Quattro Passi con Cantor, Gödel & Turing

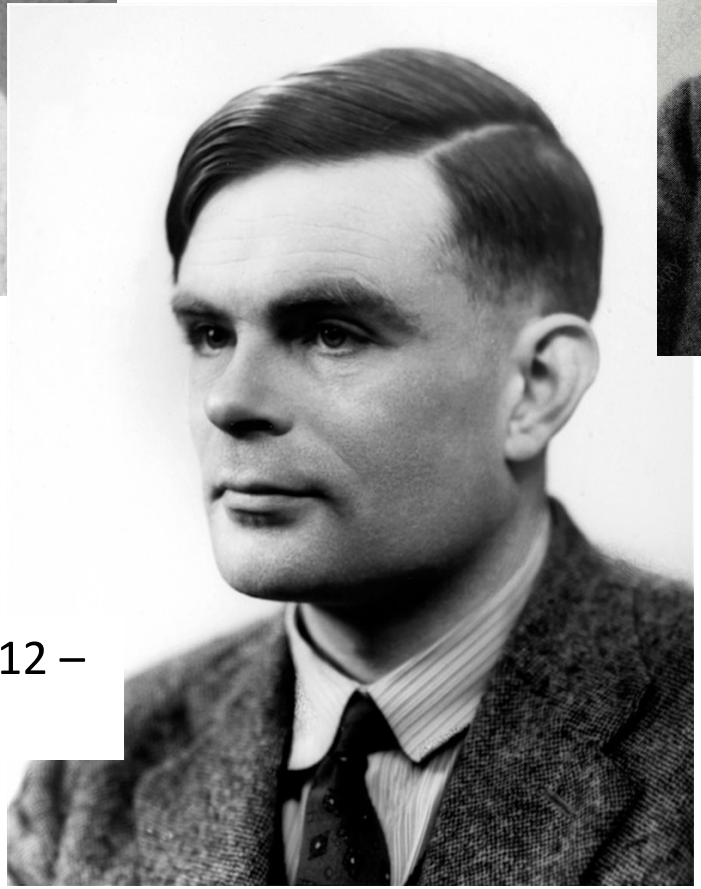
corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **20** [**1/12/23**]

SAPIENZA
UNIVERSITÀ DI ROMA

Kurt Godel (Brno 1909 – Princeton 1978)



Alan Turing (Londra 1912 – Manchester 1954)

George Cantor (San Pietroburgo 1845 – Halle 1918)

Codifiche (computabili) di Strutture Dati

corso di laurea in **Matematica**

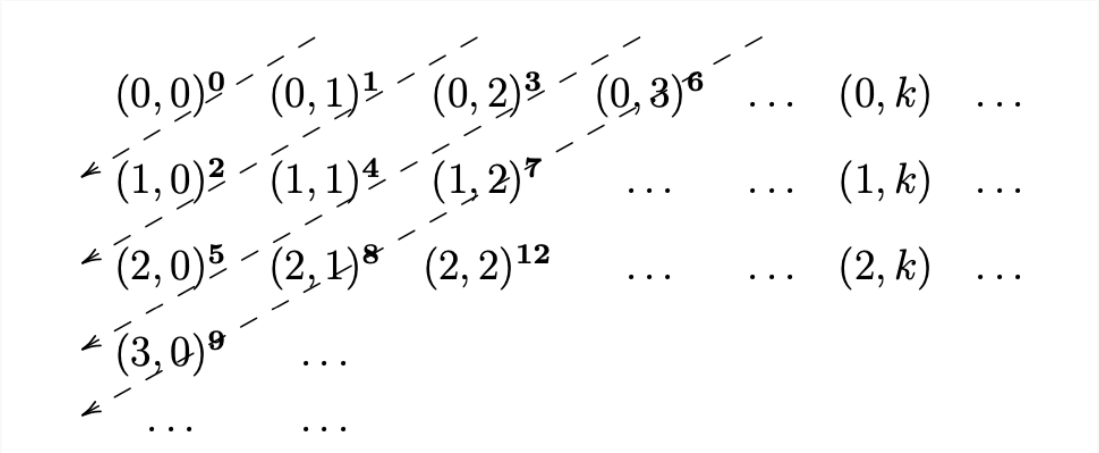
Informatica Generale, **Ivano Salvo**

Lezione **20(a)** **[1/12/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Codifica (computabile) delle coppie

Sicuramente conoscete la "codifica **a coda di rondine**" dei numeri razionali per dimostrare che $|\mathbb{N}| = |\mathbb{Q}|$.



Si può trovare facilmente una formula "chiusa" che effettua la codifica: la coppia (m, n) viene dopo $m + n$ diagonali, lunghe rispettivamente $1, 2, \ldots, m + n$ e poi devo fare n-m passi per arrivare a (m, n) sulla diagonale numero $m+n$. Risultato: $\frac{(m+n)(m+n+1)}{2} + n - m$.

Dato un naturale, basta invertire la formula per trovare la corrispondente coppia (m, n) corrispondente 😉 oppure si scrive un programma che "scorre" le diagonali… (⏵ **Esercizio**).

Codifiche computabili dei Razionali

Spesso, l'equipotenza di $\mathbb{N}$ e $\mathbb{Q}$ viene poi **dimostrata indirettamente**, facendo appello al bellissimo (e altamente non costruttivo) **Teorema di Cantor/Bernstein/Schröder** che dimostra come dati due insiemi A e B e due funzioni iniettive, $f : A \to B$ e $g : B \to A$ con uno smodato uso dell'assioma della scelta l'esistenza di una biiezione $h : A \leftrightarrow B$.

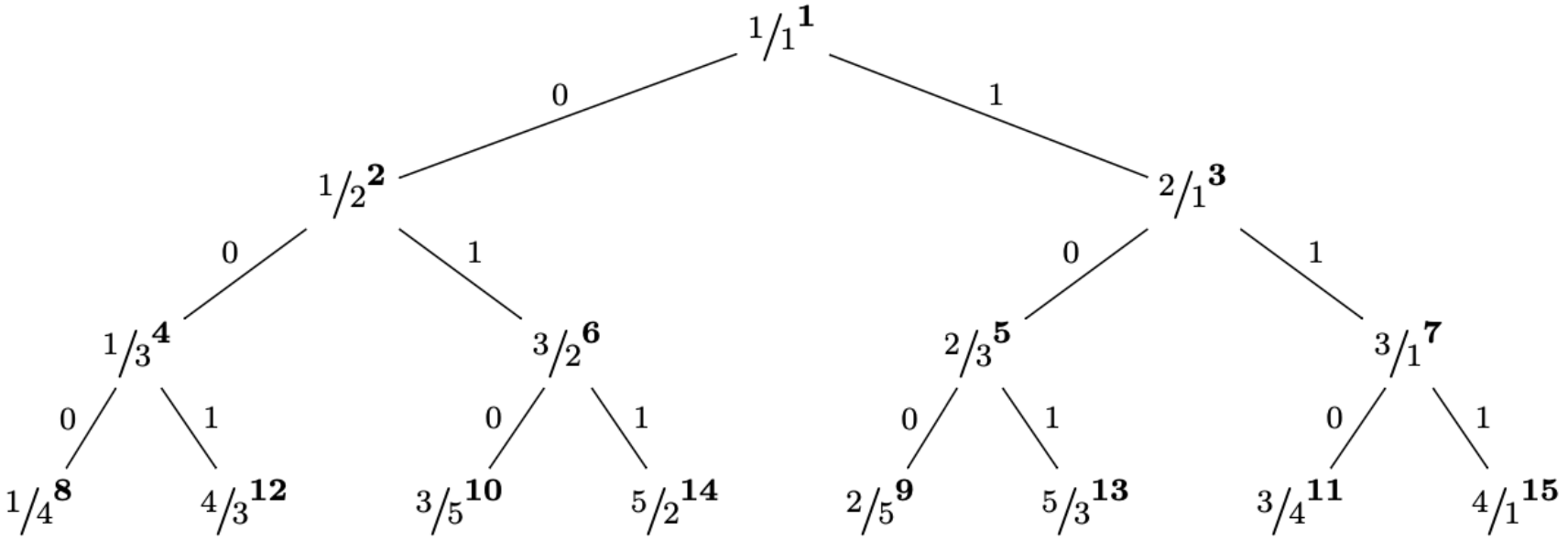
Ma noi siamo **informatici** (almeno per **oggi**) e gradiremo una **codifica diretta** e **computabile**.

Ci sono vari modi di fare questa codifica, ma uno particolarmente affascinante si ottiene dalla seguente osservazione:

*Ogni razionale è in corrispondenza biunivoca con le **computazioni** dell' algoritmo di **Euclide/Dijkstra*** (vedi Lez. **3**), che esegue le **stesse computazioni** su tutte le **coppie equivalenti** come razionali.

D'altra parte, le **computazioni** di tale algoritmo possono essere viste come **sequenze finite di 0** (mcd(m, n) = mcd(m-n, n)) **e 1** (mcd(m, n)= mcd(n-m, m)) codificabili come numeri naturali.

L'albero dei Razionali (Calkin-Wilf)



L'albero ha una facile idea di costruzione: se il **padre codifica** il razionale m/n, il **figlio sinistro codifica** $m/m+n$ e **il destro** $m+n/n$.

Qui vediamo i primi 4 livelli. **Numeriamo i razionali per livelli**: il primo ha 1 nodo, il secondo 2, il terzo 4, …, l'n-ensimo 2^{n-1}. La sequenza binaria delle computazioni di MCD ci dà un naturale tra 0 e 2^{n-1}: ma ho già "sistemato" 2^n-1 elementi nei livelli precedenti…

Se la sequenza mcd(m, n) rappresenta k ed è lunga l, rappresento m/n con il naturale 2^l-1+k+1=2^l+k.

Codifica delle tuple

Abbiamo codificato le coppie di naturali, e immaginiamo di avere le funzioni di codifica $\pi : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ e $\pi_1 : \mathbb{N} \to \mathbb{N}$ (prima componente della coppia codificata da n) e $\pi_2 : \mathbb{N} \to \mathbb{N}$ (seconda componente della coppia codificata da n). Ovviamente $\pi(\pi_1(n), \pi_2(n)) = n$.

Posso codificare le triple? Beh, facile: $\pi^3(n_1, n_2, n_3) = \pi(n_1, \pi(n_2, n_3))$ che avrà come funzioni inverse ${\pi_1}^3 = \pi_1$, ${\pi_2}^3 = \pi_1 \circ \pi_2$ e ${\pi_3}^3 = \pi_2 \circ \pi_2$.

E così via… senza scomodare difficili formule…

Osservate che queste codifiche sono biiettive con $\mathbb{N}$: di conseguenza lo **stesso numero naturale codifica una certa coppia** se lo interpreto come codifica delle coppie **e una terna** (a priori completamente **diversa**) se lo interpreto come codifica delle triple.

Quindi la **decodifica necessita di sapere** che **tipo di informazione** ho codificato. Questo è **esattamente** ciò che succede nei linguaggi di programmazione, in cui il **tipo di una variabile influenza l'interpretazione** della **sequenza di bit** trovata in memoria.

Codifica delle sequenze

Ma se volessi **codificare** tutte le **sequenze finite** di numeri naturali comunque lunghe? Nessun problema… basta sfruttare il fatto che ogni numero ha **un'unica scomposizione in fattori primi**.

Codifichiamo una **sequenza finita** $[n_1, n_2, n_3, \ldots, n_k]$ $(\forall i.\ n_i > 0)$ con il numero naturale $p_1^{n_1} \cdot p_2^{n_2} \cdot p_3^{n_3} \cdot \ldots \cdot p_k^{n_k}$, dove p_i è l'i-esimo **numero primo**. Se voglio codificare sequenze di naturali (0 compreso) basta comporla con la biezione $f : \mathbb{N} \leftrightarrow \mathbb{N} \setminus \{0\}$ definita da $f(n) = n+1$.

Per la **decodifica**, è sufficiente calcolare la **scomposizione in fattori primi** e calcolare con quale esponente appare ciascun primo.

◗ **Esercizio**: adattate l'algoritmo che calcola il massimo fattore primo allo scopo (Lezione **3**). Assumere di avere un "**vettore abbastanza grande**" oppure usate una **lista** (che avete visto in C).

Viceversa le **sequenze infinite** di naturali **non sono numerabili**.

◗ **Esercizio**: adattate l'argomento diagonale di Cantor per dimostrare questo fatto. Oppure trovate una biiezione con $\mathbb{R}$, o più facile trovate una iniezione di $\mathbb{R}$ nelle sequenze infinite di naturali.

La Funzione Universale (di un Universo di Funzioni)

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **20(b)** **[1/12/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Programmi: sequenze finite di istruz.

Un **programma** è semplicemente una **sequenza finita di istruzioni**.

Se **numero le istruzioni**, poi posso **numerare i programmi** come sequenze finite, come visto in precedenza.

In questo contesto teorico, conviene usare un **frammento minimale** sufficiente a definire tutte le funzioni computabili.

Noi abbiamo `TinyPython`, ma possiamo usare linguaggi più semplici. Si usano di solito le **Macchine di Turing (MdT)** o le **Macchine a Registri (MaR)**. Noi useremo le seconde che sono più simili a `TinyPython` e alla pratica della programmazione.

Le **variabili** (= sequenze finite di caratteri, chiamate spesso **registri** in questo formalismo) si possono facilmente numerare. Chiamiamole direttamente x_1, x_2, …, x_n, …: ogni **singolo programma** ne nomina un **numero finito**, ma ho bisogno di un numero arbitrariamente grande.

Macchine a Registri

Un programma di una Macchina a Registri (MaR) è una sequenza di istruzioni che possono essere di soli 4 tipi:

x_i = x_j (**assegnazione**: scrivo nella variabile **x_j** nella variabile **x_i**)

succ(x_i) (**successore**: incremento di 1 il contenuto della variabile **x_i**)

zero(x_i) (**azzeramento** della variabile **x_i**)

jeq x_i x_j n (**salto condizionato**: salto all'istruzione n, se $x_i = x_j$)

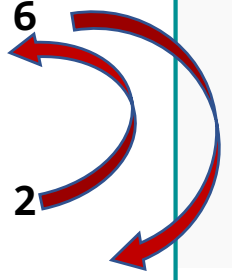
Ecco un programma MaR (si assume l'**input pronto** in $x_1, \ldots, x_k$)

➧ **Esercizio**: Provate a "tradurre" le "istruzioni complesse" di TinyPython possono essere tradotte in un programma per una MaR: ad esempio **while** B: C oppure x = exp, oppure **if** exp_1 == exp_2 **then** C_1 **else** C_2.

Applicare la vostra "traduzione" al **programma della somma** e verificare se è più o meno semplice di quello scritto a lato.

Curiosamente, **è più difficile vedere il contrario** (TinyPython calcola come MaR)!

```
# x1 = m, x2 = n
1: zero(x3)
2: jeq x3 x2 6
3: succ(x3)
4: succ(x1)
5: jeq x2 x2 2
6: x0 = x1
```



Nozione di Computabilità per MaR

Definizione: Una **funzione parziale** $f : \mathbb{N}^k \to \mathbb{N}$ è una funzione da $\mathbb{N}^k \to \mathbb{N} \cup \{\uparrow\}$. Il "valore" $\uparrow$ serve per modellare computazioni che **non terminano** o **falliscono** (divisioni per zero etc.).

Definizione: Una funzione parziale $f : \mathbb{N}^k \to \mathbb{N}$ è **computabile** se **esiste un programma** P tale che per **ogni** k**-upla** di numeri naturali $n_1, n_2, \ldots, n_k$, cominciando la computazione coi valori $n_1, n_2, \ldots, n_k$ memorizzati nelle variabili $x_1, \ldots, x_k$:

- $f(n_1, n_2, \ldots, n_k) = n \in \mathbb{N}$ allora P **termina** il valore n memorizzato nella variabile x_0;
- $f(n_1, n_2, \ldots, n_k) = \uparrow$ se e solo se P **non termina**.

Siccome posso numerare le k-uple di naturali, possiamo sempre considerare le funzioni $f : \mathbb{N} \to \mathbb{N}$.

Proposizione: $f : \mathbb{N}^k \to \mathbb{N}$ **è computabile** se e solo se è esiste una funzione **computabile unaria** $g : \mathbb{N} \to \mathbb{N}$ tale che:

$$g(n) = f(\pi_1^k(m), \pi_2^k(m), \ldots, \pi_k^k(m)) \text{ dove } m = \pi_1(n) \text{ e } k = \pi_2(n)$$

[per applicare correttamente le codifiche devo conoscere k]

Numerazione dei Programmi

Per numerare i programmi MaR, ci rimane il problema di numerare le istruzioni (avendo già numeratole sequenze). Vediamo.

$$C(\texttt{x}_i \texttt{ = } \texttt{x}_j) = 4\,\pi(i, j)$$

$$C(\texttt{succ(x}_i\texttt{)}) = 4i + 1$$

$$C(\texttt{zero(x}_i\texttt{)}) = 4i + 2$$

$$C(\texttt{jeq } \texttt{x}_i \texttt{ x}_j \texttt{ n}) = 4\,\pi^3(i, j, n) + 3$$

Il prodotto per 4 serve per discrimare nella decodifica il tipo di istruzione: basta prendere il resto della divisione per 4 per sapere quali delle 4 forme si tratti e quindi sapere quali altre funzioni inverse applicare.

Siccome ho numerato i programmi, ho in realtà **numerato** anche tutte le **funzioni computabili**.

Indichiamo con φ_n la funzione calcolata dal programma P_n e con $\varphi_n(m)$ il valore della funzione φ_n su m.

Osservate che **ogni funzione** ha sempre **una infinità** di **indici diversi**: basta ad esempio aggiungere istruzioni inutili 🙃 ma anche due algoritmi diversi originano due programmi diversi.

funz. di programmi e funz. universale

Avendo numerato i programmi con numeri naturali, una funzione $f : \mathbb{N}^k \to \mathbb{N}$ può essere vista anche come una funzione che ha **come argomenti programmi** (o le funzioni da essi calcolate).

Una **vera funzione di funzioni computabili** deve però avere la seguente proprietà: $f(n_1, n_2, \ldots, n_k) = f(n'_1, n'_2, \ldots, n'_k)$ per ogni coppia di k-uple per cui $\varphi_{n_1} = \varphi_{n'_1}$, $\varphi_{n_2} = \varphi_{n'_2}$, …, $\varphi_{n_k} = \varphi_{n'_k}$.

Vediamo la **funzione universale $\mathcal{U}$** (**binaria**, ma con poco sforzo si può fare quella k-aria) cioè la funzione che incorpora tutte le funzioni computabili (**unarie**) nel seguente senso:

$$\mathcal{U}(n, m) = \varphi_n(m)$$

cioè **interpreta** il **primo parametro** come la **codifica di un programma** (o di una funzione computabile) e lo **esegue sull'input** m, cioè il suo **secondo parametro**.

Vediamo subito che **$\mathcal{U}$ è computabile!** (la scriviamo in `TinyPython`).

Funzione universale

```
def funUniv(x, y):
    L, pc = lunghezza(x), 0
    while pc < L: #supero l'ultima istr.
        k = decodIstrPC(x, pc)
        q, t = div(k, 4) # tipo istr.
        if t == 0: # assegnazione
            i, j = π1(t), π2(t)
            v[i], pc = v[j], pc+1
        else if t == 1: # successore
            v[q], pc = v[q] + 1, pc+1
        else if t == 2: # azzeramento
            v[q], pc = 0, pc+1
        else: # salto condizionato
            i, j, l = π3_1(q), π3_2(q), π3_3(q)
            if v[i]=v[j]: pc=l
                else: pc = pc+1
    return v[0] # valore in x0 fine esecuz.
```

pc è il **program counter** e mantiene l'indice della prossima istruzione da eseguire.

Aiutandosi con un **array** di istruzioni posso fare **un'unica decodifica**.

Questo è piuttosto un **interprete** che ogni volta deve tradurre la prossima istruzione da eseguire.

Il vettore v serve a tenere i valori delle variabili del programma P_x che viene eseguito.

Conseguenze pratiche e concettuali

L'esistenza della funzione universale dimostra che si possono costruire **macchine programmabili** generali.

Non solo, **dati** e **programmi** possono risiedere nella **stessa memoria**, come **spiegò Alan Turing** a **John von Neumann** intento nella progettazione del primo calcolatore "moderno", l'ENIAC (magari, come accade, con **codifiche meno arcane**).

La funzione universale dimostra che si possono scrivere i **compilatori**, ma più in generale che si possono scrivere **programmi che "analizzano" altri programmi**.

Il fatto che la **funzione universale** si possa calcolare con un **programma `while`** (occorrerebbe dimostrare ovviamente che o sono anche **tutte le codifiche e decodifiche necessarie** … laborioso, ma non difficile: ripensate alla funzione **`massimoPrimo`** della Lezione **3**) permette di dimostrare il **Teorema di Böhm-Jacopini**…

… e tantissimi importanti e affascinanti risultati di Informatica Teorica (**computabilità**), di cui fra poco vedremo il più classico.

Sentieri Diagonali

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **20(c)** **[1/12/23]**

SAPIENZA
UNIVERSITÀ DI ROMA

Funzioni non computabili

Abbiamo visto che i **programmi sono numerabili**.

Quindi le **funzioni calcolabili sono** necessariamente **numerabili**.

Per contro, la classe delle funzioni del tipo $\mathbb{N}^k \to \mathbb{N}$ (ma anche quelle in $\mathbb{N}^k \to \{0, 1\}$, i cosiddetti **problemi decisionali**) sono evidentemente **non numerabili**.

Infatti, le funzioni in $\mathbb{N} \to \{0, 1\}$ si possono facilmente mettere in corrispondenza biunivoca con $\mathcal{P}(\mathbb{N})$ e **nessun insieme** può essere messo in **corrispondenza biunivoca** con il suo **insieme potenza**.

Infine è abbastanza evidente che: $|\mathbb{N} \to \{0,1\}| \leq |\mathbb{N}^k \to \{0,1\}| \leq |\mathbb{N}^k \to \mathbb{N}|$.

Sorge spontanea una domanda: **esistono funzioni interessanti non computabili**?

O sono **non computabili** solo oscure funzioni di nessun interesse pratico **costruite per diagonalizzazione** (à la Cantor)?

Tesi di Church e Halting Problem

Nell'anno mirabilis dell'Informatica primordiale, il **1936**, Alonzo Church enuncia la seguente tesi (**indimostrabile per costruzione!**):

Tesi di Church: *Tutte le funzioni computabili sono λ-definibili.*

che potete leggere col vostro modello di calcolo preferito (MaR, Macchine di Turing, TinyPython): è indimostrabile perché ciò che è computabile è esso stesso definito attraverso un modello di calcolo.

★ ★ ★

Consideriamo il seguente problema (di enorme interesse pratico) noto come **Halting Problem** (o **Problema della Fermata**).

È possibile costruire un programma P che prende **input** un **programma** P', un input x e risponde 1 se P' termina sull'input x e 0 altrimenti?

Avendo numerato le funzioni computabili, questo corrisponde a chiedersi **se la seguente funzione** $h : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$:

$$h(m, n) = \begin{cases} 1 & se\ \varphi_m(n)\ termina \\ 0 & se\ \varphi_m(n)\ \boldsymbol{non}\ termina \end{cases}$$

sia computabile

Indecidibilità dell'Halting problem

Teorema: *L'Halting Problem* ***non è computabile***.

Dimostrazione: Assumiamo che esista una funzione computabile h che risolve l'Halting problem.

Allora, **è computabile** la funzione **`burlona`** (vedi pseudocodice in calce). La funzione **`burlona`** **termina sull'input n** se e solo se la funzione **φ_n non termina sull'input n** ($\varphi_n(n)\uparrow$).

Siccome la funzione **`burlona`** è computabile, **allora ha un indice** nella **numerazione delle funzioni computabili** e quindi è **φ_{n_0}** per un qualche naturale n_0.

Ma cosa accade se passiamo il valore n_0 alla funzione **`burlona`**?

`burlona` termina sull'input n_0 se e solo se $h(n_0, n_0) = 0$, cioè **se e solo se** $\varphi_{n_0}(n_0)$ non termina, cioè se e solo se **`burlona` non termina** su n_0.

Assurdo. Quindi ***h* non è computabile**. □

```
def burLona(x):
    while h(x, x)==1:
        x = x
    return 1
```

Più in generale…

Più in generale sono **non è computabile nessuna proprietà non banale** dei programmi (**Teorema di Rice**), ad esempio:

- φ_n e φ_m sono la stessa funzione. Cioè il programma P_n è equivalente al programma P_m
- φ_n è totale. Cioè il programma P_n termina su ogni input
- φ_n è costante
- φ_n produce un numero finito/infinito di output distinti
- … (e chi più ne ha più ne metta!)

Attenzione! ciò non significa che **non sia possibile dimostrare l'equivalenza** tra due programmi o la **terminazione** (come peraltro abbiamo fatto spesso, più o meno informalmente).

Non **esiste un programma** che lo riesca a fare in modo sistematico!

Uno sguardo oltre…

La tecnica usata ha un illustre precedente: i **teoremi di incompletezza** di Gödel.

Bertrand Russell e Norman Whitehead avevano individuato l'origine dei paradossi e credevano di poterli eliminare dalla matematica **stratificando le proposizioni** (tipi logici).

Un enunciato paradossale, come il **paradosso del mentitore**: "sto mentendo" si origina perché la proposizione contiene **un enunciato sulla proposizione stessa** (**auto-referenza**).

Kurt Gödel per primo usa la tecnica della numerazione (detta infatti anche goedelizzazione) per far vedere che in una **teoria logica abbastanza forte** da contenere l'**aritmetica**, si possono definire sempre enunciati auto-referenziali perché una **proposizione sui numeri naturali** è anche una **proposizione di meta-teoria**.

Di conseguenza, una teoria logica **è necessariamente incompleta** (ci sono enunciati non dimostrabili, perché sia l'**enunciato** che la sua **negazione** sono **contradditori**) o **inconsistente** (tutti gli enunciati sono teoremi, e quindi, di fatto, non dimostra nulla).

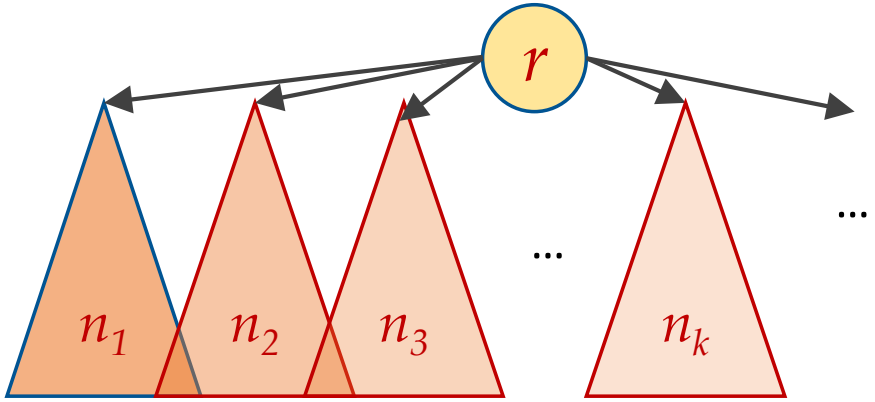
Un paradosso per finire…

Conoscete probabilmente tutti il **paradosso di Russell**: "l'insieme di tutti gli insiemi che non contengono sé stessi, contiene sé stesso"?

e forse quello del **barbiere**: "In un villaggio c'è un unico barbiere che rade tutti i cittadini che non si radono da soli.
Il barbiere si rade da solo?" [pare dovuto a Russell anche questo].

Un **albero** necessariamente **infinito** è **ripetitivo** se c'è ha tra i suoi sottoalberi sé stesso. Sono ripetitivi per esempio l'albero filiforme. Oppure l'albero n-ario completo (ogni nodo ha sempre n figli).

Costruiamo ora un albero con una radice e come sottoalberi tutti gli alberi non-ripetitivi. **È ripetitivo**?



That's all Folks!

corso di laurea in **Matematica**

Informatica Generale, **Ivano Salvo**

Lezione **20** [**1/12/23**]

SAPIENZA
UNIVERSITÀ DI ROMA