, \\ & bcb, bcc, cba, cbb, cbc, cca, ccb, ccc, bcba, bcbb, bcbc, bcca, bccb, bccc, \\ & caba, cabb, cabc, caca, cacb, cacc\ \} \end{aligned}$$

1.3.1.6 Potenze

Sia L un linguaggio. La sua potenza L^n sarà ottenuta *concatenando* L con se stesso per n volte.

$$L^n = \underbrace{L \cdot L \cdot \ldots \cdot L}_{n \text{ volte}}$$

Definizione induttiva:

1. $L^0 = \{\epsilon\}$
2. $L^i = L^{i-1} \cdot L$

Si noti che il punto 1 prende il nome di **caso base**, mentre il punto 2 si chiama **passo induttivo**.
Il *passo induttivo* implica che almeno una parte del problema sia stato risolto (tramite **ipotesi induttiva**). In questo caso, l'*ipotesi induttiva* è data dalla relazione che riguarda L^{i-1}.

Esempi:

- $L^2 = L \cdot L$
- $L^3 = L \cdot L \cdot L$
- $L^4 = L \cdot L \cdot L \cdot L$

1.3.1.7 Chiusura di Kleene

Sia L un linguaggio. Allora l'operatore *stella di Kleene* su L è definito come:

$$L^* = \bigcup_{n=0}^{\infty} L^n$$

Analogamente, è possibile definire l'operatore *più di Kleene*:

$$L^+ = \bigcup_{n=1}^{\infty} L^n$$

Proprietà:

1. $L^* = L^+ \cup L^0 = L^+ \cup \{\epsilon\}$
2. $L^+ = L \cdot L^*$
3. $L^+ = L^* \Leftrightarrow \epsilon \in L$

Poiché la concatenazione **non è commutativa**, la chiusura di Kleene **non è riflessiva**.

2 Automi a stati finiti - *FSA*

Un **automa a stati finiti** (o *FSA*, dall'Inglese *Finite State Automaton*) è il più semplice modello di astrazione. Esso rappresenta un sistema che ammette un *insieme finito* di stati (e di conseguenza un numero limitato di configurazioni). In seguito ad un determinato ingresso (a sua volta formato da un insieme finito di valori), potrà avvenire una transizione tra due stati distinti.
Gli *FSA* rappresentano il più semplice modello di computazione: molti dispositivi possono essere modellati come tali, seppure con alcune limitazioni.
Per poter usare gli *FSA* per riconoscere linguaggi, è importante identificare:

1. Le condizioni *iniziali* del sistema
2. Gli stati *finali* del sistema

Affinché sia possibile costruire un modello adatto, è necessario riconoscere gli elementi al suo interno:

- Gli **stati**, tra i quali si identificano:
  - Stato **iniziale**
  - Stati **finali**
- Le **transizioni**
- L'**ingresso**

2.1 Stati, Transizioni e Ingressi

Gli stati sono rappresentati come dei cerchi con all'interno la loro etichetta di riferimento. La loro rappresentazione è illustrata in Figura 1.

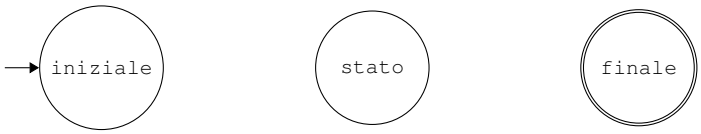


Figura 1: Stati iniziali e finali di un *FSA*

Un *FSA* è definito su un alfabeto. I simboli dell'alfabeto rappresentano l'ingresso del sistema.

Quando il sistema riceve un ingresso, cambia il proprio stato interno. Il passaggio tra stati diversi avviene tramite **transizioni**. Una **transizione** può avere un'etichetta che denomina l'azione che viene intrapresa nel momento della sua attivazione.
Una transizione è rappresentata mediante frecce, come in Figura 2.

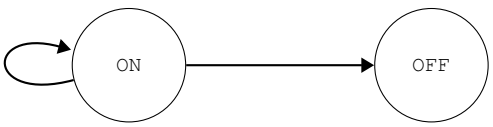


Figura 2: Transizioni in un *FSA*

2.2 Definizione formale *FSA*

Formalmente, un *FSA* è una tupla di 5 elementi $\langle Q, A, \delta, q_0, F \rangle$ dove:

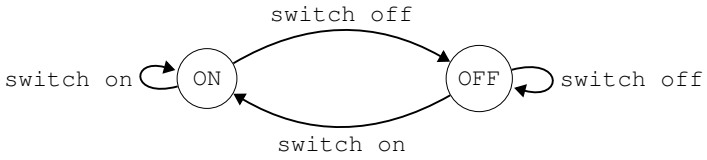
- Q è un insieme di **stati**, finito

- A è l'**alfabeto** di **ingresso**
- δ è la **funzione** di **transizione**:
  - $\delta : Q \times A \to Q$
  - la funzione di transizione è detta **parziale** se non tutte le transizione da tutti i possibili stati per tutti i possibili elementi dell'alfabeto sono definite
  - un *FSA* con una funzione di transizione totale è detto **completo**
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati finali**

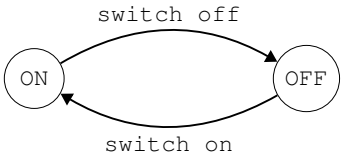
Si noti che nonostante sia definire **un solo** stato iniziale, un *FSA* ammette **più** stati finali. Inoltre, uno stato può essere contemporaneamente iniziale e finale.

2.2.1 *FSA* con transizione totale

Come già enunciato, una funzione di transizione completa implica che essa sia definita per tutti i possibili stati per ogni possibile elemento dell'alfabeto di ingresso. Il *FSA* in Figura 3a ha una funzione di transizione *totale*, mentre quello in Figura 3b presenta una funzione di transizione *parziale*.



(a) *FSA* con funzione di transizione totale



(b) *FSA* con funzione di transizione parziale

Il *FSA* in Figura 3b, al contrario di quello in Figura 3a, non presenta le transizioni in corrispondenza dell'azione di `switch off` sullo stato `off` e `switch on` sullo stato `on`.

2.2.2 Sequenza di mosse

Una **sequenza di mosse** inizia da uno stato iniziale ed è di **accettazione** se raggiunge uno degli stati *finali*.

Formalmente:

- **Sequenza di mosse**:
  - $\delta^* : Q \times A^* \to Q$
  - Il **dominio** è definito come il prodotto cartesiano tra stati e stella di Kleene di sequenze di simboli
  - Il **codominio** coincide con l'insieme degli stati
- δ^* è definita induttivamente a partire da δ:
  1. $\delta^*(q, \epsilon) = q$
  2. $\delta^*(q, yi) = \delta\left(\delta^*(q, y), i\right)$
- Stato **iniziale** $q_0 \in Q$
- Stati **finali** $F \subseteq Q$

2.2.3 Condizione di accettazione di un *FSA*

Il linguaggio relativo al *FSA* è costituito dalle stringhe x che appartengono a δ^*. *Formalmente*:

$$\forall\, x \in L \Leftrightarrow \delta^*(q_0, x) \in F$$

I linguaggi riconosciuti dagli *FSA*, come verrà analizzato più in dettaglio in seguito *(per esempio nella Sezione 6.2)*, prendono il nome di **regolari**.

2.3 Trasduttori a stati finiti - *FST*

Idea: usare gli automi come traduttori di linguaggi. Nasce il **FST** *(dall'Inglese finite state transducer)*, cioè un *FSA* che lavora su due nastri. È una specie di macchina traduttrice.
La rappresentazione più semplificata di un *FST* è rappresentata in Figura 4.

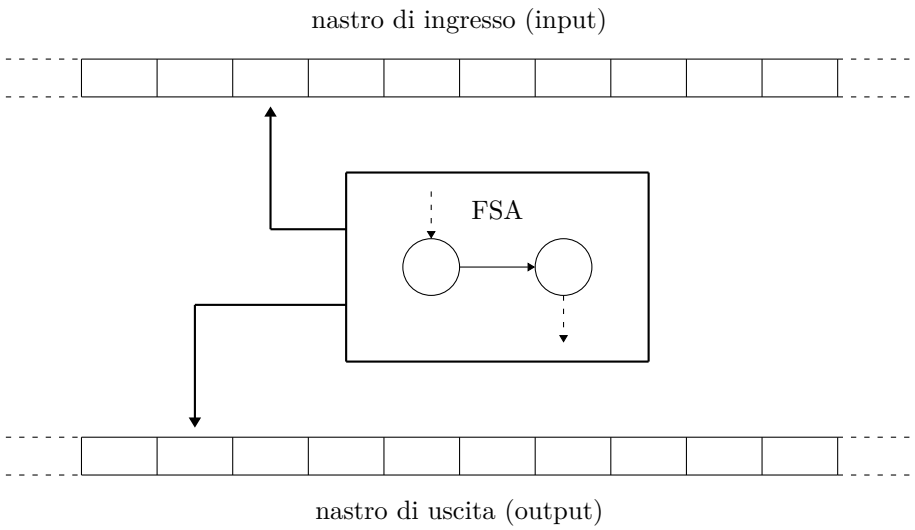


Figura 4: Diagramma semplificato di un FST

Un *FST* è composto da:

1. Una stringa di **ingresso** x
2. Una stringa di **uscita** y
3. Una funzione $\tau : L_1 \to L_2$ tale che $y = \tau(x)$

2.3.1 Definizione formale *FST*

Un trasduttore a stati finiti (*FST*) è una tupla di 7 elementi $\langle Q, A, \delta, q_0, F, O, \eta \rangle$:

- Q è un insieme finito di **stati**
- A è l'**alfabeto** di **ingresso**
- δ è la **funzione** di **transizione**:
  - $\delta : Q \times A \to Q$
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati finali**
- O è l'**alfabeto** di **uscita**
- η è la **funzione** di **uscita**:
  - $\eta : Q \times I \to O^*$
  - il **dominio** è il prodotto cartesiano degli stati per l'alfabeto di ingresso
  - il **codominio** è la stella di Kleene dell'alfabeto di uscita

La definizione di Q, A, δ, q_0, F è analoga a quella avvenuta nella Sezione 2.2 riguardante gli *FSA*.
La condizione di accettazione resta la stessa degli accettori. Di conseguenza, **la traduzione avviene solo su stringhe accettate**.

2.3.2 Traduzione di una stringa

Analogamente a quanto già illustrato nella definizione della sequenza di mosse δ (Sezione 2.2.2), η^* verrà definito induttivamente. Infatti, ricordando che $\eta^* : Q \times I^* \to O^*$:

1. $\eta^*(q, \epsilon) = \epsilon$
2. $\eta^*(q, y \cdot i) = \eta^*(q, y) \cdot \eta\left(\delta^*(q, y), i\right)$

sarà quindi valida la relazione:

$$\forall\, x, \tau(x) = \eta^*(q_0, x) \Leftrightarrow \delta^*(q_0, x) \in F$$

2.3.3 Pumping lemma

Si consideri il *FSA* in Figura 5.

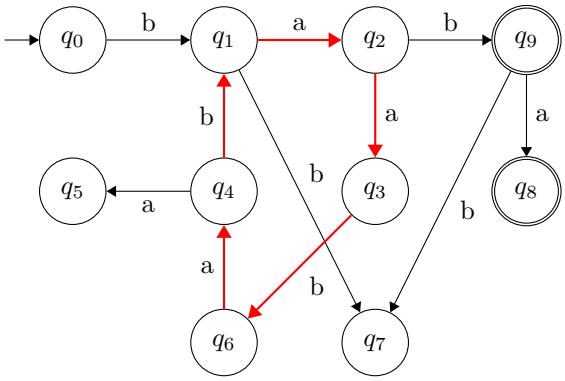


Figura 5: Pumping lemma

Se si ammette la possibilità che il ciclo $q_1 \to q_2 \to q_3 \to q_6 \to q_4 \to q_1 \to \dots$ venga attraversato una volta, allora è possibile che esso venga attraversato un numero indefinito *(potenzialmente infinito)* di volte.
Più formalmente:

- Se $x \in L, |x| \geq |Q|$, allora esistono uno stato $q \in Q$ a una stringa $w \in I^+$ tali che:
  - $x = ywz$
  - $\delta^*(q, w) = q$
- Perciò varrà anche $\forall\, n \geq 0\ yw^n z \in L$

Questo fenomeno prende il nome di **pumping lemma**, e porta a due possibili *conseguenze*:

1. $L = \emptyset$
   - $\exists\, x \in L \Leftrightarrow \exists\, y \in L,\ |y| < |Q|$
   - È sufficiente rimuovere tutti i *cicli* dall'*FSA* che accetta x
2. $L = \infty$
   - Il *FSA* deve verificare in modo analogo se $\exists\, x \in L,\ |Q| \leq |x| < 2 \cdot |Q|$
   - *La lunghezza della stringa x è tale da compiere meno di due cicli*

Nella pratica, ciò porta alle seguenti *implicazioni*:

- Un linguaggio di programmazione che ammette 0 programmi corretti **non è rilevante**
- Analogamente, un linguaggio di programmazione che ammette *un numero finito* di programmi corretti **non è rilevante**

$\Rightarrow$ Il *pumping lemma*, **quindi, non unicamente è un aspetto negativo** poiché permette di creare linguaggi **infiniti**

2.3.3.1 Conseguenza negativa del *pumping lemma*

Si consideri il linguaggio

$$L = \{a^n b^n \mid n > 0\}$$

e si supponga che sia riconosciuto da un qualsiasi *FSA*.

Si consideri ora la stringa

$$x = a^m b^m,\ m > |Q|$$

e si applichi il *pumping lemma*. Come conseguenza dello stesso, poiché la lunghezza della stringa è superiore al numero di stati, dovrebbe esistere una costante k per la quale, se $x \in L, |x| \geq k$, la stessa x può essere scritta come ywz con $1 \leq |w| \leq k, yw^i z \in L \,\forall\, i \geq 0$.
La dimostrazione avviene *per assurdo*. Scomponendo x, si possono identificare 3 possibili forme di essa che verranno accettate:

- $x = ywz, w = a^r, r > 0$, quindi anche $a^{s+k\cdot r} b^n \,\forall\, k \geq 0$ **dovrebbe essere accettato**
- $x = ywz, w = b^r, r > 0$, quindi anche $a^n b^{s+k\cdot r} \,\forall\, k \geq 0$ **dovrebbe essere accettato**
- $x = ywz, w = a^r b^s, r > 0, s > 0$, quindi anche $a^{n-r}(a^r b^s)^k b^{n-s} \,\forall\, k \geq 0$ **dovrebbe essere accettato**

Tutte e tre queste forma, tuttavia, violano la forma di x indicata nell'ipotesi. Quindi si può affermare che esistono linguaggi non riconoscibili tramite *FSA*.
Più precisamente, esistono dei *linguaggi infiniti* che non possono essere riconosciuti dagli *FSA*. Questa affermazione può essere giustificata informalmente affermando che *"gli FSA non possono contare"*.
La famiglia dei linguaggi che sono riconosciuti dagli *FSA* prende il nome di **regolare**.

2.3.3.2 Provare che un linguaggio sia infinito

Per poter provare che:

- Il linguaggio riconosciuto dal *FSA* sia infinito, è necessario provare le infinite stringhe che appartengono al linguaggio su cui esso è definito
- Il linguaggio riconosciuto dal *FSA* sia vuoto, è necessario provare le infinite stringhe che appartengono al linguaggio su cui esso è definito

Non è possibile effettuare un test con un numero infinito di stringhe in un tempo finito. Quindi, per effettuare una ricerca esaustiva e trovare una risposta a questa domanda, si provano le stringhe strettamente inferiori al numero di stati:

⇒ Se una stringa **viene accettata**, allora **tutte** le stringhe verranno accettate
⇒ Se una stringa **non viene accettata**, allora **nessuna** stringa verrà accettata

Il numero di queste stringhe, e quindi il numero di test da effettuare, è limitato.

2.3.3.3 Provare la presenza di un ciclo

Se un automa accetta un certo linguaggio, allora una stringa di lunghezza superiore al numero di stati verrà accettata se e solo se è presente un ciclo all'interno del corrispondente *FSA*.

2.4 Operazioni sugli *FSA*

Prima di poter definire le operazioni sugli *FSA*, è necessario definire il concetto di **chiusura**:
Chiusura in *matematica*: un insieme S è **chiuso** rispetto a una operazione OP se, quando OP è applicata agli elementi di S, il risultato è ancora un elemento di S
Leggermente diverso è il caso dei *linguaggi*, che verrà esaminato più nel seguente paragrafo.

2.4.1 Chiusura per i linguaggi

Siano:

- $\mathcal{L} = \{L_i\}$ una *famiglia* di **linguaggi**
  - $\mathcal{L}$ è chiuso rispetto all'operazione OP se e solo se, $\forall\, L_1, L_2 \in \mathcal{L},\ L_1\ OP\ L_2 \in \mathcal{L}$
- $\mathcal{R}$ la *famiglia* dei **linguaggi regolari**
  - $\mathcal{R}$ è chiuso rispetto alle operazioni insiemistiche, alla concatenazione e all'operatore $*$ *(stella di Kleene)*

2.4.1.1 Intersezione

Siano A_1 e A_2 due generici *FSA*, caratterizzati dalle loro espressioni

$$A_1 = \langle Q_1, I_1, \delta_1, q_{01}, F_1 \rangle$$
$$A_2 = \langle Q_2, I_2, \delta_2, q_{02}, F_2 \rangle$$

Si supponga che $I_2 = I_1 = I$ e che $Q_1 \cap Q_2 = \emptyset$. Entrambe le supposizioni non comportano alcuna perdita di generalità perché:

1. Se I_1 ed I_2, fossero diversi, posto che la funzione di transizione non deve essere necessariamente uguale, i due atomi possono essere considerati definiti su $I_1 \cup I_2$
2. Se Q_1 e Q_2 non fossero disgiunti, i nomi degli stati comuni possono essere banalmente cambiati

FSA che riconosce entrambi i linguaggi (e quindi $I_1 \cap I_2$) è costruito come:

$$C = \langle A_1, A_2 \rangle = \langle Q_1 \times Q_2, I, \delta, \langle q_{01}, q_{02} \rangle, F_1 \times F_2 \rangle$$
$$\delta(\langle q_1, q_2 \rangle, i) = \langle \delta_1(q_1, i), \delta_2(q_2, i) \rangle$$

La dimostrazione che

$$L\left(\langle A_1, A_2 \rangle\right) = L\left(A_1\right) \cap L\left(A_2\right)$$

può avvenire un ragionamento per induzione alla funzione δ.

Intuitivamente, l'*esecuzione parallela* di due *FSA* può essere simulata *accoppiandoli* tramite prodotto cartesiano. L'intersezione di due *FSA* può essere vuota, non riconoscendo alcun linguaggio (ricordando che $\emptyset \neq \{\epsilon\}$).

Esempio: siano A e B due *FSA* rappresentati come segue:

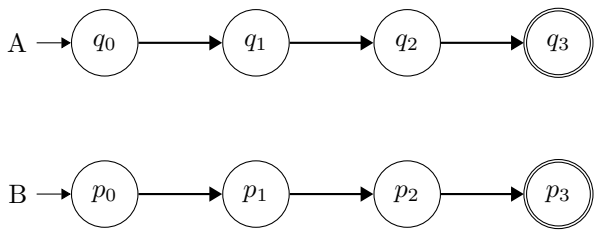


Figura 6: Intersezione tra FSA

La loro *intersezione* $C = A \cap B$ può essere rappresentata come segue:

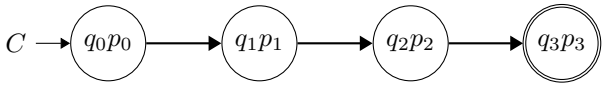


Figura 7: Intersezione tra FSA

2.4.1.2 Unione

Analogamente a quanto avviene nell'intersezione (Sezione 2.4.1.1), con le stesse ipotesi, l'unione di due *FSA*:

$$A_1 = \langle Q_1, I_1, \delta_1, q_{01}, F_1 \rangle$$
$$A_2 = \langle Q_2, I_2, \delta_2, q_{02}, F_2 \rangle$$

La loro unione sarà data da:

$$C = \langle A_1, A_2 \rangle = \langle Q_1 \times Q_2, I, \delta, \langle q_{01}, q_{02} \rangle, F_1 \times Q_2 \cup Q_1 \times F_2 \rangle$$
$$\delta(\langle q_1, q_2 \rangle, i) = \langle \delta_1(q_1, i), \delta_2(q_2, i) \rangle$$

2.4.1.3 Complemento

Si consideri il *FSA*:

$$A = \langle Q, I, \delta, q_0, F \rangle$$

Per poterlo complementare, è prima necessario costruire un *FSA* A' aggiungendo un nuovo stato $\overline{q}$ ad A in modo che la funzione di transizione di A' conduca a $\overline{q}$ ogni qualvolta che è indefinita in A. Si imponga inoltre che l'automa, una volta in $\overline{q}$, ci rimanga per qualsiasi simbolo di ingresso.
In questo modo, la funzione di transizione di A (e quindi la sua corrispondente δ^*) è totale.
Proprietà del complemento:

- Se l'intera stringa di ingresso viene scandita, allora per complementare il risultato basta *scambiare il* `si` *con il* `no`
- Se la fine della stringa non viene raggiunta, allora lo scambio di F con $Q - F$ non funziona
- Nel caso degli *FSA* il *"trucco"* consiste nel completare la funzione di transizione δ
- In generale, *non è possibile considerare equivalenti la risposta negativa a una domanda e la risposta positiva della domanda opposta*

Grazie a questa operazione, è possibile definire l'unione di due *FSA* (Sezione 2.4.1.2) tramite le leggi di *De Morgan*:

$$L(A_1) \cup L(A_2) = \overline{\overline{L(A_1)} \cap \overline{L(A_2)}} = \left(L(A_1)^C \cup L(A_2)^C\right)^C$$

con il medesimo risultato.

3 Pushdown automaton - *PDA*

Come mostrato prima nell'applicazione del *pumping lemma* al linguaggio $L = \{a^n b^n \mid n > 0\}$ (Sezione 2.3.3.1), dimostra l'impossibilità di riconoscere alcuni linguaggi tramite *FSA*, a causa della loro inabilità nel contare una quantità di simboli sconosciuta a priori.
A causa di questa mancanza, essi sono inadatti al riconoscimento di molti linguaggi di interesse pratico. Per ovviare a questo limite, vengono introdotti i *PDA* (in Italiano *automi a pila* or *AP*), il cui diagramma semplificato è mostrato in Figura 8.

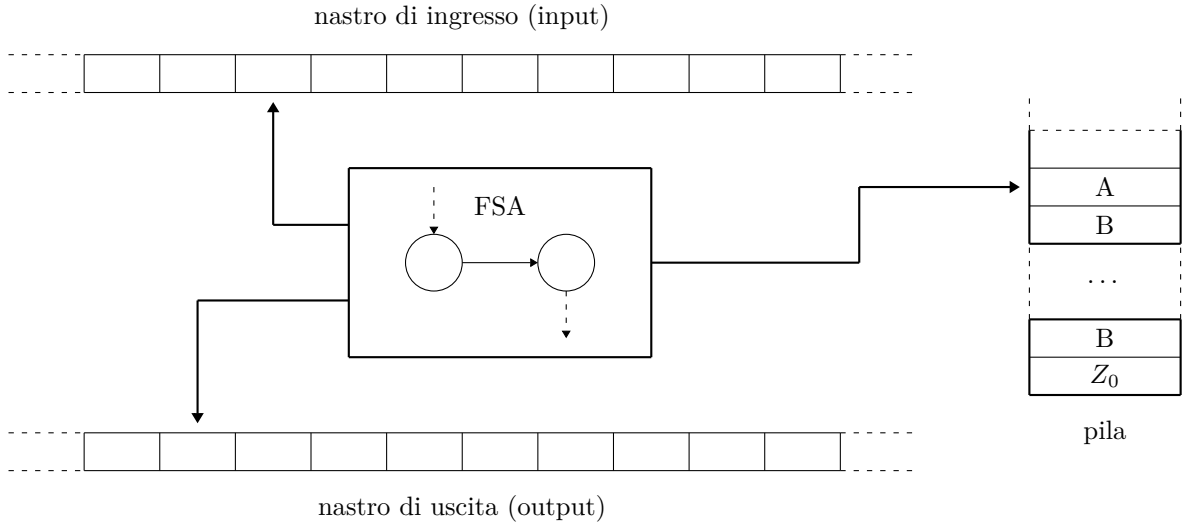


Figura 8: Diagramma semplificato di un PDA

La **pila** è una forma di memoria in cui:

- I nuovi simboli sono **inseriti in cima**
- La pila viene **letta dalla cima**
- Un simbolo letto viene **estratto dalla cima**

  $\rightarrow$ si attua una politica *LIFO*
- L'ultimo elemento in basso della pila è occupato da un particolare simbolo Z_0

I *PDA* differiscono dagli automi a stati finiti in due modi:

1. Possono usare la cima della pila per decidere quale transizione effettuare
2. Possono manipolare la pila durante una transizione

3.1 Definizione formale *PDA*

Formalmente, un *PDA* é una tupla di 7 elementi $\langle Q, A, \Gamma, \delta, q_0, Z_0, F \rangle$, dove:

- Q è un **insieme di stati**, finito
- A è l'**alfabeto** di ingresso
- Γ è l'**alfabeto** di pila
- δ è la funzione di **transizione**:
  - $\delta : Q \times (I \cup \{\epsilon\}) \times \Gamma \rightarrow Q \times \Gamma^*$
  - Il **dominio** è definito come il prodotto cartesiano tra stati, alfabeto di ingresso unito all'elemento nullo e alfabeto di pila
  - Il **codominio** è definito come il prodotto cartesiano tra stati e *stella di Kleene* dell'alfabeto di pila
- $Z_0 \in \Gamma$ è il **simbolo iniziale** (il primo in *basso*) di pila

- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'insieme di **stati finali**, finito

Si noti che la definizione di Q, A, δ, q_0, F sono analoghi a quanto illustrato in Sezione 2.2 riguardo gli *FSA*.

3.2 Mosse di un *PDA*

In base a:

- Il simbolo **letto** dall'ingresso *(opzionalmente)*
- Il simbolo **letto** dalla cima della pila
- Lo **stato** del dispositivo di controllo

il *PDA*:

- Cambia il proprio **stato**
- Sposta in avanti la **stato**
- Sostituisce il simbolo letto dalla **pila** con una stringa α *eventualmente vuota*

Sono anche ammesse delle delle mosse *spontanee* (dette ϵ-mosse) che avvengono anche senza che venga letto il simbolo dalla stringa, ignorandolo. Per queste mosse la funzione di transizione è definita come:

$$\delta(q, \epsilon, A) = \langle p, \alpha \rangle$$

Se una ϵ-mossa è stata definita per una coppia $\langle \overline{Q}, \overline{A} \rangle$ *(stato e simbolo in cima alla pila)*, non è possibile definire un'altra mossa non-ϵ per lo stesso simbolo $\overline{A}$ dallo stesso stato $\overline{Q}$.
Gli *FSA* non hanno ϵ-mosse, in quanto esse sono una caratteristica dei *PDA*. Essi non implicano che il carattere in cima alla stringa sia ϵ ma che la stringa non viene letta.

3.3 Accettazione di una stringa

Alla fine della computazione, la stringa di ingresso x è *riconosciuta* (o *accettata*) se valgono entrambe le condizioni:

1. Il *PDA* la legge **completamente**
2. Il *PDA* si trova in uno stato di accettazione (**finale**) quando la fine di x è stata raggiunta

Non ci sono condizioni sulla pila sia vuota affinché la stringa x sia accettata. In particolare, **non è necessario che la pila sia vuota** alla fine della computazione *(anche se è una condizione che si può verificare)*.

3.4 Configurazione di un *PDA*

Una **configurazione** è una generalizzazione della nozione di stato. Essa mostra:

- Lo **stato corrente** del dispositivo di controllo
- La **porzione di stringa** di ingresso a destra dalla testina
  - indica la parte di essa ancora non letta
  - ciò chè è già stato letto non è rilevante poiché è già stato consumato
- Il **contenuto** della pila

Può essere vista come una *istantanea* del *PDA*, mostrando nel tempo il suo stato interno.

Formalmente, la configurazione c è una tripla $\langle q, x, \gamma \rangle$:

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x \in I^*$ è la **porzione non letta** della stringa di ingresso
- $\gamma \in \Gamma^*$ è la **stringa di simboli** nella pila.

3.5 Transizioni di un *PDA*

Le **transizioni** tra configurazioni dipendono dalla funzione di transizione e si indicano con il simbolo $\vdash$. Una sequenza di transizioni è indicata col simbolo $\vdash^*$.
Esse illustrano come commutare tra una *istantanea* e la sua successiva. Per un dato *PDA* A, la transizione $\vdash_A$ nello spazio di tutte le possibili configurazioni di A è definita da

$$c = \langle q, x, \gamma \rangle \vdash_A c' = \langle q', x', \gamma' \rangle$$

solo se vale una delle due condizioni:

1. La funzione di transizione è definita per un simbolo di ingresso

$$\begin{aligned} x = ay &\mapsto \ x' = y \\ \gamma = A\beta &\mapsto \gamma' = \alpha\beta \\ \delta(q, a, A) &= \langle q', \alpha \rangle \end{aligned}$$

2. La funzione di transizione è definita per una ϵ-mossa

$$\begin{aligned} x &\mapsto x', \\ \gamma = A\beta &\mapsto \gamma' = \alpha\beta \\ \delta(q, \epsilon, A) &= \langle q', \alpha \rangle \end{aligned}$$

3.5.1 Nota sulle transizioni

Si consideri un generico *PDA* con le caratteristiche già indicate, sia δ la sua funzione di transizione.
Si indica con il simbolo $\perp$ il risultato di una transizione che **non è definita** per i parametri indicati.
Allora:

- Una ϵ-mossa è una mossa **spontanea**:
  - se $\delta(q, \epsilon, A) \neq \perp$ *(δ non è indefinita)* e A è il simbolo in cima alla pila, la transizione **può sempre essere eseguita**
- Se $\delta(q, \epsilon, A) \neq \perp$, allora $\forall\, i \in I, \delta(q, i, A) = \perp$
  - se questa proprietà non fosse soddisfatta, entrambe le transizioni **sarebbero consentite**
  - questa condizione **garantisce il determinismo** poiché non sarebbe possibile definire unicamente quale delle più transizioni scegliere partendo dallo stato
  - più avanti verrà mostrato il funzionamento degli automi in assenza di determinismo (Sezione 5)

3.5.2 Notazione grafica delle transizioni di un *PDA*

Il diagramma mostrato in Figura 9 mostra una transizione di un *PDA* da uno stato q_0 a uno stato p_0.

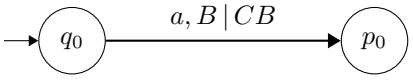


Figura 9: Notazione grafica dei *PDA*, *push*

La notazione implica che se:

1. Il *PDA* si trova nello **stato** q_0
2. Il **carattere** a viene letto dalla stringa di input
3. Il **carattere** B è in cima alla pila

allora succederanno i seguenti eventi:

1. Il carattere C viene **aggiunto** in cima alla pila *(operazione di push)*

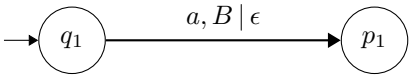


Figura 10: Notazione grafica dei *PDA*, *pop*

2. Lo stato attuale diventa p_0

Il diagramma mostrato in Figura 10 mostra una transizione di un *PDA* da uno stato q_1 a uno stato p_1.
La notazione implica che se:

1. Il *PDA* si trova nello **stato** q_1
2. Il **carattere** a viene letto dalla stringa di input
3. Il **carattere** B è in cima alla pila

allora succederanno i seguenti eventi:

1. Il carattere B viene **rimosso** dalla pila *(operazione di pop)*
2. Lo stato attuale diventa p_1

3.6 Condizione di accettazione di un *PDA*

Sia $\vdash^*$ la *chiusura riflessiva* e *transitiva* della relazione $\vdash$. La condizione di accettazione è:

$$\forall\, x \in I^* \mid x \in L \Leftrightarrow c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_F = \langle q, \epsilon, \gamma \rangle,\ q \in F$$

in cui:

1. La configurazione iniziale parte dallo **stato iniziale** con la **stringa ancora interamente da leggere** e la **pila vuota**
2. Viene applicato un certo numero di **transizioni**
3. La configurazione finale ha la **stringa completamente letta** e lo **stato attuale è nell'insieme degli stati finali** mentre non ci sono condizioni sullo stato della pila

Informalmente, una stringa è accettata da un *PDA* se esiste un cammino coerente con x su di esso che va dallo stato iniziale allo stato finale. La stringa deve essere letta in tutta la sua interezza.

3.7 *PDA* vs *FSA*

Alcuni linguaggi, a causa del *pumping lemma*, non possono essere riconosciuti da un *FSA*. Essi tuttavia possono essere riconosciuti da un *PDA* e prendono il nome di **linguaggi regolari**.
I *PDA* sono quindi **più espressivi** degli *FSA*.
Infatti, dato un *FSA* $A = \langle Q, I, \delta, q_0, F \rangle$ è immediato costruire un *PDA* $A' = \langle Q', i', \Gamma', q_0', Z_0', F' \rangle$ tale che $L(A) = L(A')$.
Informalmente, i *PDA*, rispetto agli *FSA*, hanno la capacità di invertire le stringhe di lunghezza indefinita e contare, proprio grazie alla loro memoria a pila.

3.8 *PDA* ciclici e aciclici

A differenza degli *FSA*, i *PDA* potrebbero non fermarsi dopo un numero finito di mosse: sono possibili **cicli** di ϵ-mosse. Tuttavia, i *PDA* ciclici non aggiungono potere espressivo alla classe dei *PDA* e questa loro caratteristica potrà essere rimossa.
Formalmente, si consideri un *PDA* A. La notazione

$$\langle q, x, \alpha \rangle \vdash_d^* \langle q', y, \beta \rangle$$

indica che:

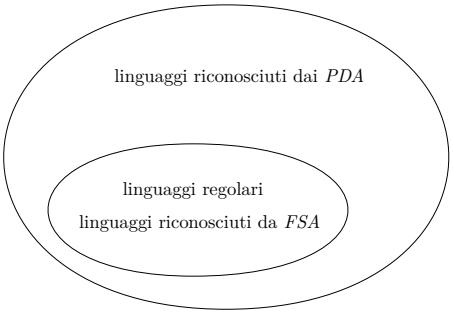


Figura 11: Linguaggi riconosciuti da *PDA* e *FSA*

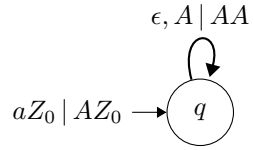


Figura 12: *PDA* ciclico

1. Il *PDA* evolve dalla *configurazione iniziale* $\langle q, x, \alpha \rangle$ alla configurazione $\langle q', y, \beta \rangle$ tramite operatore $\vdash^*$
2. Per $\beta = Z\beta'$, $\delta(q', \epsilon, Z) = \perp$ (è quindi *indefinita*)
    - $\vdash_d^*$ è una sequenza di mosse che porta a una configurazione da cui **non è possibile procedere** con una ϵ-mossa
    - per evolvere da questa configurazione è necessario leggere un simbolo in ingresso

Ciò detto, un *PDA* è **aciclico** se e solo se

$$\forall\, x \in I^*\ \exists\, (q, y) \mid \langle q_0, x, Z_0 \rangle \vdash_d^* \langle q, \epsilon, \gamma \rangle$$

e che quindi:

1. Legge sempre l'intera stringa di ingresso e, al termine di essa, si ferma
2. Non può ripetere un ciclo indefinitamente con ϵ-mosse
3. Si ferma dopo un numero finito di mosse

Ogni *PDA* ciclico può (e deve) essere trasformato nel proprio equivalente aciclico. Un *PDA* che presenta cicli, infatti, potrebbe non raggiungere mai la fine della stringa e il suo corrispondente stato di accettazione, rimanendo per sempre in un ciclo di ϵ-mosse.

3.9 Operazioni sui *PDA*

Nelle successive Sezioni (3.9.1-3.9.4) ci si occuperà di trattare la chiusura dei linguaggi riconosciuti da *PDA* rispetto a varie operazioni.

3.9.1 Complemento

La classe dei linguaggi riconosciuti da *PDA* **è chiusa** rispetto alla complementazione. Il complemento può essere costruito:

- Eliminando i cicli
- Aggiungendo stati di errore
- Scambiando stati finali con non finali
- Occupandosi di ϵ-mosse che collegano stati finali e non finali:
    - Un *PDA* potrebbe imporre l'accettazione solo alla fine di una sequenza *(finita)* di ϵ-mosse
    - Se così fosse, scambiare gli stati finali e iniziali non porterebbe a una complementazione corretta

3.9.2 Unione

La classe dei linguaggi riconosciuti dai *PDA* **non è chiusa** rispetto all'unione.

Per esempio, non esiste alcun *PDA* che riconosca $\{a^n b^n \mid n \geq 1\} \cup \left\{a^n b^{2n} \mid n \geq 1\right\}$. Tuttavia:

- $A = \{a^n b^n \mid n \geq 1\}$ è riconoscibile via *PDA*
- $B = \left\{a^n b^{2n} \mid n \geq 1\right\}$ è riconoscibile via *PDA*

Esempio intuitivo della dimostrazione:

- Se si procede riconoscendo una stringa di A ma trovando una stringa di B, pur sapendo che rimangono n caratteri b da leggere si ha perso l'informazione sul valore stesso di n
- Analogamente, riconoscendo una stringa di B ma trovando una stringa di A, pur sapendo che sono rimasti almeno n simboli nella pila non sarà possibile verificare se essi sono effettivamente n

3.9.3 Intersezione

La classe dei linguaggi riconosciuti da *PDA* **non è chiusa** rispetto all'intersezione.
Applicando la legge di *De Morgan* si verifica che:

$$\cup B = \overline{\left(\overline{A} \cap \overline{B}\right)} = \left(A^C \cap B^C\right)^C$$

Poiché i linguaggi dei *PDA* sono chiusi rispetto al complemento, se fossero chiusi rispetto all'intersezione dovrebbero essere chiusi anche rispetto all'unione, in contrario a quanto affermato nella Sezione 3.9.4

3.9.4 Differenza

La classe dei linguaggi riconosciuti da *PDA* **non è chiusa** rispetto alla differenza.
Applicando le leggi insiemistiche, si verifica che:

$$A \cap B = A - \overline{B} = A - B^C$$

Poiché i linguaggi dei *PDA* sono chiusi rispetto al complemento, se fossero chiusi rispetto alla differenza dovrebbero essere chiusi anche rispetto all'intersezione, in contrario a quanto affermato nella Sezione 3.9.3

3.10 Trasduttori a pila - *PDT*

3.10.1 Definizione formale *PDT*

Un **trasduttore a pila** (*pushdown transducer* o *PDT*) è una tupla $\langle Q, I, \Gamma.\delta, q_0, Z_0, F, O, \eta\rangle$, dove:

- Q è un **insieme di stati**, finito
- I è l'**alfabeto di ingresso**
- Γ è l'**alfabeto di pila**
  - $Z_0 \in \Gamma$ è il simbolo iniziale (il primo in *basso*) di pila
- δ è la **funzione di transizione**
- $q_0 \in Q$ è lo **stato iniziale**
- $F \subseteq Q$ è l'**insieme di stati finali**
- O è l'**alfabeto di uscita**
- η è la **funzione di uscita**:
  - $\eta : Q \times (I \cup \{\epsilon\}) \times \Gamma \to O^*$
  - **dominio**: prodotto cartesiano dell'insieme di stati, alfabeto di ingresso (compreso ϵ) e alfabeto di pila
  - **codominio**: stella di Kleene dell'alfabeto di uscita

Si noti che:

- $Q, I, \Gamma, \delta, q_0, Z_0, F$ sono definiti in modo analogo ai *PDA* nella sezione 3.1.

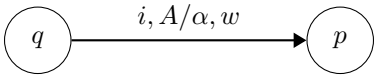


Figura 13: Transizione tra due stati in un PDT

- η è definita solo dove è definita δ
- La pila può essere necessaria perché richiesta dal linguaggio da riconoscere o perché richiesta dalla traduzione

L'illustrazione della transizione tra due stati con $\delta(q, I, A) = \langle p, \alpha \rangle$ e $\eta(q, I, A) = w$ è mostrata in Figura 13.
I PDT sono molto utilizzati per sopperire a due grosse lacune dei FST: *contare* i caratteri e *invertire* le stringhe.

3.10.2 Configurazione di un *PDT*

Una configurazione c di un PDT è una tupla $\langle q, x, \gamma, z \rangle$ in cui:

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x \in I^*$ è la **porzione non letta** della stringa d'ingresso
- $\gamma \in \Gamma$ è la stringa di **simboli nella pila**
- z è la stringa già scritta sul **nastro di uscita**

3.10.3 Transizioni di un *PDT*

Per un dato PDT T, la **relazione binaria di transizione** $\vdash_T$ nello spazio di tutte le possibili configurazioni di A è definita da:

$$\langle q, x, \gamma, z \rangle \vdash_a c' = \langle q', x', \gamma', z' \rangle, \quad z' = z\overline{z}$$

se e solo se vale una delle due condizioni:

$$x = ay,\ x' = y,\ \gamma' = A\beta,\ \gamma' = \alpha\beta,\ \delta(q, a, A) = \langle q', \alpha \rangle,\ \overline{z} = \eta(q, a, A) \tag{3.1}$$

$$x = x',\ \gamma = A\beta,\ \gamma' = \alpha\beta,\ \delta(q, \epsilon, A) = \langle q', \alpha \rangle,\ \overline{z} = \eta(q, \epsilon, A) \tag{3.2}$$

Ovverosia:

- La condizione 3.1 descrive il passaggio da una configurazione all'altra quando viene letto un simbolo in ingresso
- La condizione 3.2 prende in considerazione il caso di ϵ-mosse *(quando la testina in ingresso rimane ferma)*

3.10.4 Condizione di accettazione di un *PDT*

La condizione di accettazione di un PDT può essere descritta come:

$$\forall\, x \in I^*,\ x \in L \Leftrightarrow c_0 = \langle q_0, x, Z_0, \epsilon \rangle \vdash^* c_F = \langle q, \epsilon, \gamma, z \rangle,\ q \in F$$

La traduzione di x è definita se e solo se la stringa x è accettata

4 Macchine di Turing - *TM*

Come visto in precedenza (Sezione 3.9.2), l'unione di alcuni linguaggi riconosciuti dai *PDA* non può essere riconosciuta dai *PDA* perché la classe di questi ultimi non è chiusa rispetto all'unione.

Si consideri il linguaggio $L = \{a^n b^n c^n \mid n > 0\}$:

- La pila può essere usata per contare le a
- I simboli sulla pila possono essere usati per controllare che il numero di b sia uguale al numero di a
  - ✖ Il numero di c, di conseguenza, **non può essere contato**

Questa limitazione è dovuta alla **pila**. Infatti essa è una memoria **distruttiva**, perché una volta che il simbolo viene letto, esso è distrutto. Questa proprietà può essere dimostrata formalmente attraverso una generalizzazione del *pumping lemma* (visto in Sezione 2.3.3).
Per risolvere questo problema, sono state introdotte le **Macchine di Turing** (in Inglese *Turing Machine* o *TM*), che fanno uso di **nastri di memoria**. Essi, infatti, non sono di natura distruttiva e possono venire letti più volte.

Un diagramma semplificato di una *TM* è mostrato in Figura 14.

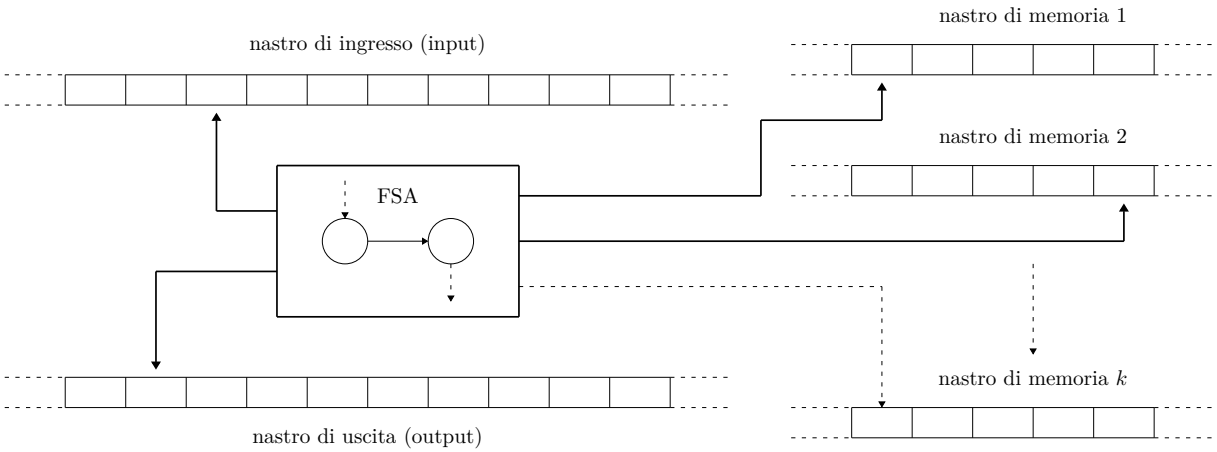


Figura 14: Diagramma semplificato di una TM

Informalmente, una *TM* è costituita da:

- **Stati** e **alfabeto** definiti come negli *FSA* (Sezione 2.2)
- Un nastro di **ingresso**, con la testina T_I
- Un nastro di **uscita**, con la testina T_F
- Un **dispositivo di controllo**
- Un **alfabeto di memoria**
- k nastri di **memoria**:
  - ogni nastro presenta una **testina di lettura**, $T_0, \ldots, T_k$
  - rappresentati come **sequenze infinite**
  - gli spazi **vuoti** sono occupati da simboli speciali detti `blank` (anche `b`, $-$)
  - inizialmente, ogni nastro di memoria è **vuoto**
  - in ogni momento, i nastri contengono **un numero finito** di simboli non-`blank`

Un passo di **computazione** di una *TM* è basato su:

- Un **simbolo letto** dal nastro di ingresso
- K simboli, uno per ogni nastro di **memoria**
- **Stato** del dispositivo di controllo

e può eseguire le seguenti operazioni **(mosse)**:

1. **Cambio di stato** del dispositivo di controllo
2. **Sovrascrittura** di un simbolo in nelle celle sottostanti alle testine dei nastri di memoria
3. **Spostamento** delle $k+1$ testine:
   - $\rightarrow$ i k nastri di memoria possono spostarsi a *sinistra* o *destra*
   - $\rightarrow$ il nastro di uscita può spostarsi solo a *destra*
4. Nessuna operazione, fermandosi **definitivamente**

La direzione di spostamento di ogni testina deve essere specificata esplicitamente, indicando:

$\rightarrow$ R per *destra di una posizione*
$\leftarrow$ L per *sinistra di una posizione*
$\downarrow$ S per *nessuno spostamento*

4.1 Definizione formale *TM*

Formalmente, una *TM* con k nastri è una tupla di 9 elementi $\langle Q, I, \Gamma, O, \delta, \eta, q_0, Z_0, F \rangle$ dove:

- Q è un insieme di **stati**, finito
- I è l'**alfabeto** di **ingresso**
- O è l'**alfabeto** di **uscita**
- Γ è l'**alfabeto** di **memoria**
- δ è la **funzione** di **transizione**:
  - $\delta : (Q - F) \times I \times \Gamma^k \rightarrow Q \times \Gamma^k \times \{R, L, S\}^{k+1}$
  - **dominio**: prodotto cartesiano di tutti gli stati meno quelli finali, alfabeto di ingresso e alfabeto di memoria *elevato al numero di nastri*
  - **codominio**: prodotto cartesiano di tutti gli stati, dell'alfabeto di memoria elevato al numero di nastri e dei possibili movimenti dei nastri *elevati a* $k+1$
    $\rightarrow$ $k+1$ perché si riferisce ai k nastri di memoria e il nastro di uscita
  - può essere **parziale**, come avviene nei *FSA* (Sezione 2.2)
  - è definita in modo tale che non ci siano transizioni uscenti da uno stato finale
- η è la **funzione** di **uscita**:
  - $\eta : (Q - F) \times I \times \Gamma^k \rightarrow O \times \{R, S\}$
  - **dominio**: coincide con quello della funzione δ
  - **codominio**: prodotto cartesiano dell'alfabeto di uscita con i possibili movimenti del nastro di uscita
  - può essere **parziale**
- $q_0 \in Q$ è lo stato **iniziale**
- $Z_0 \in \Gamma$ è il **simbolo iniziale** di **memoria**
- $F \subseteq Q$ è l'**insieme di stati finali**

Se il valore di k non è indicato, la *TM* viene semplicemente chiamata *TM multinastro (con un numero arbitrario di nastri)*.
La presenza di O e η non è obbligatoria. Essi infatti sono definiti se è previsto un *nastro di uscita*, come per esempio nelle *TM traduttrici* (Sezione 4.7).

Si consideri una transizione in cui:

- q_0, q_1 sono due **stati**
- i è il **simbolo** di **ingresso**
- A_j è il **simbolo** letto dal j-esimo nastro di memoria
- A'_j è il **simbolo** che rimpiazza A_j
- M_0 è la **direzione** della testina del nastro di ingresso T_i
- M_j, $\forall\, 1 \leq j \leq k$ è la **direzione** della testina del j-esimo nastro di memoria

La notazione grafica di questa situazione è mostrata in Figura 15.

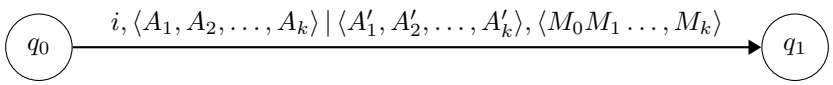


Figura 15: Transizione di una TM

4.2 Configurazione di una *TM*

Analogamente a quanto visto *PDA* (Sezione 3.4), anche le *TM* ammettono la definizione di **configurazione**. Essa dovrà includere:

- Lo **stato** del dispositivo di controllo
- La **stringa** sul nastro di ingresso e la **posizione della testina**
- la **stringa** e la **posizione della testina** per ogni nastro di memoria

Formalmente: una configurazione c di una *TM* con k nastri di memoria è la tupla di $k+3$ elementi:

$$c = \langle x \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \ldots, \alpha_k \uparrow A_k\beta_k, u \uparrow o \rangle$$

- $q \in Q$ è lo **stato corrente** del dispositivo di controllo
- $x, y \in I^*, i \in I$ rappresentano il **contenuto** del nastro di ingresso
- $\alpha_r, \beta_r \in \Gamma^*,\ A_r \in \Gamma\ \forall r\, 1 \leq r \leq k$ rappresentano il contenuto dei **nastri di memoria**
  - le testine di ogni nastro sono posizionate sulla cella che memorizza il primo simbolo della stringa che segue il simbolo $\uparrow$
  - $\uparrow \notin \{I \cup \Gamma \cup O\}$
- uo è il contenuto del **nastro di uscita** (se definito, come nelle *TM* traduttrici viste in Sezione 4.7)
  - $u \in O^*$ è la stringa già scritta sul nastro di uscita
  - $o \in O$ è l'ultimo carattere scritto

4.2.1 Configurazione iniziale

La **configurazione iniziale** c_0 di M è del tipo: $c_0 = \langle q_0, \uparrow iy, \uparrow Z_0, \ldots, \uparrow Z_0, \uparrow Z_0, \uparrow \mathtt{b} \rangle$ dove:

- $q = q_0$, lo stato attuale è quello iniziale
- $x = \epsilon$, nessun carattere della stringa è ancora stata letta
- $A_r = Z_0, \alpha_r = \beta_r = \epsilon\ \forall r$, tutti i nastri di memoria sono vuoti
- $u = \epsilon, o = \mathtt{b}$ il nastro di uscita è vuoto *(se definito)*

Inoltre, tutte le testine sono all'inizio del corrispondente nastro.

4.3 Transizioni di una *TM*

La relazione di **transizione** $\vdash_M$ (detta anche *mossa* o *passo computazionale*) fra due configurazioni c e c' di una *TM* multinastro M è definita nel modo seguente:
Siano c, c' le due configurazioni tra le quali si esegue la transizione:

$$c = \langle q, x \uparrow iy, \alpha_1 \uparrow A\beta_1, \ldots, \alpha_k \uparrow A\beta_k, u \uparrow o \rangle$$
$$c = \langle q', x' \uparrow i'y', \alpha'_1 \uparrow A'\beta'_1, \ldots, \alpha'_k \uparrow A'\beta'_k, u' \uparrow o' \rangle$$

Sia δ la funzione di transizione, definita come:

$$\delta(q, i, A_1, \ldots, A_k) = \langle p, C_1, \ldots, C_k, N, N_1, \ldots N_k \rangle$$

con:

$$p \in Q,\ N \in \{R, L, S\},\ C_r \in \Gamma,\ N_r \in \{R, L, S\}\ \forall 1 \leq r \leq k$$
$$x = \overline{xi},\ y = \overline{jy},\ \alpha_r = \overline{\alpha_r A_r},\ \beta_r = \overline{B_r \beta_r}$$

Sia η la funzione di uscita, definita come:

$$\eta(q, i, A_1, \ldots, A_k) = \langle v, M \rangle$$

con:

$$v \in O \quad M \in \{R, S\}$$

Allora la transizione $c \vdash_M c'$ (da c a c') se e solo se:

$$p = q' \tag{4.1}$$

Se vale **una delle condizioni mutuamente esclusive**:

$$x = x',\ i = i',\ y = y',\ N = S \tag{4.2}$$
$$x' = xi,\ i' = \overline{j},\ y' = \overline{y},\ N = R \tag{4.3}$$
$$x' = \overline{x},\ i' = i,\ y' = iy,\ N = L \tag{4.4}$$

Inoltre, per $1 \leq r \leq k$ deve valere anche **uno dei casi mutuamente esclusivi**:

$$\alpha'_r = \alpha_r,\ A'_r = C_r,\ \beta'_r = \beta_r,\ N_r = S \tag{4.5}$$
$$\alpha'_r = \alpha_r C_r,\ A'_r = B_r,\ \beta'_r = \overline{\beta_r},\ N_r = R \tag{4.6}$$
$$\alpha'_r = \overline{\alpha_r},\ A'_r = \overline{A_r},\ \beta'_r = C_r \beta_r,\ N_r = L \tag{4.7}$$

Infine anche **una delle due condizioni mutualmente esclusive**:

$$u' = u,\ o' = v,\ M = S \tag{4.8}$$
$$u' = uv,\ o' = \text{b},\ M = R \tag{4.9}$$

4.3.1 Singificato delle condizioni di transizione della *TM*

- La condizione 4.1 vincola lo stato di c' a essere quello di arrivo della transizione
- Le condizioni dal 4.2 al 4.4 definiscono l'evoluzione del nastro di ingresso nel passaggio da c a c':
  - Se la testina rimane ferma (condizione 4.2, in cui $N = S$), le tre parti in cui essa divide il nastro *(parte sinistra, destra e simbolo corrente)* rimangono invariate tra le due configurazioni
  - Se la testina si muove a destra (condizione 4.3, in cui $N = R$), la parte a sinistra della testina in c' conterrà anche il simbolo corrente di c, il simbolo corrente di c' sara il primo simbolo della parte destra in c e la rimanente parte destra di c sarà la parte destra di c'
  - Se la testina si muove a sinistra (condizione 4.4, in cui $N = L$), laa parte a sinistra della testina in c' conterrà tutti i simboli della parte sinistra in c, tranne l'ultimo che diverrà il simbolo corrente di c' e la parte destra di c' sarà composta dal simbolo corrente in c seguito dalla sua parte destra
- Le condizioni dal 4.5 alla 4.7 specificano l'evoluzione dei nastri di memoria in analogia con i precedenti casi. In particolare:
  - La testina rimane ferma, condizione 4.5
  - La testina si muove a destra, condizione 4.6
  - La testina si muove a sinistra, condizione 4.7
- Le condizioni 4.8 e 4.9 specificano l'evoluzione del nastro di uscita
  - Non è specificato il comportamento per $N = L$ perché la testina del nastro di uscita si muove solo a destra
  - Se la *TM* non ha un nastro di uscita, queste condizioni non vanno considerate

4.4 Condizioni di accettazione di una *TM*

Sia M una TM multinastro Una stringa $x \in I^*$ è **accettata** da M se e solo se:

$$c_0 = \langle q_0, \uparrow x, \uparrow Z_0, \ldots, \uparrow Z_0 \rangle \vdash_M^* \langle q, x' \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \ldots, \alpha_k \uparrow A_k\beta_k \rangle$$

dove:

- c_F è detta **configurazione finale**
- $\vdash_M^*$ è la chiusura riflessiva e transitiva della relazione $\vdash_M$
- $q \in F$, quindi lo stato q fa parte dell'insieme di stati finali
- Le testine possono trovarsi in un qualsiasi punto dei rispettivi nastri di memoria

Il linguaggio *accettato* da M è definito come:

$$L(M) = \{x \mid x \in I^* \text{ e } x \text{ è accettato da } M\}$$

Intuitivamente, il linguaggio riconosciuto da una TM M è composto da tutte e sole le stringhe che permettono di andare dallo stato iniziale a uno degli stati finali. Poiché il nastro di ingresso è in grado di muoversi in entrambe le direzioni, non è richiesto che al termine dell'esecuzione la testina si trovi al termine della stringa di ingresso.
Una volta che M raggiunge uno stato finale, per definizione della funzione di transizione δ, non potrà più lasciarla e la computazione termina.

4.5 Operazioni sulle *TM*

È possibile dimostrare che le TM **sono chiuse** rispetto a:

- **Intersezione**
- **Unione**
- **Concatenazione**
- **Stella di Kleene**

Perché una TM può facilmente simulare due altre TM (siano esse in *serie* o *parallelo*), si dimostra la chiusura rispetto alle prime due operazioni. Di conseguenza, si dimostra la chiusura rispetto alle altre.

Analogamente, si dimostra che **non sono chiuse** rispetto a:

- **Complemento**
- **Differenza**, conseguenza della non chiusura rispetto al complemento

La (non) chiusura rispetto a queste due operazioni è dovuta alla presenza di *cicli* all'interno delle TM.
Infatti, se essi non fossero presenti in una TM sarebbe sufficiente definire l'insieme di stati di arresto e partizionarli in stati di di *accettazione* e *non accettazione*. I problemi sorgono qualora una computazione non dovesse terminare (come visto in Sezione 5.3).

4.6 Proprietà delle *TM*

Di seguito sono enunciate alcune proprietà delle TM:

1. Ogni TM è equivalente ad un'opportuna TM dotata solo di **due stati non finali** e di **uno finale**
   - può essere necessario accrescere l'alfabeto
   - di conseguenza, sono sufficienti 3 stati per implementare qualsiasi algoritmo
2. Ogni TM è equivalente ad un'opportuna TM avente un alfabeto formato solo da **due simboli distinti**
   - può essere necessario accrescere il numero di stati
   - di conseguenza, sono sufficienti 2 simboli *(più il simbolo* `blank`*)* per implementare qualsiasi algoritmo

Le due proprietà impongono una scelta: è possibile ridurre gli stati di una TM agendo sull'alfabeto (1) o viceversa ridurre la dimensione dell'alfabeto agendo sul numero di stati (2). La scelta si riduce quindi ad un problema progettuale.

4.7 *TM* traduttrice

Quando si definiscono le *TM* con il nastro di uscita, esse diventano dei *trasduttori*, ossia possono essere usate per tradurre stringhe di I^* in stringhe di O^*. In questo caso, le *TM* vengono viste come le tuple di 9 elementi viste in Sezione 4.1.

4.7.1 Traduzione tramite *TM*

Una *TM* multinastro M definisce una **traduzione** $\tau_M : I^* \to O^*$ secondo la regola seguente:

$$\tau_M(x) = y \text{ se e solo se } \langle q_0, \uparrow x, \uparrow Z_0, \ldots, \uparrow Z_0, \uparrow \mathtt{b} \rangle \vdash_M^* \langle q, x' \uparrow iy, \alpha_1 \uparrow A_1\beta_1, \ldots, \alpha_k \uparrow A_k\beta_k, y \uparrow o \rangle$$

con $q \in F$.
In generale, una *TM* M definisce una **traduzione parziale** $\tau_M : I^* \to O^*$. Infatti, τ_M è indefinita se:

1. M raggiunge una configurazione di arresto il cui stato non appartiene a F
2. M non si ferma mai quando opera su x

Intuitivamente, una stringa x viene tradotta in una stringa y da una *TM* M se esiste un cammino che parte da una configurazione **iniziale** con x sul nastro di ingresso e termina in una configurazione **finale** con y sul nastro di uscita.

4.8 Confronto di *TM* con altre macchine

Si vuole ora confrontare la classe di macchine computazionali appartenente alle *TM* con altri tipi di macchine.

4.8.1 *TM* vs *PDA*

Come già visto, i linguaggi $a^n b^n c^n$ e $a^n b^n \cup a^n b^{2n}$ non possono essere riconosciuti da un *PDA* (Sezione 2.3.3.1) mentre tramite *TM* il riconoscimento funziona. Ogni linguaggio riconoscibile da un *PDA* può essere riconosciuto da una *TM*: si può sempre costruire una *TM* che usa un nastro di memoria come se fosse una pila.
I linguaggi accettati dalle *TM* sono detti **ricorsivamente enumerabili**.

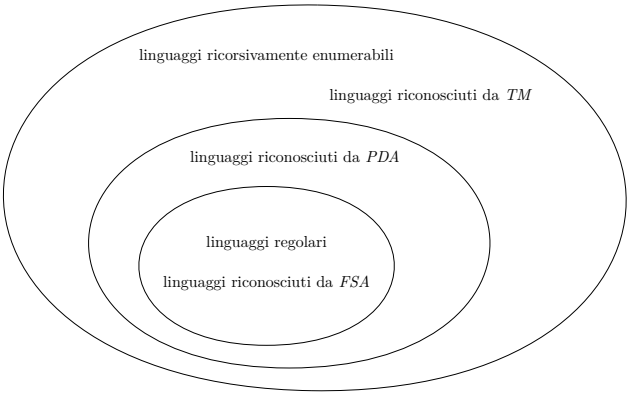


Figura 16: Relazione tra linguaggi

4.8.2 *TM* vs macchine di *Von Neumann*

Le *TM* possono simulare una macchina di *Vonn Neumann* (in Inglese *VNM*), un modello astratto di computer. La principale differenza avviene nell'accesso alla memoria: mentre nelle *TM* è *sequenziale*, nelle *VNM* è *diretto*. Tuttavia, il metodo di accesso alla memoria non influenza il potere espressivo di una macchina: non cambia la classe di problemi risolvibili con essa, ma può cambiarne la complessità. Infatti si tramite *TM* si possono anche calcolare funzioni ed eseguire algoritmi *sebbene implementare questo tipo di operazioni possa essere estremamente complicato.*

In conclusione, la *TM* è un modello più astratto di computer con accesso **sequenziale** al suo spazio di memoria.

4.9 Memoria delle *TM*

Esistono *TM* con diversi modelli di memoria:

- A nastro **singolo** (Figura 17):
  - Normalmente è **illimitato** in entrambe le direzioni
  - Serve da contemporaneamente da **ingresso, uscita e memoria**
  - È il modello più simile alla macchina originalmente ideata da *Alan Turing*
- A nastro **bidimensionale** (Figura 18):
  - È presente **una testina per ogni dimensione**
  - Può essere generalizzato a un numero *arbitrario* di dimensioni

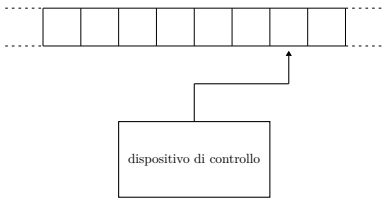


Figura 17: *TM* a nastro singolo

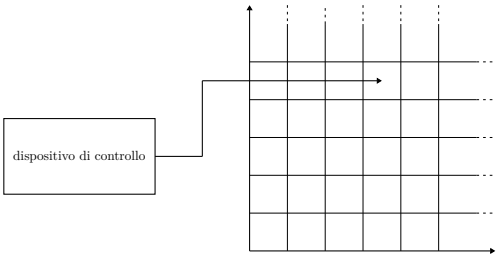


Figura 18: *TM* bidimensionale

Entrambi i modelli di *TM* sono **equivalenti**, poiché riconoscono la stessa classe di linguaggi. A tal proposito, si consulti la Figura 19, che mostra la configurazione della *TM* a nastro singolo M':

- $c(T_x)'$ rappresenta il contenuto non vuoto del nastro T_x di M alla sinistra della testina di T_x
- $c(T_x)''$ rappresenta il contenuto dello stesso nastro alla destra della testina, compreso il carattere sotto di essa
- $*$ e \$ sono simboli che non appartengono a $I \cup \Gamma \cup O$ e vengono usati per marcare i limiti fra il contenuto dei diversi nastri e posizioni della testina (rispettivamente)
- $\overline{q}$ è un'opportuna codifica dello stato M

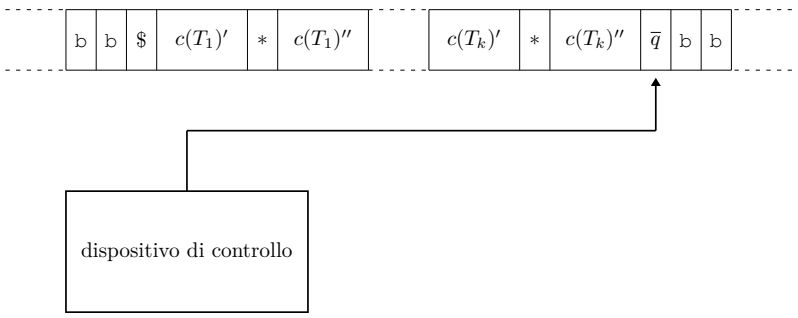


Figura 19: Equivalenza tra *TM* multinastro e *TM* a nastro singolo

5 Modelli operazionali non deterministici

Tutti i modelli considerati fino ad ora (*FSA*, *PDA*, *TM*, ...) sono **deterministici**: una volta fissato uno *stato iniziale* e un *ingresso*, la loro evoluzione è *univocamente determinata*. In certe situazioni, però, il modello che si desidera modellare non può essere descritto in modo così deterministico, in quanto l'osservatore non possiede informazioni sufficientemente accurate da consentirgli di prevederne l'esatta evoluzione per ogni configurazione. Per esempio, si consideri la serie di istruzioni:

```
if x >= y then:
  max := x

if y >= x then:
  max := y
```

Essa definisce il massimo tra due numeri `x` e `y`. Tale notazione rappresenta una specifica non deterministica: non viene illustrato il comportamento per `x == y` e l'applicazione di entrambe le regole risulta in un output valido.
Normalmente, nei linguaggi di programmazione, si usa una precedenza di tipo lessicale: viene applicata la prima condizione scelta.
Un'esempio analogo può essere fatto sfruttando gli algoritmi di *ricerca binaria*. Essi infatti compiono una *simulazione* di algoritmi non deterministici:

```
while true:
  if elemento-cercato in radice:
    end

  search in left-subtree
  search in right-subtree
```

Poiché non è possibile determinare a priori quale dei due cammini *(sinistro o destro)* sia migliore, la scelta della priorità nei cammini è spesso **arbitraria**. Alternativamente, non viene fatta una scelta del cammino da intraprendere ma entrambi vengono esplorati in **parallelo**.
In entrambi i casi, grazie al non determinismo, è possibile implementare l'algoritmo senza la necessità di usare *backtracking*, che sarebbe invece obbligatorio nel caso si cercasse di usare un sistema deterministico.

Il **non determinismo** (*ND* per brevità) è quindi un modello di computazione. Viene spesso sfruttato nei linguaggi di programmazione per consentire la computazione parallela.
Il *ND* viene applicato a vari modelli computazionali, inclusi tutti quelli visti fin'ora.

5.1 *FSA* non deterministici

Un *FSA* non deterministico (*NFA*) è una tupla di 5 elementi $\langle Q, I, \delta, q_0, F\rangle$:

- Q, I, q_0, F sono definiti come in un *FSA* (Sezione 2.2)
- δ è la **funzione di transizione**:
  - $\delta : Q \times I \to \wp(Q)$
    $\to$ $\wp(Q)$ rappresenta l'insieme delle parti di Q
    $\to$ gli elementi di $\wp(Q)$ sono insiemi di stati
  - δ^* è definito induttivamente a partire da δ:
    1. $\delta^*(q, \epsilon) = \{q\}$
    2. $\delta^*(q, y \cdot i) = \bigcup_{q' \in \delta^*(q,y)} \delta(q', i)$
       $\to$ i è l'ultimo carattere della stringa di ingresso

Nel caso di *FSA* accettori con *ND*, si dice che $x \in I^*$ è **accettata** da un *NFA* $\langle Q, I, \delta, q_0, F \rangle$ se e solo se $\delta^*(q_0, x) \cap F \neq \emptyset$.
Informalmente: tra le varie possibili esecuzioni (a parità di ingresso) dell'*NFA*, è sufficiente che una di esse vada a buon fine *(la stella di Kleene delle transizioni δ^* ha almeno uno stato che fa parte dell'insieme di stati finali)*.
Questa definizione prende il nome di **non determinismo esistenziale**, che si oppone al non determinismo **universale**: $\delta^*(q_0, x) \subseteq F$.

Gli *NFA* non sono più potenti degli *FSA*, ma presentano due grossi vantaggi:

- Può essere più semplice progettare un *NFA*
- Il numero di stati di un *NFA* può essere esponenzialmente più minore dell'analogo *FSA*

Per maggiori dettagli, si consulti la Sezione 5.1.2

5.1.1 Esempio di un *NFA*

In Figura 20 è mostrato un esempio di *NFA*. In esso, infatti, $\delta(q_0, a) = \{q_1, q_2\}$ e quindi la funzione di transizione **non è univocamente definita**.

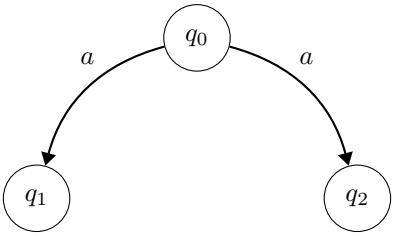


Figura 20: Esempio di un *NFA*

5.1.2 *DFA* vs *NFA*

Si osservi il *NFA* in Figura 21. Partendo da q_0 e leggendo `ab`, l'automa raggiunge uno stato che appartiene all'insieme $\{q_3, q_4, q_5\}$. Negli *NFA* viene ancora chiamato **stato** l'insieme dei possibile stati in cui esso può trovarsi durante l'esecuzione.

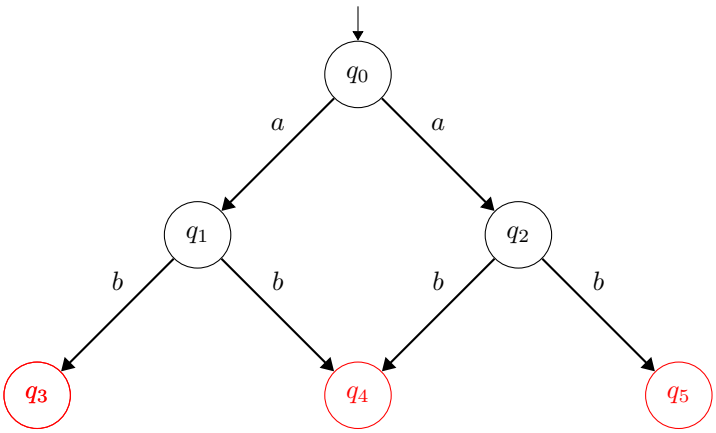


Figura 21: *DFA* e *NFA*

Formalmente, *NFA* e *DFA* **hanno lo stesso potere espressivo.** Dato un *NFA*, A, è possibile costruire un *DFA* A_D che accetti il medesimo linguaggio.

Sia $A = \langle Q, I, \delta, q_0, F \rangle,\ \delta : Q \times I \to \wp(Q)$. Si definisca $A_D = \langle Q_D, I, \delta_D, q_{0D}, F_D \rangle$ con:

- $Q_D = \wp(Q)$
  $\to$ l'insieme dei suoi stati è uguale all'insieme delle parti degli stati di A
- $\delta_D(q_D, i) = \bigcup_{q \in q_D} \delta(q, i)$
  $\to$ se un insieme di stati è raggiungibile a partire da uno stato del *NFA*, allora tale relazione viene preservata nel *DFA* sfruttando la costruzione degli stati come insiemi di stati del *NFA*
- $F_D = \{q_D \mid q_D \in Q_D \wedge F \cap q_D \neq \emptyset\}$
  $\to$ l'insieme dei suoi stati finali è dato dall'insieme di stati finali di A raggiungibili senza *ND*
- $q_{0d} = \{q_0\}$
  $\to$ il suo stato iniziale è uguale allo stato iniziale di A

Gli *NFA* non sono quindi più potenti dei corrispondenti *DFA* ma sono di dimensione ridotta. Il precedente teorema, infatti, produce un insieme di stati di cardinalità 2^n partendo da n stati di partenza del *NFA*. Inoltre, la formalizzazione più naturale di un problema è quella descritta mediante un *NFA*.

5.1.3 Conversione da *NFA* a *DFA*

Operativamente, per convertire un *FSA* in un *DFA* è necessario:

1. Creare l'insieme vuoto di stati del *DFA* Q'
2. Creare la tabella di transizione del *DFA* T'
3. Aggiungere lo stato iniziale del del *NFA* a Q'
4. Aggiungere le transizioni dello stato iniziale alla tabella di transizione T'
   $\to$ se lo stato di partenza porta a più stati per un certo input, allora essi vanno trattati come un singolo stato del *DFA*
5. Se un nuovo stato è presente in T':
   $\to$ Aggiungere il nuovo stato a Q'
   $\to$ Aggiungere le sue transizioni a T'
6. Ripetere il passo 5 finché non vengono più aggiunti nuovi stati a T'
7. T' è ora la tabella di transizione completa del *DFA* ricercato

5.2 *PDA* non deterministici

Un *PDA* non deterministico (*NPDA*) è una tupla di 7 elementi $\langle Q, I, \Gamma, \delta, q_0, Z_0, F \rangle$:

- $Q, I, \gamma, q_0, Z_0, F$ sono definiti come in un *PDA* (Sezione 3.1)
- δ è la **funzione di transizione**:
  - $\delta : Q \times (I \cup \{\epsilon\}) \times \Gamma \to \wp_F(Q \times \Gamma^*)$
  - $\wp_F$ indica l'insieme delle parti **finito** dell'insieme Q
    $\to$ Γ^* è un insieme **infinito**

Inoltre, la relazione $\vdash$ su $Q \times I^* \times \Gamma^*$ è definita da $\langle q, x, \gamma \rangle \vdash \langle q', x', \gamma' \rangle$ se e sono se è valida una delle due condizioni mutualmente esclusive:

$$x = ay, \quad x' = y, \quad \gamma = a\beta, \quad \gamma' = \alpha\beta, \quad \langle q', \alpha \rangle \in \delta(q, a, A) \tag{5.1}$$

$$x = x', \quad \gamma = a\beta, \quad \gamma' = \alpha\beta, \quad \langle q', \alpha \rangle \in \delta(q, \epsilon, A) \tag{5.2}$$

La stringa $x \in I^*$ è accettata dall'automa se e solo se:

$$\langle q_0, x, Z_0 \rangle \vdash^* \langle q, \epsilon, \gamma \rangle, \quad q \in F, \gamma \in \Gamma^*$$

Informalmente, una stringa è accettata da un *NPDA* se esiste un cammino coerente con x che va dallo stato iniziale a uno stato finale quando essa viene letta **interamente**.

I *PDA*, tuttavia, sono intrinsecamente non deterministici. Nella loro definizione era infatti stato aggiunto il vincolo per cui:

$$\delta(q, \epsilon, A) \neq \perp \Rightarrow \delta(q, I, A) = \perp,\ \forall i \in I$$

Informalmente, se una transizione da uno stato è una ϵ-mossa, allora la stessa transizione non può essere definita per nessun altro ingresso.
Rimuovere questo vincolo, infatti, priverebbe i *PDA* del loro non determinismo. Analogamente è possibile avere non determinismo cambiando la funzione di transizione di *PDA*, modificando al contempo la transizioni tra configurazioni e la condizione di accettazione.

5.2.1 Chiusura delle *NPDA*

Come per gli altri formalismi con *ND*, è possibile dimostrare che un *NPDA* può riconoscere ogni linguaggio riconoscibile tramite *DPDA*. Tuttavia, siccome gli *NPDA* sono più potenti dei *DPDA*, la classe di linguaggi riconosciuti dai primi è più ampia di quella riconosciuta dai secondi e di conseguenza non è scontato che valgano le stesse proprietà di chiusura.
Gli *NPDA*, infatti, sono chiusi rispetto all'unione. È possibile costruire un *NPDA* che è collegato con due ϵ-mosse agli stati iniziali di due *DPDA*, così mostrando una chiusura rispetto all'unione (come mostrato nella Figura 22). Tuttavia, non rimanendo chiusi rispetto all'intersezione, non possono *(e non sono)* essere chiusi rispetto al complemento.

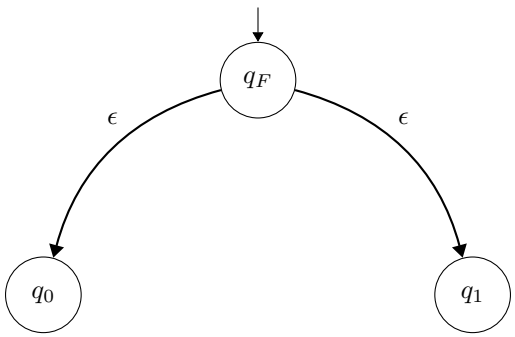


Figura 22: Intersezione di *PDA* tramite *NPDA*

5.2.2 Complemento e *ND*

Se una macchina è deterministica e la sua computazione termina, il **complemento** può essere ottenuto:

1. **Completando** la macchina
2. **Scambiando gli stati** di accettazione con quelli di non accettazione

Tuttavia, il non determinismo *(analogamente alla non terminazione)* rende questo approccio inapplicabile. Come nei *DPDA*, le computazioni nei *NPDA* possono sempre essere fatte terminare.
Tuttavia, si possono avere due computazioni del tipo:

$$c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_1 = \langle q_1, \epsilon, \gamma \rangle$$
$$c_0 = \langle q_0, x, Z_0 \rangle \vdash^* c_2 = \langle q_2, \epsilon, \gamma \rangle$$

Con:

$$q_1 \in F \quad q_2 \notin F$$

Con questa configurazione, scambiando stati di accettazione e non, x verrebbe comunque accettato (e di conseguenza il complemento non sarebbe valido).

5.2.3 Conseguenze della chiusura delle *NPDA*

Grazie a questa proprietà degli *NPDA* è possibile riconoscere il linguaggio generato dall'unione di linguaggi $\{a^n b^n \mid n \geq 1\} \cup \left\{a^n b^{2n} \mid n \geq 1\right\}$.

5.2.4 Linguaggi riconosciuti dalle *NPDA*

I linguaggi riconosciuti da *NPDA* prendono il nome di **linguaggi non contestuali** (o **context-free**).

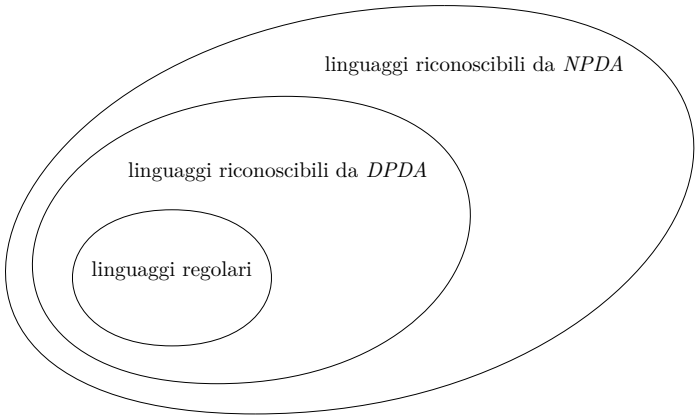


Figura 23: Linguaggi non contestuali

5.3 *TM* non deterministiche

Un *TM* deterministico (*NTM*) è un tupla di 9 elementi $\langle Q, I, \Gamma, O, \delta, \eta, q_0, Z_0, F\rangle$:

- $Q, I, \Gamma, O, q_0, Z_0, F$ sono definiti come in una *TM* (Sezione 4.1)
- δ è la **funzione di transizione**:
  - $\delta : (Q - F) \times I \times \Gamma^k \to \wp\left(Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times \{R, S\}\right)$
- η è la **funzione di uscita**:
  - $\eta : (Q - F) \times I \times \Gamma^k \to \wp\left(O \times \{R, S\}\right)$

Contrariamente a quanto avvenuto per la definizione delle *NPDA* (Sezione 5.2), non è necessario specificare che nella definizione δ l'insieme delle parti sia finito. Ciò è dovuto al fatto che l'insieme $Q \times \Gamma^k \times \{R, L, S\}^{k+1} \times \{R, S\}$ è finito, poiché costruito dal prodotto cartesiano di insiemi finiti.

5.3.1 Albero di computazione di una *NTM*

Per come è definita la funzione di transizione di M, se si considera una computazione di M su una stringa in ingresso, essa è ben descritta da un albero di configurazioni in cui è inserita ogni configurazione raggiungibile dalla configurazione iniziale. Una parola viene accettata se esiste almeno un cammino che termina in una configurazione finale. Un esempio di albero di configurazione per una *NTM* è mostrato in Figura 24, dove:

- I cerchi indicano configurazioni di accettazione
- I rettangoli indicano configurazioni di `halt` ma non di accettazione
- Le linee tratteggiate configurazione che non terminano

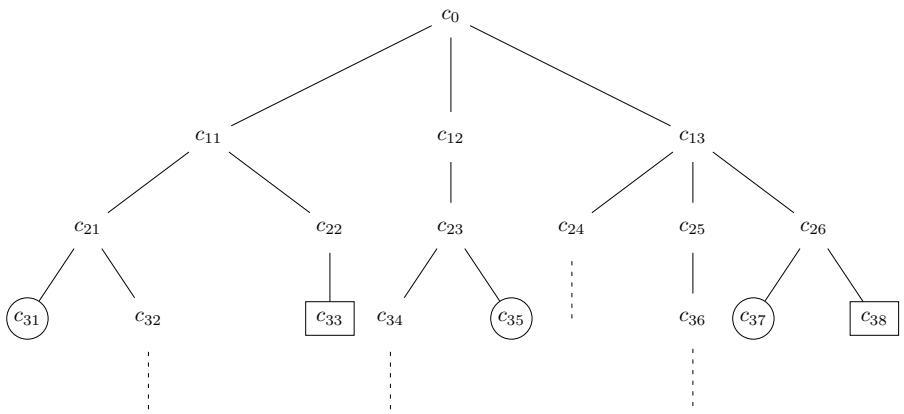


Figura 24: Albero delle computazioni di una *NTM*

5.3.2 Accettazione di una stringa da un *NTM*

Una stringa $x \in I^*$ è accettata da una *NTM* se e solo se esiste una computazione che termina in uno stato di accettazione. Il problema dell'accettazione di una stringa si può ridurre quindi alla visita di un albero di computazione.
Per cercare la via verso uno stato di accettazione, si riconoscono due tipi di approccio:

- Visita in *"profondità"*, detta **depth first**
  - Un percorso viene seguito fino al suo termine
  - Quando viene raggiunta la fine, un'altro ramo viene selezionato ed esplorato tramite *backtracking*
- Visita in *"ampiezza"*, detta **breadth first**
  - Si crea una coda di nodi da visitare
  - Ogni volta che un nuovo nodo viene trovato, i suoi successori si aggiungono alla fine della coda

Si noti tuttavia che se parola non viene accettata, una *TM* può entrare in uno ciclo infinito e quindi non terminare mai la sua computazione. Questo comportamento può essere causato dalla presenza di rami di lunghezza infinita all'interno dell'albero delle configurazioni. Una ricerca di tipo **depth first** non può funzionare per questo tipo di problema, perché finirebbe con molta probabilità in un ciclo infinito.
Una macchina deterministica può simulare l'attraversamento di un albero tramite una ricerca **breadth first**.

5.3.3 *DTM* vs *NTM*

È possibile costruire una *DTM* che visita un'albero di computazione costruito da una *NTM* livello dopo livello, implementando una ricerca di tipo **breadth first**.
Quindi, data una *NTM*, è possibile costruire:

- Una *DTM* analoga che determina se la *NTM* riconosce una stringa x
- La sua equivalente *DTM*

Infine, può essere dimostrato che il *ND* **non aggiunge potere espressivo** alle *TM*.

5.3.4 *NPDA* vs *NTM*

Nella Figura 25 è possibile osservare varie possibilità di relazione confronto tra *NPDA* e *NTM*, specialmente nei casi:

a) $NTM \cup NPDA \neq \emptyset$, quindi *NTM* e *NPDA* possono riconoscere dei linguaggi in comune
b) $NPDA \subseteq NTM$, quindi le *NTM* rappresentano una sotto categoria delle *NPDA*
c) $NTM \subseteq NPDA$, quindi le *NPDA* rappresentano una sotto categoria delle *NTM*

d) $NTM \equiv NPDA$, quindi le due categorie coincidono

Tuttavia, si dimostra che i casi:

- (a), (c) sono **falsi** perché una *NTM* può simulare un *NPDA* usando il nastro come pila
- (d) è **falso** perché la pila é una memoria distruttiva, al contrario del nastro

Quindi rimane vero il caso (c) e le *NTM* hanno potere espressivo superiore.

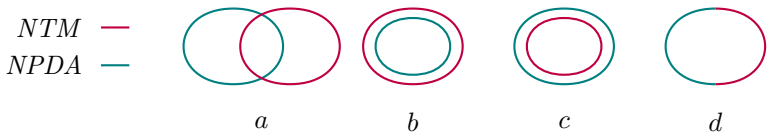


Figura 25: Confronto tra *NPDA* e *NTM*

6 Grammatiche

Fino ad ora, gli automi sono stati usati come modelli astratti nei problemi di riconoscimento dei linguaggi. Dato un *riconoscitore* A e il suo linguaggio di ingresso I, il linguaggio da esso definito è:

$$L(A) = \{x \mid x \in I^*,\ x \text{ è accettata da } A\}$$

Tuttavia, esistono altri tipi di formalismi per descrivere un linguaggio: le **grammatiche formali**. Essi sono costituiti da un **insieme di regole** (sono quindi di tipo *generativo*) che costruiscono le frasi di un linguaggio. Una grammatica formale genera stringhe di un linguaggio attraverso un processo detto di *riscrittura*. Esso è costituito da un insieme di tecniche per sostituire sottotermini di una formula con altri termini. Non si applica solo ai linguaggi in senso naturale ma a un'ampia gamma di contesti.

Esempi di riscrittura:

- $A \land B$ viene riscritto come $\neg(\neg A \lor \neg B)$
- $\neg A \lor B$ viene riscritto come $A \to B$

In generale, un meccanismo di riscrittura è un insieme di *regole linguistiche* che descrivono l'*oggetto principale* (come la frase) come sequenza di *componenti*. Ogni componente può essere poi *"raffinato"* da oggetti più dettagliati e così via, fino a ottenere una sequenza di *oggetti elementari*.
Una grammatica è quindi composta da:

- Oggetto principale, detto **simbolo iniziale**
- Inseme di componenti da sostituire durante il processo di derivazione, detti **simboli non terminali**
- Insieme di elementi di base, detti **simboli terminali**
- Regole di sostituzione, dette **produzioni**

Noam Chomsky, uno degli studiosi più importanti delle grammatiche formali, all'interno del libro *"On Certain Formal Properties of Grammars, Information and Control"* afferma che:

> *"A grammar can be regarded as a device that enumerates the sentences of a language"*

> *"A grammar of L can be regarded as a function whose range is exactly L"*

6.1 Definizione formale di grammatica

Una **grammatica** (detta anche *grammatica non ristretta*) G è una tupla di 4 elementi $\langle V_T, V_N, P, S\rangle$:

- V_T è un insieme finito di *simboli terminali*, detto **alfabeto terminale**
  - $\rightarrow$ gli elementi di V_T sono normalmente scritti in **minuscolo**
- V_N è un insieme finito di *simboli non terminali*, detto **alfabeto non terminale**
  - $\rightarrow$ V_N è costruito in modo che $V_T \cap V_N = \emptyset$
  - $\rightarrow$ l'alfabeto V è quindi dato da $V_T \cup V_N = V_T \oplus V_N$
  - $\rightarrow$ gli elementi di V_N sono normalmente scritti in **maiuscolo**
- P è un insieme finito, detto **insieme delle produzioni** di G
  - $P \subseteq V^* \cdot V_N^+ \cdot V^* \times V^*$
  - un elemento $p = \langle \alpha, \beta \rangle$ di P verrà indicato con $\alpha \to \beta$
  - la stringa α è detta **parte sinistra** di p
  - la stringa β è detta **parte destra** di p
  - due regole $s \to d_1, s \to d_2$ con la stessa parte sinistra si possono essere accorpate come $s \to d_1 \mid d_2$
- S è un elemento *"particolare"* di V_N, detto **assioma** o **simbolo iniziale**
  - $\rightarrow$ S non può essere mai parte destra di una derivazione

6.1.1 Relazione di derivazione immediata

Data una grammatica G, si definisce su V^* la relazione binaria di **derivazione immediata**, indicata dalla notazione

$$\alpha \underset{G}{\Longrightarrow} \beta$$

è definita se e solo se

$$\alpha = \alpha_1\gamma\alpha_2, \quad \beta = \alpha_1\delta\alpha_2$$
$$\alpha_1, \alpha_2, \delta \in V^*, \quad \gamma \in V_N^+, \quad \gamma \to \delta \in P$$

Il simbolo G in $\underset{G}{\Longrightarrow}$ viene normalmente omesso qualora il contesto sia univocamente definito, indicando quindi le derivazioni come $\Rightarrow$.

Come già visto con le operazioni tra linguaggi, (Sezione 1.3.1), $\overset{*}{\underset{G}{\Longrightarrow}}, \overset{+}{\underset{G}{\Longrightarrow}}, \overset{n}{\underset{G}{\Longrightarrow}}$ indicano rispettivamente la chiusura riflessiva e transitiva, la chiusura transitiva e la potenza n di $\underset{G}{\Longrightarrow}$.

6.1.2 Linguaggio di accettazione di una grammatica

Una grammatica $G = \langle V_N, V_T, P, S\rangle$ genera un linguaggio $L(G)$ sull'alfabeto V_T. Esso è definito come:

$$L(G) = \left\{ x \mid S \overset{*}{\underset{G}{\Longrightarrow}} X, x \in V_T^* \right\}$$

Informalmente, il linguaggio generato da una grammatica è quindi costruito da tutte le *e sole* stringhe di **soli simboli terminali** che possono essere generate a partire dall'**assioma** S applicando un numero qualsiasi di sostituzioni (*passi*). Notare che gli elementi non terminali di G non fanno parte della stringa da essa generata. Le regole non sono univoche: più sostituzioni diverse possono essere applicate alla stessa stringa di partenza. Il processo di derivazione è quindi intrinsecamente non deterministico. Inoltre, alcuni cammini di derivazione possono portare a sequenze di terminali e non terminali, generando quindi stringhe non valide.

6.2 Gerarchia di Chomsky

La definizione delle grammatiche (come vista in questi appunti) è opera principalmente di *Noam Chomsky*, linguista e filosofo statunitense. Egli ha infatti introdotto una loro classificazione, nota come **gerarchia di Chomsky** (il cui diagramma è osservabile in Figura 26). In Tabella 1 sono mostrate alcune caratteristiche di ciascuna grammatica.

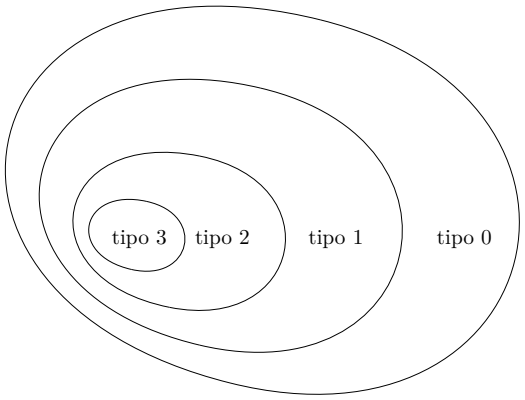


Figura 26: Gerarchia di Chomsky

tipo	*grammatica e linguaggio accettato*	*automatismo*	*forma regole*
0	non ristretta - *GG*	*TM*	*tutte*
1	dipendente dal contesto	*linear bounded automaton*	$\alpha A\beta \to \alpha\gamma\beta$
2	non contestuale - *CFG*	*NPDA*	$A \to \gamma$
3	regolare - *RG*	*FSA*	$X \to a$ o $X \to aY$

Tabella 1: Gerarchia di Chomsky

La gerarchia è composta dalle grammatiche così come segue:

Tipo 0: Include tutte le grammatiche formali

Tipo 1: Hanno regole in forma $\alpha A\beta \to \alpha\gamma\beta$

- A è un non terminale
- α, β, γ sono stringhe di terminali e non terminali
- γ è non vuota
- il contesto viene preservato *(solo il non terminale viene sostituito)*
- la regola $S \to \epsilon$ è consentita se S non appare a destra in alcuna regola

Tipo 2: Hanno regole in forma $A \to \gamma$

- A è un non terminale
- γ è una stringa di terminali e non terminali

Tipo 3: Hanno regole in forma $X \to a$ e ($X \to aY$ o $X \to Ya$)

- X, Y sono non terminali
- a è un terminale
- a può essere seguito (o preceduto) da un terminale *(le due possibilità sono mutualmente esclusive)*
- La regola $S \to \epsilon$ è consentita se S non appare a destra in alcuna regola

Le grammatiche di tipo 3 sono le **meno potenti**, mentre quelle di tipo 0 sono le **più potenti**.

6.2.1 Grammatiche lineari

Sia $G = \langle V_T, V_N, P, S\rangle$ una grammatica. Si supponga che, per ogni produzione $\alpha \to \beta \in P$:

- $|\alpha| = 1$, ossia $\alpha \in V_N$
- β sia nella forma aB, a può essere ϵ
  - $B \in V_N$, alfabeto dei non terminali
  - $a \in V_T$, alfabeto dei terminali

Allora G è una **grammatica lineare**, oppure *regolare* o di *tipo* 3 (si veda la Sezione 6.2). Le grammatiche di questa categoria hanno al massimo un non terminale nella parte destra di ognuna delle sue derivazioni.

Si riconoscono due tipi particolari di grammatiche lineari:

$\leftarrow$ Lineare **sinistra** (L-grammatica)

- Tutte le derivazioni sono nella forma $\alpha \to \alpha w$
- $|\alpha| = 1$ è vuoto o singolo non terminale
- w è una stringa di terminali

$\rightarrow$ Lineare **destra** (R-grammatica)

- Tutte le derivazioni sono nella forma $\alpha \to w\alpha$

- w è una stringa di terminali
- $|\alpha| = 1$ è vuoto o singolo non terminale

Una **grammatica** è **regolare** (in breve *RG*) se e solo se è regolare sinistra o regolare destra. Inoltre, un **linguaggio** è **regolare** se e solo se è generato da una grammatica regolare (e quindi esiste almeno una grammatica che lo genera).

6.2.2 Grammatiche non contestuali

Sia $G = \langle V_T, V_N, P, S \rangle$ una grammatica. Se, per ogni produzione $\alpha \to \beta$ si verifica:

- $\alpha \to \beta \in P$, la produzione è contenuta in P
- $|\alpha| = 1$, α è un non terminale

allora G è una **grammatica non contestuale** o *CFG* (dall'Inglese *context-free grammar*). Questa denominazione è dovuta al fatto che la riscrittura di α non dipende dal suo contesto *(la parte della stringa che la circonda)*.
Le *CFG* sono anche dette *BNF* (da *Backus-Naur Form*) e vengono impiegate per definire la sintassi di linguaggi di programmazione. Le *RG* sono anche *CFG* ma non è vero il contrario:

$$RG \subseteq CFG$$

6.2.3 Grammatiche generali

Le **grammatiche generali** (per brevità *GG* o *non ristrette*) sono tutte le grammatiche che non presentano limitazioni sulle produzioni. Corrispondono al tipo 0 della gerarchia di Chomsky (Sezione 6.2).
Sia le *RG* che le *CFG* sono **non ristrette**:

$$RG \subseteq CFG \subseteq GG$$

In questa categoria cadono le grammatiche **ricorsivamente enumerabili** e le grammatiche **ricorsive**. Per più dettagli sul significato di queste locuzioni, si consultino le Sezioni 9.11 e 9.10.

6.3 *RG* e *FSA*

Dato un *FSA* A, è possibile costruire una R-grammatica a esso equivalente, ossia in grado di generare lo stesso linguaggio riconosciuto da A e viceversa (Sezioni 6.3.1 e 6.3.2).
La conseguenza diretta è che *RG*, *FSA* ed espressioni regolari (Sezione 7) sono modelli diversi per descrivere la stessa classe di linguaggi.

6.3.1 Costruzione di *RG* partendo da *FSA*

Sia $A = \langle Q, I, \delta, q_0, F \rangle$ un *FSA*. È possibile costruire una *RG* $G = \langle V_N, V_T, S, P \rangle$ tale che:

- $V_T = I$, l'insieme dei **terminali** di G corrisponde all'**alfabeto di ingresso** di A
- $V_N = Q$, l'insieme dei **non terminali** di G corrisponde agli stati di A
- $S = q_0$, l'**assioma** di G corrisponde allo **stato iniziale** di A
- Gli elementi di P **(l'insieme delle produzioni di G)** hanno la seguente forma:
  - $B \to bC$ se e solo se $C \in \delta(B, b)$ - *transizione tra stati*
  - $B \to \epsilon$ se $B \in F$ - *transizione verso lo stato finale*
  - $\delta^*(q, x) = q'$ se e solo se $q \overset{*}{\Longrightarrow} xq'$

6.3.2 Costruzione di *FSA* partendo da *RG*

Sia $G = \langle V_N, V_T, S, P \rangle$ una *RG*. È possibile costruire un *FSA* $A = \langle Q, I, \delta, q_0, F \rangle$ tale che:

- $Q = V_N \cup \{q_F\}$, gli **stati** di A sono i **non terminali** di G
- $I = V_T$, l'**alfabeto di ingresso** di A corrisponde all'**insieme di terminali** di G
- $q_0 = S$, lo **stato iniziale** di A corrisponde all'**assioma** di G
- $F = \{q_F\}$
- La **funzione di transizione** δ ha la seguente forma:
  - $\delta(A, b) = C$ se e solo se $A \to bC$ - derivazione di terminali e non terminali
  - $\delta(A, b) = q_F$ se e solo se $A \to b$ - derivazioni di terminali

6.4 *CFG* e *NPDA*

Analogamente a quanto trattato nella Sezione 6.3.1, è possibile costruire un *NPDA* partendo da una *CFG (context-free grammar, grammatica non contestuale)* e viceversa.
Intuitivamente, la pila contiene la parte intermedia delle produzioni, fatta di terminale e non terminali. Rispettando le regole definite della *CFG*, è possibile completare le produzioni fino ad ottenere una stringa di non terminali tramite mosse **non deterministiche** del *NPDA*.
Un esempio del funzionamento del *NPDA* che emula una *CFG* con la regola $S \Rightarrow aSb \Rightarrow aabb$ è mostrato in Figura 27.

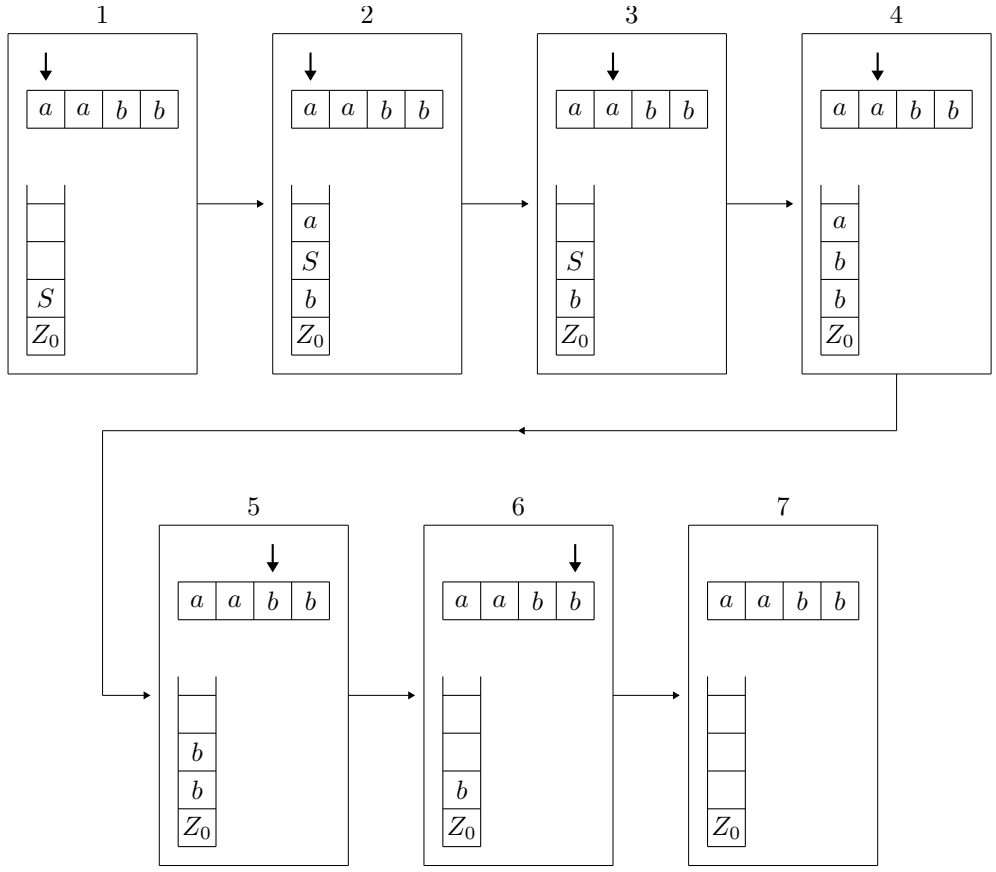


Figura 27: Analogia *CFG* e *NPDA*

6.5 *GG* e *TM*

Analogamente a quanto trattato nella Sezione 6.3.1, è possibile costruire una *TM* partendo da una *GG (grammatica generale)* e viceversa.

6.5.1 Costruzione di *GG* partendo da *TM*

Sia M una *TM* a nastro singolo. È possibile costruire una *GG* detta G tale che $L(G) = L(M)$ *(il linguaggio generato dalla GG e accettato dalla NTM sono equivalenti)*:

- Inizialmente, G genera tutte le stringhe di tipo $x\$X$ con
  - $x \in V_T^*$ e X
  - X è la copia di x composta solo dai simboli **non terminali**
- Successivamente, G simula le diverse configurazioni di M sfruttando la stringa a destra del simbolo \$
  - la derivazione $x\$X \overset{*}{\Rightarrow} x$ se e solo se x è accettata da M
  - ogni mossa di M viene **emulata** con una derivazione immediata di G
  - G ha quindi delle derivazioni della forma $x\$X \Rightarrow x\$q_0X$ *(configurazione iniziale di M)*
- Infine, per simulare le mosse di M vengono introdotte le seguenti produzioni:
  1. Se è definita $\delta(q, A) = \langle q', A', R\rangle$ si definisce in G la produzione $qA \to A'q'$
  2. Se è definita $\delta(q, A) = \langle q', A', S\rangle$ si definisce in G la produzione $qA \to q'A'$
  3. Se è definita $\delta(q, A) = \langle q', A', L\rangle$ si definisce in G la produzione $BqA \to q'BA'$, $\forall B$ dell'alfabeto di M
     $\to$ gli alfabeti di ingresso, uscita e memoria coincidono
- Per completare la costruzione è infine necessario aggiungere a M le produzioni che permettono a G di derivare da $x\$\alpha BqAC\beta$ la sola x solo nei cai in cui M giunge a una configurazione di accettazione ($x\$\alpha Bq_FAC\beta$) cancellando tutto ciò che si trova a destra di \$, esso compreso

6.5.2 Costruzione di *TM* partendo da *GG*

Sia $G = \langle V_N, V_T, S, P\rangle$ una *GG*. È possibile costruire una *NTM* M tale che $L(M) = L(G)$ *(il linguaggio accettato dalla NTM e generato dalla GG sono equivalenti)*:

- M ha un nastro di **memoria**
- La stringa di ingresso x è sul nastro di **ingresso**
- Il nastro di memoria è inizializzato con l'**assioma** Z_0S
- Il nastro di memoria in generale conterrà una **stringa** $\alpha \in V^*$:
  - Viene scandito per cercare la parte **sinistra** di una produzione P
  - Una volta trovata, M compie una scelta **non deterministica** e la parte scelta è sostituita dalla parte destra corrispondente
  - Se ad essa corrispondono **più parti destre**, ne viene scelta una in modo **non deterministico**

In questo modo vi è in G una derivazione che porta dalla stringa α alla stringa β (quindi una derivazione $\alpha \overset{*}{\Rightarrow} \beta$) se e solo se esiste una sequenza di mosse tale per cui

$$c_s = \langle q_s, Z_0\alpha\rangle \vdash^* \langle q_s, Z_0\beta\rangle$$

per qualche q_s (come illustrato in Figura 28).

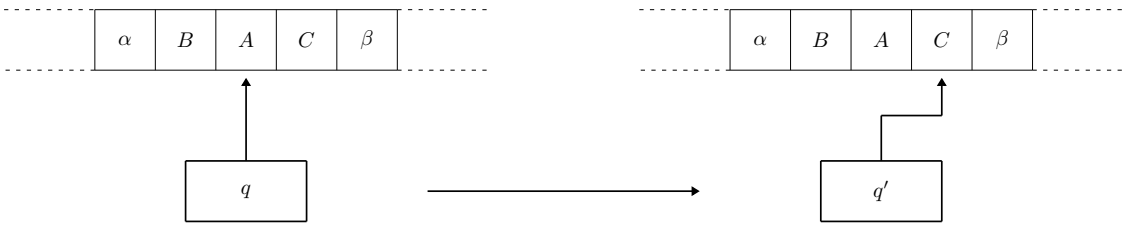


Figura 28: Derivazione $\alpha \overset{*}{\Rightarrow} \beta$ in una *TM* costruita da *GG*

Se il nastro contiene una stringa $y \in V_T^*$ (composta da soli simboli terminali), essa è confrontata con x:

✔ Se coincidono, allora x è **accettata**
✖ Altrimenti, questa particolare sequenza di mosse **non porta all'accettazione**

6.5.2.1 Note sulla costruzione di *NTM*

- Usare una NTM facilita la costruzione di una GG ma non è l'unico metodo ammesso per farlo.
  - Poiché il non determinismo può essere emulato da macchine deterministiche, una TM è sufficiente a tale scopo
- Se $x \notin L(G)$, allora M può tentare infinite computazioni, nessuna delle quali porta ad accettazione
  - alcune di queste potrebbero non terminare mai, quindi senza concludere che $x \in L(G)$
  - analogamente, non si concluderebbe se $x \notin L(G)$
- La definizione di accettazione richiede che:
  - M raggiunga una configurazione di accettazione se e solo se $x \in L$
  - essa non richiede che M termini la computazione in uno stato non finale se $x \notin L$
  - il problema del complemento è risolto e si ripresenta l'asimmetria tra risoluzione di un problema in senso positivo o negativo

7 Espressioni regolari - *RE*

Come visto fino ad ora, i linguaggi possono essere rappresentati tramite diverse **classi di modelli**, tra cui si riconoscono:

- *Insiemi*
- *Pattern*
- *Espressioni regolari*
- *Modelli operazionali*
  - Automi *(Sezione 2)*
  - Trasduttori *(Sezione 2.3)*
  - Reti di Petri
  - Diagrammi di stato
- *Modelli generativi*
  - Grammatiche *(Sezione 6)*
- *Modelli dichiarativi*
  - Logica *(Sezione 8)*

In questo capitolo si tratterà delle **espressioni regolari** (scritto come *RE* o *Regex*). Esse sono delle espressioni utilizzabili per denotare un linguaggio attraverso la scrittura delle stringhe che lo compongono.

7.1 Pattern

Prima di dare una definizione formale di *RE*, è necessario introdurre il concetto di **pattern**.

Un sistema di pattern è una tripla $\langle A, V, p \rangle$ dove:

- A è un **alfabeto**
- V è un **insieme di di variabili**

  $\rightarrow$ è definito in modo che $A \cup V = \emptyset$
- p è una stringa su $A \cup V$ detta **pattern**

Il linguaggio generato dal sistema di pattern consiste di tutte le stringhe su A ottenute da p sostituendo ogni variabile in p con una stringa su A.

Esempio: $\langle \underbrace{\{0,1\}}_{alfabeto}, \underbrace{\{v_1, v_2\}}_{variabili}, \underbrace{v_1 v_1 0 v_2}_{pattern} \rangle$

- Stringhe che iniziano con 0 ($v_1 = \epsilon$)
- Stringhe che iniziano con una stringa su A ripetuta due volte, seguita da uno 0 e da qualunque stringa (inclusa ϵ)

7.2 Sintassi e semantica delle *RE*

Dato un alfabeto di simboli terminali, mediante le seguenti regole si definiscono le **espressioni regolari** e i corrispondenti linguaggi denotati. Sono *RE* su un alfabeto Σ:

1. $\emptyset$ è una *RE* che denota il linguaggio vuoto ($\emptyset$)
2. ϵ è una *RE* che nota il linguaggio $\{\epsilon\}$
3. Ogni simbolo di σ è una *RE* che denota il linguaggio $\{\sigma\},\ \sigma \in \Sigma$
4. Se R_1 e R_2 sono *RE*, anche $R_1 \cup R_2$, *(scritto anche come* $R_1 + R_2$ *o* $R_1 \mid R_2$*)* è una *RE*
5. Se R_1 e R_2 sono *RE*, anche $R_1 \cdot R_2$, *(scritto anche come* $R_1 R_2$*)*, è una *RE*
6. Se R è una *RE*, lo è anche R^*
7. Nient'altro è una *RE*

Le *RE* seguono la stessa idea dei sistemi di pattern, ma con diverso potere espressivo. Le espressioni regolari sono diverse dai sistemi di pattern e hanno diverso potere espressivo.
Le *RE* **corrispondono esattamente** ai linguaggi regolari e hanno lo stesso potere espressivo di *RG* e *FSA*: per ogni *FSA* è possibile costruire la *RE* equivalente.

Per dimostrare l'enunciato è sufficiente osservare che:

- Ogni linguaggio denotato da una *RE* è regolare. Infatti:
  1. i casi base sono linguaggi regolari
  2. i linguaggi regolari sono chiusi rispetto a agli operatori di **concatenazione**, **unione** e **stella di Kleene**
- Data una *RG* G, è sempre possibile trovare una *RE* r tale che $L(G) = L(r)$

7.2.1 Operatori delle *RE*

Sono definiti i seguenti **operatori** delle *RE*:

- La **concatenazione**, $\cdot$
- L'**alternativa** (detta anche *pipe*), $|$
- La **stella di Kleene** (già definita nella Sezione 1.2), $*$

  $\rightarrow$ di conseguenza, anche il **più di Kleene**, $+$
- L'**opzionalità**, ?

7.2.2 *RE* e pattern

Le espressioni regolari seguono la stessa idea dei sistemi di pattern, ma con diverso potere espressivo. Infatti, per riconoscere i linguaggi generati dai pattern servirà una *TM*, mentre per le *RE* sono sufficienti le *FSA*.
Di conseguenza, le *RE* non definiscono la stessa classe di linguaggi definita dai pattern. Inoltre le due classi non sono confrontabili, e non sono una sottoclasse dell'altra.
Per la prima volta dall'inizio del corso si osserva un caso di non confrontabilità:

$$RE \neq \text{ pattern}$$

7.2.3 *RE* `POSIX`

Lo standard `POSIX` è una API standard per i sistemi operativi UNIX/LINUX e definisce anche le *RE*. Esso include:

- I **metacaratteri** ( ) [ ] ^ \ $ $*$ $+$? { }
- La notazione $[\alpha]$ indica **un singolo carattere** $\in \alpha$
- La notazione $[\hat{}\alpha]$ indica **un qualunque simbolo** $\notin \alpha$
- Il simbolo ^ indica ϵ **all'inizio di una riga** di testo
- Il simbolo $ indica ϵ **alla fine di una riga** di testo
- I simboli $*, +, |, (,)$ rappresentano gli **operatori** come già definiti *(a esempio in Sezione 1.3.1)*
- Il simbolo \ funge da **escape**

In aggiunta agli operatori delle *RE*, sono definiti:

- α? indica che α è **opzionale**

  $\rightarrow$ α appare 0 o 1 volte
- $\alpha\{n\}$ indica la **potenza** α^n

  $\rightarrow$ α appare n volte
- $\alpha\{n, m\}$ indica $\alpha^n \cup \alpha^{n+1} \cup \ldots \cup a^{m-1} \cup a^m$

  $\rightarrow$ α appare tra le m e le n volte

8 Logica nell'informatica

In questa sezione verrà analizzata la logica dal punto di vista del suo uso nell'ingegneria informatica. La logica è un *formalismo descrittivo e universale*, cioè permette di descrivere le proprietà che si vogliono ottenere (o evitare) da un sistema, senza dover per forza formalizzare anche quest'ultimo.
Grazie a questa sua proprietà, la logica può essere applicata in numerosi contesti, come:

- Le porte logiche nell'architettura dei calcolatori
- La specifica e la verifica di sistemi nell'ingegneria del software
- La definizione della semantica dei linguaggi di programmazione
- La programmazione logica
- I database

e molti altri.
All'interno del corso verranno usati la *PL* e la *FOL (introdotte nelle Sezioni 8.1 e 8.2)* per:

- Definire i linguaggi *(Sezione 8.3)*
- Specificare le proprietà di programmi *(Sezione 8.6)*
- Specificare le proprietà dei sistemi *(Sezione 8.6.1)*

8.1 Logica proposizionale - *PL*

La logica proposizionale è un linguaggio formale dalla sintassi semplice, basata su proposizioni elementari e su connettivi logici di tipo funzionale. Opera tra proposizioni *(che possono assumere il valore* `vero` *o* `falso`*)* e relazioni tra proposizioni. Le proposizioni composte sono formate concatenando proposizioni semplici tramite connettivi logici.
Contrariamente a logiche più complicate *(come la Logica del primo ordine - FOL, analizzata nella Sezione 8.2)*, non permette di operare e predicare tra oggetti non logici e quantificatori.

8.1.1 Sintassi

Sia $\mathcal{L}$ un linguaggio della **logica proposizionale**.

- L'**alfabeto**, di $\mathcal{L}$ è composto da:
  - Un insieme **numerabile** *(finito o infinito)* di **proposizioni**: $A, B, C, \ldots$
    - le proposizioni sono simboli di relazione **nullaria**
    - I simboli dell'alfabeto sono privi di significato
  - Un insieme di **simboli** $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
  - I **simboli di punteggiatura** $(,)$ - *parentesi tonde*
- L'**insieme di formule** di $\mathcal{L}$ è il più piccolo insieme tale che:
  - Ogni *proposizione* è una **formula**
  - Se F e G sono formule, allora $\neg F$, $F \wedge G$, $F \vee G$, $F \rightarrow G$, $F \leftrightarrow G$ sono **formule**
- Le **parentesi sono omesse** ovunque possibile usando l'ordine di precedenza:

$$\neg \quad \wedge \quad \vee \quad \rightarrow \quad \leftrightarrow$$

- Se A è una proposizione, allora A e $\neg A$ sono detti **letterali**
  - A è detto letterale **positivo**
  - $\neg A$ è detto letterale **negativo**
- se L è un letterale, allora $\overline{L}$ è il **letterale complementare** definito come $\neg A$ *se* $L = A$ o A *se* $L = \neg A$
- *Informalmente*, una **sotto formula** è una formula inclusa in un'altra formula
- L'insieme $\tau(F)$ delle sotto formule di $\mathcal{F}$ è definito come il più piccolo insieme di formule tale che:
  - $F \in \tau(F)$
  - se $\neg G \in \tau(F)$, allora $G \in \tau(f)$
  - se $G \wedge H, G \vee H, G \rightarrow H, G \leftrightarrow H$ appartengono a $\tau(F)$, allora $H, G \in \tau(F)$

8.1.2 Semantica

La **semantica** è introdotta per assegnare un significato alle formule.
Nella logica proposizionale, ogni formula può corrispondere a un solo valore di verità (`vero` o `falso`): è quindi una logica **a due valori**.
Un'**interpretazione** I è una funzione totale dell'insieme di proposizioni ai valori di verità. Ogni interpretazione può essere convenientemente rappresentata come l'insieme delle proposizioni vere.
Con la notazione $I \vDash F$ si indica che I **rende vera** F.

Definizioni:

- Se $I \vDash F$, allora I è un **modello** di F. Questa nozione può essere estesa agli insiemi di formule
- F è **valida** (detta anche **tautologia**) se e solo se per ogni interpretazione I vale che $I \vDash F$

  $\rightarrow$ in questo caso si può anche scrivere $\vDash F$
- F è **soddisfacibile** se e solo se esiste un'interpretazione I tale per cui $I \vDash F$
- F è **falsificabile** se e solo se esiste un'interpretazione I tale per cui $I \nvDash F$
- F è **insoddisfacibile** se e solo se per ogni interpretazione I vale $I \nvDash F$
- F è **contingente** se e solo se è sia *soddisfacibile* sia *falsificabile*
- Ogni formula del tipo $F \wedge \neg F$ è detta una **contraddizione**, indicata con $\bot$
- La formula $F \vee \neg F$ è detta **principio del terzo escluso**, indicata con $\top$
- Un insieme di formule $\mathcal{F}$ **comporta logicamente** una formula G (o G è una *conseguenza logica* di $\mathcal{F}$) se ogni modello di $\mathcal{F}$ è anche un modello di G e si scrive con $\mathcal{F} \vDash G$. *Esempio:*
  - $\{A, A \rightarrow B\} \vDash B, \{A, A \rightarrow B\} \vDash B \wedge C, \{A, A \rightarrow B\} \nvDash C$
- Per determinare sistematicamente se una formula segua da un insieme di formule si possono usare le **tabelle di verità**

Si considerino:

- La *proposizione* A
- La *formula* F
- L'*interpretazione* I

Allora:

$$I \models A \quad sse \quad I(A) = \texttt{vero}$$

Di conseguenza:

$$\begin{aligned} I \vDash \neg F \quad & \textbf{sse} \quad I \nvDash F \\ I \vDash F \wedge G \quad & \textbf{sse} \quad I \vDash F \textbf{ e } I \vDash G \\ I \vDash F \vee G \quad & \textbf{sse} \quad I \vDash F \textbf{ o } I \vDash G \\ I \vDash F \rightarrow G \quad & \textbf{sse} \quad I \nvDash F \textbf{ o } I \vDash G \\ I \vDash F \leftrightarrow G \quad & \textbf{sse} \quad I \vDash F \rightarrow G \textbf{ e } I \vDash G \rightarrow F \end{aligned}$$

Alternativamente, i connettivi possono anche essere esplicitati sotto forma di tabella (Tabella 2).

8.1.2.1 Esempio di interpretazione

Se:

- $I_1 = \{A, C\}$
- $I_2 = \{C, D\}$
- $F = (A \vee B) \wedge (C \vee D)$

Allora:

- $I_1 \vDash F$
- $I_2 \nvDash F$

F	G		$\neg F$	$F \wedge G$	$F \vee G$	$F \to G$	$F \leftrightarrow G$
vero	vero		falso	vero	vero	vero	vero
vero	falso		falso	falso	vero	falso	falso
falso	vero		vero	falso	vero	vero	falso
falso	falso		vero	falso	falso	vero	vero

Tabella 2: Tabella di verità

8.1.3 Forme normali

- Due formule F, G sono **semanticamente equivalenti** se e solo se vale sia $F \vDash G$ che $G \vDash F$
  → in questo caso si usa la notazione $F \equiv G$
- La formula G, sotto formula di F può essere sostituita da una formula H
  → per indicare la formula risultante si usa la notazione $F[G \setminus H]$
- Un insieme di connettivi è detto **funzionalmente completo** se e solo se qualunque formula proposizionale può essere trasformata in una formula semanticamente equivalente che contiene solo connettivi dell'insieme
  → L'insieme $\{\neg, \wedge\}$ è funzionalmente complesso
  → Esistono altri due connettivi booleani che singolarmente sono funzionalmente completi: `nand` e `nor`

Equivalenze notevoli:

$$
\begin{aligned}
(F \wedge F) &\equiv F && \text{idempotenza di } \wedge \\
(F \vee F) &\equiv F && \text{idempotenza di } \vee \\
(F \wedge G) &\equiv (G \wedge F) && \text{commutatività di } \wedge \\
(F \vee G) &\equiv (G \vee F) && \text{commutatività di } \vee \\
(F \wedge (G \wedge H)) &\equiv ((F \wedge (G \wedge H) && \text{associatività di } \wedge \\
(F \vee (G \vee H)) &\equiv ((F \vee (G \vee H) && \text{associatività di } \vee \\
((F \wedge G) \vee F) &\equiv F && \text{assorbimento} \\
((F \vee G) \wedge F) &\equiv F && \text{assorbimento} \\
(F \wedge (G \vee H)) &\equiv ((F \wedge G) \vee (F \wedge H)) && \text{distributività} \\
(F \vee (G \wedge H)) &\equiv ((F \vee G) \wedge (F \vee H)) && \text{distributività} \\
(\neg(\neg F)) &\equiv F && \text{doppia negazione} \\
(\neg(F \wedge G)) &\equiv (\neg F \vee \neg G) && \text{legge di De Morgan} \\
(\neg(F \vee G)) &\equiv (\neg F \wedge \neg G) && \text{legge di De Morgan} \\
(F \leftrightarrow G) &\equiv (F \to G) \wedge (G \to F) && \text{equivalenza} \\
(F \to G) &\equiv (\neg F \vee G) && \text{implicazione materiale} \\
(F \to G) &\equiv (\neg G \to \neg F) && \text{implicazione contronominale}
\end{aligned}
$$

Il *teorema di sostituzione* e le *equivalenze notevoli* possono essere utilizzate per introdurre le cosiddette **forme normali**:

- Una formula è in **forma normale negativa** se e solo se è composta solo da letterali, congiunzioni ($\wedge$) e disgiunzioni ($\vee$)
- Una formula è in **forma normale congiuntiva** (*CNF*) se e solo se ha la forma $C_1 \wedge C_2 \wedge \ldots \wedge C_n$ dove ogni C_i è la disgiunzione di letterali

$\rightarrow$ $C_1 \wedge C_2 \wedge \ldots \wedge C_n \equiv C_1 \wedge C_2 \wedge \ldots \wedge C_n \wedge \top$, quindi si può affermare che $\top$ è in *CNF* con $n = 0$

- Una formula è in **forma normale disgiuntiva** (*DNF*) se e solo se ha la forma $C_1 \vee C_2 \vee \ldots \vee C_n$ dove ogni C_i è la congiunzione di letterali

  $\rightarrow$ $D_1 \vee D_2 \vee \wedge \vee D_n \equiv D_1 \vee D_2 \vee \ldots \vee D_n \vee \bot$, quindi si può affermare che $\bot$ è in *DNF* con $n = 0$

- I C_i sono detti **clausole**, mentre i D_i sono detti **clausole duali**. Normalmente si usa una notazione insiemistica

8.1.4 Sistemi formali - *calculi*

Un *sistema formale (assiomatico deduttivo)*, oppure **calculus** in Inglese, consiste in un insieme di assiomi e un insieme di regole di inferenza che producono conseguenze logiche all'interno di una logica. Questi elementi definiscono una *relazione di derivabilità* (detta anche **dimostrabilità**) tra un insieme di formule $\mathcal{F}$ e una formula G.
Se una formula G può essere ottenuto da $\mathcal{F}$ applicando solo regole di inferenza e assiomi, si scrive che $\mathcal{F} \vdash G$. Idealmente, la relazione di derivabilità dovrebbe essere **corretta** (cioè se $\mathcal{F} \vdash G$ allora $\mathcal{F} \vDash G$) e **completa** (cioè se $\mathcal{F} \vDash G$ allora $\mathcal{F} \vdash G$).
Se una formula F piò essere derivata in una teoria $\mathcal{F}$ usando gli assiomi e le regole d'inferenza di un sistema, allora diciamo che F è un **teorema**.

8.2 Logica del primo ordine - *FOL*

La logica proposizionale (vista in Sezione 8.1) ha molte applicazioni, ma il suo potere espressivo è ristretto. Infatti, frasi come *"tutti gli esseri umani sono mortali"* e *"ogni bambino ha dei genitori"* possono essere espresse come proposizioni in un modo che non cattura le relazioni sottintese tra esseri umani, mortali, bambini e genitori.
Gottlob Frege, logico tedesco, ha sviluppato la *FOL* (**logica del prim'ordine** o **logica dei predicati**) nel 1879, estendendo la logica proposizionale con *funzioni*, *variabili* e *quantificatori*.

- Dal punto di vista *epistemologico* (stati della conoscenza) sia *PL* che *FOL* considerano **verità** e **falsità**
  - quindi vengono considerati i valori di `vero` e `falso` assunti dalla logica
- Dal punto di vista *ontologico* (ciò che esiste), *PL* considera i *fatti*, mentre *FOL* considera anche:
  - **oggetti**, quali: *persone, case, numeri, corsi, ...*
  - **proprietà** e **relazioni**, quali: *essere rosso, essere felice, essere più grande di, essere parte di, ...*
  - **funzioni**, quali: *padre di, età di, successore di, ...*

8.2.1 Sintassi

- L'**alfabeto** di un linguaggio della *FOL* $\mathcal{L}$ è composto da:
  - Un insieme infinito numerabile di **variabili**: $X, Y, Z, \ldots$
  - Un insieme di simboli di **funzione**: $f, g, \ldots$
  - Un insieme di simboli di **predicati**: $p, q, r, \ldots$
  - I seguenti **connettivi**: $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$
  - I seguenti **quantificatori**: $\exists, \forall$
  - I **simboli di punteggiatura**: $(,), ", "$ - *parentesi tonde e virgola*
- Ogni simbolo di funzione e relazione ha una **arietà** fissata che indica il numero di argomenti a esso associati:
  - la notazione p/n indica che il *predicato* (o *funzione*) p ha **arietà** n
  - un *predicato* (o *funzione*) con arietà zero è detto **nullario**
  - un *predicato nullario* $p()$ è scritto semplicemente come p
  - una *funzione nullaria* $c()$ è semplicemente scritta come c
- Le funzioni nullarie sono dette **costanti**, i predicati nullari sono detti **proposizioni**
- I **termini** denotano tutti gli oggetti che $\mathcal{L}$ può trattare
  - sono definiti induttivamente come segue:

1. Ogni *variabile* è un *termine*
2. Se f/n è un simbolo di funzione e $t_1, \ldots t_n$ sono termini, allora $f(t_1, \ldots t_n)$ è un *termine*

- i **termini generici** sono tipicamente indicati con $s, t, \ldots$

- L'insieme di **formule della *FOL*** è definito induttivamente come il più piccolo insieme tale che:
  - Se p/n è un simbolo di relazione e $t_1, \ldots t_n$ sono termini, allora $p(t_1, \ldots, t_n)$ è una formula detta **formula atomica** o **atomo**
  - se F, G sono formule e X è una variabile, allora sono formule anche:

$$\neg F \quad F \wedge G \quad F \vee G \quad F \to G \quad F \leftrightarrow G \quad \exists X F \quad \forall X F$$

- I **letterali** sono atomi *(letterali positivi)* o atomi negati *(letterali negativi)*
- Le **parentesi sono omesse ovunque possibile** usando l'ordine di precedenza tra gli operatori:

$$\exists \quad \forall \quad \neg \quad \wedge \quad \vee \quad \to \quad \leftrightarrow$$

- Per convenienza:

$$\exists X_1 (\ldots (\exists X_n(F)) \ldots) \quad \text{è abbreviato come} \quad \exists X_1, \ldots, X_n(F)$$
$$\forall X_1 (\ldots (\forall X_n(F)) \ldots) \quad \text{è abbreviato come} \quad \forall X_1, \ldots, X_n(F)$$

- Se $QX(F)$ è una formula e Q è un quantificatore, allora F si dice **ambito** di Q e Q è **applicato** a F
- Un'occorrenza di una variabile in una formula è **legata** se e solo se la sua occorrenza è entro l'ambito di un quantificatore che impiega quella variabile
  → è legata al quantificatore di ambito più piccolo che la rende legata
  → in caso contrario, è **libera**
- Una formula è **chiusa** se e solo se non contiene occorrenze libere di variabili
  - la valutazione è omessa quando si considerano formule chiuse
- Un'interpretazione $\mathcal{I}$ è detta **modello** per F sse per ogni valutazione Φ si ha $\mathcal{I} \vDash_\Phi F$
  - se $\mathcal{F}$ è un insieme di formule, un'interpretazione è un **modello** di $\mathcal{F}$ sse è un modello $\forall F \in \mathcal{F}$

8.2.1.1 Osservazioni sulle traduzioni in *FOL*

- Il connettivo principale usato con $\forall$ è $\to$
  - *"al Polimi sono tutti brillanti"* `(bleah)` si può tradurre con

$$\forall X \, (at(X, polimi) \to smart(X))$$

  - notare che la formula

$$\forall X \, (at(X, polimi) \wedge smart(X))$$

    significa che **tutti** sono al Polimi e **tutti** sono brillanti

- Analogamente, il connettivo principale da usare con $\exists$ è $\wedge$
  - *"al Polimi qualcuno è brillante"* si può tradurre con

$$\exists X \, (at(X, polimi) \wedge smart(X))$$

  - notare che la formula

$$\forall X \, (at(X, polimi) \to smart(X))$$

    significa che c'è qualcuno che o non è al Polimi o è brillante *(o entrambe)*

8.2.2 Semantica

Come per *PL*, anche *FOL* ha una semantica a due valori di verità basata sulla nozione di interpretazione.
Una interpretazione $\mathcal{I}$ di un alfabeto $\mathcal{A}$ è un **dominio** non vuoto D (indicato anche come $|\mathcal{I}|$) a una funzione che associa:

- Ogni costante $c \in \mathcal{A}$ a un elemento $c_\mathcal{I} \in D$
- Ogni simbolo di funzione $f/n \in \mathcal{A}$ a una funzione $f_\mathcal{I} : D^n \to D$
- Ogni simbolo di predicato $p/n \in \mathcal{A}$ a una relazione $p_\mathcal{I} \subseteq \underbrace{D \times \ldots \times D}_{n \text{ volte}}$

Tuttavia, prima di assegnare un significato alle formule, va definito il significato di ciascun termine. Poiché essi possono contenere variabili, occorre una **valutazione** (o **stato**) ossia una funzione dalle variabili di $\mathcal{A}$ a $|\mathcal{I}|$.
Il **significato** $\Phi_\mathcal{I}(t)$ di un termine t nell'interpretazione $\mathcal{I}$ e valutazione Φ è quindi definito induttivamente come:

1. $c_\mathcal{I}$ se t è una costante c
2. $\Phi(X)$ se t è una variabile X
3. $f_\mathcal{I}\left(\Phi_\mathcal{I}(t_1), \ldots, \Phi_\mathcal{I}(T_n)\right)$ se t è della forma $f(t_1, \ldots, t_n)$

8.2.2.1 Proprietà delle valutazioni

Si considerino:

- Una *valutazione* Φ
- Una *variabile* X
- Un'*interpretazione* $\mathcal{I}$, $c_\mathcal{I} \in |\mathcal{I}|$

Allora $\Phi[X \mapsto c_\mathcal{I}]$ è una valutazione analoga a Φ che *mappa* X in $c_\mathcal{I}$.
Il significato di una formula è un valore di verità che è definito induttivamente come segue:

1. Si indica $I \vDash_\Phi F$ una formula F vera rispetto a $\mathcal{I}$ e Φ
2. Si applicano le seguenti uguaglianze:

$$
\begin{array}{ll}
\mathcal{I} \vDash_\Phi p(t_1, \ldots, t_n) & sse \quad \langle \Phi_\mathcal{I}(t_1), \ldots, \Phi_\mathcal{I}(t_n) \rangle \in P_\mathcal{I} \\
\mathcal{I} \vDash_\Phi (\neg F) & sse \quad \mathcal{I} \nvDash_\Phi F \\
\mathcal{I} \vDash_\Phi (F \wedge G) & sse \quad \mathcal{I} \vDash_\Phi F \ e \ \mathcal{I} \vDash_\Phi G \\
\mathcal{I} \vDash_\Phi (F \vee G) & sse \quad \mathcal{I} \vDash_\Phi F \ o \ \mathcal{I} \vDash_\Phi G \\
\mathcal{I} \vDash_\Phi (F \to G) & sse \quad \mathcal{I} \nvDash_\Phi F \ o \ \mathcal{I} \vDash_\Phi G \\
\mathcal{I} \vDash_\Phi (F \leftrightarrow G) & sse \quad \mathcal{I} \vDash_\Phi (F \to G) \ e \ \mathcal{I} \vDash_\Phi (G \to F)G \\
\mathcal{I} \vDash_\Phi (\forall X(F)) & sse \quad \mathcal{I} \vDash_{\Phi[X \mapsto c_\mathcal{I}]} F \ \forall c_\mathcal{I} \in |\mathcal{I}| \\
\mathcal{I} \vDash_\Phi (\exists X(F)) & sse \quad \mathcal{I} \vDash_{\Phi[X \mapsto c_\mathcal{I}]} F \ \exists c_\mathcal{I} \in |\mathcal{I}|
\end{array}
$$

8.2.2.2 Da *PL* a *FOL*

La relazione di conseguenza logica $\vDash$ tra insiemi di formule e formule può essere esteso anche a *FOL*, così come i concetti di **validità, soddisfacibilità, falsificabilità, contingenza** e **insoddisfacibilità** (visti nella Sezione 8.1.2).
Analogamente, anche le equivalenze mostrate per *PL* (Sezione 8.1.3) possono essere estese a *FOL*, con l'aggiunta di:

$$\forall X(F) \equiv \neg(\exists(\neg F)) \qquad \textit{dualità dei quantificatori } 1$$
$$\exists X(F) \equiv \neg(\forall(\neg F)) \qquad \textit{dualità dei quantificatori } 2$$
$$\forall X(F) \wedge (\forall X)G \equiv \forall X(F \wedge G)$$
$$\exists X(F) \wedge (\exists X)G \equiv \exists X(F \vee G)$$
$$(\forall X)(\forall Y)F \equiv (\forall Y)(\forall X)F$$
$$(\exists X)(\exists Y)F \equiv (\exists Y)(\exists X)F$$
$$(\forall X(F)) \wedge G \equiv \forall X(F \wedge G) \qquad \textit{solo se } X \textit{ non è libera in } G$$
$$(\forall X(F)) \vee G \equiv \forall X(F \vee G) \qquad \textit{solo se } X \textit{ non è libera in } G$$
$$(\exists X(F)) \wedge G \equiv \exists X(F \wedge G) \qquad \textit{solo se } X \textit{ non è libera in } G$$
$$(\exists X(F)) \vee G \equiv \exists X(F \vee G) \qquad \textit{solo se } X \textit{ non è libera in } G$$

8.3 Logica e linguaggi

La logica può aiutare a descrivere i linguaggi: gli insiemi di stringhe possono essere visti come abbreviazioni di formule *FOL*. Tramite formalismi matematici, infatti, è possibile mostrare i linguaggi in modo descrittivo focalizzandosi sulle proprietà rilevanti delle stringhe anziché sul modo in cui esse sono generate o riconosciute. Per raggiungere tale obiettivo, è necessario analizzare:

- **Cosa** va descritto
- **Come** le varie parti vanno definite
- **Quali primitive** possono essere assunte

Una volta che la lingua viene definita in modo logico, occorrerebbe definire tutti i predicati e le funzioni non elementari *(come* $=, >, +, -, \times, *, \cdot, \ldots$*)*. Per evitare di doversi imbarcare in questa operazione, si fa normalmente riferimento a logiche più ristrette dalla sintassi specifica già predisposta.

8.3.1 Esempi di descrizione della lingua tramite *FOL*

8.3.1.1 Esempio 1

L'insieme:

$$\{a^n b^n \mid n \geq 0\}$$

può essere vista come una abbreviazione di:

$$x \in L \Leftrightarrow \exists n \mid n \geq 1 \wedge x = a^n b^n$$

All'interno di questa formula si possono riconoscere:

- I **predicati** $\in L, \geq, =$
- Le **funzioni** di *concatenazione, elevamento a potenza*

Come già enunciato, sarebbe necessario definire ciascuno di questi elementi sopra elencati. Per esempio, x^n può essere descritto *ricorsivamente* come:

$$\forall n \forall x ((n = 0 \Rightarrow x^n = \epsilon) \wedge (n > 0 \Rightarrow x^n = x^{n-1} \cdot x))$$

8.3.1.2 Esempio 2

Sia L il linguaggio definito come:

$$L = a^* b^*$$

Quindi L è il linguaggio delle stringhe su $\{a, b\}$ che iniziano con a. Più precisamente, fanno parte di L le stringhe:

- vuota (ϵ)
- composta da un carattere a e un suffisso appartenente a L
- composta da un prefisso appartenente a L e da un carattere b

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow (x = \epsilon) \vee (\exists y \,|\, x = at \wedge y \in L) \vee (\exists y \,|\, x = yb \wedge y \in L)$$

8.3.1.3 Esempio 3

Sia L il linguaggio delle stringhe su $\{a, b, c\}$ definito come:

$$L = a^* b^* c^*$$

Quindi L è il linguaggio delle stringhe su $\{a, b, c\}$ con tutte le a all'inizio, poi tutte le b e poi tutte le c.
Tra i molti modi possibili di rappresentare formalmente L, si consideri il seguente: è possibile definire due linguaggi *"ausiliari"* più semplici $L_1 = a^*b^*, L_2 = b^*c^*$ e affermare che una stringa appartiene a L se:

- è in L_1
- è in L_2
- composta da un carattere a e un suffisso appartenente a L
- composta da un prefisso appartenente a L seguito da un carattere c

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow (x \in L_1) \vee (x \in L_2) \vee \exists y \,((x = ay \wedge y \in L) \vee (x = yc \wedge y \in L))$$

in cui L_1, L_2 sono definite come nell'esempio precedente (Sezione 8.3.1.2).

8.3.1.4 Esempio 4

Sia L il linguaggio delle stringhe su $\{a, b\}$ in cui il numero di a sia uguale al numero di b.
Per definire questo linguaggio in *FOL* si può introdurre la funzione di *arietà* 2 $\#(a, x)$, che conta il numero di apparizioni del simbolo a nella stringa x. Questa funzione è definita formalmente dalla formula:

$$(x = \epsilon \Rightarrow \#(a, x) = 0) \wedge \forall y \,((x = ay \Rightarrow \#(a, x) = \#(a, y) + 1) \wedge (x = by \Rightarrow \#(a, x) = \#(a, y))$$

la cui definizione dipende dall'alfabeto.

Questo linguaggio può quindi essere espresso in *FOL* come:

$$x \in L \Leftrightarrow (x \in \{a, b\}^*) \wedge (\#(a, x) = \#(b, x))$$

8.3.2 Osservazioni sulla formulazione logica

Non esiste una *"formula magica"* o un sistema di risoluzione unico per ottenere la descrizione in *FOL* di un linguaggio, ma può essere utile evidenziare delle linee guida:

- Dall'esempio nel Paragrafo 8.3.1.3 si ricava che, quando si rivolge l'attenzione all'ordine con cui si susseguono le lettere in un linguaggio, si può far ricorso a formule in cui le stringhe siano decomposte nella concatenazione di sottostringhe, appartenenti ad altri linguaggi, eventualmente definiti *ricorsivamente*
- Quando è necessario contare le occorrenze di alcune lettere, risulta utile definire una funzione in grado di contare i simboli a cui si è interessati, come nell'esempio nel Paragrafo 8.3.1.4

8.4 Logica monadica del primo ordine - *MFO*

Dato un alfabeto di ingresso Σ, le formule sulla logica monadica del primo ordine (o *MFO* per brevità) sono costruite dai seguenti elementi:

- **Variabili del primo ordine**
  - rappresentate da lettere minuscole $(x, y, \ldots)$
  - interpretate sull'insieme di numeri positivi $\mathbb{N}$
- **Predicati monadici** (unari), uno per ogni simbolo di Σ
  - rappresentate come $(a(\cdot), b(\cdot), \ldots)$
  - $a(x)$ è valutata come `vero` in una stringa w se e solo se il carattere di w in posizione x è a
- La **relazione di minore** $<$ tra numeri naturali
- Le normali proposizioni connettive e i quantificatori di primo ordine

Più precisamente, sia $\mathcal{V}$ un insieme finito di variabili del primo ordine, e sia Σ un alfabeto. Le formule ben formate (*WFF*, dall'Inglese *well formed formula*) della logica *MFO* sono definite secondo la seguente sintassi:

$$\phi \coloneqq a(x) \,|\, x < y \,|\, \neg\phi \,|\, \phi \vee \phi \,|\, \exists\, x(\phi)$$

con $a \in \Sigma,\ x, y \in \mathcal{V}$.

Sono verificate le seguenti definizioni di **connettivi proposizionali**:

$$\begin{aligned}
\phi_1 \wedge \phi_2 &\triangleq \neg(\neg\phi_1 \vee \neg\phi_2) \\
\phi_1 \vee \phi_2 &\triangleq \neg(\neg\phi_1 \wedge \neg\phi_2) \\
\phi_1 \Rightarrow \phi_2 &\triangleq \neg\phi_1 \vee \phi_2 \\
\phi_1 \Leftrightarrow \phi_2 &\triangleq (\phi_1 \Rightarrow \phi_2) \wedge (\phi_2 \Rightarrow \phi_1) \\
\forall\, x(\phi) &\triangleq \nexists\, x(\neg\phi)
\end{aligned}$$

I seguenti **quantificatori del primo ordine**:

$$\begin{aligned}
x \geq y &\triangleq \neg(x < y) \\
x \leq y &\triangleq \neg(x > y) \\
x = y &\triangleq x \leq y \wedge y \leq x \\
x \neq y &\triangleq \neg(x = y) \\
x > y &\triangleq y < x
\end{aligned}$$

E le seguenti definizioni di:

- **Costante**: $x = 0 \triangleq \forall\, y \neg(y < x)$
- **Successore** di un numero naturale: $\mathrm{succ}(x, y) \triangleq x < y \neg\, \exists\, (x < z \vee z < y)$
- **Somma** di valori costanti: $y = x + k \triangleq \exists\, z_0, \ldots, z_k (z_0 = x \wedge \mathrm{succ}(z_0, z_1) \wedge \ldots \mathrm{succ}(z_{k-1}, z_k) \wedge y = z_k)$
- **Primo**: $\mathrm{first}(x) \triangleq \neg\exists\, y\, (y < x)$
  - equivalente a $x = 0$
- **Ultimo**: $\mathrm{first}(x) \triangleq \neg\exists\, y\, (y > x)$
- **Sottrazione** di valori costanti: $y = x - k \triangleq x = y + k$

  $\rightarrow$ k è una costante in $\mathbb{N}$

8.4.1 Esempi di *MFO*

- Formula che è vera se tute e sole le parole il cui primo simbolo esiste ed è a:

$$\exists\, x\, (x = 0 \wedge a(x))$$

- Formula che è vera su tutte le parole in cui ogni a è seguita da una b:

$$\forall\, x\ (a(x) \Rightarrow \exists\, y\, (y = \mathrm{succ}(x) \wedge b(y)))$$

- Parole non vuote il cui ultimo simbolo è a:

$$\exists\, x\, (\mathrm{last}(x) \wedge a(x))$$

- Parole di almeno 3 simboli in cui il terzultimo simbolo è a:

$$\exists\, x\, (a(x) \wedge \exists\, y\, (y = x + 2 \wedge \mathrm{last}(y)))$$

8.4.2 Semantica

Una formula *MFO* è interpretata su una stringa $w \in \Sigma^+$ rispetto all'assegnazione $v : \mathcal{V} \to U$ con $U = \{0, \ldots, |w| - 1\}$, che mappa $\mathcal{V}$ a una posizione nella stringa w.
La relazione di assegnamento (indicata con $\vDash$) è definita nel modo seguente:

$$\begin{aligned}
&w, v \vDash a(x) && \textit{sse } w = uav, |u| = v(x)\\
&w, v \vDash x < y && \textit{sse } v(x) < v(y)\\
&w, v \vDash \neg\Phi && \textit{sse } w, v \nvDash \Phi\\
&w, v \vDash \Phi_1 \wedge \Phi_2 && \textit{sse } w, v \vDash \Phi_1 \wedge w, v \vDash \Phi_2\\
&w, v \vDash \forall\, x(\Phi) && \textit{sse } w, v' \vDash \Phi\, \forall\, v', v'(y) = v(y), y \neq x
\end{aligned}$$

Data una stringa ϕ, il linguaggio $L(\phi)$ è definito come:

$$L(\phi) = \left\{ w \in \Sigma^+ \mid \exists\, v : w, v \vDash \phi \right\}$$

8.4.3 Proprietà di *MFO*

I linguaggi esprimibili mediante *MFO* sono **chiusi** rispetto a:

- **Unione**
- **Intersezione**
- **Complemento**

Mentre non **non sono chiusi** rispetto alla stella di Kleene. Infatti, i linguaggi definiti tramite questa logica prendono il nome di *star-free*, cioè definibili unicamente tramite unione, intersezione, complemento e concatenazione di linguaggi finiti.

Ne consegue che:

- *MFO* è strettamente meno potente degli *FSA*
  - data una formula *MFO* si può sempre costruire un *FSA* equivalente
  - L_P può invece essere riconosciuto facilmente mediante *FSA*
- In *MFO* non è possibile esprimere il linguaggio L_P rappresentante tutte e sole le parole di lunghezza pari con $l = \{a\}$
  - La formula *MFO* $a(0) \wedge a(1) \wedge \mathrm{last}(1)$ definisce il linguaggio L_{P2} fatto dalla sola parola $\{aa\}$ di lunghezza 2
  - Poiché $L_P = L_{P2}^*$, a causa della mancata chiusura rispetto alla stella di Kleene, si spiega perché

8.5 Logica monadica del secondo ordine - *MSO*

Per ottenere lo stesso potere espressivo degli *FSA* è necessario poter quantificare sui predicati monadici. Quindi la **logica monadica del secondo ordine** è costruita sugli stessi elementi della *MFO* (visti nella Sezione 8.4) con in aggiunta:

- **Variabili del secondo ordine**
  - rappresentate da lettere maiuscole $X, Y, \ldots$
  - interpretate su *insiemi* di numeri naturali

Le formule ben formate *(WFF)* della logica *MSO* sono definite secondo la seguente sintassi:

$$\phi \coloneqq a(X) \mid X(x) \mid x < y \mid \neg\phi \mid \phi \vee \phi \mid \exists\, x(\phi) \mid \exists\, X(\phi)$$

con $a \in \Sigma,\ x, y \in \mathcal{V}_1, X \in \mathcal{V}_2$.

Le definizioni della logica *MFO* sono sempre valide, con l'aggiunta di:

$$\begin{aligned} x \in X &\triangleq X(x) \\ X \subseteq Y &\triangleq \forall\, x\,(x \in X \Rightarrow x \in Y) \\ X = Y &\triangleq (X \subseteq Y) \wedge (Y \subseteq X) \\ X \neq Y &\triangleq \neg(X = Y) \end{aligned}$$

con $a \in \Sigma,\ x, y \in \mathcal{V}_1,\ X, Y \in \mathcal{V}_2$.

8.5.1 Semantica

Una formula *MSO* è interpretata su una stringa $w \in \Sigma^+$ rispetto alle assegnazioni:

$$\begin{aligned} v_1 &: \mathcal{V}_1 \to \{0, \ldots, |w| - 1\} \\ v_2 &: \mathcal{V}_2 \to \wp(\{0, \ldots, |w| - 1\}) \end{aligned}$$

dove:

- v_1 mappa ogni variabile di prim'ordine di $\mathcal{V}_1$ a una posizione nella stringa w
- v_2 mappa ogni variabile di second'ordine di $\mathcal{V}_2$ a un insieme di posizioni della stringa w

Quindi, la relazione di assegnamento per le formule della logica *MSO* è definita nel modo seguente:

$$\begin{aligned} w, v_1, v_2 &\vDash a(x) && \text{sse } w = w_1 a w_2 \wedge |w_1| = v_1(x) \\ w, v_1, v_2 &\vDash X(x) && \text{sse } v_1(x) \in v_2(X) \\ w, v_1, v_2 &\vDash x < y && \text{sse } v_1(x) < v_1(y) \\ w, v_1, v_2 &\vDash \neg\phi && \text{sse } w, v_1, v_2 \nvDash \phi \\ w, v_1, v_2 &\vDash \phi_1 \vee \phi_2 && \text{sse } w, v_1, v_2 \vDash \phi_1 \vee w, v_1, v_2 \vDash \phi_2 \\ w, v_1, v_2 &\vDash \exists\, x(\phi) && \text{sse } w, v_1', v_2 \vDash \phi \text{ per un } v_1',\ v_1'(y) = v_1(y)\, \forall\, y \in \mathcal{V}_1 \backslash \{x\} \\ w, v_1, v_2 &\vDash \exists\, X(\phi) && \text{sse } w, v_1, v_2' \vDash \phi \text{ per un } v_2',\ v_2'(Y) = v_2(Y)\, \forall\, Y \in \mathcal{V}_2 \backslash \{X\} \end{aligned}$$

8.5.2 Espressività della logica *MSO*

Poiché ogni formula *MFO* è anche formula di *MSO*, e poiché è risultato impossibile descrivere il linguaggio L_P (come nella Sezione 8.4.3), si deduce immediatamente che la logica *MSO* è più espressiva della logica *MFO*. Per lo stesso motivo, si può dimostrare che la *MSO* ha la stessa espressività degli *FSA*, e quindi per ogni *FSA* sarà possibile costruire una formula *MSO* e viceversa.

8.5.2.1 Da *FSA* a *MSO*

In generale, grazie alle quantificazioni del second'ordine è possibile trovare, per ogni *FSA*, una formula *MSO* equivalente. L'idea generale della costruzione consiste nell'usare una variabile di secondo ordine X_q per ogni stato q dell'*FSA* M. Il valore di ciascun X_q corrisponde all'insieme delle posizioni di tutti i caratteri che M può leggere in una transizione partendo dallo stato q.
Assumendo che l'insieme di stati di M sia $Q = \{0, 1, \ldots, k\}$ per un qualsiasi k, con 0 che indica lo stato iniziale, la definizione di M che riconosce il linguaggio L è data dalla congiunzione di molteplici condizioni, ognuna traducente una parte della definizione di M.

8.5.2.2 Da *MSO* a *FSA*

Data una formula *MSO* Φ, è possibile costruire un *FSA* che accetta un linguaggio L definito da Φ tramite il teorema di *Büchi-Elgot-Trakhtenbrot*.

I passaggi che portano alla costruzione non sono oggetto di questo corso e non saranno mostrati.

8.6 Precondizioni e postcondizioni

Quando si programma una funzione è importante definire precisamente cosa fa, senza necessariamente specificare come lo fa. Questo è lo scopo di **precondizioni** e **postcondizioni**.

A questo scopo viene introdotta la **notazione di Hoare**:

$$\{\text{Precondizione: } Pre\}$$
$$\text{Funzione } F$$
$$\{\text{Postcondizione: } Post\}$$

dove:

- La **precondizione** indica cosa deve valere **prima** che la funzione F sia invocata
- La **postcondizione** indica cosa deve valere **dopo** che la funzione F ha finito terminato la propria esecuzione

Le precondizioni e postcondizioni possono essere definite in modi diversi, come:

- Linguaggi naturali
- Linguaggi per asserzioni
- Linguaggi ad hoc

Tra di questi si riconosce su tutte la *FOL*, che può essere usata a questo scopo.

8.6.1 Specifiche

Una specifica deve essere usata come un *"contratto"*, che contiene tutte le informazioni necessarie senza assunzioni a priori. Quando una qualche condizione viene eliminata dalla precondizione, la specifica diventa insoddisfacente.

Una **specifica formale** è una descrizione matematica di un sistema. Come per le specifiche, esistono più linguaggi di specifica diversi. Dopo aver specificato i requisiti di un algoritmo (o di un sistema), è necessario verificare la correttezza del medesimo. Utilizzando un modello matematico dell'implementazione costruita, è possibile ottenere la prova di correttezza come una dimostrazione di teorema.

Il problema nella specifica dei sistemi è detto **frame problem**. Esso rappresenta il problema di esprimere un dominio in logica senza dover descrivere esplicitamente le condizioni che non sono modificate da un'azione. Per questo motivo situazioni estremamente semplici possono richiedere formalizzazioni complesse.

8.6.2 Esempi di specifiche

8.6.2.1 Algoritmo di ricerca

Sia F una funzione che implementa la ricerca di un elemento x in un array ordinato a di n elementi.
Allora, *informalmente*:

- *Precondizione*: l'array a è ordinato
- *Postcondizione*: la variabile logica `found` deve essere vera se e solo se l'elemento x esiste nell'array a

Formalizzando le precedenti definizioni in *FOL* si ottiene:

- *Precondizione:* $\forall\, i\, (1 \leq n \leq n-1 \rightarrow a[1] \leq a[i+1])$
- *Postcondizione:* `found` $\leftrightarrow \exists\, i\, (1 \leq i \leq n \wedge a[i] = x)$

8.6.2.2 Agoritmo di ordinamento

Sia ORD un programma che ordina un'array a di n elementi senza ripetizioni.
Allora, *informalmente*:

- *Precondizione*: l'array a non contiene ripetizioni
- *Postcondizione*: l'array ottenuto o è ordinato

Tuttavia questa definizione **non è sufficiente**: è necessario anche indicare che ogni elemento di a deve essere presente in o. Si aggiunge un array b (non utilizzato da ORD) con gli stessi elementi di a per poter fare rifermento a quest'ultimo prima che venga modificato.
Formalizzando le precedenti definizioni in *FOL* si ottiene:

- *Precondizione:*

$$\neg\exists\, i, j\, (1 \leq i \leq n \wedge 1 \leq j \leq n \wedge i \neq j \wedge a[i] = a[j]) \wedge \forall\, i\, (1 \leq i \leq n \rightarrow a[i] = b[i])$$

- *Postcondizione:*

$$\begin{aligned} &\forall\, i\, (i \leq i \leq n \rightarrow a[i] \leq a[i+1]) \wedge \\ &\forall\, i\, (1 \leq j \leq n \rightarrow \exists\, j\, (1 \leq j \leq n) \wedge a[i] = b[j]) \wedge \\ &\forall\, j\, (1 \leq i \leq n \rightarrow \exists\, i\, (1 \leq i \leq n) \wedge a[i] = b[j]) \end{aligned}$$

8.6.3 Esempi di specifica formale

8.6.3.1 Comportamento di una lampada

Se premo il tasto, la luce si accende entro Δ unità di tempo.
È necessario introdurre dei predicati opportuni:

- $P_B(t)$: istante di pressione del tasto
- $L_{ON}(T)$: istante di accensione della luce

A questo punto si potrebbe **erroneamente** definire la equivalente formula *FOL* della specifica come:

$$\forall\, t\, (P_B(t) \rightarrow \exists\, t_1\, ((t \leq t_1 \leq t + \Delta) \wedge L_{ON}(t_1)))$$

Tuttavia:

- **Non è specificato** che qualcuno debba premere il pulsante affinché la luce si possa accendere.
- **Non è specificato** cosa succede dopo che la luce si è accesa
- **Non è specificato** cosa succede se si preme il pulsante quando la luce è accesa
- **Non è specificato** cosa succede se si preme il pulsante due volte
- **Non è specificato** se la luce può essere accesa senza premere il tasto

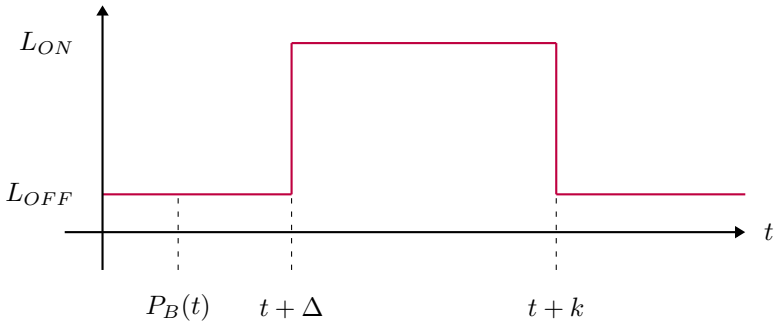


Figura 29: Schema di funzionamento della lampada

Per poter dare una specifica corretta, si altera leggermente la definizione di funzionamento, supponendo che dopo essersi accesa la lampada rimanga nel suo stato per k unità di tempo per poi spegnersi. Lo schema di funzionamento è riportato nella Figura 29.
La specifica corretta sarà:

$$\forall t \left((P_B(t) \wedge L_{OFF}(t)) \rightarrow \forall t_1 \left(t \leq t_1 \leq t + k \right) \rightarrow L_{ON}(t_1) \wedge L_{OFF}(t + k) \right)$$
$$\wedge$$
$$\forall t_3, t_4 \left((L_{OFF}(t_3) \wedge \forall t_5 \left(t_3 \leq t_5 \leq t_4 \right) \rightarrow \neg P_B(t_5)) \rightarrow L_{OFF}(t_4) \right)$$

9 Teoria della computabilità

Dopo aver illustrato un numero significativo di modelli atti a descrivere problemi di elaborazione delle informazioni e la loro soluzione e dopo averne studiato le capacità e i limiti, ci dedicheremo ora a studiare quali problemi possano essere affrontati e risolti mediante macchine da calcolo. Prima di poter continuare, è importante riassumere alcuni concetti appresi fino ad ora:

1. Automi, grammatiche e altri formalismi possono essere considerati dispositivi meccanici per risolvere problemi matematici (e quindi i problemi pratici che essi modellano)
2. Alcuni formalismi sono più potenti di altri, ossia sono in grado di riconoscere alcuni linguaggi che altri sono incapaci di riconoscere
3. Nessun formalismo, tra quelli visti fino ad ora, è più potente di una *TM*
   → alcuni hanno tuttavia la stessa medesima potenza delle *TM*
   → questi modelli verranno d'ora in poi chiamati **formalismi massimi**

Una volta costruita una visione di insieme più completa, può venire naturale porsi delle domande del tipo:

- *"I formalismi introdotti sono adeguati per catturare l'essenza di un solutore meccanico?"*
- *"La capacità di un meccanismo di risolvere un problema dipende da come è formalizzato?"*
- *"Esistono formalismi e modelli più potenti delle TM?"*
- *"Una volta che un problema è stato formalizzato adeguatamente, è sempre possibile risolverlo tramite dispositivi meccanici?"*

Per rispondere a queste *(e altre)* domande, è nata una branca dell'informatica detta **teoria della computabilità**.

9.1 Formalizzazione di un problema matematico

I formalismi fino ad ora studiati sono adeguati per tutti i problemi con domini numerabili, ovverosia gli insiemi i cui elementi possono essere messi in corrispondenza biunivoca con gli elementi di $\mathbb{N}$. Il problema originale, quindi, si riduce a quello del calcolo di una funzione $f : \mathbb{N} \to \mathbb{N}$, esattamente come una traduzione o come un riconoscimento di linguaggio in un determinato alfabeto.

Si consideri, come *esempio*, il problema relativo alla ricerca delle soluzioni di un sistema di equazioni del tipo:

$$\begin{cases} a_1 x_1 + a_2 x_2 = c_1 \\ b_1 x_1 + b_2 x_2 = c_2 \end{cases}$$

Esso può essere descritto come il calcolo della funzione f da $\mathbb{Z}^6$ a $\mathbb{Q}^2$ definita come:

$$f(a_1, a_2, b_1, b_2, c_1, c_2) = \langle x_1, x_2 \rangle$$

dove x_1, x_2, se esistono, sono i numeri razionali che soddisfano il sistema.
Si nota poi che è possibile far corrispondere biiettivamente $\mathbb{Z}$ *(numeri interi)* e $\mathbb{Q}$ *(numeri razionali)* con $\mathbb{N}$ *(numeri interi positivi)*, riducendo il problema al calcolo dell'opportuna funzione $f : \mathbb{N} \to \mathbb{N}$ come proposto prima.
Poiché tutti e 3 gli insiemi sono **enumerabili**, la correlazione esiste e può essere costruita. Un esempio di tale correlazione è mostrata nella Tabella 3.

$\mathbb{Z}$	0	1	-1	2	-2	3	-3	...
$\mathbb{N}$	0	1	2	3	4	5	6	...

Tabella 3: Correlazione tra $\mathbb{Z}$ e $\mathbb{N}$

9.2 Riconoscimento e traduzione

Il **riconoscimento** e la **traduzione** sono due formulazioni di un problema che possono essere ridotte l'una nell'altra.
Infatti, il problema di stabilire se, per una data stringa x e un dato linguaggio L, $x \in L$ si può anche impostare come la traduzione τ_L, dove

$$\tau_L(x) = \begin{cases} 1 & \text{se } x \in L \\ 0 & \text{altrimenti} \end{cases}$$

Viceversa, data una traduzione $\tau : V_1^* \to V_2^*$ si può definire il linguaggio:

$$L_\tau = \{z \mid z = x\$y,\ x \in V_1^*,\ y \in V_2^*,\ \$ \notin (V_1 \cup V_2),\ y = \tau(x)\}$$

cioè il linguaggio delle stringhe formate da una stringa su V_1^*, seguita dalla sua traduzione separata dal carattere speciale \$.
Un dispositivo che riconosce L_τ può essere usato come trasduttore che calcola τ: per ogni x, è possibile enumerare **tutte le infinite** $y \in V_2^*$ e verificare se $x\$y \in L_\tau$. Se ciò avviene, si può concludere che la stringa y è la traduzione rispetto a τ della stringa x, cioè $\tau(x) = y$. Tuttavia questo processo termina se e solo se $\tau(x)$ è definita, perché se così non fosse si procederebbe a enumerare le $y \in V_2^*$ senza mai trovare una stringa che appartenga a L_τ.

Infine, è importante ricordare che:

- Tutti i formalismi esaminati sono discreti, perché domini matematici numerabili definiti in modo finito
- La classe dei problemi che possono essere risolti da una *TM* è indipendente dall'alfabeto scelto *(a patto sia composto da almeno due simboli)*

9.3 *TM* e linguaggi di programmazione

Data una *TM* M, è possibile scrivere un programma che la simuli in un qualsiasi linguaggio. Analogamente, dato un qualunque programma, in un qualsiasi linguaggio di programmazione, è possibile costruire una *TM* che calcola la stessa funzione eseguita dal primo.
Quindi, è immediato dedurre che **le *TM* hanno lo stesso potere espressivo dei linguaggi di programmazione di alto livello** e di conseguenza sarà possibile sfruttare una *TM* per eseguire un algoritmo.

9.3.1 Algoritmi

Il concetto di **algoritmo** è uno dei concetti centrali dell'informatica. *Intuitivamente*, esso indica la procedura di risoluzione di un problema mediante un dispositivo automatico di calcolo, come un calcolatore elettronico. Alternativamente può indicare un metodo per descrivere una serie astratta di comandi elementari, indipendenti dal linguaggio comprensibile a un calcolatore, per risolvere un dato problema.
Una definizione formale di algoritmo sarebbe di difficile formulazione, quindi ci si limita a descriverne delle proprietà in modo informale:

A) Un algoritmo contiene una sequenza **finita** di istruzioni
B) Ogni istruzione deve essere **immediatamente eseguibile** tramite qualche procedimento meccanico
C) Il processore è dotato di **dispositivi di memoria** in cui possono essere immagazzinati i risultati intermedi
D) La computazione è un processo **discreto**
   → l'informazione è codificata in forma digitale
   → la computazione procede attraverso passi discreti
E) Gli algoritmi sono eseguiti in modo **deterministico**
F) **Non esiste un limite** alla quantità di **dati** di ingresso e uscita
G) **Non esiste un limite** alla quantità di **memoria** richiesta per effettuare i calcoli
H) **Non esiste limite** al numero di passi discreti richiesti per effettuare un calcolo
   → si possono avere computazioni infinite

Le ipotesi dalla (A) alla (D) si applicano agli algoritmi nei calcolatori digitali, mentre quelle dalla (E) alla (H) si applicano in senso generale. Il punto (D) relativo al non determinismo è una semplificazione *(analogamente al punto (C) riguardante la memoria)* rispetto al formalismo delle *TM*.

9.4 Tesi di Church

La tesi di Church è intrinsecamente non dimostrabile perché è basata solo sull'esperienza precedente e sull'evidenza intuitiva. Per questo motivo non è considerato un teorema.
È divisa in due parti.

9.4.1 Prima parte

La prima parte della **tesi di Church** afferma che:

> *"Non esiste alcun formalismo, per modellare una determinata computazione meccanica, che sia più potente delle TM e dei formalismi a esse equivalenti."*

Con la locuzione *potenza di calcolo* si intende la classe di problemi risolvibili tramite un determinato formalismo e non lo *"sforzo"* che la macchina impiega nel calcolo della soluzione.
Questa tesi è da verificare ogni qualvolta viene inventato un modello più potente di calcolo *(come i computer quantistici)* ma tutt'oggi non è ancora stata provata la sua falsità.

9.4.2 Seconda parte

La seconda parte della **tesi di Church** afferma che:

> *"Ogni algoritmo per la soluzione automatica di un problema può essere codificato in termini di una TM (o di un formalismo equivalente)."*

Ciò implica che nessun algoritmo può risolvere problemi che non possano già essere risolti da una *TM*: essa infatti rappresenta il più potente calcolatore che sia possibile costruire.

Per la definizione di algoritmo, si faccia riferimento alla Sezione 9.3.1.

9.5 Enumerazione delle *TM* e *TM* universali

Fino ad ora le *TM* sono state analizzate in quanto dispositivi in grado di risolvere **un particolare** problema. In particolare, è stato definito il problema definito dalla funzione di transizione δ, facente parte della struttura della macchina stessa.
Per questo motivo, le *TM* possono essere considerate come macchine calcolatori astratti, specializzati e non programmabili. Una volta caricato il programma nella memoria *(l'automa di controllo)*, la macchina potrà eseguire solo quella particolare sequenza di istruzioni.
A questo punto può venire naturale chiedersi:

> *"È possibile modellare un calcolatore programmabile con una TM? Le TM sono in grado di calcolare tutte le funzioni da $\mathbb{N}$ a $\mathbb{N}$?"*

Le prossime Sezioni si dedicheranno alla ricerca di una soluzione a entrambe le domande.

9.5.1 Enumerazione algoritmica

Dato un qualsiasi insieme S, esso può essere **numerato algoritmicamente** se è possibile stabilire una relazione biettiva fra lo stesso S e l'insieme dei numeri naturali $\mathbb{N}$, tramite un algoritmo o una *TM (grazie alla tesi di Church)*. È possibile dimostrare formalmente che le *TM* possono essere enumerate **algoritmicamente**.

Per chiarire le idee sulla enumerazione, si consideri il seguente *esempio*:
Sia L l'insieme $\{a, b\}^*$ delle stringhe sull'alfabeto $\{a, b\}$. Tale insieme è algoritmicamente enumerabile: per esempio, si possono ordinare le stringhe per lunghezza crescente e, a parità di lunghezza, assegnare un ordine tra gli elementi dell'alfabeto su cui sono definite. Il risultato è mostrato nella Tabella 4.

ϵ	a	b	aa	ab	ba	bb	aaa	aab	$\dots$
0	1	2	3	4	5	6	7	8	$\dots$

Tabella 4: Enumerazione algoritmica su $\{a, b\}^*$

9.5.2 Algoritmo di enumerazione

Per semplicità, si fissi un alfabeto A e si consideri l'insieme $\{TM_A\}$ delle TM a nastro singolo e senza stati finali, aventi A come insieme di simboli. Le TM in $\{TM_A\}$ accettano una stringa se e solo se arrivano in uno stato di `halt`. In caso contrario, compierebbero infinite mosse, entrando in un loop da cui non possono più uscire. Queste semplificazioni non comportano perdita di generalità perché queste operazioni sono possibili su ciascuna TM vista fino ad ora.
L'obiettivo di questa dimostrazione sarà spiegare come $\{TM_A\}$ possa essere numerato, ossia come sia possibile stabilire una biiezione $\mathscr{E} : \mathbb{N} \leftrightarrow \{TM_A\}$.
Si osservi in primo luogo che $\forall k \in \mathbb{N}$ esiste un numero finito di TM, con k stati e con alfabeto A. Infatti, la funzione di transizione δ sarà definita su dominio e codominio finiti. In senso più generale, data una funzione $f : D \to R$, con D, R finiti, vi sono esattamente $|R|^{|D|}$ funzioni totali f diverse.
Seguendo lo stesso ragionamento per le TM, poiché la loro funzione di transizione di transizione parziale δ è definita come

$$\delta : Q \times A \to Q \times A \times \{R, L, S\}^{k+1} \cup \{\perp\}$$

esisteranno esattamente:

$$(1 + 3 \times |Q| \cdot |A|)^{|Q| \cdot |A|} = (1 + 3 \times h \cdot k)^{h \cdot k}$$

TM, aventi $|A| = h, |Q| = k$ *(cardinalità dell'alfabeto e degli stati)*. Il termine unitario è dovuto alla presenza di $\perp$ nel codominio.
Per poter enumerare le TM dell'insieme sarà però prima necessario ordinarle in base all'*ordine lessicografico*. A tal fine è sufficiente imporre un ordinamento all'interno degli insiemi Q, A ed $\{R, L, S\}$, ottenendo un insieme ordinato:

$$\mathscr{E} : \{TM_A\} \mapsto \mathbb{N}$$

L'enumerazione $\mathscr{E}$ è algoritmica: è possibile implementare la costruzione con un algoritmo *(sia esso implementato in un linguaggio di programmazione o a sua volta una TM)* che emetta in uscita tutti gli elementi di $\{TM_A\}$ nell'ordine lessicografico appena imposto. Analogamente, sarà possibile costruire un algoritmo che faccia l'operazione inversa, cioè che partendo da una TM ne estragga il corrispondente $k \in \mathbb{N}$.

Infine l'enumerazione calcolabile da una TM prende il nome di **Gödelizzazione** e il numero naturale associato verrà chiamato **numero di Gödel** dell'oggetto. Nella precedente enumerazione, ogni TM è biiettivamente identificata dal suo numero di Gödel.

9.5.3 Riassunto dell'enumerazione

Per riassumere i concetti visti nell Sezione precedente, si ottiene un enunciato fondamentale:

1. Per ogni alfabeto A, il corrispondente l'insieme di TM $\{TM_A\}$ può essere numerato algoritmicamente, perché può essere sempre stabilita una biiezione $\mathscr{E} : \{TM_A\} \leftrightarrow \mathbb{N}$
2. Tutte le funzioni **algoritmicamente computabili** si possono enumerare algoritmicamente
3. Tutti i linguaggi **algoritmicamente riconoscibili** si possono enumerare algoritmicamente

Si noti che, sebbene $\mathscr{E}$ sia una funzione biettiva, le enumerazioni delle funzioni computabili e dei linguaggi riconoscibili non sono sempre tali, perché si verifica che molte TM sono in grado di calcolare la medesima funzione.

Per comodità, nel resto del corso si farà riferimento a TM che si comportano come dispositivi che calcolano funzioni da $\mathbb{N}$ a $\mathbb{N}$. Per ogni funzione $f : D \to R$, con D, N diversi da $\mathbb{N}$, si presupporrà implicitamente la codifica algoritmica di D e R in funzione di $\mathbb{N}$. Analogamente si farà riferimento alla y-esima TM generata dalla enumerazione $\mathscr{E}$ con M_y (quindi $M_y = \mathscr{E}(y)$).

9.5.4 Macchina di Turing universale - *UTM*

Fino a ora le *TM* sono state considerate come calcolatori astratti, specializzati e non programmabili. In questa sezione verrà introdotta la **macchina di Turing universale** (o *UTM*, dall'Inglese *Universal Turing Machine*), ossia una *TM* in grado di modellare dispositivi di risoluzione dei problemi, in cui il problema da risolvere non viene codificato nella struttura del dispositivo ma viene fornito in ingresso insieme ai dati su cui operare.

La *UTM* può essere definita come una *TM* che calcola la funzione $g(y, x) = f_y(x)$, dove:

- $y \in \mathbb{N}$ indica l'indice della **funzione** f_y calcolata dalla *TM* M_y
- $x \in \mathbb{N}$ rappresenta l'**ingresso** su cui opera M_y

Apparentemente la *UTM* così definita non sembrerebbe appartenere all'insieme $\{TM_A\}$ in quanto le funzioni a esso associate sono in una variabile, mentre g è definita a due variabili. Questo problema si dimostra facilmente risolvibile in quanto esiste **sempre** una biezione $\mathbb{N} \times \mathbb{N} \mapsto \mathbb{N}$. Un esempio di tale biiezione è dato dalla funzione:

$$d(x, y) = \frac{(x + y)(x + y + 1)}{2} + x$$

che permette di associate una coppia $x, y \in \mathbb{N}$ a un numero $k \in \mathbb{N}$ in modo biunivoco. Allo stesso modo sarà possibile costruire la funzione inversa d^{-1}.
La Figura 30 illustra il funzionamento della biezione.

Poiché si dimostra che d e d^{-1} sono computabili, la *TM* sarà **sempre** un grado di calcolarla.

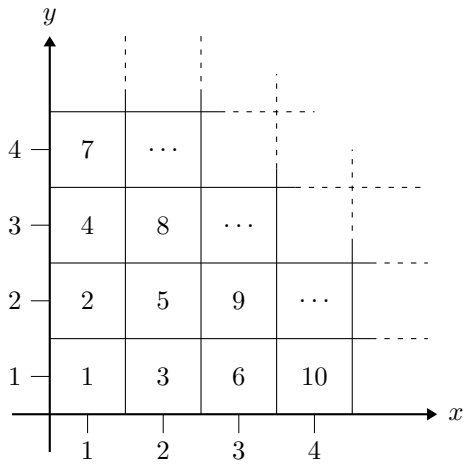


Figura 30: Illustrazione dell'associazione biunivoca

Si osservi che g è una funzione *TM*-computabile, ossia esiste un $i \in \mathbb{N}$ tale che $f_i = g$ ed è possibile calcolarla mediante i passi seguenti:

1. Si sceglie un alfabeto finito A per codificare ogni informazione richiesta per la computazione.
   → qualunque A con $|A| \geq 2$ è adatto allo scopo
2. Si traduce la rappresentazione di n nella opportuna coppia $\langle x, y \rangle$ tramite la funzione biettiva scelta
3. Si traduce il numero y in un'opportuna codifica della *TM* y-esima nella enumerazione $\mathscr{E}$, M_y
   → La traduzione viene memorizzata sul nastro della *UTM*
4. Si simula la computazione di M_y su x
   → la *UTM* lascia sul nastro $f_y(x)$ se e solo se M_y termina la sua computazione su x

Quindi viene dimostrato che la *UTM* è in grado di calcolare $g(x, y) = f_y(x)\ \forall\, x, y$, cioè il comportamento di ogni altra *TM*. La *UTM* è quindi in grado di calcolare anche il comportamento di **sé stessa**.

Con questo enunciato viene risposta la prima domanda posta nella Sezione 9.5.1.

9.6 Problemi algoritmicamente irrisolvibili

Prima di potersi occupare della risolvibilità dei problemi *(e della loro relativa irrisolvibilità)*, è necessario studiare il numero delle stesse funzioni computabili. Come concluso nella Sezione 9.5.3, infatti, tutte le funzioni computabili $f_y : \mathbb{N} \to \mathbb{N}$ sono enumerabili. Questo significa che la cardinalità del loro insieme è $\aleph_0$.
Analogamente, la cardinalità della classe $\mathscr{F}$ delle funzioni su $\mathbb{N}$, cioè $\mathscr{F} = \{f \mid f : \mathbb{N} \to \mathbb{N}\}$ è:

$$|\mathscr{F}| = |\mathbb{N}| \times |\mathbb{N}| = \aleph_0 \times \aleph_0 = 2^{\aleph_0}$$

Questo valore è noto come **cardinalità del continuo** perché pari alla cardinalità dell'insieme $\mathbb{R}$ dei numeri reali.
Quindi rapportando le due cardinalità *(rispettivamente $\aleph_0$ e $2^{\aleph_0}$ per le funzioni risolvibili e per tutte le funzioni)* si conclude che la prima è minore della seconda ($\aleph_0 < 2^{\aleph_0}$), e quindi **non tutte le tutte le funzioni sono risolvibili**.

Così facendo è stata data risposta anche alla seconda domanda posta nella Sezione 9.5.1.

9.6.1 Problemi definibili

Intuitivamente, le conclusioni raggiunte nella sezione precedente non forniscono grandi limitazioni. Infatti, nonostante la classe di tutte le funzioni $f : \mathbb{N} \to \mathbb{N}$ sia di cardinalità superiore alla classe delle funzioni computabili, le sole funzioni *interessanti* da calcolare sono quelle possono essere *definite*.
Per poter quindi definire un problema, è necessario usare una stringa di un qualche linguaggio, sia esso di tipo naturale o formale. Per esempio, alcune descrizioni di problemi possono essere:

- *il numero che moltiplicato per se stesso è uguale a y*
- $f(x) = x^4 - \pi \cdot x^2$
- $f(x) = \displaystyle\int_0^x z \cdot sin(z)dz$
- $f(x) = \displaystyle\lim_{x\to\infty} \frac{e^x}{x^2 + x}$

Ogni linguaggio è un sottoinsieme di A^* *(il linguaggio delle UTM prese in esame)* e quindi è numerabile. Così si prova che l'insieme dei problemi che si possono definire è esso stesso numerabile e:

$$\{\text{problemi risolvibili}\} \subseteq \{\text{problemi definibili}\}$$

ma, *purtroppo*, i due insiemi non coincidono **e parte dei problemi definibili non sarà risolvibile**. La relazione tra i due insiemi sarà quindi:

$$\{\text{problemi risolvibili}\} \subset \{\text{problemi definibili}\}$$

In pratica, la classe dei problemi non definibili è rappresentata da tutti quei problemi che, per carenze di tipo linguistico *(sia esso naturale, matematico o altro)* non è possibile esprimere. L'assenza di una soluzione per essi non è rilevante in quanto non è neanche possibile porsi il problema di cercarla.

9.7 Problema della terminazione - *halting problem*

Una delle proprietà che si vuole garantire durante la stesura di un programma, qualunque sia il linguaggio, è la capacità del programma stesso di *terminare* correttamente, ossia che *"non vada in loop"*.
Sarebbe estremamente utile poter determinare a priori, prima di eseguire una funzione con un determinato input, se essa porti a risultati in tempo finito o cada in loop infiniti che la portino a non terminare mai.
Questo problema, detto **problema della terminazione** (o in Inglese *halting problem*), è *descrivibile*, ma purtroppo *non decidibile*.

Analizzando il problema in modo *formale*, si indicherà con M_y la *TM* y-esima secondo la numerazione $\mathscr{E}$ e con f_y la funzione calcolata da M_y. Allora, è verificato che nessuna *TM* può calcolare la funzione totale $g : \mathbb{N} \times \mathbb{N} \to \{0, 1\}$ definita nel seguente modo:

$$g(x, y) = \begin{cases} 1 \text{ se } f_y(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

e che quindi nessuna *TM* può decidere se, per una generica *TM* M e un generico ingresso x, M si arresta in uno stato finale *(denotato come* $f_y(x) = \perp$*)* con l'ingresso x.
La dimostrazione avverrà tramite diagonalizzazione, dopo l'introduzione della tecnica stessa nella prossima Sezione.

9.7.1 Dimostrazioni per diagonalizzazione

Il ragionamento originale per diagonalizzazione fu usato da *Georg Cantor* nel 1891 per dimostrare che $\mathbb{R}$ ha una cardinalità maggiore di $\mathbb{N}$. È un metodo spesso usato per mostrare l'indecidibilità di alcuni problemi famosi e si riferisce a uno schema comune: si cerca una contraddizione quando esiste un insieme di funzioni $A \to B$ indicizzato su A. Se f_k è una funzione indicizzata da $k \in A$, un ragionamento per diagonalizzazione è dato dall'analisi degli elementi sulla diagonale della tabella ottenuta enumerando le f_k, come nella Tabella 5 (in cui la diagonale è evidenziata in **bianco su grigio**).

n	1	2	3	4	...
f_1	$f_1(1)$	$f_1(2)$	$f_1(3)$	$f_1(4)$	...
f_2	$f_2(1)$	$f_2(2)$	$f_2(3)$	$f_2(4)$	...
f_3	$f_3(1)$	$f_3(2)$	$f_3(3)$	$f_3(4)$	...
f_4	$f_4(1)$	$f_4(2)$	$f_4(3)$	$f_4(4)$	...
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\ddots$

Tabella 5: Dimostrazione per diagonalizzazione

Allora si definisce l'elemento diagonale come una funzione $d : A \to B$ che differisce dalla diagonale in ogni valore. Poiché d è una funzione $A \to B$, essa appare nella lista.
Visto che essa differisce anche da ogni elemento della lista sulla diagonale, la dimostrazione è avvenuta **per assurdo**.

9.7.2 Dimostrazione della cardinalità del continuo

La dimostrazione avviene per assurdo ed è strutturata in passi:

1. Si supponga, per assurdo, che l'intervallo $[0, 1]$ sia numerabile
2. Ciò implica che gli elementi di $[0, 1]$ possano essere messi in corrispondenza biunivoca con i numeri naturali, tramite successione di numeri reali $\{n_1, n_2, \ldots\}$ che esaurisce tutti i numeri reali compresi tra 0 e 1
3. È possibile rappresentare ciascun numero della successione in forma decimale e visualizzare la successione di numeri reali come una matrice infinita che avrà circa questo aspetto:

$$
\begin{aligned}
n_1 &= 0,\ 3\ 1\ 3\ 5\ 6\ 3\ 1\ \ldots \\
n_2 &= 0,\ 6\ 8\ 1\ 3\ 9\ 8\ 7\ \ldots \\
n_3 &= 0,\ 1\ 6\ 6\ 5\ 7\ 1\ 1\ \ldots \\
n_4 &= 0,\ 1\ 3\ 6\ 6\ 5\ 5\ 8\ \ldots \\
n_5 &= 0,\ 8\ 5\ 2\ 4\ 1\ 8\ 4\ \ldots \\
n_6 &= 0,\ 4\ 1\ 2\ 4\ 2\ 5\ 7\ \ldots \\
n_7 &= 0,\ 7\ 8\ 8\ 4\ 6\ 6\ 3\ \ldots
\end{aligned}
$$

4. Si osservino ora le cifre lungo la diagonale della matrice, cioè sulla successione il cui k-esimo elemento è la k-esima cifra decimale di r_k come mostra la figura:

$$\begin{array}{lcccccccc} n_1 = 0, & \boxed{3} & 1 & 3 & 5 & 6 & 3 & 1 & \dots \\ n_2 = 0, & 6 & \boxed{8} & 1 & 3 & 9 & 8 & 7 & \dots \\ n_3 = 0, & 1 & 6 & \boxed{6} & 5 & 7 & 1 & 1 & \dots \\ n_4 = 0, & 1 & 3 & 6 & \boxed{6} & 5 & 5 & 8 & \dots \\ n_5 = 0, & 8 & 5 & 2 & 4 & \boxed{1} & 8 & 4 & \dots \\ n_6 = 0, & 4 & 1 & 2 & 4 & 2 & \boxed{5} & 7 & \dots \\ n_7 = 0, & 7 & 8 & 8 & 4 & 6 & 6 & \boxed{3} & \dots \end{array}$$

   Questa successione di cifre sulla diagonale, vista come un'espansione decimale, definisce un numero reale (che in questo caso può corrispondere a $0.3866153\ldots$).
5. Si consideri ora un nuovo numero reale x che abbia tutte le cifre differenti dalla sequenza sulla diagonale, definendolo come:
   - se la k-esima cifra decimale di n_k è 5, allora la k-esima cifra di x è 4
   - se la k-esima cifra decimale di n_k non è 5, allora la k-esima cifra di x è 5
6. All inizio della dimostrazione si era supposto che la lista $\{n_1, n_2, \ldots\}$ enumerasse tutti i numeri reali compresi tra 0 e 1, quindi dovrebbe essere verificato che $n_k = x_k$ per un qualche k e poiché x non ha dei 9 tra le cifre decimali, la sua rappresentazione è unica e sarà presente nella riga k
7. A questo punto emerge la contraddizione: sia a la n-esima cifra decimale di $n_k = x$. Essa può essere 4 o 5: per la definizione, a potrà essere 4 se e solo se è uguale a 5, e 5 negli altri casi. Questo è impossibile e ne segue che l'ipotesi di partenza è falsa.

9.7.3 Dimostrazione del Problema di Terminazione

Come già annunciato, la dimostrazione avviene tramite tecnica diagonale. Si assuma, per assurdo, che

$$g(y,x) = \begin{cases} 1 \text{ se } f_y(x) \neq \perp & \textit{termina} \\ 0 \text{ altrimenti} & \textit{non termina} \end{cases}$$

```
int g(y, x) {
  if (halts(f_y(x)))
    return 1;

  return 0;
}
```

(descritta sia matematicamente che in pseudocodice) sia computabile.
Partendo da g si definisce ora una funzione h tale che:

$$h(x) = \begin{cases} 1 \text{ se } g(x,x) = 0 & f_x(x) \textit{ termina} \\ \perp \text{ altrimenti} & f_x(x) \textit{ non termina} \end{cases}$$

```
int h(x) {
  if (g(x, x) == 0)
    return 1;

  while (1) {};
}
```

dove $h(x) = 1 \Rightarrow g(x,x) = 0,\ f_x(x) = \perp$. L'indefinizione non implica che la funzione h non sia calcolabile, ma significa che la funzione calcolata dalla corrispondente TM non termina.
Usando h, ci si trova quindi nella *diagonale* della tabella delle $f_y(x)$.

Se g fosse computabile, allora anche h lo sarebbe. Analogamente, se h fosse computabile, allora si verificherebbe che $h(i) = f_i(i)$ per una qualche i.

Si calcoli ora $h(i)$ ricordando che $f_x(i) = h(i)$ e applicando la definizione appena data:

$$h(i) = \begin{cases} 1 \Rightarrow f_i(i) \neq \perp \Rightarrow g(i,i) = 0 \Rightarrow f_i(i) = \perp & \textbf{assurdo} \\ \perp \Rightarrow f_i(i) = \perp \Rightarrow g(i,i) = \perp \Rightarrow f_i(i) \neq \perp & \textbf{assurdo} \end{cases}$$

Quindi entrambi i casi sono inammissibili e la dimostrazione è terminata.

9.7.4 Lemma 1

Sia h' la funzione definita come:

$$h'(x) = \begin{cases} 1 \text{ se } f_x(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Allora h' **non è computabile.**
Si noti che $h'(x)$ *"si muove"* ancora sulla diagonale (coincide con $g(x,x)$) ed è quindi una semplificazione di $f(x)$. È un problema analogo al problema di terminazione, di cui è appunto un lemma, ma non dipende da esso.

9.7.4.1 Dimostrazione del Lemma 1

La dimostrazione segue la traccia della dimostrazione del Teorema di Terminazione, nel avvenuta nella Sezione 9.7.3.

Si definisca inizialmente la funzione

$$h(x) = \begin{cases} 1 \text{ se } k(x) = 0 \\ \perp \text{ altrimenti} \end{cases}$$

Si osservi ora che h è computabile se k è computabile. Una volta ottenuta una *TM* M in grado di calcolare k, sono sufficienti solo semplici modifiche per adattarla al calcolo di h.
Considerando ora x_0 il numero di Gödel di una terminata *TM* M_{x_0} che calcola h, ossia $h = f_{x_0}$ e quindi h è la funzione calcolata dalla x_0-esima *TM*.
Si verifica facilmente, tuttavia, che $h(x_0) = 1$ implica $f_{x_0}(x_0) = 1$, mentre $h(x_0) = \perp$ implica $h(x_0) = 1$.
Quindi $f_{x_0}(x_0) = h(x_0) = 1$, che è una contraddizione.

9.7.5 Lemma 2

La funzione $h'(x)$ è un caso particolare della funzione $g(y,x)$ perché:

- $h'(x) = g(x,y)$
- g non è computabile, perché $g(y,x)$ è essa stessa non computabile

Se un problema non è risolvibile, allora un suo caso speciale può essere risolvibile. Contrariamente, la generalizzazione di un problema non risolvibile è necessariamente non risolvibile.
Infine, se un problema è risolvibile, allora una sua generalizzazione potrebbe non essere risolvibile, mentre qualunque sua specializzazione è certamente risolvibile.

9.7.6 Lemma 3

Sia $k(y)$ la funzione definita come:

$$k(y) = \begin{cases} 1 \text{ se } \forall\, x \in \mathbb{N},\ f_y(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Allora $k(y)$ **non è computabile.**
Questa è una quantificazione su tutti i possibili dati in ingresso. In qualche caso potrebbe essere possibile stabilire se $f_y(x) \neq \perp$ per un qualche x specifico, ma non sarà mai possibile per ogni $x \in \mathbb{N}$ *(perché è un insieme infinito)*.
Se ciò, *per assurdo*, fosse possibile, determinare se f_y è una funzione totale sarebbe impossibile.

9.7.6.1 Dimostrazione del Lemma 3

La dimostrazione segue la traccia della dimostrazione del Teorema di Terminazione, nel avvenuta nella Sezione 9.7.3.
Per ipotesi, sia $k(y)$ la funzione definita come:

$$k(y) = \begin{cases} 1 \text{ se } \forall\, x \in \mathbb{N},\ f_y(x) \neq \perp \\ 0 \text{ altrimenti} \end{cases}$$

Sempre per ipotesi, $k(y)$ è **totale** e **computabile**.
Allora sarà possibile definire $g(x) = w$, con w pari al numero di Gödel della x-esima *TM* (definita nell'insieme $\mathscr{E}$) che calcola una funzione totale.
Se k fosse veramente computabile e totale, allora lo sarebbe anche g. Infatti:

1. Sarebbe possibile calcolare $k(0),\ k(1),\ \ldots$
2. Sia w_0 il primo valore tale per cui $k(w_0) = 1$
3. Sia $g(0) = w_0$

La procedura è algoritmica, e dato che esistono infinite funzioni totali, $g(x)$ è sicuramente definita per ogni $x \in \mathbb{N}$, quindi è **totale** e **strettamente monotona**, perché $w_{x+1} > w_x$.
Di conseguenza, anche g^{-1} è strettamente monotona ma non totale, perché g^{-1} è definita solo se w è il numero di Gödel di una funzione totale.
Allora si definisce la funzione $h(x)$ come

$$h(x) = f_{g(x)} + 1 = f_w(x) + 1$$

e poiché $f_w(x)$ è computabile e totale, anche $h(x)$ lo sarà.
Per qualche valore di w_i, infine, si verifica $h(x) = f_{w_i}(x)$ e poiché h è totale, $g^{-1}(w_i) = x_i \Rightarrow g^{-1}(w_i) \neq \perp$.
Confrontando gli ultimi due valori ottenuti, si giunge alla conclusione che:

$$\begin{aligned} h(x_i) = f_{g(x_i)}(x_i) + 1 = f_{w_i}(x_1) + 1 \\ h(x) = f_{w_i}(x) \Rightarrow h(x_i) = f_{w_i}(x_1) \end{aligned}$$

Cadendo quindi in una evidente contraddizione.

9.8 Osservazioni sulla risolvibilità

È importante notare che sapere che un problema è risolvibile non implica che il metodo di risoluzione sia noto. In matematica è spesso necessario dare dimostrazioni non costruttive: si mostra che un oggetto matematico esiste, ma non lo si calcola.
In questo caso:

- Un problema è **risolvibile** se esiste una *TM* che lo risolve
- Per alcuni problemi si può concludere che esiste una *TM* che li risolve, **senza tuttavia poterla costruire**

9.9 Problema di decisione

Un **problema di decisione** è una domanda che ha due possibili risposte: `SI` o `NO`. La domanda *(e la consecutiva risposta)* sono relative a un qualche particolare ingresso.

Esempi di problemi di decisione:

- Dato un grafo G e un insieme di vertici K, quest'ultima è una cricca *(clique)*?
- Dato un grafo G e un insieme di lati M, quest'ultimo è un albero ricoprente *(spanning tree)*?
- Dato un insieme di assiomi, un insieme di regole e una formula, è possibile dimostrare quest'ultima a partire dai primi due? *problema di incompletezza*

9.9.1 Decidibilità

Un problema si dice **decidibile** se esiste un algoritmo *(procedura di decisione)* tale che:

- È un procedimento **automatico** *(da definizione di algoritmo)*
- Se il problema è **vero** per un determinato ingresso, allora ritorna `SI`
- Se il problema è **falso** per un determinato ingresso, allora ritorna `NO`

9.9.2 Semidecidibilità

Un problema si dice **semidecidibile** se esiste un algoritmo *(procedura di semidecisione)* tale che:

- È un procedimento **automatico** *(da definizione di algoritmo)*
- Se il problema è **vero** per un determinato ingresso, allora ritorna `SI`
- Negli altri casi, **potrebbe non terminare mai**

Il problema della terminazione (Sezione 9.7) è **indecidibile** e **semidecidibile** allo stesso tempo.

9.9.2.1 Problema della terminazione e semidecidibilità

È dimostrato che il problema della terminazione (Sezione 9.7) è **semidecidibile**. Si consideri infatti l'analoga formulazione $\exists\, z \mid f_x(z) \neq \perp$.
Schema di dimostrazione:

- Se si calcola $f_x(0)$ e ne risulta che $f_x(0) \neq \perp$ allora la risposta è `SI`
- Se la computazione non termina, come è possibile accorgersene?
- Dimostrazione con metodo diagonale:
  1. Si simuli 1 passo di esecuzione di $f_x(0)$. Se termina, allora la risposta è `SI`
  2. Altrimenti, si simuli un passo di computazione di $f_x(1)$. Se termina, allora la risposta è `SI`
  3. Ancora una volta si simuli 2 passi di $f_x(0)$, 1 passo di $f_x(2)$, 2 passi di $f_x(1)$, 3 di $f_x(0)$ e così via, secondo lo schema in Figura 31 *(già vista nella Sezione relativa alla Macchina di Turing universale - UTM)*

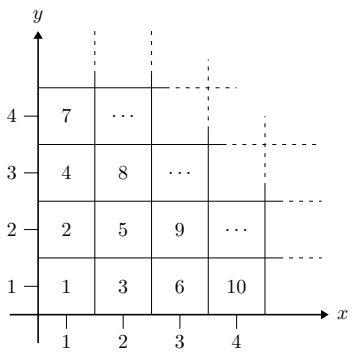


Figura 31: Illustrazione dell'associazione biunivoca

9.9.2.2 Osservazioni

Grande parte dei problemi indecidibili è **anche** semidecidibile. Un tipico esempio di questa affermazione è data dagli errori a *runtime* nei programmi.

Si nota che in questi casi il problema semidecidibile è dato dalla ricerca della presenza dell'errore, non nella dimostrazione della sua assenza. Ciò porta a una importante implicazione sulla verifica basata sul testing. Secondo Dijkstra:

> *"Il testing può dimostrare la presenza di errori e non la loro assenza."*

Molti problemi, seppur ben definiti, non possono essere risolti mediante procedimenti algoritmici. Il risultato sarà trovato tramite tecniche differenti.
Allo stesso modo esistono problemi la cui soluzione algoritmica è nota, senza che il procedimento stesso sia noto.

9.10 Insiemi ricorsivi

In questa sezione ci si dedicherà a studiare i sottoinsiemi di $\mathbb{N}$, che verranno indicati d'ora in poi con la lettera S. Dati $x \in \mathbb{N}, S \subseteq \mathbb{N}$, si vuole studiare l'appartenenza dell'elemento x allo stesso S.
Questa formalizzazione del problema è del tutto generale perché applicabile a qualsiasi insieme che si possa mettere in corrispondenza biunivoca ed effettiva con $\mathbb{N}$, ossia per tutti gli elementi numerabili.

9.10.1 Funzione caratteristica di un insieme

Dato un insieme S, la sua **funzione caratteristica** $c_s : \mathbb{N} \to \{0, 1\}$ è definita come:

$$c_s = \begin{cases} 1 \text{ se } x \in S \\ 0 \text{ altrimenti} \end{cases}$$

L'appartenenza di un elemento a un insieme e la ricorsività dello stesso è un problema la cui risolvibilità dipende dalla computabilità della sua funzione caratteristica.

9.10.2 Definizione di insieme ricorsivo

Un insieme S viene detto **ricorsivo** (oppure *decidibile*) se e solo se la sua funzione caratteristica c_s è **computabile**.
Si ricordi che, per definizione, c_s è **totale** per ogni insieme S. Infatti, dato un qualsiasi elemento $x \in \mathbb{N}$, esso appartiene (se $c_1 = 1$) o non appartiene (se $c_s = 0$) a S.

9.11 Insieme ricorsivamente enumerabile

Un insieme viene detto **ricorsivamente enumerabile** (oppure *semidecidibile*) se e solo se è l'**insieme vuoto** oppure è l'**immagine di una funzione** totale e computabile g_s, cioè:

$$S = I_{g_s} = \{x \mid \exists\, y,\ y \in \mathbb{N} \wedge x = g_s(y)\}$$

Gli insiemi ricorsivamente enumerabili devono il loro nome al fatto che il problema dell'appartenenza può essere risolto **meccanicamente** *(oppure algoritmicamente)*. Un calcolatore meccanico in grado di implementare la funzione caratteristica di un insieme fornirà necessariamente una risposta alla domanda $x \in S\ \forall\, x$. Gli insiemi ricorsivamente enumerabili sono una definizione *più debole* degli insiemi *ricorsivi*.

Il termine *"semidecidibile"* puo essere spiegato intuitivamente come:

> *"se si suppone che $x \in S$, allora enumerando gli elementi di S anche x verrà trovato e sarà possibile rispondere alla domanda. Ma se $x \notin S$?"*

9.12 Teorema $\frac{1}{2} + \frac{1}{2} = 1$

Sia S un insieme. Allora:

A) Se S è **ricorsivo**, è anche **ricorsivamente enumerabile**
B) S è **ricorsivo** se e solo se sia S che $\overline{S} = \mathbb{N} - S$ sono **ricorsivamente enumerabili**

Corollario: la classe di insiemi decidibili *(linguaggi, problemi, ...)* è chiusa rispetto al complemento.

9.12.1 Dimostazione punto (A)

Sia c_s la funzione caratteristica di S. Se S è vuoto, allora è ricorsivamente enumerabile per definizione. Altrimenti, esiste almeno un elemento $k \in S$.
Sia $g_s(x)$ la funzione definita come:

$$g_s(x) = \begin{cases} x \text{ se } c_s(x) = 1 \\ k \text{ altrimenti} \end{cases}$$

Allora $g_s(x)$ è totale, computabile e $I_{g_s} = S$. Quindi poiché S é l'immagine di una funzione totale e computabile, è ricorsivamente enumerabile per definizione.

Questa è una definizione **costruttiva**. Infatti, non è definito né se $S = \emptyset$ né l'algoritmo per trovare un k specifico. Viene definita solo l'esistenza di g_s per $S \neq \emptyset$.

9.12.2 Dimostazione punto (B)

Dimostrazione che se S è ricorsivo, è anche ricorsivamente enumerabile.
Se S è vuoto o coincide con $\mathbb{N}$, allora l'enunciato è ovvio.
Altrimenti, esistono due funzioni totali e computabili $g_s, g_{\overline{s}}$ tali che $S = I_{g_s}, \overline{s} = I_{g_{\overline{s}}}$ nella forma:

$$S = \{g_s(0), g_s(1), \ldots\}$$
$$\overline{S} = \{g_{\overline{s}}(0), g_{\overline{s}}(1), \ldots\}$$

Per costruzione

$$S \cup \overline{S} = \mathbb{N} \quad S \cap \overline{S} = \emptyset$$

e quindi:

$$\forall\, x \in \mathbb{N}\, (\exists\, y \,|\, x = g_s(y) \vee x = g_{\overline{s}} \wedge \neg(\, \exists\, z, w \,|\, x = g_s(z) \wedge x = g_{\overline{s}}(w)))$$

cioè ogni x appartiene in modo esclusivo a S o a $\overline{S}$.

Si consideri ora l'enumerazione:

$$\mathscr{E}(L) = \{g_s(0), g_{\overline{s}}(0), g_s(1), g_{\overline{s}}(1), \ldots\}$$

Poiché $\mathscr{E} \equiv \mathbb{N}$, $\forall\, x, x \in \mathscr{L}$ e, se x compare in $\mathscr{L}$ in posizione parti, allora non compare mai in posizione dispari e viceversa.
Si definisca c_s come:

$$c_{s_i} = \begin{cases} 1 \text{ se } x = s_i \in \mathscr{L} \text{ e } i = 2 \cdot k + 1 \\ 0 \text{ altrimenti} \end{cases}$$

Chiaramente c_s è ben definito, poiché x potrà apparire in $\mathscr{L}$ solo in posizione o pari o dispari (e non entrambe). Inoltre risulta totale e computabile, in quanto l'enumerazione $\mathscr{L}$ può essere effettuata da una qualche TM. Basta infatti cercare se x (elemento dato) appare in posizione pari o dispari di $\mathscr{L}$ per concludere se appartiene o meno a S.

Dimostrazione che se S è ricorsivo, allora anche $\overline{S}$ lo è.
Per definizione, la funzione caratteristica di $\overline{S}$ è definita come:

$$g_{\overline{s}}(x) = \begin{cases} 1 \text{ se } c_s(x) = 1 \\ 0 \text{ altrimenti} \end{cases}$$

Quindi in virtù del punto (B) del teorema appena dimostrato, sia S che $\overline{S}$ sono ricorsivamente enumerabili.

9.12.3 Osservazioni di interesse pratico

Si consideri l'insieme S con le seguenti caratteristiche:

- $i \in S \rightarrow f_i$ totale (cioè S) contiene solo indici di funzione **totali** e **computabili**
- f è **totale** e **computabile**, $\exists\, i \in S \mid f_i = f$ e quindi S contiene ogni i

S è quindi l'insieme di indici di funzioni totali e computabili e non è ricorsivamente enumerabile.
Questa affermazione è dimostrabile per diagonalizzazione. Quindi non esiste un formalismo ricorsivamente enumerabile che possa definire tutte le funzioni computabili e totali e solo quelle. Gli *FSA* possono definire solo alcune *(non tutte)* funzioni totali, mentre le *TM* definiscono tutte le funzioni computabili, anche quelle non totali.
Il linguaggio `C` permette di codificare qualunque algoritmo, ma anche quelli che non terminano. Non esiste nessun sottoinsieme del linguaggio `C` che definisca esattamente tutti i programmi che terminano.

La relazione tra insiemi di problemi è illustrata nella Figura 32.

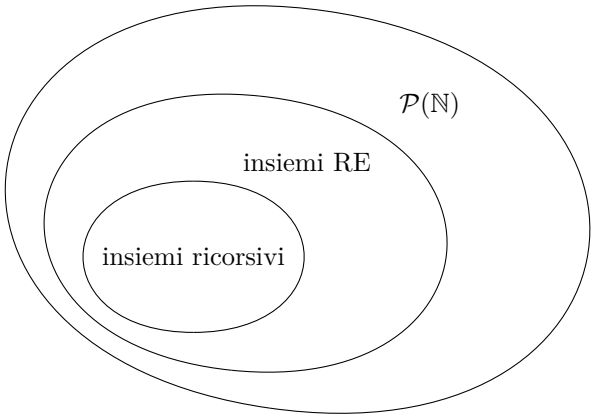


Figura 32: Relazione tra insiemi di problemi

9.13 Teoremi di *Kleene* e *Rice*

9.13.1 Teorema del punto fisso di *Kleene*

Sia t una funzione **totale** e **computabile**. Allora è sempre possibile trovare un intero p tale che:

$$f_p = f_{t(p)}$$

La funzione f_p è detta **punto fisso** di t perché t trasforma *(una macchina che calcola)* f_p in *(una macchina che calcola)* f_p stessa.

9.13.1.1 Dimostrazione del Teorma di *Kleene*

Sia u un qualsiasi numero naturale. Si definisca un *TM* che realizza la seguente procedura applicata al valore in ingresso x:

1. Calcola $z = f_u(u)$
2. Quando la computazione di $f_u(u)$ si ferma, se ciò avviene, calcola $f_z(x)$ che non è detto che sia definito

Poiché la procedura precedente è effettiva, esiste una *TM* in grado di implementarla. Inoltre è possibile costruire tale macchina e calcolare poi il suo indice $g(u)\,\forall\, u$, usando la numerazione $\mathscr{G}$ introdotta nella Sezione 9.5.

9.13.2 Teorema di *Rice*

Sia F un insieme qualunque funzioni **computabili**.
L'insieme $S = \{x \mid f_x \in F\}$ degli indici di TM che calcolano le funzioni di F è **decidibile** (e quindi **ricorsivo**) se e solo se $F = \emptyset$ oppure F è l'insieme di **tutte** le funzioni computabili.
Di conseguenza in tutti i casi non banali S **non è decidibile**.

9.13.2.1 Dimostrazione del Teorema di *Rice*

Per assurdo, si supponga che S sia ricorsivo, che $F \neq \emptyset$ e che F non coincida con l'insieme di tutte le funzioni computabili.
Si consideri ora la funzione caratteristica c_s di S definita come:

$$c_s(x) = \begin{cases} 1 \text{ se } f_x \in F \\ 0 \text{ altrimenti} \end{cases}$$

Per ipotesi assurda, si supponga che c_s sia computabile.
Sempre per ipotesi, enumerando effettivamente tutte le TM M_i si può trovare:

a) Il primo $i \in S$ per cui $f_i \in F$
b) Il primo $j \notin S$ per cui $f_j \notin S$
c) La funzione $\overline{c}_s(x)$, computabile e totale, definita come

$$\overline{c}_s = \begin{cases} i \text{ se } f_x \notin F \\ j \text{ altrimenti} \end{cases}$$

d) Per il teorema di Kleene, un $\overline{x}$ tale che $f_{\overline{c}_s(\overline{x})} = f_{\overline{d}}$

Supponendo che $\overline{c}_s(\overline{x}) = i$, per (C), segue che $f_{\overline{x}} \notin F$. Tuttavia, per (D), si ha che $f_{\overline{x}}$ e per (A) si ha che $f_i \in F$, una **contraddizione**.
Supponendo invece che $\overline{c}_s(\overline{x}) = k$, per (D) si ha che $f_{\overline{x}} = f_j$. Tuttavia, per (C) si ha che $f_{\overline{x}} = f_j$ e, per (B) si ha che $f_j \notin F$, ottenendo ancora una **contraddizione**.

9.13.2.2 Conseguenze

Il teorema di *Rice* ha un forte impatto pratico negativo.
Ad esempio, ponendo $f = \{g\}$, quindi un insieme composto dalla sola funzione g, per il teorema **non è decidibile** se una generica TM calcoli g o meno. Tuttavia, per la tesi di *Church* (Sezione 9.4) il risultato non è ristretto alle TM e al formalismo delle funzioni. Non è quindi possibile stabilire algoritmicamente se un dato algoritmo sia in grado di risolvere un determinato problema o se due programmi siano equivalenti *(cioè se calcolano la stessa funzione)*.
Inoltre, non è possibile stabilire se un algoritmo risolve un problema che goda di una qualsiasi proprietà non banale *(e quindi una proprietà che non sia appartenenza ad una categoria non esistente o coincidente con tutti i problemi risolvibili)*.

9.14 Riduzione di problemi

Un problema P' è **ridotto** ad un problema P se un algoritmo per risolvere P viene usato per risolvere P'.
Quindi P è **risolvibile** ed esiste un algoritmo che, per ogni data istanza di P':

1. **Determina** una corrispondente istanza di P
2. **Costruisce** algoritmicamente la soluzione dell'istanza di P' dalla soluzione dell'istanza di P

Formalmente:

1. Si vuole verificare se $x \in S$
2. Si è in grado di verificare se $y \in S'$

3. Se esiste una funzione t, computabile e totale tale che

$$x \in S \Leftrightarrow t(x) \in S'$$

allora è possibile determinare algoritmicamente se $x \in S$

Esempio: si supponga che il problema di cercare un elemento in un insieme sia risolvibile. Allora si può risolvere anche il problema di calcolare l'intersezione tra due sistemi, attuando una riduzione.

9.14.1 Implicazione della riduzione

Il procedimento della riduzione funziona anche al contrario:

- Si vuole sapere se è possibile risolvere $x \in S$
- Si sa che non è possibile risolvere $y \in S'$ (quindi S' *non è decidibile*)
- Se esiste una funzione t, computabile e totale, tale che

$$y \in S' \Leftrightarrow t(y) \in S$$

allora è possibile concludere che $x \in S$ non è decidibile

In conclusione, è immediato dedurre che:

- Se P^* è **risolvibile**, lo è anche P
- Se P è **irrisolvibile**, lo è anche P^*

10 Complessità del calcolo

Fin'ora è stato affrontato la determinazione della computazione di un problema, senza che venisse però analizzato il vero *"costo"* della soluzione.
Si supponga infatti di avere accesso al più potente super calcolatore al mondo. Anche sfruttando tutta la sua capacità di calcolo, alcuni problemi *(come la determinazione della partita perfetta a scacchi)* può richiedere una quantità di tempo superiore all'intera vita dell'universo *(e si dimostra che questa tesi corrisponde alla realtà).*
Di conseguenza, alcuni problemi vengono definiti **intrattabili**, perché pur esistendo un algoritmo per giungere alla sua soluzione, la sua esecuzione richiederebbe un tempo *"inaccettabile"*. L'intrattabilità costituisce una grossa limitazione alle applicazioni pratiche, in quanto esistono parecchi problemi *"interessanti"* che sono anche intrattabili.
Il concetto di **trattabilità**, è strettamente legato a quello della **complessità**, che può essere vista come la misura del *"costo"* della risoluzione: all'aumentare della prima, il secondo lieviterà. L'obiettivo di questa Sezione sarà definire in modo formale la nozione e lo studio del costo della soluzioni dei problemi, per poi essere in grado di progettare e combinare algoritmi e strutture dati in modo da realizzare soluzioni efficienti e non solo corrette.

10.1 Analisi della complessità

La tecnica che prende il nome di **analisi della complessità** consiste nel costruire strumenti per valutare la complessità di un calcolo, per poter successivamente analizzare algoritmi e strutture di nati notevoli.
Essi verranno sottoposti ad analisi quantitative su:

- Tempo necessario a completare il calcolo, cioè la **complessità temporale**
- Spazio di memoria occupato, cioè la **complessità spaziale**

L'analisi si limiterà a criteri di costo **oggettivi** e formalizzabili, quindi non terrà conto di parametri come i costi di sviluppo e *trade off* tra obiettivi contrastanti.

Per la Tesi di Church *(Sezione 9.4)*, un problema è calcolabile indipendentemente dallo strumento usato, purché esso sia *Turing completo*. Lo stesso non può essere detto riguardo alla complessità di calcolo: non è verosimile affermare che modelli di calcolo differenti impieghino lo stesso tempo *(o necessitino di egual quantità di memoria)* per eseguire programmi equivalenti. Infatti, poiché non esiste una *"Tesi di Church per la complessità"*, sarà necessario costruire uno strumento in grado di valutare la complessità temporale e spaziale di calcolo che tralasci *"considerazioni superflue"* e che sia utilizzabile per la maggioranza dei modelli di calcolo.
Contrariamente a quanto appena affermato, nonostante l'inesistenza di tale teorema, sarà tuttavia possibile correlare in modo sistematico soluzioni proposte per modelli diversi. Poiché non esiste un formalismo di calcolo perfettamente adatto a costruire il modello, l'analisi avverrà a partire dalle *TM* deterministiche.

10.2 Complessità temporale e spaziale

Sia M una *TM* **deterministica** con k nastri e sia $c_0 \vdash^* c_r$ una computazione su M. Allora è possibile definire:

- La **complessità temporale** come:
  - $T_M(x) = r$ se M termina in c_r
  - ∞ se M non termina
  - poiché M è deterministica, la computazione sarà **unica** sull'ingresso x
- La **complessità spaziale** come:
  - $S_M(x) = \sum_{j=1}^{k} \max_{i \in \{0,\dots,r\}} (|\alpha_{ij}|)$
    $\rightarrow$ $(|\alpha_{ij}|)$ indica il contenuto del j-esimo nastro alla i-esima mossa
    $\rightarrow$ la complessità è pari alla somma delle quantità massime di nastro occupate per ogni nastro
  - la complessità **spaziale** è sempre minore della complessità **temporale**
    $\rightarrow$ $\forall\, x \ \frac{S_M(x)}{k} \leq T_M(X)$

In queste definizioni tuttavia si tiene conto della natura dell'ingresso x, mentre risulta più conveniente studiare la complessità di un algoritmo in funzione della **dimensione** di x. Quindi si semplifica lo studio di $T_M(x)$ ed $S_M(x)$ ponendo $n = |x|$ *(la dimensione di x)* e studiando $T_M(n)$ e $S_M(n)$.
Questa semplificazione implica une perdita di generalità perché ingressi diversi, seppur di dimensioni uguali, potrebbero avere complessità diverse. *Formalmente*:

$$|x_1| = |x_2| \;\not\Rightarrow\; T_M(x_1) = T_M(x_2)$$
$$|x_1| = |x_2| \;\not\Rightarrow\; T_S(x_1) = T_S(x_2)$$

Per gestire la variabilità dell'ingresso così introdotta, è necessario distinguere tra:

- **Caso pessimo** $T_M(n) = \max_{n=|x|} (T_M(x))$
- **Caso ottimo** $T_M(n) = \min_{n=|x|} (T_M(x))$
- **Caso medio** $T_M(n) = \dfrac{\sum_{n=0}^{|x|} T_M(x)}{|I|^n}$
  - $\rightarrow$ somma dei tempi per parole lunghe n *diviso* il numero di parole lunghe n
  - $\rightarrow$ è una media pesata

Tipicamente, si considera il caso pessimo, perché è normalmente il più rilevante *(fornisce un limite superiore)* e l'analisi risulta più semplice che nel caso medio *(che dovrebbe tenere conto di ipotesi probabilistiche sulla distribuzione dei dati, complicandone profondamente la valutazione)*.

10.2.1 Crescita di $T_M(n)$ e $S_M(n)$

Subito ci si può accorgere di come i valori esatti di $T_M(n)$ e $S_M(n)$ non siano particolarmente utili ai fini dello studio della complessità di un particolare problema. Ciò che risulta rilevante, infatti, è il comportamento **asintotico** delle funzioni di costo, ossia quando $n \to \infty$.
Questa è una semplificazione *"aggressiva"*, perché porta a perdere la distinzione tra casi del tipo

$$T_M(n) = n^4 - 5n^2 + 2$$
$$T_M(n) = 10^{80} \cdot n^4$$

entrambi *ridotti* a n^4, seppur con andamenti radicalmente diversi per valori finiti.

Si introducono quindi tre notazioni per indicare il comportamento asintotico di una funzione:

- $\mathcal{O}$-grande: limite asintotico **superiore**
- Ω-grande: limite asintotico **inferiore**
- Θ-grande: limite asintotico sia **superiore** che **inferiore**

L'uso di questa notazione (tramite gli insiemi $\mathcal{O}$, Ω, Θ) permette di evidenziare con facilità la parte più importante di una funzione di complessità.
Questo modello tuttavia presenta delle limitazioni:

- Potrebbe non essere sufficientemente accurato per valori **piccoli** di n
  - la notazione Ω-grande è più precisa rispetto alla notazione $\mathcal{O}$-grande per i valori piccoli di n
- Algoritmi con complessità asintotica maggiore possono essere più veloce di uno a complessità minore per valori **piccoli** di n

Per ovviare a questi problemi, si faranno uso di tutte e tre le notazioni in base ai casi presi in analisi.

10.2.2 Notazione $\mathcal{O}$-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\mathcal{O}(g(n))$ è l'insieme di funzioni definito come:

$$\mathcal{O}(g(n)) = \{f(n) \,|\, \exists\, c > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq f(n) \leq c \cdot g(n)\}$$

Informalmente, le funzioni in $\mathcal{O}(g(n))$ sono *dominate* da $c \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 33.

Per brevità, è comune scrivere $f(n) = \mathcal{O}(g(n))$ al posto della più corretta notazione $f(n) \in \mathcal{O}(g(n))$.

10.2.3 Notazione Ω-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\Omega(g(n))$ è l'insieme di funzioni definito come:

$$\Omega(g(n)) = \{f(n) \,|\, \exists\, c > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq c \cdot g(n) \leq f(n)\}$$

Informalmente, le funzioni in $\Omega(g(n))$ *dominano* $c \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 34.

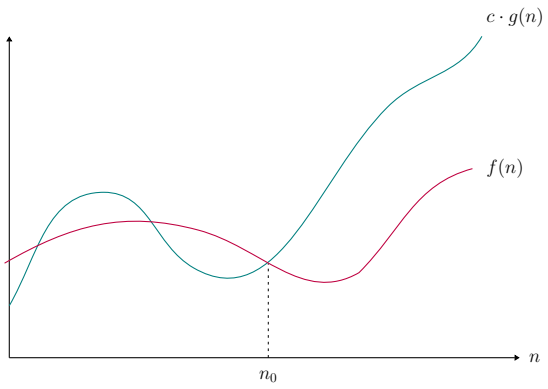


Figura 33: Notazione $\mathcal{O}$-grande

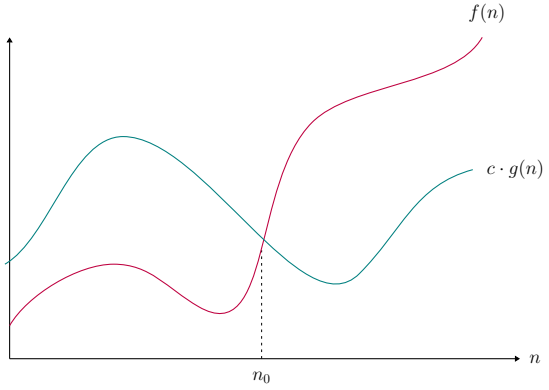


Figura 34: Notazione Ω-grande

10.2.4 Notazione Θ-grande

Definizione: siano $f(n),\ g(n)$ funzioni $\mathbb{N} \to \mathbb{R}^+$. Allora $\Theta(g(n))$ è l'insieme di funzioni definito come:

$$\Theta(g(n)) = \{f(n) \,|\, \exists\, c_1 > 0, c_2 > 0, n_0 > 0 \text{ tali che } \forall\, n > n_0,\ 0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n)\}$$

Informalmente, le funzioni in $\Theta(g(n))$ *sono comprese* tra $c_1 \cdot g(n)$ e $c_2 \cdot g(n)$ a partire da n_0. Un'illustrazione di questa relazione è rappresentata in Figura 35.

10.2.4.1 Definzioni come limiti

Se è vero che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = c, \quad c \neq 0, \quad c \neq \infty$$

allora $f(n) \in \Theta(g(n))$ e gli andamenti asintotici di $f,\ g$ differiscono per una costante moltiplicativa.

Allo stesso modo, se è vero che:

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

allora $f(n) \in \mathcal{O}(g(n)),\ f(n) \notin \Theta(g(n))$ e f cresce più velocemente di g.
Alternativamente si usa la notazione $\Theta(f(n)) < \Theta(g(n))$.

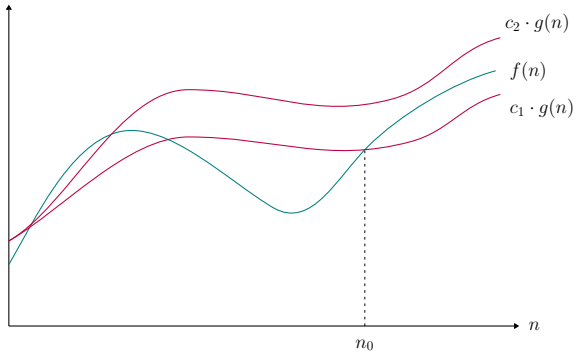


Figura 35: Notazione Θ-grande

10.2.5 Proprietà di $\mathcal{O}$, Ω, Θ

- Per definizione:
  - $f \in \Theta(g) \Leftrightarrow f \in \mathcal{O}(g) \vee f \in \Omega(g)$
- Proprietà **transitiva**
  - se $f(n) = \Theta(g(n)),\ g(n) = \Theta(h(n)) \Rightarrow f(n) = \Theta(h(n))$
  - se $f(n) = \mathcal{O}(g(n)),\ g(n) = \mathcal{O}(h(n)) \Rightarrow f(n) = \mathcal{O}(h(n))$
  - se $f(n) = \Omega(g(n)),\ g(n) = \Omega(h(n)) \Rightarrow f(n) = \Omega(h(n))$
- Proprietà **riflessiva**
  - $f(n) = \Theta(f(n))$
  - $f(n) = \mathcal{O}(f(n))$
  - $f(n) = \Omega(f(n))$
- **Simmetria**
  - $f(n) = \Theta(g(n)) \Leftrightarrow g(n) = \Theta(f(n))$
- **Simmetria trasposta**
  - $f(n) = \mathcal{O}(g(n)) \Leftrightarrow g(n) = \Omega(f(n))$

10.2.6 Complessità dei modelli deterministici di calcolo

Si vuole ora studiare la complessità dei modelli **deterministici** di calcolo studiati fin'ora. Il risultato della trattazione è illustrato in Tabella 6.

modello	*complessità spaziale*	*complessità temporale*	*note*
FSA	$S(n) \in \Theta(1)$	$T(n) \in \Theta(n)$	più precisamente $T(n) = n$
PDA	$S(n) \in \mathcal{O}(n)$	$T(n) \in \Theta(n)$	
TM a nastro singolo	$S(n) \in \Theta(n)$	$T(n) \in \Theta(n^2)$	meno efficienti di *PDA*

Tabella 6: Complessità dei modelli di calcolo

10.3 Teorema di accelerazione lineare

Quattro enunciati analoghi del Teorema di accelerazione lineare sono mostrati nelle seguenti Sezioni, seguiti ai relativi risvolti pratici.

10.3.1 Enunciato 1

> *"Se L è accettato da una TM M a k nastri in $S_M(n)$, per ogni $c \in \mathbb{R}^+$ è possibile costruire una TM M' a k nastri che accetta L con $S_{M'}(n) < c \cdot S_M(n)$"*

Il Enunciato mostra che è possibile avere una accelerazione lineare *(da qua il nome)* per un certo fattore $c \in \mathbb{R}^+$ se $0 < c < 1$.

10.3.1.1 Dimostrazione Enunciato 1

Schema della dimostrazione:

1. Si sceglie una fattore di compressione r tale che $r \cdot c > 2$
2. Per ogni alfabeto Γ_i dell'i-esimo nastro di M si costruisce Γ_i' dell'i-esimo nastro di M' assegnando un elemento per ogni $s \in \Gamma_i^r$
3. Si costruisce l'organo di controllo di M' in modo tale per cui:
   - calcoli i nuovi simboli sui nastri emulando le mosse di M spostando le testine sui nastri ogni r movimenti di M
   - memorizzi la posizione della testina arricchendo ulteriormente gli alfabeti di nastro Γ_i **oppure** arricchendo l'insieme degli stati

10.3.2 Enunciato 2

> *"Se L è accettato da una TM M a k nastri in $S_M(n)$, è possibile costruire una TM M' a 1 nastro che accetta L con $S_{M'}(n) = S_M(n)$, concatenando tutti i nastri su uno solo"*

10.3.3 Enunciato 3

> *"Se L è accettato da una TM M a k nastri in $S_M(n)$, per ogni $c \in \mathbb{R}^+$ è possibile costruire una TM M' a 1 nastro che accetta L con $S_{M'}(n) < c \cdot S_M(n)$"*

Il risultato è analogo a quello del Enunciato 2, con in aggiunta la compressione già vista nel Enunciato 1.

10.3.4 Enunciato 4

> *"Se L è accettato a una TM M a k nastri in $T_M(n)$, per ogni $c \in \mathbb{R}^+$, è possibile costruire una TM M' a $k+1$ nastri che accetta L con $T_{M'}(n) = \max(n+1,\ c \cdot T_M(n))$"*

Il risultato è analogo a quello del Enunciato 2, con in aggiunta la compressione già vista nel Enunciato 1.

10.3.4.1 Dimostrazione Enunciato 4

L'approccio della dimostrazione è analogo a quello già visto in 10.3.1.1

1. Si codifica in modo compresso i simboli dell'alfabeto di M
2. Poiché la compressione avviene a *runtime* il caso ottimale è $T_{M'}(n) = n$
3. Comprimendo r simboli in 1 nel caso pessimo servono 3 mosse di M' per emularne $r+1$ di M

10.3.5 Conseguenze pratiche

Questi teoremi portano alle seguenti conseguenze pratiche:

- Lo schema di dimostrazione usato per le *TM* vale anche per il modello di calcolatore di Von Neumann
  $\rightarrow$ È possibile avere accelerazioni lineari arbitrariamente grandi **aumentando il parallelismo fisico**

- Miglioramenti più che lineari nel tempo di calcolo possono essere ottenuti solo **cambiando algoritmo**
  → concepire ed utilizzare algoritmi **efficienti** è molto più efficiente di procedere di *forza bruta*
- Un calcolatore è in grado di eseguire operazioni aritmetiche su tipi a dimensione finita in tempo costante, mentre la *TM* richiede di propagare gli effetti al singolo `bit`, uno alla volta
  → il calcolatore opera su un alfabeto più vasto, di dimensione $|I| = 2^w$, con w dimensione della parola
  → un calcolatore può accedere direttamente ad una cella di memoria, mentre una *TM* impiega $\Theta(n)$, con n pari alla distanza di quest'ultima dalla testina

10.4 Macchina *RAM*

L'introduzione della macchina *RAM* consente di fare un passo per arrivare ad un livello di astrazione più vicino alla realtà. È infatti un modello classico, ispirato all'architettura di *Von Neumann*.
La **macchina *RAM*** è dotata di un nastro di **lettura** (`IN`) e uno di **scrittura** (`OUT`), simili a quelli di cui può essere disposta una *TM*. Il programma è cablato all'interno dell'organo di controllo tramite una serie *(finita)* di istruzioni in un certo linguaggio. Esse **non sono alterabili** durante il funzionamento della macchina. L'indice dell'istruzione da eseguire in ogni dato momento è indicato dal *program counter*.
L'accesso alla memoria avviene tramite indirizzamento diretto, in cui ogni cella (contenente un intero) viene letta o scritta dalla *unità aritmetica*. Ogni cella di memoria è quindi associata ad un intero e definita come `N`$[n]$, $n \in \mathbb{N}$. Grazie alla sua struttura, l'accesso ad una cella di memoria non implica uno scorrimento delle stesse. Le istruzioni del programma caricato nell'organo di controllo usano come primo operando sorgente o destinazione il primo elemento della memoria, `N`[0], detto **accumulatore**.

La struttura di una macchina *RAM* è mostrata nella Figura 36.

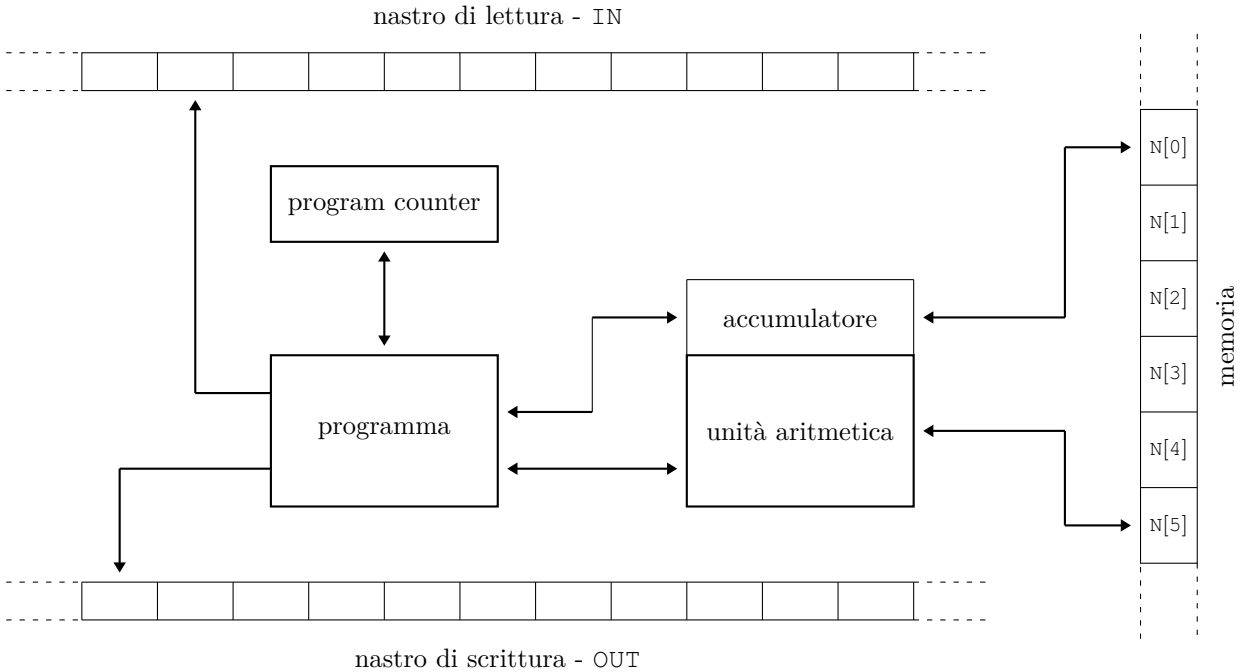


Figura 36: Struttura di una macchina *RAM*

I programmi all'interno dell'unità di controllo sono codificati tramite una semantica simile al codice `Assembly`, seppur di molto semplificato. Viene definita una specifica dei comandi (un *Instruction Set*) che permette operazioni matematiche, logiche e di salto.
Un'esempio di tale linguaggio è mostrato nella Tabella 7.

istruzione	*semantica*
`LOAD X`	N[0] ← N[X]
`LOAD= X`	N[0] ← X
`LOAD* X`	N[0] ← N[N[X]]
`STORE X`	N[X] ← N[0]
`STORE* X`	N[N[X]] ← N[0]
`ADD X`	N[0] ← N[0] + N[X]
`SUB X`	N[0] ← N[0] − N[X]
`MUL X`	N[0] ← N[0] × N[X]
`DIV X`	N[0] ← N[0] ÷ N[X]

istruzione	*semantica*
`ADD= X`	N[0] ← N[0] + X
`ADD* X`	N[0] ← N[0] + N[N[X]]
`SUB= X`	N[0] ← N[0] − X
`SUB* X`	N[0] ← N[0] − N[N[X]]
`MULT= X`	N[0] ← N[0] × X
`MULT* X`	N[0] ← N[0] × N[N[X]]
`DIV= X`	N[0] ← N[0] ÷ X
`DIV* X`	N[0] ← N[0] ÷ N[N[X]]

istruzione	*semantica*
`READ X`	N[X] ← IN
`READ* X`	N[N[X]] ← IN
`WRITE X`	OUT ← N[X]
`WRITE= X`	OUT ← X
`WRITE* X`	OUT ← N[N[X]]
`JUMP L`	PC ← L
`JZ L`	PC ← L se N[0] = 0
`JGZ L`	PC ← L se N[0] > 0
`JLZ L`	PC ← L se N[0] < 0
`HALT`	–

Tabella 7: Instruction set della macchina *RAM*

10.4.1 Criterio di costo logaritmico

L'approssimazione di costo tramite Ω, Θ e $\mathcal{O}$ è corretta ma presenta molti **limiti**.
Infatti non si tiene conto di una limitatezza della memoria: una singola parola di memoria non può sempre contenere in un solo simboli tutti gli interi usati in un determinato algoritmo. Poiché non si considera il numero di cifre necessarie a rappresentare un numero, il calcolo della complessità può risultare molto errato in certe circostanze. Questo criterio di costo prende il nome di **costante** e si contrappone al criterio di costo **logaritmico**.

Per effettuare una approssimazione più simile alla realtà, è necessario tenere conto del numero di cifre necessarie a rappresentare i numeri su cui si opera. Di conseguenza, al variare dell'ordine di grandezza degli operandi, potrà essere più o meno complicato eseguire istruzioni matematiche. Analogamente, le operazioni di accesso alla memoria *(`LOAD` e `STORE`)* avranno complessità variabile in funzione della lunghezza dei parametri.
Il criterio prende il nome di **logaritmico** perché per rappresentare un numero decimale x in base b serviranno un numero di simboli pari a $\lceil \log_b(x) \rceil$.
La complessità logaritmica delle operazioni elementari della macchina *RAM* è mostrato nell'elenco seguente, diviso per ogni categoria di operazione.

- Il costo delle operazioni **aritmetiche** e **logiche** elementari dipende dall'operazione. Ponendo $d = \log_2(i)$:
  - → **lettura** e **scrittura** (`READ`, `WRITE`) ha costo $\Theta(d)$
  - → **operazioni di memoria** (`LOAD`, `SAVE`) ha costo $\Theta(d)$
  - → **addizioni** e **sottrazioni** (`ADD`, `SUB`) ha costo $\Theta(d)$
  - → **divisioni** e **moltiplicazioni** (`MUL`, `DIV`) con **metodo scolastico** ha costo $\Theta(d^2)$
    - → usando **algoritmi più efficienti** ha costo $\Theta\left(d \log\left(d\right)\right)$
- Il costo delle operazioni di **salto** (`JZ`, `JGZ`, `JLZ`) sarà costante:
  - → sono tutte $\Theta(1)$
  - → viene sommato un *offset* di lunghezza fissa al *program counter (anch'esso di lunghezza fissa)*

10.4.2 Scelta del criterio di costo

La scelta del criterio di costo (tra logaritmico o costante) dipende da una serie di fattori. Infatti, se:

- L'elaborazione **non altera l'ordine di grandezza** dei dati di ingresso
- La memoria allocata inizialmente **non varia durante l'esecuzione** del programma
  - la memoria **non dipende** dai dati
  - di conseguenza **una singola cella è elementare** e con essa le operazioni relative
  - la dimensione di ogni singolo elemento in ingresso **non varia significativamente** nel tempo

allora il **criterio di costo costante** è adeguato. In caso contrario è indispensabile il criterio di costo logaritmico, l'unico in grado di approssimare con correttezza il costo della funzione.

La conseguenza immediata di questa differenza di complessità implica che con macchine diverse lo stesso algoritmo può dare luogo a complessità diverse. Non è possibile identificare un modello *"migliore"* tra i vari formalismi studiati e non esiste un'analogo alla Tesi di Church (Sezione 9.4), però è possibile stabilire una relazione di maggiorazione a priori tra le complessità dei diversi modelli di calcolo. Essa prende il nome di *Tesi di correlazione polinomiale.*

10.4.2.1 Relazione tra i criteri di costo

La relazione tra i due criteri di costo **non è fissa**. Infatti, non è possibile sempre affermare che sia valida la relazione:

$$C_{\log} = C_{\text{cost}} \log(n)$$

Questa uguaglianza sarà tuttavia valida nei casi in cui $C_{\text{cost}} = n$ o $C_{\text{cost}} = 1$.

10.5 Tesi di correlazione polinomiale

La Tesi di correlazione polinomiale, in analogia con la Tesi di Church, prova una relazione tra le complessità di diverse modelli di automi. Essa afferma che:

> *"Se un problema è risolvibile mediante il modello $\mathscr{M}_1$ con complessità spaziale o temporale $C_1(n)$, allora è risolvibile da un qualsiasi altro modello Turing completo $\mathscr{M}_2$ con complessità $C_2(n) \leq \pi(C_1(n))$, con π polinomio "*

10.5.1 Dimostrazione della Tesi di correlazione polinomiale

L'obiettivo di questa sezione sarà dimostrare la Tesi di correlazione polinomiale *(che d'ora in poi prenderà quindi la definizione di Teorema).*
Essa sarà strutturata in due parti, che mostreranno rispettivamente la simulazione di una *TM* tramite *RAM* e viceversa.

10.5.1.1 Simulazione di una *TM* a k nastri tramite *RAM*

Verrà ora dimostrato il funzionamento della simulazione delle azioni una *TM* a k nastri tramite macchina *RAM*. Inizialmente, sarà necessario compiere questi passi per inizializzare la memoria della *RAM*

- La *TM* viene **mappata** sulla *RAM*:
  - lo stato della *TM* verrà posto sulla prima cella di memoria della *RAM*
  - verrà usata una cella di memoria della *RAM* per ogni cella del nastro
  - la memoria rimanente viene divisa in blocchi da 4 celle
- I blocchi vengono **riempiti** con la seguente strategia:
  - Il blocco 0 conterrà le posizioni delle k testine
  - I blocchi n-esimi, $n > 0$, conterranno l'n-esimo simbolo di ognuno dei k nastri
- La *RAM* emula la lettura di un carattere sotto la testina con un accesso indiretto
  - viene usato l'indice contenuto nel blocco 0

Una volta inizializzata la memoria, sarà possibile simulare propriamente le azioni della *TM*. In particolare:

- **Lettura** dei nastri
  - lettura del blocco 0 e dello stato: $\Theta(k)$ *mosse*
  - lettura dei valori sui nastri in corrispondenza delle testine: $\Theta(k)$ *accessi indiretti*
- **Scrittura** dei nastri
  - scrittura dello stato: $\Theta(1)$

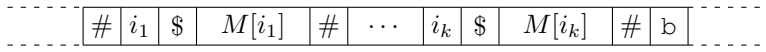


Figura 37: Inizializzazione della memoria *RAM*

- scrittura della celle dei nastri: $\Theta(k)$ *accessi indiretti*
- scrittura nel blocco 0 per aggiornare le posizioni delle k testine: ($\Theta(k)$ *mosse*

Per valutare la complessità, è necessario considerare che la *RAM* emula una mossa della *TM* con $\Theta(k)$ mosse:

- $T_{RAM}(n) = \Theta\left(T_{\mathscr{M}}(n)\right)$ per il costo **costante**
- $T_{RAM}(n) = \Theta\left(T_{\mathscr{M}}(n)\right) \log\left(T_{\mathscr{M}}(n)\right)$ per il costo **logaritmico**

10.5.1.2 Simulazione di una *RAM* tramite *TM* a k nastri

In questo paragrafo verrà dimostrata la simulazione di una *RAM* tramite *TM* a k nastri, omettendo le operazioni `MUL` e `DIV` per semplicità.

- Un nastro di memoria della *TM* viene **inizializzato** come in Figura 37
  - le celle di memoria indicizzate da i sono state coinvolte da almeno una operazione di `STORE`
  - il simbolo $ è usato come delimitatore tra indice della cella e contenuto della cella
  - il simbolo # è usato per separare tra di loro le celle di memoria
- Un secondo nastro di memoria ospita il contenuto dell'**accumulatore** `N`[0]
  - il valore viene memorizzato con codifica **binaria**
- Un terzo nastro viene utilizzato come memoria **temporanea**
  - viene usato per spostare i dati qualora sia necessario memorizzare `N`$[i_j]$
    - deve valere `N`$[i_k]$, `N`$[i_l]$, $i_k < i_j < i_l$

Una volta inizializzata la memoria, sarà possibile simulare propriamente le azioni della macchina *RAM*.
In particolare:

- L'operazione `LOAD x` avverrà come:
  1. ricerca di `x` sul nastro principiale
  2. copia di dati letti sulla cella corrispondente a `N`[0] usando il nastro di supporto
- L'operazione `STORE x` avverrà come:
  1. ricerca di `x` sul nastro principiale
  2. se esso non viene trovato, è necessario creare dello spazio usando il nastro di servizio *(se necessario)*
  3. valore di `N`[0] viene salvato in `N`[x]
- L'operazione `ADD x` avverrà come:
  - ricerca di `x`
  - copia di `N`$[x]$ sul nastro di supporto
  - calcolo della somma
  - scrittura del valore risultante in `N`[0]

In generale, la simulazione di una mossa della *RAM* richiede alla *TM* un numero di mosse n inferiore alla lunghezza del nastro principale per una costante opportuna. In simboli:

$$n \leq c \cdot \text{ lunghezza nastro principale}$$

Di conseguenza:

$$T_{RAM} = \mathcal{O}(T_{\mathscr{M}})$$

10.5.2 Lemma della Tesi di correlazione polinomiale

In seguito all'enunciazione e alla dimostrazione del Teorema di correlazione polinomiale, risulta che:

> *"Lo spazio occupato sul nastro principale è* $\mathcal{O}\left(T_{RAM}(n)\right)$*"*

10.5.2.1 Dimostrazione del Lemma

- Ogni cella i_j-esima della *RAM* occupa $\log\left(i_j\right) + \log\left(\texttt{N}[i_j]\right)$
- Ogni cella della *RAM* viene materializzata solo se la *RAM* effettua una operazione di `STORE`
- L'operazione `STORE` ha un costo per la *RAM* pari a $\log\left(i_j\right) + \log\left(\texttt{N}[i_j]\right)$
- Per riempire r celle, la *RAM* impiega $\sum_{j=1}^{r} \log\left(i_j\right) + \log\left(\texttt{N}[i_j]\right)$ unità di tempo

  $\rightarrow$ la stessa quantità di spazio occupata sul nastro

10.5.3 Conclusioni sulla Tesi di correlazione

A valle delle dimostrazioni, si può concludere che:

- La *TM* impiega **al più** $\Theta(T_{\text{RAM}}(n))$ per simulare **una mossa** della *RAM*
- Se la *RAM* ha **complessità** $T_{\text{RAM}}(n)$, essa effettua **al più** $T_{\text{RAM}}(n)$ mosse (ogni mossa costa almeno 1)
- La simulazione completa della *RAM* da parte della *TM* costa **al più** $\Theta\left((T_{\text{RAM}}(n))^2\right)$
- Il legame tra $T_{\text{RAM}}(n)$ e $T_{\text{TM}}(n)$ è **polinomiale**

Infine, è possibile fare le seguenti *osservazioni*:

- Per quanto grande possa essere il grado di un polinomio, sarà sempre inferiore ad una funzione esponenziale

  $\rightarrow$ è sufficiente confrontare l'andamento delle funzioni n^k e 2^n
- Grazie al teorema di correlazione polinomiale è possibile parlare dei problemi risolvibili in tempo e/o spazio polinomiale astraendo dalla classe del calcolatore
- *classe dei problemi trattabili in pratica* $\equiv$ *classe dei problemi risolvibili in tempo polinomiale*
  - questa classe prende il nome di $\mathcal{P}$
  - i problemi di interesse pratico che sono in $\mathcal{P}$ hanno anche grado del polinomio *"accettabile"*

11 Algoritmi

L'obiettivo di questa sezione sarà introdurre i concetti di base dei grafi e degli algoritmi, per poi iniziare ad analizzare alcuni problemi di ordinamento. Una analisi più approfondita dei grafi verrà fatta più avanti *(nella Sezione 13)*.

11.1 Introduzione ai Grafi

Un grafo è una coppia $G = \langle V, E \rangle$ in cui:

- V è un insieme finito di **nodi** (detti anche **vertici**)
- $E \subseteq V \times V$ è una relazione binaria *(costituente un insieme finito)* su V di coppie di nodi
  - gli elementi della relazione sono detti **archi**
  - se u e v sono nodi del grafo, la coppia $\langle u, v \rangle$ è un arco.

L'arco è detto **orientato** se u e v sono coppie di nodi ordinati, ovvero se esiste una relazione di ordinamento tra i due *(Figura 38a)*. In caso contrario, è detto **non orientato** *(Figura 38b)*. In questi ultimi l'ordine dei vertici negli archi non è rilevante.

(a) Grafo con arco orientato (b) Grafo con arco non orientato

Figura 38: Grafo orientato e grafo non orientato

Proprietà dei grafi:

- Se tutti gli archi di un grafo sono **orientati**, allora il grafo è detto **orientato**
  - in caso contrario, è detto **non orientato**
- Un **cammino** è una sequenza di nodi v_0, cv_1, $c \ldots$, cv_n tali che tra ogni coppia di nodi della sequenza $\langle v_i, v_{i+1} \rangle$ esiste un arco
  - i nodi v_0, cv_1, $c \ldots$, cv_n appartengono tutti al cammino
  - la lunghezza del cammino è data da n *(il numero di vertici* -1*)*
- In un grafo **non orientato**, il cammino forma un **ciclo** se $v_0 = v_n$
  - un grafo che non contiene cicli è detto **aciclico**
- Un grafo **non orientato** è connesso se tra ogni coppia di vertici esiste un cammino
- Un grafo **non orientato** e **aciclico** prende il nome di *DAG*, o *directed acyclic graph*
- $|V|$ indica il numero di **vertici**, $|E|$ denota il numero di **archi**
  - tra i due intercorre la relazione $0 \leq |E| \leq |V|^2$
- un grafo è detto **denso** se $|E| \approx |V|^2$ (il numero di lati è prossimo al numero massimo)
  - in caso contrario è detto **sparso**

11.1.1 Alberi

Un **albero** è un grafo connesso, aciclico e non orientato. È detto **radicato** se un nodo viene indicato come **radice**.
Ogni nodo del grafico è **raggiungibile** dalla radice tramite un **cammino**. Esso sarà **unico** in quanto il grafo è aciclico. Gli ultimi nodi dei cammini dalla radice sono detti **foglie**. Viene detta **altezza** dell'albero la distanza massima tra la radice e una sua foglia.
Un albero è detto **completo** se sono valide le seguenti proprietà:

- Tutte le **foglie** hanno la stessa **profondità** *(distanza dalla radice) h*
- Tutti i **nodi interni** hanno esattamente 2 figli.

Ogni **nodo** ha un **padre** *(tranne la radice)* e uno o più **figli** *(tranne le foglie)*. Più precisamente vengono chiamati:

- **Nodi interni**: tutti i nodi dei cammini tra la radice e le foglie
- **Profondità** di un nodo N: la **distanza** di N dalla radice
- **Antenato** di un nodo N: ogni nodo che **precede** N sul cammino dalla radice a N
- **Padre** di un nodo N: il nodo che **precede** immediatamente N lungo il cammino dalla radice a N
- **Figlio** di un nodo N: ogni nodo che **succede** immediatamente N lungo il cammino dalla radice a N
- **Fratelli** di un nodo N: ogni nodo che ha lo stesso padre di N

Infine un albero è detto **binario** se ogni nodo ha **al più** 2 figli.

Un esempio di albero è mostrato in Figura 39.

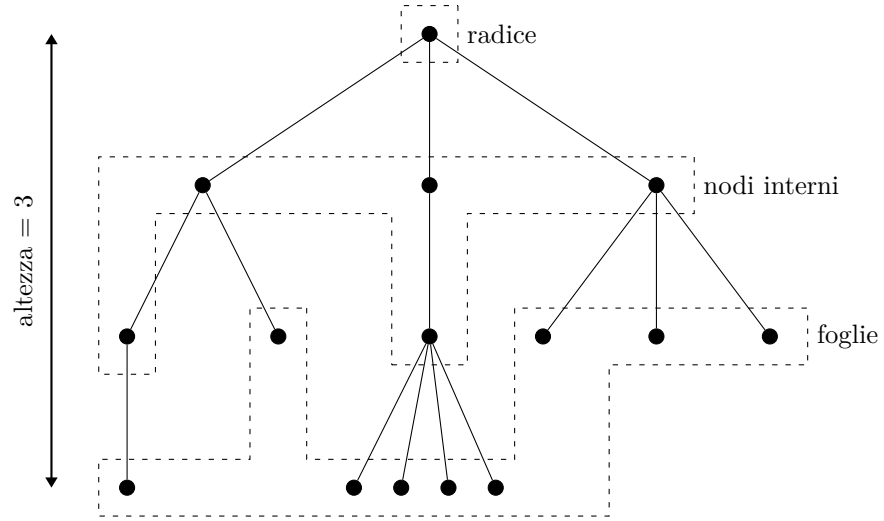


Figura 39: Esempio di albero *non binario*

11.1.1.1 Lemma di altezza dell'albero binario

Il Lemma di altezza dell'albero binario recita che:

> *"Ogni albero binario di altezza h ha un numero di foglie al più pari a* 2^h*."*

La dimostrazione è banale e avviene per induzione.

11.1.2 Heap

Uno **heap binario** è un albero binario detto **quasi completo**. Infatti tutti i suoi livelli sono completi *(su ogni livello ogni nodo ha due figli)*, tranne l'ultimo che potrebbe essere completo solo fino ad un certo punto da sinistra.
Un **max-heap** è uno heap tale che, per ogni nodo x dell'albero, il valore contenuto nel padre di x è $\geq$ del contenuto di x. Considerando i singoli nodi, $A(\lfloor i/2 \rfloor) \geq A[i]$. Nei **min-heap**, al contrario, il valore contenuto nel padre di x è $\leq$ del contenuto di x.

È sempre possibile costruire e rappresentare un heap tramite un **array**. L'albero binario che deriva da questa interpretazione è quasi completo.

Un esempio di **max-heap** e la sua corrispondenza con un array è mostrata nella Figura 40.

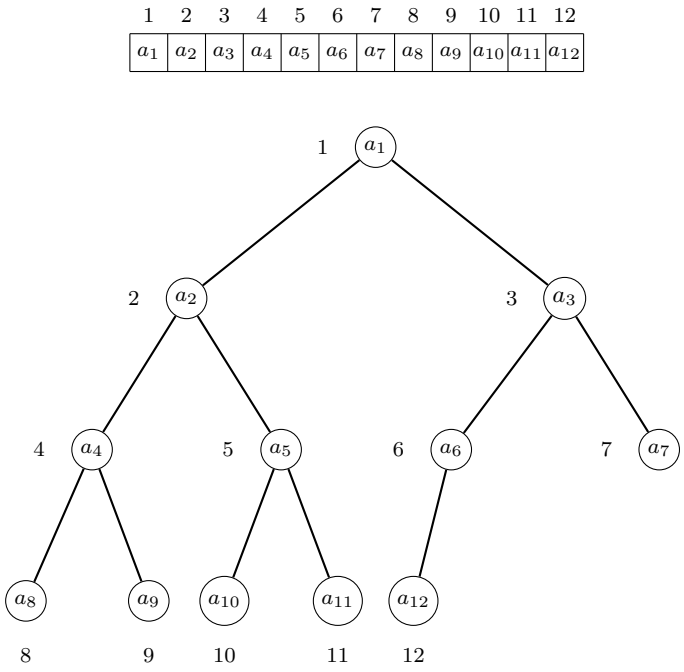


Figura 40: Esempio di max heap

11.2 Introduzione agli Algoritmi

Informalmente, un **algoritmo** è una qualsiasi procedura ben definita che prende un valore *(o un insieme di valori)* in **input** e restituisce un valore *(o un insieme di valori)* in **output**, calcolato tramite una serie ben definita di operazioni. L'algoritmo può essere visto anche come uno strumento per risolvere un particolare problema di computazione.
In particolare, si definiscono:

- **Problema**: il compito da svolgere
  - quali *output* si vuole ottenere a fronte di certi *input*
  - quali *operazioni* devono essere effettuate (la funzione che deve svolgere)
- **Algoritmo**: i passi da eseguire per risolvere un problema
  - prende gli *input* in ingresso ad un problema e li trasforma in opportuni *output*
  - è formato da una sequenza di *operazioni concrete*
  - è eseguibile da una **macchina**
  - è sempre **corretto**

Gli algoritmi possono essere descritti in diversi linguaggi *(ad esempio, in* `C`*,* `Java`*,* `Python`*, ...)*. La complessità di un algoritmo non varia significativamente in base al linguaggio in cui è implementato. Infatti essa potrà differire solo di un fattore moltiplicativo. All'interno del corso si userà lo **pseudocodice**, che assomiglia nella sintassi ad un linguaggio vero e proprio senza tuttavia esserlo.

11.2.1 Pseudocodice

La sintassi dello **pseudocodice** è la seguente:

- **Assegnamento**: `i := j`
  - è ammesso l'assegnamento multiplo: `i := j := e`
  - le variabili sono locali alla procedura
- **Cicli** e **salti** condizionali: `while`, `for`, `if-then-else`
  - la struttura a blocchi è determinata dall'indentazione

    - non si usano le parentesi
    - nei cicli **`for`** il valore dell'indice termina al valore massimo **incluso**
        - il valore indicato da **`to`** sarà uguale al valore massimo dell'indice
- **Commenti**:
    - si indicano con la notazione `// commento`
    - sono sempre a riga singola
- **Array**: la notazione è analoga a quella del `C`
    - il primo elemento può avere indice diverso da 0
        - normalmente il primo elemento dell'array `A` è `1`
    - `A[j]` è l'elemento j dell'array `A`
    - `A[i..j]` è il sotto array che inizia dall'elemento i e termina all'elemento j
- Gli **oggetti** sono strutture che raggruppano dati composti
    - possiedono degli attributi, anche detti campi
    - per accedere all'attributo `attr` dell'oggetto `x`, si usa `x.attr`
    - gli **array** sono dati composti, quindi sono assimilabili ad oggetti
      $\rightarrow$ ogni array ha un attributo `length` che indica la lunghezza dell'array
- Una **variabile** che corrisponde ad un oggetto è un puntatore all'oggetto
    - analogo a quanto avviene in `Java`
    - un puntatore che non fa riferimento ad un oggetto ha valore `NIL`
- Il **passaggio dei parametri** avviene per valore
    - la procedura invocata ottiene una copia dei parametri passati
    - passando un oggetto come parametro alla procedura, si passa il puntatore all'oggetto

Per la valutazione della complessità di algoritmi scritti in pseudocodice si userà il **criterio di costo costante** *(Sezione 10.4.1)*, poiché non verranno manipolati dati più grandi rispetto a quelli in ingresso. Come conseguenza diretta, ogni istruzione `i` di pseudocodice verrà eseguita in un tempo costante $c_{\texttt{i}}$ Grazie a questa assunzione sarà possibile trascurare il modello di calcolo della macchina che esegue lo pseudocodice *(normalmente la macchina RAM)*.
La complessità temporale sarà studiata con più dovizia di quella spaziale perché più rilevante nella realtà. Infatti, mentre la memoria ha un costo limitato, il tempo è una risorsa più significativa e costosa ai fini della risoluzione di un determinato problema.

11.2.2 Divide et Impera

La tecnica chiamata **Divide et Impera** *(dall'Inglese Divide and Conquer)* è una tecnica algoritmica molto comune. Essa si applica a problemi grossi e difficili da risolvere e si articola nei seguenti passi:

1. **Divide** - divisione del problema in problemi più piccoli da risolvere
2. **Impera** - risoluzione dei problemi più piccoli
3. **Combina** - le soluzioni vengono combinate al fine di ottenere il risultato finale

Dividendo a sufficienza i problemi, inevitabilmente si ottengono sotto problemi più facili, risolubili senza che essi debbano essere divisi ulteriormente. Questa è una tecnica di natura ricorsiva, in quanto i sotto problemi sono risolti tramite l'algoritmo stesso.

In generale, un algoritmo **divide et impera** presenta le seguenti caratteristiche:

- Il problema viene **diviso** a in sotto problemi, ognuno di dimensione $1/b$ di quello originale
- Se il sotto problema ha dimensione n sufficientemente piccola *(tale per cui $n < c$ con c costante)* allora esso può essere **risolto** in tempo costante *(quindi in $\Theta(1)$)*
- Indicando con $D(n)$ e $C(n)$ i costi di divisione e ricombinazione, è possibile esprimere il costo totale $T(n)$ come:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < c \\ D(n) + aT\left(n/b\right) + C(n) & \text{altrimenti} \end{cases}$$

11.2.3 Ricorrenze

Una **ricorrenza** è una caratteristica delle funzioni ricorsive, ossia quelle funzioni caratterizzate da:

- uno o più **casi base**
- una o più **chiamate** a sé stessa, con eventuali parametri più piccoli

Esistono 3 tecniche principali per il risolvimento delle ricorrenze:

1. **Sostituzione**
2. **Albero di ricorsione**
3. **Teorema dell'Esperto** (dall'Inglese *Master Theorem*)

Tutte queste tecniche verranno analizzate più nel dettaglio nelle seguenti Sezioni.

11.2.4 Metodo di sostituzione

Il metodo di sostituzione si articola in 2 passi:

1. **Ipotizzare** una soluzione
2. **Trovare le costanti** e dimostrare che la soluzione è corretta tramite induzione

Può essere usato per trovare il limite superiore o il limite inferiore di una ricorrenza.
Questo metodo porta ad un risultato esatto, ma può essere applicato solo laddove è inizialmente facile ipotizzare una soluzione.

11.2.5 Metodo dell'albero di ricorsione

Il metodo dell'albero di ricorsione fornisce un risultato approssimato la cui validità va provata con altre tecniche *(normalmente per sostituzione)*. Questo metodo, infatti, semplifica di molto il primo passo del metodo di sostituzione, ossia l'ipotesi della soluzione.

1. L'albero di ricorsione viene disegnato
   $\rightarrow$ ad ogni livello la somma della dimensione di ciascun nodo è pari alla dimensione della radice
2. La complessità di ciascun livello viene sommata per trovare la complessità totale

Questo tecnica è molto facile da applicare nei problemi di tipo *Divide et Impera*, come sarà mostrato nel prossimo Paragrafo.

11.2.5.1 Albero di ricorsione e tecnica *Divide et Impera*

Come annunciato precedentemente, questa tecnica si adatta particolarmente agli algoritmi che adottano la tecnica *Divide et Impera*.
Infatti, se una ricorrenza è nella forma

$$T(n) = T(a) + T(b), \quad a + b = 1,\ a \geq b$$

il corrispondente albero delle ricorrenze si esaurirà al livello k, in corrispondenza del quale il parametro a avrà valore **unitario**.
Ciò avverrà quando $n \cdot a^k = 1$, quindi per $k = \log_{1/a}(n)$. La complessità risultante andrà calcolata come la somma del costo del livello, che essendo composto da parametri unitari, è pari ad n:

$$T(n) = n \log_{1/a}(n)$$

Di conseguenza:

$$T(n) = \mathcal{O}(n \log(n))$$

Questo risultato va sempre verificato con il metodo di sostituzione.

11.2.6 Teorema dell'Esperto

Il **Teorema dell'Esperto** fornisce un metodo *"da manuale"* per risolvere le ricorrenze nella forma:

$$T(n) = aT(n/b) + f(n)$$

Il teorema richiede di memorizzare tre casi, ma permette di trovare una soluzione alla ricorrenza con relativa semplicità.

Formalmente, siano $a \geq 1,\ b > 1$ due costanti e $f(n)$ una funzione dal comportamento asintotico positivo. A partire da essi è possibile definire $T(n),\ n \in \mathbb{N}$ tramite la ricorrenza:

$$T(n) = aT(n/b) + f(n), \quad a \geq 1,\ b > 1$$

Il valore di n/b è considerato arrotondato per difetto o per eccesso (quindi rispettivamente $\lfloor n/b \rfloor$ o $\lceil n/b \rceil$).
$T(n)$ può essere limitato asintoticamente come segue:

1) Se $f(n) = \mathcal{O}\left(n^{\log_b(a)-\epsilon}\right)$ per una costante $\epsilon > 0$, allora $T(n) = \Theta\left(n^{\log_b(a)}\right)$
2) Se $f(n) = \Theta\left(n^{\log_b(a)}\right)$, allora $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$
3) Se $f(n) = \Omega\left(n^{\log_b(a)+\epsilon}\right)$ per una costante $\epsilon > 0$ e se $a \cdot f(n/b) \leq c \cdot f(n)$ per una costante $c < 1$ *(condizione di regolarità)* e dei valori di n sufficientemente grandi, allora $T(n) = \Theta(f(n))$

Informalmente, il teorema porta a confrontare $f(n)$ con la funzione $n^{\log_b(a)}$.
Se, come nel caso (1), la funzione $n^{\log_b(a)}$ è la più grande delle due, allora la soluzione è $T(n) = \Theta\left(n^{\log_b(a)}\right)$.
Se, invece, è $f(n)$ ad essere la più grande, come nel caso (3), allora la soluzione è $T(n) = \Theta(f(n))$. Infine, se come nel caso (2), le due funzioni sono della stessa dimensione, moltiplicando per un fattore logaritmico risulta che la soluzione è $T(n) = \Theta\left(n^{\log_b(a)} \log(n)\right)$.

Prima di poter applicare il teorema, è necessario tenere conto di alcuni dettagli non *ovvi*.
Nel primo caso, (1), non solo $f(n)$ deve essere più piccola di $n^{\log_b(a)}$, ma deve essere **polinomialmente più piccola**. Essa sarà, asintoticamente, più piccola di un fattore $n^{\epsilon},\ \epsilon > 0$.
Nel terzo caso, (3), non solo $f(n)$ deve essere più grande di $n^{\log_b(a)}$, ma deve essere **polinomialmente più grande** e deve soddisfare la condizione di *"regolarità"* che impone $a \cdot f(n/b) \leq c \cdot f(n)$. Questa condizione è soddisfatta dalla maggior parte delle funzioni polinomialmente limitate che possono essere incontrate.

E importante realizzare che i tre casi del teorema **non coprono tutti i possibili comportamenti** asintotici di $f(n)$. C'è infatti un *"vuoto"* tra i casi (1) e (2) quando $f(n)$ è più piccola di $n^{\log_b(a)}$ ma non **polinomialmente** più piccola. In modo analogo, la stessa problematica si presenta tra i casi (2) e (3) quando $f(n)$ è più grande di $n^{\log_b(a)}$, ma non **polinomialmente** più grande. Se la funzione $f(n)$ cade in uno di questi *"vuoti"*, o se la condizione di regolarità del caso (3) non è valida, il teorema del maestro **non è applicabile**.

11.2.6.1 Confronto polinomiale

Per cercare di chiarire il concetto di polinomialmente *"più piccolo"* e *"più grande"*, verranno ora esposti degli *esempi*.

- n è polinomialmente **più piccolo** di n^2
- $n \log(n)$ è polinomialmente **più grande** di $n^{1/2}$
- $n \log(n)$ è più grande di n ma **non polinomialmente**

In generale, una funzione $f(n)$ è polinomialmente **più grande** di una funzione $g(n)$ se vale la relazione:

$$\frac{f(n)}{g(n)} = \textit{ polinomio di grado } \geq 1$$

11.2.6.2 Casi particolari

Il Teorema dell'Esperto si semplifica notevolmente qualora $f(n)$ è una funzione $\Theta(n^k)$, con k costante:

1) Se $k < \log_b(a)$, allora $T(n) = \Theta\left(n^{\log_b(a)}\right)$
2) Se $k = \log_b(a)$, allora $T(n) = \Theta\left(n^k \log(n)\right)$
3) Se $k > \log_b(a)$, allora $T(n) = \Theta\left(n^k\right)$

Nel caso (3) la condizione aggiuntiva di regolarità è automaticamente verificata.

Un'altro caso particolare è identificato nel caso in cui la ricorrenza abbia forma

$$T(n) = \begin{cases} \Theta(1) & \text{se } n \leq m \leq h \\ \sum_{1 \leq i \leq h} a_i \cdot T(n-i) + c \cdot n^k & \text{altrimenti} \end{cases}$$

in cui i coefficienti $a_i \in \mathbb{N} \, \forall \, i$.

Ponendo $a = \sum_{i \leq i \leq h} a_i$, se:

$$\begin{cases} a = 1 \;\Rightarrow & T(n) = \mathcal{O}(n^{k+1}) \\ a \geq 2 \;\Rightarrow & T(n) = \mathcal{O}(a^n n^k) \end{cases}$$

11.3 Algoritmi di supporto

11.3.1 Operazioni sugli heap

```
PARENT(i):                LEFT(i):          RIGHT(i):
  return floor(i / 2)       return 2 * i      return 2 * i + 1
```

Ogni array `A` che rappresenta uno heap ha 2 attributi:

- `A.length`, rappresentante il numero totale di elementi dell'array
- `A.heap-size`, rappresentante il numero totale di elementi dell'heap
  - `A.heap-size` $\leq$ `A.length` $\forall$ `A`
  - solo gli elementi fino a `A.heap-size` hanno le proprietà dello heap
  - l'array potrebbe però contenere elementi dopo l'indice `A.heap-size`

11.3.2 Algoritmo `MAX-HEAPIFY`

L'algoritmo di `MAX-HEAPIFY` è una funzione che ordina i nodi di un **heap**, trasformandolo in un **max-heap**. La proprietà caratteristica dei *max-heap* è che il valore di ogni nodo dell'albero è sempre maggiore o uguale di quello dei figli.
L'algoritmo è di tipo ricorsivo ed è illustrato nel Listato 1.

```
1  MAX-HEAPIFY(A, i)
2    l := LEFT(i)
3    r := RIGHT(i)
4    if l < A.heap-size and A[l] > A[i]
5      max := l
6    else
7      max := i
8    if r < A.heap-size and A[r] > A[max]
9      max := r
10   if max != i
11     swap A[i], A[max]
12     MAX-HEAPIFY(A, max)
```

Listato 1: Costruzione di un `MAX-HEAP`

11.3.2.1 Analisi dell'algoritmo `MAX-HEAPIFY`

La complessità temporale di `MAX-HEAPIFY` è pari a

$$T = \mathcal{O}(h) = \mathcal{O}(\log(n))$$

dove h corrisponde all'altezza dell'albero, che essendo quasi completo è inferiore a $\log(n)$.

Sarebbe stato possibile giungere alla stessa soluzione usando il Teorema dell'Esperto per la ricorrenza

$$T(n) = T\left(\frac{2n}{3}\right) + \Theta(1)$$

che rappresenta il tempo di esecuzione di `MAX-HEAPIFY` nel caso pessimo.
In questo caso, infatti, l'ultimo livello dell'albero è pieno esattamente a metà. L'algoritmo viene applicato ricorsivamente sul sotto albero sinistro (contenente $\leq {}^{2n}/_{3}$, con n nodi totali).

11.3.3 Algoritmo `BUILD-MAX-HEAP`

L'algoritmo `BUILD-MAX-HEAP` permette di costruire un **max-heap** partendo da un **array**. La costruzione avviene *bottom-up*, partendo dalle foglie fino alla radice. Lo pseudocodice dell'algoritmo è mostrato nel Listato 2. Funziona grazie a due proprietà dello heap:

- Tutti gli elementi oltre l'indice `A.length / 2` sono delle **foglie**, gli altri sono dei **nodi interni**
- I sotto alberi fatti di solo foglie sono max-heap in quanto costituiti da **un solo elemento**

```
1  BUILD-MAX-HEAP(A)
2    A.heap-size := A.length
3    for i := A.length / 2 downto 1
4      MAX-HEAPIFY(A, i)
```

Listato 2: Algoritmo `BUILD-MAX-HEAP`

11.3.3.1 Analisi dell'algoritmo `BUILD-MAX-HEAP`

Il costo dell'algoritmo `BUILD-MAX-HEAP`, a priori, potrebbe risultare pari a $\mathcal{O}(n \log(n))$. Tuttavia, questo limite non fornisce una precisione sufficiente.
Infatti, si osserva che:

- L'altezza di un albero quasi completo di n nodi è $\lfloor \log_2(n) \rfloor$
- Definendo come *"altezza di un nodo di uno heap"* la lunghezza del cammino più lungo che porta ad una foglia, il costo dell'algoritmo invocato su un nodo di altezza h è $\mathcal{O}(h)$
- Il numero massimo di nodi di altezza h di uno heap è $\left\lceil \frac{n}{2^{h+1}} \right\rceil$ volte ad ogni altezza h

Grazie a queste proprietà, il costo di computazione dell'algoritmo `BUILD-MAX-HEAP` è pari a:

$$\sum_{n=0}^{\lfloor \log_2(n) \rfloor} \left\lceil \frac{n}{n^{h+1}} \right\rceil \mathcal{O}(h) = \mathcal{O}\left(n \sum_{n=0}^{\lfloor \log_2(n) \rfloor} \frac{h}{2^h} \right) = \mathcal{O}(n)$$

Il passo finale della soluzione deriva dall'uguaglianza

$$\sum_{h=0}^{\infty} \frac{1}{2^h} = \frac{1}{1 - {}^{1}/_{2}}$$

come sommatoria di una serie geometrica.

11.4 Problemi di ordinamento

L'**ordinamento** degli elementi di una sequenza è un problema molto comune e fornisce un classico esempio di problema risolvibile tramite algoritmi. Esistono parecchi algoritmi di ordinamento *(insertion sort, selection sort, bubble sort, ...)*, ognuno con la propria complessità.

Nelle prossime Sezioni alcuni di essi verranno analizzati in dettaglio.

11.4.1 INSERTION-SORT

L'algoritmo INSERTION-SORT è uno dei più semplici algoritmi di ordinamento.
Idea dell'algoritmo: per ogni elemento x di una sequenza A di elementi, partendo dalla prima posizione, inserisce l'elemento viene prelevato e l'array viene scorso fino a quando non viene trovata una posizione valida in cui inserire x.
L'array viene quindi ordinato senza sul posto, senza bisogno di spazio di memoria ausiliario.

```
1  INSERTION-SORT(A)
2    for j := 2 to A.length
3      key := A[j]
4      i := j - 1
5      while i > 0 and A[i] > key
6        A[i + 1] := A[i]
7        i := i - 1
8      A[i + 1] := key
```

riga	*costo*	*ripetizioni*
2	c_1	n
3	c_2	$n-1$
4	c_3	$n-1$
5	c_4	$\sum_{j=2}^{n} t_j$
6	c_5	$\sum_{j=2}^{n}(t_j-1)$
7	c_6	$\sum_{j=2}^{n}(t_j-1)$
8	c_7	$n-1$

11.4.1.1 Analisi dell'algoritmo INSERTION-SORT

Il tempo di esecuzione di INSERTION-SORT(A) è:

$$T(n) = c_1 \cdot n + c_2 \cdot (n-1) + c_3 \cdot (n-1) + c_4 \cdot \sum_{j=2}^{n} t_j + c_5 \cdot \sum_{h=2}^{n} (t_j - 1) + c_6 \cdot \sum_{h=2}^{n} (t_j - 1) + c_7 \cdot (n-1)$$

È quindi possibile identificare i casi ottimi e pessimi per poter calcolare l'effettiva complessità temporale:

- Caso **ottimo**:
  - gli elementi sono già ordinati, $t_2 = \ldots = t_n = 1$
  - $T(n) \approx an + b \;\Rightarrow\; T(n) = \Theta(n)$
  - si può anche affermare che $T(n) = \Omega(n)$ perché il limite inferiore è funzione di $\Theta(n)$
- Caso **pessimo**:
  - gli elementi sono già ordinati ma in ordine decrescente $t_2 = 2,\ t_3 = 3,\ \ldots$
  - $T(n) \approx an^2 + bn + c \;\Rightarrow\; T(n) = \Theta(n^2)$
  - si può anche affermare che $T(n) = \mathcal{O}(n^2)$ perché il limite superiore è funzione di $\Theta(n^2)$

11.4.2 MERGE-SORT

Idea dell'algoritmo: Se l'array da ordinare ha meno di 2 elementi, allora è ordinato per definizione. Altrimenti:

- l'array viene diviso in 2 sotto array, ognuno con la metà degli elementi di quello originario
- i 2 sotto array vengono ordinati applicando di nuovo l'algoritmo
- grazie all'algoritmo MERGE i 2 sotto array vengono combinati
- gli elementi dell'array ottenuto sono ordinati

Il MERGE-SORT è un algoritmo ricorsivo e adotta la tecnica **divide ed impera**. Per completare l'algoritmo, infatti, è necessario definire un sotto algoritmo MERGE che combina le soluzioni dei problemi divisi.
Idea dell'algoritmo MERGE:

1. Partendo dall'inizio dei 2 sotto array, viene cercato il minimo dei 2 elementi correnti
2. Il minimo viene inserito all'inizio dell'array da restituire
3. L'inizio dell'array da cui è stato prelevato il minimo viene avanzato di 1
4. L'esecuzione riprende dal passo 1

```
1  MERGE-SORT(A, p, R)
2    if p < r
3      q := floor((p + r) / 2)
4      MERGE-SORT(A, p, q)
5      MERGE-SORT(A, q + 1, r)
6      MERGE(A, p, q, r)
```

Listato 3: Pseudocodice dell'algoritmo `MERGE-SORT`

```
1  MERGE(A, p, q, r)
2    n1 := q - p + 1
3    n2 := r - q
4    alloca L[1 .. n1 + 1]
5    alloca R[1 .. n2 + 1]
6    for i := 1 to n1
7      L[i] := A[p + i - 1]
8    for j := 1 to n2
9      R[j] := A[q + j]
10   L[n1 + 1] := ∞
11   R[n2 + 1] := ∞
12   i := 1
13   j := 1
14   for k := p to r
15     if L[i] <= R[j]
16       A[k] := L[i]
17       i := i + 1
18     else
19       A[k] := R[j]
20       j := j + 1
```

Listato 4: Pseudocodice dell'algoritmo `MERGE`

riga	*costo*	*ripetizioni*
2	c_1	1
3	c_2	1
4	$\Theta(\mathtt{n})$	1
5	$\Theta(\mathtt{n}_1)$	$\mathtt{n}_1$
8	$\Theta(\mathtt{n}_2)$	$\mathtt{n}_2$
10	c_3	1
11	c_4	1
12	c_5	1
13	c_7	1
14	$\Theta(\mathtt{n})$	$\mathtt{r}-\mathtt{p}$
15	c_8	1
16	c_9	1
17	c_{10}	1
19	c_{11}	1
20	c_{12}	1

- `A` è l'array di **input** di dimensione n *(da ordinare)*
- `p` è la posizione iniziale del primo elemento
- `r` è la posizione dell'ultimo elemento

11.4.2.1 Analisi dell'algoritmo `MERGE`

Nell'algoritmo `MERGE` prima vengono copiati gli elementi dei 2 sotto array `A[p .. q]` e `A[q+1 .. r]` in 2 array temporanei L e R e poi vengono fusi in `A[p .. r]`. Per non dover controllare se L e R sono vuoti si usa un valore particolare (∞), più grande di ogni possibile valore, in ultima posizione negli array.
La dimensione dei dati in input è $\mathtt{n} = \mathtt{r} - \mathtt{p} + 1$ ed è composto da 3 cicli **`for`**:

- 2 cicli di inizializzazione (linee 10 e 11) per assegnare i valori a L e R
  1. il primo è eseguito n_1 volte, con $\Theta(n_1) = \Theta(\mathtt{q} - \mathtt{p} + 1) = \Theta(n/2) = \Theta(n)$
  2. il secondo è eseguito n_2 volte, con $\Theta(n_2) = \Theta(\mathtt{r} - \mathtt{q}) = \Theta(n/2) = \Theta(n)$

Quindi combinando i due passi, $T(n) = \Theta(n)$

11.4.2.2 Analisi delle ricorrenze dell'algoritmo `MERGE-SORT`

Siano:

- $a = b = 2$ perché il problema viene diviso in due sotto problemi di dimensione *quasi* analoga
- $D(n) = \Theta(1)$ perché la divisione del problema ha costo costante
- $C(n) = \Theta(n)$ perché è necessario scorrere tutti gli elementi dei sotto array per poterli ricombinare

La ricorrenza totale corrisponderà quindi a:

$$T(n) = \begin{cases} \Theta(1) & \text{se } n < 2 \\ \Theta(1) + 2T\left(\frac{n}{2}\right) + \Theta(n) & \text{altrimenti} \end{cases}$$

Per essere più precisi, la ricorrenza dovrebbe essere $T(\lfloor n/2 \rfloor) + T(\lfloor n/2 \rfloor)$ e non $2T(n/2)$, ma l'approssimazione è sufficiente perché non influisce sul comportamento asintotico della funzione.

Riscrivendo la ricorrenza dell'algoritmo senza fare uso delle notazioni *Theta*, risulta che:

$$T(n) = \begin{cases} c & \text{se } n < 2 \\ 2T\left(\frac{n}{2}\right) + cn & \text{altrimenti} \end{cases}$$

Per un appropriata costante c.
La ricorrenza può essere visualizzata tramite un **albero di ricorsione**, scegliendo per semplicità il caso in cui n è una potenza di 2 *(e quindi ogni ramo viene diviso in due sotto rami senza problemi di arrotondamento)*.
Una rappresentazione dell'albero di ricorsione è mostrata in Figura 41.

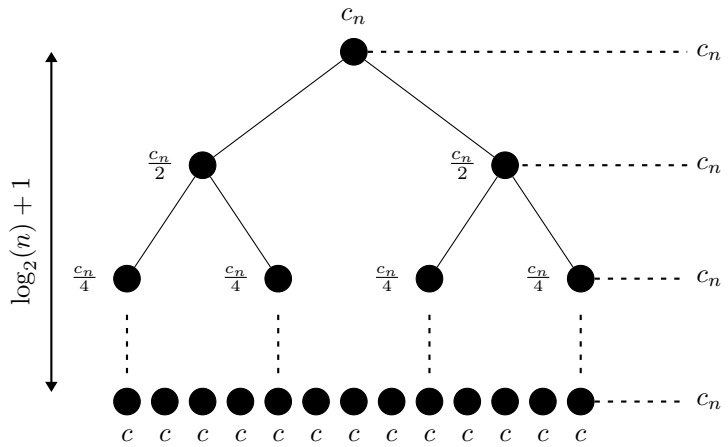


Figura 41: Albero di ricorsione per l'algoritmo `MERGE-SORT` su un array di dimensione $n = 2^k$.

Sommando i costi dei vari livelli si ottiene:

$$T(n) = cn \log(n) + cn \quad \Rightarrow \quad T(n) = \Theta(n \log(n))$$

Analogamente sarebbe stato possibile applicare il Teorema dell'Esperto, giungendo alla stessa soluzione. Infatti:

$$T(n) = 2T(n/2) + \Theta(n)$$

dove $a = b = 2$, $f(n) = n$, come nel caso (2) del Teorema.
Di conseguenza, $n^{\log_b(a)} = n^1 = n$ e la complessità temporale è quindi:

$$T(n) = \Theta(n \log(n))$$

11.4.3 HEAPSORT

L'algoritmo HEAPSORT, *come suggerisce il nome*, sfrutta le proprietà degli heap *(Sezione 11.1.2)* per ordinare gli elementi di un array. Questo algoritmo farà uso degli algoritmi *di supporto* BUILD-MAX-HEAP *(Sezione 11.3.3)* e MAX-HEAPIFY *(Sezione 11.3.2)*.

Idea dell'algoritmo: Ad ogni ciclo viene preso l'elemento più grande *(primo dell'array, in quanto max-heap)* in fondo alla parte di array ancora da ordinare *(corrispondente allo heap)*. La dimensione dell'heap viene decrementata di 1 e viene ricostruito il max-heap *(tramite MAX-HEAPIFY)* usando come radice l'ultima foglia a destra dell'ultimo livello *(l'ultimo elemento dell'array)*.

```
1 HEAPSORT(A)
2   BUILD-MAX-HEAP(A)
3   for i := A.length downto 2
4     swap A[1], A[i]
5     A.heap-size := A.heap-size - 1
6     MAX-HEAPIFY(A, 1)
```

Listato 5: Pseudocodice dell'algoritmo HEAPSORT

Dove A è l'array di **input** di dimensione n *(da ordinare)*

11.4.3.1 Analisi dell'algoritmo HEAPSORT

La complessità dell'algoritmo HEAPSORT è data dalla somma delle seguenti ricorrenze:

- BUILD-MAX-HEAP è invocato una volta, con costo $\mathcal{O}(n)$
- MAX-HEAPIFY è invocato $n-1$ volte, con costo $\mathcal{O}(\log(n))$

Quindi la complessità totale sarà:

$$T(n) = \mathcal{O}(n \log(n))$$

11.4.4 QUICKSORT

L'algoritmo QUICKSORT è un algoritmo in stile *Divide et Impera*, che ordina sul posto. Nel caso pessimo ha complessità $\Theta(n^2)$, superiore all'HEAPSORT, ma nel caso medio ha complessità $\Theta(n \log(n))$.
Idea dell'algoritmo, dato un array A[p..r] da ordinare:

- **Divide**: A viene diviso in 2 sotto array A[p..q-1] e A[q+1..r]
  - q è il minimo elemento del sotto array A[p..q-1] (A[n] $\leq$ A[q] $\forall$ n $\in$ [p, q-1])
  - q è il massimo elemento del sotto array A[q+1..r] (A[n] $\geq$ A[q] $\forall$ n $\in$ [q+1, r])
- **Impera**: i sotto array A[p...q-1] e A[q+1...r] vengono ordinati utilizzando QUICKSORT
- **Combina**: l'array A[p..r] è già ordinato

La parte *"complicata"* da implementare riguarda la partizione: essa viene effettuata tramite l'algoritmo PARTITION, che avrà l'incarico di scegliere un elemento attorno al quale gli elementi ruoteranno, detto pivot. Proprietà dell'elemento pivot:

- Tutti gli elementi **minori** di pivot vengono posti nel sotto array **sinistro**
- Tutti gli elementi **maggiori** di pivot vengono posti nel sotto array **destro**

Il pivot può essere scelto **a piacere**: esso normalmente è scelto come il primo o l'ultimo elemento dell'array. Viene dimostrato che prendere un elemento casuale porta alla stessa complessità temporale.

```
1 QUICKSORT(A, p, r):
2   if p < r:
3     q := PARTITION(A, p, r)
4     QUICKSORT(A, p, q - 1)
5     QUICKSORT(A, q + 1, r)
```

Listato 6: Pseudocodice di QUICKSORT

```
1 PARTITION(A, p, r):
2   x := A[r]
3   i := p - 1
4   for j := p to r - 1
5     if A[j] <= x
6         i := i + 1
7         swap A[i], A[j]
8   swap A[i + 1], A[r]
9   return i + 1
```

Listato 7: Pseudocodice di PARTITION

- A è l'array di **input** di dimensione n *(da ordinare)*
- p è la posizione iniziale del primo elemento
- r è la posizione dell'ultimo elemento

11.4.4.1 Analisi di QUICKSORT

La complessità di PARTITION è $\Theta(n),\ n = r - p + 1$.

Il tempo di esecuzione di QUICKSORT dipende da come viene partizionato l'array:

- Se ogni volta uno dei 2 sotto array è vuoto mentre l'altro contiene $n - 1$ elementi si ha il caso **pessimo**:
  - la ricorrenza è $T(n) = T(n-1) + \Theta(n)$
  - risolvendo per sostituzione o tramite il Teorema dell'Esperto dimostra che la soluzione è $\Theta(n^2)$
  - questo caso si può verificare quando l'array è già ordinato in senso decrescente
- Se ogni volta i due sotto array hanno una dimensione pari a $n/2$ si ha il caso **ottimo**:
  - la ricorrenza è $T(n) = 2T(n/2) + \Theta(n)$
  - risolvendo si dimostra che la soluzione è $\Theta(n \log(n))$

Se la proporzione di divisione fosse diversa da $1/2$ e $1/2$, la complessità rimarrebbe la stessa.

Il caso **medio** va analizzato in modo diverso. Qualora le partizioni fossero bilanciate (e quindi i sotto array fossero di pari lunghezze $n/2$), infatti, la complessità sarebbe più bassa.
Più nel dettaglio:

- Il costo di una divisione **bilanciata**: $\Theta(n)$
- Il costo di una divisione **non bilanciata**: $\Theta(n)$

Quindi il costo di una catena di divisioni è la stessa e quindi l'algoritmo sarà $\Theta(n \log(n))$.

Lo stesso risultato sarebbe stato raggiunto anche applicando il Teorema dell'Esperto.
Nel caso ottimo, la ricorrenza è $T(n) \leq 2T(n/2) + \Theta(n)$. La soluzione sarà quindi pari a:

$$T(n) = \mathcal{O}(n \log(n))$$

Nel caso pessimo, *in cui l'algoritmo produce una divisione* 9 *a* 1, la ricorrenza è $T(n) \leq T(9n/10) + T(n/10) + c \cdot n$.
La soluzione sarà quindi pari a:

$$T(n) = \mathcal{O}\left(n^2\right)$$

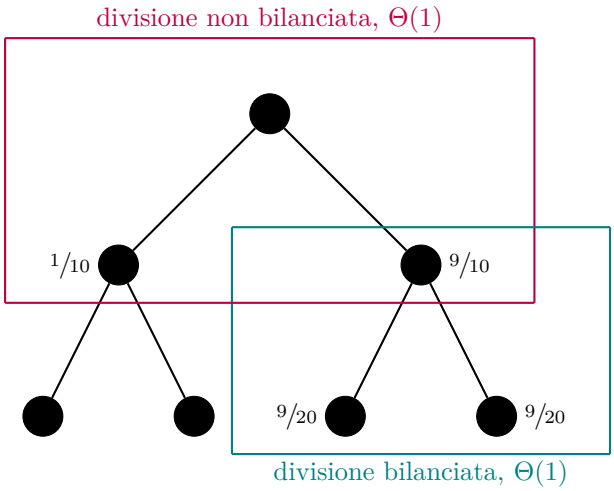


Figura 42: Partizione bilanciata e sbilanciata

11.4.5 COUNTING-SORT

L'algoritmo COUNTING-SORT è un algoritmo di ordinamento che richiede che tutti i valori in ingresso $a_0, \ldots, a_k$ corrispondano a tutti i numeri interi compresi tra 0 ed una certa costante m, ammettendo ripetizioni *(quindi $k \geq m$)*.
Idea di dell'algoritmo: se nell'array ci sono m_e valori più piccoli di un certo elemento e di valore v_e, nell'array ordinato quest'ultimo sarà in posizioni $m_e + 1$. Quindi per ordinare l'array basta contare quante copie dello stesso valore v_e sono presenti all'interno dello stesso: questa informazione viene usata per determinare, per ogni elemento e, quanti elementi sono più piccoli di e. Infine è necessario tenere conto degli elementi ripetuti.

```
1  COUNTING-SORT(A, B, K):
2    for i := 1 to k
3      C[i] := 0
4    for j := 1 to A.length
5      C[A[j]] := C[A[j]] + 1
6    for i := 2 to k
7      C[i] := C[i] + C[i - 1]
8    for j := A.length downto 1
9      B[C[A[j]]] := A[j]
10     C[A[j]] := C[A[j]] - 1
```

Listato 8: Pseudocodice algorimo COUNTING-SORT

In cui i parametri sono:

- A è l'array di **input** di dimensione n *(da ordinare)*
- B è l'array di **output** di dimensione n *(ordinato)*
- k è il valore massimo tra i valori di A

11.4.5.1 Analisi dell'algoritmo COUNTING SORT

La complessità dell'algoritmo COUNTING-SORT è data dai 4 cicli **for**:

- Il **for** del ciclo che inizia alla riga 2 ha complessità $\Theta(k)$.
- Il **for** del ciclo che inizia alla riga 4 ha complessità $\Theta(n)$.
- Il **for** del ciclo che inizia alla riga 7 ha complessità $\Theta(k)$.
- Il **for** del ciclo che inizia alla riga 10 ha complessità $\Theta(n)$.

La complessità totale sarà quindi $\Theta(n+k)$.
Se $k = \mathcal{O}(n)$, allora l'algoritmo `COUNTING-SORT` ha complessità $\mathcal{O}(n)$ *(lineare)*. `COUNTING-SORT` è più veloce di `MERGE-SORT` e `HEAPSORT` perché fa delle forti assunzioni sulla distribuzione dei valori da ordinare *(assumendo che ogni valore sia* $\leq k$*)*.
In caso contrario, la complessità dell'algoritmo `COUNTING-SORT` sarà superiore a quella degli altri algoritmi.

Si noti che è possibile ottenere una versione semplificata dell'algoritmo senza fare uso dell'array `B`, perdendo tuttavia stabilità. Se nell'array da ordinare ci fossero più elementi con lo stesso valore, questi appariranno nell'array ordinato mantenendo il loro ordine iniziale *(proprietà garantita dall'uso dell'array `B`)*.
Questa non è una proprietà particolarmente interessante nell'ordinare i numeri, ma ordinando oggetti tramite un loro attributo chiave.

11.4.6 Limite inferiore della complessità nei problemi di ordinamento

Nelle Sezioni precedenti sono stati introdotti ed analizzati svariati algoritmi di ordinamento che sono in grado di ordinare n numeri in tempo $\mathcal{O}(n \log(n))$. Questi algoritmi condividono una importante proprietà: *l'ordine da essi determinato dipende solo da confronti tra gli elementi di input.* Essi prendono quindi il nome di **algoritmi di ordinamento comparativi**.
Questi algoritmi possono essere visti in modo astratto tramite un **albero di decisione**, un albero binario che rappresenta i confronti tra gli elementi fatti da un determinato algoritmo di ordinamento con un certo input. L'esecuzione dell'algoritmo corrisponde al tracciamento di un percorso dalla radice alla foglia contenente gli elementi in ordine.
Poiché ogni algoritmo deve poter creare le $n!$ permutazioni, incluse quelle che possono apparire più volte, l'albero avrà un numero di foglie superiore a $n!$.

Un esempio di albero di decisione è mostrato nella Figura 43. Un nodo con la notazione $i : j$ indica la permutazione tra gli elementi i e j. Il percorso verde mostra la decisione dell'algoritmo di ordinamento quando avendo come input $\langle a_1 = 9, a_2 = 6, a_3 = 2\rangle$. La permutazione $\langle 3, 2, 1\rangle$ nella foglia verde indica l'ordinamento corretto. Poiché ci sono 3 elementi, ci saranno $3! = 6$ foglie.

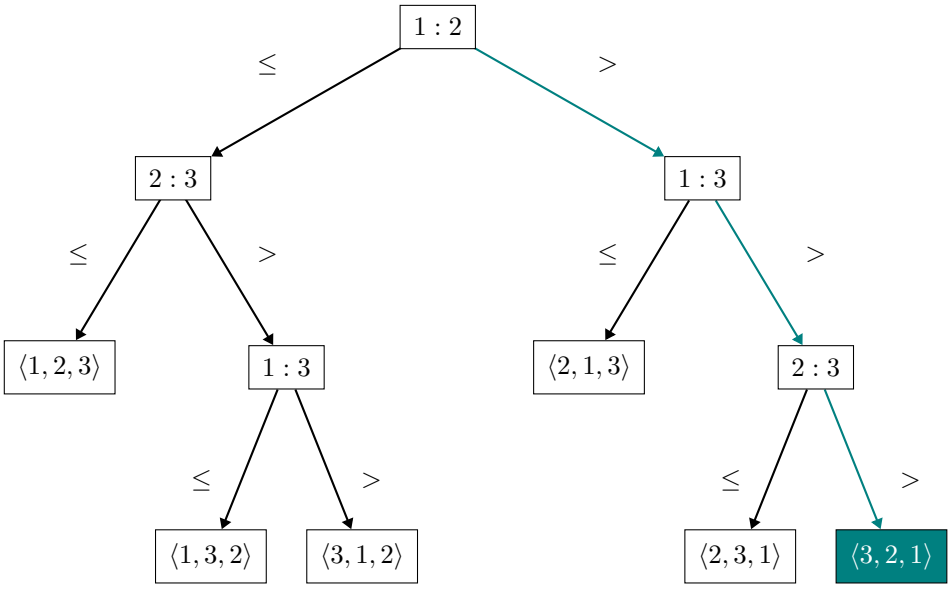


Figura 43: Albero di decisione

Poiché è possibile costruire un albero di decisione per ogni dato n, tenendo conto del Lemma di altezza dell'albero binario, *(Paragrafo 11.1.1.1)*, si ottiene il seguente teorema:

> *"Ogni albero di decisione di ordinamento di n elementi ha altezza $\Omega(n \log n)$."*

11.4.6.1 Dimostrazione del limite inferiore della complessità

Sia f il numero di foglie dell'albero di decisione di ordinamento di n elementi. Come visto prima, $f \geq n!$. Per il Lemma sarà valida la relazione:

$$n! \leq f \leq 2^h \;\Rightarrow\; 2^h \geq n!$$

Quindi il limite inferiore della complessità è:

$$h \geq \log(n!) \geq \log\left(\left({}^{n}/_{2}\right)^{n/2}\right) = \frac{n}{2}\log\left(\frac{n}{2}\right) = \Omega(n\log(n))$$

come volevasi dimostrare.

Alternativamente sarebbe stato possibile ottenere lo stesso risultato tramite l'approssimazione di Stirling:

$$n! > \left(\frac{n}{e}\right)^n \Rightarrow h \geq \log\left(\left({}^{n}/_{e}\right)^n\right) = n\log\left({}^{n}/_{e}\right) = n\log(n) - n\log(e) = \Omega(n\log(n))$$

12 Strutture dati

Le strutture dati sono usate per **contenere oggetti**. Essi rappresentano *collezioni (o insiemi dinamici)* di oggetti.
Normalmente, gli oggetti di una struttura dati presentano:

- Una **chiave**, usata per indicizzare l'oggetto
- Dei **dati satellite**, ognuno associato ad una delle chiavi

Sono definiti 2 tipi di operazioni sulle strutture dati:

- operazioni che **modificano** la collezione
- operazioni che **interrogano** la collezione

Le operazioni più usate, in pseudocodice, sono:

- `SEARCH(S, k)` - *interrogazione*
  - restituisce il **riferimento all'oggetto** `x` corrispondente alla chiave `K` nella collezione `S`
  - se nessun elemento corrisponde alla chiave, restituisce `NIL`
- `INSERT(S, x)` - *modifica*
  - **inserisce** l'oggetto `x` nella collezione `S`
- `DELETE(S, x)` - *modifica*
  - **elimina** l'oggetto `x` dalla collezione `S`
- `MINIMUM(S)` - *interrogazione*
  - restituisce l'oggetto nella collezione `S` con la **chiave più piccola**
- `MAXIMUM(S)` - *interrogazione*
  - restituisce l'oggetto nella collezione `S` con la **chiave più grande**
- `SUCCESSOR(S, x)` - *interrogazione*
  - restituisce l'oggetto nella collezione `S` con la **chiave successiva** alla chiave dell'oggetto `x`
  - necessita di una qualche forma di ordinamento all'interno di `S`, definita dalla struttura dati
- `PREDECESSOR(S, x)` - *interrogazione*
  - restituisce l'oggetto nella collezione `S` con la **chiave precedente** alla chiave dell'oggetto `x`
  - necessita di una qualche forma di ordinamento all'interno di `S`, definita dalla struttura dati

12.1 Pila - *stack*

Una **pila** è una collezione di oggetti sulla quale è possibile compiere le seguenti operazioni:

- **Controllare** se è vuota
- **Inserire** un elemento nella collezione - operazione di `PUSH`
- **Cancellare** l'elemento **in cima** alla collezione, restituendolo - operazione di `POP`

Una pila è gestita quindi con una politica *LIFO (last-in, first-out)*: l'elemento che viene cancellato è l'**ultimo** ad essere stato inserito. Eseguendo una operazione `PUSH` di un elemento `e` su una pila `S`, seguita immediatamente da una `POP` di nuovo su `S`, l'elemento restituito dall'ultima operazione sarà di nuovo `e`.

La pila può essere implementata come un `array`. Infatti, se la pila può contenere al massimo n elementi, allora è possibile allocare un array di dimensione n e usarlo come pila. Per tenere traccia dell'indice dell'elemento che è stato inserito per ultimo viene introdotto un attributo chiamato `top` che indica l'indice dell'ultimo elemento inserito.
Usando questa implementazione, considerando una pila `S`, se:

- `S.top = t`, allora `S[1], S[2], ..., S[t]` contengono tutti gli elementi della pila

- `S.top = 0`, allora la pila è **vuota** e nessun elemento può essere **cancellato**
- `S.top = n`, allora la pila è **piena** e nessun elemento può essere **inserito**

Lo pseudocodice di queste funzioni è mostrato nei Listati 9 e 10.

```
1  PUSH(S, x)
2    if S.top = S.length
3      error "overflow"
4    else
5      S.top := S.top + 1
6      S[S.top] := x
```

Listato 9: `PUSH` in pila

```
1  POP(S)
2    if S.top = 0
3      error "underflow"
4    else
5      S.top := S.top - 1
6    return S[S.top + 1]
```

Listato 10: `POP` in pila

La complessità temporale delle operazioni è $\mathcal{O}(1)$.

12.2 Coda - *queue*

Una **coda** è una collezione di oggetti sulla quale è possibile compiere le seguenti operazioni:

- **Controllare** se è vuota
- **Inserire** un elemento della collezione - operazione di `ENQUEUE`
- **Cancellare** un elemento **dal fondo** della collezione, restituendolo - operazione do `DEQUEUE`

Una coda è gestita quindi con una politica *FIFO (first-in, first-out)*: l'elemento che viene cancellato è il **primo** ad essere stato inserito.

La coda può essere implementata come un array. Infatti, se la coda può contenere al massimo n elementi, allora è possibile allocare un array di dimensione $n+1$ e usarlo come coda. Per tenere traccia degli indici del primo e dell'ultimo elemento inserito *(rispettivamente)*, vengono introdotti due attributi chiamati `head` e `tail`. Usando questa implementazione, gli attributi della coda `Q` indicheranno:

- `Q.head` l'indice dell'elemento da più da più tempo nell'array
- `Q.tail` l'indice in cui il prossimo elemento dovrà essere inserito
- `Q.tail - 1` l'indice dell'elemento da meno da meno tempo nell'array

La presenza di uno spazio aggiuntivo nell'array rispetto alla dimensione effettiva della coda è dovuto alla gestione dell'elemento aggiunto quando essa è piena. In tal caso i puntatori `head` e `tail` combacerebbero, facendo collassare la struttura. Questo non è l'unico stratagemma impiegato per risolvere il problema ma esistono strategie alternative.

Inoltre, per poter scrivere lo pseudocodice, è necessario analizzare ulteriormente il funzionamento della coda:

- Gli elementi della coda `Q` avranno indici `Q.head`, `Q.head + 1`, ..., `Q.tail - 1`, `Q.tail`
- La coda ha funzionamento circolare:
  - se `Q.tail` = `Q.length` e un nuovo elemento viene inserito, il prossimo valore di `Q.tail` sarà 1
  - se `Q.head` = `Q.tail` allora la coda è vuota
  - se `Q.head` = `Q.tail + 1` allora la coda è piena
- Se la coda non è piena, c'è sempre almeno una cella libera tra `Q.tail` e `Q.head`

Lo pseudocodice di queste funzioni è mostrato nei Listati 11 e 12.

```
1  ENQUEUE(Q, x)
2    if Q.head = Q.tail + 1:
3      error "overflow"
4    if Q.tail = Q.length
5      Q.tail := 1
6    else
7      Q.tail := Q.tail + 1
```

Listato 11: `ENQUEUE` in coda

```
1  DEQUEUE(Q)
2    if Q.head = Q.tail:
3      error "underflow"
4    x := Q[Q.head]
5    if Q.head = Q.tail
6      Q.head := 1
7    else
8      Q.head := Q.head + 1
9    return x
```

Listato 12: `DEQUEUE` in coda

La complessità temporale delle operazioni è $\mathcal{O}(1)$.

12.3 Lista *(doppiamente)* concatenata - *deque*

Una lista concatenata è una struttura dati in cui gli elementi sono sistemati in un ordine lineare, in modo simile ad un array. L'ordine non è dato dagli indici degli elementi *(che possono essere ordinati in un qualsiasi modo nella memoria del calcolatore)*, ma da una *catena* di puntatori.
Una **lista doppiamente concatenata** *(deque)* è costituita da oggetti con 3 attributi:

- `key`, rappresentante il **contenuto** dell'oggetto
- `next`, puntatore all'oggetto **seguente**
- `prev`, puntatore all'oggetto **precedente**

Sia `x` un oggetto della lista. Allora:

- Se `x.next` = `NIL`, allora x non ha successore ed è l'**ultimo** elemento della lista
- Se `x.prev` = `NIL`, allora x non ha predecessore ed è il **primo** elemento della lista
- Ogni lista `L` ha un attributo `L.head` che **punta al primo** elemento della lista

Una **lista *(singolarmente)* concatenata** ha un funzionamento analogo, con le stesse funzioni, ma senza avere il riferimento all'oggetto precedente *(ha solamente il riferimento all'elemento successivo, `next`)*

Su una lista doppiamente concatenata è possibile eseguire 3 operazioni:

- **Ricercare** un elemento, tramite la funzione `LIST-SEARCH`
- **Inserire** un elemento, tramite la funzione `LIST-INSERT`
- **Cancellare** un elemento, tramite la funzione `LIST-DELETE`

Un esempio di lista doppiamente concatenata è mostrata in Figura 44.

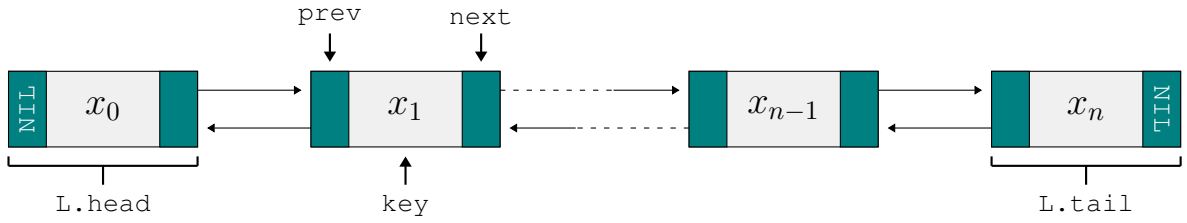


Figura 44: Lista doppiamente concatenata

12.3.1 Operazioni sulle liste

Nei prossimi Paragrafi verranno illustrati gli pseudocodici delle operazioni sulle liste, insieme alle loro complessità temporali.

12.3.1.1 Ricerca di un elemento

Input: la lista `L` in cui cercare e la chiave `k` dell'elemento desiderato.
Output: il puntatore all'elemento desiderato se esso è trovato nella lista, `NIL` in caso contrario.
Complessità temporale: $\Theta(n)$ nel caso pessimo *(la chiave non è nella lista)*.

Lo pseudocodice è riportato nel Listato 13.

```
1  LIST-SEARCH(L, k)
2    x := L.head
3    while x != NIL and x.key != k
4      x := x.next
5    return x
```

Listato 13: Pseudocodice dell'algoritmo `LIST-SEARCH`

12.3.1.2 Inserimento di un elemento

Input: la lista `L` in cui cercare e l'oggetto `x` da aggiungere *(inizializzato dalla chiave)*.
Output: nessuno, la lista `L` viene alterata senza che ne venga creata una nuova.
Complessità temporale: $\mathcal{O}(1)$.

Lo pseudocodice è riportato nel Listato 14.

```
1  LIST-INSERT(L, x)
2    x.next := L.head
3    if L.head != NIL
4      L.head.prev := x
5    L.head := x
6    x.prev := NIL
```

Listato 14: Pseudocodice dell'algoritmo `LIST-INSERT`

12.3.1.3 Cancellazione di un elemento

Input: la lista `L` in cui cercare e l'oggetto `x` da cancellare. Non è sufficientemente passare solo la chiave dell'oggetto: **è necessario passare tutto l'oggetto**.
Output: nessuno, la lista `L` viene alterata senza che ne venga creata una nuova.
Complessità temporale: $\Theta(1)$.

Lo pseudocodice è riportato nel Listato 15.

```
1  LIST-DELETE(L, x)
2    if x.prev != NIL
3      x.prev.next := x.next
4    else
5      L.head := x.next
6    if x.next != NIL
7      x.next.prev := x.prev
```

Listato 15: Pseudocodice dell'algoritmo `LIST-DELETE`

12.3.1.4 Cancellazione di un elemento data la chiave

Input: la lista `L` in cui cercare e l'oggetto `x` da cancellare. Contrariamente alla funzione analizzata precedentemente, **è sufficiente passare solo la chiave dell'oggetto**.
Output: nessuno, la lista `L` viene alterata senza che ne venga creata una nuova.
Complessità temporale: $\Theta(n)$ nel caso pessimo *(l'oggetto ricercato non è nella lista)*.

Lo pseudocodice è riportato nel Listato 16.

```
1  LIST-DELETE-KEY(L, k)
2    x = LIST-SEARCH(L, k)
3    if x = NIL:
4      return
5    LIST-DELETE(L, x)
```

Listato 16: Pseudocodice dell'algoritmo `LIST-DELETE-KEY`

12.3.2 Altri tipi di liste

Oltre alle liste doppiamente concatenate, delle liste comunemente usate sono:

- **Ordinate**
  - l'ordinamento degli elementi corrisponde a quello delle chiavi
  - il primo elemento ha la chiave minima, l'ultimo la massima
- **Non ordinate**
- **Circolari**
  - il puntatore `prev` della testa `(head)` punta alla coda (`tail`)
  - il puntatore `next` della coda `(tail)` punta alla testa (`head`)

12.4 Dizionario

Un **dizionario** è una struttura dati costituita da un insieme dinamico di oggetti che supporta solo le operazioni di:

- **Ricerca** di un elemento della collezione
- **Inserimento** di un elemento nella collezione
- **Cancellazione** di un elemento dalla collezione

Gli oggetti all'interno di un dizionario sono indirizzati tramite le loro chiavi, che si assume essere sempre numeri naturali *(se così non fosse, si potrebbe comunque usare la loro rappresentazione in memoria come stringhe di bit)*.
Se la cardinalità m dell'insieme delle possibili chiavi U *(quindi* $m = |U|$*)* è ragionevolmente piccola, il dizionario può essere implementato tramite un array di m elementi. In questo caso l'array prende il nome di **tabella ad indirizzamento diretto**. Ogni elemento `T[k]` dell'array contiene il riferimento all'oggetto con chiave `k` se un tale oggetto esiste nella tabella, altrimenti `NIL`.

Lo pseudocodice delle funzioni appena elencate è mostrato nei Listati 17, 18 e 19.
I parametri `T` e `k` rappresentano rispettivamente il **dizionario** e la **chiave** dell'elemento da cercare.

```
1  DIRECT-ADDRESS-SEARCH(T, k)
2    return T[k]
```

Listato 17: Ricerca

```
1  DIRECT-ADDRESS-INSERT(T, k)
2    T[x.key] = x
```

Listato 18: Inserimento

```
1  DIRECT-ADDRESS-DELETE(T, k)
2    T[x.key] := NIL
```

Listato 19: Cancellazione

La complessità delle operazioni è $\mathcal{O}(1)$, quindi i dizionari sono le strutture dati **più efficienti** tra quelle analizzate fin'ora. Tuttavia, se il numero di chiavi memorizzate è di molto inferiore alla cardinalità dell'insieme delle possibili chiavi, lo spazio occupato sarà inutilmente alto.

Un esempio di dizionario è mostrato nella Tabella 45.

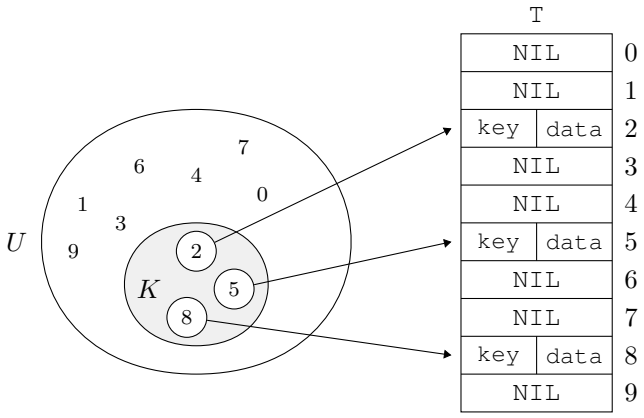


Figura 45: Dizionario

12.5 Tabella Hash

Una **tabella hash** usa una memoria proporzionale al numero di chiavi effettivamente memorizzate nel dizionario, indipendentemente dalla cardinalità dell'insieme U di chiavi.
Idea: un oggetto di chiave `k` è memorizzato in tabella in una cella di indice $h(\texttt{k})$, h è una **funzione di hash**:

- Se m è la dimensione della tabella, allora h è una funzione definita come $h :\to \{0, \ldots, m-1\}$
  - il **dominio** corrisponde al possibile numero di elementi dell'insieme
  - il **codominio** corrisponde all'indice degli elementi
  - $h(\texttt{k})$ è detto valore **hash** della chiave `k`
- La tabella `T` ha m celle $\texttt{T}[0], \texttt{T}[1], \ldots, \texttt{T}[m-1]$

12.5.1 Collisioni

Sia U l'insieme delle possibili chiavi di un dato dizionario. Ammettendo di avere $|U|$ chiavi, la funzione h dovrà mappare ognuna di esse su un numero $m < |U|$ di righe della tabella. Di conseguenza, chiavi diverse daranno

origine ad hash uguali:

$$\exists\, k_1, k_2 \;\Rightarrow\; h(k_1) = h(k_2)$$

Questo fenomeno prende il nome di **collisione**. Una delle tecniche per risolvere le collisioni prende il nome di **concatenamento**.
Idea della tecnica: gli oggetti che vengono mappati sullo stesso slot vengono posti in una lista concatenata.

La collisioni e le loro risoluzioni sono mostrate rispettivamente nelle Figure 46 e 47.

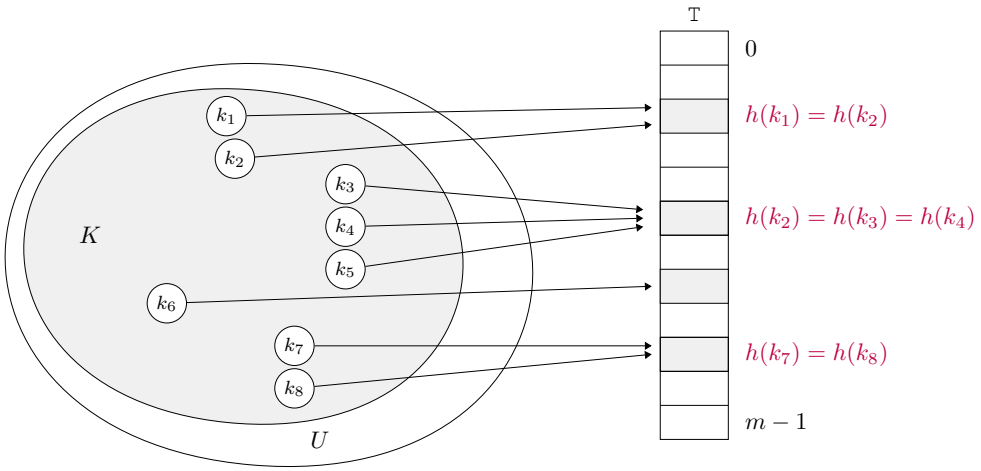


Figura 46: Collisioni in un dizionario

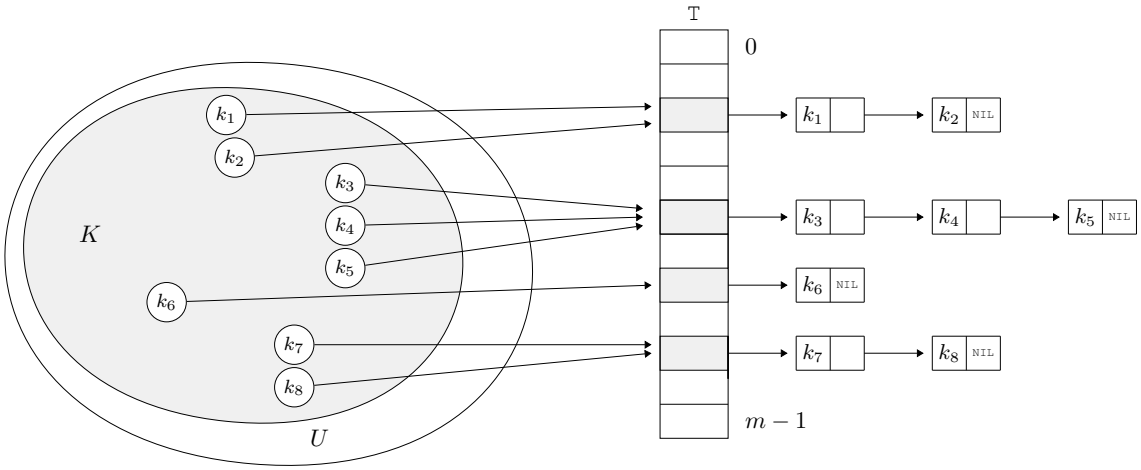


Figura 47: Risoluzione delle collisioni tramite concatenamento

12.5.2 Operazioni sulle tabelle hash

```
1    CHAINED-HASH-SEARCH(T, k)
2      cerca un elemento con chiave k nella lista T[h(k)]
```

Listato 20: Ricerca

```
1  CHAINED-HASH-INSERT(T, x)
2    inserisci X in testa alla lista T[h(x.key)]
```

Listato 21: Inserimento

```
1  CHAINED-HASH-DELETE(T, x)
2    cancella x dalla lista T[h(x.key)]
```

Listato 22: Cancellazione

Le complessità di queste operazioni è:

- $\mathcal{O}(1)$ per l'**inserimento** - `CHAINED-HASH-INSERT`
  - per ipotesi, l'elemento non è già nella tabella
- $\mathcal{O}(\texttt{T}[h(k)])$ per la **ricerca** - `CHAINED-HASH-SEARCH`
  - $\mathcal{O}(n)$ con $n =$ numero di elementi nella lista `T`[h(k)]
- $\mathcal{O}(1)$ per la **cancellazione** - `CHAINED-HASH-DELETE`
  - se la lista è singolarmente concatenata, allora la complessità è $\mathcal{O}(\texttt{T}[h(x.key)])$

Nel caso pessimo, in cui tutti gli n elementi memorizzati finiscono nello stesso slot, la complessità è quella di una ricerca in una lista di n elementi *(quindi $\mathcal{O}(n)$)*. Nella realtà, tuttavia, si dimostra che la complessità è più bassa.

Sia `T` una tabella hash di dimensione m contenente n elementi. Allora si definisce il **fattore di carico** α come il numero medio di elementi per ogni slot della tabella *(la lunghezza della catena di una cella)*: $\alpha = {}^{n}/_{m}$.
Poiché il numero di elementi è compreso tra 0 e il numero massimo di chiavi *(infatti $0 \leq n \leq |U|$)*, per ogni tabella è verificata la relazione $0 \leq \alpha \leq {}^{|U|}/_{m}$.

Ipotizzando ora che ogni chiave abbia la stessa probabilità di finire in una delle qualsiasi m celle di `T` *(la probabilità è quindi ${}^{1}/_{m}$)*, la lunghezza della media di una lista è:

$$E[n_j] = \frac{1}{m}\sum_{i=1}^{m} n_i = \frac{n}{m} = \alpha$$

Questa condizione è detta **ipotesi dell'hashing uniforme semplice**.
Il tempo medio necessario a cercare una chiave k non presente nella lista *(il caso peggiore peggiore per la ricerca)* è $\Theta(1+\alpha)$. Il termine 1 è dato dalla complessità del calcolo di $h(k)$, che viene considerato costante.

Allo stesso tempo, il tempo medio per la ricerca di un elemento presente nella lista è $\Theta(1+\alpha)$. Questo valore richiede una dimostrazione aggiuntiva.
Infatti, se $n = \mathcal{O}(m)$, allora il fattore di carico è $\alpha = {}^{n}/_{m} = {}^{\mathcal{O}(m)}/_{m} = \mathcal{O}(1)$. Di conseguenza, la complessità temporale è, in media, **costante per tutte le operazioni**.

12.5.3 Funzioni hash

Fin'ora non è stato discusso il modo in cui una funzione possa creare un hash, ma solo dell'esistenza della stessa. La domanda che sorge spontanea è quindi:

> *"Come si definisce una funzione hash? Quali devono essere le sue caratteristiche?"*

In primo luogo, una funzione deve soddisfare l'ipotesi di hashing uniforme semplice. A tale scopo, essa deve essere in grado di *"separare"* tra di loro chiavi *"vicine"*. Se esse fossero distribuite **uniformemente** in un intervallo $[0, \ldots, K-1]$, sarebbe sufficiente prendere una funzione del tipo $h(k) = \lfloor {}^{k}/_{L} \cdot m \rfloor$. Tipicamente, si impiegano delle funzioni euristiche basate sul dominio delle chiavi.

Un'altra ipotesi delle funzioni hash è che la chiave k sia un **intero non negativo** *(e quindi* $\forall\, k \in K,\ k \in \mathbb{N}$*).* Questa ipotesi non introduce un problema difficile da risolvere in quanto ogni valore può essere trattato come intero positivo interpretando correttamente la sua codifica *(per esempio, facendo un cast da float ad intero senza segno).*

12.5.3.1 Metodo della divisione

Nel **metodo della divisione**, una chiave k viene mappata in uno degli m slot disponibili nella tabella considerando il resto di k diviso da m. La funzione di hash è quindi:

$$h(k) = k \mod m$$

Questa tecnica fornisce funzioni di hashing molto facili da calcolare *(necessitano di una sola operazione).*
Per evitare risultati vicini tra di loro *(per ridurre le collisioni)* è necessario non prendere valori di k nella forma $m = 2^p$. Se così non fosse, $h(k)$ dipenderebbe solo dai p bit meno significativi di k. A tal scopo si scelgono valori di m **primi** e sufficientemente **lontani da una potenza di** 2.

12.5.3.2 Metodo della moltiplicazione

Nel **metodo della moltiplicazione**, il valore di k viene inizialmente moltiplicato per una costante A reale compresa tra 0 ed 1:

$$A \in \mathbb{R},\ 0 < A < 1$$

Dopodiché si considera la parte frazionaria di kA moltiplicata per m, prendendone la parte intera.
La funzione $h(k)$ ha quindi forma:

$$h(k) = \lfloor m \cdot (kA \mod 1) \rfloor$$

In questa formula $x \mod 1 = x - \lfloor x \rfloor$ corrisponde alla parte frazionaria di x.
Tramite questa funzione il valore di m non è critico, perché le eventuali collisioni dipenderanno dal valore di A.
Normalmente si scelgono valori di m pari a potenza di 2 *(cioè* $m = 2^p$*)* per rendere più semplici i calcoli.
Una rappresentazione di questo processo è illustrata nella Figura 48.

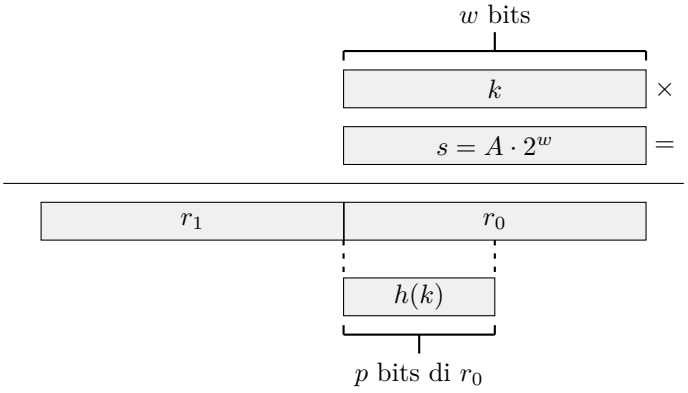


Figura 48: Metodo della moltiplicazione

È utile prendere come A un valore che sia nella forma $s/2^w$, con w pari alla dimensione della parola di memoria del calcolatore e $0 < s < 2^w$.
Se k è minore della lunghezza della parola ($k < 2^w$), allora $k \cdot s = k \cdot A \cdot 2^w$ è il numero di $2w$ bit della forma $r_1 2^w + r_0$ ed i suoi w bit meno significativi *(rappresentato da* r_0*)* costituiscono $kA \mod 1$. Il valore dell'hash cercato *(se* $m = 2^p$*)* è costituito dai p bit più significativi di r_0.
Un valore di A proposto da *David Knuth* è pari all'inverso della sezione aurea:

$$A = \frac{\sqrt{5} - 1}{2} \approx 0.6180339887\ldots$$

Applicando il calcolo precedente, è necessario prendere come valore di A la frazione nella forma $s/2^w$ più vicina a questo valore.

12.5.4 Hashing universale

Scegliendo accuratamente un insieme di chiavi, può accadere che la funzione di hash le assegni tutte allo stesso slot, portando il tempo medio di accesso ai dati a $\Theta(n)$ *(a causa della necessità di scorrere la lista associata allo slot)*. Ogni funzione di hash è vulnerabile a questo tipo di problema.
Come soluzione, la funzione di hash viene scelta **casualmente** in una classe di funzioni all'inizio dell'esecuzione. Grazie alla sua caratteristica aleatoria, l'algoritmo può comportarsi in modo diverso ad ogni esecuzione del programma, anche per input uguali.
La scelta di alcuni algoritmi potrebbe portare a prestazioni ridotte con un determinato insieme di chiavi, ma la probabilità di tale avvenimento è limitata ed uniforme per ogni funzione scelta.

Sia $\mathscr{H}$ un **insieme finito di funzioni** di hash che mappano un dato universo U di chiavi nell'insieme $\{0, 1, \ldots, m-1\}$. Questo insieme è detto **universale** se per ogni paio di chiavi distinte $k, l \in U$ il numero di funzioni di hash $h \in \mathscr{H}$ tale per cui la probabilità che $h(k) = h(l)$ è al massimo $\dfrac{|\mathscr{H}|}{m}$.

In altre parole, con una funzione di hashing scelta casualmente in $\mathscr{H}$, la probabilità di una collisione tra chiavi diverse k, l è non più della probabilità $1/m$ di una collisione se $h(k)$ ed $h(l)$ fossero scelte casualmente ed indipendentemente dall'insieme $\{0, 1, \ldots, m-1\}$.

12.5.5 Indirizzamento aperto

La tecnica dell'**indirizzamento aperto** serve a ridurre le collisioni. In questo caso la tabella contiene tutte le chiavi possibili e non sarà necessaria memoria ulteriore per le liste concatenate. Di conseguenza il fattore di carico α non sarà mai maggiore di 1 *($\alpha < 1 \forall |U|$)*.
Il vantaggio sta nel minor uso di memoria: non dovendo impiegare puntatori, sarà possibile costruire tabelle di memoria più grandi a parità di memoria occupata. Di conseguenza si generano meno collisioni e la ricerca degli elementi è più veloce.
Per effettuare inserzioni usando l'indirizzamento aperto, le celle della tabella vengono analizzate in ordine finché non ne viene trovata una vuota. La sequenza di esplorazione delle celle avviene secondo una **sequenza di ispezione** calcolata dalla funzione di hash in base alla chiave dell'oggetto. Le celle analizzate potrebbero *(e in molti casi lo sono)* non essere in ordine numerico.
La funzione di hash avrà forma:

$$h : U \times \{0, 1, \ldots, m-1\} \to \{0, 1, \ldots, m-1\}$$

Ogni sequenza di permutazione generata da h, costruita come:

$$\langle h(k, 0),\ h(k, 1),\ \ldots\ ,\ h(k, m-1) \rangle$$

è una permutazione di $\langle 0, \ldots, m-1 \rangle$

12.5.6 Operazioni in caso di indirizzamento aperto

Nei prossimi Paragrafi verranno illustrati gli pseudocodici delle operazioni sulle tabelle hash con indirizzamento aperto, insieme alle loro complessità temporali.

12.5.6.1 Inserimento

```
1  HASH-INSERT(T, k)
2     i := 0
3     repeat
4        j := h(k, i)
5        if T[j] = NULL
6           T[j] := k
```

```
7         return j
8       else
9         i := i + 1
10    until i = m
11    error "hash table overflow"
```

Listato 23: Pseudocodice dell'algoritmo HASH-INSERT

Per inserire un elemento di chiave k nella collezione T l'algoritmo procede secondo la **sequenza di ispezione**, controllando il contenuto di ogni corrispondente cella *candidata*.

12.5.6.2 Ricerca

```
1  HASH-SEARCH(T, k)
2    i := 0
3    repeat
4      j := h(k, i)
5      if t[j] = k
6        return j
7      else:
8        i := i + 1
9    until t[j] = NIL or i = m
10   return NIL
```

Listato 24: Pseudocodice dell'algoritmo HASH-SEARCH

L'algoritmo per la ricerca della chiave k controlla **la stessa sequenza di slot** usata nell'algoritmo di inserimento. La ricerca può quindi terminare con successo qualora venisse trovato uno slot vuoto, poiché k sarebbe stata inserita in quest ultimo e non in uno slot successivo nella sequenza.
L'algoritmo prende come input una tabella di hash T ed una chiave k, restituendo j se lo slot j contiene la chiave k oppure NIL se la chiave non è presente nella tabella.

12.5.6.3 Cancellazione

```
1    HASH-DELETE(T, k)
2      i := HASH-SEARCH(T, k)
3      t[i] = DELETED
```

Listato 25: Pseudocodice dell'algoritmo HASH-DELEETE

La cancellazione è una operazione più complicata, in quanto impostare a NIL lo slot desiderato non è possibile. Se così fosse, la ricerca di chiavi successive a quella cancellata non sarebbe permesso.
Una possibile soluzione sta nel rimpiazzare il valore NIL con un valore convenzionale DELETED. Così facendo tuttavia le complessità non dipenderebbero più dal fattore di carico *(come si vedrà più avanti)*.

12.5.6.4 Complessità temporale

Il tempo impiegato per trovare lo slot desiderato *(sia esso relativo ad un oggetto da eliminare o da inserire)* dipende dalla sequenza di ispezione restituita dalla funzione h, quindi dipende dalla sua implementazione.
Ipotizzando che le chiavi e le sequenze di ispezione abbiano distribuzione uniforme *(per l'ipotesi di hashing uniforme)*, ognuna delle $m!$ permutazioni di $\langle 0, \ldots, m-1 \rangle$ ha la stessa probabilità di essere scelta. Questa è una estensione alla sequenza di ispezione della precedente condizione di hashing uniforme semplice.
L'analisi della complessità viene fatta in funzione del fattore di carico α, che grazie alla struttura della tabella rispetterà le condizioni:

$$0 \leq \alpha \leq 1,\ \alpha = {}^{n}/_{m} \Rightarrow n \leq m$$

Sotto ipotesi di hashing uniforme, il numero medio di ispezioni necessarie per effettuare l'inserimento di in nuovo oggetto nella tabella è m se $\alpha = 1$ *(la tabella è piena)* e non più di $1/(1-\alpha)$ se $\alpha < 1$ *(se la tabella ha spazio disponibile)*. Il numero medio di ispezioni necessarie per trovare un elemento presente in tabella è:

$$T(\alpha) \begin{cases} = \dfrac{m+1}{2} & \text{se } \alpha = 1 \\ < \dfrac{1}{\alpha} \log\left(\dfrac{1}{1-\alpha}\right) & \text{se } \alpha < 1 \end{cases}$$

12.5.7 Tecniche di ispezione

Nella pratica, costruire funzioni di hash che soddisfino l'ipotesi di hashing fin'ora ricercate è molto difficili. Per questo motivo si accettano delle approssimazioni che si rivelano soddisfacenti. Esistono 3 tecniche:

1. Ispezione **lineare**
2. Ispezione **quadratica**
3. **Doppio hashing**

Seppur nessuna di queste tecniche produca le $m!$ permutazioni che sarebbero necessarie per soddisfare l'ipotesi di hashing uniforme, nella pratica forniscono una distribuzione di chiavi sufficientemente uniforme.
Tutte le tecniche necessitano di una **funzione di hash ausiliaria** nella forma $h' : U \to \{0, 1, \ldots, m-1\}$.

12.5.7.1 Ispezione lineare

Data una funzione di hash ordinaria $h' : U \mapsto \{0, 1, \ldots, m-1\}$ *(detta funzione di hash ausiliaria)*, il metodo di **ispezione lineare** usa la funzione di hash definita come:

$$h(k, i) = (h'(k) + i) \mod m, \quad i = 0, 1, \ldots, m-1$$

Data una chiave k, la sequenza inizia dallo slot $T([h'(k)])$ per proseguire con $T([h'(k)+1])$ fino a $T[m-1]$. Dopodiché l'algoritmo ricomincia da $T[0]$ esplorando tutti gli slot di T.
Poiché il valore iniziale determina l'intera sequenza di ispezione, solo m sequenze distinte possono essere generate.

Questo algoritmo, facile da implementare, soffre di un problema noto come **addensamento primario** *(primary clustering)*. La presenza di uno slot vuoto preceduto da i slot pieni viene riempito con probabilità pari a $p = (i+1)/m$. Serie lunghe di slot occupati tendono ad allungarsi ulteriormente, aumentando di conseguenza il tempo medio di ricerca.

12.5.7.2 Ispezione quadratica

Il metodo di **ispezione quadratica** usa una funzione di hash nella forma:

$$h(k, i) = \left(h'(k) + c_1 i + c_2 i^2\right), \ c_2 \neq 0, \ i = 0, 1, \ldots, m-1$$

in cui $h'(k)$ prende il nome di *funzione di hash ausiliaria* e c_1, c_2 sono chiamate *costanti ausiliarie*.
La posizione iniziale della sequenza è $T[h'(k)]$, seguita da posizioni la cui distanza dipende in modo quadratico dal valore di i.
I valori delle costanti c_1, e c_2 non sono scelti in modo libero ma sono forzati dalla forma della tabella.

Se due chiavi hanno gli stessi elementi iniziali della sequenza di ispezione (se $h(k_1, 0) = h(k_2, 0)$) allora condivideranno la stessa sequenza ($h(k_1, i) = h(k_2, i) \, \forall \, i$). Questa proprietà porta ad una forma di addensamento, detta **addensamento secondario** *(secondary clustering)*.
Come nell'ispezione lineare, poiché lo slot di partenza determina tutta la sequenza, esisteranno solo m sequenze distinte.

12.5.7.3 Doppio hashing

Il metodo di **doppio hashing** una funzione di hash nella forma:

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m, \quad i = 0, 1, \ldots, m-1$$

dove h_1, h_2 sono funzioni di hash ausiliarie.
La posizione iniziale della sequenza è $T[h_1(k)]$ mentre le posizioni successive hanno distanza dalla precedente pari a $h_2(k) \mod m$.

Contrariamente ai metodi visti fin'ora, la sequenza generata dal doppio hashing dipende in due modi dalla chiave k, dato che la posizione iniziale e la distanza tra posizioni successive della sequenza possono variare.
Il valore di $h_2(k)$ non deve avere divisori comuni *(al di fuori di* 1*)* con la dimensione m della tabella di hash per permettere la ricerca dell'intera tabella stessa.
Un modo comodo per assicurare questa è scegliere m pari ad una potenza di 2 e scegliere h_2 in modo che produca sempre numeri dispari.
Alternativamente si può scegliere m numero primo e scegliere h_2 in modo che produca sempre un intero positivo minore di m. Allora h_1 e h_2 assumono la forma:

$$\begin{aligned} h_1(k) &= k \mod n \\ h_2(k) &= 1 + (k \mod n') \end{aligned}$$

Il numero di sequenze generate tramite questa tecnica è quindi $\Theta(m^2)$ *(e non più* $\Theta(m)$ *come visto fin'ora)* visto che ogni possibile coppia $\langle h_1(k), h_2(k) \rangle$ genera una sequenza differente.
Le performance del doppio hashing sono molto vicine alle performance dello schema ideale di hashing uniforme. Di conseguenza, questo è considerato il metodo migliore per generare sequenze di ispezione.

13 Grafi

13.1 Alberi binari

Come già introdotto nella Sezione 11.1.1, un **albero binario** è composto da 3 elementi:

- un nodo **radice**
- un albero binario, suo **sotto albero sinistro**
- un albero binario, suo **sotto albero destro**

Normalmente gli alberi binari sono rappresentati nella memoria del calcolatore usando delle strutture di dati concatenate, come le liste concatenate *(viste nella Sezione 12.3)*. Alternativamente, come per gli *heap (Sezione 11.1.2)*, possono essere rappresentati tramite array *(enumerando i nodi)*.
Ogni nodo dell'albero è rappresentato da un oggetto che ha i seguenti attributi:

- `key`, la **chiave** del nodo
  - può rappresentare il contenuto del nodo
  - può avere anche dati satelliti
- `parent`, il **puntatore** al nodo **padre**
  - a volte viene chiamato semplicemente `p`
  - se `x.parent = NIL`, allora `x` è la radice
- `left`, il **puntatore** alla radice del **sotto albero sinistro**
  - se `left = NIL`, allora il sotto albero sinistro è **vuoto**
- `right`, il **puntatore** alla radice del **sotto albero destro**
  - se `right = NIL`, allora il sotto albero destro è **vuoto**

Inoltre ogni albero `T` ha un attributo `root`, il **puntatore** alla **radice** dell'albero stesso.

Un esempio di albero binario è mostrato nella Figura 49.

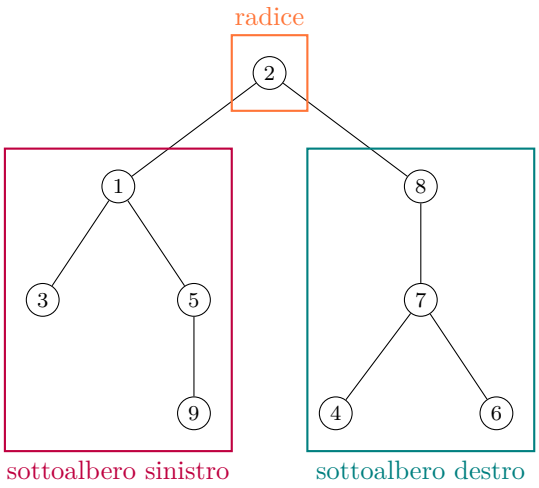
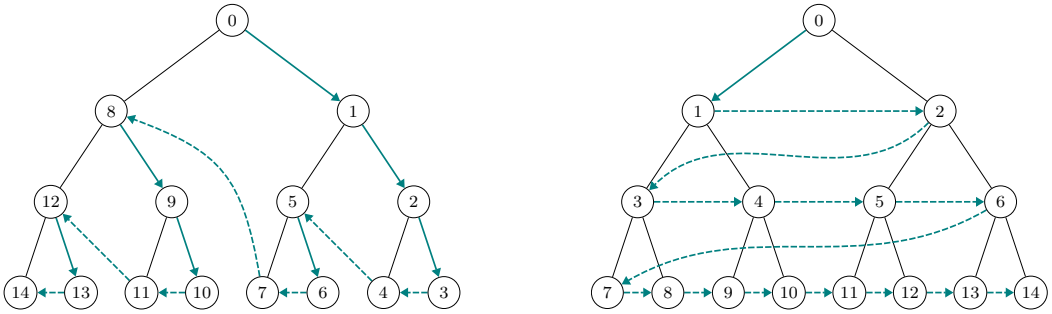


Figura 49: Esempio di albero binario

13.1.1 Operazioni sugli alberi binari

Nei Paragrafi seguenti *(13.1.1.1 - 13.1.1.3)* verranno analizzate delle operazioni comunemente applicate agli alberi binari, insieme a delle immagini esplicative.



(a) Attraversamento tramite *BFS*

(b) Attraversamento tramite *DFS*

Figura 50: Confronto tra attraversamenti di un albero binario, in ogni nodo è indicato l'ordine di visita

13.1.1.1 Attraversamento

L'attraversamento di un albero binario può avvenire esplorando prima in **larghezza** o prima in **profondità**. Nel primo caso si userà l'algoritmo `BFS` *(Breadth First Search)*, mentre nel secondo si userà l'algoritmo *DFS (Depth First Search)*.

- Nel caso del `BFS`, si cerca di visitare il nodo **più lontano** dalla radice, a patto che sia figlio di un nodo già visitato
  - **non è necessario ricordare** i nodi che sono stati visitati
- Nel caso del `DFS`, si cerca di visitare il nodo **più vicino** alla radice, a patto che non sia già stato visitato
  - **è necessario tenere traccia** di quali nodi sono già stati visitati

Entrambi gli algoritmi verranno analizzati più nel dettaglio, nelle loro applicazioni ai grafi, nei Paragrafi 13.5.1.1 e 13.5.1.2.

Un confronto grafico tra i due algoritmi e l'ordine di attraversamento è mostrato nelle Figure 50a e 50b.

13.1.1.2 Inserimento

La procedura di inserimento di un nodo in un albero binario è leggermente diversa se essa vuole operare su un nodo **foglia** *(esterno)* o un nodo **interno**:

- nel primo caso basta segnare il nodo come sotto albero
- nel secondo caso è necessario cambiare uno dei sotto alberi, assegnandoli al nuovo nodo `n`

Il processo di inserimento è illustrato nella Figura 51.

13.1.1.3 Cancellazione

La cancellazione di un elemento da un nodo non è sempre possibile senza creare ambiguità: se un nodo con due sotto alberi viene eliminato, non sarà possibile costruire un nuovo albero binario.
Per eliminare un nodo

- **con un solo sotto albero**, è sufficiente assegnare al padre di `n` il sotto albero di `n` nell'attributo corretto *(sia esso `left` o `right`)*
- **senza sotto alberi**, è sufficiente assegnare al padre di `n` il valore di `NIL` nell'attributo corretto *(sia esso `left` o `right`)*

Il processo di cancellazione è illustrato nella Figura 52.

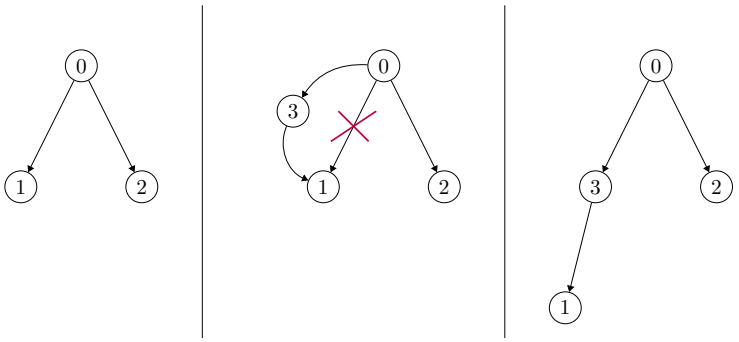


Figura 51: Inserimento di un nodo in un albero binario

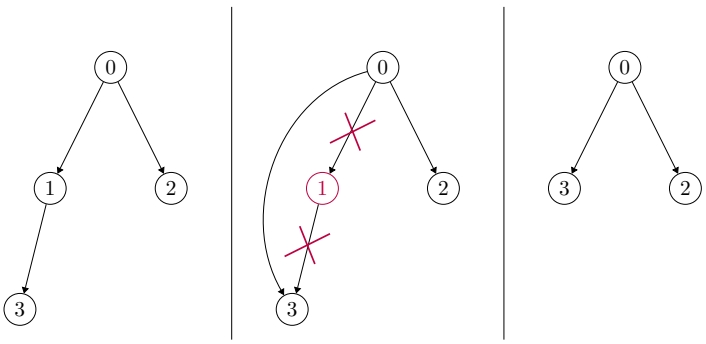


Figura 52: Cancellazione di un nodo in un albero binario

13.2 Alberi binari di ricerca - *Binary Search Trees*, *BST*

Un **albero binario di ricerca** *(in Inglese Binary Search Tree, BST)* è un albero binario che soddisfa la seguente proprietà:
Sia `x` un nodo di un *BST*. Se `y` è parte del:

- sotto albero **sinistro** di `x`, allora `y.key` $\leq$ `x.key`
- sotto albero **destro** di `x`, allora `y.key` $\geq$ `x.key`

Più semplicemente, il figlio sinistro di un nodo sarà sempre *minore o uguale* del nodo stesso. Allo stesso modo, il figlio destro sarà *maggiore o uguale*.

Un esempio di *BST* è mostrato nella Figura 53.

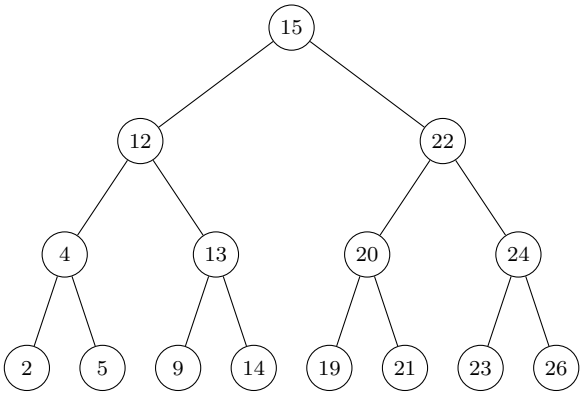


Figura 53: Esempio di binary search tree - *BST*

Grazie alla loro costruzione, è possible attraversare un *BST* visitando le chiavi in ordine crescente tramite un semplice algoritmo ricorsivo, detto attraversamento simmetrico o `INORDER-TREE-WALK`. Il suo nome deriva dalla sua caratteristica di funzionamento:

1. Il sotto albero **sinistro** viene visitato ed i suoi nodi vengono restituiti
2. La **radice** viene restituita
3. Il sotto albero **destro** viene visitato ed i suoi nodi vengono restituiti

.

Esistono definiti altri algoritmi che funzionano in modo analogo:

- **anticipato**, nel `PREORDER-TREE-WALK`, in cui la radice è restituita dopo i sotto alberi
- **posticipato**, nel `POSTORDER-TREE-WALK`, in cui la radice è restituita dopo i sotto alberi

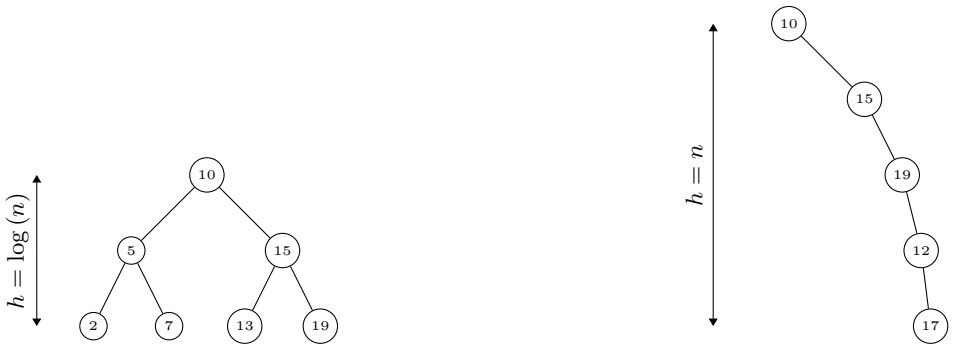
13.2.1 Altezza di un *BST*

In generale, l'altezza di un *BST* è pari a:

$$h \leq \log\left(\frac{n+1}{2}\right), \quad \forall\, n \geq 1$$

Ragionando in termini asintotici l'altezza è $\Theta(\log(n))$ per un albero bilanciato mentre è $\Theta(n)$ nel caso pessimo, in cui tutti i nodi sono in linea. Il confronto tra i due casi avviene nella Figura 54

É poi possibile dimostrare che l'altezza attesa di un albero costruito inserendo le chiavi in ordine casuale con distribuzione uniforme è $\mathcal{O}(\log(n))$.



(a) albero completo (b) albero in linea

Figura 54: Altezza di un albero binario

13.2.2 Attraversamento di *BST*

13.2.2.1 `INORDER-TREE-WALK`

```
1  INORDER-TREE-WALK(x)
2    if x != NIL
3      INORDER-TREE-WALK(x.left)
4      print(x.key)
5      INORDER-TREE-WALK(x.right)
```

Listato 26: Attraversamento simmetrico di *BST*

dove `x` è la radice del *BST* che si vuole visitare.

Chiamando questo algoritmo su un *BST* `T`, tutti i suoi elementi verranno stampati in ordine crescente.
Con n nodi nel sotto albero, il tempo di esecuzione è $\Theta(n)$. Questo limite può essere dimostrato per induzione:

1. se l'albero è **vuoto**, l'algoritmo è eseguito in tempo costante c
2. se l'albero ha 2 sotto alberi di dimensioni k e $n-k-1$, allora $T(n)$ è dato dalla ricorrenza

$$T(n) = T(k) + T(n-k-1) + d$$

13.2.2.2 `PREORDER-TREE-WALK` e `POSTORDER-TREE-WALK`

```
1  PREORDER-TREE-WALK(x)
2    if x != NIL
3      print(x.key)
4      PREORDER-TREE-WALK(x.left)
5      PREORDER-TREE-WALK(x.right)
```

Listato 27: Attraversamento anticipato di *BST*

```
1  POSTORDER-TREE-WALK(x)
2    if x != NIL
3      POSTORDER-TREE-WALK(x.left)
4      POSTORDER-TREE-WALK(x.right)
5      print(x.key)
```

Listato 28: Attraversamento posticipato di un *BST*

Il funzionamento di queste due funzioni è analogo a quello di `INORDER-TREE-WALK`, ma le chiavi vengono stampate in ordine diverso. La complessità è la stessa *(*$\Theta(n)$*)*.

13.2.3 Operazioni sui *BST*

Nei Paragrafi seguenti *(13.2.3.1 e 13.2.3.5)* verranno analizzate delle operazioni comunemente applicate ai *BST*, insieme alle loro relative complessità.

13.2.3.1 Ricerca

Per effettuare la ricerca di un nodo con una data chiave in un *BST*, dato un puntatore alla radice dell'albero e la chiave `k`, si usa la procedura riportata nel Listato 29. Essa ritornerà il puntatore al nodo `x` se presente nell'albero, altrimenti ritornerà `NIL`.
La procedura inizia dal nodo radice e traccia un percorso verso le foglie dell'albero. Per ogni nodo `x` che incontra, confronta il valore `x.key` con `k`:

- Se sono uguali o se `x` non ha un sotto albero, allora la ricerca termina
- Se `k` è minore di `x.key`, allora la ricerca continua nel sotto albero sinistro
- Se `k` è maggiore di `x.key`, allora la ricerca continua nel sotto albero destro

Lo stesso algoritmo può essere implementato in modo *ricorsivo* tramite un ciclo while. Un esempio di questo codice è mostrato nel Listato 30.

```
1  TREE-SEARCH(x, k)
2    if x = NIL or k = x.key
3      return x
4    if k < key[x]
5      return TREE-SEARCH(x.left, k)
6    else
7      return TREE-SEARCH(x.right, k)
```

Listato 29: Ricerca di un nodo in un *BST*

```
1  ITERATIVE-TREE-SEARCH(x, k)
2    while x != NIL and k != x.key:
3      if k < x.key
4        x = x.left
5      else
6        x = x.right
```

Listato 30: Ricerca iterativa di un nodo in un BST

La complessità temporale dell'algoritmo, per entrambe le soluzioni, è $\mathcal{O}(h)$, con $h =$ altezza dell'albero.

13.2.3.2 Massimo e minimo

Dato un *BST*, può essere importante trovare gli elementi le cui chiavi rappresentano i valori massimi o minimi. Un elemento dell'albero binario il cui valore è il minimo è sempre trovato seguendo il sotto albero sinistro *(tramite i puntatori `left`)* partendo dalla radice finché non si incontra un nodo senza sotto albero (il cui puntatore ha valore `NIL`). Allo stesso modo, il valore massimo è trovato seguendo sempre il sotto albero destro *(tramite i puntatori `right`)*.
I gli pseudocodici delle due funzioni sono illustrati nei Listati 31 e 32. Essi sono reciprocamente simmetrici.

```
1  TREE-MINIMUM(x)
2    while x.left != NIL
3      x = x.left
4    return x
```

Listato 31: Minimo di un BST

```
1  TREE-MAXIMUM(x)
2    while x.right != NIL
3      x = x.right
4    return x
```

Listato 32: Massimo di un BST

dove `x` è il puntatore alla radice dell'albero e il valore restituito sarà il puntatore al valore massimo o minimo, a seconda del caso.

La complessità temporale di entrambi gli algoritmi è $\mathcal{O}(h)$, con $h =$ altezza dell'albero.

13.2.3.3 Successore e predecessore

Dato un nodo di un *BST*, può essere importante trovare il suo successore o predecessore.
Se tutte le chiavi sono diverse, il successore di n nodo `x` è definito come il nodo con la chiave più piccola che è maggiore di `x.key`. Allo stesso modo il predecessore di `x` è il nodo con la chiave più grande che è minore di

x.key. La struttura del *BST*, tuttavia, permette di di eseguire queste operazioni senza dovere effettivamente confrontare le chiavi.
Gli pseudocodici delle due funzioni sono illustrati nei Listati 33 e 34.

```
1  TREE-SUCCESSOR(x)
2    if x.right != NIL
3      return TREE-MINIMUM(x.right)
4    y := x.parent
5    while y != NIL and x = y.right
6      x := y
7      y := y.parent
8    return y
```

Listato 33: Successore di di un nodo

```
1  TREE-PREDECESSOR(x)
2    if x.left != NIL
3      return TREE-MAXIMUM(x.left)
4    y := x.parent
5    while y != NIL and x = y.left
6      x := y
7      y := y.parent
8    return y
```

Listato 34: Predecessore di di un nodo

Gli algoritmi restituiscono rispettivamente il successore e il predecessore di un nodo x.

Se il sotto albero destro di x è vuoto, il successore di x è il primo elemento y che si incontra risalendo nell'albero da x tale che x è nel sotto albero sinistro di y. Allo stesso modo se il sotto albero sinistro di x è vuoto, il predecessore di x è il primo elemento y che si incontra risalendo nell'albero da x tale che x è nel sotto albero destro di y.

La complessità temporale di entrambi gli algoritmi è $\mathcal{O}(h)$, con h = altezza dell'albero.

13.2.3.4 Inserimento

L'operazione di inserimento causa modifiche nella rappresentazione del *BST*. La struttura dati deve essere modificata in modo da rispettare la proprietà dei *BST*.
L'inserimento di un nuovo valore è una operazione relativamente immediata.
Per inserire un nuovo valore v all'interno del *BST* T, viene usato l'algoritmo TREE-INSERT. Esso prende come parametro un puntatore al nodo z tale per cui z.key = v, z.left = NIL, z.right = NIL. T e alcuni dei campi di z vengono poi modificati in modo che il nuovo nodo sia inserito nella posizione appropriata dell'albero.

Lo pseudocodice dell'algoritmo è mostrato nel Listato 35.

```
1  TREE-INSERT(T, z)
2    y := NIL
3    x := T.root
4    while x != NIL
5      y := x
6      if z.key < x.key
7        x := x.left
8      else
9        x := x.right
10   z.parent := y
11   if y = NIL
12     T.root := z
13   else if z.key < y.key
14     y.left := z
15   else
16     y.right := z
```

Listato 35: Inserimento di un nuovo nodo

La complessità temporale di entrambi gli algoritmi è $\mathcal{O}(h)$, con h = altezza dell'albero.

13.2.3.5 Cancellazione

Analogamente a quanto già descritto per la procedura di inserimento, l'operazione di cancellazione causa modifiche nella struttura del *BST*.
Infatti, per eliminare un nodo `z` da un *BST* `T`:

- Se `z` **non ha sotto alberi**, allora è sufficiente modificare il padre di `z`, rimuovendo i suoi figli
- Se `z` **ha un solo sotto albero**, allora è sufficiente modificare il padre di `z`, aggiungendo come sotto albero `z`
- Se `z` **ha due sotto alberi**, è necessario:
  - trovare il successore `s` di `z`
  - copiare `s.key`, `s.left`, `s.right` in `z` e cancellare `s`

Lo pseudocodice dell'algoritmo è mostrato nel Listato 36, mentre una raffigurazione dei tre casi è mostrata nelle Figure 55, 56 e 57.

```
1  TREE-DELETE(T, z):
2    if z.left = NIL or z.right = NIL
3      y := z
4    else
5      y := TREE-SUCCESSOR(z)
6    if y.left != NIL
7      x := y.left
8    else
9      x := y.right
10   if x != NIL
11     x.parent := y.parent
12   if y.parent = NIL
13     T.root := x
14   else if y = y.parent.left
15     y.parent.left := x
16   else
17     y.parent.right := x
18   if y != z
19     z.key := y.key
20   return y
```

Listato 36: Cancellazione di un nodo

Dove `T` è la radice dell'albero e `z` è il nodo da eliminare.

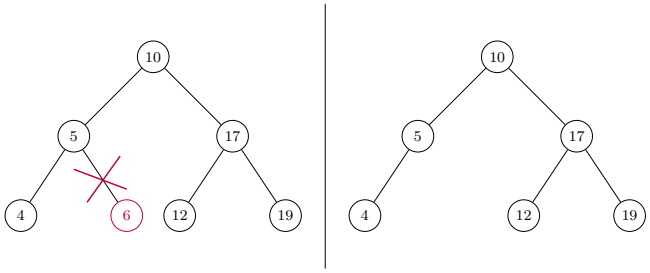


Figura 55: Cancellazione di un nodo senza sotto alberi

Funzionamento dell'algoritmo:

- Il nodo `y` è quello effettivamente da eliminare

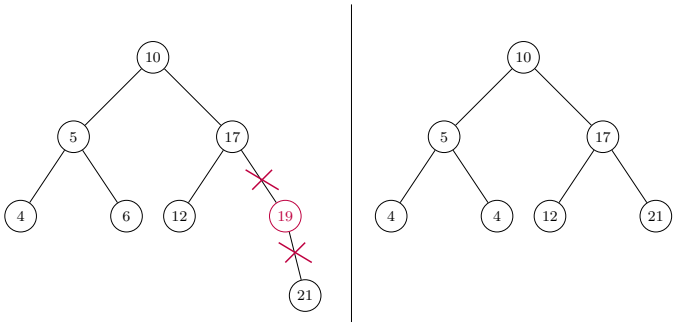


Figura 56: Cancellazione di un nodo con un solo sotto albero

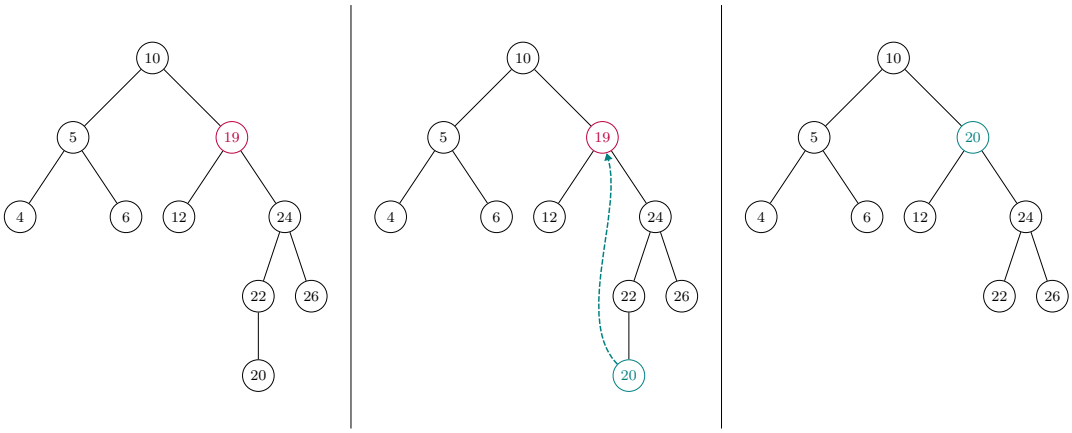


Figura 57: Cancellazione di un nodo con due sotto alberi

- Se `z` non ha più di un sotto albero, allora `y` coincide con `z`. In caso contrario, `y` è il nodo successore di `z` *(linee* $2-4$*)*
- Nelle linee $5-7$ viene assegnato a `x` la radice del sotto albero di `y` se presente o `NIL` altrimenti
- Le linee $8-14$ sostituiscono `y` con il suo sotto albero, avente `x` come radice
- Nelle linee $15-16$, se `z` ha 2 sotto alberi, la chiave di `z` è sostituita con quella del suo successore `y`

La complessità temporale di entrambi gli algoritmi è $\mathcal{O}(h)$, con $h =$ altezza dell'albero.

13.3 Alberi bilanciati

Esistono svariate definizioni di albero bilanciato.
Informalmente, un albero è detto **bilanciato** se e solo se non esistono due foglie nell'albero tali che la distanza dalla radice di una sia molto maggiore dell'altra.
Una possibile definizione formale *(definita da Adelson-Velskii e Landis)*, è la seguente:

> *"Un albero è bilanciato se e solo se, per ogni nodo `x` dell'albero, le altezze dei due sotto alberi di `x` differiscono al massimo di* 1*."*

Esistono varie tecniche per mantenere un albero bilanciato. Nella sezione seguente verranno esplorati gli alberi **rosso-neri**.

13.4 Alberi rosso-neri - *RB*

Gli **alberi rosso-neri** (o dall'Inglese *red-black*, in breve *RB*) sono alberi **bilanciati**. Le operazioni fondamentali, mostrate nella Sezione precedente, applicate agli alberi *RB* avranno complessità temporale pari a $\mathcal{O}(\log(n))$, con $h =$ altezza dell'albero, minore di quanto riscontrato negli *BST*.
Ogni nodo dell'albero contiene gli attributi `color`, `key`, `left`, `right` e `parent`. Se un nodo non ha un padre o dei sotto alberi, allora il valore relativo ai corrispondenti attributi è `NIL`.
I nodi degli alberi *RB* hanno quindi un attributo aggiuntivo rispetto ai *BST*, relativo al colore, che assume i valori **rosso** o **nero**. Distribuendo i colori nel modo corretto, viene garantito che nessun percorso dalla radice alle foglie sia più lungo di un altro.

Un albero binario è da considerarsi rosso-nero se ha le seguenti 5 *proprietà*:

1. Ogni nodo è **rosso** o **nero**
2. La **radice** è **nera**
3. Ogni **foglia** è **nera**
4. Se un nodo è **rosso**, entrambi i suoi figli sono **neri**
5. Per ogni **nodo**, ogni percorso dal nodo alle foglie discendenti contiene lo stesso numero di **nodi neri**
    - il numero di nodi neri nel percorso che part dal nodo `x` è detto $bh(\texttt{x})$
    - il nodo stesso non è conteggiato in $bh(\texttt{x})$ anche se nero

Per comodità, spesso tutte le foglie vengono rappresentate da un solo valore `NIL` anziché avere puntatori diversi. Questo nodo sarà accessibile come attributo dell'albero `T` e ogni riferimento a `NIL` sarà rimpiazzato da un puntatore a `T.NIL`.

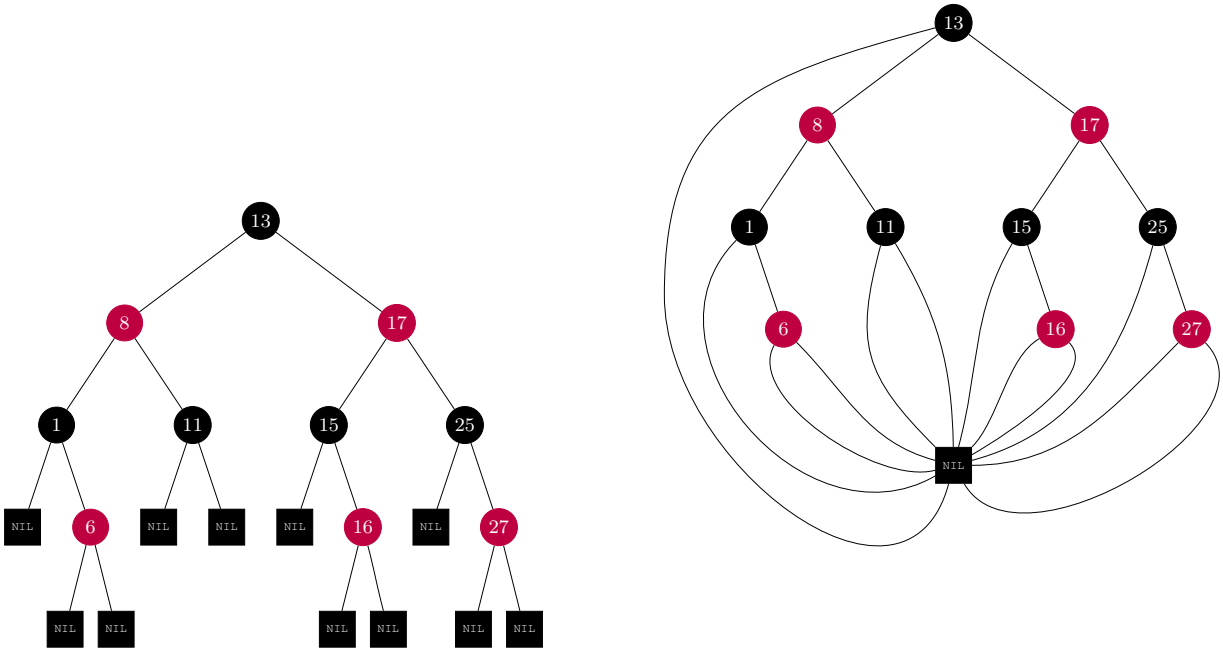
Il confronto tra le due rappresentazioni è mostrato nella Figura 58.

Grazie al modo in cui è costruito, un albero *RB* con n nodi interni (quindi n nodi con chiave) ha altezza:

$$h \leq 2\log_2(n+1)$$

Il numero di nodi interni di un albero *(o di un sotto albero)* con radice `x` è sempre $\geq 2^{bh(\texttt{x})-1}$. Questa proprietà può essere dimostrata per induzione sull'altezza di `x`.
Per la proprietà (5) degli alberi, almeno metà dei nodi dalla radice `x` *(esclusa)* ad una foglia sono neri. Di conseguenza $bh(\texttt{x}) \geq x/2$, $n \geq 2^{h/2}$ e quindi:

$$h \leq 2\log_2(n+1)$$



(a) Ogni foglia è rappresentata come `NIL`

(b) Tutte le foglie sono rappresentate da un solo nodo

Figura 58: Confronto tra le rappresentazioni degli alberi *RB*

13.4.1 Operazioni sugli alberi *RB*

Nei Paragrafi seguenti *(13.4.1.1-13.4.1.4)* verranno analizzate delle operazioni comunemente applicate ai *BST*, insieme alle loro relative complessità.

13.4.1.1 Ricerca, massimo, minimo, successore e predecessore

Lo pseudocodice delle operazioni di:

- **Ricerca**
- **Massimo** e **Minimo**
- **Successore** e **Predecessore**

è analogo a quello mostrato nelle operazioni dei *BST*, nella Sezione 13.2.3.
Sfruttando la formula relativa all'altezza di un albero, ricavata nella Sezione precedente, è possibile ricalcolare la complessità temporale di tutte le operazioni fin'ora. Essa sarà pari a:

$$\mathcal{O}(h) = \mathcal{O}(\log(n))$$

Dove n indica il numero di nodi dell'albero.

13.4.1.2 Rotazioni

Le operazioni di Inserimento e Cancellazione (Sezioni 13.4.1.3 e 13.4.1.4), hanno la caratteristica di alterare l'albero. La struttura risultante da queste operazioni potrebbe quindi violare le proprietà elencate nella Sezione 13.4.
Affinché esse siano sempre rispettate, il colore di alcuni nodi nell'albero e la struttura dei puntatori potrebbero dover essere modificati. Quest'ultima viene alterata tramite una operazione detta **rotazione**, che può essere **sinistra** o **destra**.

Una rotazione **destra** su un nodo di un albero *RB* richiede che il suo figlio **destro** non sia `T.NIL`, mentre il suo figlio sinistro può essere un nodo qualsiasi. Allo stesso modo, una rotazione **sinistra** richiede che il figlio **sinistro** non sia `T.NIL`.
Una rotazione sinistra sul nodo `x` con figlio `y` *"ruota"* attorno all'arco tra i due, trasformando `y` nella nuova radice con `x` come figlio sinistro e il figlio sinistro di `y` come figlio destro di `x`. La rotazione sinistra avrà comportamento speculare.

Gli pseudocodici delle funzione `LEFT-ROTATE` e `RIGHT-ROTATE` sono mostrati nei Listati 37 e 38.

```
1  LEFT-ROTATE(T, x)
2    y := x.right
3    x.right := y.left
4    if y.left != T.NIL
5      y.left.parent := x
6    y.parent := x.parent
7    if x.parent = T.NIL
8      T.root := y
9    else if x.parent.left = x
10     x.parent.left := y
11   else
12     x.parent.right := y
13   y.left := x
14   x.parent := y
```

Listato 37: Rotazione sinistra di un nodo in un RB

```
1  RIGHT-ROTATE(T, x)
2    y := x.left
3    x.left := y.right
4    if y.right != T.NIL
5      y.right.parent := x
6    y.parent := x.parent
7    if x.parent = T.NIL
8      T.root := y
9    else if x.parent.right = x
10     x.parent.right := y
11   else
12     x.parent.left := y
13   y.right := x
14   x.parent := y
```

Listato 38: Rotazione destra di un nodo in un RB

Entrambe le procedure hanno complessità temporale $\mathcal{O}(1)$. L'operazione di rotazione su un *RB* produce sempre un *RB (l'operazione di rotazione è chiusa rispetto all'insieme degli alberi RB)*. Inoltre, applicando due volte la rotazione *(prima sinistra e poi destra)* su un nodo si ottiene lo stesso albero di partenza.

Una rappresentazione grafica dell'operazione è mostrata nell'immagine 59.

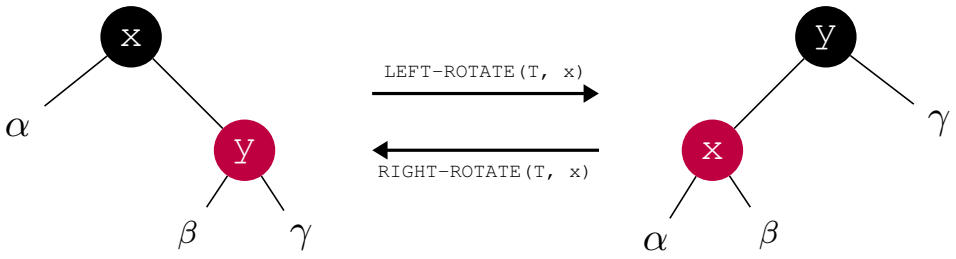


Figura 59: Rotazione sinistra e destra di un albero *RB*

13.4.1.3 Inserimento

L'inserimento è analogo a quello dell'*BST*, illustrato nel Paragrafo 13.2.3.4. Tuttavia a fine procedura è necessario ristabilire le proprietà degli alberi *RB*, se esse sono state violate, tramite l'algoritmo `RB-INSERT-FIXUP`.

Gli pseudocodici dei due algoritmi sono mostrati nei Listati 39 e 40.

```
1  RB-INSERT(T, z)
2    y := T.NIL
3    x := T.root
4    while x != T.NIL
5      y := x
6      if z.key < x.key
7        x := x.left
8      else
9        x := x.right
10   z.parent := y
11   if y = T.NIL
12     T.root := z
13   else if z.key < y.key
14     y.left := z
15   else
16     y.right := z
17   z.left := T.NIL
18   z.right := T.NIL
19   z.color := RED
20   RB-INSERT-FIXUP(T, z)
```

Listato 39: Inserimento in un RB

```
1  RB-INSERT-FIXUP(T, z)
2    while z.parent.color = RED
3      if z.parent = z.parent.parent.left
4        y := z.parent.parent.right
5        if y.color = RED
6          // caso 1
7          z.parent.color := BLACK
8          y.color := BLACK
9          z.parent.parent.color := RED
10         z = z.parent.parent
11       else if z = z.parent.right
12         // caso 2
13         z := z.parent
14         LEFT-ROTATE(T, z)
15       // caso 3
16       z.parent.color := BLACK
17       z.parent.parent.color := RED
18       RIGHT-ROTATE(T, z.parent.parent)
19     else
20       come le linee 3-18 con
21         left and right invertiti
22   T.root.color := BLACK
```

Listato 40: Fixup dopo inserimento

Per comprendere il funzionamento della procedura `RB-INSERT-FIXUP` è necessario dividere il codice in 3 casi principali, per poi analizzare l'obiettivo del ciclo **`while`** delle linee $2-21$.
Le proprietà (1) e (3) continuano ad essere valide poiché i figli di entrambi i nodi rossi inseriti sono `T.NIL`. Allo stesso modo la proprietà (5), che regola il numero di nodi neri tra su ogni percorso, è ancora soddisfatta.
Le uniche proprietà escluse sono:

- la (2), che richiede che la radice sia nera
  - violata solo se `z` è la radice
- la (4), che richiede che entrambi i figli di un nodo rosso siano neri
  - violata solo se il padre di `z` è rosso

Il ciclo **`while`** del codice nel Listato 40 mantiene valide le seguenti 3 proprietà:

1. Il nodo `z` è rosso
2. Se `z.parent` è la radice, allora `z.parent` è nero
3. Se si dovesse presentare una violazione delle proprietà dell'albero RB, allora è sicuramente una violazione della proprietà (2) o (4):
   - se la proprietà (2) è violata, allora `z` è la radice ed è rossa
   - se la proprietà (4) è violata, allora sia `z` che `z.parent` sono rossi

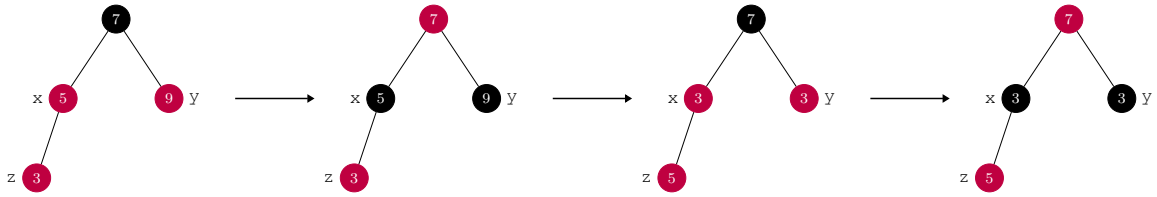
Quindi, come evidenziato nel Listato 40, sono evidenziati 3 casi:

Caso 1: `y` è rosso, `x` è rosso, `y` è rosso *(Figura 60a)*.
- la proprietà (4) è violata
- è necessario ripetere la procedura su `x.parent` perché suo padre potrebbe essere rosso
- la situazione è evoluta nel caso 2

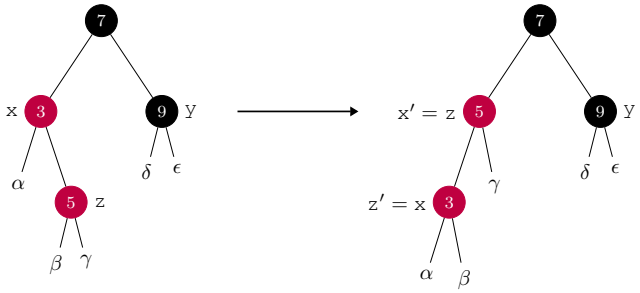
Caso 2: `y` è nero, `z` è il figlio destro di `x` *(Figura 60b)*.
- la proprietà (2) è violata

- la situazione è evoluta nel caso 3

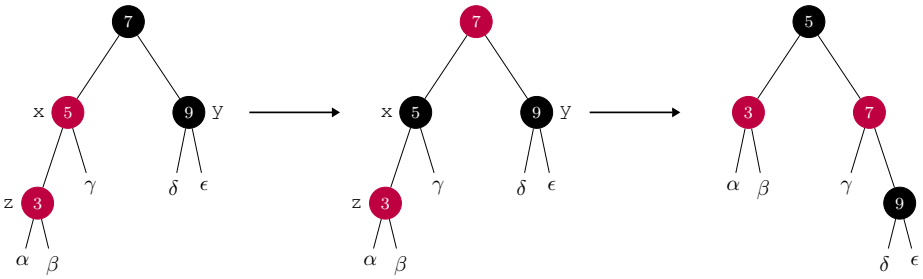
Caso 3: y è nero, z è il figlio sinistro di x *(Figura 60c)*.



(a) caso 1 dell'algoritmo RB-INSERT-FIXUP



(b) caso 2 dell'algoritmo RB-INSERT-FIXUP



(c) caso 3 dell'algoritmo RB-INSERT-FIXUP

Figura 60: Casi di RB-INSERT-FIXUP

Ogni volta che RB-INSERT-FIXUP viene eseguito, esso può terminare *(come nei casi* 2 *e* 3*)*, oppure può venire applicato ricorsivamente risalendo 2 livelli nell'albero *(come nel caso* 1*)*. In quest'ultimo caso, vengono effettuate 2 rotazioni *(durante l'ultima chiamata)*.
Quindi, nel caso peggiore, esso viene invocato al massimo h volte, dove h è l'altezza dell'albero. Di conseguenza la complessità temporale sarà $\mathcal{O}(h) = \mathcal{O}(\log(n))$.

13.4.1.4 Cancellazione

La procedura RB-DELETE è una versione adattata dell'analogo TREE-DELETE *(Paragrafo 13.1.1.3)*. Dopo l'eliminazione del nodo, una procedura secondaria chiamata RB-DELETE-FIXUP viene chiamata per assicurarsi che le proprietà dell'albero *RB* siano mantenute.
Il codice delle procedure è mostrato nel Listati 41 e 42.

```
1  RB-DELETE(T, z)
2    if z.left = T.NIL or z.right = T.NIL
3      y := z
```

```
4    else
5      y := TREE-SUCCESSOR(z)
6    if y.left != T.NIL
7      x := y.left
8    else
9      x := y.right
10   x.parent := y.parent
11   if y.parent = T.NIL
12     T.root := x
13   else if y.parent.left = y
14     y.parent.left := x
15   else
16     y.parent.right := x
17   if y != z
18     z.key := y.key
19   if y.color = BLACK
20     RB-DELETE-FIXUP(T, x)
```

Listato 41: Cancellazione in un RB

```
1  RB-DELETE-FIXUP(T, x)
2    if x.color = RED or x.parent = T.NIL
3      // caso 0
4      x.color := BLACK
5    else if x = x.parent.left
6      w := x.parent.right
7      if w.color = RED
8        // caso 1
9        w.color := BLACK
10       x.parent.color := RED
11       LEFT-ROTATE(T, x.parent)
12       w := x.parent.right
13     if w.left.color = BLACK and w.right.color = BLACK
14       // caso 2
15       w.color := RED
16       RB-DELETE-FIXUP(T, x.parent)
17     else if w.right.color = BLACK
18       // caso 3
19       w.left.color := BLACK
20       w.color := RED
21       RIGHT-ROTATE(T, w)
22       w := x.parent.right
23     w.color := x.parent.color
24     x.parent.color := BLACK
25     w.right.color := BLACK
26     LEFT-ROTATE(T, x.parent)
27   else
28   come le linee 2-26 con
29     left and right invertiti
```

Listato 42: Fixup dopo cancellazione

Il funzionamento di `RB-DELETE`, come già introdotto, è molto simile alla cancellazione di un nodo in un albero binario. Possono essere evidenziate le seguenti differenze:

- Viene usato il nodo sentinella `T.NIL` al posto del valore `NIL`
- Se un nodo **rosso** viene cancellato, **non è necessario** modificare il valore degli altri nodi
- Per la struttura di dell'algoritmo, se viene **cancellato** un nodo `y` con al massimo **un figlio**, allora `y` è **nero**

Se `y` è **nero**, le proprietà dell'albero *RB* potrebbero non essere mantenute:

- il numero di nodi neri incontrati in un cammino potrebbe essere diversa *(Proprietà (5) degli alberi RB)*
- due nodi neri potrebbero essere adiacenti *(Proprietà (5) degli alberi RB)*
- `y` sarebbe potuta essere la radice, quindi essa potrebbe non essere più nera *(Proprietà (2) degli alberi RB)*

Il nodo passato alla funzione `RB-DELETE-FIXUP` è uno tra:

- Il nodo figlio *(unico)* di `y` prima che esso venisse eliminato
- Il nodo sentinella *(`T.NIL`)* se `y` non aveva figli

Nel secondo caso, l'assegnamento nella linea 10 assicura che il padre di `x` e ora il nodo precedentemente padre di `y`, sia esso un nodo con valore o `T.NIL`.
All'interno del codice si delineano 4 casi:

Caso 0: `x` è un nodo **rosso**, oppure è la **radice**.
Caso 1: `x` è un nodo **nero**, suo fratello destro `w` è **rosso** *(Figura 61a)*.

- il padre di `x` è **nero**
- la struttura evolve nei casi 2, 3 o 4

Caso 2: `x` è un nodo **nero**, suo fratello destro `w` è **nero** con i figli **entrambi neri** *(Figura 61b)*.

- se si proviene dal caso 1, allora `x.parent` è **rosso**. L'invocazione di `RB-DELETE-FIXUP` termina subito

Caso 3: `x` è **nero**, suo fratello destro `w` è **nero** con figlio sinistro **rosso** e figlio destro **nero** *(Figura 61c)*.

- la struttura evolve nel caso 4

Caso 4: `x` è **nero**, suo fratello destro `w` è **nero** con figlio destro **rosso** *(Figura 61d)*.

Ogni volta che `RB-DELETE-FIXUP` viene invocata, essa può terminare *(casi* 0, 1, 3, 4*)* o venire applicato ricorsivamente risalendo un livello *(caso* 2 *non proveniente da* 1*)*. Una catena di invocazioni esegue al massimo 3 rotazioni *(evoluzione* $1 \rightarrow 3 \rightarrow 4$*)*.
`RB-DELETE-FIXUP` può essere quindi invocata al massimo h volte, dove h è l'altezza dell'albero. Di conseguenza la complessità temporale sarà $\mathcal{O}(h) = \mathcal{O}(\log(n))$.

13.5 Rappresentazione dei grafi in memoria

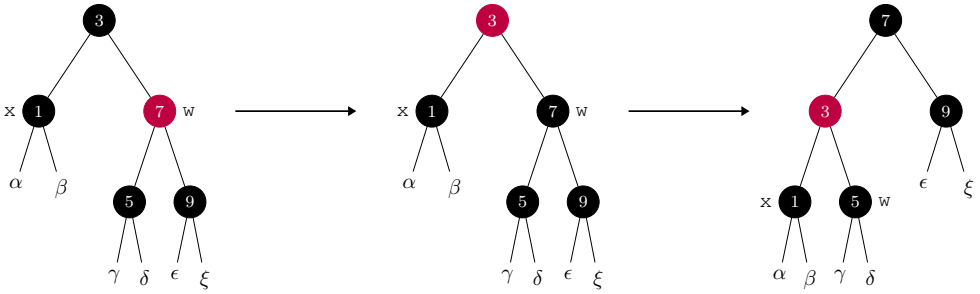
Esistono 2 tecniche principali per rappresentare i grafi in memoria:

- Tramite **liste di adiacenza**
  - implementato tramite un array di liste
  - per ogni vertice v, la corrispondente lista contiene i vertici adiacenti a v
- Tramite **matrici di adiacenza**
  - in una matrice di adiacenza M l'elemento m_{ij} è 1 se c'è un arco da i a j, altrimenti è 0

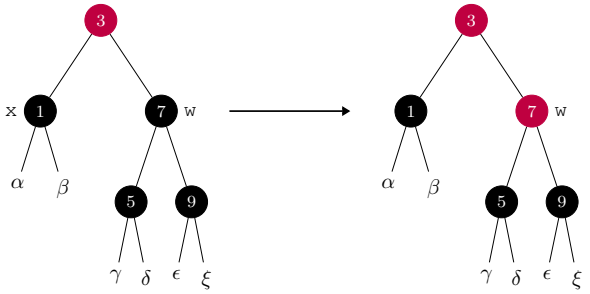
In entrambe le rappresentazioni, dato un nodo `u` appartenente ad un grafo G, l'attributo `u.adj` rappresenta l'insieme di vertici adiacenti a `u` all'interno di G.

La dimensione della rappresentazione è:

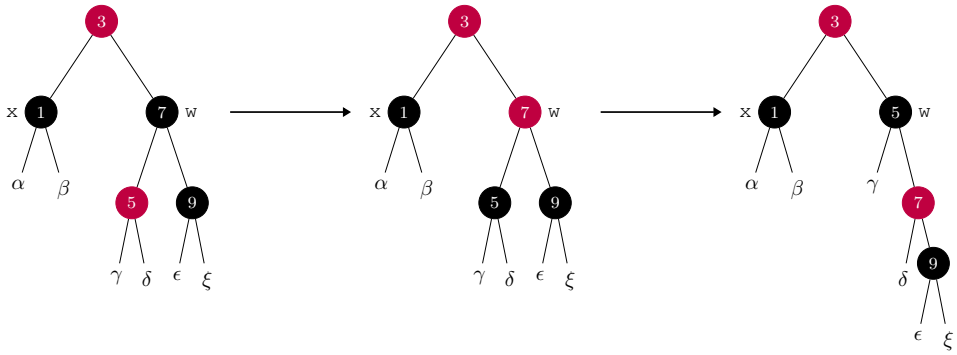
- matrici di adiacenza: $\mathcal{O}(|V|^2)$
  - la trasposta di una matrice di adiacenza relativa ad un grafo non orientato è uguale alla matrice stessa ($A^T = A$)



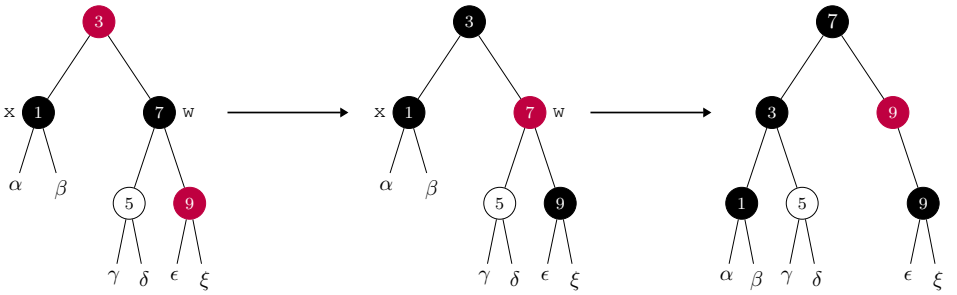
(a) caso 1 dell'algoritmo `RB-DELETE-FIXUP`



(b) caso 2 dell'algoritmo `RB-DELETE-FIXUP`



(c) caso 3 dell'algoritmo `RB-DELETE-FIXUP`



(d) caso 4 dell'algoritmo `RB-DELETE-FIXUP`

Figura 61: Casi di `RB-DELETE-FIXUP`

- liste di adiacenza: $\mathcal{O}(|V| + |E|)$
  - il numero totale di elementi nelle liste è $|E|$
  - il numero di elementi nell'array è $|V|$

Quindi le liste di adiacenza sono in generale migliori quando il grafo è **sparso** (il numero di lati è elevato) mentre le matrici sono più adatte ai grafi **densi**.

13.5.1 Operazioni sui grafi

13.5.1.1 Visita in ampiezza - *BFS*

L'algoritmo *Breadth First Search (BFS)* è caratterizzato da:

- *Input*: un grafo G *(sia diretto che non)*, un nodo s *(sorgente)* di G
- *Output*: visitare tutti i ndi di G che sono raggiungibili da S
  - un nodo u è raggiungibile da s se c'è un cammino nel grafo che collega s a u
  - il cammino tra s e un nodo u attraversato da *BFS* è il più breve possibile

L'algoritmo *BFS* esplora sistematicamente i lati di G espandendo la frontiera tra vertici visitati e da visitare: esso scopre tutti i vertici di distanza k da s prima di scoprire i vertici di distanza $k + 1$.
Per tenere traccia dei nodi già visitati, l'algoritmo colora di *bianco, grigio* e di *nero* ogni nodo presente nel grafo:

- Tutti i vertici sono inizialmente colorati di **bianco**
- Un nodo tutti i cui vicini sono stati scoperti viene colorato di **nero**
- Un nodo scoperto per la prima volta viene colorato di **grigio**
  - i nodi grigi formano la frontiera da espandere

Inoltre, ad ogni nodo viene aggiunto un attributo `dist` che indica la distanza di u dal nodo s.
La lista di grafi da visitare è implementata tramite una coda *(gestita con politica FIFO)*. Ad ogni iterazione, un nodo viene scelto ed i nodi bianchi adiacenti vengono esplorati.

Il Listato 43 contiene il codice dell'algoritmo appena descritto.

```
1  BFS(G, s)
2    for each u ∈ G.V - {s}
3      u.colour := WHITE
4      u.dist := ∞
5    s.colour := GREY
6    s.dist := 0
7    Q := ∅
8    ENQUEUE(Q, s)
9    while Q is not empty:
10     u := DEQUEUE(Q)
11     for each v ∈ u.adj:
12       if v.colour = WHITE:
13         v.colour := GREY
14         v.dist := u.dist + 1
15         ENQUEUE(Q, v)
16     u.colour := BLACK
```

Listato 43: Algoritmo `BFS`

Per calcolare la complessità temporale dell'algoritmo, è necessario analizzarlo in tutte le sue componenti. Infatti esso è composto da:

- Fase di **inizializzazione** *(linee* $2 - 8$*)* con complessità $\mathcal{O}(|V|)$
- Fase di visita dei nodi *(linee* $9 - 15$*)* con complessità $\mathcal{O}(|E|)$

La complessità totale sarà quindi $\mathcal{O}(|V| + |E|)$.

13.5.1.2 Visita in profondità - *DFS*

L'algoritmo *Depth First Search (DFS)*, al contrario dell'algoritmo *BFS*, si basa sull'idea di visitare i nodi con una politica *LIFO (il BFS preferisce una politica FIFO)*. L'idea alla base è che i vicini di ogni nodo vengono visitati non appena questo è aggiunto in cima alla stack, senza dover aspettare gli altri nodi. Questa scelta porta a visitare prima i nodi più lontani.
La procedura ha le stesse caratteristiche e funzionamento dell'algoritmo *BFS* appena analizzato, con cui condivide anche la complessità. Lo pseudocodice può essere ottenuto partendo dal *DFS* e cambiando `ENQUEUE` con `PUSH` e `DEQUEUE` con `POP`. Al suo interno i nodi vengono ulteriormente manipolati, aggiungendo gli attributi:

- `d` che indica il tempo relativo all'inizio della scoperta di un nodo *(discovery time)*
- `f` che indica il tempo relativo alla fine della scoperta di un nodo *(finishing time)*

Lo pseudocodice è mostrato nei Listati 44 e 45.

```
1  DFS(G, s)
2    for each u ∈ G.V - {s}
3      u.colour := WHITE
4    time := 0
5    for each u ∈ G.V
6      if u.colour = WHITE
7        DFS-VISIT(u)
```

Listato 44: Algoritmo `DFS`

```
1  DFS-VISIT(G, s)
2    u.colour := GREY
3    time := time + 1
4    u.dist := time
5    for each v ∈ u.adj
6      if v.colour = WHITE
7        DFS-VISIT(v)
8    u.colour := BLACK
9    time := time + 1
```

Listato 45: Funzione di supporto a `DFS`

La complessità temporale dell'algoritmo è data da:

- Inizializzazione dei nodi e delle variabili *(linee* $2-3$ *dell'algoritmo* `DFS`*)* con complessità $\Theta(|V|)$
- L'algoritmo `DFS-VISIT` è ripetuto *(linee* $5-7$ *dell'algoritmo* `DFS`*)* una volta durante l'esecuzione del ciclo **`for`** *(linee* $5-7$ *dell'algoritmo* `DFS-VISIT`*)* per ogni lato con complessità $\Theta(|E|)$

La complessità totale sarà quindi $\mathcal{O}(|V|+|E|)$ *(analoga al BFS).*

13.5.1.3 Ordinamento topologico

Un ordinamento topologico di un *DAG* (un *directed acyclic graph*, come descritto nella Sezione 11.1) è un ordinamento lineare dei nodi di tutti i vertici di un grafo G tale per cui se G contiene un lato (u, v), allora u appare prima di g nell'ordinamento. I nodi saranno quindi disposti in una linea orizzontale.
È importante notare che gli ordinamenti topologici non sono univoci e che un dato *DAG* può ammettere più ordinamenti diversi tra di loro.
L'ordinamento rispetterà quindi le precedenze tra eventi. Esso sarà caratterizzato da:

- *input*: un *DAG* G
- *output*: una lista, ordinamento topologico di G

Idea dell'algoritmo:

- il *DAG* viene vistato con un algoritmo *DFS*
- quando un nodo viene colorato di nero, esso viene inserito in testa alla lista
- una volta che tutti i nodi sono stati visitati, la lista costruita è un ordinamento topologico di G

I Listati 46 e 47 contengono il codice dell'algoritmo appena descritto.

```
1  TOPOLOGICAL-SORT(G)
2    L := ∅
3    for each u ∈ G.V
4      u.colour := WHITE
5    for each u ∈ G.V
6      if u.colour = WHITE
7        TOPSORT-VISIT(L, u)
8    return L
```

Listato 46: Algoritmo TOPOLOGICAL-SORT

```
1  TOPSORT-VISIT(L, u)
2    u.colour := grey
3    for each v ∈ u.adj
4      TOPSORT-VISIT(L, v)
5    x := elemento di lista x
6    x.key = u
7    LIST-INSERT(L, x)
8    u.colour = BLACK
```

Listato 47: Algoritmo TOPSORT-VISIT

Il tempo di esecuzione di TOPSORT è lo stesso di *DFS*, quindi la sua complessità temporale è $\Theta(|V| + |E|)$. Infatti le linee $6 - 8$ di TOPSORT-VISIT hanno complessità $\Theta(1)$ ed il resto dell'algoritmo è uguale a DFS, con l'assenza della variabile TIME.

14 Argomenti avanzati

14.1 Cammini minimi

Il **cammino minimo** tra due vertici *(o nodi)* di un grafo è il percorso che collega i vertici minimizzando la somma dei costi associati all'attraversamento di ciascun lato.
Il costo di attraversamento di un lato è dato da una funzione detta di **peso** definita come $w : E \to \mathbb{R}$.
Il problema di ricerca del cammino minimo si mira a trovare un cammino $P = (e_{v_1 v_2}, e_{v_2 v_3}, \ldots, e_{v_n v'_n})$, *dal nodo v al nodo v'*, che tra tutti i cammini possibili abbia il peso minore. Normalmente i cammini minimi sono calcolati partendo da un nodo fissato s detto **sorgente**, dato del problema. Affinché il cammino minimo resti ben definito non è possibile percorrere cicli negativi, poiché uno di essi verrebbe percorso infinite volte portando il costo a $-\infty$.

Il peso di un cammino è definito come:

$$W(p) = \sum_{i=1}^{k} w(v_{i-1}, v_i)$$

Il peso del cammino minimo dal nodo v al nodo v' è quindi:

$$\delta(v, v') = \begin{cases} \min\left\{w(p) : v \xrightarrow{P} v'\right\} & \text{se } \exists \text{ cammino da } v \text{ a } v' \\ \infty & \text{altrimenti} \end{cases}$$

L'uscita dell'algoritmo fornisce, per ogni vertice, il costo del cammino dal nodo di partenza fino ad esso. Per un nodo v, il costo viene rappresentato come $d[v] = \delta(s, v)$. Il predecessore di un nodo v nel cammino minimo è indicato con $Pi[v]$.
Inizialmente il costo viene impostato come $d[v] = \infty$ ed esso viene progressivamente ridotto tramite rilassamenti. Allo stesso modo il predecessore di ogni nodo viene inizializzato a `NIL`.
Operativamente, il rilassamento di un lato tra i nodi u e v è definito come:

$$\text{se } d[v] > d[u] + w(u, v) \quad \Rightarrow \quad d[v] \coloneqq d[u] + w(u, v),\ Pi[v] = u$$

Informalmente, *"costa di meno"* passare per il lato (u, v).

Lo stesso algoritmo in pseudocodice è mostrato nel Listato 48.

```
1  RELAX(u, v, adj, d, Pi)
2    if d[v] > d[u] + adj[u][v]
3      d[v] := d[u] + adj[u][v]
4      Pi[v] := u
```

Listato 48: Algoritmo per il rilassamento di un lato

14.1.1 Algoritmo di Bellman-Ford

L'**algoritmo di Bellman-Ford** trova il cammino minimo tra una sorgente `s` e tutti i nodi di un grafo, a patto che i costi siano tutti positivi,
Idea dell'algoritmo: rilassare, uno alla volta e partendo da s, ogni cammino. Ad ogni iterazione si aumenta di un passo ogni cammino e dopo $|V| - 1$ iterazioni ogni nodo raggiungibile sarà stato visitato.

Lo pseudocodice dell'algoritmo è mostrato nel Listato 49.

```
1  BELLMAN-FORD(adj, s)
2    V := vettore dei nodi di adj
3    alloca d, array di dimensione adj
4    alloca Pi, array di dimensione adj
5    for i := 0 to adj.length
```

```
6       d[i] := ∞
7       Pi[i] := NIL
8     d[s] := 0
9     for i := 0 to adj.length - 1
10      for each u ∈ V
11        for each v ∈ adj[u]
12          RELAX(u, v, adj, d, Pi)
13    return d, Pi
```

Listato 49: Algoritmo di Bellman-Ford

14.2 Programmazione dinamica

La programmazione dinamica, in analogia con la tecnica *Divide et Impera*, si basa sull'idea di scomporre il problema da risolvere in sotto problemi finché non sono abbastanza semplici da essere risolti con facilità. Questa tecnica è applicabile quando i problemi non sono indipendenti, cioè condividono dei sotto problemi in comune. Una volta risolto uno di questi, la soluzione viene salvata in una tabella per permetterne il riuso in seguito. Il termine programmazione non deriva dalla codifica in linguaggi di programmazione ma dal fatto che è una tecnica tabulare.
La programmazione dinamica è spesso usata per problemi di ottimizzazione. La soluzione cercata è uno dei multipli sotto problemi ammessi.

I passi normalmente usati per applicare questa tecnica sono:

1. **Caratterizzazione** della struttura delle soluzioni ottimali
2. **Definizione** ricorsiva dei valori di una soluzione ottimale del problema
3. **Calcolo** di una soluzione ottimale tramite algoritmi bottom-up *(partendo dai problemi più semplici)*
4. **Costruzione** della soluzione ottimale del problema richiesto

14.2.1 Algoritmo di programmazione dinamica

Applicando un algoritmo ricorsivo per risolvere un certo problema tramite programmazione dinamica, può risultare che la complessità sia eccessivamente elevata *(anche* $\Omega(2^n)$*)* poiché certe soluzioni vengono calcolate più volte.
Facendo uso di memoria aggiuntiva *(dove permesso)* è possibile ridurre il tempo di esecuzione, fino ad avere complessità polinomiale. Questa scelta introduce un *trade-off* spazio temporale: aumentando la complessità spaziale, quella temporale si riduce.

Idea della tecnica: il risultato dei sotto problemi già calcolati viene memorizzato. Così facendo non è necessario calcolarli più di una volta, perché se essi servissero sarebbero disponibili nella tabella. Se il costo dei singoli problemi da distinguere è polinomiale, allora la soluzione dell'intero problema richiederebbe tempo polinomiale. Questa tecnica prende il nome di **memoization**.

14.2.2 Approcci top-down e bottom-up

All'interno della programmazione dinamica si identificano due tipi principali di approccio: **top-down** e **bottom-up**.
L'approccio top-down consiste nel cercare di risolvere dapprima il problema di dimensione n *(il più grande)* per poi concentrarsi su problemi sempre più piccoli. La dimensione della tabella dei risultati e dei parametri passati aumenta man mano che la ricerca della soluzione procede.
L'approccio bottom-up, al contrario, si concentra dai problemi più piccoli procedendo fino ai problemi più grandi. Una volta risolto il problema di dimensione n, tutti i problemi di dimensione $< n$ sono già stati tutti risolti.

14.3 Algoritmi golosi

Gli algoritmi golosi (dall'Inglese greedy algorithm) rappresentano una classe particolare di algoritmi di ricerca. Essi infatti si concentrano sul trovare la soluzione ottimale per ogni passo di risoluzione.
In generale, gli algoritmi golosi cercano gli ottimi locali valutando ad ogni passo la scelta migliore su cui continuare. Questi algoritmi potrebbero portare a soluzioni non ottimali a livello globale ma, in alcuni casi, si dimostra che l'ottimo viene raggiunto.
Sono spesso impiegati in problemi difficili in cui l'ottimo locale fornisce una *"buona approssimazione"* dell'ottimo globale.

14.4 Complessità e non determinismo

L'obiettivo di questa Sezione sarà cercare di costruire una sorta di classe universale di complessità. cioè una funzione di complessità $T(n)$ tale che ogni problema al suo interno ha soluzione che impiega al più $T(n)$. Inoltre verrà analizzato il non determinismo in relazione alla sua influenza sulla complessità di soluzione dei problemi. Prima di poter cercare una risposta, è necessario dare alcune definizioni:

- Data una funzione $T(n)$, prende il nome $DTIME(T)$ la classe di problemi tale per cui esiste un algoritmo che li risolve in tempo $T(n)$
- Allo stesso modo, si indica con $DSPACE(T)$ l'insieme riconoscibili in spazio $T(n)$ mediante *TM* deterministiche
- $NTIME(T)$ e $NSPACE(T)$ rappresentano i loro analoghi con soluzioni non deterministiche

Un problema può essere identificato nel riconoscimento di un linguaggio (per semplicità, ricorsivo) e come algoritmo si sceglie una *TM*.

Si dimostra che data una funzione totale e computabile $T(n)$, esiste un linguaggio ricorsivo che non è in $DTIME(T)$. È quindi possibile costruire una gerarchia di linguaggi (e allo stesso modo di problemi) organizzata in base alla complessità temporale deterministica. Allo stesso modo, si possono costruire delle gerarchie per $DSPACE(T)$, $NTIME(T)$ e $NSPACE(T)$.

Ci si dedicherà ora alle computazioni non deterministiche. Data una *TM* non deterministica $\mathcal{M}$, la sua complessità temporale $T_{\mathcal{M}}(x)$ relativa al riconoscimento di una stringa x è definita come la lunghezza della computazione più breve tra tutte quelle che accettano x. $T_{\mathcal{M}}(n)$ sarà poi il caso pessimo tra tutti i $T_{\mathcal{M}}(x)$ con $|X| = n$.
$NTIME(T)$ sarà la classe dei linguaggi ricorsivi riconoscibili in tempo T mediante *TM* non deterministiche.
Tuttavia, la soluzione di un problema attraverso algoritmi non deterministici avviene tramite meccanismi di calcolo effettivamente deterministici. Se fosse possibile costruire una tecnica *"poco costosa"* per passare da una formulazione non deterministica ad una deterministica sarebbe possibile risolvere problemi *"interessanti"* in modo efficiente.
Questo metodo, ovviamente, non esiste. Come mostrato nel passaggio da *FSA* a *NFA* (avvenuto nella Sezione 5.1.3), infatti, il meccanismo si complica in modo esponenziale. Lo stesso concetto può essere mostrato per qualsiasi altro meccanismo di calcolo.

15 Conclusioni sparse delle precedenti Sezioni

L'obiettivo di questa Sezione sarà ricapitolare delle conclusioni tratte nelle precedenti sezioni, con particolare riguardo agli esercizi ed alle applicazioni pratiche.

15.1 Ripasso di matematica

- Formula di Gauss: $\sum_{i=1}^{k} i = \frac{k(k+1)}{2}$
- Sommatoria di una serie geometrica: $\sum_{n=0}^{+\infty} q^n = \frac{1}{1-q}$ con $|q| < 1$
- Approssimazione di Stirling: $n! \approx \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$

15.1.1 Proprietà dei logaritmi

- Definizione di logaritmo: $a^{\log_a(b)} = b,\ a, b > 0,\ a \neq 1$
- Logaritmo del prodotto: $\log_a(b \cdot c) = \log_a(b) \cdot \log_a(c)$
- Logaritmo del rapporto: $\log_a(b/c) = \log_a(b) - \log_a(c)$
- Regola dell'esponente: $\log_a(b^c) = c \cdot \log_a(b)$
- Formula di cambio di base per i logaritmi: $\log_a(b) = \frac{\log_c(b)}{\log_c(a)}$ con $a, b, c > 0,\ c \neq 1$
- Approssimazione: $\log(n!) \approx n\log(n)$ per $n \to \infty$

15.2 Scala di potenza delle classi di automi

Le classi di automi possono essere rappresentati in una scala, in ordine dal **più** al **meno** potente. Ciò avviene nella Figura 62.
Si noti che *NFA* e *FSA* hanno potenza equivalente in quanto esiste un algoritmo per convertire i primi nei secondi. Allo stesso modo *TM* e *NTM* hanno potenza equivalente perché il non determinismo non aggiunge potenza.

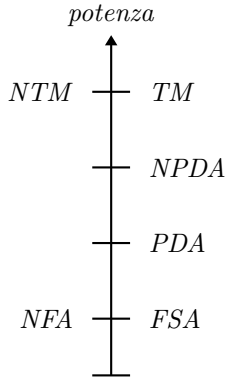


Figura 62: Scala di potenza degli automi

15.3 Chiusura degli automi rispetto alle operazioni

La chiusura degli automi rispetto alle operazioni è mostrata all'interno della Tabella 8.

15.4 Automi e grammatiche di Chomsky

All'interno della Tabella 9 è mostrata la relazione tra grammatiche secondo la caratterizzazione di Chomsky e corrispondente formalismo a potenza minima che la riconosce.

classe di automi	*unione* $\cup$	*intersezione* $\cap$	*complemento* A^c	*differenza* $\setminus$	*stella di Kleene* A^*	*concatenazione* $\cdot$
FSA, NFA	✔	✔	✔	✔	✔	✔
PDA	✖	✖	✔	✖	✖	✖
NPDA	✔	✖	✖	✖	✔	✔
TM, NTM	✔	✔	✖	✖	✔	✔

Tabella 8: Chiusura degli automi rispetto alle operazioni

grammatica	*formalismo*	*note*
0	*TM*	*grammatiche* ***generali*** *e* ***dipendenti*** *dal contesto*
1	*linear bounded automaton*	*non trattati nel corso*
2	*NPDA*	*grammatiche* ***non dipendenti*** *dal contesto*
3	*FSA, NFA*	*grammatiche* ***regolari***

Tabella 9: Relazione tra grammatiche e formalismi

15.5 Pattern tipici delle grammatiche

Si riconoscono pattern tipici per generare delle strutture note all'interno di grammatiche. Essi sono mostrati in Tabella 10.

linguaggio	*pattern*	*note*
$a^n b^n, n \geq 0$	$S \to aSb \mid \epsilon$	*stella di Kleene*
$a^n b^n, n \geq 1$	$S \to aSb \mid ab$	*più di Kleene*
$ww^r,\ w = (a \mid b)^*$	$S \to aSa \mid aBb \mid a \mid b \mid \epsilon$	*stringhe palindrome*
$(ab^+)^*$	$\begin{cases} S \to ah \mid \epsilon \\ h \to bh \mid bS \end{cases}$	

Tabella 10: Pattern tipici

15.5.1 Sintetizzazione di grammatiche di tipo 0

Idea generale: simulare i nastri di una *TM* tramite le regole della grammatica. In dettaglio:

- I nastri memorizzano i caratteri non terminali
- Altri caratteri non terminali simulano le testine
- I simboli nei nastri non mossi attraverso regole di *"swap"*

15.6 Linee guida sulla decidibilità

15.6.1 Casi immediati

- C'è una domanda di tipo **booleano** la cui risposta non dipende da alcun parametro esterno?

  $\Rightarrow$ La domanda è **chiusa** e il problema è **decidibile**

- La funzione in questione consiste di un **numero finito di casi**, tutti singolarmente decidibili e calcolabili?
  ⇒ La funzione è **calcolabile** e il problema è **decidibile**

15.6.2 Caso del programmatore

È possibile scrivere un programma in un generico linguaggio (sia esso `C`, `Java`, `...`) che risolve il problema dato?

⇒ Se **sì**, la funzione è **calcolabile** e il problema è **decidibile**

Ciò implica che sia possibile scrivere un programma che per ogni possibile ingresso sia in grado di calcolare il valore corretto dell'uscita. Se per qualche valore dell'ingresso l'uscita non è definita, è sufficiente far entrare il programma in un loop infinito.

15.6.3 Riduzioni

- Esiste un problema **indecidibile** che è un caso particolare del problema in analisi
  ⇒ Il problema in analisi è **indecidibile**
- Esiste un problema **decidibile** che è un caso particolare del problema in analisi
  ⇒ Il problema in analisi è **indecidibile**

15.6.4 Applicazione del teorema di Rice

Il teorema di Rice può essere applicato nei casi in cui si deve verificare se un programma *(o analogamente una TM o un algoritmo)*:

- Ha una data **proprietà** relativa alla funzione da esso calcolata
- Calcola una **funzione** tra quelle di un insieme dato

Infatti, se:

- L'insieme di funzioni identificato **non è banale**
  ⇒ Il problema in analisi è **indecidibile**
- L'insieme di funzioni identificato **è banale**
  ⇒ Il problema in analisi è **decidibile**

Si ricordi che un insieme di funzioni è banale se è **vuoto** o se è l'insieme di **tutte le funzioni computabili**.

15.6.5 Ricorsività di un insieme S di numeri naturali

- S è **finito**
  ⇒ S è **ricorsivo**
- S è **infinito**
  - La funzione caratteristica di S è **computabile**
    ⇒ S è **ricorsivo**
  - S può essere espresso come insieme di indici di TM con una **proprietà comune relativa alla funzione che calcolano**
    ⇒ **Si può** usare il teorema di Rice
  - S può essere espresso come insieme di indici di TM con una **proprietà comune non relativa alla funzione che calcolano**
    ⇒ **Non si può** usare il teorema di Rice

15.7 Complessità

- Tutti i logaritmi sono nello stesso Θ si può non omettere la **base**
  - la formula di cambio base comporta un fattore moltiplicativo
  - è possibile non indicare la base del logaritmo quando si valuta l'andamento asintotico
- Contrariamente a quanto avviene nei logaritmi, negli esponenziali è importante specificare la **base**
- A meno che non sia esplicitamente indicato, negli esercizi con pseudocodice si usa il criterio di costo costante

15.7.1 Complessità dei cicli

Si supponga che un ciclo viene eseguito n volte. Se, ad ogni ciclo, il contatore:

- Viene incrementato di un valore costante `k`: la complessità è $T(n) = \Theta(n)$
- Viene moltiplicato per un valore costante `k`: la complessità è $T(n) = \Theta\left(\log_{\mathtt{k}}(n)\right) = \Theta\left(\log(n)\right)$
- Viene elevato ad un valore costante `k`: la complessità è $T(n) = \Theta\left(\log\left(\log(n)\right)\right)$

15.8 Complessità delle operazioni sui grafi

La complessità delle operazioni comuni sui grafi *(studiati nella Sezione 11.1.1)*, nei loro rispettivi casi peggiori, sono mostrate nella Tabella 11.

tipo di albero	*ricerca*	*inserimento*	*cancellazione*
albero binario	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
BST	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
RB tree	$\mathcal{O}(h) = \mathcal{O}(\log(n))$	$\mathcal{O}(h) = \mathcal{O}(\log(n))$	$\mathcal{O}(h) = \mathcal{O}(\log(n))$

Tabella 11: Complessità delle operazioni comuni sui grafi - **caso peggiore**

16 Codice relativo agli algoritmi

In questa sezione vengono presentati gli algoritmi mostrati durante il corso, implementati in C.

16.1 Algoritmi di ordinamento

```c
/**
 * @brief Swap two elements of an array.
 * Complexity: O(1)
 *
 * @param a array containing the elements to swap
 * @param p index of the first element
 * @param q index of the second element
 */
void swap(int *a, int p, int q)
{
  if (p == q)
    return;

  int t = a[p];
  a[p] = a[q];
  a[q] = t;
}

/**
 * @brief Support function for build_max_heap. Fix a node in a max heap.
 * Complexity: O(log n).
 *
 * @param a array containing the heap
 * @param i index of the node to fix
 * @param len length of the array
 */
void max_heapify(int *a, int i, int len)
{
  // left and right children of the node
  int left = 2 * i + 1;
  int right = 2 * i + 2;
  // largest child of the node
  int max = i;

  // check if the children are larger than the node
  if (left < len && a[left] > a[i])
    max = left;
  if (right < len && a[right] > a[max])
    max = right;

  // if the node is not the largest, swap it with the largest child
  if (max != i)
  {
    swap(a, i, max);
    // apply the same procedure to the child
    max_heapify(a, max, len);
  }
}

/**
 * @brief Create a max heap from an array.
 * Complexity: O(n log n).
 *
 * @param sequence array to be converted
 * @param len length of the array
 */
void build_max_heap(int *sequence, int len)
{
  // iterate over the internal nodes, as the leaves
  // are already in the correct order
  for (int i = len / 2 - 1; i >= 0; i--)
    max_heapify(sequence, i, len);
```

```
}

/**
 * @brief Heap sort.
 * Complexity: O(n log n).
 *
 * @param sequence array to be ordered
 * @param len length of the array
 */
void heap_sort(int *sequence, int len)
{
  // in a max heap, the root is the largest element
  build_max_heap(sequence, len);
  for (int i = len - 1; i > 0; i--)
  {
    // take root (first element) and put it in the last position
    swap(sequence, 0, i);
    // fix the heap discarding the last element
    max_heapify(sequence, 0, i);
  }
}

/**
 * @brief Insertion sort.
 * Complexity: O(n^2).
 *
 * @param sequence array to sort
 * @param len length of the array
 */
void insertion_sort(int *sequence, int len)
{
  // iterate over each element of the array
  for (int j = 1; j < len; j++)
  {
    int key = sequence[j];
    int i = j - 1;

    // find the first element that is smaller than the key
    while (i >= 0 && sequence[i] > key)
    {
      sequence[i + 1] = sequence[i];
      i--;
    }

    // insert the key in the correct position
    sequence[i + 1] = key;
  }
}

/**
 * @brief Selection sort.
 * Complexity: O(n^2).
 *
 * @param sequence array to sort
 * @param len length of the array
 */
void selection_sort(int *sequence, int len)
{
  // iterate over the array
  for (int i = 0; i < len - 1; i++)
  {
    int min = i;

    for (int j = i + 1; j < len; j++)
    {
      // find the first place where the element is smaller than the current
      if (sequence[j] < sequence[min])
        min = j;
    }
    swap(sequence, i, min);
```

```
  }
}

/**
 * @brief Bubble sort.
 * Complexity: O(n^2).
 *
 * @param sequence array to order
 * @param len length of the array
 */
void bubble_sort(int *sequence, int len)
{
  // iterate over the array
  for (int i = 0; i < len - 1; i++)
  {
    // compare each element with the next one
    for (int j = 0; j < len - i - 1; j++)
    {
      // if the current element is larger than the next one, swap them
      if (sequence[j] > sequence[j + 1])
        swap(sequence, j, j + 1);
    }
  }
}

/**
 * @brief Support function for the merge sort.
 * Complexity: O(n).
 *
 * @param sequence array to order
 * @param p index of the first element
 * @param q index of the middle element
 * @param r index of the last element
 */
void merge(int *sequence, int p, int q, int r)
{
  // left and right portions of the array
  int n1 = q - p + 1;
  int n2 = r - q;

  // create two temporary arrays
  int L[n1 + 1], R[n2 + 1];

  // copy the elements to the temporary arrays
  for (int i = 0; i < n1; i++)
    L[i] = sequence[p + i];

  for (int i = 0; i < n2; i++)
    R[i] = sequence[q + 1 + i];

  // insert the sentinel element
  L[n1] = INT_MAX;
  R[n2] = INT_MAX;

  // initialize the two pointers
  int i = 0;
  int j = 0;

  // merge the two arrays
  for (int k = p; k <= r; k++)
  {
    if (L[i] <= R[j])
    {
      // the item on the left is smaller
      sequence[k] = L[i];
      // increase the position of the left pointer
      i++;
    }
    else
    {
```

```
      // the item on the right is smaller
      sequence[k] = R[j];
      // increase the position of the right pointer
      j++;
    }
  }
}

/**
 * @brief Merge sort.
 * Complexity: O(n log n).
 *
 * @param sequence array to order
 * @param p index of the first element
 * @param r index of the last element
 */
void merge_sort(int *sequence, int p, int r)
{
  if (p < r)
  {
    // find the middle element
    int q = (p + r) / 2;
    // sort the two positions
    merge_sort(sequence, p, q);
    merge_sort(sequence, q + 1, r);
    // merge the two sorted portions
    merge(sequence, p, q, r);
  }
}

/**
 * @brief Support function of the quick sort.
 * Complexity: O(n).
 *
 * @param sequence array to order
 * @param p index of the first element
 * @param r index of the last element
 * @return pivot position
 */
int partition(int *sequence, int p, int r)
{
  // choose the last element as the pivot
  int x = sequence[r];
  // initialize the pointer for the biggest element
  int i = p - 1;
  for (int j = p; j < r; j++)
  {
    // if the current element is smaller than the pivot
    if (sequence[j] <= x)
    {
      // increase the position of the pointer
      i++;
      // swap the current element with the one on the left
      swap(sequence, i, j);
    }
  }

  swap(sequence, i + 1, r);
  return i + 1;
}

/**
 * @brief Quick sort.
 * Complexity: O(n log n).
 *
 * @param sequence array to order
 * @param p index of the first element
 * @param r index of the last element
 */
void quick_sort(int *sequence, int p, int r)
```

```
{
  if (p < r)
  {
    // find the pivot position
    int q = partition(sequence, p, r);
    // sort the left and right portions
    quick_sort(sequence, p, q - 1);
    quick_sort(sequence, q + 1, r);
  }
}

/**
 * @brief Counting sort.
 * Complexity: O(n + k).
 *
 * @param sequence array to order
 * @param len length of the array
 * @param ordered array to store the ordered sequence
 * @param max maximum value of the array
 */
void counting_sort(int *sequence, int len, int *ordered, int max)
{
  int C[max + 1];

  // initialize C
  for (int i = 0; i <= max; i++)
    C[i] = 0;

  // count the elements = i
  for (int j = 0; j < len; j++)
    C[sequence[j]]++;

  // count the elements <= i
  for (int i = 1; i <= max; i++)
    C[i] += C[i - 1];

  for (int j = len; j >= 0; j--)
  {
    // order the elements according to the position stored in C
    ordered[C[sequence[j - 1]] - 1] = sequence[j - 1];
    // decrease the position of the element
    C[sequence[j - 1]]--;
  }
}
```