Riassunto Calcolabilità e complessità

# Introduzione alle macchine di Turing

La macchina di Turing è stata inventata dal suo omonimo creatore nel 1936 ed ha le seguenti caratteristiche:

* è molto semplice da utilizzare dal momento che hanno un solo tipo di istruzione, ciò permette di effettuare l’analisi dei programmi, dei costi, eccetera;
* utilizzano un modello di memoria sequenziale.

Per macchina di Turing si intende un dispositivo che esegue un dato numero di operazioni sulla sola base dell’input, essa possiede queste caratteristiche:

* Un nastro semi-infinito (ha un inizio ma non una fine);
* Una testina che legge una cella alla volta;
* Sulla cella ci può essere un simbolo;
* I simboli sono presi da un alfabeto di lavoro;
* Alcune celle sono vuote e possono essere indicate con B o lo spazio;



La macchina ha uno stato e opera su una funzione di transizione che associa una nuova posizione e scrive un carattere sulla posizione in cui si trova in quel momento.

Esempio: δ(q0.a)= (q’,a’,m) dove m={L,R,S}

La testina si trova nello stato qo con il carattere a sotto di essa, allora farà qualcosa che la sposterà nello stato q’ scrivendo il carattere a’ prima di muoversi in base a come gli dice m

(rispettivamente sinistra, destra, fermo).

Come regola generale, per ogni coppia (q,s) dove q è lo stato attuale della macchina e s il carattere della cella corrente, esiste una e una sola funzione di transizione associata.

Per convenzione ogni macchina parte da una configurazione iniziale che è:

Spiegato: si trova nello stato q0 con il carattere vuoto sotto la testina, allora va nello stato q1 spostando la testina a destra (R) e scrivendo il carattere vuoto prima del movimento.

La computazione di una macchina di Turing termina quando non esiste una funzione di transizione associata alla coppia (q,s).

La terminazione NON dipende dal movimento della testina dato che la macchina di Turing può continuare ad andare anche se la testina sta continuamente ferma.

## Realizzazione di una macchina di Turing

Come già detto in precedenza, la macchina di Turing ha memoria sequenziale, tuttavia il programma non lo è dal momento che l’ordine in cui vengono scritte le funzioni di transizione non ha importanza e tutto dipende dalla configurazione istantanea.

La creazione di cicli nelle macchine di Turing avviene tramite l’utilizzo degli stati, precisamente si rimane in uno stesso stato e di cicla spostando la testina finchè ce n’è bisogno.

A volte è necessario utilizzare uno stato per ogni situazione, ad esempio:

Si vuole leggere la stringa “ciao” nella macchina di Turing, se viene letta, si cancella e si scrive “salve” al suo posto.

In questo esempio si ha bisogno di quattro stati per leggere “ciao” (uno per ogni lettera) e ne servono altri 5 per scrivere “salve” (anche qui uno per ogni lettera), si ha quindi una situazione in cui lo stato è utilizzato come memoria.

### Considerazioni:

* Nel caso in cui si ha bisogno di memoria finita, si utilizzano gli stati;
* Se invece si ha bisogno di memoria infinita, è opportuno tornare a un dato punto del nastro utilizzando dei caratteri segnaposto oppure implementando un “contatore” con una stringa di caratteri da cancellare a ogni iterazione.

# Funzioni

Una funzione f:A→ B è un’associazione di elementi di A verso quelli di B tale che ogni elemento del primo insieme sia associato a un solo elemento del secondo, ciò lascia aperte tre possibilità:

* Dato a in A, potrebbero non esserci elementi di B associati, quindi f non è definita per a e si indica con f(a)↑, la funzione in questo caso viene detta non totale o parziale;
* Vi possono essere più elementi di A distinti con lo stesso elemento di B associato, la funzione in tal caso è detta non iniettiva;
* Potrebbe esistere un elemento di B che non è associato a nessun elemento di A, la funzione in tal caso è detta non suriettiva.

Vi sono comunque casi in cui una funzione totale può anche essere parziale, in tal caso una funzione parziale può essere anche totale quando verifica le condizioni di totalità (ogni elemento di A è associato a uno di B).

## Algoritmi e programmi

Un algoritmo è la descrizione di un procedimento di calcolo.

Esempio: La funzione fp:N→ N associa il numero di cellule all’i-esimo anno di vita di p.

Questa definizione è chiara ma non indica come viene calcolato il tutto.

Un programma è una resa di un algoritmo in un linguaggio di programmazione, se ne avessimo uno, si potrebbe calcolare il numero di cellule dato un numero naturale n tenendo in considerazione il tempo impiegato, lo spazio occupato e altri eventuali costi.

In questo contesto le funzioni non contano, quello che interessa è la proprietà delle funzioni di essere calcolabili, quindi se esiste un algoritmo che riesce a farlo, eccetera.

Questo ragionamento porta ai concetti di funzioni definibili e calcolabili, in particolare si vuole sapere se questi concetti sono equivalenti o meno, oppure se esistono funzioni definibili che sono anche calcolabili.

# Storia della svolta dell’informatica

Il concetto di funzioni definibili e calcolabili è legato alla storia dell’informatica e dei computer.

La spinta avvenne ai primi del 1900 nella balistica per il calcolo delle traiettorie dei proiettili, i “computer” in quest’epoca erano esseri umani adibiti al calcolo in base a delle istruzioni date, queste istruzioni dovevano seguire i seguenti punti:

* dovevano essere di numero finito;
* dovevano essere completate in tempo finito;
* non dovevano necessitare di particolari competenze;
* non doveva esserci originalità nello svolgimento, tutti dovevano eseguirle allo stesso modo;
* dovevano essere ripetibili.

Queste istruzioni venivano dette procedure di calcolo effettive dal momento che queste portano a un risultato, queste non erano comunque esenti da errori e per questo motivo si davano a più “computer” le stesse istruzioni al fine di confrontare i risultati e quindi correggere il tutto.

Tutto ciò divenne insufficiente quando partì la ricerca per la bomba atomica, quindi iniziarono a essere costruiti i primi sistemi che automatizzano il tutto, per fare ciò servivano delle istruzioni semplici ma allo stesso tempo complete da permettere automaticità e la scrittura di algoritmi per ogni possibile funzione.

Nel 1936 Alan Turing propose la macchina che porta il suo nome, essa è semplice per permettere l’analisi chiara e meno contorta, proprio per questo motivo Turing ebbe l’intenzione di sostituire la persona a favore del sistema automatico.

# Definizione formale di macchina di Turing

Una macchina di Turing è una quintupla (Q,∑,Г,δ,q0) in cui:

* Q è l’insieme di stato;
* ∑ è l’alfabeto di input;
* Г è l’alfabeto di lavoro (che contiene tutto ∑ più il carattere Б;
* δ è la funzione di transizione, essa è parziale dal momento che non è definita per tutti gli stati;

δ:Q x Г → Q x Г x Δ dove Δ={L, R, S}

* q0 è lo stato iniziale e appartiene all’insieme Q.
* In alcuni casi è anche presente un insieme F indicante gli stati finali, essi sono utili per la decisione di un linguaggio.

In ogni istante la macchina di Turing si trova in un certo stato e la testina osserva una certa cella, si può quindi definire il concetto di configurazione istantanea come la descrizione di tutte le componenti della macchina in un dato istante, precisamente lo stato in cui si trova, la cella osservata e la stringa sul nastro.

Esempio:

C = q0|=>БabacБaaБББ…

L’esempio descrive una configurazione istantanea in cui la macchina si trova allo stato q0, osserva la prima cella e contiene la stringa “abac aa” nel nastro.

Si può notare che dopo l’input le celle hanno tutte valore Б, quindi per rappresentare le configurazioni basta rappresentare solo l'input inserito.

Data una configurazione C e una funzione di transizione δ(q,a), se δ(q,a) passa a (q’,a’,R), allora viene creata una nuova configurazione C’ in cui:

* q’ è il nuovo stato;
* a’ è il carattere da scrivere al posto di a;
* la testina viene spostata di una cella verso destra.

Si può quindi dire che C’ è raggiungibile da C in un passo.

Quindi in generale:

## Situazione particolare

Data una configurazione C = q0|=>БabacБaaБББ… e una funzione di transizione δ(q,a)=>(q’,a’,L), si ha una situazione in cui la testina cade dal nastro, questo è un errore e quindi si specifica che:

C=qaα dove q è uno stato in Q, a è un carattere dell’alfabeto di lavoro Г e α è una stringa di caratteri:

q in Q, a in Г, α in Г\*

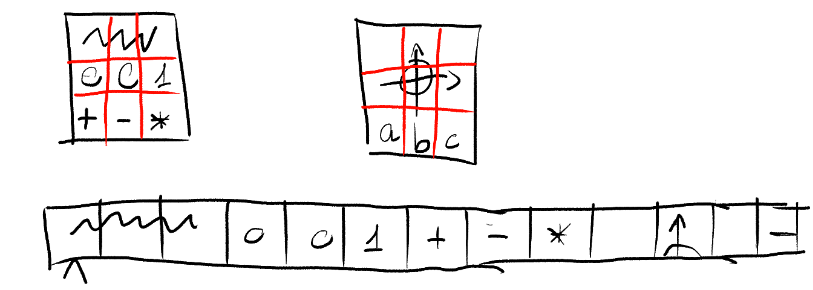
Una computazione è una sequenza finita di configurazioni tale che la configurazione Ci permetta il passaggio alla sua successiva in una sola transizione:

Il numero finito di configurazione è il motivo per cui la funzione di transizione è parziale (non totale), se così non fosse le computazioni sarebbero infinite.

## Discussioni sulle macchine di Turing

Il modello della macchina di Turing è ragionevole nonostante il nastro sia semi-infinito, ciò può essere un problema per via della limitatezza delle risorse nei normali computer. Questo comunque non è un problema dal momento che Q, Г e ∑ sono per definizione insiemi finiti, inoltre le configurazioni che interessano riguardano solamente quelle finite. Di conseguenza la macchina di Turing è utilizzata come oggetto finito nel senso che ci sarà un numero di configurazioni finito in cui, dopo un certo numero di passi finiti, ci sarà solo un numero finito di celle scritte. Ciò apre all’ipotesi di avere un input di lunghezza infinita ma non ha senso dal momento che non si potrebbe leggere.

Il significato del nastro semi-infinito è dato dal fatto che non non si ragiona sulla limitatezza dei dati e quindi non ci si pone il problema che il nastro finisca, per questo motivo è più giusto dire che il nastro sia illimitato.

L’idea del nastro parte dall’analisi che gli esseri umani hanno a disposizione carta e penna in cui viene compilato ogni foglio e che, quando questo non ha più spazio per la scrittura, venga cambiato, allo stesso tempo però è anche possibile tornare indietro per correggere eventuali errori.

Se si dividono questi figli in righe e colonne, si può notare che ogni riquadro corrisponde a una cella del nastro, la testina invece è il modo in cui l’essere umano legge il foglio.

Gli alfabeti Г e ∑ sono finiti perché, dal momento che ogni cella è uno spazio finito, se così non fosse vi sarebbero simboli simili a livello infinitesimale e quindi sarebbe impossibile distinguerli. Q invece è finito dal momento che rappresenta lo stato mentale dell’essere umano, anche qui si può fare un ragionamento simile al precedente: se gli stati mentali fossero infiniti, questi sarebbero tra loro indistinguibili.

La configurazione inoltre è finita dato che l’essere umano è capace di leggere poche parole alla volta, “dimenticandosi” quelle precedenti, ciò spiega anche il fatto che il numero di simboli non sia infinito.

# Accettazione e decisione di un linguaggio

Dalla definizione della macchina di Turing detta in precedenza, è possibile dire che:

* un macchina accetta un linguaggio se essa termina per ogni input, ciò vuol dire che se un dato input non appartiene a questo linguaggio, esso non farà terminare la computazione;
* una macchina decide un linguaggio se essa termina su uno stato contrassegnato come finale per ogni input, quindi tutti gli input che fanno terminare la macchina in stati non finale non appartengono al linguaggio.

# Modelli di calcolo

Un modello di calcolo indica tutto il necessario che dà un senso al calcolabile, nel caso delle macchine di Turing, ad esempio, bisogna definire sia il formalismo, sia le componenti hardware che la compongono.

Quello che essenzialmente si deve fare per definire un modello di calcolo è indicare come è fatta e come accetta i linguaggi, l’accettazione infatti permette l’analisi del modello in modo semplice.

Esempio: data una macchina di Turing standard e una in cui la testina non può stare ferma, è possibile dire che queste accettano lo stesso insieme di linguaggi (?

Per dimostrare ciò, si divide la dimostrazione in due parti:

* Questa parte è banale dal momento che la macchina di Turing è equivalente a quella standard ma con un vincolo in più, le funzioni di transizione hanno corrispondenza 1 a 1 tra le due macchine;
* Per dimostrare ciò, data una macchina M standard, bisogna costruire M’ in modo che i due linguaggi accettati siano equivalenti (Llr(M’) = Ls(M)).

M=(Q,∑,Г,δ,q0)

M’=(Q’,∑,Г’,δ’,q0)

Per quanto riguarda le funzioni di transizioni, se queste portano a uno spostamento a destra/sinistra della testina, esse vengono mantenute:

δ’(q,a) = δ(q,a) per ogni (q,a) con q in Q e a in ∑

Se invece le funzioni di transizione non portano a nessuno spostamento, bisogna sostituirle con due spostamenti in modo da simulare il funzionamento corretto.

Questo spostamento deve essere prima a destra e poi a sinistra, non si può fare al contrario dal momento che, se così non fosse, la testina rischierebbe di cadere dal nastro.

Per fare ciò nel modo corretto, si effettua il primo spostamento andando verso uno stato di supporto, il secondo spostamento invece porterà il tutto verso lo stato corretto:

se δ(q,a)=(q’,a’,S) → δ’(q,a)=(ql,a’,R)

δ’(ql,ai)=(q’.ai,L) per ogni ai in Г

Si può notare che per il secondo spostamento vengono definite tante funzioni di transizione quanti sono i simboli dell’alfabeto di lavoro, questo perché quando si torna indietro non si sa quale carattere viene visto dalla testina.

* Dalla precedente regola, si può dire che Q’ non è altro che l’insieme di stati Q originale più tutti gli stati di supporto;
* L’alfabeto di lavoro è rimasto lo stesso dato che non è stato aggiunto alcun simbolo..

Un ragionamento simile avviene tra i linguaggi accettati e decisi: sono equivalenti?

## Dimostrazione

### Lt ⊇ Lf

Data una macchina M che decide un linguaggio L, costruire una macchina M’ che accetti lo stesso linguaggio. Per fare ciò, bisogna fare in modo che la macchina non termini per tutti gli input non appartenenti a L, quindi:

* se la funzione di transizione è definita per (q,a) in M, allora essa rimane tale anche per M’;

δ’(q,a)=δ(q,a)

* se invece la funzione non è definita e q non è finale, si crea una nuova funzione in cui, per ogni stato q, si rimane sullo stesso stato senza muovere la testina;

δ’(q,a)=(q,a,S) per ogni q in Q tale che q non finale

### Lf ⊇ Lt

Data una macchina M che accetta un linguaggio L, costruire una macchina M’ che decida lo stesso linguaggio.

Dal momento che un dato input x appartiene al linguaggio L se e solo se la macchina M termina oppure se termina in uno stato finale, una possibile soluzione è definire M’ in modo che tutti gli stati in Q siano finali:

M’=(Q,∑,Г,δ,q0,Q)

Se invece si vuole avere solamente uno stato finale, occorre ragionare nel seguente modo:

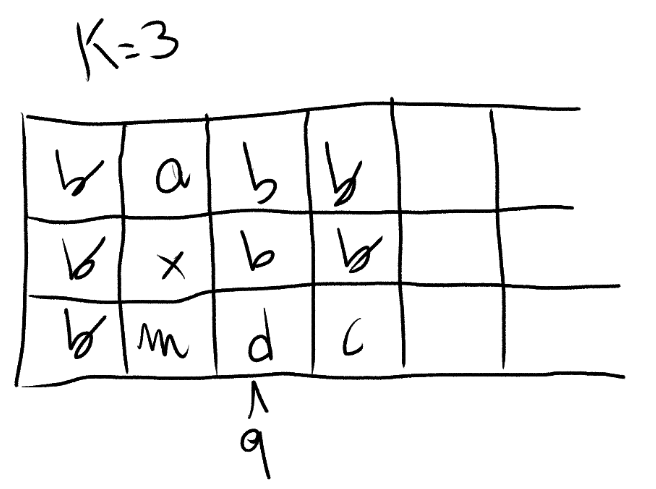
* se la funzione di transizione è definita in M per (q,a), allora essa rimane tale anche per M’.

δ’(q,a)=δ(q,a)

* se invece non è così, allora si aggiunge una funzione che da (q,a) passa a uno stato finale q lasciando la testina ferma.

δ’(q,a)=(qf,a,S)

# Macchina di Turing multitraccia

Questo modello di calcolo è una macchina di Turing simile a quella standard ma presenta un nastro avente k tracce per ogni cella.

Dal momento che la macchina è multitraccia, la funzione di transizione è differente:

k>0

La funzione di transizione prende quindi uno stato e k caratteri come input e va verso un altro stato, scrive k caratteri sul nastro e muove la testina.

La configurazione iniziale prevede tutte le tracce vuote tranne la prima in cui è contenuto l’input.

Questo modello non aggiunge potenza in più alla macchina di Turing, di conseguenza accettano lo stesso insieme di linguaggi.

## Dimostrazione

### Lkt ⊇ Ls

Data una macchina M standard, costruire M’ con k tracce che accetta lo stesso linguaggio.

Si può vedere la macchina di Turing standard come un caso particolare di quella a k tracce (in cui k=1), quindi:

se (q,a)=(q’,a’,m)

allora ’(q,a,Б,...,Б) = (q’,a’,Б,...,Б,m)

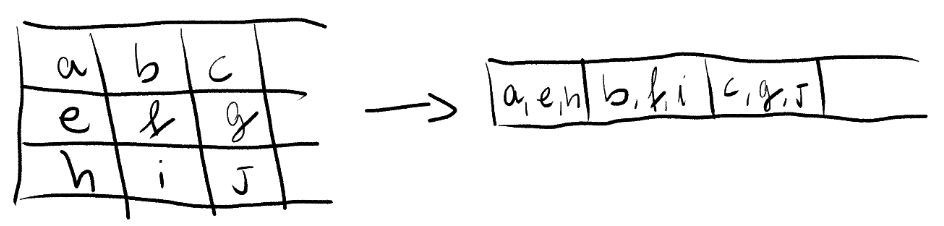
### Ls ⊇ Lkt

Data una macchina M a k tracce, costruire M’ standard in modo che accetti lo stesso linguaggio.

Una possibile idea sta nel ridefinire l’alfabeto di lavoro nel seguente modo:

’= x x … x

’ sarà quindi una k-upla di valori appartenenti all’alfabeto di lavoro della macchina M.

Un’altra idea è dare una corrispondenza univoca ai simboli dell’alfabeto di lavoro nel seguente modo:

per ogni x in ∑, x=(x,Б,...,Б)

per tutti gli altri simboli che non rispettano questa forma, si utilizza un nuovo simbolo.

# Macchina di Turing con nastro infinito

Questo modello è simile alla macchina standard con la differenza che il nastro è illimitato da entrambe le parti, di conseguenza la testina non può cadere dal nastro (la nozione di raggiungibilità cambia). La configurazione iniziale non cambia ed è possibile dimostrare che la potenza è equivalente rispetto alla macchina standard.

## Dimostrazione

### L2w ⊇ Ls

Data una macchina M standard, costruire M’ con nastro infinito in modo che accetti lo stesso linguaggio:

M=(Q,∑,Г,δ,q0)

M’=(Q’,∑,Г’,δ’,q0’)

Per fare in modo che la macchina M’ simuli la caduta dal nastro, si utilizza un nuovo simbolo dell’alfabeto di lavoro da scrivere nella cella antecedente a quella iniziale, quindi:

Г’=Г U {\*}

Per quanto riguarda la funzione di transizione, bisogna fare in modo che il carattere \* venga scritto nella cella giusta:

δ’(q0’,Б)=(q\*,Б,L)

δ’(q\*,\*)=(q0,\*,R)

per ogni a in Г tale che a!=\*, δ’(q,a)=δ(q,a)

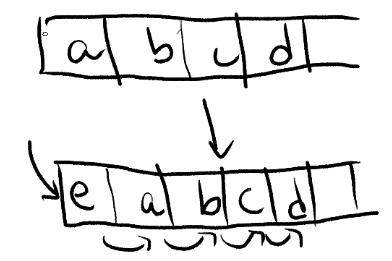
δ’(q,\*)=(q,\*,S)

Come si può notare dalle funzioni di transizione scritte sopra, lo stato iniziale q0 della macchina M non è equivalente a quello di M’, di conseguenza Q e Q’ non sono equivalenti:

q0 != q0’ e quindi Q’=Q U {q0’,q\*}

### Ls ⊇ L2w

Data una macchina M con nastro infinito, costruire M’ standard in modo che accetti lo stesso linguaggio:

M=(Q,∑,Г,δ,q0)

M’=(Q’,∑,Г’,δ’,q0’)

Una possibile idea è quella di definire delle funzioni di transizione che permettano lo shift dei simboli da sinistra a destra, quindi:

δ’=δ U {funzioni per lo shift}

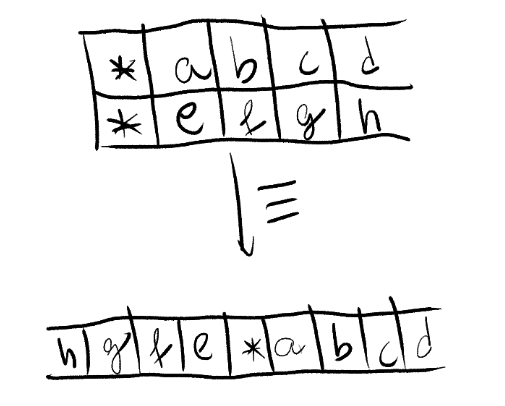
Q’=Q U {stati utili per lo shift}

La macchina M’ NON deve terminare durante uno shift, ciò non succede dal momento che è definita per ogni stato e per ogni simbolo dell’alfabeto di lavoro.

Un’altra idea è quella di utilizzare un nastro multitraccia, in tal caso si ha bisogna di un simbolo \* per indicare, in base allo stato attuale, la traccia di destinazione.

Г’=,Г U {\*}

Oltre agli stati in Q, vengono definiti degli stati qs e qi per indicare rispettivamente che si sta considerando la traccia superiore o inferiore.

Q’=Q U {qs,qi}

Il cambio di lettura della traccia avviene utilizzando le seguenti funzioni di transizione:

δ’(qs,\*)=(qi,\*,R)

δ’(qs,\*)=(qi,\*,R)

δ’(q,a)=δ(q,a)

In questo modo, la macchina M’ non può terminare durante “lo scambio”.

Bisogna inoltre definire delle funzioni di transizione che, a partire da qs/qi e un simbolo a, si vada in uno degli stati di Q, allo stesso modo non può terminare su \* dato che c’è una δ definita per ogni stato.

# Macchina di Turing multinastro

La macchina di Turing multinastro è una macchina avente k nastri con testine indipendenti, la differenza principale rispetto al modello standard riguarda quindi la funzione di transizione:

δ:Q x Г x Г x … x Г → Q x (Г x Δ) x (Г x Δ) x … x (Г x Δ)

Bisogna quindi specificare nella funzione di transizione il movimento di ogni testina.

Nella configurazione iniziale, tutti i nastri sono vuoti e hanno la testina all’inizio tranne il primo in cui è scritta la stringa di input.

Anche in questo caso, l’insieme dei linguaggi accettati da questo modello è lo stesso rispetto a quello standard.

## Dimostrazione

### Lkn ⊇ Ls

Data una macchina M standard, costruire una macchina M’ a k nastri che accetta lo stesso linguaggio. Questa parte della dimostrazione è semplice dal momento che M è un caso particolare di M’ in cui viene utilizzato un solo nastro, la funzione di transizione diventa quindi la seguente:

Dato δ(q,a)=(q’,a’,m) δ’(q,a,Б,...,Б)=(q’,a,m,Б,S,...,Б,S);

### Ls ⊇ Lkn

Data una macchina di Turing M a k nastri, costruire una macchina M’ standard che accetti lo stesso linguaggio.

Per dimostrare ciò si costruisce una macchina multitraccia M’’ aventi n=2k tracce (dove k è il numero di nastri).

le tracce dispari emulano effettivamente i nastri mentre quelle pari simulano il movimento delle rispettive testine.

Dal momento che viene letta solo una testina alla volta, è opportuno andarle a cercare, questi simboli vengono “memorizzati” dalla macchina utilizzando gli stati come memoria.

Vi saranno quindi funzioni di transizioni che permetteranno lo spostamento delle singole “testine”, per fare ciò bisogna nuovamente cercarle ed effettuare lo spostamento.

Tutto questo è possibile dal momento che il numero di stati aggiuntivi è finito così come le funzioni di transizioni in più.

Si può inoltre notare che questa macchina non può terminare durante la ricerca delle testine durante la loro lettura/scrittura, questo perché vi sono molteplici funzioni di transizione che non lo permettono.

Fatto ciò, basta convertire la macchina multitraccia in macchina standard.

# Macchine di Turing non deterministiche

Una macchina di Turing non deterministica è una macchina in cui la funzione di transizione è una relazione di transizioni, ciò vuol dire che per ogni coppia (q,a) possono esserci più destinazioni anziché una sola:

, con d = grado di non determinismo (max numero di diramazioni per una delta)

Tutte le possibili opzioni vengono computate tutte insieme in parallelo.

La computazione iniziale è la stessa di una macchina standard, quello che cambia è la computazione: dal momento che tutto viene fatto insieme, viene creato un albero di configurazioni in cui ogni nodo ha tanti figli quant’è il grado di non determinismo (in questo caso è il numero massimo di possibili transizioni), il passo della macchina diventa quindi il passaggio da un livello all’altro della macchina.

Un cammino dell’albero è una computazione deterministica, ciò non deve essere confuso con quello della macchina standard dal momento che non vi è un ordine prefissato delle regole.

L’accettazione di una stringa avviene se almeno una configurazione della macchina termina.

Una macchina non deterministica è potente quanto una macchina di Turing standard.

## Dimostrazione

### Ls ⊇ Lr

L’idea è quella di visitare in ampiezza l’albero delle computazioni, evitando quindi l’esecuzione parallela.

### Lr ⊇ Ls

Data una macchina M standard, costruire una macchina M’ non deterministica che accetta lo stesso linguaggio. M’ ha le stesse caratteristiche di M, la differenza p che il grado di non determinismo della prima è uguale a 1.

# Accettazione di un linguaggio di una macchina di Turing

Un linguaggio che viene accettato da una macchina di Turing è detto ricorsivamente enumerabile mentre se viene deciso da essa è detto ricorsivo.

Parlare di linguaggi è però limitante, quindi è opportuno parlare di insiemi di stringhe, per farlo però bisogna avere le informazioni in una certa forma. Ciò che serve è quindi una rappresentazione dal momento che la macchina di Turing lavora solo su stringhe ( si può quindi parlare di insiemi ricorsivi e ricorsivamente enumerabili).

Nell’accettazione per terminazione, data la seguente macchina:

δ(q0,Б)=(q1,Б,R)

δ(q1,1)=(q1,1,R)

δ(q0,0)=(q1,0,S)

L’insieme accettato sono tutti i numeri naturali rappresentabili con sequenze di 1 (in totale sono 2^n -1 con n>=1).

Con la simulazione si può passare da una rappresentazione a un altra.

## Dimostrazione

Data una macchina M avente linguaggio unario, costruire Mi con linguaggio binario.

Il procedimento consiste essenzialmente nel costruire il modulo che permetta la conversione da una rappresentazione a un’altra, quindi deve esistere una macchina di Turing che permetta ciò.

Una rappresentazione è detta ragionevole quando è convertibile in un’altra rappresentazione in tempo polinomiale e conservi la stessa quantità di informazioni, inoltre i concetti di linguaggio ricorsivamente enumerabile e di linguaggio ricorsivo hanno senso solo con questo tipo di rappresentazioni.

## Funzione calcolata da una macchina di Turing

Data una funzione f:∑\* → ∑\* e una macchina di Turing avente la stringa x come input, si può dire che esiste b tale che. f(x) = b? Per capirlo bisogna far fare cose alla macchina di Turing come:

* parcheggiare la testina all’inizio del nastro;
* fermarsi in un dato stato;
* eccetera.

Tutte queste soluzioni sono tra loro equivalenti, quindi prendendo la macchina di Turing citata prima si può ad esempio parcheggiare la testina nel caso in cui f(x) non sia definito.

## Funzioni calcolabili secondo Turing

Una funzione f:∑\* → ∑\* è detta calcolabile secondo Turing se e solo se esiste una macchina di Turing che la accetta, si indica con ψM1 come l’insieme di tutte le funzioni calcolabili dalla macchina M aventi un solo argomento.

Esempio:

ψM1:N → N, ψM1(n)=n+1 in notazione unaria

ψM1(x)=x se x!=2^n -1 con n>1

2^(n+1)-1 altrimenti

E’ possibile anche utilizzare rappresentazioni differenti per input e output

### Funzioni a più argomenti

ψM2: N^2 → N con argomenti separati da Б

ψM2(n,m)=n+m+2 in notazione unaria

ψM3: N^3 → N

ψM3(n,m,0)=n+m+0+3; sempre in notazione unaria.

Si può quindi dire che ogni funzione f che viene calcolata da una macchina di Turing è calcolabile secondo Turing.

## Tesi di Church-Turing

La tesi di Church-Turing indica che tutte le funzioni calcolabili sono quelle calcolabili secondo Turing, quindi tutto il computabile può essere effettuato su una macchina di Turing.

# Problemi di Decisione

I problemi di decisioni sono un insieme di problemi che hanno come risposta sì oppure no, quindi dal punto di vista delle istanze, esse possono essere divide in istanze sì (quelle che caratterizzano il problema) e istanze no.

Dal momento che le istanze sì di un problema sono quelle che lo caratterizzano, si può ridurre il problema stesso a un insieme ricorsivamente enumerabile o ricorsivo:

* nel primo caso il problema è detto semi-decidibile, quindi è possibile dare sì alle istanze sì ma non è detto che è possibile farlo anche con le istanze no);
* nel secondo invece il problema è detto decidibile.

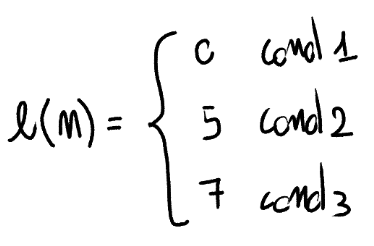
Quando un problema non è ricorsivamente enumerabile, esso è detto indecidibile.

Riprendendo in mano la tesi di Church-Turing, si può quindi dire che tutti i problemi decidibili secondo Turing e così via, da qui si può definire la cosiddetta Turing-Completezza, ovvero che un dato modello di calcolo ha una potenza equivalente a quella di una macchina di Turing.

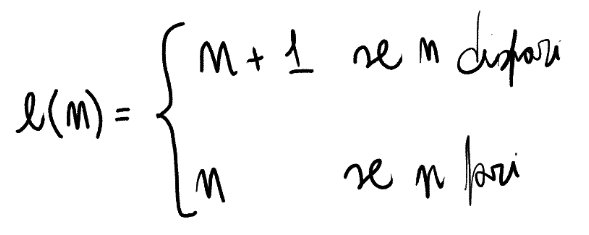
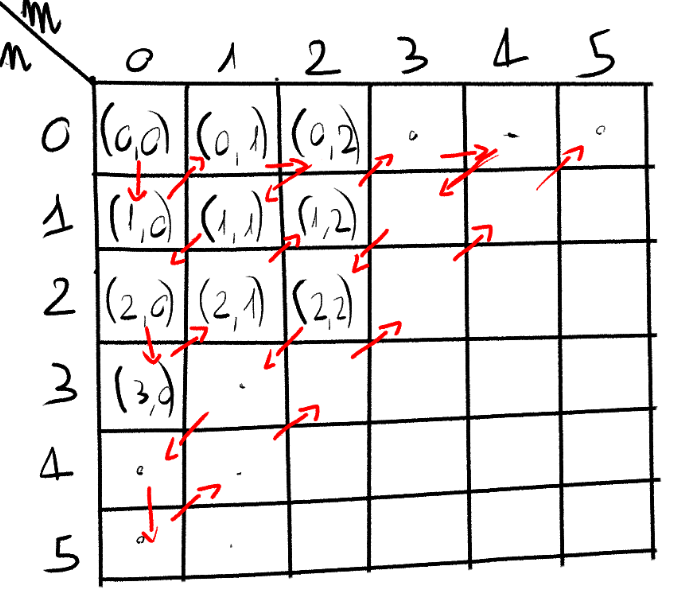
## Enumerabilità

Dato un insieme S, esso è detto enumerabile se esiste una funzione e:N→ S che sia totale e suriettiva.

Esempio:

* Z10={0,...,9} è enumerabile dal momento che esiste e tale che e(n)=n%10;
* Z057={0,5,7} è enumerabile dato che esiste e tale che dia come output 0, 5 o 7 in base a tre differenti condizioni;
* In generale ogni insieme finito è enumerabile

Nel caso degli insiemi infiniti:

* l’insieme dei numeri pari è enumerabile dal momento che la funzione e (che esiste) restituisce il numero stesso se è pari e il suo successivo se è dispari;
* L’insieme dei naturali N è enumerabile e la funzione e è l’identità: e(n)=n
* L’insieme Z dei numeri interi positivi e non è anch’esso enumerabile, l’idea è quella di alternare numeri positivi e negativi, ricavando quindi la seguente funzione:
* I numeri razionali Q sono anch’essi enumerabili, l’idea è quella di rappresentarli nella forma n/m con n e m naturali, quindi si può compilare una tabella con tutte le possibili combinazioni e andarle a prendere come in figura, questa tecnica è detta DoveTailing. In questa tecnica, ogni coppia n/m deve comparire in un numero finito di passi.

# Enumeratori ricorsivi

Un enumeratore ricorsivo è una macchina di Turing che enumera qualcosa, essa ha k nastri inizialmente vuoti e nell’ultimo di questi enumera tutti gli elementi di un insieme. Ogni elemento di un insieme S deve quindi comparire nel nastro in un numero finito di passi.

Un insieme S è ricorsivamente enumerabile se e solo se esiste un enumeratore ricorsivo che lo enumera.

## Dimostrazione (→)

Dato un enumeratore E, bisogna costruire una macchina di Turing che accetta S. In tal caso, questa macchina di Turing deve emulare E controllando l’input x per ogni elemento dell’insieme.

Il procedimento da eseguire è il seguente:

* si genera x e si mette come input della macchina di Turing;
* se viene accettato, allora X appartiene a S;

Si possono anche generare tutte le stringhe x e farle andare in parallelo, le stringhe x però sono potenzialmente infinite e/o possono esserci infiniti passi di computazione.

A fronte di questo problema, si procede nel seguente modo: alla macchina di Turing avente stringa xi si fa fare un numero j di passi, se essa viene accettata allora vuole dire che la coppia (xi, j+n) (con n>=1) viene comunque accettata, quindi (xi,j) viene inserita nell’insieme.

## Enumerazione delle stringhe

Per quanto riguarda l’enumerazione delle stringhe, l’ordine alfabetico non va bene dal momento che non è adatto a un insieme infinito di stringhe, al suo posto si utilizza quello lessicografico il quale enumera prima le stringhe di una data lunghezza per poi passare a quelle successive.

Da ciò si può derivare il concetto di enumeratore ordinato, ovvero un enumeratore che utilizza l’ordine lessicografico per gli elementi dell’insieme.

Un insieme S è ricorsivo se e solo se è enumerato da un enumeratore ordinato, allo stesso tempo deve esiste una macchina M che lo decida, quindi l’idea è quella di costruire un enumeratore ordinato basandosi su M.

Ogni input x che viene generato appartiene al linguaggio se viene deciso da M.

Per dimostrare che, se esiste un enumeratore ordinato E per S, allora questo insieme è ricorsivo, bisogna costruire una macchina di Turing M a partire da E.

Precisamente, E genera delle stringa y0, y1,... e le confronta con l’input x della macchina:

* se queste sono uguali, allora x è decisa da M e quindi fa parte di S;
* se x è maggiore di yi, vuol dire che x viene dopo nell’ordine lessicografico e quindi si passa a yi+1;
* se x è minore di yi, si restituisce no dato che yi viene dopo nell’ordinamento lessicografico.

## Note sugli enumeratori ordinati

Se esiste una funzione e:N→ S tale che S sia enumerabile, allora S è ricorsivamente enumerabile se e solo se un enumeratore lo enumera se e solo se la funzione e è calcolabile.

Inoltre, dati due numeri i e j tale che i<j, se e(i) < e(j) allora l’enumeratore è ordinato.

Dato un enumeratore E, si può assegnare la funzione e:N→ S tale che e(i) = xi, in tal caso e è calcolabile se è possibile costruire una macchina di Turing che simula E.

Se questa macchina di Turing riesce a simulare un enumeratore ordinato, allora E è ordinato.

Supponendo infatti che la funzione e sia calcolabile, è possibile costruire una macchina di Turing M tale che M(n)=e(n) che simula l’enumeratore E.

## Relazione tra ricorsività e ricorsività enumerabile

Se un insieme S è ricorsivo, allora è anche ricorsivamente enumerabile, ciò è dimostrabile dato che:

* Se un insieme è ricorsivo, allora esiste una macchina di Turing che lo decide;
* Se un insieme è ricorsivamente enumerabile, esiste una macchina di Turing che lo accetta;
* Da una macchina che decide S è possibile ricavare la variante che accetta.

Un insieme S è ricorsivo se e solo se S e il suo complementare sono ricorsivamente enumerabili.

### Dimostrazione ( → )

Se S è ricorsivo, allora esiste una macchina di Turing che lo decide e quindi che lo accetta, di conseguenza S è ricorsivamente enumerabile.

Dato che si ha una macchina che accetta S, è possibile costruirne una che accetta il suo complementare facendo tutto il contrario (va in loop negli stati finali e termina in tutti gli altri), quindi il complementare di S è ricorsivamente enumerabile.

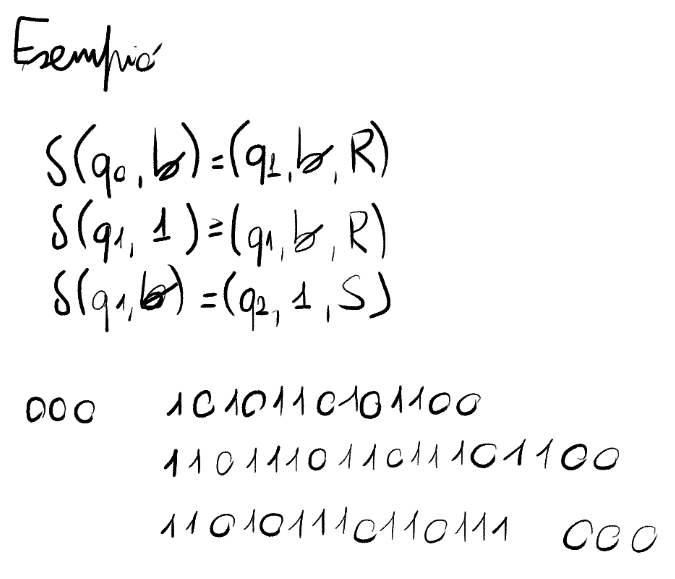
### Dimostrazione ( ← )

Date le macchine M e M’ che accettano rispettivamente l’insieme S e il suo complementare, per costruirne una che decide S si procede nel seguente modo:

* Si eseguono M e M’ in parallelo, una delle due deve per forza terminare dato che sono complementari;
* Tutto questo viene fatto facendo uno step alla volta per entrambe le macchine.

Allo stesso modo è possibile dimostrare che il complementare di S è ricorsivo, tuttavia per farlo basta costruire una macchina di Turing complementare.

### Codifica delle macchine di Turing

Data una macchina di Turing standard aventi stati finali, parcheggio della testina e altri parametri fissati, essa è individuata dalla sua funzione di transizione, si può quindi codificare una macchina da essa:

* Gli stati qi vengono codificati in una stringa di i+1 uni;
* i simboli vengono codificati allo stesso modo;
* per lo spostamento, L, R e S diventano rispettivamente 1, 11 e 111;
* la funzione di transizione segue i punti sopra separando ogni stato/simbolo/spostamento con uno 0, per separare le diverse righe si usa 00;
* l’intera sequenza viene inglobata tra 000 e 000.

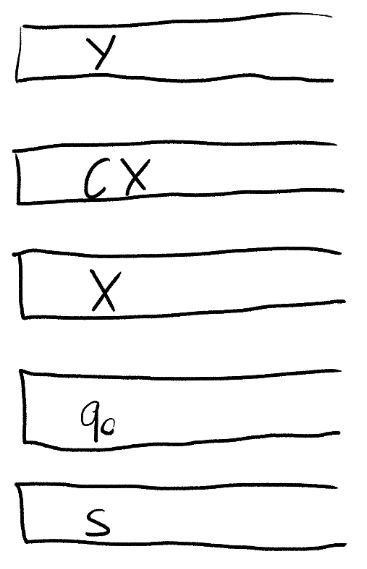
L’insieme C è l’insieme di tutte le stringhe x tali che codificano una macchina di Turing, quindi data una stringa x in C:

Dal momento che l’insieme C è ricorsivo, è possibile sapere se esiste una macchina di turing con una riga della fdt che inizia con uno stato qi e/0 se il simbolo osservato è sj.

## Macchina di Turing universale

La macchina di Turing universale è una macchina che prende una stringa y come input, essa cerca di parsificarla come cx dove c è una codifica e x è l’input.

Ciò è possibile dal momento che c è ben definita e quindi è distinguibile da x, basta infatti controllare se vi sono i due 000.

Questa macchina ha un certo numero di nastri con cui scrive lo stato e il simbolo attuale, su di essi andrà poi a scrivere le controparti nuove.

L’unico limite di questa macchina di Turing riguarda il numero di stati, ne servirebbe infatti un numero arbitrario ma tutto ciò si può “evitare” con una computazione più lunga.

I nastri della macchina universale esistono per convenienza, quindi tutto si può emulare con una macchina standard.

Data la macchina universale U e una stringa y=cx, si può dire che:

U(y)=Mc(x)

Come tutte le macchine, quella universale ha una codifica u, quindi:

U(uy)=U(y)=Mc(x) se y=cx

L’insieme di codifiche C è ricorsivo ed è formato da stringhe finite e parsificabili, dato che ogni c in C corrisponde a una macchina, si può dire che |C|=|MdT|.

Rispetto all’insieme dei naturali N, esiste una funzione e:N→ C, inoltre C è ricorsivo e quindi si può costruire un enumeratore ordinato.

Se C è ricorsivo allora è anche ricorsivamente enumerabile, di conseguenza si può costruire un enumeratore ricorsivo che lo enumera, quindi |C|<=|N|.

Dato che |C|<=|N|, si può dire che |C|<=|MdT| in quanto a una stessa macchina è possibile associare più stringhe equivalenti nel funzionamento.

Dato l’insieme di funzioni calcolabili Fc, se esiste una macchina che calcola f in Fc, allora si può associare f a c. Per costruire una macchina differente ma che calcola la stessa f basterebbe combinare gli stati e/o aggiungere delle linee della fdt.

Ciò significa che ogni funzione f in Fc è associabile a infinite macchine di Turing, quindi |Fc|<=|MdT|, Fc è una quantità al massimo enumerabile ed è un insieme ricorsivamente enumerabile dato che esistono delle macchine che le calcolano.

## Cardinalità delle funzioni definibili

SI consideri

f(0) = 0, f(1) = 1, f(2) = 1, ….

f = {0,1,1,....} -> 0,011…. => |f| = R

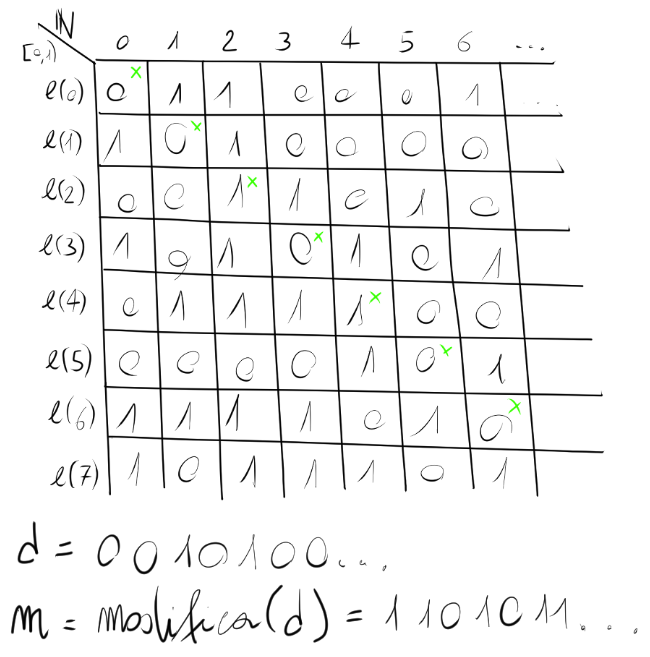
Per quanto riguarda le funzioni definibili, esse possono essere viste come stringhe binarie infinite, mettendo “0,” davanti a essi, il numero ottenuto sta nell’intervallo [0,1).

Ciò significa che f è definibile se e solo se sono in corrispondenza biunivoca con l’intervallo [0,1) in R.

Dato che |[0,1)| = |R| > |N| con |R|=2^|N|, vi sono più funzioni definibili rispetto a quelle calcolabili dal momento che queste ultime sono al massimo |N|.

### Dimostrazione

Per assurdo si suppone che esiste una funzione e:N → [0,1) totale e suriettiva, per fare la dimostrazione si utilizza la diagonalizzazione di Cantor:

* si compila una tabella con tutte le possibili combinazioni;
* si prende la diagonale e si modifica in ogni punto;
* si va a controllare se la diagonale modificata combacia con una delle righe della tabella.

Dal momento che vi possono essere infinite combinazioni, la diagonale sarà una sequenza infinita così come la sua controparte modificata.

Quindi se si cerca di far combaciare la diagonale modificata con una delle righe della tabella, si scopre che questa è diversa in almeno un valore, ciò significa che essa è un nuovo elemento e quindi la funzione non è totale.

Anche aggiungendo questo nuovo elemento in testa alla lista, il risultato non cambierebbe dal momento che si otterrebbe una nuova diagonale (e quindi un nuovo valore).

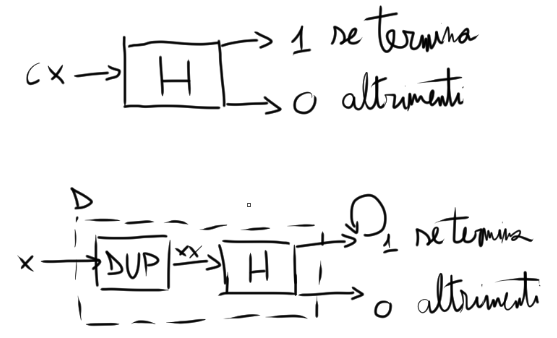
### Problema dell’Halt

La funzione Halt(c,x) è una funzione che, data una codifica di una macchina c ed un input x, restituisce 1 se la macchina termina e 0 altrimenti.

Questa funzione non è però calcolabile.

### Dimostrazione

Per assurdo si suppone che Halt(c,x) sia calcolabile, quindi esiste una macchina di Turing H che la calcola.

Si modifica H nel seguente modo:

* manda in loop dove viene restituito 1;
* si aggiunge un duplicatore all’inizio.

Si ottiene in questo modo una macchina di Turing D avente codifica d, dando d come input a D:

* il duplicatore produce dd;
* H(dd) indica che D(d) termina se e solo se Md(d) non termina, ciò significa che D(d) non termina.

In sintesi:

* D(d) termina se e solo se Md(d) non termina se e solo se D(d) non termina;
* D(d) non termina se e solo se Md(d) termina se e solo se D(d) termina.

In conclusione, tutto ciò è assurdo e quindi Halt(c,x) non è calcolabile.

### Considerazioni sull’insieme degli input di Halt(c,x)

Si prende l’insieme H come l’insieme di tutte le stringhe cx tale che c sia una codifica in C, x sia un input e Mc(x) termina, questo insieme non è ricorsivo dal momento che, se lo fosse, Halt(c,x) sarebbe calcolabile (quindi esisterebbe una macchina di Turing che lo calcola).

L’insieme H è però RE dal momento che, utilizzando una macchina universale U, è possibile capire se una stringa cx è in H o meno, quindi:

* Mh(cx) termina se e solo se cx è in H;
* Mh(cx) non termina se e solo se cx non è in H;
* U(cx) termina se e solo Mc(x) termina se e solo se cx è in H.

Si può quindi concludere che U(cx) è la macchina che accetta H e ciò rende questo insieme ricorsivo.

### Considerazione sugli insiemi ricorsivi e ricorsivamente enumerabili

Considerando ora il complementare di H, ovvero tutte le stringhe cx non parsificabili in cui c'è una codifica in C e x è in {0,1}\*\H, si può dire che:

* l’insieme non è ricorsivo dal momento che, se lo fosse, esisterebbe una macchina che lo calcola e, a partire da essa, sarebbe possibile costruirne una che calcola il suo complementare;
* L’insieme non è nemmeno ricorsivamente enumerabile dal momento che lo è anche H e quindi, se lo fosse, H sarebbe anche ricorsivo.

Si considera ora l’insieme K degli input x in cui la macchina Mx(x) termina, quindi dato che x è una codifica ma anche l’input, si può dire che l’insieme C è contenuto in {0,1}\*

L’insieme K è ricorsivamente enumerabile dal momento che si può duplicare x e dare la stringa xx a una macchina universale.

Per capire se K è anche ricorsivo, la macchina di Turing che decide K dovrebbe terminare in uno stato qf se Mx(x) termina, altrimenti termina in uno stato q non finale. Ciò però risolverebbe il problema dell’Halt e quindi K non è ricorsivo e, di conseguenza, il suo complementare non è ricorsivamente enumerabile.

Detto ciò, si può quindi concludere che un insieme X ricorsivamente enumerabile ma non ricorsivo ha un complementare che non lo è.

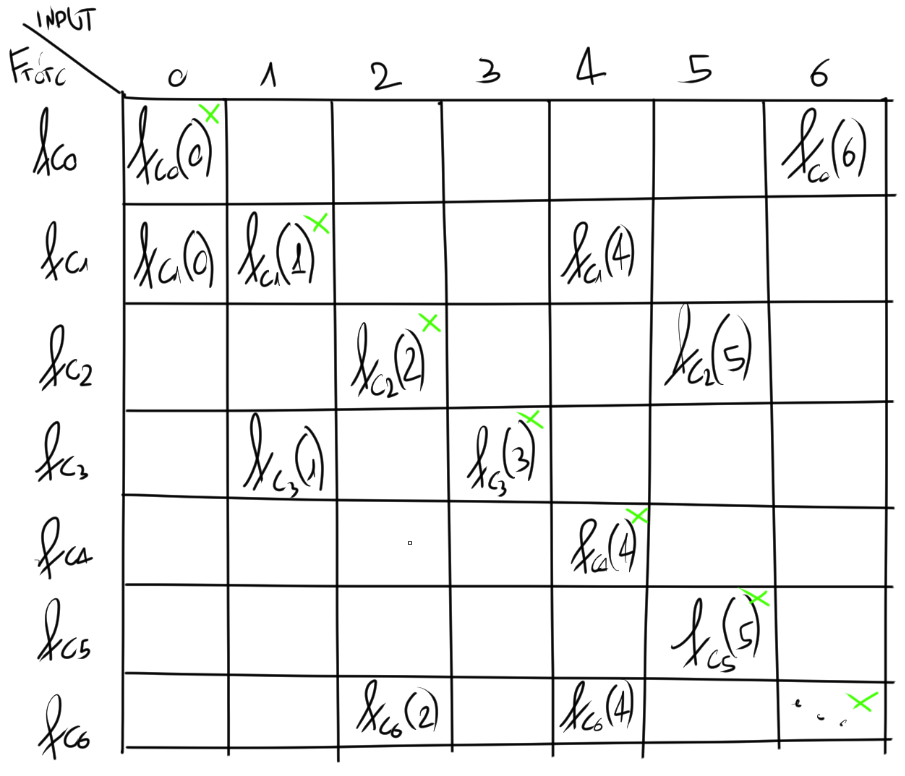
## Funzioni calcolabili e la loro codifica

Si prende ora in considerazione l’insieme delle funzioni totali e calcolabili Ftotc che è contenuto in C, per assurdo si suppone che questo insieme non sia ricorsivamente enumerabile, quindi non esiste una funzione enumerabile/una macchina di Turing che lo accetta/un enumeratore ricorsivo.

Per assurdo si suppone l’esistenza di una funzione di enumerazione e:N-->Ftotc ricorsiva.

Per dimostrare che Ftotc sia enumerabile, si prende in considerazione C: dato che Ftotc è contenuto in C e quest’ultimo è enumerabile, allora |Ftotc|<=|C|<=|N|.

Per verificare che Ftotc si anche ricorsivamente enumerabile, si utilizza la diagonalizzazione:

Si prende la diagonale d={fci(i)} e si modifica in ogni punto:

m={di + 1}

Se si confronta m con tutte le righe della tabella a fianco, si scopre che è diversa in almeno un punto, la funzione che dà m come risultato è quindi completa e non è in Ftotc, questo perchè, in caso contrario, bisogna vedere se esista una macchina che la calcola oppure se la funzione è totale.

Quest’ultimo punto è dimostrato, per quanto riguarda il primo invece:

* la funzione g(n)=fc0(n)+1 è una funzione che richiama un programma diverso per ogni n;
* Quindi si ricava da n la configurazione cn=e(n) e si dà la stringa cn n in input alla macchina universale, ciò permette il calcolo di g(n);
* g(n) è quindi calcolabile e di conseguenza appartiene all’insieme Ftotc;
* ciò però non è dimostrato dato che Ftotc non sarebbe enumerabile in tal caso, quindi g(n) non è in Ftotc.

L’ipotesi della calcolabilità è cruciale dal momento in cui si dice che l’enumeratore E è calcolabile, di conseguenza esiste una macchina di Turing che lo calcola (lo stesso discorso vale anche se non esiste E).

Ftotc è quindi enumerabile, il problema però sta nella sua calcolabilità, utile per costruire una macchina di Turing.

Tutto questo serve a dire che la funzione e non è calcolabile e quindi Ftotc non è ricorsivamente enumerabile.

Dal momento che la diagonale modificata m risulta sia calcolabile, sia in Ftotc, c’è da chiedersi se:

* m sia effettivamente calcolabile, cosa che è stata dimostrata prima e quindi m è in Ftotc;
* m sia effettivamente in Ftotc dato che, essendo una diagonale modificata, non dovrebbe appartenere a questo insieme.

Si può quindi dire che m sia uguale alla diagonale d dal momento che vi sono funzioni in cui fci non è definita, ciò indica che per ogni funzione fci vi è almeno un punto in cui non è definita.

Quando si effettua la diagonalizzazione bisogna quindi prendere la diagonale, modificarla e assicurarsi che la modifica sia stata effettuata in ogni punto.

## Importanza del ciclo while

Un risultato simile al precedente si ottiene andando a verificare l’importanza del ciclo while, ovvero un ciclo di cui non si sa la durata. Un programma senza cicli while termina, quindi si può dire che il while introduce la non terminazione ma anche altro.

Dato un linguaggio senza while Lnw,per dimostrare che l’insieme Pnw di tutti i programmi scritti in Lnw è ricorsivo o meno, si utilizza la diagonalizzazione:

Si considera una f:N→ N che prende un input x e restituisce y su cui si accetta l’output.

Come al solito si prende la diagonale d={Pi(i)} e si modifica in ogni punto, m={di+1} è calcolabile attraverso la macchina universale: U( e(n), n)+1

Dato che si è certi che m è diversa in ogni punto, si può dire che è calcolabile ma senza il ciclo while non è possibile, quindi Lnw non è un linguaggio Turing-Completo.

### Dimostrazione dell’Halt con la diagonalizzazione

Per assurdo si suppone che esiste una macchina di Turing H che, dato un input nella forma cx, restituisce sì se termina, altrimenti restituisce no.

La tabella a destra indica se una configurazione ci=e(i) con l’input xi i termina oppure no, ciò corrisponde all’aggiunta del duplicatore.

L'insieme delle stringhe 01 è enumerabile, quindi si può costruire e-1:{0,1}\* → N calcolabile.

Un ragionamento simile si può fare per le codifiche:

Data e’:N→ {0,1}\*, si può ricavare f:{0,1}\* → C che associa xi a e’(i)=ci, ciò permette di associare ogni stringa a una macchina di Turing e quindi si può considerare {0,1}\* come insieme di codifiche di macchine.

Si possono quindi sostituire i vari ci con xi, come sempre si prende la diagonale d={sì, sì, no,...} e si modifica in ogni punto (fare in cui si manda in loop la macchina sul sì).

m={loop(di)}

Mandare l’halt in loop significa invertire le terminazioni: dove Mxi(xi) termina, non termina più e viceversa. Con m calcolabile si ha quindi una macchina di Turing che termina dove l’altra non termina nonostante abbia la stessa codifica e lo stesso input.

Questa dimostrazione è molto simile a quella delle funzioni calcolabili dato che si sono non terminazioni su d, la modifica a d è possibile perchè fatta in ogni punto ma per farla è necessario decidere l’halt.

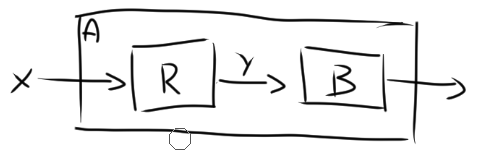
## Ricorsività dell’insieme Ftotc

Come già detto in precedenza, l’insieme Ftotc={c | per ogni x Mc(x) termina} non è un insieme ricorsivo dato che questo insieme e il suo complementare dovrebbero essere ricorsivamente enumerabili.

Precisamente nessuno dei due insiemi è ricorsivamente enumerabile dato che:

* Ftotc deve far valere la terminazione per ogni input (e sono infiniti);
* il suo complementare richiede la non terminazione, cosa che non è possibile fare.

### Dimostrazione complementare non RE

Si sa che Ftotc non è ricorsivo e nemmeno ricorsivamente enumerabile. Si può provare a utilizzare la diagonalizzazione, tuttavia c’è il rischio che la funzione g che modifica la diagonale non sia totale, in tal caso g potrebbe essere nell’insieme complementare di Ftotc.

Una seconda soluzione è la riduzione: una trasformazione delle istanze di un problema A in quelle di un problema B in cui vengono mantenute le istanze sì.

Quindi se il problema B è risolvibile, allora lo è anche A, l’unica incognita sta nel capire se la riduzione effettuata è calcolabile o meno.

Nel caso in cui la riduzione R è calcolabile e conserva le istanze sì, se A è ricorsivo e/o ricorsivamente enumerabile, allora lo è anche B.

La riduzione R non deve essere necessariamente iniettiva dal momento che viene utilizzata in un solo verso.

## Esempio: problema della stampa

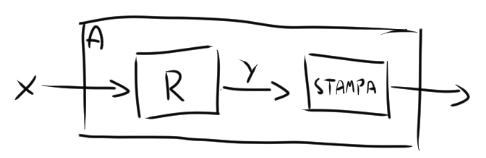
Data una macchina c, un input x e un carattere a, il predicato stampa(c,x,a) restituisce 1 se Mc(x) stampa il carattere a, 0 altrimenti.

Da ciò si possono fare le seguenti considerazione:

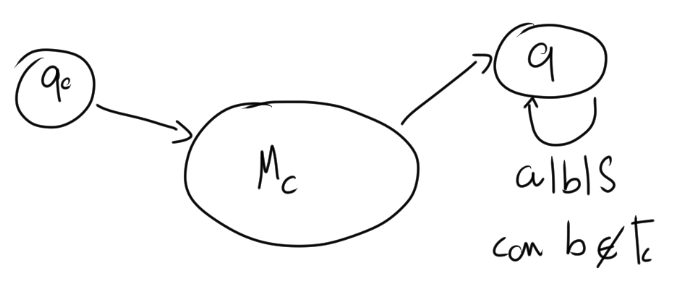
* se la stampa è calcolabile, l’insieme S={(c,x,a) | Mc(x) stampa a} è ricorsivo e il problema è decidibile;
* se invece è calcolabile la stampa parziale ( quella che va n loop su 0), allora ST è ricorsivamente enumerabile e il problema è semi-decidibile.

Il problema della stampa non è decidibile.

### Dimostrazione

Si prende un problema A non decidibile tale che A<=Stampa. Dal momento che la stampa parziale è calcolabile, mettere una funzione non calcolabile al posto di A non permetterebbe la riduzione, quindi insieme come Ftotc non vanno bene per questo scopo.

Il problema da mettere al posto di A deve essere ricorsivamente enumerabile ma non ricorsivo, come ad esempio l’Halt.

L’Halt con la riduzione applicata deve portare alla stampa, quindi da una configurazione cx bisogna ottenere c’x’a:

Halt(cx)=1 se e solo se Stampa(c’,x’,a)=1

Si suppone Mc come il grafo della macchina di Turing di c, se c termina allora può raggiungere un arco che permette la stampa.

Quello che essenzialmente si fa è aggiungere una regola nella funzione di transizione, precisamente δc’(q,a)=δc(q,a) se la regola esiste, altrimenti δc’(q,a)=(q,b,S).

Le istanze sì di Mc vengono mantenute dato che viene stampato il carattere b quando termina, al contrario se Mc non termina, non arriverà mai a q e quindi non stamperà b.

La riduzione R prende la vecchia c e produce c’ avente delle linee di delta in più, lasciando il resto invariato, precisamente:

1. Scorre c->Г;Q;
2. se δ(q,a) non è presente in delta, allora δ(q,a)=(q’,b,S)

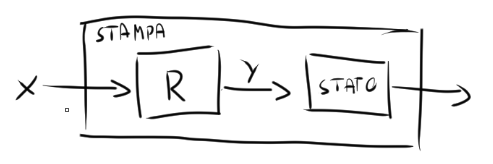
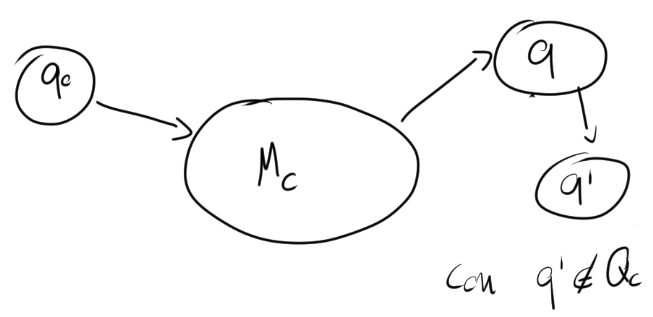
Tutto questo è fattibile e quindi R è calcolabile.

Da ciò si può quindi concludere che il problema della stampa non è decidibile dal momento che non lo è quello dell’Halt.

## Problema dello stato

Data una macchina c, un input x e uno stato q, il predicato Stato(c,x,q) restituisce 1 se la macchina Mc(x) entra in q, 0 altrimenti.

L’insieme STAT={(c,x,q)|Mc(x) entra in q} è ricorsivamente enumerabile dato che si può costruire una macchina che termina nello stato q, ciò è possibile partendo dalla macchina universale.

Per dimostrare che STAT è anche ricorsivo, bisogna dimostrare che il suo complementare non sia ricorsivamente enumerabile oppure si può sfruttare il problema della stampa.

Quel che si fa è aggiungere linee di delta: δc’(q,a)=δc(q,a) se la regola esiste, altrimenti δc’(q,a)=(q’,a,S).

La riduzione R conserva le istanze sì dal momento che se Mc va in q. allora L’Halt va in q’ e, se MC non termina, q non verrà mai raggiunto e quindi non può far terminare l’Halt.

R è inoltre calcolabile dato che, come per il problema della stampa, dell’aggiunta di righe di delta per permettere l’Halt è fattibile.

## Riduzione di -H in -Ftotc

Si sa che il Ftotc non è ricorsivamente enumerabile dal momento che decidere una codifica c per quest’insieme implica che esiste una c in cui non termina l’input.

### Dimostrazione

Si sa che:

* Se la macchina Mc(x) non termina, sicuramente c non è totale, quindi è in -Ftotc;
* Se Mc(x) termina, allora non è in -Ftotc e quindi non va bene.

Quello che si intende fare è sfruttare la terminazione e la non terminazione di Mc per costruire Mc’. Il primo punto viene rispettato da Mc’ mentre il secondo no, per farlo rispettare bisogna quindi propagare tutto:

per ogni y, Mc’(y)=Mc(x), ovvero quello che Mc fa in un solo punto, Mc’ lo fa su tutti, quindi:

Mc’=Mc(x) se Mc(x) termina (quindi è totale), altrimenti non termina (quindi non è totale).

Tutto questo dimostra che la riduzione R mantiene le istanze sì, inoltre è anche calcolabile dato che si può costruire una macchina che cancella l’input x, scrive y ed esegue la computazione.

## Considerazioni sulle funzioni costanti e calcolabili

Si considera l’’insieme delle funzioni costanti totali Ftcc={c| per ogni x, Mc(x) termina ed esiste k: Mc(x)=k}, esso è un sottoinsieme di Ftotc, quindi dato che quest’ultimo non è ricorsivamente enumerabile, anche Ftcc potrebbe non esserlo.

Si prende in considerazione la configurazione 000 000, essa è una macchina di Turing che termina in ogni caso, di conseguenza è una funzione totale.

Se si prende l’insieme di tutte le funzioni fatte in questo modo, da ciò che è stato detto prima, non dovrebbe essere ricorsivamente enumerabile, eppure si può calcolare ed è un insieme finito (quindi esiste un enumeratore ordinato e, di conseguenza, è ricorsivo).

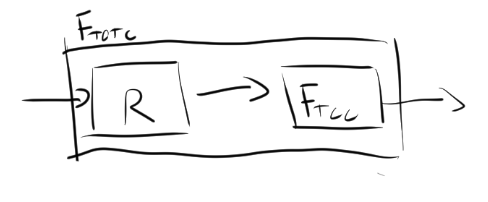
Se Ftcc è ricorsivamente enumerabile, allora dovrebbe esistere una macchina di Turing che, per ogni input x, restituisce uno stesso valore k.

Dal momento che non esiste una macchina che faccia infiniti controlli su ogni possibile x (non terminerebbe), Ftcc non dovrebbe essere ricorsivamente enumerabile.

### Dimostrazione

Utilizzando la diagonalizzazione si può notare che la diagonale viola la definizione di Ftcc, quindi non appartiene all’insieme.

Questo metodo non va bene e quindi bisogna provarne un altro, come ad esempio la riduzione.

Con la riduzione, supponendo che Ftcc non sia ricorsivamente enumerabile, si può ridurre partendo da Ftotc.

Dato che la macchina Mc termina per ogni input x (la funzione è totale), bisogna cercare di costruire Mc’ che termina e restituisce k, per farlo basta aggiungere delle righe di delta:

δc’(q,a)=δc(q,a) se esiste δc(q,a)

δc’(q,a)=”cancella nastro e scrive k” altrimenti.

La riduzione R conserva quindi le istanze sì del problema dal momento che la funzione è totale e termina sempre, inoltre la cancellazione del nastro e la scrittura di k sono fattibili (rendendo R calcolabile).

Da ciò si può concludere che Ftcc non è ricorsivamente enumerabile.

## Considerazioni sull’insieme di funzioni che assumono il valore z0 in almeno un punto

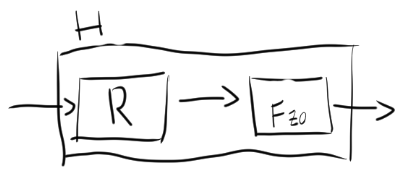
L’insieme Fzo={c | esiste x:Mc(x) termina e Mc(x)=z0} è un insieme di codifiche, quindi deve essere calcolabile. Per dire se esiste o meno una macchina di Turing che che verifica questa condizione, bisognerebbe provarla per ogni input dato che ne basta una che restituisca z0. Ciò significa che il problema è semidecidibile, per provarlo bisogna costruire una macchina che lo accetta. Per fare ciò si può provare a utilizzare il DoveTailing dato che, se è presente z0 tra i numeri, prima o poi dovrà uscire, quindi Fzo è ricorsivamente enumerabile.

Per verificare che Fzo sia anche ricorsivo si prova a vedere se -Fzo è ricorsivamente enumerabile, per farlo si utilizza la riduzione partendo da -Ftotc.

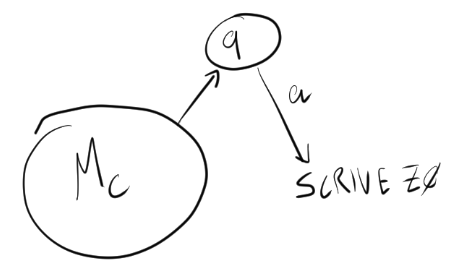
A partire da c in - Ftotc, costruire c’ in -Fzo, quindi se il risultato è uguale a z0, la macchina c va in loop, altrimenti termina.

Questo metodo non preserva le istanze sì del problema dato che:

* se c è in -Ftotc, allora c’ è in -Fzo;
* se invece c non è in -Ftotc, non è detto che c’ non sia in -Fzo, quindi non va bene.

Si può provare a ridurre Fzo partendo da H dal momento che hanno proprietà simili.

A partire da una macchina c con input x bisogna costruire c’, quindi Mc(x) termina se e solo se Mc’ restituisce z0.

Se la macchina Mc termina, bisogna aggiungere delle linee di delta in modo da scrivere z0.

Se invece Mc non termina su un input x, allora per lo stesso input non termina nemmeno Mc’ dato che non arriverebbe allo stato q (e quindi non scriverebbe z0).

La riduzione R conserva quindi le istanze sì del problema, inoltre è anche calcolabile dato che questa la costruzione di Mc’ è fattibile.

Da tutto ciò si può concludere che Fzo non è ricorsivo.

## Considerazioni sull’insieme di funzioni costanti che restituiscono 0

Si considera l’insieme Fc0 = {c | per ogni c, Mc(x)=0 }, per dimostrare che l’insieme è ricorsivamente enumerabile, Mc non deve terminare per x, quindi bisogna costruire una macchina che accetta se c appartiene a Fc0, altrimenti non termina.

Dal momento che l’insieme Fc0 è composto da funzioni costanti che restituiscono 0, la diagonalizzazione non va bene dal momento che la diagonale modificata sarebbe una sequenza di 1, che va contro la definizione dell’insieme stesso.

Si può provare a ridurre Ftc0 in Fco, da ciò si può dire che Fc0 non è ricorsivamente enumerabile dato che la riduzione è la stessa di Ftcc.

Si può osservare che non esiste un programma che indica la restituzione o meno, tutto ciò tenendo conto della valenza della tesi di Church.

# Relazione tra la calcolabilità coi linguaggi e le grammatiche

Le grammatiche e i linguaggi regolari sono ricorsivi perché rappresentabili con un automa a stati finiti, simulabile da una macchina di Turing.

In generali linguaggi e grammatiche sono ricorsivi perchè rappresentabili con un automa a pila, quest’ultimo si può simulare con una macchina di Turing utilizzando ad esempio due nastri: uno per l’input e l’altro per la pila.

Un automa a pila non è simulabile con quello a stati finiti dal momento che alcuni linguaggi context-free non sono regolari, ciò significa che questi due modelli non sono equivalenti.

## Gerarchia di Chomsky

La gerarchia di Chomsky indica un insieme di classi con cui classificare le grammatiche, esse sono:

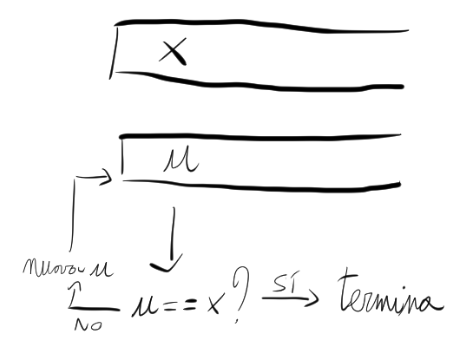
* Grammatiche senza restrizioni (Tipo 0): formata da tutte le regole u→ v tali che u è in (V+∑)+ e v è in (V+∑)\*, la parte sinistra della regola quindi non può essere vuota.
* Grammatiche Context-sensitive (Tipo 1);
* Grammatiche Context-free (Tipo 2);
* Grammatiche regolari (Tipo 3).

### Grammatiche senza restrizioni

Un linguaggio è ricorsivamente enumerabile se e solo se viene generato da una grammatica senza restrizioni, quindi ciò che si può fare con queste grammatiche lo si fa anche con una macchina di Turing.

#### Dimostrazione

Per dimostrare ciò, occorre costruire una macchina di Turing che accetta un linguaggio senza restrizioni.

Supponendo che la macchina di Turing sia non deterministica, si generano tutte le possibili stringhe in base alla grammatiche e le scrive sul nastro. Il linguaggio sarà accettato dalla macchina se almeno una di queste stringhe termina.

Dall’altra parte, se un linguaggio L ricorsivamente enumerabile e una macchina che lo accetta, per indicare che esiste una grammatica senza restrizioni G che lo genera, occorre fargli produrre tutte le possibili stringhe. Se G genera la stringa in input, allora termina, altrimenti va in loop generando stringhe.

Precisamente, G produce tutte le possibili stringhe seguite da [q0 Б x]:S→ \* x[q0 Б x], quindi se la macchina ha una regola δ(q,a)=(q’,a’,R) allora G ha una regola q a b → a’q’b.

Se in G compare [u qf w] → ε, tutto quanto viene cancellato a eccezione dell’input x.

La grammatica G simula quindi una macchina e, se termina, genera l’input x, quindi il linguaggio è ricorsivamente enumerabile se G lo genera.

### Linguaggi Context-sensitive

Una grammatica context-sensitive è una grammatica avente forma aAb→ aub dove u,a e b sono in (V+∑)\*, oppure hanno regole u → v in cui |u| <= |v|.

Se linguaggio context-sensitive sono ricorsivamente enumerabili dato che sono linguaggio senza restrizioni con dei vincoli in più, inoltre sono anche ricorsivi.

#### Dimostrazione

Data una grammatica G context sensitive che genera un linguaggio L, bisogna costruire una macchina che lo decida. Per farlo si generano tutte le possibili stringhe u in modo non deterministico e, se u è uguale all’input x, la macchina termina e accetta, altrimenti vi possono essere i seguenti casi:

* Se la lunghezza di u è minore di quella di x, allora viene generata una nuova u e riprova;
* se invece non è così, termina e rifiuta.

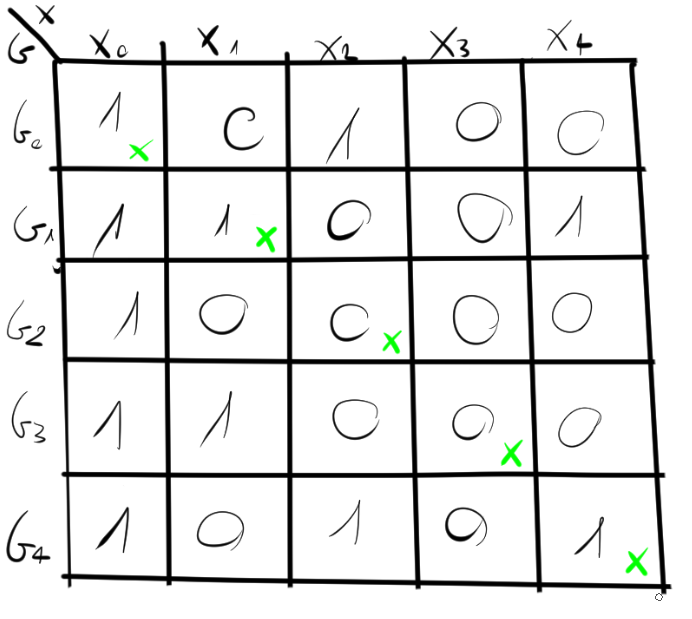
Si può notare come i due punti siano la definizione di enumeratore ricorsivo.

Esistono comunque linguaggi ricorsivi che non sono context sensitive.

#### Dimostrazione

Dato l’insieme G delle grammatiche context-sensitive, è possibile capire se queste lo sono o meno? Sì e per farlo si controlla se, per ogni regola u→ v, la lunghezza di u sia minore o uguale a quella di v.

Se ciò si può fare, allora l’insieme G è ricorsivo.

Per costruire un linguaggio ricorsivo non context-sensitive, si utilizza la diagonalizzazione.

Dati degli input xi e delle grammatiche Gi, si scrive 1 in tabella se la grammatica Gi genera xj e 0 altrimenti.

Prendendo la diagonale modificata m, si scopre che non sta in tabella, quindi m appartiene a un linguaggio L’, quest’ultimo è ricorsivo dal momento che su può costruire un enumeratore ordinato nel seguente modo: per ogni coppia (xi,Gi) se xi è generato da Gi, si restituisce 1, altrimenti 0.

Il linguaggio L’ non è context sensitive dato che non può stare nella tabella.

Si può quindi concludere che tutti i linguaggi context-sensitive sono ricorsivi ma non è vero il contrario.

I linguaggi context-sensitive sono decisi da MdT aventi spazio lineare detti Linear Bounded Automata e, anche se sono decidibili, è opportuno verificare lo spazio occupato e il tempo impiegato.

# Complessità

Col termine complessità strutturale si intende una valutazione intrinseca dei problemi, utile per scegliere il modello di calcolo più adatto e/o effettuare eventuali ottimizzazioni.

Ciò che si intende fare è lavorare in modo uniforme rispetto ai modelli e alle risorse, queste ultime variano a seconda del modello utilizzato.

Queste risorse possono essere:

* Tempo impiegato;
* Spazio occupato;
* La quantità di comunicazione (nel contesto delle comunicazioni);
* La quantità di inchiostro (il numero di scritture di una macchina di Turing);
* eccetera.

# Complessità in tempo

La complessità in tempo indica il tempo impiegato da un algoritmo, una macchina o un programma per produrre una soluzione. Nel caso delle macchine di Turing, questo partono con una configurazione C0(x) e finiscono con Ct, in tal caso il tempo di computazione tCM(x)=t dal momento che vengono effettuati t passi, uno per configurazione (assumendo che t termina).

La complessità si definisce attraverso una funzione:

La funzione f calcola il tempo massimo rispetto a tutte le stringhe x di lunghezza n.

Esempio: macchina di Turing per le stringhe palindrome:

* Con n=1 bisogna controllare tCM(0) e tCM(1), entrambe fanno due passi e quindi f(1)=2;
* Con n=2 si effettuano 4 passi dal momento che, una volta arrivati alla fine della stringa, bisogna tornare indietro per controllarla;
* Il caso peggiore è una stringa lunga n palindroma, in tal caso si effettuano 2\*(n-1)/n passi, ciò corrisponde a una O(n^2)

La complessità nel caso peggiore è quella che interessa maggiormente dal momento che si sa che le cose non possono andare peggio di così, in alcuni casi però è utile sapere la complessità nel caso migliore o nel caso medio.

Esempio: quicksort

La complessità del quicksort migliore è O(nlogn) e O(n^2) nel caso peggiore.

Nel caso medio ha inoltre bisogno della distribuzione dell’input per essere calcolabile, quindi si preferisce il caso peggiore.

Si utilizza la lunghezza dell’input come riferimento in quanto essa è legata alla quantità di informazioni rappresentabili, fornendo quindi una misura semplice.

Nell’esempio delle stringhe palindrome, il costo è O(n^2) usando una macchina a un nastro, con due nastri la complessità diminuisce a O(n), quale delle due vale? Si può passare da un modello all’altro considerando tutto questo?

Per ipotesi, la funzione f(n) deve essere maggiore o uguale a n in quanto la macchina la Turing non riuscirebbe a leggere tutto l’input se così non fosse (in tal caso f(n) è logaritmica).

Quindi, la macchina di Turing ha bisogno di avere il tempo di leggere tutto l’input.

Esempio: Nastro infinito v Nastro finito

Data una macchina M1 con nastro infinito e una M2 con nastro finito:

* per simulare M2 con M1 basta effettuare due operazioni per mettere un limite al nastro, quindi la complessità rimane la stessa dato che M1 è una M2 con un vincolo in meno:

f1(n)=f2(n)

* per simulare M1 con M2, occorre spostare f(n)/2 celle nel caso peggiore, quindi dato che per le prime n celle si fanno n passi, si ha che f2(n)=f1(n)^2.

Sfruttando la macchina multitraccia è possibile rendere questo processo lineare.

## Differenze in tempo tra macchine

### Standard v destra-sinistra

Considerando la macchina standard e quella destra-sinistra, l’idea è quella di effettuare è guardare la simulazione e vedere il costo:

* Simulare la destra-sinistra con quella standard non comporta un aumento della complessità, dato che nella prima c’è un vincolo in più:

fLR(n) = fS(n)

Al contrario simulare la macchina standard con quella destra-sinistra, si ha un aumento di fattore due a causa del vincolo in più in quest’ultima

fS(n)=2\*fLR(n)

### Standard v k tracce

Confrontando la macchina standard con quella a k tracce,

* la simulazione della prima sulla seconda non cambia dato che è un caso particolare in cui k è uguale a 1;

fkT(n)=fS(n)

* al contrario, la complessità aumenta:

fS(n)=fkT(n)+2n+2

### Standard v k nastri

Come per la macchina a k tracce, la macchina standard è una macchina a k tracce con k uguale a 1, quindi:

Per dimostrare il contrario, si sfrutta la macchina multitraccia ponendo c=2k tracce, un passo della multinastro corrisponde a:

* la scansione dell’intero nastro per trovare le testine;
* prendere i valori delle testine;
* tornare indietro ed effettuare tutte le operazioni.

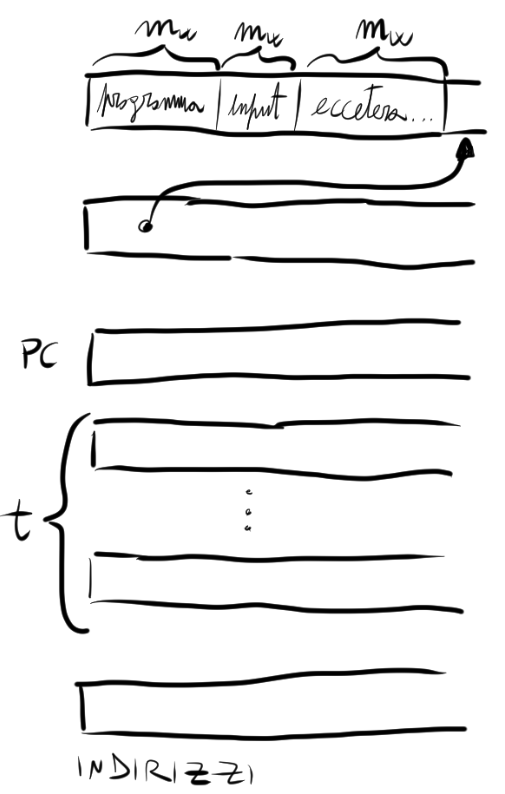
Quindi, con un numero k di nastri e un costo t per trovare l’ultima testina, si ha che:

## Computer simulato da una macchina di Turing

In questi casi è bene avere qualcosa da rapportare ai modelli di calcolo data la complessità delle macchine di Turing.

Si suppone di avere un computer con le seguenti caratteristiche:

* una memoria di parole lunghe mw;
* un set finito di istruzioni, ognuna con al massimo t operandi e modifica t celle di memoria;
* L’operazione più gli operandi deve stare al massimo in una parola di memoria;
* L’allocazione può allocare un numero ma di parole al massimo;
* gli operatori possono essere input, costanti e/o variabili, indirizzabili in modo indiretto o diretto.

Un programma è una sequenza finita di istruzione la cui esecuzione è sequenziale (a meno di salti), la sua complessità per un dato input x è data dal numero di istruzioni eseguite nel caso peggiore.

Per simulare questo modello con una macchina di Turing, servono k nastri:

* un nastro viene utilizzato come la RAM della situazione, dividendo le celle in gruppetti grandi mw. La memoria contiene il programma, gli input e tutto il necessario;
* Le parole sul nastro non vengono mai deallocate, quindi servirà un puntatore per riferire la prima parola libera, per fare ciò si utilizza un nastro;
* Un nastro è il Program Counter;
* t nastri di lavoro, in cui effettuare la computazione, appoggiandosi agli altri per lettura e scrittura;
* un ultimo nastro per indicare gli indirizzi, utile per recuperare la cella in cui si scrive un dato valore.

Quanto costa tutto questo?

L’esecuzione di un’operazione ha un costo costante dal momento che la lunghezza dell’operando è una costante che dipende dall’istruzione.

Il recupero della prossima istruzione (Fetch) ha bisogno di scorrere il programma, quindi può essere minore o uguale alla sua lunghezza.

Per quanto riguarda il caricamento in memoria, vi sono quattro possibili casi:

* Il caricamento di una costante, il costo è la lunghezza del programma;
* il caricamento di un input è la lunghezza del programma più quella di tutte le istruzioni;
* in caso di indirizzamenti diretto: mw + mk dove mk è il numero di celle occupate al k-esimo passo di computazione (nel caso peggiore la cella da caricare è l’ultima);
* in caso di indirizzamento indiretto: 2\*(mw + mk).

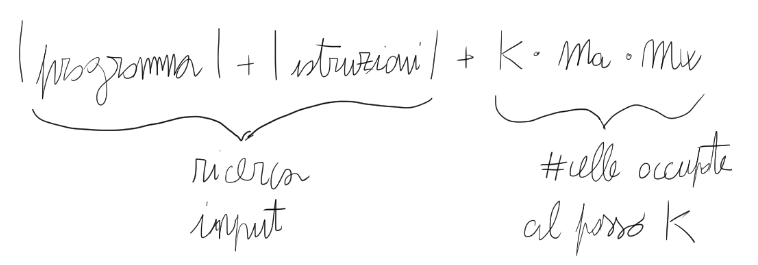
Al k-esimo passo bisogna effettuare il fetch per recuperare input e valori, tutto ciò viene fatto per t operandi sia per l’operazione di LOAD, sia per quella di STORE:

La memoria occupata è la seguente:

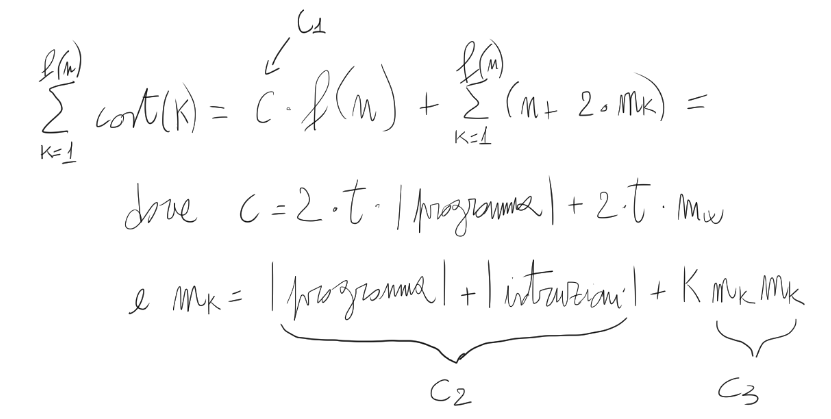
in cui:

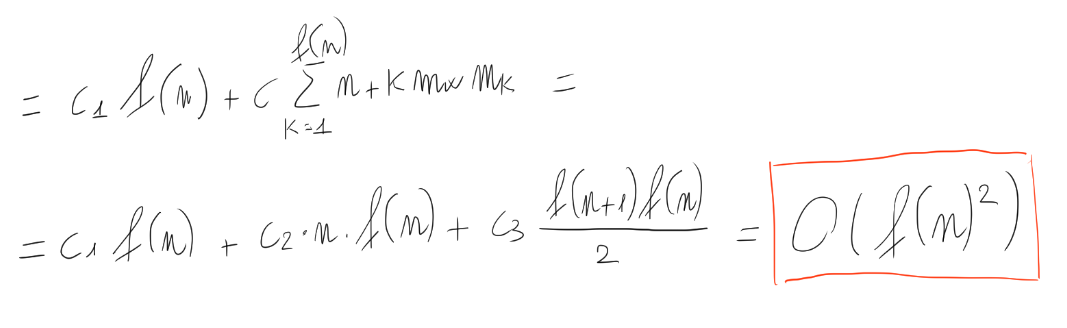
* serve per la ricerca dell’input;
* sono le celle occupate al passo k.

Il costo totale è quindi il seguente:



Il costo totale è il seguente:





Come si può notare, il rallentamento resta in ogni caso quadratico, ciò deriva dall’accesso sequenziale della memoria delle macchine di Turing.

## Rappresentazioni

Un altro problema riguarda la rappresentazione dell’input e dell’output: dati due modelli m1 e m2, occorre trovare un modo per passare da una rappresentazione all’altra. Se ciò riguarda solo l’input, allora lo si converte e si impostano le righe di δ per farla lavorare allo stesso modo.

Dato l’insieme S contenuti nei numero naturali N, gli elementi nel formato 1X sono nell’insieme N mentre quelli 0X no con X = n in binario.

S è comunque un insieme ricorsivo, quindi esiste una macchina di Turing che termina qualora il formato sia 1X mentre va in loop con 0X.

Prendendo invece un insieme ricorsivo che da binario passa ai numeri reali R senza rappresentazioni, non è possibile passare da un modello mb a uno mr, ciò significa che mr è più potente.

Il modello mr non è tuttavia quello che interessa dal momento che non dà alcuna informazione, escluderlo infatti permette di considerare le sole rappresentazioni ragionevoli.

Dal punto di vista della complessità, le rappresentazioni possono variarla dal momento che il tempo dipende dalla lunghezza dell’input e quest’ultimo dipende dalla rappresentazione.

Nel caso dei grafi, le varie rappresentazioni cambiano la lunghezza dell’input, infatti:

* la matrice di adiacenza è lunga #nodi^2;
* la lista di incidenza è lunga #nodi + # archi

Confrontandole si scopre che queste rappresentazioni sono legate polinomialmente, quindi la differenza tra lunghezze è al massimo quadratica.

Date due rappresentazioni R1 e R2:

|(x)R1| <= poly1( | (x)R2 | )

|(x)R2| <= poly2( | (x)R1 | )

Le lunghezze delle rappresentazioni ragionevoli sono legate polinomialmente, non sono però ancorate a nulla. Quel che si può fare è provare a prendere una base fissa e basarsi su quella per le rappresentazioni, rendendole in questo modo concise ma non troppo.

### Rappresentazione dei numeri

Si prende in considerazione la rappresentazione unaria, binaria e ternaria e, sulla base di esse, si costruiscono tre macchine di Turing che, dato un numero n in input, effettuino tanti decrementi fino a portarli a 0.

Si può osservare che:

* nella rappresentazione unaria si utilizzano n+1 celle per rappresentare un numero n, quindi il costo è O(n);
* nella rappresentazione binaria la lunghezza del numero è logaritmica dato che vengono utilizzati più caratteri, precisamente:

|(n)2| = log2(n)

Per scrivere tutte le n celle, il costo è 2^|(x)2|

* Con la rappresentazione ternaria si ha un caso uguale al precedente ma con un costo differente:

3^|(x)3| con |(x)3|=log3(n)

Confrontando le tre lunghezze, si può notare un rapporto esponenziale tra esse:

|(x)k| = k^|(x)k|

Confrontando i logaritmi invece, si può dire che:

log2(x) = logk(x) / log2(k) con k>1

Ciò significa che log2(x) e logk(x) sono legati attraverso una costante.

Le rappresentazioni concise sono quelle che ad esempio dei numeri naturali come quelle viste in precedenza.

Dire che una rappresentazione è concisa o meno vuol dire che in base k si utilizza lo spazio in modo efficiente mentre in base 1 lo spazio è inefficiente dal momento che con un dato numero di celle occupate, si rappresenta un solo valore.

Il fatto di non essere troppo concise riguarda l’estrazione delle informazioni e la loro computazione. Quello che interessa comunque è la computazione in quanto l’estrazione delle informazioni riguarda la rappresentazione in sè.

Riprendendo l’esempio di prima, la base 2 risulta troppo concisa?

Dato un input n, questo può contenere al più 2^n possibili valori, quindi sono stati effettuati 2^n decrementi, alcuni di essi però trattavano i numeri dispari, con essi si ottengono 2\*2^n /2 decrementi.

Saltando un carattere si hanno 2\*2\*2^n/2^2 decrementi, quindi si può indicare il numero di decrementi totale col seguente pattern:

#dec= base\*#celle\*2^n / 2^#celle

il costo è quindi esponenziale sul numero n e non sulla lunghezza effettiva dell’input, quindi è O(m).

Questa rappresentazione è però concisa,infatti in complessità si utilizzano basi k>1 dal momento che la base 1 non è naturale e non dà informazioni.

## Tesi di Church forte

La tesi di Church forte indica che i modelli ragionevoli sono tra loro legati polinomialmente alle macchine di Turing.

Con ragionevoli si intendono i modelli non realizzabili, come le macchine non deterministiche o la computazione quantistica.

Il passaggio da un modello all’altro ha al massimo una perdita polinomiale, il danno fatto da questi ultimi riguarda le rappresentazioni legate polinomialmente.

La granularità del problema è infatti legata al polinomio, per fare distinzioni bisogna fissare il polinomio e le rappresentazioni.

## Classi di complessità

* Time(f(n)) è la classe dei problemi tali che esiste una macchina di Turing che li decide in tempo f(n), essa è anche detta DTime in quanto la macchina è deterministica;
* L’unione di tutte le classi Time(n^k) forma la classe P, ovvero la più piccola classe dal punto di vista del tempo. Inoltre è una classe robusta in quanto la composizione di due macchine polinomiali genera una macchina polinomiale.

La classe P è la classe dei problemi considerati facili dato che la complessità cresce lentamente in base all’input.

## Teorema di accelerazione lineare

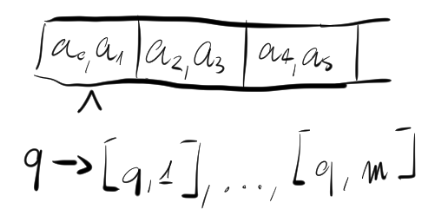
Il teorema dell’accelerazione lineare indica che, dato un linguaggio L deciso da una macchina M in tempo f(n), per ogni ε>0 esiste una macchina M’ che decide L a tempo ε\*f(n)+2n+2.

Da ciò che è stato detto prima, si può dire che, per ε>=1, la macchina ha solo più tempo a disposizione. Qualora invece ε sia compreso tra 0 e 1, vi possono essere due casi:

* se f(n)=ω(n), f(n) domina e quindi permette a M’ di essere più veloce;
* se invece f(n)=Ө(n), allora è 2n + 2 a dominare.

Tutto questo rimane lineare e permette di sottintendere O(n).

### Dimostrazione

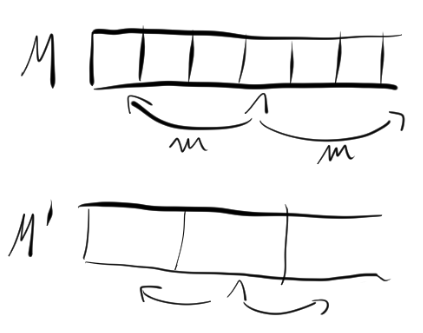
Per dimostrare il teorema, l’idea è che, data una testina di una macchina che permette la lettura di più celle, lo scopo è che, se f(n) tende a ε\*f(n), la testina deve essere grande 1/ε per far quadrare tutto.

Tutto questo non cambia l’hardware della macchina dato che ciò è possibile sfruttando il prodotto cartesiano.

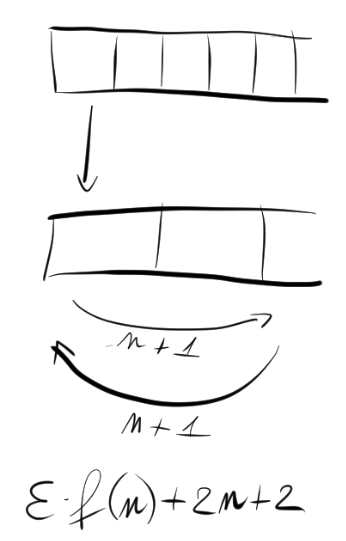
Questa nuova macchina però non può andare a caso e deve simulare quella indicata nella figura sopra.

Si possono sfruttare gli “stati come memoria” per sapere dove si trova la testina:

dato lo stato q, δ(q,a)=(q’,a’,m1) è la regola che fa a un passo, in quello successivo ci sarà la seguente:

δ(q’,b)=(q’’,b’,m2)

Ciò significa che lo stato q’ dipende da q e a e, allo stesso modo, q’’ dipende da q’ e b, quindi se si conosce lo stato q e il nastro si conosce tutta la catena e di conseguenza la computazione. Precisamente, se si conosce lo stato q e le m celle a destra e a sinistra del nastro (per un totale di 2m+1, tutta l’informazione necessaria), si conoscono tutti gli m passi della computazione.

Per eseguire m passi in un colpo solo, la macchina M deve conoscere le seguenti informazioni per simulare un passo di M’ :

* la cella attuale;
* m cella a sinistra (la supercella precedente);
* m celle a destra (la supercella successiva).

La simulazione di un passo avviene nel seguente modo:

* bisogna leggere la cella attuale e quelle immediatamente più vicine ( possibile dopo m passi);
* sapere quali celle vengono modificate e come, questo vale per entrambe le macchine;
* sapere lo stato di destinazione;
* sapere dove sarà la testina in entrambe le macchine.

La macchina simulante M farà un passo a destra e due a sinistra per leggere le informazioni essenziali, dopodiché va a modificarle dove necessario.

Nel caso peggiore, bisogna modificare le due celle laterali e tornare nel mezzo, per fare ciò la macchina impiega sempre cinque passi, questa costante dipende dalla grandezza delle supercelle m.

M’ in k passi esegue m passi di M, quindi ci mette un tempo di k\*f(n)/m e, di conseguenza, si può concludere che ε=k/m dove k è una costante e m è l’accelerazione lineare utile per far venire il tutto.

2n+2 è dovuto a una fase preliminare in cui si prepara il nastro di M, precisamente ci vogliono n+1 passi per scrivere l’input e altrettanti per tornare all’inizio.

Questa implementazione dimostrata non è efficiente, vale solo nella teoria e giustifica O() dato che, se la macchina M ha complessità εn^2 +4n+4 <100n^2, basta porre ε=1/100 per indicare che si può costruire M’ con complessità n^2.

In poche parole, basta sapere l’ordine di grandezza dal momento che si può costruire una macchina più efficiente, tutto questo dipende dalla complessità del problema.

## Funzioni Time Constructible

Una funzione f(n) è time constructible se esiste una macchina di Turing con input x che calcola una stringa lunga f(|x|) in tempo O(f(n)+n) dove n è la lunghezza dell’input e f(n) è la complessità del problema.

Esempio: quadrato di n

Nell’ultimo nastro vi sono n^2 uni e ci mette un tempo di 2n+2+n^2 per scriverli tutti.

Tutte le funzioni usuali come logaritmi e polinomi sono time constructible.

### Lemma

Data una macchina M che accetta un linguaggio L, M accetta L in tempo f(n) se per ogni stringa l in L, M(l) accetta in tempo f(n):

Se f è time constructible e il linguaggio L viene accettato da una macchina in tempo f(n), allora L è deciso da una macchina in tempo O(f(n))

#### Dimostrazione

L’idea è quella di utilizzare un cronometro costruito sfruttando un nastro della macchina, esso è una sequenza di 1 lunga f(n) i quali vengono cancellati a ogni step.

Se la macchina finisce la stringa di 1, termina e rifiuta dal momento che, avendo superato il tempo f(n), si è sicuri che non accetta.

Questa macchina M’ deve simulare una macchina M.

La computazione di M’ è O(2\*f(n)) per la costruzione dell’orologio più altrettanti per la simulazione di M, si può quindi concludere che la macchina ci mette O(f(n)).

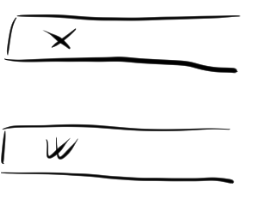
## Non determinismo

Dal momento che anche la macchina non deterministica N utilizza un certo tempo, considerando l’albero delle computazioni, se quest’ultimo termina allora si può definire un tempo di computazione della macchina N(x) come la profondità dell’albero.

Da questa funzione si può definire la classe NTime(f(n)) = {π|π è deciso da N in tempo f(n)} e da qui la classe NP come l’unione di tutti gli NTime(n^k).

## Modello Guess & Verify

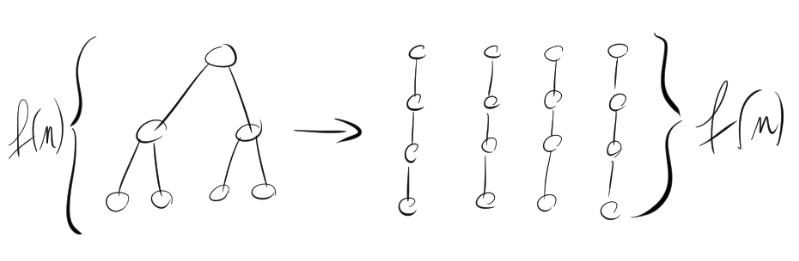
Il modello guess & verify è una macchina di Turing che lavora in due fasi:

* Guess: genera una stringa w da scrivere nel secondo nastro;
* Verify: verifica le stringhe x e w rispondendo sì o no, essa deve lavorare in tempo f(n).

Se x appartiene al linguaggio L ,allora verify restituirà sì dato che ha indovinato w, altrimenti restituirà no.

Per via della fase di Guess, questa macchina di Turing non è realistica, tuttavia è l’equivalente di una macchina non deterministica.

### Dimostrazione

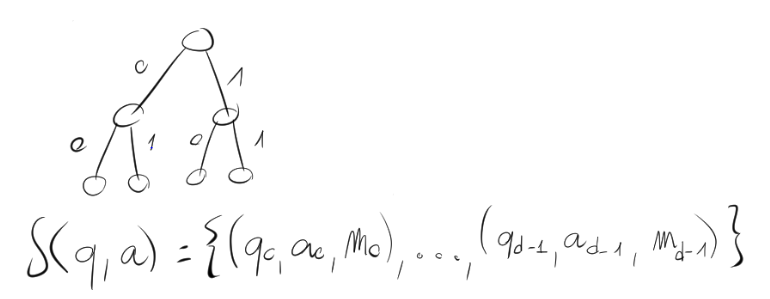
Data una macchina G guess & verify che decide un linguaggio L in tempo f(n), per costruire una macchina N non deterministica che termina sempre con tempo f(n), si scrive w nel procedere con le scelte non deterministiche.

Questa tecnica permette di scrivere ogni possibile stringa w in tempo f(n).

La fase di verifica viene effettuata prendendo ogni stringa w e testandola, ognuna di esse è lunga f(n) e, se almeno una di queste termina, allora N accetta l’input x per w.

Data una macchina non deterministica N, per costruire G di tipo guess & verify, per farlo si numerano le scelte non deterministiche.

G deve indovinare le scelte non deterministiche e quindi w è la rappresentazione di tutte le scelte.

La fase di verify percorre l’albero delle computazioni in base alle scelte fatte, se x è una stringa del linguaggio L allora esiste un ramo dell’albero delle computazioni che accetta.

La macchina G deve indovinare w e verificare il percorso controllando se termina o meno, quindi potrebbe indovinare il percorso che accetta.

Se x non è nel linguaggio L, allora N non termina dato che la stringa w è limitata a f(n).

### Osservazioni

Una macchina N non deterministica ha nodi dell’albero che accettano l’input x, quindi si può definire e

Se f è time constructible, allora esiste una macchina N’ che termina in tempo O(f(n)) dove N’ costruisce l’orologio (utilizzato da tutti i rami dell’albero) e simula N.

Se si prende una classe non deterministica, essa dà informazioni riguardanti la struttura del linguaggio anziché sul tempo.

Esempio: battaglia navale

La battaglia navale è un problema in NP e si rappresenta l’istanza nel seguente modo:

n,m,F,S;{0,...,n-1} x {0,...,m-1} → {N,A,?} x {R,C}

dove,n e m sono le dimensioni, F indica i quadretti in ogni riga/colonna, S è una soluzione parziale ed R e C sono rispettivamente la riga e la colonna.

Per dimostrare che questo problema è in NP, si costruisce una macchina guess & verify che indovina la soluzione completa e la verifica. La lunghezza della soluzione rispetta quella dell’input: |Sf| <= n

La fase di verifica è ancora polinomiale dato che il controllo è lineare (al massimo quadratico)

## Massimo sottoinsieme indipendente

Il massimo sottoinsieme indipendente (MSI) indica che, dato un grafo con nodi pesati, bisogna trovare il sottoinsieme S più pesante tale che i nodi di S non siano vicini.

Questo problema è di ottimizzazione, la sua variante per decisione indica che, dato un grafo G e un valore k, viene restituito sì se esiste un MSI(G) con valore >=k.

Con grafi lineari e alberi si può utilizzare la programmazione dinamica mentre per grafi normali il problema è NP.

### Dimostrazione

Ciò che è stato detto prima si può dimostrare con una macchina guess & verify:

* la fase di guess indovina un sottoinsieme S di dimensione lineare nell’input;
* la fase di verify controlla l’indipendenza tra i nodi e se la somma dei pesi è >= a k (si fa polinomialmente).

## Soddisfacibilità booleana (SAT)

SAT è un problema sulle forme booleane normali congiuntive.

### Definizione

* x e -x indicano i letterali;
* x1 v x2 v x3 sono le clausole;
* più clausole in OR formano la forma normale congiuntiva;
* a:V→ {T,F} è un assegnamento, una funzione che assume valore T o F.

SAT è un problema in NP e lo si può dimostrare con una guess & verify:

* la fase di guess indovina il valore di un assegnamento, producendo una stringa w contenente tutti i risultati;
* la fase di verify, applica le variabili alla formula (operazione lineare su w per trovare i valori): se la clausola risulta T allora si può saltare. Il caso peggiore è controllare l’ultima variabile in una clausola.

## Rapporto tra P e NP

La classe P si trova all’interno della classe NP in quanto il determinismo un caso particolare del non determinismo.

Da ciò si può quindi dire che le macchine polinomiali sono equivalenti a una macchina guess & verify in cui si ignora la prima fase.

Risolvere un problema in NP con una macchina deterministica quanto costa?

Dal momento che si può simulare una macchina non deterministica con una standard sfruttando la visita in ampiezza sull’albero delle computazioni lungo f(n) e con grado d, il costo è <= d^f(n) = 2^O(f(n)).

Si può quindi immaginare la macchina di Turing deterministica con due famiglie di nastri:

* la prima mantiene le configurazioni;
* la seconda mantiene le transizioni da effettuare per ogni configurazione.

Dopodiché la macchina scambia i due nastri.

Tutte queste configurazioni sono lunghe quanto l’input fino a tempo n, con tempi maggiori la dimensione totale diventa la seguente:

Dalla formula indicata sopra si può quindi concludere che la classe NTime(f(n)) è contenuta nella classe Time(2^(f(n)), da essa si ricava la classe EXP come l’unione di tutte le classi Time(2^(n^k)).

## P, NP ed EXP

Le classi P e NP sono equivalenti?

Per dire ciò, l’idea è guardare i problemi difficili di NP e lavorare su di essi, se un problema p in NP è anche P, allora lo sono anche tutti gli altri se p è difficile.

Il confronto tra due problemi avviene attraverso la riduzione polinomiale:

Se la riduzione R è polinomiale, è opportuno che sia anche trascurabile rispetto a tutto il resto in modo da far emergere la complessità dei due problemi.

R deve essere totale ma non necessariamente suriettiva o iniettiva, ciò che interessa è che dia una risposta al problema A sfruttando B.

Se l’algoritmo di B è polinomiale, allora la combinazione con R genera un algoritmo polinomiale per A, lo stesso discorso vale anche per le macchine di Turing che decidono, infatti: se B è in EXP, allora lo è anche A e così via.

Dal momento che il problema A non può essere più difficile di B per via della riduzione polinomiale, se B è in P, allora A non può essere fuori da questa classe.

Esempio: Considerando il problema 3SAT (versione di SAT con clausole grandi 3), si vuole ridurre il problema a un vertex cover.

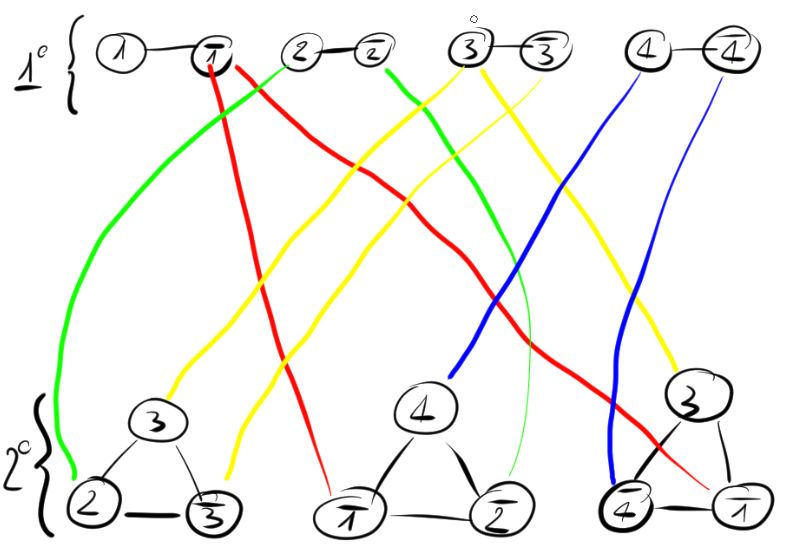


L’idea è quella di costruire un grafo con due famiglie di sottografi:

* per ogni variabili vengono definite coppie di nodi legate da un arco;
* per ogni clausola vengono definite delle triplette di nodi legate da tre archi.

Si va a completare il grafo collegando le due famiglie con degli archi, precisamente collegando i nodi di una famiglia con quelli corrispondenti nell’altra.

La formula booleana è soddisfacibile se il ricoprimento comprende m+2k nodi, quindi si pone questo valore come massimo.

Il ricoprimento degli archi del primo sottografo implica che almeno un nodo deve essere ricoperto, nel secondo invece ne servono almeno due.

Per ricoprire tutti gli archi, per ogni nodo deve esserci almeno un arco che connetta i due sottografi. Occorre quindi prendere in considerazioni le variabili che verificano le singole clausole (le quali soddisfano la formula in SAT), quindi ricoprire tutti gli altri dato che questi sono già stati presi in considerazione.

Infine, si ricoprono tutti i nodi che non hanno archi che ricoprono.

Considerando i nodi 1, -2, -3 e 4, il ricoprimento funziona dato che il secondo grafo ha ricoprimento, quindi si scelgono tutti gli altri nodi dell’assegnamento:

a(1)=T a(2)=F a(3)=F a(4)=T

L’assegnamento indicato sopra soddisfa la formula.

Per dimostrare che questa riduzione è polinomiale, la macchina di Turing che la calcola deve:

* calcola il primo sottografo scorrendo la formula una prima volta;
* scorrendo la formula una seconda volta, calcola il secondo sottografo e gli archi che lo collegano col primo.

Questo calcolo è al massimo quadratico, quindi la riduzione è polinomiale.

## Tipologia di problemi

In generale, data una classe di problemi C, un problema c è detto C-Hard se per ogni problema c’ in C, c è riducibile polinomialmente a c’, ciò vale indipendentemente dal fatto che c sia dentro la classe C o meno.

Se il problema c è C-Hard ed è nella classe C, c è detto C-Completo, quindi c contiene tutta la complessità della classe C.

Il problema C rappresenta quindi il problema più difficile della classe C.

Quindi se P è diverso da NP, una risoluzione di un problema np difficile in NP in tempo polinomiale comporterebbe che P sia uguale a NP dato che la riduzione è polinomiale.

## Teorema di Cook

Il teorema di Cook indica che SAT è NP-Hard e quindi NP completo dato che è in NP. Inoltre ogni problema p in NP si può ridurre a SAT in modo polinomiale, ciò significa che esiste una riduzione che riduce p in SAT.

### Dimostrazione

Dato che p è in NP, esiste una macchina N non determinista che lo decide in tempo polinomiale, bisogna quindi costruire una riduzione che converte x in φx mantenendo le istanze sì.

φ indica che esiste una configurazione accettante di N su input x di lunghezza ρ(|x|).

Partendo da una variabile Qik indicante che N è nello stato k all’istante i con (0<=k<=|Q|-1 e 0<=ρ(n)), all’istante i, la macchina deve essere in un certo stato, si può quindi riscrivere tutto come segue:

(Qi,0 v … v Qi,|Q|-1) //La macchina è in alcuno uno stato all’istante i.

In questo modo però c’è la possibilità che la macchina sia in più stati, quindi si risolve tutto aggiungendo:

La variabile Hi,j indica che all’istante i la testina è nella cella j, anche in questo caso bisogna imporre dei limiti sulla testina:

Con Si,j,k si indica che il simbolo sk è nella cella j all’istante i, anche qui si procede come prima:

La configurazione iniziale è la seguente:

Q0,0 H0,0 S0,0,x0 ...S0,n,xn ...S0,ρ(n),0

La funzione di transizione δ(qj,sk)=(qt,sh,mi) si impone utilizzando gli le variabili Q, S e H nel modo giusto, precisamento utilizzando l’implicazione logica:

Il tutto viene poi convertito come segue:

Bisogna inoltre aggiungere delle regole qualora δ(qj,sk) non sia definita:

Tutte le cose scritte fino a ora vanno tutte in OR tra loro.

Bisogna inoltre aggiungere delle regole:

Inoltre, se la computazione è accettante, si può supporre che vi sia un unico stato finale q|Q|-1:

La funzione ρ(n) indica la complessità nel caso peggiore (ovvero quando termina).

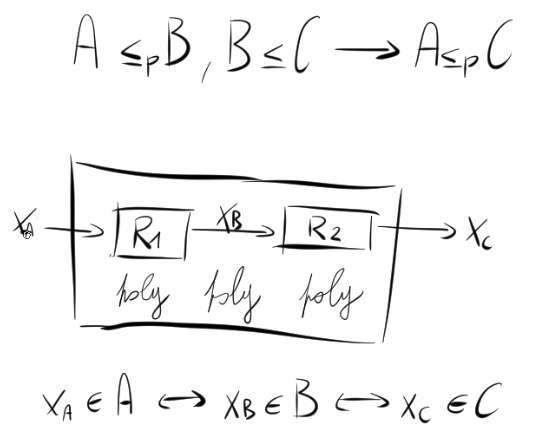
Il numero totale di clausole è il seguente:

* per la prima famiglia di Q: c\*ρ(n)\*|Q|\*log(ρ(n));
* per la seconda: c’\*ρ(n)\*|Q|\*log(ρ(n))

Lo stesso discorso vale per H e S.

La lunghezza di queste formule è polinomiale così come il tempo per scriverle e il numero di δ, inoltre il non determinismo non è un problema dato che “non fa esplodere” la macchina.

Le δ non definite non sono un problema.

Dato che alla fine c’è Qρ(n),|Q|-1, la formula viene soddisfatta se a tempo ρ(n) la macchina di Turing è nello stato q|Q|-1, le istanze sì vengono quindi conservate e tutto viene fatto polinomialmente, la riduzione è quindi calcolabile.

Tutto ciò è possibile anche con una macchina guess & verify in cui tutte le S0,n+1,0...ρ(n) vengono sostituite con delle clausole per scrivere w dopo l’input x, quindi si concentra tutto il non determinismo in w.

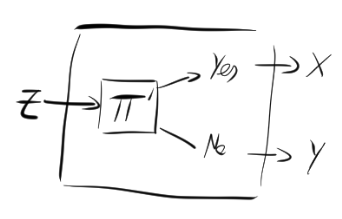
Dati tre problemi A, B e C, se A è riducibile a B e quest’ultimo fa lo stesso per C, allora A è riducibile a C.

Il problema SAT è riducibile alla tricolorabilità e viceversa.

## Considerazioni su problemi difficili

Problemi come SAT sono NP-Completi, quindi se P è uguale a NP, tutti i problemi NP-Completi sarebbero anche P-completi.

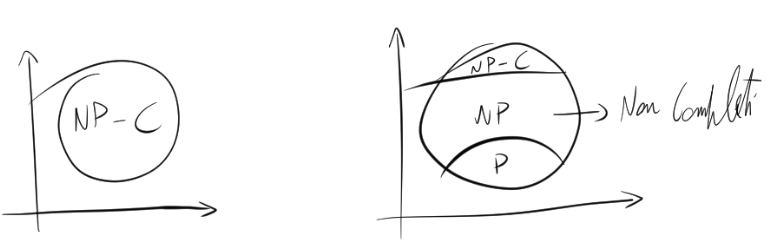
Considerando i problemi polinomiali, se un problema p ha almeno un’istanza sì e una no, allora è P-Completo.

Per essere P-Hard, per ogni problema q polinomiale, q deve essere riducibile a p in modo polinomiale, quindi data un’istanza z.

La modifica alla macchina sopra è comunque polinomiale e quindi P-Completa.

Con P uguale a NP, p è sia P-Completo, sia NP-Completo, quindi se esiste q in NP che non è NP-Completo, allora P è diverso da NP.

Ciò implica che P per essere uguale a NP ha bisogno che tutti i problemi q in NP siano NP-Completi.



## Complessità in spazio

Data una macchina di Turing, si introduce un modello con I/O avente due nastri speciali (uno per l’input e uno per l’output) più k nastri di lavoro.

La funzione di transizione funzione come sempre sui nastri di lavoro, per l’I/O invece:

* l’input non può essere modificato perchè è read-only;
* l’output si comporta come un normale nastro con l’eccezione che non va a sinistra.

La complessità in spazio indica lo spazio massimo occupato su ogni nastro di lavoro:

La separazione tra l’input e i nastri serve a separare l’input dal lavoro, cosa non possibile col tempo in quanto bisogna leggerlo.

Proprio per questo motivo è possibile ottenere complessità sub-lineari in spazio.

### Confronto tra le varie macchine di Turing

Confrontando la macchina standard con gli altri modelli, in termini di spazio si può notare che:

* la versione destra-sinistra non cambia rispetto a quella standard dato che le differenze non influenzano il nastro;
* per lo stesso motivo non cambia niente per la macchina che accetta per terminazione rispetto a quella che lo fa per stati finali;
* Per via del prodotto cartesiano, lo spazio occupato dalla macchina a k tracce non cambia;
* Sfruttando la macchina a k tracce, la macchina a k nastri occupa lo stesso spazio.
* Confrontando il computer con le macchine di Turing, si può notare che lo spazio utilizzato dai registri è costante, stessa cosa vale per PC dato che dipende dalla lunghezza del programma.

Per quanto riguarda la memoria, in tempo f(n) vengono allocate un numero di ma\*mw\*f(n) celle, di conseguenza il nastro rimane dello stesso ordine di grandezza dato che viene riutilizzato.

## Teorema di compressione del nastro

Il teorema di compressione del nastro è equivalente a quello di accelerazione lineare in spazio: dato un linguaggio L deciso da una macchina con I/O M in spazio s(n), per ogni ε>0 L è deciso da una macchina M’ con I/O in spazio ε\*s(n).

Per fare in modo che tutto ciò avvenga, si utilizza il prodotto cartesiano sui nastri di lavoro, in questo modo lo spazio dei nastri compressi diventa s(n)/m.

L’input non ha bisogno di essere compresso dato che non serve e occuperebbe spazio lineare, quello che si intende fare infatti è risparmiare spazio, non risparmiare tempo.

Per comprimere bisogna calcolare m=1/ε e definire una funzione di transizione sulla base del nuovo alfabeto (prodotto cartesiano) e che funzioni allo stesso modo.

## Funzioni space constructible

Una funzione s è space constructible se usa spazio s(|x|) dato un input x. Inoltre s è space constructible se costruisce un timer lungo s(n) sfruttando un nastro e non va oltre quella lunghezza.

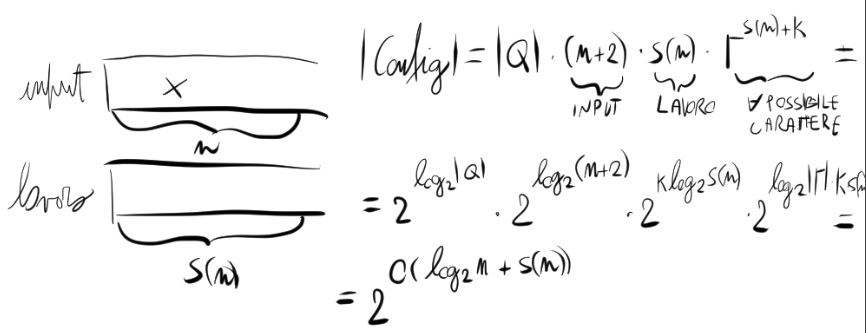
In questi casi la compressione del nastro potrebbe rivelarsi utile.

Se la macchina M lavora in tempo f(n), essa non usa spazio maggiore di f(n), quindi si può dire che la classe Time(f(n)) è contenuta in Space(f(n)).

Una funzione che utilizza spazio s(n) quanto ci mette in tempo?

Dato che una computazione è un insieme di configurazioni, se una di queste appare più di una volta, allora si è sicuri di trovarsi in un ciclo e quindi la macchina non termina. Ciò permette di dire che la terminazione vuole tutte le configurazioni siano distinte.

Nel caso si riesca a mettere un numero massimo di configurazioni distinte, si ottiene un massimo in tempo di computazione, il numero di configurazioni è quindi il seguente:



Dato che nella formula dominano log2(n) e s(n), si può dire che la classe Space(s(n)) è contenuta in Time() se s(n) termina.

Se una macchina M accetta un linguaggio L in spazio s(n), allora é deciso anche da una macchina M’ in spazio O(s(n)) con s space constructible.

Se un input x non appartiene al linguaggio L, M non termina e/o utilizza più spazio.

In generale la macchina M’ simula M e, se non termina in un tempo limitato, rifiuta, lo stesso avviene se utilizza più spazio.

Nei limiti dello spazio e del tempo, M’ funziona esattamente come M.

Per indicare lo spazio, si costruisce un timer sfruttando un nastro della macchina di Turing, oltre quel limite non si può andare, altrimenti rifiuta.

Dal momento che non si può utilizzare lo stesso metodo del tempo, si procede come segue:

* si scrive l’orologio com un numero in base c da decrementare a ogni passo;
* Quando l’orologio arriva a zero, sono stati fatti esattamente c^s(n) passi.

## Teorema della gerarchia temporale

Il teorema di gerarchia temporale indica che, data una funzione f time constructible, la classe Time(f(n)) è contenuta strettamente in Time(f(n)\*(log(f(n))^2)

### Dimostrazione

Per dimostrare tutto questo, bisogna trovare una linguaggio L tale che L non sia in Time(f(n)) e sia in Time(f(n)\*(log(f(n))^2).

Si considera L’Halt con limite temporale Hp={x|Mx(x) termina in tempo f(n)}, dal suo complementare -Hp si costruisce una macchina N non deterministica con input x che costruisce l’orologio ed effettua una sorta di diagonalizzazione che simula la macchina M per f(n) passi:

* N accetta se Mx(x) rifiuta;
* N accetta se Mx(x) ci mette più tempo;
* N rifiuta se Mx(x) accetta;

Tutto questo permette di dire che N decide -Hp.

Il fatto che Mx(x) ci metta più tempo dipende dalla simulazione dal momento che l’orologio ci mette O(f(n)) a essere costruito.

Per dimostrarlo si suppone che M con codifica u che decide -Hp con tempo f(n), ciò significa che il caso 2 non si può verificare dato che non va oltre f(n), negli altri casi Mu(u) si comporta diversamente e ciò non permette la verifica.

Questo teorema si esegue in due passi:

* Si genera un linguaggio L in Time(f(n)\*log(f(n))^2);
* Si dimostra che L non è in Time(f(n))

Dove per ogni funzione f(n), esiste un linguaggio L non appartenente a Time(f(n)).

## Teorema del Gap

Il teorema del gap indica che, data una funzione g(n) calcolabile, esiste una funzione f(n) calcolabile tale che la classe Time(f(n)) sia uguale a Time(g(f(n)).

Ponendo g(n)=2^(2^n), si ha che Time(f(n)) sia equivalente a Time(2^(2^f(n))), quindi nonostante l’aumento della risorsa, non vi sono nuovi linguaggi.

La differenza con il teorema precedente è che quest’ultima dice che f è time constructible.

## Teorema della gerarchia spaziale

Il teorema della gerarchia spaziale indica che,date due funzioni s1 e s2, s1(n)=O(s2(n)) con s2 space constructible e s2(n) maggiore o uguale a n.

### Dimostrazione

Per dimostrare tutto questo, bisogna trovare un linguaggio L che stia in s2 ma non in s1.

Quindi, data una macchina N non deterministica, N segna s2(n) e simula Mx(x):

* se Mx(x) accetta, N rifiuta;
* se Mx(x) rifiuta, N accetta;
* se Mx(x) usa più spazio, N accetta

Per segnare i nastri, la macchina N ha bisogno di spazio s2(n).

Ora si considera una macchina M con codifica u che accetta L in spazio s1(n), ciò non permette di verificare il terzo caso dato che non supera il limite, negli altri invece Mu(u) non decide lo stesso linguaggio dato che fa tutto il contrario che fa N.

s1(n)=O(s2(n)) non indica esplicitamente che s1 sia più piccolo di s2, se così fosse esisterebbe un numero n0 in cui, per ogni n>n0, s1(n)<s2(n).

Il linguaggio L accettato non è propriamente un H limitato a s2 dato che, per simulare ogni simbolo di Mx, N usa uno spazio di log2|Г|, lo spazio utilizzato è quindi s(n)/log2|Г|.

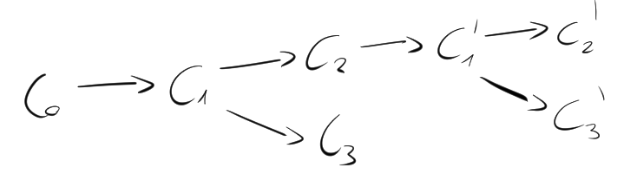
Se si prova a dare come input di N la sua codifica, N accetta dato che proverebbe a utilizzare più spazio.

## Classi non deterministiche

Data una macchina non deterministica N con input x, lo spazio utilizzato è lo spazio massimo su uno dei rami sCN(x):

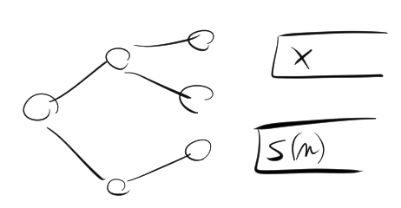
s(n)=max(|x|=n) sCN(x)

Da ciò si definisce la classe NSpace(s(n)) come la classe dei problemi decidi da una macchina non deterministica in spazio s(n).

Inoltre la classe NTime(f(n)) è contenuta in NSpace(f(n)) dato che ogni step occupa una cella di memoria.

NSpace è contenuto in NTime per qualche s(n)?

Dato l’albero delle computazioni, se c3’ accetta, allora anche c3 accetta, quindi NSpace(s(n)) è contenuta in NTime(2^(O(s(n))+log(n))) se la funzione s è space constructible.

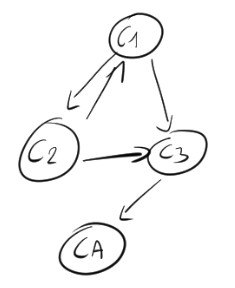
Le classi Space(s(n)) e Time(f(n)) sono rispettivamente contenute in NSpace(s(n)) e NTime(f(n)) perchè il determinismo è un caso particolare del non determinismo.

NTime(f(n)) è contenuta in Space per qualche f time constructible?

Dato che la macchina è non deterministica e l’albero è lungo f(n), si dà un numero a ogni scelta e si vede la configurazione. Dal momento che ogni configurazione è lunga f(n), si ha che NTime(f(n)) è contenuta in Space(f(n)).

NSpace(s(n)) è contenute in Time per qualche s?

In base all’albero, lo spazio su ogni ramo è s(n), quindi supponendo che la funzione sia space constructible, l’albero è profondo 2^(O(s((n))+log(n))

Si può comprimere l’albero in un grafo:

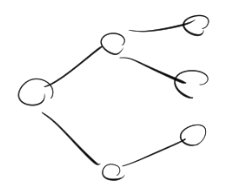
In tal caso, se si raggiunge Ca, vuol dire che la computazione è un cammino, per trovarlo si utilizza un algoritmo di visita.

La costruzione del grafo si può fare in real-time durante la visita, da ciò si può concludere che la classe NSpace(s(n)) è contenuta in Time(2^(O(s((n))+log(n)))

## Teorema di Savitch

Il teorema di Savitch indica che, data una funzione s(n)=ω\*log(n) (funzione almeno lineare) time constructible , la classe NSpace(s(n)) è contenuta in Space(s(n)^2).

### Dimostrazione



Dato l’albero, la sua profondità è 2^O(s(n)), l’idea di Savitch è quella di basarsi su un predicato P(Ci,Cj,k) indicante che se il predicato è vero, esiste un cammino da Ci a Cj raggiungibile in 2^k passi.

Ciò è vero se e solo se esiste una configurazione C’ tale che P(Ci,C’,k-1) e P(C’,Cj,k-1), quindi esistono configurazioni intermedie.

Supponendo che il tempo utilizzato sia 2^s(n) e ci sia una sola configurazione accettante Ca nella macchina non deterministica N, si genera ogni possibile Ci:

* Dire che N accetta per un dato input x vuol dire che P(C0(x),Ca, s(n));
* Quindi esista C’ tale che P(Co(x),C’,s(n)-1) e P(C’,Ca,s(n)-1);
* Dai due predicati di prima si può dire che esiste C’’ tale che … e così via;
* Alla fine si ha che P(Ci,Cj,0), ciò significa che le due configurazioni sono raggiungibili in un passo (Concetto di raggiungibilità).

Dal momento che l’albero di Ci è profondo s(n), se P(Ci,Cj,0) è verificato, allora si sale di livello per poi scendere da un’altra parte.

Ogni ipotesi è lunga s(n) e tutte queste sono s(n), quindi si ha tutto in spazio s(n)^2.

Se s(n) è più grande, allora si considera O(s(n)) nel caso vi siano più configurazioni accettanti, ciò si ripercuote sulla generazione delle configurazioni.

## Classi di complessità

### Tempo deterministico

* P: classe dei problemi risolvibili in tempo polinomiale;
* EXP: classe dei problemi risolvibili in tempo esponenziale;

### Tempo non deterministico

* NP: classe dei problemi risolvibili in tempo polinomiale con una macchina non deterministica;
* NEXP: classe dei problemi risolvibili in tempo esponenziale con una macchina non deterministica;

### Spazio deterministico

* L: classe dei problemi risolvibili in spazio logaritmico;
* PSpace: classe dei problemi risolvibile in spazio polinomiale;

### Spazio non deterministico

* NL: classe dei problemi risolvibili in spazio logaritmico con una macchina non deterministica;
* NPSpace: classe dei problemi risolvibile in spazio polinomiale con una macchina non deterministica;

### Considerazioni sulle classi

Per via della solita regola del non determinismo, le classi deterministiche sono contenute nella loro variante non deterministica.

Una macchina di Turing occupa sempre spazio deterministico, indipendentemente dal fatto che lo sia o meno in tempo.

Dal momento che una macchina non deterministica può essere simulata da un macchina deterministica in tempo 2^f(n), si ha che:

I problemi che utilizzano spazio logaritmico sono risolvibili in tempo lineare dato che NSpace(log(n)) Time(n^O(log(n))) = Time(n^k)

Per il teorema della raggiungibilità:

Per il teorema di Savitch:

Si può quindi concludere che:

Inoltre, si ha che:

* per via del teorema della gerarchia temporale;
* per via del teorema della gerarchia spaziale