Elementi di Calcolo e Complessità

Federico Matteoni

A.A. 2019/20

Indice

Introduzione

Prof. Pierpaolo Degano `pierpaolo.degano@unipi.it`
Con Giulio Masetti `giulio.masetti@isti.snr.it`
Esame: compitini/scritto + orale

Capitolo 1

Calcolabilità

1.1 Teoria della Calcolabilità

Illustra **cosa può essere calcolato da un computer** senza limitazioni di risorse come spazio, tempo ed energia. Vale a dire:

- Quali sono i **problemi *solubili*** mediante una **procedura effettiva** (qualunque linguaggio su qualunque macchina)?

- Esistono **problemi *insolubili***? Sono interessanti, realistici, oppure puramente artificiali?

- Possiamo raggruppare i problemi in **classi**?

- Quali sono le **proprietà** delle classi dei problemi solubili?

- Quali sono le relazioni tra le classe dei problemi insolubili?

Astrazione Utilizzeremo **termini astratti per descrivere la possibilità di eseguire un programma ed avere un risultato**. Questa astrazione è un **modello** che non tiene conto di dettagli al momento irrilevanti.
Un po' come l'equazione per dire quanto ci mette il gesso a cadere che non tiene conto delle forze di attrito dell'aria.

Problema della Decisione Un problema è risolto se si conosce una **procedura** che permette di decidere con un numero **finito** di operazioni di decedere se una proposizione logica è vera o falsa.

1.2 Algoritmo

Un algoritmo è un insieme **finito** di istruzioni.

Istruzioni Elementi da un insieme di **cardinalità finita** ed ognuna ha **effetto limitato** (localmente e "*poco*") sui dati (che devono essere **discreti**). Un'istruzione deve richiedere tempo finito per essere elaborata.

Computazione Successione di istruzioni finite in cui ogni passo dipende solo dai precedenti. Verificando una porzione finita dei dati (**deterministico**). Non c'è limite alla memoria necessaria al calcolo (è finita ma illimitata). Neanche il tempo è limitato (necessario al calcolo). Tanto tempo e tanta memoria quante ce ne servono.

Un'eccezione a questa definizione di algoritmo è costituita dalle macchine concorrenti/interattive, dove gli input variano nel tempo. Inoltre vi sono formalismi che tengono conto di algoritmi probabilistici e stocastici. Altre eccezioni sono gli algoritmi non deterministici, ma per ognuno di essi esiste un algoritmo deterministico equivalente (Teorema `3.3.6`)

1.3 Macchina di Turing

Introdotta da **Alan Turing** nel 1936, confuta la speranza "*non ignorabimus*" di poter risolvere qualsiasi cosa con un programma.
Turing originariamente la presenta supponendo di aver un impiegato precisissimo ma stupido, con una pila di fogli di carta ed una penna, ed un foglio di carta con le istruzioni che esegue con estrema diligenza. Non capisce quello che fa, e si chiama "**computer**".

Struttura matematica Una Macchina di Turing (MdT) è una quadrupla:

$$\mathrm{M} = (\mathrm{Q}, \sum, \delta, \mathrm{q}_0)$$

Q = $\{\mathrm{q}_i\}$ è l'insieme finito degli **stati** in cui si può trovare la macchina.
Indicheremo con lo stato speciale h la fine corretta della computazione, h non $\in$ Q.

$\sum = \{\sigma, \sigma' \dots\}$ è l'insieme finito di **simboli**. Ci sono elementi che devono per forza esistere:

- $\#$ carattere **bianco**, vuoto
- $\triangleright$ carattere di inizio della memoria, chiamato **respingente**, che funziona come un inizio file

$\delta \subseteq (\mathrm{Q} \times \sum) \to (\mathrm{Q'} \bigcup \{h\}) \times \sum' \times \{\mathrm{L, R, -}\}$ è **funzione di transizione**.
Mantiene determinismo perché funzione, ad un elemento associa un solo elemento (la transizione è univoca).
Transizioni finite perché prodotto cartesiano di insiemi finiti.
$\delta(\mathrm{q}, \triangleright) = (\mathrm{q'}, \triangleright, \mathrm{R})$, cioè se sono a inizio file possono solo andare a destra.
Può essere vista come una relazione di transizione, $\delta \subseteq (Q \times \sum) \times (Q \bigcup \{h\}) \times \sum \times \{L,\ R,\ -\}$

$\mathrm{q}_0 \in \mathrm{Q}$ lo **stato iniziale**

Mappatura a coda di rondine, bigezione tra (m, n) $\to$ k, cioè $N^2 \to N$.
Costruire un modello per il calcolo dopo aver posto delle condizioni affinché qualcosa si possa chiamare algoritmo.

1.3.1 $\sum$

$\sum^0 = \{\epsilon\}$, con ϵ = parola vuota, che non contiene caratteri

$\sum^{i+1} = \sum \cdot \sum^i = \{\sigma \cdot u \mid \sigma \in \sum \wedge u \in \sum^i\}$

$\sum^* = \bigcup_{i \in N} \sum^i$, insieme di tutte le possibili combinazioni di simboli

$\sum^f = \sum^* \cdot (\sum - \{\#\} \bigcup \{\epsilon\})$, cioè linsieme di tutte le stringhe che terminano con un carattere non bianco ma può terminare con la stringa vuota

Esempio $\sum_B = \{0,1\} \longrightarrow \sum_B^* = \{\epsilon, 0, 1, 01, 10, 010, 110010, \dots\}$ tutti i numeri binari

1.3.2 Transizioni

La **situazione corrente** di una macchina di Turing può essere scritto come (q, u, σ, v) dove:

q è lo **stato attuale**, $q \in Q$

u è la **stringa a sinistra** del carattere corrente, $u \in \sum^*$

σ è il **carattere corrente**, $\sigma \in \sum$

v è il **resto della stringa** che termina con un carattere non nullo, $v \in \sum^f$

Può anche essere più comodamente espressa come $(q, u\,\underline{\sigma}\,v)$

1.3.3 Computazione

Una computazione è una transizione $(q, x) \longrightarrow (q', \omega)$. Una macchina di Turing parte **sempre** da $(q_0, \underline{\triangleright} x)$.
Ogni computazione può esprimere il numero di passi necessari, ad esempio $\gamma \longrightarrow^n \gamma'$.
$\forall$ *computazione* $\gamma \Rightarrow \gamma \longrightarrow^0 \gamma$. Inoltre se $\gamma \longrightarrow \gamma' \wedge \gamma' \longrightarrow^n \gamma''$ allora $\gamma \longrightarrow^{n+1} \gamma''$

Esempio Macchina di Turing che esegue la semplice somma di due semplici numeri romani.

q	σ	$\delta(q,\sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	I	(q_0, I, R)
q_0	$+$	(q_1, I, R)
q_1	I	(q_1, I, R)
q_1	$\#$	$(q_2, \#, L)$
q_2	I	$(h, \#, -)$

$(q_0, \underline{\triangleright} II + III) \longrightarrow (q_0, \triangleright \underline{I} I + III) \longrightarrow (q_0, \triangleright I\underline{I} + III) \longrightarrow$
$(q_0, \triangleright II \underline{+} III) \longrightarrow (q_1, \triangleright III\underline{I}II) \longrightarrow (q_1, \triangleright IIII\underline{I}I) \longrightarrow$
$(q_1, \triangleright IIIII\underline{I}) \longrightarrow (q_1, \triangleright IIIIII\underline{\#}) \longrightarrow (q_2, \triangleright IIIII\underline{I}) \longrightarrow$
$(h, \triangleright IIIII)$

Esempio Macchina di Turing che verifica se una stringa di lettere a, b è palindroma o no.

q	σ	$\delta(q,\sigma)$
q_0	$\triangleright$	$(q_0, \triangleright, R)$
q_0	a	$(q_A, \triangleright, R)$
q_0	b	$(q_B, \triangleright, R)$
q_0	$\#$	$(h, \#, -)$
q_A	a/b	$(q_A, a/b, R)$
q_A	$\#$	$(q_{A'}, \#, L)$
$q_{A'}$	a	$(q_R, \#, L)$
q_B	a/b	$(q_B, a/b, R)$
q_B	$\#$	$(q_{B'}, \#, L)$
$q_{B'}$	a	$(q_R, \#, L)$
q_R	a/b	$(q_R, a/b, R)$
q_R	$\triangleright$	$(q_0, \triangleright, R)$

$(q_0, \underline{\triangleright} abba) \longrightarrow (q_0, \triangleright \underline{a} bba) \longrightarrow (q_A, \triangleright\triangleright \underline{b} ba) \longrightarrow$
$(q_A, \triangleright\triangleright b\underline{b}a) \longrightarrow (q_A, \triangleright\triangleright bb\underline{a}) \longrightarrow (q_A, \triangleright\triangleright bba\underline{\#}) \longrightarrow$
$(q_{A'}, \triangleright\triangleright bb\underline{a}) \longrightarrow (q_R, \triangleright\triangleright b\underline{b}) \longrightarrow (q_R, \triangleright\triangleright \underline{b}b) \longrightarrow (q_R, \triangleright\underline{\triangleright} bb)$
$\longrightarrow (q_0, \triangleright\triangleright \underline{b}b) \longrightarrow (q_B, \triangleright\triangleright\triangleright \underline{b}) \longrightarrow (q_B, \triangleright\triangleright\triangleright b\underline{\#}) \longrightarrow$
$(q_{B'}, \triangleright\triangleright\triangleright \underline{b}) \longrightarrow (q_R, \triangleright\triangleright \underline{\triangleright}) \longrightarrow (h, \triangleright\triangleright\triangleright)$

1.4 Linguaggi di Programmazione

Un primo formalismo di algoritmo, come abbiamo visto, è la **macchina di Turing**: attenendosi alle richieste di tempo e spazio arbitrariamente grandi ma finiti, risolve un **problema**.
Un secondo formalismo sono i **linguaggi di programmazione**.

1.4.1 Sintassi

Sintassi astratta Definiamo la **sintassi** dello scheletro di un semplice linguaggio di programmazione imperativo.
Una **sintassi astratta** è una sintassi non concreta, cioè che non tiene conto di alcune cose come la procedenza tra gli operatori.
Sintassi

Expr → E ::= x | n | E + E | E · E | E - E

Bexpr → B ::= tt | ff | E < E | ¬B | B ∨ B

Comm → C ::= skip | x = E | C;C | if B then C else C | for i = E to E do C | while B do C

Abbiamo una serie di insiemi da definire ulteriormente

x ∈ Var, l'insieme delle **variabili**

n ∈ N, **numeri naturali**.

Abbiamo anche la **memoria** per poter **assegnare ad una variabile il suo significato**

σ : Var $\rightarrow_{fin} N$
Si dice "a dominio finito", indicata dal fin sotto la freccia, per indicare che il dominio Var ha cardinalità finita.
Var dominio è quindi un sottoinsieme di Var insieme delle variabili che sarebbe infinito.
La memoria si può aggiornare, diventando $\sigma' = \sigma[\mathrm{x} \mapsto \mathrm{n}]$.
Ad esempio, $\sigma'(y) = \mathrm{n}$ se y = x, altrimenti $\sigma'(y) = \sigma(y)$

1.4.2 Funzioni di Valutazione

Inoltre, per valutare le espressioni generate dalla grammatica, servono delle **funzioni di valutazione**. Esse **trovano il significato di ogni espressione**

Funzione di valutazione delle espressioni
$\mathscr{E}$: Expr $\times$ (Var $\to N) \to N$
La sua **semantica denotazionale** è la seguente

$$\mathscr{E}[\mathrm{x}]_\sigma = \sigma(\mathrm{x}) \qquad \mathscr{E}[\mathrm{E}_1 \pm \mathrm{E}_2]_\sigma = \mathscr{E}[\mathrm{E}_1]_\sigma \pm \mathscr{E}[\mathrm{E}_2]_\sigma$$
$$\mathscr{E}[\mathrm{n}]_\sigma = \mathrm{n}$$

Importante notare come gli operatori $+$, $-$, $\cdot$ *dentro* le espressioni siano dei **semplici token denotazionali**, mentre sono gli operatori *valutati* ad eseguire il vero e proprio calcolo. Per chiarire questo aspetto, facciamo un esempio. Valutiamo con la nostra funzione $\mathscr{E}[\mathrm{E}_1 + \mathrm{E}_2]_\sigma = \mathscr{E}[\mathrm{E}_1]_\sigma$ *più* $\mathscr{E}[\mathrm{E}_2]_\sigma$. Se non definiamo l'operatore "*più*", allora se poniamo $\sigma(\mathrm{x}) = 25$ la valutazione

$$\mathscr{E}[3 + \mathrm{x}]_\sigma = \mathscr{E}[3]_\sigma \textit{ più } \mathscr{E}[\mathrm{x}]_\sigma = 3 \textit{ più } 25 = 42$$

è corretta quanto

$$\mathscr{E}[3 + \mathrm{x}]_\sigma = \mathscr{E}[3]_\sigma \textit{ più } \mathscr{E}[\mathrm{x}]_\sigma = 3 \textit{ più } 25 = 28$$

Ovviamente utilizzeremo la valutazione specificata in precedenza e gli operatori aritmetici assumeranno il loro significato standard.
L'unica eccezione è l'operatore $-$, che nel nostro caso sarà il **meno limitato** dal simbolo $\dot{-}$, la cui unica differenza è che non può dare un risultato inferiore a 0. Ad esempio, $5 \mathbin{\dot{-}} 7 = 0$

Funzione di valutazione di espressioni booleane
$\mathscr{B}$: Bexpr $\times$ (Var $\to N) \to \{\mathrm{tt}, \mathrm{ff}\}$
La cui **semantica denotazionale** è la seguente

$$\mathscr{B}[\mathrm{tt}]_\sigma = \mathrm{tt} \qquad \mathscr{B}[\neg \mathrm{B}]_\sigma = \neg\mathscr{B}[\mathrm{B}]_\sigma$$
$$\mathscr{B}[\mathrm{ff}]_\sigma = \mathrm{ff}$$
$$\mathscr{B}[\mathrm{E}_1 < \mathrm{E}_2]_\sigma = \mathscr{E}[\mathrm{E}_1]_\sigma < \mathscr{E}[\mathrm{E}_2]_\sigma \qquad \mathscr{B}[\mathrm{B}_1 \vee \mathrm{B}_2]_\sigma = \mathscr{B}[\mathrm{B}_1]_\sigma \vee \mathscr{B}[\mathrm{B}_2]_\sigma$$

Anche qua vale il medesimo discorso sulla definizione sugli effettivi operatori.

1.4.3 Semantica Operazione Strutturale

Structural Operationale Semantics Metodo attraverso il quale viene fornita la semantica dei comandi. Parte da un **insieme di configurazioni** Γ

$$\Gamma = \{(\mathrm{C}, \sigma) \mid \mathrm{FV(C)} \subset \mathrm{dom}(\sigma)\} \cup \{\sigma\}$$

dove FV(C) sono le **variabili del programma** e con $\mathrm{FV(C)} \subset \mathrm{dom}(\sigma)$ si richiede che tutte le variabili del programma abbiano un valore nella memoria fornita. Si fa l'unione con la sola memoria σ perché la situazione finale è $(\ , \sigma)$ che, analogamente allo stato fittizio h nella macchina di Turing, segnala la fine dell'esecuzione. Inoltre si hanno le **transizioni** $\to$

$$\to \subset \Gamma \times \Gamma$$

Definiamo quindi un **insieme di transizioni (Γ, $\to$)** tramite delle **regole di inferenza** del tipo $\frac{premesse}{conclusione}$. In assenza di premesse, $-$, la regola di inferenza si dice **assioma**.

$$\frac{-}{(skip, \sigma) \to \sigma} \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_1, \sigma)} \mathscr{B}[B]_\sigma = tt$$

$$\frac{-}{(x = E, \sigma) \to \sigma[x \mapsto n]} \mathscr{E}[E]_\sigma = n \qquad \frac{-}{(if\ B\ then\ C_1\ else\ C_2, \sigma) \to (C_2, \sigma)} \mathscr{B}[B]_\sigma = ff$$

$$\frac{(C_1, \sigma) \to (C_1', \sigma')}{(C_1; C_2, \sigma) \to (C_1'; C_2, \sigma')} \qquad \frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \to \sigma} \mathscr{B}[E_2 < E_1]_\sigma = tt$$

$$\frac{-}{(for\ i = E_1\ to\ E_2\ do\ C, \sigma) \to (i = n_1;\ C;\ for\ i = n_1 + 1\ to\ n_2\ do\ C, \sigma)} \mathscr{B}[E_2 < E_1]_\sigma = ff \wedge [E_1]_\sigma = n_1 \wedge [E_2]_\sigma = n_2$$

$$\frac{-}{(while\ B\ do\ C, \sigma) \to (if\ B\ then\ C;\ while\ B\ do\ C, \sigma)}$$

1.5 Calcolabilità

1.5.1 T-Calcolabile

Dati Σ alfabeto della macchina, Σ_0 alfabeto di input e Σ_1 alfabeto di output, con $\#, \triangleright \notin \Sigma_0 \cup \Sigma_1 \subset \Sigma$

$$\text{M=(Q, } \Sigma, \delta, \text{q}_0) \text{ calcola } f : \Sigma_0^* \longrightarrow \Sigma_1^* \Leftrightarrow (\forall w \in \Sigma_0^* \wedge f(w) = x \Rightarrow M(w) \to_{fin} (\text{h}, \triangleright\text{z}))$$

Si dice che la **funzione** f è **T-Calcolabile**.
Cioè, esiste una macchina di Turing che per ogni stringa finita in input arriva, con un numero finito di passi, all'arresto lasciando sul nastro la stringa di output corretta. Notare come non viene data nessuna interpretazione al risultato della f.

1.5.2 `while`-Calcolabile

$$\text{C calcola } f : \text{Var} \to N \Leftrightarrow (\forall\ \sigma : \text{Var} \to N \wedge f(x) = n \Rightarrow \text{C}(\sigma) \to_{fin} \sigma' \wedge \sigma'(x) = n)$$

Si dice che la funzione f è **`while`-Calcolabile**.
Cioè esiste un programma C che calcola il risultato corretto in un numero finito di passi.

Invariante Tutti i risultati visti fin'ora **sono invarianti rispetto al modello dei dati**, e questo vale anche per la T-Calcolabilità e la `while`-Calcolabilità.
In particolare, se ho i dati in un formato A allora posso codificarli nel formato B in cui opera la macchina, calcolare il risultato in formato B e decoficarlo nel formato A di partenza. Questo vale se **le codifiche sono funzioni biunivoche e "*facili*"**. Vedremo cosa significa essere "facili", ma per adesso basti pensare ad un numero finito di passi e che terminano sempre.

1.5.3 Esempio di codifica

	0	1	2	3	4	5
0	0	2	5	9	14	
1	1	4	8	13		
2	3	7	12			
3	6	11	...			
4	10	16				
5	15					

Codifica a coda di rondine

Codifica $(x,\ y) \mapsto \frac{1}{2}(x^2 + 2xy + y^2 + 3x + y)$
Es. $(3,\ 1) \mapsto \frac{1}{2}(9 + 6 + 1 + 9 + 1) = \frac{26}{2} = 13$

Decodifica $n \mapsto (n - \frac{1}{2}k(k+1),\ k - (n - \frac{1}{2}k(k+1)))$
con $k = \lfloor \frac{1}{2}(\sqrt{1 + 8n} - 1) \rfloor$

Es $8 \mapsto (8 - 6,\ 6 - 8 + 3) = (2,\ 1)$
$k = \lfloor \frac{1}{2}\sqrt{1 + 8 \cdot 8} - 1 \rfloor = 3$
$\frac{k(k+1)}{2} = 6$

1.6 Notazione

Una **funzione** f è $\subset$ A $\times$ B, con A spazio di partenza e B codominio. Quindi f(a) = b si può esprimere anche con (a, b) $\in f$, con a $\in$ A e b $\in$ B.

$f(a) = b \wedge f(a) = c \Rightarrow b = c$

Considereremo **funzioni parziali**, cioè funzioni con A contenente punti dove f non è definita. Non è quindi detto che $\forall$ a $\in$ A $\exists$ b $\in$ B | f(a) = b

f **converge** su a, cioè $f(a)\downarrow \Leftrightarrow \exists\, b \mid f(a) = b$

f **diverge** su a, cioè $f(a)\uparrow \Leftrightarrow \nexists\, b \mid f(a) = b$

Dominio di f: $dom(f) = \{a \mid f(a)\downarrow\}$

Immagine di f: $imm(f) = \{b \mid \exists\, a \in A \Rightarrow f(a) = b\}$

Rapporto tra algoritmi A e funzioni f f è un **insieme potenzialmente infinito di coppie**, ma non posso assegnare due f diverse allo stesso insieme, mentre esistono tanti algoritmi diversi che calcolano la stessa funzione. Ad esempio, $f = \emptyset$ è calcolata da `while(true) do skip` ma anche da `while(true) do skip;skip`.

1. Quali sono le funzioni calcolabili?
   Nelle ipotesi iniziali di definizione di algoritmo, per adesso conosciamo le T-Calcolabili e le `while`-Calcolabili.

2. Quali proprietà hanno?
   Posso combinarle?

3. Esistono funzioni non calcolabili?

4. Sono interessanti?
   Esistono a prescindere dalla macchina?

Algoritmi e calcolabilità Per ora abbiamo definito gli algoritmi in base al loro comportamento, sotto forma di **configurazioni che si susseguono** del tipo (istr. corrente + . . . , memoria). Abbiamo anche diversi modi di affrontare la calcolabilità:

1. **Hardware**, con la macchina di Turing
   Questo è uno dei primi esempio di calcolo, è semplice da capire e si descrivono direttamente macchina che eseguono gli algoritmi. Uno dei primi approcci allo studio della complessità.
   Cambio programma → Cambio macchina

2. **Software**
   Ho l'interprete, cioè la semantica, fissi. Se cambio il programma non devo cambiare la macchina

   (a) Programmi `while`
   Base della programmazione iterativa, dalla semantica operazionale e anch'essi usati per lo studio della complessità

   (b) Funzioni ricorsive
   Base della programmazione funzionale

1.7 Funzioni ricorsive primitive

Per formalizzare i vari modi con cui possiamo esprimere le funzioni, usiamo quella che si chiama **λ-notazione**. Queste espressioni individuano gli argomenti all'interno di un'espressione che descrive una funzione, scritta seguendo un'opportuna sintassi.

$$\lambda < \textit{variabili} > . < \textit{espressione} >$$

Esempio $\lambda\, x,\, y.expr$
Gli **argomenti** dell'espressione *expr* sono x, y. Si dice anche che x, y **appaiono legate da λ in *expr***.
Invece un qualsiasi altro simbolo di variabile w in *expr* **non è da considerarsi argomento** dell'espressione, e viene definito **libero** in *expr*.
Altri **esempi** per evidenziare la **notazione**:

$\lambda\, y.x + y$

$\lambda\, x\lambda\, y.x + y$ che può essere riscritta come $\lambda\, x,\, y.x + y$ ed equivale a dire $somma(x, y) = x + y$ dando così il nome $somma$ alla funzione.

$\lambda\, x_1, x_2, \ldots, x_n.expr$ riscritta come $\lambda\, \overrightarrow{x}.expr$

1.7.1 Classe C

La classe C delle **funzioni ricorsive primitive** è la **minima classe** di funzioni che obbediscono alle seguenti regole di inferenza, regole di sintassi per definire le funzioni.
Casi base

Zero: $\lambda\overrightarrow{x}.0$
Prende un vettore di argomenti e restituisce 0.

Successore: $\lambda x.x + 1$
Prende un valore e restituisce il suo successore.

Proiezione/Identità: $\lambda\overrightarrow{x}.x_i$
$\overrightarrow{x} = x_1, \ldots, x_n$, $1 \leq i \leq n$

Casi iterativi

Composizione
$g_1, \ldots, g_n \in C$ con k argomenti ("a k posti") e
$h \in C$ a n posti
$\Rightarrow \lambda x_1, \ldots, x_n.h(g_1(\overrightarrow{x}), \ldots, g_n(\overrightarrow{x})) \in C$

Ricorsione primitiva
$h \in C$ a $n + 1$ posti e
$g \in C$ a $n - 1$ posti

$$\Rightarrow \begin{cases} f(0, x_2, \ldots, x_n) = g(x_2, \ldots, x_n) \\ f(x_1 + 1, x_2, \ldots, x_n) = h(x_1, f(x_1, x_2, \ldots, x_n), x_2, \ldots, x_n) \end{cases}$$

$f \in C \Leftrightarrow$ esiste una successione $f_0, \ldots,\, f_n = f \mid \forall f_1$ è ottenuto con i casi base oppure f_i è ottenuto con i casi iterativi da f_j con $j < i$

Esempio Esempio di funzioni ricorsive
$f_1 = \lambda x.x$
$f_2 = \lambda x.x + 1$
$f_3 = \lambda x_1, x_2, x_3.x_2$
$f_4 = f_2(f_3(x_1, x_2, x_3))$

$$\begin{cases} f_5(0, x_2) = f_1(x_2) \\ f_5(x_1 + 1, x_2) = f_4(x_1, f_5(x_1, x_2), x_2) \end{cases}$$

Proviamo a calcolare $f_5(2, 3) =$

Regola di valutazione interna-sinistra: valuto per primo quello che sta dentro i parametri partendo da sinistra.
$= f_5(1 + 1, 3) =$
$= f_4(1, f_5(1, 3), 3) =$
$= f_4(1, f_4(0, f_5(0, 3), 3), 3) =$
$= f_4(1, f_4(0, f_1(3), 3), 3) =$
$= f_4(1, f_4(0, 3, 3), 3) =$
$= f_4(1, f_2(f_3(0, 3, 3)), 3) =$
$= f_4(1, f_2(3), 3) =$
$= f_4(1, 4, 3) =$
$= f_2(f_3(1, 4, 3)) =$
$= f_2(4) =$
$= 5$

Vediamo cosa succede con una **regola di valutazione esterna**: $f_5(2, 3) =$
$= f_4(1, f_5(1, 3), 3) =$
$= f_2(f_3(1, f_5(1, 3), 3)) =$
$= f_3(1, f_5(1, 3), 3) + 1 =$
$= f_5(1, 3) + 1 =$
$= f_4(0, f_5(0, 3), 3) + 1 =$
$= f_2(f_3(0, f_5(0, 3), 3) + 1 =$
$= f_3(0, f_5(0, 3), 3) + 1 + 1 =$
$= f_5(0, 3) + 2 =$
$= f_1(3) + 2 =$
$= 3 + 2 =$
$= 5$

Meno Limitato Non ritorna mai numeri negativi, ma 0.
$f_7(x, y) = y$

$f_8(x, y) = x$

$$\begin{cases} pred(0) = 0 \\ pred(x + 1) = f_8(x, pred(x)) \end{cases}$$

$f_9(x, y, z) = pred(f_3(x, y, z)$

$$\begin{cases} f_{10}(0, y) = f_1(y) \\ f_{10}(x + 1, y) = f_9(x, f_{10}(x, y), y) \end{cases}$$

$\Rightarrow x \dot{-} y = f_{10}(f_7(x, y), f_8(x, y))$

Somma Non è altro che generalizzazione del successore, applico il successore tante volte quante servono.

$$\begin{cases} 0 + y = y \\ (x + 1) + y = (x + y) + 1 \end{cases}$$

Prodotto Sfrutto la somma

$$\begin{cases} 0 * y = 0 \\ (x + 1) * y = (x * y) + y) \end{cases}$$

Potenza Generalizza il prodotto

$$\begin{cases} x^0 = 1 \\ x^{y+1} = (x^y) * x \end{cases}$$

C'è un modo per generalizzare la potenza? $\Rightarrow$ **Ackerman**.

Relazione Diciamo che la relazione $R(x_1, \ldots, x_n) \subset N^n$ è **ricorsiva primitiva** se lo è la sua **funzione caratteristica** χ_R definita come

$$\chi_R(x_1, \ldots, x_n) = \begin{cases} 1 & se\ (x_1, \ldots, x_n) \in R \\ 0 & se\ (x_1, \ldots, x_n) \notin R \end{cases}$$

Quindi se χ_R è ricorsiva primitiva allora anche R è ricorsiva primitiva.

Esempio $P = \{\ n \in N \mid n$ è un numero primo$\}$ è ricorsiva primitiva. Questo per il teorema di fattorizzazione unica.
$\forall x \in N\ \exists$ numero finito di esponenti $x_1 \neq 0 \mid x = p_0^{x_1} \cdot p_1^{x_1} \cdot \ldots \cdot p_n^{x_n}$
Come trovare tali esponenti con f ricorsiva primitiva.
M = (Q, Σ, δ, q_0)
Q = {q_0, ..., q_k), Σ = {σ_0, ..., σ_n)
Kurt Gödel: rappresentare algoritmi come numeri: **Gödelizzazione** data macchina di turing M trovo i che è il suo numero di Gödel.

1.7.2 Funzione di Ackermann

La funzione di Ackermann **non è definibile** mediante gli schemi di ricorsione primitiva definiti in precedenze, ma è totale ed ha una definizione intuitivamente accettabilissima.

$A(0,0,y) = y$

$A(0,x+1,y) = A(0,x,y) + 1$

$A(1,0,y) = 0$

$A(z+2,0,y) = 1$

$A(z+1,x+1,y) = A(z, A(z+1,x,y), y)$ **doppia ricorsione**

La **doppia ricorsione** presente non è un problema: tutti i valori su cui si ricorre decrescono, quindi i valori di $A(z,x,y)$ sono definiti in termini di un numero finito di valori della funzione A. Quindi intuitivamente A è calcolabile. Inoltre **cresce più rapidamente di ogni funzione ricorsiva primitiva** ma **non è ricorsiva primitiva**.
Ma cosa calcola? Una sorta di esponenziale generalizzato, infatti:

$A(0,x,y) = y + x$

$A(1,x,y) = y * x$

$A(2,x,y) = y^x$

$A(3,x,y) = y^{y^{\cdot^{\cdot^{\cdot y}}}} \}\ x \textit{ volte}$

1.7.3 Realizzazione

Con il linguaggio-`while` e il linguaggio-`for` posso riprodurre i casi base della ricorsione. In particolare, per ogni programma `for` esiste una funzione ricorsiva primitiva e viceversa.

Un programma che calcola lo 0 è un programma che legge gli ingressi e scrive 0 in uscita.

Il successore lo realizzo con un assegnamento uscita = ingresso +1

La proiezione consiste nel leggere in memoria la variabile x_i cercata e metterla in uscita

Realizzo h tale che $h(g_1(x,y,z), g_2(x,y,z))$, con programma p_1 associato a g_1, p_2 associato a g_2 e p_3 associato a h.
Il programma che realizza la composizione sarà quindi p_1;p_2;p_3.

Per la ricorsione primitiva $\begin{cases} f(0,y) = g(y) \to p_1 \\ f(x+1,y) = h(x, f(x,y), y) \to p_2 \end{cases}$ che dopo qualche passaggio abbiamo visto che $f(x+1,y) = h(x, f(x,y), y) = h(x, h(x+1, f(x+1,y)), y)$. Associando p_1 a g e p_2 a h, lo realizzo con il programma-`for`

```
t1 = g(y);
for (i = 1 to x + 1):
        t1 = g(i, t1, y);
end
```

Per la **funzione caratteristica** $\chi_I(n) = \begin{cases} 1 \ n \in I \\ 0 \ \textit{altrimenti} \end{cases}$

1.8 Diagonalizzazione

Esiste un formalismo che esprime tutte e sole le funzioni totali calcolabili? No
Dobbiamo necessariamente avere a che fare con funzioni parziali, ma perché "no"?
Qualunque formalismo o esprime solo funzioni totali ma non tutte, oppure esprime anche funzioni parziali. La dimostrazione è fondamentale per la teoria della calcolabilità: prende il nome di **diagonalizzazione**.

Dimostazione Fissiamo il formalismo delle funzioni ricorsive primitive, posso prendere l'algoritmo di Gödel per numerarle. Quindi avrò $f_0, f_1, \ldots, f_n, \ldots$
Definisco $g(n) = f_n(n) + 1$ (*diagonalizzazione* viene da usare lo stesso indice per indice e parametro).
Se g è una ricorsiva primitiva, allora è numerabile. Diciamo che g ha come numero i: $f_i(n) = g(n) = f_n(n) + 1$
Se diagonalizzo avrò $f_i(i) = g(i) = f_i(i) + 1$ ma **non può essere** che $f_i(i) = f_i(i) + 1$
$\Rightarrow g$ non è ricorsiva primitiva.

Se io prendo le funzioni parziali, posso applicare lo stesso ragionamento?
$\phi(x) = \psi_x(x) + 1$
Diciamo come prima che $\phi(x)$ ha indice i, quindi $\psi_i(x) = \phi(x) = \psi_x(x) + 1$
Se $\psi_i(x)$ diverge, allora $\psi_x(x)$ diverge e anche $\psi_x(x) + 1$ diverge, quindi sono uguali. Non si applica il ragionamento della diagonalizzazione nel caso delle funzioni parziali.

1.9 μ-ricorsive

Minima classe $\mathscr{R}$ che, allo schema fino alla ricorsione primitiva, si aggiunge:

Minimizzazione $\phi(\overrightarrow{x}, y) \in \mathscr{R}$
$\psi(\overrightarrow{x}) = \mu y.[\phi(\overrightarrow{x}, y) = 0 \wedge \forall z < y \mid \phi(\overrightarrow{x}, z) \downarrow]$
$\mu x.[I]$ è il minimo elemento di I insieme.

Cosa significa? Data una funzione appartenente a $\mathscr{R}$ (che ovviamente può essere una ricorsiva primitiva), la vado a calcolare sugli argomenti $\overrightarrow{x}$ della ψ e su una certa y. Se vale 0, il risultato è y, altrimenti **deve** convergere e vado avanti incrementando y di 1 e ricalcolando fino a che non trovo un risultato pari a 0.
Quindi le μ-ricorsive definiscono anche funzioni non totali, al contrario delle ricorsive primitive che definiscono solo funzioni totali.

Esempio $\phi(x, y) = 42$ è costante, quindi ricorsiva primitiva, quindi anche μ-ricorsiva.
$\psi(x) = \mu y.[\phi(x, y) = 42]$ ovunque indefinita perché non tornerà mai 0 quindi $\nexists y$.

Quindi **terminazione e non terminazione sono cruciali**.
Se la definisco per casi? Ad esempio $f(x) = \begin{cases} \mu y.[y < g(x) \mid h(x, y) = 0] & \textit{se} \exists \textit{ taley} \\ 0 & \textit{altrimenti} \end{cases}$ con g, h ricorsive primitive.
f è ricorsiva primitiva, perché composizione di ricorsive primitive, ed è anche totale, perché converge sempre.
Quindi **se pongo dei limiti al numero di tentativi**, dato da $y < g(x)$, si ricade nelle ricorsive primitive e non ci sono problemi di parzialità.

1.9.1 Notazione

Per ragioni storiche, una relazione $I \subset N^n$ è **ricorsiva** (sinonimo di totale) $\Leftrightarrow$ la sua funzione caratteristica χ_I è **calcolabile totale**.
Inoltre, come già detto, I è ricorsiva primitiva $\Leftrightarrow \chi_I$ è ricorsiva primitiva.

1.10 Tesi di Church-Turing

Le funzioni intuitivamente calcolabili sono tutte e sole le T-calcolabili.
In realtà è un'ipotesi, ma è così forte che viene presa come tesi. Ci permette di non considerare il formalismo con cui formalizziamo gli algoritmi, poiché tutti i formalismi rappresentano la stessa *classe di elementi*. Ci limiteremo a dire algoritmo, Macchia di Turing... indifferentemente, poiché grazie a questa tesi possiamo dire che un algoritmo è equivalente qualsiasi sia il linguaggio in cui è scritto.

1.10.1 Risultati

Indichiamo con ϕ_i è la **funzione calcolata dall'i-esimo algoritmo** M_i

ϕ_i è funzione $\rightarrow$ **semantica**

M_i è algoritmo $\rightarrow$ **sintassi**

Può succedere per $i \neq j$ che $\phi_i = \phi_j$ ma $\mathrm{M}_i \neq \mathrm{M}_j$ (ad esempio `while(true) do skip` e `while(true) do skip;skip`).

T-calcolabili = `while`-calcolabili = μ-calcolabili

D'ora in avanti parliamo solo di funzioni calcolabili, quindi ϕ_i *è* calcolabile.

Tempo di calcolo $\exists$? una funzione calcolabile totale $t(i,n)$ che maggiora il tempo di calcolo di M_i(n)? No. Vediamo come dimostrarlo, introducendo una **diagonalizzazione**.

$$t(i,n) = \begin{cases} k & \textit{se } M_i(n) \downarrow \textit{in meno di } i \textit{ passi} \\ 0 & \textit{altrimenti} \end{cases}$$

Sia T_i la misura **esatta** del tempo di calcolo di M_i. $\mathrm{T}_i(\mathrm{n}) \leq t(i,n)$ è calcolabile totale e $T_x(x)$ **tempo di calcolo effettivo**, $t(x,x)$ **tempo di calcolo stimato**

$$\psi(x) = \begin{cases} \phi_x(x) + 1 & \textit{se } T_x(x) \leq t(x,x) \\ 0 & \textit{altrimenti} \end{cases}$$

Quindi anche ψ è calcolabile totale. Applico Church-Turing, quindi $\phi_i(i) = \psi(i) = \begin{cases} \phi_i(i) + 1 & \textit{se } T_i(i) \leq t(i,i) \\ 0 & \textit{altrimenti} \end{cases}$

Ma siccome ϕ_i è calcolabile totale, non può succedere che, quando termina, sia $\phi_i(i) = \phi_i(i) + 1$, **è un assurdo**.
Quindi $t(i,n)$ non è calcolabile totale, di conseguenza **non c'è modo di stimare il tempo di calcolo**.
Per lo stesso motivo non possiamo imporre limiti allo spazio.

Spazio di calcolo $\exists$? una funzione calcolabile totale che dato M_i, x dice quante celle di memoria uno specifico calcolatore C userà per calcolare $\mathrm{M}_i(x)$?

$$h(i,x) = \begin{cases} 1 & \textit{se } M_i(x) \uparrow \textit{(su C)} \\ 0 & \textit{altrimenti} \end{cases}$$

Sia n la cardinalità di Σ, $m-1$ cardinalità di Q e k il numero di celle di C.
Posso quindi scrivere n^k **stringhe diverse**: il cursore può stare su k posizioni diverse e la macchina in m stati diversi.
Il **numero massimo di configurazioni diverse** (stato con posizione del cursore e stringa su nastro) è $l = n^k \cdot k \cdot m$, con n^k possibilità di nastro scritto, k possibili posizioni del cursore e m stati ($m - 1 + 1$ per lo stato h di **halt**)
Dopo l passi, quindi, la configurazione si ripete necessariamente. Si può dire che **la macchina è in loop**.
Siccome la macchina attraversa un **numero finito di configurazioni superiormente limitato da** l, se la macchina non si è arrestata prima di l passi allora troverò per forza una configurazione già vista in precedenza, che mi porterà in una configurazione già vista e così via. **Quindi non terminerà mai**.
Ho dimostrato che h è calcolabile totale, ma posso scrivere quindi t come nella dimostrazione precedente, ma giungo all'assurdo già visto. **Quindi non posso mettere un limite al nastro.**

I seguenti risultati sono **invarianti rispetto all'enumerazione scelta**.

1.10.2 Teorema 1: Le Funzioni Calcolabili sono tante quante i numeri naturali

Le f calcolabili sono $\#N$. Anche le f calcolabili totali sono $\#N$.
Esistono funzioni *non* calcolabili, molte di più di quelle calcolabili.

Dimostrazione Non sono più di $\#N$ perché posso calcolare le macchine di Turing. Almeno $\#N$ perché posso costruire una macchina che per qualsiasi input lascia un numero naturale sul nastro, quindi di queste ce ne sono almeno quanti sono i numeri naturali.

Quindi indichiamo con ϕ_i la funzione (in generale, parziale) calcolata dall'algoritmo M_i e i **indice della macchina**. Come detto prima, può darsi che per $i \neq j$ sia $\phi_i = \phi_j$ ma sicuramente $\mathrm{M}_i \neq \mathrm{M}_j$.

1.10.3 Teorema 2: Ogni funzione calcolabile ϕ_i ha infiniti (numerabili) indici

Anche detto **padding lemma**.
Non solo, posso costruire un insieme infinito di indici A_i tale che $\forall\, j$ in A_i $\phi_j = \phi_i$ mediante una funzione ricorsiva primitiva.

Dimostrazione Sia M_i un programma `P`. Prendo `P;skip`, poi `P;skip;skip`... metto tanti `;skip` quanti voglio. Posso generare un numero infinito di programmi che calcolano tutti la stessa funzione.

1.10.4 Teorema 3: Forma Normale

Prendendo un qualsiasi algoritmo, posso riscriverlo in una **forma canonica/normale**, che non è per forza migliore ma è una forma specifica e da noi **privilegiata**.
$\exists$ un predicato $T(i, x, y)$ e una funzione $U(y)$ ricorsive primitive calcolabili totali tali che $\forall$ funzione calcolabile i, $x.\phi_i(x) = \mu\mathrm{y}.[U(T(i, x, y))]$
Corollario: tutte le funzioni T-calcolabili sono anche μ-calcolabili. Non solo, ma $\mu\mathrm{y}$ corrisponde al `while`, e T, U a due programmi `for`. Quindi ogni funzione calcolabile può essere ottenuta da due programmi scritti con il linguaggio `for` ed una sola applicazione del linguaggio `while`.

Dimostrazione Devo costruire il predicato T e la funzione U.
$T = (i, x, y)$ è detto **predicato di Kleene** ed è **vero** $\Leftrightarrow$ y **è la codifica di una computazione di** $\mathbf{M}_i(x)$ **terminante**.
Per calcolare, T prende i e recupera M_i. Comincia a scandire i valori y, li decodifica e, uno alla volta dato x ingresso, controlla se il risultato è una computazione terminante. Definisce U in modo che $U(y) = z$, con z risultato della computazione (ciò che rimane sul nastro della MdT corrispondente).
C'è un solo y nelle macchine deterministiche, se c'è.

1.10.5 Teorema 4: Teorema di enumerazione

$\exists\, z$ tale che $\forall\, i$, x $\phi_z(i, x) = \phi_i(x)$. Quindi z è **MdT universale**. z interprete, i è programma.
Fra tutti gli algoritmi, ce ne sono infiniti numerabili in grado di eseguire tutti gli altri algoritmi.

Dimostrazione $\phi_i(x) = \mu y.U(T(i, x, y)$ per il terzo teorema. Quindi $\phi_i(x)$ è un algoritmo, avrà un indice per Church-Turing che indicheremo con z.
Diciamo $\phi_i(x) = \phi_z(i, x) = \mu y.U(T(i, x, y))$ (senza y come argomento perché quantificata, non libera ma legata, una sorta di *variabile di lavoro*). Applico la transitività dell'uguaglianza e ho finito, $\phi_i(x) = \phi_z(i, x)$.

1.11 Macchina di Turing Universale

Ottimo modello per le macchine Von Neumann.

Due insiemi Utili per la rappresentazione
$Q_* = \{q_0, q_1, \ldots\} \not\ni h$, insieme numerabile ma **non è l'insieme degli stati** della MdTU ma è un insieme ausiliario.
$\Sigma_* = \{\sigma_0, \sigma_1, \ldots\} \not\ni L, R, -$ insieme di simboli

Codifica K $K : Q_* \cup \{h\} \cup \Sigma_* \cup \{L, R, -\} \longrightarrow \{\,|\,\}^*$
Ognuno degli elementi q_i, σ_i ecc. viene codificato in questo modo.
$h \mapsto |$
$q_i \mapsto |^{\,i+2}$, con la barra verticale ripetuta $i+2$ volte
$L \mapsto |$, $R \mapsto ||$, $- \mapsto |||$
$\sigma_i \mapsto |^{\,i+4}$
C'è un problema: $|||||$ è uno stato ($\in Q_*$) o un simbolo ($\in \Sigma_*$)? Vedremo come disambiguare.
Q_* contiene **tutti i possibili stati delle MdT**, e Σ_* contiene **tutti i possibili simboli delle MdT**. Quanto visto fin'ora è ausiliario alla costruzione della MdTU.

Costruzione Prendo una MdT qualunque $M = (Q, \Sigma, \delta, s)$. Ordiniamo gli stati ed i simboli.
$Q = \{q_{i_1}, q_{i_2}, \ldots, q_{i_k}\}$, finito perché gli stati sono finiti.
$\Sigma = \{\sigma_{j_1}, \sigma_{j_2}, \ldots, \sigma_{j_l}\}$.
Con $i_1 < i_2 < \ldots < i_k$ e $j_1 < j_2 < \ldots < j_l$. Con questo abbiamo supposto Q e Σ **totalmente ordinate**.
Considero questo alfabeto: $\{\,|, c, d, \#, \triangleright\}$ con $|, c, d \notin Q_* \cup \Sigma_*$.
Adesso possiamo codificare le quintuple $\in \delta$. $\delta(q_{i_p}, \sigma_{j_q}) = (q, \sigma, D)$ con D un certo simbolo di direzione $(L, R, -)$. Come codificarla? Attraverso la seguente stringa

$$s_{p,q} = cK(q_{i_p})cK(\sigma_{j_q})cK(q)cK(\sigma)cK(D)c$$

con c che funge da **separatore**, **tra la codifica di uno stato e la codifica di un simbolo**.
$s_{p,q} = cK(q_{i_p})cK(\sigma_{j_q})cdcdcd$ se $\delta(q_{i_p}, \sigma_{i_q})$ è indefinita

Esempio $\overline{M} = \{\{q_2\}, \{\sigma_1, \sigma_3, \sigma_5\}, \delta, q_2\}$, con $i_1 = 2$, $j_1 = 1$, $j_2 = 3$, $j_3 = 5$

$q_2 \to q_{i_1}$

$\sigma_1 \to \sigma_{j_1}$

$\sigma_3 \to \sigma_{j_2}$

$\sigma_5 \to \sigma_{j_3}$

$\delta(q_2, \sigma_1) = (h, \sigma_5, -) \mapsto s_{1,1} = c|^4c|^5c|c|^9c|^3c$ perché $K(q_2) = |^{2+2}$, $K(\sigma_1) = |^{1+4}$, $K(h) = |$, $K(\sigma_5) = |^{5+4}$, $K(-) = |||$
$\delta(q_2, \sigma_3) = (q_2, \sigma_1, R) \mapsto s_{1,2} = c|^4c|^7c|^4c|^5c|^2c$
$\delta(q_2, \sigma_5)$ non definita $\mapsto s_{1,3} = c|^4c|^9cdcdcdc$

Codifica della macchina ρ Possiamo ora definire una funzione di "*codifica*" (in realtà iniettiva, perché in fase di decodifica possiamo non trovare una M corrispondente) della macchina in esame. (s senza indici è lo stato iniziale)
$\rho(M) = cK(s)cs_{1,1}s_{1_2} \ldots s_{1,l}c \ldots cs_{k,1}s_{k_2} \ldots s_{k,l}c$
Praticamente si mettono "in coda" tutte le codifiche dei δ(stato,simbolo) $\to s_{stato,simbolo}$, separando con c il "cambio" dello stato.
$\rho(M) = c$ <codifica di s> c <codifiche dello stato 1 per simbolo> $c \ldots c$ <codifiche dello stato k per simbolo> c

$\rho(M(w)) = \rho(M)\tau(w)$ con $\tau(\sigma'_0, \ldots, \sigma'_n) = cK(\sigma'_0)cK(\sigma'_1)c \ldots cK(\sigma'_n)c$

Esempio $\rho(\overline{M}) = c|^4c\ \ c|^4c|^5c|c|^9c|^3c\ \ c|^4c|^7c|^4c|^5c|^2c\ \ c|^4c|^9cdcdcdc\ \ c$

Dato che $\exists\, z \mid \forall\, i, x$ si ha $\phi_z(i, x) = \phi_i(x)$, allora vogliamo che la MdTU si comporta esattamente come $\overline{M}$ quando codifica $\overline{M}$. Quindi

$(s, \underline{\triangleright} w) \longrightarrow^M_n (h, u\underline{a}v) \Rightarrow (s_U, \underline{\triangleright}\rho(M)\tau(w)) \longrightarrow^U_m (h, \tau(uav)\underline{\#})$

$(s_U, \underline{\triangleright}\rho(M)\tau(w)) \longrightarrow^U_n (h, u'a'v')$ dovrebbe succedere che $a' = \#$ e $v' = \epsilon$, ma anche che $u' = \tau(uav)$. Se succede $\Rightarrow (s, \underline{\triangleright} w) \longrightarrow^M_m (h, uav)$