Domande Informatica Teorica

1. Cos'è un problema di decisione?

   Solution: Un problema di decisione è una domanda a cui rispondere *Sì* o *No*.
   Formalmente, è costituito da 3 elementi:

   - Nome del problema
   - Istanza degli oggetti considerati
   - Domanda, ovvero proprietà che gli oggetti possono o meno soddisfare

2. Come dimostrare l'esistenza di problemi non decidibili, senza mostrarne un esempio?

   Solution: Per studiare la decidibilità di un problema Π ci sono due possibili approcci:

   - Trovare un programma P_Π che calcola la funzione soluzione

     $$\Phi_\Pi : D \to \{0,1\} \quad \text{t.c.} \quad \Phi_\Pi(x) = \begin{cases} 1 & \text{se } p(x) \\ 0 & \text{se } \neg p(x) \end{cases}$$

     di conseguenza $\Phi_\Pi \in \mathcal{T}$
   - Se $\Phi_\Pi \in \mathcal{T}$, allora esiste un programma che la calcola

   Di conseguenza, i problemi risolvibili PROG sono isomorfi alle funzioni calcolabili, quindi numerabili, mentre tutti i possibili problemi sono rappresentati dalle funzioni da $\mathbb{N}$ a $\mathbb{N}$ (dato quanto già visto), isomorfe a $\mathbb{N}_\perp^\mathbb{N}$, quindi:

   $$\text{DATI} \sim \text{PROG} \sim \mathbb{N} \nsim \mathbb{N}_\perp^\mathbb{N}$$

   Per dimostrare che DATI $\sim \mathbb{N}$: serve una funzione tale che permetta una biezione tra dati e $\mathbb{N}$, come la funzione coppia di Cantor

   $$\langle \_, \_ \rangle : \mathbb{N} \times \mathbb{N} \to \mathbb{N}^+$$

   (banalmente estendibile a tutto $\mathbb{N}$).

   Per dimostrare che PROG $\sim \mathbb{N}$: una volta definito un sistema di calcolo e i relativi comandi, si può mostrare una codifica per questi che porta a una biezione con $\mathbb{N}$.

   Alternativamente, un problema di decisione può essere comparato al riconoscimento di un linguaggio $L \subseteq \Sigma^*$, per un alfabeto Σ di conseguenza:

- Il numero possibile di linguaggi è $P(\Sigma^*) \sim \mathbb{R}$
- I sistemi di calcolo sono in quantità PROG $\sim \mathbb{N}$

Per forza devono esistere linguaggi non decidibili.

3. Il problema dell'arresto: definizione e dimostrazione.

Solution: Definizione del problema dell'arresto:

- Nome: AR
- Istanza: $x, y \in \mathbb{N}$
- Domanda: $\varphi_y(x) \downarrow$?

Teorema 0.1. *AR è indecidibile.*

Dimostrazione. Assumiamo per assurdo che AR sia decidibile, allora esiste una funzione soluzione

$$\Phi_{\text{AR}}(x, y) = \begin{cases} 0 & \text{se } \varphi_y(x) \uparrow \\ 1 & \text{se } \varphi_y(x) \downarrow \end{cases}$$

Valutando il caso in cui $x = y$

$$\Phi_{\text{AR}}(x, x) = \begin{cases} 0 & \text{se } \varphi_x(x) \uparrow \\ 1 & \text{se } \varphi_x(x) \downarrow \end{cases}$$

Visto che $\Phi_{\text{AR}} \in \mathcal{T}$, anche la funzione

$$f(x) = \begin{cases} 0 & \text{se } \Phi_{\text{AR}}(x) = 0 \equiv \varphi_x(x) \uparrow \\ \varphi_x(x) + 1 & \text{se } \Phi_{\text{AR}}(x) = 1 \equiv \varphi_x(x) \downarrow \end{cases} \in \mathcal{T}$$

Sia $\alpha \in \mathbb{N}$ la codifica di A tale che $\varphi_\alpha = f$. Valutiamo φ_α in α:

$$\varphi_\alpha(\alpha) = \begin{cases} 0 & \text{se } \varphi_\alpha(\alpha) \uparrow \\ \varphi_\alpha(\alpha) + 1 & \text{se } \varphi_\alpha(\alpha) \downarrow \end{cases}$$

Ma tale funzione non può esistere:

- Nel primo caso $\varphi_\alpha(\alpha) = 0$ se $\varphi_\alpha(\alpha) \uparrow$, ma è una contraddizione
- Nel secondo caso $\varphi_\alpha(\alpha) = \varphi_\alpha(\alpha) + 1$, ma tale relazione non vale per nessun naturale

Siamo a un assurdo, AR è indecidibile.

□

4. Sistemi di calcolo visti in Teoria della Calcolabilità.

Solution: I sistemi di calcolo visti sono:

- Sistema RAM: infiniti registri, R_0 contiene l'output, R_1 l'input, si ha un program counter L, le istruzioni sono
  - Incremento: $R_k \leftarrow R_k + 1$
  - Decremento: $R_k \leftarrow R_k \dot{-} 1$
  - Salto condizionato: `if` $R_k = 0$ `goto` m, con $m \in \{1, \ldots, |P|\}$
- Sistema WHILE: 21 registri, x_0 output, x_1 input, sono presenti dei comandi base:
  - $x_k := x_j + 1$
  - $x_k := x_j \dot{-} 1$
  - $x_k := 0$

  e dei comandi definiti induttivamente:
  - Comando composto

    `begin` $C_1, \ldots, C_n$ `end`

    dove ogni C_i è un qualsiasi comando
  - Comando while

    `while` $x_k \neq 0$ `do` C

    dove C è un qualsiasi comando

  Di conseguenza, un programma WHILE è un comando composto

5. Approfondimento su RAM (struttura, istruzioni, stato prossimo, computazione)

Solution: Il sistema RAM permette infiniti registri, tra i quali R_0 contiene l'output e R_1 l'input, si ha inoltre un program counter L per tenere traccia dell'istruzione da eseguire. Un programma P è un insieme ordinato di istruzioni.

Le istruzioni sono:

- Incremento: $R_k \leftarrow R_k + 1$
- Decremento: $R_k \leftarrow R_k \dot{-} 1$

- Salto condizionato: `if` $R_k = 0$ `goto` m, con $m \in \{1, \ldots, |P|\}$

Ogni istruzione fa passare la macchina da uno stato a un altro; la semantica operazione di un'istruzione è formata dalla coppia degli stati prima e dopo l'istruzione.

La computazione del programma P è una sequenza di stati $\mathcal{S}_i$, infinita se non termina, altrimenti si ha uno stato finale $\mathcal{S}_{fin}$ in cui viene posto in R_0 il risultato della computazione.

Lo stato è una funzione

$$\mathcal{S} : \{L, R_i\} \to \mathbb{N}$$

ovvero, che dato un registro e un valore del program counter L, restituisce il contenuto del registro.

Uno stato finale $\mathcal{S}_{fin}$ è un qualsiasi stato tale che $\mathcal{S}(L) = 0$.

Lo stato iniziale è tale che

$$\mathcal{S}_{init}(R_i) = \begin{cases} 1 & \text{se } R_i = L \\ n & \text{se } R_i = R_1 \\ 0 & \text{altrimenti} \end{cases}$$

Per definire l'esecuzione del programma si usa la funzione stato prossimo

$$\delta : \text{STATI} \times \text{PROG} \to \text{STATI}_\perp$$

tale che

$$\delta(\mathcal{S}, P) = \mathcal{S}'$$

Dove $\mathcal{S}$ rappresenta lo stato in seguito all'esecuzione del comando P.

La funzione è tale che:

- Se $\mathcal{S}(L) = 0$, $\mathcal{S}' = \perp$ in quanto l'esecuzione è terminata
- Se $\mathcal{S}(L) > |P|$ non si ha una terminazione esplicita, quindi

$$\mathcal{S}'(R) = \begin{cases} 0 & \text{se } R = L \\ \mathcal{S}(R_i) & \text{altrimenti} \end{cases}$$

- Se $1 \leq \mathcal{S}(L) \leq |P|$, si considera l'istruzione $\mathcal{S}(L)$-esima:
  - incremento/decremento su R_k:

$$\mathcal{S}'(R) = \begin{cases} \mathcal{S}(R) + 1 & \text{se } R = L \\ \mathcal{S}(R) \pm 1 & \text{se } R = R_k \\ \mathcal{S}(R) & \text{altrimenti} \end{cases}$$

– salto condizionato su R_k

$$\mathcal{S}'(R) = \begin{cases} m & \text{se } R = L \wedge R_k = 0 \\ \mathcal{S}(L) + 1 & \text{se } R = L \wedge R_k \neq 0 \\ \mathcal{S}(R) & \text{altrimenti} \end{cases}$$

L'esecuzione di un programma genera una sequenza di stati, definita secondo la funzione δ.

6. Struttura e istruzioni WHILE.

Solution: Il sistema di calcolo WHILE usa esattamente 21 registri ed è strutturato, quindi non necessita di program counter. Sono presenti istruzioni base e istruzioni definite induttivamente. Comandi base:

- $x_k := x_j + 1$
- $x_k := x_j \dot{-} 1$
- $x_k := 0$

Comandi induttivi:

- Comando composto

  $$\texttt{begin } C_1; \ldots; C_n \texttt{ end}$$

  dove ogni C_i è un qualsiasi tipo di comando
- Comando while

  $$\texttt{while } x_k \neq 0 \texttt{ do } C$$

  dove C è un qualsiasi comando

Per dimostrare proprietà di un programma $P \in W$-PROG la si verifica prima sui comandi base, poi induttivamente su comando composto e while.

Dati comando da eseguire e stato corrente, la funzione stato prossimo restituisce lo stato successivo:

$$[\![\_]\!](\_) : W\text{-COM} \times W\text{-STATI} \to W\text{-STATI}_\perp$$

Si può definire la semantica induttivamente, a partire dagli assegnamenti:

$$[\![x_k := x_j \pm 1]\!](\underline{x}) = \begin{cases} x_i & \text{se } i \neq k \\ x_j \pm 1 & \text{se } i = k \end{cases}$$

$$[\![x_k := 0]\!](\underline{x}) = \begin{cases} x_i & \text{se } i \neq k \\ 0 & \text{se } i = k \end{cases}$$

Mentre per i comandi induttivi, partendo dal comando composto:

$$[\![\texttt{begin } C_1; \ldots; C_n \texttt{ end}]\!](\underline{x}) = ([\![C_n]\!] \circ \cdots \circ [\![C_1]\!])(\underline{x})$$

dove i comandi C_i sono noti per I.H.

Per il comando while:

$$[\![\texttt{while } x_k \neq 0 \texttt{ do } C]\!](\underline{x}) = \begin{cases} [\![C]\!]^e(\underline{x}) & \text{se } \exists e \text{ t.c. } [\![C]\!]^e(\underline{x})[k] = 0 \\ \bot & \text{altrimenti} \end{cases}$$

Di conseguenza, dato che un programma while $W \in W$-PROG è un comando composto, si può rappresentare la semantica del programma come

$$\Psi_W(n) = \text{Pro}_0^{21}([\![W]\!](w\text{-}in(n)))$$

dove $w\text{-}in(n)$ restituisce lo stato iniziale della macchina con input n.

7. Relazione tra RAM e WHILE, con dimostrazione che il secondo è contenuto nel primo e viceversa.

Solution: Si può dimostrare che le funzioni calcolabili dai due sistemi sono le stesse, ovvero che $F(\text{RAM}) = F(\text{WHILE})$. Per dimostrarlo si deve mostrare che $F(\text{WHILE}) \subseteq F(\text{RAM})$ e $F(\text{RAM}) \subseteq F(\text{WHILE})$.

Dati due sistemi di calcolo $\mathcal{C}_1$ e $\mathcal{C}_2$, si definisce traduzione una funzione che associa i programmi di uno ai programmi dell'altro, con le proprietà di programmabilità, correttezza e completezza, rispetto ai programmi presenti nei due insiemi. Se esiste un traduttore da $\mathcal{C}_1$ a $\mathcal{C}_2$ allora $F(\mathcal{C}_1) \subseteq F(\mathcal{C}_2)$, dato che per ogni programma P_1 del primo sistema di calcolo esiste (per completezza) $P_2 = t(P_1)$ nel secondo con la stessa semantica (per correttezza).

Per dimostrare che $F(\text{WHILE}) \subseteq F(\text{RAM})$, costruiamo un compilatore $C : W\text{-PROG} \to$ PROG. Si può definire induttivamente il compilatore, in quanto WHILE è definito induttivamente. I 21 registri vengono mappati nei primi 21 RAM, dato che sono infiniti non ci sono problemi.

Passo base: gli assegnamenti

- $x_k := 0$, si itera sul registro da azzerare finché è diverso da 0 (usando $R_{21} = 0$ come condizione di coda del ciclo)
- $x_k = x_j \pm 1$, se $k = j$ è banale, altrimenti bisogna
  - Spostare il valore di R_j in R_{22}, azzerandolo
  - Azzerare R_k
  - Rigenerare R_j e R_k a partire da R_{22}
  - Effettuare l'incremento/decremento effettivo

Passo induttivo:

- Il comando composto è una composizione di comandi noti per I.H.
- Il comando while si può compilare come un ciclo che esegue il comando, noto per I.H., finché $R_k \neq 0$

Il compilatore così definito è programmabile, compila ogni $W \in W$-PROG e mantiene la semantica, di conseguenza

$$F(\text{WHILE}) \subseteq F(\text{RAM})$$

Per mostrare che $F(\text{RAM}) \subseteq F(\text{WHILE})$ vogliamo scrivere un interprete I_W che prende in input un programma $P \in \text{PROG}$ e $x \in \mathbb{N}$ e restituisce l'esecuzione di P su x, ovvero $\varphi_P(x)$. Non crea nulla di intermedio, si limita a eseguire il programma.

Dovendo prendere due input (P e x), bisogna condensarli tramite Cantor, quindi, con $n = cod(P)$, l'input diventa $\langle x, n \rangle$. Di conseguenza:

$$\forall x, n \in \mathbb{N},\ \Psi_{I_W}(\langle x, n \rangle) = \varphi_n(x) = \varphi_P(x)$$

Come salvare lo stato della macchina nell'interprete? Un problema sono i registri, in RAM possono essere infiniti, ma non ne verranno mai usati veramente infiniti, quindi il contenuto di questi può essere rappresentato tramite una lista $\langle R_0, \ldots, R_n \rangle$.

L'interprete I_W salva lo stato della macchina nel modo:

- $x_0 \leftarrow \langle R_0, \ldots, R_n \rangle$
- $x_1 \leftarrow L$
- $x_2 \leftarrow y$, dato su cui lavora P

- $x_3 \leftarrow n$, "listato" del codice
- $x_4 \leftarrow I$, istruzione attuale, prelevata dal listato grazie al PC

Inizialmente l'input $\langle x, n \rangle$ si trova in x_1. L'interprete deve quindi solamente estrarre, nell'ordine stabilito dal PC, le istruzioni dal listato, decodificare ed eseguirle, finché il PC è diverso da zero, ovvero $x_1 \neq 0$.

Avendo l'interprete, si può costruire un compilatore $C : \text{PROG} \to W\text{-PROG}$ semplicemente chiamando l'interprete dopo aver posto in x_1 l'input $\langle x_1, x_2 \rangle$, dove x_1 contiene l'input di P e x_2 contiene $cod(P)$.

Abbiamo quindi trovato un compilatore programmabile, completo e corretto da RAM a WHILE, mostrando che

$$F(\text{RAM}) \subseteq F(\text{WHILE})$$

Di conseguenza

$$F(\text{RAM}) = F(\text{WHILE})$$

8. Differenza tra compilatore e interprete.

Solution: Un compilatore prende in input un programma e lo trasforma in un altro in un sistema di calcolo destinazione, mantenendone la semantica.

Un interprete è una funzione che prende come argomenti un programma e un input e simula l'esecuzione del programma passo passo, restituendo l'esecuzione del programma originale sull'input fornito.

9. Come passare un programma RAM all'interprete.

Solution: L'interprete accetta in input la codifica del programma (quindi un numero, dato che PROG $\sim \mathbb{N}$) e ricava l'istruzione da prelevare grazie al valore del PC, per poi decodificarla.

10. Perché è impossibile che il destro di una coppia di Cantor sia 0, a meno che non sia l'ultima istruzione codificata.

Solution: Dopo aver applicato iterativamente Cantor su una lista $\langle x_1, \langle \dots \langle x_n, 0 \rangle \dots \rangle \rangle$, quando vengono estratti i valori, come sin verrà estratto l'ultimo valore inserito, come des rimarrà la codifica di tutti gli altri valori della lista, che quindi potrà essere 0 solo nel caso in cui questa è l'ultima istruzione codificata.

11. Gerarchia classi di complessità e descrizione.

Solution: Le classi di complessità sono

$$L \subseteq P \subseteq NP \subseteq \text{EXPTIME} \subseteq \mathscr{P}$$

E sono definite come:

- $L = \text{DSPACE}(\log n)$
- $P = \text{DTIME}\left(n^k\right)$
- $NP = \text{NTIME}\left(n^k\right)$
- $\text{EXPTIME} = \text{DTIME}\left(2^{n^k}\right)$
- $\mathscr{P}$ tutte le funzioni calcolabili

12. Descrizione DTM per calcolare lo spazio.

Solution: Data la DTM $M = (Q, \Sigma, \Gamma, \delta, q_0, F)$ si definisce $S(x)$ il numero di celle occupate sul nastro di memoria durante la computazione di M su x.

La complessità in spazio viene conseguentemente definita come

$$s(x) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

ovvero come il massimo numero di celle usate per il calcolo di un input, per ogni taglia di input.

Ma in questo modo viene considerato anche lo spazio dedicato all'input, non permettendo complessità sublineari. Per risolvere, si aggiunge un altro nastro dedicato all'input, con dei terminatori $\notin \Sigma$. Si deve modificare la funzione di transizione per gestire il moto delle due testine e permettere di leggere sia da nastro di input che di lavoro.

Per ogni $x \in \Sigma^*$, $S(x)$ è ora dato solo dal numero di celle usate solamente sul nastro di lavoro, non considerando così l'interferenza dovuta all'input.

Il linguaggio $L \in \Sigma^*$ è riconosciuto in spazio deterministico $f(n)$ se e solo se esiste una DTM M tale che $L = L_M$ e $s(n) \leq f(n)$.

13. Com'è definita la computazione in una DTM.

Solution: La computazione è una sequenza di mosse (passo che, dato lo stato corrente e simbolo letto, porta a un nuovo stato, con eventuale scrittura e spostamento) dettate dalla funzione di transizione.

La funzione di transizione è definita come

$$\delta : Q \times \Gamma \to Q \times (\Gamma \setminus \{blank\}) \times \{-1, 0, 1\}$$

14. Qual è la condizione di terminazione in una DTM.

Solution: Una DTM termina quando entra in uno stato finale, ovvero con $\delta(q, \gamma) = \perp$, oppure quando la funzione di transizione non è definita (si tratta di una funzione parziale).

15. Definizione ricorsive primitive partendo da ELEM.

Solution: L'insieme ELEM contiene solo le operazioni di successore, proiettore e azzeramento. Aggiungendo l'operatore di composizione

$$\text{Comp}(h, g_1, \ldots, g_k) : \mathbb{N}^n \to \mathbb{N} \mid \text{Comp}(h, g_1, \ldots, g_k)(\underline{x}) = h(g_1(\underline{x}), \ldots, g_n(\underline{x}))$$

e l'operatore di ricorsione primitiva

$$\text{RP}(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(f(\underline{x}, y), y - 1, \underline{x}) & \text{se } y > 0 \end{cases}$$

per poi chiudere l'insieme rispetto alle due operazioni, si ottiene l'insieme RICPRIM

$$\text{ELEM}^{\{\text{Comp,RP}\}} = \text{RICPRIM}$$

16. Definizione ricorsive parziali. Come abbiamo definito le funzioni calcolabili?

Solution: A partire da RICPRIM (definito sopra), per aggiungere all'insieme delle funzioni calcolabili le ricorsive parziali si usa l'operatore di minimalizzazione:

$$\mathrm{MIN}(f)(\underline{x}) = \mu_y(f(\underline{x}, y) = 0) = \begin{cases} y & \text{se } f(\underline{x}, y) = 0 \wedge (\forall y' < y, f(\underline{x}, y) \downarrow \wedge f(\underline{x}, y') \neq 0) \\ \perp & \text{altrimenti} \end{cases}$$

Informalmente, restituisce il più piccolo valore di y che azzera $f(\underline{x}, y)$, ovunque precedentemente definita su y'. Questo operatore permette le presenza di funzioni parziali e chiudere RICPRIM rispetto alla minimalizzazione permette di ottenere

$$\mathrm{ELEM}^{\{\mathrm{Comp,RP,MIN}\}} = \mathscr{P}$$

ovvero l'insieme delle funzioni ricorsive parziali.

La tesi di Church-Turing sostiene che l'insieme delle funzioni intuitivamente calcolabili coincide con $\mathscr{P}$ (alternativamente, le funzioni calcolabili tramite DTM).

17. Come abbiamo definito la potenza computazionale $F(\mathrm{RAM})$?

Solution: La potenza computazionale del sistema RAM è definibile come

$$F(\mathrm{RAM}) = \left\{ f \in \mathbb{N}_\perp^{\mathbb{N}} \mid \exists P \in \mathrm{PROG} \text{ t.c. } \varphi_P = f \right\} = \{\varphi_P \mid P \in \mathrm{PROG}\}$$

18. RICPRIM = $F(\mathrm{FOR})$

Solution: Tutte le funzioni in RICPRIM hanno inizio e fine ben definiti, sono tutte funzioni totali (ma non rappresenta tutte le funzioni totali), quindi sulla base di questo si può definire il costrutto FOR che si serve di una variabile di controllo che parte da un valore ed arriva a un valore limite.

Le funzioni calcolabili tramite FOR sono funzioni totali con numero di iterazioni noto, esattamente come per RICPRIM.

19. Relazione tra $\mathscr{P}$ e $F(\mathrm{WHILE})$.

Solution: Per confrontare $\mathscr{P}$ e $F(\mathrm{WHILE})$ si può cominciare dal domandarsi se $\mathscr{P} \subseteq F(\mathrm{WHILE})$. L'insieme $\mathscr{P}$ può essere visto come definito induttivamente, ovvero

- Se $f \in \text{ELEM}$ allora $f \in \mathscr{P}$
- Se $h, g_1, \ldots, g_k \in \mathscr{P}$ allora $\text{Comp}(h, g_1, \ldots, g_k) \in \mathscr{P}$
- Se $h, g \in \mathscr{P}$ allora $\text{RP}(h, g) \in \mathscr{P}$
- Se $f \in \mathscr{P}$, allora $\text{MIN}(f) \in \mathscr{P}$

Di conseguenza, proviamo per induzione strutturale su $\mathscr{P}$ che le funzioni sono while-programmabili:

- Passo base: le funzioni elementari sono semplicemente programmabili tramite while
- Passi induttivi:
  - Per $\text{Comp}(h, g_1, \ldots, g_k) \in \mathscr{P}$: si tratta di una composizione di istruzioni base, programmabili per I.H.
  - Per $\text{RP}(h, g) = f(\underline{x}, y)$: si usa un comando while e una variabile di controllo per iterare un singolo comando, programmabile per I.H., finché la variabile di controllo non ha raggiunto il valore di y
  - Per $\text{MIN}(f) \in \mathscr{P}$: un ciclo while può incrementare il valore di 1 finché non azzera la funzione, ottenendo la stessa semantica della minimalizzazione

Quindi $\mathscr{P} \subseteq F(\text{WHILE})$.

Per $F(\text{WHILE}) \subseteq \mathscr{P}$: si può rappresentare la semantica di un programma $W \in W\text{-PROG}$ con

$$\Psi_W = \text{Pro}_0^{21}([\![W]\!](w\text{-}in(x)))$$

Quindi, sapendo che l'operatore di proiezione è $\in \mathscr{P}$, se la funzione di stato prossimo è ricorsiva parziale abbiamo verificato l'inclusione.

Definiamo la funzione numero prossimo f_C che applica Cantor all'array degli stati; si può passare da $[\![C]\!](x)$ a $f_C(x)$ usando solo funzioni ricorsive parziali. Verifichiamo che f_C sia ricorsiva parziale per induzione strutturale:

- Passo base:
  - $C \equiv x_k := 0$

    $$f_C = \left[\text{Pro}_0^{21}(x), \ldots, 0, \ldots, \text{Pro}_2^{21}0(x)\right]$$

    con lo 0 in posizione k
  - $C \equiv x_k := x_j \pm 1$

    $$f_C = \left[\text{Pro}_0^{21}(x), \ldots, \text{Pro}_j^{21}(x) + 1, \ldots, \text{Pro}_2^{21}0(x)\right]$$

Avendo usato solo funzioni $\in \mathscr{P}$, i due comandi sono $\in \mathscr{P}$

- Passo induttivo:
  - $C \equiv$ `begin` $C_1; \ldots; C_n;$ `end`

    $$f_C = f_{C_n}(\ldots f_{C_1}(x)\ldots)$$

  - $C \equiv$ `while` $x_k \neq 0$ `do` C_b

    $$f_C(x) = f_{C_b}^{e(x)}(x) \text{ con } e(x) = \mu_y(\text{Pro}(k, f_{C_b}^y(x)))$$

    $e(x)$ non è costante, ma si può definire $T(x,y) = f_{C_b}^y(x)$, facilmente implementabile tramite ricorsione primitiva, quindi $\in \mathscr{P}$, facendo diventare $e(x)$ una minimalizzazione di funzioni $\in \mathscr{P}$, quindi $e(x) \in \mathscr{P}$.

  Quindi anche questi sono $\in \mathscr{P}$

Abbiamo provato anche che $F(\text{WHILE}) \in \mathscr{P}$.

Di conseguenza, in totale

$$\mathscr{P} = F(\text{WHILE})$$

20. Perché ha senso il corso di informatica teorica? Perché ha senso studiare la teoria della calcolabilità?

Solution: ~~Perché mi servono credit~~ Il corso si pone le domande di *cosa* è in grado di fare l'informatica e *come* è in grado di farlo.

La teoria della calcolabilità vuole dare una caratterizzazione generale, sotto forma matematica, di ciò che è calcolabile, ovvero i problemi risolvibili per via automatica. Ha l'obiettivo di delineare la portata dell'informatica stessa.

21. Definire la potenza di calcolo.

Solution: La potenza di calcolo di un sistema è definita come l'insieme di tutte le funzioni che quel sistema è in grado di calcolare.

Definendo un sistema di calcolo come

$$\mathcal{C} : \text{PROG} \to \text{DATI}, \text{ con } P \in \text{PROG} \text{ t.c. } P : \text{DATI} \to \text{DATI}_\perp$$

Con $\mathcal{C}(P, x)$ si indica la semantica del programma P su input x nel sistema di calcolo $\mathcal{C}$, ovvero il risultato dell'esecuzione.

La potenza di calcolo è quindi definita come

$$F(\mathcal{C}) = \{\mathcal{C}(P, \_) \mid P \in \text{PROG}\}$$

22. Cosa significa che $\mathbb{N}$ non è isomorfo a $\mathbb{N}_\perp^\mathbb{N}$?

Solution: Quando due insiemi sono isomorfi vuol dire che si può avere una biezione tra essi, $\mathbb{N}$ è "meno numeroso" di $\mathbb{N}_\perp^\mathbb{N}$, di conseguenza non si può avere una biezione.

Per dimostrarlo: per assurdo, supponiamo che $\mathbb{N} \sim \mathbb{N}_\perp^\mathbb{N}$, ovvero entrambi gli insiemi sono numerabili e di conseguenza listabili. Ricordando la definizione

$$\mathbb{N}_\perp^\mathbb{N} = \{f : \mathbb{N} \to \mathbb{N}_\perp\}$$

Possiamo listare tutte le funzioni:

- Sulla prima riga $f_0(0)$, $f_0(1)$, $f_0(2)$, …
- Sulla seconda riga $f_1(0)$, $f_1(1)$, $f_1(2)$, …
- …

Costruiamo una funzione $\varphi : \mathbb{N} \to \mathbb{N}_\perp$ tale che

$$\varphi(x) = \begin{cases} 1 & \text{se } f_x(x) = \perp \\ f_x(x) + 1 & \text{se } f_x(x) \downarrow \end{cases}$$

Questa funzione appartiene a $\mathbb{N}_\perp^\mathbb{N}$ ma non alla lista definita precedentemente, assurdo sotto l'assunzione che $\mathbb{N}_\perp^\mathbb{N}$, di conseguenza

$$\mathbb{N}_\perp^\mathbb{N} \nsim \mathbb{N}$$

23. Quali sono le funzioni in ELEM?

Solution: Le funzioni all'interno di ELEM sono

$$\text{ELEM} = \{ \begin{array}{rll} \text{successore:} & s(x) = x + 1, & x \in \mathbb{N} \\ \text{zero:} & 0^n(x_1, \ldots, x_n) = 0, & x_i \in \mathbb{N} \\ \text{proiettori:} & \text{Pro}_k^n(x_1, \ldots, x_n) = x_k, & x_i \in \mathbb{N} \end{array} \}$$

Queste sono funzioni "basilari" che qualunque idea di calcolabilità deve considerare come calcolabili.

24. Operatore di composizione, operatore di ricorsione primitiva. Quale è il nome dell'insieme ottenuto dopo la ricorsione primitiva?

Solution: L'operatore di composizione è definito come

$$\text{Comp}(h, g_1, \ldots, g_k)(\underline{x}) = h(g_1(\underline{x}), \ldots, g_k(\underline{x}))$$

L'operatore di ricorsione primitiva è definito come

$$\text{RP}(h, g) = f(\underline{x}, y) = \begin{cases} g(\underline{x}) & \text{se } y = 0 \\ h(f(\underline{x}, y-1), y-1, \underline{x}) & \text{se } y > 0 \end{cases}$$

Chiudendo l'insieme ELEM rispetto alle operazioni di composizione e ricorsione primitiva si ottiene

$$\text{ELEM}^{\{\text{Comp,RP}\}} = \text{RICPRIM}$$

25. RICPRIM $\subseteq F$(WHILE)

Solution: Per dimostrare che RICPRIM $\subseteq F$(WHILE) consideriamo una definizione induttiva di RICPRIM, questo è definibile come

- Se $f \in$ ELEM, $f \in$ RICPRIM
- Se $h, g_1, \ldots, g_k \in$ RICPRIM, Comp$(h, g_1, \ldots, g_k) \in$ RICPRIM
- Se $h, g \in$ RICPRIM, RP$(h, g) \in$ RICPRIM
- Nient'altro in RICPRIM

Di conseguenza mostrare per induzione strutturale che ognuno di questi punti è while-programmabile permette di verificare l'inclusione.

Passo base: le funzioni $\in$ ELEM sono banalmente $\in W$-PROG.

Passo induttivo:

- La composizione di funzioni è un susseguirsi di comandi base, $\in W$-PROG per I.H., di conseguenza implementabile in WHILE

- La ricorsione primitiva è una forma di iterazione definita (implementabile tramite comando while e una variabile di controllo) che esegue un comando, quest'ultimo $\in W$-PROG per I.H.

Di conseguenza abbiamo verificato che

$$\text{RICPRIM} \subseteq F(\text{WHILE})$$

26. Operatore di minimalizzazione.

Solution: L'operatore di minimalizzazione consiste, informalmente, del più piccolo valore che permette di azzerare una funzione, ovunque precedentemente definita.

Formalmente

$$\text{MIN}(f)(\underline{x}) = \begin{cases} y & \text{se } f(\underline{x}, y) = 0 \wedge (\forall y' < y, f(\underline{x}, y') \neq 0 \wedge f(\underline{x}, y') \downarrow) \\ \perp & \text{altrimenti} \end{cases}$$

27. Dimostrare $F(\text{WHILE}) \subseteq \mathscr{P}$.

Solution: Per dimostrare che $F(\text{WHILE}) \subseteq \mathscr{P}$ bisogna mostrare che la semantica di ogni programma $\in W$-PROG è esprimibile sotto forma di operazioni $\in \mathscr{P}$.

Sapendo che un programma while W è definibile come funzione stato prossimo sul programma stesso, la semantica su input x è esprimibile come

$$\Psi_W = \text{Pro}_0^{21}[\![W]\!](w\text{-}in(x))$$

Dove:

- La funzione $[\![C]\!](x)$ restituisce lo stato della macchina in seguito all'esecuzione del comando C (una tupla)
- Pro_0^{21} restituisce il valore del primo registro (quello di output), a partire dalla tupla contenente lo stato della macchina
- $w\text{-}in(x)$ definisce lo stato iniziale della macchina con input x

Di conseguenza, mostrando che la funzione di stato prossimo è ricorsiva parziale viene verificata l'inclusione.

Il primo problema è che $[\![\_]\!](\_)$ lavora su $\mathbb{N}^{21}$, mentre gli elementi di $\mathscr{P}$ hanno codominio $\mathbb{N}$. Per risolvere viene definita la funzione "numero prossimo" f_C, che condensa lo stato della macchina in un singolo valore tramite Cantor (useremo notazione $[\_]$).

Vogliamo quindi mostrare che $f_C \in \mathscr{P}$. Lo si può fare per induzione strutturale:

- Passo base: assegnamenti:
  - $C \equiv x_k := 0$

    $$f_C(x) = [\mathrm{Pro}(0,x), \dots, 0, \dots, \mathrm{Pro}(20,x)]$$

    dove lo 0 è in posizione k
  - $C \equiv x_k := x_j \pm 1$

    $$f_C(x) = [\mathrm{Pro}(0,x), \dots, \mathrm{Pro}(j,x) \pm 1, \dots, \mathrm{Pro}(20,x)]$$

    sempre lavorando sulla posizione k

  Si può notare che, per ora, f_C è composta da solo funzioni $\in \mathscr{P}$
- Passo induttivo:
  - $C \equiv \texttt{begin}\ C_1; \dots; C_k;\ \texttt{end}$

    $$f_C(x) = f_{c_k}(\dots f_{C_1}(x) \dots)$$
  - $C \equiv \texttt{while}\ x_k \neq 0\ \text{do}\ C_b$

    $$f_C(x) = f_{C_b}^{e(x)}(x) \quad \text{con} \quad e(x) = \mu_y(\mathrm{Pro}(k, f_{C_b}^y(x)) = 0)$$

    ma $e(x)$ non è costante. Si può definire però $T(x,y) = f_{C_b}^y(x)$, implementabile tramite ricorsione primitiva:

    $$T(x,y) = \begin{cases} x & \text{se } y = 0 \\ f_{C_b}(T(x, y-1)) & \text{se } y > 0 \end{cases}$$

    Di conseguenza $T(x,y) \in \mathscr{T}$, quindi

    $$e(x) = \mu_y(\mathrm{Pro}(k, f_{C_b}^y(x)) = 0) = \mu_y(\mathrm{Pro}(k, T(x,y)) = 0) \in \mathscr{T}$$

    In conclusione

    $$f_C(x) = T(x, e(x))$$

    quindi anche questo rientra $\in \mathscr{P}$

28. SPA.

Solution: Un Sistema di Programmazione Accettabile SPA è un sistema di calcolo $\mathcal{C}$ che rispetta gli assiomi di Rogers, ovvero

1. Aderisce alla tesi di Church-Turing

$$F(\mathcal{C}) = \mathscr{P}$$

2. Ammette interprete universale

$$\exists u \in \mathbb{N} \text{ t.c. } \forall x, y \in \mathbb{N},\ \varphi_u(\langle x, y\rangle) = \varphi_y(x)$$

3. Rispetta il teorema S_n^m, ovvero esiste $S_n^m \in \mathcal{T}$ tale che

$$\forall n \in \mathbb{N}, \underline{x} \in \mathbb{N}^m, \underline{y} \in \mathbb{N}^n,\ \varphi_n(\langle \underline{x}, \underline{y}\rangle) = \varphi_{S_n^m(n,\underline{y})}(\underline{x})$$

29. Teorema di ricorsione con dimostrazione.

Solution: Il teorema di ricorsione dice che, dato un SPA φ_i, $\forall t : \mathbb{N} \to \mathbb{N}$ vale che

$$\exists n \in \mathbb{N} \text{ t.c. } \varphi_n = \varphi_{t(n)}$$

Ovvero, qualunque sia la natura di t, esisterà sempre un programma che mantiene la semantica in seguito alla trasformazione.

Dimostrazione. Partiamo mostrando che

$$\varphi_{\varphi_i(i)}(x) \overset{(2)}{=} \varphi_{\varphi_u(i,i)}(x) \overset{(2)}{=} \varphi_u(x, \varphi_u(i,i)) \rightsquigarrow f(x,i) \in \mathscr{P}$$

Di conseguenza

$$\rightsquigarrow f(x,i) \overset{(1)}{=} \varphi_e(x,i) \overset{(3)}{=} \varphi_{S_1^1(e,i)}(x) \quad (\dagger)$$

Consideriamo la funzione $t\left(S_1^1(e,i)\right) \in \mathscr{P}$, di conseguenza

$$\exists m \in \mathbb{N} \text{ t.c. } t\left(S_1^1(e,i)\right) = \varphi_m(i) \quad (\dagger\dagger)$$

Poniamo ora $n = S_1^1(e,m)$

$$\varphi_{t(n)}(x) = \varphi_{t\left(S_1^1(e,m)\right)}(x) \overset{(\dagger\dagger)}{=} \varphi_{\varphi_m(m)}(x)$$

$$\varphi_n(x) = \varphi_{S_1^1(e,m)}(x) \overset{(\dagger)}{=} \varphi_{\varphi_m(m)}(x)$$

$$\implies \varphi_n(x) = \varphi_{t(n)}(x)$$

□

30. Struttura di una DTM.

Solution: Una macchina di Turing deterministica DTM è un dispositivo formato da un nastro potenzialmente infinito su cui leggere/scrivere e da un controllo a stati finiti. Può essere definita tramite la sestupla

$$M = (Q, \Sigma, \Gamma, \delta, q_0, F)$$

Dove

- Q è l'insieme degli stati che può assumere la macchina
- Σ è l'alfabeto di input
- Γ è l'alfabeto di lavoro
- $\delta : Q \times \Gamma \to Q \times \Gamma \setminus \{blank\} \times \{1, 0, -1\}$ è la funzione di transizione che, dato uno stato e un simbolo letto, definisce lo stato successivo, il simbolo scritto e il movimento sul nastro
- q_0 è lo stato iniziale
- $F \subseteq Q$ è l'insieme di stati finali in cui la macchina si arresta

Per descrivere lo stato di una DTM si usa una configurazione $C = (q, k, w)$, dove

- q è lo stato della macchina
- k è la posizione della testina
- w è il contenuto (non $blank$) del nastro

La principale funzionalità di una DTM è riconoscere linguaggi, un linguaggio $L \subseteq \Sigma^*$ è riconoscibile da una DTM se solo se esiste una DTM M tale che $L = L_M$. Di conseguenza, codificando insiemi in un linguaggio, si possono riconoscere insiemi e problemi di decisione. Le DTM sono anche in grado di calcolare funzioni, la computazione termina con $f(x)$ sul nastro se $f(x) \downarrow$, va in loop altrimenti.

31. Complessità temporale.

Solution: Data una DTM M, con $T(x)$ si intende il numero di mosse della computazione di M su x, la complessità in tempo è definita di conseguenza come

$$t : \mathbb{N} \to \mathbb{N} \text{ t.c. } t(n) = \max\{T(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

ovvero il tempo peggiore sugli input di una certa taglia.

Si dice che il linguaggio $L \subseteq \Sigma^*$ è riconoscibile in tempo deterministico $f(n)$ se esiste una DTM M tale che $L = L_M$ e $t(n) \leq f(n)$.

32. Complessità spaziale.

Solution: Data una DTM M, con $S(x)$ si intende il numero di celle occupate sul nastro di lavoro durante la computazione di M su x, la complessità in spazio è definita di conseguenza come

$$s : \mathbb{N} \to \mathbb{N} \text{ t.c. } s(n) = \max\{S(x) \mid x \in \Sigma^* \wedge |x| = n\}$$

ovvero il maggior spazio occupato per la computazione di una certa taglia di input.

Si dice che il linguaggio $L \subseteq \Sigma^*$ è riconoscibile in spazio deterministico $f(n)$ se esiste una DTM M tale che $L = L_M$ e $s(n) \leq f(n)$.

33. Relazione tra DTIME e NTIME.

Solution: Si può facilmente notare che

$$\text{DTIME}\left(f(n)\right) \subseteq \text{NTIME}\left(f(n)\right)$$

In quanto, data una DTM M, una NDTM si può comportare esattamente come M ignorando la parte non deterministica della computazione.

Per la relazione inversa, l'idea è quella di far generare e riconoscere a una DTM tutte le strutture che potrebbe considerare una NDTM, di conseguenza deve verificare in tempo $O(f(n))$ un numero di strutture generate $|\Gamma|^{O(f(n))}$, quindi il tempo richiesto diventa

$$t(n) = O(f(n)) \cdot |\Gamma|^{O(f(n))} = O\left(f(n) \cdot 2^{O(f(n))}\right)$$
$$\implies \text{NTIME}(f(n)) \subseteq \text{DTIME}\left(f(n) \cdot 2^{O(f(n))}\right)$$

34. La classe degli insiemi riconosciuti dalle macchine di Turing coincide con?

Solution: La classe degli insiemi riconosciuti da una DTM coincide con la classe degli insiemi ricorsivamente numerabili. Dato un insieme A, consideriamo la codifica dell'insieme L_A, allora l'insieme è riconoscibile da una DTM se esiste M tale che $L_A = L_M$. Il risultato della computazione può essere:

- $x \in A$: M si arresta
- $x \notin A$: M va in loop (se ricorsivamente numerabile) o si arresta rifiutando (l'insieme è ricorsivo)

Un algoritmo deterministico per il riconoscimento di $A \subseteq \mathbb{N}$ è una DTM M tale che $L_A = L_M$ e la computazione si arresta su ogni input. La classe degli insiemi riconosciuti da algoritmi deterministici coincide con la classe degli insiemi ricorsivi.

35. Convincimi del fatto che esistono Quine nei SPA.

Solution: I programmi di Quine sono programmi che hanno loro stessi come semantica, ovvero $\varphi_j(x) = j$, la domanda quindi diventa

$$\exists j \in \mathbb{N} \text{ t.c. } \varphi_j(x) = j, \ \forall x \in \mathbb{N}?$$

Sappiamo che esiste un programma P tale che $\varphi_P(x) = j$, di conseguenza si può codificare P tramite una funzione ricorsiva totale $cod(P) = g(j)$. Per il teorema di ricorsione vale quindi che

$$\exists j \in \mathbb{N} \text{ t.c. } \varphi_{g(j)}(x) = \varphi_j(x) = j$$

Quindi la risposta alla domanda è sì.

36. Teorema di Rice (enunciato e dimostrazione).

Solution: Un insieme di programmi $I \subseteq \mathbb{N}$ rispetta le funzioni se è chiuso per semantica, ovvero

$$(a \in I \wedge \varphi_a = \varphi_b) \implies b \in I$$

Il teorema di Rice dice che un insieme $I \subseteq \mathbb{N}$ che rispetta le funzioni non è ricorsivo, a meno che $I = \emptyset$ o $I = \mathbb{N}$.

Dimostrazione. Per assurdo, supponiamo che $I \neq \emptyset$ e $I \neq \mathbb{N}$ con I ricorsivo, vuol dire che esistono $a \in I$ e $b \notin I$. Definiamo quindi la funzione

$$t : \mathbb{N} \to \mathbb{N} \text{ t.c. } t(x) = \begin{cases} a & \text{se } x \notin I \\ b & \text{se } x \in I \end{cases}$$

E questa funzione è $\in \mathcal{T}$ e calcolabile grazie al programma P_I che decide I, il quale esiste per assurdo. Per il teorema di ricorsione sappiamo che

$$\exists d \in \mathbb{N} \text{ t.c. } \varphi_d = \varphi_{t(d)}$$

Quindi, per d ci sono due possibilità:

- $d \in I \implies t(d) \in I \implies b \notin I$, assurdo
- $d \notin I \implies t(d) \notin I$ e $t(d) = a \in I$, ma I rispetta le funzioni quindi $d \in I$, assurdo

□

37. Il teorema di Rice dice quando certi insiemi non sono ricorsivi. Fai un esempio di insieme che non rispetta le funzioni? Inventane uno. Questo è ricorsivo? Può avere anche altre caratteristiche questo insieme (essere ricorsivamente numerabile o altro)?

Solution: Degli esempi di insiemi che non rispettano le funzioni sono:

$$A = \{P \mid cod(P) \leq 100\}$$

In generale, se l'appartenenza all'insieme è determinata dalla struttura dei valori al posto che dalla loro semantica l'insieme non rispetta le funzioni (molto generalmente).

L'insieme A si può notare che non rispetta le funzioni (facilmente dimostrabile per TR), ed è ricorsivo: si può trovare un programma che calcola la codifica di un programma input e si arresta sempre determinando l'appartenenza o meno di tale input all'insieme (ovvero un programma che implementa la funzione caratteristica $\mathcal{X}_A \in \mathcal{T}$). Dato che è ricorsivo, è anche ricorsivamente numerabile.

38. Che linguaggi sono i DSPACE(1) e cosa si usa per riconoscerli? Perché basta un'automa e non tutta la DTM?

Solution: La classe dei linguaggi in DSPACE(1) sono quelli riconoscibili in spazio costante da una DTM, ovvero la classe dei linguaggi regolari.

Per definizione, un linguaggio è regolare se esiste un'automa che lo riconosce. Una DTM che usa spazio costante non può mantenere informazioni sulla computazione diverse dallo stato attuale, ovvero esattamente come una DFA, il quale si basa solo su stato attuale e simbolo letto. I linguaggi regolari non richiedono confronti tra parti non contigue della parola o conteggi non limitati.