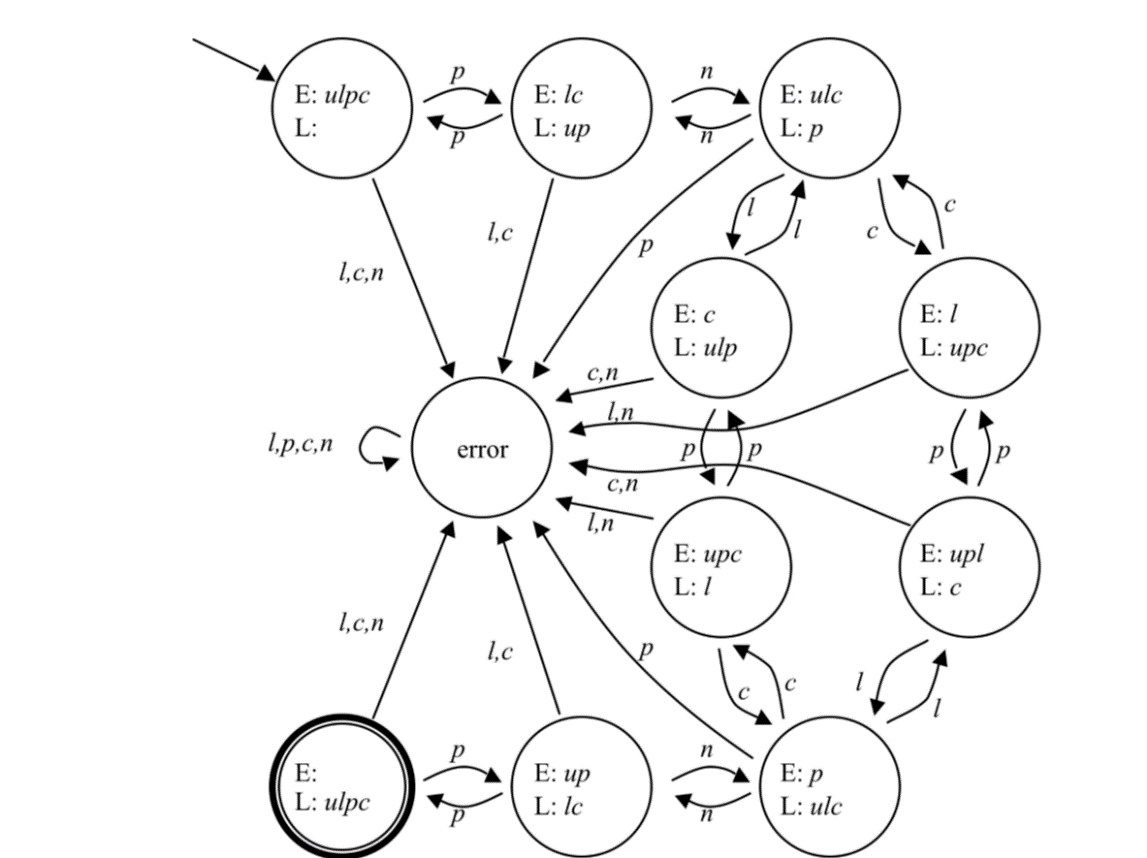
**Elementi di Teoria della Computazione**

***Sipser, Capitolo 0 –* Alfabeti, Stringhe, Linguaggi**

**Def.** Un *insieme* è una collezione non ordinata di oggetti o elementi.  
Gli insiemi sono scritti tra { }, mentre gli elementi sono inseriti tra le parentesi, separati da una virgola.  
 **Def.** Per ogni insieme S, *w ∈ S* indica che w è un elemento di S.  
 **Ordine e ridondanza non contano**: {a, b, c} ha elementi a, b, c. {a, b, c} e {b, a, b, c, c} sono lo stesso insieme. {a} ed a sono cose diverse: {a} è un insieme che contiene solo l’elemento a.  
 **Es.** L’ insieme dei numeri dispari è {1, 3, 5, 7, 9, 11, 13, ...} = {2n+1 | n = 0, 1, 2, ...} = {2n+1 | n ∈ N}.  
**Es.** Se A = {2n | n ∈ N}, allora 4 ∈ A, ma 5 ∉ A.  
  
**Def.** La *cardinalità* |S| di S è il numero di elementi in S.  
**Es.** Se S = {ab, bb}, allora |S| = 2. Se T = {an | n > 1}, allora |T| = ∞. Se T = ∅, allora |T| = 0.  
  
**Def.** Un insieme S è *ﬁnito* se |S| < ∞. Se S non è ﬁnito, allora è detto *inﬁnito*.  
**Es.** Se S = {ab, bb}, allora |S| = 2 e S è ﬁnito. Se T = {an | n > 1}, allora |T| = ∞ e T è inﬁnito.  
  
Un **alfabeto** è un insieme ﬁnito di elementi fondamentali (chiamati lettere o simboli).  
**Es.** L’ alfabeto binario è Σ = {0, 1}.  
  
Una **stringa** su un alfabeto è una sequenza ﬁnita di simboli dell’alfabeto.  
**Es.** 0131 è una stringa sull’ alfabeto B = {0, 1, 2, ..., 9}.  
  
Data una stringa s, la **lunghezza** di s è il numero di simboli in s. La lunghezza di s è denotata con *lunghezza(s)* o |s|.  
**Es.** lunghezza(mom) = |mom| = 3. La stringa vuota ε è la stringa contenente nessun simbolo, |ε| = 0.   
  
**Def.** Dato un alfabeto Σ, la *chiusura di Kleene di Σ* è Σ\*: l’ insieme di tutte le possibili stringhe su Σ.  
**Es.** Se Σ = {a, b}, allora Σ\* = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, abb, ...}.  
  
**Def.** Date due stringhe u e v, la concatenazione di u e v è la stringa uv.  
**Es.** Se u = abb e v = ab, allora uv = abbab e vu = ababb.  
 Se u = ε e v = ab, allora uv = ab.  
 Se u = bb e v = ε, allora uv = bb.  
 Se u = ε e v = ε, allora uv = ε; cioè εε = ε.  
  
**Def.** Per una stringa w, deﬁniamo wn per n ≥ 0 induttivamente:   
 w0 = ε,  
 wn+1 = wnw, per ogni n ≥ 1.  
**Es.** Se w = cat, allora: w0 = ε, w1 = cat, w2 = catcat, w3 = catcatcat, ...  
  
**Def.** Data una stringa s, una *sottostringa* di s è una qualsiasi parte di simboli consecutivi della stringa s, cioè w è una sottostringa di s se esistono stringhe x e y (eventualmente vuote) tali che **s = xwy**.  
**Es:** 567 è una sottostringa di 56789.  
 567 è una sottostringa di 45678.  
 567 è una sottostringa di 34567.  
**Es.** La stringa 472 ha sottostringhe ε, 4, 7, 2, 47, 72, 472; ma 42 non è sottostringa di 472.  
  
**Def.** Un *Linguaggio formale* (Linguaggio) è un insieme di stringhe su un alfabeto.  
 **Es.** Linguaggi per computer, quali C, C++ o Java, sono linguaggi formali con alfabeto {a, b, ..., z, A, B, ..., Z, 0, 1, 2, ..., 9, >, <, =, +, −, ∗, /, (, ), …}. Le regole della sintassi deﬁniscono le regole del linguaggio. L’insieme di nomi validi di variabili è un linguaggio formale.  
**Es.** Alfabeto A = {x}. Linguaggio L = {ε, x, xx, xxx, xxxx, ...} = {xn|n = 0, 1, 2, 3, ...}.  
Nota. x0 = ε, quindi stringa vuota in L.  
**Es.** Alfabeto A = {x}. Linguaggio L = {x, xxx, xxxxx, ...} = {x2n+1|n = 0, 1, 2, 3, ...}.  
**Es.** Alfabeto A = {0, 1, 2, ..., 9}. Linguaggio L = {qualsiasi stringa che non inizia con 0} = {ε, 1, 2, ..., 9, 10, ...}.  
**Es.** Sia A = {a, b}, deﬁniamo un Linguaggio L formato da tutte le stringhe che iniziano con a seguita da 0 o più b; cioè L = {a, ab, abb, abbb, ...} = {abn|n ≥ 0}.  
**Nota.** L’insieme vuoto ∅ è l’ insieme che non contiene alcun elemento. ∅ ≠ {ε} poichè ∅ non ha elementi. In generale, ε ∉ ∅.  
  
**Def.** Siano S e T insiemi. Diciamo che S ⊆ T (S *sottoinsieme* di T) se w ∈ S implica w ∈ T (cioè, se ogni elemento di S è anche un elemento di T).  
**Es.** Se S = {ab, ba} e T = {ab, ba, aaa}, allora S ⊆ T ma T ⊄ S.  
 Se S = {ba, ab} e T = {aa, ba}, allora S ⊄ T e T ⊄ S.  
  
**Def.** Due insiemi S e T sono *uguali* (S = T) se S ⊆ T e T ⊆ S.  
**Es.** Siano S = {ab, ba} e T = {ba, ab}; allora S ⊆ T e T ⊆ S; quindi S = T.  
 Siano S = {ab, ba} e T = {ba, ab, aaa}; allora S ⊆ T, ma T ⊄ S; quindi S ≠ T.  
  
**Def.** Dati due insiemi S e T, la loro *unione* S ∪ T = {w | w ∈ S oppure w ∈ T}.  
S ∪ T contiene tutti gli elementi contenuti in S oppure in T (o in entrambi).  
**Es.** Se S = {ab, bb} e T = {aa, bb, a}, allora S ∪ T = {ab, bb, aa, a}.  
 Se S = {a, ba} e T = ∅, allora S ∪ T = S.  
 Se S = {a, ba} e T = {ε}, allora S ∪ T = {ε, a, ba}.  
  
**Def.** Dati due insiemi S e T, la loro *intersezione* S ∩ T = {w | w ∈ S e w ∈ T}.  
S ∩ T contiene tutti gli elementi comuni ad S e T.  
**Def.** Due insiemi S e T si dicono *disgiunti* se S ∩ T = ∅.  
**Es.** Siano S = {ab, bb} e T = {aa, bb, a}; allora S ∩ T = {bb}.  
 Siano S = {ab, bb} e T = {aa, ba, a}; allora S ∩ T = ∅, quindi S e T sono disgiunti.  
  
**Lemma.** Se S e T sono disgiunti (cioè S ∩ T = ∅), allora |S ∪ T| = |S| + |T|.  
**Lemma.** Se S e T sono tali che S ∩ T < ∞, allora |S ∪ T| = |S| + |T| − |S ∩ T|.  
  
**Def.** Dati due insiemi S e T, la loro *sottrazione* S − T = {w | w ∈ S e w ∉ T}.  
**Es.** Siano S = {a, b, bb, bbb} e T = {a, bb, bab}; allora S − T = {b, bbb}.  
 Siano S = {ab, ba} e T = {ab, ba}; allora S − T = ∅.  
  
**Def.** Dato un insieme universale *U*, il *complemento* di un insieme S ⊆ U è *C(S)* = {w | w ∈ U, w ∉ S}.  
C(S) è l’ insieme di tutti gli elementi considerati (elementi di U) che non sono in S (quindi C(S) = U − S).  
**Es.** Siano U l’insieme delle stringhe su alfabeto {a, b}, e S l’insieme delle stringhe su alfabeto {a, b} che iniziano con b; allora C(S) è l’insieme delle stringhe su alfabeto {a, b} che non iniziano con b.  
N.B.: NON l’insieme stringhe che iniziano con a (es. stringa vuota ε).  
  
**Def.** Dati 2 insiemi S e T di stringhe, la *concatenazione* (o prodotto) di S e T è ST = {uv | u ∈ S, v ∈ T}.  
ST è l’ insieme di stringhe che possono essere divise in 2 parti: la prima parte coincide con una stringa in S, e la seconda parte coincide con una stringa in T.  
**Es.** Se S = {a, aa} e T = {ε, a, ba}, allora ST = {a, aa, aba, aaa, aaba} e TS = {a, aa, aaa, baa, baaa}.  
aba ∈ ST, ma aba ∉ TS. Quindi ST ≠ TS.  
  
**Def.** Una *sequenza* di oggetti è una lista di questi oggetti in qualche ordine.  
Ordine e ridondanza sono importanti in una sequenza (non in un insieme).  
  
**Def.** Sequenze ﬁnite sono dette *tuple*. Una k-tupla ha k elementi nella sequenza.  
**Es.** (4, 2, 7) è una 3-tupla o tripla.  
 (9, 23) è una 2-tupla o coppia.  
  
**Def.** Dati due insiemi A e B, il *prodotto cartesiano* A × B è l’insieme di coppie A × B = {(x, y) | x ∈ A, y ∈ B}.  
**Es.** Siano A = {a, ba, bb} e B = {ε, ba}; allora:  
- A × B = {(a, ε), (a, ba), (ba, ε), (ba, ba), (bb, ε), (bb, ba)};  
- B × A = {(ε, a), (ε, ba), (ε, bb), (ba, a), (ba, ba), (ba, bb)}.  
Nota. (ba, a) ∈ B × A, ma (ba, a) ∉ A × B; quindi, B × A ≠ A × B.  
Nota. Il prodotto cartesiano è diverso dalla concatenazione: AB = {a, aba, ba, baba, bb, bbba} ≠ A × B.  
  
Possiamo anche deﬁnire il prodotto cartesiano di più di 2 insiemi. A1 × ... × Ak+l è l’insieme di k-tuple  
 A1 × ... × Ak = {(x1, ..., xk) | xi ∈ Ai, 1 ≤ i ≤ k}.  
  
**Es.** Siano A1 = {ab, ba, bbb}, A2 = {a, bb}, A3 = {ab, b}. Allora A1 × A2 × A3 = {(ab, a, ab), (ab, a, b), (ab, bb, ab), (ab, bb, b), (ba, a, ab), (ba, a, b), (ba, bb, ab), (ba, bb, b), (bbb, a, ab), (bbb, a, b), (bbb, bb, ab), (bbb, bb, b)}.  
  
**Def.** Per ogni insieme S, *l’insieme potenza* *P(S)* è P(S) = {A | A ⊆ S}.  
Cioè, è l’insieme di tutti possibili sottoinsiemi di S (inclusi ∅ e S stesso).  
**Es.** Se S = {a, bb}, allora P(S) = {∅, {a}, {bb}, {a, bb}}.  
  
**Lemma.** Se |S| < ∞ , allora |P(S)| = 2|S|.  
Cioè, ci sono 2|S| diﬀerenti sottoinsiemi di S.  
  
**Def.** Dato un insieme S di stringhe, sia  
 S0 = {ε},  
 Sk = {w1w2 ... wk | wi ∈ S, i = 1, 2, ..., k} = SS...S, k > 1,  
ossia la *concatenazione* di S con se stesso per k volte.  
Nota. Sk è l’insieme di stringhe ottenute concatenando k stringhe di S, con possibili ripetizioni. In particolare, S1 = S.  
**Es.** Se S = {a, bb}, allora:  
- S0 = {ε};  
- S1 = {a, bb};  
- S2 = {aa, abb, bba, bbbb};  
- S3 = {aaa, aabb, abba, abbbb, bbaa, bbabb, bbbba, bbbbbb}.  
  
**Def.** La *chiusura* (o *Kleene star*) di un insieme di stringhe S è S\* = S0 ∪ S1 ∪ S2 ∪ S3 ∪ ...  
Nota. S\* è l’insieme di tutte le stringhe ottenute concatenando zero o più stringhe di S, potendo usare la stessa stringa più volte.  
 S\* = {w1w2 ... wk | k ≥ 0, wi ∈ S, i = 1, 2, ..., k},  
dove per k = 0, la stringa w1w2 ... wk = ε è la stringa vuota.  
  
**Es.** Se S = {ba, a}, allora S\* = {ε, a, aa, ba, aaa, aba, baa, aaaa, aaba, ...}.  
**Es.** Se A = {a, b}, allora A∗ = {ε, a, b, aa, ab, ba, bb, aaa, aab, aba, ...}, tutte le possibili stringhe su alfabeto A.  
**Es.** Se S = ∅, allora S\* = {ε}.  
**Es.** Se S = {ε}, allora S\* = {ε}.  
  
**(S\*)\*** è l’ insieme di stringhe formate concatenando stringhe di S\*.  
Nota. (S\*)\* = S\* per ogni insieme S di stringhe.  
  
**S+** è l’insieme di stringhe formate concatenando una o più stringhe di S.  
**Es.** Se S = {x}, allora S+ = {x, xx, xxx, xxxx, ...}.  
  
Per ogni stringa w, l’**inverso** di w, scritto *reverse(w)* o wR, è la stessa stringa di simboli scritta in ordine inverso.  
Se w = w1w2 ... wn, dove ogni wi è un simbolo, allora wR = wnwn−1 ... w1.  
**Es.** (cat)R = tac.  
**Es.** εR = ε.

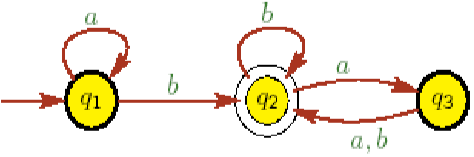
**Automi finiti**

Un problema classico per essere introdotti agli automi può essere il seguente. Un uomo viaggia con un lupo, una pecora ed un cavolo: egli vuole attraversare il fiume con una barca che può contenere soltanto l’uomo più uno dei suoi possedimenti; inoltre, il lupo mangia la pecora se soli insieme, così come la pecora mangia il cavolo se soli insieme. Come può attraversare il fiume senza perdite?  
Le *mosse* possono essere rappresentate da simboli: uomo attraversa con lupo (*l*), uomo attraversa con pecora (*p*), uomo attraversa con cavolo (*c*), uomo attraversa con niente (*n*). La sequenza delle mosse può essere rappresentata da una stringa; un esempio di soluzione è la stringa ***p****n****l****p****c****n****p*** (in grassetto quando l’uomo attraversa con, viceversa quando l’uomo ritorna con): prima attraversa con pecora, poi ritorna con niente, poi attraversa con lupo, …  
Un problema, risolvibile da un automa, può essere raffigurato mediante un **diagramma delle transizioni**, principalmente costituito da stati e transizioni. Ogni mossa porta il puzzle da uno stato ad un altro: si dice che la mossa provoca una *transizione*. Una mossa è detta *legale* se ciò che ne consegue non causa un errore (in questo esempio, lasciare lupo e pecora oppure pecora e cavolo assieme). L’insieme degli stati *raggiungibili* è costituito da tutti gli stati in cui può trovarsi un determinato problema. Ogni stato è raffigurato graficamente da un cerchio, mentre le transizioni tra stati sono raffigurate da una freccia da uno stato all’altro. Lo stato *iniziale* (start) è lo stato iniziale del problema, ed è contraddistinto da una freccia che parte dal vuoto. Lo stato *finale* (scopo/obiettivo raggiunto) è lo stato in cui un problema è considerato risolto, ed è contraddistinto da un doppio cerchio.  
Come detto in precedenza, un *cammino* può essere rappresentato da una stringa w ∈ {*l*, *p*, *c*, *n*}\*. Una sequenza di mosse è una *soluzione* se e solo se la stringa corrispondente è un cammino dallo stato iniziale allo stato finale del diagramma. Il diagramma definisce il *linguaggio delle soluzioni*:  
 {x ∈ {*l*, *p*, *c*, *n*}\* | iniziando in stato start e seguendo transizioni di x, terminano nello stato finale}.  
Nell’esempio specifico, il problema è trovare una sequenza di mosse che permette di trasportare capra, cavolo e lupo; il linguaggio corrispondente è l’insieme delle stringhe sull’alfabeto {*l*, *p*, *c*, *n*}. Le stringhe corrispondono a sequenze di mosse. Tra esse cerchiamo una stringa che corrisponde ad una soluzione del problema.  
Per alcune stringhe, l’automa non sa cosa fare: vogliamo automi che sappiano sempre cosa fare. Ad esempio, possiamo usare uno stato addizionale che raffigura un errore. Di seguito è posto il diagramma delle transizioni del problema in questione.



In teoria dei sistemi dinamici, un **automa** è un sistema dinamico discreto (nella scansione del tempo e nella descrizione del suo stato) e tempo-invariante (il sistema si comporta alla stessa maniera indipendentemente dall'istante di tempo in cui agisce).  
Quando l'automa si trova in un dato *stato*, esso può accettare solo un sottoinsieme dei simboli del suo alfabeto. L'evoluzione di un automa parte da un particolare stato detto **stato iniziale**. Un sottoinsieme privilegiato dei suoi stati è detto insieme degli **stati finali** o *marcati*.  
In genere gli automi sono **deterministici**, ovvero dato uno stato ed un simbolo in ingresso è possibile una sola transizione. Esistono comunque anche automi non deterministici, o stocastici.

Nel diagramma dell’esempio precedente c’è esattamente una transizione per ogni stato e per ogni lettera nell’alfabeto Σ. Esso fornisce una *procedura computazionale* per decidere se una stringa è una soluzione o meno:  
- parti nello stato start;  
- segui una transizione per ogni simbolo in input;  
- se alla fine arrivi nello stato obiettivo, allora accetta altrimenti rifiuta l’input.  
Un automa **finito** (talvolta noto come “macchina a stati finiti”) è un modello semplice di calcolatore avente una quantità finita di memoria. L’idea di base del funzionamento è la seguente: si ha in input una stringa w su un alfabeto Σ, si leggono i simboli di w da sinistra a destra e, dopo aver letto l’ultimo simbolo, esso indica se accetta o rifiuta la stringa w.

La figura a lato riguarda un esempio di automa  
finito deterministico (DFA) con alfabeto Σ = {a,  
b}, dove la stringa arriva in input ed esso legge un  
simbolo per volta, dal primo all’ultimo,  
accettando o rifiutando la stringa.  
q1, q2 e q3 sono gli stati: q1 è lo stato start e q2 è  
uno stato accetta. Gli archi dicono come muoversi  
quando l’automa si trova in uno stato ed un simbolo di Σ viene letto. Dopo aver letto l’ultimo simbolo: se DFA è in uno stato accetta, allora la stringa è accettata, altrimenti è rifiutata. Ad esempio, la stringa abaa è accettata, aba è rifiutata, ε è rifiutata.

**Def.** Un *automa finito deterministico* (DFA, detto anche “automa finito”) A è una 5-tupla M = (Q, Σ, f, q0, F), dove:  
1. Q è un insieme finito di stati;  
2. Σ è un alfabeto, e il DFA processa stringhe su Σ;  
3. f : Q × Σ → Q è la funzione di transizione;  
4. q0 ∈ Q è lo stato start;  
5. F ⊆ Q è l’insieme di stati accetta (o finali).

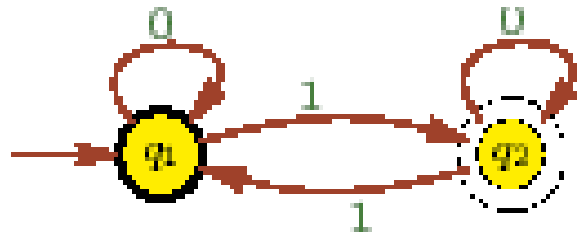
La **funzione di transizione** di un DFA f : Q × Σ → Q, per ogni stato e per ogni simbolo di input, dice in quale stato spostarsi. Cioè, f(r, a) è lo stato che il DFA occupa quando, trovandosi nello stato r, legge a, per stato r ∈ Q ed ogni simbolo a ∈ Σ.  
Nella figura precedente, (q1, b) → q2, in quanto se ci si trova nello stato q1 e si ha in input b, allora si passa allo stato q2. Inoltre, esiste esattamente un arco uscente da r con label a, quindi la macchina è **deterministica**: una macchina è deterministica se il suo comportamento è specificatamente definito.  
In sintesi, M = (Q, Σ, f, q1, F), con Q = {q1, q2, q3}, Σ = {a, b}, f è descritta dalla tabella a lato,  
q1 è lo stato start, F = {q2}.

Usando un linguaggio più formale, un DFA esibisce il seguente funzionamento (**computazione**).  
Sia M = (Q, Σ, f, q1, F) un DFA. Esso considera una stringa w = w1w2 ... wn su Σ, dove ogni wi è in Σ. M *accetta* w se esiste una sequenza di stati r0, r1, …, rn in Q tali che:  
1. r0 = q0 (il primo stato della sequenza è quello iniziale);  
2. rn in F (l’ultimo stato in sequenza è uno stato accetta);  
3. f(ri-1, wi) = ri, per ogni i = 1, 2, …, n (la sequenza di stati corrisponde a transizioni valide per la stringa w).

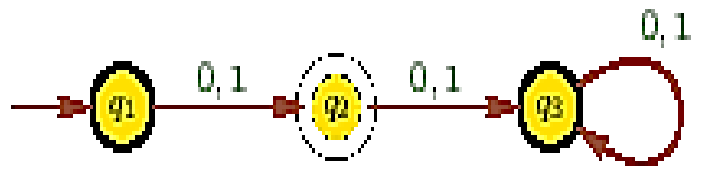
Nell’esempio in questione, la stringa abaa è accettata se e solo se esiste una sequenza di stati q1, q1, q2, q3, q2.

**Def.** Se L è l’insieme di tutte le stringhe che la macchina M accetta, allora si dice che L è il *linguaggio della macchina* M, e M *riconosce* (o *accetta*) L.  
Scriviamo anche L(M) = L.

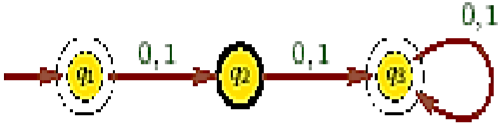
**Def.** Un linguaggio è *regolare* se è riconosciuto da qualche DFA.  
**Es.** Nell’esempio precedente, L(M) è l’insieme di tutte le stringhe sull’alfabeto {a, b} della forma {a}\*{b}{b,aa,ab}\*.

**Es.** Si consideri il DFA M1 con alfabeto Σ = {0, 1}, posto a lato.  
La stringa 010110 è accettata, ma 0101 è rifiutata. L(M1) è il linguaggio di  
stringhe su Σ in cui il numero totale di 1 è dispari.

**Esercizio.** Dare un DFA che riconosce il linguaggio di stringhe su Σ con  
un numero pari di 1.  
🡪 Basta invertire gli stati accetta in stati non accetta, e viceversa, del DFA M1.

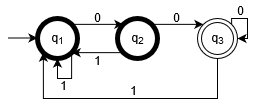
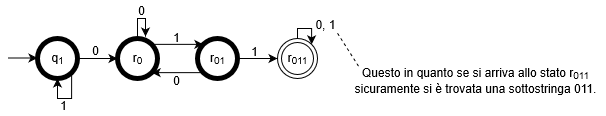
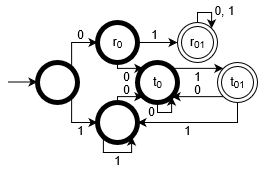
**Es.** Si consideri il DFA M2 con alfabeto Σ = {0, 1}, posto a lato.  
L(M2) è il linguaggio su Σ dell’insieme delle stringhe aventi  
lunghezza 1, cioè L(M2) = {w in Σ\* | |w| = 1}.

Si ricordi che C(L(M2)), il *complemento* di L(M2), è l’insieme di  
stringhe su Σ che non sono in L(M2).

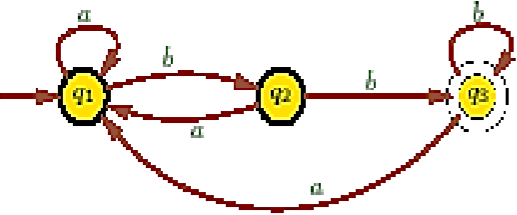
**Es.** Si consideri il DFA M3 con alfabeto Σ = {0, 1}, posto a lato.  
L(M3) è il linguaggio su Σ dell’insieme delle stringhe aventi  
lunghezza diversa da 1, cioè L(M2) = {w in Σ\* | |w| ≠ 1}.  
Si noti che è presente più di uno stato accetta, e che anche lo stato  
start è uno stato accetta. *In generale, un DFA accetta ε se lo stato start è anch’esso uno stato accetta.*

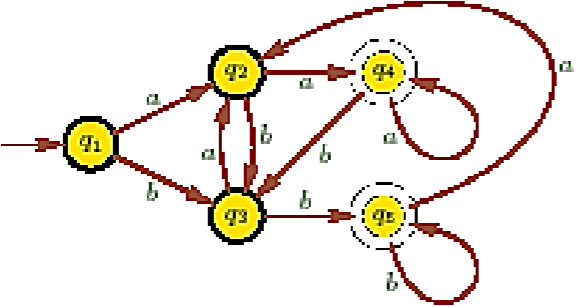
La funzione di transizione f può essere **estesa** a f\*, una funzione che opera su stati e *stringhe* (anziché su stati e simboli). In generale, è possibile definirla induttivamente:  
- f\*(q, ε) = q;  
- f\*(q, xa) = f(f\*(q, x), a).  
Il linguaggio accettato da M è, quindi, L(M) = {w | f\*(q0, w) ∈ F}.  
Informalmente, la funzione di transizione estesa dà come risultato lo stato dell’automa dopo aver letto tutti i simboli di una stringa, partendo da un dato stato dell’automa.

**Es.** Si consideri il DFA M1 precedentemente definito. La funzione di transizione estesa f\* è:  
- f\*(q1, ε) = q1;  
- f\*(q1, 001) = f(f\*(q1, 00), 1) *= f(q1, 1) = q2*; // si è proceduto a ritroso  
- f\*(q1, 00) = f(f\*(q1, 0), 0) *= f(q1, 0) = q1*; // si è proceduto a ritroso  
- f\*(q1, 0) = f(f\*(q1, ε), 0) = f(q1, 0) = q1.

**Esercizio.** Fornire DFA per i seguenti linguaggi sull’alfabeto {0, 1}:  
1. Insieme di tutte le stringhe che terminano con 00.  
  
2. Insieme di tutte le stringhe con tre zeri consecutivi.  
-  
  
3. Insieme delle stringhe con 011 come sottostringa.  
  
  
4. Insieme delle stringhe che cominciano o finiscono (o entrambe le cose) con 01.  
 **□**

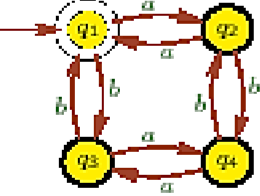
In generale, dato un DFA M per un linguaggio L, è possibile costruire DFA M’ per il **complemento** C(L) da M:  
- rendendo tutti gli stati accetta in M stati non-accetta in M’;  
- rendendo tutti gli stati non-accetta in M stati accetta in M’.  
Più formalmente, se un linguaggio L su un alfabeto Σ ha un DFA M = (Q, Σ, f, q1, F), allora il DFA per il complemento C(L) è M’ = (Q, Σ, f, q1, Q-F).

**Es.** Si consideri il DFA M4 con alfabeto Σ = {a, b}, posto a lato.  
L(M4) è il linguaggio di stringhe su Σ che terminano con bb, cioè  
L(M4) = {w in Σ\* | w = sbb per qualche stringa s}.

**Es.** Si consideri il DFA M5 con alfabeto Σ = {a, b}, posto a lato.  
L(M5) = {w | w = saa o w = sbb per qualche stringa s}.

**Es.** Si consideri il DFA M6 con alfabeto Σ = {a, b}, posto a lato. Esso accetta tutte le  
possibili stringhe su Σ, cioè L(M6) = Σ\*.  
In generale, ogni DFA in cui tutti gli stati sono stato accetta riconosce il linguaggio Σ\*.

**Es.** Si consideri il seguente DFA M7 con alfabeto Σ = {a, b}, posto a lato. Esso non  
accetta stringhe su Σ, cioè L(M7) = ∅.  
In generale, un DFA può non avere stati accetta.

**Es.** Si consideri il seguente DFA M8 con alfabeto Σ = {a, b}, posto a lato:  
- ogni a muove verso destra o sinistra.  
- ogni b muove verso l’alto o il basso.  
In sintesi, esso riconosce il linguaggio di stringhe su Σ con un numero pari di a  
ed un numero pari di b.

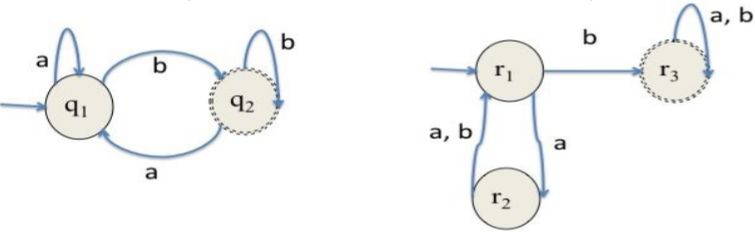
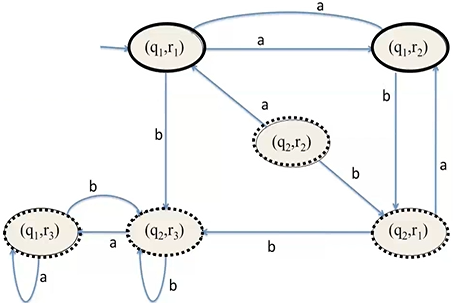
Siano A e B linguaggi. Allora:  
- A ∪ B = {w | w ∈ A o w ∈ B} è l’unione dei due linguaggi;  
- AB = {vw | v ∈ A, w ∈ B} è la concatenazione dei due linguaggi;  
- A\* = { w1w2 ... wk | k ≥ 0 e ogni wi ∈ A} è la chiusura (Kleene star) del linguaggio A.

**Def.** Una collezione S di oggetti è *chiusa* per un’operazione f se, applicando f a membri di S, tale operazione restituisce un oggetto presente in S.  
**Es.** N = 0, 1, 2, …} è chiuso per addizione, ma non per sottrazione.

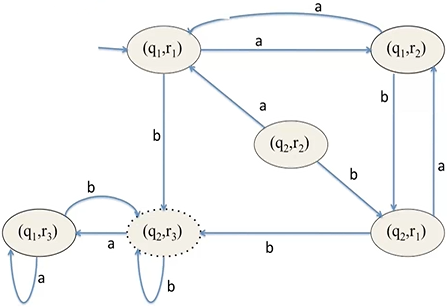
Abbiamo visto che, dato un DFA M1 per un linguaggio L, possiamo costruire un DFA M2 per un linguaggio complemento L’ rendendo tutti gli stati accetta in M1 stati non-accetta in M2, e tutti gli stati non-accetta n M1 stati accetta in M2. Quindi, L regolare ⇒ C(L) regolare.

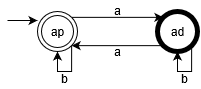
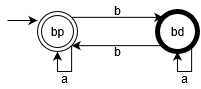
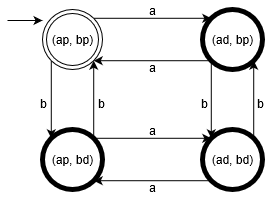
**Teorema.** *L’insieme dei linguaggi regolari è chiuso per l’operazione di complemento.*

**Teorema.** *La classe dei linguaggi regolari è chiusa per l’unione. Cioè, se L1 e L2 sono linguaggi regolari, allora anche L1 ∪ L2 è un linguaggio regolare.*  
DIM. L1 ha come DFA M1, e L2 ha come DFA M2. Una stringa w è in L1 ∪ L2 se e solo se w è accettata da M1 oppure da M2. Serve un DFA M3 che accetta w se e solo se w è accettata da M1 o M2: deve sapere se uno dei due automi accetta la stringa w in input, e per ottenere ciò è possibile porre *in parallelo* M1 e M2. Quindi, costruiamo un DFA M3 tale:  
1. da tener traccia di dove l’input sarebbe se fosse contemporaneamente in input a M1 e M2;  
2. che accetta la stringa se e solo se M1 o M2 la accetta.  
A tal punto, siano L1 e L2 definiti su uno stesso alfabeto Σ. Supponiamo che il DFA M1 riconosce L1, dove M1 = (Q1, Σ, f1, q1, F1), e che il DFA M2 riconosce L2, dove M2 = (Q2, Σ, f2, q2, F2).  
Costruiamo il DFA M3 = (Q3, Σ, f3, q3, F3) in questo modo:  
- Q3 = Q1 × Q2 = {(x, y) | x in Q1, y in Q2};  
- l’alfabeto di M3 è Σ;  
- M3 ha f3 : Q3 × Σ → Q3 tale che per ogni x in Q1, y in Q2, a in Σ: f3((x, y), a) = (f1(x, a), f2(y, a));  
- lo stato iniziale di M3 è q3 = (q1, q2) in Q3;  
- l’insieme di stati accetta di M3 è F3 = {(x, y) in Q3 | x in F1 o y in F2}.  
M3 è un DFA che riconosce l’unione. Poiché Q3 = Q1 × Q2, allora il numero di stati in M3 è |Q3| = |Q1| ∙ |Q2|. Quindi, |Q3| è finito poiché |Q1| e |Q2| sono finiti. **□**

**Es.** Si considerino i seguenti DFA e linguaggi su Σ = {a, b}:  
DFA M1 riconosce linguaggio A1 = L(M1) DFA M2 riconosce linguaggio A2 = L(M2).  
  
  
  
  
  
  
  
  
Il seguente è un DFA per l’**unione** di A1 e A2. In particolare, si procede nel seguente modo.  
1. Otteniamo gli stati dal prodotto cartesiano di Q1 e Q2.  
In particolare, lo stato iniziale è la coppia costituita dagli stati  
iniziali dei due automi;  
2a. Se ci si trova nello stato (q1, r1) e si legge a, allora si nota  
che il primo automa resta nello stato q1, mentre il secondo  
automa passa allo stato r2, quindi il risultato della funzione  
di transizione è lo stato (q1, r2) ed il nuovo automa passa  
in quest’ultimo stato;  
2b. se ci si trova nello stato (q1, r1) e si legge b, allora si nota  
che il primo automa passa allo stato q2, mentre il secondo  
automa passa allo stato r3, quindi il risultato della funzione  
di transizione è lo stato (q2, r3) ed il nuovo automa passa  
in quest’ultimo stato;  
…  
3. Otteniamo gli stati finali come coppie dove è presente o lo stato finale dell’automa A o lo stato finale dell’automa B (o entrambi). In questo esempio, gli stati finali sono: (q2, r1), (q2, r2), (q2, r3), (q1, r3).  
N.B.: Lo stato (q2, r2) non è mai raggiungibile da alcuno stato dell’automa (compreso lo stato iniziale), per cui è possibile ometterlo.

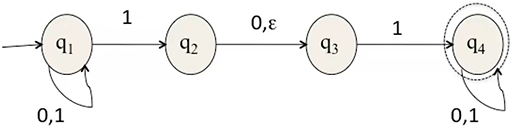
**Teorema.** *La classe dei linguaggi regolari è chiusa per l’intersezione. Cioè, se L1 e L2 sono linguaggi regolari, allora anche L1 ∩ L2 è un linguaggio regolare.*  
DIM. L1 ha come DFA M1, e L2 ha come DFA M2. Una stringa w è in L1 ∩ L2 se e solo se w è accettata sia da M1 che da M2. Serve un DFA M3 che accetta w se e solo se w è accettata da M1 e M2: deve sapere se ambedue gli automi accettano la stringa w in input, e per ottenere ciò è possibile porre *in parallelo* M1 e M2. Quindi, costruiamo un DFA M3 tale:  
1. da tener traccia di dove l’input sarebbe se fosse contemporaneamente in input a M1 e M2;  
2. che accetta la stringa se e solo sia M1 che M2 la accettano.  
A tal punto, siano L1 e L2 definiti su uno stesso alfabeto Σ. Supponiamo che il DFA M1 riconosce L1, dove M1 = (Q1, Σ, f1, q1, F1), e che il DFA M2 riconosce L2, dove M2 = (Q2, Σ, f2, q2, F2).  
Costruiamo il DFA M3 = (Q3, Σ, f3, q3, F3) in questo modo:  
- Q3 = Q1 × Q2 = {(x, y) | x in Q1, y in Q2};  
- l’alfabeto di M3 è Σ;  
- M3 ha f3 : Q3 × Σ → Q3 tale che per ogni x in Q1, y in Q2, a in Σ: f3((x, y), a) = (f1(x, a), f2(y, a));  
- lo stato iniziale di M3 è q3 = (q1, q2) in Q3;  
- l’insieme di stati accetta di M3 è F3 = {(x, y) in Q3 | x in F1 e y in F2}.  
M3 è un DFA che riconosce l’intersezione. Poiché Q3 = Q1 × Q2, allora il numero di stati in M3 è |Q3| = |Q1| ∙ |Q2|. Quindi, |Q3| è finito poiché |Q1| e |Q2| sono finiti. **□**

**Es.** Presi gli stessi automi, linguaggi ed alfabeto dall’  
esempio mostrato in precedenza, si nota che un automa  
che riconosce l’**intersezione** di A1 e A2 cambia soltanto  
gli stati finali rispetto all’unione. Bisogna, cioè, ottenere  
un automa che accetti stringhe che verrebbero accettate  
sia da M1 sia da M2: quindi, gli stati finali sono le coppie  
dov’è presente sia lo stato finale di A1 sia lo stato finale  
di A2; lo stato finale è, in questo caso, (q2, r3).

**Es.** Si consideri l’automa A che riconosce tutte le stringhe aventi un numero pari di a ed un numero pari di b. Cioè, L(A) = {w | w ha un numero pari di a AND w ha un numero pari di b}.  
Siccome L(A) è risultato di un’intersezione, lo scriviamo in questo modo:  
L(A) = {w | w ha un numero pari di a} ∩ {w | w ha un numero pari di b}. Essendo un’intersezione, è possibile costruire prima l’automa che riconosce tutte le stringhe aventi un numero pari di a, così come l’automa che riconosce tutte le stringhe aventi un numero pari di b, per poi costruire l’automa-intersezione.

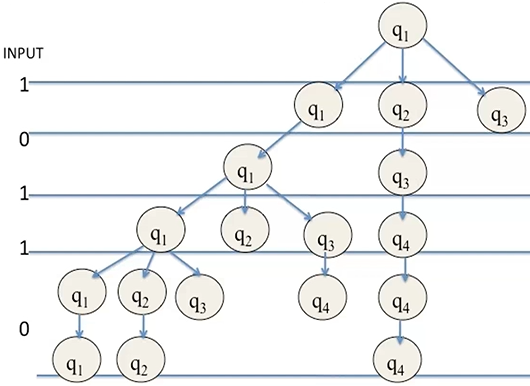
Per ottenere l’automa-unione A’ che riconosce tutte le stringhe aventi un numero pari di a o un numero pari di b (cioè, L(A’) = {w | w ha un numero pari di a OR w ha un numero pari di b}), basta trasformare anche gli stati (ad, bp) e (ap, bd) in stati accetta.

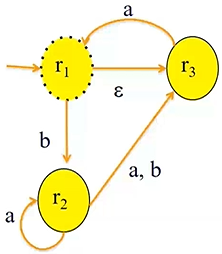
**Teorema.** *La classe dei linguaggi regolari è chiusa per la concatenazione. Cioè, se L1 e L2 sono linguaggi regolari, allora anche L1L2 è un linguaggio regolare.*  
Si noti che è possibile (ma laborioso) costruire un DFA per L1L2, dati i DFA per L1 ed L2. Introduciamo, invece, un nuovo tipo di macchina.

In un DFA, lo stato successivo occupato in corrispondenza di un dato input è unicamente determinato: quindi, le macchine sono deterministiche. La funzione di transizione in un DFA è f : Q × Σ → Q, che restituisce sempre un singolo stato.  
Gli automi finiti non deterministici (NFA) permettono più scelte per il prossimo stato per un dato input. Per uno stato q, l’NFA può:  
- avere più archi uscenti da q labellati con lo stesso simbolo a, per i vari a ∈ Σ;  
- prendere ε-edge senza leggere simboli in input.  
**Es.** Sia definito il seguente NFA N1 con alfabeto A = {0, 1}.  
Si nota subito che lo stato iniziale q1, quando riceve in input  
il simbolo 1, può sia restare nello stato q1 che passare agli  
stati q2 e q3: questo perché quando l’automa è nello stato  
q2 è possibile che venga ricevuta in input una ε vuota (una  
tale transizione prende il nome di *epsilon-transition*), di conseguenza si passa direttamente anche allo stato q3. Inoltre, può capitare che in uno stato l’automa non possa accettare un simbolo dell’alfabeto (in questo caso, nello stato q3 non è possibile ricevere una label 0).

Se l’NFA è in uno stato con più scelte (nell’esempio precedente, nello stato q1 e l’input è 1) la macchina si divide in più copie di se stessa: ogni copia continua una computazione indipendentemente dalle altre. Quindi, l’NFA può essere in un **insieme di stati** in uno stesso momento; segue che esso segue ogni possibile computazione in parallelo, e:  
- se una copia giunge in uno stato accetta, l’NFA accetta la stringa;  
- se nessun cammino giunge in uno stato accetta, allora l’NFA non accetta la stringa in input.

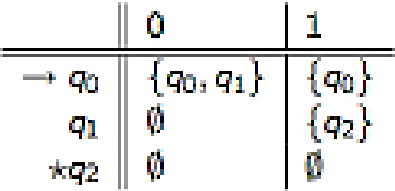
Se l’NFA è in uno stato con ε-transition, senza leggere input:  
- l’NFA si divide in più copie, ed ognuna segue una possibile transizione;  
- ogni copia continua indipendentemente da altre copie;  
- l’NFA segue ogni possibile cammino in parallelo;  
- l’NFA continua *non deterministicamente* come prima.  
Nell’esempio precedente, se si riceve in input 10110, questa è la computazione che l’automa effettua.

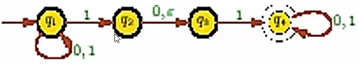
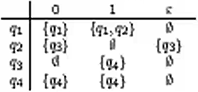


**Es.** Dato l’NFA posto a lato, è possibile notare che esso accetta stringhe ε, a, aa, baa,  
baba, …; non accetta, invece, stringhe b, ba, bb, …

**Def.** Dato un alfabeto Σ, *l’alfabeto* *Σε* è l’alfabeto ottenuto da Σ aggiungendo ε.  
Cioè, Σε = Σ ∪ {ε}.

**Def.** Un *automa finito non-deterministico* (NFA) A è una 5-tupla M = (Q, Σ, f, q0, F), dove:  
1. Q è un insieme finito di stati;  
2. Σ è un alfabeto, e il DFA processa stringhe su Σ;  
3. f : Q × Σε → P(Q) è la funzione di transizione, dove P(Q) è l’insieme di tutti i sottoinsiemi possibili di Q;  
4. q0 ∈ Q è lo stato start;  
5. F ⊆ Q è l’insieme di stati accetta (o finali).  
Nota. La differenza tra DFA e NFA è nella funzione di transizione f, la quale:  
- ammette mosse di tipo ε;  
- restituisce un insieme di stati anziché un solo stato.  
**Es.** Sia definito l’NFA N = ({q0, q1, q2}, {0, 1}, δ, q0, {q2}), posto di seguito, dove δ è la funzione di transizione descritta nella tabella a lato.

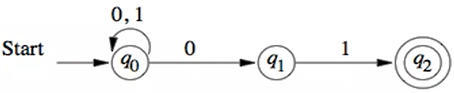
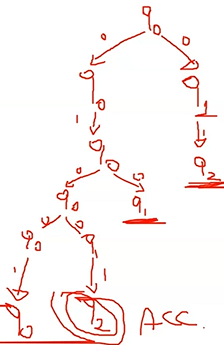


**Es.** Sia definito l’NFA N2 = ({q1, q2, q3, q4}, {0, 1}, δ, q1, {q4}), posto di seguito, dove δ è la funzione di transizione descritta nella tabella a lato.  
  
  
  
Si ricordi che, una volta l’automa è nello stato q2, esso può passare allo stato q3 in quanto è presente una ε-transition in tale stato.

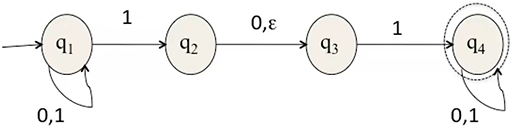
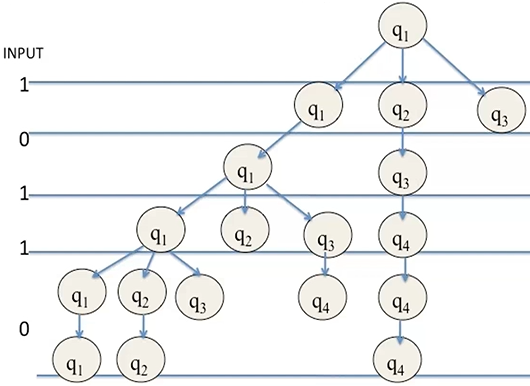
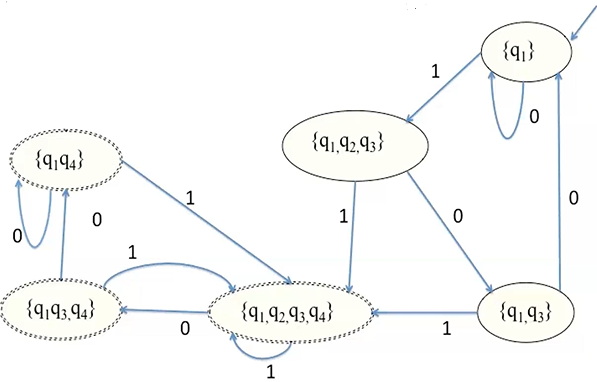
Sia N = (Q, Σ, f, q0, F) un NFA e w una stringa su A; allora, N **accetta** w se w = y1y2 ∙∙∙ ym, dove yi ∈ Σε, ed esiste una sequenza di stati r0, r1, …, rm in Q tale che:  
1. r0 = q0;  
2. ri+1 ∈ f(ri, yi+1) per ogni i = 0, 1, 2, …, m-1;  
3. rm ∈ F.  
Informalmente, vuol dire che N accetta w se esiste una possibile computazione che porta allo stato di accettazione. Si noti che la stringa generica yi non appartiene a Σ, bensì appartiene a Σε in quanto è possibile che essa sia un ε per effettuare una epsilon-transition; è questa l’unica differenza tra la computazione di un DFA e la computazione di un NFA.

**Def.** L’insieme di stringhe accettate da un NFA N è il linguaggio *riconosciuto* da N, ed è denotato con L(N).

Nota. Ogni DFA è anghe un NFA. Infatti, vedremo che è possibile esprimere un NFA attraverso un corrispondente DFA, e che i primi sono semplicemente un modo più compatto per descrivere i secondi.

**Es.** Sia definito il NFA N posto a lato. Allora esso accetta il linguaggio  
{x01 | x ∈ Σ\*}. Ad esempio, se l’input è 01001 abbiamo il seguente  
albero di computazione:  
- 0, da q0 si passa sia a q0 che a q1;  
- 1, da q0 si passa nuovamente a q0, e da q1 si passa a q2 (in quest’ultimo stato, la  
computazione si arresta);  
- 0, da q0 si passa sia a q0 che a q1;  
- 0, da q0 si passa sia a q0 che a q1, e da q1 con 0 si arresta la computazione;  
- 1, da q0 si passa a q0 e da q1 si passa a q2, e in quest’ultimo stato la stringa viene  
accettata.

**Def.** Due macchine sono *equivalenti* se riconoscono lo stesso linguaggio.

**Teorema.** *Ogni NFA N ha un equivalente DFA M; cioè, se N è un NFA, allora esiste un DFA M tale che L(M) = L(N).*DIM. Costruiamo, a partire da N, il DFA M equivalente. L’idea è simile a quella usata per costruire l’automa che riconosce l’unione di due linguaggi: simuliamo tutte le computazioni fatte dal NFA in parallelo. Consideriamo il seguente automa.  
  
  
  
  
  
L’automa deterministico dovrà, partendo dallo stato iniziale in  
presenza di input 1, ricordare all’interno dello stato che andrà  
ad occupare che l’automa non deterministico può essere in  
ognuno degli stati q1, q2 e q3: è possibile ottenere ciò andando  
a considerare uno stato che è dato dall’insieme degli stati  
{q1, q2, q3}. Allo stesso modo, in presenza di input 0, seguendo  
la prima computazione allora si passa da q1 a q1, mentre per  
la seconda si passa da q2 a q3, e la terza computazione si arresta: questo si può considerare nell’automa deterministico come uno stato dato dall’insieme degli stati {q1, q3}. Si procede in questo modo per ogni possibile computazione. Otteniamo, quindi, il seguente DFA.  
Si osservi che dallo stato {q1, q2, q3}, in presenza di input 0, si  
passa allo stato {q1, q3}.  
Gli stati finali del DFA saranno tutti quelli in cui compare  
*almeno uno* tra gli stati finali del NFA: in questo caso, solo gli stati  
in cui compare q4.

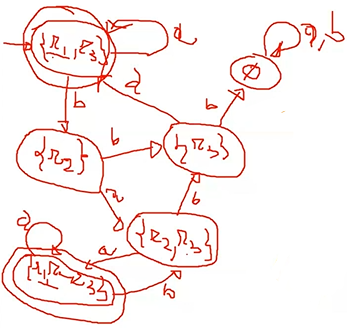
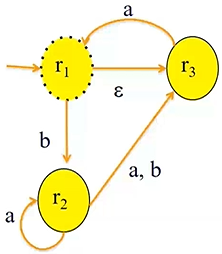
Formalizzando, sia L = L(N) per un NFA N = (QN, Σ, fN, qN, FN); allora, costruiamo un DFA D = (QD, Σ, fD, qD, FD).

Partiamo da un automa D’ che non considera le ε-transition, dove, per ogni R ∈ Q e a ∈ Σ:  
- Q = P(QN), ossia ogni stato in D’ sarà un sottoinsieme dell’insieme potenza degli stati del NFA N;  
- f(R, a) = , ossia la funzione di transizione darà come risultato l’unione dei risultati (per ogni elemento r dell’insieme R), della funzione di transizione fN(r, a);  
- q = {qN}, ossia lo stato iniziale sarà l’insieme in cui compare lo stato iniziale del NFA N;  
- F = {R ∈ Q | R ∩ FN ≠ ∅}, ossia gli stati finali saranno tutti quegli stati che contengono al loro interno uno stato finale del NFA N.

Per ogni stato in fN, se vi è una ε-transition allora dobbiamo considerarla. A tale scopo, definiamo l’insieme **E(R)** = R ∪ {q | q è raggiungibile da uno stato in R con 1 o più archi labellati ε}, cioè l’insieme che considera l’unione tra uno stato di D’ (in quanto R ∈ Q) e l’insieme degli stati raggiungibili da uno stato in R che ammettono almeno una ε-transition.  
**Es.** Considerando l’esempio precedente, si ha ad esempio che E({q1}) = {q1}, e E({q2}) = {q2, q3}: questo perché q1 non accetta epsilon-transition, a differenza di q2.

Quindi, avremo il DFA D:  
- QD = P(QN), che coincide con quanto definito per D’;  
- fD(R, a) = , ossia la funzione di transizione darà come risultato l’unione dei risultati (per ogni elemento r dell’insieme R), della E della funzione di transizione fN(r, a);  
- qD = E({qN}), ossia lo stato iniziale sarà la E dell’insieme in cui compare lo stato iniziale del NFA N;  
- FD = {R ∈ QD | R ∩ FN ≠ ∅}, che coincide con quanto definito per D’.

**Es.** Considerando l’esempio precedente, si ha ad esempio che fN(q1, 1) = {q1, q2} se non consideriamo le ε-transition. Se consideriamo le ε-transition, invece, si ha che E(fN(q1, 1)) = {q1, q2, q3}.

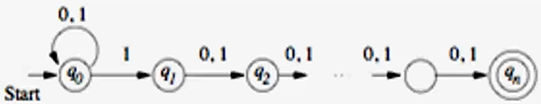
**Es.** Considerando l’automa posto a lato, otteniamo che il suo stato iniziale sarà  
E({r1}) = {r1, r3}. Abbiamo, inoltre, il suo DFA equivalente:  
- {r1, r3}, con input ‘a’ si ha che da r1 non si passa ad alcuno stato, mentre *da r3 si passa  
allo stato r1*. A questo punto, si applica nuovamente la funzione E (in quanto r1 accetta  
una ε-transition) e *si passa ancora dallo stato r1 allo stato r3*. Il prossimo stato sarà, quindi,  
ancora {r1, r3};  
- {r1, r3}, con input ‘b’ si ha che *da r1 si passa a r2*, mentre da r3 non si passa ad alcuno  
stato. Lo stato r2 non ammette ε-transition, quindi il nuovo stato sarà {r2};  
- {r2}, con input ‘a’ si ha che *da r2 si passa ad r2* e *da r2 si passa ad r3*, quindi il nuovo stato  
sarà {r2, r3};  
- {r2}, con input ‘b’ si ha che *da r2 si passa ad r3*, quindi il nuovo stato sarà {r3};  
- {r2, r3}, con input ‘a’ da r2 vale il punto precedente, mentre *da r3 si passa ad r1*,  
ma in quest’ultimo caso applicando la E otteniamo che *si passa anche da r1 a r3*.  
Di conseguenza, il nuovo stato sarà {r1, r2, r3};  
- {r2, r3}, con input ‘b’ da r2 vale il punto precedente, mentre da r3 non c’è  
nessuna transizione. Di conseguenza, il nuovo stato sarà {r3};  
- {r3}, con input ‘a’ *da r3 si passa a r1*, ma applicando la E otteniamo che *si passa  
anche da r1 a r3*. Il nuovo stato sarà, quindi, lo stato iniziale {r1, r3};  
- {r3}. con input ‘b’ non si passa ad alcuno stato. Tuttavia, l’automa è  
deterministico per cui non possiamo ignorare un eventuale input, per cui  
*si passa allo stato vuoto*;   
- considerando lo stato vuoto, per ogni input si passa nuovamente allo stato vuoto;  
- {r1, r2, r3}, con input ‘a’ ogni stato passa a quello successivo, per cui il nuovo stato sarà ancora {r1, r2, r3};  
- {r1, r2, r3}, con input ‘b’ si ha che *da r1 si passa ad r2*, *da r2 si passa ad r3*, mentre da r3 non si passa ad alcuno stato. Quindi, il nuovo stato sarà {r1, r3}.  
Ora, bisogna identificare gli stati finali: sono tutti quegli stati al cui interno è presente lo stato r1. L’automa deterministico corrispondente avrà, quindi, come stati finali {r1, r3} e {r1, r2, r3}. □

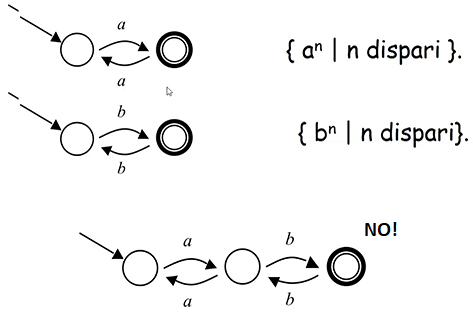
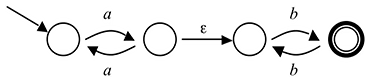
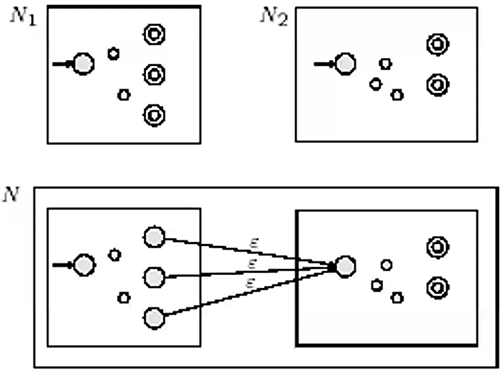
Nota. f\*N(qN, x) = insieme di stati raggiungibili da qN su input x. Ciò vuol dire che la funzione estesa di un automa non deterministico può dare come risultato più stati, in quanto è possibile avere diverse copie di computazione.  
**Es.** Nell’esempio all’inizio della pagina precedente, si ha che f\*N(q1, 10110) = {q1, q2, q4}.

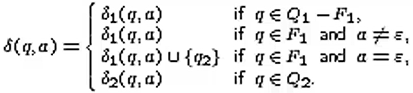
Formalmente, si ha che:  
- , quindi non ci sono lettere lette ed applichiamo la sola epsilon-transition;  
- , quindi si prendono tutti gli stati in cui potrebbe trovarsi l’automa dopo aver letto la stringa x ∈ Σ\*, e per ognuno di questi stati applichiamo la funzione di transizione del NFA, applichiamo poi la funzione E, facciamo l’unione di ciò che abbiamo ottenuto ed otteniamo l’insieme di stati raggiungibili quando l’input è la stringa xa.

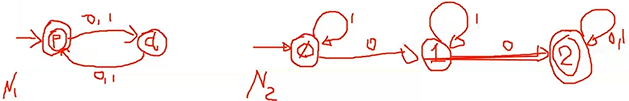
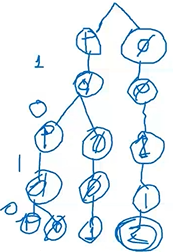
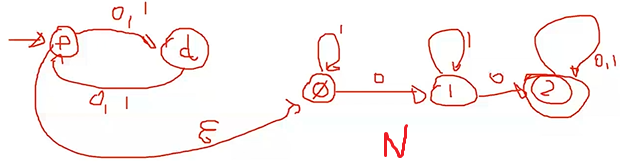
Risulta, per ogni x ∈ Σ\*, , cioè che le funzioni di transizione estese del NFA e del DFA corrispondente coincidono (ciò significa che i due automi accettano le stesse stringhe); questo risultato si può dimostrare induttivamente.  
Quindi, sappiamo che D simula N su ogni input x. Inoltre, D accetta x se e solo se N accetta x. Infine, il linguaggio L = L(N) = L(D). Ciò significa che D e N sono equivalenti.  
DIM. Mostriamo per induzione su |w|, ponendo q0 = qN, che .  
*Base:* w = ε. L’enunciato segue dalla definizione.  
*Passo*: Supponiamo che l’uguaglianza è vera per una certa stringa x, e consideriamo una stringa xa:  
 (per definizione)  
 (per ipotesi induttiva)  
 (per definizione)  
 . □

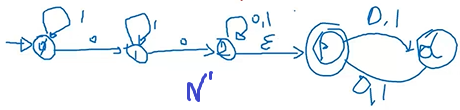
**Cor.** Un linguaggio L è regolare se e solo se esiste un NFA che riconosce L.  
DIM. Se L è regolare, allora esiste un DFA che lo riconosce; ma ogni DFA è anche un NFA, quindi esiste un NFA per L. Dal teorema precedente, ogni NFA ha un equivalente DFA. Quindi, se esiste un NFA che riconosce L, allora esisce un DFA che riconosce L.  
Formalmente: L = L(N) = L(D) ⇒ ∃ D : L(D) = L(N) = L; quindi, L è un linguaggio regolare. □

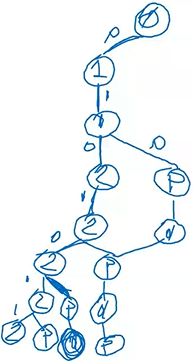
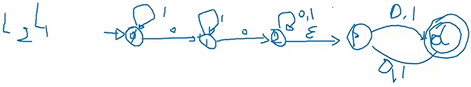
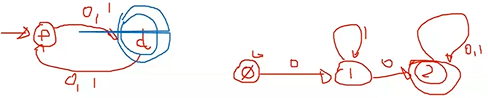
Si osservi che, sebbene sia possibile ottenere un DFA equivalente da un determinato NFA, è bene (spesso) utilizzare questi ultimi perché sono più facili da rappresentare.  
**Nota.** Esiste un NFA N con n+1 stati che non ha nessun DFA equivalente con meno di 2n stati. Il linguaggio dell’automa a dx è L(N) = {x1c2c3 ∙∙∙ | x ∈ {0, 1}\*, ci ∈ {0, 1}}.  
Questo automa è simile all’automa che riconosce tutte le  
stringhe che terminano con 01, ma è costruito in modo tale  
che lo stato iniziale cicli sia con 0 che con 1, dopodiché ha  
una transizione con 1 da q0 a q1 (quindi si ha uno sdoppiamento della computazione); inoltre, dallo stato q1 allo stato qn si avanza con qualsiasi simbolo che riceve in input. Quindi, una stringa per essere accettata deve avere qualunque simbolo all’inizio, un 1 e poi n-1 simboli qualsiasi.  
In sintesi, mentre un NFA può sdoppiarsi ad ogni stato ed avere più computazioni, un DFA deve necessariamente rappresentare gli stati in modo tale che possa ricordare gli ultimi n simboli che ha letto. Questo non è semplice da realizzare, in quanto ci sono 2n sequenze di bit (da rappresentare con opportuni stati) a1a2 ∙∙∙ an da ricordare.

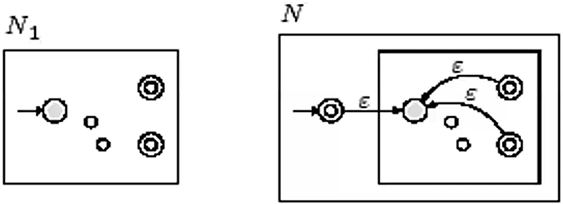
Si ricordi che, dati due linguaggi A, B regolari, la **concatenazione** è l’insieme AB = {vw | v ∈ A, w ∈ B}.  
**Teorema.** *La classe dei linguaggi regolari è chiusa per l’operazione di concatenazione.*  
DIM. Siano definiti i due automi a lato, i quali accettano rispettivamente tutte le  
stringhe aventi un numero pari di a (linguaggio A), e tutte le stringhe aventi un  
numero pari di b (linguaggio B).  
Si potrebbe pensare all’automa posto in basso alla figura, che accetta il linguaggio  
AB = {aibj | i, j dispari}. Tuttavia, esso è errato in quanto accetterebbe anche la  
stringa abbbab, che non rispetta tale linguaggio (che accetta un numero dispari di  
a *seguite* da un numero dispari di b).  
 L’automa di cui a sinistra, invece, è l’automa corretto. La  
 ε-transition, che va dallo stato finale del primo automa allo  
 stato iniziale del secondo automa, consente di considerare prima  
 la sottostringa formata da un numero dispari di a, poi di considerare la sottostringa formata da un numero dispari di b.  
Questo risultato è generalizzabile secondo l’esempio di cui a destra.  
Sia definito un linguaggio generico L1, riconosciuto dall’automa N1, e un  
linguaggio generico L2, riconosciuto dall’automa N2. *Si vuole costruire un  
NFA N che riconosca il linguaggio L1L2; si ha che:*  
*- lo stato iniziale di N coincide con lo stato iniziale di N1;  
- ogni stato finale di N1 è collegato da una epsilon-transition allo stato  
iniziale di N2;  
- gli stati finali di N coincidono con gli stati finali di N2.*  
Supponiamo di avere una stringa w = uv, dove u ∈ L1 e v ∈ L2. Quando  
si dà in input u al primo automa, al termine dell’esecuzione in N1 essa  
terminerà in un suo stato finale, in quanto u ∈ L1; dando la stringa v al secondo automa, essa terminerà in un suo stato finale, in quanto v ∈ L2.  
Dando la stringa w = uv in input all’automa N, accade che la sottostringa u viene accettata dal primo “sotto-automa”; siccome l’automa N è non deterministico, la computazione si può sdoppiare al termine di u (per la ε-transition) e la computazione prosegue con la lettura della sottostringa v dal secondo automa, che la accetterà. Di conseguenza, la stringa w = uv verrà accettata da N.  
Analogamente, supponiamo di avere una stringa w = uv, non appartenente a L1L2: questo significa che u ∉ L1 OR v ∉ L2. Accaderà, quindi, che la stringa non verrà accettata da (almeno) uno dei sottoautomi, di conseguenza w non verrà accettata dall’automa N. □

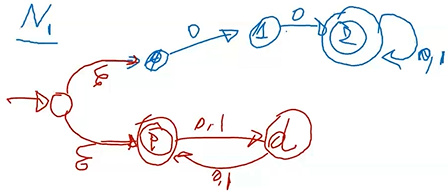
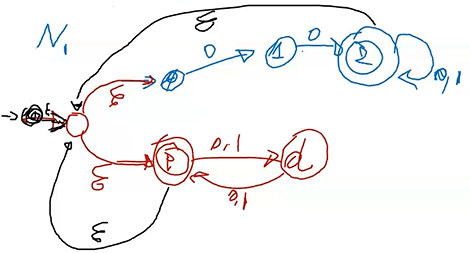
Formalmente, supponiamo che il linguaggio L1 sia riconosciuto da un NFA N1 = (Q1, Σ, δ1, q1, F1), e che il linguaggio L2 sia riconosciuto da un NFA N2 = (Q2, Σ, δ2, q2, F2). Per costruire un automa non deterministico che rispetti il linguaggio L dato dalla concatenazione tra L1 ed L2, si ha un NFA N = (Q, Σ, δ, q1, F2) tale che:  
- Q = Q1 ∪ Q2;  
- il suo alfabeto è lo stesso dei due linguaggi L1 ed L2;  
- il suo stato start è q1, lo stesso di N1;  
- i suoi stati finali appartengono all’insieme F2, gli stessi di N2;  
- la funzione di transizione è regolata dalla figura a lato.  
In particolare, la funzione di transizione *δ(q, a)*, dato un generico stato *q* ed una stringa *a*:  
- se *q* è uno stato non finale del primo automa, allora la transizione è quella specificata dal primo automa;  
- se *q* è uno stato finale del primo automa e *a* è diversa da epsilon, allora la transizione è quella specificata dal primo automa;  
- se *q* è uno stato finale del primo automa e *a* è ε, allora alle transizioni che già si avevano per il primo automa vanno aggiunte anche le ε-transition verso lo stato iniziale del secondo automa;  
- se *q* è uno stato del secondo automa, allora le transizioni sono quelle specificate dal secondo automa.

**Es.** Siano L1 = {w | |w| è pari} e L2 = {w | w contiene almeno due 0}. Vogliamo costruire gli automi per questi due linguaggi, e a partire da questi ultimi un automa per la loro concatenazione L1L2. A tal scopo, sia N1 l’automa per L1, e sia N2 l’automa per L2. Otteniamo il seguente risultato.  
  
  
  
  
  
Quindi, sia L(N) = L1L2. Dall’idea dimostrativa del teorema precedente, otteniamo il seguente automa N.  
Notiamo che, ad esempio, la stringa 10100 è accettata da N, dato che è gestibile come stringa costituita dalle sottostringhe 10 ∈ L1 e 100 ∈ L2. Questa stringa è gestita dall’automa come descritto dall’albero a destra.

**Es.** Dati i linguaggi L1 ed L2 precedenti, vogliamo costruire un automa per la loro concatenazione L2L1. A tal scopo, sia N1 l’automa per L1, e sia N2 l’automa per L2. Quindi, sia L(N’) = L2L1. Le stringhe accettate devono, quindi, essere costituite da una sottostringa contenente due o più 0, e da una sottostringa ad essa concatenata di lunghezza pari. Dall’idea dimostrativa del teorema precedente, otteniamo il seguente automa N’.

**Es.** Siano L1 = {w | |w| è dispari} e L2 = {w | w contiene almeno due 0}. Vogliamo costruire gli automi per questi due linguaggi, e a partire da questi ultimi un automa per la loro concatenazione L1L2. A tal scopo, sia N1 l’automa per L1, e sia N2 l’automa per L2. Otteniamo il risultato sottostante. Ad esempio, la  
stringa 1010111 (gestibile come 1010 ∈ L2 e 111 ∈ L1) viene accettata. La 0101011 (gestibile  
come 0101 ∈ L2 e 011 ∈ L1) segue l’alberocomputazionale di cui a destra.

La concatenazione è stata definita per due linguaggi, ma in realtà è possibile iterare ed andare a concatenare più di due linguaggi: per concatenare L1, L2, L3, ad esempio, si può prima concatenare L1 ed L2 per poi procedere con la concatenazione di L3. In generale, questo risultato si può estendere alla **Kleene-star**:  
 L\* = {x1x2 ∙∙∙ xk | k ≥ 0, xi ∈ L, i = 1, 2, …, k}.  
**Teorema.** *La classe dei linguaggi regolari è chiusa per l’operazione Kleene-star.*  
DIM. Sia N1 un automa che riconosce un linguaggio L.  
*Costruiamo N* (che riconosce L\*) *da N1, in cui ogni stato  
finale è collegato da una ε-transition allo stato iniziale* (di N1)*.*  
In questo modo, si consente all’automa di tornare  
indietro una volta arrivato in uno stato finale, in modo  
tale da poter gestire un numero qualsiasi di  
concatenazioni di stringhe appartenenti allo stesso  
linguaggio. Siccome la Kleene-star contiene anche la stringa vuota ε,  
allora *occorre inserire in N un nuovo stato iniziale* (che è anche finale, appunto per gestire quest’ultimo caso) *da collegare allo stato iniziale di N1*.  
Se, quindi, partiamo da un automa N1 = (Q1, Σ, δ1, q1, F1), allora N = (Q, Σ, δ, qn, F), dove:  
- Q = Q1 ∪ {qn}, dove qn è il nuovo stato iniziale;  
- F = F1 ∪ {qn};  
- δ(q, a) = δ1(q, a) se q ≠ qn e q ∉ F1,   
 δ(q, a) = δ1(q, a) ∪ {q1} se q ≠ qn e q ∈ F1,  
 δ(q, a) = q1 se q = qn e a = ε. □

**Es.** Sia L = {w | |w| è pari OR w = 00x}. Un automa non deterministico che riconosca L è il seguente (N1, a sinistra), dove in rosso è posto il sottoautoma che accetta le stringhe pari, e in blu è posto l’automa che riconosce le stringhe che iniziano con 00.  
Notiamo che l’OR, in un automa non deterministico, è semplicemente rappresentabile aggiungendo uno stato iniziale generico collegato con una ε-transition agli “ex stati iniziali” dei sottoautomi. Questo è possibile in quanto, con i NFA, la computazione si sdoppia direttamente all’inizio e prosegue nei due sottoautomi.  
A destra, invece, è posto l’automa N che riconosce la Kleene-star di L, cioè L\* = L(N).  
  
  
  
  
  
  
  
Ad esempio, la stringa 0011 (considerabile come  
unica stringa che inizia con due 0, oppure come  
stringa costituita da x1 = 00 concatenata con  
x2 = 11) viene accettata.

Più in generale, per la costruzione dell’OR (**unione**)supponiamo di avere due linguaggi, L1 ed L2, riconosciuti rispettivamente da due automi N1 e N2. Per costruire l’automa N, che riconosce il linguaggio L1 ∪ L2:  
1. si crea un nuovo stato, che sarà lo stato iniziale dell’automa N;  
2. colleghiamo lo stato iniziale di N con i due stati iniziali di N1 e N2.  
In questo modo, abbiamo un NFA che riconosce tutte le stringhe dell’unione, in quanto (per ogni stringa) la computazione si sdoppia ed avremo due rami computazionali: uno eseguito da N1 e uno eseguito da N2. Ora:  
- se la stringa appartiene all’unione, allora essa o arriva in uno stato finale di N1 o arriva in uno stato finale di N2 (o in entrambi), e viene accettata dall’automa N;  
- se la stringa non appartiene all’unione, allora essa non arriva in alcuno stato finale di N1 o N2, e non viene accettata dall’automa N.

**Espressioni regolari**

Un FA (NFA o DFA) è un metodo per costruire una macchina che riconosce linguaggi regolari.  
Un’**espressione regolare** è un modo dichiarativo per descrivere un linguaggio regolare.  
**Es.** 01\* ∪ 10\*.  
Le espressioni regolari sono usate, ad esempio, in comandi UNIX (grep), strumenti per l’analisi lessicale di UNIX (Lex – Lexical analyzer generator – e Flex – Fast Lex).  
Sia A = {0, 1}. Per brevità, nelle espressioni regolari: 0 indica {0}, 1 indica {1}. In pratica, eliminiamo la notazione insiemistica per brevità.  
**Es.** 0 ∪ 1 indica {0} ∪ {1}, cioè {0, 1}.  
**Es.** (0 ∪ 1)0\* è {0, 1}{0}\* (dove {0}\*. = {ε, 0, 00, 000, …}.), cioè l’insieme di stringhe binarie che iniziano con 0 oppure 1 e continuano con degli 0 (anche nessuno).  
**Es.** (0 ∪ 1)\* è {0, 1}\*, cioè l’insieme di tutte le stringhe su {0, 1}.

Le espressioni regolari possono essere definite formalmente utilizzando una definizione induttiva. La base è:  
- a è espressione regolare per ogni a nell’alfabeto, e denota l’insieme {a};  
- ε è espressione regolare, e denota l’insieme {ε};  
- ∅ è espressione regolare, e denota l’insieme {∅}.  
Siano, ora, R1 e R2 espressioni regolari che rappresentano i linguaggi L1 e L2. Si ha che:  
- l’*unione* (R1 ∪ R2) rappresenta l’insieme L1 ∪ L2;  
- la *concatenazione* (R1R2) rappresenta l’insieme L1L2;  
- la *chiusura di Kleene* (R1)\* rappresenta l’insieme L1\*.  
Ogni espressione regolare può essere costruita a partire dal risultato ottenuto.  
**Es.** Sia R1 = 0 l’e.r. che rappresenta L1 = {0}, e sia R2 = 1 l’e.r. che rappresenta L2 = {1}. Allora:  
- l’unione delle due espressioni regolari (R1 ∪ R2) = 0 ∪ 1 rappresenta l’insieme L1 ∪ L2 = {0, 1};  
- la chiusura di Kleene (R1)\* = 0\* rappresenta l’insieme L1\* = {0}\*.  
Siano E1 = 0 ∪ 1 e E2 = 0\* espressioni regolari che rappresentano rispettivamente gli insiemi L1’ = {0, 1} e L2’ = {0}\*; allora si ha che la concatenazione E1E2 = (0 ∪ 1)0\* è un’espressione regolare che rappresenta il linguaggio L1’L2’ = {0, 1}{0}\*.

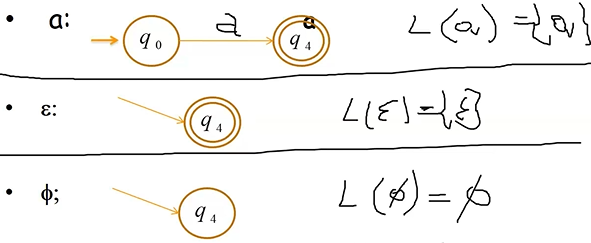
**Def.** Se R è un’espressione regolare, allora L(R) denota il linguaggio *generato* da R.  
**Es.** R = (0 ∪ 1)0\* è un’espressione regolare che genera il linguaggio L(R) = {0, 1}{0}\*.

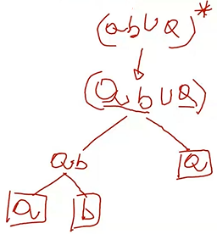
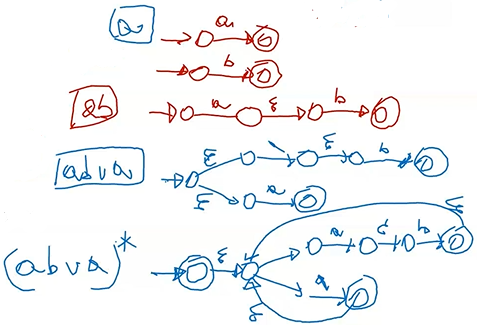
**Definizione induttiva di espressione regolare (e.r.).**  
BASE:  
- ε e ∅ sono espressioni regolari: L(ε) = {ε} e L(∅) = ∅;  
- se a ∈ Σ, allora a è un’espressione regolare: L(a) = {a}.  
PASSO: Se R1 e R2 sono espressioni regolari, allora:  
- R1 ∪ R2 è un’espressione regolare che rappresenta L(R1) ∪ L(R2);  
- R1R2 è un’espressione regolare che rappresenta L(R1)L(R2);  
- R1\* è un’espressione regolare che rappresenta (L(R1))\*.

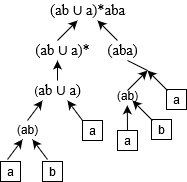
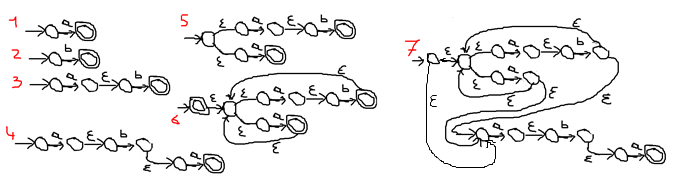
Le espressioni regolari possono essere create applicando le operazioni di unione, concatenazione e Kleene star. Possiamo utilizzare le parentesi per cambiare l’ordine usuale di precedenza per le operazioni regolari, che è il seguente: 1. Kleene star (\*); 2. Concatenazione (∙); 3. Unione (∪).  
**Es.** 01\* ∪ 1 è raggruppato in (0(1)\*) ∪ 1, cioè il linguaggio che riconosce la stringa 1 oppure le stringhe che iniziano con 0 e terminano con zero o più 1.  
**Es.** 00 ∪ 101\* è è il linguaggio formato da una stringa 00 e da stringhe inizianti con 10 seguite da zero o più 1.  
**Es.** Sia 0(0 ∪ 101)\* un’espressione regolare. Prima si effettua la concatenazione di 101, poi si effettua l’unione tra 0 e 101, di quest’unione si effettua la Kleene star, ed infine si effettua la concatenazione. Allora:  
- 0101001010 appartiene al linguaggio;  
- 00101001 non appartiene al linguaggio;  
- 0000000 appartiene al linguaggio;  
- 101 non appartiene al linguaggio.

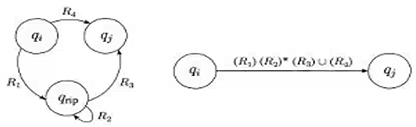
**Es.** Sia A = {0, 1}. Allora:  
1. (0 ∪ 1) genera il linguaggio {0, 1};  
2. 0\*10\* genera il linguaggio {w | w ha un solo 1};  
3. A\*1A\* genera il linguaggio {w | w ha almeno un 1};  
4. A\*001A\* genera il linguaggio {w | w ha 001 come sottostringa};  
5. (AA)\* genera il linguaggio {w | |w| è pari}, in quanto (AA)\* = ({0, 1}{0, 1})\* = {00, 01, 10, 11}\*;  
6. (AAA)\* genera il linguaggio {w | |w| è multiplo di 3}.

**Es.** Sia EVEN-EVEN su A = {a, b} l’insieme di stringhe con un numero pari di a e un numero pari di b. Ad esempio, aababbaaababab ∈ EVEN-EVEN. Si ha che l’e.r. (aa ∪ bb ∪ (ab ∪ ba)(aa ∪ bb)\*(ab ∪ ba))\* genera il linguaggio EVEN-EVEN.  
Questo esempio fa comprendere che le espressioni regolari consentono di definire una classe di linguaggi che è esattamente quella dei linguaggi regolari.

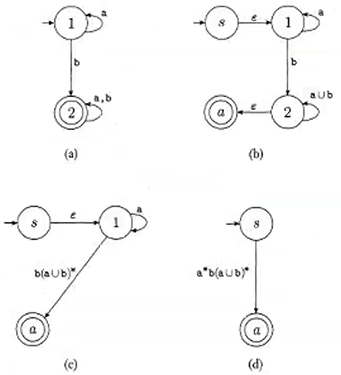
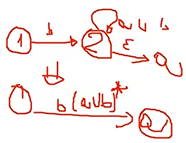
**Teorema** (TEOREMA DI KLEENE)**.** *Un linguaggio L è regolare se e solo se esso ammette una espressione regolare.*  
Questo significa che, poiché un linguaggio è regolare se e solo se è riconosciuto da un automa finito, questo risultato stabilisce che la classe dei linguaggi riconosciuti dagli automi finiti è esattamente la classe dei linguaggi descrivibili mediante un’espressione regolare.  
DIM. Sappiamo che: Linguaggio L è regolare ⇔ L è riconosciuto da NFA ⇔ L è riconosciuto da DFA.  
Si può dimostrare che:  
(1) per ogni DFA A possiamo costruire un’espressione regolare R con L(R) = L(A);  
(2) per ogni espressione regolare R esiste un NFA A tale che L(A) = L(R).  
Dimostrando queste due affermazioni, si ha che L è riconosciuto da DFA ⇔ L ammette un’espressione regolare. Quindi: Linguaggio L è regolare ⇔ L ammette un’espressione regolare.  
DIM (2): *L ammette un’e.r. ⇒ L riconosciuto da NFA.*  
1. Dimostriamo per casi base: a, ε e ∅.  
  
  
  
  
  
  
  
2. Assumiamo la correttezza per R1 e R2, quindi proviamo per R1 ∪ R2, R1R2, R1\*. Questo è immediato: basta costruire opportuni NFA per unione, concatenazione e Kleene star.  
Dato R1, sappiamo che esiste un automa N1, per ipotesi induttiva, tale che L(N1) = L(R1); analogamente, sappiamo che esiste un automa N2, per ipotesi induttiva, tale che L(N1) = L(R1). Di conseguenza, si ha che:  
- L(R1 ∪ R2) = L(R1) ∪ L(R2) = L(N1) ∪ L(N2) = L(N);  
- L(R1R2) = L(R1)L(R2) = L(N1)L(N2) = L(N);  
- L(R1\*) = (L(R1))\* = (L(N1))\* = L(N). □

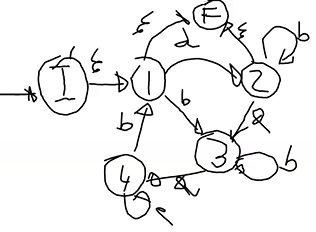
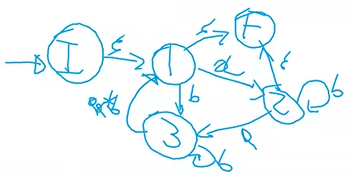
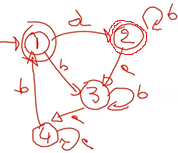
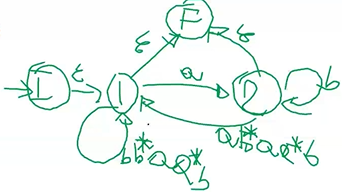
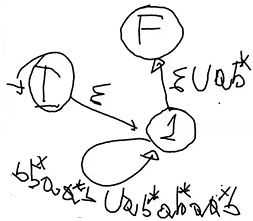
**Es.** Costruire un NFA per (ab ∪ a)\*.  
Induttivamente, sappiamo che (ab ∪ a)\* è il risultato della Kleene star su (ab ∪ a), che è il risultato dell’unione di (ab) e (a) *(base)*, mentre sappiamo che ab è il risultato della concatenazione tra a *(base)* e b *(base)*.  
Basandoci su questa costruzione, possiamo costruire l’automa che riconosce il linguaggio di  
questa espressione regolare. Procediamo così:  
1. costruiamo l’automa che riconosce a;  
2. costruiamo l’automa che riconosce b;  
3. costruiamo l’automa che riconosce ab;  
4. costruiamo l’automa che riconosce ab ∪ a;  
5. costruiamo l’automa che riconosce (ab ∪ a)\*.  
La figura a destra include tutti gli automi.

**Es.** Costruire un NFA per (ab ∪ a)\*aba.  
Induttivamente, sappiamo che:  
a) (ab ∪ a)\*aba è il risultato della concatenazione tra (ab ∪ a)\* e aba; quindi:  
 b) banalmente, aba è il risultato della concatenazione tra ab ed a (*base*), mentre ab è il risultato della  
 concatenazione tra a *(base)* e b *(base)*;  
 c) (ab ∪ a)\* è il risultato della Kleene star su (ab ∪ a), che è il risultato dell’unione tra ab e a *(base)*, mentre ab è  
 il risultato della concatenazione tra a *(base)* e b *(base)*.  
Basandoci su questa costruzione, possiamo costruire l’automa che riconosce il linguaggio di questa espressione regolare. Procediamo così:  
1. costruiamo l’automa che riconosce a;  
2. costruiamo l’automa che riconosce b;  
3. costruiamo l’automa che riconosce ab;  
4. costruiamo l’automa che riconosce aba; (*punto b concluso*)  
5. costruiamo l’automa che riconosce l’unione tra ab e a;  
6. costruiamo l’automa che riconosce la Kleene star di (ab ∪ a); (*punto c concluso*)  
7. costruiamo l’automa che riconosce la concatenazione tra (ab ∪ a)\* e aba. (*punto a concluso*).  
Di seguito sono posti tutti gli automi.

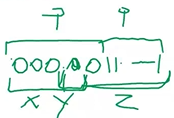
DIM(1): *L riconosciuto da NFA ⇒ L ammette un’e.r.*  
Procediamo convertendo dei DFA in espressioni regolari equivalenti. Costruiamo la prima variazione di NFA: le transizioni possono avere delle espressioni regolari per etichette, invece che solo gli elementi dell’alfabeto o ε. Per comodità, vogliamo che:  
- lo stato iniziale non ha transizioni da qualsiasi altro stato;  
- c’è soltanto un singolo stato accetta, che non ha frecce uscenti e non coincide con lo stato start.  
Per convertire un DFA in un’espressione regolare:  
- aggiungiamo un nuovo stato start con una ε-transition collegata al vecchio stato start;  
- aggiungiamo un nuovo stato accetta collegato con frecce ε-transition (partendo d)ai vecchi stati accetta.  
Successivamente, selezioniamo uno stato isolandolo dalla macchina e sistemando il resto in modo che lo stesso linguaggio sia ancora riconosciuto. Qualsiasi stato va bene, a condizione che non sia uno stato start o accetta. Chiamiamo lo stato rimosso qrip. Dopo aver rimosso lo stato, modifichiamo la macchina alterando le espressioni regolari che etichettano le restanti frecce. Le nuove etichette compensano l’assenza di qrip aggiungendo le computazioni perdute nella modifica. La nuova etichetta, passando da uno stato qi a uno stato qj è una espressione regolare che descrive tutte le stringhe che avrebbe utilizzato la macchina da qi a qj direttamente attraverso qrip.  
La figura a destra è un semplice esempio. Lo stato eliminato, qrip,  
aveva una freccia entrante da qi (con espressione R1), un ciclo  
su se stesso (con espressione R2) e una freccia uscente verso  
qj (con espressione R3). Eliminando questo stato, resta la sola  
freccia da qi a qj, con espressione regolare *(R1)(R2)\*(R3) ∪ (R4)*.  
Chiaramente, esisteva già una freccia da qi a qj (con espressione R4) prima dell’eliminazione, per cui il risultato sarà una *unione* tra R4 e l’espressione creata dall’eliminazione; quest’ultima sarà una *concatenazione* tra (R1), (R2)\* (in quanto *ogni ciclo sullo stato da eliminare si trasforma in Kleene-star*) e R3.

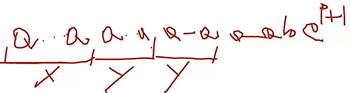
Nella pagina seguente è posto un ulteriore esempio.

Abbiamo l’automa di partenza **(a)**, di cui vogliamo costruire l’e.r. nel  
modo visto in precedenza. Quindi, partiamo con l’aggiungere un nuovo  
stato iniziale ed un nuovo stato finale, dove: il nuovo stato iniziale è  
connesso da un’ε-transition al vecchio stato iniziale, ed il vecchio stato  
finale è connesso da un’ε-transition al nuovo stato finale **(b)**.  
Una transizione *a, b* si trasforma nell’e.r. *a ∪ b* (si noti lo stato 2).  
Successivamente, eliminiamo un primo stato **(c)**. Consideriamo prima  
l’eliminazione dello stato 2. Prendiamo, quindi, il sottoautoma seguente  
(in rosso) per rendere più chiara la modifica.  
 Lo stato 2 ha una freccia entrante dallo stato 1  
 con label b, un ciclo su se stesso con label  
 a ∪ b, e una epsilon-transition verso il nuovo  
 stato finale. Dunque, si elimina lo stato 2 e si  
 ottiene un arco da 1 allo stato finale *a*, con  
 espressione regolare *b(a* *∪ b)\**, in quanto bisogna  
concatenare la label da 1 a 2 (cioè *b*) con la Kleene-star della label di 2 (cioè *a ∪ b*, in quanto era un ciclo su se stesso), per poi concatenarla con la label della freccia uscente da 2 ad a (che, tuttavia, è omessa in quanto epsilon-transition).  
Ora, bisogna eliminare lo stato 1 **(d)**. Notiamo che abbiamo un’unica transizione da *s* a 1 (che è un’epsilon-transition), un ciclo con label a, e un arco uscente da 1 ad *a* con la label prodotta in precedenza. Di conseguenza, si ottiene ***a\*b(a ∪ b)\**** (quindi la concatenazione tra *a\**, cioè la label del ciclo sullo stato 1, e l’arco uscente), che è l’espressione regolare dell’automa (a).  
Questo risultato, ponendo E = a\*b(a ∪ b)\*, abbiamo che L = L(E), cioè abbiamo dimostrato che se L è riconosciuto da un DFA allora esso ammette un’e.r. □

**Es.** Si consideri il DFA in rosso, e si costruisca una espressione regolare che corrisponda al suo linguaggio.  
1) Aggiungiamo i nuovi start e final state. Il risultato è l’automa posto in nero.  
2) Procediamo con l’eliminazione dello stato 4. Notiamo che esso ha: un’unica transizione entrante dallo stato 3, ed ha un’unica transizione uscente che va allo stato 1 (labellate rispettivamente a e b) oltre che ad un ciclo labellato a. Questo sottoautoma si trasforma in un arco da 3 a 1 avente espressione regolare *aa\*b*. Il risultato è l’automa posto in azzurro.  
3) Procediamo con l’eliminazione dello stato 3. Notiamo che esso ha: due transizioni entranti (una da 1 e una da 2, labellate rispettivamente con b e a), una transizione uscente verso 1 (labellata R = aa\*b) e un ciclo (labellato b). Considerando i due sottoautomi A = 1 → 3 → 1, e B = 2 → 3 → 1, otteniamo che:  
- il sottoautoma A viene sostituito dalla transizione da 1 a 1, la cui label è *bb\*R*;  
- il sottoautoma B viene sostituito dalla transizione da 2 a 1, la cui label è *ab\*R*.  
Quindi, si ha come risultato un arco da 1 a 1 (avente label bb\*aa\*b) e un arco da 2 a 1 (avente label ab\*aa\*b). Il risultato è l’automa posto in verde.  
4) Procediamo con l’eliminazione dello stato 2. Notiamo che esso ha: un’unica transizione entrante da 1 (labellata a), due transizioni uscenti (una verso 1, labellata R1 = ab\*aa\*b, e una ε-transition verso F) e un ciclo (labellato b). Considerando i due sottoautomi C = 1 → 2 → 1, e D = 1 → 2 → F, otteniamo che:  
- il sottoautoma C viene sostituito dalla transizione da 1 a 1, la cui label è *ab\*R1*;  
- il sottoautoma D viene sostituito dalla transizione da 1 a F, la cui label è *ab\**.  
Quindi, si ha come risultato un arco da 1 a 1 (che già esiste, quindi verrà sostituito dall’unione tra la preesistente bb\*aa\*b e la nuova ab\*ab\*aa\*b) e un arco da 1 a F (che già esiste, quindi verrà sostituito dall’unione tra la preesistente ε-transition e la nuova ab\*). Il risultato è posto in nero, in basso.  
5) Resta da eliminare lo stato 1. Notiamo che esso ha: un’unica ε-transition entrante da I, un’unica transizione verso F (labellata ε ∪ ab\*) e un ciclo (labellato R2 = bb\*aa\*b ∪ ab\*ab\*aa\*b). Otteniamo, quindi, l’automa finale avente unica transizione da I a F, la cui label è ***(bb\*aa\*b ∪ ab\*ab\*aa\*b)\*(ε ∪ ab\*)***.  
Il risultato è l’automa posto in blu.  
  
Nella pagina seguente sono posti tutti gli automi nei vari step.  
  
  
  
  
  
  
  
  
  
  
  
  
  
  


Non tutti i linguaggi sono regolari. Ad esempio, il linguaggio L = {anbn | n ≥ 0} non è regolare, cioè non è riconosciuto da alcun automa finito: non è possibile, infatti, avere un automa che riconosca un linguaggio di questo tipo. Se tentiamo di costruire un DFA che riconosce L = {anbn | n ≥ 0}, l’automa deve verificare che la stringa è fatta da un numero *n* di a seguito da uno stesso numero *n* di b: ci servirebbero infiniti stati per contare il numero di a ed il numero di b.  
Useremo il seguente lemma per mostrare che un linguaggio non è regolare.  
**Lem** (PUMPING LEMMA)**.** *Per ogni linguaggio regolare L, esiste una costante positiva p tale che, per ogni stringa w in L di lunghezza |w| ≥ p, esistono stringhe x, y, z tali che w = xyz, che soddisfano:  
- |y| > 0;  
- |xy| ≤ p;  
- xyiz in L, per ogni i ≥ 0.*  
DIM. Siano M un automa che riconosce L, e p il numero di stati di M.  
Considerando una stringa w = xyz tale che |w| ≥ p, allora sappiamo che nella computazione esiste (almeno) uno stato ripetuto, cioè esiste (almeno) uno stato che viene visitato più di una volta (sia r il primo stato ripetuto). A questo punto, sappiamo che:  
- x (la prima parte della stringa w) porta la computazione dallo stato iniziale q1 allo stato r (cioè, f\*(q1, x) = r);  
- y (la seconda parte della stringa w) porta la computazione dallo stato r allo stato r (cioè, f\*(r, y) = r);  
- z (la terza parte della stringa w) porta la computazione dallo stato r allo stato finale (cioè, f\*(r, z) ∈ F).  
Verifichiamo che questa suddivisione soddisfa le tre proprietà del lemma:  
- |xy| ≤ p (con r primo stato ripetuto) è soddisfatta, in quanto se così non fosse allora avremmo lo stato ripetuto prima;  
- |y| ≥ 1 segue dalla precedente, in quanto abbiamo bisogno di almeno un simbolo per poter ciclare su r;  
- xyiz appartiene al linguaggio L, dato che tale stringa porta dallo stato iniziale ad r, *da r ad r per i volte* (anche 0 volte), e infine da r a uno stato finale. □

**Teorema.** Sia L l’insieme di tutte le stringhe 0n1n (con n ≥ 0) su {0, 1} aventi un uguale numero di 0 e di 1. Il linguaggio L non è regolare.  
DIM. Dimostriamo questo teorema per contraddizione usando il Pumping Lemma (PL). Supponiamo, a tal scopo, che L è regolare, quindi applichiamo il lemma. Sia, quindi, p la costante del PL. Consideriamo una stringa w = 0p1p. Il pumping lemma implica che esistono xyz = 0p1p, tali che |xy| ≤ p, y è non vuota e che xykz appartiene a L *per ogni k ≥ 0.*  
Ma se la stringa presa è 0p1p, allora i primi p simboli sono tutti 0. Tuttavia, se il PL dice che la lunghezza della sottostringa xy è effettivamente minore o uguale a p, allora sappiamo che essa sta entro quei p simboli 0; inoltre, siccome y è non vuota, abbiamo che y è una sottostringa di almeno uno 0.  
Ora, consideriamo la stringa xykz. Presa in input tale stringa, per contraddire il PL occorre  
esibire un valore k per cui la stringa xykz non appartiene al linguaggio.  
Consideriamo k = 2 (quindi, y si ripete due volte). Per appartenere al linguaggio, questa stringa  
dovrà avere un numero p di 0 ed un numero p di 1: abbiamo una stringa del tipo 0|x|0|y|0|y|0 ∙∙∙ 01p = 0(p + |y|)1p; poiché |y| > 1, si ha che questa stringa non appartiene al linguaggio L, dato che p + |y| > p.  
Ricapitolando, preso un qualsiasi k > 1, allora xykz e xyz hanno (tra di loro) diverso numero di 0 e stesso numero di 1. Questa è una contraddizione. □

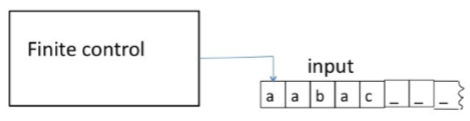
**Es.** Sia L = {anbam | n < m}. Usiamo il Pumping Lemma per dimostrare che il linguaggio L non è regolare.  
Supponiamo P.A. che L sia regolare; di conseguenza, esso soddisferebbe il PL, quindi esisterebbe una costante p tale che,per ogni stringa w in L di lunghezza |w| ≥ p, esistono stringhe x, y, z tali che w = xyz, che soddisfano: |y| > 0, |xy| ≤ p, xyiz in L (per ogni i ≥ 0).  
Sia definita la stringa, legata a p, w = apbap+1 = xyz, dove |xy| ≤ p; da questo  
consegue che xy è costituita da un numero finito di a.  
Bisogna dimostrare che la terza condizione non è soddisfatta, cioè che esiste i ≥ 0  
tale che la stringa xyiz ∉ L. Sia, ad esempio, i = 2, quindi xy2z è la stringa a lato.  
Notiamo che questa stringa non appartiene al linguaggio, dato che (|x| + |y| + # a prima del simbolo b) = p, e quindi il numero totale di a prima del simbolo b è p + |y|; siccome |y| > 1, abbiamo che p + |y| ≥ p + 1, quindi la condizione n < m del linguaggio non è soddisfatta. Dal Pumping Lemma, otteniamo che il linguaggio L non è regolare. Questa è una contraddizione.

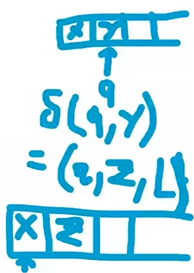
**(?) Esercizio.** Sia L = {anbam | n ≥ m}. Usare il PL per dimostrare che il linguaggio L non è regolare.  
🡪 Supponiamo P.A. che L sia regolare; di conseguenza, esso soddisferebbe il PL, quindi esisterebbe una costante p tale che,per ogni stringa w in L di lunghezza |w| ≥ p, esistono stringhe x, y, z tali che w = xyz, che soddisfano: |y| > 0, |xy| ≤ p, xyiz in L (per ogni i ≥ 0).  
Sia definita la stringa, legata a p, w = apbap = xyz, dove |xy| ≤ p; da questo consegue che xy è costituita da un numero finito di a.  
Bisogna dimostrare che la terza condizione non è soddisfatta, cioè che esiste i ≥ 0 tale che la stringa xyiz ∉ L. Sia, ad esempio, i = 0, quindi xy0z è la stringa considerata. Notiamo che questa stringa non appartiene al linguaggio, dato che, siccome |xy| ≤ p, e siccome |y| > 0, necessariamente otteniamo che |x| < p; da ciò consegue che (|x| + # a prima del simbolo b) = p - |y| **< p**. Quindi, la condizione del linguaggio n ≥ m non è soddisfatta. Dal Pumping Lemma, otteniamo che il linguaggio L non è regolare. Questa è una contraddizione. □

**Macchine di Turing**

Gli automi finiti sono estremamente semplici ed economici da costruire: sono particolarmente adatti ad alcuni compiti importanti, come pattern matching, lavastoviglie, telecomandi, semafori, circuiti sequenziali, …  
Tuttavia, essi non sono sufficientemente “potenti” per risolvere numerosi problemi importanti. Necessitiamo di una macchina più semplice, ma allo stesso tempo potente quanto i computer convenzionali.

**Alan Mathison Turing** (1912 – 1954) è stato un matematico, logico e crittanalista britannico, considerato uno dei padri dell’informatica e uno dei più grandi matematici del Novecento. Introdusse la macchina ideale ed il test omonimi. Fu uno dei più brillanti decrittatori che operavano in Inghilterra, durante la seconda guerra mondiale, per decifrare i codici delle Potenze dell’Asse. Decifrò enigma, il codice usato dai sottomarini tedeschi. Morì mangiando una mela al cianuro, in seguito ad una persecuzione omofobica condotta nei suoi confronti.

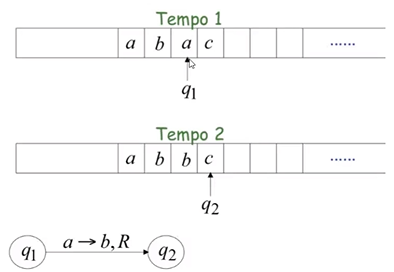
Nel 1936, Alan Turing schematizzò i limiti delle macchine calcolatrici, ponendo le definizioni di quella che sarebbe diventata famosa come “macchina di Turing” (MdT).  
Una **Turing machine** è una macchina a stati finiti con un nastro *semi-infinito*, avente un inizio ma non avente una fine. La testina può muoversi in entrambe le direzioni: essa può leggere e scrivere in ogni cella del nastro, e quando la MdT raggiunge uno stato *accept/reject*, allora accetta/rifiuta immediatamente.  
La MdT è schematizzata come nella figura a sinistra. “Finite control”  
è il diagramma degli stati, ed il nastro “input” ha un inizio e non ha una  
fine. Il simbolo \_ (*blank*) segna la fine dell’input, cioè indica una casella  
vuota.

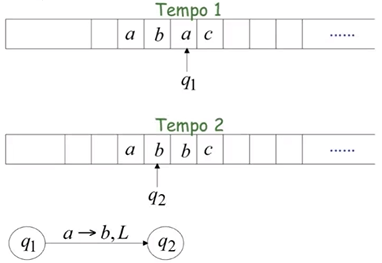
**Def.** Una *Macchina di Turing* è una settupla (Q, Σ, Г, δ, q0, qaccept, qreject), dove:  
- Q è l’**insieme** (finito) **degli stati** della macchina;  
- Σ è l’**alfabeto di lavoro** (\_ ∉ Σ) che la macchina processa;  
- Г (*gamma*) è l’**alfabeto del nastro** (\_ ∈ Г, Σ ⊂ Г), contenente tutti i simboli che possono essere scritti all’interno di una cella di memoria (tecnicamente, l’unione tra l’alfabeto di lavoro e l’insieme dei simboli non processabili dalla macchina, come \_);  
- δ: Q × Г → Q × Г × {*L, R*} è la **funzione di transizione**, dove le lettere lette sono all’interno  
del nastro (in ogni istante si hanno dei simboli nelle varie celle, e la cella a cui punta la testina  
rappresenta lo stato q in cui si trova la macchina). Nell’esempio di cui a destra, la testina (stato  
q) legge y e passa dallo stato q allo stato r: z è la lettera che viene scritta nella cella al posto  
della lettera corrente, e l’ultimo elemento della tripla risultante (L = *left*, R = *right*) è il movimento  
della testina in seguito alla transizione;  
- q0 è lo **stato iniziale**;  
- qaccept è lo **stato di accept**;  
- qreject è lo **stato di reject**.

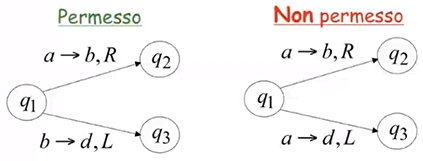
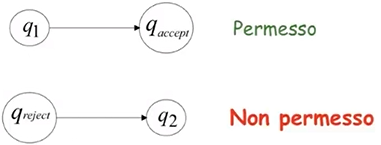
Sia M una Macchina di Turing definita da (Q, Σ, Г, δ, q0, qaccept, qreject); allora:  
- ad ogni istante, M occupa uno degli stati in Q;  
- la testina si trova in un quadrato del nastro contenente un qualche simbolo γ ∈ Г;  
- la funzione di transizione δ: Q × Г → Q × Г × {*L, R*} dipende dallo stato q e dal simbolo di nastro γ.  
Il *range* della funzione di transizione sono triple (q’, γ’, d), con:  
- q’ ∈ Q;  
- γ’ ∈ Г è il simbolo scritto dalla testina sulla cella del nastro su cui la testina si trova *all’inizio* della transizione;  
- d ∈ {L, R} è la direzione in cui la testina muove un passo, con L = *left* ed R = *right*.

**Es.** Dato l’esempio posto a destra, si ha che i passi 1, 2, 3 – per passare dal  
tempo 0 al tempo 1 – sono rappresentati da una funzione di transizione  
δ(q, a) = (r, k, L), con q ed r stati generici per supposizione.

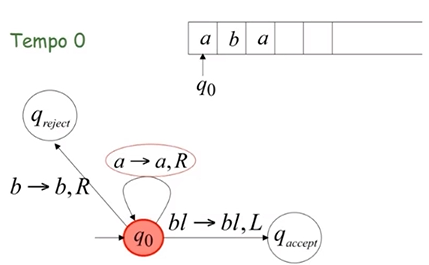
**Es.** Dato l’esempio posto a destra, se la testina puntasse alla seconda  
cella, scrivesse *f* e si muovesse a destra, allora avremmo  
δ(r, b) = (r’, *f*, R), con r ed r’ stati generici per supposizione.

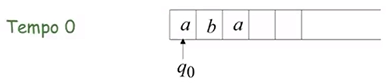
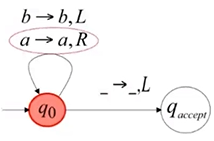
La computazione parte sempre da uno stato iniziale q0, con l’input scritto sul nastro: quest’ultimo inizialmente contiene l’input, ed in particolare l’inizio dell’input è scritto all’inizio del nastro. La testina si trova nella prima cella a sinistra del nastro (cella 0). A questo punto, la macchina è pronta per l’esecuzione, ed effettuerà le transizioni in accordo alla funzione δ.  
**Es.** Con questo esempio, vediamo come la funzione δ può essere  
rappresentata in modo compatto all’interno del diagramma degli stati.  
La figura a destra dice che la testina, al tempo 1, si trova nello stato q1 e sulla  
cella 2, dove è scritto il simbolo ‘a’; al tempo 2, la testina passa allo stato q2 e si  
trova sulla cella 3, dove è scritto il simbolo ‘c’, mentre nella cella dove essa  
puntava al tempo 1 è stato scritto il simbolo ‘b’.  
Ciò è sintetizzabile secondo la transizione in basso nella figura, che equivale  
a δ(q1, a) = (q2, b, R). Con *a* → *b* indichiamo che la nuova lettera scritta  
sarà *b*, ci si muove verso *R* e si passa da *q1* a *q2*.

**Es.** La figura a destra dice che la testina, al tempo 1, si trova nello stato q1 e  
sulla cella 2, dove è scritto il simbolo ‘a’; al tempo 2, la testina passa allo stato  
q2 e si trova sulla cella 1, dove è scritto il simbolo ‘b’, mentre nella cella dove  
essa puntava al tempo 1 è stato scritto il simbolo ‘b’.  
Ciò è sintetizzabile secondo la transizione in basso nella figura, che equivale  
a δ(q1, a) = (q2, b, L). Con *a* → *b* indichiamo che la nuova lettera scritta  
sarà *b*, ci si muove verso *L* e si passa da *q1* a *q2*.

La computazione termina quando *M* raggiunge: uno stato accettazione qaccept (che implica una **Computazione Accept**), o uno stato rifiuto qrejeft (che implica una **Computazione Reject**).  
Parliamo di Macchina di Turing deterministica, in cui *non ci sono  
ε-transition*: nella funzione di transizione, *per ogni stato e per  
ogni lettera esiste uno ed un solo risutato*.  
Nella figura a destra, notiamo che non è permesso che, nello  
stato q1, leggendo *a* si possa scrivere o *b* o *d* e muoversi o verso  
*R* o verso *L*; ciò avviene in quanto abbiamo due possibili mosse  
per δ(q1, a).

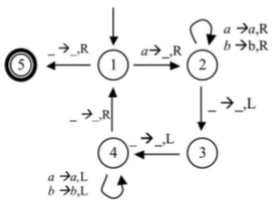
Gli **stati di arresto** non hanno archi uscenti: per definizione, in uno stato  
di arresto (accept o reject) la computazione termina.

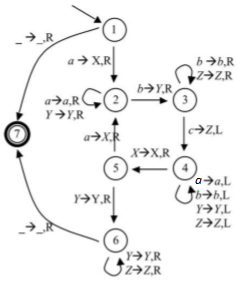
Una macchina di Turing può accettare, non accettare o avere problemi con una stringa.  
**Es.** L’esempio di cui a destra mostra un caso in cui la stringa non viene  
accettata.  
L’alfabeto di lavoro è Σ = {a, b}, mentre l’alfabeto di del nastro è Г = {a, b, \_}.  
Possiamo denotare il blank sia con \_ che con *bl*. Lo stato iniziale è q0: se si  
trova in q0 e legge ‘a’, allora non si cambia il contenuto della cella e la testina  
si sposta a R, e non si ha un cambio di stato; se si trova in q0 e legge ‘b’, allora  
non si cambia il contenuto della cella e la testina si sposta a R, passando allo  
stato qreject; se si trova in q0 e legge ‘*bl*’, allora non si cambia il contenuto della cella e la testina si sposta a L, passando allo stato qaccept. Si noti che la stringa, in questo esempio, viene rifiutata leggendo solo i primi due simboli, per cui non è necessario arrivare a fine stringa per determinare l’accettazione o il rifiuto.  
Se avessimo avuto in input la stringa aaa\_\_, allora la stringa sarebbe stata accettata.

**Es.** Nell’esempio di cui a destra, l’alfabeto di lavoro è Σ = {a, b}, mentre  
l’alfabeto di del nastro è Г = {a, b, \_}. Lo stato qaccept non è raggiungibile,  
in quanto si entra in un loop in cui si leggono sempre i primi due simboli.  
La computazione non termina, di conseguenza l’input non viene accettato.

Ciò mostra che la macchina di Turing può andare in loop: questo è uno dei  
grandi problemi che evidenzia che esistono problemi non risolvibili con un  
calcolatore.

Esiste un’*eccezione* nella funzione di transizione, secondo cui se ci troviamo all’inizio del nastro (prima cella) e leggiamo un simbolo qualsiasi per poi muoverci a sinistra, il risultato sarà sempre che la testina punterà alla prima cella. Nell’esempio precedente, se avessimo avuto in input la stringa bb\_ avremmo ottenuto che la funzione δ(q0, b) = (q0, b, L) non avrebbe spostato la testina, che quindi punterà sempre alla cella corrente (e, in questo esempio, entrerebbe comunque in un loop). Questo non è considerato un errore.

**Es** (Strategia per accettare {anbn | n ≥ 0})**.** Si consideri l’alfabeto Σ = {anbn | n ≥ 0}. Possiamo *cancellare ripetutamente* la prima occorrenza di *a* e l’ultima occorrenza di *b*: se la stringa appartiene all’alfabeto Σ, allora non rimangono simboli. Tutto ciò lo possiamo formulare in cinque passi:  
1. Se leggi \_, vai al punto 5. Se leggi a, scrivi \_ e vai al punto 2;  
2. Spostati a destra (R) di tutti a e b. Al primo \_, muovi a sinistra (L) e vai al  
punto 3;  
3. Se leggi b, scrivi \_ e vai al punto 4;  
4. Spostati a sinistra (L) di tutti a e b. Leggendo \_, muovi R e vai al punto 1;  
5. Accept.  
La figura a destra mostra una macchina di Turing che implementa tale algoritmo.  
Lo stato *reject* manca: spesso, questo non viene spesso considerato per non avere un numero elevato di stati. Prima di costruire la MdT, si può assumere che *tutte le transizioni che non compaiono vanno implicitamente in uno stato, non inserito nel diagramma, di rifiuto*. Ad esempio, notiamo che se siamo nello stato 3 e leggiamo a, allora non c’è una transizione: in questo caso, si assume che la stringa viene rifiutata.

**Es.** La seguente è una MdT per stringhe *anbncn*. Dunque, operiamo sull’alfabeto Σ = {a, b, c}. Inoltre, l’alfabeto del nastro è Г = {\_, a, b, c, X, Y, Z}.  
Il “cammino” computazionale di tale macchina può riassumersi in 4 passi:  
1) cancella la prima *a*. In questo esempio, sovrascriviamo le *a* con una *X*;  
2) avanza (scorrendo la stringa), e cerca la prima occorrenza di *b*. Quando viene  
trovata la *b*, la sovrascriviamo con una *Y*;  
3) avanza (scorrendo la stringa), e cerca la prima occorrenza di *c*. Quando viene  
trovata la *c*, la sovrascriviamo con una *Z*;  
4) ritorna al passo 1).  
Ad esempio, si riceve in input la stringa aabbcc. Al primo passo, si sostituisce la  
prima *a* con una *X*, e ci si sposta a destra; successivamente, si avanza fin quando non  
si trova una *b* (in questo caso, soltanto una volta). Trovata la *b*, la si sovrascrive con *Y*  
e ci si sposta a destra; successivamente, si avanza fin quando non si trova una *c* (in questo caso, soltanto una volta). Trovata la *c*, la si sovrascrive con *Z* e ci si sposta a sinistra. A questo punto, l’iterazione è finita e la testina può iniziare a spostarsi a sinistra per tornare all’inizio del nastro e ad effettuare una nuova iterazione (il passo 4 serve proprio a tornare indietro sul nastro, per qualsiasi simbolo letto a parte una *X*, che inizia a far muovere la testina verso destra).  
Anche in questo esempio, ogni transizione da uno stato ad un altro non inserita, se si verifica porta la macchina in uno stato qreject che rifiuta la stringa.

Una **configurazione** di una Macchina di Turing è una descrizione concisa di stato e contenuto del nastro. Trattasi di una stringa C = *uqv*, dove:  
- *q* è lo *stato* occupato dalla macchina M;  
- *uv* è il *contenuto del nastro* (sinistra – destra);  
- la testina punta sul primo (cioè, più a sinistra) simbolo di *v* (su primo blank \_ se v = ε);  
- dopo *v* sono presenti solo simboli blank \_.  
**Es.** Dato l’esempio precedente (MdT per stringhe *anbncn*), al tempo in cui il nastro contiene la stringa XXYYZZ e punta alla seconda cella (cella 1, contenente l’ultimo X), la configurazione di MdT in questo tempo è  
*C = X4XYYZZ*, dove *u* = X, *q* = 4, *v* = XYYZZ.  
A partire da questa configurazione C, sapendo che δ(4, X) = (5, X, R), possiamo ottenere la configurazione successiva *C’ = XX5YYZZ*, dove *u* = XX, *q* = 5, *v* = YYZZ.

In generale, la configurazione cambia ad ogni mossa. Si dice che C1 *produce* C2 (si indica con C1 → C2) se una mossa della MdT può far andare la macchina da C1 a C2.

**Es.** Se *a*, *b*, *c* ∈ Г (cioè, sono simboli), *u*, *v* ∈ Г\* (cioè, sono stringhe), *qi*, *qj* ∈ Q (cioè, sono stati),  
allora *uaqibv* → *uqjacv* se δ(qi, b) = (qj, c, L) è una *mossa a sinistra*,  
oppure  *uaqibv* → *uacqjv* se δ(qi, b) = (qj, c, R) è una *mossa a destra*.

Di seguito alcuni casi particolari.  
Data una configurazione *qibv* → *qjcv*, se la testina è ad inizio nastro e la prossima è una *mossa a sinistra*, allora la posizione della testina non cambia (senza che la Macchina di Turing se ne accorga).  
La configurazione *uaqi* è equivalente a *uaqi\_*, cioè la parte vuota del nastro viene riempita con simboli blank (\_).

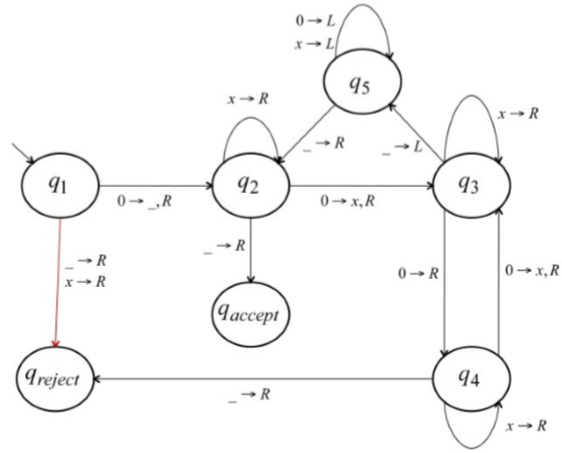
Una configurazione di M si dice **di start** su un input *w* (*q0w*) se e solo se lo stato *q0* è lo stato iniziale, il nastro contiene *w* e la testina è posizionata sulla prima cella del nastro.  
Una configurazione di M si dice **di accettazione** (Accept) se essa raggiunge uno stato qaccept.  
Una configurazione di M si dice **di rifiuto** (Reject) se essa raggiunge uno stato qreject.  
Una configurazione di M si dice **di Halt** se essa raggiunge o uno stato qreject o uno stato qaccept (cioè, una qualsiasi configurazione Accept o Reject).

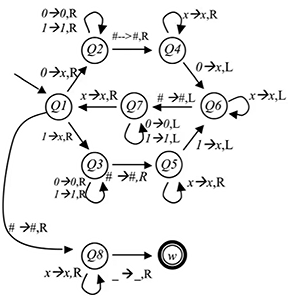
Una Macchina di Turing M **accetta** una parola *w* se esiste una computazione (*sequenza di configurazioni*) di M del tipo C1, C2, …, Ck tale che:  
1. C1 = *q0w* è la configurazione iniziale di M con input *w*;  
2. Ci → Ci+1 per ogni i = 1, 2, …, k - 1;  
3. Ck è la configurazione di accettazione (Accept) di M.  
Una Macchina di Turing M **rifiuta** una parola *w* se esiste una computazione (*sequenza di configurazioni*) di M del tipo C1, C2, …, Ck tale che:  
1. C1 = *q0w* è la configurazione iniziale di M con input *w*;  
2. Ci → Ci+1 per ogni i = 1, 2, …, k - 1;  
3. Ck è la configurazione di rifiuto (Reject) di M.

Una Macchina di Turing M può raggiungere tre *risultati computazionali*:  
1. M *accetta* – se si ferma in *qaccept*;  
2. M *rifiuta* – se si ferma in *qreject*;  
3. M ***cicla/loop*** – se non si ferma mai.  
Mentre M funziona, non si può dire se essa è in loop, in quanto si potrebbe fermare in seguito oppure no.

Data una Macchina di Turing M, il **linguaggio di M** è l’insieme delle stringhe che M accetta, e viene denotato con L(M).  
Un linguaggio si dice **Turing riconoscibile** se esiste una macchina di Turing che lo riconosce. Formalmente, un linguaggio L si dice Turing riconoscibile se esiste una Macchina di Turing M tale che L(M) = L. Di conseguenza, la macchina accetta tutte le stringhe del linguaggio. Se, invece, una stringa w ∉ L, allora la stringa può essere o *rifiutata* o mandare la macchina *in loop*.

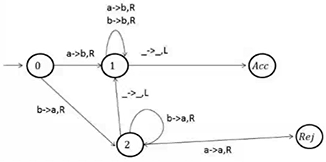
Abbiamo visto che è difficile dire se una MdT è in loop. Possiamo evitare questo problema costruendo i macchine che si fermano (accettando o rifiutando) *su ogni input*: trattasi dei **deciders**.  
Si dice che un decider *decide* il linguaggio L se esso riconosce L.  
Un linguaggio si dice **Turing decidibile** se esiste una MdT che lo decide. Formalmente, un linguaggio L si dice Turing decidibile se esiste una Macchina di Turing M tale che L(M) = L e M è un decider.  
La differenza tra un linguaggio L Turing riconoscibile e Turing decidibile risiede nel fatto che i primi possono mandare le MdT che li riconoscono in loop, mentre i secondi possono far sì che le MdT che li decidono o accettino o rifiutino le loro stringhe *su ogni input*.

**Es.** Consideriamo il linguaggio L = { | n > 0} dell’insieme di stringhe di 0 la cui lunghezza è potenza di 2. Vogliamo costruire una MdT M2 che lo decide.  
Nota. Il linguaggio non è regolare (*ciò è dimostrabile mediante Pumping Lemma*).  
🡪 Si noti che *n* è potenza di 2 se e solo se ripetute divisioni per 2 danno resto 1.  
Sia *w* l’input; allora, un algoritmo tale che M2 decida tale linguaggio può essere il seguente:  
1. Scorrere il nastro da sinistra a destra cancellando ogni SECONDO 0; *//divide per 2 il numero di 0*  
2. Se rimane solo uno 0, allora ACCEPT;  
3. Se rimane un numero di 0 dispari ≥ 3, allora REJECT; *//in quanto il numero di 0 non è potenza di 2*  
4. Se rimane un numero pari di 0, allora riporta la testina all’inizio del nastro;  
5. Ritorna al passo 1.  
Allora, l’implementazione della MdT M2 può essere la seguente.  
Si noti che spesso si utilizza la forma contratta *x* → *R*, che equivale  
a *x* → *x, R* (cioè, quando si sovrascrive lo stesso carattere letto).  
L’alfabeto della macchina è Σ = {0}. Inoltre, l’alfabeto del nastro è  
Г = {0, x, \_}.  
Ad esempio, sia *w* = 0000. Quando ci si trova nello stato iniziale q1,  
con un *passo tecnico* si sostituisce (*il primo*) 0 con \_, per poi passare  
alla cella successiva (ed allo stato q2). Passati allo stato q2, si  
sostituisce (*il secondo*) 0 con x e si passa alla cella successiva (ed allo  
stato q3). Nello stato q3, si lascia (*il terzo*) 0 invariato e si passa alla  
cella successiva (ed allo stato q4). Nello stato q4, si sostituisce (*il  
quarto*) 0 con x e si passa alla cella successiva (ed allo stato q3). Ora,  
la testina punta su \_, quindi abbiamo scandito tutta la stringa in input; dobbiamo iterare di nuovo: si sposta la testina all’indietro (passando allo stato q5, dove per ogni simbolo letto si sposta la testina a sinistra, fino ad arrivare all’inizio) fin quando si riconosce il simbolo \_ (questo è il motivo per cui abbiamo utilizzato un carattere diverso, per simboleggiare l’inizio della stringa), tornando allo stato q2: qui siamo pronti ad una nuova iterazione.  
La testina si sposta fin quando non trova uno 0 (in questo caso, *il terzo*), che viene sostituito con x e si muove la testina a destra (e si passa allo stato q3). Nello stato q3, con le x si avanza: si arriva alla fine del nastro, quindi si legge il simbolo \_ e si torna in q5, dove si torna all’inizio della stringa e si passa quindi a q2 (muovendo la testina a destra, quindi al secondo simbolo della stringa, cioè la prima x); tornati in q2, infine, si scandisce ogni simbolo x sul nastro e si arriva alla fine della stringa: si legge, quindi, \_ e si passa allo stato qaccept, e quindi la stringa 0000 viene *accettata*. □

**Es.** Consideriamo il linguaggio L = {w#w | w ∈ {0, 1}\*}. Di seguito un’idea per verificare se una stringa s ∈ L:  
1. Leggi il primo carattere;  
2. Memorizzalo e cancellalo;  
3. Cerca # e guarda il carattere successivo;  
4. Se esso è uguale al carattere memorizzato, allora cancellalo;  
5. Ritorna all’inizio al primo carattere non cancellato;  
6. Ripeti 1-5 fino a considerare tutta la stringa in input: se si trova qualcosa di “inatteso”, allora REJECT; altrimenti, ACCEPT.  
Vediamo, ora, come costruire una MdT M1 per il linguggio L. Sia definita in input una stringa w; allora, M1 deve:  
1. Memorizzare il simbolo più a sinistra e cancellarlo (scriviamo x);  
2. Avanzare sul nastro fino a superare #, se non si trova allora REJECT;  
3. Confronta il primo simbolo diverso da x con il simbolo memorizzato, se è diverso allora REJECT;  
4. Se è uguale, cancella il simbolo confrontato e ritorna a inizio nastro;  
5. Vai al punto 1.  
Vediamo, ora, la funzione di transizione per M1. Si ha che Σ = {0, 1, #}, e Г = {0, 1, #, x, \_}. Nella descrizione, lo stato qreject e tutte le transizioni in ingresso sono state omesse. Ovunque vi sia una transizione mancante, va in qreject.  
Nota. Abbiamo utilizzato il termine “memorizzare”: le MdT non possono  
memorizzare simboli in quanto non hanno memoria, ma può essere  
ottenuta una memorizzazione progettando il diagramma degli stati in un  
modo preciso: in particolare, si useranno stati diversi a seconda che l’input  
sia un 1 o uno 0. A tal scopo, gli stati q2 e q3 memorizzano il bit 0, mentre  
gli stati q4 e q5 memorizzano il bit 1.  
In altre parole, questi due segmenti sono identici, e in ogni segmento si  
utilizza il valore memorizzato.

\*Esempio del funzionamento alle slide 116 – 148.\*

Le Macchine di Turing possono avere le proprie funzioni di transizione rappresentate utilizzando una **tabella**. Data una funzione di transizione generica δ: Q × Г → Q × Г × {*L, R*}, è possibile costruire una tabella di 5 colonne ed n righe (le n righe rappresentano le etichette complessive della funzione), dove:  
- la prima colonna rappresenta lo stato attuale (qi ∈ Q);  
- la seconda colonna rappresenta il simbolo letto (s ∈ Г);  
- la terza colonna rappresenta il nuovo stato (qj ∈ Q);  
- la quarta colonna rappresenta il simbolo da scrivere (t ∈ Г);  
- la quinta colonna rappresenta il movimento della testina ({L, R}).  
Ad esempio, data la tabella a lato otteniamo che:  
- la riga 4 rappresenta la funzione δ(q1, 0) = (q1, 0, L);  
- la riga 6 rappresenta la funzione δ(q1, \_) = (q2, \_, R);  
- …

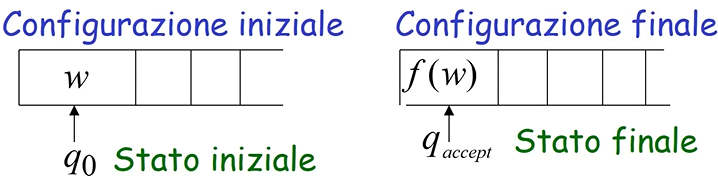
**Esercizio.** Data la macchina in figura, fornire la configurazione iniziale e le computazioni su input *b* e *bba*.  
🡪  
b) La configurazione iniziale della macchina in figura è ***0b***.  
A questo punto, essendo nello stato 0, leggendo *b* la testina scrive  
*a* ed essa punterà alla cella successiva, che ha il carattere \_. Quindi,  
la configurazione *0b* produce la configurazione *a2\_* (***0b → a2\_***).  
Nello stato 2, leggendo \_, la testina si sposta a sinistra (quindi, punterà  
ad *a*) e si va nello stato 1: quindi, la configurazione *a2\_* produce la  
configurazione *1a* (***a2\_* *→ 1a***).  
Nello stato 1, leggendo *a* si resta nello stato 1 e si scrive *b* e ci si sposta a destra (quindi, si punterà a \_), di conseguenza la configurazione *1a* produce la configurazione *b1\_* (***1a → b1\_***).  
Nello stato 1, leggendo \_ si va nello stato di accettazione e ci si sposta a sinistra (quindi, si punterà a *b*), di conseguenza la configurazione *b1\_* produce la configurazione *Accb* (***b1\_ → Accb***).  
Dunque, la stringa *b* viene accettata con queste configurazioni:  
0b → a2\_ → 1a → b1\_ → Accb. **✓**

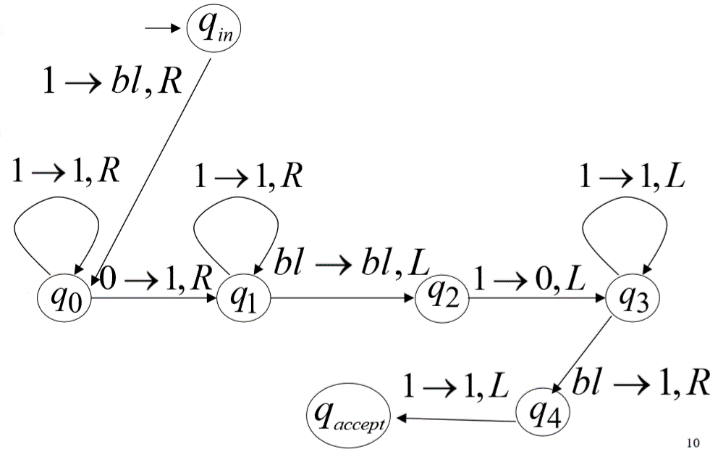
bba) La configurazione iniziale della macchina in figura è ***0bba***.  
A questo punto, essendo nello stato 0, leggendo *b* si passa allo stato 2 e la testina scrive *a* e si sposta alla cella a destra (che ha il carattere *b*). Quindi, la configurazione *0bba* produce la configurazione *a2ba* (***0bba → a2ba***).  
Nello stato 2, leggendo *b* si resta nello stato 2 e la testina scrive *a* e si sposta alla cella a destra (che ha il carattere *a*). Quindi, la configurazione *a2ba* produce la configurazione *aa2a* (***a2ba → aa2a***).  
Nello stato 2, leggendo *a* si passa allo stato di rifiuto e la testina si sposta a destra (che ha il carattere \_). Quindi, la configurazione *aa2a* produce la configurazione *aaaRej\_* (***aa2a → aaaRej\_***).  
Dunque, la stringa *bba* viene rifiutata con queste configurazioni:  
0bba → a2ba → aa2a → aaaRej\_. □

In queste tipologie di esercizi, è possibile che una macchina vada in loop: per accorgersene, la macchina entra in una configurazione già incontrata in precedenza.

**Calcolare funzioni con Macchine di Turing**

Sia definita una funzione f: D → S che opera su stringhe; cioè, data una stringa w ∈ D si ottiene un’altra stringa f(w) ∈ S.  
Una funzione può dipendere da più variabili. Ad esempio, la funzione addizione f(x, y) = x + y è una funzione a due variabili.  
Consideriamo come l’insieme degli interi. Se usiamo la notazione decimale e vogliamo dare in output 5, allora il simbolo della MdT sarà 5 (con alfabeto Σ = {0, 1, …, 9}. Se usiamo la notazione binaria e vogliamo dare in output 5, allora il simbolo della MdT sarà 101 (con alfabeto Σ = {0, 1}). Noi useremo la *notazione unaria*, avente come alfabeto Σ = {1}, che dà in output tanti 1 quant’è il valore del numero in input (es. 5 → 11111): questa notazione è più semplice, anche se non è molto efficace.

**Idea.** Una funzione *f* si dice *calcolabile* se esiste una macchina di Turing *M* tale che, per ogni w ∈ D, essa parte con un input w in uno stato iniziale q0 e termina in uno stato finale qaccept avendo sul nastro il valore f(w).

**Es.** La funzione è calcolabile. Infatti, dati  
x, y interi, allora possiamo costruire una Macchina di Turing  
che calcola questa funzione: ad una stringa in input *x0y*  
(unario) è associata una stringa in output *xy0* (unario). Lo 0  
viene usato all’inizio semplicemente per separare le due  
stringhe, mentre viene usato al termine per eseguire  
eventualmente altre operazioni.  
Ad esempio, dato l’input 2 + 2 (11011) ricaviamo l’output 4  
(1111).  
A destra è posta la MdT che esegue l’operazione appena vista.

È importante che, al termine del calcolo dell’output, la testina che si trova alla fine torni all’inizio del nastro per andare nella situazione finale (qaccept).

\*Esempio del funzionamento alle slide 11 – 24.\*

**Es.** La funzione è calcolabile. Infatti, dato un numero x intero, allora possiamo costruire una Macchina di Turing che calcola questa funzione: ad una stringa in input *x* (unario) è associata una stringa in output *xx* (unario).  
Si scriva, per esercizio, la Macchina di Turing corrispondente.

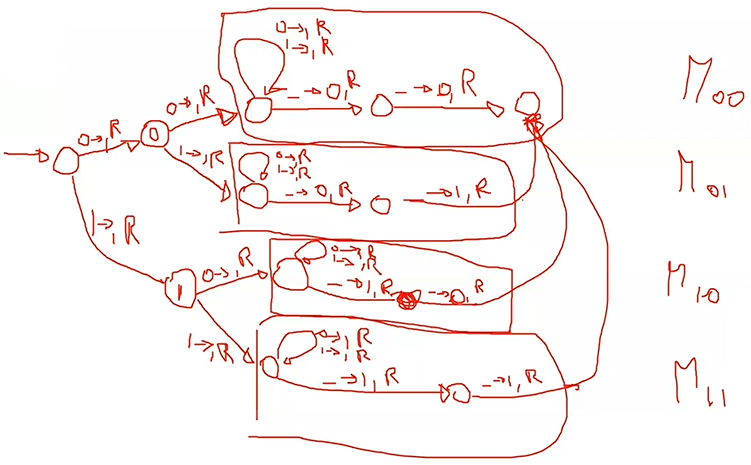
**Es.** La funzione è calcolabile. La MdT, ad esempio, parte con un input 111011\_ e dà in output 1\_ (in quanto 111 = 3 > 2 = 11). Dunque, bisogna vedere se il numero di 1 prima dello 0 è maggiore rispetto al numero di 1 dopo lo 0.  
Si scriva, per esercizio, la Macchina di Turing corrispondente.

**Varianti delle Macchine di Turing**

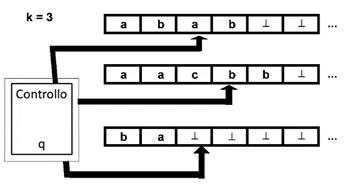
Esistono definizioni alternative di Macchina di Turing, chiamate *varianti*. Tra queste, vedremo delle MdT a più nastri e MdT non deterministiche. Mostreremo, inoltre, che tutte le varianti “ragionevoli” hanno la stessa capacità computazionale.  
Quando proviamo che una MdT ha una certa proprietà, non ci poniamo domande sulla grandezza di tale macchina o su quanto è complesso programmarla. Per il momento, ci interessano solo alcune proprietà essenziali.

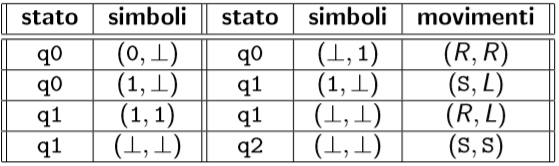
Uno **stayer** è una Macchina di Turing la cui testina può rimanere sulla stessa cella del nastro durante una transizione; formalmente, essa è la stessa settupla M = (Q, Σ, Г, δ, q0, qaccept, qreject), con la sola differenza che la funzione di transizione è δ: Q × Г → Q × Г × {*L, S, R*}, dove **S** indica che la testina resta ferma durante la transizione (S sta per “stay”).  
*Diciamo che il potere computazionale di due modelli è lo stesso se riconoscono la stessa classe di linguaggi*.  
Una MdT può essere facilmente simulata da uno stayer (basta banalmente non usare la possibilità di stare nella funzione di transizione), per cui lo stayer include, per definizione, una MdT semplice. Da ciò consegue che il potere computazionale di uno stayer è almeno pari al potere computazionale di una MdT convenzionale.  
Resta da provare il seguente.  
**Corollario.** Il potere computazionale di una MdT è almeno pari al potere computazionale di uno stayer ⇔ per ogni stayer esiste una MdT che riconosce lo stesso linguaggio.  
DIM. Assumiamo che M sia uno stayer, e mostriamo un MdT M’ che simula M.  
M’ è definita esattamente come M, tranne che ogni transizione S è sostituita da due transizioni in M’:  
1. la prima porta M’ in uno stato speciale (addizionale) e muove la testina a destra (R);  
2. la seconda ritorna allo stato originale muovendo la testina a sinistra (L).  
Quindi, M e M’ riconoscono lo stesso linguaggio. □

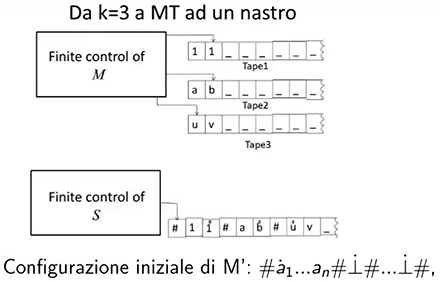
In generale, quando vogliamo provare che due varianti di MdT hanno lo stesso potere computazionale, mostriamo che possiamo simulare una con l’altra.

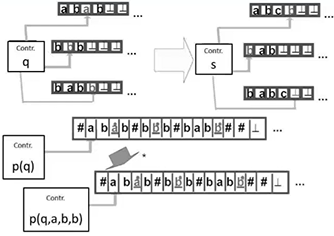
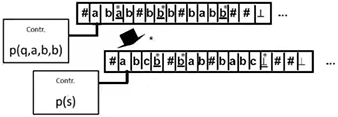
In alcune occasioni avremo bisogno di **memorizzare** l’informazione letta sul natro: *possiamo usare gli stati per memorizzare informazioni*. L’approccio è simile a quello visto nell’esempio sulle MdT convenzionali, dove per memorizzare il bit 0 andavamo in una parte della macchina, mentre per memorizzare il bit 1 andavamo in un’altra parte della macchina.  
**Es.** Supponiamo di avere una MdT M che deve leggere i primi due bit letti sul nastro e scriverli alla fine dell’input (al posto dei due \_ più a sinistra). Un’idea può essere la seguente:  
- costruiamo una MdT M00 che legge i primi due bit, cerca la fine dell’input e scrive 00 alla fine dell’input;  
- replichiamo il passo precedente per tutte le possibili coppie (in questo caso M01, M10 e M11);  
- M: dopo aver letto i primi due bit in input, si sposta sulla replica corrispondente ai due bit letti (e quindi li scrive alla fine dell’input).  
Notiamo che in base ai primi  
due bit in input si va in una  
determinata “sottomacchina”.  
Poiché i primi due bit, sull’  
alfabeto {0, 1}, possono dare  
22 combinazioni diverse, si  
costruiscono 4 diverse parti di  
macchina che operano in  
dipendenza dei bit letti.

Nota. Questa tecnica è utilizzabile *in generale*.  
Se vogliamo che M memorizzi una sequenza di k simboli, possiamo:  
- avere una replica per ogni possibile sequenza di k simboli;  
- M si sposta sulla replica che corrisponde alla sequenza mentre la legge.  
Questo implica che sono necessari circa |Σ|k stati aggiuntivi. Infatti, se |Σ|k = c, allora avremo c (stati dopo aver letto il primo simbolo) + c2 (per ogni stato precedente, ci sono altri c stati) + … + ck = c + c2 + … + ck = O(ck).

Una MdT **multinastro** (a k nastri) è una normale Macchina di Turing avente k nastri. Inizialmente, l’input compare sul primo nastro e gli altri sono vuoti. La sua funzione di transizione è δ: Q × Гk → Q × Гk × {*L, S, R*}k; dunque, essa muove (in modo indipendente) le testine dei vari nastri.  
Questa variante è simile ad una MdT convenzionale: usa k nastri, con k ≥ 1. Ciascun nastro ha una propria testina, e con una mossa si specificano (oltre al nuovo stato): i k simboli letti, i k simboli da scrivere e i k movimenti delle k testine. Come configurazione iniziale, essa avrà l’input sul primo nastro ed i rimanenti nastri saranno vuoti.  
Nella figura a destra, ad esempio, abbiamo una MdT a 3 nastri, dove  
in quel momento lo stato occupato dalla macchina è q, la prima testina  
punta alla cella 2 (a), la seconda testina punta alla cella 3 (b), la terza  
testina punta alla cella 2 (\_). In questo caso, quindi, la funzione di  
transizione δ(q, (a, b, \_)) ci restituirà un nuovo stato, tre simboli che  
saranno scritti in quelle celle, e i movimenti delle tre testine.

In maniera compatta, usiamo l’espressione δ(qi, a1, …, ak) = (qj, b1, …, bk, D1, …, Dk) per indicare che se la MdT a k nastri M si trova nello stato qi, e la testina i legge ai (per i = 1, …, k), allora M va nello stato qj, la testina scrive bi e si muove nella direzione Di ∈ {L, S, R} (per i = 1, …, k).  
**Es.** Premessa: spesso, si userà il simbolo ⟂ per identificare il blank (\_).  
Questa macchina MdT a 2 nastri per 0n1n, la cui funzione di transizione è descritta dalla tabella a lato:  
1. Scorre il primo nastro verso destra fino al primo 1: per ogni 0,  
scrive un 1 sul secondo nastro;  
2. Scorre il primo nastro verso destra e il secondo nastro verso  
sinistra: se i simboli letti sono diversi, allora termina in uno stato  
non finale;  
3. Se legge \_ su entrambi i nastri, allora termina in uno stato finale.

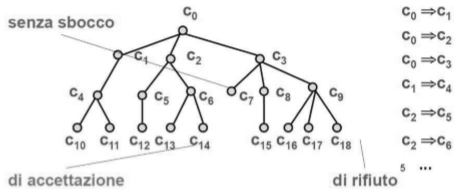
Le MdT multinastro sembrerebbero più potenti delle MdT ordinarie. In realtà, possiamo dimostrare che le due varianti sono equivalenti.  
**Teorema.** *MdT e MdT multinastro sono modelli equivalenti*.  
DIM. L’implicazione diretta è ovvia, in quanto una MdT è una MdT multinastro, con k = 1. Vogliamo dimostrare che per ogni MdT multinastro è possibile costruire una MdT convenzionale che riconosce lo stesso linguaggio. A tal scopo, effettuiamo una simulazione. Utilizziamo come esempio la MdT multinastro, con k = 3, definita in precedenza.  
Una soluzione è immaginare i k nastri affiancati, ognuno con indicata la posizione della testina. Dobbiamo codificare questa informazione su un solo nastro. Per fare ciò, quindi, possiamo concatenare il contenuto dei k nastri, su k blocchi consecutivi separati da un carattere particolare (in questo esempio, #):  
- ogni blocco avrà lunghezza variabile che dipende dal contenuto del nastro corrispondente;  
- un elemento marcato (con ˙) nel blocco i-esimo indica la posizione della testina i-esima (ad esempio, se la testina S punta ad un elemento del primo blocco e legge γ˙, allora la testina del primo nastro è in questa posizione e legge γ);  
- usiamo un alfabeto esteso Г2 tale che γ˙ ∈ Г2 per ogni γ ∈ Г.  
Nella figura a destra, notiamo che il primo nastro (input 11) punta alla  
cella 1 (1), il secondo nastro (input ab) punta alla cella 1 (b), il terzo  
nastro (input uv) punta alla cella 0 (u). Sotto è posta la macchina M’  
convenzionale.

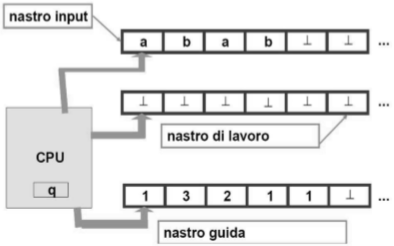
*Per ogni istruzione della MdT multinastro M, la MdT M’:  
1. scorre i k nastri e “raccoglie” informazioni sui k simboli letti;  
2. applica la transizione, scorrendo i k nastri e applicando su ciascuno scrittura e spostamento.*A lato è posto un esempio, dove si applica la transizione δ(q, a, b, b) = (s, c, a, c, R, L, R). Nella macchina ad un nastro, prima legge il contenuto dei nastri (quindi, sa che sul primo  
nastro viene letta a, sul secondo viene letta b, sul terzo viene letta  
b); ora, sa che la la mossa è δ(q, a, b, b) = (s, c, a, c, R, L, R), per cui  
si dovrà scrivere:  
nastro 1) c al posto di a˙, per poi spostarsi a destra (quindi, si dovrà  
sostituire la b in cella 4 con una b˙);  
nastro 2) a al posto di b˙, per poi spostarsi a sinistra (quindi, si dovrà  
sostituire la b in cella 6 con una b˙);  
nastro 3) c al posto di b˙, per poi spostarsi a destra (quindi, si dovrà  
sostituire il # in qualche modo).  
Nella parte finale della figura notiamo il cambiamento: nel nastro 3,  
abbiamo shiftato a destra tutti i caratteri dopo la c, ed inserito un ⟂˙  
per indicare che quella è la fine del nastro. Questo è possibile, dato  
che la lunghezza del nastro di una MdT è variabile nel tempo.

Per ogni mossa del tipo δ(qi, a1, …, ak) = (qj, b1, …, bk, D1, …, Dk):  
- S scorre il nastro verso destra, fino al primo ⟂, memorizzando nello stato i simboli marcati sui singoli nastri. Si ricordi che la memorizzazione viene implementata con uno stato aggiuntivo per ogni possibile sequenza di ≤ k simboli di Г;  
- la testina si riposiziona all’inizio del nastro;  
- poi il nastro viene scorso di nuovo, eseguendo su ogni sezione (cioè un nastro di M) le azioni che simulano quelle delle testine di M, ossia scrittura e spostamento.  
Durante il secondo passaggio, S scrive su tutti i simboli “marcati”, e sposta il marcatore alla nuova posizione della testina corrispondente di M.

Nota. Se una testina di M muove su ⟂ più a sinistra del suo nastro, allora la testina virtuale (il puntino su un simbolo) sul segmento corrispondente del nastro di S si sposta sul delimitatore #. In questo caso, S deve spostare di una posizione tutto il suffisso del nastro a partire da #˙ fino all’ultimo # a destra. Dopo, S scrive ⟂˙ nella prima posizione soggetta a shift (cioè, dove c’era #˙).

Riagganciandoci alla descrizione a inizio pagina,  
per ogni istruzione di MdT multinastro M, la MdT S:  
1. scorre i k nastri e “raccoglie” informazioni sui k simboli letti;  
2. applica la transizione, scorrendo i k nastri e applicando su ciascuno scrittura e spostamento;  
3. *infine la MdT entra nello stato che ricorda il nuovo stato di S e riposiziona la testina all’inizio del nastro*.  
Questo comporta molti più stati e mosse, ma otteniamo comunque che S simula M. □

Una MdT **non deterministica** (NMdT) è una macchina di Turing convenzionale avente funzione di transizione δ: Q × Г → P(Q × Г × {*L, R*}), dove P è l’*insieme potenza* di tale prodotto cartesiano, invece di δ: Q × Г → Q × Г × {*L, R*}; il non determinismo, anche in questo caso, permette di avere associato ad uno stato e ad un simbolo letto sul nastro più transizioni: per ogni transizione, la computazione proseguirà autonomamente in parallelo.  
La computazione di una NMdT è ben descritta da un albero contenente ogni configurazione raggiungibile dalla iniziale (c0) su un dato input: se un cammino (da c0) raggiunge lo stato accetta, allora la NMdT accetta l’input, anche se altri cammini raggiungono uno stato reject.  
Siccome l’insieme potenza contiene anche il vuoto, è possibile che  
una computazione si fermi a seguito di input incorretto: nella figura  
a lato, ad esempio, c7 non produce alcuna nuova configurazione.  
In questo caso, la macchina si ferma dopo tre passi: avendo  
raggiunto una configurazione di accettazione (c14), la stringa in  
input verrà accettata, indipendentemente da ciò che accade sugli  
altri cammini.

Le NMdT sembrerebbero più potenti delle MdT ordinarie. In realtà, possiamo dimostrare che le due varianti sono equivalenti.  
**Teorema.** *MdT e NMdT sono modelli equivalenti.*  
DIM. In un verso è ovvio, in quanto una MdT è anche una NMdT, per la definizione stessa.  
Dimostriamo, ora, che se abbiamo un linguaggio riconosciuto da una NMdT allora esso è riconosciuto anche da una MdT. A tal scopo, guardiamo alla computazione di una NMdT N come a delle configurazioni raggiungibili da quella iniziale c0 su input w: esaminiamo l’albero della figura precedente, e simuliamo questo comportamento seguendo l’albero delle configurazioni, eseguendo tutte le computazioni.  
Visitiamo l’albero con strategia *BFS* – Breadth-First Search (cioè, *ricerca in ampiezza*) – ed eseguiamo un ramo per volta fin quando non lo esaminiamo per intero. Partendo dalla configurazione iniziale c0 andiamo in c1, c2 e c3; al secondo step, torniamo al primo nodo e visitiamo tutti i nodi che esso produce (c4), torniamo al secondo nodo e visitiamo tutti i nodi che esso produce (c5 e c6), torniamo al terzo nodo e visitiamo tutti i nodi che esso produce (c7, c8, c9); proseguiamo in questo modo fin quando non raggiungiamo una configurazione di accettazione (o di rifiuto).  
Formalmente, per ogni input w:  
- M deve eseguire tutte le possibili computazioni di N su w;  
- M deve accettare se e solo se almeno una raggiunge lo stato accetta.  
Utilizziamo una BFS dell’albero delle computazioni (eseguendo tutte le simulazioni in contemporanea), in quanto alcune delle computazioni di N possono essere infinite, e ciò implica che:  
- alcuni cammini sono infiniti;  
- se M esegue la simulazione e segue un cammino infinito, va in loop (anche se su un altro cammino raggiungerebbe lo stato accetta).  
L’algoritmo specificato in precedenza è formalizzato come segue:  
1. Esegui il primo passo di ogni computazione. Se almeno una accetta, allora accetta;   
2. Esegui il secondo passo di ogni computazione. Se almeno una accetta, allora accetta;  
…  
i. Esegui il passo i-esimo di ogni computazione. Se almeno una accetta, allora accetta.  
Questa simulazione la facciamo mediante una MdT D a 3 nastri. Inizialmente, il nastro 1 contiene l’input di N e gli altri due sono vuoti; la simulazione di N procede come segue:  
1. Copia l’input sul nastro 2 (usato per eseguire la computazione);  
2. Esegui un prefisso di questa computazione, usando il contenuto del  
nastro 3. Se si raggiunge una conﬁgurazione accetta, allora accetta l’input;  
3. Aggiorna il contenuto del nastro 3 per ottenere la successiva computazione  
(si incrementa il contenuto di 1, fin quando non si arriva al numero b, dato  
che la cifra è b-aria);  
4. Vai al passo 1.  
Per gestire il contenuto del nastro 3, formalmente, sappiamo che N ha più  
possibili transizioni da una stessa configurazione:  
- per ogni configurazione di N, la MdT D codifica tutte le possibili transizioni e le enumera;  
- sia COMPi il prefisso lungo *i* di una computazione di N (dove *i* rappresenta il numero di passi da eseguire). Codifichiamo COMPi con stringa b1…bi di *i* simboli, dove b è il massimo numero di transizioni possibili da qualsiasi configurazione. Allora:  
 - D parte con la configurazione iniziale;  
 - b1 è il numero della transizione al primo passo di COMPi;  
 - per 2 ≤ j ≤ i, bj è il numero della transizione attuale al passo j-esimo di COMPi.  
Ad esempio, la stringa 421 codifica un prefisso di lunghezza 3 (quindi, b = 3) in cui:  
- al primo passo, N esegue la quarta transizione nella lista delle transizioni possibili (dalla configurazione iniziale);  
- al secondo passo, N esegue la seconda transizione nella lista delle transizioni possibili (dalla configurazione raggiunta al passo 1);  
- al terzo passo, N esegue la prima transizione nella lista delle transizioni possibili (dalla configurazione raggiunta al passo 2).  
Nota. Alcune configurazioni possono ammettere meno di b scelte, quindi non tutte le sequenze b-arie rappresentano un prefisso di computazione.  
Riassumendo, la simulazione di N con D risulta:  
1. Scrivi 1 sul nastro 3;  
2. Copia input sul nastro 2;  
3. Se il prefisso del numero b-ario è la codifica di un prefisso di una computazione di N, allora eseguilo. Se la computazione accetta, allora accetta, altrimenti vai al passo 4;  
4. Incrementa di 1 il numero b-ario sul nastro 3 (con i numeri che vanno da 1 a b);  
5. Vai al passo 2. □

Non abbiamo detto *come* funziona, in quanto la macchina deterministica diventa molto complicata da costruire. Non diamo maggiori dettagli, ma ci limitiamo a dire come dovrebbe funzionare: questo perché a noi basta sapere che una tale macchina esiste.

Nella metà del Novecento, vari gruppi di matematici pensavano al concetto di **computabilità**:  
- Alonzo Church (matematico americano) creò un metodo per definire le funzioni computabili, *λ-calcolo*;  
- Alan Turing creò la *Macchina di Turing*;  
- Church, Kleene e Rosser diedero la definizione formale di una classe di funzioni il cui valore può essere ottenuto mediante *ricorsione*.  
Si è dimostrato che questi tre modelli di calcolo sono tra loro equivalenti. A noi basta conoscere l’equivalenza tra MdT e calcolatori (programmi o algoritmi).  
**Tesi di Church-Turing.** “*Se esiste un algoritmo per eseguire un calcolo, allora questo calcolo può essere eseguito da una MdT (o equivalenti).*”  
Da questa Tesi ricaviamo la sua negazione: infatti, ciò è equivalente a dire che se non esiste una MdT allora non esiste un algoritmo. In seguito vedremo, basandoci su questo fatto, che esistono problemi per cui non è possibile trovare un algoritmo che risolve quel problema: cioè, esistono problemi che *non sono computabili*.

**Problemi di decisione**

I **problemi di decisione** sono problemi che hanno come soluzione una risposta SI o NO.  
**Es.** - PRIMO: Dato un numero *x*, *x* è primo?  
**Es.** - CONNESSO: Dato un grafo *G*, *G* è connesso? Si ricordi che un grafo di dice *connesso* se per ogni coppia di nodi esiste un cammino che li collega all’interno del grafo.  
**Es.** - ACCETTAZIONE DI UN DFA: Dato un DFA *ß* e una stringa *w*, l’automa *ß* accetta *w*?

Se si vuole risolvere un problema di decisione utilizzando una MdT, in qualche modo occorre trasformare il problema in un linguaggio: questo perché le MdT sono essenzialmente accettatori di linguaggi.   
Ricorda: l’input per una MdT è sempre una stringa. Se vogliamo dare in input altri oggetti, questi devono essere codificati come stringhe: ciò rappresenta il primo step per effettuare questa trasformazione. Qualsiasi oggetto, infatti, può essere codificato come stringa.

Rappresenteremo, quindi, i problemi di decisione mediante linguaggi.  
**Es.** Il linguaggio che rappresenta il problema “PRIMO” è  
dove *〈x〉* denota una rappresentazione sotto forma di stringa su un alfabeto Σ dell’oggetto *x* (*codifica*).  
Utilizzeremo sempre la notazione **〈x〉** per indicare una tale rappresentazione. Quindi, se *x* è un intero, ed usiamo come alfabeto Σ = {0, 1}, denoteremo con 〈x〉 una codifica binaria che rappresenta tale intero.  
Nota. 〈x〉 ∈ *P* **se e solo se** PRIMO ha risposta SI su input *x*.

**Es.** Il linguaggio che rappresenta il problema “CONNESSO” è  
dove *〈G〉* denota una “ragionevole” codifica di *G* mediante una stringa su un alfabeto Σ.  
Dunque, se *G* è connesso allora 〈*G*〉 ∈ *A* e la stringa viene aggettata. Se *G’* non è connesso, allora 〈*G*〉 ∉ *A* e la stringa non viene accettata.  
Per rappresentare un grafo, possiamo prendere Σ = {0, 1, …, 9, (, ), ,} e 〈G〉 = ({1, 2, 3}, {(1, 2), (2, 3), (3, 1)}), cioè una sequenza in cui il primo elemento è l’insieme dei vertici ed il secondo elemento è un insieme di *edge*, che sono tutti gli edge del grafo. Questa è la rappresentazione che useremo.  
Un modo alternativo per rappresentare un grafo è prendere Σ = {0, 1, (, ), #} e codificare *G* mediante una stringa (1#10#11)#((1#10)#(10#11)#(11#1)).

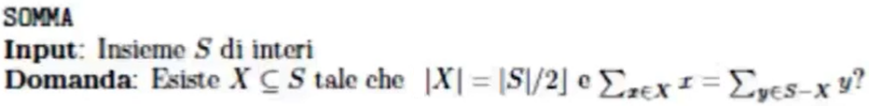
Sia un linguaggio di stringhe che rappresentano grafi connessi (non orientati). Si ha che 〈G〉 ∈ *A* se e solo se G è istanza SI per CONNESSO. Risolvere CONNESSO equivale a decidere il linguaggio *A*.

In questo modo esprimiamo un problema computazionale in termini di riconoscimento di un linguaggio (cioè, l’insieme delle codifiche di istanze SI per il problema).  
**Es.** Si ha in input un grafo G. Un modo per verificare se G è connesso è il seguente:  
1. Seleziona un nodo di G e marcalo;  
2. Ripeti finche si trovano nuovi nodi da marcare:  
 2.1. Per ogni nodo *v* in G, se *v* è connesso ad un nodo marcato, allora marcalo;  
3. Se tutti i nodi risultano marcati allora accetta, altrimenti reject.

Vogliamo convincerci che il risultato precedente è realizzabile mediante una MdT. Quindi, vogliamo una MdT che riconosce l’insieme .  
Il grafo può essere rappresentato mediante due liste: la lista dei *nodi* (numeri naturali) e la lista degli *edges* (coppie di numeri). Ad esempio, abbiamo il grafo 〈G〉 = ({A, B, C, D}, {(A, B), (A, D), (B, C), (C, D)}).  
Nota. Non specifichiamo l’alfabeto (binario, decimale, …). Ignoriamo i dettagli non importanti del’implementazione.

Se facciamo questo, bisogna controllare prima di tutto che l’input sia:  
- una lista di nodi (digit) senza ripetizioni;  
- una lista di coppie di nodi (digit presenti nella lista precedente).

Un’implementazione può essere la seguente:  
*1) Seleziona un nodo di G e marcalo:* Marca il primo nodo sulla lista (ad esempio, con un ∙ sul digit più a sinistra);  
*2) Ripeti finché i nuovi nodi sono marcati:  
 2.1) Per ogni nodo v in G, se v è connesso ad un nodo marcato, allora marcalo:*  
 2.1.1) Sottolinea il primo nodo n1 senza ∙ sulla lista (sottolineando il digit più a inistra);  
 2.1.2) Sottolinea il primo nodo n2 con ∙ sulla lista;  
 2.1.3) Cerca se esiste edge (n1, n2): se SI, allora marca n1 con ∙ e vai a 2); se NO, sia n2 il successivo nodo  
 con ∙ sulla lista, sottolinealo e vai a 2.1.3)  
*3) Se tutti i nodi risultano marcati allora accetta, altrimenti reject*: Scorri la lista dei nodi: se tutti i nodi hanno ∙ allora accetta, altrimenti reject.

**Esercizio.** Dato il problema  
  
  
  
Definire il linguaggio *LSOMMA* corrispondente, spiegando la corrispondenza.  
🡪 Sia   
  
Abbiamo il problema SOMMA, il quale prende in input un insieme S di interi. Vogliamo sapere se esiste un sottoinsieme X di S la cui cardinalità è uguale alla parte intera inferiore di |S|/2 tale che, se si fa la somma su tutti gli elementi *x* di X, questa equivale alla somma su tutti gli elementi *y* di S – X.  
In pratica, ci chiediamo se possiamo dividere l’insieme S in due parti in modo tale che la somma degli elementi di una “metà” è uguale alla somma degli elementi dell’altra “metà”.  
Un modo per definire un tale linguaggio può essere il seguente. Nel linguaggio LSOMMA ci saranno codifiche di insiemi che rispettano la proprietà **(1)**:  
In particolare, è la seconda condizione a stabilire che l’insieme è un’istanza SI del problema. □

**Esercizio.** Dato il problema  
  
  
Definire il linguaggio *LP* corrispondente, spiegando la corrispondenza. Supponendo di usare per gli interi la rappresentazione binaria: (110, 1001) ∈ *LP*? (111, 101) ∈ *LP*? (110, 100) ∈ *LP*?  
🡪 Abbiamo il problema PRIMI, il quale prende in input una coppia di interi (x, y). Vogliamo stabilire se i due interi x ed y sono *relativamente primi* (cioè, non hanno divisori in comune tra loro).  
Un modo per definire un tale linguaggio può essere il seguente. Nel linguaggio LP ci saranno codifiche di coppie di interi che rispettano tale proprietà:  
Supponendo di usare per gli interi la rappresentazione binaria, abbiamo che:  
- (110, 1001) ∉ *LP*, dato che 6 e 9 non sono relativamente primi;  
- (111, 101) ∈ *LP*, dato che 7 e 5 sono relativamente primi;  
- (110, 100) ∉ *LP*, dato che 6 e 4 non sono relativamente primi.  
Ad esempio 〈6, 11〉 ∈ *LP*, dato che la coppia (110, 1011) è costituita da interi relativamente primi. □

**Decidibilità ed indecibilità**

Con la decibilità/indecidibilità iniziamo la seconda parte del corso. L’obiettivo è analizzare i *limiti* della risoluzione di problemi mediante algoritmi: ne studieremo il potere computazionale nella soluzione dei problemi, e proveremo che esistono problemi che possono essere risolti mediante algoritmi ed altri no.

Nell’esempio “PRIMO”, abbiamo espresso un problema computazionale come un problema di *riconoscimento di un linguaggio* (cioè, l’insieme delle codifiche di istanze SI per il problema): dunque, risolvere “PRIMO” equivale a decidere un linguaggio .

Studiamo questa tipologia di problemi per sapere che esistono problemi non risolvibili con un computer. I problemi indecidibili non sono affatto lontani da problemi di interesse informatico; ad esempio:  
- il problema generale della verifica del software non è risolvibile mediante computer;  
- compressione dati ottimale, cioè trovare il programma più corto per produrre un’immagine data;  
- individuazione dei virus, cioè rispondere alla domanda “Questo programma è un virus?”.

Dunque, ci poniamo come **obiettivo** presentare un problema irrisolvibile. Sia data in input una stringa *w* ad una MdT *M*: vogliamo sapere rispondere alla domanda “*M* si arresta su input *w*?”. Il linguaggio corrispondente a questo problema è .

Per ora, introduciamo degli strumenti con cui lavoreremo: *cardinalità di insiemi (infiniti)* e *diagonalizzazione (metodo introdotto da Cantor*).  
La **cardinalità** di un insieme è banalmente la sua taglia. Due insiemi hanno la stessa cardinalità se è possibile stabilire una corrispondenza tra i loro elementi.  
**Es.** A = {1, 2, 3}, B = {4, 3, 5} ⇒ 1 – 4, 2 – 3, 3 – 5.

Cosa possiamo dire per insiemi infiniti? Possiamo applicare una tecnica simile.  
**Es** (*Paradosso di Hilbert*)**.** Nel paese senza confini esiste il più grande di tutti gli alberghi, cioè un albergo con infinite stanze. Tuttavia, anche gli ospiti sono infiniti, e il proprietario ha esposto un cartello con la scritta “Completo”. Ad un tratto, si presenta un viaggiatore che ha assolutamente bisogno di una camera per la notte. Egli non fa questione di prezzo e infine convince l’albergatore, il quale trova il modo di alloggiarlo. *Come fa?*  
Sposta tutti i clienti nella camera successiva (l’ospite della 1 alla 2, l’ospite della 2 alla 3, …); in questo modo, è possibile, essendo l’albergo infinito, sistemare (nella camera 1) il nuovo ospite anche se l’albergo è pieno.  
[Passo completato]  
Poco dopo, arriva una comitiva di *infiniti turisti*, anche in questo caso l’albergatore si lascia covincere (in fondo si tratta di un grosso affare), e trova posto ai *nuovi infiniti ospiti* con la stessa facilità con cui aveva alloggiato l’ospite in più. *Come fa* (senza ripetere infinite volte il passo visto prima)*?*  
Sposta ogni ospite nella stanza con un numero doppio rispetto a quello attuale (dalla 1 alla 2, dalla 2 alla 4, …), lasciando ai nuovi ospiti tutte le camere con i numeri dispari, che sono essi stessi infiniti, risolvendo dunque il problema. Questa operazione, infatti, fa sì che le sole stanze pari siano occupate, il che implica che le stanze dispari siano libere.  
Gli ospiti sono tutti, dunque, sistemati, benché l’albergo fosse pieno.  
[Passo completato]  
Ancora più complesso: ci sono infiniti alberghi con infinite stanze, tutti al completo. Tutti gli alberghi chiudono, tranne uno. Tutti gli ospiti vogliono alloggiare nell’unico albergo rimasto aperto. *Come fa* (senza ripetere infinite volte il passo visto prima)*?*  
Assegna ad ogni persona una coppia di numeri (*n*, *m*) in cui *n* indica l’albergo di provenienza, e *m* la relativa stanza. Gli ospiti sono quindi etichettati come segue:  
  
  
  
  
  
  
A questo punto, basta assegnare le nuove stanze agli ospiti secondo un criterio ordinato, ad esempio per le diagonali, come indicato nella pagina seguente.



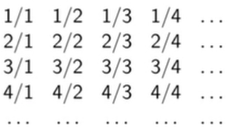
Possiamo, quindi, indicare una corrispondenza tra cardinalità di insiemi infiniti? Ad esempio, l’insieme dei numeri naturali è INFINITO, così come l’insieme dei numeri naturali pari e l’insieme dei numeri naturali dispari. Anche l’insieme dei numeri reali è INFINITO.  
La quantità di numeri reali è la stessa di quella dei numeri naturali? Come si misura la cardinalità di insiemi infiniti?

Il **Metodo della diagonalizzazione** è stato introdotto da Cantor nel 1973 mentre cercava di determinare come stabilire se, dati due insiemi infiniti, uno è più grande dell’altro. Cantor osservò che due insiemi finiti hanno la stessa cardinalità se gli elementi dell’uno possono essere messi in corrispondenza uno a uno con quelli dell’altro. Successivamente, estese questo concetto agli insiemi infiniti.  
**Def.** Una *funzione* è una relazione input-output, dove *X* è l’insieme dei possibili input (*dominio*) e *Y* è l’insieme dei possibili output (*codominio*). Per ogni input *x* ∈ *X* esiste un solo output *y* = f(*x*) ∈ *Y*.  
**Def.** Una funzione è *iniettiva* se ∀*x*, *x*’ ∈ *X*, *x* ≠ *x*’ ⇒ f(*x*) ≠ f(*x*’).  
**Def.** Una funzione è *suriettiva* se ∀*y* ∈ *Y*, *y* = f(*x*) per qualche *x* ∈ *X*.  
**Def.** Una funzione è una funzione *biiettiva* di *X* su *Y* (o una *biezione* tra *X* e *Y*) se f è iniettiva e suriettiva.  
Una funzione biiettiva è una corrispondenza uno a uno tra gli elementi del dominio e gli elementi del codominio.

**Es.** , con , , è una funzione, ma non è né iniettiva né suriettiva.  
**Es.** , con , , è una funzione iniettiva ma non suriettiva.  
**Es.** , con , , è una funzione suriettiva ma non iniettiva.  
**Es.** , con , , è una funzione biiettiva.  
**Es.** , con è una funzione biiettiva.

**Def.** Due insiemi *X* e *Y* hanno *la stessa* *cardinalità* se esiste una funzione biiettiva di *X* su *Y*.

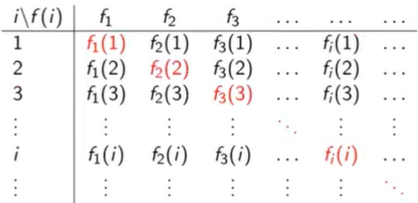
**Def.** Un insieme è *enumerabile* (o *numerabile*) se ha la stessa cardinalità di un sottoinsieme di ℕ.  
Se A è numerabile, allora possiamo “numerare” gli elementi di *A* e scrivere una lista (a1, a2, …); cioè, per ogni numero naturale *i*, allora possiamo specificare l’elemento *i*-esimo della lista.  
**Es.** Per l’insieme dei numeri naturali pari, l’elemento *i*-esimo della lista corrisponde a 2*i*.

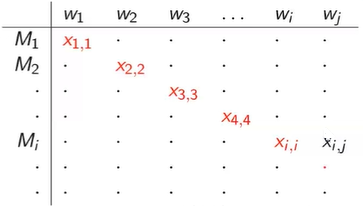
Per quest’ultima proprietà, quindi, possiamo associare ad un insieme *A* (*finito*) un sottoinsieme dell’insieme ℕ (ovviamente, ℕ è *infinito*) attraverso una funzione biettiva . Ciò vuol dire che avremo associato un unico elemento di *A* ad un unico elemento di ℕ.  
**Es.** L’insieme dei numeri razionali è numerabile. Mostriamo che possiamo  
formare una lista di tutti i numeri razionali. Formiamo un rettangolo infinito,  
come mostrato nella figura a destra. Questa tabella, con infinite righe e colonne,  
è corretta, e si noti che se scorre per le righe s’incrementa man mano il  
numeratore, mentre se si scorre per le colonne s’incrementa man mano il  
denominatore.  
Per far vedere che questo insieme è numerabile, bisogna mostrare la biezione (o, equivalentemente, bisogna listare tutti i numeri razionali). Sicuramente non possiamo procedere per righe, dato che se prendiamo tutta la prima linea non arriviamo mai alla seconda; allo stesso modo, non possiamo procedere per colonne. Un’idea può essere procedere *in diagonale* (usando la secondaria, in questo caso). Dunque, prendiamo tutti gli elementi della prima diagonale (1/1) per poi prendere tutti gli elementi della seconda diagonale (2/1, 1/2), e così via.  
Nota. Dovremmo eliminare i duplicati (ad esempio, 1/1 = 1 e 2/2 = 1), ma è  
solo una questione tecnica.  
Costruita la lista (a1, a2, …), possiamo quindi numerarla. Ciò vuol dire che  
esiste la biezione tra l’insieme dei numeri razionali e l’insieme dei numeri reali,  
il che significa che l’insieme dei numeri razionali è numerabile.

L’insieme Σ\* è numerabile: per dimostrarlo, listiamo prima la stringa vuota, poi le stringhe (in ordine lessicografico) lunghe 1, poi 2, e così via. A questo punto, come nell’esempio precedente, possiamo numerare la lista (partendo da 1) e quindi *numerare* l’insieme Σ\*.  
**Es.** Σ = {0, 1} ⇒ *w0* = ε, *w1* = 0, *w1* = 1, *w3* = 00, …  
In questo caso, possiamo sapere anche la posizione in cui compare una determinata stringa nella lista. Presa 001, ad esempio, sappiamo che |001| = 3, e che essa compare in posizione 20 + 21 + 22 + 2 (quest’ultimo in quanto è la *seconda* stringa in ordine lessicografico tra tutte le stringhe di lunghezza 3) = 9; quindi, la stringa 001 si trova in posizione 9 nella lista.

**L’insieme delle descrizioni di MdT** è numerabile: è possibile codificare una MdT *M* con una stringa su un alfabeto Σ.

In generale, *per determinare se un insieme è numerabile bisogna mostrare che esiste una biezione con ℕ*, cioè per ogni elemento possiamo far vedere che posizione occupa all’interno dell’insieme.

**Fatto.** L’insieme dei numeri reali non è numerabile.  
DIM. Sia per assurdo ℝ numerabile; allora possiamo costruire la lista f(1), f(2), f(3), …  
Per ogni *i* ≥ 1, scriviamo f(*i*) = f0(*i*),f1(*i*)f2(*i*)f3(*i*)… Cioè, sappiamo che ogni f(*i*) è un numero reale, ed essendo tale ha una parte intera ed una parte decimale. Nella rappresentazione precedente, f0(*i*) è la parte intera, separata da una virgola dalla parte decimale f1(*i*)f2(*i*)f3(*i*)… Ad esempio, se f(1) = 4,256… allora f0(1) = 4, f1(1) = 2, f2(1) = 5, f3(1) = 6, …  
Organizziamoli in una matrice, posta a lato, in cui le colonne sono  
indicizzate con gli interi 1, 2, 3, …, *i* e la riga *i*-esima è l’elemento  
f(*i*) che compare nella lista. Quindi, ad esempio, nella riga 1  
compaiono le cifre decimali del primo elemento; stiamo ignorando  
la parte intera di ogni numero.  
A questo punto, consideriamo la diagonale di questa matrice. Sia  
*x* ∈ (0, 1) il numero *x* = 0,*x*1*x*2*x*3…*xi*… ottenuto scegliendo *xi* ≠ f*i*(*i*)  
per ogni *i* ≥ 1. Chiaramente, *x* ∈ ℝ.  
A questo punto, risulta *x* nella lista? Se *x* = f(*j*), allora il suo *j*-esimo digit soddisfa *xj* = f*j*(*j*): ma *xj* ≠ f*j*(*j*) per definizione di *x*. Questa è una contraddizione.  
Quindi, *x* ∈ ℝ non può comparire nella lista e ℝ non è numerabile. □

Abbiamo detto che Σ\* = {*w1*, *w2*, …} e l’insieme delle descrizioni di MdT su Σ (cioè {*M1*, *M2*,…}) sono numerabili. Anche in questo caso possiamo usare il Metodo della diagonalizzazione; quindi, costruiamo la tabella seguente, dove xi,j = 1 se *wj* ∈ L(*Mi*), xi,j = 0 altrimenti.  
Possiamo sfruttare la diagonale principale (x1,1, x2,2, ..., xi,j, ...) per  
costruire un nuovo linguaggio . Questo  
linguaggio è il “complemento della diagonale”:  
- se l’elemento (*Mi*, *wi*) della diagonale xi,i = 1, allora *wi* ∉ L;   
- se l’elemento (*Mi*, *wi*) della diagonale xi,i = 0, allora *wi* ∈ L.  
Vogliamo stabilire se L può comparire nella lista, cioè se L = L(*Mh*)  
per qualche *h*; supponiamo che L = L(*Mh*), allora:  
- *wh* ∈ L ⇒ xh,h = 0 ⇒ *wh* ∉ L(*Mh*) = L, che è una contraddizione;   
- *wh* ∉ L ⇒ xh,h = 1 ⇒ *wh* ∈ L(*Mh*) = L, che è una contraddizione.

Da quest’ultimo risultato, otteniamo che L ≠ L(*Mh*) per ogni h, il che significa che L non può comparire nella lista. Da ciò segue che “esistono più linguaggi che Macchine di Turing”.  
**Corollario.** Esistono linguaggi che non sono Turing riconoscibili.

Una MdT **universale** *U* simula la computazione di una qualsiasi MdT *M*.  
Allora *U* riceve in input una *rappresentazione* di *M* e di un possibile input *w* di *M*.  
**N.B.** Abbiamo visto che è possibile codificare una MdT *M* e una stringa *w* con una stringa su un alfabeto Σ.  
**Es.** = “codifica di *M*”#”codifica di *w*”, cioè la concatenazione (#) tra le due codifiche.

Una tale macchina è chiamata “universale” perché la computazione di una qualsiasi MdT può essere simulata da *U*.  
Nota. *U* può anche andare in loop.

Dunque, abbiamo costruito una macchina universale *U* il cui linguaggio è .  
**Teorema.** *Il linguaggio  
è Turing riconoscibile.*  
DIM. Definiamo una MdT *U* che accetta : sull’input , dove *M* è una MdT e *w* è una stringa:  
1. Simula *M* sull’input *w*;  
2. Se *M* accetta, allora accetta; se *M* rifiuta, allora rifiuta. □

Abbiamo visto una MdT che simula un automa. Simulare una MdT *M* con un’altra MdT risulta molto simile:  
1. Marca lo stato iniziale di *M* (stato corrente) e il primo simbolo sul nastro (posizione corrente della testina);  
2. Cerca la prossima transizione (nella parte che descrive la funzione di transizione). Sia (q, *x*) → (q’, *x’*, D);  
3. Esegui la transizione;  
4. Aggiorna lo stato corrente (marca q’) e la posizione corrente della testina (marca simbolo a D);  
5. Se lo stato corrente risulta qaccept/qreject decidi di conseguenza, altrimenti ritorna al passo 2.

**Nota.** *U* è detta MdT universale.  
**Nota.** *U* riconosce *ATM*: accetta ogni coppia 〈*M*, *w*〉 ∈ *ATM*.  
**Nota.** *U* cicla su 〈M, *w*〉 se (e solo se) *M* cicla su *w*. Quindi *U* **non decide** *ATM*.

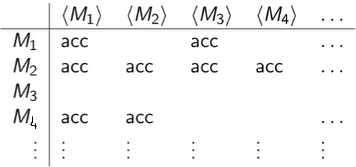
**Teorema** (INDECIDIBILITÀ DEL PROBLEMA DELLA FERMATA)**.** *Il linguaggio  
non è decidibile.*  
Notiamo che un tale linguaggio può essere trasformato in un problema di decidibilità/indecidibilità su input *M*, *w* e che risponde alla domanda “*M* accetta *w*?”. Non è possibile, però, decidere se una data macchina si ferma su un dato input.

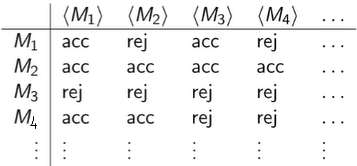
**Es** (Paradosso di Bertrand Russel)**.** In un paese vive un solo barbiere, un uomo ben sbarbato, che rade tutti e soli gli uomini del villaggio che non si radono da soli. *Chi sbarba il barbiere?*  
- Se il barbiere rade sé stesso, allora per definizione il barbiere non rade se stesso;  
- se il barbiere non rade se stesso, allora, dato che il barbiere rade tutti quelli che non si radono da soli, il barbiere rade se stesso.  
Si tratta di un’**antinomia**: compresenza di due affermazioni contraddittorie che possono essere entrambe dimostrate o giustificate.  
In generale, Russel pose il problema dell’insieme di tutti gli insiemi che non contengono se stessi. L’*autoreferenza* può causare problemi.

Possiamo usare un’autoreferenzialità del genere per dimostrare l’indecidibilità di .  
DIM. Supponiamo per assurdo che sia decidibile, cioè che esiste una macchina di Turing *H* con due possibili risultati di computazione (accettazione, rifiuto) e tale che  
Si noti che, a differenza di *U*, la macchina *H* è un decisore (o accetta o rifiuta, ma non va mai in loop).  
Costruiamo una nuova MdT *D* che usa *H* come sottoprogramma *D* sull’input 〈*M*〉, dove *M* è una MdT:  
1. Simula *H* sull’input 〈*M*, 〈*M*〉〉;  
2. Fornisce come output l’opposto di *H*, cioè se *H* accetta, allora *rifiuta* e se *H* rifiuta, allora *accetta*.

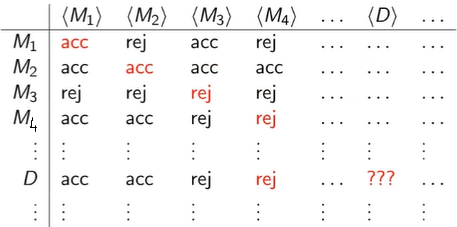
Quindi  
Se ora diamo in input a *D* la sua stessa codifica 〈*D*〉 abbiamo  
Cioè, *D* accetta 〈*D*〉 se e solo se *D* non accetta 〈*D*〉.  
Questo è assurdo. *Tutto è causato dall’assunzione che esiste H*. Quindi, *H* non esiste. □

Riepilogando la dimostrazione:  
1. Definiamo ;  
2. *Assumiamo che sia decibilie; sia H una MdT che lo decide*;  
3. Usiamo *H* per costruire una MdT *D* che inverte le decisioni: *D*(〈*M*〉) accetta se *M* non accetta 〈*M*〉 e rifiuta se *M* accetta 〈*M*〉;  
4. Diamo in input a *D* la sua codifica 〈*D*〉: *D*(〈*D*〉) accetta se e solo se D rifiuta 〈*D*〉. Contraddizione.

Anche se “nascosta”, la dimostrazione precedente utilizza la  
diagonalizzazione. Consideriamo la tabella a lato, dove le *Mt* sono  
tutte le MdT numerate, e 〈*Mt*〉 sono le loro descrizioni (stringhe): se  
nella cella (i, j) c’è “acc”, allora 〈*Mj*〉 ∈ L(*Mi*); se non c’è nulla, allora  
〈*Mj*〉 ∉ L(*M*j).



Consideriamo *H*: la MdT *H* rifiuta anche se *Mi* va in loop (oltre a se *Mi*  
rifiuta). La sua tabella è posta a destra.

Consideriamo, ora, *D* e *D*(〈*D*〉). Dobbiamo considerare la diagonale.  
Dove sono posti i tre punti interrogativi (???) non eravamo in grado  
di mettere né accetta né rifiuta; infatti *D*, in corrispondenza della sua  
descrizione, non può né accettare né rifiutare.

Nella prova precedente, quindi, è stato usato il metodo della  
diagonalizzazione. In conclusione, è Turing riconoscibile ma  
è *indecidibile*.  
Che differenza c’è tra le due dimostrazioni? Cioè, che differenza c’è tra *U* e *D*? Chiaramente, la differenza tra le due macchine è proprio nell’esistenza del loop.

Sappiamo che esistono linguaggi che non sono Turing riconoscibili. Vogliamo individuare uno specifico linguaggio non Turing riconoscibile (, cioè il complemento di ).  
**Def.** Diciamo che un linguaggio *L* è *co-Turing riconoscibile* se il suo complemento è Turing riconoscibile.  
**Teorema.** *Un linguaggio L è decidibile se e solo se L è Turing riconoscibile e co-Turing riconoscibile.*  
DIM. (⇒) Se *L* è decibilie, allora esiste una macchina di Turing *M* con due possibili risultati di una computazione (accettazione, rifiuto) e tale che *M* accetta *w* se e solo se *w* ∈ *L*. Allora *L* è Turing riconoscibile. Inoltre, è facile costruire una MdT che accetta *w* se e solo se *w* ∉ *L*:  
(⇐) Supponiamo che *L* e il suo complemento siano entrambi Turing riconoscibili. Sia *M1* una MdT che riconosce *L* e sia *M2* una MdT che riconosce . Definiamo una MdT *N* a due nastri, la quale si limiterà a simulare in parallelo le macchine *M1* e *M2*.

La macchina N, su input *x*, funzionerà come segue:  
1. Copia *x* sui nastri di *M1* e *M2*. Quindi, un nastro lo associamo alla macchina *M1* ed uno alla macchina *M2*¸dopodiché ogni mossa sarà data dalla coppia di mosse che farebbero *M1* e *M2*;  
2. Simula *M1* e *M2* in parallelo (usa un nastro per *M1*, l’altro per *M2*);  
3. Se *M1* accetta, allora *N* accetta. Se *M2* accetta, allora *N* rifiuta.  
Segue che *N* decide *L*. Infatti, per ogni stringa *x* abbiamo due casi:  
1. *x* ∈ *L*. Ma *x* ∈ *L* se e solo se *M1* si arresta e accetta *x*. Quindi *N* accetta *x*;  
2. *x* ∉ *L*. Ma *x* ∉ *L* se e solo se *M2* si arresta e accetta *x*. Quindi *N* rifiuta *x*.  
Poiché una e solo una delle due MdT accetta *x*, allora *N* è una MdT con soli due possibili risultati di una computazione (accettazione, rifiuto) e tale che *N* accetta *x* se e solo se *x* ∈ *L*. □

**Teorema.**  *non è Turing riconoscibile.*  
DIM. Supponiamo per assurdo che sia Turing riconoscibile. Sappiamo che è Turing riconoscibile. Quindi, è Turing riconoscibile e co-Turing riconoscibile. Per il precedente teorema, è decidibile. Questo è assurdo, in quanto abbiamo dimostrato che non è decidibile. □

È importante riconoscere che un problema *P* è indecidibile. Si può estendere la classe dei problemi indecidibili in due modi:  
1) Supporre l’esistenza di una MdT che decide *P* e provare che questo conduce ad una contraddizione (come fatto con );  
**2)** Considerare un problema *P’*  di cui sia nota l’indecidibilità (ad esempio, ) e dimostrare che *P’*  *“non è più difficile”* del problema in questione *P*.

**Es.** Sia Σ = {0, 1}. Consideriamo i problemi:  
Sia *w* ∈ *L* e sia *n* il corrispondente decimale di *w*. È facile costruire la MdT *INCR* che incrementa il valore di *n*:

Possiamo dire che *EVEN* “non è più difficile” di *ODD*: se esiste una MdT *R* che decide *ODD*, allora la MdT *S* decide *EVEN*.

Viceversa, se *EVEN* è indecidibile proviamo così che anche *ODD* lo è: se per assurdo esistesse una MdT *R* che decide *ODD*, allora la MdT *S* deciderebbe *EVEN*. □

**Esercizio.** Possiamo dire che *ODD* “non è più difficile” di *EVEN*? In che modo?  
🡪

**Es** (Il “vero” problema della fermata)**.**  
Abbiamo . L’unico caso che fa sì che non sia un’istanza SÍ è il caso in cui la macchina va in loop su *w*.  
Vogliamo far vedere che questo *è anch’esso indecidibile*, utilizzando una *riduzione* da un problema indecidibile. Se *HALTTM* fosse decidibile, allora potremmo decidere anche *ATM* (cioè, se avessimo un decisore per *HALTTM* allora potremmo costruire un decisore per *ATM*, il che ci porterà ad una contraddizione per l’indecidibilità di *ATM*).  
Sia *R* una MdT che decide *HALTTM* (per assurdo). Costruiamo quindi *S* (che può decidere *ATM*) che sull’input , dove *M* è una MdT e *w* è una stringa, *simula R su* :  
- *se R rifiuta*, allora *S* rifiuta (poiché *M* va in loop quando *w* ∉ *L(M)*);  
- *se R accetta* (cioè, *M* si ferma su *w*), allora simula *M* finché *M* si arresta su *w*.

Se *M* ha accettato, allora S accetta l’input (*w* ∈ *L(M)*); se *M* ha rifiutato, allora S rifiuta l’input (*w* ∉ *L(M)*). In definitiva, *S* accetta se e solo se .  
Se esistesse *R* che decide *HALTTM*, allora otterremmo *S* che decide *ATM*. Poiché sappiamo che *ATM* è indecidibile, allora *R* non può esistere e *HALTTM* deve essere indecidibile (contraddizione).

In generale, lo **schema di riduzione** dal problema A al problema B si utilizza come segue:  
1. Sappiamo che il problema noto A risulta indecidibile;  
2. Vogliamo provare che B è indecidibile;  
3. Assumiamo (per assurdo) B decidibile ed usiamo questa assunzione per provare A decidibile;  
4. La contraddizione ci fa concludere che B è indecidibile.

Di seguito un riepilogo per l’esempio della prova dell’indecidibilità di :  
*1. Sappiamo che il problema noto A risulta indecidibile*. Abbiamo posto A = ; *2. Vogliamo provare che B è indecidibile*. gioca il ruolo di B; *3. Assumiamo (per assurdo) B decidibile ed usiamo questa assunzione per provare A decidibile.* Proviamo che se è decidibile allora è decidibile; *4. La contraddizione ci fa concludere che B è indecidibile.* Giungiamo alla contraddizione.

**Def.** Una funzione è *calcolabile* se esiste una MdT *M* tale che, su ogni input *w*, *M* si arresta con f(*w*) (e solo con f(*w*)) sul suo nastro.  
Cioè, una funzione è calcolabile se esiste una Macchina di Turing che la calcola.

**Es.** Le seguenti funzioni aritmetiche sono calcolabili (dove *n*, *m* ∈ ℕ):  
- ;   
- ;   
- ;   
- ;   
- .

**Def.** Un linguaggio *A* è *riducibile a un linguaggio B* () se esiste una funzione calcolabile tale che  
La funzione *f* è chiamata la *riduzione di A a B*.

Una riduzione fornisce un modo per convertire problemi di appartenenza ad *A* in problemi di appartenenza a *B*. Se un problema *A* è riducibile a *B*, e sappiamo risolvere *B*, allora sappiamo risolvere *A*: ciò implica che *A* “non è più difficile” di *B*.

**Teorema.** *Se e B è decidibile, allora A è decidibile.*  
DIM. Siano *M* il decider per *B* ed *f* la riduzione da *A* a *B*. Costruiamo un decider *N* per *A* che, su input *w*:  
- calcola f(*w*);  
- “utilizza” *M* su f(*w*) e dà lo stesso output.  
A questo punto: (perché *f* è una riduzione da *A* a *B*) . Quindi, *N* decide *A*.  
Dunque: .  
D’altronde: . □

**Teorema.** *Se e B è Turing riconoscibile, allora A è Turing riconoscibile.*  
DIM. Siano *RA* un riconoscitore per *A* e *RB* un riconoscitore per *B*; allora:

**Corollario.** Se e *A* è indecidibile, allora *B* è indecidibile.  
DIM. Se *B* fosse decidibile lo sarebbe anche *A*, in virtù del teorema precedente. □

**Corollario.** Se e *A* non è Turing riconoscibile, allora *B* non è Turing riconoscibile.  
DIM. Se *B* fosse Turing riconoscibile lo sarebbe anche *A*, in virtù del teorema precedente. □

**Es.** Siano definiti i seguenti linguaggi:  
Un esempio di riduzione è . Il linguaggio prende in input una MdT *M* e si pone come domanda “L(*M*) = ∅?”. Questa riduzione ci dirà che decidere se il linguaggio di una MdT è vuoto è un problema indecidibile. Dobbiamo far vedere che esiste la funzione di riduzione, ossia una funzione che mappa stringhe del primo linguaggio in stringhe del secondo linguaggio e, per ogni stringa che non appartiene al primo linguaggio, il risultato sarà una stringa che non appartiene al secondo linguaggio.  
Consideriamo tale che , dove *M1* su input *x*:  
1. Se *x* ≠ *w*, allora *M1* si ferma e rifiuta *x*;  
2. Se *x* = *w*, allora *M1* simula *M* su *w* e accetta *x* se *M* accetta *w*.  
*f* è una riduzione di a ?  
Abbiamo che:  
In questo punto, si ha che la funzione definita mappa una stringa del primo linguaggio () in una stringa che appartiene al secondo linguaggio; in definitiva, abbiamo mostrato che   
Ci resta da mostrare il caso in cui la stringa non appartiene ad :

In sintesi, la funzione *f* è calcolabile, e *M* accetta *w* (cioè, ) se e solo se (cioè, se e solo se ). □

Per completezza, in base ad uno dei corollari definiti in precedenza, abbiamo che  
Quindi, anche *ETM* è indecidibile (*la decidibilità non risente dalla complementazione*).  
Nota. Non si conosce una riduzione da *ATM* a *ETM*.

**Es.** Siano definiti i linguaggi:  
Un esempio di riduzione è . Il linguaggio contiene tutte le descrizioni di MdT il cui linguaggio è regolare (cioè, si ricordi, se il linguaggio è accettato da qualche automa finito). Questo linguaggio prende in input una MdT *M* e si pone come domanda “*L*(*M*) è regolare?”. Questa riduzione ci dirà che decidere se il linguaggio di una MdT è regolare è un problema indecidibile.  
Non abbiamo alcuna restrizione su come costruire la funzione *f*. L’unica cosa importante è che questa funzione sia calcolabile.  
Consideriamo la funzione come riduzione da a , dove *R* su un input *x*:  
1. Se , allora *R* si ferma e accetta *x*;  
2. Se , allora *R* simula *M* su *w* e accetta *x* se *M* accetta *w*.  
Posto per brevità, abbiamo che:  
posto Σ = {0, 1}. Sapendo che Σ\* (= *L*(*R*)) è regolare, allora .  
Ci resta da mostrare l’implicazione inversa:  
il quale non è regolare (mostrato in passato con il Pumping Lemma).

Questo fatto implica che . A questo punto, abbiamo dimostrato le due implicazioni.

In sintesi, la funzione *f* è calcolabile, e:  
- *L*(*R*) = Σ\* (regolare) se *M* accetta *w*;  
- *L*(*R*) = {0n1n | n ∈ ℕ} (non regolare) altrimenti. □

**Es.** Siano definiti i linguaggi:  
Un esempio di riduzione è . Il linguaggio prende in input due MdT *M1* e *M2* e si pone come domanda “*L*(*M1*) = *L*(*M2*)?”. Questa riduzione ci dirà che decidere se il linguaggio di due MdT sono uguali è un problema indecidibile.  
Consideriamo la funzione come riduzione da a , dove se e solo se ; occorre dimostrare che la funzione è calcolabile, e che vale il sse precedente.  
Sia *M1* una MdT tale che *L*(*M1*) = ∅. *La funzione sarà* , la quale è calcolabile, e bisogna far vedere che effettivamente questa è una riduzione di a .  
Mostriamo le implicazioni:

**Es.** Siano definiti i linguaggi:  
Un esempio di riduzione è . Procediamo come nell’esempio precedente.  
Consideriamo la funzione come riduzione da a , dove se e solo se ; occorre dimostrare che la funzione è calcolabile, e che vale il sse precedente.  
L’idea è la seguente. Data , consideriamo le MdT *M1* e *M2* tali che:  
- *M1* accetta *x* (qualunque input) ;  
- *M2* simula *M* su *w*. Se *M* accetta *w*, allora *M2* accetta *x* .  
La funzione è calcolabile, in quanto costruire una macchina che accetta tutte le stringhe è semplice, così come la seconda parte di *f* è *M* stessa, quindi è l’input (basta concatenare la descrizione di *M1* a *M* stessa, ed otteniamo il risultato della funzione). Ora, mostriamo che la funzione *f* è una riduzione.  
Mostriamo le implicazioni:  
In sintesi, è una riduzione da a . □

**Esercizio.** Siano definiti i linguaggi:  
Un esempio di riduzione è .  
🡪 L’idea per la dimostrazione è la seguente. Data , consideriamo una MdT *M1* che *accetta l’insieme vuoto* e una macchina *M2* che accetta Σ\* se *M* accetta *w*. Cioè, per ogni input *x*:  
- *M1* rifiuta *x*;  
- *M2* simula *M* su *w*. Se *M* accetta, allora *M2* accetta.  
Dimostriamo che la funzione è una riduzione da a .

…

**Teorema.** *non è né Turing riconoscibile né co-Turing riconoscibile.*  
DIM. Supponiamo per assurdo che sia Turing riconoscibile. Allora  
Quindi sarebbe Turing riconoscibile: questo è assurdo.  
Supponiamo per assurdo che sia co-Turing riconoscibile, cioè che sia Turing riconoscibile. Allora  
Quindi sarebbe Turing riconoscibile: questo è assurdo. □

**Esercizio.** Mostrare che