Lezione 10 – riduzioni e introduzione alla Teoria della Complessità Computazionale

Lezione del 09/04/2024

Usare "a scatola nera"

- Torniamo alla dimostrazione dell'indecidibilità dell'Halting Problem
  - o, equivalentemente, alla non accettabilità del suo complemento
- Siamo partiti supponendo di avere una macchina T in grado di decidere L_H, e poi:
  - senza sapere come era fatta T (senza "aprirla")
  - abbiamo costruito una serie di altre macchine – T, T', T'', T* - che ci hanno portato dove volevamo
- La macchina T' la abbiamo costruita senza sapere come era fatta T
  - e come avremmo potuto saperlo? T manco esiste…
- Che, poi, è quello che facciamo sempre quando utilizziamo, ad esempio una classe delle API di Java
  - per dire, chi di voi, prima di utilizzare il metodo show di JFrame si andrebbe a vedere come è implementato?!
- Questo utilizzo "a scatola nera" di T corrisponde esattamente al concetto di "invocazione di funzione"

Usare "a scatola nera" – (1)

- Ora, quando T' usava T, T' passava (come "parametro") a T il suo stesso input (i,x)
- In generale, possiamo utilizzare una macchina T_0 all'interno di un'altra macchina T_1 in un modo un po' più complesso
  - in effetti, il linguaggio deciso/accettato da T_0 potrebbe anche essere molto diverso da quello deciso/accettato da T_1
  - allora, potrebbe essere necessario "modificare" l'input di T_1 prima di "darlo in pasto" a T_0
- Esempio (scemo): voglio costruire una macchina che decida il linguaggio L_{P12} che contiene tutte (e sole) le parole palindrome di lunghezza pari costituite dai caratteri '1' e '2'
  - caspiterina, quanto assomiglia a L_{PPAL} questo linguaggio L_{P12}, però!
  - quasi quasi, invece di mettermi lì a ri-progettare ex novo un'altra macchina, proverei a ri-utilizzare T_{PPAL} – che decide L_{PPAL}
  - peccato che T_{PPAL} lavori sull'alfabeto {a,b} invece che sull'alfabeto {1,2}
  - Uhm… quasi quasi, provo a trasformare le parole di L_{P12} in parole di L_{PPAL}…

Usare "a scatola nera" – (1)

- Voglio costruire una macchina che decida il linguaggio L_{P12} che contiene tutte (e sole) le parole palindrome di lunghezza pari costituite dai caratteri '1' e '2'
  - voglio costruire una macchina T_{P12} che utilizzi "a scatola nera" T_{PPAL}
  - peccato che T_{PPAL} lavori sull'alfabeto {a,b} invece che {1,2}
  - devo trasformare le parole di L_{P12} in parole di L_{PPAL}...
- Facile: prendo il mio x $\in${1,2}* e procedo così: assumendo $x = x_1 x_2 \ldots x_n$, per ogni h =1, 2, ... , n
  - se x_h = '1' allora poniamo y_h = 'a'
  - se x_h = '2' allora poniamo y_h = 'b'
  - infine, poniamo $y = y_1 y_2 \ldots y_n$.
- Quello che ho ottenuto è quindi una parola y $\in$ {a,b}* che ha le seguenti caratteristiche
  - **se x $\in$ L_{P12} allora y $\in$ L_{PPAL}**
  - **se x $\notin$ L_{P12} allora y $\notin$ L_{PPAL}**

Riduzioni (many-to-one)

- Quello che abbiamo fatto, in realtà, è qualcosa di più di una semplice trasformazione di una parola in un'altra parola
- Abbiamo progettato una funzione f : {1,2}* → {a,b}* tale che
- 1) **f è totale e calcolabile** – ossia,
  - è definita per ogni parola x ∈ {1,2}* e, inoltre,
  - esiste una macchina di Turing di tipo trasduttore T_f tale che, per ogni parola x ∈ {1,2}*, la computazione T_f (x) termina con la parola f(x) ∈ {a,b}* scritta sul nastro di output
- 2) per ogni x ∈ {1,2}* vale che: x ∈ L_{P12} se e solo se f(x) ∈ L_{PPAL}
  - che in matematichese si scrive : $\forall x \in \{1,2\}^* [x \in L_{P12} \leftrightarrow f(x) \in L_{PPAL}]$
- la funzione f si chiama ***riduzione*** da L_{P12} a L_{PPAL}
- e si dice che L_{P12} è ***riducibile*** a L_{PPAL} e si scrive $L_{P12} \preccurlyeq L_{PPAL}$

Riduzioni (many-to-one)

- Quello che abbiamo detto sino ad ora può essere generalizzato
- Dati due linguaggi, $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$, diciamo che **L_1 è *riducibile* a L_2** e scriviamo **$L_1 \preccurlyeq L_2$** se
- Esiste una funzione $f : \Sigma_1^* \rightarrow \Sigma_2^*$ tale che
- 1) f è totale e calcolabile – ossia,
  - è definita per ogni parola $x \in \Sigma_1^*$ e, inoltre,
  - esiste una macchina di Turing di tipo trasduttore T_f tale che, per ogni parola $x \in \Sigma_1^*$, la computazione $T_f(x)$ termina con la parola $f(x) \in \Sigma_2^*$ scritta sul nastro di output
- 2) per ogni $x \in \Sigma_1^*$ vale che: $x \in L_1$ se e solo se $f(x) \in L_2$
  - **$\forall x \in \Sigma_1^* [x \in L_1 \leftrightarrow f(x) \in L_2]$**
- Siamo al paragrafo 5.5

Decidibilità, accettabilità e riduzioni

- Il concetto di riduzione si rivela molto utile come strumento per dimostrare che un linguaggio è decidibile/accettabile: dato un linguaggio L_3
  - se dimostro che **$L_3 \preccurlyeq L_4$** , per un qualche altro linguaggio L_4 ,
  - se io so che **L_4 è decidibile**
  - allora, posso concludere che anche **L_3 è decidibile**
- Infatti, sia $L_3 \subseteq \Sigma_3^*$ e $L_4 \subseteq \Sigma_4^*$
  - **L_4 è decidibile** : allora esiste una macchina T_4 tale che, per ogni $x \in \Sigma_4^*$, $T_4(x)$ termina in q_A se $x \in L_4$, $T_4(x)$ termina in q_R se $x \notin L_4$
  - **$L_3 \preccurlyeq L_4$** : allora esiste una un trasduttore T_f tale che, per ogni $x \in \Sigma_3^*$, $T_f(x)$ termina con una parola $y \in \Sigma_4^*$ scritta sul nastro di output tale che $y \in L_4$ se $x \in L_3$, e $y \notin L_4$ se $x \notin L_3$
- Ora, costruiamo una macchina T_3 , a 2 nastri, che, con input $x \in \Sigma_3^*$:
  - prima simula $T_f(x)$ scrivendo l'output y sul secondo nastro
  - poi simula $T_4(y)$: se $T_4(y)$ accetta allora anche T_3 accetta, se $T_4(y)$ rigetta allora anche T_3 rigetta,

Decidibilità, accettabilità e riduzioni

- Abbiamo costruito una macchina T_3 , a 2 nastri, che, con input $x \in \Sigma_3^*$:
  - prima simula $T_f(x)$ scrivendo l'output y sul secondo nastro
  - poi simula $T_4(y)$: se $T_4(y)$ accetta allora anche T_3 accetta, se $T_4(y)$ rigetta allora anche T_3 rigetta,
- E a che ci serve?! Beh,
  - poiché è vero che $y \in L_4$ se $x \in L_3$, e $y \notin L_4$ se $x \notin L_3$, allora:
  - se $x \in L_3$ allora $y \in L_4$ e, quindi, $T_4(y)$ accetta; quindi, T_3 accetta le parole in L_3,
  - se , $x \notin L_3$ allora $y \notin L_4$ e, quindi, $T_4(y)$ rigetta; quindi T_3 rigetta le parole che non sono in L_3.
- In conclusione, T_3 decide L_3. Ossia, **L_3 è decidibile**
- Con una dimostrazione simile (che vi fate per esercizio) si dimostra che dato un linguaggio L_3
  - se dimostro che **$L_3 \preccurlyeq L_4$** , per un qualche altro linguaggio L_4 ,
  - se io so che **L_4 è accettabile**
  - allora, posso concludere che anche **L_3 è accettabile**

Decidibilità, accettabilità e riduzioni

- Il concetto di riduzione si rivela molto utile come strumento per dimostrare che un linguaggio è **non** decidibile/non accettabile: dato un linguaggio L_2
  - se dimostro che $L_1 \preccurlyeq L_2$, per un qualche altro linguaggio L_1 ,
  - se io so che **L_1 è non decidibile**
  - allora, posso concludere che anche **L_2 è non decidibile**
- Infatti, sia $L_1 \subseteq \Sigma_1^*$ e $L_2 \subseteq \Sigma_2^*$
  - **se L_2 fosse decidibile** (per assurdo): allora, poiché $L_1 \preccurlyeq L_2$, per quello che abbiamo appena dimostrato (nelle ultime due slides) anche **L_1 sarebbe decidibile** contraddicendo l'ipotesi che **L_1 è non decidibile**
- Con una dimostrazione simile (che vi fate per esercizio) si dimostra che dato un linguaggio L_2
  - se dimostro che $L_1 \preccurlyeq L_2$, per un qualche altro linguaggio L_1 ,
  - se io so che **L_1 è non accettabile**
  - allora, posso concludere che anche **L_2 è non accettabile**

Decidibilità, accettabilità e riduzioni

- In conclusione, il concetto di riduzione può essere utilizzato in due "direzioni":

- 1) riducendo un linguaggio **"sconosciuto"** ad un linguaggio accettabile/decidibile dimostriamo che il linguaggio **"sconosciuto"** è anch'esso accettabile/decidibile
  - ad esempio, dimostrando che $L_{P12} \preccurlyeq L_{PPAL}$ e sapendo che L_{PPAL} è decidibile, abbiamo dimostrato che L_{P12} è decidibile
  - ad esempio, dimostrando che $L_{H0} \preccurlyeq L_{H}$ e sapendo che L_{H} è accettabile, abbiamo dimostrato che L_{H0} è accettabile

- 2) riducendo un linguaggio non accettabile/non decidibile ad un linguaggio **"sconosciuto"** dimostriamo che il linguaggio **"sconosciuto"** è anch'esso non accettabile/non decidibile
  - ad esempio, dimostrando che $L_{H} \preccurlyeq L_{H0}$ e sapendo che L_{H} è non decidibile, abbiamo dimostrato che L_{H0} è decidibile

Riduzioni (many-to-one): esempio

- Facciamo un esempio di dimostrazione di non decidibilità di un linguaggio mediante riduzioni
  - è una modifica dell'esempio a pag. 7 della dispensa 5 – non tanto facile
- Consideriamo il linguaggio

  L_{H0} = { i ∈ ℕ: i è la codifica di una macchina di Turing T_i e T_i (0) termina}

  - ossia, i ∈ L_{H0} se i è la codifica di una macchina di Turing T_i e la computazione di T_i con input la parola costituita dal solo '0' termina
- caspita quanto assomiglia a L_H questo L_{H0}!
  - In effetti, L_{H0} è un sottoinsieme di L_H : L_{H0} contiene solo coppie (i,0)
  - questo si esprime dicendo che L_{H0} è una *restrizione* di L_H
  - e implica (ovviamente!) che un algoritmo che "si occupa" di L_H può "occuparsi allo stesso modo" anche di L_{H0}
  - e, quindi, la macchina di Turing che accetta L_H accetta anche L_{H0}
  - ma L_{H0} sembra "più facile" di L_H : quindi, magari, L_{H0} è decidibile...
- E, invece, sorprendentemente, non è vero che L_{H0} è "più facile" di L_H !

Riduzioni (many-to-one): esempio

- Non è vero che L_{H0} è "più facile" di L_H !
  - E come si fa a dimostrarlo?
- Riducendo L_H a L_{H0}!
  - ossia, individuando una funzione totale e calcolabile f che trasforma parole di L_H in parole di L_{H0} e parole non in L_H in parole non in L_{H0}
- Ricordiamo:
  L_H = {(i,x) ∈ ℕ×ℕ: i è la codifica di una macchina di Turing T_i e T_i (x) termina}
  e **L_{H0} = { i ∈ ℕ: i è la codifica di una macchina di Turing T_i e T_i (0) termina }**
- Consideriamo una qualsiasi coppia **(i,x)** ∈ ℕ×ℕ , con **x = $x_1x_2...x_n$**, e da essa deriviamo un intero **k = $k_{(i,x)}$** = f(**i,x**) nel modo seguente:
- a) se i non è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $M_{(i,x)}$ che entra in loop qualunque sia il suo input
- b) se i è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $M_{(i,x)}$ che, con input 0, simula **T_i (x)**
  - vedi prossima pagina
- c) $k_{(i,x)}$ è la codifica di $M_{(i,x)}$, ossia f(**i,x**) = $k_{(i,x)}$

Riduzioni (many-to-one): esempio

- b) se i è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $\mathbf{M_{(i,x)}}$ che
  - 1) nello stato iniziale $\mathbf{q^1}$: se legge 0 sul nastro allora scrive $\boldsymbol{x_1}$ e entra nello stato $\mathbf{q^2}$ muovendosi a destra, altrimenti rigetta
  - 2) nello stato $\mathbf{q^2}$: se legge ␣ sul nastro, scrive $\boldsymbol{x_2}$ e entra nello stato $\mathbf{q^3}$ muovendosi a destra, altrimenti rigetta
  - ... (eccetera eccetera) ...
  - n) nello stato $\mathbf{q^n}$: se legge ␣ sul nastro, scrive $\boldsymbol{x_n}$, riposiziona la testina sul primo carattere dell'input e entra nello stato iniziale $\mathbf{q_0}$ di $\mathbf{T_i}$, altrimenti rigetta
  - a questo punto, $\mathbf{M_{(i,x)}}$ inizia a simulare $\mathbf{T_i(x)}$ a scatola aperta
- Osservazione 1. Il numero degli stati $\mathbf{M_{(i,x)}}$ dipende da **x** e potrebbe sembrare non costante: non è così!
  - Innanzi tutto, **x** non è input per $\mathbf{M_{(i,x)}}$ (che si attende come input un solo carattere 0).
  - Poi, ricordiamo che abbiamo considerato una coppia **(i,x)** e solo dopo averla fissata abbiamo costruito $\mathbf{M_{(i,x)}}$. In altre parole, per ogni **x** abbiamo una $\mathbf{M_{(i,x)}}$, ossia **x** è costante per $\mathbf{M_{(i,x)}}$

Riduzioni (many-to-one): esempio

- Consideriamo una qualsiasi coppia **(i,x)** ∈ $\mathbb{N}\times\mathbb{N}$, con **x =** $x_1x_2...x_n$, e da essa deriviamo un intero $\mathbf{k_{(i,x)}}$ = f(**i**,**x**) nel modo seguente:
  - b) se i è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $\mathbf{M_{(i,x)}}$ che
    - 1) nello stato iniziale $\mathbf{q^1}$: se legge 0 sul nastro allora scrive x_1 e entra nello stato $\mathbf{q^2}$ muovendosi a destra, altrimenti entra in loop
    - 2) nello stato $\mathbf{q^2}$: se legge sul nastro, scrive x_2 e entra nello stato $\mathbf{q^3}$ muovendosi a destra, altrimenti rigetta
    - ... (eccetera eccetera) ...
    - n) nello stato $\mathbf{q^n}$: se legge sul nastro, scrive x_n , riposiziona la testina sul primo carattere dell'input e entra nello stato iniziale $\mathbf{q_0}$ di $\mathbf{T_i}$, altrimenti rigetta
    - a questo punto, $\mathbf{M_{(i,x)}}$ inizia a simulare $\mathbf{T_i(x)}$ a scatola aperta
- Sappiamo ben calcolare $\mathbf{M_{(i,x)}}$:
  - sia nel caso a), in cui l'intero **i** che non è la codifica di una macchina di Turing
  - sia nel caso b) in cui possiamo effettivamente costruire $\mathbf{M_{(i,x)}}$: avendo l'intero **i** che è la codifica (che ben conosciamo!) di $\mathbf{T_i}$, possiamo costruire le quintuple di $\mathbf{M_{(i,x)}}$ utilizzando tale codifica.
- Quindi, f è totale e calcolabile.

Riduzioni (many-to-one): esempio

- Fissando una qualsiasi **(i,x)** ∈ ℕ×ℕ, con **x =** $x_1x_2...x_n$, abbiamo costruito $\mathbf{M_{(i,x)}}$:
  - a) se i non è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $\mathbf{M_{(i,x)}}$ che entra in loop qualunque sia il suo input
  - b) se i è la codifica di una macchina di Turing, allora abbiamo costruito una macchina di Turing $\mathbf{M_{(i,x)}}$ che, con input 0, termina se e solo se $\mathbf{T_i(x)}$ termina
  - c) $\mathbf{k_{(i,x)}}$ è la codifica di $\mathbf{M_{(i,x)}}$
- L_{H0} = { i ∈ ℕ: i è la codifica di una macchina di Turing T_i e T_i (0) termina}
- Per ogni **(i,x)** ∈ ℕ×ℕ , poiché $\mathbf{k_{(i,x)}}$ è la codifica di se $\mathbf{M_{(i,x)}}$,
  - ossia $\mathbf{T_{k_{(i,x)}}} = \mathbf{M_{(i,x)}}$
- allora $\mathbf{k_{(i,x)}}$ ∈ L_{H0} se e solo se $\mathbf{T_{k_{(i,x)}}}$ (0) = $\mathbf{M_{(i,x)}}$(0) termina
- resta da capire in quali casi $\mathbf{M_{(i,x)}}$(0) termina

Riduzioni (many-to-one): esempio

- Fissando una qualsiasi **(i,x)** $\in \mathbb{N}\times\mathbb{N}$, con $\mathbf{x = x_1x_2\ldots x_n}$, abbiamo costruito $\mathbf{M_{(i,x)}}$:
  - a) se i non è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $\mathbf{M_{(i,x)}}$ che entra in loop qualunque sia il suo input
  - b) se i è la codifica di una macchina di Turing, allora abbiamo costruito una macchina di Turing $\mathbf{M_{(i,x)}}$ che, con input 0, termina se e solo se $\mathbf{T_i(x)}$ termina
  - c) $\mathbf{k_{(i,x)}}$ è la codifica di $\mathbf{M_{(i,x)}}$
- Per ogni **(i,x)** $\in \mathbb{N}\times\mathbb{N}$, se $\mathbf{M_{(i,x)}}(0)$ termina allora $\mathbf{M_{(i,x)}}$ è stata costruita eseguendo il passo b)
  - perché la macchina costruita eseguendo il passo a) non termina mai!
  - e quindi **i** è la codifica di una macchina di Turing $\mathbf{T_i}$
- e inoltre, $\mathbf{T_i(x)}$ termina,

- Ricapitolando: se $\mathbf{k_{(i,x)}} \in L_{H0}$ allora $\mathbf{T_{k_{(i,x)}}}(0) = \mathbf{M_{(i,x)}}(0)$ termina
- e se $\mathbf{T_{k_{(i,x)}}}(0) = \mathbf{M_{(i,x)}}(0)$ termina allora **i** è la codifica di una macchina di Turing e $\mathbf{T_i(x)}$ termina
- ossia: se $\mathbf{k_{(i,x)}} \in L_{H0}$ allora **(i,x)** $\in L_H$

Riduzioni (many-to-one): esempio

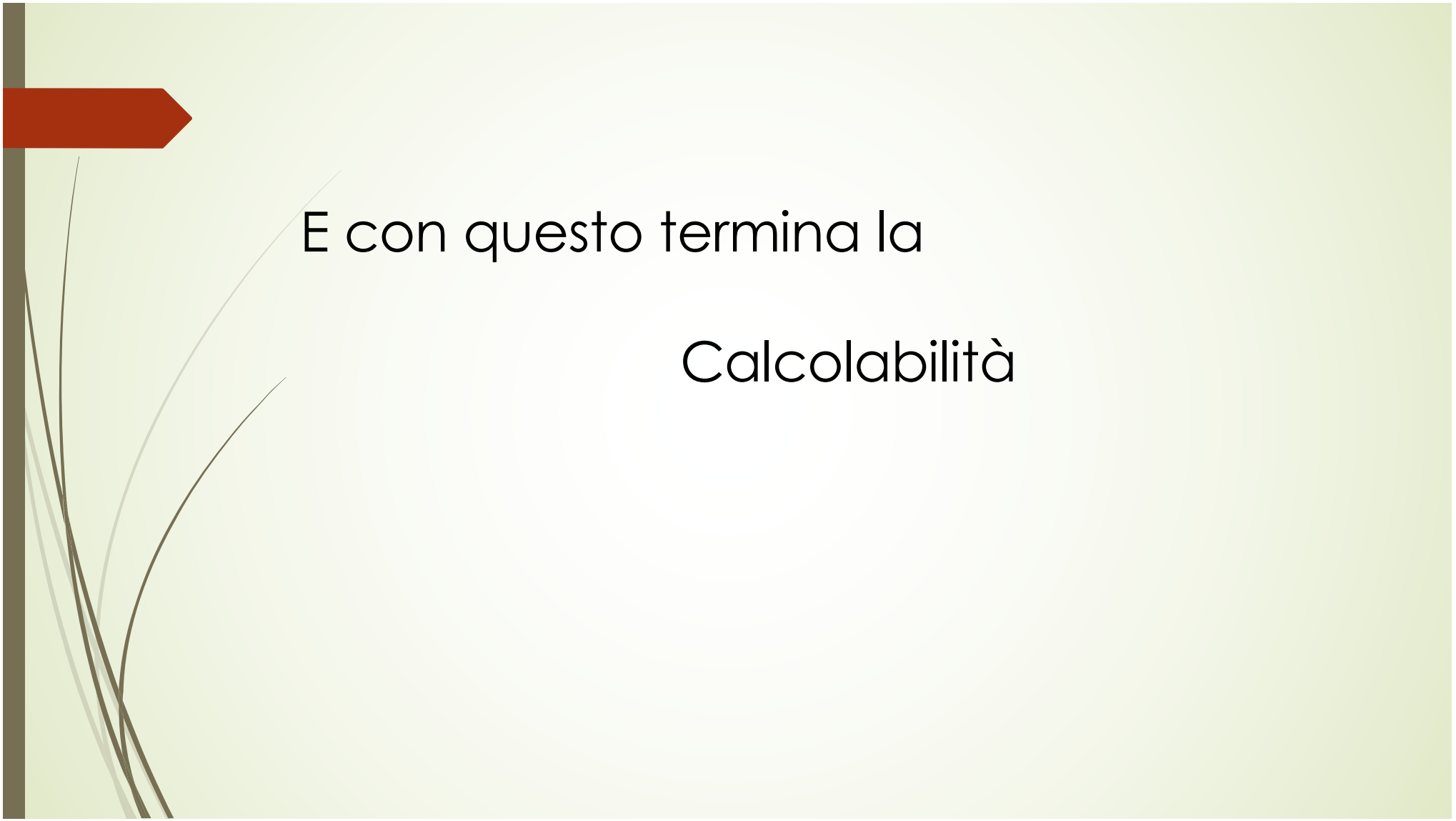
- Fissando una qualsiasi **(i,x)** ∈ ℕ×ℕ, con **x =** $x_1x_2...x_n$, abbiamo costruito $\mathbf{M_{(i,x)}}$:
  - a) se i non è la codifica di una macchina di Turing, allora costruiamo una macchina di Turing $\mathbf{M_{(i,x)}}$ che entra in loop qualunque sia il suo input
  - b) se i è la codifica di una macchina di Turing, allora abbiamo costruito una macchina di Turing $\mathbf{M_{(i,x)}}$ che, con input 0, termina se e solo se $\mathbf{T_i(x)}$ termina
  - c) $\mathbf{k_{(i,x)}}$ è la codifica di $\mathbf{M_{(i,x)}}$

- Per ogni **(i,x)** ∈ ℕ×ℕ , se $\mathbf{M_{(i,x)}}(0)$ non termina allora
  - o $\mathbf{M_{(i,x)}}$ è stata costruita eseguendo il passo a) e quindi **i** non è la codifica di una macchina di Turing $\mathbf{T_i}$ e quindi $\mathbf{M_{(i,x)}}(0)$ non termina
  - oppure $\mathbf{M_{(i,x)}}$ è stata costruita eseguendo il passo b), e quindi **i** è la codifica di una macchina di Turing $\mathbf{T_i}$, ma $\mathbf{T_i(x)}$ non termina,

- Ricapitolando: se $\mathbf{k_{(i,x)}} \notin L_{H0}$ allora $\mathbf{T_{k_{(i,x)}}}(0) = \mathbf{M_{(i,x)}}(0)$ non termina
- e se $\mathbf{T_{k_{(i,x)}}}(0) = \mathbf{M_{(i,x)}}(0)$ non termina allora **(i,x)** $\notin L_H$
- ossia, se $\mathbf{k_{(i,x)}} \notin L_{H0}$ allora allora **(i,x)** $\notin L_H$

Riduzioni (many-to-one): esempio

- Ricapitolando, abbiamo considerato una qualsiasi coppia **(i,x)** ∈ $\mathbb{N}\times\mathbb{N}$, con **x** = $x_1x_2\ldots x_n$, e da essa abbiamo derivato un intero $\mathbf{k}_{(i,x)}$ nel modo seguente:
  - a) se i non è la codifica di una macchina di Turing, allora abbiamo costruito una macchina di Turing $\mathbf{M}_{(i,x)}$ che entra in loop qualunque sia il suo input
  - b) se i è la codifica di una macchina di Turing, allora abbiamo costruito una macchina di Turing $\mathbf{M}_{(i,x)}$ che, con input 0, termina se e solo se $\mathbf{T_i(x)}$ termina
  - c) abbiamo calcolato $\mathbf{k}_{(i,x)}$ come la codifica di $\mathbf{M}_{(i,x)}$ ponendo, $\mathbf{k}_{(i,x)}$= f(**i**,**x**).
- 1) Poiché abbiamo descritto il procedimento che permette di calcolare $\mathbf{k}_{(i,x)}$ a partire da ogni coppia **(i,x)**, allora f è totale e calcolabile
- 2) per ogni **(i,x)** ∈ $\mathbb{N}\times\mathbb{N}$ vale che: **(i,x)** ∈ L_H se e solo se $\mathbf{k}_{(i,x)}$= f(**i**,**x**) ∈ L_{H0}
- Dunque, $\mathbf{L_H \preccurlyeq L_{H0}}$ e ciò dimostra che $\mathbf{L_{H0}}$ non è decibile!


E con questo termina la
Calcolabilità

Complessità: si parte!

- Abbiamo studiato cosa si intende per problema risolvibile
  - o meglio, per linguaggio decidibile
  - o linguaggio accettabile
  - o funzione calcolabile
- E abbiamo visto che esistono problemi non risolvibili.
- Ma anche (va da sé) problemi risolvibili.

- Uhmmm… Siamo davvero sicuri di poter risolvere i problemi risolvibili?

La Torre di Hanoi

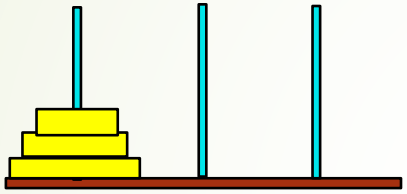
- *Narra la leggenda che, in un tempio nascosto nella foresta vicino ad Hanoi, sia custodito un grande piatto di ottone dal quale partono tre aste verticali di diamante. All'inizio dei tempi, su una delle tre aste vennero impilati da Brahma 64 dischi d'oro, di grandezze diverse gli uni dagli altri, a formare una torre: alla base era posto il disco più grande, sopra di esso quello immediatamente più piccolo, e così via, fino alla sommità della torre costituita dal disco più piccolo di tutti gli altri. Dall'inizio dei tempi, compito dei monaci è, semplicemente, spostare la torre dall'asta sulla quale Brahma l'aveva impilata ad un'altra asta. Due regole i monaci sono tenuti a rispettare: un solo disco alla volta può essere spostato da un'asta all'altra, e mai un disco può essere appoggiato su un disco più piccolo. Secondo la leggenda, quando i monaci termineranno il loro compito, quando, cioè, l'ultimo disco sarà finalmente piazzato a formare di nuovo la torre su un'asta diversa da quella sulla quale Brahma l'ha posta, allora arriverà la fine del mondo e tutto si trasformerà in polvere.*
- *Dobbiamo preoccuparci? Beh, prima di farlo, cerchiamo almeno di capire come funziona lo spostamento di una torre.*
  - Da "Il sentiero dei problemi impossibili", di cui sono autrice, Franco Angeli ed.

La Torre di Hanoi

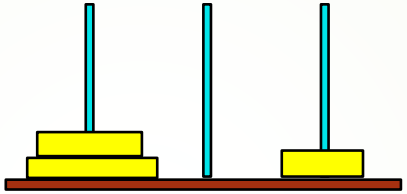
- Consideriamo una torre di 3 soli dischi impilata, diciamo, sull'asta a sinistra: l'obiettivo è spostarla sull'asta a destra
  - l'asta centrale avrà la funzione di "asta d'appoggio".
- Portiamo a termine il compito eseguendo le seguenti mosse (fig. nella prossima pag.):
  - 1) poiché possiamo spostare un solo disco alla volta, spostiamo il disco più piccolo sull'asta a destra;
  - 2) ora possiamo spostare il disco di grandezza intermedia e, poiché non possiamo appoggiarlo sul disco più piccolo, lo impiliamo nell'asta al centro;
  - 3) a questo punto, spostiamo il disco più piccolo sull'asta al centro, appoggiandolo sul disco di grandezza intermedia;
  - 4) spostiamo il disco più grande sull'asta a destra;
  - 5) spostiamo il disco più piccolo dall'asta centrale sull'asta a sinistra ;
  - 6) spostiamo il disco di grandezza intermedia sull'asta a destra, appoggiandolo sul disco più grande
  - 7) Infine, spostiamo il disco più piccolo sull'asta a destra, appoggiandolo sul disco di grandezza intermedia: fatto!

La Torre di Hanoi (3 dischi)

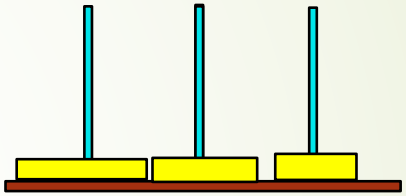
1)

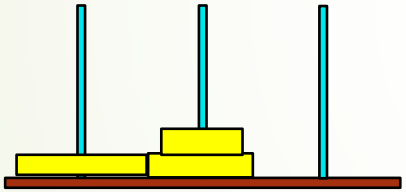


2)

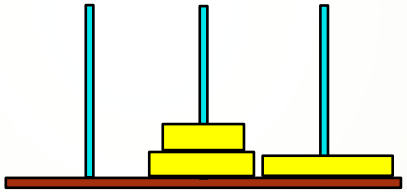


3)

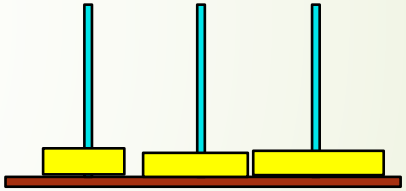




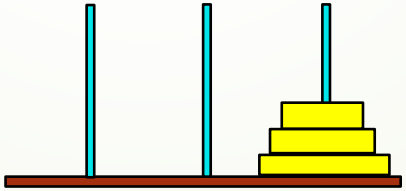
4)



5)



6)



7)

La Torre di Hanoi

- Dunque, abbiamo spostato una torre di 3 dischi utilizzando 7 spostamenti di dischi singoli
  - e non è possibile realizzare il nostro compito utilizzando un numero inferiore di spostamenti di dischi singoli.
- Per spostare una torre di 4 dischi è necessario:
  - spostare la sotto-torre costituita dai 3 dischi più piccoli dall'asta di sinistra a quella centrale,
  - poi spostare il disco più grande sull'asta di destra ,
  - e, infine, spostare la sotto-torre costituita dai 3 dischi più piccoli dall'asta centrale a quella di destra.
  - E non possiamo far di meglio!

La Torre di Hanoi

- Questo procedimento è generalizzabile
- Per spostare una torre di n dischi è necessario:
  - spostare la sotto-torre costituita dagli n-1 dischi più piccoli dall'asta di sinistra a quella centrale (configurazione 4) nella figura),
  - poi spostare il disco più grande sull'asta di destra (configurazione 5) nella figura),
  - e, infine, spostare la sotto-torre costituita dagli n-1 dischi più piccoli dall'asta centrale a quella di destra (configurazione 7) nella figura).
  - E non possiamo far di meglio!
- Quindi, se indichiamo con M(n) il numero di spostamenti di dischi singoli *necessario* a spostare una torre di n dischi, vale la seguente relazione di ricorrenza:

$$M(n) = 2\,M(n-1) + 1$$

- Che ha come soluzione $M(n) = 2^n - 1$
- E non possiamo far di meglio!

La Torre di Hanoi

- Quindi, se indichiamo con M(n) il numero di spostamenti di dischi singoli *necessario* a spostare una torre di n dischi, abbiamo che
  $$M(n) = 2^n-1$$
- E non possiamo far di meglio!
- Ma che significa?
- Che per spostare la Torre di Hanoi occorrono (sono necessari)
  $$2^{64} - 1 = 18.446.744.073.709.551.615$$
  spostamenti di dischi
- e che, anche se i monaci riuscissero a spostare un disco in 1 secondo, occorrerebbero almeno 18.446.744.073.709.551.615 secondi per spostare la torre
- che corrispondono a circa 5.845.580.504 secoli
- un tempo così lungo che quando il sole diverrà una gigante rossa e brucerà la Terra, il gioco non sarà ancora stato completato.
- ...

E allora?

- Intanto, possiamo stare ragionevolmente tranquilli: se sarà fine del mondo, non sarà per colpa dei monaci di Hanoi…
- Ma, soprattutto: lo sappiamo risolvere o no il problema della torre di Hanoi?
- Certo, che lo sappiamo risolvere!
  - vi ho mostrato il procedimento che sposta una torre di n dischi da un'asta all'altra!
- Tuttavia…
- Tuttavia, anche se *sappiamo come fare* a spostare una torre grande quanto ci pare, se la torre è abbastanza grande l'intera nostra vita non sarà sufficiente a vedere la torre spostata
- Se il tempo necessario a calcolare la soluzione di (un'istanza di) un problema è troppo elevato, saper calcolare quella soluzione è equivalente a non saperla calcolare
- …

La Teoria della Complessità Computazionale

- Studia la "quantità di risorse" **necessarie** a risolvere un problema
  - meglio: a decidere un linguaggio
- E suddivide i problemi in "trattabili" e "intrattabili"
  - dipendentemente dal fatto che la "quantità di risorse" necessarie cresca come un polinomio o più di un polinomio
- Ma perché la crescita polinomiale è discriminante fra trattabilità e intrattabilità?
  - Beh, lo avete visto quanto è grande 2^{64}: un numero di 20 (venti!) cifre. Invece, 64^2 è il minuscolo 4096. Piccolo.
  - Chiara l'idea?
- Una funzione più che polinomiale cresce infinitamente più velocemente di una funzione polinomiale!
  - e, se quella funzione rappresenta la "quantità di risorse" necessaria a risolvere un problema…

La Teoria della Complessità Computazionale

- Sì, ma qui stiamo parlando di funzioni che rappresentano la "quantità di risorse" necessaria a risolvere un problema
- Ma qual è *l'argomento* di queste funzioni?
  - Cioè: *in funzione di cosa* esprimiamo la complessità di un problema?
- E, poi, quali sono le "risorse" che prendiamo in considerazione?

- Che dire? La risposta nelle prossime puntate…